Level-Of-Details Rendering with Hardware Tessellation

Thibaud Lambert

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

par **Thibaud Lambert**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Rendu de niveaux de détails avec la Tessellation Matérielle

Date de soutenance : 18 décembre 2017

Devant la commission d'examen composée de :

Xavier Granier ..	Professeur des universités, Institut d'Optique .	Président
Tamy Boubekeur	Professeur, Telecom ParisTech	Rapporteur
Raphaëlle Chaine .	Professeur des universités, Université de Lyon	Rapporteur
Cyril Crassin	Chercheur, Nvidia	Examinateur
Gaël Guennebaud	Chargé de recherche, Inria	Directeur
Pierre Bénard	Maître de conférences, Université de Bordeaux	Encadrant

– 2017 –

Résumé Au cours des deux dernières décennies, les applications temps réel ont montré des améliorations colossales dans la génération de rendus photoréalistes. Cela est principalement dû à la disponibilité de modèles 3D avec une quantité croissante de détails. L'approche traditionnelle pour représenter et visualiser des objets 3D hautement détaillés est de les décomposer en un maillage basse fréquence et une carte de déplacement encodant les détails. La tessellation matérielle est le support idéal pour implémenter un rendu efficace de cette représentation. Dans ce contexte, nous proposons une méthode générale pour la génération et le rendu de maillages multi-résolutions compatibles avec la tessellation matérielle. Tout d'abord, nous introduisons une métrique dépendant de la vue capturant à la fois les distorsions géométriques et paramétriques, permettant de sélectionner la le niveau de résolution approprié au moment du rendu. Deuxièmement, nous présentons une nouvelle représentation hiérarchique permettant d'une part des transitions temporelles et spatiales continues entre les niveaux et d'autre part une tessellation matérielle non uniforme. Enfin, nous élaborons un processus de simplification pour générer notre représentation hiérarchique tout en minimisant notre métrique d'erreur. Notre méthode conduit à d'énormes améliorations tant en termes du nombre de triangles affiché qu'en temps de rendu par rapport aux méthodes alternatives.

Title Level-Of-Details Rendering with Hardware Tessellation

Abstract In the last two decades, real-time applications have exhibited colossal improvements in the generation of photo-realistic images. This is mainly due to the availability of 3D models with an increasing amount of details. Currently, the traditional approach to represent and visualize highly detailed 3D objects is to decompose them into a low-frequency mesh and a displacement map encoding the details. The hardware tessellation is the ideal support to implement an efficient rendering of this representation. In this context, we propose a general framework for the generation and the rendering of multi-resolution feature-aware meshes compatible with hardware tessellation. First, we introduce a view-dependent metric capturing both geometric and parametric distortions, allowing to select the appropriate resolution at render-time. Second, we present a novel hierarchical representation enabling on the one hand smooth temporal and spatial transitions between levels and on the other hand a non-uniform hardware tessellation. Last, we devise a simplification process to generate our hierarchical representation while minimizing our error metric. Our framework leads to huge improvements both in terms of triangle count and rendering time in comparison to alternative methods.

Keywords Real-time rendering, Hardware tessellation, Level of detail, Displacement mapping

Mots-clés Rendu temps réel, Tessellation matérielle, Niveaux de details, Carte de déplacement

Laboratoire d'accueil Inria Bordeaux Sud-Ouest

Remerciements

Je tiens tout d'abord à remercier mes deux encadrants de thèse, Gaël Guennebaud et Pierre Bénard pour leur grande disponibilité, leurs conseils et pour m'avoir guidé tout au long de ces trois années. Grâce à eux, mon doctorat a été une expérience plaisante et enrichissante. Je tiens également à remercier les membres de l'équipe Manao et plus précisément Pascal Barla, Xavier Granier, Romain Pacanowski et Patrick Reuter pour leur accueil, leur aide et leur retour sur mes travaux de recherche. Je souhaite aussi remercier Anne-Laure Gautier pour l'aide administrative qu'elle m'a apporté.

Tamy Boubekeur et Raphaëlle Chaine ont accepté d'être les rapporteurs de cette thèse, et je les en remercie. Je remercie également Xavier Granier, et Cyril Crassin leur participation à mon jury de thèse. Je les remercie donc tous pour le temps qu'ils m'ont consacré et pour leurs remarques constructives.

Cette thèse n'aurait pas pu exister sans le soutien de l'agence nationale de la recherche qui a financé le projet RichShape dans lequel s'inscrit ma thèse. Je remercie aussi l'Inria et son personnel pour m'avoir accueilli pendant ces trois années.

Je remercie Brett Ridel avec qui j'ai eu le plaisir de travailler dans le cadre de l'enseignement à l'IOGS. Je remercie également les membres de l'open-space, à savoir Antoine, Arthur, Boris, Carlos, David, George, Loïs, et Thomas pour la bonne ambiance et les moments de détentes partagés ensemble. Enfin, je remercie ma famille et mes amis pour le soutien qu'ils m'ont apporté.

Contents

Introduction

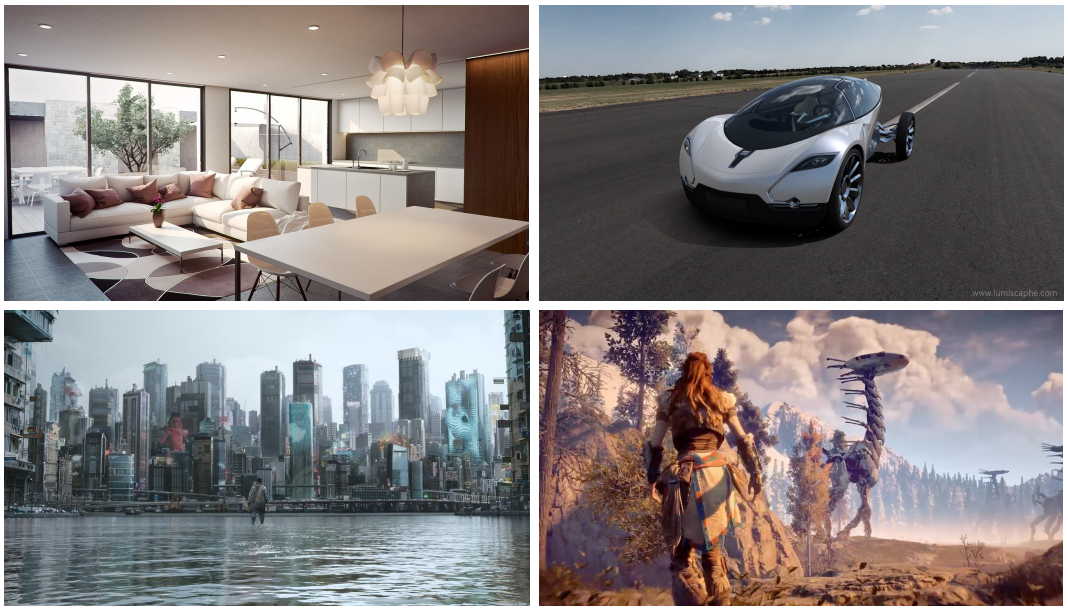


Figure 1 – Image synthesis — Top left: an architectural design. Top right: rendered car from Lumiscaphe. Bottom left: real-world image mixed with a digital one from "Ghost in the Shell". Bottom right: image from the game "Horizon Zero Dawn".

This thesis falls in the scope of 3D images synthesis that aims at generating digital images from virtual worlds through computer algorithms. Image synthesis is central in numerous application domains such as digital entertainment, training simulation, scientific visualization, advertisement, medical imaging, etc. (Figure 1). For most of these applications, one of the goals is to produce so called photo-realistic rendering. Two conditions need to be met to achieve this goal. First, rendering algorithms have to be developed to reproduce the physical laws of light transports and light-matter interactions with the highest accuracy possible. Second, the digital scenes must closely reproduce the complexity of real-world environments, thus requiring a great amount of details. These two aspects are essential in the film industry for special visual effects where real-world images are mixed with digital ones, or

(a) Offline rendering **(b)** Real-time rendering

Figure 2 – Offline rendering versus real-time rendering — **(a)** An image taken from the movie "The Good Dinosaur". **(b)** An image taken from the video game "Uncharted 4". Contrary to real-time renderings, offline renderings are very difficult to differentiate from real world images.

in architecture and the automobile industry to preview and investigate possible designs. In this context, an enormous amount of details is necessary to represent as realistic as possible 3D models. Moreover, the release of displays with high density of pixels also motivates the use of highly detailed models to leverage the full benefits of such displays.

Rendering algorithms can be classified into two main categories: offline algorithms and real-time algorithms. The former are subject to little time constraints; they are used for films and animation movies. Offline rendering applications have thus the option to use expensive algorithms. They generally use global illumination techniques combined with highly detailed models to obtain images that are almost not differentiable from photographs (Figure 2**(a)**). This is achieved at the price of a long computation time; the rendering of a single image can take hours or even days.

Real-time rendering, on the other hand, is concerned with generating images in a very small amount of time. This is essential for interactive applications such as video games, interactive simulations, VR applications, etc. to offer a smooth experience to the user. Each image has to be computed in less than 30ms to achieve a comfortable frame rate. Images rendered in real-time are thus still easy to differentiate from real ones (Figure 2**(b)**). This is due to the use of both coarser models and approximated light transport algorithms that run much faster than offline algorithms.

To achieve such performance, all real-time rendering engines entirely rely on dedicated Graphics Processing Units (GPUs). Modern GPUs are massive parallel processors composed of thousands of cores. They are specifically dedicated to the real-time rendering of a specific representation of 3D models: the triangle mesh combined with textures (Figure 3). The triangle mesh represents the general shape of the model, while the textures encode the finer details and surface attributes such as colors, normals, and lighting properties. Such mod-

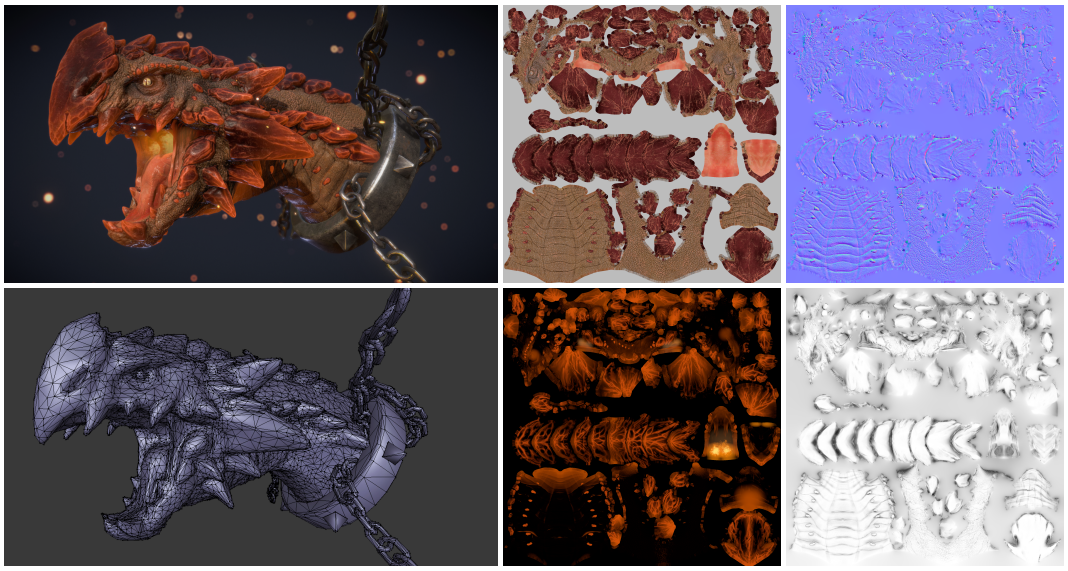


Figure 3 – "Blood and Fire" 3D model courtesy of Rico Cilliers — A 3D model composed of a triangle mesh (left) with albedo, normals, emission, and ambient occlusion textures (right).

els can either be created by artists using 3D modeling or sculpting software, or obtained by scanning real world objects. In both cases, the number of triangles per object have been constantly increasing in the past years to represent even more detailed objects. In addition, 3D scenes have also become more and more complex, and can be composed of several thousand objects. This results in a very large amount of triangles that raises memory, performance and filtering issues.

A variety of methods have been proposed to address these problems. First, only objects that are visible in the final image need to be rendered. So objects outside the field of view of the camera are discarded. For the same reason, large objects that cross the boundaries of the field of view are often cut along these boundaries in a process called clipping, and the parts outside the field of view are discarded. For opaque and solid objects, which is the case of most objects in a typical video-game scene, back-facing triangles with respect to the current viewpoint cannot be visible and are thus safely discarded. The notion of culling can be extended to occlusions of one object by others, this is called occlusion culling. This is generally done by pre-computing some form of hierarchy of potentially visible sets of objects, and then this hierarchy is used at runtime to identify what is visible and what is not. Clipping and culling are very popular mechanisms to greatly reduce the complexity of a scene and thus to speed up the rendering. However, when using highly detailed meshes, these mechanisms are not sufficient. For example, a single mesh can contain several millions of triangles and if it is far away, it can be projected on a dozen

Figure 4 – Level of details in "The Witcher 3" — The resolution of the trees is adapted according to the view distance to reduce the complexity of the scene.

of pixels, resulting in a particularly inefficient rendering.

Level of details (LODs) rendering aims at reducing drastically the number of visible triangles in a scene. The key idea of level of details is to adapt the resolution of the object according to the view-point. Distant objects fill less space on the screen than close ones, and thus need fewer polygons to be accurately rendered (Figure 4). Based on these observations, a multi-resolution representation is pre-computed for each object and at render time, the appropriate resolution is used according to a view-dependent criterion.

The simplest LOD approach for 3D models consists in a fixed set of meshes with a decreasing number of polygons. At render time, the mesh with lowest-resolution satisfying the view-dependent criterion is displayed. The lack of transition when changing from one level to another leads to visible popping artifacts: the 3D object becomes suddenly more detailed. The famous Progressive Mesh technique [Hoppe, 1996], addresses this issue by proposing a continuous-resolution representation. It consists in a coarse mesh and a set of operations that indicates how to refine the coarse mesh back into the original one. In addition to be more compact than discrete LODs, it enables fine-grain control of the vertex density with smooth temporal transitions. However, a fundamental issue with progressive meshes is that it cannot leverage the full power of massive parallel GPU architectures due to the dependence of the operations with each other.

These LOD representations are generally generated using a polygon reduction algorithm. The goal of such algorithms is to reduce the number of polygons of the mesh while preserving as best as possible its visual appearance. To this

Figure 5 – Rendered image from Unigine Heaven — The geometry of the 3D scene (left) is amplified using displacement mapping with hardware tessellation (right).

end, they need to measure throughout the simplification the differences with the original mesh. This error metric is at the heart of simplification methods, as it will define the order in which the simplification operations will be applied, but also drive the local optimizations to relocate appropriately the vertices throughout the simplification. A good error metric will better preserve important features of the mesh, and will thus allow to reduce further the scene complexity for the same visual quality.

Selecting the appropriate level to reduce as much as possible the complexity of the scene while preserving its visual appearance is also a central but challenging task. Indeed, the visual appearance of a 3D model depends on numerous parameters: not only its geometry, but also its material, the illumination environment, the camera viewpoint, the display, the human visual system, the lighting condition in the room, etc. These parameters have complex interactions with each other and it is therefore difficult to take into account all of them at once. Moreover, the evaluation of such a metric must be extremely fast to be used in a real-time applications. As a result, existing methods are only based on the view distance and the geometry of the object, letting plenty of room for significant improvements.

An alternative method to represent highly detailed meshes is to use displacement maps. The highly detailed mesh is decomposed during a pre-process into a coarse surface and a 2D map encoding the details, i.e., the relief, of the 3D model. Such a map is called a displacement map. More precisely, the displacement map is an offset function describing how to deform the coarse surface to reproduce the original mesh. This representation can be efficiently rendered using the hardware tessellation engine of modern GPU which enables a dynamic control of the mesh resolution. Each face, called in this case a patch, of the input coarse mesh is subdivided at a given resolution according to a fixed and uniform subdivision pattern. Then, the generated vertices are moved to the desired 3D location using the displacement map.

The specificities of hardware tessellation imply new challenges. First, a

recurrent challenge when rendering displacement maps with hardware tessellation is to ensure continuous temporal transitions. The goal is to solve popping artifacts without introducing swimming artifacts, i.e., visible fluctuations of the reconstructed surface due to the undersampling of the displacement map.

Displacement mapping with hardware tessellation offers a much more compact representation than progressive mesh, thanks to the tessellation pattern storing implicitly the topology. Unfortunately, this compact representation comes at the price of less flexibility. For the same amount of triangles, it introduces generally a higher error than fine-grained approaches such as the progressive mesh. The second challenge is then how to enable feature-aware hardware tessellation. This problem is particularly difficult due to the fixed hardware tessellation pattern, which prevents the use of existing feature-preserving mesh simplification algorithms.

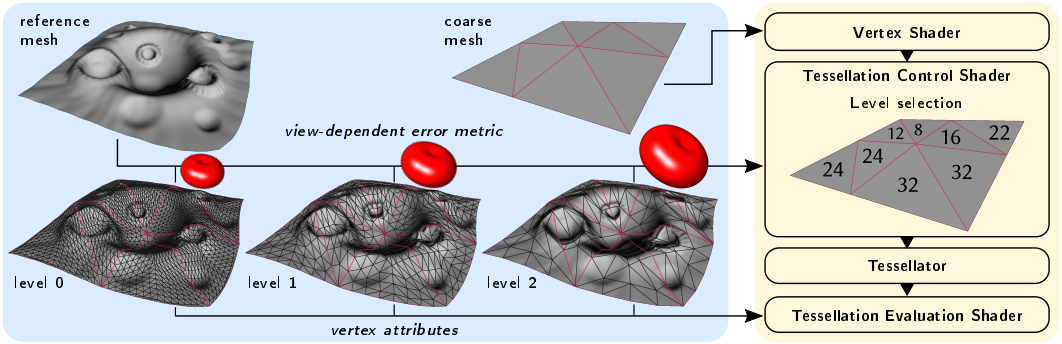
Contributions

In this thesis, we tackle the following challenges:

- How to best exploit GPU tessellation to more finely control the repartition of tessellated vertices and best preserve the features of the detailed mesh?
- How to generate high quality LOD compatible with GPU tessellation?
- How to achieve smooth temporal transitions between the LOD while avoiding geometry fluctuations?
- How to quickly select the best LOD to ensure high visual quality with a minimal number of polygon?

To address these challenges, we present a general framework (Figure 6) for the generation and the rendering of feature-aware LODs compatible with hardware tessellation. From an input detailed mesh and a corresponding decomposition into patches, our methods produces a hierarchy of LOD compatible with hardware tessellation, and a view-dependent metric measuring, for each patch and each level of the hierarchy, its view-dependent error with respect to the input mesh. This representation is then used in a custom controllable non-uniform tessellation pass avoiding swimming artifacts and providing significant performances gains compared to alternative methods. More precisely, we make the following contributions:

Geometric and attributes view-dependent metric Our first contribution is a patch-based view-dependent metric which estimates both the geometric and attributes distance between the LOD and the reference detailed mesh.



(a) LOD generation **(b)** GPU evaluation

Figure 6 – Full processing pipeline — **(a)** as a pre-process, the LOD is generated from the reference mesh by decimation and, for each patch, at each level, its approximation error with respect to the reference surface is summarized into a compact view-dependent metric, **(b)** these are then used during hardware tessellation to select the most appropriate patch level according to the current viewing distance and direction.

It is accomplished by considering *difference vectors* matching points with the same texture coordinates on the LOD and reference surface. We show that the error induced by these vectors can be tightly and conservatively summarized by a simple representation, which enables a fast GPU evaluation of the approximation error along any view directions (Chapter 2).

Controllable custom tessellation Then, we introduce a novel interpolation scheme between tessellation levels that enables controllable custom tessellation while avoiding *swimming* artifacts. It allows to finely control the location of the topological changes between levels, paving the way for non-uniform tessellation. Our method only requires the storage of an additional index per vertex to link vertices from coarse to fine levels (Chapter 3).

Feature-preserving simplification To exploit the flexibility introduced by our representation, we design a feature-preserving mesh simplification algorithm. Starting from a base mesh tessellated and displaced at the finest level, our algorithm progressively decimates the mesh while matching the hardware tessellation pattern at successive levels. We devise a novel simplification heuristic based on our new error metric to optimize simultaneously the positions and texture coordinates of the relocated vertices (Chapter 4).

Chapter 1

Related Work

Whether it is for collision detection, finite element analysis, or real-time rendering, simplified meshes are used to speed up computation while still providing accurate enough results. In this context, the user needs a way to control the error introduced by the simplification process to maintain some desired level of accuracy of the subsequent computation. For real-time rendering, the development of quantitative measures of the simplification error is a crucial ingredient of LOD methods. Those measures play an important role throughout the whole process, whether it is for the generation, the evaluation or the rendering of LODs.

We start by presenting the Graphics Pipeline. Next, we present the metrics that have been proposed to measure the differences between two meshes (Section 1.2). We then compare the different LOD representations and generation methods for level of details (Section 1.3). Finally we present an alternative representation – the displacement maps – for representing highly detailed meshes, and show that it can be efficiently rendered with hardware tessellation (Section 1.4).

1.1 Graphics Pipeline

The graphics pipeline on current GPUs is accessible using a graphics API such as OpenGL (cross-platform) or DirectX (Windows only). In the following, we will use the OpenGL nomenclature. It consists of programmable shader stages and fixed function stages (Figure 1.1) allowing to transform a 3D scene composed of polygonal meshes into a pixel image. In the following, we will explain the role of each stage except the three hardware tessellation stages which are detailed latter.

Inputs The typical representation for 3D objects is to use a set of polygons, called a mesh. A mesh is composed of the mesh geometry, represented by the

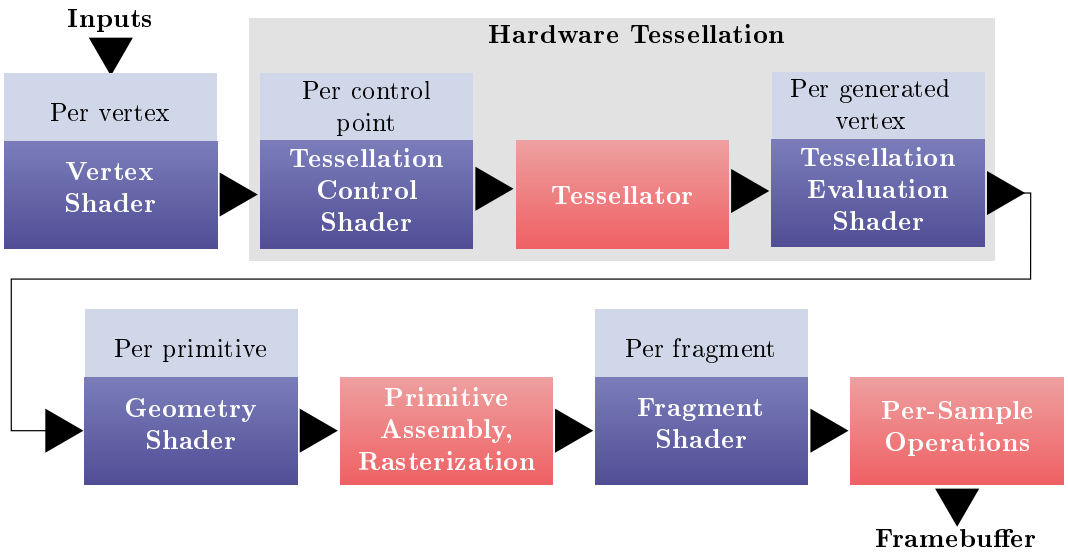


Figure 1.1 – OpenGL Graphics pipeline — The programmable stages are in blue and the fixed stages are in red.

vertices, and the mesh connectivity, represented by the edges or faces that connect the vertices. The graphics pipeline is specifically designed for triangle, and thus even if quads are used, there will be an additional triangularization step.

In addition to the geometry, polygonal meshes can have a material, properties describing the visual appearance of the object, represented using some form of attributes attached to the surfaces. Color, normals, and lighting properties are example of such attributes. This attributes can be specified per faces, per vertex or to a finer scale using texture. To represent highly complex appearance, the texture approach is the most adapted as it allows to store attributes with high frequency variation without the connectivity and geometry overhead. Thus the attributes are generally decoupled from the geometry and stored into a 2D textures Cohen et al. [1998]. Then the only remaining per vertex attributes is the textures coordinates used to fetch the other attributes.

Vertex Shader Once the mesh data sent to the pipeline, the Vertex Shader is ran on each input vertex. It typically performs per-vertex operation such as transformations, animations, or skinning.

Geometry shader The Geometry shader is an optional stage allowing various geometric manipulations. It takes as input a primitive (point, line, triangle, ...) and outputs zero or more primitives. The Geometry Shader is usually used for doing computational tasks on the GPU and get the data back through a Transform Feedback buffer. While the Geometry shader can also be used to amplify the resolution of a model, it is not as efficient as the tessellation

mechanism.

Primitive Assembly and Rasterization The purpose of the Primitive Assembly is to group the vertices into a sequence of individual base primitives. This is done using the list of indices specified in input. Then the rasterization transforms these primitives into a pixel image. Each primitive is split into discrete elements called fragments: A fragment is generated for each pixel covered by the primitive.

Fragment shader Fragments generated by the rasterization are then processed by the Fragment Shader. The input of the fragment shader are computed by interpolation of the primitives attributes. Each fragment shader invocation outputs a depth value and zero or more color values. The color output of the fragment is usually computed according to the light, normal, and textures. Finally, the fragments are written to the frame buffer after per-fragment operations such as depth test and blending steps.

1.2 Mesh Comparison Error Metrics

Error metrics are the key ingredient of level of detail methods. First at rendering time, the goal is to choose the level of detail with the minimum number of triangles without impacting the visual appearance. To accomplish this, it is necessary to define for each model some form of measure of the error introduced by each coarser level. Then to reduce the complexity of our 3D scene, we need to be able to evaluate the screen-space error of a particular LOD when viewed from a given viewpoint.

Second, during the generation of the coarser levels of detail, the input geometry needs to be simplified while minimizing some error metrics that quantify the differences introduced during the simplification. Many simplification algorithms consist in choosing the best simplification operation to apply among a large number of available choices. The best operation is thus defined with respect to those error metrics. The more accurate the error metric, the better will be the choices throughout the simplification process. Some simplification operations also require a placement strategy of the simplified vertices, this optimization can also be guided by those error metrics, thus proposing better choice available to simplification process and resulting in better quality meshes.

Finally, some simplification approaches use local metrics or heuristics to guide the simplification process and therefore have no clue about the global error introduced during the simplification. It is thus useful to have tools, such as Metro [Cignoni et al., 1996], to evaluate the global simplification error afterwards. Metro allows to compare the difference between a pair of meshes

by computing the maximum and mean geometric error. It has been used in several research papers to compare different simplification methods.

1.2.1 Geometry-Based Metrics

The simplification of a polygonal mesh reduces the number of its faces, changing as a result the shape of the surface. Preserving this shape helps the object to cover the correct pixels on screen and to maintain an accurate silhouette. Finding the distance between two surfaces let us measure the geometrical differences, and thus the modifications of the shape introduced by the simplification.

The Hausdorff distance is certainly the most common geometric error metric when it comes to comparing two surfaces. It defines the distance between two point sets, but as a surface can be seen as a continuous set of points, it can also be applied to surfaces.

In the context of geometry processing, the Hausdorff distance measures the maximum distance between two surfaces. Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be two surfaces, and $\mathbf{x}$ be a point belonging to $\mathcal{S}_1$. Then the distance d from $\mathbf{x}$ to $\mathcal{S}_2$ is expressed by the following equation:

$$\mathrm{d}(\mathbf{x}, \mathcal{S}_2) = \inf_{\mathbf{y} \in \mathcal{S}_2} \|\mathbf{x} - \mathbf{y}\| \, . \tag{1.1}$$

Thus the distance h from $\mathcal{S}_1$ to $\mathcal{S}_2$ is defined by the relation:

$$\mathrm{h}(\mathcal{S}_1, \mathcal{S}_2) = \sup_{\mathbf{x} \in \mathcal{S}_1} \mathrm{d}(\mathbf{x}, \mathcal{S}_2) = \sup_{\mathbf{x} \in \mathcal{S}_1} \inf_{\mathbf{y} \in \mathcal{S}_2} \|\mathbf{x} - \mathbf{y}\| \, . \tag{1.2}$$

The equation (1.2) is called the one-sided Hausdorff distance. This is not a symmetric distance as in general $\mathrm{h}(\mathcal{S}_1, \mathcal{S}_2) \neq \mathrm{h}(\mathcal{S}_2, \mathcal{S}_1)$. Every point of $\mathcal{S}_1$ have been matched with a point of $\mathcal{S}_2$, but there may be unmatched points of $\mathcal{S}_2$ with potentially a greater distance to $\mathcal{S}_1$. The Hausdorff distance H is the two-sided distance between the two surfaces and is defined as:

$$\mathrm{H}(\mathcal{S}_1, \mathcal{S}_2) = \max(\mathrm{h}(\mathcal{S}_1, \mathcal{S}_2), \mathrm{h}(\mathcal{S}_2, \mathcal{S}_1)) \, . \tag{1.3}$$

It is worthy to mention that if $\mathrm{H}(\mathcal{S}_1, \mathcal{S}_2) = 0$, then the two surfaces $\mathcal{S}_1$ and $\mathcal{S}_2$ are geometrically the same. The Hausdorff distance provides the tightest error bound on the maximum distance between two surfaces. This is useful for applications such as medical or scientific visualization, because it allows to guarantee that the error is never superior to a given tolerance.

In other applications, the average error may be a more reasonable choice as it provides a better indication of the error across the entire surface. In a similar vein, we define the one-sided mean error m as:

$$\mathrm{m}(\mathcal{S}_1, \mathcal{S}_2) = \int_{\mathcal{S}_1} \inf_{\mathbf{y} \in \mathcal{S}_2} \|\mathbf{x} - \mathbf{y}\| d\mathbf{x} \ , \tag{1.4}$$

and the symmetric mean error M as:

$$\mathrm{M}(\mathcal{S}_1, \mathcal{S}_2) = \frac{1}{2}(\mathrm{m}(\mathcal{S}_1, \mathcal{S}_2) + \mathrm{m}(\mathcal{S}_2, \mathcal{S}_1)) \ . \tag{1.5}$$

Only minimizing the mean average error may create few location with high error. Reciprocally, minimizing the maximum error can lead to a large increase of the average error. Ideally, simplification methods should find some form of compromise between the two.

1.2.2 Attribute Error Metrics

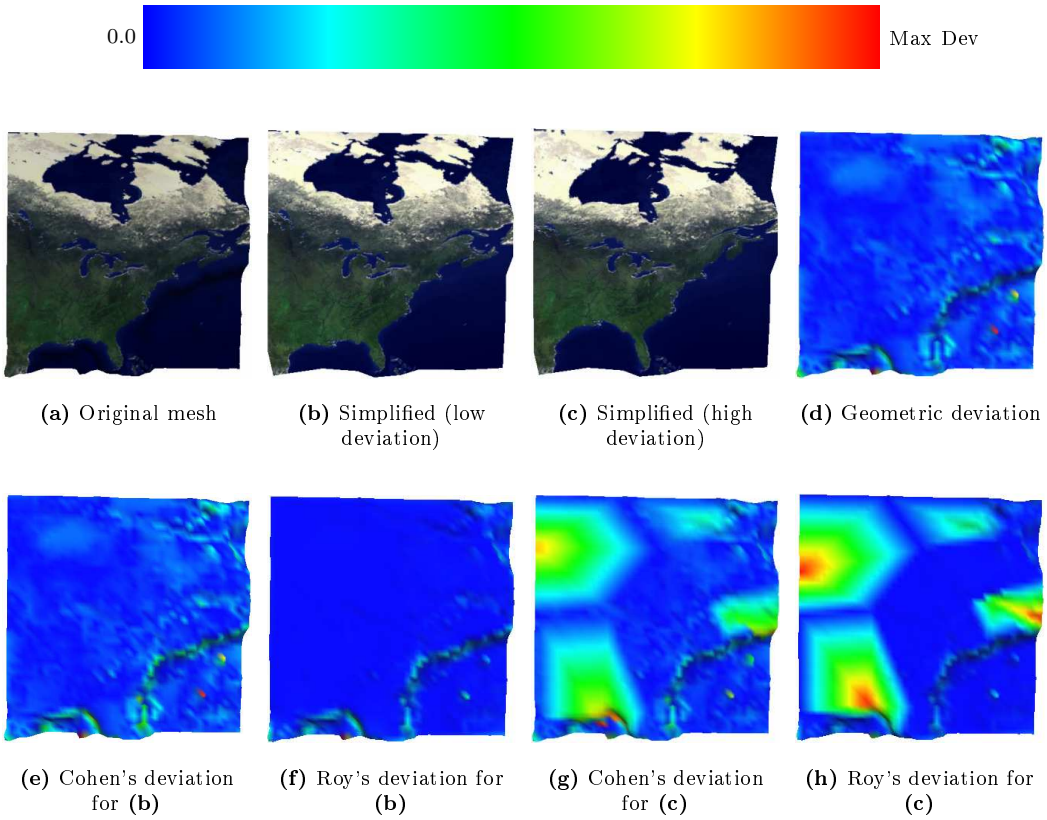
Blindly minimizing geometric errors is usually not sufficient to preserve the visual aspect of a model. Indeed, in addition to the shape, the simplification process should preserve the attributes attached to the surfaces. In our context of rendering such attributes include color, normals, texture coordinates, etc, and are used for shading purpose. Attributes stored per vertex will be affected by the simplification, thus measuring the attributes error between reference and simplified meshes is essential to carry out more meaningful a representative simplification error.

Roy et al. [2004] propose a mesh comparison method to compute the local differences between the attributes of two meshes. They define the attribute deviation $\mathrm{d_a}$ between the surface $\mathcal{S}_2$ and a point $\mathbf{p}$ belonging to the surface $\mathcal{S}_1$ as the difference of the value of the attributes a between $\mathbf{p}$ and the nearest point $\mathbf{p}'$ on the surface $\mathcal{S}_2$:

$$\mathrm{d_a}(\mathbf{p}, \mathcal{S}_2) = \|a_i(\mathbf{p}) - a_i(\mathbf{p}')\| \tag{1.6}$$

If the nearest point $\mathbf{p}'$ is not unique, i.e., there are several points on the surface $\mathcal{S}_2$ having the same distance to the point $\mathbf{p}$, the attribute deviation is the minimum deviation between $\mathbf{p}$ and those candidates. Note that this attribute deviation measure is not symmetric, the nearest point to $\mathbf{p}'$ on the surface $\mathcal{S}_1$ is not necessarily $\mathbf{p}$ and so $\mathrm{d_a}(\mathbf{p}, \mathcal{S}_2)$ is not necessarily equal to $\mathrm{d_a}(\mathbf{p}', \mathcal{S}_1)$.

This metric captures only the effective attribute deviation, and ignores the geometric deviation, as shown in Figure 1.2 **(f)** and **(h)**. It can thus be used to detect precisely regions with high attribute distortions. However, it is difficult to combine this attribute deviation with another geometric error as these two measures have different units and scales. In addition, the point-to-point mapping constructed by the attribute deviation is neither continuous nor



(a) Original mesh **(b)** Simplified (low deviation) **(c)** Simplified (high deviation) **(d)** Geometric deviation

(e) Cohen's deviation for **(b)** **(f)** Roy's deviation for **(b)** **(g)** Cohen's deviation for **(c)** **(h)** Roy's deviation for **(c)**

Figure 1.2 – Comparison between Cohen's texture coordinates deviation ((**e**) and (**g**)) and Roy's attribute deviation ((**f**) and (**h**)). Roy et al. [2004]

bijective. Thus to capture all the differences, the attribute deviation should be computed in both directions (from $\mathcal{S}_1$ to $\mathcal{S}_2$ and from $\mathcal{S}_2$ to $\mathcal{S}_1$). However in that case, it does not provide a single mapping but rather relies on two potentially conflicting mappings.

Cohen et al. [1998] assume that the attributes are decoupled from the geometry and stored into 2D textures. This way, they reduce the problem of measuring the attribute deviation to measuring the texture coordinates deviation. Using correspondences defined by the common parametrization, they define the texture coordinate deviation for a single point as the distance between this point and the point with the same texture coordinates on the other surface. The deviation between two surfaces $\mathcal{S}_1$ and $\mathcal{S}_2$ is then define as:

$$d_{tex}(\mathcal{S}_1, \mathcal{S}_2) = \sup_{(u,v)} \|\mathcal{M}_1(u,v) - \mathcal{M}_2(u,v)\| \tag{1.7}$$

where $\mathcal{M}_1 : (u,v) \to (x,y,z)$ for $\mathcal{S}_1$ and $\mathcal{M}_2$ for $\mathcal{S}_2$ are the functions which associate to every texture coordinates their corresponding 3D position.

As illustrated in Figure 1.2, this texture coordinate deviation has for main advantage to capture at the same time the geometric and the parametric distortion between two models. Indeed, this formulation naturally combines the geometry and the parametrization without unit nor scaling problems.

1.2.3 Perceptual Metrics

As explained above, error metrics can be used to select the appropriate LOD such that the visual differences between the simplified model and the original one are not visible. All the error metrics presented so far compute some form of objective distances in object or attribute space to quantify the difference between the two meshes, but they may not match well the perceived visual difference. Thus several approaches strive to quantify the perceptual difference using a variety of known properties of the human visual system.

Since the goal of rendering is to produce an image, it makes sense to use image quality metrics to evaluate the visual error introduced by simplification algorithms. To do so, many approaches compare a pair of images obtained by rendering the simplified and original mesh from a given view point to drive the simplification process [Lindstrom and Turk, 2000; Lindstrom, 2000; Luebke and Hallen, 2001; Williams et al., 2003; Qu and Meyer, 2008]. Reddy [1997], Dumont et al. [2003], Zhu et al. [2010] analyze pre-rendered images at preprocessing time to then drive the selection of the appropriate LOD at render time. The main advantage of this type of approaches is that they can take advantage of the vast literature on perceptual metrics in image processing, and more importantly it takes implicitly into account mesh attributes (colors, normals, ...), and the illumination environment.

However, image-based approaches suffer from several limitations. First, rendering multiple images many times throughout the simplification process is very expensive and slows down considerably the computation. Second, image-based measures depend on a very large set of rendering parameters (lights, view-points, shading models, etc). Yet, for interactive applications, the error metric should be as independent of these parameters as possible. Indeed those can greatly vary for interactive applications, and measuring the error for all possible combinations of all possible values is not feasible. Finally, when using animated meshes, there is no guarantee that the actual visual quality is correlated with the measured static image quality.

Other approaches rely directly on the perceptual analysis of 3D information. Lee et al. [2005], Song et al. [2014] respectively use multi-scale curvature-based analysis or spectral processing to find salient regions, visually interesting regions, on a mesh. [Lavoué, 2011; Wang et al., 2012; Torkhani et al., 2014; Dong

et al., 2014] provide perceptual measures also based on curvature. However, all these approaches do not take into account surface attributes.

1.2.4 Summary

To choose the most adapted level of details, the ideal error metric should quantify with maximal accuracy the geometric and attribute error introduced by a given LOD for a given viewing distance and direction. Perceptual image-based approaches are appealing at first glance as they evaluate directly the final result of the rendering and thus indirectly take into account all these factors. However, they are limited to static meshes, depends on other parameters that are difficult to track in real-world applications, and it is difficult to summarize in a compact representation these metrics. In addition, it is difficult to use them to guide the simplification process, due to their expensive computation.

Other approaches compute an error for a single aspect of the model (geometry or attributes). They are often convenient to formulate optimization problems and thus easy to use to guide the simplification. Combining a geometric and an attribute error metric to obtain an estimation of the visual simplification error is possible, but requires arbitrary weights. Those weights are often chosen empirically and thus do not provide a generic and elegant way to measure the simplification error.

Conversely, texture coordinate deviation, as proposed by Cohen et al. [1998], elegantly combines the geometric and attribute errors. Minimizing this error will guarantee that the object covers the correct pixels on screen and that each pixel has the right attributes. We propose an extension of this metric to take into account the view direction in Chapter 2.

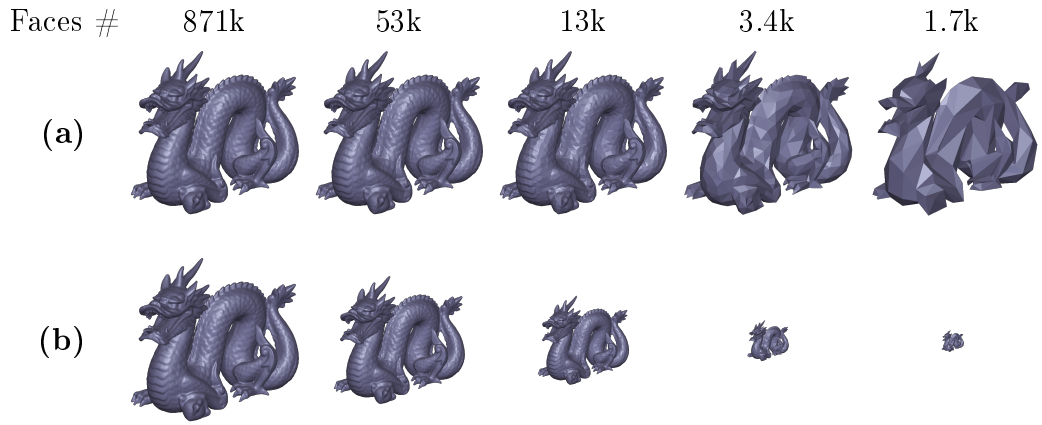


Figure 1.3 – Discrete Level of details — **(a)** Models of the Standford Dragon at different resolutions. **(b)** According to the view distance, it possible to use a coarser version of the model without affecting the visual appearance.

1.3 Level of Details

To achieve real-time performances, highly detailed geometric models are usually rendered using level-of-details (LOD) [Luebke et al., 2002]. The initial detailed model is iteratively simplified during a preprocessing stage, and stored in a compact hierarchical representation. At render time, according to a view-dependent criterion, the appropriate level in the hierarchy is selected. Choosing this criterion properly greatly reduces the complexity of the scene while preserving the visual appearance.

In the previous section, we have seen different approaches to evaluate the error introduced by a given LOD. We now discuss the benefits and limitations of the existing LOD representations (Section 1.3.1), and then present the different simplification algorithms proposed to generate LOD (Section 1.3.2). Finally, we review the metrics used to guide the simplification process (Section 1.3.3).

1.3.1 LOD Representation

The simplest approach to store and represent LOD consists in storing several meshes at different resolutions to represent the same object (Figure 1.3**(a)**). We call this approach DLOD for discrete level of detail. These meshes are precomputed using any polygon reduction technique. At render time, the appropriate version of the object is displayed according to a view-dependent criterion (Figure 1.3**(b)**). There is no transition mechanism between the different levels. As the view changes, the meshes are simply swapped to satisfy the view error criterion. This approach is still widely used in the game industry because it is simple, and more importantly because it is GPU efficient and

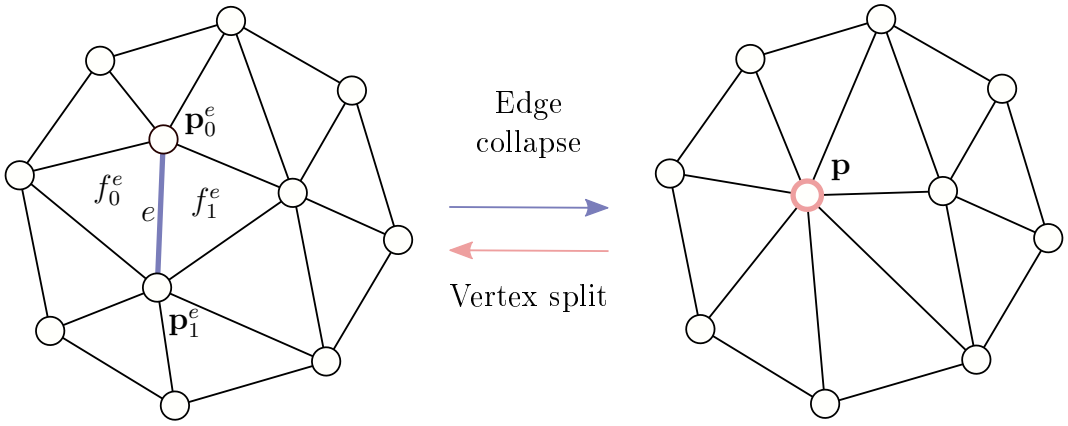


Figure 1.4 – Example of an edge collapse and its inverse operation: vertex split.

easily fit in any rendering pipeline.

However, the lack of any transition between levels lead to highly visible popping artifacts unless the view error criterion is chosen such that there is no visual differences when the changes occur. In practice, however, such criterion does not allow enough simplification of the scene and thus does not sufficiently accelerate the rendering. In addition, DLOD offers a poor control over the model resolution as it is limited to few precomputed meshes. Indeed, it does not support selective refinement, i.e., addition of details only in desired areas of the model. For large models, only a small part of the mesh can be close to the view. Selective refinement allows to refine only those areas, leaving the rest of the model at a low resolution and thus improving even more the rendering performance.

To overcome these limitations, Hoppe [1996] designed a continuous-hierarchical representation for level of details called Progressive Meshes (PM). The input detailed mesh $\mathcal{M}^n$ is simplified using a series of n reversible operations into a coarse mesh $\mathcal{M}^0$. Using the inverse operations, the coarse mesh $\mathcal{M}^0$ can be refined into a series of meshes $\mathcal{M}^1$,$\mathcal{M}^2$,... and up to the original mesh $\mathcal{M}^n$. Therefore, rather than storing a discrete set of LOD, he store the coarse mesh $\mathcal{M}^0$ and the set of n operations that indicates how to refines $\mathcal{M}^0$ back into the original mesh $\mathcal{M}^n$, providing a quite efficient, lossless, and continuous representation.

Progressive meshes use two operations: edge collapse and vertex split. Edge collapse has been proposed by Hoppe et al. [1993] and is one of the most used simplification operator. As illustrated in Figure 1.4, this operator collapses an edge $e = (\mathbf{p}_0^e, \mathbf{p}_1^e)$ into a single vertex $\mathbf{p}$. This yields the suppression of the edge e and of the two adjacent faces f_0^e and f_1^e if they exists. The inverse operation of a edge collapse is called vertex split. It splits a vertex $\mathbf{p}$ in two

vertices $\mathbf{p}_0^e, \mathbf{p}_1^e$ and adds the edge $(\mathbf{p}_0^e, \mathbf{p}_1^e)$ and the two faces f_0^e and f_1^e.

A nice property of the edge collapse and the vertex split is that they naturally enable smooth visual transition between two successive levels of details, thus avoiding all the popping artifacts of the discrete LOD techniques. Smooth transitions are achieved by doing a linear interpolation over time from the split vertex position to the final positions of the two generated vertices.

The PM representation also supports real-time selective refinement [Xia and Varshney, 1996; Hoppe, 1997; El-Sana and Varshney, 1999], i.e., adding details only in desired areas of the model. There are, however, constraints on the order in which the vertex split operations can be applied: the candidate vertex for this split must exist, and other neighbor preconditions have to be satisfied. From those constraints, they establish a hierarchy describing the dependencies between the vertex split operations. Depending on the choice of those constraints, the hierarchy will have a different structure: a tree [Xia and Varshney, 1996], a binary tree [El-Sana and Varshney, 1999], or a forest [Hoppe, 1997]. Each *front* in those structures corresponds to a different possible level of details. Then by modifying progressively this front it is possible to refine or simplify the mesh in the desired areas.

Liang Hu et al. [2010] present different dependency conditions enabling an efficient implementation of Progressive Mesh entirely on programmable graphics hardware. However their implementation is limited to halfedge collapses and do not support smooth temporal transitions.

1.3.2 Level of details generation

The typical way to generate LOD is to use some polygon reduction technique to create a hierarchy of meshes with decreasing number of polygons. Polygon reduction is an optimization process reducing the number of polygons while preserving as much as possible the visual appearance. To do so, it tries to preserve the overall shape, the volume and the boundaries of the model. Polygon reduction approaches vary widely in their support of attributes. Some reduction methods, especially the earliest ones, offer no support of the attributes, and are based on purely geometric error metrics. Other algorithms use some form of interpolation to carry the attributes of the original model to the simplified one without taking them into account during the course of the optimization. Most sophisticated methods try to estimate the attribute error introduced by the reduction, and combine it with the geometric error to guide the optimization.

Polygon reduction techniques can be further separated into two categories: local and global ones. The techniques that belong to the first category rely on local simplification operators such as edge collapses which are iteratively applied to reduce the number of polygons. The global approaches, on the other

hand, operate over large regions or even the entire mesh, and simplify a large number of primitives in a single step.

Global reduction techniques comprise a large variety of approaches. For example, vertex clustering merges all the vertices within a given cell, or cluster, into a single representative vertex. The representative vertex is chosen from one of the collapsed vertices, or computed as some form of average over all the vertices within the cell. The clustering is based on geometric proximity, it can be done using a rectilinear grid [Rossignac and Borrel, 1993], an octree [Luebke and Erikson, 1997], floating-cells [Low and Tan, 1997], or Centroidal Voronoi Diagrams [Valette and Chassery, 2004]. The level of simplification depends on the cell resolution: larger cells yield high simplification rates at low computation costs. Other approaches embed the mesh in a volumetric grid, then apply simplification operators in the volumetric domain, and finally reconstruct a triangle mesh using iso-surface extraction [He et al., 1996; Nooruddin and Turk, 2003]. However, global approaches suffer generally from a major shortcoming: As the reduction happens in a single step, they cannot provide a continuous representation across levels.

Local algorithms start from the original mesh and iteratively apply simplification operators until a certain stopping condition is met. They thus naturally enable continuous transitions for LOD.

For most operators, applying them in different orders lead to different results. Finding the best order in which the operations need to be applied to minimize the differences between the resulting and the original mesh is likely a NP-hard problem. As finding the optimal solution is not feasible in a reasonable time, most approaches have opted for greedy algorithms. Based on some heuristics, these approaches make the locally optimal choice at each step. Although it is not guaranteed to produce the best solution, it may result in a good approximation of the optimal solution in a reasonable amount of time.

The pseudocode of Algorithm 1 illustrates the general principle of such approaches. Using some error metric, the algorithm computes the cost of each possible operation, and stores them in a priority queue. Then, as long as the queue is not empty or the stopping condition is not met, the operation with minimum cost is pulled from the queue and applied to the mesh. Applying an operation affects the geometry and the connectivity of the mesh, thereby the costs of adjacent operations need to be updated.

Simplification algorithms can have different stopping conditions. The user can provide a fidelity criterion with respect to the input mesh that the simplified mesh must satisfy. The simplification algorithm attempts to minimize the

Algorithm 1 Greedy simplification algorithm

```
1: Input: Original mesh M, Stopping criterion crit
2: H ← Empty heap
3: for each possible operation op do
4:     c ← Compute cost of op
5:     Insert (c,op) in H
6: end for
7: while H not empty and M satisfies crit do
8:     op ← Minimum of H
9:     Remove op from H
10:    Apply op on M
11:    for each neighbor operation op_i do
12:        c_i ← Compute cost of op_i
13:        Update (c_i,op_i) in H
14:    end for
15: end while
```

number of triangles while respecting the fidelity constraint. The fidelity criterion is usually defined as a measure of the differences between the simplified mesh and the original model. This strategy is better suited for applications in which the visual fidelity is the central component.

Otherwise the stopping condition can be defined as a triangle-budget. In this case, the algorithm tries to minimize the error metric while not exceeding the target number of triangles. Since this method generates a fixed number of triangles, it is easier to obtain a fixed frame rate. It is thus best suited for interactive applications and often used for real-time LOD.

The simplification operators and the error metrics are the main components of simplification algorithms. In general, the choice of the error metric is highly dependent on the operators. Next, we present the different operators used for mesh simplification. Each of these operators simplifies the geometry and the connectivity in a local region, removing a small number of primitives (vertex, edge or face). Most of them are designed specifically for triangle meshes. It is thus often required to triangulate the input mesh in a preprocessing step, if it is not composed exclusively of triangles.

Vertex Removal

The most straightforward operator when simplifying geometry is the vertex removal operator [Schroeder et al., 1992; Soucy and Laurendeau, 1996; Klein et al., 1996]. It consists in removing a single vertex and its adjacent faces, and then triangulating the resulting hole (Figure 1.5 **(a)**). Each vertex removal

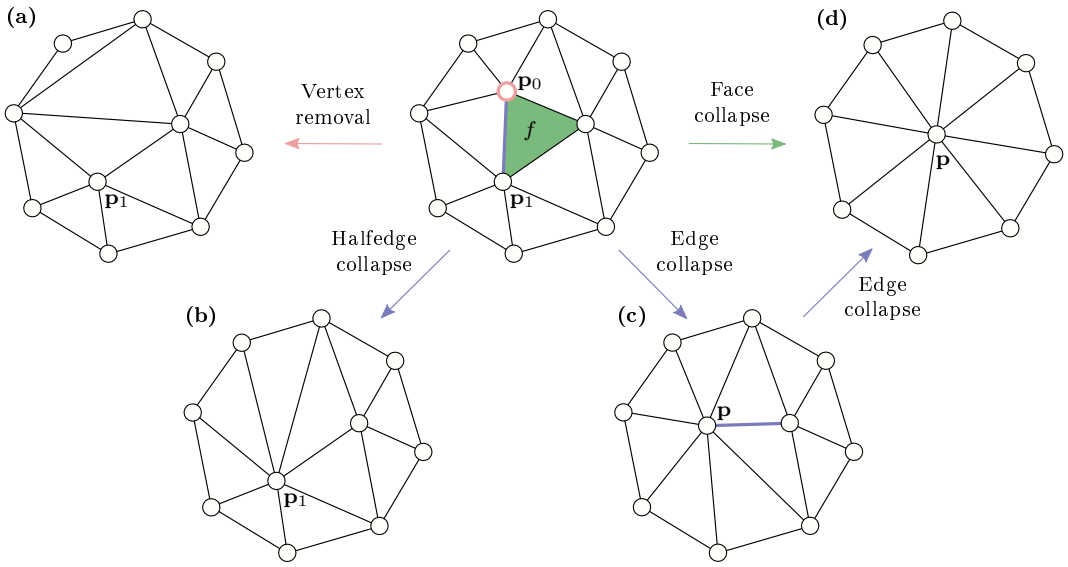


Figure 1.5 – Local simplification operators — **(a)** Vertex removal: The vertex $\mathbf{p}_0$ is removed and the resulting hole is triangulated. **(b)** Halfedge collapse: The halfedge $(\mathbf{p}_0, \mathbf{p}_1)$ is collapsed to $\mathbf{p}_1$ **(c)** Edge collapse: The two vertices $\mathbf{p}_0$ and $\mathbf{p}_1$ are merge into a new vertex $\mathbf{p}$. **(d)** Face collapse: The three vertices of the face f are collapsed into the new vertex $\mathbf{p}$. The same result can be obtain by collapsing a second edge from **(c)**.

operation reduces the complexity of a mesh by one vertex and two faces. Apart from the removal order, the triangulation of the holes is the only way to affect the geometric quality of the simplified mesh. Therefore it is important to create triangles with good aspect ratios and that approximate the original surface as closely as possible.

Edge Collapse

As presented in Section 1.3.1, edge collapse merges the two extremities of an edge into a single vertex, removing one vertex and two faces. The position of the resulting vertex is usually chosen as the position minimizing the cost metric. Although the edge collapse operator is simple to implement, it can introduce mesh foldover or nonmanifold edges. Figure 1.6**(a)** shows an example of contraction introducing a foldover. When contracting the edge (p_0^e, p_1^e), the orientation of the face f gets flipped. This can results in visual artifacts, such as lighting and textures discontinuities. Therefore edge collapse operations introducing foldover are removed from the priority queue or get prohibitive costs. To detect foldover, it is necessary to keep track of the normals before and after the contraction. If there is a large variation of the normal direction, usually greater than 90°, then a foldover has occurred.

Nonmanifold edges are edges with three or more adjacent triangles. They

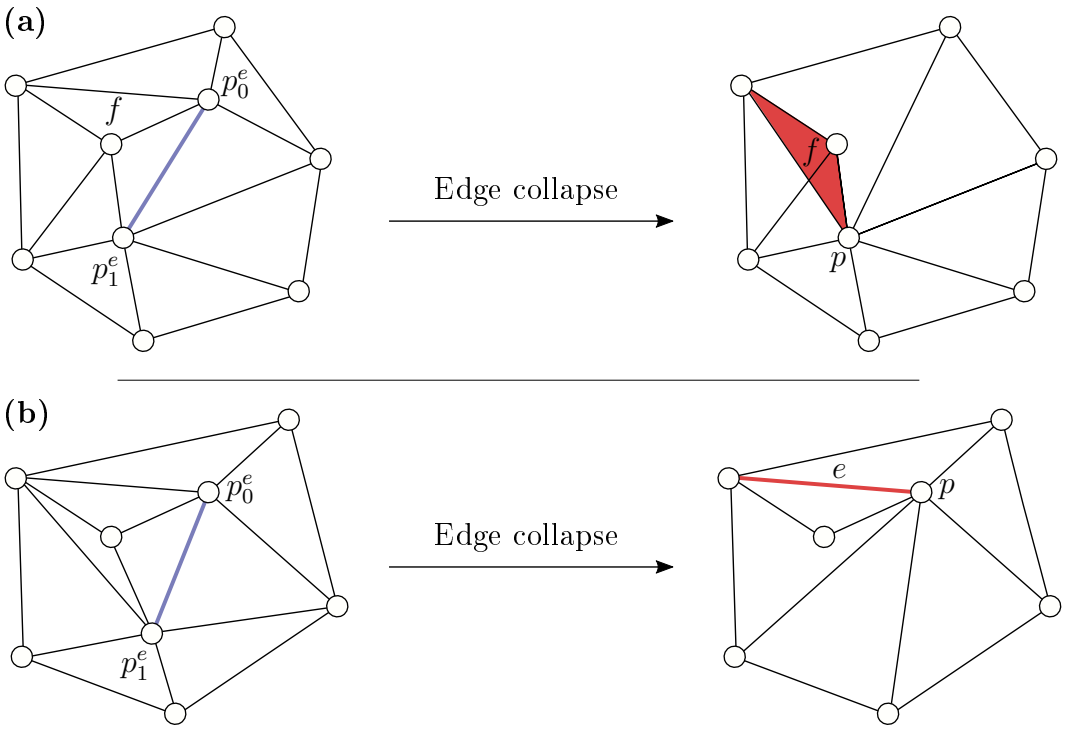


Figure 1.6 – Invalid edge collapse operations — **(a)** When contracting the edge $(\mathbf{p}_0^e, \mathbf{p}_1^e)$, a foldover appears: The face f flips direction. **(b)** The contraction of $\mathbf{p}_0^e$ and $\mathbf{p}_1^e$ leads to a nonmanifold edge e.

appear when contracting an edge whose extremities share three or more neighboring vertices (as shown in Figure 1.6**(b)**). Since many algorithms rely on manifold connectivity, it is crucial to avoid introducing such edges.

Halfedge Collapse

Halfedge collapse consists in collapsing the edge to one of its end points. It can be seen as a simpler version of edge collapse. It generally gives poor quality results compared to edge collapse since the locally simplified geometry is not optimized.

Vertex-Pair Collapse

The vertex-pair collapse is an extension of the edge collapse operation to unconnected vertices [Garland and Heckbert, 1997]. Although no face are removed when merging two unconnected vertices, it allows to close holes and gaps. This is especially useful to simplify meshes with a lot of unconnected components or holes when the topology is less important than the overall shape. Typical examples include foliage.

A mesh with n vertices can accept potentially n^2 vertex-pair contractions.

It is generally too costly to consider all existing vertex-pair contractions. Most of the simplification algorithms using vertex-pair collapses therefore limit them to a subset of pairs that are close in space.

Triangle Collapse

Similar to edge collapse, the triangle collapse operator [Hamann, 1994; Gieng et al., 1997] simplifies a mesh by collapsing the three vertices of a triangle to a single vertex. The position of the new vertex is also the result of some form of optimization. A triangle collapse is equivalent to two edge collapses (as shown on Figure 1.5) and thereby provides little practical benefits compared to edge collapse.

Edge Flip

Although edge flip is not technically a simplification operator, it is often used in combination of those to improve the mesh quality. It consists in replacing the two triangles adjacent to an edge with two new triangles. In addition to potentially improve the mesh quality, edge flips can be used to solve foldover introduced by edge collapses. However, edge flips are usually not compatible with continuous LOD, as, e.g., Progressive Meshes.

1.3.3 Metrics for Mesh Simplification

Mesh simplifications are driven by error metrics. In general, those metrics try to estimate the error introduced by a single simplification operation. They are involved in two kinds of optimizations during the simplification. The first one is the cost used for ordering the simplification operations. The second use case is to drive the optimization of the locally simplified geometry, such as the position of the new vertex after an edge collapse. These two optimizations are tightly related, and most of the time they share a great deal of code. For instance, the cost is often defined as the residual of the geometric optimization, whereas the latter strive to minimize this residual. We now present several of those metrics.

Distance to average plane

In the context of vertex removal, Schroeder et al. [1992] estimate the geometric deviation through the distance between the vertex to be removed and the average plane of the adjacent faces. The average plane is computed using the normals $\mathbf{n_i}$, centers $\mathbf{c_i}$, and areas A_i of the adjacent faces $\mathcal{T}$:

$$\mathbf{n} = \text{normalize}\left(\sum_{i\in\mathcal{T}} \mathbf{n_i} A_i\right),$$
$$\mathbf{c} = \frac{\sum_{i\in\mathcal{T}} \mathbf{c_i} A_i}{\sum_{i\in\mathcal{T}} A_i}$$

The distance of the vertex $\mathbf{p}$ to the average plane is then:

$$E_{\text{avg}}(\mathbf{p}) = |\mathbf{n}^T(\mathbf{p} - \mathbf{c})| \tag{1.8}$$

This error criterion only considers the deviation from the actual surface and not from the input mesh. Therefore, it is difficult to have any estimation on the error accumulated during the overall simplification.

Surface to surface distance

Soucy and Laurendeau [1996], Klein et al. [1996] and Hu et al. [2017] estimate the surface-to-surface distance by maintaining a mapping between the original mesh and the simplified one. Each deleted vertex is linked to the closest face of the simplified mesh. Soucy and Laurendeau [1996] define the cost of a vertex removal as the maximum distance from either a deleted neighbor vertex or the vertex itself to the retriangulated surface. This criterion returns an underestimation of the Hausdorff distance because the L^∞ surface to surface distance might not be located on a vertex of the original mesh. Klein et al. [1996] use this mapping to compute the actual Hausdorff distance. Hu et al. [2017] estimate the Hausdorff distance using a uniform sampling per face with additional samples on edges and vertices.

For all those approaches, the computation time is, however, proportional to the number of deleted vertices. The more the simplification proceeds, the higher the complexity of each cost estimation increases.

Maximum Supporting Plane Distance

Ronfard and Rossignac [1996] point out that every vertex is at the intersection of all the planes defined by its adjacent faces. Following this observation, they associate a set of planes $\mathcal{P}$ with each vertex, and define the error for a vertex $\mathbf{p}$ as the maximum distance from $\mathbf{p}$ to the planes in its set. Let $(\mathbf{n}, d)$ be a plane in $\mathcal{P}$ where $\mathbf{n}$ is the unit normal of the plane, and d is the offset from the origin. Then vertices belonging to the plane satisfy the equation $\mathbf{n}^T\mathbf{p} + d = 0$. This gives the following formulation for the error metric:

$$E_{\text{spd}}(\mathbf{p}) = \max_{(\mathbf{n},d)\in\mathcal{P}} (\mathbf{n}^T\mathbf{p} + d)^2 \,. \tag{1.9}$$

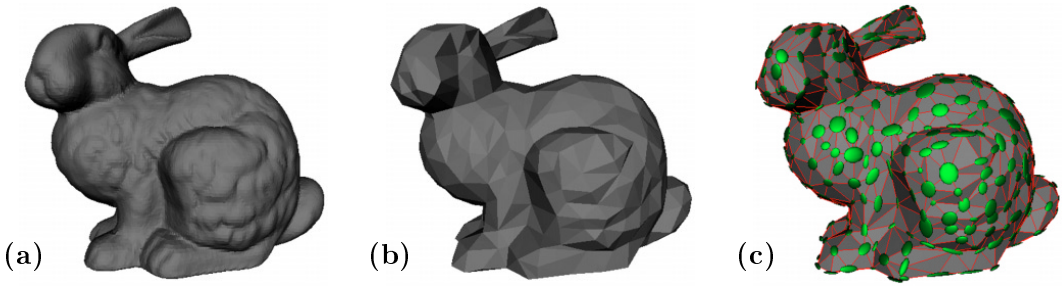


Figure 1.7 – Quadric Error Metric [Garland and Heckbert, 1997] — **(a)** Original bunny model with 69,451 triangles. **(b)** Quadric error displayed as an ellipsoid for each vertex. **(c)** A simplified version using only 1,000 triangles.

Those sets are initialized from the adjacent faces of each vertex, therefore the initial error at each vertex is null. When an edge is collapsed, the two sets are merged. This measure provides an efficient heuristic for ordering edge collapse operations, but the set of planes need to be tracked explicitly during the simplification. In addition, this method does not provide any guarantees about the maximum and the average geometric deviation introduced by the simplification.

Quadric Error Metric

Based on the same observation than Ronfard and Rossignac [1996], Garland and Heckbert [1997] define their error as the sum of squared distances of $\mathbf{p}$ to all planes of the set $\mathcal{P}$:

$$\begin{aligned} E_{\text{QEM}}(\mathbf{p}) &= \sum_{t=(\mathbf{n},d)\in\mathcal{P}} (\mathbf{n}^T\mathbf{p} + d)^2 \\ &= \sum_{t=(\mathbf{n},d)\in\mathcal{P}} (\mathbf{p}^T\mathbf{n}\mathbf{n}^T\mathbf{p} + 2d\mathbf{n}^T\mathbf{p} + d^2) \\ &= \sum_{t=(\mathbf{n},d)\in\mathcal{P}} e_{\text{QEM}}(\mathbf{p}, t) \ , \end{aligned} \tag{1.10}$$

where e_{QEM} is the error for a single plane.

e_{QEM} is a quadratic form: $\mathbf{p}^T\mathbf{A}\mathbf{p} + 2\mathbf{b}^T\mathbf{p} + c$ (Figure 1.7). Using homogeneous coordinates, this quadric form can be represented by a single 4×4 symmetric matrix $\mathbf{Q}_{(\mathbf{n},d)}$:

$$E_{\text{QEM}}(\mathbf{h}) = \mathbf{h}^T \left(\sum_{(\mathbf{n},d)\in\mathcal{P}} \mathbf{Q}_{(\mathbf{n},d)} \right) \mathbf{h} = \mathbf{h}^T \mathbf{Q} \mathbf{h} \ , \tag{1.11}$$

$$\text{with } \mathbf{Q}_{(\mathbf{n},d)} = \begin{bmatrix} \mathbf{A} & \mathbf{b} \\ \mathbf{b}^T & c \end{bmatrix} = \begin{bmatrix} \mathbf{n}\mathbf{n}^T & d\mathbf{n} \\ d\mathbf{n}^T & d^2 \end{bmatrix} .$$

Here $\mathbf{h}$ is the homogeneous vector $\begin{bmatrix} \mathbf{p}^T & 1 \end{bmatrix}^T$. This formulation is very convenient because it shows that we need a single matrix Q to find the sum of the squared distance of an entire set of planes. Following this observation, they implicitly keep track of the set of planes by storing such a matrix Q for each vertex. When contracting two vertices $\mathbf{p}_1$ and $\mathbf{p}_2$, the quadric associated to the new vertex is simply defined as the sum of the two quadrics of the collapsed vertices $\mathbf{Q} = \mathbf{Q}_1 + \mathbf{Q}_2$. This may introduce some imprecision into the error measurement because a plane may be counted multiple times. However they advocate against a more complicated method correcting this problem, as it will provide little benefit.

Since the error function E_{QEM} is quadratic, finding its minimum is a linear problem. The new vertex position $\mathbf{p}$ which minimize E_{QEM} is the point for which all the derivatives vanish: $\frac{\partial E_{\text{QEM}}}{\partial x} = \frac{\partial E_{\text{QEM}}}{\partial y} = \frac{\partial E_{\text{QEM}}}{\partial z} = 0$. This yields to the following linear system:

$$\begin{gathered} \mathbf{Q_{11}}\ \mathbf{p} = -\mathbf{q_{12}} \\ \text{with } \mathbf{Q} = \begin{bmatrix} \mathbf{Q_{11}} & \mathbf{q_{12}} \\ \mathbf{q_{12}}^T & q_{22} \end{bmatrix} \end{gathered} \tag{1.12}$$

The main computational cost in solving this minimization is to invert the 3×3 matrix $\mathbf{Q_{11}}$. If the planes composing the quadric are almost parallel, the matrix may not be invertible. Even if the matrix is numerically invertible, the solution might be far away from the neighborhood. In these cases, they use a fall-back strategy such as selecting one of the two end points or the midpoint of the collapsed edge.

The mesh boundaries are preserved by the mean of additional proxy planes. For each boundary edge, they add to the endpoints a plane orthogonal to the adjacent face, weighted by a large factor.

Salinas et al. [2015] use a similar idea to make the greedy decimation aware of the general structure of the mesh. A set of planar proxies detected in a preprocessing step are converted and added to the quadrics in order to guide

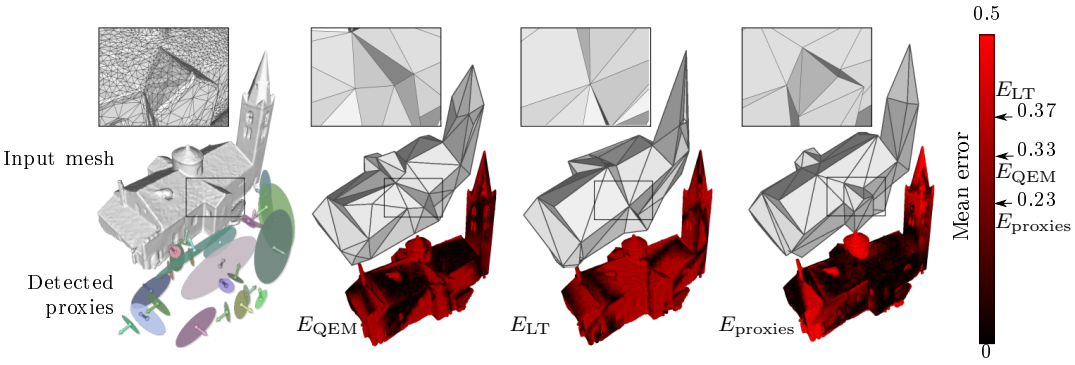


Figure 1.8 – Structure-aware mesh decimation [Salinas et al., 2015] — A set of planar proxies are converted and added to the quadrics in order to guide the simplification. This results in a better preservation of the structure of the mesh.

the simplification (Figure 1.8). However, this is not enough on its own to guarantee that the simplification converges towards the coarse-scale structures. Additional structure-preserving rules, prohibiting some edge collapses, need to be added to recover and preserve the structures defined by the proxies.

Quadric Error Metric with attributes

Several approaches have been proposed to extend the quadric error metric to support surfaces with attributes. Garland and Heckbert [1998] extend the quadric to a n-dimensional space. The method remains exactly the same. They construct a quadric to measure the squared distance of any point in $\mathbb{R}^n$ to a plane. For a triangle $(\tilde{\mathbf{p}}_0, \tilde{\mathbf{p}}_1, \tilde{\mathbf{p}}_2)$, vertices would have the form $\tilde{\mathbf{p}} = \begin{bmatrix} x & y & z & a_0 & a_1 & \dots & a_m \end{bmatrix}^T$, with $a_0, a_1, ..., a_m$ the attributes of the surface. Then to characterize the plane of the triangle $(\tilde{\mathbf{p}}_0, \tilde{\mathbf{p}}_1, \tilde{\mathbf{p}}_2)$, we need to compute two orthogonal vectors lying in this plane. Denoting $\tilde{\mathbf{p}}_{\mathbf{ij}}$ the vector $\tilde{\mathbf{p}}_{\mathbf{j}} - \tilde{\mathbf{p}}_{\mathbf{i}}$, these two vectors are obtained using the following equations:

$$\mathbf{v_1} = \frac{\tilde{\mathbf{p}}_{\mathbf{01}}}{\|\tilde{\mathbf{p}}_{\mathbf{01}}\|} , \tag{1.13}$$

$$\mathbf{v_2} = \frac{\tilde{\mathbf{p}}_{\mathbf{02}} - (\mathbf{v_1}^T \tilde{\mathbf{p}}_{\mathbf{02}})\mathbf{v_1}}{\|\tilde{\mathbf{p}}_{\mathbf{02}} - (\mathbf{v_1}^T \tilde{\mathbf{p}}_{\mathbf{02}})\mathbf{v_1}\|} . \tag{1.14}$$

Let $\tilde{\mathbf{p}}$ be a point in $\mathbb{R}^n$. We can compute e^n_{QEM} the squared distance to the plane $\tilde{t}$ defined by $\mathbf{v_1}$ and $\mathbf{v_2}$ as:

$$e^n_{\text{QEM}}(\tilde{\mathbf{p}}, \tilde{t}) = \|\tilde{\mathbf{p}} - \tilde{\mathbf{p}}_0\|^2 - ((\tilde{\mathbf{p}} - \tilde{\mathbf{p}}_0)^T \mathbf{v_1})^2 - ((\tilde{\mathbf{p}} - \tilde{\mathbf{p}}_0)^T \mathbf{v_2})^2 . \tag{1.15}$$

As before, we can rewrite it as a quadric form:

$$e^n_{\text{QEM}}(\tilde{\mathbf{p}}, \tilde{t}) = \tilde{\mathbf{p}}^T \mathbf{A} \tilde{\mathbf{p}} + 2\mathbf{b}^T \tilde{\mathbf{p}} + c \tag{1.16}$$

$$\begin{aligned} \text{where } \mathbf{A} &= \mathbf{I} - \mathbf{v_1}^T \mathbf{v_1} - \mathbf{v_2}^T \mathbf{v_2} \ , \\ \mathbf{b} &= (\tilde{\mathbf{p}}_\mathbf{0}^T \mathbf{v_1}) \mathbf{v_1} + (\tilde{\mathbf{p}}_\mathbf{0}^T \mathbf{v_2}) \mathbf{v_2} - \tilde{\mathbf{p}}_\mathbf{0} \ , \\ c &= \tilde{\mathbf{p}}_\mathbf{0}^T \tilde{\mathbf{p}}_\mathbf{0} + (\tilde{\mathbf{p}}_\mathbf{0}^T \mathbf{v_1})^2 + (\tilde{\mathbf{p}}_\mathbf{0}^T \mathbf{v_2})^2 \ . \end{aligned}$$

This generalized quadric has the same structure as the previous 3-dimensional quadric but here $\mathbf{A}$ is a $n \times n$ matrix and $\mathbf{b}$ is an n-vector. The generalized quadric error is then defined as:

$$E^n_{\text{QEM}}(\tilde{\mathbf{p}}) = \sum_{\tilde{t} \in \mathcal{P}} e^n_{\text{QEM}}(\tilde{\mathbf{p}}, \tilde{t}) \ . \tag{1.17}$$

The optimal placement $\tilde{\mathbf{p}}$ minimizing $E^n_{\text{QEM}}(\tilde{\mathbf{p}})$ is computed by proceeding exactly as for the standard QEM. It is worthwhile to mention that the size of the quadric matrix grows quadratically with the number of attributes. Consequently, handling too many properties can lead to an important memory cost as well as an increased computational time.

One important constraint of this approach is that the scale of the attributes matter. Hence, to obtain consistent results, the 3D model has to be scaled so that the positions and the various attributes have the same scale. The scale of each attribute determines how it will be preserved to the detriment of the other attributes.

Hoppe [1999] proposed a different approach to simplify meshes with attributes. He defines the error metric as the standard quadric error $e_{\text{QEM}}(\mathbf{p}, t)$ and a sum of attribute errors $e_{\hat{a}_j}(\tilde{\mathbf{p}}, a_j)$:

$$E^a_{\text{QEM}}(\tilde{\mathbf{p}}) = \sum_{t \in \mathcal{P}} e^a_{\text{QEM}}(\tilde{\mathbf{p}}, t) \ , \tag{1.18}$$

$$\text{with } \ e^a_{\text{QEM}}(\tilde{\mathbf{p}}, t) = e_{\text{QEM}}(\mathbf{p}, t) + \sum_{j=1}^{m} e_{\hat{a}_j}(\mathbf{p}, a_j) \ , \tag{1.19}$$

where $\tilde{\mathbf{p}} = (\mathbf{p}^T, a_0, a_1, ..., a_m)$, and $e_{\hat{a}_j}$ is the error for the j-th attribute. The geometric error $e_{\text{QEM}}(\mathbf{p}, t)$ is the squared distance from $\mathbf{p}$ to its projection $\mathbf{p}'$ on the plane.

The attribute error $e_{\hat{a}_j}(\tilde{\mathbf{p}}, a_j)$ is the squared deviation between a_j and the value a'_j linearly interpolated from the vertices of the face at that projected point $\mathbf{p}'$. The attributes are interpolated using a linear function $\hat{a}_j(\mathbf{p}) = \mathbf{g}_j^T \mathbf{p} + d_j$ such that it interpolates the three face vertices and it is constant in

the direction orthogonal to the face $\mathbf{n}_f^T \mathbf{g}_j = 0$. The attribute error $e_{\hat{a}_j}(\tilde{\mathbf{p}}, a_j)$ is thus defined as:

$$\begin{aligned} e_{\hat{a}_j}(\tilde{\mathbf{p}}, a_j) =& (\hat{a}_j(\mathbf{p}) - a_j)^2 \\ =& (\mathbf{g}_j^T \mathbf{p} + d_j - a_j)^2 \ . \end{aligned} \tag{1.20}$$

This gives the following sparse linear system:

$$\begin{bmatrix} \mathbf{n}\mathbf{n}^T + \sum_j \mathbf{g}_j \mathbf{g}_j^T & -\mathbf{g}_1 \cdots - \mathbf{g}_m \\ -\mathbf{g}_1 & \\ \vdots & I \\ -\mathbf{g}_m & \end{bmatrix} \tilde{\mathbf{p}} = - \begin{bmatrix} d\mathbf{n} + \sum_j d_j \mathbf{g}_j \\ -d_1 \\ \vdots \\ -d_m \end{bmatrix} \ . \tag{1.21}$$

Here again the scale of attributes matters. They use a set of weight λ_j to scale the attribute errors relative to the geometric error. Compared to the n-dimensional quadric, the matrix A is relatively sparse and therefore less expensive in memory for an important number of attributes.

Lindstrom-Turk Metric

Lindstrom and Turk [1998] propose a very different approach in the sense that it does not retain any information about the original model during the course of the simplification. After each edge collapse, their goal is to find the optimal position of the new vertex by minimizing several geometric properties. More precisely, they combine several linear equality constraints so that the original volume and shape of the model is locally preserved.

Let $\mathcal{T}_e$ be the set of triangles adjacent to at least one endpoint of the edge e and $\mathbf{p}$ the resulting vertex of the contraction of e. The difference of volume before and after the contraction can be characterized by the set of tetrahedrons formed by a triangle $t = (\mathbf{p}_0^t, \mathbf{p}_1^t, \mathbf{p}_2^t)$ of $\mathcal{T}_e$ and $\mathbf{p}$ (Figure 1.9). The signed volume of a tetrahedron is considered positive if $\mathbf{p}$ is above the plane of t and negative if $\mathbf{p}$ is below this plane. Then, by summing the contribution of each tetrahedron, we can compute the change of volume between the two surfaces. To preserve the volume, this sum needs to be equal to zero:

$$\sum_{t \in \mathcal{T}_e} \mathrm{Vol}(\mathbf{p}, \mathbf{p}_0^t, \mathbf{p}_1^t, \mathbf{p}_2^t) = 0 \tag{1.22}$$

The equation (1.22) can be rewritten as the following linear constraint:

$$\left(\sum_{t \in \mathcal{T}_e} \mathbf{n}_t^T \right) \mathbf{p} = \sum_{t \in \mathcal{T}_e} \begin{vmatrix} \mathbf{p}_0^t & \mathbf{p}_1^t & \mathbf{p}_2^t \end{vmatrix} \tag{1.23}$$

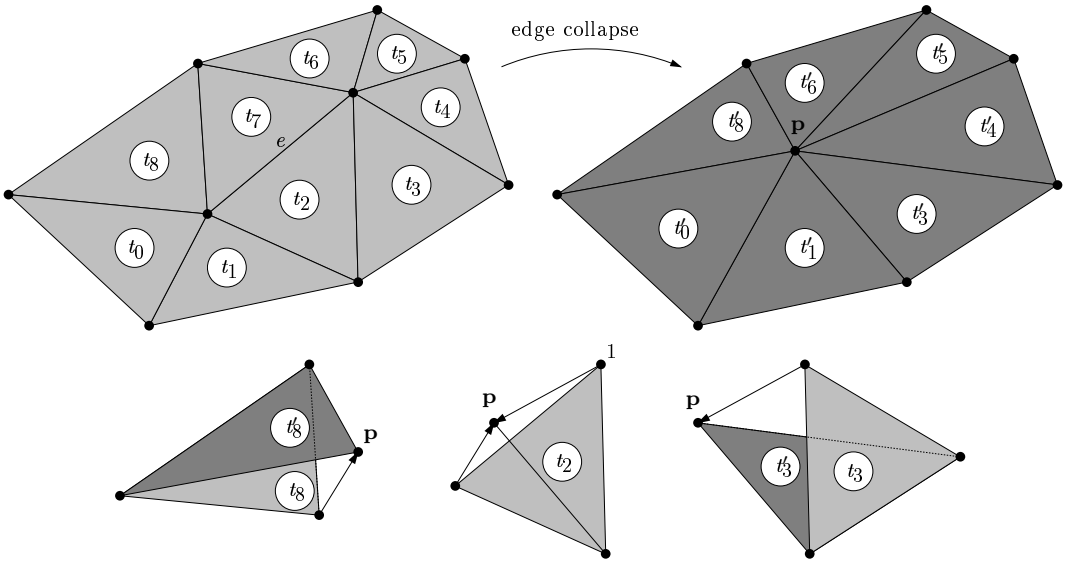


Figure 1.9 – Lindstrom-Turk Metric [Lindstrom and Turk, 1998] — Example of tetrahedral volumes associated with triangles t_2, t_3 and t_8.

where $\mathbf{n}_t$ is the normal of triangle t with magnitude equaled to twice the area of t. Equation 1.23 only restricts $\mathbf{v}$ to a plane. The remaining constraints are defined as the minimization of the unsigned volume of each individual tetrahedron. These constraints attempt to minimize the distance between the two surfaces and lead to the following objective function:

$$\begin{aligned} E_{\text{LT}}(\mathbf{p}) &= \sum_{t \in \mathcal{T}_e} \text{Vol}(\mathbf{p}, \mathbf{p}_0^t, \mathbf{p}_1^t, \mathbf{p}_2^t)^2 \\ &= \sum_{t \in \mathcal{T}_e} (\mathbf{n}_t^T \mathbf{p} - \begin{vmatrix} \mathbf{p}_0^t & \mathbf{p}_1^t & \mathbf{p}_2^t \end{vmatrix})^2 \end{aligned} \tag{1.24}$$

If the triangles of $\mathcal{T}_e$ are coplanar, then equation (1.24) yields an infinite plane of solution and is redundant with the previous constraint. In this case, an additional term is added to optimize the aspect ratio of the remaining triangles.

In addition to these constraints, they propose a set of constraints specifically dedicated to boundaries. These constraints are similar to the volume preservation and optimization detailed before but with the area enclosed by the mesh boundary. Finally, the cost associated to each edge collapse operation is defined using the objective functions from the minimization.

As all constraints are derived directly from the current surface, no information needs to be stored or maintained through the simplification. This method is therefore memory efficient and enables the simplification of very large polygonal models at a fast rate.

1.3.4 Summary

Iterative decimation methods simplify a mesh while preserving as mush as possible the features of the mesh. To do so, we saw that numerous metrics have been developed to guide the simplification process. They are used for ordering the simplification operations but also, in the case of edge collapses, to compute the position of the new vertices. Although all of them aim to minimize the Hausdorff distance, they all have to rely on some heuristics to achieve reasonable performance. A vast majority of them does not take into account the attributes of the mesh. Those supporting attributes, all depend on arbitrary weights to combine the geometric and the attribute errors, and thus do not provide a generic and elegant metric to optimize simultaneously the position and attributes of the mesh.

The output of an iterative decimation can be stored as a compact and continuous representation. In particular, in the context of edge collapse, we saw that Progressive Meshes support selective refinement, allowing to refine the mesh only in desired areas of the model. However even parallel GPU implementations of Progressive Mesh cannot leverage the full power of modern graphics hardware. We show a novel LOD representation adapted to modern GPUs in Chapter 3 and a way to generate it in Chapter 4.

1.4 Displacement Mapping

1.4.1 Definition

Another approach to render highly detailed 3D models is to decompose the model into a low-frequency mesh, often called control mesh, and an offset function defined over the surface of this mesh (Figure 1.10). The offset function is usually discretized and stored in a texture called a *displacement map*. It can be computed using a ray casting process [Lee et al., 2000]: the displacement map is sampled by shooting rays from the control mesh along the surface normals until they intersect the original mesh. Displacement mapping [Cook, 1984] thus refers to the methods of perturbing the coarse surface by the displacement map to reconstruct the original model. It offers a more compact representation than Progressive Meshes, as the connectivity of detailed mesh is not stored. In addition, it enable low-cost animations. Animations are applied to the control mesh and then the detailed surface is reconstructed using the displacement map.

Each texel of the displacement map encodes a displacement. If those displacements are scalar, then the map is called a scalar displacement map or a height map. A height map can be seen as a gray-scale texture. The scalar-displaced surface $\mathcal{S}_h$ generated from a height map is defined in the parametric space as:

$$\mathcal{S}_h(u,v) = \mathcal{P}(u,v) + \mathcal{H}(u,v)\mathcal{N}(u,v) \tag{1.25}$$

$\mathcal{S}_h(u,v)$ is reconstructed by displacing the points of the coarse surface $\mathcal{P}(u,v)$ along the surface normal $\mathcal{N}(u,v)$ by the offset $\mathcal{H}(u,v)$ fetched from the height map.

Displacements can also be stored as 3D vectors. In this case, the map is called a *vector displacement map*. The vector-displaced surface $\mathcal{S}_v$ can be formulated as follows:

Figure 1.10 – Displacement Mapping — A detailed mesh (right) decomposed into a low-frequency mesh (middle) and a offset function (left) defined over the surface of this mesh.

$$\mathcal{S}_v(u,v) = \mathcal{P}(u,v) + \mathcal{R}(u,v)\mathcal{D}(u,v) \tag{1.26}$$

where $\mathcal{D}(u,v)$ is the displacement vector in the local frame of the face and $\mathcal{R}(u,v)$ is the 3×3 matrix transforming vectors from the local tangential frame to the object space. Vector displacements enable the representation of more complex shapes such as folds, which are impossible to reproduce with a single height map. Moreover, the tangential components of the displacement vectors allows potentially to slightly redistribute the samples to better capture the features of the original model.

Displacement mapping was originally proposed for offline rendering pipelines such as RenderMan [Apodaca and Gritz, 1999]. In this context, the surface is tessellated such that the generated triangles are smaller than a pixel, then new generated vertices are moved using the displacement map to reconstruct the details surface [Cook et al., 1987]. This produces very accurate results at the cost of a long computational time. Although such an approach was not suitable for real-time rendering before the introduction of the hardware tessellation, different methods have been proposed to exploit displacement maps in real-time. We start by presenting those methods in Section 1.4.2 and then review displacement mapping methods using hardware tessellation in Section 1.4.3

1.4.2 Pre-Hardware Tessellation

There is two kind of approaches using displacement maps in real-time: per vertex or per fragment displacement mapping In per fragment displacement mapping, the vertex shader transforms only the coarse mesh, and the height map is taken into account when fragment are processed. To ensure all necessary fragment are generated, the coarse mesh must be a bounding mesh of the detailed model. In the fragment shader, it is to late to change the geometry, thus the visibility problem needs to be manually solved for each fragment. This can be done by finding the textures coordinates of the visible point which will be used to fetch color and normal data. However finding rapidly those coordinates is a difficult problem.

Parallax mapping [Kaneko et al., 2001; Welsh and Corporation, 2004; Tatarchuk, 2005] aims to tackle this problem by fetching the height map only once to obtain more accurate texture coordinates according to the value of the height map and the view angle. This method does not compute the actual visible point but rather gives an illusion of depth due to parallax effects as the view changes. Iterative parallax mapping [Premecz, 2006] attempts to improve the solution by iteratively exploring the height field, in a ray marching manner. Other approaches iteratively explore the height field to find the visible point using a binary search [Yerex and Jagersand, 2004], a linear search [McGuire

and McGuire, 2005; Tatarchuk, 2006], some kind of pre-computed data encoding the empty spaces which is then used to find safe step in the ray marching [Kolb and Rezk-Salama, Kolb and Rezk-Salama; Oh et al., 2006; Donnelly, 2005], or a combinations of previous approaches [Policarpo et al., 2005]. All those methods can be classify in two categories : safe or unsafe algorithms. The former find the correct visible point in all cases but unfortunately are very costly. The latter are usually much faster but are not guarantee to return the correct solution. For a comprehensive survey on this techniques see [Szirmay-Kalos and Umenhoffer, 2006].

Per vertex displacement mapping can be done either in the vertex shader or in the geometry shader. The offset is fetched from the displacement map using the vertex texture coordinates, and then it is simply applied to the given vertex. The challenge here is how to generate the appropriate refined geometry. Refining the mesh on the CPU and transmitting the data to the GPU at each frame would be to costly. The ideal approach would be to use only a coarse mesh at the CPU level and then to refine on-the-fly the mesh on the GPU.

In the field of fast mesh refinement, a large amount of work has been done to generate subdivision surfaces [Catmull and Clark, 1998; Doo and Sabin, 1998; Loop, 1987; Kobbelt, 2000], which are the standard modeling primitives in the animation industry. Given a coarse polygonal mesh, subdivision surfaces define a smooth surface computed as the limit of a recursive subdivision scheme. The desired refinement level is obtained by iteratively subdividing the mesh until a certain criterion is met, creating at each step a smoother mesh containing more polygons. This refinement scheme has become very attractive for real-time rendering in authoring tools and video games. Displacement maps can be applied on top of subdivision surfaces, adding high frequency details to the smooth surfaces.

A number of approaches have been proposed to render subdivision surfaces on GPU. Most straightforward implementations [Bunnell, 2005; Shiue et al., 2005; Patney et al., 2009; Weber et al., 2015] iteratively apply the subdivision rules using a GPU compute kernel updating a dense mesh stored in GPU memory. The resulting mesh is then rendered in a second pass. However this approach involves numerous memory transfers between the GPU multiprocessors and the global memory.

The geometry shader [Blythe, 2006] can create new vertices, and triangles from a single primitive. It could then be used to amplify the mesh resolution on-the-fly. The coarse mesh is sent as input to the rendering pipeline. The vertex shader transforms the coarse geometry and then the geometry shader tessellates it as required and applies the displacement map. However, the

amount of data output by the geometry shader is limited and thus the number of vertices that can be generated by a single geometry invocation is also limited. This can be overcome if the data is fed back to geometry shader again to perform further subdivision, however it does not achieve reasonable performance for large subdivision in real-time.

Vlachos et al. [2001] suggested to add a stage into the graphics pipeline located just after the vertex shader. This stage replaces each triangle of the original mesh by a "curved PN-triangle" providing a smooth surface with smaller triangles. Such tessellation unit has been implemented in the ATI Radeon 8500 in 2001, but it did not allow to control the position of the resulting vertices, making it not compatible with techniques such as displacement mapping. Boubekeur and Schlick [2005] proposed to use instantiation of generic refinement patterns to create additional vertices. They use a single refinement pattern for each resolution level. Each vertices of the refinement pattern is described using barycentric coordinates. At render time, for each face of the coarse mesh, the refinement pattern at the desired level is instantiated. The positions of the new vertices are interpolated from their barycentric coordinates, the attributes of the coarse face and the displacement map. The refinement patterns are kept in GPU memory. The amount of data transmitted between the CPU and GPU thus becomes independent of the refinement level, as only the coarse mesh need to be transmitted at runtime. Boubekeur and Schlick [2008] extended this methods to support different refinement levels for adjacent faces by adding patterns to handle all configurations of adjacent refinement levels. Then at run-time, the appropriate patterns are chosen to obtain a spatially-continuous and adaptive subdivision. The hardware tessellation is based on a similar approach but dedicated steps have been added to the graphics pipeline.

1.4.3 Hardware Tessellation

Since DirectX 11 [MICROSOFT, 2009] and OpenGL 4.0 [Segal and Akeley, 2010], GPUs expose a dedicated unit to amplify the polygonal density of the mesh on the fly: the *hardware tessellation* [Moreton, 2001] (Figure 1.11). For a comprehensive survey on hardware tessellation see [Nießner et al., 2016]. In this context, a coarse representation defined as a set of patches is used. At runtime the resolution of the model is controlled on a per patch basis, enabling patch-grain LOD control with smooth spatial and temporal transitions. This makes hardware tessellation the ideal support to implement for displacement mapping as well as to render subdivision surfaces.

To use hardware tessellation, the pipeline takes as input a control mesh composed of patches. Its vertices are the control points of the patch, and will

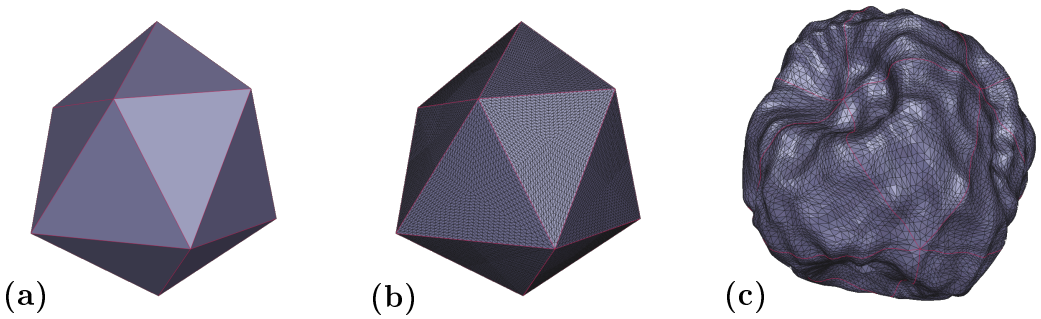


Figure 1.11 – Example of hardware tessellation — **(a)** A Coarse mesh for asteroid. **(b)** Tessellated mesh at factor 32 for every patch with no displacement. **(c)** Tessellated mesh with displacement.

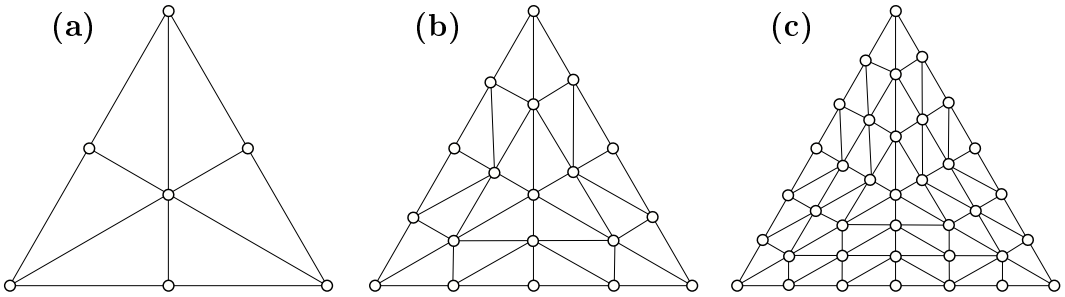


Figure 1.12 – Triangle Hardware Tessellation Pattern — Example of the predefined tessellation pattern output by the Tessellator for different uniform factors. From left to right: 2 ,4 ,6.

later be used to compute the final position of the generated vertices. The patch control points are processed normally by the Vertex Shader. Since the control mesh has fewer vertices than the original mesh, it enables low-cost animations in the vertex shader. The tessellation process is divided into two programmable stages: the Tessellation Control Shader (TCS) and the Tessellation Evaluation Shader (TES) and a fixed function stage: the Tessellator. These new steps are located between the vertex shader and the geometry shader (Figure 1.1).

The Tessellation Control Shader (TCS) controls how each patch gets tessellated by computing per-path tessellation factors. Using two mechanisms which are discussed in more detail later, it enables smooth spacial and temporal transitions.

The following stage is the Tessellator. It is a fixed-function stage which subdivides the patches according to a predefined and uniform subdivision pattern. It generates the topology and the new vertices from the factors set in the TCS. Examples of tessellation patterns are shown in Figure 1.12 for triangle patches.

The Tessellation Evaluation Shader is then invoked for each generated vertex. It outputs their final position and other per-vertex attributes. New

vertices are described in terms of tessellation coordinates which are the location of the new vertex within the patch. It corresponds to the barycentric coordinates for triangles and standard bilinear coordinates for quads. From these coordinates and the control points, the attributes of each vertex have to be interpolated. It is at this stage that offsets are applied for displacement mapping, or limit positions are computed for subdivision surfaces. Then the pipeline continues normally with, optionally the Geometry Shader, and the rasterization of the generated primitives.

Although the hardware tessellation is the ideal mechanism to render displacement maps, it has several shortcomings. After presenting the different challenges when using hardware tessellation with displacement maps, we discuss alternative representations trying to solve these issues.

Generating the control mesh

The control mesh and the displacement map can be directly generated using multi-resolution authoring tools such as Mudbox or ZBrush. The control mesh can also be obtained as the result of a simplification algorithm. In the case of hardware tessellation, the goal is slightly different than usual mesh simplification. In addition to approximate well the original surface, the control mesh need to have a uniform distribution of feature across its patches to achieve maximum performance. Indeed, Yuan et al. [2016] have shown that the variation of the tessellation level on different patches has an impact on the rendering time. They thus propose to count the number of features per triangles throughout the simplification. They define a feature vertex as a vertex whose Gaussian curvature is larger than a predefined threshold. Then, when collapsing an edge, if the number of feature vertices for a triangle exceeds a pre-defined threshold, they cancel the operation. The simplification process continues until all triangles have reached the maximum number of features or the desired number of triangles is achieved.

Tessellation level selection

Selecting the appropriate factors to reach maximum efficiency without introducing visual artifacts remains also an open question. Previous approaches only consider either the projected patch size [Cantlay, 2001] – fully neglecting its actual content after tessellation – or isotropic metrics, such as the Hausdorff distance between each LOD and the detailed mesh [Schafer et al., 2013], which overestimates the error along some view directions and does not take into account the surface attributes.

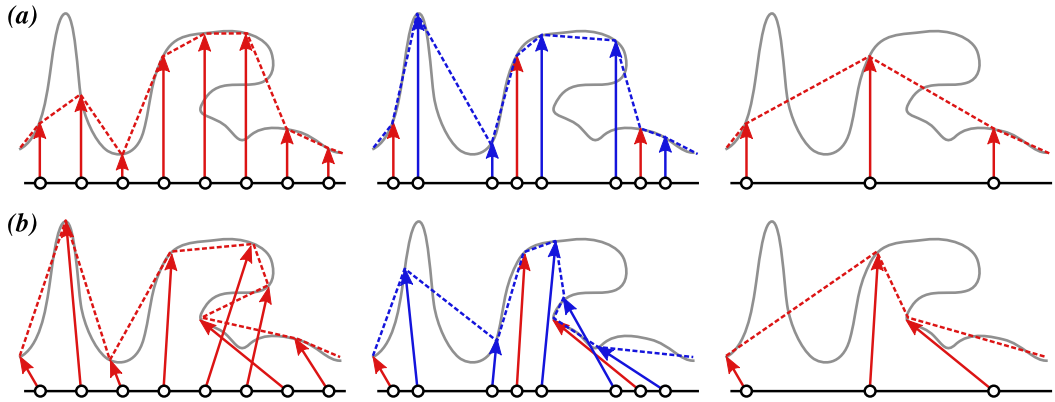


Figure 1.13 – Schematic comparison of scalar **(a)** and vectorial **(b)** displacement mapping for the same geometry (gray curve) at two successive integer levels (in red) and a fractional level (in blue). Vectorial displacement better represents the input signal, and enables folds. Yet both approaches suffer from swimming artifacts when the pattern continuously changes with fractional tessellation (middle column); the surface appears to fluctuate due to the undersampling of the displacement map.

Temporal continuity

The temporal continuity is achieved through fractional tessellation factor (Figure 1.14 **(a)**). To transition between two levels, new vertices appear from existing ones and move progressively to their final positions. The hardware tessellation exposes two modes of transition: between odd levels, or between even ones. Figure 1.14 **(a)** shows a transition between even levels 4 and 6. The tessellation pattern progressively changes to transition between two tessellation levels. However continuously changing the sampling pattern creates *swimming* artifacts when a displacement map is used (Figure 1.13). The surface appears to fluctuate due to the undersampling of the displacement map. As shown in Figure 1.13, both scalar and vectorial displacements suffer from this issue.

Spacial continuity

The spacial continuity is achieved because patch tessellation density is controlled by several tessellation factors. The outer tessellation factors control how the patch gets subdivided on its borders, and the inner tessellation factors control the interior of the patch. For a triangle patch, there are three outer tessellation factors, one for each edge, and one inner tessellation factor. Figure 1.14 **(b)** shows a patch with different outer tessellation factors. The Tessellator will automatically connect seamlessly the borders and the interior of the patch regardless of the selected factors. Thus to guarantee the spacial continuity, it is only necessary to ensure that edges shared by two patches are set to the same factor.

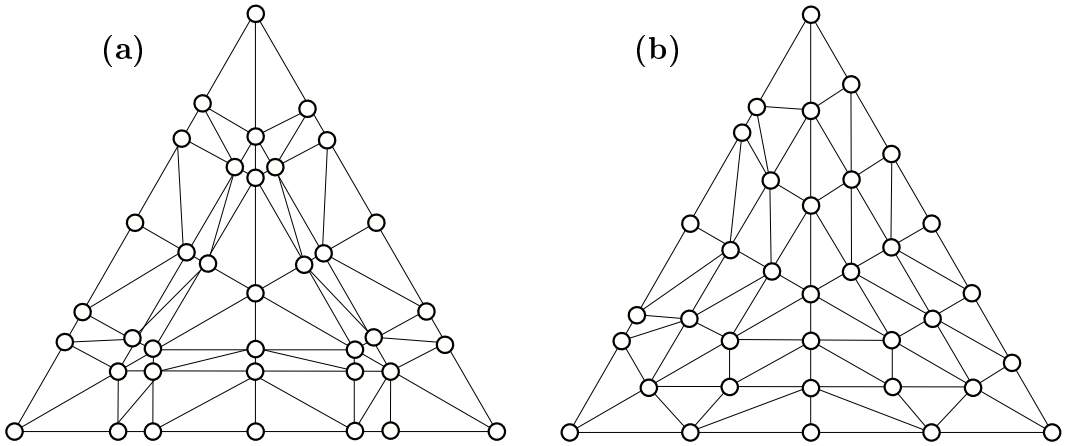


Figure 1.14 – Fractional Hardware Tessellation — **(a)** Transition between even levels with a fractional factor of 4.9. **(b)** tessellation pattern with different outer tessellation factors.

However, owing to texture coordinate discontinuities introduced when computing the 2D parametrization by unfolding the mesh, cracks can occur along seams when using displacement maps. 3D Points on seams corresponds to several points in texture space, the displacement values interpolated at these points can differ because of the different sampling rates on both side of the seam or texture misalignment. Bilinear texture interpolation or mipmapping amplify even more this problem. Such a problem is not specific to displacement maps, color and normal maps suffer also from these discontinuities. In the case of displacement mapping, however, a tiny difference leads to highly visible cracks in the rendered surface.

Feature adaptive tessellation

A more fundamental issue of hardware tessellation is that patches are uniformly subdivided according to a predefined pattern. This is especially problematic with standard scalar displacement along vertex normals (Figure 1.13(a)). For a single level, vectorial displacement can mitigate this issue because vertices can be moved to (almost) arbitrary locations [Jang and Han, 2012], which is essential to faithfully reproduce features and folds (Figure 1.13(b)). However, once the vectorial displacements have been computed for the finest level, vertices cannot be spatially redistributed to construct the coarser ones since the subdivision mechanism is fixed and uniform. Conversely, if starting from the coarser level, it is not possible to control topologically where new vertices will be added. This severely limits the ability to construct feature-aware LOD.

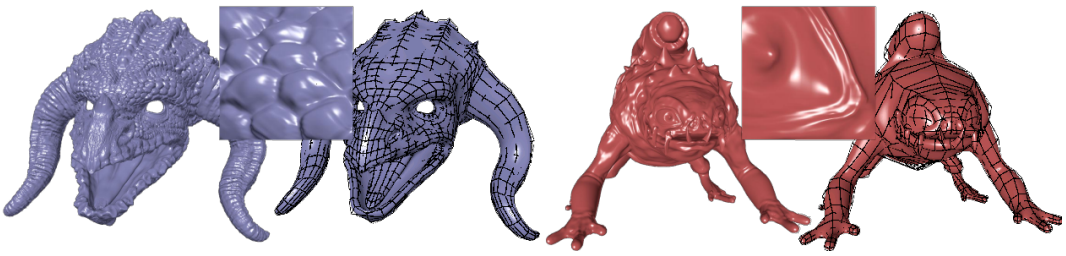


Figure 1.15 – Analytic displacement applied on top of Catmull-Clark subdivision surfaces using hardware tessellation Nießner and Loop [2013]. The normals are directly derived from the displacement function allowing a fast and accurate shading. The models are courtesy of the Blender Foundation and Bay Raitt, respectively.

1.4.3.1 Analytic displacement

To obtain a faithful visual aspect of the model, it is crucial to retrieve the normals of the detailed surface at a fragment scale to achieve accurate shading. In the case of scalar displacement, these normals can be computed from the displacement map using bump mapping [Blinn, 1978]. However, this method requires several look-up in the displacement map to compute the normals. The most common method is rather to use normal mapping, i.e., to use an additional map to retrieve the normals of the detailed surface. In addition to be more efficient that bump mapping, it allows to add micro scale details independently of the displacement map. It is however complicated to modify in real time the displacement when using normal mapping, as it will be necessary to re-compute on-the-fly the normal maps. Furthermore, to allow deformation of the base surface, the normals need to be stored in tangent space. The computation of tangent frames globally consistent across the mesh edges is costly and difficult.

Nießner and Loop [2013] propose to store the displacements as a smooth analytic function. The normals can thus be directly derived from this representation. Therefore, it is no longer necessary to use a normals map, which avoids all the normal mapping problems. More precisely, their displacement surface $\mathcal{S}_a$ is defined as:

$$\mathcal{S}_a(u,v) = \mathcal{C}(u,v) + \mathcal{N}_s(u,v)\mathcal{D}(u,v) \tag{1.27}$$

where $\mathcal{C}(u,v)$ is a base Catmull-Clark limit surface defined by the coarse base mesh, $\mathcal{N}_s(u,v)$ is its corresponding smooth normal field, and $\mathcal{D}(u,v)$ is a scalar-valued displacement function. Their method is not specific to Catmull-Clark subdivision surface and could be extended to other subdivision schemes as long as the limit surface is C^2 continuous, except at a limited number of extraordinary vertices where it is still C^1. The displacement function $\mathcal{D}(u,v)$

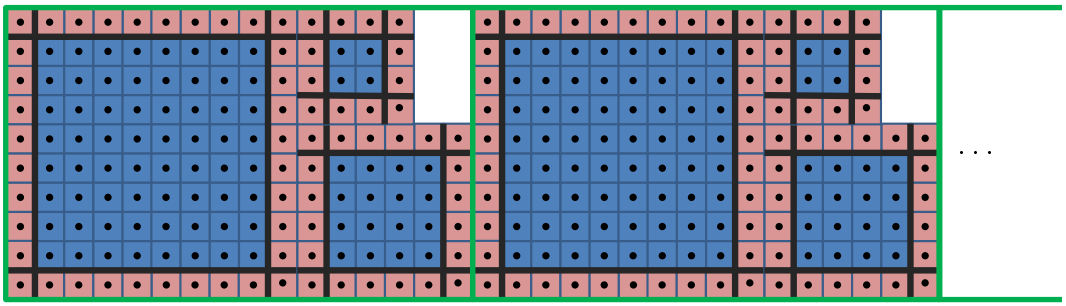


Figure 1.16 – Example of the tile-based texture format used to stored displacement data Nießner and Loop [2013]. The displacement data (blue) and overlap (red) of two tiles (green outline) are stored in tightly packed representation. Each tile is composed of three mip levels.

is then defined as a scalar-valued biquadratic B-spline with a Doo-Sabin subdivision surface structure [Doo and Sabin, 1998], which is C^1 continuous with vanishing first derivative at extraordinary vertices. Therefore, the displaced surface $\mathcal{S}_a(u, v)$ is also C^1 continuous, allowing direct evaluation of the surface normals everywhere.

The coefficients of the biquadratic B-spline are stored in the displacement maps in a tile-based format similar to Ptex Burley and Lacewell [2008]. It stores a separate texture per quad patch of the coarse mesh. These small textures are tightly packed in a single map, leading to a compact representation adapted for the GPU.

To avoid introducing cracks along the tile boundaries, each tile is stored in an axis-aligned fashion with a one-texel overlap corresponding to adjacent tiles. This overlap enables coherent filtering on both sides of the seams. To avoid problems at extraordinary vertices, where not exactly four tiles meet, they force the tile corners to have the same values, avoiding any filtering problem at the cost of limiting the modeling flexibility.

Nießner and Loop [2013] tackle the *swimming* artifacts problem as a signal sampling problem. The displacement map can be seen as a high frequency signal, then *swimming* artifacts correspond to aliasing artifacts, i.e., the sampling rate is too small compared to the signal frequency. Aliasing is a well-known problem in computer graphics that can be solved using pre-filtering [Williams, 1983]. Mipmapping consists in a sequence of images representing the same image but the dimensions of successive images are each time divided by two. Each image of the sequence is thus a prefiltered version of the original image at lower resolutions.

Nießner et al. thus precompute a mipmap for each patch. An overview of their storage scheme is illustrated in Figure 1.16. At runtime, they use the

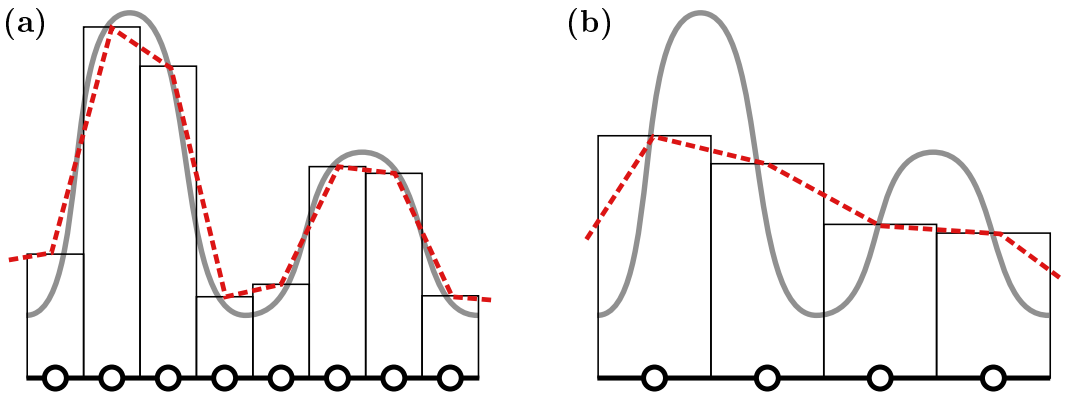


Figure 1.17 – A continuous signal (gray line) is stored in a texture (black rectangle). The displaced surface (red line) is reconstructed for two different mip levels. The displaced surface **(a)** represents well the input signal, but at lower resolution **(b)** the displaced surface is completely smooth and thus fail to represent the signal.

appropriate mipmap level based on the tessellation density to avoid undersampling artifacts. In addition to avoid *popping* artifacts when changing the mipmap level, they linearly interpolate the resulting displacements between the two mipmap levels.

Although analytic displacement enables an accurate shading of the displaced surface, it is restricted to subdivision surfaces with smooth scalar displacements and therefore it can not represent sharp edges. This severely limits the range of models that can be represented. In addition, the manipulation of biquadratic B-spline is more costly than regular scalar or vector displacements.

Whereas the use of mip-mapping certainly avoids typical artifacts that appears when using a displacement map, this severely deteriorate the LOD quality. Mip-mapping tends to over smooth the displacement, increasing the distance between the reconstructed surface and the original one (Figure 1.17). The original surface could most often be better represented if the samples were spatially redistributed to construct feature-aware LOD.

1.4.3.2 Indirect scalar mapping

Vectorial displacements allow to reconstruct more accurately the original surface by sampling the features of the mesh as shown in Figure 1.18**(b)**. However when deforming the base surface, the displacement vectors may intersect each other, leading to visible unbearable artifacts (Figure 1.18**(e)**). The greater are the vector displacement, the higher are the chances that this problem happens. This makes it difficult to use for anything other than static meshes or rigidly transformed surfaces.

Conversely, scalar displacements mitigate this problem because the vertices

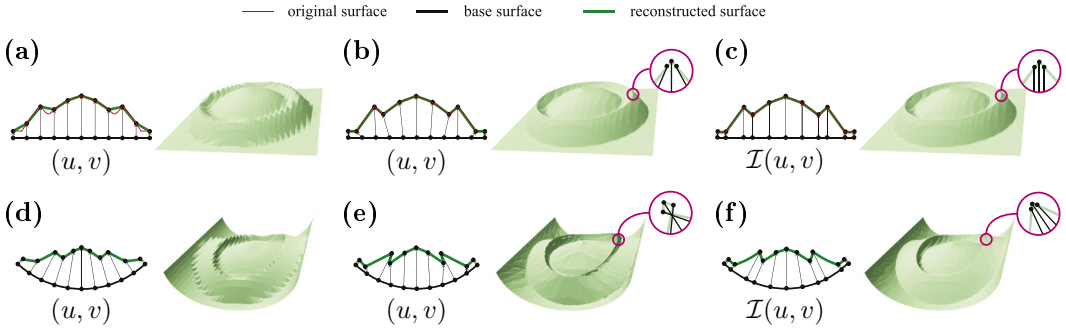


Figure 1.18 – Comparison of scalar (left), vector (middle) and indirect scalar (right) displacements Jang and Han [2013]. Scalar displacements **(a)** fail to represent the feature of the mesh, on the contrary the vector **(b)** and indirect scalar **(c)** displacements reconstruct accurately the original surface by sampling the features of the mesh. However, when the base surface is deformed, the vector displacements **(e)** introduce self-intersections, while scalar **(d)** and indirect scalar **(f)** displacements remain artifacts free.

are always displaced along the normals (Figure 1.18**(d)**). However, as the sampling is uniform and the vertices can not be redistributed, it often fails to reconstruct the features of the model (Figure 1.18**(a)**).

To reduce the apparition of self-intersections while accurately representing the features of the model, Jang and Han [2013] decomposed the vectorial displacement map into a height map and an indirect scalar map. The indirect scalar map redistributes the vertices in parametric space, which are then displaced along the surface normals using the height map and their new parametric coordinates. The surface reconstructed using indirect scalar displacement mapping is defined as:

$$\mathcal{S}_i(u,v) = \mathcal{P}(\mathcal{I}(u,v)) + \mathcal{H}(u,v)\mathcal{N}(\mathcal{I}(u,v)) \tag{1.28}$$

$\mathcal{S}_i$ is reconstructed by displacing the point of the base surface $\mathcal{P}$ along the surface normal $\mathcal{N}$ by the offset $\mathcal{H}$ fetched from the height map. The indirect scalar map $\mathcal{I}$ is used to redistribute non-uniformly the samples on the base surface. The surface features are accurately reconstructed by optimizing the indirect scalar map such that the surface features are well sampled (Figure 1.18**(c)**). In addition, the displacement occurs along the normal, making this method more suitable for surface deformations (Figure 1.18**(f)**). However, it still does not allow a spatial redistribution of all vertices for the coarser levels once the indirection map has been computed for the finest one.

1.4.3.3 Multi-resolution attributes

Schafer et al. [2013] avoid texture-related artifacts by using a data structure

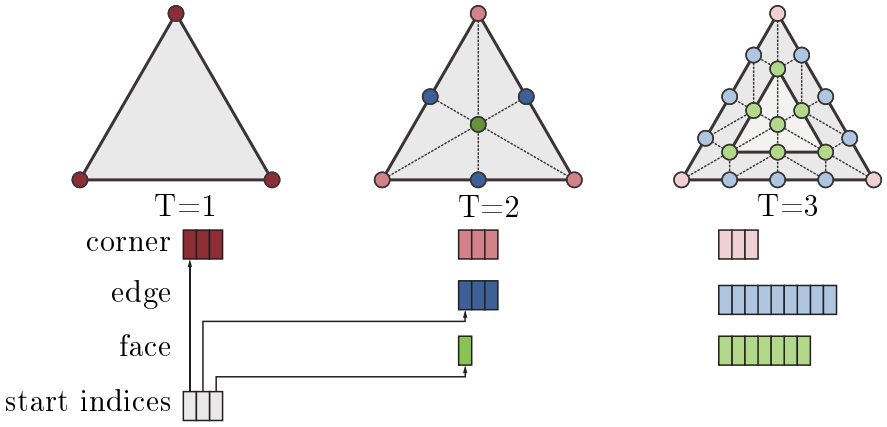


Figure 1.19 – Attributes vertices located on the corners, the edges or the face of a patch are stored separately in a compact 1D array, providing a seamless representation [Schafer et al., 2013].

that follows the hardware tessellation pattern. Instead of storing the spatially varying attributes (e.g., displacements) into texture, they proposed to store the attributes as vertex attributes in a compact 1D array. This provides a bijective mapping between the vertices generated by the Tessellator and their attributes. This one-to-one mapping guarantees coherent evaluations along patch boundaries, thus preventing cracks and discontinuities. Moreover, it overcomes under-sampling artifacts appearing when sampling attributes from textures. It also provide a more compact storage than texture, thanks to the bijective mapping.

Taking advantage of the fixed tessellation pattern, they store attributes for each generated vertices for different tessellation factors. Vertex attributes are stored separately for vertices located on the corners, the edges or the interior of the patch, as illustrated on Figure 1.19 for a single triangle. For multi-resolution attributes, they store the data corresponding to the different levels of tessellation one after another. They choose to store attributes only for power-of-two tessellation factor similarly to texture mipmapping.

For edges and corners, they store attributes only once for all adjacent patches, and pack all attribute data in a single buffer. Vertices located on patch boundaries thus share the same attributes, resulting in consistent evaluation on both sides of the boundaries. To recover the right patch data at run-time, they store an additional lookup table containing for each patch the indices to the first attributes of the face, edges and corners. To handle different face orientations, they use signed indices to indicate the read direction for edges.

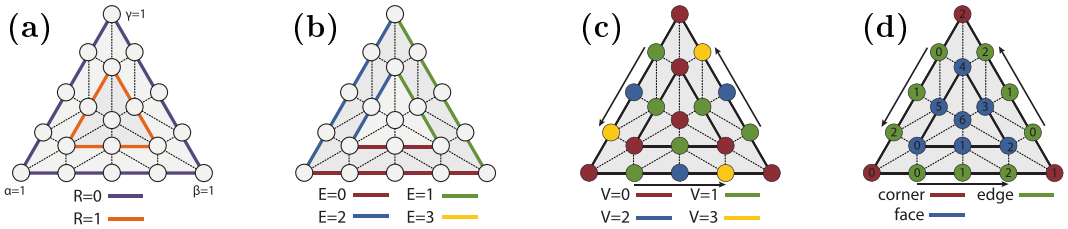


Figure 1.20 – Indexing scheme of Schafer et al. [2013]. A unique index **(d)** is computed from the barycentric coordinates by determining the ring **(a)**, the edge **(b)**, and the vertex index **(c)**.

To fetch the attributes during the rendering step, in addition to the lookup indices, another index is required to identify each vertex inside the patch. Current GPUs do not expose such an index, but Schäfer et al. deduce it from the fractional barycentric coordinates of the generated vertices.

In the following, we describe their index computation for triangle patterns, they also derive a similar index computation for quad patterns, but it will not be detailed here. For a triangle patch at tessellation factor t, the tessellation pattern consists of $\lfloor t/2 \rfloor + 1$ concentric topological rings. Edges on each ring are composed of $t - 2R$ segments. Thus each vertex can be uniquely identified by the triplet (R, E, V) (Figure 1.20 **(a,b,c)**), where $R \in [0, \lfloor t/2 \rfloor]$ is the ring index, $E \in [0, 2]$ the edge index, $V \in [0, t - 2R]$ the position of the vertex on that edge. These three indices (R, E, V) are computed from the barycentric coordinates. The ring, edge, vertex (R, E, V) indices are then used to compute a linear index as illustrate on Figure 1.20 **(d)**.

They show that nearest neighbor, bilinear and trilinear filtering can be easily applied using the (R, E, V) indices. Those interpolations allow to support arbitrary tessellation factors while storing attributes for power-of-two tessellation factors only. However, in the case of displacement attributes, using arbitrary levels still leads to *swimming* artifacts. To avoid them, they limite the tessellation level to power-of-two factors. Note that such a constraint guarantees that vertices of a given level will also be present with the exact same barycentric coordinates at the next level. Indeed, when using only power-of-two factors, increasing the level adds new vertices in the middle of each edges of the previous level. Therefore, unlike when using fractional factors, sampling positions of existing vertices do not vary. It thus bypass completely the undersampling problem at the cost of discarding the fractional tessellation transition mechanism and the benefits of finer-grain patterns (odd or even). The transition between levels is achieved by a simple linear blend of the two nearest power-of-two levels.

This representation proposed by Schafer et al. [2013] partially solves the feature preserving problem. Their representation provides a one-to-one mapping between the generated vertices and their attributes, it is thus possible to spatially redistribute all the vertices at every levels. To preserve artifacts-free temporal transition, existing vertices should not move too far from their previous positions between levels or it will produce severe *swimming* artifacts. As new vertices cannot be added at arbitrary topological positions, this severely limits the possibilities of redistribution.

1.4.4 Summary

Hardware tessellation allows fast and efficient rendering of detailed meshes. To this purpose, the detailed mesh is decomposed into a control mesh and a displacement map. Several solutions have been proposed to solve the visual artifacts that may appear due to the undersampling of the displacement map. Nießner and Loop [2013] combine a tile-based format with mip-mapping, guaranteeing coherent access along patch boundaries and solving the problem of undersampling. Schafer et al. [2013] store the displacements in a per vertex basis avoiding all the texture-related problems.

The storage proposed by Schafer et al. allows in addition to control precisely the position of all generated vertices for all levels. However, optimizing independently each level, without taking into account how the transitions between levels are done, will reintroduce swimming artifacts. Indeed, their approach is still limited by the possibility to control topologically where new vertices will be added. Thus the ability to construct feature-aware LOD remains an open challenge.

Chapter 2

View-Dependent LOD selection

In the previous chapter, we have seen that the hardware tessellation allows to subdivide each patch at different tessellation factors. Selecting the appropriate factors to reach the maximum efficiency without introducing visual artifacts is a challenging task. Indeed, the visual appearance of a model depends on numerous parameters: its geometry, its material, the illumination environment and the view characteristics. These parameters have highly complex interactions with each other and it is too complicated to take into consideration all of them at the same time. Our metric for LOD selection focus on measuring the geometrical error and the attribute deviation depending on the view distance and direction.

In this chapter, we describe how we compute (Section 2.1) and summarize (Section 2.2), for each patch and at each level of the LOD, the error relative to the reference mesh in a view-dependent fashion so that adequate tessellation factors can be efficiently determined at render time (Section 2.4).

Our metric takes as input a detailed reference mesh and a hierarchy of LOD. We require that these meshes are consistently parametrized. As in most modern rendering pipelines, we further assume that all spatially varying attributes, such as normals and colors, are stored within texture maps. This way, attribute distortions are conservatively measured by the parametric distortions even though they are not strictly equivalent. For example, important parametric distortions may not be visible for models with low variations of attributes.

2.1 Mapping and Difference Vectors

We first focus on establishing a view dependent error metric for a single patch at a given level. For now, we only consider the impact of the view direction; the scaling factors due to the perspective projection and pixel resolution are taken into account at render time (Section 2.4). Our error metric is based on the bijection offered by the parametrization shared between the LOD and the

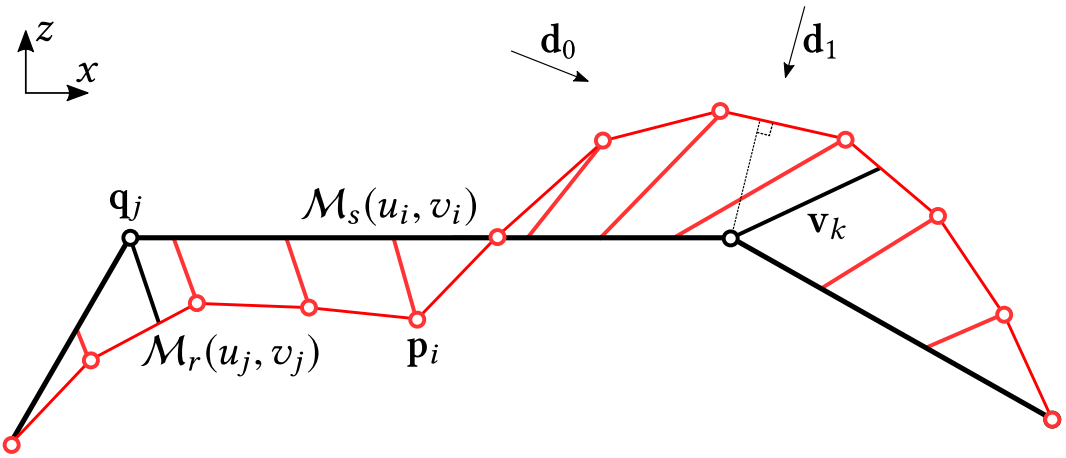


Figure 2.1 – $\mathcal{M}_s$ and $\mathcal{M}_r$ allow to define the difference function $\mathcal{V}$ between the reference mesh (in red) and the simplified mesh (in black) using their shared parametrization. As illustrated with the difference vector $\mathbf{v}_k = \mathcal{V}(u_k, v_k)$, this mapping generally differs significantly from the orthogonal projection of one mesh on the other. Also note how the error varies according to the view direction; for instance, it is much larger when seen from $\mathbf{d}_0$ than from $\mathbf{d}_1$.

reference mesh. In the following, $(\mathbf{p}_i, u_i, v_i)$ (resp. $(\mathbf{q}_j, u_j, v_j)$) refers to the 3D position and texture coordinates of the i^{th} (resp. j^{th}) vertex of the reference (resp. simplified) mesh. Let $\mathcal{M}_s : (u, v) \to (x, y, z)$ for the simplified patch and $\mathcal{M}_r$ for the reference mesh be the functions which associate to every texture coordinates their corresponding 3D position. In particular, $\mathbf{p}_i = \mathcal{M}_r(u_i, v_i)$ and $\mathbf{q}_j = \mathcal{M}_s(u_j, v_j)$ (Figure 2.1). Taking the difference between the two surfaces in parametric space yields the vector-valued function

$$\mathcal{V}(u, v) = \mathcal{M}_s(u, v) - \mathcal{M}_r(u, v) \ , \tag{2.1}$$

defining for every point a so called *difference vector*.

For a given unit view direction $\mathbf{d}$, the error introduced by a single vector $\mathbf{v} = \mathcal{V}(u, v)$ is equal to the norm of its projection on screen, i.e., $\sin(\text{angle}(\mathbf{d}, \mathbf{v}))\, \|v\|$ which can be written as $\|\mathbf{v} \times \mathbf{d}\|$. As depicted in Figure 2.3(a), if the view direction is aligned with the difference vector, the error vanishes. In contrast, the error becomes maximal when the view direction is orthogonal to the difference vector. For a patch and a direction $\mathbf{d}$, it is thus possible to compute a conservative error $E(\mathbf{d})$ (Figure 2.3**(b)**), that is equal to the maximum error introduced by all vectors for this direction:

$$E(\mathbf{d}) = \sup_{(u,v)} \|\mathcal{V}(u, v) \times \mathbf{d}\| \ . \tag{2.2}$$

As $\mathcal{M}_s$ and $\mathcal{M}_r$ are continuous piecewise linear functions and as $\mathcal{V}$ is a linear combination of those, it is also continuous and piecewise linear. In

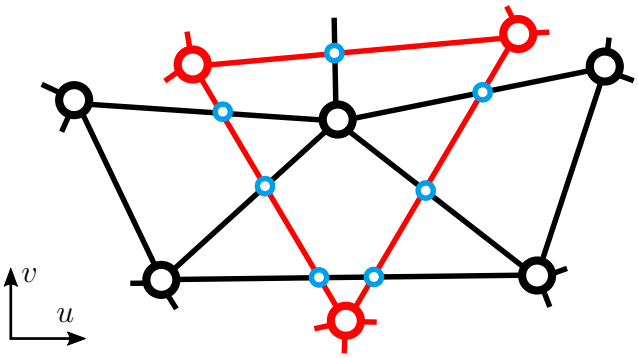


Figure 2.2 – To evaluate E, the vectors originating from the vertices (red and black circle) of the two meshes and from all edge intersections (blue circle) in parametric space have to be considered.

addition, in equation (2.2), the cross product performs a linear combination of the components of $\mathcal{V}(u, v)$, and the norm transforms this continuous piecewise linear function into a continuous piecewise monotone function. Therefore, as we are only interested in maximum error values, we can evaluate E only for a discrete set of difference vectors $\mathbf{v_k} \in V$ corresponding to the bounds of each linear piece:

$$E(\mathbf{d}) = \max_{\mathbf{v_k} \in V} \|\mathbf{v}_k \times \mathbf{d}\| \ . \tag{2.3}$$

In 2D, the linear pieces are segments bounded by the union of the vertices of the reference and simplified meshes. As illustrated in Figure 2.1, the difference vectors thus match the vertices $\mathbf{q}_j$ of the simplified mesh with their image on the reference mesh $\mathcal{M}_r(u_j, v_j)$, and reciprocally the vertices $\mathbf{p}_i$ of the reference mesh with their image on the simplified mesh $\mathcal{M}_s(u_i, v_i)$.

In 3D, the functions $\mathcal{M}_s$ and $\mathcal{M}_r$ are also linear on each triangular face of their respective domain. Thus the function $\mathcal{V}$ is linear on each polygonal face formed by the intersection of the two meshes in parametric space, as depicted on the Figure 2.2. Consequently, to evaluate E, we need to consider the vectors originating from the vertices of the two meshes and from all edge intersections in parametric space. For a given point $\mathbf{p}_i$ on the reference mesh (resp. $\mathbf{q}_j$ on the simplified mesh), we thus need to efficiently find the point $\mathcal{M}_s(u_i, v_i)$ on the simplified mesh with the same texture coordinates (resp. $\mathcal{M}_r(u_j, v_j)$ on the reference mesh). To accelerate this search, we use two 2D AABB-trees built in parametric space over the triangles of the reference and simplified meshes. These 2D AABB-trees are used to find the triangle containing the desired texture coordinates. From the triangle and the point texture coordinates, we compute its barycentric coordinates, and then deduce its 3D position. We also need to consider the difference vectors occurring at edge-edge intersections in the 2D parametric space. To this end, we implemented a edge-to-edge intersection technique, based on a 2D AABB-tree in parametric space on the

edges.

Owing to the large number of difference vectors (in the order of 10^4 for a single patch), the definition of $E(\mathbf{d})$ as in equation (2.3) is not suitable for fast evaluation. One can notice however that only a fraction of them are actually contributing to the *maximum* error, those belonging to the convex hull of the vector set (centered at the same origin); the other ones can be discarded. Although the number of vectors is much smaller after this pruning (from dozens to a few hundreds) it is still impractical for real-time GPU evaluation: a more compact representation is required.

2.2 Approximate Error Metric

To evaluate E efficiently, we need to find an approximation $\tilde{E}(\mathbf{d}) \approx E(\mathbf{d})$ which is fast to evaluate and can be stored with a low memory footprint. The approximation $\tilde{E}$ has to minimize the mean squared distance to E over all directions: $\min \int (\tilde{E}(\mathbf{d}) - E(\mathbf{d}))^2 d\mathbf{d}$. Following the previous remark on the convex-hull reduction, a natural idea would be to further summarize the set of difference vectors by a simpler enclosing primitive such as, for instance, an ellipsoid as detailed in Section 2.2.1. In Section 2.2.2 we will present a more pragmatic approach that consists in directly approximating the error function E instead of approximating the difference vectors.

2.2.1 Bounding Ellipsoid

To summarize our set of difference vectors by a bounding ellipsoid, we have to solve two different issues: how to evaluate the error represented by an ellipsoid in a given direction and how to find the enclosing ellipsoid that best approximates E for all view directions.

Ellipsoid Evaluation

First, let us focus on how to evaluate the error represented by an enclosing ellipsoid for a given direction $\mathbf{d}$. An arbitrarily oriented ellipsoid centered at $\mathbf{c}$, is defined by the following implicit equation:

$$(\mathbf{x} - \mathbf{c})^T \mathbf{A} (\mathbf{x} - \mathbf{c}) = 1 \ , \tag{2.4}$$

where $\mathbf{A}$ is a positive definite matrix. The eigenvectors of $\mathbf{A}$ define the principal axes of the ellipsoid. The eigenvalues of $\mathbf{A}$ are the inverse of the square of the semi-axes length.

To estimate our error, we need to project it along the view direction. This can be done by computing a basis $\mathrm{K}_{\mathbf{d}}$ orthogonal to the view direction $\mathbf{d}$ and then projecting the ellipsoid on this basis using the inverse matrix $\mathbf{A}^{-1}$. Please

note that the orthogonal projection of an ellipsoid on a 2D plane is always an ellipse. The new ellipse is defined by the 2×2 matrix $\mathbf{A_d}$ obtained as follows: $\mathbf{A_d^{-1}} = \mathrm{K}_\mathbf{d}^T \mathbf{A}^{-1} \mathrm{K}_\mathbf{d}$. To obtain our conservative error for the current view direction, we need to find the length of the longest semi-axis of the projected ellipse. As for ellipsoids, the length of the semi-axis of an ellipse is proportional to the corresponding matrix eigenvalues. Putting all together, we arrive at the following formula:

$$\tilde{E}_{ell}(\mathbf{d}) = \sqrt{\lambda_{max}(\mathbf{A_d^{-1}})} \tag{2.5}$$

where λ_{max} extracts the longest eigen value. Although the control tessellation shader is not the bottleneck of the rendering pipeline, equation (2.5) is not cheap to evaluate. To speedup a little the evaluation, the matrix $\mathbf{A}^{-1}$ can be stored rather than $\mathbf{A}$.

Minimum Volume Enclosing Ellipsoid

Due to eigenvalue extraction and maximum operation, finding the best ellipsoid that enclose our vector set is a difficult problem. To find the enclosing ellipsoid that best approximates E for all view directions, a simple heuristic is to choose the bounding ellipsoid of minimum volume. The volume of an ellipsoid can be expressed using the determinant of $\mathbf{A}$:

$$Vol = \frac{4}{3}\pi \left|\mathbf{A}^{-1}\right|^{\frac{1}{2}} \tag{2.6}$$

Thus to minimize the volume of the ellipsoid, we must minimize the determinant of $\mathbf{A}^{-1}$. This give us the following formulation for the problem of the minimum volume ellipsoid problem enclosing all the difference vectors $\mathbf{v_k}$:

$$\begin{aligned} \text{minimize} \quad & \left|\mathbf{A}^{-1}\right| \\ \text{subject to} \quad & \forall \mathbf{v_k},\ (\mathbf{v_k} - \mathbf{c})^T \mathbf{A} (\mathbf{v_k} - \mathbf{c}) \leq 1 \\ & \forall \mathbf{z} \neq \mathbf{0}, \mathbf{z}^T \mathbf{A} \mathbf{z} > 0\ . \end{aligned} \tag{2.7}$$

The second constraint guarantee that $\mathbf{A}$ is a positive semi-definite matrix. This is not a convex optimization problem. However it is possible to obtain one by a change of variable. Then by solving the dual problem, a solution up to a relative accuracy of ϵ can be obtained in polynomial time [Khachiyan, 1996]. Our implementation of computing the minimum volume enclosing ellipsoid is based on [MOSHTAGH, 2005].

A plot of the minimum volume enclosing ellipsoid approximation $\tilde{E}_{mve}$ is depicted on Figure 2.3(**d**). The minimum volume bounding ellipsoid is not necessarily the one that best approximates our set of vectors. In some cases, the optimization leads to ellipsoids with semi-axes of very different lengths in

order to minimize the volume. This lead to huge overestimation of the error along directions orthogonal to the longest semi-axis.

We have tested other heuristics based on principal component analysis, or the distance to the error vectors convex hull, but none of them were satisfactory.

Enclosing Ellipsoid using a SQP solver

A more sophisticated approach to find the ellipsoid, strive to directly fit the error function E. For this purpose, we first discretize the problem by taking a very large set of directions $\mathbf{d}_i$ uniformly distributed on a sphere. In our experiments, we used 10^4 directions generated with a Spherical Fibonacci mapping [Keinert et al., 2015], which yields a nearly uniform distribution on the unit hemisphere.

Then to find the six coefficients of $\mathbf{A}$, we minimize in a least-square sense the difference between $\tilde{E}_{sqp}(\mathbf{d}_i)$ and $E(\mathbf{d}_i)$ for all directions. To guarantee a conservative approximation, we add inequality constraints ensuring that for any direction, $\tilde{E}_{sqp}$ is greater or equal to E. This lead to the following nonlinear problem:

$$\begin{aligned} \text{minimize} \quad & \sum_i \left(\tilde{E}_{sqp}(\mathbf{d}_i) - E(\mathbf{d}_i) \right)^2 \\ \text{subject to} \quad & \tilde{E}_{sqp}(\mathbf{d}_i) \geq E(\mathbf{d}_i) \ . \end{aligned} \tag{2.8}$$

Starting from the minimum volume solution, we perform a nonlinear minimization of equation (2.8) using a Successive Quadratic Programming solver (SQP) (Figure 2.3**(e)**). Although it gives good result, the minimization is computationally expensive due to the nonlinearity of the problem, which makes the solver slow to converge in some cases. The nonlinearity of the problem is directly linked to the maximum eigenvalue extraction of the ellipsoid evaluation of equation (2.5).

2.2.2 Spherical Harmonic Approximation of E

Evaluating the error represented by an ellipsoid in a direction $\mathbf{d}$ involves a somewhat expensive eigenvalue extraction after a projection on a 2D plane. For the same reason, finding the enclosing ellipsoid that best approximate E for all view directions is a difficult task.

Keeping the same approach to directly approximating the error function E instead of approximating the difference vectors, we propose to use a different representation which allows at the same time a linearization of equation (2.8) and a faster evaluation at run-time.

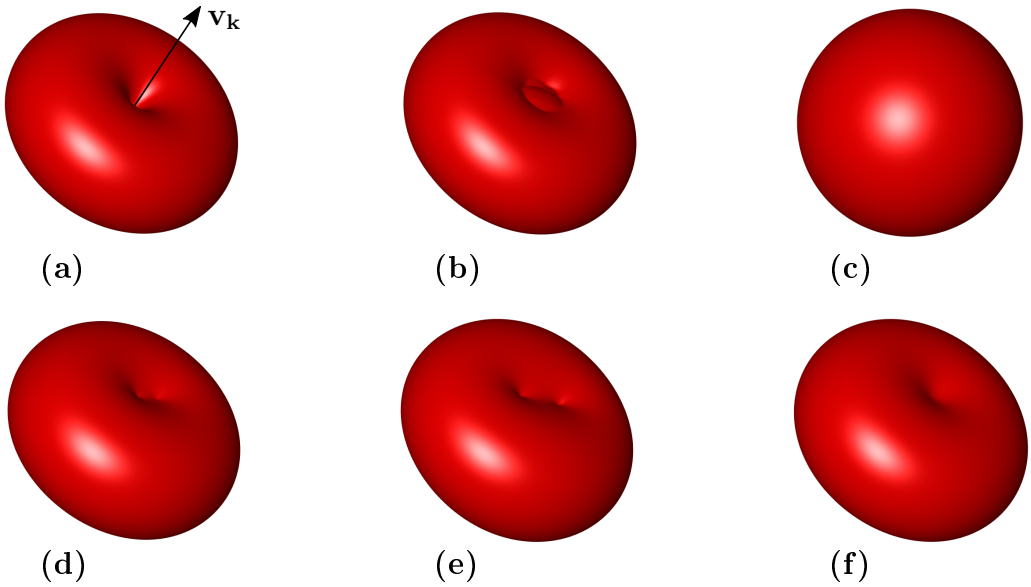


Figure 2.3 – Error metric visualization – we plot the error function $E(\mathbf{d})$ for a single **(a)** and a set **(b)** of difference vectors, as well as the isotropic approximation E_{iso} **(c)**, the minimum volume enclosing ellipsoid $\tilde{E}_{ell}(\mathbf{d})$ **(d)**, the enclosing ellipsoid with SQP optimization $\tilde{E}_{sqp}(\mathbf{d})$ **(e)** and our conservative error metric $\tilde{E}(\mathbf{d})$ **(f)**.

To find such a representation, let us make a few observations. First, we can observe in equation (2.3) that the error function $E(\mathbf{d})$ is centrally symmetric: $E(\mathbf{d}) = E(-\mathbf{d})$. Second, in the case of a single difference vector $\mathbf{v} = (v_x, v_y, v_z)$ and a direction $\mathbf{d} = (x, y, z)$, equation (2.2) becomes:

$$\begin{aligned} E(\mathbf{d}) &= \|\mathbf{v} \times \mathbf{d}\| \\ &= \sqrt{(v_y z - v_z y)^2 + (v_z x - v_x z)^2 + (v_x y - v_y x)^2} \ , \end{aligned}$$

which is the square-root of a quadratic trivariate polynomial limited to the second-order terms. A natural choice to approximate $E(\mathbf{d})$ thus consists in seeking for functions $\tilde{E}$ of the same form (see Figure 2.3 **(a)**), but with generalized coefficients:

$$\tilde{E}(\mathbf{d}) = \sqrt{a_1 x^2 + a_2 y^2 + a_3 z^2 + a_4 xy + a_5 xy + a_6 yz} \ . \tag{2.9}$$

We observe that since the vector $\mathbf{d} = (x, y, z)$ is unitary, the function space of the squared of $\tilde{E}$ is equivalent to the one defined by the constant and the five second-order basis functions of real spherical harmonics. Like $E(\mathbf{d})$, the function $\tilde{E}(\mathbf{d})$ is by construction centrally symmetric. It only requires the storage of six scalar coefficients, and involves very few arithmetic operations to be evaluated compared to the ellipsoid approximation.

To compute the best coefficients for $\tilde{E}(\mathbf{d})$, similarly to the SQP enclosing ellipsoid, we discretize the problem by taking a very large set of directions $\mathbf{d}_i$ uniformly distributed on a hemisphere. Then, in order to linearize the problem, we minimize in the least-square sense the distance between the square of our metric $\tilde{E}(\mathbf{d}_i)^2$ and the square error $E(\mathbf{d}_i)^2$ for all directions. Moreover, we guarantee a conservative approximation by adding inequality constraints ensuring that for any direction, $\tilde{E}$ is larger or equal to E. This boils down to a convex quadratic problem, which can be formally summarized as follows:

$$\begin{aligned} &\text{minimize} \quad \sum_i \left(\tilde{E}(\mathbf{d}_i)^2 - E(\mathbf{d}_i)^2\right)^2 \\ &\text{subject to} \quad \tilde{E}(\mathbf{d}_i)^2 \geq E(\mathbf{d}_i)^2 \quad \forall \mathbf{d}_i \ . \end{aligned} \tag{2.10}$$

This equation is efficiently solved using a QP solver. The approximation $\tilde{E}$ can represent a single minimum (Figure 2.3**(f)**), thus when the error is small for multiple view directions, it can suffer some overestimations. On the contrary, the ellipsoid representations $\tilde{E}_{sqp}$ and $\tilde{E}_{mve}$ (Figure 2.3**(d)** and **(e)**) can represent two minimum. We present a quantitative comparison in the following section.

2.3 Accuracy Evaluations

To evaluate the accuracy of these approximations, we consider both the average error A and maximum error M with respect to E over all directions on the unit sphere Ω, normalized by the maximal value of E:

$$\begin{aligned} M &= \sup_{\mathbf{d}}(\tilde{E}(\mathbf{d}) - E(\mathbf{d}))/\sup_{\mathbf{d}} E(\mathbf{d}) \\ A &= \frac{1}{4\pi}\int_{\Omega}(\tilde{E}(\mathbf{d}) - E(\mathbf{d}))/\sup_{\mathbf{d}} E(\mathbf{d}) \ . \end{aligned}$$

For comparison purpose, we also compute these indicators with an isotropic version of our error metric (Figure 2.3(c)), defined as:

$$E_{iso} = \sup_{\mathbf{d}} E(\mathbf{d}) = \max_k \|\mathbf{v}_k\| \ .$$

The normalization by the maximal value of E compensates for large variations of the magnitude of E across levels, patches and meshes. These measures are summarized in Figure 2.4 for four different meshes depicted in Figure 4.6. In these plots, for each level we take the maximum of M and average of A over all patches of the given mesh.

Our conservative metric $\tilde{E}$ clearly outperforms the isotropic approximation. In the worst case, it overestimates the actual LOD deviation by about 30%,

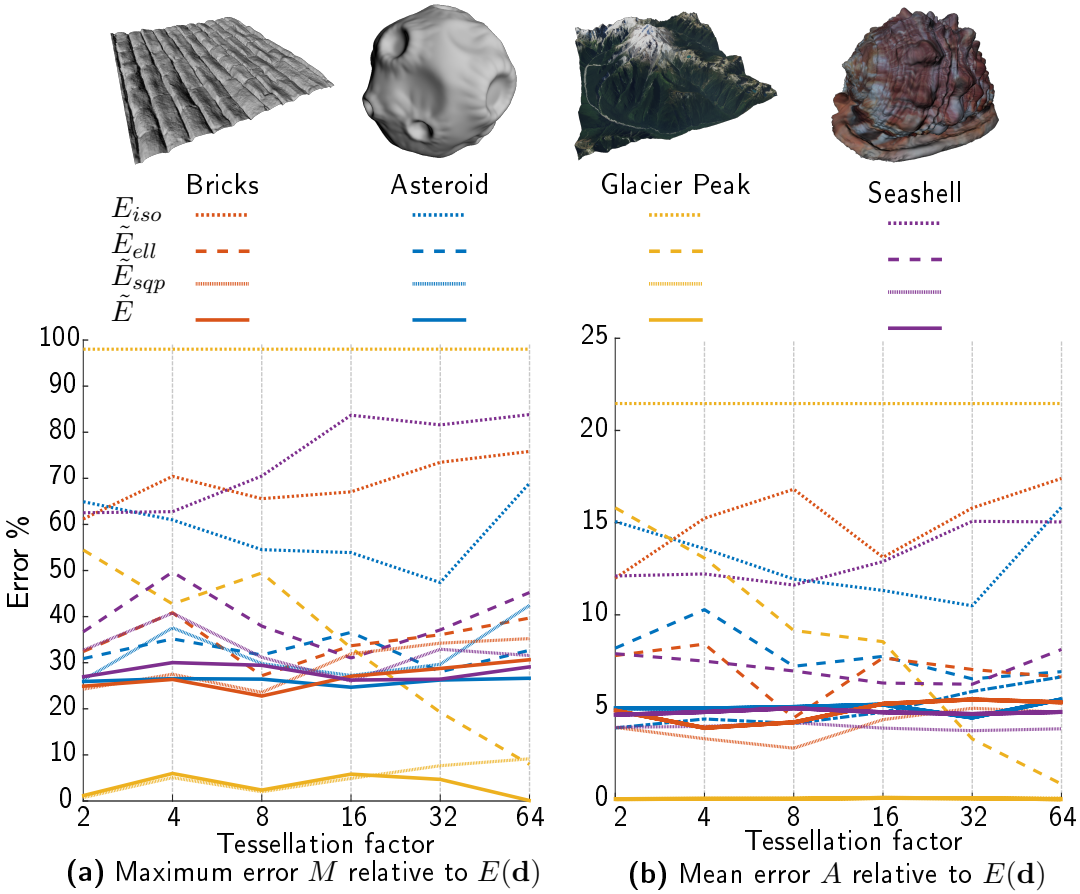


Figure 2.4 – Error metric comparison – for each 3D, at each tessellation factor, the maximum **(a)** and mean **(b)** error of three approximate error metrics have been computed and averaged over all patches generated using the simplification algorithm described in Chapter 4.

while the isotropic error overestimation ranges from 50 to 90% (Figure 2.4a). The approximation provided by the ellipsoids of minimal volume is somewhat in-between, but is less stable across meshes. Our metric $\tilde{E}$ and the sqp optimization of the ellipsoid $\tilde{E}_{sqp}$ give similar results. The latter performs slightly better with respect to the mean error, but our metric is slightly better regarding the maximum error. Our metric can suffer some overestimation because the quadratic polynomial fails to represent well multiple minima, which can happen when the error is small for multiple view directions. On the contrary, the ellipsoid representation can represent two minima, which explains why the enclosing ellipsoid with SQP optimization is better on average (Figure 2.5a and b). Yet the average error is very small, about 5% for both metrics, compared to 15-20% with the isotropic metric and 5-10% with the minimum volume ellipsoids (Figure 2.4b). This is especially noticeable for height-field models such as the "Glacier Peak" terrain, for which an almost unique direction is favored (the

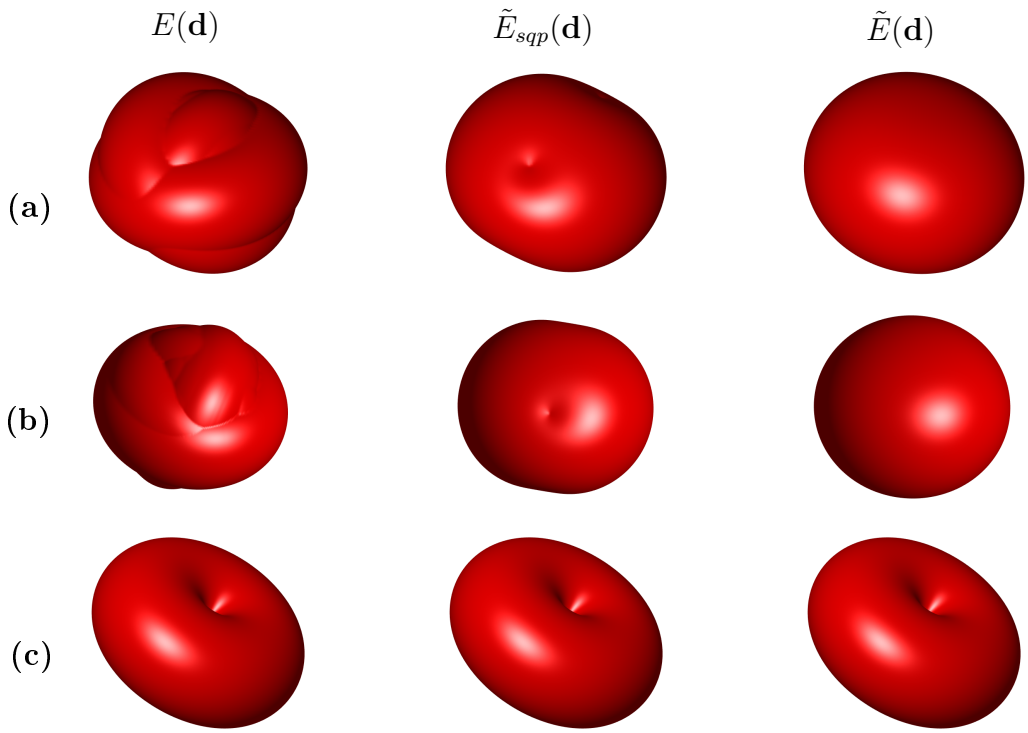


Figure 2.5 – Error metric visualization for different patches – we plot the error function $E(\mathbf{d})$, the enclosing ellipsoid with SQP optimization $\tilde{E}_{sqp}(\mathbf{d})$ and our conservative error metric $\tilde{E}(\mathbf{d})$ for one patch of the Bricks **(a)**, Seashell **(b)** and Glacier Peak **(c)** models.

direction of elevation) at every level. In such cases, both our approximation and the enclosing ellipsoid with SQP optimization represent perfectly the error function $E(\mathbf{d})$ (Figure 2.5c). This demonstrates that our approximation is very accurate in most cases, and that a more sophisticated approximation like the SQP optimization leads to very small gains and considering its computational overhead, using such an approach thus finds little justification.

2.4 LOD Selection

Once the coefficients of the error metric $\tilde{E}$ have been computed for all patches at all levels, they are stored in a 1D texture buffer indexed by the patch primitive ID. For power-of-two levels, this implies a negligible memory cost of 6 levels $\times$ 6 coefficients $\times$ 4 bytes $=$ 144 bytes per patch.

Our rendering pipeline then follows the standard hardware tessellation steps. For each patch, the tessellation level is computed in the tessellation control shader according to the current camera position and orientation. To guarantee that adjacent patches are seamlessly connected, we first compute a

tessellation level for each patch-border, and then assign the maximum level computed on the boundaries to the patch interior. For a patch-border, we evaluate our metric for the two adjacent patches. More precisely, the direction of evaluation $\mathbf{d}$ is computed as the normalized difference between the camera and edge center position. Our metric is thus evaluated six times for a patch: one time for each adjacent patch and three times for the patch at each patch-border center. Evaluating the patch error at each boundary center, has the advantage of approximating the variation of the view direction over the patch.

Since our metric is defined in object space, we apply the camera rotation and object transformation to the direction d. Then, starting from the lowest tessellation level, we iteratively fetch the coefficients of the quadratic polynomial using the *primitive ID* of the patch, and we evaluate the metric using equation (2.9) until the error is inferior to a user-defined pixel bound. To take into account perspective distortions, and hence the viewing distance, we project the error centered on the edge orthogonally to the view direction, making the assumption that the scaling factor and view-direction are locally constant, this is equivalent to a first-order approximation of perspective projection.

Let us call $l+1$ the lowest level satisfying the user-defined pixel bound e_{bound}. To support smooth transition we have to compute an additional fractional factor $\alpha \in]0,1]$ to transition progressively between the levels $l+1$ and l. α is computed by linearly interpolate between the pixel error e_{l+1} of level $l+1$ and the pixel error e_l of level l:

$$\alpha = \frac{e_l - e_{bound}}{e_l - e_{l+1}} \ . \tag{2.11}$$

The fractional factor α is used later to provide smooth transition between the level l and $l+1$ as detailed in Section 3.2.

2.5 Results

Figure 2.6 illustrates the view-dependent nature of our selection metric, it reports the number of rendered triangles when rotating around a single instance of the previous 3D objects. Highest rate variations are observed for the "Bricks" and "Glacier Peak" models that both exhibit a privileged displacement direction: the error introduced by the simplification is thus strongly anisotropic, which is especially well captured by our metric. In both cases, the number of rendered triangles is maximal for grazing view directions, and minimal for nearly orthogonal view directions (between 0.5 and 1.5s). Note that unlike the "Bricks" model, the "Glacier Peak" model is an extreme case where the x and y coordinates of the input reference mesh matches with the uv-coordinates. Even though the other two 3D models are closed meshes, relatively high triangle count variations (roughly $\times 2$) are still observed as the amount of geometric

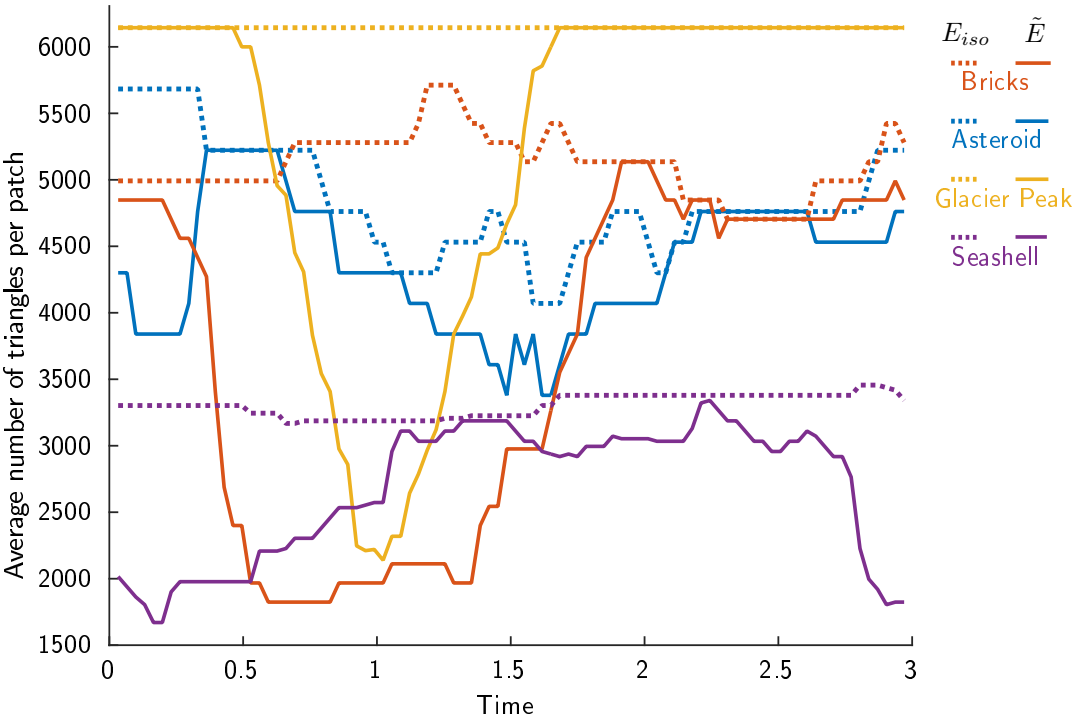


Figure 2.6 – Average number of triangles per patch along predefined view paths for our four reference models and for both the isotropic and our view-dependent error selection metric.

details is not evenly distributed among the different patches. Compared to the isotropic selection metric, the gain in triangle count range from 0% for some rare view directions up to 66%. Comparing the plots of the two metrics, we can estimate whether the variation in triangle count comes from the change of distance or orientation during the rotation. The plot of the isotropic metric for the "Seashell" model is flatter than for the others because we selected a more distant point of view: the relative distance of the patch to the camera is thus more constant during the rotation.

2.6 Discussions

Visibility From a given view direction, only a subpart of the model is generally visible due to self-occlusions. This is particularly true for patches with extreme folds but also when looking at a patch at a grazing angle. To take this into consideration, we could extend our error E by adding a visibility term $\delta_{vis}(\mathbf{v_k}, \mathbf{d})$ to filter vectors $\mathbf{v_k}$ that are not visible from the view direction $\mathbf{d}$.

$$E(\mathbf{d}) = \max_{\mathbf{v_k} \in V} (\|\mathbf{v}_k \times \mathbf{d}\| \, \delta_{vis}(\mathbf{v_k}, d)) \; . \tag{2.12}$$

However, this visibility term is not trivial to define for a vector: a vector can be partially visible. A simple approach would be to take into account only the visible part of the vector. In that case, we can define $\delta_{vis}(\mathbf{v_k}, \mathbf{d})$ as a scalar function in the range $[0, 1]$. $\delta_{vis}(\mathbf{v_k}, \mathbf{d})$ will be equal to 1 for fully visible vectors and 0 for entirely hidden vectors. For partially visible vectors, it will be proportional to the part of the vector that is visible.

Taking into account the visibility breaks the central symmetry of E. Therefore, $\tilde{E}$ will no longer be suitable for approximating such an error function. This may be solved by adding more coefficients to our approximation, for example the three first-order basis functions of real harmonics. However, a full analysis of E would be required to determine the appropriate basis functions for the approximation.

Shadows In the context of displaying highly detailed models, shadows play a crucial part for enhancing the realism. Although it is not our main focus, it is important to keep in mind how the tessellation interacts with other parts of the rendering process. As the shadow computation is based on the geometry of the models, choosing a too coarse LOD can lead to very inaccurate shadows. The two most common shadow computation techniques used for real-time rendering are shadow mapping [Williams, 1978] and shadow volumes [Crow, 1977].

The key idea behind shadow mapping is simple: everything that is visible from the light's point of view is lit and the rest is in the shadow. The shadow mapping algorithm involves two major steps. First, the scene is rendered from the light view, storing the depth of every visible surface. Then the scene is rendered from the regular viewpoint and for each fragment, by comparing the distance between the corresponding 3D point and the light position to the depth previously computed, we can determine whether it is lit or not. For the test to be coherent, it is crucial that the two rendering passes are done with the same geometry. Implying that the same tessellation level have to be set for the two passes. Taking into account only the view position and direction can lead to choose coarse LOD for occluders and thus to coarse shadows. To solve this problem, a simple approach would be to take also into account the position and the direction of the light into the LOD selection process. However for complex scenes with several lights, this could lead to over-tessellation and performance issues.

In the case of shadow volumes, the idea is to extend the silhouette of an object from the light source. Then when an object is inside these volumes, it is therefore in shadow. Changing the level between the silhouette extrusion and the actual rendering would cause shadow artifacts on silhouettes. Then as for shadow mappings, we need to use the same tessellation level for the two steps and thus have the same trade-off between shadow resolution and performance.

Chapter 3

Controllable fractional tessellation

In the previous chapter, we have seen how to choose the appropriate LOD to achieve the best performance without impacting the visual quality. This chapter focuses on rendering the LOD once the tessellation level have been selected. We present our flexible LOD blending scheme, which extends the GPU fractional tessellation in two ways. First, it avoids *swimming* artifacts by applying fractional interpolation on the displaced surface rather than in the parametric domain. Second, during the transition between two levels, any vertex of the fine level can be smoothly introduced from any nearby vertex of the coarse one. In contrast to standard fractional tessellation, this relationship needs to be stored explicitly inside a custom representation, as described in Section 3.1. Furthermore, our approach completely bypasses both the fractional and adaptive tessellation mechanisms built in the GPU: it makes only use of integer tessellation with equal factors for patch interior and boundaries. Both mechanisms are reintroduced in a custom and controllable manner, as detailed in Section 3.2 and Section 3.3 respectively.

3.1 Representation and storage

Our framework is based on a hierarchical representation of the LODs that is stored using Schäfer et al.'s linear indexing [Schafer et al., 2013]. To fetch any attribute attached to a vertex dynamically generated by the hardware tessellation, a unique index is required to identify each vertex. Current GPUs do not expose such an index, Schäfer et al. showed that it can be inferred from the patch index and the fractional barycentric coordinates of the generated vertices. Within a given patch of index i_p at a level l, each generated vertex is identified by a unique index $i_v \in [0, N(l)[$, where $N(l)$ is the number of vertices per patch at level l.

We devise a simpler indexing scheme than the one proposed by Schafer et al. [2013]. By looking at the tessellation patterns (Figure 3.1), one can

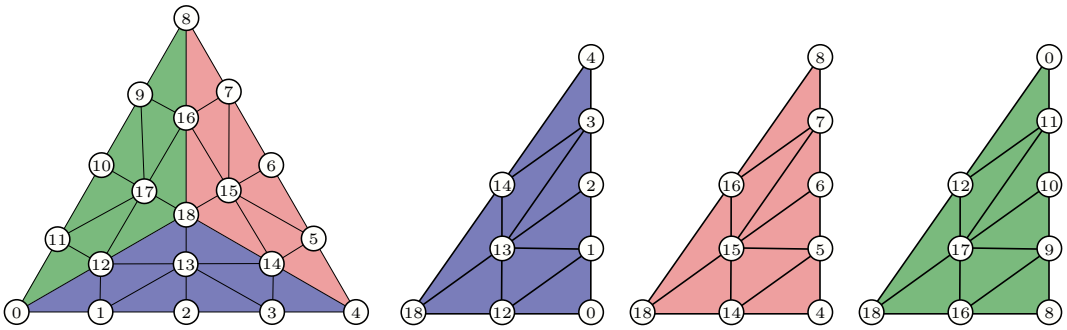


Figure 3.1 – Indexing scheme — A triangle patch at factor 6 with indices and the three patch pieces unfolded.

Algorithm 2 i_v computation

Input: barycentric coordinates $\mathbf{c}$, tessellation factor f
$(c_{min}, j) = \min(\mathbf{c})$ ▷ *Find minimum coordinate and its index*
$\mathbf{c} = \sigma^{j+1}(\mathbf{c})$ ▷ *Apply cyclic permutation σ on the coordinate*
$(x, y) = \text{round}\left(\begin{pmatrix} 1 & 1 & -2 \\ 0 & 1 & -1 \end{pmatrix} \mathbf{c} f\right)$ ▷ *Change coordinate basis*
$i_{ring} = y + x((j+1) \ \% \ 3)$ ▷ *Compute index on the ring*
if $x \neq 0$ **then**
    $i_{ring} = i_{ring} \ \% \ 3u$ ▷ *Special case of the center vertex*
end if
$i_{offset} = 3(\frac{f}{2}(\frac{f}{2}+1) - \frac{x}{2}(\frac{x}{2}+1))$ ▷ *Compute offset for other rings*
return $i_v = i_{ring} + i_{offset}$

notice that they are composed of concentric rings. The key idea behind our indexing scheme is to find the index of the ring and the position of the vertex along it. To do so, the triangular patch is divided into three parts: each part can easily be identified by looking at the smallest barycentric coordinate. Then we apply a transformation matrix to unfold this part on a regular grid as shown in Figure 3.1. The x coordinate in the grid basis informs us about the index of the ring, with the center of the patch having index 0. From the y coordinate, we can compute the offset of the vertex inside the ring. From the combination of these informations, we can deduce the final index i_v. The precise computation is detailed in Algorithm 2.

Any vertex of the tessellated mesh can thus be uniquely identified through the triplet (l, i_p, i_v). In our work, this indexing serves two purposes. First, this index is used to establish the relationship between vertices of adjacent levels as explained in the following section. Second, it permits to precisely assign any attribute (e.g., the displacement vectors) to the generated vertices by compactly storing them into a global 1D array. In our implementation, this

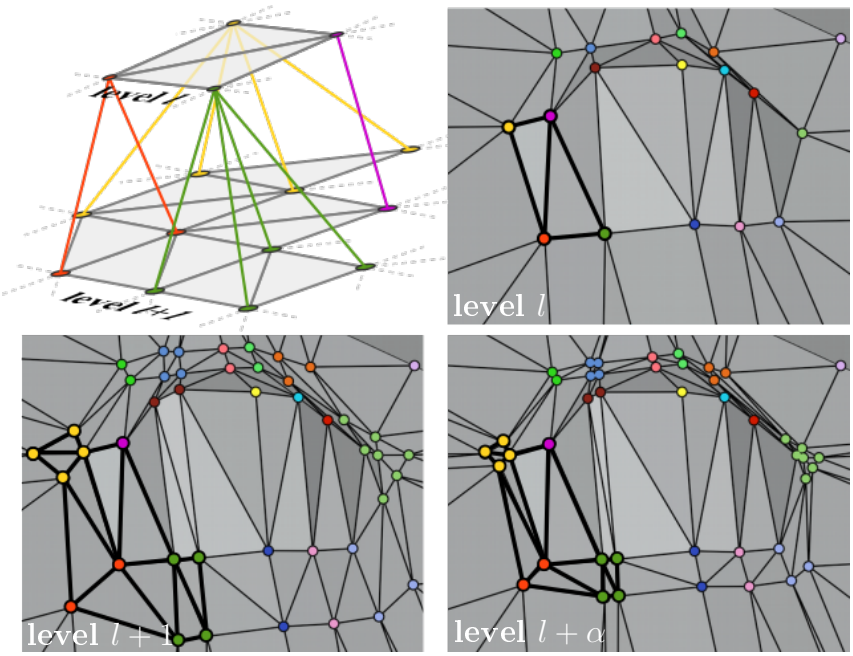


Figure 3.2 – Transition between two successive levels l and $l+1$. At the intermediate level $l+\alpha$, the vertex positions are obtained by linear interpolation between their parent positions at level l and their final positions at level $l+1$. The top-left figure depicts the respective hierarchical representation for the thicker subset of triangles.

linearization follows the lexicographic order of the triplet.

We now detail how rendering a discrete tessellation level. For each vertex we store in this 1D array a displacement vector and texture coordinates. At render-time, in the evaluation shader, we first compute the index from the barycentric coordinates, and fetch those attributes. The final position of the vertex is obtained by interpolating the position of three path corners and adding the fetched displacement. The displacement is expressed in tangent space to allow animation and deformation of the control mesh. Other attributes such as normals and colors are stored in texture and fetch using the texture coordinates in the fragment shader for high quality per-pixel shading.

3.2 Continuous LOD

To support smooth temporal transition between levels, we need to store an additional index per vertex in the 1D buffer. This index is used to establish the relationship between vertices of adjacent levels. Namely, each vertex $V = (l+1, i_p, i_v)$ is associated to a parent vertex $V' = (l, i_p, i'_v)$ of the coarser level by storing its index i'_v as an additional attribute, that we call the *blending* index.

This leads to a hierarchical representation of the LOD depicted in Figure 3.2. We now detail how vertices are interpolated during a transition from level l to the next level $l+1$.

Our hierarchical representation is directly amenable for fractional-like tessellation. For the sake of simplicity, let us consider a single patch. To interpolate between two subsequent levels l and $l+1$ at a fraction $\alpha \in]0,1]$, the idea is to smoothly move each vertex of level $l+1$ towards its parent vertex of the coarser level l as α vanishes. This behavior is depicted in Figure 3.2. More precisely, a patch at a fractional level $l+\alpha$ is first instantiated at the level $l+1$ through integer hardware tessellation, e.g., for levels restricted to power-of-two factors, it corresponds to a tessellation factor of 2^{l+1}. Then, for each generated vertex, its own and parent attributes are fetched and linearly interpolated according to the parameter α. However, in order to avoid swimming artifacts, we do not directly interpolate displacement attributes. Instead, we interpolate the object space positions $\mathbf{p}_i$ and $\mathbf{p}_{i'}$ of the current vertex i and parent vertex i' respectively.

Unlike $\mathbf{p}_i$ which is easily computed, $\mathbf{p}_{i'}$ is more challenging to obtain as we do not have direct access to its barycentric coordinates. Recovering them from the blending index i' would involve numerous prohibitive integer divisions and modulo. Thus, since the tessellation patterns are fixed, we propose to simply precompute and store them once and for all, at every needed tessellation factor. This array requires only 4291 entries in total for power-of-two factors, which is very lightweight and does not impact performance.

This representation and blending scheme are very versatile but, since the tessellation pattern of each level is fixed, the hierarchy defined by the blending indices must satisfy some constraints. First, when α vanishes, the degenerated pattern of the higher level $l+1$ must exactly match the pattern of the coarser level l. This hard constraint forces the intermediate levels to match even or odd tessellation factors. In practice storing the displacement attributes for all even or odd factors would be too expensive anyway, and we thus store and interpolate between power-of-two levels only. (For the sake of clarity, some of the illustrations of this manuscript have been made using even-levels though.) To prevent dependencies across adjacent patches, we also forbid vertices of a given patch to be paired with vertices of a different one, which explains why storing the in-patch vertex index i'_v as blending index is sufficient. As a consequence, vertices generated along patch-borders can only be parented with analogous vertices. Finally, to prevent swimming artifacts during morphing, vertices should be paired to geometrically close ones. All these constraints have to be considered during the LOD generation as described in Chapter 4.

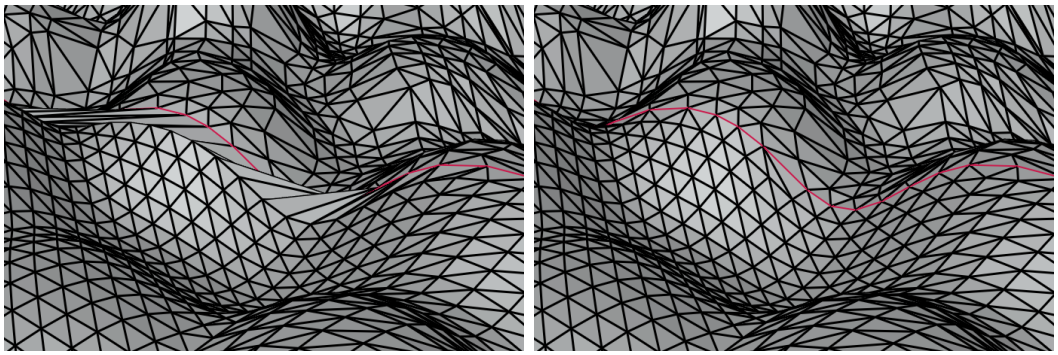


Figure 3.3 – Adaptive LOD along patch-borders. **Left:** with our LOD representation, the hardware adaptive tessellation produces invalid geometry along patch boundaries (in red) whose level differs from the interior level. **Right:** our manual stitching technique solves this problem with a negligible overhead.

3.3 Adaptive LOD

As explained in Section 1.4.3, adjacent patches can be tessellated at different resolutions, so we need to define how they can be seamlessly connected. A naive approach would be to assign a common tessellation factor to the shared patch-borders, and let the hardware tessellator connects the interior vertices and boundary vertices for us. As illustrated in Figure 3.4(a-d), since we do not necessarily pair boundary vertices in the same way the tessellator does, edges connecting the interior with the boundary can suddenly flip when its interior factor changes. This not only introduces popping artifacts, but more importantly it creates invalid geometry in saddle-like area as shown in Figure 3.3.

Our solution still defines a common fractional factor for the shared patch-borders, but we accomplish the stitching manually using our hierarchical representation as illustrated in Figure 3.4(e-f). Patch boundaries are subdivided at the same rate as the patch interior, thus completely bypassing the hardware adaptive tessellation. Let us assume that the current patch boundary is subdivided at level $l+1$ (e.g., 6 in Figure 3.4), and that the target fractional level for the boundary is $l-1+\beta$ (e.g., 3.8 in Figure 3.4). To reach this fractional level, we apply the blending scheme presented in the previous section between the parent and grand-parent of each boundary vertex. This mechanism can be applied recursively to handle resolution differences of more than one levels, even though this is very unlikely to happen especially when using power-of-two levels.

Interior vertices may have ancestors that lie on a patch boundary, hence having a different tessellation factor. Let us assume that the fractional levels of the patch interior and border are $l+\alpha$, and $l-1+\beta$ respectively. If $\beta > 1$ then the actual integer subdivision levels for both the interior and border are the same. Such a vertex will be positioned at a ratio α between its initial position $\mathbf{p}_i$ and its parent position $\mathbf{p}_{i'}$, which is the desired behavior.

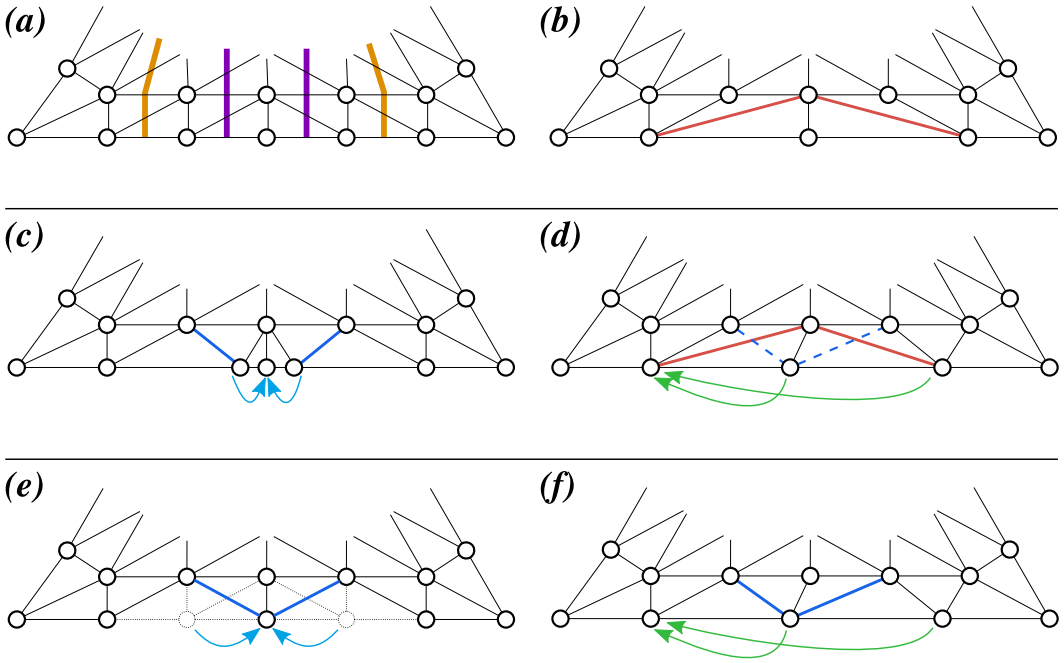


Figure 3.4 – Transition of a patch-border from factor 6 to 3.8 while the interior factor remains constant at 6. **(a)** The hardware tessellator wants to contract the orange strip to obtain the pattern in **(b)**. Our simplification algorithm decided to collapse the purple strip, which implies the blending depicted in **(c)** with blue arrows. This works as long as the integer boundary factor is 6, but once it changes to 4, the blue edges are flipped to produce the red ones **(d)**. To solve this problem, we set the integer boundary factor at the patch factor (i.e., 6 instead of 4); **(e)** we conceptually put boundary vertices at their parent position to reach factor 4 (blue arrows), and **(f)** we apply our blending mechanism with the grand-parent to achieve 3.8 fractional tessellation (green arrows).

However, as soon as β becomes smaller than one, the previous mechanism will suddenly put this vertex on the patch-border at a position $\mathbf{p}_\beta$ corresponding to the ratio β between its two subsequent ancestors ($\mathbf{p}_{i'}$ and $\mathbf{p}_{i''}$). Instead, to avoid popping artifacts, such a vertex must be progressively moved towards the patch border. This is easily accomplished by computing its actual position as the ratio α between its initial position $\mathbf{p}_i$ and target position $\mathbf{p}_\beta$ (leading to red point).

3.4 Results

We have seen that our custom controllable fractional tessellation provide both spatial and temporal smooth transition. Figure 3.6 shows two examples rendered using our custom controllable fractional tessellation. Patches with dif-

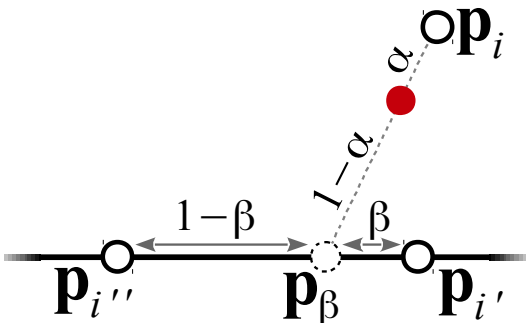


Figure 3.5 – Blending scheme for a patch interior vertex collapsing on a vertex locating on a patch boundary.

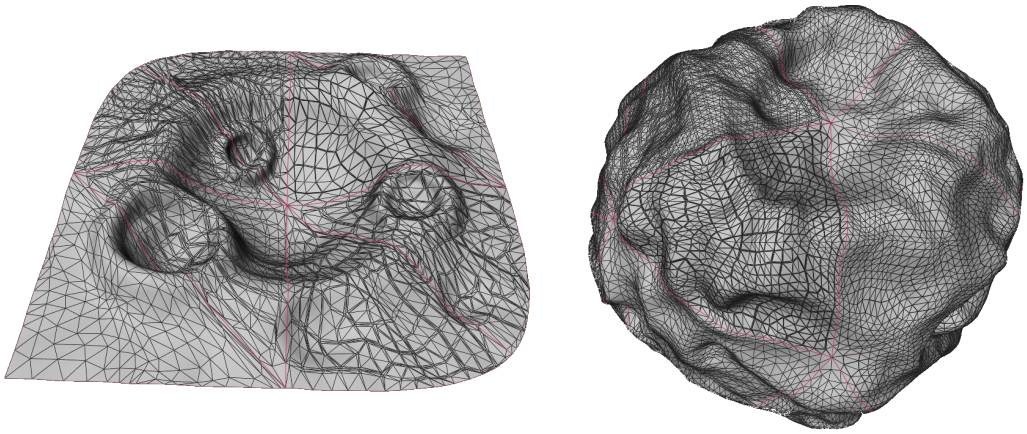


Figure 3.6 – Two models render using our custom controllable fractional tessellation. Adjacent patches with different tessellation factors are seamlessly connected.

ferent tessellation levels are seamlessly connected. In addition, thanks to our hierarchical representation, we are able to propose different transition between levels. More precisely, we can control the topological changes that happen during a transition, allowing to redistribute vertices in the areas of interest of the model. The next chapter present how to generate the hierarchical representation. More results are presented in this chapter.

3.5 Discussion

Memory consumption Regardless of the LOD rendering technique, the 3D positions of the detailed mesh (i.e., at level 64) and of the intermediate LODs need to be stored. As motivated in Schafer et al. [2013], storing those into 2D mip-mapped textures is not optimal due to empty regions in texture atlases, and the need for oversampling to guarantee an injective mapping from tessellated vertices to texel. In contrast, storing them per vertex is memory

optimal. The memory overhead of our approach comes from one integer index per vertex for the blending, and two UV-coordinates to apply textures. In our prototype implementation, each scalar attribute is simply stored as a 32 bits float or integer value, leading to a total of 24 Bytes per vertex: 3 floats for displacement, 2 floats for texture coordinates, and one integer for the blending index. Neglecting per-patch attributes, this implies a cost of $24 \times N$ Bytes to store the N vertices of the finest level. For intermediate power-of-two levels, the number of vertices is divided by four. To store all attributes for all power-of-two levels, it implies a cost of about $32 \times N$ Bytes. Our approach thus remains very memory efficient since simply storing the detailed textured mesh as a traditional indexed-triangle-list without LOD would already requires $44 \times N$ Bytes because of the storage of the connectivity, and previous progressive LOD rendering systems based on vertex-splits require at least $69 \times N$ bytes [Liang Hu et al., 2010].

Moreover, our storage requirement could be significantly reduced in two ways. First, the blending indices can be stored as 12 bits integers for tessellation factors up to 64 (which is the limit of current GPUs), and the texture coordinates could be heavily compressed by expressing them relatively to the interpolated patch UVs, for instance using 16 bits fixed-point precision scalars. Likewise, displacement vectors could likely be compressed the same way without impact on the visual quality. Second, one could limit the maximum tessellation factor on a per patch basis, similarly to the work of Schafer et al. [2013], thus saving the finer level storage for patches of lower geometric complexity.

Power-of-two factors Similarly to Schäfer et al.'s method for swimming-free displacement mapping, the main limitation of our approach is its practical restriction to power-of-two factors. Even though our approach can handle a finer granularity up to even-factors, the memory required to store all the levels becomes prohibitively expensive. For power-of-two factors, the required memory is approximatively doubled compared to the original mesh, whereas it is squared in the case of even-factors. We argue that this limitation is largely compensated by LODs of much higher geometric quality allowing to use lower tessellation factors for the same visual quality.

In addition, the transition between two even-factor levels is very unlikely to reduce the visual error unless all the vertices are slightly redistributed. Yet, in this case, temporal artifacts will be reintroduced since all vertices will constantly and very rapidly move back and forth. On the other hand, we agree that even-factor levels are very useful when over-tessellating the mesh such that the projected tessellated triangles are below one pixel: swimming artifacts are effectively avoided, but at a prohibitive rendering cost.

Indexing The indices computed from barycentric coordinates are used to fetch attributes from a 1D texture. Achieving coherent access in this buffer

can provide an important performance boost. To obtain coherent access, the indexing scheme has to match the scheduling of the vertices generating by the tessellator. Using the transform feedback buffer of the geometry shader, we recovered the scheduling of the vertices for one patch. Unfortunately, our indexing scheme does not match perfectly with this scheduling. We experiment an alternative indexing scheme matching this scheduling, however, the computation was too costly and it runs slower than our indexing scheme.

Ideally, this index could be directly provided by the driver. If future hardware would expose such index, it would remove completely the overhead of the index computation and enable coherent access to fetch attribute data, improving the performance and simplifying the evaluation shader implementation.

Chapter 4

Strip-based Mesh Simplification

To exploit the flexibility of our representation presented in the previous chapter, we devise a feature-aware simplification algorithm that computes the final position of the vertices at every level, and the blending indices required to smoothly transition between those. Our general objective is to distribute the vertices of each level to best represent the most significant features of the input geometry. Whereas this is a well studied topic in the literature [Botsch et al., 2010], our context implies two specific problems. First, at each level, the simplified geometry must coincide with the predefined tessellation pattern. Second, as seen in Section 3.2, we need to find a continuous mapping between two subsequent levels which guarantees that the collapsed topology of the mesh at level $l+1$ matches the topology at level l when the transition starts.

The first problem could be addressed by applying surface fitting methods [Kobbelt et al., 1999; Yeh et al., 2011; Nivoliers et al., 2014] on each level. However, finding a consistent match between the levels as requested by the second criterion would then be extremely difficult, and even impossible without adding explicit dependencies between the levels during their optimization. In contrast, this second problem is naturally overcome using any of the iterative mesh decimation algorithms we discussed in Section 1.3.2. In this case the matching can be directly established during edge-collapse. On the other hand, it becomes extremely difficult to make the decimation process to converge to a predefined mesh connectivity.

Actually, for both approaches, the core of the problem can be reduced into the problem of deciding whether a given graph is a *minor* of another, which is NP-complete. If the given graph is fixed, it can be solved in polynomial time [Kawarabayashi et al., 2012], yet with a constant that depends super-exponentially on the size of the graph. This problem is thus intractable in the general case. However, we are not dealing with arbitrary graphs and we know at least one solution: the default transition of the hardware tessellation engine (Figure 4.1). We show in the following how to construct many others.

We also argue that both problems need to be solved at the same time, and

Algorithm 3 Strip-based mesh simplification

Input: tessellated coarse mesh at factor $f \leftarrow f_{\max}$
+ reference mesh
loop
  if f is power-of-two **then** ▷ *desired tessellation level*
    Refit current mesh to the reference mesh. ▷ *4.7*
    Compute $\tilde{E}$ for each patch. ▷ *2.2*
    Store attributes for the current level.
  end if
  if $f = f_{\min}$ **then** *break.*
  for each patch corner **do**
    Update $\tilde{E}$ for each adjacent patch.
    Find the strip minimizing ▷ *4.4*
    the edge selection cost C_{edge}. ▷ *4.6*
    Contract the edges of the strip. ▷ *4.5*
  end for
  $f \leftarrow f - 2$
end loop

we thus follow the principle of edge-collapse decimation algorithms, and extend them to a strip-based flavor which produces at every step a mesh compatible with hardware tessellation.

4.1 Algorithm overview

Our simplification procedure is sketched in Algorithm 3. Our method takes as input a detailed reference mesh and a corresponding decomposition into fully tessellated patches, called *level 0*, matching the fixed hardware tessellation pattern. Depending on the production pipeline, the level 0 can be obtained by different means; one option is to start from a coarse decomposition into patches uniformly tessellated in the parametric space, and to sample the reference mesh or a displacement map, as detailed in Section 4.2.

From those two input meshes, our method produces a hierarchy of LOD compatible with hardware tessellation that minimize our view-dependent error with respect to the reference mesh.

Starting from the level 0, the next level is constructed by collapsing a set of adequate strips of edges. We properly defines the set of feasible contractions in Section 4.3, and we show how to quickly explore it to find the best contractions in Section 4.4. We take advantage of our error metric to better drive the edge-collapses (Sections 4.5 and 4.6) and to design a global optimization procedure that refits the vertex positions of the simplified mesh to the reference surface

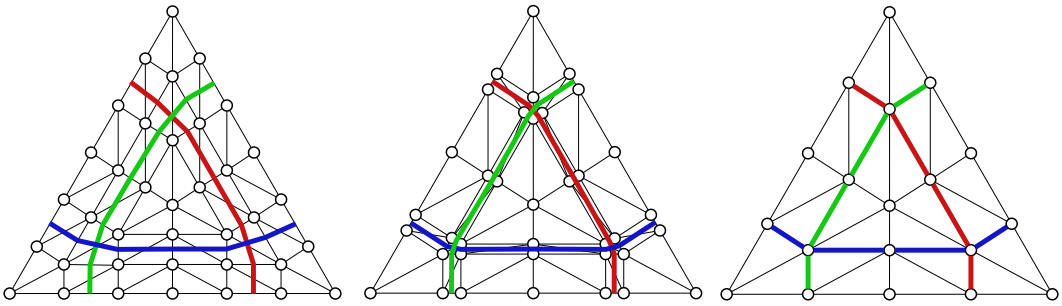


Figure 4.1 – Fractional tessellation patterns at factors 6, 4.9 and 4. Colored lines highlight the three strips which are collapsed/split during the transition.

(Section 4.7).

4.2 Level 0 initialization

To ensure that the first level in the LOD hierarchy matches the fixed hardware tessellation pattern, we take as input the detailed reference mesh and a coarse decomposition into triangular patches sharing the same parametrization. The decomposition is either obtained from the modeling process itself, or a posteriori using any decimation algorithm that preserves the texture coordinates.

The initial level 0 is then generated by instantiating each patch of the coarse mesh at the highest tessellation factor ($f_{\max} = 64$ for current GPU) with texture coordinates linearly interpolated from the patch corners. The first refitting process (step 4 of Algorithm 3) will take care of computing the respective 3D positions. Better starting points could be obtained by non-uniformly spreading the vertices in the parametric space [Yuan et al., 2016], but we did not investigated such an approach.

4.3 Feasible contractions

For triangular patches, the default transition of the tessellation engine consists in collapsing three "straight" strips as shown in Figure 4.1. We extend this notion of strip to give more flexibility to the transitions. To effectively transition between even tessellation levels, we first observe that $3f$ edges need to be collapsed, including two edge-collapses per patch boundary. To preserve the regularity of the pattern, pairs of collapsed boundary edges have to be connected through strips of collapsed edges, with one strip per patch boundary. To make the definition of such strips tractable, we restrict them to have equal length and to connect a pair of "opposite" half patch boundaries. An example is given in Figure 4.2(a). In other words, each patch boundary is associated to one strip of $f + 1$ graph nodes connecting half-parts of its two

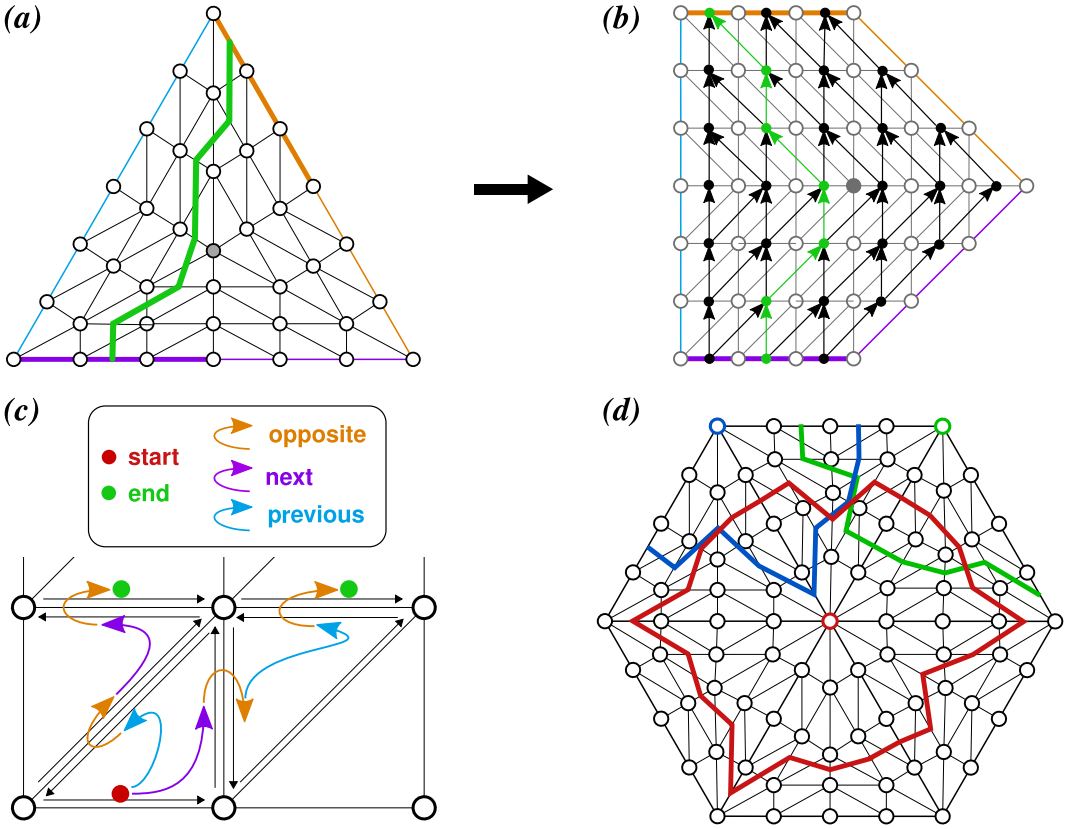


Figure 4.2 – **(a)** A strip (in green) is a continuous list of edges joining two "opposite" half-patch-borders (thick purple and orange lines). **(b)** Unfolding the patch into a uniform grid, a strip can be seen as a path in the DAG constructed over the patch edges. **(c)** This implicit graph can be easily traversed walking on a half-edge data structure. **(d)** Strips extend to x-strips when they crosses multiple patches since they have to be connected on patch boundaries. Closed x-strips form rings (in red).

adjacent patch boundaries. For a given boundary, its set of possible associated strips is easily defined after a reparametrization of the triangle such that its vertices lie on a uniform grid with the boundary and half-boundaries aligned on the vertical and horizontal axes respectively, as depicted in Figure 4.2(b). Since the length of a strip is fixed, the only possibility is to perform one edge-collapse per horizontal line direction. Each horizontal edge being connected by two adjacent triangles, this operation produces a continuous strip, with the additional requirement that collapsed edges form a continuous strip.

The set of possible strips going from one half-boundary to the opposite one are best defined as a directed acyclic graph (DAG) whose nodes are the edges of the mesh. In this dual representation, one strip corresponds to one path from a half-boundary to another. In practice, this graph does not have to be explicitly constructed. As illustrated in Figure 4.2(c), using an half-edge data structure, a given node h has at most two children that are reached using the following

operator compositions: op(next(op(prev(h)))) and op(prev(op(next(h)))). If the intermediate edge reached by prev(h) or next(h) is a patch-border, then the respective child does not exist.

Discussions In order to easily track the set of feasible contractions, we restrict this set to strip collapses. As a result, our algorithm ignores a few possible contractions that could potentially lead to even more accurate LODs. The entire space of solutions could be explored using edge-collapses and a brute-force backtracking algorithm to handle configurations that do not match the target pattern. However, as the problem is NP-complete, this approach is only feasible for very coarse levels. A potential direction to explore would be to start from a feasible strip obtained using our algorithm and apply path mutations in the spirit of Thorpe [1984]: small modifications can be applied to a feasible strip to explore step by step the space of all possible contractions. Although, it will not give the optimal solution, it would allow to find a local optimal solution. The challenge is then to define the set of possible modifications allowing to pass from one solution to another.

4.4 Strip collapse

Our LOD simplification algorithm consists in finding the strips that minimize some feature-preserving error metric. We define the error of a strip as the sum of the error of each edge-collapse. Any cost function can be used, and we will present in Section 4.6 one based on our view-dependent error metric that we presented in Chapter 2.

Usually the coarse mesh is composed of several adjacent patches and we need to ensure the coherence between them. As illustrated in Figure 4.2(d), strips ending along the same half-patch-border must be connected, or cracks will appear. This creates extended-strips (x-strips) that must be considered as a whole during the error minimization. Each x-strip can be associated to a unique vertex of the coarse mesh, and vice versa. Its length is then proportional to the valence of the coarse mesh vertex. If the vertex lies on an open boundary then the two extremities are not constrained. Otherwise, the x-strip must form a closed ring (e.g., the red line in Figure 4.2(d)).

Using the aforementioned dual representation of the feasible strips, finding the best x-strip associated to a given vertex of the coarse mesh boils down to a shortest path search. For an open x-strip, both starting and ending nodes can be freely chosen. This case can be solved in a single pass by connecting all starting and ending nodes to two virtual nodes, and computing the shortest path between those. For a closed x-strip, the ending node must coincide with the starting node. In this case, we pick one arbitrary patch boundary and run a shortest path algorithm for each of the $f/2$ possible starting/ending nodes

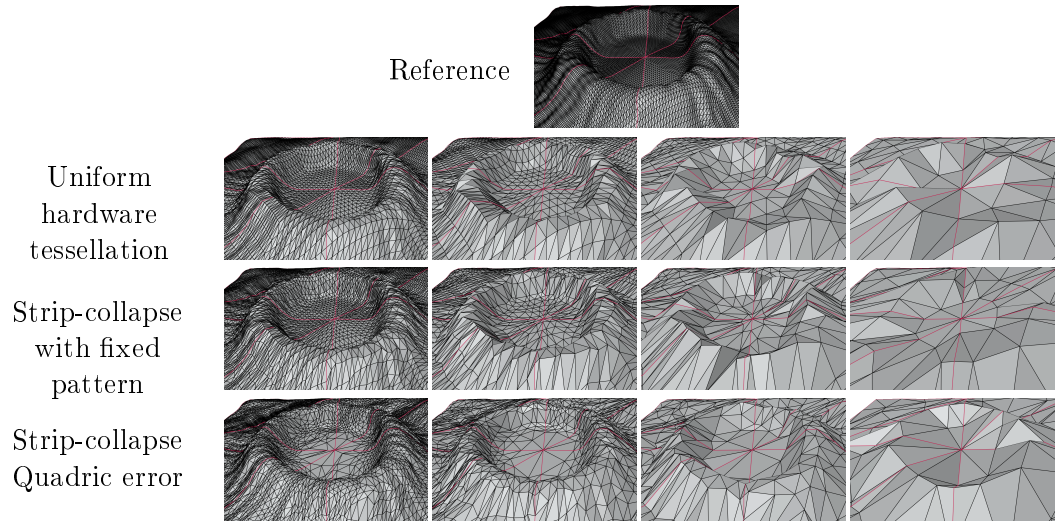


Figure 4.3 – Starting from the same 8 patches and displacement map, comparison of the LOD representation generated by three approaches for the factors 32, 16, 8 and 4 (from left to right). **Top:** Uniform hardware tessellation. **Middle:** Mesh decimation constrained to choose the same strips as the tessellation pattern. **Bottom:** Our method, i.e., mesh decimation collapsing the strips that minimize an error metric. For each level, our approach better preserves the input geometry by spatially redistributing the polygonal size and density.

lying on the corresponding half-patch-border. Since we are dealing with a directed acyclic graph, each search is efficiently accomplished using a variant of Dijkstra algorithm with linear complexity [Cormen et al., 1990]. It is also possible to compute in a single pass the best ring, using some form of advancing front techniques (see Algorithm 5 in Annex for more details).

Once an optimal x-strip has been found, each of its edges is collapsed to a single vertex positioned at the minimum of the error metric.

In Figure 4.3 we compare our strip-based decimation algorithm with regular uniform hardware tessellation and a variant of our strip-based decimation algorithm which is constrained to choose the straight strips of the tessellation pattern. At every level, our approach better represents the features of the input geometry by simplifying flat areas while preserving regions of higher curvature, such as the top of the "crater".

Discussions There are fundamental differences between our strip-based simplification algorithm and classical edge-collapse decimation. The latter contracts one edge after the other, selecting the one with the smallest error every time. After each collapse, the error of surrounding edges must be recomputed, and a partially sorted list of edges must be updated. In our case, edge-collapses defining a strip are generally not independent: Depending on the metric, the cost and the placement of the vertices may be affected by adjacent collapses.

If that is the case, then both the selected strip and result of the contractions will depend on the arbitrary choice of the contraction order along the strip. Otherwise, the influence between strips is marginal, and the order in which strips are selected and collapsed has a minor impact on the final result This is because the three strips of a patch travel in a different direction of contraction, and each strip crosses the two others only once.

In our implementation, x-strips are processed in an arbitrary order. The dependence between edge-collapses within a strip are thus not taken into account, leading to a suboptimal solution. Considering the dependences between collapses is not possible with a Dijkstra variant algorithm. Owing to the inter-dependences, the whole path have to be considered to compute its cost, increasing considerably the complexity of our problem. As stated in the previous section, path mutations could improve our solution as it will allow, once an initial solution obtained, to find a local optimal solution taking into account edge-collapse dependences.

We experiment different order of contraction for edge within a x-strip such as increasing or decreasing cost order, but the results were not conclusive. Therefore, once the x-strip has been selected, the order of edge contractions is only of minor importance.

4.5 Vertex placement strategy

In this section we revisit the local vertex placement strategy applied during the contraction of an edge (step 12 in Algorithm 3) based on our view-dependent metric we developed. Our view-dependent metric heavily relies on the texture coordinates of the simplified mesh, it is thus crucial to preserve a good parametrization during the edge-collapse simplification.

To this end, we propose a new placement strategy which optimizes jointly the position and texture coordinates of the merged vertices. Our approach borrows the unsigned volume optimization constraint of Lindstrom and Turk [1998] but expresses it in a mixed texture and position space. Our construction is depicted in Figure 4.4. For each face $T_k = (k_0, k_1, k_2)$ adjacent to the collapsed vertices $\mathbf{p}_0^e$ and $\mathbf{p}_1^e$, we derive three volume minimization objectives, one for each position component, by constructing tetrahedrons from the position component and the two uv-coordinates of the vertices.

Let $\tilde{\mathbf{p}} = (x, y, z, u, v)$ be the unknown position and texture coordinates of the merged vertex. For the x component, the unsigned volume objective can be written as:

$$f_x(\tilde{\mathbf{p}}) = \sum_{T_k} \left([u\, v\, x]\, \mathbf{n}_{T_k}^{uvx} - \begin{vmatrix} u_{k_0} & u_{k_1} & u_{k_2} \\ v_{k_0} & v_{k_1} & v_{k_2} \\ x_{k_0} & x_{k_1} & x_{k_2} \end{vmatrix} \right)^2 \tag{4.1}$$

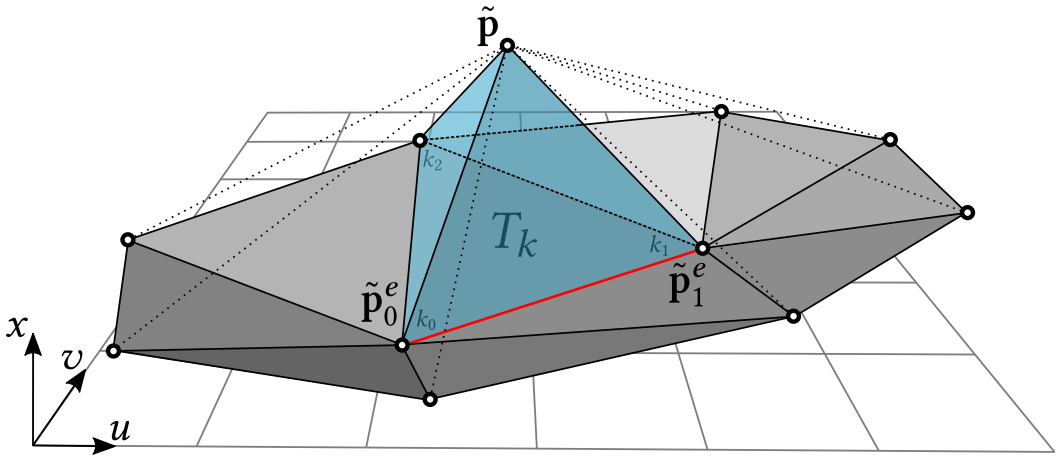


Figure 4.4 – Construction of our vertex placement metric – the two vertices $\tilde{\mathbf{p}}_0^e$ and $\tilde{\mathbf{p}}_1^e$ are collapsed into the merged vertex $\tilde{\mathbf{p}}$. For each of the three position components, here x, each adjacent face $T_k = (k_0, k_1, k_2)$ is connected to $\tilde{\mathbf{p}}$ to form tetrahedrons in the (u, v, x) space (one of them is highlighted in blue). Their unsigned volumes are minimized in the least-square sense to find both the 3D position and uv-coordinates of $\tilde{\mathbf{p}}$.

where $\mathbf{n}_{T_k}^{uvx}$ is the unnormalized normal of triangle T_k obtained as the cross product of two edges in the (u, v, x) space.

$$\mathbf{n}_{T_k}^{uvx} = \begin{vmatrix} u_{k_1} - u_{k_0} \\ v_{k_1} - v_{k_0} \\ x_{k_1} - x_{k_0} \end{vmatrix} \times \begin{vmatrix} u_{k_2} - u_{k_0} \\ v_{k_2} - v_{k_0} \\ x_{k_2} - x_{k_0} \end{vmatrix} . \tag{4.2}$$

Deriving similar objectives for the y and z components and summing them, we define the following objective function:

$$f(\tilde{\mathbf{p}}) = f_x(\tilde{\mathbf{p}}) + f_y(\tilde{\mathbf{p}}) + f_z(\tilde{\mathbf{p}}) . \tag{4.3}$$

The objective $f(\tilde{\mathbf{p}})$ is scale-independent but yields a plane of solution when all the adjacent faces become coplanar in the five dimensional space. We thus add a regularization term attracting the new vertex towards the center of the collapsed edge, the full objective function is:

$$F(\tilde{\mathbf{p}}) = f(S\tilde{\mathbf{p}}) + \epsilon \left\| \tilde{\mathbf{p}} - \frac{\tilde{\mathbf{p}}_0^e + \tilde{\mathbf{p}}_1^e}{2} \right\|_S^2 \tag{4.4}$$
$$\text{where} \quad \|x\|_S^2 = x^T S x$$

To make this term scale-independent, we normalize the positions and texture coordinates by a scaling matrix S constructed from the length of the diagonal of the bounding box enclosing the adjacent faces in 3D space and

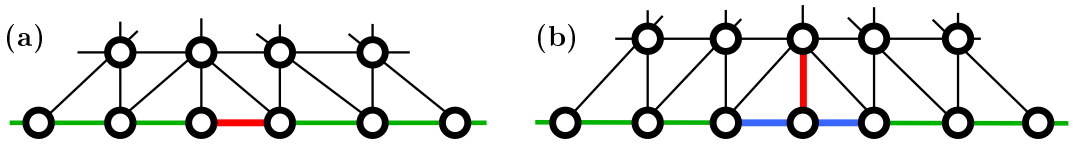


Figure 4.5 – Edge-collapses involving a patch boundary (green) – the position of the merged vertex is either constrained **(a)** on the collapsed-edge (red), or **(b)** on the two adjacent boundary edges (blue).

parametric space respectively. This non-uniform scaling further allows us to set the ϵ factor once and for all, irrespectively of the input mesh. To this end, we analyzed the numerical conditioning of this energy without the regularization term for typical mesh configurations, and empirically we found that taking $\epsilon = 10^{-4}$ is a good trade-off between the placement flexibility and numerical stability.

Associated to this new placement strategy, the natural cost for collapsing the corresponding edge is the residual of $f(\tilde{\mathbf{p}})$ (equation (4.3)) without the aforementioned regularization and rescaling. To the best of our knowledge, this is the first edge-collapse cost function taking into account both the geometric and parametric distortions in a scale invariant manner.

Patch boundaries. The GPU tessellation requires that vertices of the patch boundaries remain on the boundary edge, otherwise artifacts may appear when a boundary edge receives a different tessellation level than the patch interior. However an edge-collapse can affect a patch boundary in two situations (red edges in Figure 4.5): (1) when contracting an edge belonging to the boundary, and (2) when collapsing an edge that joins a vertex on the boundary and a vertex on the patch interior. In both cases, the texture coordinates of the merged vertex needs to be a linear interpolation of two vertices on the boundary. There is only one possibility in the former configuration, but two in the latter as depicted by the blue edges in Figure 4.5b. In this case, we evaluate the solution for both edges, and keep the one with minimal residual. Moreover, to handle degenerated cases in this particular configuration, we use a modified regularization term in equation (4.3) that attracts the merged vertex toward the edge extremity which is on the patch boundary.

In practice, to constrain the texture coordinates on an edge, the unknowns u and v are replaced by a new unknown α such that:

$$[u, v] = \alpha[u_0^e, v_0^e] + (1 - \alpha)[u_1^e, v_1^e] \quad \text{with } 0 \leq \alpha \leq 1.$$

With such a formulation, the objective function of equation 4.3 leads to a quadratic problem that can be solved similarly for patch boundaries, open mesh boundaries and texture coordinate discontinuities.

Discussions. This objective function shares strong similarities with our view-dependent metric derived in Chapter 2: for a single position component, minimizing the unsigned volumes of the attached tetrahedrons corresponds to minimizing the continuous integral of the difference function $\mathcal{V}$ computed between the two meshes before and after the contraction. Even though the difference vectors are not taken relative to the reference mesh, it establishes an indirect relation with our error metric E.

4.6 Edge-collapse Selection Strategy

All edge-collapse simplification algorithms use an edge selection strategy to prioritize contractions. The cost-function that we just defined in Section 4.5 is already a great improvement over prior work, but it remains very local and it is only indirectly related to our patch-based metric E. To get closer to the minimization of E, we propose to modify this strategy to favor edge-collapses that do not increase E from any view direction, and thus results in a better distribution of the error over the entire patch (step 11 of Algorithm 3).

Each contraction modifies the difference vectors in the one-ring neighborhood of the collapsed-edge. As explained in Section 2.1, we can efficiently extract these difference vectors using two 2D AABB-trees built in parametric space over the reference and simplified meshes. As the reference mesh is constant, its associated AABB-tree only needs to be computed once and is used thorough the simplification process. The simplified mesh AABB-tree, however, would need to be updated after each edge-collapse. In our implementation, this AABB-tree is used only once to initialize, for each face of the simplified mesh, the list of the reference mesh vertices that map to this face. This list is then quickly updated after each edge-collapse by re-assigning the difference vectors in the one-ring neighborhood of the collapsed-edge to the new adjacent faces of the merged vertex. Since the computation of edge intersections is a costly operation, difference vectors originating from such edge-edge intersections are ignored.

The natural solution to evaluate whether the error metric would be increased by this contraction would be to compute the view-dependent metric before and after each tentative edge-collapse, but repeatedly solving the convex quadratic problem of equation (2.10) for each edge would be prohibitively expensive. We thus propose the following faster alternative.

First, before each strip-collapse, we ensure that the coefficients of our analytic error metric $\tilde{E}_{curr}$ is up-to-date for each considered patch using equation (2.10) (step 10 of Algorithm 3). Since the number of strip-collapses is orders of magnitude smaller than the number of edge-collapses, this is a reasonable compromise. Then, for each tentative edge-collapse, following equa-

tion (2.3), we define the error

$$E_{col}(\mathbf{d}) = \max_{\mathbf{v}_k \in V_{edge}} \|\mathbf{v}_k \times \mathbf{d}\|$$

introduced by the difference vectors V_{edge} affected by this contraction. To speed up the computation, the set V_{edge} is again reduced to its convex hull, as in Section 2.2. Finally, we compare E_{col} with $\tilde{E}_{curr}$ for a reasonably small set of directions D, and average the positive differences, yielding the following cost function:

$$C_{edge} = \frac{1}{|D|} \sum_{\mathbf{d} \in D} \max(0, E_{col}(\mathbf{d}) - \tilde{E}_{curr}(\mathbf{d})). \tag{4.5}$$

Negative differences must be ignored because they only imply that the corresponding vectors are not contributing to the error in this direction, not that the error E would decrease. In our experiments we used 200 uniformly distributed directions.

Then, in step 11 of Algorithm 3, we search for the strip in the binary DAG whose sum of edge costs is the lowest. At each node of the graph, if the two possible paths have the same cost, which is often the case when contractions do not increase the error in any direction (i.e., $C_{edge} = 0$), we fallback on the standard edge selection cost of the respective simplification algorithm (i.e., $f(\tilde{\mathbf{p}})$ of equation (4.3) in the case of our placement strategy). In practice this implies that both costs need to be computed and propagated.

4.7 Refitting

In this section, we detail our global refitting step that is applied after reaching each power-of-two levels (step 4 in Algorithm 3). During edge-collapses, merged vertices are optimized one after the other, making impossible to reach any local minimal of any global error. The goal of this additional optimization step is thus reach such a local minimum by refitting the vertex positions of the simplified mesh to the input mesh. This refitting step also compensates for the fact that all local placement strategies (including ours) does not retain any precise information from the initial input mesh. Moreover, since our local placement strategy does not include any volume-preservation term, the model can locally shrink or grow during the simplification. This effect is annihilated by this refitting step. As a matter of fact, we deliberately designed our local placement strategy without any volume-preservation term because we planed in advance to include such a global refitting step. Moreover, it would not have been possible to include such a term without strong scale dependence issues.

Taking advantage of our view-dependent error metric, our goal here is to globally optimize the vertex positions such as to reduce the error $E(\mathbf{d})$ for all

patches and any view direction $\mathbf{d}$. Since it would be intractable to directly minimize our metric E, we devise a related but simpler optimization problem. Let us first observe that the error E is highly related to the magnitudes of the difference vectors. Therefore, reducing them should also reduce E. To this end, we assume that the texture coordinates (u_j, u_j) of the vertices of the current simplified mesh are fixed, and we seek to find their respective 3D positions $\mathbf{q}_j$ that minimize the length of the difference vectors $\mathbf{v}_k$ of V, i.e., that minimize the average distance between the simplified and reference meshes in the least-square sense. This can be formulated as the following weighted least-square optimization problem:

$$\min \sum_{\mathbf{v}_k \in V} w_k \left\| \mathbf{v}_k \right\|^2 \ , \tag{4.6}$$

where, for now, $w_k = 1$, and V is the set of all difference vectors linking the reference and simplified meshes. For a given vertex $(\mathbf{p}_i, u_i, v_i)$ of the reference mesh, the respective difference vector $\mathcal{V}(u_i, v_i)$ going to the point $\mathcal{M}_s(u_i, v_i)$ of the simplified mesh can be expressed using barycentric coordinates as:

$$\begin{aligned} \mathcal{V}(u_i, v_i) &= \mathcal{M}_s(u_i, v_i) - \mathbf{p}_i \\ &= \sum_{j \in T_i} \lambda_j \mathbf{q}_j - \mathbf{p}_i \, . \end{aligned}$$

The barycentric coordinates λ are computed in parametric space by finding the triangle T_i of the simplified mesh that contains the point (u_i, v_i), such that $(u_i, v_i) = \sum_{j \in T_i} \lambda_j (u_j, v_j)$. Reciprocally, for a vertex $(\mathbf{q}_j, u_j, v_j)$ of the simplified mesh, we get:

$$\mathcal{V}(u_j, v_j) = \mathbf{q}_j - \sum_{i \in T_j} \lambda_i \mathbf{p}_i \, .$$

Using the same logic for the difference vectors corresponding to the edge intersections, we end up with a very sparse linear system where the components x, y, z of the unknown $\mathbf{q}_j$ are independent:

$$\min \left\| \mathbf{CQ} - \mathbf{P} \right\|_w^2 \ , \tag{4.7}$$

where $\mathbf{Q}$ is the matrix of the unknown $\mathbf{q}_j$, $\mathbf{P}$ the matrix of the corresponding position $\mathbf{p}_i$ on the reference mesh, and $\mathbf{C}$ the matrix doing the matching. In our implementation, we solve this system through the normal equation and the direct sparse Cholesky solver of the Eigen library [Guennebaud et al., 2016].

The error E can be further reduced by observing that E is mostly influenced by the longest difference vectors, whereas reducing the length of the shorter vectors, that are unlikely to take part in the convex hull of V, would leave the global error unchanged. In order to take into account this behavior while

Algorithm 4 Refitting step

1: **Input:** Reference mesh $\mathcal{R}$, Mesh $\mathcal{M}$
2: $\mathcal{V} \leftarrow$ Compute difference vectors between $\mathcal{R}$ and $\mathcal{M}$
3: $(\mathbf{C}, \mathbf{P}) \leftarrow$ Construct linear system from $\mathcal{V}$
4: $\mathbf{W_0} \leftarrow$ Identity
5: **loop** ▷ *t is the iteration number*
6: $\quad \mathbf{Q_t} \leftarrow (\mathbf{C}^T\mathbf{W_t}\mathbf{C})^{-1} - \mathbf{C}^T\mathbf{W_t}\mathbf{P}$
7: $\quad \mathbf{V_t} \leftarrow \mathbf{CQ} - \mathbf{P}$
8: $\quad e_t \leftarrow$ Evaluate integral of $\tilde{E}$ for $\mathbf{V_t}$ over the mesh
9: $\quad$ **if** $e_{t-1} \leq e_t$ **then**
10: $\quad\quad$ **break**
11: $\quad$ **end if**
12: $\quad \mathbf{W_{t+1}} \leftarrow$ Comnpute weights from $\mathbf{V_t}$ ▷ *equation (4.8)*
13: **end loop**
14: Apply $\mathbf{Q_{t-1}}$ on $\mathcal{M}$

keeping a simple and fast optimization procedure, we extend the above least-square optimization by iteratively re-weighting the difference vectors according to their ratio to the largest vector:

$$w_k = \gamma + \frac{\|\mathbf{v}_k\|}{\sup_{\mathbf{v}_i \in V} \|\mathbf{v}_i\|} , \tag{4.8}$$

where γ is a small constant to avoid close to zero weights for small vectors ($\gamma = 0.02$ in all our experiments). In practice, after a first refitting iteration with unit weights, we apply a few passes of weighted refitting until the mean error over all directions stops decreasing (Algorithm 4). Note that, at each iteration, we only need to recompute the weights; the above barycentric coordinates only need to be computed once.

This iteratively re-weighted refitting process is applied each time a tessellation level is stored (step 4 in Algorithm 3). To avoid cracks between adjacent patches tessellated at different levels, the position and texture coordinates of the patch corners cannot change during the simplification. Consequently, they are only optimized during the initial refitting step of the level 0 and considered fixed during the subsequent ones.

As the refitting is applied for power-of-two levels only (steps 4 of Algorithm 3), it is reasonable to compute edge-edge intersections. Furthermore, since the computation of the error metric $\tilde{E}$ (steps 5 of Algorithm 3) exploits the same set of difference vectors, it can thus be established only once for both.

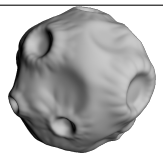
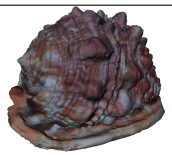
	Asteroid	Bricks	Seashell	Glacier Peak
reference mesh				
# triangles	491k	786k	1474k	786k
# patches	20	32	60	32

Figure 4.6 – Test scenes used for the evaluation.

mesh	**Total**	**VolUV** $+\tilde{E}$	**Edge-edge intersec.**	**Refitting**	C_{edge}
Asteroid	219	10	2	6	158
Bricks	504	17	6	10	294
Seashell	1227	35	18	21	663
Glacier Peak	475	17	6	9	275

Table 4.1 – Pre-computation time (in seconds) of our full LOD generation method and its main components.

4.8 Results

We evaluated the performance of our method on a Intel i7-4790K @ 4GHz CPU with a Nvidia Geforce 980 GTX. In our experiments, the target tolerance error for the LOD selection has been set to 1 pixel for a window resolution of 1920 × 1140 pixels.

4.8.1 Pre-computation times

The pre-computation times of our LOD generation procedure is reported in Table 4.1 for the same four meshes with the breakdown for the main components of our algorithm. Our new vertex placement strategy is referenced as *VolUV*, for *Volumetric UV* optimization. The column *VolUV* represents the cost of the overall generation process if the edge-edge intersections, refitting, and our edge-collapse cost (equation (4.5)) are omitted. It includes the initialization and propagation of the vertex-to-mesh mappings as well as the resolution of the QP problems to compute $\tilde{E}$. This part of the algorithm is rather fast, and the LS refitting step and the computation of the edge-edge intersections add only a small overhead to the generation process. In contrast, our edge selection cost C_{edge} greatly dominates the overall cost. However, our C_{edge} implementation is not optimized at all, and we believe that its cost could be reduced by one or two orders of magnitude by computing this cost in parallel for each edge, and exploiting SIMD instruction sets to evaluate E_{col} over hundreds of

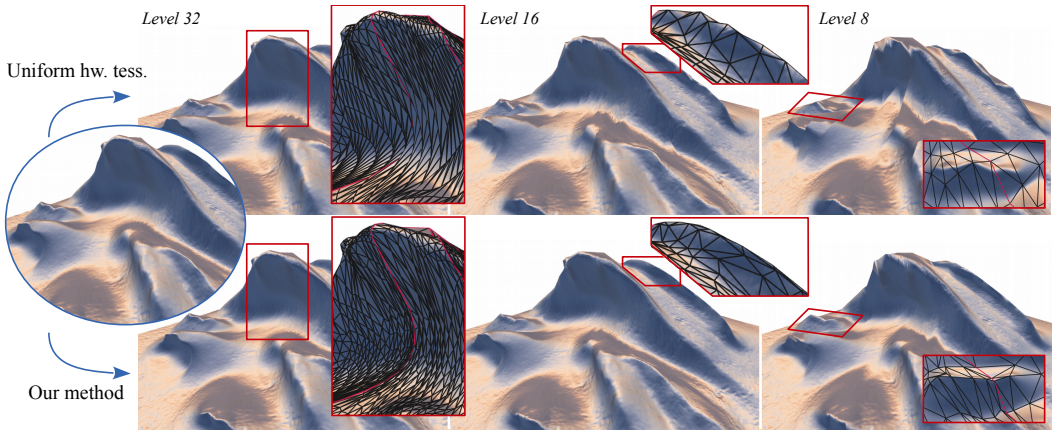


Figure 4.7 – Comparison on the "Snowdrifts" test scene (8 patches). Uniform hardware tessellation ***(top)*** fails at representing accurately sharp features and areas of high curvature, such as the top and deep part of the drifts, which produces tessellation artifacts. Our method ***(bottom)*** better preserves those regions by adapting the triangle size and aligning their edges with those features.

directions.

4.8.2 Comparison to regular hardware tessellation

The quality of our generated LODs and its superiority to both standard displacement maps and strip-collapse algorithm constrained to follow the fixed tessellation patterns have already been demonstrated in Figure 4.3. Even though the later variant slightly improves upon displacement maps by repositioning the vertices of the coarser level according to the quadratic error metric, this figure shows that most of the gains of our approach actually come from its non-uniformity in choosing optimal strips. Figure 4.7 shows a more complicated example with folds and sharp edges. Even at the factor 32, standard tessellation introduces strong oscillations in the concave region of the drifts. Those become prominently visible at factor 16, even with a normal-map-based shading. At level 16, sharp features are already significantly degraded by the uniform sampling of the displacement map, whereas our approach manages to properly align triangle edges with sharp features. As a result, our approach remains very close to the reference image, even at a factor 8 for which 98.5% of the vertices have been removed.

Figure 4.8 further shows that these well-represented features remain very stable during the continuous transition between LODs. It also illustrates that, when transitioning between two subsequent levels with power-of-two tessellation, we obtain a very different behavior than standard fractional tessellation as many more strips split or collapse simultaneously.

Figure 4.9 provides comparisons on larger scale models for equivalent tessellation factors. In both cases, triangle edges are better aligned with features

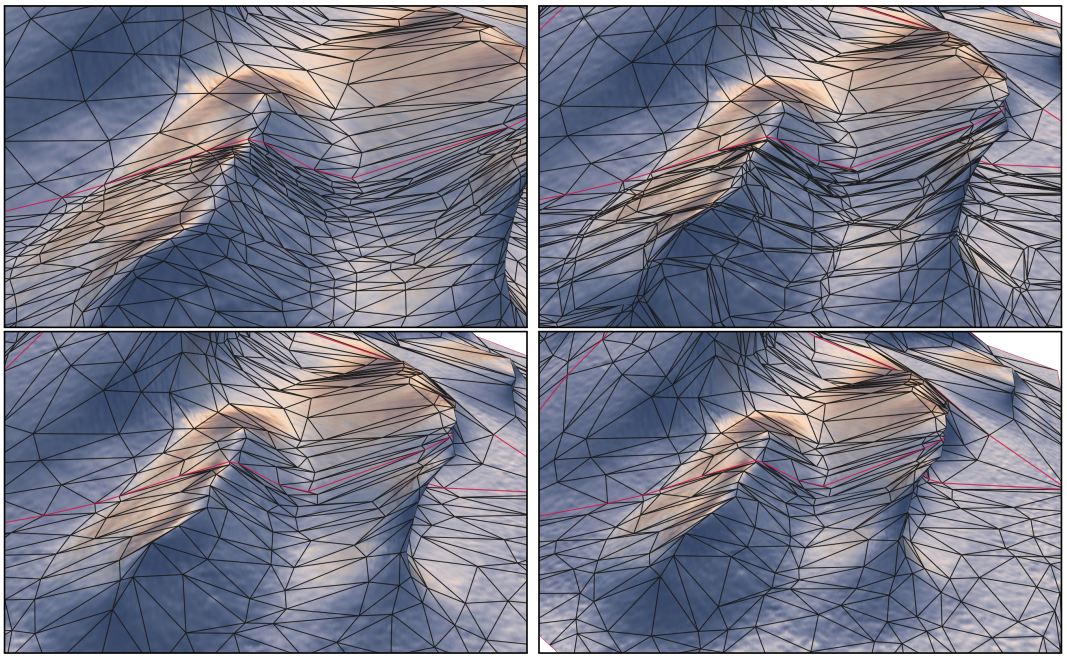


Figure 4.8 – Zoom-out sequence generated by our controllable fractional tessellation. Observe how the sharp features are well preserved during the continuous simplification. The bottom patch transitions from integer factors 32 to 16, whereas the top patch stays at an integer factor 16 but nearly approaches the next power-of-two factor 8 at the last frame.

thus avoiding distorted sharp edges and shading artifacts produced by invalid geometry.

4.8.3 Quantitative Evaluation

The impact of our refitting step, vertex placement strategy and selection strategy are evaluated in Figure 4.10 using the integral of our error metric E over the mesh as an objective comparison criterion. For this experiment, we compare our new placement strategy (yellow curves), the QEM-based (purple curves) and LTM-based (green curves) vertex placement strategies relatively to the LOD produced by the classic hardware tessellation pipeline (i.e., by sampling mipmapped displacement maps). Since we require the interpolation of texture coordinates, we use a QEM variant that constraints the vertex placement on the collapsed-edge with linear texture coordinates interpolation, and used a simple Shepard interpolation [Shepard, 1968] of the neighboring vertex texture coordinates for the LTM. To ensure a fair comparison, all the methods start with the same initial tessellated mesh (level 0) obtained by sampling an intermediate displacement map. This explains why, without refitting, they all exhibit the exact same error for the highest tessellation factor.

It can be seen that our new vertex placement strategy alone, i.e., with-

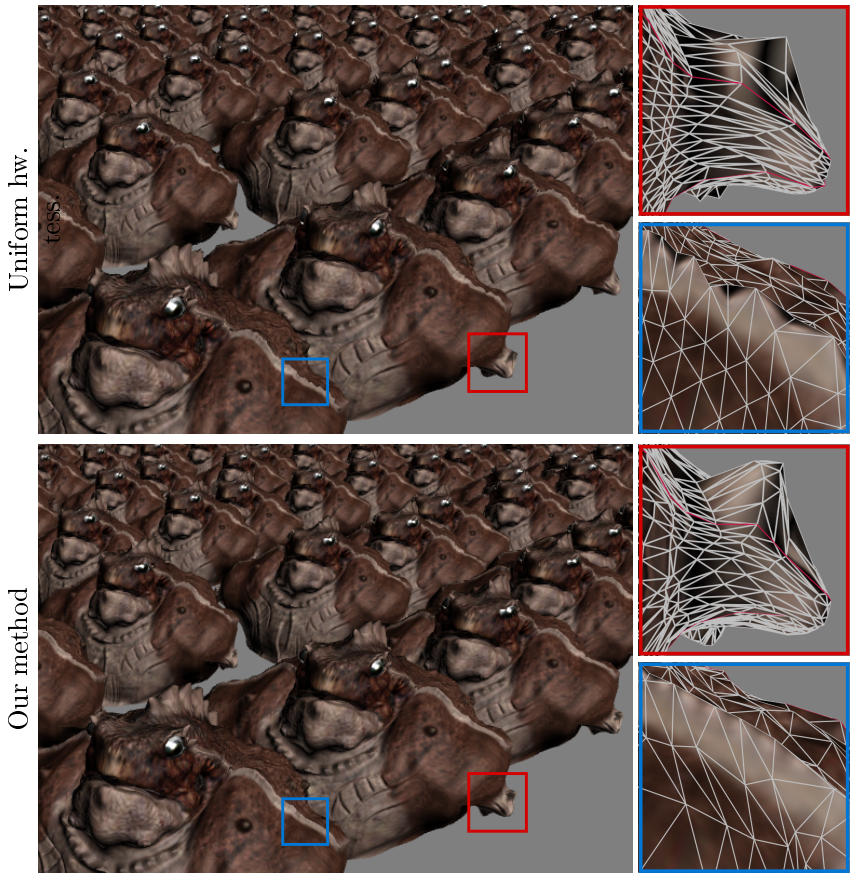


Figure 4.9 – "Toad army" – **Left:** Image and **Right:** close-up views (same tessellation but different viewpoint). Uniform hardware tessellation ***(top)*** degrades or misses major geometric features of the displacement map, such as the shoulder crest and the sculpted stump, whereas our LOD mechanism ***(bottom)*** represents and simplifies them faithfully.

out the previously described global refitting step, already outperforms both the QEM and LTM strategies. As expected, for all placement strategies, the refitting step has a huge impact on the quality of the produced LOD. The higher accuracy of the texture coordinates computed with our method combined with the refitting step leads us to substantial improvements over the classic hardware tessellation. Our cost strategy, applied in conjunction with the two previous extensions, hence called *Full*, allows to further reduce the view-dependent error by up to 60% compared to the regular hardware tessellation.

4.8.4 View-Dependent Metric Performance

Figure 4.11 compares the performance of our selection metric $\tilde{E}$ with the isotropic version E_{iso} on the four test scenes of Figure 4.6 for three LOD gen-

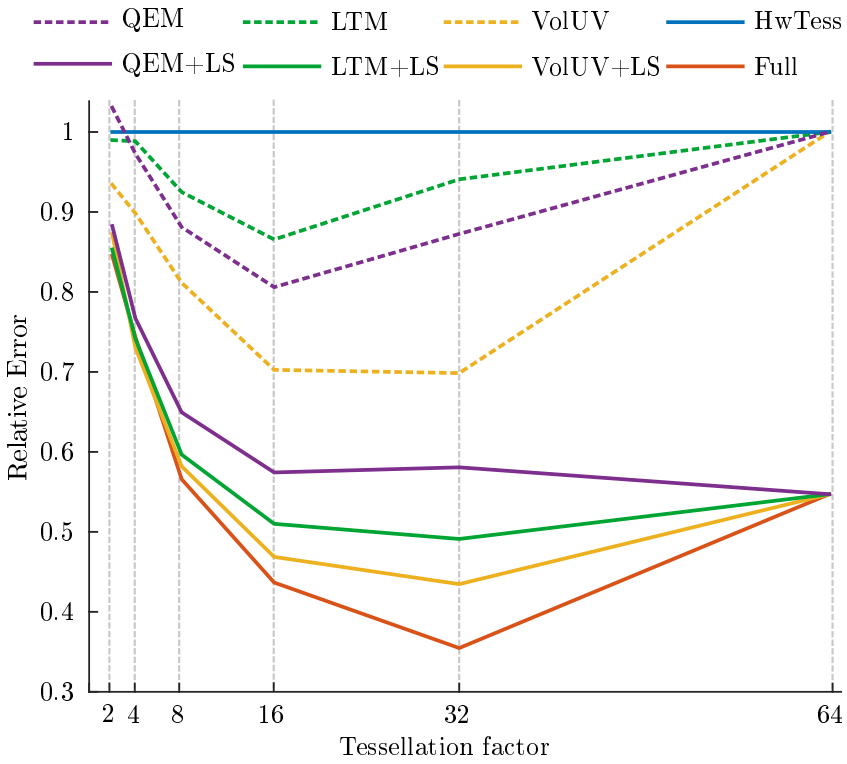


Figure 4.10 – Simplification algorithms comparison – the error is expressed relatively to the mesh produced by the regular hardware tessellation (HwTess) and averaged over all patches of the four 3D models shown in Figure 4.6. We compare the QEM- and LTM-based vertex placement strategies with our new Volumetric UV optimization with and without least-square refitting, and eventually its *full* combination with our novel edge selection cost.

eration approaches: the regular hardware tessellation with mip-mapped displacement maps, a QEM-based simplification method, and our full algorithm. On all test scenes, instantiated 400 times on a uniform 20×20 2D grid with random orientations, for a given LOD generation method, our view-dependent selection metric reduces the number of polygons generated by the tessellator by an average of 10% to 35% at equal visual quality. When comparing the LOD generation techniques with each others, our method reduces drastically the number of required triangles compared to previous work, accelerating the rendering time by a factor ranging from 2 to 3 on average.

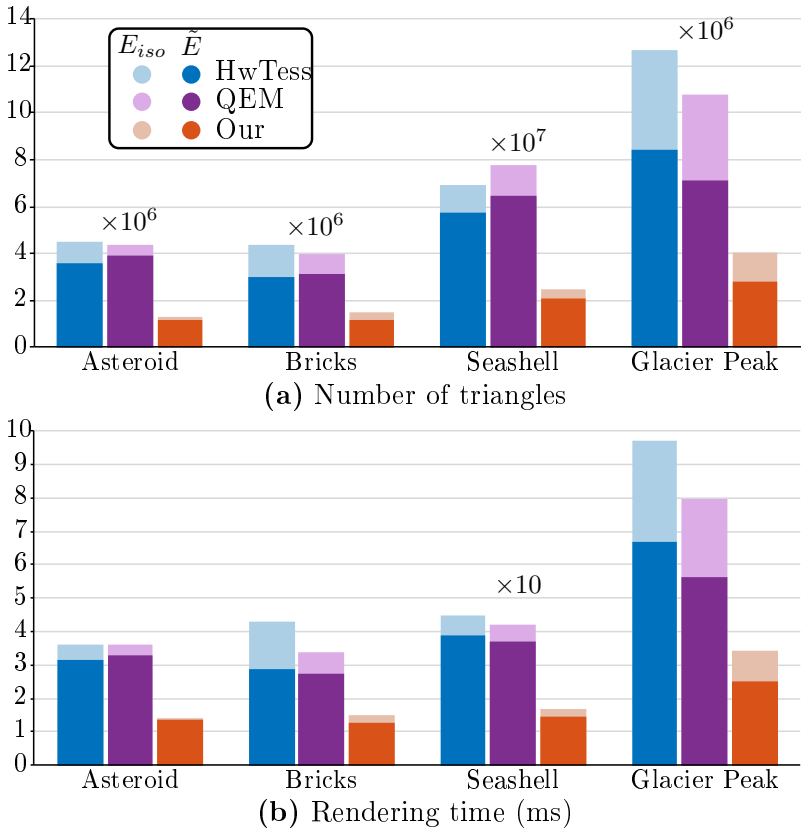


Figure 4.11 – Rendering performance comparison – for a given viewpoint, we measure the number of triangles **(a)** after level selection using our view-dependent metric $\tilde{E}$ and the isotropic version E_{iso}, as well as the associated rendering time **(b)**. We compare these indicators for three LOD generation methods on the four test meshes of Figure 4.6 instantiated 400 times on a 2D grid with random orientations.

Conclusion

Summary

In this thesis, we proposed a general framework for the generation and rendering of non-uniform LODs compatible with the hardware tessellation engine and made the following contributions.

First, we introduced a patch-based view-dependent metric capturing both geometric and parametric distortions. This metric can serve multiple purposes. First, it can directly be used to compare simplifications algorithms in an objective manner. More importantly, we showed how to take insights from this metric to design better simplifications algorithms. Finally, we demonstrated how to accurately and compactly summarize this metric over a given patch to drive the selection of appropriate LOD at rendering time. It provides significant gains compared to an isotropic metric.

Second, we presented a novel hierarchical representation of multi-resolution meshes for hardware tessellation that allows us to finely control the topological locations of vertex splits and merges, relaxing the regularity of fractional tessellation, while retaining the efficiency of the respective GPU's units. To exploit the flexibility introduced by our novel representation, we presented a dedicated mesh decimation scheme that matches the hardware tessellation pattern at successive levels.

Last, we built upon the insights gained by our metric to revisit iterative simplification algorithms. In particular, we proposed the first vertex placement strategy that optimizes for both geometric and parametric distortions in a scale independent manner. We then showed that performing global optimization steps on the whole simplified mesh can greatly improve the quality of the LOD with little additional pre-computation time. Last but not least, we also described how to use our error metric to schedule the local simplification operations to minimise even further the overall simplification error.

Our framework enables for the first time the rendering of feature-aware LOD with hardware tessellation, leading to huge improvements both in terms of triangle count and rendering time in comparison to alternative methods. This framework opens the door to many opportunities of improvements.

For example, both our new vertex-placement strategy and refitting procedures could be integrated within classical edge-collapse simplification algorithms. The integration of the former is straightforward, and refitting passes could be triggered on a regular basis during the simplification. If this combination is as successful as in our hardware tessellation tessellation context, this would lead to a highly competitive tool for the decimation of parametrized meshes.

Perspectives

We present in this section four directions for future work that could overcome some of the limitations of our framework or leads to additional performance gains.

Animation In Section 1.4.3.2, we observed that vector displacement mapping tends to produce more artifacts after deforming the base mesh than scalar displacements. Since our method is likely to create large tangential displacements to reposition vertices on the mesh features, it suffers from the same limitation. It could be mitigated using some form of indirect scalar representation. Rather than storing a vector for each vertex, we could store, with the same memory cost, the two new barycentric coordinates and a scalar displacement along the normal. However, converting an arbitrary vector in this representation is not a trivial task. First, such a conversion is neither guaranteed to exist nor to be unique. Second, the conversion can produce barycentric coordinates of vertices outside the patch, leading to even stronger artifacts when the base surface is animated.

Our view-dependent error E changes through the animation. It has been designed with rigidly deformed objects in mind. For reasonably small deformations, transforming our view-depend metric by a local rigid approximation on a patch-basis is likely to behave well. Handling large deformations, however, remains an open problem. It is difficult to apply directly the deformation induced by the animation to our approximation $\tilde{E}$. Indeed, the deformation is generally not constant over a patch and all the difference vectors are affected differently. It is thus impossible to precisely know how our error $\tilde{E}$ will be affected by the deformation. A possible approach would be to compute several errors $\tilde{E}$ for different key-frames and interpolate between them during the animation.

Input In order to have a full automatic pipeline capable of generating LOD compatible with hardware tessellation from any input mesh with a parametrization, we need to decompose the input mesh into a set of patches. This can easily be done by any simplification algorithm that maintains a good parametrization

through the simplification. For example, our VolUV method in combination with our refitting process can be used. However, the stake is to create patches of similar complexity [Yuan et al., 2016]. In addition, to minimize our metric, it would be interesting to construct patches where details can be represented by relatively collinear displacements.

Once the decomposition into patches is obtained, the input mesh for our decimation algorithm is generated by uniformly sampling for each patch the parametric domain according to the finest tessellation pattern. The first refitting step takes care of computing the corresponding 3D positions. For the finest level, the quality gain of our method in comparison to the regular tessellation is thus only due to our refitting step. Therefore, better algorithms still need to be investigated to generate the initial finer level. We believe that a better redistribution of the samples in the parametric space would increase even more the quality of the finest level. As the finest level is the level with the large number of triangles, it could at run-time reduce greatly the number of triangles and provide significant performance gains. On top of that, as we use a decimation algorithm to generate the subsequent levels, increasing the quality of the finest level will impact the quality of the coarser ones, thus leading to potentially even more performance gains.

Tessellation Pattern The hardware tessellation pattern has been designed with uniform sampling and regular-shaped triangles in mind. It is thus well suited for representing subdivision surfaces and displacement mapping. However, in the context of feature-aware LOD, we had to devise a dedicated mesh decimation scheme based on the contraction of strips to match the hardware tessellation pattern at successive levels. Although it provides impressive results, it is still much less flexible than standard edge-collapse algorithms.

Therefore, it would be interesting to investigate alternative tessellation patterns more adapted to the rendering of feature-aware LODs. The idea would be to design a pattern allowing more flexibility during the transitions. In addition to retaining the continuous spatial and temporal transitions between levels, such a pattern needs to have the following properties:

- We have to be able to efficiently enumerate all the possible sets of contractions to pass from one level to another.
- This set of contractions has to be as diverse as possible to enable a finer control of topological changes.

This would allow to exploit all the flexibility introduced by our representation, and thus to provide finer-grain LOD compatible with hardware tessellation.

Geometry filtering As the tessellation factor decreased, geometric details are removed from the surface. To keep the contribution of those details in

the final pixel values, we need to use a normal map. This is, however, not sufficient to achieve accurate shading. Indeed, simplifying the geometry also affects shadows (as discussed in Section 2.6) and self-occlusions. The latter is problematic since parts of the model which normally would have been hidden, will contribute to the value of some pixels. To achieve an accurate shading, it is thus necessary to integrate the removed details contribution into the reflectance properties of the surface. Several approaches [Olano and Baker, 2010; Dupuy et al., 2013] have already studied this issues at a micro-scale, however, at a macro-scale geometry filtering remains an open problem.

Appendix A

Optimal x-strip algorithm

Algorithm 5 Find optimal x-strip

Input: Mesh $\mathcal{M}$, Patch corner c
$\mathcal{B} \leftarrow$ Find a half-patch boundary of c ▷ *An open one if it exists*
for each halfedge h_b on $\mathcal{B}$ **do**
    Insert h_b in P ▷ P *is the list of remaining vertex to process*
    for each halfedge h_p in adjacent patch of c **do** ▷ *Initilization*
        $\mathrm{D}[h_b, h_p] \leftarrow 0$ ▷ *Distance from h_b to h_p*
        $\mathrm{prev}[h_b, h_p] \leftarrow$ undefined ▷ *Previous halfedge in optimal path*
    end for
end for
while P not empty **do**
    $h \leftarrow$ Pop a halfedge of P
    if h is on open boundary **then**
        N $\leftarrow$ halfedge of $\mathcal{B}$ ▷ *Connects manually the two open boundaries*
    else
        N $\leftarrow$ following halfedge of h ▷ *As explained in Section 4.3*
    end if
    for each vertex n of N **do**
        if $\mathrm{prev}[any, n] =$ undefined **then** ▷ *Has this node been visited?*
            Insert n in P
            for each halfedge h_b on $\mathcal{B}$ **do**
                $\mathrm{D}[h_b, n] \leftarrow \mathrm{D}[h_b, h]+$ cost of h
                $\mathrm{prev}[h_b, n] \leftarrow h$
            end for
        else
            for each halfedge h_b on $\mathcal{B}$ **do**
                if $\mathrm{D}[h_b, n] > \mathrm{D}[h_b, h]+$ cost of h **then**
                $\mathrm{D}[h_b, n] \leftarrow \mathrm{D}[h_b, h]+$ cost of h
                $\mathrm{prev}[h_b, n] \leftarrow h$
                end if
            end for
        end if
    end for
end while
cost $\leftarrow \infty$
ring $\leftarrow$ undefined
for each halfedge h_b on $\mathcal{B}$ **do** ▷ *Extracts optimal ring*
    if $\mathrm{D}[h_b, h_b] <$ cost **then**
        cost $\leftarrow \mathrm{D}[h_b, h_b]$
        ring $\leftarrow$ Get ring of h_b ▷ *Using* prev *information*
    end if
end for
return ring

Bibliography

Anthony A. Apodaca and Larry Gritz. 1999. *Advanced RenderMan: Creating CGI for Motion Picture* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

James F. Blinn. 1978. Simulation of Wrinkled Surfaces. *SIGGRAPH Comput. Graph.* 12, 3 (Aug. 1978), 286–292. https://doi.org/10.1145/965139.507101

David Blythe. 2006. The Direct3D 10 System. *ACM Trans. Graph.* 25, 3 (July 2006), 724–734. https://doi.org/10.1145/1141911.1141947

Jeffrey Bolz and Peter Schröder. 2002. Rapid Evaluation of Catmull-Clark Subdivision Surfaces. In *Proceedings of the Seventh International Conference on 3D Web Technology (Web3D '02)*. ACM, New York, NY, USA, 11–17. https://doi.org/10.1145/504502.504505

Jeff Bolz and Peter Schröder. 2003. Evaluation of subdivision surfaces on programmable graphics hardware. (2003).

Mario Botsch, Leif Kobbelt, Mark Pauly, Pierre Alliez, and Bruno Lévy. 2010. *Polygon Mesh Processing*. A K Peters/CRC Press.

Tamy Boubekeur and Christophe Schlick. 2005. Generic Mesh Refinement on GPU. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware (HWWS '05)*. ACM, New York, NY, USA, 99–104. https://doi.org/10.1145/1071866.1071882

Tamy Boubekeur and Christophe Schlick. 2008. A Flexible Kernel for Adaptive Mesh Refinement on GPU. *Computer Graphics Forum* 27, 1 (2008), 102–114. https://doi.org/10.1111/j.1467-8659.2007.01040.x

Lévy Bruno and Nicolas Bonneel. 2012. Variational Anisotropic Surface Meshing with Voronoi Parallel Linear Enumeration. In *IMR - 21st International Meshing Roundtable - 2012*. San José, United States. https://hal.inria.fr/hal-00804558

Michael Bunnell. 2005. Adaptive Tessellation of Subdivision Surfaces with Displacement Mapping. In *GPU Gems 2*. 109–122.

Brent Burley and Dylan Lacewell. 2008. Ptex: Per-Face Texture Mapping for Production Rendering. 27 (06 2008), 1155–1164.

Iain Cantlay. 2001. *DirectX 11 Terrain Tessellation*. Technical Report. https://developer.nvidia.com/sites/default/files/akamai/gamedev/files/sdk/11/terraintessellation_whitepaper.pdf

E. Catmull and J. Clark. 1998. Seminal Graphics. ACM, New York, NY, USA, Chapter Recursively Generated B-spline Surfaces on Arbitrary Topological Meshes, 183–188. https://doi.org/10.1145/280811.280992

Paolo Cignoni, Claudio Rocchini, and Roberto Scopigno. 1996. *Metro: Measuring Error on Simplified Surfaces*. Technical Report. Paris, France, France.

Jonathan Cohen, Marc Olano, and Dinesh Manocha. 1998. Appearance-preserving Simplification. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '98)*. ACM, 115–122.

Robert L. Cook. 1984. Shade Trees. *SIGGRAPH Comput. Graph.* 18, 3 (Jan. 1984), 223–231. https://doi.org/10.1145/964965.808602

Robert L. Cook, Loren Carpenter, and Edwin Catmull. 1987. The Reyes Image Rendering Architecture. *SIGGRAPH Comput. Graph.* 21, 4 (Aug. 1987), 95–102. https://doi.org/10.1145/37402.37414

T. Cormen, C. Leiserson, , and R. Rivest. 1990. *Introduction to Algorithms*. McGraw-Hill.

Franklin C. Crow. 1977. Shadow Algorithms for Computer Graphics. *SIGGRAPH Comput. Graph.* 11, 2 (July 1977), 242–248. https://doi.org/10.1145/965141.563901

Tony DeRose, Michael Kass, and Tien Truong. 1998. Subdivision Surfaces in Character Animation. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '98)*. ACM, New York, NY, USA, 85–94. https://doi.org/10.1145/280814.280826

Lu Dong, Yuming Fang, Weisi Lin, and Hock Soon Seah. 2014. Objective visual quality assessment for 3D meshes. In *2014 Sixth International Workshop on Quality of Multimedia Experience (QoMEX)*. 1–6. https://doi.org/10.1109/QoMEX.2014.6982277

William Donnelly. 2005. Per-pixel displacement mapping with distance functions. In *GPU Gems 2*. 123–136.

D. Doo and M. Sabin. 1998. Seminal Graphics. ACM, New York, NY, USA, Chapter Behaviour of Recursive Division Surfaces Near Extraordinary Points, 177–181. https://doi.org/10.1145/280811.280991

Reynald Dumont, Fabio Pellacini, and James A. Ferwerda. 2003. Perceptually-driven Decision Theory for Interactive Realistic Rendering. *ACM Trans. Graph.* 22, 2 (April 2003), 152–181. https://doi.org/10.1145/636886.636888

Jonathan Dupuy, Eric Heitz, Jean-Claude Iehl, Pierre Poulin, Fabrice Neyret, and Victor Ostromoukhov. 2013. Linear Efficient Antialiased Displacement and Reflectance Mapping. *ACM Transactions on Graphics* 32, 6 (Nov. 2013), Article No. 211. https://doi.org/10.1145/2508363.2508422

H. Edelsbrunner, D. Kirkpatrick, and R. Seidel. 1983. On the shape of a set of points in the plane. *IEEE Transactions on Information Theory* 29, 4 (July 1983), 551–559. https://doi.org/10.1109/TIT.1983.1056714

Herbert Edelsbrunner and Ernst P. Mücke. 1994. Three-dimensional Alpha Shapes. *ACM Trans. Graph.* 13, 1 (Jan. 1994), 43–72. https://doi.org/10.1145/174462.156635

Jihad El-Sana and Amitabh Varshney. 1998. Topology Simplification for Polygonal Virtual Environments. *IEEE Transactions on Visualization and Computer Graphics* 4, 2 (April 1998), 133–144. https://doi.org/10.1109/2945.694955

Jihad El-Sana and Amitabh Varshney. 1999. Generalized View-Dependent Simplification. *Computer Graphics Forum* 18, 3 (1999), 83–94. https://doi.org/10.1111/1467-8659.00330

David R. Forsey and Richard H. Bartels. 1988. Hierarchical B-spline Refinement. *SIGGRAPH Comput. Graph.* 22, 4 (June 1988), 205–212. https://doi.org/10.1145/378456.378512

Michael Garland and Paul S. Heckbert. 1997. Surface Simplification Using Quadric Error Metrics. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '97)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 209–216. https://doi.org/10.1145/258734.258849

Michael Garland and Paul S. Heckbert. 1998. Simplifying Surfaces with Color and Texture Using Quadric Error Metrics. In *Proceedings of the Conference*

on Visualization '98 (VIS '98). IEEE Computer Society Press, Los Alamitos, CA, USA, 263–269. http://dl.acm.org/citation.cfm?id=288216.288280

T. S. Gieng, B. Hamann, K. I. Joy, G. L. Schussman, and I. J. Trotts. 1997. Smooth hierarchical surface triangulations. In *Visualization '97., Proceedings*. 379–386. https://doi.org/10.1109/VISUAL.1997.663906

Gaël Guennebaud, Benoît Jacob, et al. 2016. Eigen v3. http://eigen.tuxfamily.org. (2016).

Bernd Hamann. 1994. A data reduction scheme for triangulated surfaces. *Computer Aided Geometric Design* 11, 2 (1994), 197–214. https://doi.org/10.1016/0167-8396(94)90032-9

Taosong He, Lichan Hong, A. Varshney, and S. W. Wang. 1996. Controlled topology simplification. *IEEE Transactions on Visualization and Computer Graphics* 2, 2 (Jun 1996), 171–184. https://doi.org/10.1109/2945.506228

Hugues Hoppe. 1996. Progressive Meshes. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '96)*. ACM, New York, NY, USA, 99–108. https://doi.org/10.1145/237170.237216

Hugues Hoppe. 1997. View-dependent Refinement of Progressive Meshes. In *Proceedings of ACM SIGGRAPH '97*. 189–198.

Hugues Hoppe. 1999. New Quadric Metric for Simplifiying Meshes with Appearance Attributes. In *Proceedings of the Conference on Visualization '99: Celebrating Ten Years (VIS '99)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 59–66. http://dl.acm.org/citation.cfm?id=319351.319357

Hugues Hoppe, Tony DeRose, Tom Duchamp, Mark Halstead, Hubert Jin, John McDonald, Jean Schweitzer, and Werner Stuetzle. 1994. Piecewise Smooth Surface Reconstruction. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '94)*. ACM, New York, NY, USA, 295–302. https://doi.org/10.1145/192161.192233

Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. 1993. Mesh Optimization. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '93)*. ACM, New York, NY, USA, 19–26. https://doi.org/10.1145/166117.166119

Kaimo Hu, Dong-Ming Yan, David Bommes, Pierre Alliez, and Bedrich Benes. 2017. Error-Bounded and Feature Preserving Surface Remeshing with Minimal Angle Improvement. *IEEE Transactions on Visualization and Computer Graphics* (Jan. 2017). https://doi.org/10.1109/TVCG.2016.2632720

Hanyoung Jang and JungHyun Han. 2012. Feature-Preserving Displacement Mapping With Graphics Processing Unit (GPU) Tessellation. *Comput. Graph. Forum* 31, 6 (Sept. 2012), 1880–1894. https://doi.org/10.1111/j.1467-8659.2012.03068.x

Hanyoung Jang and JungHyun Han. 2013. GPU-optimized indirect scalar displacement mapping. *Computer-Aided Design* 45, 2 (feb 2013), 517–522. https://doi.org/10.1016/j.cad.2012.10.034

Tomomichi Kaneko, Toshiyuki Takahei, Masahiko Inami, Naoki Kawakami, Yasuyuki Yanagida, Taro Maeda, and Susumu Tachi. 2001. Detailed shape representation with parallax mapping. In *In Proceedings of the ICAT 2001*. 205–208.

Hyeong Yeop Kang, Hanyoung Jang, Chang Sik Cho, and Jung Hyun Han. 2015. Multi-resolution terrain rendering with GPU tessellation. *Visual Computer* 31, 4 (2015), 455–469. https://doi.org/10.1007/s00371-014-0941-6

Ken-ichi Kawarabayashi, Yusuke Kobayashi, and Bruce Reed. 2012. The disjoint paths problem in quadratic time. *Journal of Combinatorial Theory, Series B* 102, 2 (2012), 424–435.

Benjamin Keinert, Matthias Innmann, Michael Sänger, and Marc Stamminger. 2015. Spherical Fibonacci Mapping. *ACM Transactions on Graphics* 34, 6 (2015), 193:1–193:7.

Leonid G. Khachiyan. 1996. Rounding of Polytopes in the Real Number Model of Computation. *Math. Oper. Res.* 21, 2 (May 1996), 307–320. https://doi.org/10.1287/moor.21.2.307

R. Klein, G. Liebich, and W. Strasser. 1996. Mesh reduction with error control. In *Visualization '96. Proceedings.* 311–318. https://doi.org/10.1109/VISUAL.1996.568124

Leif Kobbelt. 2000. $\sqrt{3}$ subdivisionn. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '00)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 103–112. https://doi.org/10.1145/344779.344835

Leif P. Kobbelt, Jens Vorsatz, and Ulf and Labsik. 1999. A Shrink Wrapping Approach to Remeshing Polygonal Surfaces. *Computer Graphics Forum* 18, 3 (1999), 119–130.

Andreas Kolb and Christof Rezk-Salama. Efficient Empty Space Skipping for Per-Pixel Displacement Mapping.

Denis Kovacs, Jason Mitchell, Shanon Drone, and Denis Zorin. 2009. Real-time Creased Approximate Subdivision Surfaces. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games (I3D '09)*. ACM, New York, NY, USA, 155–160. https://doi.org/10.1145/1507149.1507174

Venkat Krishnamurthy and Marc Levoy. 1996. Fitting Smooth Surfaces to Dense Polygon Meshes. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '96)*. ACM, New York, NY, USA, 313–324. https://doi.org/10.1145/237170.237270

Thibaud Lambert, Pierre Bénard, and Gaël Guennebaud. 2016. Multi-Resolution Meshes for Feature-Aware Hardware Tessellation. *Computer Graphics Forum* 35, 2 (2016), 253–262.

Guillaume Lavoué. 2011. A Multiscale Metric for 3D Mesh Visual Quality Assessment. *Computer Graphics Forum (Proceedings of Eurographics Symposium on Geometry Processing 2011)* 30 (July 2011), 1427–1437. https://hal.archives-ouvertes.fr/hal-01354459

Aaron Lee, Henry Moreton, and Hugues Hoppe. 2000. Displaced Subdivision Surfaces. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '00)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 85–94. https://doi.org/10.1145/344779.344829

Chang Ha Lee, Amitabh Varshney, and David W. Jacobs. 2005. Mesh Saliency. *ACM Trans. Graph.* 24, 3 (July 2005), 659–666. https://doi.org/10.1145/1073204.1073244

Liang Hu, Pedro V Sander, and Hugues Hoppe. 2010. Parallel View-Dependent Level-of-Detail Control. *IEEE Transactions on Visualization and Computer Graphics* 16, 5 (2010), 718–728.

Peter Lindstrom. 2000. *Model Simplification Using Image and Geometry-based Metrics*. Ph.D. Dissertation. Atlanta, GA, USA. Advisor(s) Turk, Greg. AAI9994428.

Peter Lindstrom and Greg Turk. 1998. Fast and Memory Efficient Polygonal Simplification. In *Proceedings of the Conference on Visualization '98 (VIS '98)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 279–286. http://dl.acm.org/citation.cfm?id=288216.288288

Peter Lindstrom and Greg Turk. 2000. Image-driven Simplification. *ACM Trans. Graph.* 19, 3 (July 2000), 204–241. https://doi.org/10.1145/353981.353995

Charles Loop. 1987. *Smooth Subdivision Surfaces Based on Triangles*. Ph.D. Dissertation. https://www.microsoft.com/en-us/research/publication/smooth-subdivision-surfaces-based-on-triangles/

Charles Loop and Scott Schaefer. 2008. Approximating Catmull-Clark Subdivision Surfaces with Bicubic Patches. *ACM Trans. Graph.* 27, 1, Article 8 (March 2008), 11 pages. https://doi.org/10.1145/1330511.1330519

Kok-Lim Low and Tiow-Seng Tan. 1997. Model Simplification Using Vertex-clustering. In *Proceedings of the 1997 Symposium on Interactive 3D Graphics (I3D '97)*. ACM, New York, NY, USA, 75–ff. https://doi.org/10.1145/253284.253310

David Luebke and Carl Erikson. 1997. View-dependent Simplification of Arbitrary Polygonal Environments. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '97)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 199–208. https://doi.org/10.1145/258734.258847

David Luebke and Benjamin Hallen. 2001. Perceptually Driven Simplification for Interactive Rendering. In *Proceedings of the 12th Eurographics Conference on Rendering (EGWR'01)*. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 223–234. https://doi.org/10.2312/EGWR/EGWR01/223-234

David Luebke, Benjamin Watson, Jonathan D. Cohen, Martin Reddy, and Amitabh Varshney. 2002. *Level of Detail for 3D Graphics*. Elsevier Science Inc.

Morgan McGuire and Max McGuire. 2005. Steep Parallax Mapping. *I3D 2005 Poster* (2005). http://www.cs.brown.edu/research/graphics/games/SteepParallax/index.html

Nicolas Menzel and Michael Guthe. 2010. Towards Perceptual Simplification of Models with Arbitrary Materials. *Computer Graphics Forum* 29, 7 (2010), 2261–2270. https://doi.org/10.1111/j.1467-8659.2010.01815.x

MICROSOFT. 2009. Direct3D 11 Features. (2009).

Henry Moreton. 2001. Watertight Tessellation Using Forward Differencing. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware (HWWS '01)*. ACM, New York, NY, USA, 25–32. https://doi.org/10.1145/383507.383520

NIMA MOSHTAGH. 2005. MINIMUM VOLUME ENCLOSING ELLIPSOIDS. (2005). https://doi.org/10.1.1.116.7691

Ashish Myles, Young In Yeo, and Jörg Peters. 2008. GPU Conversion of Quad Meshes to Smooth Surfaces. In *Proceedings of the 2008 ACM Symposium on Solid and Physical Modeling (SPM '08)*. ACM, New York, NY, USA, 321–326. https://doi.org/10.1145/1364901.1364946

G. Nader, K. Wang, F. Hetroy-Wheeler, and F. Dupont. 2015. Just Noticeable Distortion Profile for Flat-Shaded 3D Mesh Surfaces. *IEEE Transactions on Visualization and Computer Graphics* PP, 99 (2015), 1–1. https://doi.org/10.1109/TVCG.2015.2507578

Georges Nader, Kai Wang, Franck Hétroy-Wheeler, and Florent Dupont. 2016. Visual Contrast Sensitivity and Discrimination for 3D Meshes and their Applications. *Computer Graphics Forum* 35, 7 (Oct. 2016), 497–506. https://doi.org/10.1111/cgf.13046

Ahmad H. Nasri. 1987. Polyhedral Subdivision Methods for Free-form Surfaces. *ACM Trans. Graph.* 6, 1 (Jan. 1987), 29–73. https://doi.org/10.1145/27625.27628

T. Ni, Y. Yeo, A. Myles, V. Goel, and J. Peters. 2008. GPU smoothing of quad meshes. In *2008 IEEE International Conference on Shape Modeling and Applications*. 3–9. https://doi.org/10.1109/SMI.2008.4547938

M. Nießner, B. Keinert, M. Fisher, M. Stamminger, C. Loop, and H. Schäfer. 2016. Real-Time Rendering Techniques with Hardware Tessellation. *Comput. Graph. Forum* 35, 1 (Feb. 2016), 113–137. https://doi.org/10.1111/cgf.12714

Matthias Nießner and Charles Loop. 2013. Analytic Displacement Mapping Using Hardware Tessellation. *ACM Trans. Graph.* 32, 3, Article 26 (July 2013), 9 pages. https://doi.org/10.1145/2487228.2487234

Vincent Nivoliers, Dong-Ming Yan, and Bruno Lévy. 2014. Fitting polynomial surfaces to triangular meshes with Voronoi squared distance minimization. *Engineering with Computers* 30, 3 (2014), 289–300.

F. S. Nooruddin and G. Turk. 2003. Simplification and repair of polygonal models using volumetric techniques. *IEEE Transactions on Visualization and Computer Graphics* 9, 2 (April 2003), 191–205. https://doi.org/10.1109/TVCG.2003.1196006

Kyoungsu Oh, Hyunwoo Ki, and Cheol-Hi Lee. 2006. Pyramidal displacement mapping: A GPU based artifacts-free ray tracing through an image pyramid. (01 2006), 75-82 pages.

Marc Olano and Dan Baker. 2010. LEAN Mapping. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D '10)*. ACM, New York, NY, USA, 181–188. https://doi.org/10.1145/1730804.1730834

Manuel M. Oliveira, Gary Bishop, and David McAllister. 2000. Relief Texture Mapping. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '00)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 359–368. https://doi.org/10.1145/344779.344947

Anjul Patney, Mohamed S. Ebeida, and John D. Owens. 2009. Parallel View-dependent Tessellation of Catmull-Clark Subdivision Surfaces. In *Proceedings of the Conference on High Performance Graphics 2009 (HPG '09)*. ACM, New York, NY, USA, 99–108. https://doi.org/10.1145/1572769.1572785

Fábio Policarpo, Manuel M. Oliveira, and João L. D. Comba. 2005. Real-time Relief Mapping on Arbitrary Polygonal Surfaces. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games (I3D '05)*. ACM, New York, NY, USA, 155–162. https://doi.org/10.1145/1053427.1053453

Jovan Popović and Hugues Hoppe. 1997. Progressive Simplicial Complexes. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '97)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 217–224. https://doi.org/10.1145/258734.258852

Mátyás Premecz. 2006. Iterative parallax mapping with slope information. In *In Central European Seminar on Computer Graphics*. 2006.

Lijun Qu and Gary W. Meyer. 2008. Perceptually Guided Polygon Reduction. *IEEE Transactions on Visualization and Computer Graphics* 14, 5 (Sept. 2008), 1015–1029. https://doi.org/10.1109/TVCG.2008.51

Martin Reddy. 1997. *Perceptually modulated level of detail for virtual environments*. Ph.D. Dissertation. University of Edinburgh, UK.

Rémi Ronfard and Jarek Rossignac. 1996. Full-range approximation of triangulated polyhedra. *Computer Graphics Forum* 15, 3 (1996), 67–76. https://doi.org/10.1111/1467-8659.1530067

Jarek Rossignac and Paul Borrel. 1993. Multi-resolution 3D approximations for rendering complex scenes. In *Modeling in Computer Graphics: Methods and Applications*. Springer Berlin Heidelberg, 455–465.

Michaël Roy, Sebti Foufou, and Frédéric Truchetet. 2004. Mesh comparison using attribute deviation metric," Int. *Journal of Image and Graphics* 4 (2004), 1–14.

D. Salinas, F. Lafarge, and P. Alliez. 2015. Structure-Aware Mesh Decimation. *Comput. Graph. Forum* 34, 6 (Sept. 2015), 211–227. https://doi.org/10.1111/cgf.12531

Henry Schafer, Magdalena Prus, Quirin Meyer, Jochen Submuth, and Marc Stamminger. 2013. Multiresolution Attributes for Hardware Tessellated Objects. *IEEE Transactions on Visualization and Computer Graphics* 19, 9 (Sept. 2013), 1488–1498. https://doi.org/10.1109/TVCG.2013.44

William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. 1992. Decimation of Triangle Meshes. *SIGGRAPH Comput. Graph.* 26, 2 (July 1992), 65–70. https://doi.org/10.1145/142920.134010

Mark Segal and Kurt Akeley. 2010. The OpenGL Graphics System : A Specification (Version 4.0 (Core Profile) - March 11, 2010). (2010).

Donald Shepard. 1968. A Two-dimensional Interpolation Function for Irregularly-spaced Data. In *Proceedings of the 1968 23rd ACM National Conference (ACM '68)*. ACM, 517–524.

Le-Jeng Shiue, Ian Jones, and Jörg Peters. 2005. A Realtime GPU Subdivision Kernel. *ACM Trans. Graph.* 24, 3 (July 2005), 1010–1015. https://doi.org/10.1145/1073204.1073304

Ran Song, Yonghuai Liu, Ralph R. Martin, and Paul L. Rosin. 2014. Mesh Saliency via Spectral Processing. *ACM Trans. Graph.* 33, 1, Article 6 (Feb. 2014), 17 pages. https://doi.org/10.1145/2530691

Marc Soucy and Denis Laurendeau. 1996. Multiresolution Surface Modeling Based on Hierarchical Triangulation. *Computer Vision and Image Understanding* 63, 1 (1996), 1–14. https://doi.org/10.1006/cviu.1996.0001

Jos Stam. 1998. Exact Evaluation of Catmull-Clark Subdivision Surfaces at Arbitrary Parameter Values. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '98)*.

ACM, New York, NY, USA, 395–404. https://doi.org/10.1145/280814.280945

L. Szirmay-Kalos and T. Umenhoffer. 2006. Displacement Mapping on the GPU - State of the Art. *Computer Graphics Forum* 27, 1 (2006).

Natalya Tatarchuk. 2005. Practical Dynamic Parallax Occlusion Mapping. In *ACM SIGGRAPH 2005 Sketches (SIGGRAPH '05)*. ACM, New York, NY, USA, Article 106. https://doi.org/10.1145/1187112.1187240

Natalya Tatarchuk. 2006. Practical Parallax Occlusion Mapping with Approximate Soft Shadows for Detailed Surface Rendering. In *ACM SIGGRAPH 2006 Courses (SIGGRAPH '06)*. ACM, New York, NY, USA, 81–112. https://doi.org/10.1145/1185657.1185830

Charles E. Thorpe. 1984. Path Relaxation: Path Planning for a Mobile Robot. In *Proceedings of the Fourth AAAI Conference on Artificial Intelligence (AAAI'84)*. AAAI Press, 318–321. http://dl.acm.org/citation.cfm?id=2886937.2886997

Fakhri Torkhani, Kai Wang, and Jean-Marc Chassery. 2014. A curvature-tensor-based perceptual quality metric for 3D triangular meshes. *Machine Graphics & Vision* (Feb. 2014), 1–25. https://hal.archives-ouvertes.fr/hal-00995716

Sébastien Valette and Jean-Marc Chassery. 2004. Approximated Centroidal Voronoi Diagrams for Uniform Polygonal Mesh Coarsening. *Computer Graphics Forum* 23, 3 (2004), 381–389. https://doi.org/10.1111/j.1467-8659.2004.00769.x Eurographics 2004 proceedings.

Alex Vlachos, Jörg Peters, Chas Boyd, and Jason L. Mitchell. 2001. Curved PN Triangles. In *Proceedings of the 2001 Symposium on Interactive 3D Graphics (I3D '01)*. ACM, New York, NY, USA, 159–166. https://doi.org/10.1145/364338.364387

Kai Wang, Fakhri Torkhani, and Annick Montanvert. 2012. A fast roughness-based approach to the assessment of 3D mesh visual quality. *Computers and Graphics* 36, 7 (Nov. 2012), 808–818. https://doi.org/10.1016/j.cag.2012.06.004

Zhou Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. 2004. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing* 13, 4 (April 2004), 600–612. https://doi.org/10.1109/TIP.2003.819861

Thomas Weber, Michael Wimmer, and John D. Owens. 2015. Parallel Reyes-style Adaptive Subdivision with Bounded Memory Usage. In *Proceedings of the 19th Symposium on Interactive 3D Graphics and Games (i3D '15)*. ACM, New York, NY, USA, 39–45. https://doi.org/10.1145/2699276.2699289

Terry Welsh and Infiscape Corporation. 2004. Parallax Mapping with Offset Limiting: A PerPixel Approximation of Uneven Surfaces. (2004).

Lance Williams. 1978. Casting Curved Shadows on Curved Surfaces. *SIGGRAPH Comput. Graph.* 12, 3 (Aug. 1978), 270–274. https://doi.org/10.1145/965139.807402

Lance Williams. 1983. Pyramidal Parametrics. *SIGGRAPH Comput. Graph.* 17, 3 (July 1983), 1–11. https://doi.org/10.1145/964967.801126

Nathaniel Williams, David Luebke, Jonathan D. Cohen, Michael Kelley, and Brenden Schubert. 2003. Perceptually Guided Simplification of Lit, Textured Meshes. In *Proceedings of the 2003 Symposium on Interactive 3D Graphics (I3D '03)*. ACM, New York, NY, USA, 113–121. https://doi.org/10.1145/641480.641503

J. C. Xia and A. Varshney. 1996. Dynamic view-dependent simplification for polygonal models. In *Visualization '96. Proceedings.* 327–334. https://doi.org/10.1109/VISUAL.1996.568126

I-Cheng Yeh, Chao-Hung Lin, Olga Sorkine, and Tong-Yee Lee. 2011. Template-based 3D Model Fitting Using Dual-Domain Relaxation. *IEEE Transactions on Visualization and Computer Graphics* 17, 8 (2011), 1178–1190.

Keith Yerex and Martin Jagersand. 2004. Displacement Mapping with Raycasting in Hardware. In *ACM SIGGRAPH 2004 Sketches (SIGGRAPH '04)*. ACM, New York, NY, USA, 149–. https://doi.org/10.1145/1186223.1186410

Yazhen Yuan, Rui Wang, Jin Huang, Yanming Jia, and Hujun Bao. 2016. Simplified and Tessellated Mesh for Realtime High Quality Rendering. *Comput. Graph.* 54, C (Feb. 2016), 135–144. https://doi.org/10.1016/j.cag.2015.07.011

Qing Zhu, Junqiao Zhao, Zhiqiang Du, and Yeting Zhang. 2010. Quantitative analysis of discrete 3D geometrical detail levels based on perceptual metric. 34 (02 2010), 55–65.

Rendu de niveaux de détails avec la Tessellation Matérielle

Thibaud Lambert

Cette thèse s'inscrit dans le cadre de la synthèse d'images 3D qui vise à générer des images de synthèse à partir de mondes virtuel à travers des algorithmes informatiques. La synthèse d'images est essentielle dans de nombreux domaines d'application tels que le divertissement, la simulation, la visualisation scientifique, la publicité, l'imagerie médicale, etc. Pour la plupart de ces applications, l'un des objectifs est de produire ce que l'on appelle des rendus photo-réalistes. Deux conditions doivent être remplies pour atteindre cet objectif. Tout d'abord, des algorithmes de rendu doivent être développés pour reproduire les lois physiques du transport de la lumière et des interactions lumière-matière avec la plus grande précision possible. Deuxièmement, les scènes numériques doivent reproduire fidèlement la complexité des environnements réels, ce qui nécessite une grande quantité de détails. Ces deux aspects sont essentiels dans l'industrie cinématographique pour les effets spéciaux où des images du monde réel sont mélangées avec des images numériques, ou dans l'architecture et l'industrie automobile pour prévisualiser et étudier des design possibles. Dans ce contexte, une quantité énorme de détails est nécessaire pour représenter des modèles 3D aussi réalistes que possible. De plus, l'apparition d'affichages avec une densité élevée de pixels motive également l'utilisation de modèles très détaillés pour exploiter tous les avantages de ces affichages.

Les algorithmes de rendu peuvent être classés en deux catégories : les algorithmes hors ligne et les algorithmes temps réel. Les premiers sont soumis à de faibles contraintes de temps; ils sont utilisés pour les films et les films d'animation. Les applications de rendu hors ligne ont donc la possibilité d'utiliser des algorithmes coûteux. Ils utilisent généralement des techniques d'illumination globales combinées à des modèles très détaillés pour obtenir des images qui ne sont presque pas différentiables des photographies. Ceci est réalisé au prix d'un long temps de calcul; le rendu d'une seule image peut prendre des heures voire des jours.

D'un autre côté, le rendu temps réel a pour but de générer des images en très peu de temps. Ceci est essentiel pour les applications interactives telles que les jeux vidéo, les simulations interactives, les applications de RV, etc. pour offrir une expérience fluide à l'utilisateur. Chaque image doit être calculée en moins de 30 ms pour atteindre un nombre d'image par seconde confortable. Les images rendues en temps réel sont donc encore faciles à différencier des images réelles. Cela est dû à l'utilisation de modèles plus grossiers et d'algorithmes de transport de la lumière approchés qui fonctionnent beaucoup plus rapidement que les algorithmes hors ligne.

Pour atteindre ces performances, tous les moteurs de rendu en temps réel reposent entièrement sur des unités de calcul graphique (GPU) dédiées. Les GPU modernes sont des processeurs parallèles massifs composés de milliers de cœurs. Ils sont spécifiquement dédiés au rendu temps réel d'une représentation spécifique de modèles 3D: le maillage triangulaire associé à des textures. Le maillage triangulaire représente la forme générale du modèle, tandis que les textures encodent les détails les plus fins et les attributs de surface tels que les couleurs, les normales et les propriétés d'éclairage. Ces modèles peuvent être créés par des artistes utilisant des logiciels de modélisation ou de sculpture 3D, ou obtenus en scannant des objets du monde réel. Dans les deux cas, le nombre de triangles par objet n'a cessé d'augmenter ces dernières années pour représenter des objets encore plus détaillés. De plus, les scènes 3D sont devenues de plus en plus complexes et peuvent être composées de plusieurs milliers d'objets. Cela se traduit par une très grande quantité de triangles qui soulève des problèmes de mémoire, de performance et de filtrage.

Diverses méthodes ont été proposées pour résoudre ces problèmes. D'abord, seuls les objets visibles dans l'image finale doivent être rendus. Les objets situés en dehors du champ de vision de la caméra sont donc ignorés. Pour la même raison, les grands objets qui traversent les limites du champ de vision sont souvent coupés le long de ces limites dans un processus appelé "clipping", et les parties en dehors

du champ de vision sont écartées. Pour les objets opaques et solides, ce qui est le cas de la plupart des objets d'une scène de jeu vidéo typique, les triangles orientés vers l'arrière par rapport au point de vue actuel ne peuvent pas être visibles et sont ainsi écartés en toute sécurité. La notion de "culling" peut être étendue aux occlusions d'un objet par d'autres, c'est ce qu'on appelle "occlusion culling". Ceci est généralement effectué en pré-calculant une forme de hiérarchie de groupes d'objets potentiellement visibles, puis cette hiérarchie est utilisée au moment de l'exécution pour identifier ce qui est visible et ce qui ne l'est pas. Le "culling" et le clipping sont des mécanismes très populaires pour réduire considérablement la complexité d'une scène et ainsi accélérer le rendu. Cependant, lors de l'utilisation de maillages très détaillés, ces mécanismes ne sont pas suffisants. Par exemple, un seul maillage peut contenir plusieurs millions de triangles et s'il est éloigné, il peut être projeté sur une douzaine de pixels, ce qui rend le rendu particulièrement inefficace.

Le rendu de niveaux de détail (LOD) vise à réduire de manière drastique le nombre de triangles visibles dans une scène. L'idée clé des niveaux de détail est d'adapter la résolution de l'objet en fonction du point de vue. Les objets éloignés remplissent moins d'espace sur l'écran que les objets proches et nécessitent donc moins de polygones pour être rendus avec précision. Sur la base de ces observations, une représentation multi-résolution est pré-calculée pour chaque objet et au moment du rendu, la résolution appropriée est utilisée selon un critère dépendant de la vue.

L'approche LOD la plus simple pour les modèles 3D consiste en un ensemble fixe de maillages avec un nombre décroissant de polygones. Au moment du rendu, le maillage avec la résolution la plus basse satisfaisant un critère dépendant de la vue est affiché. L'absence de transition lors du passage d'un niveau à un autre conduit à des artefacts visibles: l'objet 3D devient soudainement plus détaillé. La fameuse technique "Progressive Mesh" aborde ce problème en proposant une représentation à résolution continue. Il consiste en un maillage grossier et un ensemble d'opérations qui indique comment raffiner le maillage grossier dans le maillage original. En plus d'être plus compact que les LOD discrets, il permet un contrôle précis de la densité des vertex avec des transitions temporelles lisses. Cependant, un problème fondamental avec les maillages progressifs est qu'il ne peut pas tirer parti de toute la puissance des architectures GPU massivement parallèles en raison de la dépendance des opérations les unes avec les autres.

Ces représentations LOD sont généralement générées en utilisant un algorithme de réduction de polygone. Le but de tels algorithmes est de réduire le nombre de polygones du maillage tout en préservant le mieux possible son aspect visuel. À cette fin, ils doivent mesurer tout au long de la simplification les différences avec le maillage d'origine. Cette métrique d'erreur est au cœur des méthodes de simplification, car elle définira l'ordre dans lequel les opérations de simplification seront appliquées, mais conduira également les optimisations locales à relocaliser de manière appropriée les sommets tout au long de la simplification. Une bonne métrique d'erreur préservera mieux les caractéristiques importantes du maillage et permettra ainsi de réduire davantage la complexité de la scène pour une même qualité visuelle.

Choisir le niveau approprié pour réduire autant que possible la complexité de la scène tout en préservant son aspect visuel est également une tâche centrale mais difficile. En effet, l'aspect visuel d'un modèle 3D dépend de nombreux paramètres: non seulement sa géométrie, mais aussi son matériau, l'environnement d'éclairage, le point de vue de la caméra, l'affichage, le système visuel humain, les conditions d'éclairage de la pièce, etc. Ces paramètres ont des interactions complexes entre eux et il est donc difficile de les prendre en compte tous en même temps. De plus, l'évaluation d'une telle métrique doit être extrêmement rapide pour être utilisée dans des applications en temps réel. En conséquence, les méthodes existantes sont uniquement basées sur la distance de vue et la géométrie de l'objet, laissant beaucoup de place pour des améliorations significatives.

Une méthode alternative pour représenter des maillages hautement détaillés consiste à utiliser des cartes de déplacement. Le maillage très détaillé est décomposé au cours d'un pré-traitement en une surface grossière et une carte 2D codant les détails, c'est-à-dire le relief, du modèle 3D. Une telle carte s'appelle une carte de déplacement. Plus précisément, la carte de déplacement est une fonction de décalage décrivant comment déformer la surface grossière pour reproduire le maillage d'origine. Cette représentation peut être rendue efficacement en utilisant le moteur de pavage matériel du GPU moderne qui permet un contrôle dynamique de la résolution du maillage. Chaque face, appelée dans ce cas un

patch, du maillage grossier d'entrée est subdivisée à une résolution donnée selon un schéma de subdivision fixe et uniforme. Ensuite, les sommets générés sont déplacés vers l'emplacement 3D souhaité en utilisant la carte de déplacement.

Les spécificités de la tessellation matérielle impliquent de nouveaux défis. Tout d'abord, un défi récurrent lors du rendu des cartes de déplacement avec la tessellation matérielle est d'assurer des transitions temporelles continues. L'objectif est de résoudre les artefacts de "popping" sans introduire d'artefacts de "swimming", c'est-à-dire des fluctuations visibles de la surface reconstruite en raison du sous-échantillonnage de la carte de déplacement.

Les cartes de déplacement avec la tessellation matérielle offre une représentation beaucoup plus compacte que le maillage progressif, grâce au motif de tessellation stockant implicitement la topologie. Malheureusement, cette représentation compacte vient au prix de moins de flexibilité. Pour la même quantité de triangles, il introduit généralement une erreur plus élevée que les approches offrant un contrôle plus fin telles que le maillage progressif. Le deuxième défi est alors de savoir comment activer la tessellation matérielle. Ce problème est particulièrement difficile en raison du modèle de pavage matériel fixe, qui empêche l'utilisation d'algorithmes de simplification de maillage préservant des caractéristiques existants.

Dans cette thèse, nous abordons les défis suivants:

- Comment exploiter au mieux la tessellation GPU pour contrôler plus finement la répartition des sommets tessellés et préserver au mieux les caractéristiques du maillage détaillé?
- Comment générer des LOD de haute qualité compatibles avec la tessellation GPU?
- Comment réaliser des transitions temporelles lisses entre les LOD tout en évitant les fluctuations de la surface?
- Comment sélectionner rapidement le meilleur LOD pour assurer une haute qualité visuelle avec un nombre minimal de polygones?

Pour répondre à ces défis, nous présentons une méthode générale pour la génération et le rendu de LOD compatibles avec la tessellation matérielle. A partir d'un maillage détaillé d'entrée et d'une décomposition correspondante en patchs, nos méthodes produisent une hiérarchie de LOD compatible avec la tessellation matérielle, et une métrique dépendante de la vue mesure, pour chaque patch et chaque niveau de la hiérarchie, son erreur dépendant de la vue. le maillage d'entrée. Cette représentation est ensuite utilisée dans une passe de tessellation non uniforme, contrôlable et personnalisée, évitant les artefacts d'ondulation et fournissant des gains de performances significatifs par rapport aux méthodes alternatives. Plus précisément, nous apportons les contributions suivantes:

Métrique dépendante de la vue Notre première contribution est une métrique dépendante de la vue par patch qui estime à la fois la distance géométrique et la distance entre les attributs des niveaux de détails et du maillage détaillé de référence. Ceci est accompli en considérant des *vecteurs de différence* des points avec les mêmes coordonnées de texture sur la surface du niveau de détail et la surface de référence. L'erreur induite par ces vecteurs peut être résumée de manière conservative par une représentation simple, ce qui permet une évaluation rapide par GPU de l'erreur d'approximation dans toutes les directions de vue. Cette métrique peut remplir plusieurs objectifs. Premièrement, il peut directement être utilisé pour comparer des algorithmes de simplification de manière objective. Plus important encore, nous avons montré comment tirer parti de cette métrique pour concevoir de meilleurs algorithmes de simplification. Enfin, nous avons démontré comment résumer de manière précise et compacte cette métrique sur un patch donné afin de déterminer le niveau de détail approprié au moment du rendu. Elle fournit des gains significatifs par rapport à une métrique isotrope.

Tessellation personnalisée contrôlable Ensuite, nous présentons un nouveau schéma d'interpolation entre les niveaux de tessellation qui permet une tessellation personnalisée contrôlable tout en évitant les artefacts *swimming*. Notre méthode nécessite seulement le stockage d'un index supplémentaire par sommet pour relier les sommets des niveaux grossiers aux niveaux fins. Il nous permet de contrôler finement les emplacements topologiques des divisions et des contractions de vertex, en relâchant la régularité de

la tessellation fractionnaire, tout en conservant l'efficacité des unités respectives du GPU, ouvrant ainsi la voie à une tessellation non uniforme.

Simplification compatible avec la tessellation matérielle Pour exploiter la flexibilité introduite par notre représentation, nous proposons un algorithme de simplification itératifs compatible avec la tessellation matérielle. Partant d'un maillage de base subdivisé et déplacé au niveau le plus fin, notre algorithme décime progressivement le maillage tout en retombant successivement sur le motif de la tessellation matérielle pour les différents niveaux. Nous proposons en plus une heuristique de simplification basée sur notre nouvelle métrique d'erreur pour optimiser simultanément les positions et les coordonnées de texture des sommets relocalisés. En particulier, nous avons proposé la première stratégie de placement de sommets qui optimise à la fois les distorsions géométriques et paramétriques indépendamment de l'échelle. Nous avons ensuite montré que l'ajout d'une étape d'optimisation globale sur l'ensemble du maillage simplifié peut grandement améliorer la qualité des niveaux de détail. Dernier point, mais non des moindres, nous avons également décrit comment utiliser notre métrique d'erreur pour planifier les opérations de simplification afin de minimiser encore plus l'erreur globale.

Notre méthode permet pour la première fois le rendu de niveaux de détails avec la tessellation matérielle, conduisant à d'énormes améliorations en termes de nombre de triangles et de temps de rendu par rapport aux méthodes alternatives. Ce cadre ouvre la porte à de nombreuses opportunités d'améliorations.

Par exemple, notre nouvelle stratégie de placement de sommets et nos procédures d'optimisation globale pourraient être intégrées dans des algorithmes classiques de simplification itératifs. L'intégration de la première est simple, et des optimisations globales pourraient être déclenchées régulièrement lors de la simplification. Si cette combinaison est aussi efficace que dans le contexte de la tessellation matérielle, cela conduirait à un outil hautement compétitif pour la décimation de maillages paramétrés.