Universitat de Girona

PARALLEL SPATIAL DATA STRUCTURES FOR INTERACTIVE RENDERING

Ismael GARCÍA FERNÁNDEZ

Dipòsit legal: Gi. 385-2013
http://hdl.handle.net/10803/107998

Parallel spatial data structures for interactive rendering de Ismael García Fernández està subjecta a una llicència de Reconeixement 3.0 No adaptada de Creative Commons

© 2013, Ismael García Fernández

Doctorat del Programa Oficial de Postgrau en Tecnologia

PhD Thesis

Parallel spatial data structures for interactive rendering

Ismael García
2012

Advisor:

Dr. Gustavo Patow

Memòria presentada per optar al títol de Doctor per la Universitat de Girona

Universitat de Girona

Dr. Gustavo Patow, professor agregat del Departament d'Informàtica i Matemàtica Aplicada de la Universitat de Girona,

CERTIFICA:

Que aquest treball titulat "Parallel spatial data structures for interactive rendering", que presenta Ismael García Fernández per a l'obtenció del títol de Doctor, ha estat realitzat sota la meva direcció.

Signatura

Dr. Gustavo Patow

Girona, 25 de Juliol de 2012

To my lovely wife Cristina, for being there.

Abstract

Advances in graphic processing units (GPUs) introduce new parallel architectures with many processor cores in single computing devices. Graphic algorithms and data structures should be adapted to take advantage of the specific aspects of these current and future parallel many-core architectures – as standard graphics data structures for single or multiple CPUs or fixed-function GPUs are not scalable and flexible enough to do so. Thus, the problem of defining parallel-friendly data structures that can be efficiently created, updated, and accessed is still an ongoing research challenge.

The context of computer graphics is closely related to spatial data, usually defined by points, lines, rectangles, regions, surfaces, and volumes. The representation of such data has always played a crucial role in many applications. More imporatantly, quite often it is crucial that for data be accessed efficiently to improve algorithmic speed. As an example, irregular spatial data should be fitted or resampled into regular domains in order to support efficient parallel evaluation in modern GPUs. These regular spatial data structures allow original samples to be collected and queried in parallel very efficiently.

The main question explored in this thesis is how to define novel parallel random-access data structures for surface and image spatial data with efficient construction, storage, and query memory access patterns.

In order to address this question, representations for shape detail mapping over coarse geometries in real-time applications are explored. The key idea is to create a mapping of the input spatial data on a coarse lattice in which each cell contains a local description of shape and shading information for rendering this region of the domain. This low-bandwidth localized memory access pattern is increasingly advantageous in many-core architectures and crucial to provide a high rendering speed.

Our main contribution is a set of parallel-efficient methods to evaluate irregular, sparse or even implicit geometries and textures in different applications: a method to decouple shape and shading details from high-resolution meshes, mapping them interactively onto lower resolution simpler domains; an editable framework to map high-resolution meshes to simpler cube-based domains, generating a parallel-friendly quad-based representation; a new parallel hashing scheme compacting spatial data with high load factors, which has the unique advantage of exploiting spatial coherence in input data and access patterns.

Resum

Els avenços en les unitats de processament gràfic (GPU) introdueixen noves arquitectures paral·leles amb molts nuclis processadors en un únic dispositiu. Els algorismes gràfics i les seves estructures de dades s'han d'adaptar per aprofitar els aspectes específics de les actuals i futures arquitectures paral·leles - degut a que les estructures de dades estàndards estan diseñades per CPUs o GPUs no programables y no són prou escalables ni flexibles per integrarse al nou paradigme de computació paral.lela. Per tant, el problema de definir estructures de dades que puguin ésser creades de manera eficient, actualitzables i accessibles en paral·lel segueix essent un repte important en la investigació de computació gráfica interactiva.

El context dels gràfics per ordinador està estretament relacionat amb les dades espacials, en general definides per punts, línies, rectangles, regions, superfícies i volums. La representació d'aquestes dades sempre ha jugat un paper crucial en moltes aplicacions. De fet molt sovint és crucial que les dades s'accedeixin de manera eficient per millorar el rendiment en clau de la complexitat de temps algorísmic. A tall d'exemple, les dades espacials irregulars s'han de organitzar o tornar a mostrejar en dominis regulars per tal de donar suport a una l'avaluació paral·lela eficient en les GPUs avançades. Aquestes estructures de dades espacials regulars permeten que les mostres originals es puguin recullir i consultar en paral·lel de manera molt eficient.

La qüestió principal explorada en aquesta tesi doctoral és la forma de definir noves formes d'accés aleatori paral·lel en estructures de dades amb informació de superfícies i d'imatge, amb una construcció eficient, un emmagatzematge compacte i uns patrons d'accés a memòria coherents en les operacions de consulta.

Per tal d'abordar aquesta qüestió, s'han explorat representacions per mapejar els detalls de formes geomètriques complexes en estructures geomètriques simplificades. La idea clau és crear un mapatge de les dades d'entrada espacials sobre una graella regular i compacte en què cada cel·la conté una descripció local de la informació de forma i aparença visual dels detalls associats a aquesta regió del domini regular. Aquest patró d'accés permet reduir el cost d'ample de banda en les transferències d'informació essent molt avantatjós en moltes arquitectures multi-nucli. En resum, tots aquests elements són crucials per a proporcionar una alta velocitat de renderització.

La nostra principal aportació és un conjunt de paral·lels eficients mètodes per avaluar imatges i geometries irregulars, disperses o implícites en diferents aplicacions, i proposem: un mètode per a separar la forma i els detalls d'aparença visual partint de malles d'alta resolució, mapejant de manera interactiva la informació en dominis de més simples de baixa resolució; un marc d'edició geomètrica per convertir malles irregulars de triangles d'alta resolució en representacions més simples basades en un domini de cubs, generant una estructura fàcilment paral·lelitzable basada en primitives quadrangulars; un nou esquema de *hashing* paral·lel per a la organització i compactació de dades espacials en estructues amb un elevat factor de càrrega, la qual cosa té presenta unes avantatges expecionals per explotar la coherència espacial de les dades d'entrada i els seus patrons d'accés a memòria.

Resumen

Los avances en las unidades de procesamiento gráfico (GPU) introducen nuevas arquitecturas paralelas con muchos núcleos de procesadores en un único dispositivo. Los algoritmos gráficos y sus estructuras de datos deben adaptarse para aprovechar los aspectos específicos de las actuales y futuras arquitecturas paralelas - debido a que las estructuras de datos estándares están diseñadas para CPUs o GPUs no programables y no son lo suficientemente escalables ni flexibles para integrarse el nuevo paradigma de computación paralela. Por tanto, el problema de definir estructuras de datos que puedan ser creadas de manera eficiente, actualizables y accesibles en paralelo sigue siendo un reto importante en la investigación de computación gráfica interactiva.

El contexto de los gráficos por ordenador está estrechamente relacionado con los datos espaciales, en general definidos por puntos, líneas, rectángulos, regiones, superficies y volúmenes. La representación de estos datos siempre ha jugado un papel crucial en muchas aplicaciones. De hecho muy a menudo es crucial que los datos se accedan de manera eficiente para mejorar el rendimiento en clave de la complejidad de tiempo algorítmico. A modo de ejemplo, los datos espaciales irregulares se deben organizar o volver a muestrear en dominios regulares para dar soporte a una evaluación paralela eficiente en las GPUs avanzadas. Estas estructuras de datos espaciales regulares permiten que las muestras originales se puedan recoger y consultar en paralelo de manera muy eficiente.

La cuestión principal explorada en esta tesis doctoral es la forma de definir nuevas formas de acceso aleatorio paralelo en estructuras de datos con información de superficies y de imagen, con una construcción eficiente, un almacenamiento compacto y unos patrones de acceso a memoria coherentes en las operaciones de consulta.

Para abordar esta cuestión, se han explorado representaciones para mapear los detalles de formas geométricas complejas en estructuras geométricas simplificadas. La idea clave es crear un mapeo de los datos de entrada espaciales sobre una rejilla regular y compacta en la cual cada celda contiene una descripción local de la información de forma y apariencia visual de los detalles asociados a esa región del dominio regular. Este patrón de acceso permite reducir el coste de ancho de banda en las transferencias de información siendo muy ventajoso en muchas arquitecturas multi-núcleo. En resumen, todos estos elementos son cruciales para proporcionar una alta velocidad de renderización.

Nuestra principal aportación es un conjunto de paralelos eficientes métodos para evaluar imágenes y geometrías irregulares, dispersas o implícitas en diferentes aplicaciones, y proponemos: un método para separar la forma y los detalles de apariencia visual partiendo de mallas de alta resolución, mapeando de forma interactiva la información en dominios más simples de baja resolución; un marco de edición geométrica para convertir mallas irregulares de triángulos de alta resolución en representaciones más simples basadas en un dominio de cubos, generando una estructura fácilmente paralelizable basada en primitivas cuadrangulares; un nuevo esquema de *hashing* paralelo para la organización y compactación de datos espaciales en estructuras con un elevado factor de carga, lo que tiene presenta unas ventajas expecionals para explotar la coherencia espacial de los datos de entrada y sus patrones de acceso a memoria.

Acknowledgments

Finally after many days of hard work I have finished my PhD! I have reached the end of this phase in my life, and all this would not have been possible without the support and motivation of many people.

I am greatly indebted to my advisor Gustavo Patow, whom I have worked with since my undergraduate years. He brought me unnumbered scientific skills. His enthusiasm and passion were an invaluable support throughout my thesis, not just as an advisor but also as a colleague and friend. His natural interest in my work propelled me through the inevitable low points of research and encouraged me to take my own research directions, giving me the freedom to explore these paths, making me feel that I always was having his unconditional support.

Additionally I would like to give a strong thank you to Sylvain Lefebvre, for his mentorship during our collaborations which has helped me in many aspects of my research. His enthusiasm for Computer Graphics has provided me a great opportunity to learn, introducing me to several interesting ideas that surely will impact my future in research.

I am also very grateful to the people at the Geometry and Graphics Group (Universitat de Girona, Spain) for the friendly environment. Thanks to all my colleagues there, Adrià, Oriol, Carles, Albert, Mei, Raissel, Nacho and Lien (and many others to be listed here...!). In special, I want to thank to my closer PhD colleagues and friends Fran and Tere. We had countless coffee breaks and talks, thank you so much for the good moments we had inside and outside the lab.

I also want to thank the people at the ALICE project-team (INRIA Nancy Grand-Est, France). Bruno Levy, Nico Ray and Rhaleb Zayer gave me always a friendly welcome during my stays. Thanks to my other PhD colleagues there as well, Anass Lasram and Vincent Nivoliers. All of them made my stays more pleasant, and motivated me to return every time.

Next, I would like to thank my coauthors for all the pleasant time I had working and discussing with them. In particular, I want to thank Jiazhi Xia, Ying He and Samuel Hornus. I really enjoyed our collaboration in every joint project.

Thanks to my family, especially to my sister and parents, for their support and interest in my work, in spite of not understanding so much of its purpose, they provide me with indescribable wisdom, perspective, and love.

Last but not least, I want to thank my beloved Cristina. This thesis is dedicated to you, for all the love you give me and the unconditional support throughout this very long journey. Yet all of this would be for nothing without you.

Publications

The work presented in this thesis resulted in the following international conferences and journals:

[GP08] IGT: Inverse Geometric Textures
Ismael Garcia, Gustavo Patow
[Journal]
ACM Trans. Graphics (Proc. SIGGRAPH Asia)
DOI 10.1145/1409060.1409090
`http://ismaelgarcia.org/papers/igt_siga2008/`

[XGH^{+}11] Editable Polycube Map for GPU-based Subdivision Surfaces
Jiazhi Xia, Ismael Garcia, Ying He, Shi-Qing Xin, Gustavo Patow
[Conference]
Proceedings of I3D 2011, Symposium on Interactive 3D Graphics and Games
`http://ismaelgarcia.org/papers/epcm_i3d2011/`

[GXH12] Editable Polycube Map for GPU-based Subdivision Surfaces
Ismael Garcia, Jiazhi Xia, Ying He, Shi-Qing Xin, Gustavo Patow
[Submitted to journal]
`http://ismaelgarcia.org/papers/tvcg_i3d2011/`

[GLHL11] Coherent parallel hashing
Ismael Garcia, Sylvain Lefebvre, Samuel Hornus, Anass Lasram
[Journal]
ACM Trans. Graphics (Proc. SIGGRAPH Asia)
DOI 10.1145/2070781.2024195
`http://ismaelgarcia.org/papers/cohash_siga2011/`

[RLD^{+}12] A Runtime Cache for Interactive Procedural Modeling
Tim Reiner, Sylvain Lefebvre, Lorenz Diener, Ismael Garcia, Bruno Jobard, Carsten Dachsbacher
[Journal]
SMI 2012, Shape Modeling International, Computer & Graphics
DOI 10.1016/j.cag.2012.03.031
`http://ismaelgarcia.org/papers/hashcache_smi2012/`

Contents

Contents **17**

1 Introduction **21**
1.1 Rendering strategies . . . 22
1.2 Spatial data organization and representation . . . 23
1.2.1 Spatial data organization . . . 23
1.2.2 Surface and volume data representation . . . 24
1.3 Parallel computing . . . 26
1.3.1 Irregular spatial data and parallelism . . . 26
1.4 Problem statement . . . 27
1.5 Contributions . . . 28
1.6 Document organization . . . 29

2 Background **31**
2.1 Digital surface representation . . . 31
2.1.1 Surface definition . . . 31
2.1.2 Piecewise linear surface representation . . . 34
2.1.3 Surface parameterization and remeshing . . . 36
2.2 Level-of-detail . . . 41
2.2.1 Surface simplification . . . 41
2.2.2 Subdivision surfaces . . . 43
2.2.3 Multiresolution level of detail . . . 45
2.2.4 Error metrics . . . 46
2.3 Real-time rendering . . . 48
2.3.1 Graphics hardware pipeline . . . 49
2.3.2 Graphics processors and parallel programming . . . 51
2.4 Detail mapping data structures . . . 55
2.4.1 Irregular spatial data organization . . . 56
2.4.2 Spatial addressing . . . 56
2.4.3 Spatial data memory layout . . . 56

2.4.4 Spatial data and texture mapping . . . 57
2.4.5 Parallel spatial query access patterns . . . 59
2.4.6 Linear data structures . . . 59
2.4.7 Grid-based data structures . . . 62
2.4.8 Tree-based data structures . . . 69
2.4.9 Hashing data structures . . . 71

3 Detail mapping and simplification 77
3.1 Context: mesh parameterization and simplification . . . 79
3.1.1 Mesh attribute-preserving simplification . . . 79
3.1.2 Texture-based attribute-preserving simplification . . . 80
3.2 Inverse Geometric Textures . . . 85
3.2.1 Parameter domains and mappings: ($M \xrightarrow{\mathcal{C}} P \xrightarrow{\mathcal{P}} T \xrightarrow{\mathcal{M}} D$) . . . 86
3.2.2 Data structures (D, L, A) . . . 90
3.2.3 Constructing the inverse map $\mathcal{I}$ . . . 91
3.2.4 Querying with the inverse map $\mathcal{I}$: ($D \xrightarrow{\mathcal{I}} P$) . . . 92
3.3 Applications . . . 94
3.3.1 Vertex colors . . . 95
3.3.2 Volumetric and procedural texturing . . . 96
3.3.3 Texture mapping . . . 96
3.3.4 Texture transfer . . . 98
3.4 Results . . . 98
3.4.1 Geometry & attributes preservation with IGT: experimental evaluation . 99
3.4.2 Query evaluation . . . 100
3.5 Discussion and limitations . . . 105
3.5.1 Sparse and inconsistent topological geometries . . . 105
3.5.2 Bijectivity limitations in the mappings $\mathcal{C}$ and $\mathcal{P}$ . . . 105
3.5.3 Animation compatibility . . . 106
3.6 Conclusions . . . 106

4 Editable mapping and subdivision surfaces 107
4.1 Context: mesh parameterization and subdivision surfaces . . . 110
4.1.1 Polycube mapping . . . 110
4.1.2 Quadrangulation . . . 112
4.1.3 Subdivision surfaces . . . 112
4.1.4 Cross-parameterization . . . 113
4.2 Editable Polycube Map . . . 113
4.2.1 Overview . . . 115
4.2.2 Constructing Polycube Map . . . 117
4.2.3 Subdivision surface from the polycube map . . . 123
4.3 Applications . . . 123
4.3.1 GPU-based subdivision displacement . . . 123
4.3.2 Kit-bashing . . . 124
4.3.3 Blendshapes . . . 126
4.3.4 Dual Painting . . . 127
4.4 Results and Discussion . . . 128

4.4.1 Results . . . 128
4.4.2 Tradeoff between accuracy and regularity . . . 129
4.4.3 Stroke drawing . . . 130
4.4.4 Automatic Tunnel Slitting . . . 130
4.4.5 Comparison to [HWFQ09] . . . 132
4.5 Conclusions . . . 134
4.5.1 Limitations and Future Works . . . 134

5 Coherent parallel hashing **135**
5.1 Context: parallel hashing . . . 136
5.2 Coherent parallel hashing . . . 137
5.2.1 Notations and definitions . . . 138
5.2.2 Main algorithm and data structure . . . 139
5.2.3 Construction . . . 142
5.3 Results . . . 144
5.3.1 Hashing generic data . . . 145
5.3.2 Hashing in a Computer Graphics setting . . . 146
5.4 Applications . . . 150
5.5 Discussion, limitations and future work . . . 150
5.6 Conclusion . . . 152

6 Conclusions and future work **153**
6.1 Summary of contributions . . . 153
6.2 Perspectives and future work . . . 154
6.2.1 Real-time simplification and parallel localization with random access . . . 154
6.2.2 Efficient and robust semi-automatic parallel parameterization with interactive control . . . 154
6.2.3 Succinct hashing schemes, dynamic hash tables and variable-length data elements . . . 155

Bibliography **157**

Chapter 1

Introduction

> The focus of our study is to design and provide time- and space-efficient parallel data structures and algorithms for real-time rendering applications. This chapter reviews the main concepts and the problem statement of this dissertation.

IN the computer graphics field, rendering processes transform collections of three-dimensional geometric objects into realistic-looking images. A scene description is created by first positioning objects into the scene, then assigning materials and adding light sources. The scene is viewed from a virtual camera and the interactions between lights and materials are computed from that viewing position. The result is a computer generated image. The process of computing the color of each pixel on screen from a three-dimensional scene description is often referred to as *rendering* an image.

Rendering can be broadly divided in two categories, namely *offline* and *real-time* rendering. The focus of this thesis is on the latter, which thanks to faster processors and better algorithms allow continuously more advanced visual effects at interactive frame rates.

> A large number of operations in computer graphics are concerned with the process of collecting *spatial data* in a computer's memory, in such a way that the information can be subsequently recovered as quickly as possible in order to be processed and generate a screen image in real-time. In this context, it is important to retain and organize the spatial data in such a way that fast retrieval and evaluation are possible.

In this chapter we will describe the special conditions of most common collect, query and evaluation operations with spatial data in computer graphics applications. In general we first need to define some concepts to describe these particular problems in more detail.

1.1 Rendering strategies

In computer graphics, there are two fundamental techniques to process the input surface and volume data to generate the screen images: *rasterization* and *raytracing*.

Rasterization based on Z-buffer depth sorting [Cat74], is an object-order algorithm, processing each scene object one after the other, and defined by two main steps, sample selection and interpolation. Sample selection is performed by scanning the input geometric primitives of each object to generate their screen *surface fragments* (see Figure 1.1). Each surface fragment corresponds to a screen pixel coordinate, where additional surface detail samples may be queried, evaluated and interpolated for the final color computation in a so-called *shader program*.

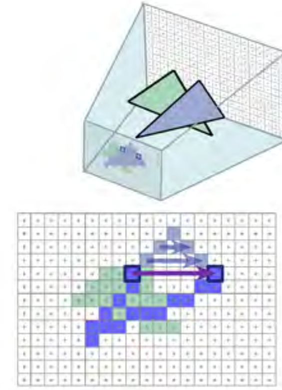


Figure 1.1: Rasterization object-order algorithm shading screen *surface fragments*.

Raytracing [App68] is an image-order algorithm which defines a *view ray* from the camera through each screen pixel. Each ray is traced in order to find *surface intersections* (see Figure 1.2). At each surface intersection point, a shader program may trace additional rays, to then query and interpolate the surface detail samples and generate the final color.

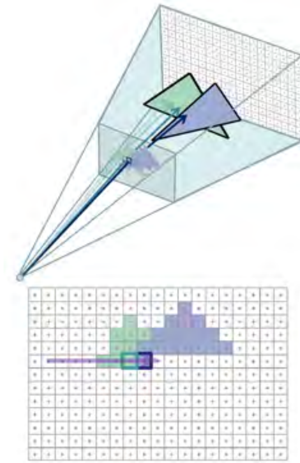


Figure 1.2: Raytracing image-order algorithm shading screen *surface intersections*.

Rasterization and raytracing are usually managed by broadband spatial data structures (see Section 1.2.1). Both techniques also use essentially the same shader programs in surface fragments or in ray intersection coordinates, respectively. At this fine-grained level, it is required to query and evaluate the additional shape and shading detail samples very efficiently.

We focus our study on this specialized level to provide novel efficient parallel data structures to improve rendering performance in real-time applications, by preparing the input spatial data to this end (see Section 1.2.2).

This typically involves defining shape and shading representations over a lattice in which each cell contains the local description required for rendering such regions of the domain, which is specially adapted to take advantage of specific aspects of many-core architectures so that they can be efficiently created, updated and accessed.

1.2 Spatial data organization and representation

The complexity of rendering very detailed objects should be balanced by using a reasonable amount of shape and shading information depending on the distance between the object and the screen point of view. This means that the object's data structures should provide a coarser representation if they appear far way, but still provide the fine-grained details when they are visualized in a close view (see Figure 1.3).

Figure 1.3: Level-of-detail techniques use simplified objects representations across the view distance.

This is specially important in order to provide a good interactive visualization performance, which reduces the rendering cost of querying and evaluating minor, distant or unimportant shape and appearance information.

1.2.1 Spatial data organization

Describing and evaluating spatial data such as surface and volume information can be a computationally complex task. For instance, modeling the interplay of light and the surfaces to create a screen image can require a high computational cost. In large scenes composed of many objects, spatial data structures are used to organize the surface and volumetric objects in three-dimensional space, grouping nearby objects in order to process them efficiently, taking advantage of the spatial locality of their data.

The organization of the spatial data in large scenes is usually addressed with hierarchical data structures [Sam90]. The main reason for using a hierarchy is that different types of queries on the data get significantly faster. Rendering a three-dimensional scene to create a screen image is usually performed using a hierarchical spatial data structure called *scene graph*. A subdivision of the entire space of the scene, with regular or irregular cells, is done, for instance, with *octrees* [Gla84], *bounding volume hierarchy* trees (BVH) [Cla76] (see Figure 1.4), or *kd-trees* [Arv88]. These hierarchical data structures are used as a *broad-band* spatial data organization of the scene. As an example, if some objects of the scene are almost invisible due to size or position, the scene graph can be used to determine that this is the case, discarding them, to avoid unnecessary computations in subsequent rendering pipeline stages.

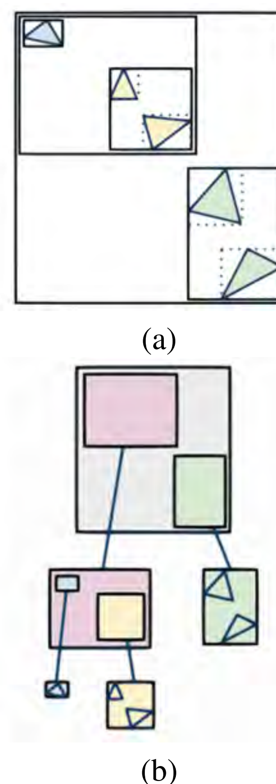


Figure 1.4: Scene objects organization with a bounding volume hierarchy (BVH) data structure. (a) Volumes grouping scene objects. (b) BVH tree of the scene.

1.2.2 Surface and volume data representation

Independently of the spatial broad-band object organization, the surface and volume information of the scene objects should have a flexible *fine-grained* representation, with different levels of detail. This will help to balance the processing time on the selected scene objects that contribute to the final image. The required data structures to this end are extensively used with a large amount of shading computations to obtain the final appearance. Therefore, they must allow very efficient parallel collecting and querying operations.

Detail mapping: Our notion of an object's *shape and shading detail information* includes the color data above the base shape, but also the fine surface and volumetric shape and shading information that creates the final appearence of the object. These high frequency details are meant to be mapped to the base shape of the object and adjusted by a level-of-detail strategy in order to balance the visual quality, the storage and processing costs.

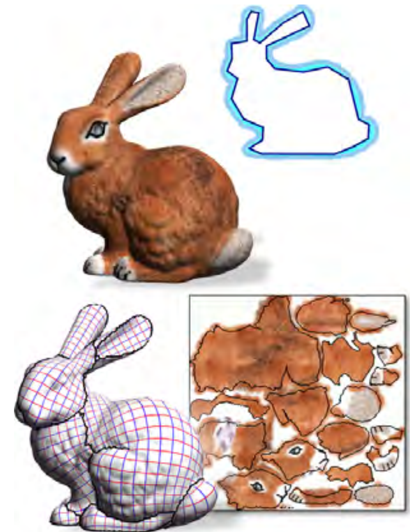


Figure 1.5: (*Top*) 3D Bunny shape. In *dark blue* illustrates the base shape of the bunny, and in *light blue* depicts the layer of shape and shading fine details mapped over the base shape. (*Bottom*) Shows the cutting and flattening of the shape and shading details layer in an image grid.

In general, a *mapping operation* is defined in any spatial object representation in order to link the shape and shading details onto the base shape of the objects. The *map* must allow a flexible representation between the gross shape and the details of the object.

The thin layer of shape and shading details defines a very sparse distribution over the 3D spatial domain. In fact, it is an embedding of a 2D surface layer in the 3D domain. Therefore, usually a uniform 2D grid is used to encode the samples of the map between the base shape and the detail layer. In this process the details are somehow resampled onto the details onto a planar domain in one or multiple patches called charts, usually flattening the surface into the rectangular domain of an image (see Figure 1.5).

Here we describe a set of general data structures for spatial data processing that are commonly used in rendering applications:

- **Mesh data structures** (like triangle and tetrahedron meshes) represent three-dimensional objects by piecewise linear surfaces and volumes defined by a set of planar polygons with an explicit, and often irregular connectivity, which means that each polygon does not have the same fixed number of neighbour polygons aroud it (see Figure 1.6). Mesh structures can provide an easy local control, support arbitrarily topologies and give an efficient display evaluation.

Figure 1.6: A surface triangle mesh data structure.

However, they will only provide a guaranteed continuity, accuracy, and being specially concise, if they are represented with a parametric or a subdivision geometric scheme (see Section 2.2.2). Furthermore, they are not an efficient representation for instance, for intersection operations.

- **Linear data structures** can represent point-sampled surfaces and volumes without a *mesh connectivity* between the samples (see Figure 1.7), which avoids the requirement of surface and volume extraction methods to generate or maintain such connectivity information during the collection or query evaluation during interactive rendering. They are useful, for instance, in presence of large sparsity in the distribution of the data elements in the spatial domain (see Section 2.4.6).

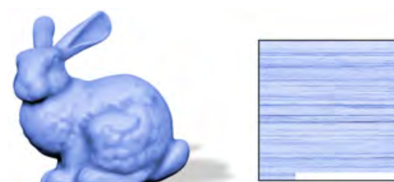


Figure 1.7: A point sampled surface with a linear data structure.

Although they can provide a compact representation without explicitly storing empty space, they still impose challenges to provide at the same time an efficient collecting process, a compact representation and random-access queries on the data. For instance, *sorted arrays* require a binary search to query the spatial data, and in case of new elements to be inserted it may require a full reconstruction, sorting again the full array.

- **Uniform grid data structures** (like 2D and 3D images) represent the data samples aligned to grid locations as pixel data (see Figure 1.8). The image grid defines an implicit connectivity between the neighboring grid locations. So, it allows constant time query operations between neighbor pixels in the grid without requiring any explicit connectivity information for neighbor query evaluations (see Section 2.4.7). However, they are not a space-efficient representation in case of sparse data, leaving most of the records empty.

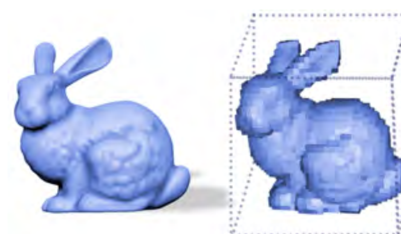


Figure 1.8: A surface sampled on a regular grid data structure.

- **Adaptive grid data structures** (like quadtrees and octrees) provide a subdivision of the bounded object domain with variable resolution (see Figure 1.9). They are useful to organize point-sampled data, boundaries of curves and surfaces, or interior regions like areas and volumes; adaptively sampling the spatial information. However, the query operations require an explicit traversal to get the neighboring data, and also the linked nodes usually have a large space overhead with respect to the sampled data and break the coherence in the memory access pattern (see Section 2.4.8).

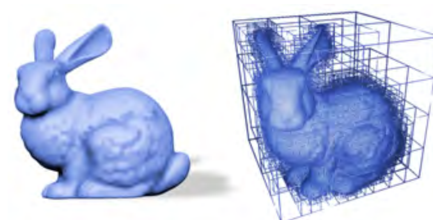


Figure 1.9: A surface sampled on an adaptive grid data structure.

- **Hash data structures** can also represent point-sampled data without *mesh connectivity* between the samples. They are useful in presence of large sparsity in the distribution of the data elements in the spatial domain and they can be constructed to be compact (see Figure 1.10), and still be able to answer queries in almost constant time (see Section 2.4.9).

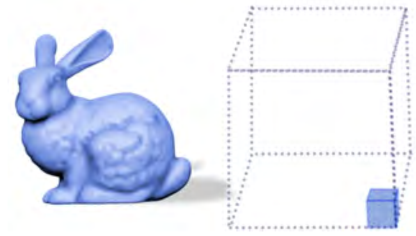


Figure 1.10: A surface sampled on a packed hashing grid data structure.

1.3 Parallel computing

Graphic processing units (GPUs) are becoming so parallel that standard data structures that proved very effective for collecting and recovering geometry and image information in CPU sequential processing, or fixed-function GPUs, are not scalable and practical in current and future many-core architectures. Many-core devices, like GPUs, should be exploited by especially suited data structures to allow collecting and recovering geometry and image information in parallel. Ideally, the data structures would be constructed on the GPU itself using an efficient parallel method to avoid being the bottelneck for a parallel application. Several hierarchical spatial data structures have been recently adapted, like octrees [LK10] and k-d trees [ZHWG08].

In general, the problem of defining parallel-friendly data structures that can be efficiently created, updated, and accessed is still an on going significant research challenge [LSK+06].

1.3.1 Irregular spatial data and parallelism

On the time of the early appearing graphic processing units, a careful optimization of meshes and images was required before the graphics processing, e.g. reducing the connectivity of the mesh to define an irregular but more compact representation, amenable for the computational capabilities of the available GPUs. These operations often required sequential CPU intensive processing but allowed to provide low-bandwidth transfer operations between the CPU and the GPU.

Nowadays the complexity of geometric models used in interactive applications is constantly increasing due to the need of more convincing, detailed, and usually realistic visualizations, requiring, for instance, large triangle meshes. However, these large irregular data structures with explicit connectivity do not scale well for current and future *data parallel processing* on many-core architectures. The large amount of shape and shading information should be presented in a way that allows a more data parallel-friendly evaluation, where each processor is able to perform the same task on different but regular pieces of the distributed data.

As an example, the geometric data represented in irregular-like mesh data structures should be somehow resampled into regular-like domains, in order to avoid the storage of the explicit connectivity. This would give support for a greater parallelism, and at the same time create coherence by providing a better spatial locality in the access pattern. This is important because a major restriction of irregular mesh data structures is that they do not allow a random-access evaluation of the input information, while regular data structures allow to evaluate original samples into screen pixels, querying input data for different pixels very efficiently with random-access operations.

Furthermore, graphics workloads are non-trivial in many ways. In the graphics processing of triangle meshes, each incoming triangle may produce a variable number of surface fragments, the exact number of which is unknown before the rasterization is complete. The number of fragments can vary wildly between different workloads, and also within a single batch of triangles of a same surface. Also, approximately half of the incoming triangles are usually culled, producing no fragments at all [Bli96].

1.4 Problem statement

Shape and shading spatial data can be represented in a variety of ways. The representation ultimately chosen for a specific task is heavily influenced by the type of operations to be performed on the data. Depending on wether the nature of the source data is static or dynamic (i.e., if the number of data points can change during execution), whether the data is defined with a uniform or a sparse spatial distribution, and whether the data fits completely *in-core* or requires *out-of-core* processing, we can take advantage of any of these assumptions to design efficient specialized parallel data structures.

> We focus on the study of novel data structures that should be specialized for in-core parallel-friendly processing. The main question is how to provide an efficient construction, storage, and memory access patterns in the queries for spatial data. The proposed data structures should be compatible with different types of static and dynamic input spatial data commonly used in interactive rendering, with a possible large range of sparsity in the spatial domain.

The set of main conditions that the proposed data structures should follow are:

***Algorithmic* requirements:**

- Exploit parallelism.
- Run on many-core processors (GPUs and CPUs).
- Provide efficient and scalable collecting and querying operations.
- Create coherent access patterns

***Spatial data structure* requirements:**

- Provide a simple and flexible configuration with user-friendly parameters.
- Exploit coherence of the spatial data.
- Provide memory cache alignment patterns (e.g. block memory transfers).
- Exploit available temporal and spatial locality.

1.5 Contributions

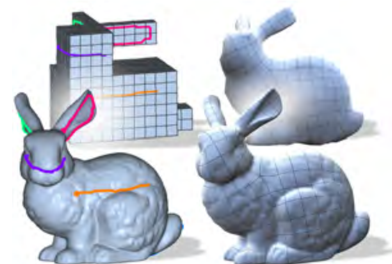
This thesis introduces three specific representations of spatial data with efficient parallel random-access for interactive rendering applications. Surface and volume representations of different topology and sparsity are handled with efficient encoding and rendering algorithms, where the key idea is to create a mapping of the input data to a virtual grid, which naturally suits for parallel graphics processing units with a Single Instruction, Multiple Data (SIMD) programming model.

> The proposed approaches create a coarse lattice in which each cell contains a local description of surface and volume information, required for rendering such regions of the domain. This low-bandwidth localized memory access pattern is increasingly advantageous in many-core architectures, were the usage of random-access parallel data structures is crucial to provide fast rendering speed and good visual quality.

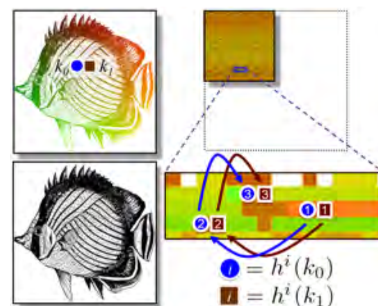
Basically, this manuscript proposes three main contributions, namely:

A detail mapping method for surface simplification based on an inverse parameterization. The coarse shape of the object representation is decoupled from its surface and shading details on high resolution triangle meshes, allowing to map the original shading and surface details onto any lower resolution simpler domain representation of the object. The triangular mesh can be simplified with almost no constraints, and the inverse parameterization allows accessing the original high resolution information, resulting in higher quality compact meshes preserving the original high-resolution surface information.

An editable mapping method for subdivision surfaces that provides a framework to map high-resolution triangular meshes to simpler cube-based domains (polycube). These are the basis to generate a quad-based parallel-friendly representation. The sketch-based interface allows the users to easily modify and fine-tune the mapping, and perform other modeling operations between the triangular mesh and the polycube. The quad-based representation is convertd to a subdivision surface specially built for quad patch-based tessellation on the GPU.



A real-time parallel hashing method that introduces a parallel-friendly hashing scheme to compact surface and volumetric spatial data with high load factors. It provides the unique advantage of exploiting spatial coherence of the input data and in the access pattern. The technique creates and accesses the data structure in the GPU, in a parallel very efficient way, leading to increased locality in the memory access, and an increased coherence in the parallel execution paths accessing the data.



These elements form a set of parallel efficient strategies to be able to map, from sparse geometries, irregular meshes, implicit surfaces, vector textures, and raster images, to memory efficient and random-access data structures for fast interactive rendering applications.

1.6 Document organization

This thesis is structured as follows. After this introduction, a background on level-of-detail, parallel programming and spatial data structures is given in Chapter 2. From Chapter 3 to Chapter 5, the three novel techniques are presented in detail. In all they define a set of parallel spatial data structures to improve the rendering performance of the real-time graphics pipeline. The thesis is completed by a conclusion in Chapter 6 which also contains a discussion and perspectives of future work.

See below the main topics of each chapter:

Chapter 1: Introduction

The focus of our study is to design and provide time- and space-efficient practical parallel data structures and algorithms for real-time rendering applications. This chapter reviews the main concepts and the problem statements of the study described in this manuscript.

Chapter 2: Background

Shape and shading representations are of great interest in computer graphics. This chapter are presented most relevant and close related developments with level-of-detail from shape and shading information, describing and identifying their key points that inspired our proposed contributions.

Chapter 3: Detail mapping and simplification

Surface simplification must deal with a problem of great importance: preserving both the shape and shading details attached to a geometric mesh structure without constraining the final quality. This chapter presents our proposed special setting with a parameterization-based spatial directory to avoid the limiting issues and generate high quality compact and parallel-friendly representations.

Chapter 4: Editable mapping and subdivision surfaces

Shape representations require *simplicity* and *regularity* on the mesh structure of many computer graphics applications. This chapter presents a framework with a sketch-based

editable mapping from complex shapes onto high quality cube-based parametric domains. The provided representation is specially useful for hardware parallel tessellation with subdivision surfaces, displacement mapping and other modeling applications.

Chapter 5: Coherent parallel hashing

Collecting *sparse* spatial data is particularly useful when having some elements of interest sparsely located in the spatial domain. This chapter presents parallel hashing methods to robustly create smaller tables with only the interesting shape and shading elements of implicit or irregular surfaces of objects, while exploiting coherence in the spatial data and the access patterns for fastest random-access parallel queries in many computer graphics applications.

Chapter 6: Conclusions and future work

The thesis is completed by a conclusion which also contains a discussion and perspectives of future work.

Chapter 2

Background

Shape and shading representations are of great interest in computer graphics. This chapter presents the most relevant and closely related developments about mathematical models and computer representations from shape and shading information, and describes and identifies the key points that inspired our proposed contributions.

Advances in 3D digital geometry processing and computer graphics have created a plenitude of novel concepts for the mathematical representation and interactive manipulation of graphics models to capture and modify the shape of physical objects. These three-dimensional geometric models form the basis of many interactive rendering applications.

2.1 Digital surface representation

In this section we describe mathematical models and the discrete geometric representations of shapes, introducing basic notions of differential geometry and the properties of piecewise linear surface representations.

2.1.1 Surface definition

Here we introduce the concepts involved in the description and computation of the geometric aspects of 2D and 3D objects. These concepts are related to the mathematical study of shape and form – which are essential for some of the geometry processing methods studied in this manuscript. To begin, we will describe the requirements of a shape as candidate *topological space* and then we will formalize the definition of a *surface*.

Given a topological space (X, T_x), where X is a set, and T_x is a collection of subsets of X, it satisfies the following axioms:

- The *empty set* and X are in T_x.
- T_x is closed under arbitrary union.
- T_x is closed under finite intersection.

Given a shape defined as a topological space (X, T_x), it must meet the following four requirements to be formaly defined as a *surface*:

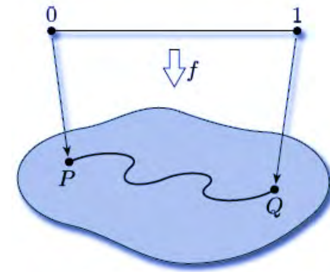


Figure 2.1: A surface must be path connected, any two points P and Q can be joined by a curve.

1) The first requirement is that the surface should be in *just one piece*. This can be ensured by requiring the surface to be *path-connected*, which means that any two points P and Q on the surface can be joined by a curve that lies entirely in the surface (see Figure 2.1). In mathematical terms, this means that there is a continuous map f from the interval $[0, 1]$ to the surface, such that $f(0) = P$ and $f(1) = Q$.

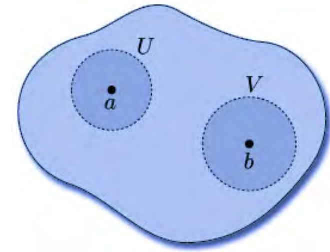


Figure 2.2: A surface must be a *Hausdorff space*, for any pair of distinct points a and b there are two *disjoint* open sets U and V, one containing a and the other containing b.

2) The second requirement is a technical one that is needed to eliminate certain awkward cases. We require a surface to be a *Hausdorff space*. This means that, given any pair of distinct points a and b in the space, there are *disjoint* open sets U and V, one containing a and the other containing b (see Figure 2.2). As $\mathbb{R}^3$ with the Euclidean topology is a Hausdorff space, any subset of it with the induced topology, such as a surface in space, is also a Hausdorff space. Thus it follows that for a shape defined as a topological space to be a surface in space, it must also be a Hausdorff space.

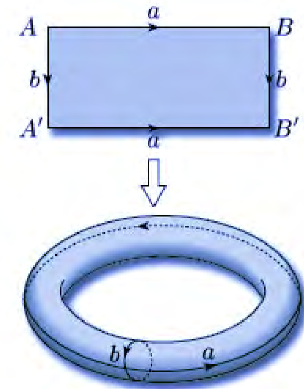


Figure 2.3: A surface must be a *compact surface* obtained from a *closed polygon* (or a finite number of polygons) by identifying boundary edges.

3) The third requirement it is to be a *compact surface*, in the sense that it can be obtained from a *closed polygon* (or a finite number of polygons) by identifying boundary edges. In Figure 2.3 we can see an example of the construction of the compact surface of a torus: the process corresponds to the identification of opposite edges in pairs following the directions indicated by the arrows. For ease of reference, we label each pair of edges that is to be identified with the same letter, a for the first pair to be identified and b for the second pair. An example of surface that is not compact is a cylinder without its bounding circles and the entire plane.

4) The fourth requirement is that the topological space (X, T_x), in order to be a surface, it must satisfy that, given any point $x \in X$, there is an open set U containing x such that U is *homeomorphic* either to an open disc in $\mathbb{R}^2$ with the Euclidean topology or to an open half-disc in the upper half-plane with the subspace topology inherited from the Euclidean topology on $\mathbb{R}^2$. As an example, in a cylinder –except for the points at the ends of the cylinder boundaries – each point has a disc-like neighborhood, but the points at the ends have no such neighbourhoods (see Figure 2.4). Instead, each such endpoint x has a half-disc-like neighborhood, being then a subset that is homeomorphic to an open half-disc.

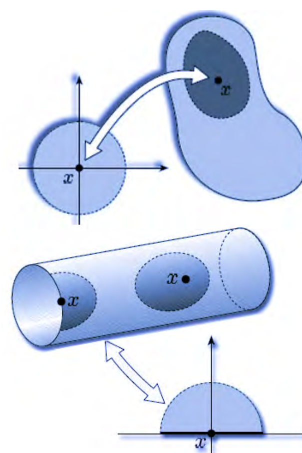


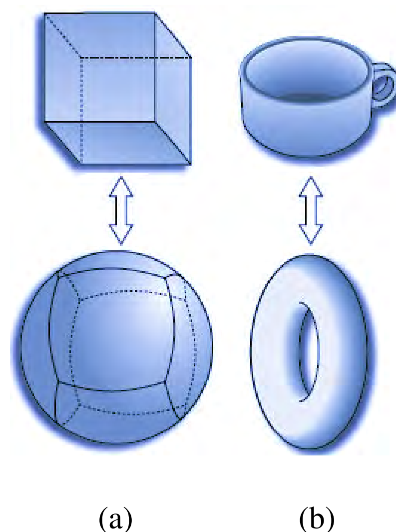
Figure 2.4: Surface open sets must define a *homeomorphism* to a disk or half-disk.

Now we can provide the formal definition of a surface:

> A surface is a compact path-connected Hausdorff topological space (X, T) with the property that, given any point $x \in X$, there is an open set U containing x such that U is homeomorphic either to an open disk in $\mathbb{R}^2$ with the Euclidean topology or to an open half-disk in the upper half-plane with the subspace topology inherited from the Euclidean topology on $\mathbb{R}^2$. A *surface in space* is a surface (X, T_x) where X is a subset of $\mathbb{R}^3$ and T_x is the subspace topology on X inherited from the Euclidean topology T on $\mathbb{R}^3$.

In a comparison between surfaces we can classify them as topologically equivalent or as different according to the *homeomorphic relation*, which is as follows:

The homeomorphic relation between two topological spaces (X, T_x) and (Y, T_y) is valid if there is a *bijection* $f : X \to Y$ that is continuous, and whose inverse f^{-1} is also continuous with respect to the given topologies. Such a function f is called a homeomorphism. The relation '*is homeomorphic to*' between topological spaces is the most fundamental relation in topology, and an important requirement for the surface definition. If two topological spaces are homeomorphic, it means that they are indistinguishable from a topological point of view, they are topologically equivalent. An alternative definition of a homeomorphism is that a *bijection* $f : X \to Y$ is a homeomorphism if and only if both f and f^{-1} map open sets to open sets. Thus, if (X, T_x) and (Y, T_y) are homeomorphic, then not only are the elements of X and Y in one-to-one correspondence, but so are their open sets. We can thus regard (Y, T_y) as being essentially the same space as (X, T_x) so far as its purely topological properties are concerned: (X, T_x) and (Y, T_y) are merely two different ways of presenting the same space.

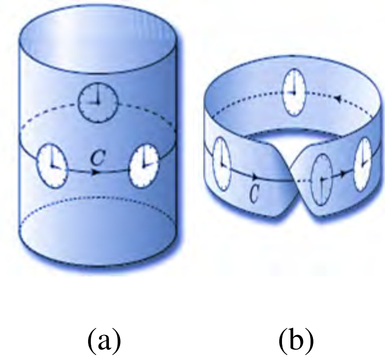


(a) (b)

Figure 2.5: Homeomorphism between 2 surfaces: (a) Between cube and sphere (genus 0). (b) Between a torus and a mug (genus 1).

Given two topological spaces as surfaces (X, T_x) and (Y, T_y), a bijective map f from X to Y is called a *diffeomorphism* if both $f : X \to Y$ and its inverse $f^{-1} : Y \to X$ are differentiable. If these functions are r times continuously differentiable, f is called a $C^r - diffeomorphism$. Two surfaces X and Y are diffeomorphic if there is a diffeomorphism between them. As an example, the relation between a cube and a sphere, and a doughnut and a teacup are homeomorphisms too, since both shapes are topologically equivalent (see Figure 2.5).

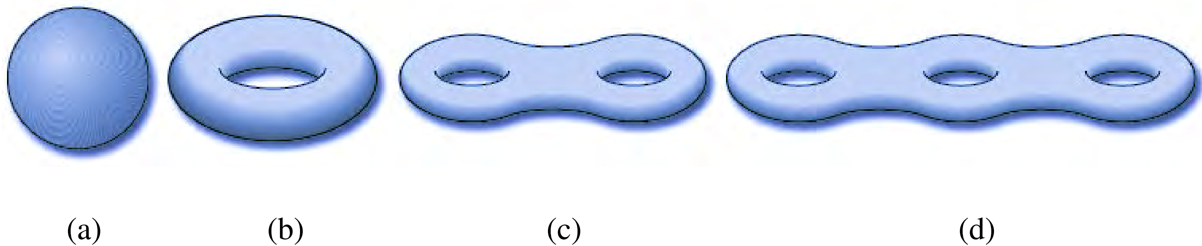
The idea of *orientability* is another fundamental concept necessary for the study of surfaces. Two surfaces that are homeomorphic have the same orientability number. We observe that this is not the case between a cylinder and a Möbius band by noticing that every cylinder has an 'inside' and an 'outside', and we can paint one red and the other blue. But if we try to paint a Möbius band in two colors, we fail, as it has just one 'side'. Any one-sided surface in space is non-orientable (see Figure 2.6).



(a) (b)

Figure 2.6: Orientable and non-orientable surfaces.

The genus of a connected, orientable surface is an integer value representing the maximum number of cuts along non-intersecting closed simple curves without rendering the resultant manifold disconnected (see Figure 2.7). It is equal to the number of handles on it. Alternatively, it can be defined in terms of the Euler characteristic X, via the relation $X = 2 - 2g$ for closed surfaces, where g is the genus. For surfaces with b boundary components, the equation reads $X = 2 - 2g - b$.



(a) (b) (c) (d)

Figure 2.7: Genus of a surface: (a) Sphere (genus 0). (b) Torus (genus 1). (c) Two-ring torus (genus 2). (d) Three-ring torus (genus 3).

2.1.2 Piecewise linear surface representation

In computer science, polygonal meshes remain the most common and flexible way to approximate surfaces. A polygonal surface model, also known as a *mesh*, is a piecewise linear surface in the three-dimensional Euclidean space $\mathbb{R}^3$. Without loss of generality, it can be assumed that the set of planar polygons defining a mesh consists entirely of triangular faces, since any non-triangular polygon may be triangulated in a preprocessing step [Sei91].

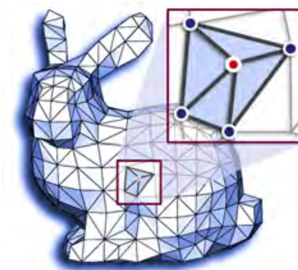


Figure 2.8: Closed mesh 2-manifold. Every edge in the mesh is shared by exactly two faces and the neighborhood of every vertex consists of a closed loop of faces.

A polygonal surface is said to be a *2-manifold* mesh, called *closed mesh* (see Figure 2.8), if every edge in the mesh is shared by exactly two faces and the neighborhood of every vertex consists of a closed loop of faces. In the case where the mesh does have a boundary, it is a 2-manifold called *open mesh* (see Figure 2.9), where the boundary edges must have only one incident face and the neighborhood of boundary vertices consists of a single fan of faces.

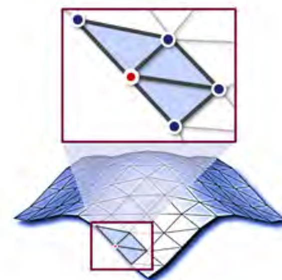


Figure 2.9: Open mesh 2-manifold. Any boundary edge must have only one incident face and the neighborhood of boundary vertices consists of a single fan of faces.

2.1.2.1 Face-vertex mesh

A face-vertex mesh $M = (V, F)$ is a structure containing a list of vertices V, and a list of triangular faces F. The vertices list $V = (v_1, v_2, \cdots, v_m)$ is an ordered sequence where each vertex may be identified by a unique integer i. The faces list $F = (f_1, f_2, \cdots, f_n)$ is also ordered, assigning a unique integer to each face. Every vertex $v_i = (x_i, y_i, z_i)$ is a vector in the Euclidean space $\mathbb{R}^3$. Each triangle $f_i = (j, k, l)$ is an ordered list of three indices identifying the corners vertices (v_j, v_k, v_l) of f_i.

2.1.2.2 Half-edge mesh data structure

A half-edge mesh data structure [Wei85] is an edge-centered structure capable of maintaining incidence information of vertices, edges and faces, for example for planar maps, polyhedra, or other orientable, two-dimensional surfaces embedded in an arbitrary dimension. Each edge is decomposed into two half-edges with opposite orientations. One incident face and one incident vertex are stored for each half-edge (see Figure 2.10). For each face and each vertex, the first incident half-edge is stored.

Figure 2.10: Half-edge mesh data structure: one incident face (shown in light blue) and one incident vertex (shown in red) are stored for each half-edge.

The half-edge data structure provides constant time access queries to the neighborhood of an arbitrary point without requiring any search operation during traversals. For instance, we can easily compute a normal for any given face on demand, even if the vertex locations are changing, or we can traverse every face touching a vertex (the *start* of the vertex) to easily estimate a normal for that vertex. For instance, if we are locally changing small portions of a mesh, this is much easier than recomputing the normals for every vertex in the mesh. Similarly, mesh simplification or subdivision algorithms such as Loop and Catmull-Clark are relatively easy to integrate for level-of-detail geometry processing. An overview and comparison of these different data structures together with a thorough description of the design implemented here can be found in [Ket99].

The orientation of a face is a cyclic order of the incident vertices. A manifold mesh is *orientable* if any two adjacent faces have *consistent orderings* (see Figure 2.11). Let f_i and f_j be adjacent faces sharing the edge (v_i, v_j). If v_i and v_j occur in this order for f_i, then they must occur in f_j in the order v_j followed by v_i. The typical choice in computer graphics applications is based on the visibility of the mesh from the camera location. The vertices are ordered counterclockwise in the plane of the face viewed from outside the mesh. This assumption defines the normal vectors directed toward the eye.

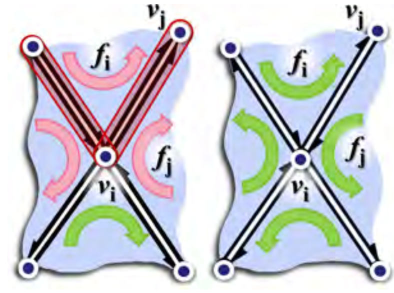


Figure 2.11: 2-manifold mesh orientation. (Left) Non-orientable surface. (Right) Orientable surface.

2.1.3 Surface parameterization and remeshing

Geometry processing methods exploits the mathematical properties of a surface representation to map the original object surface shape and shading details into more suitable domains (e.g. a planar domain), or to transfer such details onto a higher quality mesh structure generated by mapping the input mesh structure to more suitable base-complex domains. Therefore, it is highly desirable to find a diffeomorphism between a 2-manifold surface and a suitable Euclidean space domain in many computer graphics applications.

2.1.3.1 Surface planar parameterization

Mesh parameterization was introduced in computer graphics to find a one-to-one mapping from a suitable parameter domain to the given surface. Given any two surfaces with the same topology (S_T and D_T), an important goal of parameterization is to obtain *bijective* (invertible) maps, where each point on the parameter domain corresponds to exactly one point of the given surface. The mapping f between the triangular mesh S_T and the triangulation of the parameter domain D_T, is defined as being piecewise linear, associating each triangle of the original mesh with a triangle in the parameter domain (see Figure 2.12). The one-to-one mapping from the suitable parameter domain to the surface must minimize angle distortions (*conformal parameterization*) and area distortions (*equiareal parameterization*) to obtain a good quality parameterization.

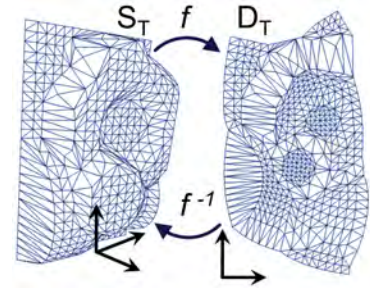


Figure 2.12: Piecewise linear mapping where each point on the parameter domain D_T corresponds to exactly one point of the given surface S_T.

Here, we first focus our interest on methods parameterizing triangulated surfaces which are homeomorphic to a disk, with piecewise linear mappings onto a planar domain.

Harmonic mapping: The harmonic map objective is to parameterize a given disk-like surface $S \subset \mathbb{R}^3$ into a unit disk D in the plane.
A *harmonic function* is a twice continuously differentiable function: $f : S \rightarrow \mathbb{R}^2$, where S is an open subset of $\mathbb{R}^2$ which satisfies the Laplacian equation $\triangle f = 0$ everywhere in S, and

$$\triangle = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \tag{2.1}$$

Harmonic functions can be generalized on arbitrary Riemannian manifolds (i.e. manifolds possessing a metric tensor such as the geodesic in a Euclidean space) as a Laplace-Beltrami operator. The Laplace-Beltrami operator is also the divergence of the gradient: $\triangle f = Div(\bigtriangledown f)$. More details can be found in Schoen and Yau [SY97].
If S and D are two Riemannian manifolds, such as 2-manifold surfaces, then a harmonic map $f : S \to D$ is defined to be a stationary point of the Dirichlet energy

$$E(f) = \frac{1}{2} \int_S \parallel \mathsf{g}rad_S f \parallel^2 \tag{2.2}$$

The *Radó Theorem* [Rad26] – proved independently by Kneser [Kne26] and Choquet [Cho45], gives the theoretic foundation for harmonic surface mapping between 2D convex domains. Suppose $S \in R^2$ is a convex domain with a smooth boundary ∂S, and suppose that D is the unit disk. Then, given any homeomorphism $\mu : \partial D \to \partial S$, there exists a unique harmonic function $f : D \to S$ such that $f = \mu$ on ∂D and μ is a diffeomorphism [SY97]. When we focus on mappings from general surfaces $S \in \mathbb{R}^3$ to a plane, we find that all of the above properties are essentially the same [FH05].

These interesting properties attracted a lot of harmonic field-based algorithms [ZH99, WGTtY, LGW$^+$07, MCK08]. Eck *et al.* [EDD$^+$95b] introduced discrete harmonic maps into the computer graphics community, working on triangular meshes S_T. The basic approach has two main steps:

1. First, the boundary mapping is fixed by mapping the polygonal boundary of S_T homeomorphically to the boundary of the unit disk triangle mesh D_T. This is equivalent to choosing the planar image of each vertex in the mesh boundary [Flo97a].

2. Second, the piecewise linear mapping $f : S_T \to D_T$ is found, which minimizes the Dirichlet energy. Consider one triangle $T = [v_1, v_2, v_3]$ in the surface S_T, then, referring to Figure 2.13, one can show that:

$$\begin{aligned} 2\int_T \parallel \mathsf{g}rad_T f \parallel^2 = \\ \mathsf{c}ot\theta_3 \parallel f(v_1) - f(v_2) \parallel^2 + \\ \mathsf{c}ot\theta_2 \parallel f(v_1) - f(v_3) \parallel^2 + \\ \mathsf{c}ot\theta_1 \parallel f(v_2) - f(v_3) \parallel^2 \end{aligned} \tag{2.3}$$

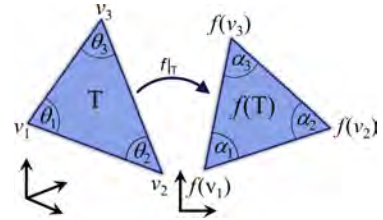


Figure 2.13: Atomic map between a mesh triangle and the corresponding triangle in parametric-space.

The normal equations for the minimization problem, can be expressed as the linear system of equations,

$$\sum_{j \in N_i} w_{ij}(f(v_j) - f(v_i)) = 0, v_i \in V_I, \quad (2.4)$$

where

$$w_{ij} = \cot\alpha_{ij} + \cot\beta_{ij} \quad (2.5)$$

and the angles α_{ij} and β_{ij} are shown in Figure 2.14.

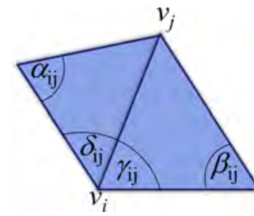


Figure 2.14: Angles for the discrete harmonic map.

The associated matrix is symmetric and positive definite, and thus the linear system is uniquely solvable. The matrix is also sparse and iterative methods are effective, e.g., conjugate gradients. Note that the system has to be solved twice, once for the x and once for the y coordinates of parameter points $f(v_i), v_i \in V_I$.
The properties of the harmonic map are exploited in our proposed global parameterization described in chapter 4.

2.1.3.2 Surface multi-chart parameterization

Segmentation: a planar parameterization is only applicable to surfaces with disk topology. Hence, closed surfaces and surfaces with genus greater than zero must be cut prior to planar parameterization, as shown in Figure 2.15. Furthermore, complex surfaces usually increase parameterization distortion, independently of the parameterization technique used. To allow parameterizations with low distortion, the surfaces must be cut to reduce the complexity.

Since cuts introduce discontinuities into the parameterization, a delicate balance between the conflicting goals of smaller distortion and shorter cuts has to be achieved. It is possible to use constrained parameterization techniques to reduce cross-cut discontinuities.

Cutting and chart generation are most commonly used when computing parameterizations for mapping of textures and other signals onto the surface (see Section 3.1). Depending on the application, mesh segmentation techniques use different criteria for creating charts.

Usually, surfaces are broken into several charts until the parametric distortion of each chart is sufficiently low, while the number of charts remains small and their boundaries are kept as short as possible [MYV93, GWH01, SWG+03].

Chart packing: chartification techniques raise an additional post-processing challenge. Following the parameterization of each individual chart, these charts need to be placed, or packed, in a common planar parameter domain, with a packing as compact as possible [LPRM02] (see Figure 2.15 (*Bottom*).

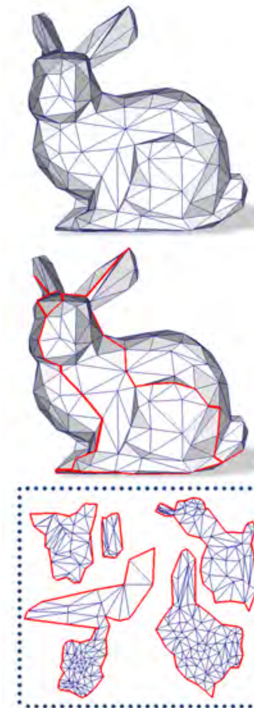


Figure 2.15: Closed surfaces and surfaces with genus greater than zero must be cut prior to planar parameterization. Following the parameterization, the charts need to be packed as compact as possible in the planar parameter domain.

2.1.3.3 Parameterization distortion metric

A parameterization distortion metric is used to assess the map quality. In general, it is defined by the relation between the geometric shape of the parameterization domain triangles and the shape of the original triangles, which in general are slightly different and result in *angle* and *area distortions*. Parameterization methods try to minimize distortion for the whole mesh, but very few meshes admit *isometric* (i.e. zero distortion) parameterizations. Maps that minimize angular distortion, or shear, are called *conformal*, and maps that minimize area distortion are called *authalic*. Most methods focus on angle preservation and then balance it with area preservation.
For each triangle t_obj_i with the 3D object space coordinates p_i, q_i, and s_i and the associated triangle t_par_i with the parameter domain coordinates u_i, v_i, and t_i, where $\angle s_i$, $\angle q_i$, $\angle p_i$ are the angles of each side, and $\| p_i - q_i \|$, $\| p_i - s_i \|$, $\| q_i - s_i \|$ are the side lengths.

Following Degener *et al.* [DMK03a], we measure the normalized angle distortion by

$$S_i = \frac{\angle s_i^2 cotan(\| p_i - q_i \|) + \angle q_i^2 cotan(\| p_i - s_i \|) + \angle p_i^2 cotan(\| q_i - s_i \|)}{4 \, area_{\triangle}(u_i, v_i, t_i)} \tag{2.6}$$

$$E_{norm.angle} \frac{\sum_{i=0}^{n} area_{\triangle}(p_i, q_i, s_i) S_i}{\sum_{i=0}^{n} area_{\triangle}(u_i, v_i, t_i)} \tag{2.7}$$

And the normalized area distortion is measured by

$$E_{norm.area} = \frac{area_{\triangle}(u_i, v_i, t_i)}{area_{\triangle}(p_i, q_i, s_i)} + \frac{area_{\triangle}(p_i, q_i, s_i)}{area_{\triangle}(u_i, v_i, t_i)} \tag{2.8}$$

For an isometric parameterization, the normalized angle and area distortion should be 1. So in practice the closer the metric is to 1, the better the quality of the parameterization (see Figure 2.16).

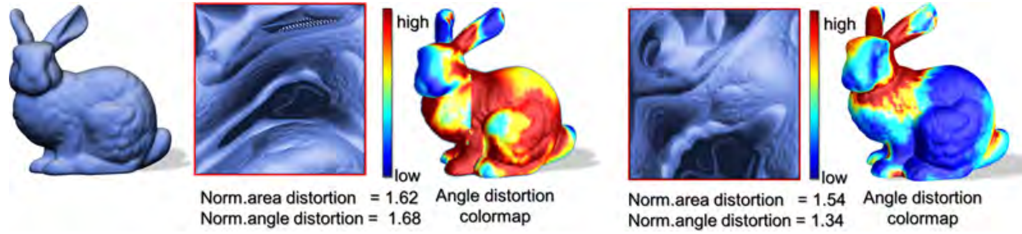


Figure 2.16: Angle and area parameterization error metrics following Degener *et al.* [DMK03a]. (Left) Parameterized bunny with a *geometry image* [GGH02a] with large angle distortion (shown in red). (Right) Parameterized bunny with a *spherical geometry image* [PH03] shows less area and angle distortion (shown in blue as the normalized angle distortion equal to 1).

2.1.3.4 Surface remeshing

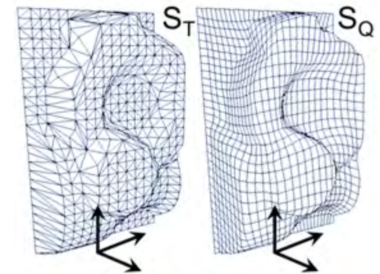


Figure 2.17: Surface quadrilateral remeshing: from an irregular triangle-based mesh structure to a regular quad-based mesh structure.

Surface remeshing is the process that, given a 3D mesh, computes another mesh whose elements satisfy some quality requirements to create a structured regular- or semi-regular mesh replacing the unstructured input mesh (see Figure 2.17).

A structured mesh, sometimes called *regular mesh*, has all internal vertices (except in the boundary) surrounded by a constant number of elements (e.g. edges and faces). This offer certain advantages over an unstructured mesh, because its connectivity graph is significantly simpler, hence it allows for efficient traversal and localization queries on the spatial data.

Semi-regular meshes are essentially piecewise-regular where most of the vertices are regular, meaning that they will have a regular *valence* of 6 edges for triangles, and 4 edges for quadrilaterals (see Figure 2.18). However, they also include a small number of *extraordinary vertices* (also called irregular or singular vertices), which have a valence different from the ones already described. In general, semi-regular meshes offer a good tradeoff between the simplicity of structured meshes and the flexibility of unstructured meshes.

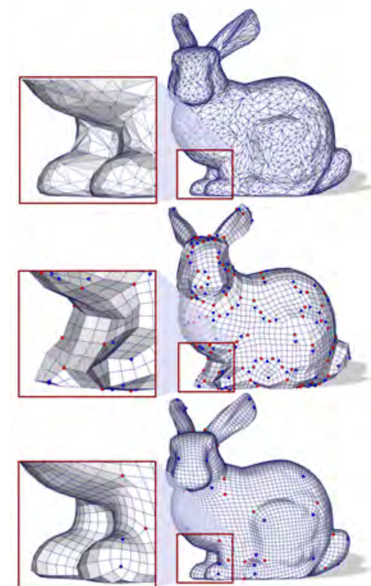


Figure 2.18: Surface semi-regular quad-based remeshing. (*Top*) Input irregular triangle mesh. (*Middle*) [RLL+06b] quad-based remeshing showing a large number of irregular vertices. (*Bottom*) [XGH+11] another quad-based remeshing showing fewer irregular vertices.

Many techniques for semi-regular remeshing [GVSS00, GGH02a, RLL+06a] use a parameterization [EDD+95b, Flo97b, HG00] to find a bijective correspondence. The parameterization plays a critical role, and any deficiencies in the process will be amplified in the output mesh. In particular, building a globally smooth parameterization is a difficult research problem. One challenge is to obtain semi-regular meshes with a prescribed low number of irregular vertices.

In general, high quality remeshing means generating a new discretization of the original geometry with a mesh that exhibits the three following properties: well-shaped elements, uniform or isotropic sampling and smooth gradation sampling.

A well-shaped triangle has aspect ratio as close to 1 as possible, and a well-shaped quadrilateral contains angles between two consecutive edges as close to $\pi/2$ as possible. Isotropic sampling means that the sampling is locally uniform in all directions. Requiring uniform sampling is even more restricting by obliging the sampling to be uniform across the entire mesh. Smooth gradation means that if the sampling density is not uniform, it should vary in a smooth manner.

2.2 Level-of-detail

In geometric modeling and computer graphics, polygonal mesh structures are commonly used as the surface representations of objects in simulations and visualization methods. Advances in acquisition systems, such as laser range scanners and medical imaging devices are easily able to provide vast spatial databases containing millions of polygons, far more than current graphics hardware can render at interactive frame rates. The key to an acceptable solution for real-time graphics of complex scenes is to have a series of geometric approximations that resemble the original complex objects from all directions, but with increasingly lower rendering costs. Furthermore, the transitions between one approximation and the next one must be barely noticeable in order to effectively reveal details as objects approach the viewpoint.

2.2.1 Surface simplification

In polygonal surface applications a tradeoff exists between the accuracy with which a surface is modeled and the amount of time required to process it. If a simplified representation of those models is used, potential gains can be obtained. Eliminating redundant geometry will reduce model representation size and improve the run-time performance of the scene being rendered. This is one of the main motivations for using surface simplification techniques, where different aspects such as geometric and visibility properties can be taken into account when simplifying a model.

Surface simplification is naturally targeted towards large and complex datasets that would be very hard to manipulate manually. Suppose we have a polygonal model M and we would like an approximation M' having fewer polygons than the original, but being as similar as possible to M, retaining all its most relevant features. The goal of polygonal surface simplification is to automatically produce such approximations. It is important to mention that the computation of the minimal-facet approximation within a certain error bound is a NP-hard problem. A characterization and classification of surface simplification methods is presented by Andujar [And99].

The most common surface simplification approach is *decimation*, working as a top-bottom strategy starting from the original surface and iteratively removing elements at each step using a *face reduction* strategy until the desired level of approximation is achieved. The main face reduction strategies using a *local operator* are summarized below:

2.2.1.1 Vertex decimation

Vertex decimation [SZL92, KLS96, Kle98] operates iteratively, selecting unimportant vertices for removal and then re-triangulating the resulting hole (see Figure 2.19). In each step of the decimation process the vertex v_r with the lowest error metric weight (being the less valuable vertex) is selected for removal. Then, the operator eliminates all the t incident triangles, resulting in a hole that is reconstructed with $t - 2$ triangles. The criterion for vertex removal in the simplest case (an interior vertex not on an edge or a corner) is the distance from the vertex to the plane approximating its surrounding vertices. It is worthwhile to note that this criterion only considers deviation from the new mesh to the mesh created in the previous iteration; deviation from the original mesh is not used in the strategy. As these algorithms typically rely on somewhat involved manifold mesh operations, they can be complicated to implement and slow to execute, not being feasible for a parallel implementation.

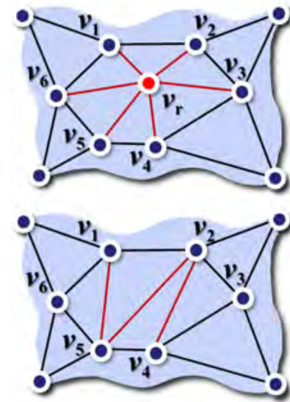


Figure 2.19: Vertex decimation simplification selects unimportant vertices for removal, and then re-triangulates the resulting hole.

2.2.1.2 Edge contraction

Edge contraction strategies consist of an iterative removal of geometric primitives with a local reduction operator chosen according to a local geometric optimality criteria. The *edge-collapse* [HDD+93] takes as a parameter the edge to be collapsed $h = \{i, j\}$. The two vertices $\{i, j\}$ are collapsed in a single vertex v_h, updating all edges that were previously incident to i and j to reference v_h. As a result of the collapse, the triangles sharing the edge degenerate into a segment and are removed (see Figure 2.20). The only computed parameter is the new vertex position, which is usually one of the two old vertices, or a weighted average.

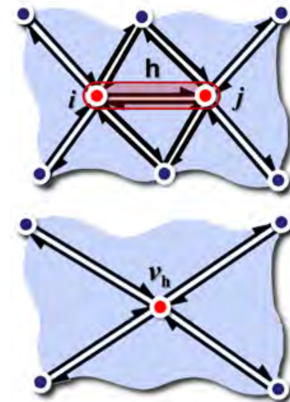


Figure 2.20: Edge-collapse simplification iteratively removes the triangles sharing the collapsed edge.

The selection of the edge that will be collapsed is the key ingredient in edge collapse methods. The process, keps some form of history about the progressively simplified surface to evaluate the current error.
Using cumulative quadric error matrices is particularly efficient and hence a popular choise [GH97]. One of the benefits of iterative contraction is the hierarchical structure created, which naturally leads to a useful multiresolution surface representation [Hop96]. On the other hand, one issue is that they can be hard to parallelize, given the inherently serial nature of the iterative process.

2.2.1.3 Vertex clustering

Vertex clustering [RB93] consists of spatially partitioning the initial vertex set into a set of clusters and unifying all vertices inside a cluster by a single vertex, called *cluster representant* (see Figure 2.21). Vertex clustering often produces relatively poor quality approximations and tends to make alterations to the topology of the original model. The results of this algorithm can be quite sensitive to the resolution and the placement of the grid cells, making it incapable of simplifying features larger than the cell size. Low and Tan [LT97] improve the basic method by using floating cells rather than a fixed grid to define the clusters, and Luebke and Erikson [LE97] alternatively form a hierarchy of clusters. Both methods improve the simplification quality, and can be used to perform view-dependent simplification. Furthermore, the algorithm is based on a linear pass through the source vertices and then a linear pass through the source triangles, so it has a relatively coherent memory access pattern, and interacts well with memory hierarchies, also being feasible for a parallel implementation [Wil11].

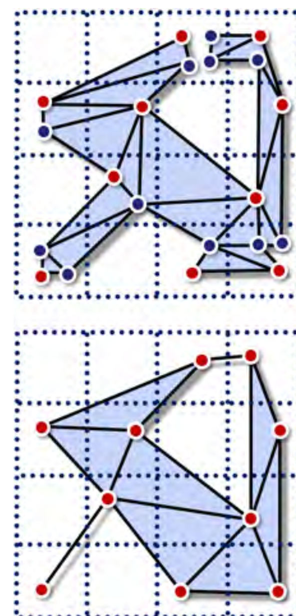


Figure 2.21:
Vertex clustering simplification consists of spatially partitioning the initial vertex set into a set of clusters and unifying all vertices inside a cluster by a single vertex.

2.2.2 Subdivision surfaces

Subdivision surfaces are an alternative level-of-detail strategy which has become a valuable tool in geometric modelling and computer graphics due to their simplicity, efficiency and ease of implementation. The subdivision surface itself is defined as the limit of repeated recursive refinement and smoothing steps. The refinement phase creates new vertices and reconnects them to create new smaller triangles, and the smoothing phase computes new positions for the vertices. The shape of the refined surface is determined by the initial structured mesh of control points and a set of subdivision rules.

The recursive nature of their definition makes subdivision surfaces suitable for many application fields, such as animation. Flexible modelling operators for 2-manifold surfaces are used to construct smooth surfaces and multiresolution representations. Subdivision surfaces are ideal for interactive multiresolution mesh editing, where the overall shape of an object is controlled by a coarse mesh, while details are added by modifying the control points of a refined mesh. Their computational efficiency, along with the compatibility with arbitrary topologies and the support of surface features with complex geometry are its main advantages. In addition to smooth surfaces, the management of boundaries and sharp features presented by some subdivision approaches allows more realistic objects to be represented.

Most subdivision schemes operate on triangular or quadrilateral meshes, where a vertex is said to be ordinary if its valence is six in a triangular mesh or four in a quadrilateral mesh. A mesh (triangular or quadrilateral) is said to be regular if all their vertices are ordinary.

Studies in this field proposed several subdivision schemes. A survey of subdivision surfaces can be found in Zorin and Schröeder [ZS00], here we offer a brief description of a few schemes with their main properties:

2.2.2.1 Loop scheme

The Loop scheme [Loo87] proposes the following rules: a new edge point is computed by $\frac{3E+E'}{4}$ where E is the midpoint of the edge and E' the midpoint of the opposite edge; a new vertex point is computed by $(1-n\beta)V+\beta P$ where V is the old vertex, n is the valence of V, P is the sum of all n neighbors of V and $\beta = \frac{3}{6}$ for $n = 3$, or, $\beta = \frac{1}{n}(\frac{3}{8}+\frac{1}{4}cos * 2\pi/n))^2)$ for $n > 3$. Special rules are needed for boundary points. Loop surfaces are C^2 at ordinary vertices and C^1 at the others. The original scheme (see Figure 2.22) was extended by Hoppe *et al.* [HDD+94] to incorporate sharp creases, darts and corner points. Schweitzer [Sch96] further extended the scheme with conical and cusp points. A fast parallel method for the approximation of Loop subdivision surfaces in real-time is proposed in by Li *et al.* [LRZM11].

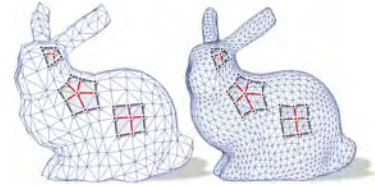


Figure 2.22: Loop subdivision scheme, shown highlighted over ordinary vertices of valence 4, and irregular vertices of valence 3 and 5.

2.2.2.2 Doo-Sabin scheme

The Doo-Sabin scheme [DS98] is conceptually quite simple, since there is only one mask used to compute the new vertices (see Figure 2.23). Let p_i, $i = 0..n$ be the vertices of a face. The new vertex in the corner i is computed by $\sum_j a_j p_j$ where $a_i = \frac{n+5}{4n}$ and $a_i = \frac{3+2cos(2\pi(i-j)/n)}{4n}$ for $j \neq i$. In Peters and Reif [PR98] the C^1 continuity of the limit surface has been proved. A special rule is required only for boundaries, where the limit curve is a quadratic spline. A fast GPU parallel rendering of a Loop subdivision surface by a patch-based tessellation algorithm is presented by Fan and Chen [FC09].

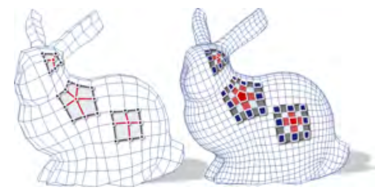


Figure 2.23: Doo-Sabin subdivision scheme, shown highlighted over ordinary vertices of valence 4, and irregular vertices of valence 3 and 5.

2.2.2.3 Catmull-Clark scheme

The Catmull-Clark scheme [CC98] defines the following rules: a new face point is computed as the average of the vertices of the face; a new edge point as the average of the endpoints of the edge and the new face points of the adjacent faces; a new vertex point is computed by $\frac{F+2E+(n-3)V}{n}$ where V is the old vertex, n is the valence of V, F is the average of the new face points of all faces incident to V and E is the average of the midpoints of all edges incident to V. Special rules are needed for boundary points. The scheme produces surfaces that are C^2 continuous everywhere except at extraordinary vertices (see Figure 2.24), where they are C^1 [BS88, PR98]. A patch-based GPU parallel tessellation method for approximate Catmull-Clark surfaces with displacement mapping is proposed by Loop and Schaefer in [LS08a].

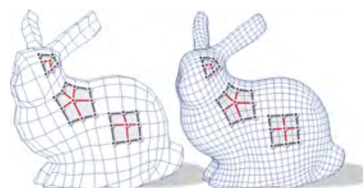


Figure 2.24: Catmull-Clark subdivision scheme, shown highlighted over ordinary vertices of valence 4, and irregular vertices of valence 3 and 5.

2.2.3 Multiresolution level of detail

Multiresolution geometric models [LWC+02] support the representation of geometric entities at different levels-of-details (LODs). Independently of the decomposition scheme adopted, different methodologies have been proposed to handle multiresolution: from a simple collection of versions of an object at different resolutions, to models that maintain relations between consecutive levels-of-detail.

2.2.3.1 Discrete LOD

Discrete LOD (DLOD) methods use a small set of often handcrafted representations and try to select the most appropriate one for a given viewing condition. These sets are small for of two reasons: memory consumption would explode for multiple representations, and the cost of an artist designing many representations would be enormous. Subdivision schemes with integer tessellation factors are another interesting discrete level-of-detail approach (see Figure 2.25).

The small number of representations of DLOD, switching from one representation to another becomes visible and causes popping artifacts [CLE06]. One method to avoid these popping artifacts is *late switching*. Here switching is performed only if the two representations have exactly the same appearance for the current viewing condition. In practice this is quite unfeasible: first of all, we often do not know when two representations will be undistinguishable, because this depends on numerous factors, such as lighting and surrounding objects, which we probable do not know beforehand. And secondly, from the point-of-view of performance we want to switch as early as possible to speed up rendering as much as possible.

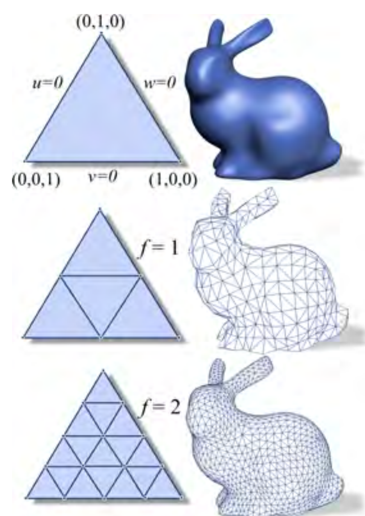


Figure 2.25: Level-of-detail Discrete LOD with the integer tessellation factors $f = 1.0$ and $f = 2.0$.

In order to avoid *popping* artifacts for discrete LODs, *LOD blending* [GW07] combines the different LODs in a single frame. To do this the rendering of the two required LODs in each frame is needed during the transition stage. *LOD interpolation* [SW08] uses the temporal coherence between the two required LODs to interpolate them in different subsequent frames during the LOD transition.

2.2.3.2 Continuous LOD

Continuous LOD (CLOD) methods work by creating an object representation specific for each viewing condition for each frame. Since similar viewing conditions result in similar representations, the change of one representation into another is perceived as smooth [Hop96, HSH10]. Continuous LOD specialized methods for a certain type of applications such as in terrain rendering [LKR+96] have proven effective. One practical way to allow for continuous level-of-detail (CLOD) without visual *popping*, is the fractional tessellation scheme [Mor01]. Fractional tesselation smoothly amplifies the geometric details of a particular mesh by subdividing the polygons of the mesh into smaller polygons to change the level of detail when a floating point tessellation factor per edge is provided (see Figure 2.26).

This CLOD tessellation pattern is integrated for triangle and quad primitives in the GPU hardware stages, as described in Section 2.3.1.

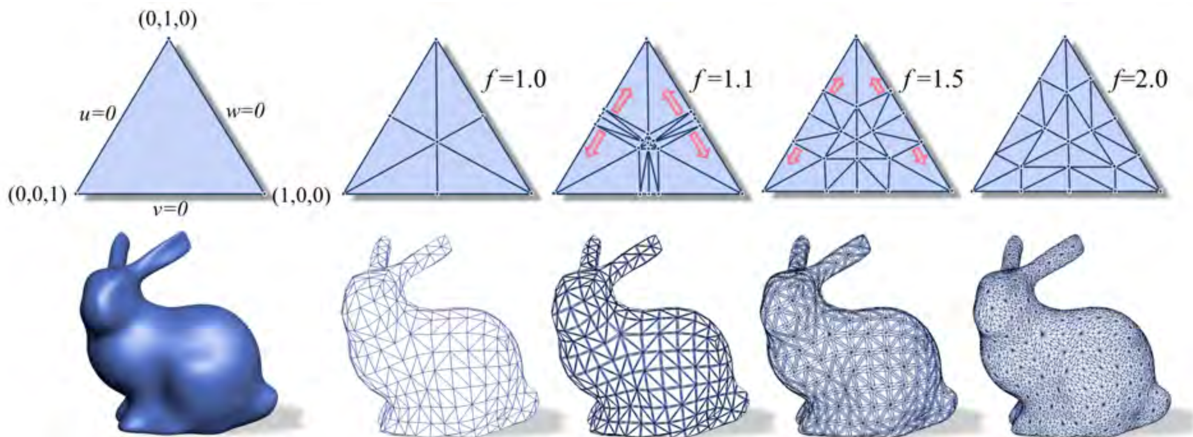


Figure 2.26: Level-of-detail Continuous LOD. Four regular fractional tessellation examples are shown with a common tessellation factor (f) on all edges, from $f = 1.0$ up to $f = 2.0$.

2.2.4 Error metrics

The assessment of the level-of-detail results are performed with either geometric error metrics over the generated surface approximations, or image-based error metrics over the generated screen images, to measure the fidelity of the shape and shading feature preservation. Both strategies are described below:

2.2.4.1 Geometry-based approximation metric

A geometry-based approximation metric allows one to compare the difference between a pair of surfaces. The approximation error between two meshes may be defined as the distance between corresponding sections of the meshes.
When comparing general surfaces, there is no single distinguished direction along which distances can be measured. Instead, we measure distances between closest pairs of points. If we denote the set of all points on the surface of a model M by $P(M)$, the distance from a point v to the model M is defined to be the distance to the closest point on the model:

$$d_v(M) = min_{w \in P(M)} \parallel v - w \parallel \tag{2.9}$$

where $\parallel \cdot \parallel$ is the usual Euclidean vector length operator.
The Hausdorff distance [CRS98] is a commonly used geometric error measure. The Hausdorff error measures the maximum deviation between two models (see Figure 2.27). In practice, it is common to formulate the approximated metric based on sampling the distance d_v at a discrete set of points. Given $P(M_1)$ and $P(M_2)$, we can select two sets $X_1 \subset P(M_1)$ and $X_2 \subset P(M_2)$ containing m_1 and m_2 sample points, respectively. These sets should, at a minimum, contain all the vertices of their respective models. The formulation of the Hausdorff error metric is

$$E_{max}(M_1, M_2) = max\left(max_{v \in X_1} d_v(M_2), max_{v \in X_2} d_v(M_1)\right) \tag{2.10}$$

If $E_{max}(M_1, M_2)$ is bounded by some known threshold ϵ, then we know that every point of the approximation is within a distance ϵ of the original surface and that every point of the original is within the ϵ distance of the approximation, respectively. The measure of the average squared distance between the two models can be defined as follows

$$E_{avg}(M_1, M_2) = \frac{1}{k_1 + k_2}\left(\sum_{v \in X_1} d_v^2(M_2)dv + \sum_{v \in X_2} d_v^2(M_1)dv\right) \tag{2.11}$$

where k_1, k_2 are the surface areas of M_1, M_2.

Usually E_{avg} generally gives a better measurement of the overall fit than E_{max} and is less sensitive to noise, even though it may over discount localized deviations.

Observe that it is not sufficient to simply consider every point on M_1 and find the closest corresponding point on M_2. Closest distances are measured in both directions between M_1 and M_2 due to the differences in results depending on the direction of the computation. The geometric error metric measures the similarity between models independently of the order of the given models.

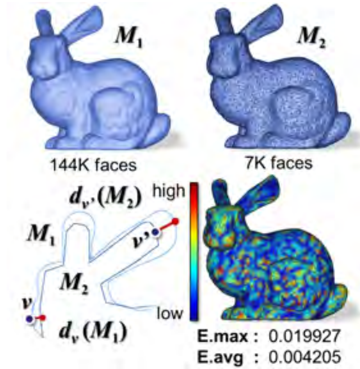


Figure 2.27: Hausdorff distance error metric: (*Top*) A bunny high-resolution mesh M_1 and its simplified mesh M_2 using [GH97]. (*Bottom*) The error is also visualized by coloring mesh M_2 with respect to the evaluated approximation error.

2.2.4.2 Image-based appearence metric

An image-based appearence metric in rendering systems measure the similarity of visual appearance as the ultimate criterion for evaluating the quality of an approximation. The visual appearence of a model M under viewing conditions ξ is determined by the raster image I^{ξ} produced by the renderer. The similarity of appearence can be seen as an image error metric that measures the overall difference between the two images. In that sense, it may say that two models M_1 and M_2 appear identical in view ξ if their corresponding images I_1^{ξ} and I_2^{ξ} are identical. If I_1 and I_2 are both $m \times m$ RGB raster images, the difference between them can be defined as the average sum of the squared differences between all corresponding pixels

$$\parallel I_1 - I_2 \parallel_{img} = \frac{1}{m^2} \sum_u \sum_v \parallel I_1(u,v) - I_2(u,v) \parallel^2 \tag{2.12}$$

where $\parallel I_1(u,v) - I_2(u,v) \parallel$ is the Euclidean length of the difference of the two RGB vectors $I_1(u,v)$ and $I_2(u,v)$. This simple metric makes it possible to measure the visual difference between a detailed input model and its approximation in a certain view. This can be viewed as a measure which is equivalent to human perception. Differential weighting for the color channels, non-linear sensitivity to radiance, and spatial filtering are some factors that can be added to improve the measure. More elaborate metrics for comparing images have been presented in [RT98]. If M_2 is a good approximation of M_1 for the given view ξ, then $\parallel I_1 - I_2 \parallel_{img}$ should be small. Given this image metric, the total difference in appearance between two models can be characterized by integrating $\parallel I_1^{\xi} - I_2^{\xi} \parallel_{img}$ over all possible views ξ.

The main advantage of an appearance-based metric is that it directly measures similarity of appearance, which is what rendering systems are interested in preserving. It also allows occluded details to be discarded. Moreover, these error metrics are useful when some finite set of viewpoints occurs. Unfortunately, in most cases adequate samples of viewpoints are not possible. If an incorrect set of samples is evaluated, significant features can be removed. Furthermore, each sample may involve an expensive rendering step.

2.3 Real-time rendering

The graphics pipeline abstracts the underlying hardware details from the graphics programmer, while providing a restricted programming model that expresses the inherent parallelism in rendering. For example, each vertex of a geometric model can be transformed in parallel and each pixel can be colored independently.

In a simplified form of the pipeline, the graphics programmer writes a shader that determines how one individual vertex should be transformed and another shader that contains instructions on how the color of one pixel should be computed. The underlying hardware then schedules and dispatches all instances of vertex and pixel shader programs.

At a coarse level, the graphics pipeline can be divided into three main stages. The first one is geometry processing, where three-dimensional models are positioned and animated. Additionally, the input geometry can be refined if needed. Rasterization follows next, where the visibility of each primitive for a specific camera position is determined. Finally, in the pixel processing stage, materials are applied to the visible parts of the geometry and the pixel color is computed and written into the resulting image.

2.3.1 Graphics hardware pipeline

The hardware graphics pipeline consists of both fixed-function stages and programmable stages. All programmable stages of the pipeline can access texture images through the GPU's memory system. Figure 2.28 shows a schematic overview of this pipeline. In the first term the geometry data of a given application is received at the graphics processor. Three-dimensional models are represented as collections of triangles, quadrilaterals or higher order primitives, where the latter are parametric surface patches. In addition to geometric primitives, the application uploads shader programs for the programmable stages and buffers holding constant values and a specific rendering state, such as the tessellation method, light positions, and camera parameters. A set of texture images needed for the shader programs are also uploaded to the GPU's memory system. With this information available from the application, below the steps through all the pipeline stages are described.

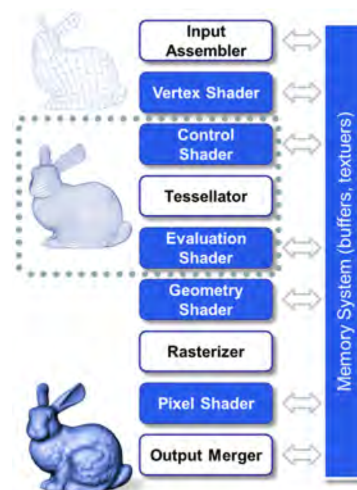


Figure 2.28: Diagram of the hardware rendering pipeline: Showing in *white* the fixed pipeline stages and in *blue* the programmable stages.

Input Assembler: The first stage of the pipeline reads arrays containing vertex data, such as arrays of positions, normal vectors and texture coordinates, and assembles vertex structures from the individual arrays. The members of the vertex structure are called *vertex attributes*. Often, a separate index list with offsets into the vertex arrays is used for more compact storage, and to avoid transforming the same vertex more than once.

Vertex Shader: The vertex shader is a user-provided program executed once for each vertex in the assembled vertex array. The program outputs a transformed position and all attributes needed by shader programs later in the pipeline. A minimal vertex shader takes a three-dimensional position, expressed in a coordinate system local to the geometric object, and multiplies it with a matrix that transforms the position into the *camera clip space*. This is a coordinate system independent of the scene scale, convenient for clipping operations against the camera's view volume. If tessellation stages are disabled, the vertices are sent directly to the rasterizer.

Control Shader: The following three pipeline stages interact to support flexible geometry amplification. These are the control shader, a fixed-function tessellator, and the evaluation shader. The control shader interprets the transformed positions from the vertex shader as control points for a parametric patch.

They are evaluated and tessellated into a large set of small triangles in the two downstream tessellation stages. The control shader is applied to the control points of an input patch, and has two tasks. The user-provided program is executed per control point of the input patch, firstly to typically change the patch's control point basis, and secondly, to compute the tessellation factors of the edges of the patch.

Tessellator: This is a fixed-function unit that, given a patch, tessellation factors from the preceding control shader and state parameters, generates a set of barycentric coordinates and connectivity in a planar domain inside the patch. In modern GPU pipelines, the tessellator works on triangular and quad patches and has a set of modes, including uniform and fractional tessellation. The barycentric positions are fed into the next pipeline stage.

Evaluation Shader: This program takes the role of the vertex shader when tessellation is enabled and transforms the geometric positions into the camera clip space. It is executed once for each generated barycentric position from the tessellator and outputs a displaced vertex. The shader has access to the transformed control points from the hull shader, and typically evaluates a parametric surface, such as a bi-degree Bézier surface, at a certain parametric coordinate. Additionally, the shader may use texture images to add local surface detail.

The transformed vertices are then passed on to the rasterizer. Recent research has shown that the tessellation stages can be combined to represent advanced surface representations, such as animated approximate subdivision surfaces [LS08b, LSNCn09].

Rasterizer: The first task of the rasterizer is to set up triangles from consecutive indices in the vertex arrays. After triangle setup, the next task is to determine which pixels on screen are covered by each triangle. At this point in the pipeline, the triangles have been transformed by the vertex or evaluation shader into the camera clip space. The rasterization stage handles clipping of primitives to the view volume and performs culling operations.

After clipping and culling, the clip space coordinates are projected on screen and visibility is determined. There are several traversal strategies for the visibility test. One simple approach is, for each triangle, to compute a bounding box on screen and for each screen space visibility sample within this bounding box, test if the sample is covered by the triangle. This process is called scan conversion or rasterization.

If a sample is covered, a fragment is generated. For each covered fragment, the vertex attributes are interpolated using the triangle's barycentric coordinates at the sample's hit point. The interpolated vertex attributes, including the depth at the hit point, are stored in the fragments. The fragments are then sent to the next pipeline stage, the pixel shader. In most graphics hardware pipelines to date, the rasterizer is a fixed-function unit.

Pixel Shader: For each fragment generated by the rasterizer, the final color is computed by a user-defined program, representing a surface material that takes the fragment's interpolated attributes as input. These pixel shader programs often contain several texture map lookups and advanced material descriptions.

The pixel shader is followed by a depth test, and if the pixel shader is guaranteed not to modify the depth interpolated in the rasterizer, this test can sometimes be moved before the pixel shader. The depth values of the currently processed sample are compared to the value stored in a depth buffer. The depth test is configurable, and a commonly used test is 'less than': only if the current fragment's depth is closer than the depth buffer value, the current color value is written to that sample position in the frame buffer, and the corresponding entry in the depth buffer is updated.

Output Merger: In the final stage of the GPU pipeline, shaded fragments are blended into the frame buffer, taking color and transparency of each fragment into account. Once the entire scene has been fully processed, the resulting color buffer is displayed on screen. The output merger is typically a fixed-function stage, with a user-configurable blending mode.

2.3.2 Graphics processors and parallel programming

2.3.2.1 Evolution to many-core devices

Graphics processing units (GPU) have evolved by outperforming CPUs' arithmetic throughput and memory bandwidth. They are designed such that more transistors are devoted to data processing (ALUs) rather than data caching (memory caches) and flow control as shown in Figures 2.29 and 2.30. The introduction of programmable shader stages started to make them an ideal processor to accelerate a variety of data parallel applications beyond computer graphics.

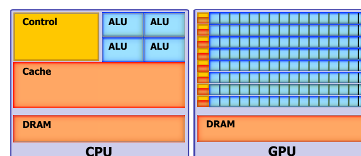


Figure 2.29: Diagrams of a CPU and GPU architectures. GPUs are designed with more transistors devoted to data processing (ALUs) rather than data caching (memory caches).

The GPU is especially well-suited to address problems that can be expressed as data-parallel computation (the same program is executed on many data elements in parallel) with high arithmetic intensity (the ratio of arithmetic operation to memory operations). This architecture was designed for image rendering (3D) and processing (video playback) but data-parallel processing can be also found in physics simulation, signal processing, computational finance or biology. An algorithm that is data-parallel is also referred to as *embarrassingly parallel*. Those algorithms can be accelerated substantially by using GPU.

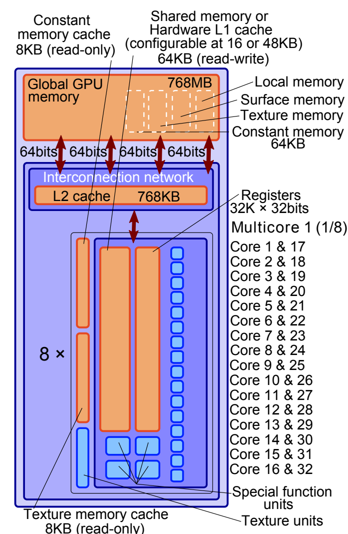


Figure 2.30: Diagram of Fermi GPU architecture [Nvi11].

2.3.2.2 Organizing data movement

GPU algorithms are typically computation-bound or memory-bound. When the execution time of the algorithm depends mainly on the access speed of the memory, it means that they are memory-bound and they can be made faster by reducing the memory footprint of the data they work on.

Organizing data movement is an important design concept for data structures. When emphasizing compute throughput on SIMD programmimg models, the cost of data movement starts to dominate the processing cost and therefore data movement must be restricted by keeping data locally as much as possible. In particular, the process of collecting data to build the data structure defines algorithmic patterns that are typically memory-bound due to the large, and sometimes incoherent memory accesses, also often including synchronization barriers between the parallel threads accessing memory.

2.3.2.3 Global memory access patterns

Global memory performance is affected by *coalescing*, which means that, for a SIMD instruction that accesses global memory, the individual accesses for each thread can be combined together by the memory subsystem into a single memory transaction if every reference falls within the same contiguous global memory segment. The performance discrepancies between coalesced and non-coalesced accesses can be as large as an order of magnitude. Bus transactions are on the order of 128 bytes, making it particularly wasteful if each thread induces a separate transaction for a single 4-byte memory reference.

2.3.2.4 GPU memory hierarchy

In the beginning, processors started using a single level of cache, but as their speed increased, two to three levels of cache hierarchies were introduced to span the growing speed gap between processor and memory. In these hierarchies, the lowest-level caches are small but fast enough to match the processor's needs in terms of high bandwidth and low latency, while higher levels of the cache hierarchy are optimized for size and speed.

The GPU PRAM memory hierarchy creates a branch in a modern computer's CPU memory hierarchy. The GPU, just like a CPU, has its own caches and registers to accelerate data access during computation (see Figure 2.31). GPUs also have their own main memory with its own address space, a fact that means programmers must explicitly copy data from CPU to GPU memory. This transfer is often a bottleneck in some GPU applications where there is a high dependence between the sequential computations performed on the CPU and the next processing to be handled on the GPU, especially in applications with real-time interactive requirements.

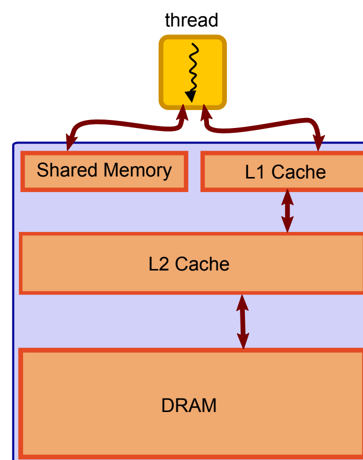


Figure 2.31: Diagram of Fermi GPU memory hierarchy.

Memory accesses are among the slowest operations of a processor. According to the Moore's law [Sch97] instruction performance has been increasing at a much greater rate than memory performance has increased. This difference in performance increase rate means that memory operations have been getting expensive compared to simple register-to-register instructions. Modern CPUs integrate large caches in order to reduce the overhead of these expensive memory accesses. GPUs use another strategy so as to cope with this issue. Massive parallelism can 'feed' the GPU with enough computational operations while waiting for pending memory operations to finish. This different execution strategy requires new implementation ideas to be sought.

On-chip memory level: a big difference between CPU and GPU is the memory hierarchy. Registers and shared memory are extremely fast while global memory is several hundred times slower. It is worth noting that the shared memory is not a hardware cache, but is a scratchpad memory [Wik04]. Each streaming multiprocessor (SM) has one of these local high-speed internal memories. It can be seen as a L1 cache, but there are crucial differences: explicit instructions are required to move data from global memory to shared memory and there is no coherency among scratchpads and global memory. When used as a cache memory, if global memory changes, scratchpad's content will not be updated. Shared memory is considered to be as fast as register memory as long as there are no bank conflicts among threads.

Unified cache memory level: more recent GPU architectures, such as the *Fermi* [Nvi11] architecture, introduce a fully coherent L2 cache common to all SMs (referred to as *unified cache*). This new cache has the same purpose as the CPU L2 cache and is a significant change from previous architectures. This key feature shows a real general purpose orientation of GPUs, i.e., the direction they are going to take in the near future. This cache does not explicitly accelerate graphic computations.

Global GPU memory (DRAM) is linked to the GPU chip through a very large data path. In real programs, the peak global memory bandwidth is difficult to obtain since memory has to be aligned and thread accesses have to be coalesced. Coalescing allows all independent thread memory access to be merged into one big access. Several thread access patterns are recognized by coalescing hardware. It also means there is severe bandwidth degradation for stride access patterns with large offsets.

2.3.2.5 Stream programming concepts

GPU programming models conceal the number of actual processors from the user. The programmer specifies only a kernel or shader program and a data stream over which the program is to be executed. The GPU then maps this data onto the available processors to compute the result. Simple examples include programs that apply the same affine transformation to every input point. However, the user cannot query which processor was used, nor is it recommended that explicit inter-processor communication be performed. Current unified computing architecture GPUs execute batches of threads in single-instruction, multiple-data (SIMD) style. The size of a batch of threads varies from tens to hundreds depending on the architecture, and branch performance is highly dependent on the coherency of the compute operations performed within a batch.

Compute Unified Device Architecture (CUDA) is a hardware architecture coupled with an extension of the C language for GPGPU computing. The programming model is tightly coupled with the architecture of NVidia graphic cards. On the other hand, *Open Computing Language* (OpenCL) [Khr08] is programming language for parallel computing compatible across heterogeneous platforms consisting of central processing unit (CPUs), graphics processing unit (GPUs), and other processors including CUDA devices.

The execution of a CUDA kernel program is performed through three general steps. In the first step, the required input data is allocated and loaded in GPU global memory. The second step is setting up the kernel call, specifing the threads organization, and finally executing the kernel program on the GPU.

The kernel call is defined by the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel by a grid of blocks of threads, and each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the block.

CPU thread vs GPU thread: Despite using the same name, the word thread has a different definition on the CPU than on GPU, and can lead to misunderstandings. Unlike in CPU, GPU threads are managed by hardware. Classic thread programming techniques do not match GPU thread design. In CUDA threads should not diverge for optimal performances. Divergent threads are not impossible to implement, but they can dramatically lower the performances.

Every GPU thread in the same warp executes the same instruction in lockstep, although all threads can branch separately. However, this would lead to extremely bad performances, even on the newer architectures.

The multiprocessor (SM) creates, manages, schedules, and executes threads in groups of 32 parallel threads called *warps*. Individual threads composing a warp start together at the same program address, but in lastest architectures [Nvi11] they have their own instruction address counter and register state, and are therefore free to branch and execute independently. Warp size is fixed by hardware and is 32 for recent GPU architectures, though this size might change in future architectures. As the basic unit of execution flow in a multiprocessor is a warp of 32 thread, it is useless to execute less that 32 threads in a block.

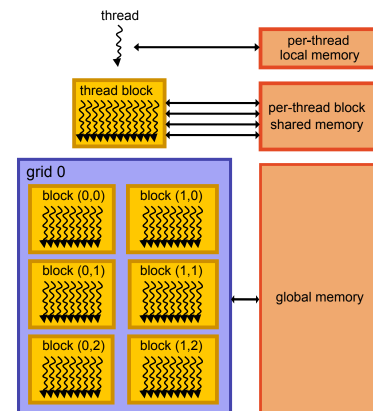


Figure 2.32: Diagram of GPU threads organization. Threads communication within the same block is performed by reading and writing data in shared memory, while between all the threads of the grid it must be performed through the DRAM global memory, which is considerably slower.

Communication between threads is achieved by reading and writing data to various shared memory spaces. The machine model exposes three levels of explicitly managed storage that vary in terms of visibility and latency. These levels are: per-thread registers and local memory, shared memory local to a collection of warps running on the same processor core, and finally a large global memory in off-chip DRAM that is accessible by all threads (see Figure 2.32). Threads must explicitly move data from one memory space to another.

Thread divergence: On previous GPU architectures [Nvi08] each conditional branch was serialized. Accordingly to [WPSAM10], the '`else clause`' is always executed first while other clauses are disabled, then the '`if clause`' is executed (and the '`else clause`' disabled). However, in newer architectures, such as the Fermi architecture, this issue has been improved because each multiprocessor features a *dual warp scheduler*. Each group of 16 cores can execute a different conditional branch. Our divergent thread example would be executed in parallel on the Fermi architecture only. Of course, it works for half-warps only.

The CUDA compiler allocates registers memory to threads. If the threads require too many registers, local memory is used. Actually, local memory does not exist in the hardware itself. Local memory is the name given to some global memory which was made private to a thread. This memory is extremely slow compared to register or shared memory, thus exceeding the maximum register memory leads to significantly slower performance.

For situations where a *race condition* (process whereby an output of the process is unexpectedly and critically dependent on the sequence or timing of other events) is difficult or impossible to avoid, atomic operations can be performed on both shared and global memory. Atomic operations perform a series of actions that cannot be interrupted; examples include incrementing a counter and conditionally setting a memory location based on its current value.

These operations are extremely helpful for threads in different thread blocks communicating with each other. However, they are costlier than a normal memory access, especially when many threads are performing the same atomic operation on the same memory location.

2.4 Detail mapping data structures

Collecting and querying spatial data of different sparsity and density distributions while preserving the interactive constraints is one of the main interests in this thesis. In this context, there are already different approaches that allow the organization of spatial data either in a regular grid, a parameterized domain, a spatial subdivision or hash table data structure. One example is virtual grids, where each grid cell defines a specialized description of the local information contained onto it, behaving as a *spatial directory*. This section provides a short description of different methodologies that provide representations for shape and shading detail data.

Note that the values stored in the base data structures could represent indices into another structure, allowing a spatial query coordinates (or key) to reference more data, as illustrated in the spatial directory descriptions of this section.

Arguably, one would be willing to incur more time on such operations as opposed to the much more frequent lookup operations, required in many rendering applications.

Here we consider data structures that may take more time in the collecting process, in order to place the spatial elements more closely together, to achieve a compact storage, to maximize the coherency between neighboring spatial data, and to minimize the variable work per retrieval.

The majority of state-of-the-art work on GPU spatial data structures constructs them on the CPU as a pre-processing step, then they are later used on the GPU to accelerate rendering applications. However, more recent work has already focused on producing these data structures directly on the GPU, using parallel construction algorithms.

2.4.1 Irregular spatial data organization

Defining an indexing of surface and volume spatial data is an important strategy to create coherence in detail mapping data structures, where memory access patterns are critical for performance. Data layout greatly influences memory access patterns, and therefore it has a large impact on overall program performance. Different strategies can be exploited to create coherence in spatial indexed data.

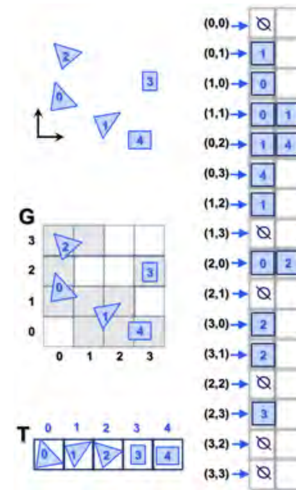


Figure 2.33: Data grid embedding partitions the spatial data into regular cells that can lead to a g reater locally allowing more efficient query operations.

Data grid embedding allows spatial data to be organized in regular partitions of the space, like cubically shaped cells, where each cell contains references to the elements that overlap it [RBW04, GPSSK07]. The greater locality of this space partitioning allows more efficient query operations over the data, not requiring all the elements of the spatial domain to be checked when asking for a specific point, but only the elements attached to the lists in the cells with the interest point. This greatly reduces the number of evaluation tests (see Figure 2.33).

2.4.2 Spatial addressing

A great amount of graphics algorithms rely on spatial proximity. Therefore techniques trying to partially preserve the proximity between neighboring data elements, from their multi-dimensional space, with respect to their location inside the data structure, it can improve the performance thanks to better coherency in the memory access patterns, i.e. two cells that are close in space are likely to be close in the total order.

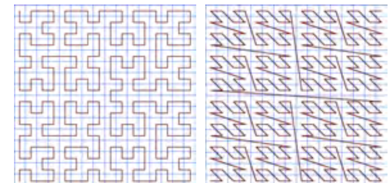


Figure 2.34: Space-filling curves allow to define spatial ordering of data by mapping multi-dimensional elements to a one-dimensional indexing.

Space-filling curves [Sag94] allow the definition of a spatial ordering of data by mapping multi-dimensional elements to a one-dimensional indexing (see Figure 2.34). The maps enumerate each grid location in space to a one-dimensional location key following the spatial curve (e.g. Peano-Hilbert curve, Lebesgue/z-ordering, gray-ordering, etc.)

2.4.3 Spatial data memory layout

In this section we describe two strategies to enhance the spatial locality of data structure memory data layouts.

Alignment: alignment of large indexed spatial datasets, such as a list of 3D point coordinates, is important [SDZ+11]. The elements can be stored in the structure-of-arrays (SoA) or the array-of-structures (AoS) formats. In SoA, the individual components are stored contiguously for all rows. AoS reverses this by storing all components for a single row in order (see Figure 2.35). AoS is the more natural format for an object-oriented method and is well-suited for cache-based or pipelined processors. However, SoA is more suited for vector processing (e.g., SSE and CUDA SIMT). The SoA format is expected to be more efficient for parallel architectures due to their coalesced global memory access (see Section 2.3.2.3) and reduced (or removed) shared memory bank conflicts.

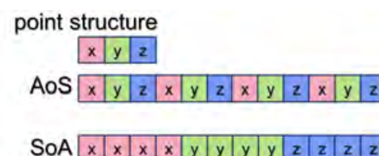


Figure 2.35: Alignment as Structure-of-Array (SoA) is more suited for vector processing (e.g. SSE and CUDA SIMT) due to coalesced global memory access and reduced (or removed) shared memory bank conflicts.

Blocking: blocking aims to align the spatial object data in consistent regularly-sized blocks, taking the data with certain locality in the spatial domain and putting them together in a single block (see Figure 2.36). In this way, when a block is brought to memory, it allows good use of the rest of the block when similar operations are performed at the same time on neighboring elements. The block size should be aligned to a power of two, in order to provide a more coherent blocking, aligned with memory transaction transfer page sizes [CNLE09, AVS+11].

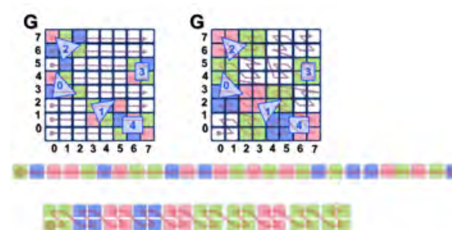


Figure 2.36: Blocking neighbor data on regular-sized blocks (e.g. 2×2) allows efficient access to the rest of the block when similar operations are performed at the same time on neighboring elements.

2.4.4 Spatial data and texture mapping

In many interactive visualizations triangle meshes require colors computed on any point of the model, which are typically calculated from an image, called texture. Texture maps are used in many places in the graphics pipeline. As seen in Figure 2.28, all programmable pipeline stages can access texture maps.

The textures are stored as 2D and 3D grids of color samples, but there is no obvious way of automatically mapping a point p on a three-dimensional surface to a point texture space. Each triangle vertex contains a set of attributes attached, containing, for example, a normal vector, color values, and texture coordinates. For each fragment generated by the rasterizer, texture coordinates specified at each vertex are interpolated to give a unique position within the texture image. The interpolated texture coordinates are used to access a small set of texels in a texture image stored in memory.

For each projected point from an object's surface to a pixel on the screen, many applications use textured surfaces to determine the color of the pixel. However, a pixel on the screen may correspond to a large area of the texture for distant objects, which necessitates filtering to avoid aliasing.

Depending on how distant an object is, a filter may integrate over too many texels of the texture. Hence, performing too many evaluations and high memory bandwidth for the final contribution of the distant object in the screen.

Rather than computing exact filter integrals, GPUs use precalculated downsamplings of the texture that are stored in a MIPmap [Wil83], which avoids aliasing of distant texture details and achieves a better performance balance, by doing less evaluations requiring a lower memory bandwidth as well.

Mipmapping: mipmapping [Wil83] is a classic technique for improving the performance and quality of texture filtering for real-time rendering. The goal of mipmapping is to accelerate the calculation of images downsampled to arbitrary scales by interpolating between precomputed power-of-two scalings of an image (see Figure 2.37).

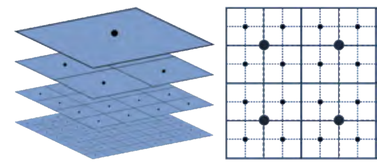


Figure 2.37: A MIPmap can be visualized as a stack of overlaid images as shown on the left. The alignment between neighboring resolutions is shown on the right.

Mipmapping has had hardware implementation since the first texturing hardware, and the original description generated downsampled images using a box filter, though other higher-order filters can be used at each level [Bur81]. The total storage takes only 4/3 of the original image's space.

Filtering quality: Mipmapping works well for sampling high-frequency textures. However, despite of improving quality by reducing texture aliasing, it may not properly sample triangles with oblique projections onto the screen. Therefore, mipmapping on certain surfaces makes them look blurry, and anisotropic filtering [GH86, MPFJ99] with an appropiate level of anisotropy should be used (see Figure 2.38). Heckbert [Hec89] described the problem of filtering a warped image in its full generality. Most subsequent works [MPFJ99, LWW06] have been on how to sample images with affine transformations of filters.

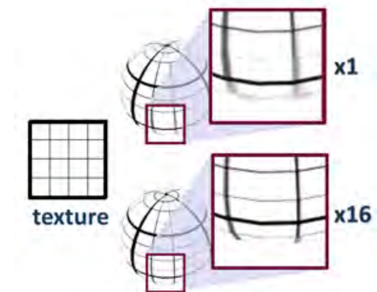


Figure 2.38: (*Top*) Mipmapping on certain surfaces can make them look blurry, because of grazing viewing directions or highly warped texture coordinates. (*Bottom*) Anisotropic hardware filtering with an appropiate anisotropy level should be used.

Texture memory bandwidth: Texture bandwidth is consumed any time a texture fetch request goes out to memory. Although modern GPUs have texture caches designed to minimize extraneous memory requests, they obviously still occur and consume a fair amount of memory bandwidth.

Mipmapping should be used on any textured surface that may be minified, because it improves texture cache utilization by localizing texture-memory access patterns for minified textures.

2.4.5 Parallel spatial query access patterns

Minimizing variable work per retrieval when several elements are being looked up in parallel at the same time is an important feature. For instance, when different threads of a block query neighbor cells of a spatial directory in parallel, and their lists have largely different lengths to be evaluated, it results in a variable number of accesses per retrieval in each thread. This usually results in all threads having to wait for the thread that performs the most accesses to retrieve its data [ASA+09]. The spatial directory structure must minimize this variability, providing similar coherent memory accesses over neighbor data elements from the spatial domain (see Figure 2.39).

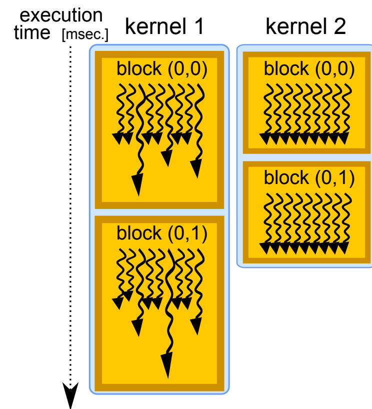


Figure 2.39: Minimizing variable work per retrieval when several elements are being looked up in parallel at the same time results in all threads having less idle time within a block.

2.4.6 Linear data structures

Linear data structures such as sorted arrays, stacks and queues contain the stored elements arranged linearly in a sequence. Parallel algorithms for linear data structures are a popular focus of research, and they are powerful building blocks of spatial data structures. Here we will describe some useful parallel techniques over linear data structures to define efficient spatial directories.

Prefix-sum: a *prefix-sum* (also known as a so-called *scan*) is parallel method which is fundamental for list-processing primitive for computing recurrence relations [HSO07, HG11]. It takes a binary associative operator $\oplus$ with identity I, and an array of n elements, where concurrent threads can cooperatively compute scatter offsets for writing data into shared structures. It takes as input

$$[a_0, a_1, \cdots, a_{n-1}],$$

and returns the array

$$[I, a_0, (a_0 \oplus a_1), \cdots, (a_0 \oplus a_1 \oplus \cdots \oplus a_{n-2})].$$

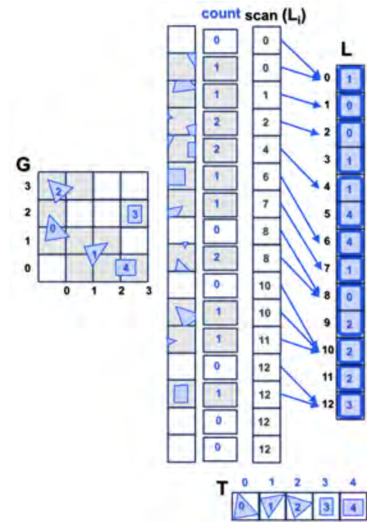


Figure 2.40: Parallel prefix-sum is a fundamental list-processing primitive for computing recurrence relations where concurrent threads can cooperatively compute scatter offsets for writing data into shared

For example, if $\oplus$ is addition, and given a *count* array, where each record counts the number of elements overlapping the corresponding cell of the grid G:

$$[0, 1, 1, 2, 2, 1, 1, 0, 2, 0, 1, 1, 0, 1, 0, 0],$$

then the *prefix-sum* operation on the *count* array would return the output array *scan* (see Figure 2.40):

$$[0, 0, 1, 2, 4, 6, 7, 8, 8, 10, 10, 11, 12, 12, 12, 12].$$

The last value of the *scan* result array gives the size of the total count of elements on the cells of G. Therefore, it allows the array L to be initialized with the required size, and index all the elements of each cell as local contiguous lists with the elements' IDs pointing to the elements' array T. There are many uses for scan, including parallel sorting, stream compaction and tree-based data structures for spatial data [Ble90].

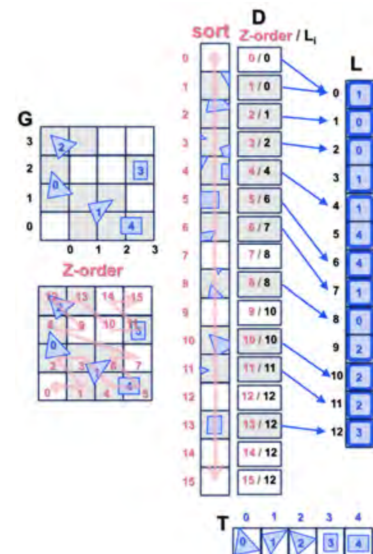


Figure 2.41: Parallel sorting methods allow the spatial data to be reordered, bringing indexed elements into an ordered sequence. Then, a parallel binary search can query a subset of the elements.

Sorting: Sorting parallel methods allow the reordering of spatial data to bring the elements with similar properties together. However, objects should first be organized by a 1D spatial index query, created, for instance with space-filling curve coordinates (see Section 2.4.1). In parallel sorting, most works have focused on producing faster and faster GPU implementations of the radix sort algorithm, which is highly parallelizable. Parallel radix sorting is currently the fastest approach for sorting elements based on they query coordinates – so-called *keys* – on both CPU and GPU processors [SHG09, MG11].

Sorted arrays can be used as an alternative strategy to define an ordered spatial directory D, by the z-order query coordinates. This allows the spatial data elements of cells organized in small lists L to be locally addressed (see Figure 2.41). The ordered spatial directory can be constructed in parallel at extremely fast rates [MG11].

After collecting the spatial data in an ordered sequence, a parallel binary search [HB10] is required to query a subset of the elements in the sequence. Retrieval timings can be highly varied. Therefore, if the queries are sorted, the branching patterns of the binary search, and memory reads will tend to be coalesced into fewer memory transactions, reducing the cost of parallel queries significantly. However, searching random elements in the array incurs as many as $O(lgN)$ probes in the worst case, which requires many more probes than hash tables (see Section 2.4.9).

Stream compaction: stream compaction parallel methods prove to be effective in case of collecting data with a sparse distribution over the spatial domain. This is particularly useful with arrays that have some elements that are interesting, but also many uninteresting ones. Stream compaction produces a smaller array with only the interesting elements (e.g. non-empty records of the array). More formally, stream compaction takes an input array G and a predicate p, and outputs only those elements in G for which $p(G)$ is true (see Figure 2.43), preserving the ordering of the input elements [Hor05]. With this smaller array D, computation is more efficient because it only requires the interesting elements to be evaluated, the transfer costs are greatly reduced (e.g. between the GPU and CPU), and it allows more coherent and efficient memory access patterns.

Stream compaction requires two steps, scan and scatter:

1. The first step generates a temporary vector A where the elements that pass the predicate p are set to 1 and the other elements are set to 0. Then, a scan is performed over A. For each element that passes the predicate, the result of the scan contains the destination address for that element in another temporary vector S.

2. The second step scatters the input elements to the output vector D using the addresses generated by the scan.

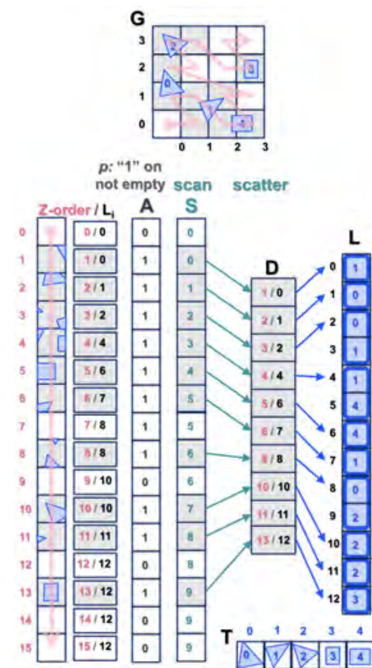


Figure 2.43: Stream compaction parallel methods collect the elements of interest from an array A, without the empty records of the domain grid G, into a compacted array D.

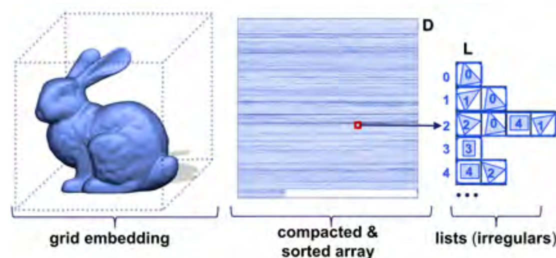


Figure 2.42: Spatial data elements can be embedded in a grid, organized by a 1D spatial index over the grid domain, and processed by parallel radix sorting and stream compaction, leaving only an ordered linear sequence with the elements of interest. This collecting operations allows efficient and coherent parallel binary search operations on the spatial data.

In summary, linear data structures generated with parallel sorting and stream compaction allow the construction of parallel efficient spatial directory, as shown in Figure 2.42, with interesting properties (see Table 2.1). Furthermore, they represent a procedural step during the construction of a parallel hierarchical data-structures, such as octrees [ZGHG11], KD-trees [ZHWG08], and bounding volume hierarchies [LGS^{+}].

Linear data structures main properties:
Space-filling curve + sorting + stream compaction
√ Convert nD to 1D spatial coherence
√ Compact space overhead (pointerless structure)
√ Easy to update, but not optimal
√ Query as parallel binary search
✗ Require a full-reconstruction on updates
✗ Variable work per retrieval specially if searching random elements

Table 2.1: Linear data structures tradeoffs.

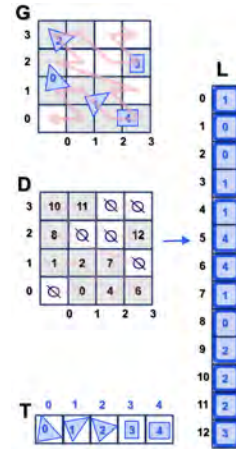


Figure 2.44: A uniform grid can be used to reference spatial elements that are known a priori to be uniformly distributed over the domain.

2.4.7 Grid-based data structures

A uniform grid was one the first proposed spatial data structures [BF79] to achieve fast query evaluations in localized range searches in regular locations (see Figure 2.44). Building a grid can be done in linear time, as opposed to other hierarchical data structures that require super-linear time. Hence, for dynamic data changing every frame, a shorter build time may compensate for a longer rendering time. Therefore, a grid can result in a shorter total time (including build and query evaluation) than other superior acceleration data structures. Building a grid is a memory-bound algorthim. Therefore, reducing the memory footprint of a grid can result in shorter build times.

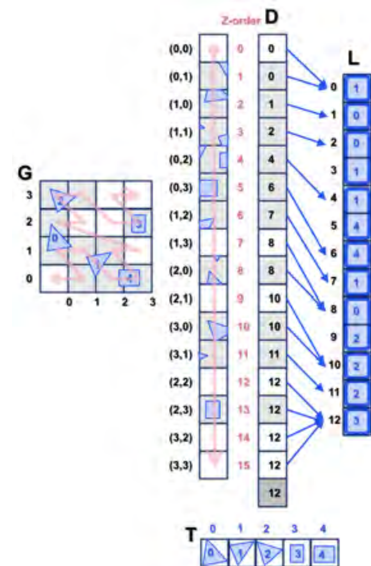


Figure 2.45: A compact grid data structure represents a grid with minimal memory requirements: one index per grid cell and exactly one index per element reference.

Compact grid: the compact grid [LD08] is a data structure that represents a grid with minimal memory requirements: more specifically, exactly one index per grid cell and exactly one index per element reference. It can be shown that the algorithm for building the data structure requires linear time.
The number of cells M should be linear in the number of elements N [BF79] or $M = \rho N$ where ρ is called the grid density.
Inserting the elements into a grid means that all the *cell/element* overlaps have to be determined on the grid domain G (see Figure 2.45). This can be done using the bounding box of the element, or with more accurate element / cell overlap tests. Using the bounding box results in shorter build times and longer rendering times, because the element is also added to cells that overlap with its bounding box only, but not the element. More accurate object cell overlap tests result in longer build times and shorter query evaluation times.

The compact grid data structure for representing a grid consists of two static arrays D and L. For each cell the array D stores the offset of the corresponding element list in L. The array L consists of the concatenation of all element lists. This data structure is static; elements cannot be inserted nor removed.

The size of the element list associated to the cell with 1D index i is given by $D[i + 1] - D[i]$. Note that this expression is invalid for the last object list, since $D[N]$ does not exist. In order to avoid an explicit check for this special case, the array D is extended by one more position.

A regular grid can be efficiently used as a spatial directory (see Figure 2.46) when the elements are known a priori to be uniformly distributed over the domain (see Table 2.2). When it is not known whether this is the case, one would probably use another method to compact the directory even further, using hashing strategies (see Section 2.4.9).

Lagae *et al.* [LD08] suggest a parallel construction of the compact grid combining the compact grid representation with the scalable sort-middle grid build method proposed by Ize *et al.* [IWR+06]. However, construction does not scale linearly with respect to the number of threads, there being a problem for dynamic large spatial data that requires extremely fast reconstructions at each frame.

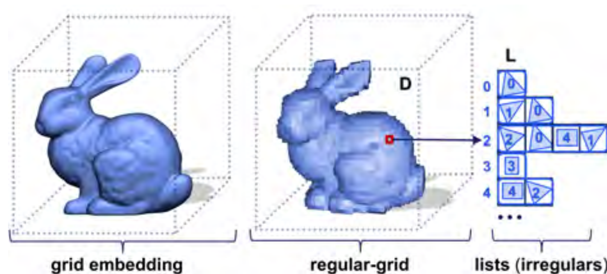


Figure 2.46: A regular grid used as an efficient spatial directory when the elements are known a priori to be uniformly distributed over the domain.

Regular grids main properties:
Grid embedding + lists
√ Block transfers
√ Good cache alignment and behavior
✗ Empty wasted space for sparse data
✗ Non-adaptive to irregular data distributions

Table 2.2: Regular grids tradeoffs.

2.4.7.1 Parameterized grid data structures

Another way to collect and retrieve sparse spatial data is by *mesh parameterization*. Mesh parameterization was introduced in computer graphics to find a one-to-one mapping from a suitable parameter domain to the given surface. For instance, a planar mapping is used to flatten and regularly sample the given surface into an image for texture mapping purposes (see Figure 2.47).

On the whole, texture mapping is a powerful technique that adds flexibility and control when designing spatial data structures to be evaluated in a shader. However, the gap between computing power and memory bandwidth is increasing for standard CPUs, and the compute versus memory access ratio is even higher for dedicated graphics hardware [Joh05]. Hence, it is critical to reduce the memory bandwidth on the construction and query operations over the data structure as much as possible.

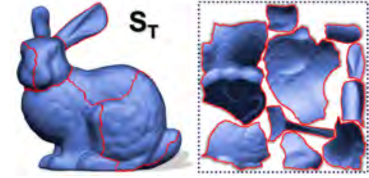


Figure 2.47: Mesh planar parameterization [ZSGS04], and resampled shape and shading data in a texture atlas.

Planar mapping: Planar parameterizations [GGH02a, LPRM02, ZSGS04, SLMB05] map a triangle mesh to the planar domain, which then allow the sampling of the shape and shading information over the regular structure of an image.

While some surfaces have a natural planar parameterization, on many others the challenge is to flatten the surface while keeping a low distortion, which often requires the surface to be cut into independently parameterized *charts*. These charts are later packed into a single image also called a *texture atlas* (see Figure 2.47).

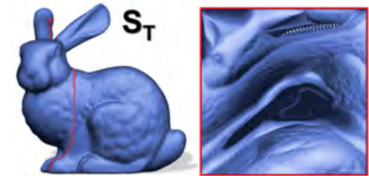


Figure 2.48: Mesh geometry image parameterization [GGH02a] flattening the surface in a single fixed-boundary chart.

- **Geometry image** parameterization [GGH02a] maps the triangle mesh S_T into a single chart with the boundary fixed to the square image (see Figure 2.48). As a result, it does not have to store any explicit connectivity anymore between the samples, as the connectivity is implicit in the image itself. During the mapping process, the surface is cut introducing a set of *seams*.

  Thanks to the cuts, the parameterization unfolds the original surface into a square domain, and then resamples the geometry on the image grid data structure. This regular layout allows more efficient parallel processing, acting as a spatial directory D either to access the irregular parameterized data by cell lists L (see Figure 2.49 (*Bottom*)), or through a regular quad-based remeshing S_Q of the original irregular structure (see Figure 2.49 (*Top*)). The constant memory access queries to the cells do not require an expensive hierarchical traversals (opposed to other data structures presented in Section 2.4.8), and proves to be a very suitable data structure for uniformly distributed spatial data.

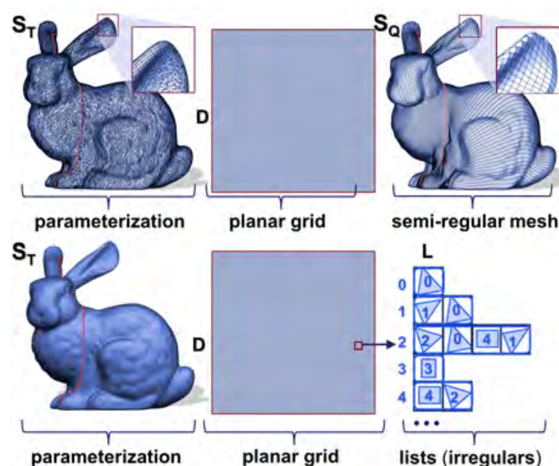


Figure 2.49: A geometry image parameterization as a spatial directory. (*Top*) Allows a regular remeshing of the original irregular structure, or (*Bottom*) to access the irregular parameterized original data by localized cell lists.

Planar parameterizations work fairly well. However, some applications are quite sensitive to the parametric distortions and discontinuities introduced by the seams that appear when cutting non-genus zero surfaces onto the planar domain. These methods cannot avoid the discontinuities generated by the cuts, produced when a 3D edge is mapped to two different edges in the boundary between different charts, quite often having largely different lengths in the parameterization domain. A mapping of this type will generate a change of scale in the parameterization which will be quite visible in applications such as texture mapping and require expensive solutions [GP09]. Thus, a *base complex* (e.g. a polycube) with the same genus as the given input surface is usually more suitable as an alternative parametric domain.

Base complex parameterizations: Base complex parameterizations such as *sphere mapping* [PH03] (for genus-0 surfaces) and *polycube mapping* [THCM04] (for any arbitrary genus) allows a *seamless* continuous mapping of a triangle mesh to the faces of regular base complex domains without requiring the surface to be cut or to the creation of discontinuities.

- **Spherical mapping** [PH03]: Some shapes are often described by closed, genus-zero surfaces. For such shapes, the sphere is the most natural parameterization domain, since it does not require cutting the surface into a disk (see Figure 2.50). A spherical parameterization must prevent parametric foldovers and guarantee a one-to-one spherical map, which can make it difficult between highly deformed shapes to create a parameterization adequately sampling all the surface regions.

A large stretch in any direction about a surface point means that the reconstruction of the surface signal will lose high-frequency detail in that region. A coarse-to-fine optimization strategy penalizes undersampling using a stretch-based parameterization metric. First, a spherical parameterization from a spherical domain D to the given closed shape $D \to S_T$ is performed. Next, another spherical parameterization is done from the sphere domain D to a polyhedron domain P, such as an octahedron: $D \to P$. Finally, the polyhedron P is unfolded into an image grid domain I: $I \to P$. All these maps are invertible, and their composition provides a map: $I \to P \to D \to S_T$.

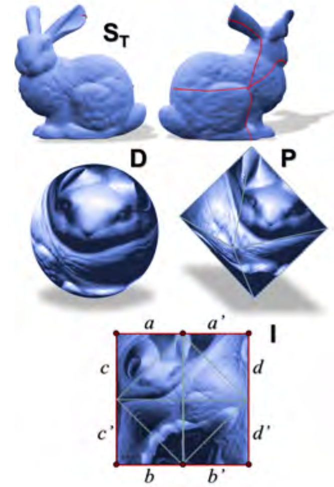


Figure 2.50: Spherical parameterization [PH03] of genus-0 surfaces allows a seamless continuous map of an irregular mesh structure to the faces of a spherical regular parametric domain, without creating discontinuities.

A spherical parameterization is created from the original surface to a seamless sphere domain. This regular seamless layout allows more efficient parallel processing, acting as a spatial directory either to access genus-0 irregular parameterized data by cell lists L (see Figure 2.51 (Bottom)), or through a quad-based regular remeshing S_Q of the original irregular structure (see Figure 2.51 (Top)). In general, this is done with less distortion near the cutting boundaries (e.g. compared to planar geometry images [GGH02a], see comparison of distortion error metrics in Section 2.1.3.3), and without creating discontinuities in the parameter domain.

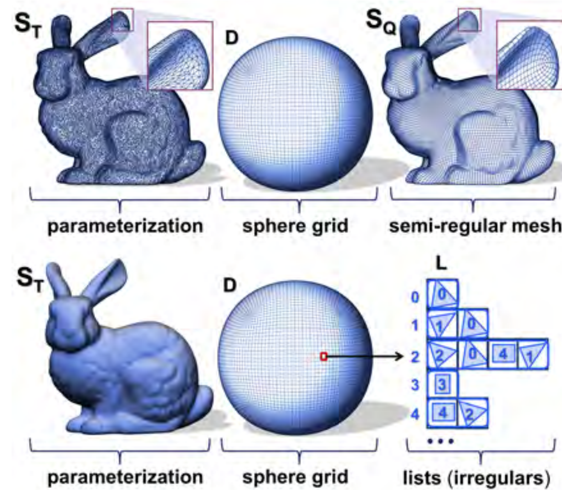


Figure 2.51: A spherical geometry image parameterization as a spatial directory. (*Top*) Allows a regular remeshing of the original genus-0 irregular structure by a seamless layout of the mesh without discontinuities, or (*Bottom*) allows access to the irregular parameterized original data by localized cell lists.

- **Polycube mapping:** A *polycube* P is a quadrilateral regular base domain surface defined by the composition of equal size cubes. Polycube mapping parameterizes the triangular mesh S_T over the set of square charts of a polycube domain P. Tarini *et al.* [THCM04] defines the polycube map from a manually built polycube domain in five main steps (see Figure 2.52): first, the user manually warps the polycube P over the surface S_T, obtaining the warped polycube P_W. Next, the vertices of S_T are projected in the normal direction over the warped polycube surface P_W. This projection may generate foldovers, mostly in regions with small features. Then, the projected mesh over P_W is warped back to the surface of P. This gives an initial positioning of the vertices of S_T that are assigned to one of the quads of the polycube P.

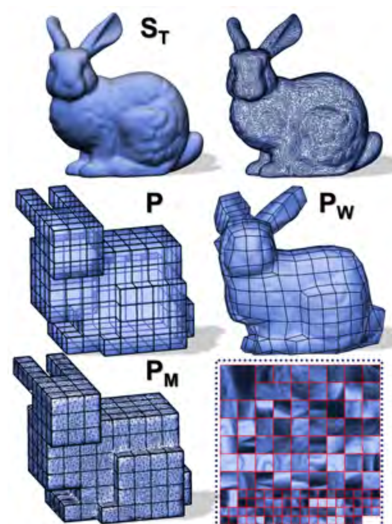


Figure 2.52: Mesh parameterization by polycube map allows arbitrary genus surfaces to be mapped and achieves a seamless continuous map of an irregular mesh structure to the faces of a polycube regular parametric domain, all without creating discontinuities in the parameter domain.

However, this usually does not define a good parameterization because the piecewise linear funtion maps each triangle of S_T to a corresponding parameter triangle over P, possibly deforming it considerably or even, not resulting in a one-to-one map in some parts.

Therefore, to optimize the mapping, the barycentric coordinates are computed by a fixed-boundary parameterization to drive an iterative process where the vertices can be reassigned between the quadrilateral faces. Finally, once the quads are parameterized, inter-chart smoothness is further optimized by a global relaxation method to obtain the final polycube map P_M.

Wang *et al.* [WHL$^+$07, WJH$^+$08] introduced two polycube mapping approaches that first map the 3D model and a manually constructed polycube to the canonical domain (e.g., sphere, euclidean plane or hyperbolic disc), and then seek the map between the two canonical domains. The resulting polycube map is guaranteed to be a diffeomorphism (see Section 2.1.1).

He *et al.* [HWFQ09] proposed an automatic polycube construction and mapping scheme, first by a consistent voxelization, and then by breaking down the model and the polycube into genus-0 rings by an uniformization metric. Finally, a piecewise map is computed, with harmonic functions (see Section 2.1.3), independently for each ring. However, harmonic mapping between rings is known to be a time-consuming non-linear process.

In general, parameterized grids offer an efficient setting for a spatial directory (see Table 2.3). In particular, *base complex*-based schemes are somehow able to map arbitrary genus surfaces S_T into the seamless and regular parameter domain of a polycube P of the same topology. The polycube map can serve as a spatial directory either to access irregular parameterized surface data of arbitrary genus, organized in cell lists L (see Figure 2.53 (*Bottom*)), or through a regular quad-based remeshing S_Q of the original irregular structure (see Figure 2.53 (*Top*)), with less distortion in general than a planar parametric mapping. However, it is difficult to control the polycube map process and may not work for models of complicated geometry and topology. In chapter 4 we explore a more flexible setting for the polycube map in order to exploit it as a suitable parametric domain for parallel efficient rendering applications.

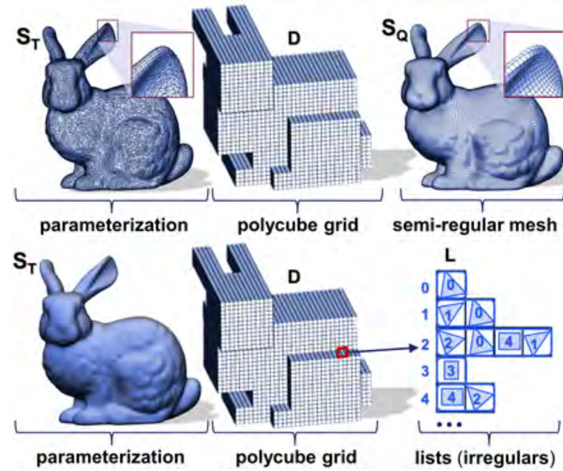


Figure 2.53: A polycube map parameterization as a spatial directory. (*Top*) Allows a regular remeshing of the original irregular structure by a seamless mapping without discontinuities, or (*Bottom*) allows access to the irregular parameterized original data by localized cell lists.

Parameterized grids main properties:
Grid embedding + lists
√ Block transfers
√ Good cache alignment and behavior
✗ Empty wasted space for sparse data
✗ Non-adaptive to irregular data distributions

Table 2.3: Parameterized grids tradeoffs.

2.4.8 Tree-based data structures

Adaptive tree-based spatial data structures allow shape and shading data to be subdivided inside a hierarchical regular grid enclosing the object. The data is stored in a spatial hierarchy, such as an octree [Sam90]. This strategy avoids a number of issues of planar parameterizations, removing the need for a global parameterization to cut and unfold the mesh in a planar domain. Furthermore, spatial hierarchical methods usually avoid the interpolation problems between the discontinuities of different cutted parts, while at the same time the hierarchy can be refined adaptively in specific areas of interest.

- **Octree textures:** Debry *et al.* [DGPR02], and Benson and Davis [BD02] have shown how 3D hierarchical data structures called *octree textures*, can be used as a spatial directory D to efficiently store shape and shading information of a surface without a spatial parameterization. The octree textures approach provides low distortion on the sampled spatial data because the shape is regularly and adaptively sampled only where the surface intersects a cell of the subdivided grid (see Figure 2.54). The memory requirements of an octree texture are lower compared to a high-resolution regular grid, but the tree still contains many unused entries in its internal nodes, and accessing the data requires a long chain of indirections, as opposed to, for instance, a regular grid.

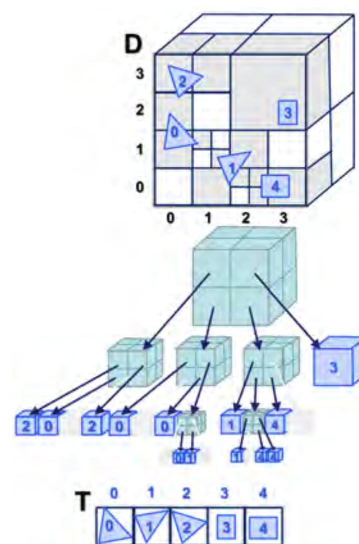


Figure 2.54: Octree texture data structure [LHN05, Lef06].

Note that octree textures may now be interactively constructed [LHN05, BHGS06, Lef06, ZGHG08], and their adaptive hierarchical structure is also suitable in the case of acquired surfaces, where their poor topological guarantees, coupled with their high density, make them hard to unfold in the plane with the use of planar parameterization over a 2D grid. Octrees textures are a spatial directory useful in applications with sparse spatial data [DT07, SZS+08], where they only locally partition the space into increasingly smaller boxes where it is required (see Figure 2.55).

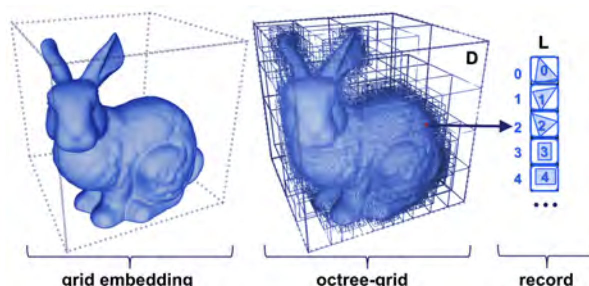


Figure 2.55: Octree texture used as an spatial directory. The hierarchy can be adaptively refined in specific areas with sparse irregular spatial data.

The volumetric hierarchical space subdivision of an octree generates satisfying clustering at coarse subdivision levels. However, it is also obvious that at finer levels, when the cells come closer to the surface, the volume-based decomposition leads to imbalanced clustering in areas where the surface is not aligned with the main directions of the data structure. However, this implies an overhead in access time and storage space, which may limit their use in some interactive applications. Consequently, other approaches have appeared, which combine the main advantages of volume-based approaches and 2D planar grids at their leaf nodes, by only coarsely subdividing the volume hierarchy until the surface can be faithfully captured by a simple planar grid without foldovers. These methods, while removing the need for a global parameterization, also achieve a more efficient packing and access than a regular octree.

- **Volume-surface trees** [BHGS06] introduced a data structore to alleviate the forementioned deficiencies with a modified version of the octree. The volume-surface tree is an hybrid octree/quadtree structure which combines a 3D scheme for the first levels of the tree, and a 2D scheme as soon as the surface can be projected onto a plane without folding. The structure contains three different types of nodes: volume nodes (comparable to octree nodes), transition nodes (being the leaves of V-nodes) and the roots of 2D hierarchies of surface nodes (comparable to quadtree nodes).

  Note that each transition node carries a local frame that is used to align its corresponding sub-quadtree. The union of all transition nodes defines the volumetric layer under which it becomes possible to implement 2D algorithms (the transition-layer). Having the transition-layer at low depth, and switching to quadtrees as soon as possible, reduces the memory overhead thanks to the 2-dimensional structure, and speeds-up traversals and tests. Evaluating whether a piece of surface will exhibit folding during the 2D projection can be done by numerically integrating the curvature over its area. The method relies on different heuristics to define a height field indicator [PG01] to ensure projections without foldings.

- **TileTree** [LD07] proposed an adaptive method more compact than a full-octree. The key idea is to use an octree to position square texture tiles around the surface. The process starts by building an octree around the spatial data. However instead of storing a single sample in each leaf, a set of 2D tiles are mapped onto the faces of the leaf (up to six tiles per leaf). The octree is subdivided until no more than one fold exists in each leaf. The tiles are packed together in arbitrary order. The resolution of the tiles can be locally changed to be adapted to changes in the detail level of the stored data. During the query evaluation over the spatial data, the surface normal gives the projection onto the tiles, to then retrieve the shape and shading information from the 2D tile map.

  In general, the shape and shading details to be stored in the spatial directory D are usually much finer than the general base shape of an object. Therefore, the octree only requires a few levels where many neighboring samples will be stored in the tiles of a same leaf, which guarantees good access coherence to the stored data (see Figure 2.56).

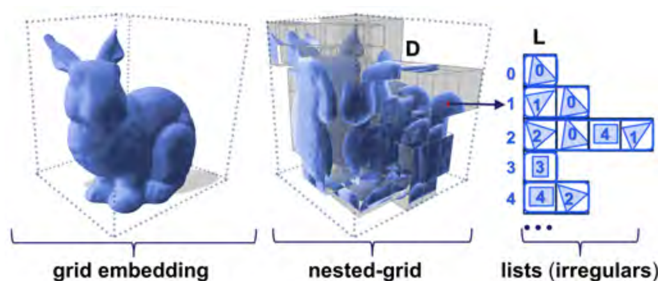


Figure 2.56: Tiletree [LD07] acting as volume-surface hierarchical spatial directory, it combines a coarse volume hierarchy, subdivided until the surface is faithfully captured by a simple 2D grid without foldovers. The 2D grid leaves act as a locally planar spatial subdirectory, achieving a more efficient packing and access evaluation than a regular octree.

Adaptive tree-based data structures main properties:
Hierarchical regular or irregular subdivision
√ Block transfers
√ Fairly good cache alignment and behavior
√ Support insertion and flexible memory allocation and layout
✗ Larger space overhead (e.g. pointers)
✗ Variable depth in hierarchy means variable latency in queries

Table 2.4: Adaptive grid data structures tradeoffs.

2.4.9 Hashing data structures

Another different strategy to collect and retrieve sparse spatial data is *spatial hashing*. By embedding the spatial data over a grid, hashing should only store those cells where data elements are located. In the presence of sparse input data, most of the cells will be completely empty, and only a small fraction of the cells require to be stored. The key idea is place the sparse data into a hash table by defining a hash function h. This strategy forgoes adaptivity and store fixed-resolution data in the hash table. The hash function maps each data element from the query coordinates – also called *keys* k – on the domain into locations $h(k)$ inside the hash table. This causes no problems until a record with a key k' has to be inserted and the location $h(k')$ is already occupied by another key k. In this case we say a *collision* has occurred. Handling collisions is the central issue in hashing.

Keys are taken in a universe U of size $|U|$. We note $D \subset U$ as being the set of *defined keys*, that is the keys from U which should be stored in the hash table. Keys which do not belong to D are called *empty keys*. We consider the *load factor* d, which corresponds to the ratio of the number of defined keys to the number of keys that the hash table can hold. A hash table with load factor $d = 1$ is a *minimal hash*, that is, a hash with no wasted space. Each defined key may be associated with some additional data. The input is thus given as a set of key-data pairs.
Static spatial data should be collected only once, and hence the hash function used to pack the data can be chosen to avoid any collision between the input keys with collision-free hashing techniques. On the other hand, dynamic data constantly changing at interactive rates requires more flexible hashing schemes able to continuolsy reconstruct and update the structure at every frame.

Collision-free: collision-free hashing methods address the issue of repeated proving by guaranteeing that every item can be located in exactly one location in the hash table, allowing all queries to be answered with a single probe. Next we will describe two different strategies useful creating compact spatial directories requiring only a single probe to query the spatial data from the hash table.

- **Perfect spatial hashing** [LH06a] is a especially interesting setting in the definition of the hash table, where the hash table should have two properties: one is that it should be minimal, which means that the hash table should be as compact as possible, avoiding any wasted memory space. The second property is that the hash function has to be perfect, so that any two distinct input elements are mapped to distinct locations in the hash table (see Figure 2.57). In other words, there are no collisions in the hash function.

  There are theoretical bounds defining the description of a hash function, which must at least have a complexity of $1.44n$ bits for the data [FHCD92], where n is the number of input entries. By this theoretical bound the description of the hash function in a compact form cannot be expected, and some auxiliary data is required.

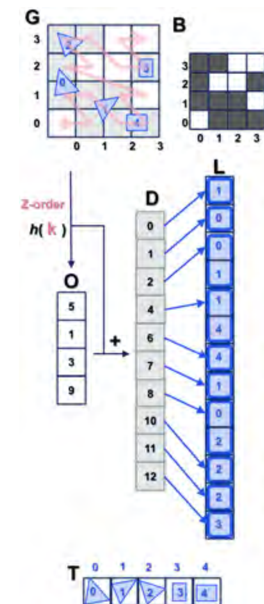


Figure 2.57: Perfect spatial hashing [LH06a].

  Lefebvre and Hoppe [LH06a] introduced an auxiliary offset table computed in a preprocessing step, to make a simple hash function perfect. In a query evaluation on the hash table, the hash function uses the query coordinates to make a look up in the offset table to retrieve an offset value. This look up in the offset table is done by using toroidal addressing.

  After retrieving the offset value, it is added to the original query coordinates to obtain a final look up in the hash table, using a toroidal addressing as well. So the evaluation of the hash function is extremely simple, just requiring two look ups: one in the offset table and one in the hash table. Furthermore, as it does not contain branching, it is ideal for SIMD parallel query evaluation.

Note that as hash function h only prevents collisions between non-zero elements of the spatial grid G, queries on empty locations are not directly supported in the hash table D. However, by encoding which elements are non-zero using domain bits in a grid B (with the same dimensions as the grid G), queries on empty locations are also supported.

The main constraint of this method is that finding a perfect hash function for the input data requires a very slow preprocessing. This means that, once constructed, we are not allowed to introduce or delete data because we would need to compute a completely new hash function again. However, the attached values of the data elements already inside the hash table can be modified without changing the element location.

- **Hash grid** [LD08] is a data structure that reduces the memory requirements of a *compact grid* (see Section 2.4.7), by using perfect hashing based on column displacement compression; being more efficient in both time and space than traditional compact grid methods (see Figure 2.58). Given a sparse grid domain G, a hash function h is computed for a hash table D, such that each non-zero element $G(i,j)$ is hashed to the position $h(i,j)$. The hash function h should be perfect and close to minimal, i.e. the hash table H should contain as few unused entries as possible. The algorithm works as follows: The first column of the grid D is copied to the hash table G at offset 0. Each subsequent column of the grid G is then copied to the hash table H at the smallest offset, such that non-zero elements do not overlap. Each offset is determined starting from the offset of the previous column. For each column i, this offset is recorded in a 1D offset table O at position i. Each non-zero element $G(i,j)$ corresponds to $D[O[i]+j]$, i.e. the hash function h is given by $h(i,j) = O[i]+j$. The queries on empty locations of the spatial grid domain can be supported by encoding which elements are non-zero using domain bits in a grid B (with the same dimensions as the grid G).

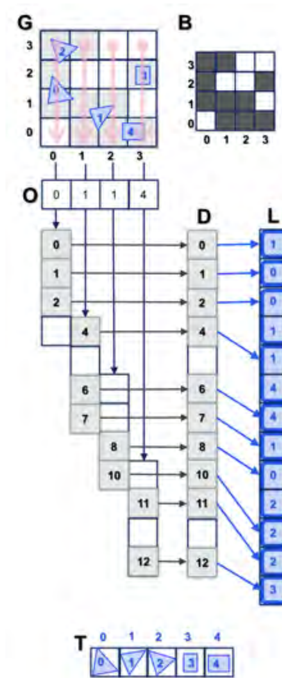


Figure 2.58: Perfect hashing by column displacement compression [LD08].

The hashed grid representation can be combined with the scalable sort-middle grid build method of Ize *et al.* [IWR+06] by parallelizing the computation of the domain bits, and then parallelizing the computation of the hash function by distributing the columns of the grid over the threads.

Multiple-choice hashing: multiple-choice hashing is a hash table scheme where each item is stored in one of $H \leq 2$ hash table buckets. The ability to choose from multiple locations when storing an item improves space utilization, while the simplicity of such schemes makes them highly amenable to a GPU-friendly implementation. Some variants, such as *cuckoo hashing*, allow items to be moved among their H choices in order to improve load balance and avoid hash table overflows. Here we consider hashing schemes that move items on insertion and deletion operations, as arguably one would be willing to spend more time on such operations as opposed to more frequent lookup operations.

- **Cuckoo hashing** [PR04] is a variation on multiple choice hashing schemes. The hash table is divided into H subtables of the same size. In a sequential construction process, items are inserted one by one. An item can be placed in one of the possible H slots in the different subtables, if any of them are empty. But if there is no room for an item at any of its H choices, instead of causing an overflow, the method considers moving the item in one of those H records to another consistent location with respect to its own H choices. This starts a recursive process were the evicted item must be reinserted into the hash table following the same procedure, or until sufficiently attempts have been made to declare a failure. Moving the items during an insertion allows them to flow intuitively towards less contested records in the table. The probability of this happening is small when using two hash functions, and even more empirically unlikely using more hash functions.

- **Two-level cuckoo hashing:** [ASA+09] defines a very efficient parallel hashing construction by relaxing the requirements on the compaction of a hash table, making possible to collect sparse data in a hash table in real-time with a GPU-friendly parallel construction and simple hash functions. The hash functions may produce collisions, but still provide query operations on the data with very few memory access operations.

  Alcantara *et al.* [ASA+09] modified the standard cuckoo hashing algorithm to work in parallel, allowing all of the key-value pairs to be inserted into the structure simultaneously. In the construction process, all threads managing items that have not yet been stored, simultaneously write their keys into the same subtable using the associated hash functions; CUDA semantics ensure that exactly one write will succeed for each record that is written into.

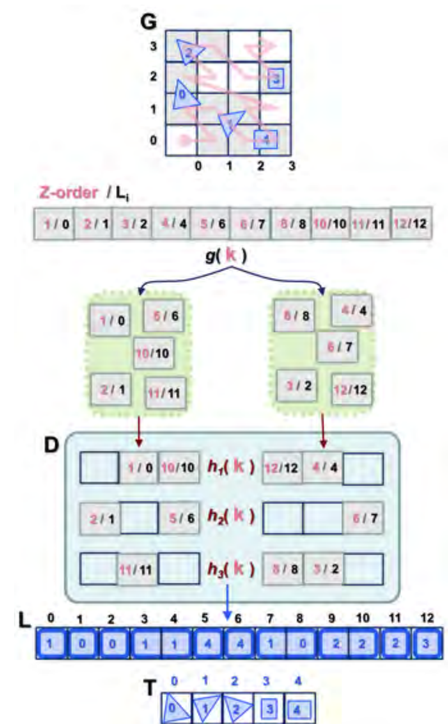


Figure 2.59: Two-level cuckoo hashing [ASA+09].

This procedure is designed to be efficient with data divided in small datasets by using a two-level hash table. In a first level, the input is partitioned into smaller buckets using a hash function $g(k)$. Then, a cuckoo hash $h(k)$ is built for each bucket in parallel, with a different CUDA thread block handling each bucket. This allows each cuckoo hash table to be built in shared memory, reducing the cost of the memory accesses incurred during construction. Moreover, it mitigates the cost of a cuckoo hashing failure since each cuckoo hash table is independent: failing to build one does not cause the others to be rebuilt from scratch (for more details see Section 5.1).

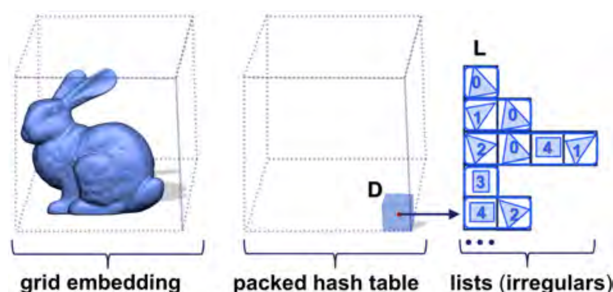


Figure 2.60: Hashing spatial directory: using a hash function maps each data element with the query coordinates into locations inside a compact hash table. This usually only requires a constant number of memory accesses to query the data elements.

Hashing data structures main properties:
Grid embedding + hashing
√ Compact storage
√ Constant time and simple query accesses
√ Support insertion and flexible memory allocation and layout
✗ Random nature of hash functions breaks coherence
✗ Construction process are slow or can easily fail at high load factors

Table 2.5: Hashing data structures tradeoffs.

Chapter 3

Detail mapping and simplification

Surface simplification methods must deal with a problem of great importance: preserving both the shape and the shading details attached to a geometric mesh structure without constraining the final quality. This chapter presents our proposed setting that uses a parameterization-based spatial directory to avoid some important simplification-limiting issues and to generate high quality compact and efficient representations.

Highly detailed geometric models are commonplace in many computer graphics applications to represent objects' surface and shading information (see Section 1.2.2).

Polygonal meshes remain the most common and flexible way to approximate surfaces, typically as dense triangle meshes (see Section 2.1.2). Nonetheless, beyond the geometric coordinates, the polygonal meshes also store shading information attached to the vertex, edge and face elements of the mesh, as so called *mesh attributes*.

Many applications often require shape and shading attributes with higher frequencies than the geometric coordinates themselves, e.g. fine color and shape details over a base surface (see Figure 3.1).

These requirements impose the definition of a mesh structure with support to this high frequency information. A straighforward method is to define a dense mesh structure, with a density enough to achieve an accurate sampling of the fine detail attributes (see Figure 3.1 (*Left*)). Alternatively, it is possible to use a mesh parameterization to map these attributes stored in a texture image, requiring only the storage of the texture mapping coordinates as mesh attributes (see Figure 3.1 (*Right*)).

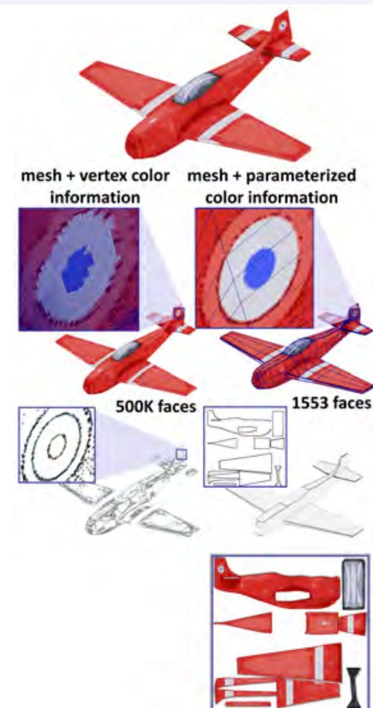


Figure 3.1: A plane 3D model with color shading information. (*Left*) The color information is attached to a sufficiently dense mesh structure as vertex colors, in order to capture the complex color details spanning over the surface, as shown below. (*Right*) The color information is stored in a texture atlas so that the mesh structure can be coarser but still define consistent chart boundaries, with lower complexity than the color field.

In general, a geometric model must allow a flexible description with different levels-of-detail (see Section 2.2), because the number of samples necessary to show the model accurately in the screen depends on the nature of the model, and its area on screen pixels, which in turn is related to its distance from the viewpoint. In order to create different levels-of-detail representations of models, surface simplification algorithms propose to reduce the mesh connectivity while preserving the surface and shading details.

In this chapter we will illustrate the problems appearing in the context of mesh simplification and the preservation of both the shape and the shading attributes attached to the mesh structure, in particular because the mesh structure simplification must deal with a difficult tradeoff between preserving the geometry coordinates and the mesh attributes.

Here we will show that a special setting of a spatial mesh parameterization can be helpful to greatly reduce the constraints of attribute preserving simplification methods, which at the same time allows querying and mapping of the high-resolution mesh attributes in any level-of-detail.

3.1 Context: mesh parameterization and simplification

In general, object fine color and shape details can arbitrarily span over the full mesh structure with high frequencies(see Figure 3.1 (*Left*)). If such information is directly stored in the mesh as vertex attributes, then, it requires a much denser mesh structure than if it were for only capturing the gross surface of the object.

in Section 2.2.1, we described simplification methods that only focused on preserving the surface properties of geometric objects, while any additional mesh attribute on the mesh structure would be compromised, or completely lost, as shown in Figure 3.2 (*Middle*).

3.1.1 Mesh attribute-preserving simplification

Mesh attribute-preserving simplification avoids the degeneration of the mesh attributes signals, introducing further constraints in the simplification process, to preserve not only the geometric coordinates, but also any other shape and shading attribute available, such as colors, normals and texture coordinates attached to mesh vertices.

Hoppe presented a seminal attribute preserving simplification approach [Hop96] to construct a single *progressive mesh* representation for all the levels-of-detail, and introduced attribute preserving metrics for vertex colors, normals and texture coordinates. Independently, Garland *et al.* [GH98] presented an extended quadric error metric for attribute-preserving simplification.

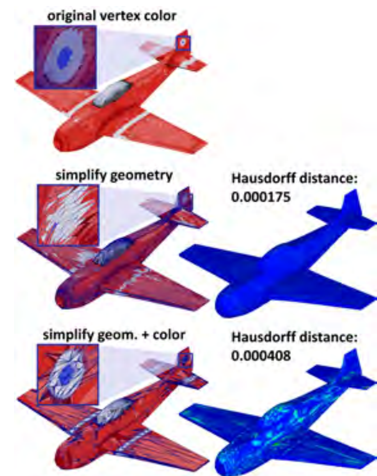


Figure 3.2: (*Top*) A dense triangle mesh with vertex colors. (*Middle*) Simplification does not preserve mesh attributes, such as the vertex colors. Rather, it destroys the surface appearance, but allows the surface properties to be preserved in coarser mesh structures. (*Bottom*) Attribute-preserving simplification must preserve the mesh structure around both the surface and the attribute features. The balance in this tradeoff can lead to less geometric fidelity.

Those methods must guarantee an adequate accuracy of the attributes to generate the different levels-of-detail of the mesh, preserving at the same time the attributes and the surface properties in the resulting simplified mesh.

However, when the surfaces to be simplified have complex attribute signals attached, preserving them may largely compromise the surface properties preservation on the mesh structure, and even make it impossible to simplify the mesh structure to coarse levels, as shown in Figure 3.2 (*Bottom*).

The main problem is that most edges around a high-frequency attribute signal changes are not simplified, and consequently other parts of the surface may be drastically contracted losing more surface quality than with purely geometric simplification methods. This geometric degradation can be measured with the Haussdorf distance between the original and the simplified meshes, as shown in Figure 3.2, while the attribute preservation quality can be measured with image-based error metrics (see Section 2.2.4).

3.1.2 Texture-based attribute-preserving simplification

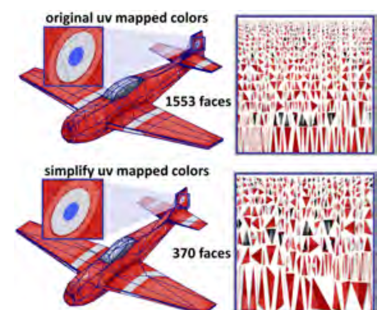


Figure 3.3: (*Top*) A triangle mesh with a per-triangle chart texture atlas. The mesh structure avoids the dense sampling of the color information by storing it into a texture atlas. (*Bottom*) A different LOD generated with simplification requires a different set of per-triangle charts texture atlas.

Texture-based attribute-preserving simplification includes a range of approaches that try to encode the objects' complex shape and shading signals directly in a texture, called a texture atlas (see Section 2.4.7.1). Therefore, they only require the storage of set of texture coordinates in the mesh structure, to map the surface triangles into the texture image. These methods aim to reduce the simplification limitations produced by the most complex attributes signals, such as color and shape detail information.

The first texture-based methods [CMR+99, COM98, TCS03, C06] defined a per-triangle mapping, placing each triangle as an independent chart in the planar parameter domain (see Figure 3.3). Then the triangle mesh attributes are resampled onto the parameter domain triangles and finally stored in the texture.

However, when the triangle mesh connectivy is simplified, and new triangles appear from the collapse operations, the new coarser levels cannot reuse the same texture map anymore. The reason for this is that once the connectivity has changed, the triangles appearing from the collapse operations might not be in the original texture map.

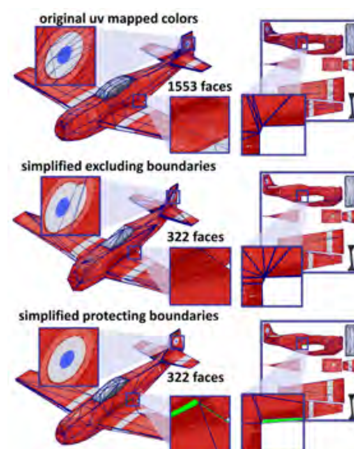


Figure 3.4: (*Top*) A 3D model with its mesh parameterized texture atlas. (*Middle*) Attribute-preserving simplification methods that exclude edges of chart boundaries, avoid texture sampling artifacts but considerably reduce the geometric quality of the surface. (*Bottom*) Attribute-preserving simplification methods that only protect the charts boundaries leaverage better geometric quality, but results in the collapse of a few boundaries out of the texture chart bounds, leading to attribute sampling artifacts (shown in green).

These methods are not compatible with CLOD level-of-detail techniques, only with DLOD techniques (see Section 2.2.3), because the mesh would dynamically change during the visualization and would forbid the usage of a predefined set of triangles stored in the texture atlas. Furthermore, they require a high texture memory footprint to define one texture map per level, and could also produce severe resampling and filtering artifacts, as shown in Figure 3.6.

Other attribute-preserving simplification methods [Hop96, GH98, SSGH01, CH02a, Moo02] take advantage of a mesh parameterization to resample the original mesh attributes, preserving as much as possible the neighboring relations between the triangles, and resampling their signals in a set of charts of a single texture image.

These methods focus on reducing attribute distortion and preserving the most relevant surface properties with respect to a single texture atlas shared by all the coarser levels-of-detail (see Figures 3.4). They operate in such a way that, to the greatest extent, neighboring triangles in the surface mesh are also meant to be neighbors in the parameter domain of the texture atlas, in the same chart. Such connectivity-preserving techniques try to use only a few charts as the layout for the parameter domain. These later simplification approaches also must deal with the complex balance of shape and attribute fields preservation.

The texture coordinates of a mesh parameterization also define a smooth scalar field over the mesh structure, with sharp discontinuities between the different charts in the parameter domain (see Figure 3.6 (*Top*)). The charts boundaries follow a subset of the triangle mesh edges, which must be preserved during the simplification process.

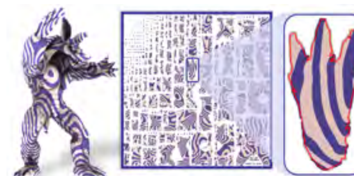


Figure 3.5: Armadillo 3D model with a complex planar parameterization.

Nevertheless, when preserving all the edges of chart boundaries during a simplification process, sampling artifacts may appear when the collapsed triangles span outside of the corresponding chart in the parameter domain, as shown in Figure 3.6 (*Bottom*).

Therefore, the parameterized mapping allows to free the simplification of the mesh structure without the constraints of any other available attribute. These other attributes usually have higher complexity than the parameterization itself, so it is beneficial to encode them in a separate texture atlas. Nonetheless, the mesh parameterization should be as good as possible, because the texture coordinates attribute often introduces discontinuities that still prevent many of the edge-collapsing operations that would be valid if we only considered geometric quality degradation.

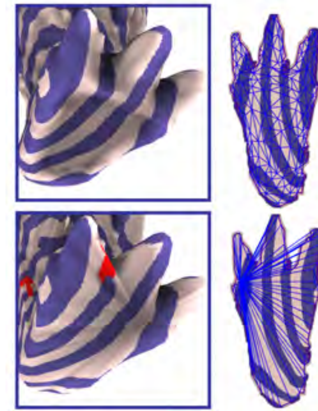


Figure 3.6: Armadillo LODs generated with attribute-preserving simplification. (*Top*) Close-view of the foot texture chart, showing in red the chart boundary edges and the original triangulation in the parameter domain. (*Bottom*) Even if the complex boundary chart edges of the foot are preserved, collapsed triangles can span out of the boundary in concave regions, leading to texture sampling artifacts (shown in red).

3.1.2.1 Trade-offs: mesh parameterization and simplification

Now we are going to overview the properties that each mesh parameterization has with respect to simplification and detail mapping.

Many parameterization methods [LPRM02, ZSGS04, SLMB05] unfold the geometry in charts of irregular and complex shape boundaries (e.g. non-convex boundaries) that simplification methods must preserve (see Figure 3.7).

These irregular charts allow a low parametric distortion in the mapping of the surface attributes on the texture. However, this setting wastes empty space in between these irregular charts.

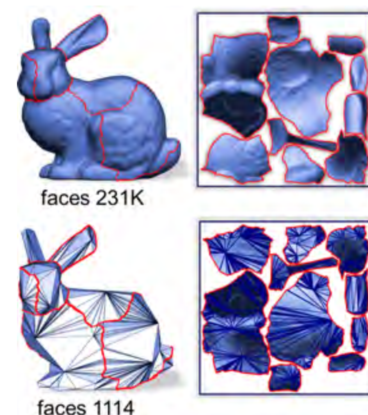


Figure 3.7: (*Top*) Bunny model parameterized and simplified with a free boundary chart parameterization [ZSGS04]. (*Bottom*) Complex chart boundaries must be protected in the simplification process. They waste texture space and limit the surface quality on coarser LODs.

Furthermore, the irregular boundaries are difficult to preserve during the simplification process, i.e. if too many edges are collapsed from an irregular chart boundary, it may be impossible to preserve its original shape. Therefore, as all the levels-of-detail take samples from the same texture atlas, all the LODs must preserve the chart boundaries as much as possible. Otherwise they will be sampling outside of the chart texture grid bounds, where invalid values would be taken.

This introduces strong limitations in the simplification process, where most edges of the coarser mesh belong almost solely to the charts boundaries, resulting in a much worse geometric quality for the simplified object (see Figure 3.7 (*Bottom*)).

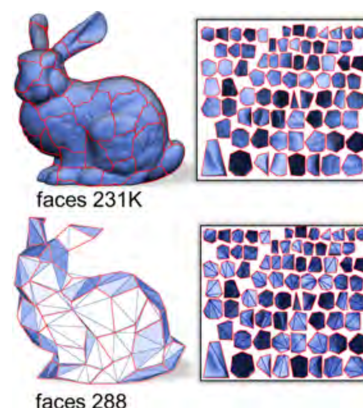


Figure 3.8: (*Top*) Bunny model parameterized and simplified with a convex straightened boundary chart parameterization [SSGH01]. (*Bottom*) Convex charts with straight line boundaries allow a less limited simplification process, wasting texture space but producing relatively good surface quality on coarser LODs.

Other approaches [COM98, SSGH01, SWG$^+$03] tried to carefully create the parameterization charts, so that each boundary is closely aligned with straight line segments, between its chart neighbors in the surface. Therefore, many edges of the boundaries are aligned in the same straight line, and they can be collapsed during simplification without compromising the charts boundary shapes, and consequently avoiding sampling the texture information out of the bounds.

As it can be seen in Figure 3.8, this allows the reduction of the simplification limiting factor, providing a better geometric quality in the coarser LODs.

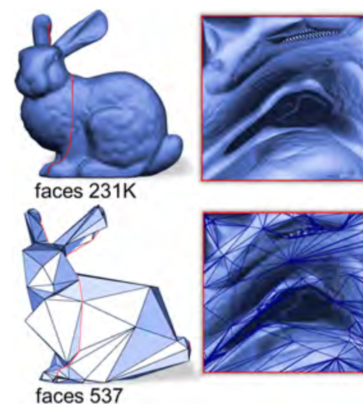


Figure 3.9: (*Top*) Bunny parameterized and simplified with a single regular boundary chart parameterization [GGH02b]. (*Bottom*) This chart does not waste texture space but it may introduce large texture parameterization distortions, and allows a fairly good geometric quality on coarser LODs.

The simplification of a texture parameter domain encapsulates some interesting properties, such as having a reduced number of chart discontinuities, and making them simpler (e.g. following regular paths over the surface), which helps to reduce the constraints of the simplification process.

For instance, a single regular chart generated with a geometry image parameterization [GGH02b] does not waste space in the texture grid domain but produces a high parametric distortion (see Section 2.1.3.3). The regular chart boundary allows the simplification process to generate coarser representations with better results (see Figure 3.9). However, the geometric quality of the coarser triangles is still fairly low.

Nonetheless, other mesh parameterizations [PH03, THCM04] try to map the object into a parameter domain of a single regular structure, but with a similar shape and the same topology as the object itself. In this case, the parameterizations do not require the object to be cut to achieve the mapping, and a *seamless* mapping without boundaries in the parameter domain is obtained.

For example, a spherical geometry image [PH03] allows to define a seamless unfolding of genus-0 surfaces into a regular chart to be defined, which connects its boundary in a pairwise manner (see Section 2.4.7.1), avoiding the introduction of irregular discontinuities in the mesh parametererization.

These simplification methods avoid any boundary constraint in the simplification process, leading to very good quality LODs, as shown in Figure 3.10 (*Bottom*), and it remains possible to sample the encoded attributes in a seamless texture domain, such a cube-map (see Figure 3.10 (*Top*)).

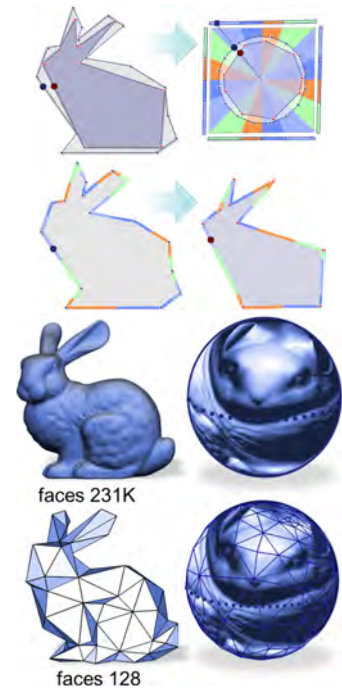


Figure 3.10: (*Top*) A spherical domain mapping defines a projection line mapping any point on a lower LOD to a corresponding point on the highest LOD without constraints. (*Bottom*) The simplification process is not limited by chart boundaries, allowing a good surface quality to be provided on coarser LODs.

> An interesting observation is that the spherical domain coordinates between two different levels-of-detail allow a *projection line* to be defined, mapping any point on a lower LOD to a corresponding point on the highest LOD without constraints.

In the case of objects of arbitrary topology, i.e. with a genus greater than 0, or objects with fairly extruded parts and having shapes not similar to a sphere, the spherical parameterization would be incompatible or would introduce too a large distortion in the attribute mapping.

Another mesh parameterization that allows a more flexible setting is polycube mapping [THCM04] (see Section 2.4.7.1), which offers a better trade-off between providing a good attribute mapping quality and, an unconstrained simplification. The polycube map provides a seamless layout parameterizing the triangular mesh over the set of square charts of a polycube that allows the object to be simplified without parameter-domain boundary constraints (see Figure 3.11).

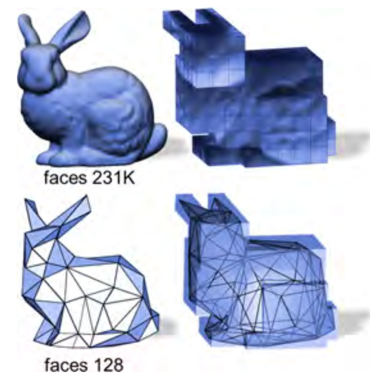


Figure 3.11: A polycube domain allows mapping any point on a lower LOD to a corresponding point on the highest LOD. The polycube configurable shape and topology allows a flexible setting in order to reduce mapping distortion, while limiting the simplification process, which provides a good surface quality on coarser LODs.

We explore the usage of the most suitable mesh parameterizations to define a spatial directory data structure and avoid the aforementioned simplification limitations. In particular, we aim to be able to sample complex attribute fields on the mesh structure or in texture atlases without resampling them, and with efficient parallel query operations for real-time rendering. The method is called *Inverse Geometric Textures*.

3.2 Inverse Geometric Textures

In this section, we introduce *Inverse Geometric Textures* (IGT) a spatial data structure that allows preservation of surface and shading details from a high resolution triangle mesh onto lower resolution ones, generated with any given simplification method, and without discontinuity or resampling constraints. At the same time IGT allows efficient local parallel queries of the original shading attributes from the high resolution triangle mesh. For this, IGT decouples surface attributes by introducing a so-called *decoupling parameterization* defined on the reference triangle mesh to generate an *inversely parameterized* regular grid, also called the *spatial directory*, where for each cell, a list stores all triangles that are mapped onto it.

Figure 3.12: Sample models with color attributes stored in a texture atlas. The mapping $\mathcal{MT}_a$ is an artist parameterization for attaching the shape and shading attributes to the triangle meshes.

In general, artists usually perform manual corrections on each representation level, to ensure that consistent surface attributes and texture maps are provided. In this way, the coarser attribute values do not produce misplacements around the discontinuities, especially where the original mesh connectivity has changed and shading artifacts may appear.

Our proposed technique can be successfully used with a high resolution mesh M with any shape and shading attributes A, avoiding manual fixes and constraints on the simplification method.

As an example, the high resolution object can have surface attributes as with *artist-provided* parameterized textures (see Figure 3.12), which are not modified or resampled, avoiding additional effort to directly use the original artist-designed content. We denote the mapping between the mesh M and the attributes A in the parameter domain, as $M \xrightarrow{\mathcal{MT}_a} A$.

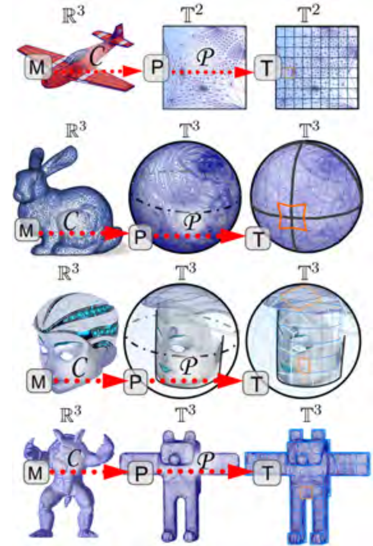


Figure 3.13: Sample models M mapped by the mapping step $\mathcal{C}$ to a parametric surface P. Next, they are projected by $\mathcal{P}$ to a parametric layout T divided in regular cells.

The proposed spatial data structure is based on the concepts of multidimensional searching [Sam90]. In particular, it is related to the *fixed grid* range searching scheme [BF79], which we adapt for real-time level-of-detail to achieve better spatial data locality and more parallel-friendly query operations on the shape and shading information. With that objective in mind, we define the inverse mapping $\mathcal{I}$ ($\mathcal{I} = \mathcal{M}^{-1}$).

Here we will define the parameter domains (P, T, and D) and the composite of map operations ($\mathcal{M}$, $\mathcal{P}$ and $\mathcal{C}$) needed to consistently define and evaluate IGT. Also, we describe its construction process and query evaluation method.

We assume that a high resolution triangle mesh M with shading surface attributes A are provided. The triangle mesh is defined by N triangles where shading attributes are attached to the vertices.

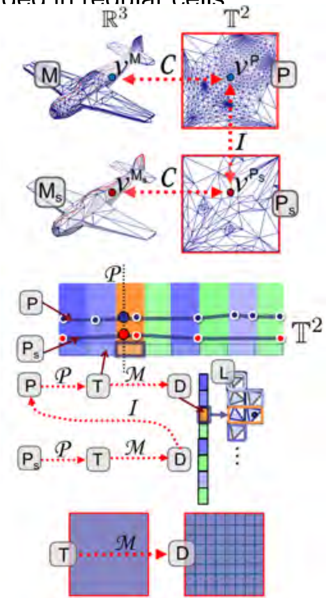


Figure 3.14: (*Top*) The points v^M of the original mesh M, and v^{M_s} of the simplified mesh M_s are mapped by $\mathcal{C}$ to corresponding points v^P and v^{P_s} in a planar parametric surface. (*Middle*) The two corresponding points v^{P_s} (shown in red) and v^P (shown in blue) are projected by $\mathcal{P}$ into a same cell of T. (*Bottom*) The triangles of P that overlap a same cell of T are mapped by $\mathcal{M}$ and stored in localized list in L of a 2D spatial directory D. This allows to define the inverse map $\mathcal{I}$.

3.2.1 Parameter domains and mappings: ($M \xrightarrow{\mathcal{C}} P \xrightarrow{\mathcal{P}} T \xrightarrow{\mathcal{M}} D$)

In this section we introduce the notation and describe the particular formulation that is relevant to the setting of IGT.

Given a triangle mesh M, the geometric coordinates of vertices in $\mathbb{R}^3$ are denoted as v^M. A simplified mesh version of M is noted as M_s, and its respective geometric coordinates as v^{M_s}.

The high resolution mesh M has attribute values v^A in the vertices, such as per-vertex shape and shading information, or the texture coordinates of a mesh parameterization – called $\mathcal{MT}_a$ – in order to attach the surface with any information stored in a texture atlas A, i.e. $M \xrightarrow{\mathcal{MT}_a} A$, as shown in Figure 3.12.

IGT requires the definition of a decoupling parameterization $\mathcal{D}$ on the reference triangle mesh M to generate an inverse parameterized directory grid texture D, where each grid cell stores a list into a linear memory buffer L, with information about all the triangles are that mapped onto it (see Section 3.2.3). The decoupling parameterization is composed of three different mapping steps $\mathcal{D} = \mathcal{M} \circ \mathcal{P} \circ \mathcal{M}$ described in Sections 3.2.1.1, 3.2.1.2 and 3.2.1.3.

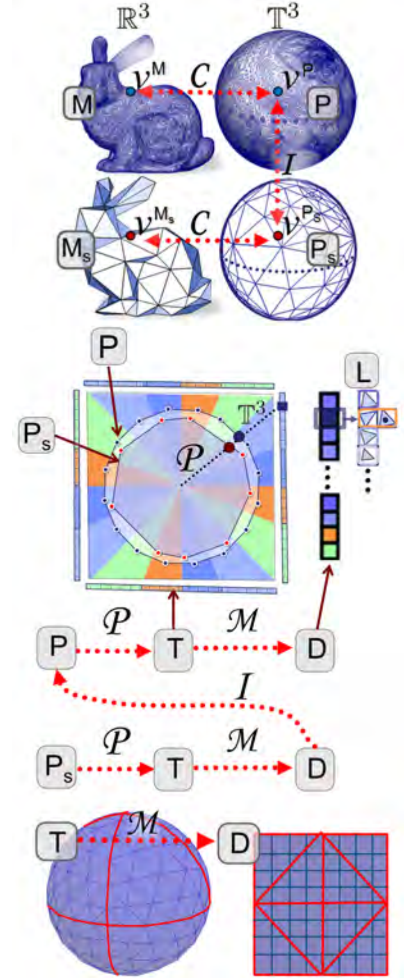


Figure 3.15: (*Top*) The points v^M of the original mesh M, and v^{M_s} of the simplified mesh M_s are mapped by $\mathcal{C}$ to corresponding points v^P and v^{P_s} in a spherical parametric surface. (*Middle*) The two corresponding points v^{P_s} (shown in red) and v^P (shown in blue) are projected by $\mathcal{P}$ into a same cell of T. (*Bottom*) The triangles of P that overlap a same cell of T are mapped by $\mathcal{M}$ and stored in localized list in L of a 2D spatial directory D. This allows the inverse map $\mathcal{I}$ to be defined.

3.2.1.1 The mapping $\mathcal{C}$: mesh parameterization ($M \xrightarrow{\mathcal{C}} P$)

The decoupling parameterization $\mathcal{D}$ should avoid the simplification limiting issues appearing over M when aiming to preserve both the surface and the shading properties.

For this reason, as a first step the mesh M should be mapped into a suitable parameter surface P with a mapping step $\mathcal{C}$. In this way we can decouple the complex attribute signals from the surface during the simplification, and then we are able to map them back efficiently without resampling to any simplifyied mesh M_s.

The mapping $\mathcal{C}$ ($M \xrightarrow{\mathcal{C}} P$) should define a parametric surface P as a one-to-one map from M. This map should have a reduced number of chart discontinuities and should allow a regular layout (see Section 3.2.1.2). Therefore, it can be any of the most suitable mesh parameterizations we described in the previous section (e.g. geometry image [GGH02a], spherical image [PH03], cylindrical map [HSV05] or a polycube map [THCM04]), as illustrated in Figure 3.13.

The parametric coordinates v^P of the surface P are stored along with the mesh structure M during the simplification process to generate M_s. In this way, the simplified mesh keeps the correspondence between a surface point with geometric coordinates v^{M_s} and the parametric coordinates v^{P_s} (i.e. $v^{M_s} \xrightarrow{\mathcal{C}} v^{P_s}$), as shown in Figures 3.13 (*Top*), 3.15 (*Top*), 3.16 (*Top*), and 3.17 (*Top*.

In the case where the mapping $\mathcal{C}$ is defined with a seamless mesh parameterization method, built in parameter domain with the same topology and with a shape similar to M, we can avoid the simplification constraints while preserving other attribute signals, and offer the best flexibility with IGT. Otherwise, in the case where the mapping $\mathcal{C}$ is defined with a mesh parameterization which introduce even simple and regular boundary discontinuities, they will have to be protected during the simplification process.

3.2.1.2 The mapping $\mathcal{P}$: parametric projection ($P \xrightarrow{\mathcal{P}} T$)

A crucial step of our proposed method is that, once we have the simplified mesh M_s (and equivalently the simplified parametric surface P_s) by using the parametric coordinates v^{P_s} our proposed solution should be able to map the original high resolution attributes v^A and the texture atlas A onto M_s.
For this reason, the parametric surface P is embedded in a *parametric layout* T overlappfing P with a set of regular cells c, with the mapping $\mathcal{P}$ ($P \xrightarrow{\mathcal{P}} T$). In this way, each cell is overlapped by a subset of the parametric surface triangles (see Figure 3.13 (*Middle*)).

In other words, the mappings $M \xrightarrow{\mathcal{C}} P \xrightarrow{\mathcal{P}} T$ define a coarse lattice in parametric domain T, where each cell contains a local description of the surface defined by the set of triangles contained in the cell.

At this point we take advantage of the setting defined by the parameterization $\mathcal{C}$ on the surface P, and its simplified counter part P_s, to define the aforementioned mapping $\mathcal{P}$. $\mathcal{P}$ defines projection lines which allows each parametric point of the simplified mesh v^{P_s} to be mapped onto a point v^P of the original parametric surface P. At the same time, both points are mapped by $\mathcal{P}$ onto a same point v^T on the same cell c of T (see Figure 3.16).

Depending on the parametric layout of the parameterization $\mathcal{C}$, a different method may be required to define the projection line of the mapping $\mathcal{P}$.

For instance, in the case of using a geometry image as $\mathcal{C}$, it defines a rectangular chart in P with the triangles from M.
If the simplification respects the parameterization boundaries, the rectangular parametric domain is still consistent in P_s from M_s (see Figure 3.14 (*Top*)).

In this setting, in a next step, the mapping $\mathcal{P}$ can find the corresponding point v^P on P from the simplified mesh P_s, through a perpendicular projection line starting in v^{P_s} from P_s through P, as shown in Figure 3.14 (*Middle*).

Equivalently, in case of using a spherical, cylindrical or a polycube mapping as $\mathcal{C}$, we would use a spherical, cylindrical or a polycube projection [THCM04] to define the respective projection lines from P_s to P (see Figures 3.15, 3.16, and 3.17).

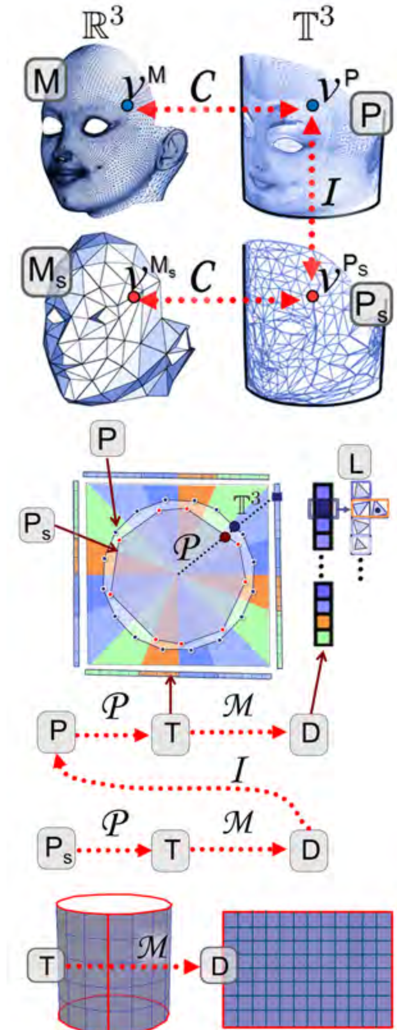


Figure 3.16: (*Top*) The points v^M of the original mesh M, and v^{M_s} of the simplified mesh M_s are mapped by $\mathcal{C}$ to corresponding points v^P and v^{P_s} in a cylindrical parametric surface. (*Middle*) The two corresponding points v^{P_s} (shown in red) and v^P (shown in blue) are projected by $\mathcal{P}$ into a same cell of T. (*Bottom*) The triangles of P that overlap a same cell of T are mapped by $\mathcal{M}$ and stored in localized list in L of a 2D spatial directory D. This allows the inverse map $\mathcal{I}$ to be defined.

3.2.1.3 The mapping $\mathcal{M}$: domain unfolding ($T \overset{\mathcal{M}}{\to} D$)

Given that the shape and shading data of M is mapped on the parametric surface P, and projected in a regular layout T, the texture mapping operation can be used over a 3D voxelization, to create a 3D spatial directory, capturing the information on the cells of the parametric domain T, and storing it into localized lists pointed by a grid-of-voxels structure.

However, since most of the voxels would be empty (except in the 2D surface layer), this option is far too wasteful, so we may decide to create a 2D spatial directory D to store the parametric surface information from T.

As illustrated in Figure 3.14 (*Middle*), the projection line of the mapping $\mathcal{P}$ maps both the point v^{P_s} and the corresponding point v^P onto the same cell c of T. This is an important feature that allos us to define the 2-dimensional spatial directory D with the cells of T, and to to provide the inverse map $\mathcal{I}$ (see Section 3.2.4). This transformation is done with a mapping $\mathcal{M}$ (i.e. $T \overset{\mathcal{M}}{\longrightarrow} D$), as described below.

Depending on the mesh parameterization used in the mapping step $\mathcal{C}$, a convenient mapping $\mathcal{M}$ should be used to unfold the cell layout T in a 2D grid. For instance, a cylindrical or a spherical map $\mathcal{C}$ requires a cylindrical or spherical projection, as the mapping $\mathcal{M}$ to create the spatial directory D in the plane (see Figures 3.15 (*Middle*), 3.16 (*Middle*)).

In this way each point v^{P_s} is mapped into a point v^D in a given cell c of the spatial directory.

In the case of a polycube mapping $\mathcal{C}$, it requires a small 3D look-up table [THCM04] to map the cells of the polycube parameter domain T into the planar grid D, as shown in Figure 3.17 (*Middle*).

Each cell c of the spatial directory D contains a localized list of parametric triangles stored in L.

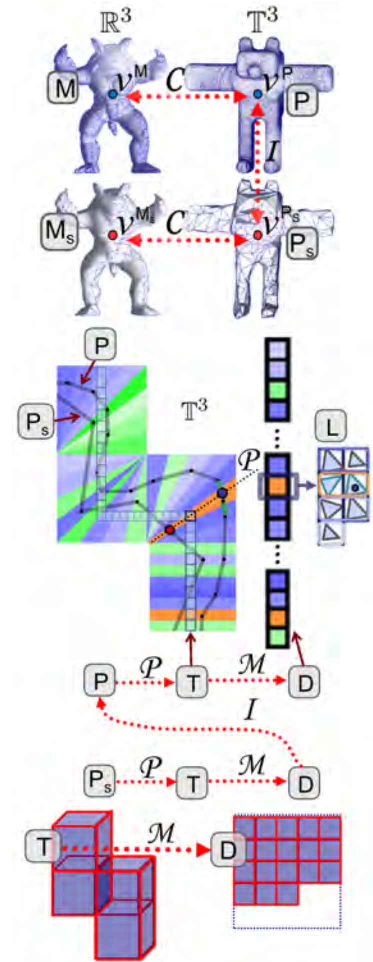


Figure 3.17: (*Top*) The points v^M of the original mesh M, and v^{M_s} of the simplified mesh M_s are mapped by $\mathcal{C}$ to corresponding points v^P and v^{P_s} in a polycube parametric surface. (*Middle*) The two corresponding points v^{P_s} (shown in red) and v^P (shown in blue) are projected by $\mathcal{P}$ into a same cell of T. (*Bottom*) The triangles of P that overlap a same cell of T are mapped by $\mathcal{M}$ and stored in localized list in L of a 2D spatial directory D. This allows to define the inverse map $\mathcal{I}$.

Thanks to an inverse mapping $\mathcal{I}$, for any valid decoupling coordinates v^D in the spatial directory D, we can identify the point v^M, and shading attributes v^A (e.g. vertex color, texture coordinates, etc.) of M, and can map them onto M_s, by an inverse mapping $\mathcal{I}$.

In general, an important feature is that the mapping $\mathcal{M}$ should provide a uniform sampling distribution of T over D. Otherwise, if $\mathcal{M}$ has a too large area distortion, some cells of the spatial directory would correspond to larger regions of the parametric surface, while others would be undersampled. In the end, this can harm the efficiency of the parallel queries evaluating the inverse map (see Section 3.2.4).

In following sections we describe the data structures (D, L and A) that define the inverse map $\mathcal{I}$, how the map is constructed, and the querying process.

3.2.2 Data structures (D, L, A)

The proposed data structure defines the inverse map $\mathcal{I}$ to break the dependence between the triangle mesh connectivity and the original shape and shading attributes. The inverse geometric textures representation is encoded with this particular setting:

- The *Directory Texture* (D) is usually a small grid defining a spatial directory in a texture, where each cell encodes the location of the corresponding list in the List Texture, together with the length of the corresponding list, with 0 representing an empty list.

- The *List Texture* (L) is encoded into another texture, where each cell list is consecutively stored. As the lists have variable-length, we simply concatenate all lists in the raster-scan order of the 2-D spatial directory D (see Section 2.4.1), letting the 2D spatial directory D contain pointers to the start of each cell list. Naturally, we also coalesce identical lists to allow for sharing between cells. Each list record contains parametric coordinates (v_1^P, v_2^P, v_3^P) for one triangle, and the pointer to other shading attributes stored in A.

- The *Attribute Texture* (A) stores the additional surface attribute information for each original triangle (vertex color, *artist-provided* parameterization coordinates, etc).

The storage required by IGT is the storage of the spatial director (i.e. the texture grid D) plus the locations of the list records L, and the compacted surface attributes in A. Usually, the size $c_w \times c_h$ of the grid D is much smaller than the triangle count N from the original triangle mesh. The total storage required by the data structures (D, L) for a given mesh M with N triangles, is $O(2N)$ [1].

The different texture sizes of D, L and A can be found for various examples in Table 3.1 of Section 3.4, but the overall memory requirements for IGT are fairly small (ranging up to 4 MB for the gargoyle model in Table 3.1.

[1] In our paper [GP08], we had a more complex linked data structure for the spatial directory. We simplified the layout as described here to achieve a better balance between the query evaluation time and the data structure memory consumption.

3.2.3 Constructing the inverse map $\mathcal{I}$

The generation of the data structures (D, L, A) needed for IGT is performed in an off-line pre-processing stage. This is done by verifying every parametric triangle (v_1^P, v_2^P, v_3^P) over T, and generating a list record in a conservative manner, even if the triangle slightly touches a cell c of T.

First, the triangles in M are mapped to T by $M \xrightarrow{\mathcal{C}} P \xrightarrow{\mathcal{P}} T$ and are checked for intersection following the projection coordinates v^T, following the projection line of the mapping $\mathcal{P}$ with the corresponding cell region c of T. Note that the composite mapping $\mathcal{C} \circ \mathcal{P}$ is noted as $\mathcal{T}$ in Figure 3.18.

Next, each cell c of T is mapped to a grid location of the spatial directory D with a mapping $\mathcal{M}$ ($T \xrightarrow{\mathcal{M}} D$). The cell of the spatial directory will point to the list with the information of all the overlapped parametric triangles.

If the intersection between the parametric triangle and the corresponding cell c is not empty, then a new record is generated and added to the respective list in L, with the parametric coordinates of the triangle and a pointer to the additional attributes stored in A (see Figure 3.18).

The cost of preprocessing N triangles in the 2D directory grid space is $O(2N)$. The whole process only takes from between a few seconds to a couple of minutes even for the most sophisticated examples we have tried. However, note that there is room of improvement in this preprocessing step, taking advantage of the efficient gather and scatter operations of the lastest GPU architectures [Nvi11] to achieve a faster construction. However, our main objective is the parallel efficient inverse map $\mathcal{I}$ in the query evaluations as described in the next section.

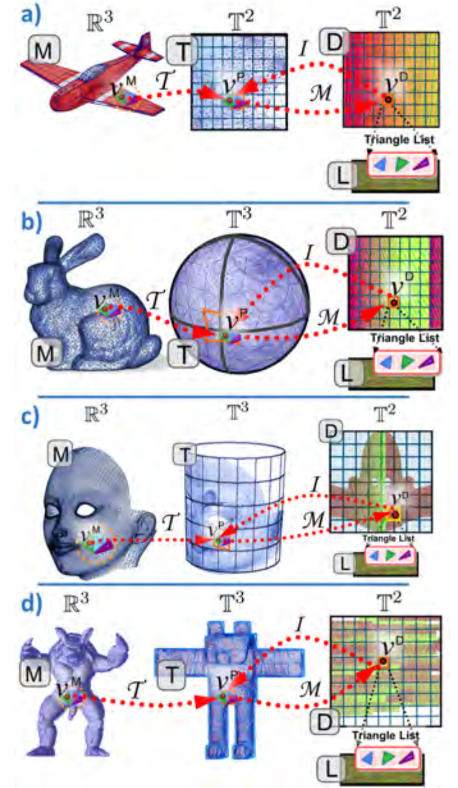


Figure 3.18: The construction of the spatial directory D and the lists L through the mapping $\mathcal{T}$ ($\mathcal{T} = \mathcal{P} \circ \mathcal{C}$) and $\mathcal{M}$.

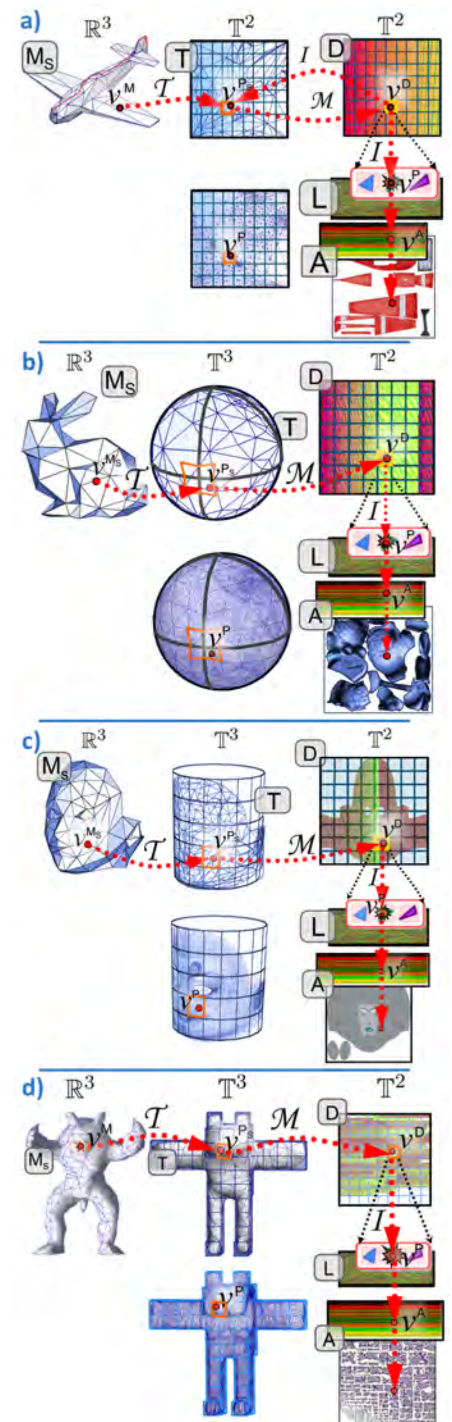


Figure 3.19: The spatial directory D allows the inverse map $\mathcal{I}$ to be performed and query the information from the high resolution mesh M. Given the projection line from the parametric coordinates v^{P_s} defined by the mapping $\mathcal{P}$, we search the intersection point v^P in any of the triangles in the corresponding localized list from L of the cell c.

3.2.4 Querying with the inverse map $\mathcal{I}$: $(D \xrightarrow{\mathcal{I}} P)$

In order to have a parallel efficient query evaluation, the composite mapping $\mathcal{M} \circ \mathcal{P}$ and the inverse mapping $\mathcal{I} = \mathcal{M}^{-1}$ should be simple, efficient and parallel friendly.

In the mesh structure M_s, each vertex v, has the geometric and parametric coordinates $(v^{M_s}|v^{P_s})$ attached. The transformation $P_s \xrightarrow{\mathcal{P}} T \xrightarrow{\mathcal{M}} D$ should be efficient in order to query the information from the high resolution mesh M with the inverse map $\mathcal{I}$ defined as localized queries in D and L (see these mappings pseudocode in Listings 3.1, 3.3 and 3.4).

When rendering a low resolution model M_s, the triangles are rasterized, generating a range of interpolated surface and parametric coordinates $(v^{M_s}|v^{P_s})$ on the triangle surface fragments. Next, a pixel shader takes the parametric coordinates v^{P_s} and the composite mapping $(v^{P_s} \xrightarrow{\mathcal{P}} v^T \xrightarrow{\mathcal{M}} v^D)$ is evaluated with the data structure (D, L). This allows querying in real-time the cell c pointed by v^D to check a small list of triangles in L, and to find the corresponding parametric coordinates v^P, and as well as the attributes v^A of the original mesh M as shown in Figure 3.19.

The inverse map $\mathcal{I}$ $(D \xrightarrow{\mathcal{I}} P)$ is performed from the spatial directory D, where given the projection line from the fragment parametric coordinates v^{P_s} (defined by the mapping $\mathcal{P}$) it searches the intersection point v^P in any of the triangles in the corresponding localized list from L of the cell c. The corresponding intersection point v^P over the projection line is found with a simple point-in-triangle method [LAM05] (see the inverse mapping pseudocode in Listing 3.5).

Once the point v^P is found, we can take its attribute coordinates v^A from A (see Listing 3.6) to compute the final interpolated shading values of the surface fragment as described in Section 3.4.2.1.

Listing 3.1: Planar mappings $\mathcal{P}$ and $\mathcal{M}$ (planar unfolding)

```
1 // Mapping Ps --\mathcal{P}--> T:
2 v_T = v_Ps;
3 // Mapping T --\mathcal{M}--> D:
4 v_D.x = v_T.x;
5 v_D.y = v_T.y;
6 cell_D = texture2D(D, v_D.xy);
```

Listing 3.2: Cylindrical mappings $\mathcal{P}$ and $\mathcal{M}$ with latitude unfolding

```
1 // Mapping Ps --\mathcal{P}--> T:
2 v_T = v_Ps;
3 // Mapping T --\mathcal{M}--> D:
4 v_D.x = atan2(v_T.y,v_T.x)/PI;
5 v_D.y = v_T.z;
6 cell_D = texture2D(D, v_D.xy);
```

Listing 3.3: Spherical mappings $\mathcal{P}$ and $\mathcal{M}$ with octahedron unfolding

```
1 // Mapping Ps --\mathcal{P}--> T:
2 // Project onto octahedron
3 v_T /= dot(vec3(1.0,1.0,1.0), abs(v_Ps));
4 // Mapping T --\mathcal{M}--> D:
5 // unfolding of the downward faces
6 if (v_T.z < 0.0f )
7 {
8   v_D.xy = (1-abs(v_T.yx))*sign(v_T.xy);
9 }
10 // Mapping to [0;1]^2 texture space of D
11 v_D.xy = v_D.xy * 0.5 + 0.5;
12 cell_D = texture2D(D, v_D.xy);
```

Listing 3.4: Polycube mappings $\mathcal{P}$ and $\mathcal{M}$

```
1 //--------------------------------------------------
2 // Mapping Ps --\mathcal{P}--> T:
3 //--------------------------------------------------
4 // Compute projection line \mathcal{P} depending on the polycube rotation and configuration
5 // Find 3D index of cubic cell
6 v_LUT = floor(v_Ps);
7 // Serialize the 3D index into a 2D index
8 // (the 3D lookup table is stored in a subpart of the 2D texture)
9 v_LUT.x = dot(v_LUT, vec3(1.0, 0.0, 16.0));
10 // Texture access to the 3D lookup table required for the mapping \mathcal{M}
11 v_LUT = v_LUT + vec3(0.5, 0.5, 0.0);
12 v_LUT = v_LUT * text_coord_normalizer + vec3(0.0, 0.0, 0.0);
13 v_T = texture2D(pcmTexture, v_LUT.xy);
14 // The rotation is stored in 1..5 bits of map.z
15 int rot = int(fmod(v_T.z * 255.0, 32.0)); // 31
16 // Configuration is stored in bits 6,7,8 of map.z)
17 int configBits = int(v_T.z * 255.0 / 32.0); //6
18 vec3 a, s, aOut, sOut;
19 // Get cube integer coordinates
20 vec4 cube_icoords = floor(v_Ps);
21 decodeFrom(rot, a, s);
22 P.xyz = apply(v_Ps_norm.xyz, a, s);
23 P.xyz = field(P.xyz, configBits);
24 P.xyz = antiRotateVectorBitWise(P.xyz, rot);
25 P.xyz = ((configBits&1) && !(configBits&2) && !(configBits&4)) ?
26         vec3(-P.x, -P.y, -P.z) : P.xyz;
27 //--------------------------------------------------
28 // Mapping T --\mathcal{M}--> D:
29 //--------------------------------------------------
30 vec4 v_D;
31 // Get the normalized [0..1)^3 of the fragment coordinates inside each cube
32 vec4 v_Ps_norm = fract(v_Ps);
33 // Compute the coordinates from the projection linne onto the spatial directory D
34 v_D.xyz = subTextCoord2d(v_P_norm.xyz, rot, configBits);
35 v_D = v_D * text_coord_normalizer_on_TS_0 + text_coord_normalizer_on_TS_1;
36 v_D = v_T * text_coord_normalizer_on_TS_times_255 + v_D;
37 cell_D = texture2D(D, v_D.xy);
```

Listing 3.5: Inverse mapping $\mathcal{I}$

```
1 //-------------------------------------------------
2 // Inverse mapping \mathcal{I} through the spatial directory (D,L)
3 //  D --\mathcal{I}--> T (\mathcal{I} = \mathcal{M}^{-1})
4 //-------------------------------------------------
5 cellListSize = cell_D.w;
6 texelSector = round(cell_D.z);
7 texelSectorUV.x = round(texelSector/16.0)
8 texelSectorUV.y = round(fmod(texelSector, 16.0));
9 cellListUV = ( round(cellListInfo.xy) + round(texelSectorUV.xy * 256.0) ) *
      cellsListsAtlasSize.zz;
10 for (iPolygon = 0; iPolygon < cellListSize; iPolygon++)
11 {
12   step = iPolygon*cellsListsAtlasSize.z;
13   polygon = texture2D(L, vec2(cellListUV.x+step, cellListUV.y), 0.0);
14   v_P0 = unpack(polygon.x);
15   v_P1 = unpack(polygon.y);
16   v_P2 = unpack(polygon.z);
17   inside = intersect_triangle(v_Ps, P, v_P0.xyz, v_P1.xyz, v_P2.xyz, t, u, v);
18   if (inside > 0.5) { break; }
19 }
```

Listing 3.6: Fetching attributes from A

```
1 // Get the attribute values from A
2 int attributeCoord = polygon.z;
3 v_A0 = texture2D(A, convert2DCoords(attributeCoord));
4 v_A1 = texture2D(A, convert2DCoords(attributeCoord + 1));
5 v_A2 = texture2D(A, convert2DCoords(attributeCoord + 2));
```

3.3 Applications

Some applications benefit by decoupling surface attributes from the simplification process, reducing the work to be done by developers once the modeler has delivered a high resolution triangle mesh M. Information is associated to a reference mesh at three different levels: at the triangle level, as with *per-face* constant colors; at the vertex level, as with *per-vertex* normals or ambient occlusion factors; and at the texture map level.

In Figure 3.20, we can see the Aikobot Robot model, which is an artist-created model provided with a multi-charts parameterization $\mathcal{MT}_a$. In this model, there are lots of color and texture discontinuities, which pose a serious problem to simplification methods trying to preserve surface attributes.

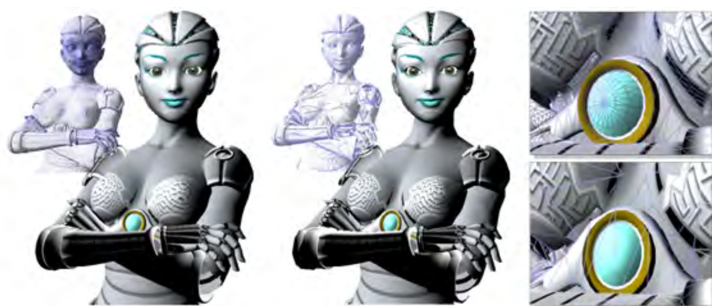


Figure 3.20: (*Left*) An artist-created model fully rendered and in wireframe (197150 triangles, 62 FPS). (*Middle*) The strongly simplified model (35834 triangles -82% reduction-, 165 FPS) rendered with our technique. (*Right*) Notice that in the inset, despite drastic simplification, the global appearance is maintained from the original model (*Top*) to the simplified one (*Bottom*).

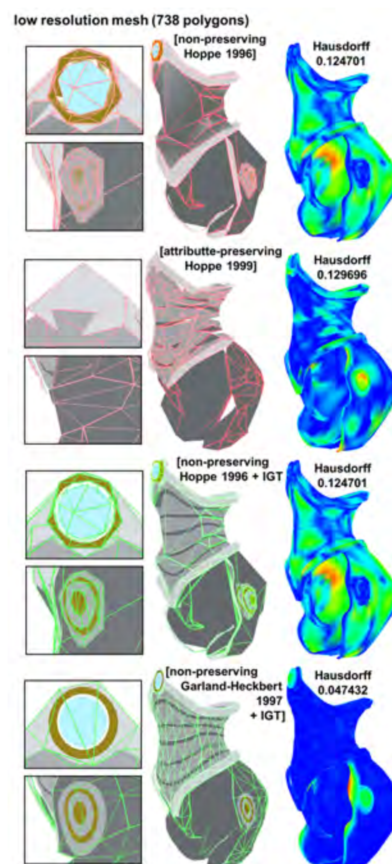


Figure 3.22: Solid color preservation: the Aikobot body armor model has artist-painted colors directly at the vertex level. With traditional methods, either the colors or the mesh quality suffer, while IGT allows their preservation (738 triangles, 650 FPS).

3.3.1 Vertex colors

As mentioned in Section 3.1, simplification methods without attribute preservation tend to mix colors at the boundaries of triangles with different solid colors. On the other hand, some attribute-preserving methods modify the shape of the boundary when simplifying the edges connectivity around it. Instead, IGT preserves sharp solid boundaries between colors or texture borders.

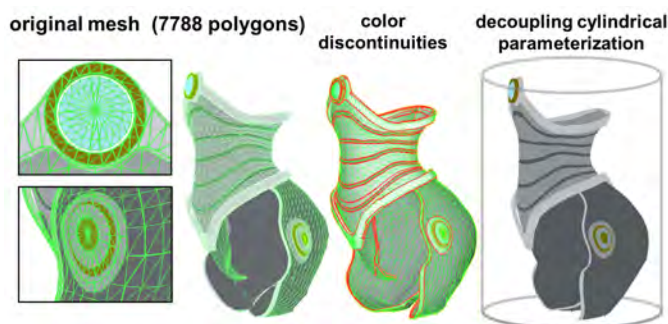


Figure 3.21: The color attributes on the Aikobot body armor defines complex color boundaries.

For example, in Figure 3.22 we can see a comparison of some simplification methods used with and without IGT. In this case, a simple cylindrical parameterization was used as the mapping step $\mathcal{C}$, and we kept the original per-vertex colors assigned by the artist in the texture buffer A.

In Figure 3.22 the effects of using a simplification method [Hop96] with the attribute preservation disabled can be clearly seen. Also, another example of a method that performs the attribute-preserving simplification can be seen [Hop99]. In the first case coloring detail is almost destroyed, while in the second it is preserved, but with a much lower geometric quality.

However, when a traditional simplification method like [GH97] is used in combination with IGT, the best of both worlds are achieved: a good geometric quality with correctly preserved per-vertex colors without constraints.

3.3.2 Volumetric and procedural texturing

Volumetric functions [KK89] are usually point-wise evaluated on the surface of the reference model M. For lower resolution models M_s the simplified triangles span a different region of the texture volume, producing noticeable changes in appearance. One solution for consistently applying a volumetric texture onto M_s is to resample it for M, transfer that information to a texture and use it for the successive LODs [CH02b]. However, IGT solves this problem, directly using the attributes from the reference model M to procedurally texture all the coarser levels (see Figure 3.23).

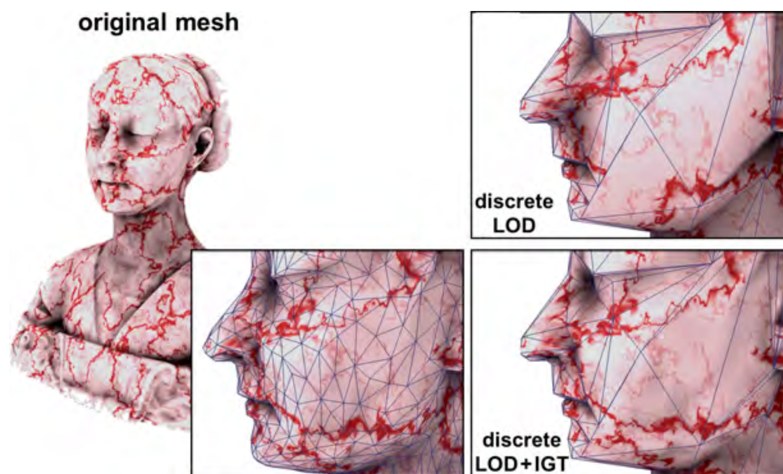


Figure 3.23: IGT correctly preserves the shape and appearance of solid textures, while direct application of the texture onto different LODs leads to non-preserving texturing (304 triangles, 535 FPS).

3.3.3 Texture mapping

As mentioned, one of the key points of IGT is its ability to decouple surface attributes and parameterized textures from simplification. As an example, in Figure 3.24, the head of the Aikobot Robot model was independently parameterized and textured for each submesh using a multi-chart method as the *artist-provided* parameterization $\mathcal{MT}_a$. Simplification of the model with traditional techniques results either in mixed textures or lower quality meshes due to the complex chart discontinuities to be preserved.

With IGT, we provide a *decoupling* parameterization $(\mathcal{M} \circ \mathcal{P} \circ \mathcal{C})$ that allows the simplification of a model by any desired method. In the Figure 3.24, the helmet was parameterized with a spherical parameterization and the head with a cylindrical one, in the mapping step $\mathcal{C}$. As can be seen, the combination of those seamless parameterizations with geometric simplification methods leads to a simplified model with a high quality mesh and with correctly preserved textures (see a further example in Figure 3.26).

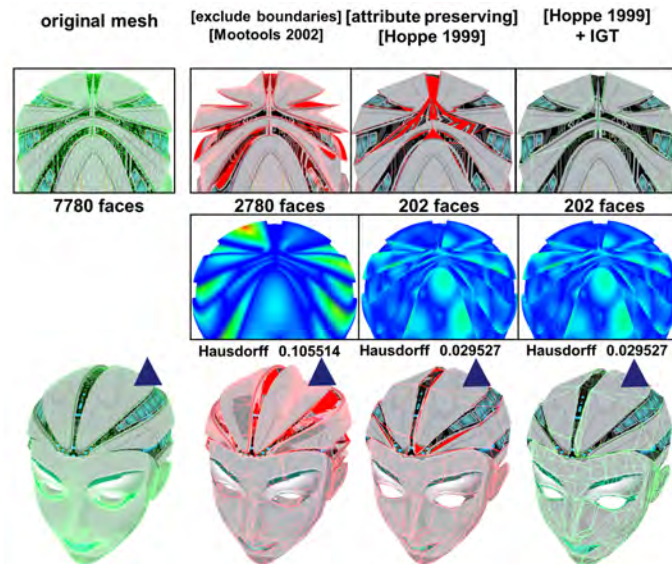


Figure 3.24: IGT decouples texturing and simplification, allowing simplification of a textured model (left column) with the most convenient method. Polygon Cruncher (second column) cannot simplify the helmet to more than 2780 triangles when excluding the attributes boundary edges, while IGT allows simplification to coarser levels (202 triangles, 450 FPS) and provides better geometric and attributes preservation. In the insets, a rear view of head details can be observed for each case.

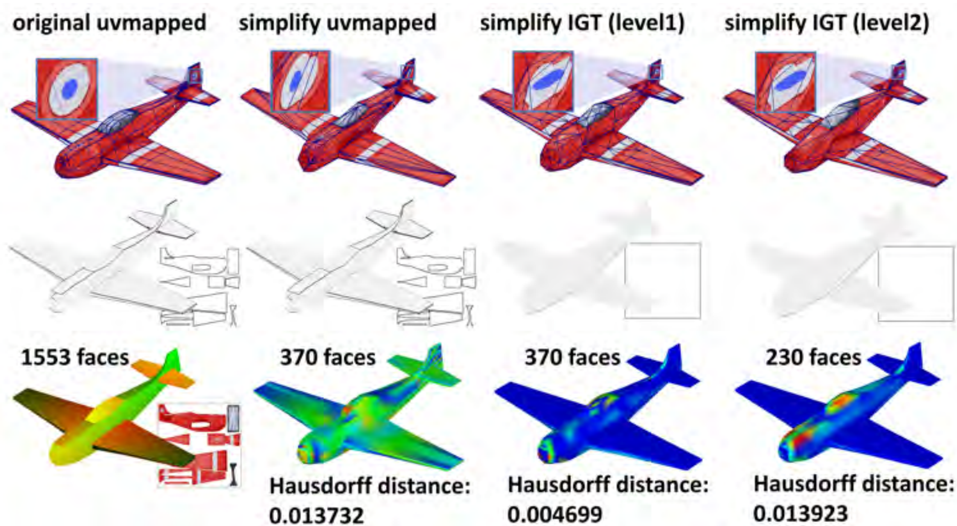


Figure 3.25: (*Left*) Example showing a plane model simplified [Moo02] preserving the boundaries of the original parameterization in the left. (*Right*) simplification with [Moo02] only requires the introduced geometry image mapping $\mathcal{C}$ to be preserved as shown in two coarser LODs. The last two show similar attribute preservation quality and better geometric preservation.

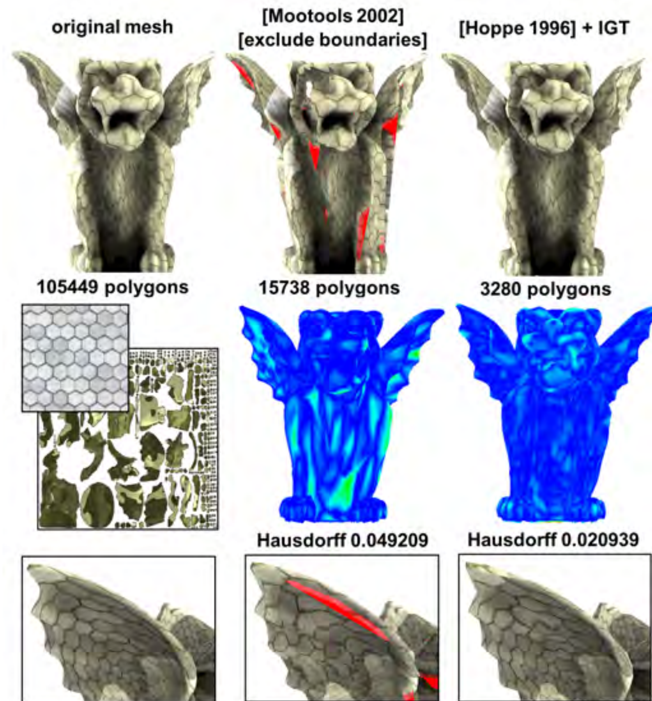


Figure 3.26: Tiling over a texture domain with IGT. From left to right, the original model (131072 triangles), the model simplified with Polygon Cruncher (only up to 15738 triangles), and the model simplified with a progressive mesh [Hop96] method (3280 triangles) combined with IGT.

3.3.4 Texture transfer

In applications that do not require dynamically changing the LOD, a resample of surface attributes is performed to map the fine surface details and shading attributes to a simplified mesh M_s by simple projection (e.g., based on coarse mesh normals [TCS03]). IGT provides access to the original reference model information A through the inverse parameterization $\mathcal{I}$, to directly resample the information into the planar grid D. This can help avoiding resampling ambiguities that may result in concave objects with normal-based methods.
Our texture transfer method is used to create a low resolution resampled texture of the original shape and shading information from M, to be used in the coarsest level-of-detail in the *shader LOD* strategy described in Section 3.4.2.2.

3.4 Results

In Table 3.1, results for different combinations of input models and parameterizations are presented. In particular, we have implemented the mapping step $\mathcal{C}$ with combinations of cylindrical [HSV05] and spherical mappings [PH03], Least Squares Conformal Maps (LSCM) multi-charts [LPRM02], Angle Based Flattening (ABF++) multi-charts [SLMB05], Polycube-maps (PCM) [THCM04], and Geometry Images (GI) [GGH02b].

From this table, we can see that the storage needs of IGT are small, almost always requiring less than a medium-resolution 1024^2 normal map (RGBA8 encoded, 4 MB).

3.4.1 Geometric & attributes preservation with IGT: experimental evaluation

As mentioned in Section 3.2.1.1, the coordinates v^P generated with mapping $\mathcal{C}$ are stored in M, and processed along the geometric simplification process, where we have successfully applied a spherical mapping to the Aikobot helmet, a cylindrical mapping to both the Aikobot body armor and the head, and a geometry image for the Aikobot body, as seen in Figures 3.24 and 3.22.

In the case of mappping $\mathcal{C}$ for helmet, head and body armor, they are defined by seamless mesh parameterizations, hence simplification algorithms such as [GH97] which can be generally applied without requiring any attribute preservation constraint.

For the rest of the Aikobot body, the simplification method used [Moo02] was forced to preserve the charts boundaries of the parametric surface P during the process.

In general, different parameterizations have different behaviors when simplified, not only restricting the kind of simplification method to choose, but also the quality of the resulting mesh, as shown in Figure 3.27.

The only requirement imposed on the combination parameterization / simplification algorithm is that the latter should preserve, if present, the chart boundaries of the parameterization at all LODs.

Multi-chart parameterizations, which are most commonly used by artists, produce seams, so they require specific constrained simplification algorithms to preserve those seams. Therefore, the model cannot be simplified with good quality, as the simplification algorithm (e.g. Polygon Cruncher) must preserve the seams to avoid texture artifacts (see Figure 3.27 (a) where the algorithm cannot simplify the model further than 15440 polygons).

In Figure 3.27 (b), it shows what happens when the texture is transferred to an Iso-charts parameterization [ZSGS04]: simplification quality is good as long as few charts are used, but significant texture distortion and blurring is observed in the close views, compared to the original texture used in Figure 3.27 (a).

But, when IGT is used to query the original artist parameterized textures, with Iso-charts [ZSGS04] as mapping $\mathcal{C}$ to fetch the original texture atlas parameterized with LSCM [LPRM02] (as shown in Figure 3.27 (c)), we can see almost perfect attribute preservation for medium quality models and a significant improvement for extremely simplified ones, though the geometric quality is still constrained as can be seen n the armadillo hands and fingers.

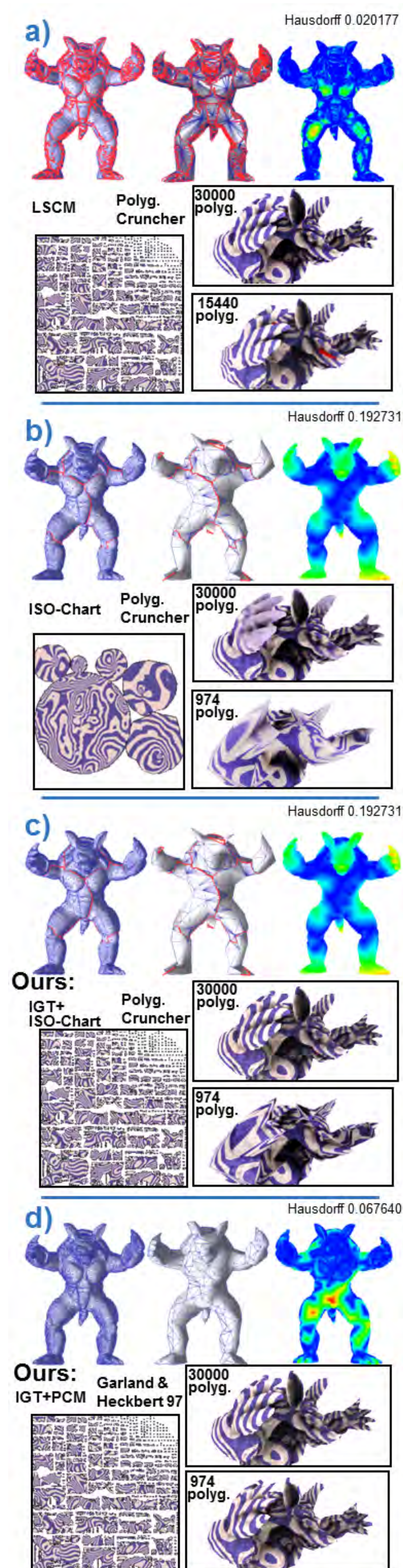


Figure 3.27: Sensitivity of IGT to different decoupling parameterizations.

Finally, Figure 3.27 (*d*) shows a much better combination: IGT with artist parameterization (LSCM), and Polycubes (PCM) in mapping step $\mathcal{C}$. Then the model can be simplified even with an unconstrained method such as the seminal QEM approach [GH97], providing excellent results in both attributes preservation and geometric quality (as shown with the Hausdorff distances).

All these comments apply even for a complex model, where IGT would provide LODs with correctly preserved high resolution details in a completely automatic manner, without requiring a manual fine-tuning step.

There is one important aspect of our technique that must be mentioned: although IGT is independent from both the type of shape and shading attributes and the simplification method used, it must be clear that simplification methods are not parameterization-independent.

IGT can be successfully combined using those ones in the mapping step $\mathcal{C}$, providing almost perfect attribute preservation and a significant improvement even for extremely simplified models.

We can conclude that the best combination with IGT is using a seamless parameterization, such as spherical, cylindrical or polycube-maps, which tolerate most simplification methods basically because the texture coordinates can be calculated from the original vertex coordinates for any LOD, without requiring a manual fine-tuning step (See Figure 3.27, lower row).

3.4.2 Query evaluation

Here we describe the different properties of the query operation performed through the inverse map $\mathcal{I}$, showing their evaluation cost.

3.4.2.1 Filtering

IGT performs a shader evaluated filtering over the spatial directory, retrieving the nearest samples (5 in our case) from the corresponding cells, and then blending them (see Figure 3.28). As suggested by [LH06b], [NH08] and [THCM04], our findings produced the same shader filtering performance factors: about x3.9 when using 4 samples per pixel.

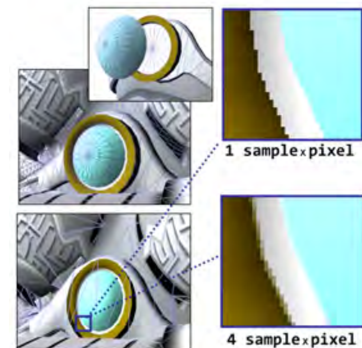


Figure 3.28: Shader evaluated filtering of the per-vertex attribute information queried from the high-resolution mesh M (e.g. vertex colors).

In the presence of an attribute texture atlas, the MIP-map level selection is defined by the approximate derivative functions (*dFdx* and *dFdy* in a GLSL pixel shader) of the fragment attribute texture coordinates v^A, once queried by the inverse map, to map the values with hardware filtering.

However, when an object is viewed at a far distance from the screen, many of the cells of the spatial directory are mapped into individual screen pixels, and the filtering breaks down, because noise can be produced if, arbitrarily, one or other of the many cells located in a same pixel get evaluated in very fast distant views of an object. In such cases, for distant views one must instead transition to a conventional texture with a MIP-map pyramid (without the spatial directory query evaluation) as we describe in Section 3.4.2.2.

3.4.2.2 Shader LOD

IGT shader evaluation should be used from close to medium-range distances, as its evaluation is more complex than a standard texture map operation. We have implemented a shader LOD technique to switch to a simpler shader as soon as the evaluation of most fine detail samples can be neglected.

When the observer cannot distinguish one another, the shader LOD, using predefined range distances, swaps to the simpler shader that only fetches the surface attributes from a precomputed resampled texture, as described in Section 3.3.4. In this way, the MIP-maps of this texture can then be used for further distances, resulting in a smooth minification of the attributes without noise.

3.4.2.3 Shader performance

It is important to mention the influence of the spatial grid size in the requirements and performance of IGT. As expected, the lists of the cells will get shorter on average, as the spatial grid increases resolution, but at the expense of an increased memory cost.

For the examples shown in Table 3.1 the average list length is short (4 triangles). However, this can vary for an irregular triangle density.

For instance, in certain cells the list length has a lower bound, as any cell that covers a vertex shared by, for instance, 6 triangles will have (at least) a length of 6 entries.

In analyzing the search evaluation cost there are two costs to take into account the accesses to D, and the inclusion tests, which test if the parametric coordinates v^{P_s} satisfy the query matching the original surface point v^P.

At most, each surface fragment performs a single query in a cell, requiring one memory access, and an average of between $3 and 5$ memory accesses in L to find the match (see Figure 3.29). Finally one additional memory access is required to get the attribute information from A.

Nevertheless, the histograms displayed in Figure 3.29 (*Middle*) show the correlation of the performance improvement achieved when larger spatial directories are used, because the cell list lengths became shorter and it translates to better query performance illustrated in the performance plot measured in frames-per-second (FPS).

In general, we can observe that our query evaluation performance is bound by the cost and number of memory accesses needed to obtain the original attribute samples from M.

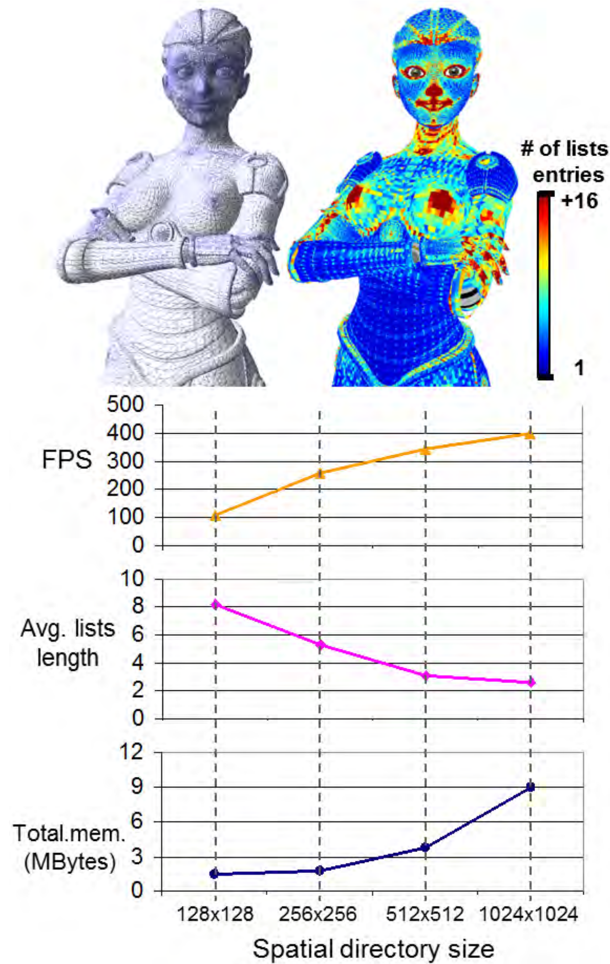


Figure 3.29: (*Top*) The Aikobot high resolution model on the left, and the heatmap visualization of the spatial directory list lenght on each surface cell (grids size 256×256). (*Bottom*) There is a trade-off between the shader performance evaluation and the memory footprint of the spatial directory, because the larger the spatial directories are, the shorter the cells list, providing a faster shader evaluation.

3.4.2.4 Level-of-detail experimental evaluation

We can get information of the overhead incurred when using IGT by doing a comparison between models simplified and rendered with different techniques to achieve a given fixed frame-rate, as shown in Figure 3.30 where all methods produce a similar visual quality without geometry or visual artifacts, from each observer distance.

In this comparison, we evaluate the performance of IGT with respect to rendering the model M simplified with different LOD techniques [Moo02, GH97, TCS03]. The renderings are made at 1024×768, on a Quad Core Pentium IV with a GeForce 8800 card.

In all the view distances shown in Figure 3.30, we analyzed the performance overhead of the shader query evaluation on IGT by selecting (with an orange box) the LOD changes when the replacement between LODs. In all the compared techniques, LOD changes are selected when they produce the smallest the visual difference (i.e. also known as *Late Switching* [GW07]), and using the IGT LOD polygon count to match a fixed frame rate.

It is important to mention that the better geometry and attribute preservation with IGT allows LOD replacements to be done earlier than with other techniques, providing better visual fidelity.

In general, better replacement strategies should be used [GW07] to avoid the *popping effect* when switching between discrete LOD, or alternatively CLOD techniques can be used as permitted with IGT over irregular triangle meshes, or with our new the representation regular representation described in chapter 4.

In Figure 3.30 (*Bottom*), it can also be observed that the memory footprint overhead introduced by IGT to store the parametric coordinates v^{P_s} in the LODs (M_s) and the data structures (D, L, A) (as shown in Table 3.1), is small compared to the other techniques ([LODs M_s] 4.3MB + $[D, L, A]$ 0.81MB = 5.1MB).

This reinforces the idea that IGT can present low overhead models much earlier than other techniques, as with the normal projection [TCS03] shown on the right in Figure 3.30), which requires a different texture map for each LOD.

As a model covers less and less pixels, less pixel shaders are needed, the overhead of IGT is reduced smoothly.

Dist.	[Mootools02] (exclude bound.)	[GH97] + IGT	[GH97] + [TH03]	FPS	Ratio IGT/Ref
10.59	6628 polyg. LOD1	3519 polyg.	6628 polyg. LOD1	45	1.9
10.59	3274 polyg. LOD2	1321 polyg.	3274 polyg. LOD2	87	2.5
10.59	2780 polyg. LOD3	1103 polyg.	2780 polyg. LOD3	95	2.5
26.51	6628 polyg. LOD1	3642 polyg.	6628 polyg. LOD1	21	1.8
26.51	3274 polyg. LOD2	1652 polyg.	3274 polyg. LOD2	42	2.0
26.51	2780 polyg. LOD3	1521 polyg.	2780 polyg. LOD3	46	1.8
50.05	6628 polyg. LOD1	5370 polyg.	6628 polyg. LOD1	16	1.2
50.05	3274 polyg. LOD2	2060 polyg.	3274 polyg. LOD2	32	1.6
50.05	2780 polyg. LOD3	1729 polyg.	2780 polyg. LOD3	35	1.3
139.9	6628 polyg. LOD1	6628 polyg.	6628 polyg. LOD1	19	1
139.9	3274 polyg. LOD2	3038 polyg.	3274 polyg. LOD2	36	1.1
139.9	2780 polyg. LOD3	2801 polyg.	2780 polyg. LOD3	39	1.1
264.1	6628 polyg. LOD1	6628 polyg.	6628 polyg. LOD1	22	1
264.1	3274 polyg. LOD2	3274 polyg.	3274 polyg. LOD2	42	1
264.1	2780 polyg. LOD3	2780 polyg.	2780 polyg. LOD3	46	1
Mem.	4.26 MB	5.11 MB	7.98 MB		

Figure 3.30: Experimental evaluation of the performance overhead and the memory footprint required by IGT compared to two other LOD techniques: [Moo02] (excluding boundaries from simplification), and [GH97] + [TCS03]. The polygon count of IGT LODs was adjusted to match the frame rate of the compared metrics showing we can provide similar visual quality even with lower resolution geometries.

In another experiment, shown in Figure 3.31 we do a performance comparison between the different LOD techniques without constraining to a fixed frame rate. We can observe a smoothly improved speed for IGT, which can be inferred from the slope in the curves that allows early switching the given LODs with IGT in Figure 3.31.

Traditional simplification techniques, which produce models without requiring special shaders, need to process a constant number of triangles for longer distance intervals to avoid the shape and shading artifacts on the LODs, making the performance curves quite flat.

Dist.	[GH97] + IGT		[Mootools02]		[GH97] + [TH03]	
10.5	76	LOD3	46	LOD1	46	LOD1
26.5	100	LOD5	43	LOD2	48	LOD3
50.5	143	LOD6	32	LOD2	58	LOD4
139.9	304	LOD6 **(*)**	36	LOD3	292	LOD5
264.1	343	LOD6 **(*)**	44	LOD3	344	LOD6

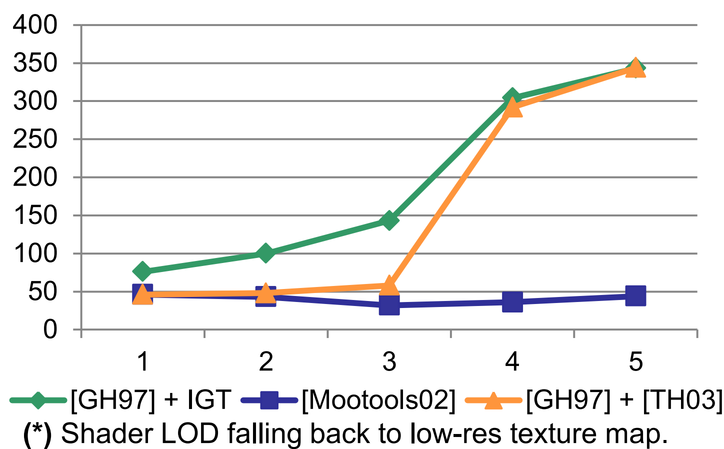


Figure 3.31: Framerates of IGT when compared with two other traditional discrete LOD methods [Moo02] (excluding boundaries from simplification), and [GH97] + [TCS03].

Models	Polyg. count (triangles)	IGT param. ($\mathcal{C}$)	Artist param. ($\mathcal{MT}_a$)	Dir. tex. (D / RGBA8)	Lists tex. (L / RGBA32)	Attrib. tex. (A / LA16)	Mem. (MB)
Armadillo	30000	PCM	LSCM	128x128	512x256	512x256	3.06
Gargoyle	131072	GI	LSCM	128x128	512x256	512x512	4.06
SpaceShip	15338	ABF++	LSCM	64x64	256x256	256x128	1.26
Laurana	10000	PCM	Procedural Tex.	128x128	512x256	512x128	2.43
Bunny	15000	PCM	LSCM	128x128	256x256	256x256	1.56
Aikobot Armor	7798	Cylindrical	—	128x128	128x128	128x128	0.43
Aikobot Helmet	6628	Spherical	LSCM	128x128	256x128	256x128	0.81
Aikobot Face	39204	Cylindrical	LSCM	128x128	512x256	512x256	3.06
Aikobot Body	49858	LSCM	LSCM	256x256	512x320	256x480	3.68

Table 3.1: Memory usage information for various examples. Here, the acronym meanings are: LSCM: Least Squares Conformal Maps, PCM: Polycube-maps, ABF++: Angle Based Flattening, GI: Geometry Images, RBGA8: a standard 4-byte/texel format, RBGA32: four channels 16-byte/texel format, LA16: two channels 4-byte/texel format,

3.5 Discussion and limitations

In this section we provide a brief discussion of the possibilities and the limitations of IGT.

3.5.1 Sparse and inconsistent topological geometries

The proposed data structure (D, L, A) is defined by the decoupling parameterization $\mathcal{D}$ in order to break the dependence between the triangle mesh connectivity and the original shape and shading attributes.

Furthermore, an important feature is that the mapping $\mathcal{M}$ allows a uniform unfolding to be distributed across the spatial grid D in $\mathbb{T}^2$ for almost any 2-manifold mesh M. In this way, most of the cell locations of the grid will be addressing local attributes descriptions.

However, in the case of a very sparse or topologically inconsistent triangle mesh M, with shape and shading attributes, they cannot be mapped by a mesh parameterization $\mathcal{C}$ into a suitable parametric surface P for our needs.

Another solution would be to define a hashing compacted grid directory D, where a sparse set of cells would replace the parameter domain P in a packed hash table as proposed in Chapter 5.

3.5.2 Bijectivity limitations in the mappings $\mathcal{C}$ and $\mathcal{P}$

IGT can present a problem when applied to non-bijectively parameterized objects either in the mapping steps $\mathcal{C}$ or in some particular cases produced by the simplification process. A drawback of our technique is that we do not strictly enforce the bijection between the original mesh M and any simplified mesh M_s generated with any given simplification method.

In the mapping step $\mathcal{P}$, when more than one triangle cover the same point v^P in the parametric surface P, IGT can only resolve the ambiguity with a heuristic –like taking the closest point– as shown in Figure 3.32.

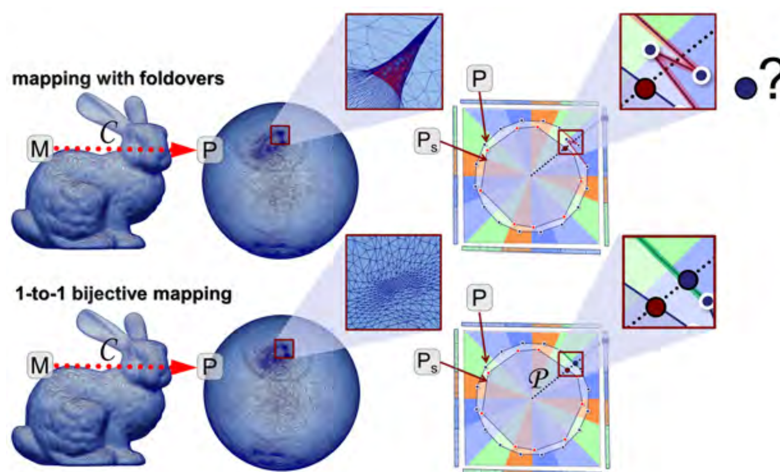


Figure 3.32: The bunny model parameterized to a spherical parametric surface P. (*Top*) Close view of a region of the spherical surface with a set of red triangles which fold over other ones, not allowing a bijective mapping in the projection line of the mapping $\mathcal{P}$. (*Bottom*) Increasing the iterative steps of the spherical mapping $\mathcal{C}$ foldovers are avoided and a bijective 1-to-1 map is provided with the mapping $\mathcal{P}$.

3.5.3 Animation compatibility

It is also worthwhile mentioning that IGT does not affect animation of the model, and can even be used to improve it. The LOD being visualized can be animated with any technique, and as IGT works entirely in the parameter domain, this means that animation would work seamlessly as long as the decoupling parameterization $\mathcal{D}$ is not modified by the process. IGT can even prove beneficial in cases where the use of traditional simplification techniques results in a LOD with insufficient triangles in joints with large stretching.

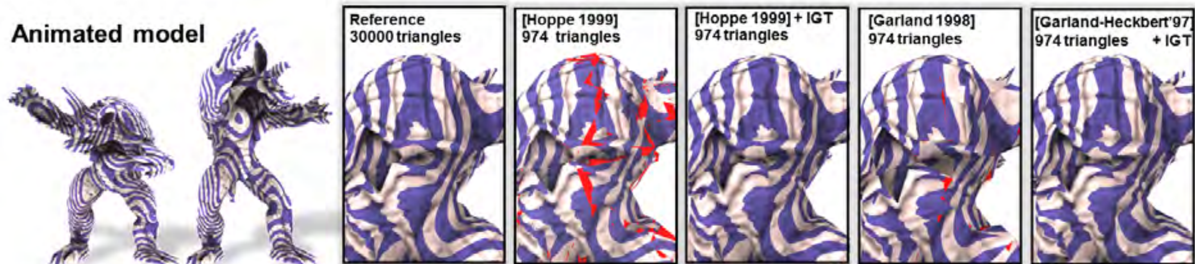


Figure 3.33: The armadillo model with skeletal animation. The closets show the attribute preservation in a LOD M_s with 974 triangles.

3.6 Conclusions

We have presented a new level-of-detail method called Inverse Geometric Textures, which allows better geometric and attribute preservation and provides a real-time mapping and evaluation over simplified triangle meshes.

IGT provides an inverse mapping in parametric space which can be used to apply information generated for a reference model onto any simplified version with DLOD or CLOD techniques.

IGT makes use of a composite mapping for simplification purposes that is *independent* of the attributes provided by the artist. This way, attributes and simplification are decoupled from each other. The best results are obtained in combination with seamless parameterizations.

For example a cylindrical, spherical, or polycube map can be used in the mapping step $\mathcal{C}$, which allows the user to choose any simplification method. Being parameterization-independent, IGT is compatible with both MIP-mapping and filtering techniques in the texture altas attributes. IGT does not provide filtering by itself, but it enables the combination of the advantages of different parameterizations in a way that was impossible before without a great deal of work (e.g. by transferring the texturing information to a texture).

This work opens three main areas of research. Adding *geometric detail* such as in Porumbescu et al. [PBFJ05] seems a logical next step, as it would allow addition of detail to the *reference* model, not only the simplified versions. Also, allowing animation in the triangles of the reference model stored in the parameter domain to be mapped on the lower quality models would permit introduction of interesting effects, such as approximate facial animation.

Finally, geometry compression techniques that would allow even higher resolution models to be used with IGT should be studied.

Chapter 4

Editable mapping and subdivision surfaces

Shape representations require *simplicity* and *regularity* on the mesh structure for many computer graphics applications. This chapter presents a framework with sketch-based editable mapping operations to map complex shapes onto high quality cube-based parametric domains. The provided representation is specially useful for hardware parallel tessellation with subdivision surfaces, displacement mapping and other modeling applications.

Many digitally captured and modeled objects generate surfaces defined by dense and irregular triangle meshes (see Figure 4.1). At the same time, many applications require a shape data structure with better simplicity, regularity, and a compact, yet flexible, mesh representation. The main reason for this is that artist modelers and animators prefer to work over cleaner and higher level abstractions of a given surface, rather than with irregular dense triangle meshes, in order to ease their modeling operations.

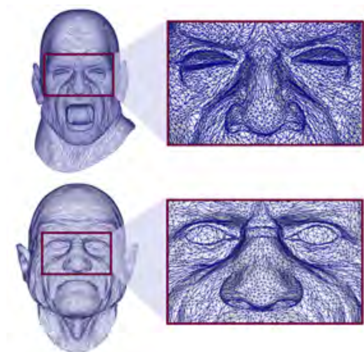


Figure 4.1: Faces of two characters modeled with triangle meshes. The mesh structure connectivities are dense, irregular and incompatible between the two.

In the previous chapter we focused on decoupling shape and shading attributes from complex irregular triangle meshes, in order to allow level-of-detail simplification methods.

In this chapter we aim to provide a new spatial data structure and a user friendly bijective parameterization to convert irregular triangle meshes into simpler quad-based representations. This setting allows a hardware-friendly LOD with subdivision surfaces with displacement mapping, and a set of modeling applications.

For instance, in the digital content creation process, modeling artists often would like to have *shape transfer* operations between different models. For this, they require user-friendly operations to map the shape and shading information into an intermediate *parameter domain*, from where they would be able to transfer this information to new models at convenience.

As an example, in character modeling, an artist may want to transfer the shape information from one character face onto another one, blending them to create variations of the two (see Figure 4.2).

Figure 4.2: (*Left*) / (*Right*): Two face models, and (*Top*) / (*Bottom*): two shape blended combinations generated from the other two.

Also, it is a well known fact that most artists prefer modeling with quads, as quad geometry provides a better flow, tessellates cleaner and deformations under animation are noticeably smoother, especially around joints [Oli06]. Therefore, any available model defined by an irregular triangle mesh, is required to be converted to a regular- or semi-regular quad-based mesh structure, in order to be integrated in the artists modeling pipeline (see Figure 4.5).

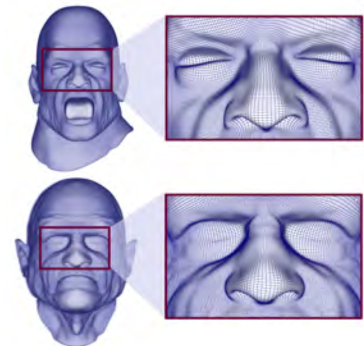


Figure 4.3: Faces of two characters modeled with converted to semi-regular quad-based mesh structures.

Surface parameterization and remeshing are used to *transfer* the shape and shading details of a given irregular mesh structure to a suitable parameter domain. The parameter domain should allow a flexible configuration to achieve a high quality surface mapping with respect to the input shape complexity and topology; i.e. avoiding to introduce surface discontinuities from cutting the mesh to map the surface into a parametric domain or losing shape detail (see Section 2.4.7.1 of Chapter 2).

The only way to avoid discontinuities is to choose a parameter domain that has both the same topology as the given input mesh and a similar shape. As an example, while any genus-0 shape is homeomorphic to a sphere parameter domain, the mapping will only show low distortion if the input surface has a quasi-spherical shape.

The natural idea is to extend the paramenter domain to more general shapes, such as a polycube (see Figure 4.4). A polycube is a natural generalization of a cubical space, which can be easily modeleted to resamble the basic shape of any arbitrary genus object. Therefore, the polycube proves to be a useful flexible parametric domain to ease the forementioned modeling operations, from irregular input models by creating a regular map of the input spatial data.

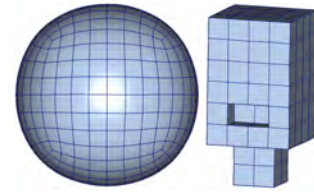


Figure 4.4: (*Left*) Quasi-spherical objects can be easily mapped to a sphere. (*Right*) More complex shapes can be more easily mapped to a configurable polycube parameter domain.

Tarini *et al.* [THCM04] pioneered the concept of polycube maps as a technique to parameterize 3D shapes to the polycube domain, useful as the parametric domain of shapes with complicated topology and geometry.

An interesting strategy to follow, would be to map the shape information from a given model onto a polycube domain with simple and user-friendly sketch-based operations, giving the artists a simple way to control the mapping of the shape features (see Figure 4.5).

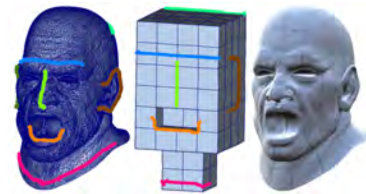


Figure 4.5: A sketch-based interface can provide control to modelers to create a mesh parameterization for modeling applications.

In this chapter, we pursue this objective with a surface parameterization and a remeshing strategy (see Section 4.2), where a consistent parameterized mapping allows to transfer not only the shape and shading details, but deformations and even animation properties of the models (see Section 4.3).

Mesh parameterization for surface remeshing plays an important role to find a bijective mapping (see Section 2.4.7.1 of Chapter 2) between the input surface, with an irregular triangle-based structure, and a regular parametric domain (e.g. the polycube).

We aim to remesh the irregular mesh structure, converting the input mesh to a semi-regular one while enforcing bijectivity. Otherwise, with a non-bijective mapping process we could lose shape information and create discontinuities in the mapping.

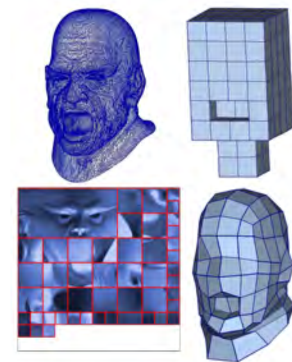


Figure 4.6: Polycube mapping and remeshing converts an irregular triangle mesh into a semi-regular quad-based coarse mesh and a set of regular charts stored in a texture atlas.

> The research challenge is to build a globally smooth parameterization, easily controllable by the user, to generate a new mesh structure with well-shaped elements, a low number of irregular vertices, and an efficient parametric representation.

The definition of the polycube parametric domain has two main objectives. On one hand, it should skip the original irregular structure complexity by separating the base shape of the object into a compact structure called *control mesh*, defined by a regular and simple polycube structure. On the other, the mapping should allow to parameterize the fine shape and shading details –guided by user sketches– onto the polycube quadrilateral faces and stored onto an image grid, as shown in Figure 4.6.

Figure 4.7: Semi-regular quad-based LODs rendered with subdivision and displacement mapping.

This specialized setting will provide a spatial data structure for efficient hardware-friendly level-of-detail (see Figure 4.7), using tessellation techniques such as subdivision surfaces (see Section 4.3.1).

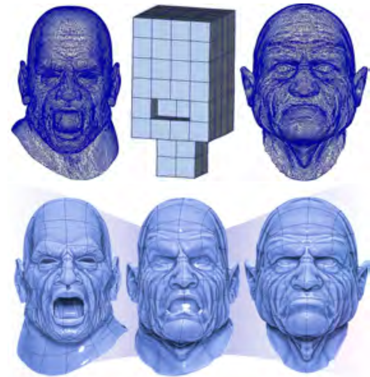


Figure 4.8: (*Top*) Sketching relevant features of two input models to be mapped on shared polycube domain allows to perform shape blending operations to create new models with shape details from the two (*Bottom*).

Compared to other global surface parameterization techniques, a polycube map has two unique features that make them promising for the mentioned graphics applications: First, the parametric domain has a regular structure that naturally supports quadrangulation, and can be easily constructed, edited and visualized. Second, the singularities (polycube corners) have fixed structures, i.e. a valence 3, 5 or 6 (see Section 2.1.3.4 of Chapter 2). The reduced number of possible singularities results in a small number of topological combinations, which is important for modern tessellation hardware (see Section 2.3 of Chapter 2).

4.1 Context: mesh parameterization and subdivision surfaces

In this section we review related works on polycube maps, surface quadrangulation, subdivision surfaces (compatible with hardware tessellation), and cross-parameterization, which provide links closely related with the proposed solution.

4.1.1 Polycube mapping

Constructing a polycube map is a challenging work. From the users' point of view, an ideal polycube mapping algorithm should have at least the following features:

- Quality: the map is a bijection with low angle and area distortion.
- User control: the user can easily control the mapping by specifying optional features on the 3D model and their desired locations on the polycube domain.
- Re-use: the user can easily modify the polycube maps and reuse resources from existing polycube maps.
- Performance: the algorithm is efficient and robust, and it is automatic with the possibility of adding editable user-specified constraints to control the map.

Table 4.1: Comparison of polycube map construction methods. Symbols: ● good, ◐ fair, ○ poor.

Features	[THCM04]	[WHL$^+$07]	[WJH$^+$08]	[LJFW08]	[HWFQ09]	Our
Map quality	●	◐	◐	●	●	●
Bijection	○	●	●	●	●	●
User control	○	○	◐	○	○	●
Editing	○	○	○	○	○	●
PC construction	○	○	○	●	●	○
Automatic	●	○	○	●	●	●
Large models	●	○	○	○	◐	●
General topology	○	○	○	○	◐	◐

There are several approaches to construct a polycube mapping [THCM04, WHL$^+$07, WJH$^+$08, LJFW08, HWFQ09]. Unfortunately, none of them has all the desired features (see Table 4.1). For instance, Tarini *et al.* [THCM04] does not guarantee a bijection, while others [WHL$^+$07, LJFW08, HWFQ09] do not allow user control. Wang *et al.* [WJH$^+$08] requires a large amount of user interaction to specify the polycube structure on the 3D model and is not suitable for large-scale models with complicated geometry and topology. More importantly, none of them allow the users to *edit* the polycube map in an easy and intuitive fashion. In general, current tools for editing in planar parametric domains can be awkward to use; and on the other hand, editing on a polycube, which more closely resembles the gross structure of the model, could indeed be simpler, especially when the user is allowed to control the rough shape of the mapping.

Tarini *et al.* [THCM04] pioneered the concept of polycube maps. They designed six projection functions that map the points inside a cell of the dual space to the polycube surface. However, their method does not produce a bijective mapping since two vertices on the same projection line share the same image, which is a fundamental feature for a large range of applications. Furthermore, their method has strict requirements on the shape of the polycube, like that the dual space should completely enclose a slightly modified version of the input model in an intermediate coordinate space.

Rather than projecting the 3D surface to the polycube, Wang *et al.* [WHL$^+$07] introduced an intrinsic approach that first maps the 3D model and the polycube to the canonical domain (e.g., sphere, euclidean plane or hyperbolic disc), and then seeks the map between the two canonical domains. The resulting polycube map is guaranteed to be a diffeomorphism. However, in this scheme it is difficult to control the polycube map, i.e., a feature on the 3D model may not be mapped to a desired location on the polycube.

In their follow-up work, Wang *et al.* [WJH$^+$08] proposed the user-controllable polycube map where the users can specify the pre-images of the polycube corners. Their method works well for shapes with simple geometry and topology, but is not feasible for complicated models since it is very tedious and error-prone to specify the polycube structure manually. Efforts have also been made to optimize the polycube map automatically.

Wan *et al.* [WYZ$^+$11] optimized the polycube mapping in the sense of area and angle distortion. But their method is still not user controllable. Instead, the method presented here provides much more user control, which allows the user to create polycube maps with a much smaller number of patches, and gives much more control over the quality of the induced subdivision surface, which is what makes this method practical for real-time rendering on modern hardware.

It is known that the polycube map quality (in terms of angle and area distortion) highly depends on the shape of the polycube. There have been some research efforts that aim to construct the

polycube automatically [LJFW08, HWFQ09].

These techniques usually break down the input model into smaller and simpler components and then use polycube primitives to approximate each one. For example, the Reeb graphs [LJFW08] and harmonic functions [HWFQ09] can be used to guide the shape segmentation. As heuristics are usually used in the segmentation and polycube approximation, these approaches may not work for models of complicated geometry and topology. As we mentioned before, none of the existing algorithms allows the users to easily edit the maps.

A well designed polycube map may serve as a boundary constraint in volumetric parameterizations. Wang *et al*. [WLL+11] constructed trivariate polycube splines for volume data. Gregson *et al*. [GSZ11] generated an all-hex mesh with a polycube domain. They proposed an automatic volumetric deformation to construct the polycube. The quality of the volumetric parameterization is directly related to the quality of surface polycube map.

4.1.2 Quadrangulation

Our work is also related to quadrangulation, which has been extensively studied in the past few years. Spectral surface quadrangulation [DBG+06] can produce simple parametric quad-based domains. However, it does not allow control over the quad alignment with respect to the geometric features. Huang *et al.* [HZM+08] extended it to take them into account. Other recent approaches such as Quadcover [KNP07], PGP [RLL+06b] and Mixed Integer [BZK09], use a set of precomputed quadrilateral-aligned features to generate a quadrilateral mesh.

However, while the generated meshes automatically can have good quality, they lack a simple user control, and may contain far too many quads to be used as parametrization domains for practical applications (see Section 4.3).

4.1.3 Subdivision surfaces

Litke *et al*. [LLS01] developed a technique to fit Catmull-Clark subdivision surfaces to a given shape within a prescribed tolerance, based on the method of quasi-interpolation. However, the control mesh of the subdivision surface must be known beforehand. Kin-Shing *et al*. [CWQ+07] proposed a fitting method from the point cloud of a given surface, to create an irregular triangle-based subdivision generated by dual marching cubes, further optimized with non-linear least squares. In contrast, our main interest is to generate a quad-based regular subdivision scheme. And, in general, none of the approaches represent a continuous parameterization, while our proposal provides this feature in a natural way.

Recently, Panozzo *et al*. [PPT+11] presented a technique for the automatic construction of adaptive quad-based subdivision surfaces. Their technique is based on a set of maps called *fitmaps*, which roughly estimate how well the mesh can be locally modeled by patches.

One difference of their method with respect to ours is that it is a ”one-click” solution, where in our case the modeler has to sketch a base polycube to control the coarse domain of the provided quad-based subdivision surface. However, this base polycube and the user strokes represent the basis of our approach, which provides high flexibility to recreate complex surfaces. Also, our technique uses vector-based displacement mapping, which allows to have surfaces with concavities represented in a simple manner without requiring a more fine base tessellation level. Finally, our approach, given the necessary user-controlling strokes, can reproduce surfaces with

fine detailed features, something which cannot be guaranteed with the scalar field of the fitmaps approach.

In the last few years there has been a growing trend to use the tessellation capabilities of modern graphics hardware to generate high-resolution models from a coarse base mesh [Bun05] [Tat08]. Loop *et al.* [LS08b, LSNCn09] presented a method for approximating subdivision surfaces with hardware-accelerated parametric patches. Our method presents a uniform, regular and user-controllable quads-only mesh with a parameterized representation suitable for subdivision surfaces, compatible with such approaches (see Section 4.3.1), and which also nicely fits into the subdivision modeling pipeline (see Sections 4.3.2, 4.3.3, and 4.3.4), with the already-mentioned benefits for the user.

4.1.4 Cross-parameterization

Another related topic is cross-parameterization. Recently, many algorithms have been developed for building the mapping between general surfaces of the same topology. A common approach is to parameterize the models over a common base mesh [LDSS99, MKFC01, PSS01, KS04]. In these approaches, the meshes are split into matching patches, each set of which is then parameterized on a common planar domain. A given set of matching feature points serve as patch corners and feature correspondences. In particular, Yeh *et al.* [YLSL10] proposed an interactive interface for correspondence placement.

However, all the mentioned approaches use points for feature correspondences, while our method supports user sketches. In practice, we find it more intuitive to draw feature lines than to only place points, which allowed us to extend the strategy to blendshape modeling (see Section 4.3.3).

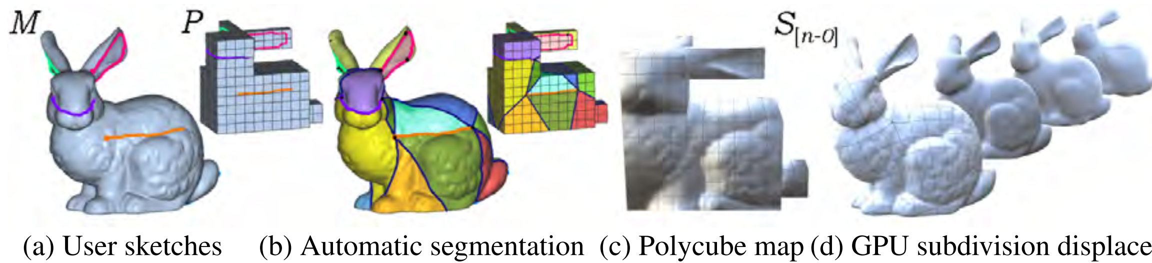
4.2 Editable Polycube Map

In this chapter, we present the editable polycube map to overcome the limitations of the existing approaches (see Figure 4.9). Our method allows the user to construct the polycube map in an intuitive and easy manner: given a 3D model M and its polycube domain P, the user is able to sketch features on M and P to specify feature correspondences. Then, our system will automatically compute the map in such a way that the features on M are mapped to the user-specified locations on P.

Later on, the user is allowed to edit the features on M, P, or both, providing precise control over the mapping. This way, the editable features can help to mark and preserve fine-grained features on the provided polycube-mapped subdivision surface.

We demonstrate the proposed editable polycube map framework with applications like GPU-friendly interpolative subdivision surfaces. The positive properties enumerated above allow an extremely efficient implementation of GPU-based subdivision surfaces. Also, the reduced number of combinations, together with watertight sampling, allow for a continuous subdivision method that smoothly integrates with current production pipelines. Finally, we are able to provide coarse regular base meshes with a reduced memory footprint.

The whole process should be compared with the traditional work artists do to create a model to be used in an environment like a computer game, which usually requires a subdivision surface



(a) User sketches (b) Automatic segmentation (c) Polycube map (d) GPU subdivision displacement

Figure 4.9: The proposed method allows the users to easily edit and control the polycube map by sketching the features/constraints on the 3D model and the polycube. The generated polycube map is conformal and with low area distortion, facilitating some graphics applications such as GPU-based displacement mapping.

to be used for modern tessellation hardware.

In general, if the new model is based on an existing model, for instance obtained from a laser scanning process, the final model generation implies manually performing re-topology operations to create a base mesh of the subdivision surface, to then transfer the original details by normal projection, which do not ensure bijectivity.

Usually, artists start from a coarse base model, quite often sculpted using a rough shape like a polycube base mesh. Then, the model is imported into an application like ZBrush [Pix10] and further subdivided with a couple of steps of Catmull-Clark subdivision. From this moment on, the artist has to model/transfer the fine-grained details onto the final model. Obviously, this is a time-consuming and quite redundant procedure.

Our proposal is to avoid this by quickly establishing a bijective correspondence between the coarse polycube and the input model, and then creating the quad-based representation on the polycube-based model by transferring the details of the high-resolution input model. The resulting model will have all the enumerated properties and would be ready for usage in a production environment with a minimal user interaction.

Another practical and extremely useful application is *kit-bashing* [PLB07] which refers to the widely used practice among artists of reusing previously made assets as accessories to quickly form a new model. In general, seasoned artists tend to keep a digital library of previously created model parts. With our scheme, the library of accessories and the newly constructed shape could be presented in a parameterized and tessellated form (see Section 4.3.2).

We also introduce shape blending as another very practical application of the techniques presented here. By mapping different shapes to a common base parametric domain, and by the introduction of specific blending operators, it is possible to seamlessly blend between shapes, or even obtain a smooth morphing animation between them (see Section 4.3.3). Finally, we also present dual painting, a tool that takes advantage of the dual parameterization of the polycube maps to help the user to easily paint complex models with concave surfaces that might be hard to reach (see Section 4.3.4).

The specific contributions of this chapter include:

- We present a method that, from a general mesh, creates a high-quality and user-controllable polycube map in an efficient and intuitive manner. Our method allows the users to easily modify the map and fine-tune the mapping. The user is also able to control the number of patches in the base mesh of the quad-based representation by the construction of the

base polycube. The provided polycube doesn't need to accurately resemble the shape of the object, as coarse polycubes are usually enough.

- We provide a subdivision surface representation specially built for quad patch-based tessellation on the GPU for object and character rendering. Also, this scheme provides a reduced number of topology combinations thanks to the low number of valence possibilities (only 3, 5 or 6), which is very important in terms both of memory footprint and of the texture fetching bandwidth, which strongly affects performance.

- This new scheme opens the door to new possibilities and applications. Interesting examples of these applications are GPU-friendly tessellated subdivision surfaces, kit-bashing, shape blending and dual painting.

The remaining of the chapter is organized as follows: Section 4.1 reviews related works on polycube map, surface quadrangulation, subdivision surfaces compatible with hardware tessellation, and cross-parameterization. Then, Section 4.2 presents the details of our editable polycube map framework, and Section 4.3 presents our applications. After that, Section 5.3 shows the experimental results and discussions. Finally, Section 4.5 draws the conclusion and discusses lines for future work.

4.2.1 Overview

As mentioned, the objective of the proposed technique is, starting from a high-resolution model coming either from an artist or a 3D scanner (plus its cleaning stage), to build a quad-based representation, feasible for subdivision surfaces that allows user control over a reduced number of singularities, and have that model ready for seamless texturing, watertight displacement, etc. This should be done in a user-controllable way, using only quads, and without wasted space in texture space.

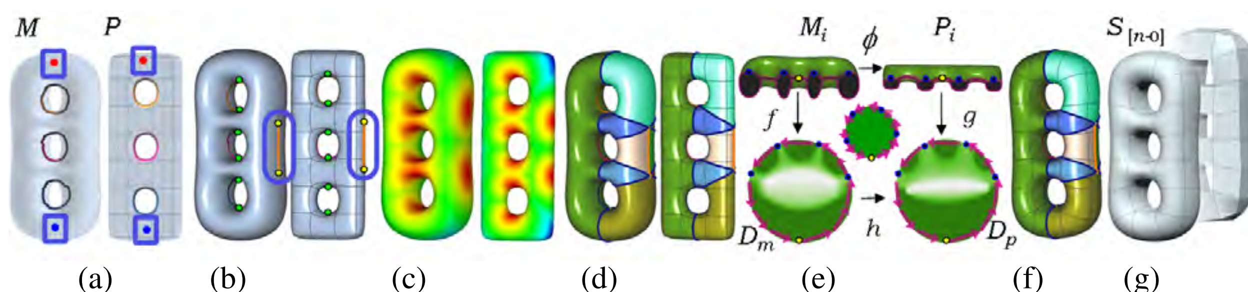


Figure 4.10: Algorithmic pipeline. (a) On non-genus 0 surfaces, first, only two user control points are required to guide an auto segmentation of the surfaces to genus 0 patches. This way the handles are cut by the corresponding auto-tunnels in M and P. (b) Next, the user sketches a few strokes as constraints on the 3D model M and the polycube P, where the yellow dots are sample points on the user sketched strokes and the green dots the saddle points computed in the auto-tunnels. (c) These sample points are used as sources to compute the distance field (warm colors are shorter distances and cold colors are larger distances between the sample points). (d) By the set of shortest paths between sample points, which do not cross the user strokes, and the auto-tunnels, a geodesic segmentation on M and P is computed, cutting both into *valid* single boundary genus-0 patches. (e) Then, a constrained map is defined between each pair of patches, which allows to transfer the surface M_i to P_i, between all pairs of patches. (f) Next, all P_i patches are seamlessly glued with a global smoothing operator. (g) And finally, the subdivision surface is reconstructed.

Our input is a high-resolution model plus a simple polycube representation, which is constructed manually by the user to mimic the input shape[1]. By controlling the position and location of the cubes, the user has control over the number and location of the singular vertices in the resulting quads-only mesh, as shown in Figure 4.23. For the convenience of the following stages, if the input model is non-genus-0, a preprocessing stage is performed to slit the model into genus-0 by the cutting tunnel loops (see Section 4.2.2.1).

Then we provide a guided feature-sketching interface enabling users to sketch controlling strokes in a natural manner (see Section 4.2.2.2).

The user-specified features are sampled with points, from which we compute the shortest distance to each other using a multi-source Dijkstra's shortest path algorithm. Although the computed distance field induces a triangulation on the 3D model, the path between two points is not straight and the resulting patch can have a complex non-triangular shape.

To ensure the quality of the triangulation, we compute the geodesic triangulation on the polycube by using the correspondence of the user-specified features on the 3D model and the polycube. This way, the resulting triangulation is smoother and more visually pleasing than the one that could be obtained directly from the multi-source Dijkstra computation.

After that, both shapes are segmented into genus-0 patches, keeping an identification between the segments in the polycube and in the high-resolution model. We compute the map between each pair of patches using a series of harmonic maps. By setting the boundary conditions carefully, the computed map is guaranteed to be continuous along the cutting boundaries.

Finally, a simple and effective global diffusion algorithm is applied to improve the map quality. The resultant map is bijective and satisfies the user-specified constraints.

The result of this process is that the polycube map induced a quad-remeshed version of the high-resolution model. An immediate benefit is that the patches of the resulting remeshed model come from tessellated square faces, so a very low-resolution version of the model can be built from the original polycube faces as shown in Figure 4.10. This low-resolution model not only has quads as its only primitive, but also only has vertices with a small and restricted valence number, only 3, 4, 5 and 6, as shown in Figure 4.15 (a). This low-resolution model is the basis of our subdivision surface with displacement mapping, see Section 4.3.1.

Then, polycube texture mapping is performed working only with the polycube map of the model, and consists of creating a 2D texture atlas which contains all the externally visible polycube faces, which is an easy task as the previous step already kept only the visible faces that are not shared by more than one cube. To build the atlas, we just placed each face in consecutive squares in the final texture atlas, see Section 4.2.3.

Finally, in runtime, only the very low-resolution model needs to be sent to the GPU, along with the texture maps built in the pre-processing stage, to generate a continuous subdivision surface with all the benefits mentioned in Section 2.2.2. As the very low-resolution model can be animated, its run-time tessellated version also can. It is important to mention that the user has full control over the process through the sketched features and the segmentation in the entire pipeline as illustrated in Figure 4.12, and in further applications described in Section 4.3.

[1]Many commercial softwares, such as Maya and 3DS Max, allow the user to easily do this.

4.2.2 Constructing Polycube Map

Given the input model M and the corresponding user-built polycube P, we present a user-controllable framework to construct the map between them. The algorithm pipeline is illustrated in Figure 4.10.

4.2.2.1 Topological preprocessing

Sketching controlling strokes in simple models of genus 0 is an easy task. However, in the case of models with complicated topology, say, with genus larger than 0, it would not be straightforward for novice users to sketch strokes keeping the correspondences between the input model M and the polycube M. In fact, it could be a tedious task to specify sketches around each handle explicitly on a high-genus model (see Figure 4.24 (a)).

In this section, we introduce a topology-aware preprocessing algorithm to slit the model M and polycube P into genus-0 patches if the input model M has a genus larger than 0, and the matching information is also computed in this stage.

The idea is motivated by the tunnel loop computation introduced by Dey *et al*. [DLS07]. Given a closed surface of genus g, there are always g tunnel loops. Cutting along a tunnel loop eliminates the handle. We can cut a high-genus model into a genus-0 surface by cutting all the handles along the corresponding tunnel loops. On the other hand, the geometry aware tunnel loop-computing methods [DLSCS08] suggests that the tunnel loops can be used to provide correspondence information in a surface map. In the following, we call tunnel loops simply as tunnels for short.

Given the input meshes and the tunnels in them, if the input object has a genus greater than 1, the matching between the tunnels of the object and the polycube it is a difficult challenge (further discussion in Section 4.4.4). So, we assume there is alignment and similarity between the object and the polycube. In case both the object and the polycube are reasonably well aligned and not drastically twisted, we provide a semi-automatic detection and matching of tunnels. In practice, considering that the polycube is constructed to mimic the input shape, these assumptions become reasonable. We adopt a hybrid strategy to achieve this matching between tunnels in the object and the polycube.

We compute the harmonic field in the object and polycube surfaces to depict their geometry shape. For this, the user only need to specify two control points on the input object and the polycube respectively, as illustrated in Figure 4.10 (a). It is worth noting that the pair of control points also serves as a part of the mapping constraints, like the feature strokes shown in Figure 4.10 (b).

In practice, we specify the two control points on two sides of the object and polycube shapes, in order to have all the tunnels between them. The computed harmonic fields are shape-aware, specifically, with value 0 at one control point, and 1 at the other one, smoothly increasing between the two points. Thus, the harmonic field values provide an ordering of the tunnel loops.

Furthermore, two saddle points are computed for each tunnel loop corresponding to the loci of the control points (shown in green in Figure 4.10 (b)). The saddle points in the tunnel loops serve as additional accurate map constraints. Note that there could be tunnel loops at the same level of the harmonic field, under the mild assumption of alignment and similarity. We

include the 3D space coordinates to enhance the ordering of the tunnels in order to eliminate this ambiguity.

Our specific algorithm is as follows:

Step 1. Given the non-genus-0 model M and the corresponding polycube P, we compute the tunnels of each handle in the 3D model and the polycube [DLSCS08].
Step 2. Given the user-specified control points $p_0, p_1 \in M$, and the corresponding $p_0^{'}, p_1^{'} \in P$, we compute the harmonic field f in M by setting $f(p_0) = 0, f(p_1) = 1$ as the boundary condition, and similarly for $f^{'}$ in P by setting $f^{'}(p_0)^{'} = 0, f^{'}(p_1)^{'} = 1$.
Step 3. We compute the two saddle points of each tunnel by selecting the points with minimum and maximum values of the computed harmonic fields in each tunnel. Under the assumption of similarity, the computed harmonic fields in the object and the polycube are similarly distributed.
Step 4. We match the tunnels in the 3D model with the tunnels in polycube. First we order the tunnels in M in ascending order by the average harmonic field value of all the points in the tunnel. If there are tunnels in the same level of the harmonic field, a further ordering is performed for those tunnels by their 3D space coordinates by selecting first the dimension with the largest difference above a threshold value between tunnels, and next in the other two dimensions. This strategy allows to order the tunnels in ascending order. In our experiment, the threshold is selected as 0.05 times the maximum of the harmonic value, or the length of the maximum direction of the bounding box.
Step 5. We cut M and P along the tunnels. In this way, the models with genus g are cut into genus-0 surfaces with $2g$ holes. Each tunnel is slitted into two circles and the 2 saddle points in the tunnel are slitted into 4 points. We match the slitted circles and saddle points of the 3D model and polycube by the consistent orientation between them.
After this preprocessing stage, the tunnel loops also serve as strokes to segment the object and the polycube (see Figure 4.10 (d)), and the saddle points serve as sample points for the polycube mapping algorithm described in Section 4.2.2.3.

4.2.2.2 Guided stroke drawing interface

Feature Guided Stroke Drawing: In our framework, the users are allowed to sketch the features freely on the models using a WYSIWYG (What You See Is What You Get) sketching interface. The WYSIWYG interface allows a natural sketching metaphor by projecting strokes from screen space onto the 3D surface.

On the object surface, the detailed geometry features are visualized and could be captured intuitively with strokes. Since the strokes serve as boundary conditions in the mapping stage, strokes on the shape features lead to more accurate mapping results that preserve the features well (see Figure 4.11).The user sketches a number of (few) features on the 3D model M, and the same number of features must be specified on the polycube P.

Stroke Types: Our system supports three types of stroke shapes: *curved line strokes*, *closed loop strokes*, and *crossed strokes*.

- *Curved line strokes* are the fundamental strokes for shape feature description, and users use them in most cases for specifying features (e.g. see the strokes in Figure 4.14).
- *Closed loop strokes* can be powerful in depicting loop-shaped features such as a character's mouth and eyes. With a closed loop stroke, the loop shaped feature on the objects'

surface can exactly be mapped onto a loop shape in the polycube domain (e.g. see the mouth stroke in Figure 4.18).

- *Crossed strokes* are a combination of several crossed *curved line strokes*. In practice, the strokes are split into connected *curved line strokes* by the crossing point (e.g. see the eyes and nose strokes in Figure 4.18).

Consistency & Topology Guided Stroke Drawing: The feature stroke set in the object must be consistent with the set in the polycube domain in the sense of having the same number of strokes and a similar spatial distribution. We present a consistency checking scheme to guide the consistency-aware stroke drawing. Specifically, after a pair of strokes are sketched in the object mesh and the polycube, our system performs the segmentation process described in Section 4.2.2.3 and validates the correspondence between the segmented patches. We perform the segmentation in an incremental manner that only computes the newly added geodesic paths in each iteration. If the segmentation results are not consistent between the object and the polycube, the system alerts that the newly added strokes could be non topologically consistent. Thanks to our editable framework, the user is always given the possibility to edit the strokes dynamically with the consistency guidance.

Furthermore, the purpose of the segmentation stage (described in Section 4.2.2.3) is to divide the mesh into *genus-0 single-boundary patches*. In the following, we refer to a patch of genus-0 with a single-boundary as a *valid patch* (e.g. Figure 4.10 (e) illustrates a pair of *valid patches* M_i and P_i).

On the interactive segmentation result, the *valid segmented patches* are shown with same consistent colors in the object and the polycube domain (see Figure 4.12 (2)). The remaining parts are still colored in grey-blue, which suggests that additional strokes are required to segment this region into *valid patches*. A complete example of the interactive stroke drawing process is illustrated in Figure 4.12.

Figure 4.11: The users can take full control of the polycube map by simple sketches. The thumbnails show the sketched constraints.

4.2.2.3 Segmentation

The divide-and-conquer approach proposed by He *et al.* [HWFQ09] can be used to construct the polycube map, by breaking down the model into genus-0 patches and then computing the piecewise map independently for each one. Although their method is able to divide the model automatically, all cutting planes must be horizontal. Thus, the segmentation highly depends on the orientation of the model and may result in too many small patches and the cutting boundary may not represent any feature.

In our framework, the users are allowed to sketch the features freely on the models (see Section 4.2.2.2). Here we assume that the user-specified features are consistent with the help the of the guided sketching interface.

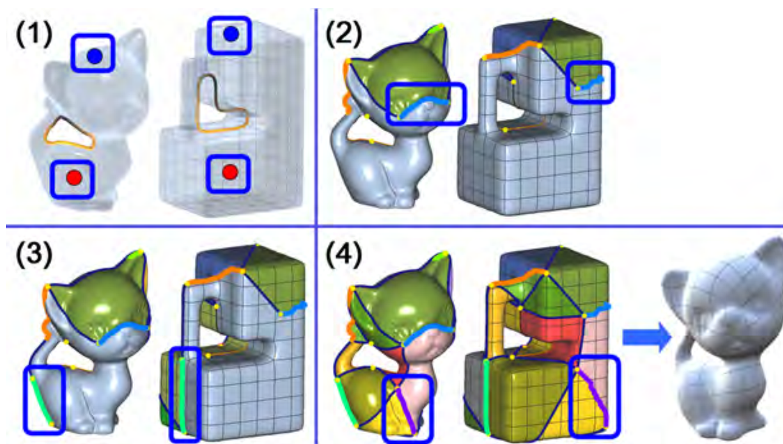


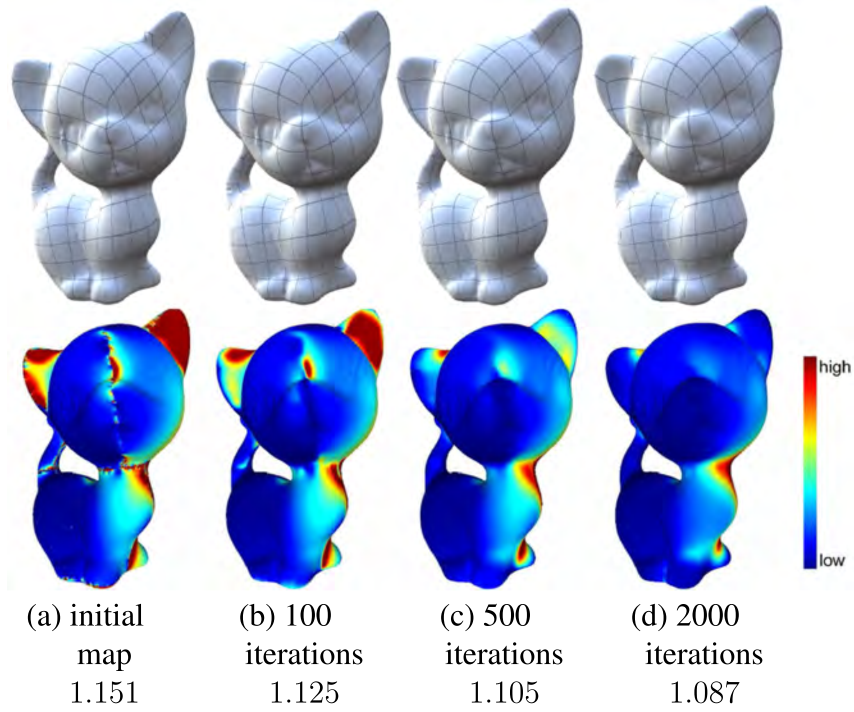
Figure 4.12: Interactive editing of Kitten (genus 1): (1) Initially, the user draws two control points and the auto-tunnels are computed. (2) Then, he draws three strokes above the object and the polycube. As a result, a set of color parts get segmented as *valid* single boundary g0 patches. (3) The remaining surface, shown in gray, means that additional strokes are further required. (4) Finally, with 5 user strokes, all the surface of both, object and polycube, is consistently segmented and the polycube map is computed.

Our segmentation algorithm is as follows:

Step 1. Given the user-specified sketches, $\gamma_i \in M$, $\gamma_i^{'} \in P$, $i = 1, \cdots, n$, we sample the sketches with a set of points. Let $S = \{p_j\}_{j=1}^{m}$ and $S' = \{p_j^{'}\}_{j=1}^{m}$ denote the sample points on M and P respectively.
Step 2. We use $p_j \in M$, $j = 1, \cdots, m$, as source points and compute the shortest distance for every point on M using a multi-source Dijkstra's algorithm. This way, each vertex v is associated with a distance $d(v, p_j)$ where p_j is the closest sample point to v. Let $c(v) \in S$ be the closest sample point of vertex v.
Step 3. We consider each mesh edge $e_{ij} = (v_i, v_j)$, where v_i and v_j are neighboring mesh vertices. If $c(v_i) \neq c(v_j)$, let $s_1 = c(v_i)$ and $s_2 = c(v_j)$ be the two sample points. We mark the two sample points s_1 and s_2 as neighbors.
Step 4. For every pair of sample points s_i and s_j which are marked as neighbors, we compute the geodesic path between them. It can be shown that two geodesic paths can only meet at the the two ending sample points. Then on the polycube P, we compute the geodesic path between $s_i^{'}$ and $s_j^{'}$. If the geodesic paths in object surface and polycube surface do not intersect with any user input strokes we add them into the *segmentation path sets*.
Step 5. We segment M and P along the computed *segmentation path sets*.
In the above algorithm, we compute a distance field on the 3D model M using the user sketched constraints. As shown in Figure 4.10 (e), this distance field naturally induces a segmentation on M. However, note that a few geodesic paths intersecting with user-strokes and auto-tunnels are canceled. Therefore not all the result patches are triangle-shaped.

4.2.2.4 Constrained map

Let $P_i \in P$ and $M_i \in M$ be a pair of segmented patches, each of which is a genus-0 surface with a single boundary. We want to find a bijective and smooth map $\phi : M_i \rightarrow P_i$ as illustrated in Figure 4.10 (e). Rather than computing the map directly, we first parameterize M_i to the unit disc using a harmonic map, i.e., $f : M_i \rightarrow \mathbb{D}_m$ such that $\triangle f = 0$ and f maps the boundary of M_i to the boundary of $\mathbb{D}_m$ using the arc length parameterization, $f(\partial M_i) = \partial \mathbb{D}_m$. Similarly,



(a) initial map 1.151 (b) 100 iterations 1.125 (c) 500 iterations 1.105 (d) 2000 iterations 1.087

Figure 4.13: Smoothing the polycube map. The initial map has only C^0 continuity along the segmentation curves and user-specified features. Thus, one can clearly see the large distortion and lack of smoothness in (a). Using the Laplacian smoothing algorithm, the distortion smoothly spreads out over the entire model, see (b)-(d). The values are the angle distortions. The step length is $\delta = 0.05$.

we also parameterize P_i to the unit disc using a harmonic map $g : P_i \rightarrow \mathbb{D}_p$. More details of discrete harmonic map can be found at [EDD$^+$95a].

We then seek a smooth map between the two unit discs $h : \mathbb{D}_m \rightarrow \mathbb{D}_p$. This map h is also computed using a harmonic map $\triangle h = 0$ and the boundary condition is set as follows: Let s_1, s_2 and s_3 be the sample points on ∂M_i, and $f(s_j) \in \partial \mathbb{D}_m$, $j = 0, 1, 2$ be the images on the boundary of unit disc. Similarly, let $g(s_j^{'}) \in \partial \mathbb{D}_p$ be the images of the sample points $s_j^{'} \in P_i$.

Then we require the function h to map $f(s_j)$ to $g(s_j^{'})$, i.e., $h \circ f(s_j) = g(s_j^{'})$, $j = 0, 1, 2$. The images for the points between $f(s_j)$ and $f(s_{(j+1)\%3})$, $j = 0, 1, 2$, are computed using an arc length parameterization.

The polycube parameterization is given by the composite map $\phi = f \circ h \circ g^{-1}$ as shown in the following commutative diagram:

$$\begin{array}{ccc} M_i & \xrightarrow{\phi} & P_i \\ \downarrow f & & \downarrow g \\ \mathbb{D}_m & \xrightarrow[h]{} & \mathbb{D}_p \end{array}$$

Finally, we glue the piecewise maps $\phi_i : P_i \rightarrow M_i$ together as illustrated in Figure 4.10 (f).

4.2.2.5 Globally smoothing map

With the above boundary conditions, the maps are consistent along the boundaries which can be glued seamlessly. The resulting map $\cup_i \phi_i$ guarantees to be C^0 continuous along the segmentation boundaries.

We use Laplacian smoothing [Fie88] to improve the continuity along the segmentation boundaries. Given the initial polycube map $\phi : P \to M$, let $p' = \phi(p) \in M$ denote the image of $p \in P$. Then we solve the following diffusion function:

$$\frac{\partial p'(t)}{\partial t} = -(\triangle p'(t))_{\parallel}, \tag{4.1}$$

where $\mathbf{v}_{\parallel} = \mathbf{v} - (\mathbf{v}, \mathbf{n})\mathbf{n}$ is the tangent component of $\mathbf{v}$, $\mathbf{n}$ is the normal vector, $(,)$ is the dot product, and t is the diffusion time.

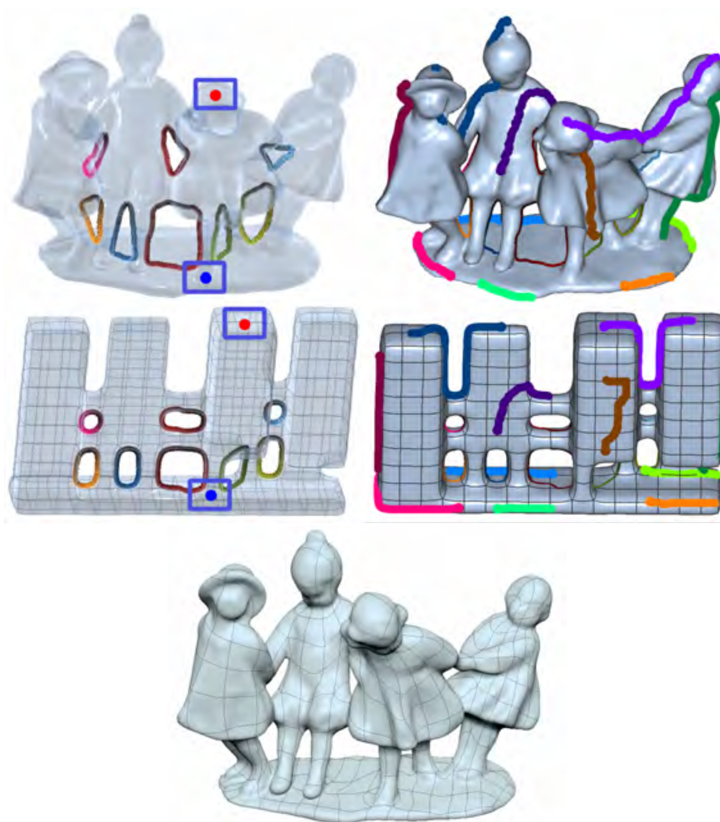


Figure 4.14: Children model editing: Using two user points, and 13 user-strokes. Bottom: The polycube map result.

Since the given polycube map ϕ is represented in a quadrilateral mesh induced by the tessellation of the polycube in which each quad in P is a square, we use the following Laplace operator:

$$\triangle p' = p' - \frac{1}{m} \sum_{pq\ is\ edge} q', \tag{4.2}$$

where m is the valence of p, and q is the one-ring neighbor vertex of p.
The above diffusion equation can be solved easily using the Euler method. We set the step length to $\delta = 0.05$ in our experiments.

Following Degener *et al.* [DMK03b] and Tarini *et al.* [THCM04], we measure the map quality in terms of angle and area distortions which integrate and normalize the values $\sigma_1\sigma_2 + 1/\sigma_1\sigma_2$ and $\sigma_1/\sigma_2 + \sigma_2/\sigma_1$, where σ_1 and σ_2 are the singular values of the Jacobian matrix of ϕ. $\epsilon_{angle} = \epsilon_{area} = 1$ when the map ϕ is isometric. As shown in Figure 4.13, our method leads to visually pleasing results in only a few hundred iterations. We must remember that Khodakovsky *et al.* [KLS03] introduced a globally smooth parameterization method. Particularly, our globally smooth algorithm is designed for polycube mapping, while their method is for mapping between triangle shaped patches.

4.2.3 Subdivision surface from the polycube map

We can generate a parameterized subdivision representation with displacement mapping out of the parameterized polycube map, which can be effectively used in many graphics application like computer-generated movies, as well as real-time applications.

As the bijection has already been established between the original surface and the polycube, the polycube mapped surface can be processed to define a quad-based subdivision surface $S_{[0-n]}$.

The base subdivision mesh S_0 will have the number of quad faces of the coarse polycube domain. The displacements defined between S_0 and the polycube mapped mesh S_n, are stored as a vector field *(direction + length)* in texture maps for the GPU-subdivision displacement applications described in Section 4.3. The folllowing steps are performed:

1. **Reverse subdivision:** First, we need to recover the coarse quads from the refined polycube-mapped surface S_n to extract the subdivision base mesh S_0 (see both in Figure 4.24 (bottom)). In order to do that we reconstruct the Catmull-Clark Subdivision scheme with the algorithm described in [LN06], which allows to obtain a valid Catmull-Clark subdivision surface $S_{[0-n]}$.

2. **Packing in texture space:** All polycube map patches are packed as consecutive squares in the texture atlas. For each patch S_0, all the interior sub-patches of S_n contained in a base patch of S_0 are assigned their respective texture coordinates in the atlas patch, as shown in Figure 4.15 (b).

3. **Raster displacement data:** We rasterize each high-resolution patch into its associated base patch as a vector field displacement in texture space, also saving the other required information as normals maps, occlusion maps, etc.

4.3 Applications

In this section we will present some applications that are possible thanks to the dual nature of the polycube parameterization, and the generated subdivsion scheme $S_{[0-n]}$: GPU-based subdivision displacement, kit-bashing, shape blending/animation, and dual painting.

4.3.1 GPU-based subdivision displacement

For real-time applications, we use the subdivision algorithm presented by Loop and Schaefer [LS08b], following the implementation described by Castaño [Cas08b]. The method presented

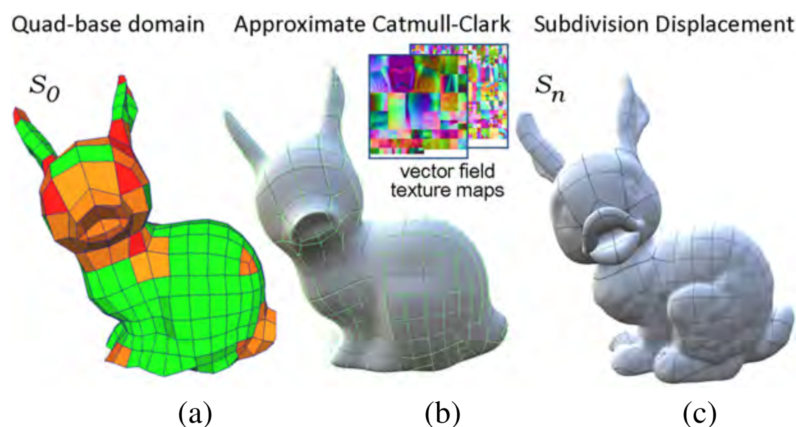


Figure 4.15: Bunny-duck kitbashed model shown with GPU-based subdivision displacement. (a) Our base mesh (level 0 of the subdivision surface). Green patches contain only valence 4 vertices, and the the orange to red levels shows the number of extraordinary vertices, red meaning patches having the greater number). (b) Tessellation with aproximate Catmull-Clark subdivision surface, with the normal and displacement vector field texture maps. (c) the subdivision surface with GPU-based subdivision displacement (the iso-parametric curves can be seen at the top right).

here is particularly well suited for this implementation as we only generate vertices with valences 3, 4, 5 or 6 as shown in Figure 4.15 (a). In this implementation, domain shaders (or vertex shaders when using instanced tessellation) are responsible for providing the final position of each new vertex from the tessellated mesh.

In order to have watertight sampling of the displacement map, it is necessary to solve the problem that the values of the texels along the chart boundaries must match exactly along both sides of the seams. For that reason, we define for each patch, the one who "owns" every single edge and corner [Cas08b, Cas08a], so all patches can agree what texture coordinate to use when sampling the displacement at those locations. This way, all patches are coordinated with respect to what texture coordinate to use when sampling the displacement map. In practice, this amounts storing, for every edge and for every corner, the texture coordinates of the owner of those features (4 texture coordinates per vertex). At runtime, only a single texture sample is needed; and the corresponding texture coordinate can be selected with a simple calculation [Cas09].

Geometry image-based tessellation is also an interesting option that becomes a sub-case of our strategy, but it is a technique mainly intended for static and not too large objects. With geometry images, no texture coordinate-specific information is required, sufficing just the ownership data to compute everything in the respective shaders.

4.3.2 Kit-bashing

Kit-bashing is a common practice among artists, who tend to accumulate models from previous projects and reuse parts of them to start building any new one. This technique is particularly used for human-like characters. Cut and paste methods are proposed in Funkhouser *et al.* [FKS+04] and Yu *et al.* [YZX+04a]. Huang *et al.* [HFAT07] proposed a merging boundary optimization for better matching. Sharf *et al.* presented SnapPaste [SBSCO06] enabling an interactive framework in a drag-and-snap manner. Kreavoy *et al.* [KJS07] computed the cut boundary automatically after the merging operation is specified for better boundary shape. And *blocks* (generalized cuboid shapes) from Leblanc *et al.* [LHP11] are introduced into this problem as a modeling primitive for composition operations to create complex objects.

With our polycube mapping technique, the artist can have a library of parts already mapped and textured. For instance, when creating a new monster character, the artist can decide to reuse the legs or the arms from a previous model. Then, the two parts of the model (the torso and the legs) can be assembled in the polycube domain supporting integer scaling and rotation factors, and a free alignment in object space, together preserving the subdivision surface representation in all the process as illustrated in Figure 4.16.

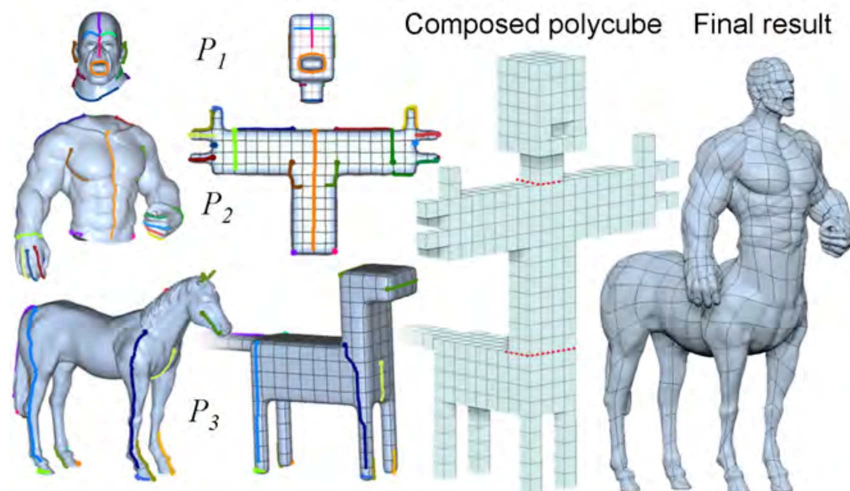


Figure 4.16: Centaur kit-bashing composition example. Left: a horse and a male head and torso are polycube mapped. Right: The parts of interest are cutted and ensambled together to generate the centaur subdivision model.

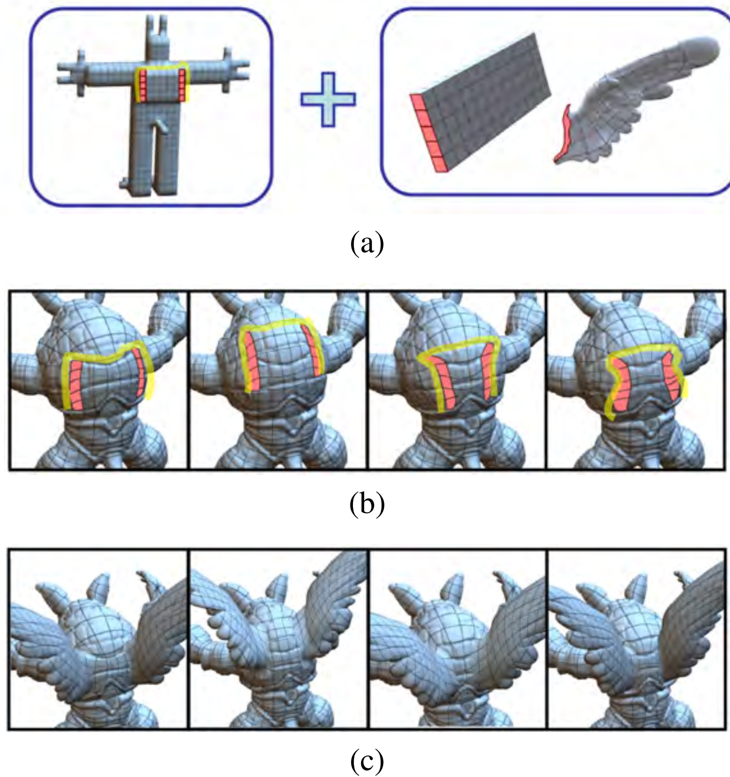
As the model parts to assemble have different atlases, creating the new unified atlas only means merging the respective patches from each texture map into a new single one, recomputing the seams boundaries, and adequately combining the new neighboring uv coordinates of each base level patch.

In Figure 4.17 we can see an example where the user stitched a couple of wings, originally from the Lucy model, to the back of the armadillo model. As the models were processed beforehand, the user just selected a few faces from both polycubes to cut the parts of interest by the selected cube faces. This leaves us with both polycube model parts perfectly matching in parametric domain.

Thanks to the bijection established between both the polycube and the original 3D model, the edges that stitched to each other on the polycube maps are bijectively identified with their corresponding edges in the original 3D model.

Once this identification has been done, the next step is to align the two original 3D model parts, at convenience by the user. In our implementation, we minimized a simple energy function consisting of the l^2 distance between the vertices. If the user is not satisfied with the final result, he/she is able to reposition the pieces by performing further adjustments to the positioning of the parts, before going to the next stage. If a larger positioning control is needed, the user is always free to place feature-lines at any place in the object and the polycube as illustrated in Figure 4.17 (b), and these features would be used in polycube map editing step.

Once aligned, we glue both parts by collapsing each pair of matching vertices, replacing them by the midpoint vertex. As this may cause crisp edges, we immediately perform a local poisson-



(a)

(b)

(c)

Figure 4.17: Armadillo-wings kit-bashing composition: The user took the polycube mapped wings from the Lucy model and stitched them onto the back of the armadillo model. (a) First, user select the parts to be glued. (b) Next, one curved stroke is added to fine-tune the wings placement. (c) And last, a poisson-based smoothing is used between the gluing parts to obtain the final result.

based smoothing step [YZX+04b] to the gluing parts as shown in Figure 4.17 (c).

4.3.3 Blendshapes

Another application of the editable polycube map approach is the creation of blendshape models to provide an intuitive way to define new models by easy fusion of a set of previous polycube mapped models. The proposed application is based on a same *shared* polycube base domain, where the user first mapped different input source shapes by a set of strokes as described in Section 4.2 and illustrated in Figure 4.18.

The polycube domain then acts then as the *combination space* of previously mapped input models. The blendshape formulation represents the surface as a linear combination of the set of shapes (i.e. like morph targets) as,

$$o = p + \sum_{i=0}^{n} B_i w_i \tag{4.3}$$

displacing the original positions p of the base polycube, with respect to the set of blendshape displacement textures B_i that contain the vertex displacements between the reference polycube base domain and their respective blendshape instances. The weight textures w_i are used to obtain the blend result o as the combined vector displacements in a final texture map.

The blendshapes B_i and weight textures w_i in the polycube domain, are unfolded in texture space on a per-face basis. The blendshape textures are combined by weights w_i defined by

user brush strokes with alpha masks or smoothstep operators. The per-face blendshape texture implementation is encoded in a texture array where we apply pinning to all corners of the mesh with an irregular valence. However, corners with valence different than 4 cannot be exactly matched with a bilinear interpolation. For that, we perform a simple process over the mesh connectivity to determine which shared corners are irregular. Then, for each shared group of faces for each irregular corner, we fix the blendshape texture. First, by determining the correct value to be stored exactly at that corner in the shared per-face texture (e.g. by simple average). Then the correction is propagated to every mipmap level of every face of that group, so that the shared corner has always the same value.

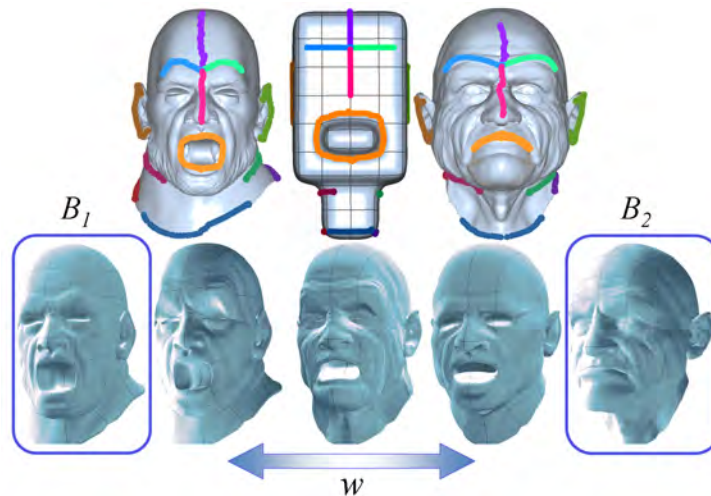


Figure 4.18: Top: The input surfaces are polycube mapped with the same base polycube *shared domain* to generate the blendshapes B_1 and B_2. Bottom: New surface models are obtained by the user with brush stroke operations to define the set of weight values for the linear combination of the blendshapes B_1 and B_2.

4.3.4 Dual Painting

Another interesting application of our dual bijective parameterization scheme is what we called *dual painting*, where a user can paint the model directly over the 3D model surface, the polycube, or any morph between the two. This can be considered a valuable addition to the tools proposed by Hanrahan and Haeberli [HH90], as the user can use the polycube map to easily access and paint concave regions that would be too difficult to paint otherwise (see Figure 4.19).

Another advantage is that cubes afford a semantic partition of the model so that users can expose occluded parts of the model by selecting and unhiding cubes other than triangles. Also, the parts that the user may assemble for kit-bashing can be textured beforehand, and our painting tools help the user with the finishing touches, using the smoothing operation to also blend the colors at the joints.

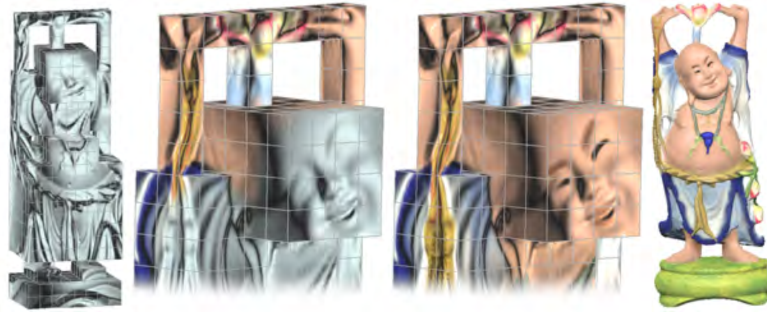


Figure 4.19: Dual painting on Happy Buddha: some parts, e.g, Buddha's cloth, are very difficult to paint due to the occlusion and concave geometry, but the polycubed equivalent is much more accessible and easier to paint.

4.4 Results and Discussion

4.4.1 Results

We tested our method with a wide-range of models with various geometry and topology configurations, as shown in Figure 4.25. Additionally, table 4.3 shows the statistics of our experiments. A good global bijective parameterization is very important for modeling and texturing, as typical projections used in Tarini *et al.* [THCM04] unavoidably have problems because two vertices on the same projection line share the same image, something that produces artifacts like the ones that are clearly visible in Figure 4.20.
We also quantitatively compared our methods with existing methods [THCM04], [WHL+07], [WJH+08] and [HWFQ09] as shown in in Table 4.2 and Figures 4.20 and 4.21.

Tarini *et al.* [THCM04] is efficient for large scale models, but it can not guarantee the bijectivity due to the projection of the 3D model to the polycube, and may result in some undesired artifacts in texture mapping and painting, see Figure 4.20.

Wang *et al.* [WHL+07] computed the polycube map in an intrinsic way by conformally parameterizing M and P to canonical domains and then seeking the map between them. For a genus-0 shape with complex geometry (like the Armadillo), the conformal spherical parameterization has very large area distortion on the elongated parts (e.g. arms, legs and tail). Thus, the induced map also has a large area distortion with uneven sampling.

Wang *et al.* [WJH+08] required to manually specify the images of the polycube corners and edges on the 3D model and then computed the map for each polycube face individually. The user-defined polycube structures on M can be considered as our constraints. However, it is very tedious and error-prone to specify them manually if the polycube is complicated, which precludes its usage for large-scale models. Furthermore, the user can not specify other features or constraints in Wang *et al.* [WJH+08].

Automatic approaches [LJFW08, HWFQ09] use heuristics and may not work well for complex models.

In He *et al.* [HWFQ09], both M and P are segmented by horizontal cutting planes, and the cutting locus serve as constraints. Thus, the generated map is orientation dependent. Due to the complex geometry of the Armadillo, e.g., the arms are not axis aligned, there are large distortions on the upper arms and shoulder, (see Figure 4.21). Compared to the existing approaches,

Table 4.2: Comparison of polycube map construction methods. Symbols: ● good, ◐ fair, ○ poor.

Features	[THCM04]	[WHL$^+$07]	[WJH$^+$08]	[LJFW08]	[HWFQ09]	Our
Map quality	●	◐	◐	●	●	●
Bijection	○	●	●	●	●	●
User control	○	○	◐	○	○	●
Editing	○	○	○	○	○	●
PC construction	○	○	○	●	●	○
Automatic	●	○	○	●	●	●
Large models	●	○	○	○	◐	●
General topology	○	○	○	○	◐	◐

our method is more intuitive and flexible in terms of user control and editing, and can generate better quality polycube-maps.

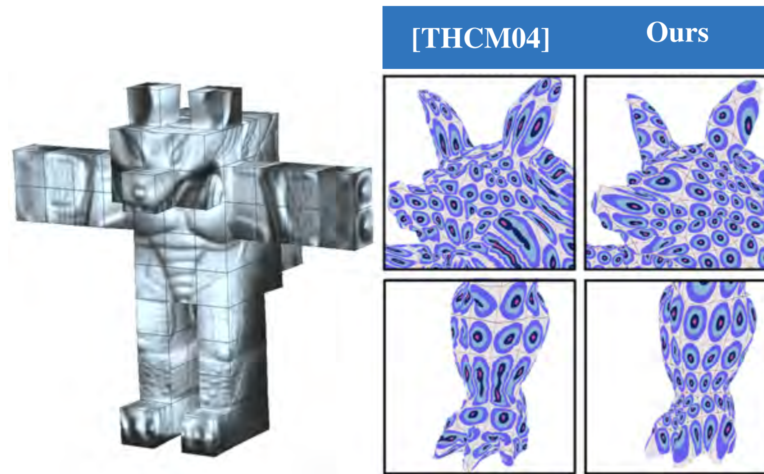


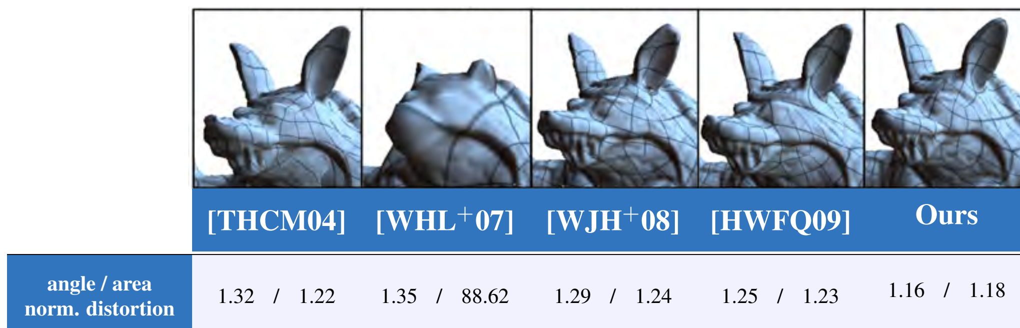
Figure 4.20: Comparison of the bijectivity between the editable polycube map and the original polycube implementation, at the left, of the original proposal [THCM04]. Top row: insets of the Armadillo head. Bottom row: insets of its feet. As we can see, non-bijectivity in [THCM04] results in a dependence of multiple points on the mesh with a single point in texture space: painting this single point stains other places than the one originally intended.

4.4.2 Tradeoff between accuracy and regularity

It is known that the distortions of polycube parameterizations highly depend on the shape of the polycube. In general, the more accurate its representation, the lower distortion of the map. However, the price to pay is the larger number of extraordinary points (polycube corners). Thus, there is a tradeoff between quality and complexity. Through our experiments, we observed that, for shapes with extruding regions, like the Armadillo's fingers (see Figure 4.23), it is usually a good idea to design an accurate polycube to model these features. In our framework, we leave the choice to the user.

It is worth noting that compared to the results of other approaches, the polycube maps of our method have higher quality using the same coarse polycube base (See Figure 4.21). Benefited by our interactive control, a better tradeoff could be achieved using our system.

In Figure 4.22 we compare the parameterization quality with different methods not using a polycube parametric domain. As it can be observed we can provide a higher quality quad-mesh structure with lower angular and area distortion, comparable in terms of low distortion to less constrained methods [ZSGS04], thanks to our configurable parameter domain.



	[THCM04]	[WHL+07]	[WJH+08]	[HWFQ09]	Ours
angle / area norm. distortion	1.32 / 1.22	1.35 / 88.62	1.29 / 1.24	1.25 / 1.23	1.16 / 1.18

Figure 4.21: Comparisons of our method with [THCM04], [WHL+07], [WJH+08] and [HWFQ09]. We use the same polycube to make the comparison fair. Our method is more intuitive and flexible in terms of user control and editing, and can generate polycube map of better quality. The values below each figure are the angle and area distortions.

4.4.3 Stroke drawing

The user input strokes play two important roles in our framework. First, they induce the segmentations: the shape of segmented patches is controlled by the shape of input strokes, which are validated during the mapping stage to ensure they provide a consistent topology in the object and the polycube. Second, the input strokes also serve as the boundary conditions in the mapping stage. Thus, the loci of input strokes brings to users full control of the mapping result (see Figure 4.11).

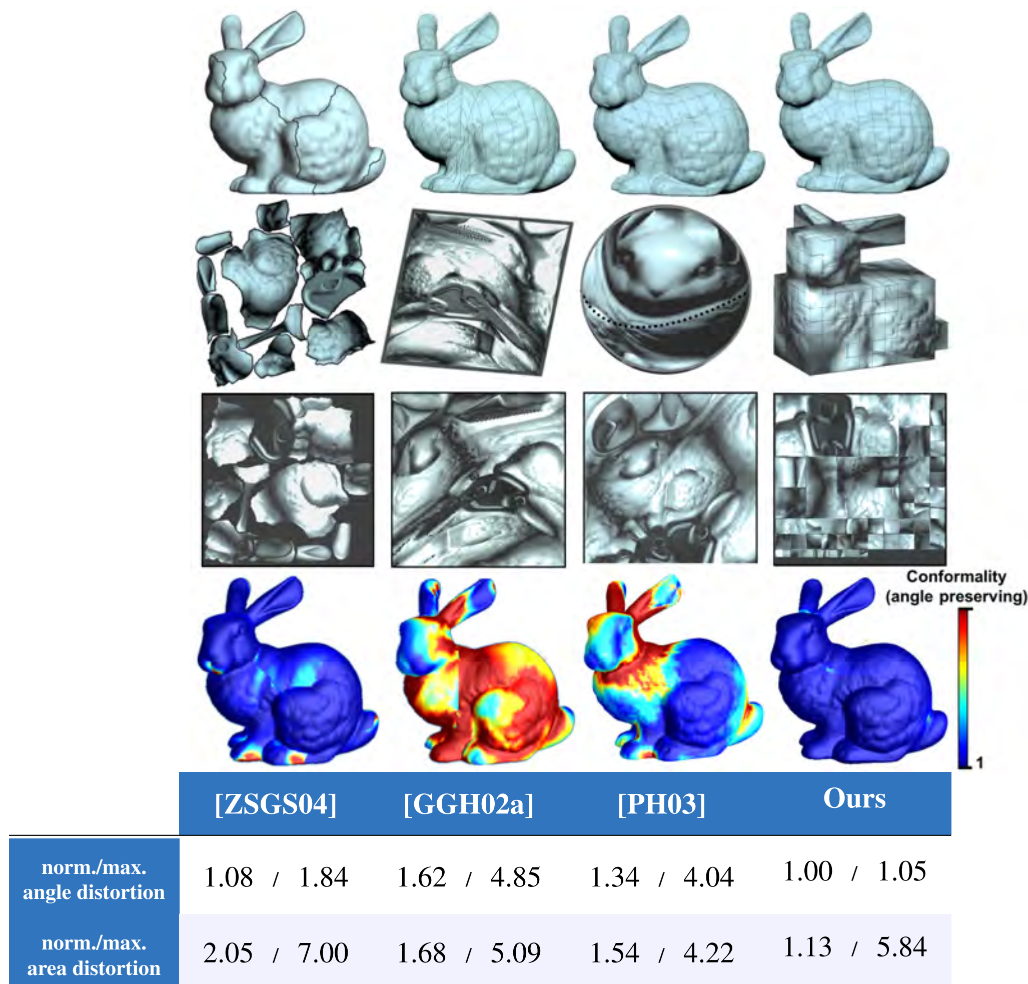
The requirements and importance of the input strokes might challenge novice users. In our observation, fortunately, the correspondences match with user's perception and can be easily managed after a few trials. In general, users add strokes around relevant detailed features, like near the ears, eyes and mouth of face models, which usually leads to satisfactory mapping results which preserve well those features, as illustrated in Figure 4.18.

4.4.4 Automatic Tunnel Slitting

The proposed topological preprocessing stage (described in Section 4.2.2.2) converts the input high-genus mesh into genus-0. The user could also map high-genus meshes without it by carefully slitting all the handles. We evaluate the performance of automatic tunnel slitting by comparing polycube mappings with and without it (see Figure 4.24).

A novice user specified 34 strokes on Buddha for polycube mapping without the auto-computed tunnel loops. Given the guidance of computed tunnel loops, 15 strokes are required by the same user. Further more, the quality of the computed tunnels is often better than with user input strokes around the handles. Thus, automatic tunnel slitting often lead to better mapping quality with less required effort.

Our algorithm would fail if the alignment and similarity assumption does not hold. If only the alignment assumption fails, the polycube or the input object could be aligned with simple rotation operations in our system. But the similarity assumption can fail if, for example, the input object is twisted too much, and the polycube can not be constructed keeping similarity and simplicity at the same time. In this case, tunnel loops are computed without the correspondence information, and users are required to input the correspondences manually by specifying one point at each tunnel loop in a corresponding order in the object and polycube.



	[ZSGS04]	[GGH02a]	[PH03]	Ours
norm./max. angle distortion	1.08 / 1.84	1.62 / 4.85	1.34 / 4.04	1.00 / 1.05
norm./max. area distortion	2.05 / 7.00	1.68 / 5.09	1.54 / 4.22	1.13 / 5.84

Figure 4.22: Comparisons of our polycube mapping method with other methods not using a polycube parametric domain: [ZSGS04], [GGH02a], [PH03]. Our user editable parameterization can provide a higher quality with lower angular and area distortion thanks to the configurable parameter domain.

The specified points also serve as the boundary condition in tunnel loops map. Thus, $2g$ point specifying operations are needed for model with genus g. It's still much easier than drawing $2g$ loops strokes to slit g tunnels manually.

Furthermore, in this extremal case, there is no currently known perfect solution. e.g. Reeb Graph [PSBM07] might be a potential solution. But it would fail in cases of symmetric shapes. More user interaction or information of 3D space alignments are also required. A complete classification of correspondence methods is described in [KZHCO10]. For general case, we believe that the assumption of alignment and similarity are mild. And our algorithm would suffice to solve most cases of this problem.

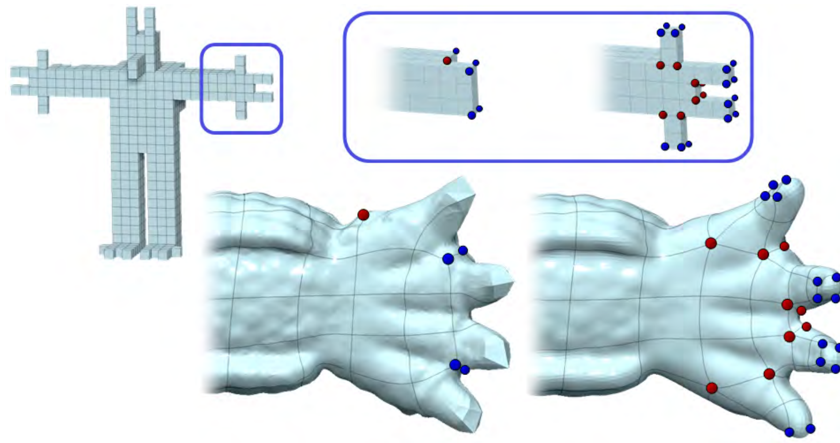


Figure 4.23: Armadillo hand polycube editing: The control of the base polycube detail and the type and location of irregular vertices are both important, because they are intrinsically linked to the geometric features on the surface, and impact the final quality. Irregular vertices shown in blue and red, are valence 3 and 5 respectively. Left: A coarser polycube hand without fingers, has less irregular vertices but limit the quality. Right: A more detailed polycube introduce more irregular vertices but capture with higher fidelity the surface details.

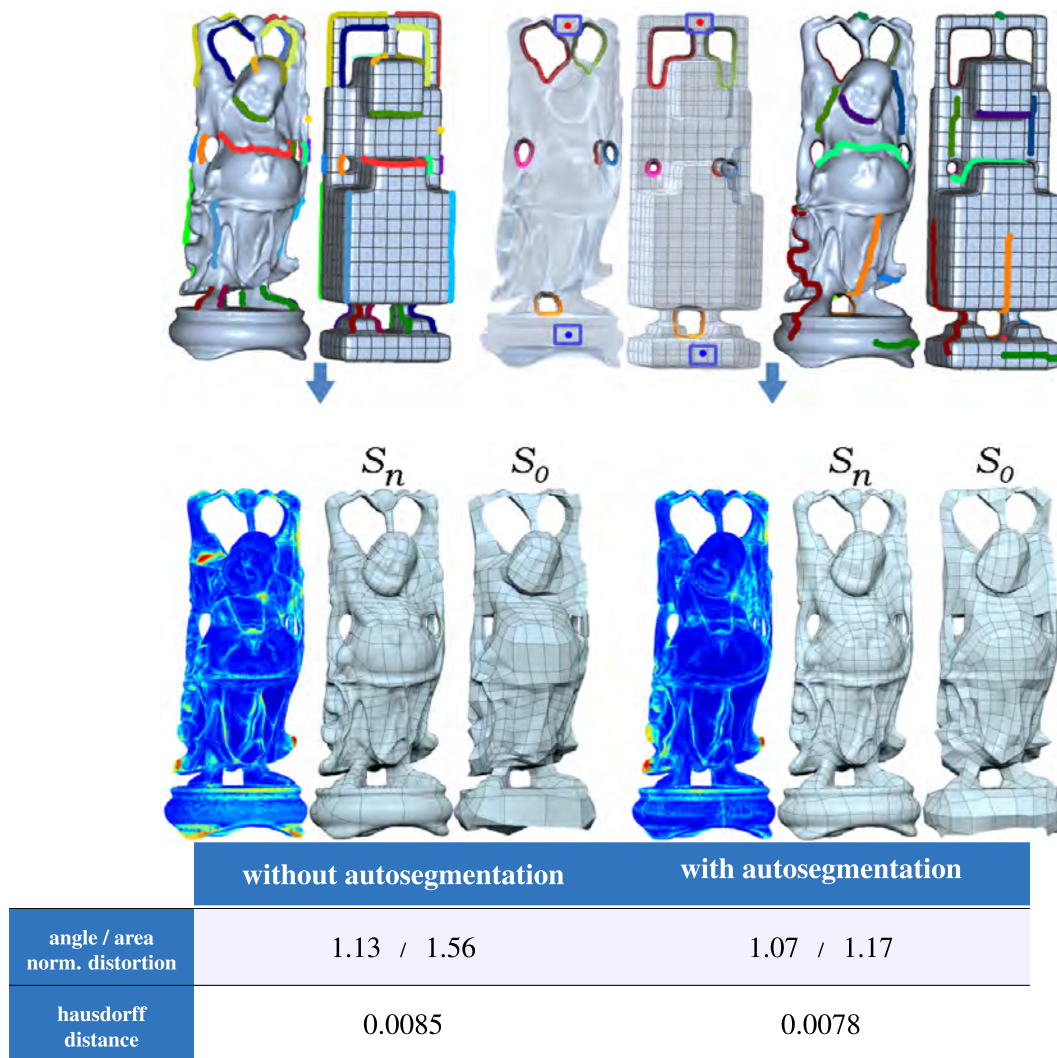
Table 4.3: Statistics of experimental results. Notation used in the table: genus: genus of M; #△ in M: # of triangles in M; #□ in S_0: # of squares in S_0; S_n levels: # of subdivision levels for the high-resolution polycube; n_c: # of corners in P; n_f: # of user-specified features; $time$: time measured in seconds; angle distortion: angle distortion metric; area distortion: area distortion metric; hausdorff M - S_n: hausdorff w.r.t bounding box diagonal between M and S_n.

model	genus	#△ in M	#□ in S_0	S_n levels	n_c	n_f	$time$	angle distortion	area distortion	hausdorff M - S_n
Armadillo	0	346K	1012	6	82	14	148	1.16	1.18	0.0054
Arthur hand	0	307K	239	5	56	7	78	1.06	1.14	0.0091
Arthur head	0	252K	57	7	12	5	76	1.02	1.13	0.0050
Buddha	6	114K	818	5	117	15	47	1.07	1.17	0.0078
Bunny	0	144K	452	5	22	6	35	1.01	1.13	0.0072
Children	8	269K	526	5	110	13	106	1.20	1.22	0.0089
Duck	0	101K	288	5	24	5	37	1.09	1.16	0.0041
El Oso head	0	100K	150	5	24	11	40	1.13	1.22	0.0076
Holes 3	3	64K	68	5	32	1	13	1.10	1.18	0.0045
Horse	0	67K	436	5	28	8	11	1.04	1.07	0.0069
Isidore horse	0	151K	74	8	14	6	48	1.01	1.07	0.0061
Kitten	1	274K	272	5	28	6	83	1.16	1.27	0.0055
Lucy	0	526K	980	5	38	15	210	1.12	1.23	0.0091
Ogre	0	478K	694	5	124	9	68	1.14	1.29	0.0089
Skull	0	52K	24	7	8	3	7	1.02	1.06	0.0035
Tylo head	0	500K	920	5	32	9	194	1.10	1.25	0.0083

4.4.5 Comparison to [HWFQ09]

We want to emphasize the differences between our method and He *et al.* [HWFQ09]. They first segmented the 3D model and polycube by horizontal planes, computing a map between each pair of segmented components, and finally smoothed the map by solving a harmonic map for the entire shape. Although using the same divide-and-conquer strategy, our method is completely different in all the following steps:

First, since the cutting loci are also the constraints of the map, He *et al.* [HWFQ09] can only map the horizontal, planar features from the 3D model. Our segmentation allows the users to cut the model by arbitrary closed curves, resulting in more flexible and meaningful constraints. Furthermore, our method supports the user-control and editing operations not supported by



	without autosegmentation	with autosegmentation
angle / area norm. distortion	1.13 / 1.56	1.07 / 1.17
hausdorff distance	0.0085	0.0078

Figure 4.24: Buddha model editing. Left: without the handles autosegmentation, a novice user specified 34 user strokes to obtain a good quality result. Right: Using two user control points for the auto-segmentation, and with only with 15 strokes, same user obtained better quality results. Bottom: The generated subdivision surface respectively with hausdorff w.r.t bounding box diagonal between M and S_n, area and angle distortion in S_n.

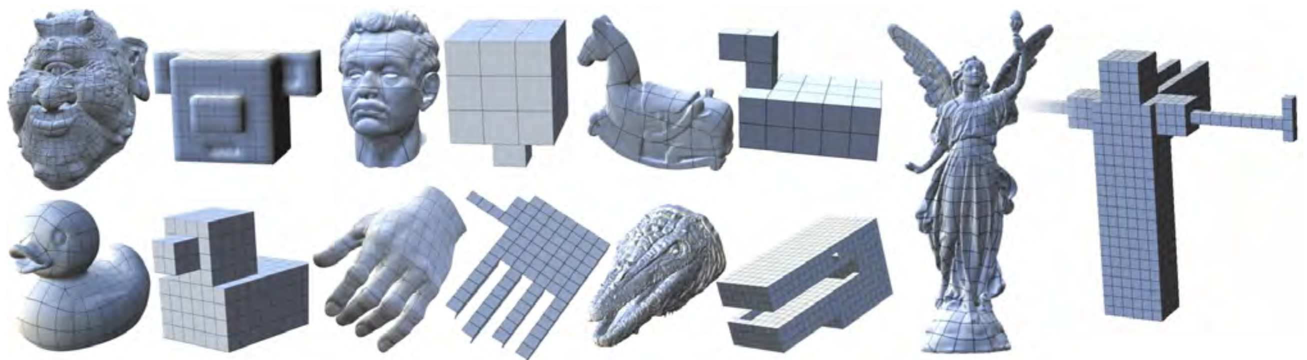


Figure 4.25: More polycube mapping results: Ogre, arthur's head, isidore horse, lucy, duck, arthur's hand and tylo's head.

them.

Second, He *et al.*[HWFQ09] mapped the segmented components to multiple connected rings by an uniformization metric, and then computed a harmonic map between the rings. It is known that computing this metric is a nonlinear time-consuming process. Our method computes the harmonic map between two topological disks, so it is much more efficient.

Finally, rather than solving a harmonic map for the entire shape, we smoothed the whole map by an iterative method to diffuse the angle distortion. As shown in Figure 4.13, our method is very effective and leads to a high quality map with only a few hundred iterations, as only very simple vertex operations are involved in each iteration.

4.5 Conclusions

We have proposed a new method that aims to improve user-control by developing a very practical and efficient system. From a general mesh with optional featured sketches, we create a user-controllable quads-only mesh with a globally smooth parameterization. The method guarantees that the computed map is bijective and conformal except at a finite number of extraordinary points (the polycube corners). As listed in Table 4.2, our method has most of the user-desired features: The created mesh is uniform, regular and is generated from the base polycube mesh in an automatic manner. During the editing stage, the user is able to control the number of patches in the base mesh of the subdivision surface by the construction of the base polycube. Then, the user still has the possibility of fully controlling the process by sketching correspondence lines between the high-resolution model and the polycube.

On the more technical side, as a result of the reduced number of topology combinations, we are able to have both a small memory footprint and a reduced texture fetching bandwidth, which strongly improves run-time performance of the tessellated result with modern hardware using instanced tessellation or the programmable units in Shader Model 5 hardware.

4.5.1 Limitations and Future Works

The proposed framework has limitations, though. First, sharp thin features, like the sharp boundaries of man-made mechanical objects may not be preserved well in our framework. There are various reasons. On one hand, the sketched input strokes may not directly capture the sharp feature exactly. On another hand, the proposed global smoothing in Section 4.2.2.5 is not feature-aware and the sharp features will be undesiredly smoothed. A simple extension to improve the sketch interface would be to aid the user snapping strokes to the exact sharp features. And, at the same time, use these sharp stroke features as constraints in the global smoothing stage.

Second, the tunnel loops matching between the object and the polycube might fail if the input mesh has complicated geometry or topology, e.g., twisted shape with handles. In this case, it is hard to align the object and corresponding polycube. More precise polycube design is able to alleviate this problem. But designing a precise polycube requires more efforts and introduces more singularities. One direction of our future work is to investigate more robust tunnel loops matching algorithm.

Chapter 5

Coherent parallel hashing

> Collecting *sparse* spatial data is particularly useful when having some elements of interest sparsely located in the spatial domain. This chapter presents a parallel hashing method to robustly create smaller tables with only the interesting shape and shading elements of implicit or irregular surfaces of objects, while exploiting the coherence in the spatial data and the access patterns for fastest random-access parallel queries in many computer graphics applications.

Sparse spatial data is very common in Computer Graphics, in previous chapter we shown that mesh parameterization is a useful strategy to unfold the surface information in a parametric domain. However, very sparse, irregular-topology or implicit surfaces are not easily parameterizable as we describe below. Other strategies are required to find a good tradeoff between theirs access performance and the efficient storage which is an ongoing challenge.

For example, raw acquired surfaces have a poor topological quality coupled with a high triangle mesh density making them hard to unfold them uniformly in a planar domain by a mesh parameterization, to achieve an efficient attributes encoding for parallel query evaluation.

In sparse image data, discontinuities like sharp vector silhouettes are generally present at only a small fraction of pixels. As an example, texture sprites usually overlay high-resolution features at sparse locations that should be efficiently stored, or image attributes like alpha masks are mainly binary, requiring additional precision at only a small subset of pixels.

Spatial hashing is a strategy to losslessly pack sparse data into a dense table. These methods have proven useful in situations enabling the data to be tightly packed while still allowing fast random access. In computer graphics, spatial hashing has been already successfully applied for texturing, rendering, collision detection and animation.

Using a spatial hash, data is stored in a single array—the *hash table*—addressed through a *hash function*. The hash function computes the data location in the hash table from the query coordinates, or *keys*. There have been several developments lately, improving query and construction times, and in particular enabling fast parallel construction on GPUs.

In Chapter 3 surface geometry was unfolded uniformly distributed in the parameter domain. However, when this is not possible, leaving a too sparse distribution of the surface data in the spatial grid directory D on the parameter domain, with too much wasted empty space, one requires to use a different data structure, such as hashing, for the directory. Instead of storing all grid cells, we store only those cells that actually contain records of surface and shading information.

5.1 Context: parallel hashing

Recent spatial hashing schemes hash millions of keys in parallel, compacting sparse spatial data in small hash tables while still allowing for fast access from the GPU. Unfortunately, available schemes suffer from two drawbacks: Multiple runs of the construction process are often required before success, and the random nature of the hash functions decreases access performance.

The first spatial hashing schemes focused essentially on reaching good load factors while having a constant time and simple access to the data from the GPU.

Lefebvre and Hoppe [LH06b] proposed a static hash construction enabling access to the data with as little as two memory accesses and one addition. However, to achieve this result the hash has to be *perfect*: All keys corresponding to defined data should map to different locations in the hash table. In other words, there are no *collisions*. Building such a constrained hash function requires an off–line construction process, limiting this approach to static cases (see further details in Section 2.4.9).

Alcantara *et al.* [ASA+09, AVS+11] propose less constrained hashing schemes that achieve fast, parallel construction on the GPU. These schemes may produce collisions. However, querying a key never requires more than four independent memory accesses. The particular hash mechanism they use is known as *cuckoo hashing*. We detail it in Section 5.1.

Both approaches achieve constant query time with a fixed number of instructions. Unfortunately, these constraints imply that construction is difficult and the process may fail, requiring several restarts especially at high load factors.

Alcantara *et al.* [ASA+09] introduced the first algorithm enabling fast, parallel hash table construction on the GPU. Millions of keys are efficiently hashed in milliseconds, outperforming previous schemes by several orders of magnitude. Since this approach is the closest to our work, we describe it in more details.

A cuckoo hash [PR04] maintains two or more different tables of the same size, each accessed through a different hash function—Alcantara *et al.* [ASA+09] use three tables. Keys are inserted in the first table, evicting already inserted keys in case of collision. Evicted keys are in turn inserted in the second table, then from the second to the third, and from the third back to the first. The process loops around until all keys are inserted or until the number of iterations reaches a maximum—which triggers a construction failure. Upon failure the process is restarted with different hash functions. Unfortunately, given a number of tables the failure rate abruptly increases above a limit load factor. Using more tables increases this limit. However, using too many tables becomes wasteful at lower load factors. In contrast, our proposed hashing scheme automatically adapts to these various situations (see Section 5.2).

The parallel construction algorithm builds a cuckoo hash in shared memory—a small but very fast memory. It starts by randomly distributing the keys in equally sized buckets using a first level hash [BZ07]. Any bucket overflow triggers a construction failure. This limits the maximum possible load factor to 0.7: higher load factors give a too large failure rate for this key distribution phase. Next, all buckets are hashed independently in parallel with cuckoo hashing.

In recent work, Alcantara *et al.* [AVS$^+$11] build a cuckoo hash in a single pass. The single pass approach is made possible by latest hardware capabilities (efficient atomic operations on NVidia Fermi devices). Their new hashing scheme also reaches higher load factors thanks to the use of four hash functions. Both schemes further introduce handling of multi-value hashing and duplicate keys in the input.

The cuckoo scheme behaves very well in practice, and the guarantee of a constant number of memory accesses to query a key is well adapted to GPUs. Its main drawback stems from an increasing failure rate at high load factors requiring to manually select more hash functions, and the loss of coherence due to randomization. Another less obvious issue is that while a fixed number of lookups are required, the average number of lookups is often higher than that of our scheme as keys tend to be uniformly distributed in all the tables, even at lower load factors.

We compare our work to parallel cuckoo hashing in Section 5.3.

5.2 Coherent parallel hashing

In this section we introduce a new parallel hashing scheme which reaches a high load factor with a very low failure rate. In addition our scheme has the unique advantage to exploit coherence in the data and the access patterns for faster performance.

Compared to existing approaches, it exhibits much greater locality of memory accesses and consistent execution paths within groups of threads. This is especially well suited to Computer Graphics applications, where spatial coherence is common. In absence of coherence our scheme performs similarly to previous methods, but does not suffer from construction failures.

Our scheme is based on the Robin Hood scheme [Cel86] modified to quickly abort queries of keys that are not in the table, and to preserve coherence. We demonstrate our scheme on a variety of data sets (see Section 5.3). We analyze construction and access performance, as well as cache and threads behavior. We relax the constraint of accessing data with a fixed number of instructions. Instead, we implement queries after trying the *maximum* number of accesses required to find a defined key. We propose a mechanism to quickly reject empty keys, thereby significantly reducing their negative impact.

In addition, we tailor our scheme to exploit the spatial coherence of rendering algorithms. In existing schemes, neighboring keys are often mapped to distant locations in the hash table. This is an issue since graphics hardware is designed to benefit from spatial coherence: Threads are organized in a grid and best access performance is achieved when nearby threads access nearby memory locations. Lefebvre and Hoppe [LH06b] were aware of this and proposed a construction process preserving some degree of coherence, however with only limited positive impact on access performance.

Linial and Sasson analyzed a *non-expansive* hashing scheme to bring similar keys close to each other in the hash table [LS96]. That scheme, however, necessitates too much space to be

practical in graphics applications.

As we shall see, the improved robustness of our scheme lets us design hash functions improving memory coherence during queries, while still affording high load factors.

Our hash is designed to reach high load factors at low failure rate, and to provide fast queries regardless of the load factor. The key insight is to exploit dynamic branching to release the constraints on the construction process, and to exploit coherence in the access patterns when available.

Our algorithm builds a unique, large hash table in one pass. Parallelism is obtained by launching many thread groups simultaneously. Each thread is responsible for inserting exactly one key.

Our main contributions are:

- A parallel hashing approach reaching high load factor with a low failure rate. It relies upon a coherent hash function exploiting coherence in memory accesses, when available. This leads to increased locality in memory accesses and increased coherence in the execution paths within a thread group accessing the data.

- An improved query scheme using a few bits of additional information per key to efficiently find defined keys, and perform early rejection of empty keys.

We introduce a complete parallel GPU implementation and analyze its behavior in details.

5.2.1 Notations and definitions

Keys are taken in a universe U of size $|U|$. We note $K \subset U$ the set of *defined keys*, that is the keys from U which should be stored in the hash table. Keys which do not belong to K are called *empty keys*.

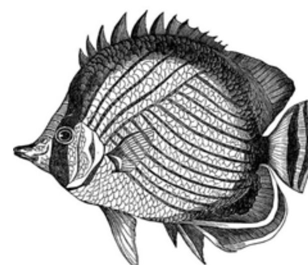
Throughout the paper we consider the *load factor* d. It corresponds to the ratio of the *number* of defined keys to the number of keys that the hash table D can hold. A hash table with load factor $d = 1$ is a minimal hash, that is a hash with no wasted space. It is worth considering another type of "load factor", which we call the *key-density*; it is defined as the ratio of the number of bits used to store the keys to the number of bits in the hash table (not counting data bits).

Each defined key may be associated with some additional data. In our setting, all defined keys are associated with data fields having a same, fixed size (e.g. an RGB color triple, or a boolean value). The input is thus given as a set of key-data pairs.

An application is said to perform a *constrained access* to the hash if the only queried keys are in the set K. The scheme of [LH06b] is especially efficient in this situation, as keys do not need to be stored. In this paper however we are mostly concerned with the *unconstrained* access scenario, where empty keys may be queried and must be detected as such.

By *coherence* we refer to the locality of the parallel memory accesses performed within a thread group. In Section 5.3.2.2 we compare query performance for the 2D case using different access patterns: Linear row major, Morton and the bit-reversal permutation. The first two offer a strong locality—neighboring threads access neighboring data—while the third has poor locality.

test1: $2.2M/4096^2$



fish: $20.5M/8192^2$

Figure 5.1: Two of the datasets used to test our hashing algorithm. They give a good spread of behaviors between randomness and structure. The number of defined keys (black pixels) and the size of the image is indicated below each.

5.2.2 Main algorithm and data structure

Our hash is at heart an open addressing scheme [Pet57]: Each input key k is associated with a sequence of probing locations $h^1(k), h^2(k), \ldots$ in the hash table. Ideally, this *probe sequence* should enumerate all locations in finite time. For now, we assume that the sequence is given. We introduce our coherent probe sequence later.

In order to add a key, the insertion algorithm iterates along the probe sequence until an empty location is found. The key is then inserted. We call the number of steps required for successful insertion the *age* of a key. If the hash table can fit all the defined keys, then the process is guaranteed to succeed as long as the sequence $h^i(k)$ enumerates all locations. Therefore the algorithm is quite robust to changes in the hash function. Querying a key proceeds similarly to construction, by walking along the probe sequence until the key is found in the hash table.

However, open addressing suffers from a severe drawback for our purpose: The age of the keys is very low on average but typically has a large maximum. This maximum age, noted M, is crucial: When querying an empty key, its absence from the hash table can only be verified by walking along the sequence of keys at least M steps. Since the data set is sparse, a large number of queries to empty keys is expected in many applications. The overall performance can dramatically suffer. Note that hitting an empty location during a query before reaching M steps is unlikely, especially under high load factors.

We next discuss how to efficiently reduce the maximum age and reject empty keys.

5.2.2.1 Reducing the maximum age

The maximum age issue has already been identified and studied in previous work. A very effective solution to it is known as Robin Hood hashing [Cel86], which is based on open addressing.

The idea is to store the age of the keys in the hash table during its construction. This additional data is discarded afterwards. Consider the case of inserting a key k_{new} at a location $h^i(k_{\text{new}})$ already occupied by a key k_{prev}. The age a of k_{prev} is compared to i. If the key being inserted is older ($i > a$), then k_{prev} is evicted. The current key is inserted at $h^i(k_{\text{new}})$ and k_{prev} is recycled into the set of keys to be inserted. Intuitively, the keys which are difficult to insert push away the keys which have been easier to insert.

This has a drastic effect on the histogram of key ages, and in particular on the maximum age as illustrated Figure 5.2.

Figure 5.2: Hashing 2^{20} defined keys randomly distributed in a universe of 2^{24} keys into a hash table of 1.3 million keys (the load factor is 0.8). Histogram of insertion ages for open addressing (top) and Robin Hood (bottom). Gray bars correspond to very few items but are non–zero. The maximum age goes down from 46 to only 5.

There are two particularly important theoretical facts about Robin Hood hashing making it especially well suited to our purpose [Cel86]: The expected maximum age in a *full* table of size n is $\Theta(\log n)$ and Devroye *et al.* [DMV04] improve the bound to $\Theta(\log_2 \log n)$ on non full tables. Furthermore, the expected query time complexity can be made constant if starting the accesses at the average age. Note that these facts are derived assuming uniform random sparse data, but our experiments show that they hold in practice on our datasets.

In our current implementation we do not start the accesses at the average age: For simplicity we always start from age one, searching for the key until the maximum age for the sequence is reached.

5.2.2.2 Empty key rejection

While Robin Hood hashing strongly reduces the maximum age M, it remains quite large compared to the few memory accesses of cuckoo hashing. This is especially important in applications querying many empty keys, as they always require M steps along the sequence. We therefore introduce a new mechanism to accelerate the rejection of empty keys.

We note that most keys have a very small age, with only a few outliers ever reaching M. Therefore, in most cases a much smaller value than M would suffice to detect empty keys. To benefit from this we store in each entry of the hash table the maximum age of all the keys mapping *first* to this location. More precisely, let $D[p]$ be the key stored at location p in the hash table D, let $\text{MAT}[p]$ be the maximum age starting from p, then we have:

$$\text{MAT}[p] = \max_{\{k \in K | h^1(k) = p\}} \left(i \text{ st. } D[h^i(k)] = k \right) \tag{5.1}$$

When querying a key k we iterate at most $\text{MAT}[h^1(k)]$ times along the probe sequence. This guarantees that defined keys are found, and affords for fast detection of empty keys.

5.2.2.3 Encoding the maximum age

Storing the maximum age table MAT requires additional memory. Fortunately, the maximum age values are small and only require a few bits. In all our tests, the maximum age was below 16. Thus, we reserve 4 bits in each hash table cell to store the maximum age. The latter can optionally be quantized to either accommodate larger values or reduce even further the number of bits used to 3 or less.

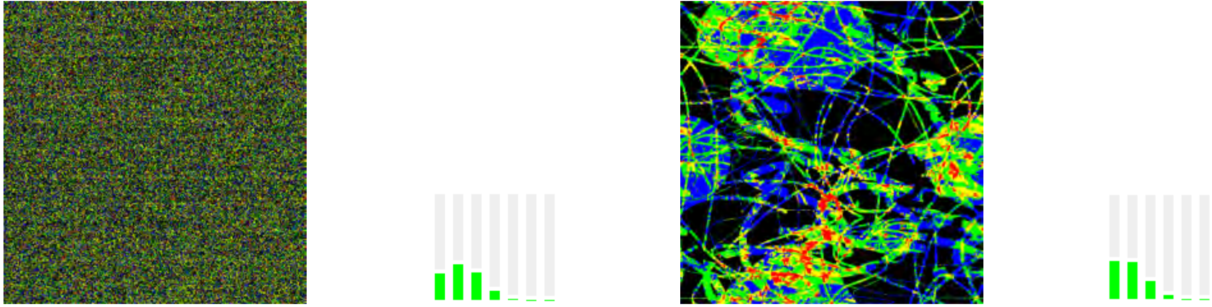


Figure 5.3: Maximum age table for test1 hashed at 0.8 load factor. Color code: 0, black; 1, blue; 2, green; 3, yellow; greater, red. *Left:* Random probe sequence. *Right:* Coherent probe sequence. Key age histograms are comparable, but note how coherence is maintained in the map on the right. We can expect more efficient dynamic branching when branching with respect to this map.

Let us assume each key is stored on k bits and their age is stored on a bits. When the user targets a data structure p times larger than the defined keys, we allocate $pk|K|$ bits of memory. In fact, due to the additional storage of the maximum age, this corresponds to a hash table storing $\frac{pk|K|}{(k+a)}$ keys. Thus, the keys will be hashed at load factor $\frac{(k+a)}{pk}$. For example, in a typical situation, we have $k = 28$ bits, $a = 4$ bits. Then, targeting a storage of $1.2 \times k \times |K|$ bits—that is 1.2 times the size of the defined keys, or a 0.83 key-density—requires hashing at 0.95 load factor, while targeting a 0.7 key-density requires a 0.82 load factor. With 64 bit cells and 60 bit keys, these load factors become 0.89 and 0.76 respectively.

The hashing scheme of Alcantara *et al.* [AVS$^+$11] requires no additional information apart from the key, in which case the load factor and the key-density are equal.

5.2.2.4 Exploiting coherence

In many computer graphics applications, data is queried in a coherent manner: Either spatial coherence within a frame, or temporal coherence due to limited motion between frames. We design a new probe sequence tailored to exploit coherence in the queries.

Note that we seek coherence of memory accesses *between neighboring threads*. This is quite different from the typical CPU notion of cache coherence where one seeks to access nearby memory locations *in sequence*. On the GPU, groups of threads access memory *simultaneously*, and it is important to group the accesses in nearby locations. This is also known as *coalesced* accesses. Similarly, the hardware performs faster when groups of threads take similar branching decisions.

Our objective is to design a different probe sequence for the keys, which favors coherence. Our new probe sequence h_{coh} preserves coherence by making neighboring keys test neighboring locations—while still ensuring that the *successive* locations of a same key are uncorrelated and perform a random walk. It corresponds to random translations of the *entire* hash table at each step i. That is, for a key k at step i in a hash table of size $|D|$:

$$h^i_{\text{coh}}(k) = k + o_i \mod |D|$$

where o_i is a sequence of offsets, independant of k. We typically set $o_0 = 0$ and use large random numbers as offsets.

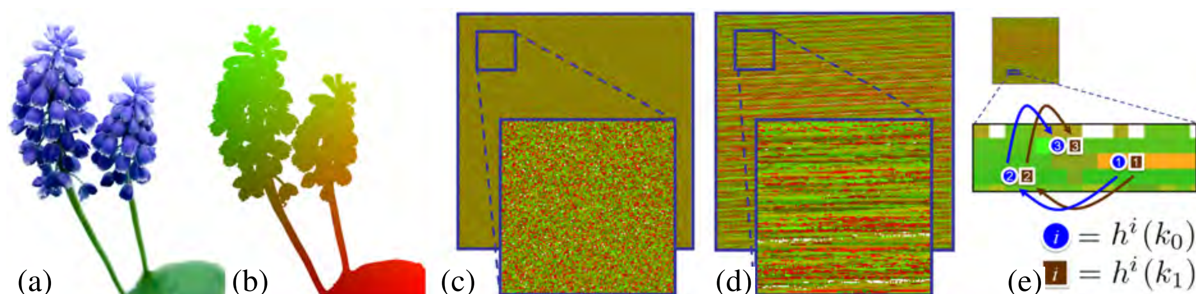


Figure 5.4: *(a)* The flower image is 3820×3820 image (14.5 million pixels) and contains 3.7 million non–white pixels. The coordinates of these pixels are shown as colors in *(b)*. We store the image in a hash table under a 0.99 load factor: the hash table contains only 3.73 million entries. These are used as keys for hashing. *(c)* The table obtained with a typical randomizing hash function: Keys are randomly spread and all coherence is lost. *(d)* Our spatially coherent hash table, built in parallel on the GPU. The table is built in 15 ms on a GeForce GTX 480, and the image is reconstructed from the hash in 3.5 ms. The visible structures are due to preserved coherence. This translates to faster access as neighboring threads perform similar operations and access nearby memory. *(e)* Neighboring keys are kept together during probing, thereby improving the coherence of memory accesses of neighboring threads.

An important property of this probe sequence is that neighboring keys remain neighbors at each step, as illustrated Figure 5.4 (e). Therefore, if two neighboring threads attempt to find neighboring keys, they will both always access neighboring memory locations until one terminates. Note that these keys do not have to be present in the hash table for coherence to happen at the thread level during queries. In fact, access coherence during queries is orthogonal to the distribution of the defined keys. Sparse random data encoded with our hash will still benefit from being queried in a coherent manner.

The map MAT encoding the maximum ages also benefits from a coherent hash function: It exhibits a much stronger spatial coherence. This implies that neighboring threads will perform similar data–dependent loops, reducing divergence. In Figure 5.3, for illustration purposes we hash the image test1 shown in Figure 5.1 in a 2D hash table – we extend the hash to 2D by applying the same computations independently to each dimension. We display the maximum age table MAT for a random and a coherent probe sequence. The second image (coherent) exhibits structure and coherence, and thus affords for more efficient dynamic branching than the first.

Our coherent probe sequence outperforms the random one when coherence is present. It strongly reduces cache misses, thread branch divergence and results in significantly faster queries. We measure these effects on various data sets in Section 5.3.

5.2.3 Construction

Our CUDA implementation runs on a graphics processor, where multiple threads are organized in groups and execute the program in parallel. We always build 1D hash tables. 2D and 3D data is linearized, as discussed in Section 5.3.2.

Building the hash table in a single pass requires global atomic operations to safely manipulate memory. Our eviction strategy requires comparing the age of the key to be inserted to the age of the key already in the hash table. These operations—compare ages and store key if greater—have to be performed atomically, or the table will quickly get corrupted by concurrent accesses. We encode the age, the key and its data in a single word (either 32 or 64 bits) with the age on 4

bits. We never observed an age above 15 in all our tests, but reaching this value would trigger a construction failure.

The insertion algorithm is detailed next. For clarity we ignore the data fields. Please note that this is a pseudo code. The actual implementation differs slightly. In particular, the atomicMax operation is not available on 64 bits words on current hardware (however, it is natively supported on 32 bit words). We thus emulate it using atomicCAS, as suggested by the CUDA programming guide. This incurs a performance penalty in the construction process: Full hardware support of 64 bits atomicMax will further improve performance.

The input defined keys are in the array K. The outputs are the hash table D and the max age table MAT. The type uint represents unsigned integers. The algorithm performs the following operations:

Listing 5.1: Coherent hash construction.

```
1  kernel(const uint *K, uint *D, uint *MAT) {
2     uint  key       = K[ global_thread_id ];
3     uint  age       = 1;
4     while( age < MAX_AGE ) {
5        uint h_k_i   = hash( key , age );
6        uint age_key = PACK( age , key );
7        uint prev    = atomicMax( & D[ h_k_i ] , age_key );
8        if ( age_key > prev ) {
9           uint h_k_1 = hash( key , 1 );
10          atomicMax( & MAT[ h_k_1 ] , age );
11          if ( AGE( prev ) > 0 ) {
12             key      = GET_KEY( prev );
13             age      = GET_AGE( prev );
14          } else {
15             return;
16          }
17       } else {
18          age++;
19       }
20    }
21 }
```

init The hash table D and the max age table MAT are allocated and initialized to zero.

2-3 The thread reads the key from the input located at index global_thread_id, which is unique for each thread. The current key is read in key, and age is the current insertion age.

4 The thread executes until the key has been inserted in an empty cell of the hash table or the maximum number of iterations for the current key has been reached.

5 The hash function is applied to key at age.

6-7 The key and its age are packed into a word age_key (32 or 64 bits). An atomicMax is used for the eviction mechanism. This instruction compares the current value in memory to age_key and replaces it if greater. The previous value is always returned.

8 Tests whether an eviction occurred, by comparing the value returned by atomicMax to age_key.

9-10 An eviction occurred and the current key has been inserted. These two lines update the max age table MAT. The first hash position of the current key is computed, and an atomicMax updates the max age table.

11-13 The age at the insertion location determines whether it was an empty slot or a previously inserted key. An age above zero implies that a key was evicted. The evicted key is recycled and becomes the current key, inserted next.

15 If the insertion location was empty, the thread has finished inserting its key and exits.

18 The key was not inserted. The age is incremented and the next insertion location will be tested.

After construction the maximum ages stored in `MAT` are optionally quantized and packed together with the keys in `D`. This is done in a second CUDA kernel. The table `MAT` is discarded after this. Note that duplicate keys in the input could be trivially handled by comparing whether `prev` equals `age_key`.

5.2.3.1 Running the kernel

The number of threads and groups is chosen so as to maximize the GPU workload. A thread that has finished its job sits idle until all threads in its group have finished as well. While our coherent probe sequence and max-age table help reducing thread divergence, some remain and idle threads do occur. To minimize their number, the number of threads per group should not be too large. It should however not be too small either for a good GPU utilization as we are limited by the maximum number of groups working simultaneously (120 in our NVidia Fermi GPU). We experimentally found that the best tradeoff is to use 192 threads per group.

Thus, we set the number of groups to $\lceil |D|/192 \rceil$ (a few threads after the end of the input array run without performing any operations).

5.3 Results

The performance of our scheme is impacted by several factors: The number of keys to be hashed, the target load factor, and whether coherence exists in the data and the access. All our tests are performed with a NVidia Fermi GTX 480 GPU.

In the following comparisons we introduce a variant of our scheme using a random probe sequence h_{rand} defined as follows:

$$h^i_{\text{rand}}(k) = c_0 + (k * c_1) + (i * c_2) \mod |D|$$

where c_0, c_1 and c_2 are large random numbers. The purpose of h_{rand} is to reveal when coherence is successfully exploited by our coherent probe sequence. Indeed, in absence of coherence we expect h_{coh} and h_{rand} to result in similar performance. When coherence is available we expect better performance from h_{coh}.

We compare our results to the methods of Alcantara *et al.* [ASA+09, AVS+11]. We use the implementations made available by the authors in the NVidia CUDPP library, using the multiplicative hash functions described in these papers. We ran all tests on a NVidia GeForce GTX 480.

In Section 5.3.1 we analyze the behavior of our hash when no particular coherence exists, and compare it to previous work. In Section 5.3.2, we discuss the behavior of our hash in a Computer Graphics setting, where spatial coherence exists.

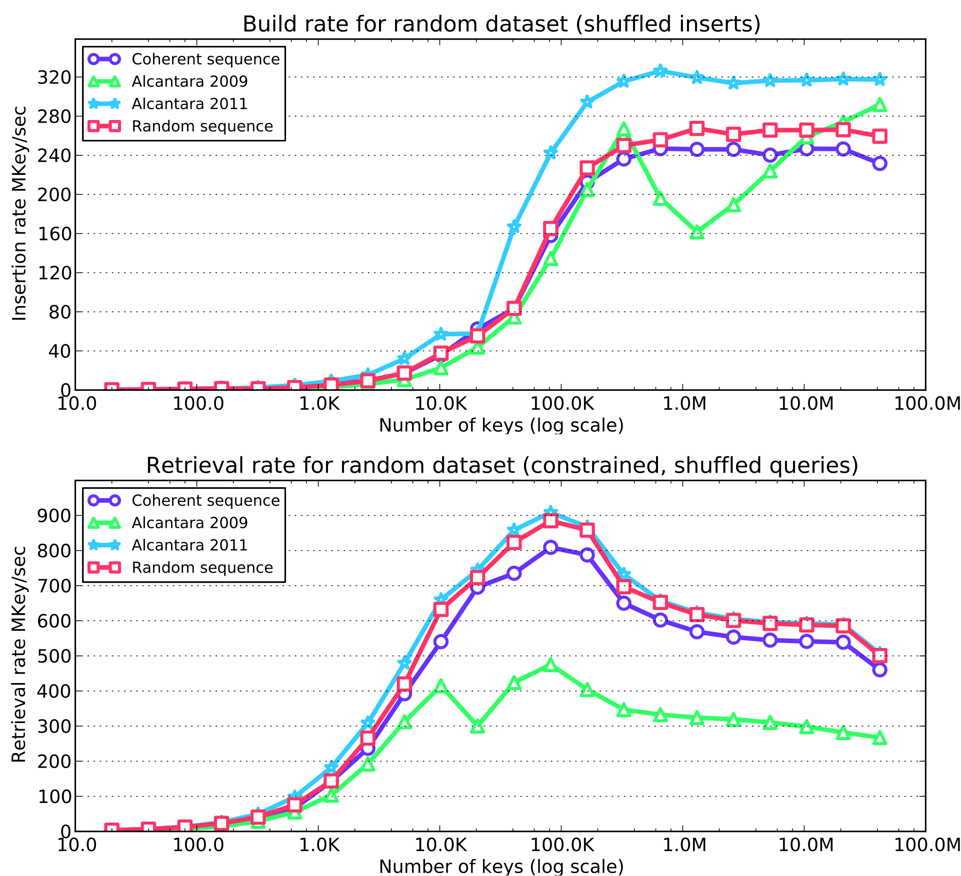


Figure 5.5: Insertion rates (top) and retrieval rates (bottom) for h_{rand} and h_{coh} and earlier schemes, for increasingly larger input, under 0.8 load factor. Timings are averaged over several runs. Please refer to the text for details on these data sets.

5.3.1 Hashing generic data

In this section we focus on hashing random keys taken in a 1D universe, assuming that no particular coherence exists neither between keys nor in the access patterns.

We consider key–data pairs of 64 bits, having a key on 32 bits, a data record on 28 bits, and using 4 bits to encode the maximum age. We randomly select an increasingly larger number of keys in the universe of 2^{32} possible keys.

We analyze insertion and retrieval of defined keys. For insertion the input is a vector of key–data records. For retrieval, the input is a vector of keys for which data must be retrieved. To avoid all bias in the measure, we shuffle the input vectors for both construction and query.

Of course, this setting exhibits no coherence and corresponds, in fact, to the worst case scenario for our hash. We will see in Section 5.3.2 that performance significantly increases in the presence of coherence.

5.3.1.1 Insertion

Figure 5.5 (top) compares construction performance of increasingly larger random sets of keys, under a 0.8 load factor. We observe that both probe sequences h_{rand} and h_{coh} behave similarly. This is explained by the fact that the random input data does not exhibit any coherence that

could be exploited during construction.

5.3.1.2 Retrieval

Figure 5.5 (bottom) compares query performance of increasingly larger sets of keys, under a 0.8 load factor. Again, on these random data sets we observe that both probe sequences h_{rand} and h_{coh} behave similarly. In these tests we only query *defined keys*. Since the input is extremely sparse, there is no coherence in the access. We will observe the benefit of coherence in Section 5.3.2.

5.3.1.3 Comparison to cuckoo hashing

Under a 0.8 load factor and for 16M (million) keys, our scheme achieves an insertion rate of 249 Mkeys/sec. In comparison, [AVS+11] achieves 318 Mkeys/sec and [ASA+09] achieves 268 Mkeys/sec. Therefore, in absence of coherence our insertion scheme is slower than both versions of the cuckoo scheme. This is essentially due to the update of the max-age table `MAT`, and the emulation of the 64 bits `atomicMax`. We will see in the next section, however, that in presence of coherence our scheme performance increases significantly.

Remarkably, our scheme consistently reaches load factor as high as 0.99. For 32 millions random keys under 0.99 load factor, our insertion rate is 112 MKeys/s and the retrieval rate is 241 MKeys/s.

5.3.1.4 Failure rates

In all our tests our scheme *never* failed to build a hash table both with h_{rand} and h_{coh}, and up to 0.99 load factor. Load factors higher than 0.99 typically generate a max age above 15 which no longer fits our simple 4 bits encoding. This robustness is an important advantage compared to the cuckoo scheme where restarts can lead to inconsistent performance behavior under high load factors.

cuckoo hashing [ASA+09] rarely fails at 0.7 load factor, however this behavior quickly degrades at higher load factors.

5.3.2 Hashing in a Computer Graphics setting

Our scheme is best suited when data is coherent—defined keys tend to be neighboring—and when the data is accessed in a coherent manner. Coherence in the data helps the construction process; However a random set of keys still benefits from a coherent access due to thread locality.

In a typical Computer Graphics application the hash table stores a sparse, structured, set of elements (texels, vector primitives, voxels, particles, triangles) which are accessed with some degree of spatial coherence. In most scenarios, a large number of empty keys are also queried.

5.3.2.1 2D data

We first consider 2D data sets. Our test consists in hashing a sparse subset of the pixels of a 2D image (e.g. all the non white pixels), and then query *all* pixels of this image to reconstruct it.

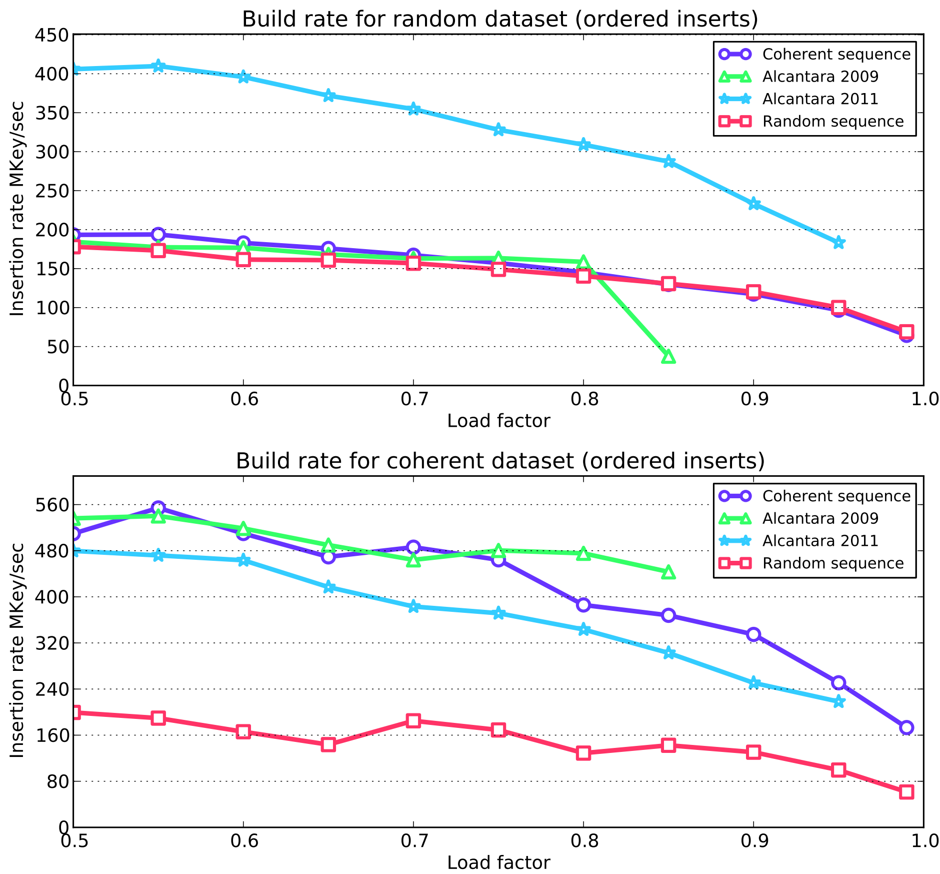


Figure 5.6: Construction times for h_{rand} and h_{coh} and earlier schemes. Times are averaged over several runs. *Top:* $1M$ random keys are inserted, taken from a universe of size $8K \times 8K$. *Bottom:* Timings for the fish image: $20.5M$ are defined in a universe of size $8K \times 8K$.

In the tests below, keys are computed from the 2D pixel coordinates with a row-major ordering. We later discuss the impact of different pixel orderings.

Figure 5.6 reports construction times for both a random data set and the fish data set. The important observation is that coherence in the fish data set—the existence of many neighboring keys—directly results in improved construction performance. The fish data set contains $20.5M$ keys. Under a 0.85 load factor, the random sequence reaches 142 Mkeys/s while our coherent hashing scheme achieves 368 Mkeys/s – an improvement of 159%. Thanks to coherence our scheme is now on par with parallel cuckoo hashing for construction times. In contrast, on random data the coherent sequence has similar performance as the random sequence.

Figure 5.7 reports the time taken to retrieve *all* the keys, both defined and empty. The distinction between querying empty or defined keys is important since empty keys are typically the most expensive to retrieve. In our scheme their cost is greatly reduced by using the the max-age table. The results are shown in Figure 5.7 for both a random set of keys and the fish dataset. Clearly, both the fish and random data sets benefit from coherence in the access. These results are consistent across all the datasets we tested. Under a 0.85 load factor our coherent hash retrieves 5324 Mkeys/s, while all other schemes achieve less than 1000 Mkeys/s: Coherence brings a very significant improvement in query performance.

The benefit of our coherent probe sequence is also clearly revealed by the *percentage of global*

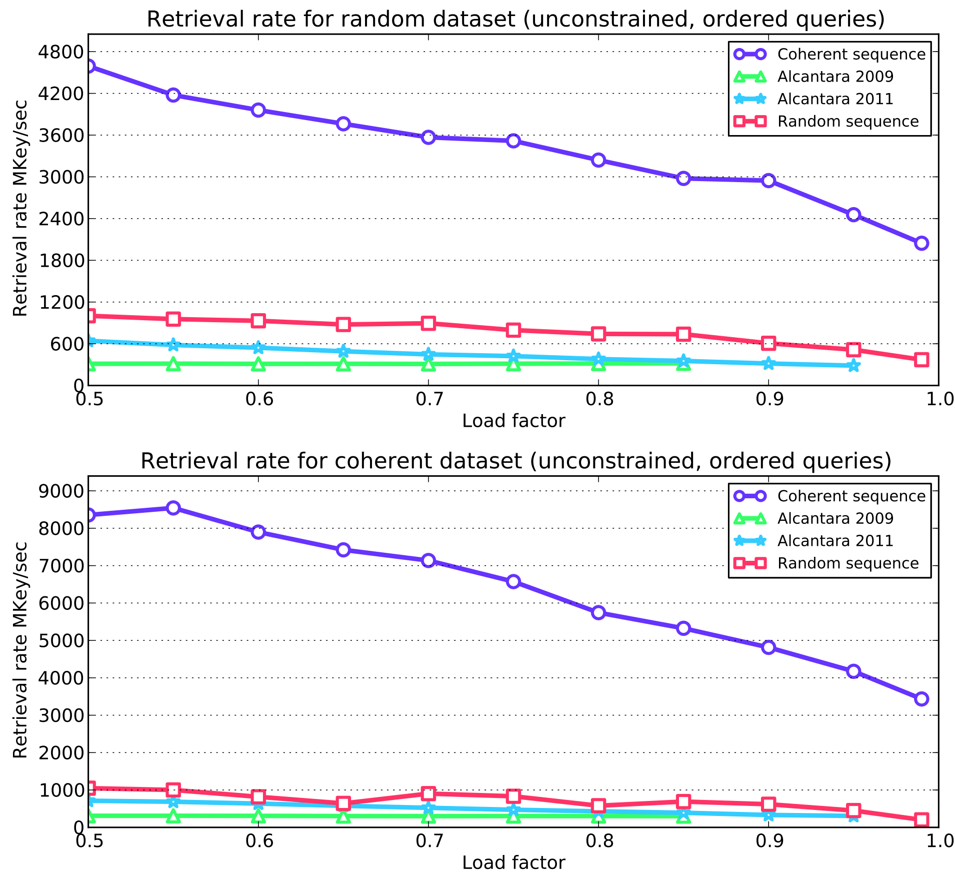


Figure 5.7: Access times for h_{rand} and h_{coh} and earlier schemes. Times are averaged over several runs. Missing data for Alcantara09 is due to construction failure at high load factors. *Top:* $8K \times 8K$ keys are queried, $1M$ of which, chosen at random, are defined. *Bottom:* Timings for the fish image: $8K \times 8K$ keys are queried, of which $20.5M$ are defined.

cache hit during queries, as shown Figure 5.8. No other scheme in our tests made any significant use of the cache. Figure 5.8 shows only L1 cache data. We ran additional tests to reveal further improvements in the number of L2 cache read requests and DRAM read requests, as reported in Figure 5.1 together with the measured branch divergence rate. In Figures 5.8 and 5.1, the data is taken from the above experiment with the fish data set.

5.3.2.2 Key layout

We now analyze the effect of different orderings of the 2D data in the 1D key universe. This is important as in many graphics applications of hashing, the keys are queried in a systematic and coherent way. For example, threads in a same group rasterize neighboring pixels that have neighboring texture coordinates, so we should strive to keep this coherence when translating the position or the texture coordinates into 1D keys. We test three orderings:

- The *linear row-major* order maps (x, y) to $x + Wy$ when W is the width of the (rectangular) domain: we should benefit from the coherent hash when we query neighboring keys on a same line of the domain.

- The *Morton* order maps (x, y) to $\mathcal{M}(x, y)$, the integer obtained by interleaving the bits of the binary representation of x and y. This improves locality along both X and Y axes

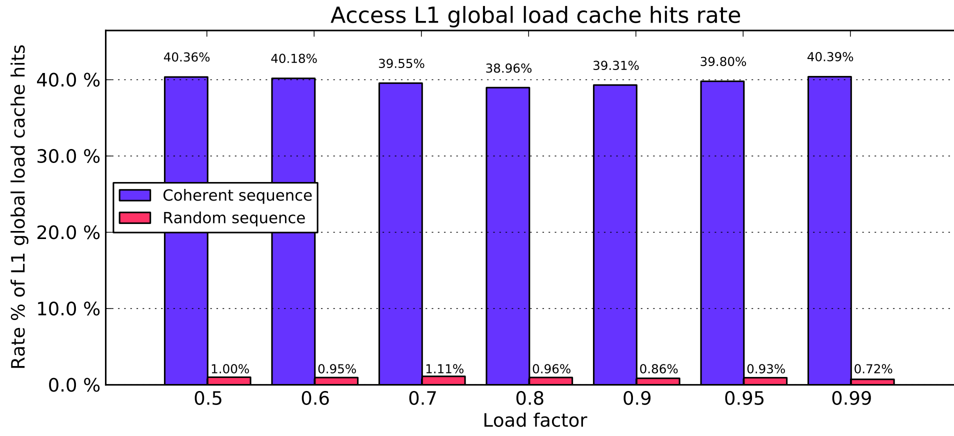


Figure 5.8: Percentage of L1 global cache hit during queries for the fish data set. The higher the better. Note that only our coherent probe sequence exhibits a significant cache reuse.

	Time	L2 requests	DRAM requests	Branch divergence
h_{rand}	97.1 ms	458.6 M	458.6 M	21.8 %
h_{coh}	12.6 ms	44.2 M	42.9 M	10.4 %

Table 5.1: L2 and DRAM read requests for the fish data set under 0.85 load factor.

(see further details in Section 2.4.1).

- The *bit-reversal* permutation $\sigma = (\sigma_i, i \in [0, 2^w))$ is obtained by reversing the bits of the binary representation $b_1^i b_2^i ... b_w^i$ of integer i: $\sigma_i = b_w^i b_{w-1}^i ... b_1^i$. For the experiment, we map (x, y) to $\mathcal{M}(\sigma_x, \sigma_y)$. This mapping exhibits no coherence at all.

The test consists in drawing a full-screen rotating image using a custom GLSL pixel shader to query the hash map. The latter is stored as a 2D texture and encodes the image color data using the three orderings above. Since GLSL offers fewer opportunities for optimization, the test reports overall lower performance than the CUDA implementation. The results are shown in Figure 5.9. The random probe sequence behaves roughly the same for each ordering and even gives slightly faster queries with the highly incoherent bit-reversal ordering. On the contrary, the coherent probe sequence gives significantly faster queries on the two other orderings since it leverages coherence in the access pattern. One can clearly see how increasingly coherent orderings translate into higher performance.

5.3.2.3 3D data

We have experimented with 3D data as well, consisting in voxelizations of the armadillo and hairy models (see Figure 5.11) in a grid of size 512^3. We hash 64 bits key-data pairs. Our experiments consisted in drawing slices of the volume at random orientations. The armadillo voxel data results in 9.2M keys. Insertion rate is 280 Mkeys/s under load factor $d = 0.8$ and 254 Mkeys/s at $d = 0.99$. Retrieval rate is 1905 Mkeys/s at $d = 0.8$ and 1182 Mkeys/s at $d = 0.99$. The hairy voxel data results in 24M keys. Insertion rate is 455 Mkeys/s at $d = 0.8$ and 182 Mkeys/s at $d = 0.99$. Retrieval rate is 1736 Mkeys/s at $d = 0.8$ and 1208 Mkeys/s at $d = 0.99$.

Regarding the key layout, we obtain similar results as the 2D results shown in Figure 5.9, with

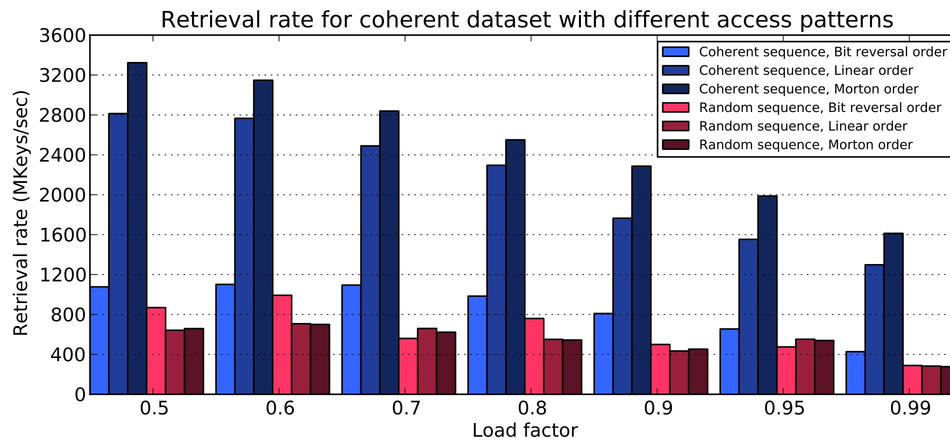


Figure 5.9: Query timings for h_{coh} using different ordering of the pixels, for the fish data set.

an even stronger advantage to the Morton ordering.

5.4 Applications

We demonstrate a sparse painting application relying on our hashing scheme, illustrated Figure 5.10. A 2D atlas is updated interactively while the user paints along the surface. Only the pixels touched by the brush are stored in the hash table. This lets us paint locally at very high resolution, while maintaining a low memory usage.

When the user paints on the model we retrieve the (u, v) coordinates of the pixels touched by the brush. If pixels are already in the hash table, we simply update their colors. If new pixels are touched we rebuild the hash table entirely: We first gather the new pixel key–color pairs and concatenate them with the current hash table from which we remove empty entries. This array is used as the input for building a new hash table. The entire process is fast enough to happen seamlessly while the user paints.

Some applications may choose to spend more memory in exchange for faster queries. This can happen seamlessly by simply rebuilding the hash table with a lower or higher load factor.

For the dataset and viewing conditions of Figure 5.10, at 0.8 load factor, our scheme with h_{rand} builds in 7 ms and reaches 299 FPS, and our scheme with h_{coh} builds in 3 ms and reaches 375 FPS.

5.5 Discussion, limitations and future work

Our scheme has two main drawbacks. First, in absence of coherence in the access patterns our scheme brings little to no benefit compared to a random probe sequence. Second, the max age table slows down construction and requires additional memory. Quantizing the max age could reduce this issue but not solve it entirely. Note that this is only problematic if empty keys are queried: In case of constrained access the max age table is not used and does not have to be built.

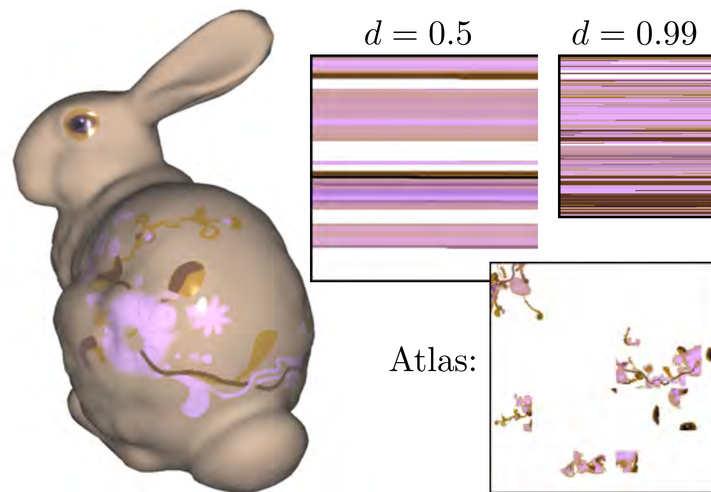


Figure 5.10: Our sparse painting application lets the user decorate an object with high resolution details. Only the painted pixels are stored, regardless of the resolution of the virtual texture. In this example the virtual texture has size 4096^2, among which $1M$ pixels are painted. For this 1024^2 viewpoint, 389586 queries are made. At 0.5 load factor the hash table is built in 3 ms and the display runs at 446 FPS, while at 0.99 load factor it is built in 10 ms and display runs at 237 FPS.

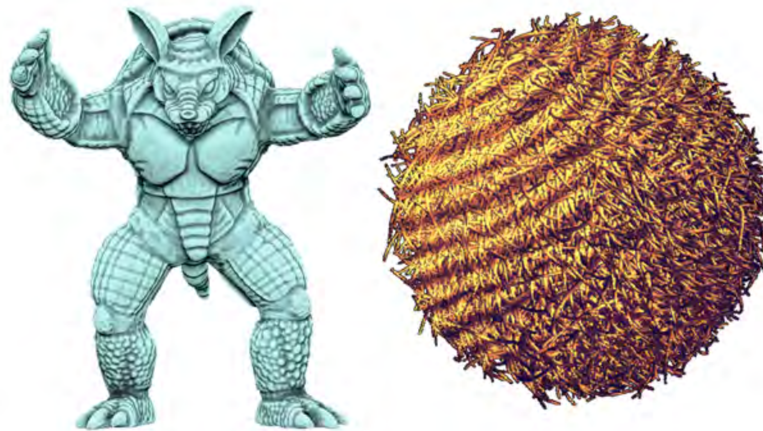


Figure 5.11: The armadillo and hairy color voxel data.

Future directions of research include handling deletion of keys as well as incremental insertions and deletions while maintaining a compact hash table.

5.6 Conclusion

Hashing is often synonymous of random access patterns. A remarkable result of our work is to demonstrate that coherence can be preserved, going against this common belief. As shown by our analysis coherence immediately translates to large improvements in cache behavior, and thus to large improvements in query time.

The CUDPP 2.0 implementation of [AVS+11], released concurrently to the publication of our work, also improves significantly the cache behavior. The authors now rely on more coherent hash functions, which resemble the translations of our coherent sequence. This is another strong indication that hashing can preserve memory access efficiency on parallel processors.

Of course, these results only hold when some degree of coherence is available, in the data for construction and in the access patterns for retrieval. This is why we strongly believe our hash is of particular interest for graphics applications, where spatial coherence is common – it was designed from the start with this goal in mind.

In addition, our hash reaches high load factors without failure and performs consistently well at all load factor settings. This is in contrast to cuckoo hashing which requires to manually select a fixed number of hash functions. Its code is very simple, there is no extra complexity to deal with restarts or to generate new hash functions. We thus believe it offers a strong alternative for storing sparse data in a computer graphics context.

Chapter 6

Conclusions and future work

6.1 Summary of contributions

In this thesis we have proposed a set of data structures with efficient parallel query evaluation for mapping shape and shading information on geometries of objects. We addressed this problem from three different perspectives, each of them can be preferred to the other two depending on the specific requirements the application scenario.

To begin with, we decoupled high-resolution irregular triangle meshes from the shape and shading attributes during the simplification process. Our proposed solution avoids the limiting issues of geometric and attribute preservation during the simplification process, thanks to the definition of a decoupling parameterization and an inverse map to create the correspondence between the simplified geometry and the original mesh structure attributes. In this way, for any simplified mesh, we can directly map the original attributes without resampling them, in cases where this procedure would not be recommended.

However, neither the arbitrarily choosen simplification process nor our proposed decoupling parameterization, were designed to completely ensure a bijection (with a 1-to-1 map) from any simplified model to the original high-resolution one. Furthermore, the latest GPUs introduce tessellation units that allow more flexible and efficient level-of-detail techniques based on tessellation strategies, such as subdivision surfaces, rather than traditional interactive DLOD and CLOD techniques [LWC+02].

For this reason, we proposed a data structure based on a polycube mapping to allow a user-friendly sketch-based parameterization of irregular triangle meshes. The proposed method creates a quad-based representation, defined in such a way that bijectively preserves the shape details of the given triangle mesh. This solution proves useful when resampling the original data into a new semi-regular geometric structure is preferred to ease user modeling operations and to improve the rendering performance with hardware tessellation units. We demonstrate its feasibility through hardware-friendly subdivision surfaces with displacement mapping, and by integrating their representation in further modeling applications exploiting our sketch-based interface.

Nonetheless, we still found that, in some cases, many applications produce geometries which can be topologically inconsistent (e.g. when directly extracted from a 3D scanning), as they are sparse by nature or even defined directly by implicit surfaces. In any case, these situations can not be easily handled by parameterization methods with neither of our previously proposed

solutions.

Therefore, we have explored a third alternative data structure to pack the spatial data information of such complex geometries, in a compact way, and provide an efficient construction and query evaluation; all without requiring a parameterization. Our proposed data structure is based on spatial hashing and exploit the coherence in the spatial data and in the access pattern to pack and map the shape and shading details. Our experiments have shown that it allows faster construction than previous spatial hashing methods and most parameterizations, and with almost full memory utilization and a similar query access performance as that the obtained with the parameterized mappings. However, the proposed spatial hashing is not flexible enough to capture multi-scale phenomena when compared to tree-based and adaptive grid-based data structures.

In general each specific problem may require to choose one of the different proposed solutions. Nevertheless, all of them still have room for improvement. Fortunately, there are also many new challenges to be pursued in the quest for more flexible and efficient parallel data structures. In next section we will briefly describe our future work perspectives.

6.2 Perspectives and future work

In this thesis, we tackled the problem of level-of-detail techniques coupled with detail preservation and efficient parallel query evaluation in modeling and rendering applications. We see several interesting future research directions based on our work, as described below.

6.2.1 Real-time simplification and parallel localization with random access

We have developed a level-of-detail technique which decouples and evaluates the shape and shading attributes of high resolution geometries over any generated level-of-detail. While the query evaluation is efficiently processed in the programmable stages of the GPU, the preprocessing step to map the surface into a suitable parametric domain, and the creation of the localizing spatial data directory, is an expensive preprocessing step on the CPU. Therefore, our approach is limited the be used on static high-resolution geometries that do not drastically change their mesh structure or topology during the interactive visualization. Also, we require the levels-of-detail to closely resemble the original shape.

Therefore, dynamic spatial data captured by real-time capturing sensor devices cannot be prepared for LOD with IGT, because the simplification and the decoupling parameterization would need to be interactively computed for each captured frame.

One of the focuses of our future work will be to investigate alternative real-time simplification methods [Wil11], interactive parameterization [TSS$^+$11] methods and other spatial data structures suitable to accelerating geometric queries on dynamically changing data.

6.2.2 Efficient and robust semi-automatic parallel parameterization with interactive control

We exploited the embedding of simple polycubes as parameter domains to generate quad-based representations with several good properties: semi-regularity, few irregular vertices, uniform

tessellation densities and well-shaped quads. However, the task of automatically (or semi-automatically) producing this kind of parameterization in a robust way is still an open problem.

In this context, it could be of great interest to be able to automatically create a polycube that fits the most relevant singular vertices of a given triangle mesh to correspond with subset of the irregular vertices of the polycube domain, while at the same time being locally bijective, and in general, having an overall shape not too different from the one in the represented object.

In terms of efficiency and interactive control, in many cases the mapping may require user adjustments, best achieved through direct manipulation from a high-level generated abstraction, not necessarily a polycube [TPP+11]. This requires even more efficient parallel algorithms to modify the resulting quad mesh and computate the global parameterization.

Finally, in terms of robustness, many capturing sensor devices produce geometric spatial data with topological noise or artifacts that typically require many preprocessing operations before being able to extract high-level parametric abstractions. New methods that can efficiently take the raw spatial data to directly and robustly create such abstractions [PTSZ11] are of great interest.

6.2.3 Succinct hashing schemes, dynamic hash tables and variable-length data elements

The performance of hashing schemes is measured mainly by its construction, lookup, and space consumption properties.

In terms of the construction process, one interesting property is that we provide almost full memory utilization from the hash table size, achieving high load factors. Nonetheless, an interesting new feature would be to adapt it to be a dynamic dictionary with efficient parallel key deletion and updates. We believe the deletion algorithm proposed by Celis [Cel86] could be implemented. However, this is not straightforward since it requires a more complex management of the access and the max age table. A simpler approach would be to directly erase the keys since the query operation ignores empty slots. This would require updating the max age table. Fortunately, detecting the key with maximum age to be deleted is easy, though the behavior of the hash after many deletions and insertions should be further analyzed.

In terms of lookup evaluation, the random-access queries are still memory-bound by long latencies. Therefore, beyond exploiting the coherency of the spatial data and improving the coherency of the memory access pattern and the execution paths, we found that the global memory bandwidth is still limited, so prefetching methods [KH10] can probably help in some specific cases. New experiments are required to understand its application.

Also, when all the threads are waiting for their memory access results, and there is a very small number of independent instructions between the memory access instructions and the consumption of the data accessed, large memory latencies appear. A useful, complementary solution to this problem is to prefetch the next data elements while consuming the current data elements, which increases the number of independent instructions between the memory accesses and the computation instructions of the accessed data.

In terms of space consumption, in our current approach we store in the records of the hash table $D[u]$ the key along the data $< k, age, data >$. While in some specific applications the key k can be useful if stored, in other ones which only require the $data$ to be queried, the space overhead

of storing the key is not justified. Therefore, adaptation to a succinct hashing scheme, requiring a lower space bound, would be interesting.

For this, we have a coherent hash function defined with translations at every age step, from an offset table $o[]$, and many applications have a spatial domain with a bounded size S, and only N defined keys to be stored in a hash table D (e.g. as it happens for instance in raytracing applications). An interesting observation is that we are using the same translations for all keys (i.e. the permutation offsets are in range hash table size $|D|$, $[0..|D|-1]$ stored in $o[]$). Therefore, only keys whose k are such that $u = (k + o[age]) mod |D|$, they can collide. In general, we can at most have S/H of those keys colliding (i.e. only those $|D|$ keys appart). Hence, we could store the age and the multiplicative factor r (as $r = k/|D|$) instead of k, as $< r, age, data >$. The key point is that, the smaller the size $|D|$ of the hash table to store the N defined keys, the better the ratio we will obtain when storing $r = k/|D|$, which is much less expensive than storing the coordinates k.

Finally, we also believe that some applications do require variably-sized data to be stored in dictionaries such as hash tables. The envision of parallel hashing schemes for variable-length data is also of great interest, because traditional hashing schemes have generally focused on external storage of data records (out of the hash table), indexed by sparse integers. Consequently, these schemes are generally not well adapted for their use with spatially coherent multidimensional data, which happens in many computer graphics applications (e.g. vector textures, or variable-length 3D layers of surface data), where the coherent access cannot be exploited with typical hash functions. A proposed solution [Hop07] requires costly CPU preprocessing to define the perfect hash of such variable-rate data, and more parallel-friendly construction approaches would be of great interest.

Bibliography

[And99] Carlos Andújar. Octree-based simplifications of polyhedral solids, 1999.

[App68] Arthur Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, AFIPS '68 (Spring), pages 37–45, New York, NY, USA, 1968. ACM.

[Arv88] J. Arvo. Linear-time voxel walking for octrees, 1988. Ray Tracing News 12(1).

[ASA+09] Dan A. Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta. Real-time parallel hashing on the gpu. *ACM Trans. Graph.*, 28:154:1–154:9, December 2009.

[AVS+11] Dan Alcantara, Vasily Volkov, Shubhabrata Sengputa, Michael Mitzenmacher, John Owens, and Nina Amenta. *Building an Efficient Hash Table on the GPU*. Morgan Kaufmann, 2011.

[BD02] David Benson and Joel Davis. Octree textures. *ACM Trans. Graph.*, 21:785–790, July 2002.

[BF79] Jon Louis Bentley and Jerome H. Friedman. Data structures for range searching. *ACM Comput. Surv.*, 11:397–409, December 1979.

[BHGS06] Tamy Boubekeur, Wolfgang Heidrich, Xavier Granier, and Christophe Schlick. Volume-surface trees. *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2006)*, 25(3):399–406, 2006.

[Ble90] Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.

[Bli96] Jim Blinn. *Jim Blinn's corner: a trip down the graphics pipeline*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[BS88] A. A. Ball and D. J. T. Storry. Conditions for tangent plane continuity over recursively generated b-spline surfaces. *ACM Trans. Graph.*, 7:83–102, April 1988.

[Bun05] Michael Bunnell. *GPU Gems 2*, chapter Adaptive Tessellation of Subdivision Surfaces with Displacement Mapping. Addison Wesley, 2005.

[Bur81] P J Burt. Fast filter transforms for image processing. *Computer Graphics and Image Processing*, 16(1):20–51, 1981.

[BZ07] Fabiano C. Botelho and Nivio Ziviani. External perfect hashing for very large key sets. In *Proceedings of the sixteenth ACM Conference on Conference on Information and Knowledge Management*, CIKM '07, pages 653–662. ACM, 2007.

[BZK09] David Bommes, Henrik Zimmer, and Leif Kobbelt. Mixed-integer quadrangulation. *ACM Trans. Graph.*, 28(3):1–10, 2009.

[C06] Chih-Chun Chen 0002 and Jung-Hong Chuang. Texture adaptation for progressive meshes. *Comput. Graph. Forum*, 25(3):343–350, 2006.

[Cas08a] Ignacio Castaño. Next-generation rendering of subdivision surfaces. SIGGRAPH 2008, 2008.

[Cas08b] Ignacio Castaño. Tessellation of subdivision surfaces in direct3d 11. Gamefest 2008, 2008.

[Cas09] Ignacio Castaño. Ownership-based zippering. http://castano.ludicon.com/blog/2009/01/10/ownership-based-zippering/, 2009.

[Cat74] Edwin Earl Catmull. *A subdivision algorithm for computer display of curved surfaces.* PhD thesis, 1974. AAI7504786.

[CC98] E. Catmull and J. Clark. *Recursively generated B-spline surfaces on arbitrary topological meshes*, pages 183–188. ACM, New York, NY, USA, 1998.

[Cel86] Pedro Celis. *Robin Hood Hashing*. PhD thesis, University of Waterloo, Ontario, Canada, 1986.

[CH02a] Nathan A. Carr and John C. Hart. Meshed atlases for real-time procedural solid texturing. *ACM Trans. Graph.*, 21:106–131, April 2002.

[CH02b] Nathan A. Carr and John C. Hart. Meshed atlases for real-time procedural solid texturing. *ACM Trans. Graph.*, 21(2):106–131, 2002.

[Cho45] G. Choquet. Sur un type de transformation analytique généralisant la représentation conforme et défini au moyen de fonctions harmoniques. *Bulletin des Sciences Mathématiques*, 69:156–165, 1945.

[Cla76] J. H. Clark. Hierarchical geometric models for visible surface algorithms, 1976. Communications of the ACM, Vol. 19, No. 10, pp. 547-554.

[CLE06] *Evaluation of suprathreshold perceptual metrics for 3d models*, New York, NY, USA, 2006. ACM. 106063.

[CMR+99] P. Cignoni, C. Montani, C. Rocchini, R. Scopigno, and M. Tarini. Preserving attribute values on simplified meshes by resampling detail textures. *The Visual Computer*, 15(10):519–539, 1999.

[CNLE09] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. Gigavoxels : Ray-guided streaming for efficient and detailed voxel rendering, feb 2009. to appear.

[COM98] Jonathan Cohen, Marc Olano, and Dinesh Manocha. Appearance-preserving simplification. *Computer Graphics (Proc. SIGGRAPH)*, 32:115–122, 1998.

[CRS98] P. Cignoni, C. Rocchini, and R. Scopigno. Metro: Measuring error on simplified surfaces. *Computer Graphics Forum*, 17(2):167–174, 1998.

[CWQ+07] Kin-Shing Cheng, Wenping Wang, Hong Qin, Kwan-Yee Wong, Huaiping Yang, and Yang Liu. Design and analysis of optimization methods for subdivision surface fitting. *IEEE Transactions on Visualization and Computer Graphics*, 13:878–890, 2007.

[DBG+06] Shen Dong, Peer-Timo Bremer, Michael Garland, Valerio Pascucci, and John C. Hart. Spectral surface quadrangulation. *ACM Trans. Graph.*, 25(3):1057–1066, 2006.

[DGPR02] David (grue) DeBry, Jonathan Gibbs, Devorah DeLeon Petty, and Nate Robins. Painting and rendering textures on unparameterized models. *ACM Trans. Graph.*, 21:763–768, July 2002.

[DLS07] Tamal K. Dey, Kuiyu Li, and Jian Sun. On computing handle and tunnel loops. In *In IEEE Proc. NASAGEM 07*, pages 357–366, 2007.

[DLSCS08] Tamal K. Dey, Kuiyu Li, Jian Sun, and David Cohen-Steiner. Computing geometry-aware handle and tunnel loops in 3d models. *ACM Trans. Graph.*, 27:45:1–45:9, August 2008.

[DMK03a] P. Degener, J. Meseth, and R. Klein. An adaptable surface parameterization method. In *In Proceedings of the 12th International Meshing Roundtable*, pages 201–213, 2003.

[DMK03b] Patrick Degener, Jan Meseth, and Reinhard Klein. An adaptable surface parameterization method. In *IMR '03*, pages 201–213, 2003.

[DMV04] Luc Devroye, Pat Morin, and Alfredo Viola. On worst-case robin hood hashing. *SIAM Journal on Computing*, 33:923–936, 2004.

[DS98] D. Doo and M. Sabin. *Behaviour of recursive division surfaces near extraordinary points*, pages 177–181. ACM, New York, NY, USA, 1998.

[DT07] Christopher DeCoro and Natalya Tatarchuk. Real-time mesh simplification using the gpu. In *SI3D '07: Proceedings of the 2007 Symposium on Interactive 3D graphics and games*, page TBD, New York, NY, USA, 2007. ACM Press.

[EDD+95a] M. Eck, T. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery, and W. Stuetzle. Multiresolution analysis of arbitrary meshes. In *SIGGRAPH '95*, pages 173–182, 1995.

[EDD+95b] Matthias Eck, Tony DeRose, Tom Duchamp, Hugues Hoppe, Michael Lounsbery, and Werner Stuetzle. Multiresolution analysis of arbitrary meshes. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '95, pages 173–182, New York, NY, USA, 1995. ACM.

[FC09] Fengtao Fan and Fuhua Cheng. Gpu supported patch-based tessellation for dual subdivision. In *Computer Graphics, Imaging and Visualization, 2009. CGIV '09. Sixth International Conference on*, pages 5 –10, aug. 2009.

[FH05] Michael S. Floater and Kai Hormann. Surface parameterization: a tutorial and survey. In N. A. Dodgson, M. S. Floater, and M. A. Sabin, editors, *Advances in multiresolution for geometric modelling*, pages 157–186. Springer Verlag, 2005.

[FHCD92] Edward A. Fox, Lenwood S. Heath, Qi Fan Chen, and Amjad M. Daoud. Practical minimal perfect hash functions for large databases. *Commun. ACM*, 35(1):105–121, January 1992.

[Fie88] David A Field. Laplacian smoothing and delaunay triangulations. *Communications in Applied Numerical Methods*, 4:709–712, 1988.

[FKS+04] Thomas Funkhouser, Michael Kazhdan, Philip Shilane, Patrick Min, William Kiefer, Ayellet Tal, Szymon Rusinkiewicz, and David Dobkin. Modeling by example. *ACM Trans. Graph.*, 23(3):652–663, 2004.

[Flo97a] Michael S. Floater. Parametrization and smooth approximation of surface triangulations. *Comput. Aided Geom. Des.*, 14:231–250, April 1997.

[Flo97b] Michael S. Floater. Parametrization and smooth approximation of surface triangulations. *Computer Aided Geometric Design*, 14:231–250, 1997.

[GGH02a] Xianfeng Gu, Steven J. Gortler, and Hugues Hoppe. Geometry images. *ACM Trans. Graph.*, 21:355–361, July 2002.

[GGH02b] Xianfeng Gu, Steven J. Gortler, and Hugues Hoppe. Geometry images. *ACM Trans. Graph.*, 21(3):355–361, 2002.

[GH86] Ned Greene and Paul S. Heckbert. Creating raster omnimax images from multiple perspective views using the elliptical weighted average filter. *IEEE Comput. Graph. Appl.*, 6(6):21–27, June 1986.

[GH97] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '97, pages 209–216, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[GH98] Michael Garland and Paul S. Heckbert. Simplifying surfaces with color and texture using quadric error metrics. In *Proceedings of the conference on Visualization '98*, VIS '98, pages 263–269, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[Gla84] Andrew S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics & Applications*, 4(10):15–22, October 1984.

[GLHL11] Ismael Garcia, Sylvain Lefebvre, Samuel Hornus, and Anass Lasram. Coherent parallel hashing. *ACM Trans. Graph.*, 30(6):161:1–161:8, December 2011.

[GP08] Ismael Garcia and Gustavo Patow. Igt: inverse geometric textures. *ACM Trans. Graph.*, 27(5):137:1–137:9, December 2008.

[GP09] Francisco González and Gustavo Patow. Continuity mapping for multi-chart textures. *ACM Trans. Graph.*, 28(5):109:1–109:8, December 2009.

[GPSSK07] Ismael Garcia, Gustavo Patow, Mateu Sbert, and Laszlo Szirmay-Kalos. Multi-layered indirect texturing for tree rendering. In S. Mérillou D. Ebert, editor, *Eurographics Workshop on Natural Phenomena*, 2007.

[GSZ11] James Gregson, Alla Sheffer, and Eugene Zhang. All-hex mesh generation via volumetric polycube deformation. *Computer Graphics Forum*, 30(5):1407¨C1416, 2011.

[GVSS00] Igor Guskov, Kiril Vidimče, Wim Sweldens, and Peter Schröder. Normal meshes. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '00, pages 95–102, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[GW07] Markus Giegl and Michael Wimmer. Unpopping: Solving the image-space blend problem for smooth discrete lod transitions. *Computer Graphics Forum*, 26(1):46–49, March 2007.

[GWH01] Michael Garland, Andrew Willmott, and Paul S. Heckbert. Hierarchical face clustering on polygonal surfaces. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, I3D '01, pages 49–58, New York, NY, USA, 2001. ACM.

[GXH12] Editable polycube map for gpu-based subdivision surfaces, 2012. (submitted).

[HB10] Jared Hoberock and Nathan Bell. Thrust: A parallel template library, 2010. Version 1.3.0.

[HDD+93] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh optimization. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '93, pages 19–26, New York, NY, USA, 1993. ACM.

[HDD+94] Hugues Hoppe, Tony DeRose, Tom Duchamp, Mark Halstead, Hubert Jin, John McDonald, Jean Schweitzer, and Werner Stuetzle. Piecewise smooth surface reconstruction. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, SIGGRAPH '94, pages 295–302, New York, NY, USA, 1994. ACM.

[Hec89] Paul S. Heckbert. Fundamentals of texture mapping and image warping. Technical report, Berkeley, CA, USA, 1989.

[HFAT07] Xiaohuang Huang, Hongbo Fu, Oscar Kin-Chung Au, and Chiew-Lan Tai. Optimal boundaries for poisson mesh merging. In *SPM '07*, 2007.

[HG00] K. Hormann and G. Greiner. MIPS: An efficient global parametrization method. In P.-J. Laurent, P. Sablonnière, and L. L. Schumaker, editors, *Curve and Surface Design: Saint-Malo 1999*, Innovations in Applied Mathematics, pages 153–162. Vanderbilt University Press, Nashville, TN, 2000.

[HG11] Mark Harris and Michael Garland. *Optimizing Parallel Prefix Operations for the Fermi Architecture*, pages 253–269. Morgan Kaufmann, Waltham, MA, 2011.

[HH90] Pat Hanrahan and Paul Haeberli. Direct wysiwyg painting and texturing on 3d shapes:. *SIGGRAPH '90*, 24(4):215–223, 1990.

[Hop96] Hugues Hoppe. Progressive meshes. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '96, pages 99–108, New York, NY, USA, 1996. ACM.

[Hop99] Hugues H. Hoppe. New quadric metric for simplifying meshes with appearance attributes. In David Ebert, Markus Gross, and Bernd Hamann, editors, *IEEE Visualization '99*, pages 59–66, San Francisco, 1999.

[Hop07] Hugues Hoppe. Perfect hashing of variably-sized data, 10 2007.

[Hor05] Daniel Horn. Stream reduction operations for gpgpu applications. In Matt Pharr, editor, *GPU Gems 2*, pages 573–589. Addison-Wesley, 2005.

[HSH10] Liang Hu, Pedro V. Sander, and Hugues Hoppe. Parallel view-dependent level-of-detail control. *IEEE Trans. Vis. Comput. Graph.*, 16(5):718–728, 2010.

[HSO07] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel prefix sum (scan) with CUDA. In Hubert Nguyen, editor, *GPU Gems 3*, chapter 39, pages 851–876. Addison Wesley, August 2007.

[HSV05] Toon Huysmans, Jan Sijbers, and Brigitte Verdonk. Parameterization of tubular surfaces on the cylinder. *The journal of WSCG*, pages 97–104, 2005.

[HWFQ09] Ying He, Hongyu Wang, Chi-Wing Fu, and Hong Qin. A divide-and-conquer approach for automatic polycube map construction. *Computer & Graphics*, 33(3):369–380, 2009.

[HZM+08] Jin Huang, Muyang Zhang, Jin Ma, Xinguo Liu, Leif Kobbelt, and Hujun Bao. Spectral quadrangulation with orientation and alignment control. *ACM Trans. Graph.*, 27(5):147, 2008.

[IWR+06] Thiago Ize, Ingo Wald, Chelsea Robertson, Steven G. Parker, Thiago Ize, Ingo Wald, Chelsea Robertson, and Steven G. Parker. An evaluation of parallel grid construction for ray tracing dynamic scenes. In *In Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 47–55, 2006.

[Joh05] Streaming architectures and technology trends, 2005. In GPU Gems 2, pages 457-470.

[Ket99] Lutz Kettner. Using generic programming for designing a data structure for polyhedral surfaces. *Comput. Geom. Theory Appl*, 13:65–90, 1999.

[KH10] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[Khr08] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*, 8 December 2008.

[KJS07] Vladislav Kreavoy, Dan Julius, and Alla Sheffer. Model composition from interchangeable components. In *Proceedings of the 15th Pacific Conference on Computer Graphics and Applications*, pages 129–138, 2007.

[KK89] J. T. Kajiya and T. L. Kay. Rendering fur with three dimensional textures. *Computer Graphics (Proc. SIGGRAPH)*, 23:271–280, 1989.

[Kle98] Reinhard Klein. Multiresolution representations for surfaces meshes based on the vertex decimation method. *Computers and Graphics*, 22(1):13–26, 1998.

[KLS96] Reinhard Klein, Gunther Liebich, and Wolfgang Strasser. Mesh reduction with error control. In *Proceedings of the 7th conference on Visualization '96*, VIS '96, pages 311–318, Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.

[KLS03] A. Khodakovsky, N. Litke, and P. Schröder. Globally smooth parameterizations with low distortion. *ACM Trans. Graph.*, 22:350–357, 2003.

[Kne26] H. Kneser. Lösung der aufgabe 41. *Jahresbericht der Deutschen Mathematiker-Vereinigung*, 35:123–124, 1926.

[KNP07] Felix Kälberer, Matthias Nieser, and Konrad Polthier. Quadcover - surface parameterization using branched coverings. *Comput. Graph. Forum*, 26(3):375–384, 2007.

[KS04] V. Kreavoy and A. Sheffer. Cross-parameterization and compatible remeshing of 3d models. *ACM Trans. Graph.*, 23:861–869, 2004.

[KZHCO10] Oliver Kaick, van, Hao Zhang, Ghassan Hamarneh, and Daniel Cohen-Or. A survey on shape correspondence. In *Proc. of Eurographics State-of-the-art Report*, pages 1–24, 2010.

[LAM05] Marta Lofsted and Tomas Akenine-Moller. An evaluation framework for ray-triangle intersection algorithms. *Journal of Graphics Tools*, 10(2):13–26, 2005.

[LD07] Sylvain Lefebvre and Carsten Dachsbacher. Tiletrees. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, I3D '07, pages 25–31, New York, NY, USA, 2007. ACM.

[LD08] Ares Lagae and Philip Dutré. Compact, fast and robust grids for ray tracing. *Computer Graphics Forum (Proceedings of the 19th Eurographics Symposium on Rendering)*, 27(4):1235–1244, June 2008.

[LDSS99] A. W. F. Lee, D. Dobkin, W. Sweldens, and P. Schröder. Multiresolution mesh morphing. In *SIGGRAPH '99*, pages 343–350, 1999.

[LE97] David Luebke and Carl Erikson. View-dependent simplification of arbitrary polygonal environments. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '97, pages 199–208, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[Lef06] Aaron E. Lefohn. *Glift: generic data structures for graphics hardware*. PhD thesis, Davis, CA, USA, 2006. AAI3236026.

[LGS+] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast BVH construction on GPUs. *Computer Graphics Forum*, 28(2):375–384.

[LGW+07] X. Li, X. Guo, H. Wang, Y. He, X. Gu, and H. Qin. Harmonic volumetric mapping for solid modeling applications. In *Proc. ACM symp. on Solid and physical modeling*, pages 109–120, 2007.

[LH06a] Sylvain Lefebvre and Hugues Hoppe. Perfect spatial hashing. *ACM Trans. Graph.*, 25:579–588, July 2006.

[LH06b] Sylvain Lefebvre and Hugues Hoppe. Perfect spatial hashing. In *ACM SIGGRAPH 2006 Papers*, SIGGRAPH '06, pages 579–588, New York, NY, USA, 2006. ACM.

[LHN05] Sylvain Lefebvre, Samuel Hornus, and Fabrice Neyret. Gpu gems 2 - programming techniques for high-performance graphics and general-purpose computation, 2005.

[LHP11] Luc Leblanc, Jocelyn Houle, and Pierre Poulin. Modeling with blocks. *The Visual Computer (Proc. Computer Graphics International 2011)*, 27(6-8):555–563, June 2011.

[LJFW08] Juncong Lin, Xiaogang Jin, Zhengwen Fan, and Charlie C. L. Wang. Automatic polycube-maps. In *GMP'08*, pages 3–16, 2008.

[LK10] Samuli Laine and Tero Karras. Efficient sparse voxel octrees. In *Proceedings of ACM SIGGRAPH 2010 Symposium on Interactive 3D Graphics and Games*, pages 55–63. ACM Press, 2010.

[LKR+96] Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hodges, Nick Faust, and Gregory A. Turner. Real-ti continuous level of detail rendering of height fields. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '96, pages 109–118, New York, NY, USA, 1996. ACM.

[LLS01] Nathan Litke, Adi Levin, and Peter Schröder. Fitting subdivision surfaces. In *VIS '01*, pages 319–324, 2001.

[LN06] Sandrine Lanquetin and Marc Neveu. Reverse catmull-clark subdivision. In *WSCG '06*, 2006.

[Loo87] C. Loop. Smooth subdivision surfaces based on triangles. Department of mathematics, University of Utah, Utah, USA, August 1987.

[LPRM02] Bruno Lévy, Sylvain Petitjean, Nicolas Ray, and Jérome Maillot. Least squares conformal maps for automatic texture atlas generation. *ACM Trans. Graph.*, 21(3):362–371, 2002.

[LRZM11] Guiqing Li, Canjiang Ren, Jiahua Zhang, and Weiyin Ma. Approximation of loop subdivision surfaces for fast rendering. *Visualization and Computer Graphics, IEEE Transactions on*, 17(4):500 –514, april 2011.

[LS96] Nathan Linial and Ori Sasson. Non-expansive hashing. In *Proceedings of the twenty-eighth annual ACM Symposium on Theory of Computing*, STOC '96, pages 509–518. ACM, 1996.

[LS08a] Charles Loop and Scott Schaefer. Approximating catmull-clark subdivision surfaces with bicubic patches. *ACM Trans. Graph.*, 27(1):8:1–8:11, March 2008.

[LS08b] Charles Loop and Scott Schaefer. Approximating catmull-clark subdivision surfaces with bicubic patches. *ACM Trans. Graph.*, 27(1):1–11, 2008.

[LSK+06] Aaron E. Lefohn, Shubhabrata Sengupta, Joe Kniss, Robert Strzodka, and John D. Owens. Glift: Generic, efficient, random-access gpu data structures. *ACM Trans. Graph.*, 25:60–99, January 2006.

[LSNCn09] Charles Loop, Scott Schaefer, Tianyun Ni, and Ignacio Castaño. Approximating subdivision surfaces with gregory patches for hardware tessellation. *ACM Trans. Graph.*, 28(5):1–9, 2009.

[LT97] Kok-Lim Low and Tiow-Seng Tan. Model simplification using vertex-clustering. In *Proceedings of the 1997 symposium on Interactive 3D graphics*, I3D '97, pages 75–ff., New York, NY, USA, 1997. ACM.

[LWC+02] David Luebke, Benjamin Watson, Jonathan D. Cohen, Martin Reddy, and Amitabh Varshney. *Level of Detail for 3D Graphics*. Elsevier Science Inc., New York, NY, USA, 2002.

[LWW06] Zhouchen Lin, Lifeng Wang, and Liang Wan. First order approximation for texture filtering, 206.

[MCK08] Tobias Martin, Elaine Cohen, and Mike Kirby. Volumetric parameterization and trivariate b-spline fitting using harmonic functions. In *Proceedings of the 2008 ACM symposium on Solid and physical modeling*, SPM '08, pages 269–280, New York, NY, USA, 2008. ACM.

[MG11] Duane Merrill and Andrew Grimshaw. High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing. *Parallel Processing Letters*, 21(02):245–272, 2011.

[MKFC01] T. Michikawa, T. Kanai, M. Fujita, and H. Chiyokura. Multiresolution interpolation meshes. In *PG '01*, pages 60–69, 2001.

[Moo02] Mootools. Polygon cruncher, 2002. http://www.mootools.com/.

[Mor01] Henry Moreton. Watertight tessellation using forward differencing. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, HWWS '01, pages 25–32, New York, NY, USA, 2001. ACM.

[MPFJ99] Joel McCormack, Ronald Perry, Keith I. Farkas, and Norman P. Jouppi. Feline: fast elliptical lines for anisotropic texture mapping. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '99, pages 243–250, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[MYV93] J. Maillot, H. Yahia, and A. Verroust. Interactive texture mapping. In *Proc. of SIGGRAPH-93: Computer Graphics*, pages 27–34, Anaheim, CA, 1993.

[NH08] Diego Nehab and Hugues Hoppe. Random-access rendering of general vector graphics. In *ACM SIGGRAPH Asia 2008 papers*, SIGGRAPH Asia '08, pages 135:1–135:10, New York, NY, USA, 2008. ACM.

[Nvi08] Nvidia. Geforce 8800 whitepaper, 2008.

[Nvi11] Nvidia. Fermi compute architecture whitepaper, 2011.

[Oli06] Gary Oliverio. *Maya 8: Character Modeling*. Jones & Bartlett Publishers, 2006.

[PBFJ05] Serban D. Porumbescu, Brian Budge, Louis Feng, and Kenneth I. Joy. Shell maps. *ACM Trans. Graph.*, 24(3):626–633, 2005.

[Pet57] W. W. Peterson. Addressing for random-access storage. *IBM Journal of Research and Development*, 1(2):130–146, 1957.

[PG01] Mark Pauly and Markus Gross. Spectral processing of point-sampled geometry. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, pages 379–386, New York, NY, USA, 2001. ACM.

[PH03] Emil Praun and Hugues Hoppe. Spherical parametrization and remeshing. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pages 340–349, New York, NY, USA, 2003. ACM.

[Pix10] Pixologic. Zbrush. http://www.pixologic.com/, 2010.

[PLB07] Zack Petroc, Kevin Lanning, and Timur Baysal. *Character Modeling 2*. Ballistic Publishing, 2007.

[PPT+11] Daniele Panozzo, Enrico Puppo, Marco Tarini, Nico Pietroni, and Paolo Cignoni. Automatic construction of adaptive quad-based subdivision surfaces using fitmaps. *IEEE Trans. Vis. and Comp. Graph.*, 17(10):1510–1520, october 2011.

[PR98] Jörg Peters and Ulrich Reif. Analysis of algorithms generalizing b-spline subdivision. *SIAM J. Numer. Anal.*, 35:728–748, April 1998.

[PR04] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[PSBM07] Valerio Pascucci, Giorgio Scorzelli, Peer-Timo Bremer, and Ajith Mascarenhas. Robust on-line computation of reeb graphs: simplicity and speed. *ACM Trans. Graph.*, 26, July 2007.

[PSS01] E. Praun, W. Sweldens, and P. Schröder. Consistent mesh parameterizations. In *SIGGRAPH '01*, pages 179–184, 2001.

[PTSZ11] Nico Pietroni, Marco Tarini, Olga Sorkine, and Denis Zorin. Global parametrization of range image sets. *ACM Transactions on Graphics (proceedings of ACM SIGGRAPH ASIA)*, 30(6):149:1–149:10, 2011.

[Rad26] T. Radó. Aufgabe 41. *Jahresbericht der Deutschen Mathematiker-Vereinigung*, 35:49+, 1926.

[RB93] Jarek Rossignacand and Paul Borrel. Multi-resolution 3d approximations for rendering complex scenes. In *Modeling in Computer Graphics: Methods and Applications*, pages 455–465, 1993.

[RBW04] Ganesh Ramanarayanan, Kavita Bala, and Bruce Walter. Feature-based textures. In *Rendering Techniques*, pages 265–274, 2004.

[RLD+12] Tim Reiner, Sylvain Lefebvre, Lorenz Deiner, Ismael Garcia, Bruno Jobard, and Carsten Dachsbacher. A runtime cache for interactive procedural modeling. *Computers and Graphics (Special Issue of Shape Modeling International)*, 36(3), 2012.

[RLL+06a] Nicolas Ray, Wan-Chiu Li, Bruno Lévy, Alla Sheffer, and Pierre Alliez. Periodic global parameterization. *ACM Trans. Graph.*, 25(4):1460–1485, 2006.

[RLL+06b] Nicolas Ray, Wan Chiu Li, Bruno Lévy, Alla Sheffer, and Pierre Alliez. Periodic global parameterization. *ACM Trans. Graph.*, 25:1460–1485, October 2006.

[RT98] Y. Rubner and C. Tomasi. Texture metrics. In *Systems, Man, and Cybernetics, 1998. 1998 IEEE International Conference on*, volume 5, pages 4601 –4607 vol.5, oct 1998.

[Sag94] Hans Sagan. *Space-Filling Curves*. Springer, 1 edition, September 1994.

[Sam90] Hanan Samet. *The design and analysis of spatial data structures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[SBSCO06] Andrei Sharf, Marina Blumenkrants, Ariel Shamir, and Daniel Cohen-Or. Snap-paste: an interactive technique for easy mesh composition. *Vis. Comput.*, 22:835–844, September 2006.

[Sch96] Jean E. Schweitzer. Analysis and application of subdivision surfaces, 1996.

[Sch97] R.R. Schaller. Moore's law: past, present and future. *Spectrum, IEEE*, 34(6):52 –59, jun 1997.

[SDZ+11] Christopher P. Stone, Earl P. N. Duque, Yao Zhang, David Car, John D. Owens, and Roger L. Davis. Gpgpu parallel algorithms for structured-grid cfd codes. In *Proceedings of the 20th AIAA Computational Fluid Dynamics Conference*, number 2011-3221, June 2011.

[Sei91] Raimund Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Comput. Geom. Theory Appl*, 1:51–64, 1991.

[SHG09] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore gpus. *Parallel and Distributed Processing Symposium, International*, 0:1–10, 2009.

[SLMB05] Alla Sheffer, Bruno Lévy, Maxim Mogilnitsky, and Alexander Bogomyakov. Abf++: fast and robust angle based flattening. *ACM Trans. Graph.*, 24(2):311–330, 2005.

[SSGH01] Pedro V. Sander, John Snyder, Steven J. Gortler, and Hugues Hoppe. Texture mapping progressive meshes. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, pages 409–416, New York, NY, USA, 2001. ACM.

[SW08] Daniel Scherzer and Michael Wimmer. Frame sequential interpolation for discrete level-of-detail rendering, June 2008.

[SWG+03] P. V. Sander, Z. J. Wood, S. J. Gortler, J. Snyder, and H. Hoppe. Multi-chart geometry images. In *SGP '03: Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pages 146–155. Eurographics Association, 2003.

[SY97] R.M. Schoen and S.T. Yau. *Lectures on harmonic maps*. Conference proceedings and lecture notes in geometry and topology. International Press, 1997.

[SZL92] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. Decimation of triangle meshes. *SIGGRAPH Comput. Graph.*, 26(2):65–70, July 1992.

[SZS+08] Xin Sun, Kun Zhou, Eric Stollnitz, Jiaoying Shi, and Baining Guo. Interactive relighting of dynamic refractive objects. *ACM Trans. Graph.*, 27:35:1–35:9, August 2008.

[Tat08] Natalya Tatarchuk. Advanced topics in GPU tessellation. Gamefest'08, 2008.

[TCS03] Marco Tarini, Paolo Cignoni, and Roberto Scopigno. Visibility based methods and assessment for detail-recovery. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, VIS '03, pages 60–, Washington, DC, USA, 2003. IEEE Computer Society.

[THCM04] Marco Tarini, Kai Hormann, Paolo Cignoni, and Claudio Montani. Polycube-maps. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 853–860, New York, NY, USA, 2004. ACM.

[TPP+11] Marco Tarini, Enrico Puppo, Daniele Panozzo, Nico Pietroni, and Paolo Cignoni. Simple quad domains for field aligned mesh parametrization. *ACM Transactions on Graphics, Proceedings of SIGGRAPH Asia 2011*, 30(6), 2011.

[TSS+11] Kenshi Takayama, Ryan Schmidt, Karan Singh, Takeo Igarashi, Tamy Boubekeur, and Olga Sorkine. Geobrush: Interactive mesh geometry cloning. *Computer Graphics Forum (proceedings of EUROGRAPHICS)*, 30(2):613–622, 2011.

[Wei85] Kevin Weiler. Edge-based data structures for solid modeling in curved-surface environments. *IEEE Comput. Graph. Appl.*, 5:21–40, January 1985.

[WGTtY] Yalin Wang, Xianfeng Gu, Paul M. Thompson, and Shing tung Yau. 3d harmonic mapping and tetrahedral meshing of brain imaging data. In Proc. Medical Imaging Computing and Computer Assisted Intervention (MICCAI), St. Malo, France, Sept. 26-30.

[WHL+07] Hongyu Wang, Ying He, Xin Li, Xianfeng Gu, and Hong Qin. Polycube splines. In *Proceedings of ACM symposium on Solid and Physical Modeling*, pages 241–251, 2007.

[Wik04] Wikipedia. Scratchpad memory, 2004.

[Wil83] Lance Williams. Pyramidal parametrics. *SIGGRAPH Comput. Graph.*, 17(3):1–11, July 1983.

[Wil11] Andrew Willmott. Rapid simplification of multi-attribute meshes. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, pages 151–158, New York, NY, USA, 2011. ACM.

[WJH+08] Hongyu Wang, Miao Jin, Ying He, Xianfeng Gu, and Hong Qin. User-controllable polycube map for manifold spline construction. In *Proceedings of ACM symposium on Solid and Physical Modeling*, pages 397–404, 2008.

[WLL+11] Kexiang Wang, Xin Li, Bo Li, Huanhuan Xu, and Hong Qin. Restricted trivariate polycube splines for volumetric data modeling. *TVCG*, page In Print, 2011.

[WPSAM10] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying GPU microarchitecture through microbenchmarking. pages 235–246, March 2010.

[WYZ+11] Shenghua Wan, Zhao Yin, Kang Zhang, Hongchao Zhang, and Xin Li. A topology-preserving optimization algorithm for polycube mapping. *Computer & Graphics*, 35(3):639–649, 2011.

[XGH+11] Jiazhi Xia, Ismael Garcia, Ying He, Shi-Qing Xin, and Gustavo Patow. Editable polycube map for gpu-based subdivision surfaces. In *Symposium on Interactive 3D Graphics and Games*, I3D "11, pages 151–158, 2011.

[YLSL10] I-C. Yeh, C.-H. Lin, O. Sorkine, and T.-Y. Lee. Template-based 3d model fitting using dual-domain relaxation. *IEEE Trans. Vis. and Comp. Graph.*, accepted, 2010.

[YZX+04a] Yizhou Yu, Kun Zhou, Dong Xu, Xiaohan Shi, Hujun Bao, Baining Guo, and Heung-Yeung Shum. Mesh editing with poisson-based gradient field manipulation. In *SIGGRAPH '04*, pages 644–651, 2004.

[YZX+04b] Yizhou Yu, Kun Zhou, Dong Xu, Xiaohan Shi, Hujun Bao, Baining Guo, and Heung-Yeung Shum. Mesh editing with poisson-based gradient field manipulation. *ACM Trans. Graph.*, 23:644–651, August 2004.

[ZGHG08] Kun Zhou, Minmin Gong, Xin Huang, and Baining Guo. Highly parallel surface reconstruction, 2008.

[ZGHG11] Kun Zhou, Minmin Gong, Xin Huang, and Baining Guo. Data-parallel octrees for surface reconstruction. *IEEE Transactions on Visualization and Computer Graphics*, 17:669–681, May 2011.

[ZH99] Dongmei Zhang and Martial Hebert. Harmonic maps and their applications in surface matching. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR '99)*, volume 2, 1999.

[ZHWG08] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph.*, 27:126:1–126:11, December 2008.

[ZS00] Denis Zorin and Peter Schröeder. Subdivision for modeling and animation. *New York*, 98, 2000.

[ZSGS04] Kun Zhou, John Synder, Baining Guo, and Heung-Yeung Shum. Iso-charts: stretch-driven mesh parameterization using spectral analysis. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, SGP '04, pages 45–54, New York, NY, USA, 2004. ACM.