EUROGRAPHICS 2019
A. Giachetti and H. Rushmeier
(Guest Editors)

Volume 38 (2019), Number 2
STAR – State of The Art Report

Rethinking Texture Mapping

Cem Yuksel[1] Sylvain Lefebvre[2] Marco Tarini[3]

[1]University of Utah, USA [2]INRIA, Nancy, France [3]Università degli Studi di Milano, Italy

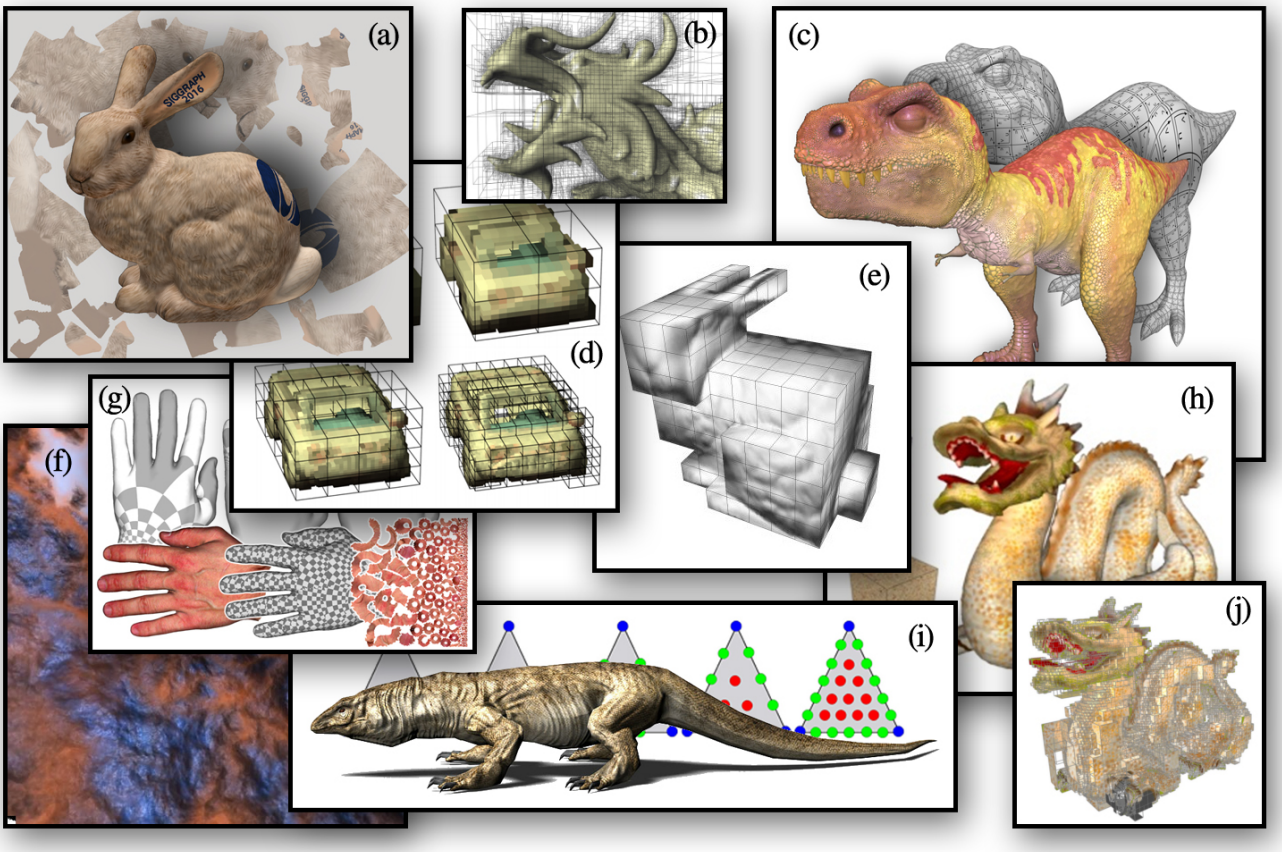


Figure 1: *Examples alternatives to texture mapping: (a) volume-encoded uv-maps [Tar16], (b) octree textures [LHN05], (c) Ptex [BL08] (©Walt Disney Animation Studios), (d) brickmaps [CB04], (e) polycube-maps [THCM04], (f) gigavoxels [CNLE09], (g) invisible seams [RNLL10], (h) perfect spatial hashing [LH06], (i) mesh colors [YKH10], and (j) tiletrees [LD07].*

Abstract

The intrinsic problems of texture mapping, regarding its difficulties in content creation and the visual artifacts it causes in rendering, are well-known, but often considered unavoidable. In this state of the art report, we discuss various radically different ways to rethink texture mapping that have been proposed over the decades, each offering different advantages and trade-offs. We provide a brief description of each alternative texturing method along with an evaluation of its strengths and weaknesses in terms of applicability, usability, filtering quality, performance, and potential implementation related challenges.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Image manipulation—Texturing

1. Introduction

In computer graphics, texture mapping is the fundamental means by which high-frequency signals, such as diffuse colors, normals, displacement, and other shading parameters, are defined over 3D surfaces. The principle is to store the signal in 2D high-resolution texture images, and then define a mapping from the 3D surface to the 2D image, by assigning a *uv* coordinate to each mesh vertex. This approach is ubiquitously adopted by virtually all computer graphics applications and implemented on all available graphics hardware, from high-end to smartphone GPUs.

However, traditional texture mapping has a number of fundamental issues. Creating uv-maps is time consuming and involves extensive manual effort in practice. As a consequence, texture mapping continues to occupy a substantial portion of artist time, which dominates the cost of AAA video game production. The result is tailored for a specific mesh and connectivity and it does not necessarily work through different resolution versions of the mesh, used for level of detail (LoD). Distortions and seams introduced by the mapping complicate texture authoring, filtering, and procedural synthesis. Seams cause visual artifacts with signal interpolation, and cracks on the surface with displacement mapping. Representing the mapping requires vertex duplications, which can complicate data structures and procedural texturing operations. Any change to the geometry or the connectivity implies updating the mapping and the texture image. Furthermore, texturing other surface representations than polygonal meshes (e.g. implicit surfaces, point clouds) requires a conversion to a mesh format.

Since the early days of texturing, there has been a constant research effort to alleviate the issues and/or bypass the limitations of traditional texture mapping (Figure 1). Unfortunately, the ubiquitous adoption of texture mapping implies that it is seldom questioned as the method of choice, and both authoring pipelines and rendering engines have been engineered around its intrinsic limitations, thereby making it harder for alternatives to be adopted. Yet, the industry recently started to recognize the advantages of alternative approaches to texture mapping.

This article surveys such alternative approaches, discussing their advantages and trade-offs regarding versatility, ease of authoring, storage cost, rendering quality and performance, and implementation difficulty. Our aim is to provide the knowledge needed for determining the best candidate for replacing texture mapping for any application, which we expect to be different based on the constraints of the application.

1.1. The Scope of this STAR

We begin by discussing the traditional texture mapping approach in detail (Section 2), explaining its strengths and its shortcomings. Then, we present alternative approaches to traditional texture mapping, grouped in four categories.

Perfecting the Traditional Texture Mapping (Sec. 3). The methods in this group address specific shortcomings of the traditional texture mapping approach by carefully-designed mapping operations. This group includes:

- *Invisible Seams* [RNLL10] (Sec. 3.1),
- *Seam Erasure* [LFJG17] (Sec. 3.1),
- *Seamless Toroidal/Cylindrical Textures* [Tar12] (Sec. 3.2),
- *Seamless Texture Atlases* [PCK04] (Sec. 3.3).

Connectivity-based Representations (Sec. 4). These methods rely on the inherent parameterization of each separate element of the 3D model, without explicitly defining a separate mapping of the surface. This group includes:

- *Ptex* [BL08] (Sec. 4.1),
- *Mesh Colors* [YKH10] (Sec. 4.2),
- *Mesh Color Textures* [Yuk16] (Sec. 4.3).

Sparse Volumetric Textures (Sec. 5). These methods store the texture data in sparse volumetric structures embedding the surface, rather than 2D images mapped onto the model surface. Thus, mapping is implicitly defined by the 3D positions on the surface. This group includes:

- *Adaptive Texture Maps* [KE02] (Sec. 5.1),
- *Octree Textures* [BD02, LHN05] (Sec. 5.2),
- *Brick Maps* [CB04] (Sec. 5.3),
- *Perfect Spatial Hashing* [LH06, GLHL11] (Sec. 5.4),

Volume-based Parameterizations (Sec. 6). These methods define the mapping using the 3D position of the surface, either directly or by using a volumetric intermediate representation. The texture data is stored as a 2D image, as in traditional texture mapping. This group includes:

- *TileTrees* [LD07] (Sec. 6.1),
- *PolyCube-Maps* [THCM04] (Sec. 6.2),
- *Volume-Encoded UV-Maps* [Tar16] (Sec. 6.3).

techniques above includes two items, labelled as [OLDER], which consists in the direct adoption of basic, well established mechanisms.

1.2. Evaluation Criteria

We evaluate traditional texture mapping and its alternatives based on various criteria that examine their properties in five groups: applicability, usability, filtering quality, performance, and implementation. We provide a separate summary table for the evaluation of each method. The tables are color-coded to highlight positive traits using green, negative/problematic traits using red, and mixed or neutral traits in yellow. While the summary tables and their color-coding aim to provide a general assessment, the importance of each trait can be application-dependent.

Applicability. The applicability criteria examine the types of 3D model representations that can be textured with each method. While all methods we discuss primarily target *polygonal meshes*, a few of them also support other surface representation, such as *point clouds*, or *implicit surfaces*. Some methods impose *shape/topology limits* on the models that can be textured. Most methods support *subdivisions* of the textured polygonal model, where the original edges are split, but otherwise maintained. Some methods even provide *tessellation independence*, so that, for example, the same texture can be used with different resolution representations of a model in an LoD pyramid.

Usability. The usability criteria examine the impact of the methods on the 3D production pipeline, mainly from the perspective of content creation. One important factor is whether a mapping must be manually constructed for the surface, or if, conversely, an *automated mapping* can be used. *Mapping customizibility* can be important for tailoring the mapping according to user needs. Using some methods, *model editing after painting* the texture data may require reproducing the texture data fully or partially. The *resolution readjustment* property presents whether local resolution changes on the texture data can be easily handled. Some methods support *texture repetition* that allows using the same texture data over multiple parts of a model, which can be used for exploiting symmetries or tiling, and thereby reduce the texture storage requirements. Another important criterion is the ability to store the texture data as a *2D image representation* in a way that would allow using existing 2D image editing and painting tools. While any texture data can be converted into a 2D image, this criterion examines whether the resulting 2D image is suitable for manual painting operations.

Filtering Quality. The filtering quality criteria examine the abilities of the method for producing effective texture filtering. We consider this for three forms of filtering operations: *magnification filtering*, which is necessary when the signal is up-sampled on screen pixels and typically consists of on-the-fly bilinear interpolation; *minification filtering*, which occurs when the signal is down-sampled on screen pixel and is typically handled using precomputed textures for different filter sizes using mipmapping; and *anisotropic filtering*, which allows skewed filter shapes along a direction and is particularly important when the textured surface is viewed at an angle.

Performance. The performance criteria indicate the computation time and memory storage requirements of each method. *Vertex data duplication* is a typical performance overhead that leads to storing and processing multiple copies of some mesh vertices. *Storage overhead* reports the extra memory cost in addition to the texture data storage. If sampling the texture data requires indirect accesses, they are reported as a part of the *access overhead* property. *Computational overhead* reports the additional computations required by the method for sampling the texture data. The *hardware filtering* property presents if the method can utilize the texture filtering hardware available on GPUs. This is particularly important for real-time rendering applications, since hardware texture filtering can be an order of magnitude faster than software implementations.

Implementation. Replacing traditional texture mapping with an alternative method may require implementing custom tools and algorithms. We report separate estimates for the *asset production* phase, including algorithms and tools to create and edit textures and required mappings, and the *rendering* phase, considering the custom algorithms required for texture filtering operations.

Table 1: *Traditional 2D Texture Mapping*

Applicability	
Polygonal Meshes	Yes
Point Clouds	Single color per point
Implicit Surfaces	With implicit mapping
Shape/Topology Limits	None
Subdivisions	Yes
Tessellation Independence	If seams are preserved
Usability	
Automated Mapping	Often requires manual intervention
Mapping Customizability	Yes
Model Editing after Painting	Problematic
Resolution Readjustment	Problematic
Texture Repetition	Yes
2D Image Representation	Yes
Filtering Quality	
Magnification Filtering	Yes, with seam artifacts
Minification Filtering	Yes, with seam artifacts
Anisotropic Filtering	Yes, with seam artifacts
Performance	
Vertex Data Duplication	Yes
Storage Overhead	2D mapping and wasted texture space
Access Overhead	None
Computation Overhead	None
Hardware Filtering	Yes
Implementation	
Asset Production	Numerous existing tools
Rendering	Full GPU support

2. Traditional 2D Texture Mapping

Before we survey alternatives, we examine the traditional approach to texture mapping, which is the current status quo of computer graphics applications. Graphics APIs, GPU hardware, 3D models production pipelines, game engines, standard file formats for 3D models, and almost all modeling software are all designed around this approach. This section serves as the background for our discussions of alternative methods, and the "baseline" against which to compare them.

Though the texture data can be stored as 1D, 2D, or 3D *texel* arrays, in this survey we exclusively concentrate on 2D textures that are used for defining colors on 3D surfaces. The texture values on the surface are determined using a *mapping* from the surface to the 2D texture image. This mapping is defined by assigning *uv* coordinates to mesh vertices that indicate their image-space locations. In effect, mapping planarizes the 3D model and in almost all typical cases it must include *seams*. Seams are defined as the set of edges that the two faces they connect map to different locations on the 2D image. Thus, the seam edges map to two separate places.

Texture mapping can be used with any polygonal mesh. Point clouds can be represented by carefully assigning *uv* coordinates to each point, such that each *uv* pair corresponds to a texel of the texture image. Implicit surface representations require implicit mapping functions, which can be challenging to define, depending on the complexity of the implicit model. Texture mapping does not impose any shape or topology limits on the 3D mesh and it can easily handle arbitrary subdivisions of a mesh. Traditional texture mapping does not offer tessellation independence, so a complete remeshing of the surface can be difficult to handle. More specifically, unless remeshing preserves the edges along seams, simply

defining a new mapping would not be sufficient and the texture image may need to be regenerated for the new mesh topology and the new mapping. This limitation is particularly important for generating different resolution versions of a mesh for LoD optimizations.

Some of the most important drawbacks of texture mapping are related to its usability. First and foremost, texture mapping requires defining a mapping and texture data on a given 3D model cannot be generated before defining the mapping. While it is possible to define a mapping automatically, in practice mapping cannot be fully automated. This is partially because the process of defining a desirable mapping must also involve the knowledge of how the texture data would be generated and used. Therefore, even when an automated method is used, manual customization of mapping is often needed, which can be a tedious process. Once the mapping is defined and the texture data is generated, editing the 3D model can be problematic. This is because geometrical or topological modifications may require completely or partially altering the mapping, which typically needs to be done manually. This may in turn invalidate the existing texture data stored in the 2D image. Local modifications to the 3D model do not always have local effect on the mapping. Furthermore, locally changing the texture resolution on a part of the model requires altering the mapping and the related texture data, which makes this process problematic as well. Changing the entire texture resolution is much easier, but it can lead to substantial inefficiencies, such as unnecessarily increasing the resolution unnecessarily on other parts of the model. Similarly, editing or processing the texture image itself in 2D is made problematic by presence of seams and mapping distortion, thus requires either careful editing or to drop traditional 2D painters tool in favour of specialized 3D painter.

Traditional texture mapping also has some advantages in terms of usability, which are not supported by all alternative methods. In particular texture repetitions on a surface can be handled easily and efficiently. Though most applications require an injective mapping, traditional texture mapping supports non-injective mapping as well, so the same texture data can be repeated on multiple parts of a model. Furthermore, when the mapping is designed accordingly, the resulting 2D image that stores the texture data can be a good representation of the 3D model surface. Thus, existing 2D image editing tools can be used for texture authoring.

Another set of important drawbacks of traditional texture mapping is its limitations on rendering-time filtering quality. Seams in mapping lead to discontinuities in texture filtering. These discontinuities lead to inconsistencies in texture filtering on either side of the seams, which can reveal the seams in rendered images. When used with displacement mapping, seams lead to cracks on the surface, hindering the widespread use of displacement mapping in practice. Seams also cause inconsistencies in anisotropic filtering. Therefore, seams must be carefully placed while defining the *uv* mapping, so that the inconsistencies in filtering operations that reveal the seams would have a minimal impact on the final rendered image quality.

Since traditional texture mapping has been the standard for handling surface textures, graphics hardware and APIs are designed to optimize the texture mapping performance. Nonetheless, texture mapping has some performance-related drawbacks as well. Since vertices along seams are mapped to multiple locations on the 2D image, vertex data duplication is often unavoidable. Also, mapping often involves wasted space on the texture image, either as unused texels (due to imperfect packing) or unnecessarily high resolution texture data for some parts of the model (due to distortions in mapping). Furthermore, the mapping information must be stored and passed through the rendering pipeline, such as a pair of *uv* coordinate values per vertex.

There are numerous tools for asset production with traditional texture mapping and the existing rendering pipelines are designed to support it. Therefore, we assert that no additional implementation is required for using texture mapping, when existing tools and algorithms can be utilized.

3. Perfecting Traditional Texture Mapping

Several methods have been proposed to use the traditional texture mapping approach in specific, restricted ways so that a few of its shortcomings are avoided or mitigated with at most minimal modifications.

A range of methods strives to automatize the effort required to workaround the limitations of texture mapping. For example, image filtering algorithms have been recently highly specialized [PKCH18] to work in presence of texture seams and texture distortions. In a similar spirit, algorithms have been proposed to automatize the adaptations of the *uv* map that is necessary after 3D shape modifications [DGM18] (for limited deformations). As another example, a specific solutions addressing the poor interoperability of LoD-pyramids and atlas-based textures exists [SSGH01] (for a specific type of LoD-pyramids: progressive meshes). These methods can be useful in specific situations, but their adoption impacts significantly the asset-production pipelines.

3.1. Invisible Seams

One of the problems of traditional texture mapping is the rendering artifacts due to discontinuities along seams, caused by inconsistent bilinear filtering performed on either side of the seams (see Fig. 2a). The *invisible seams* [RNLL10] approach introduces some constraints on the way that seams are mapped onto the 2D texture image to ensure that bilinear filtering operations on either

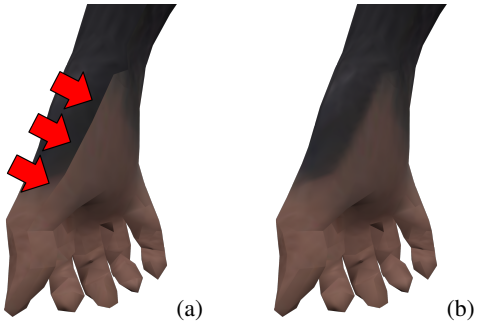


Figure 2: ***Invisible seams:*** *(a) a visible seam with traditional texture mapping is (b) erased by altering the uv mapping [LFJG17].*

side of the seams would produce consistent results, thereby effectively hiding the seams. Other names have also been used for referring to similar approaches, such as "seamless cuts" or "erased seam" [LFJG17]. A parametrization where all seams are invisible is sometimes called "globally seamless" or "griddable."

A simple example mapping rule for making seams invisible is placing both sides of the seams axis-aligned in *uv* space (either horizontally or vertically) with matching lengths, and assigning texels values that are exactly replicated along the seam. While not being the main motivation (or even being mentioned), this is the case for several texture mapping approaches, such as cylindrical/toroidal mappings [Tar12] and PolyCubeMaps [THCM04].

This set of constraints can be relaxed in several ways. The invisible seams method [RNLL10] explores two (independent) generalizations. First, integer resolution jumps across seams are allowed by only requiring the parametric length (in the 2D texture image space) of one side of the seam to be n times the other, where $n \geq 1$ and $n \in \mathbb{Z}$. This preserves seam invisibility, as long as the texel values on the longer side are linear interpolations of the values on the shorter side. This observation is exploited for generating *uv* layouts with motorcycle graphs [SPGT18]. Second, oblique placement of seams in the 2D texture space is allowed. Here, the two sides of the seams are constrained to be matching in parametric space up to an integer translation and rotations by an integer multiple of $\pi/2$, resulting in texel grids which are aligned across seams. The invisible seams method [RNLL10] borrows this setup straight from the contexts where parametrizations are constructed for remeshing rather than for *uv* maps [BZK09,RLL*06,JTPSH15], thereby tapping into a large number of existing automatic potential parametrization construction approaches developed for this purpose.

Making seams invisible solves the magnification filtering problems of traditional texture mapping. This improvement comes at a cost of minor usability drawbacks, by making the process of defining a *uv* mapping more complicated. Minification pre-filtering (including anisotropic) can be dealt with by making the seam invisible at lower-res MIP-map levels (resulting in an increased cost).

More recently, the concept of invisible seams have been further generalized [LFJG17]. Here, no constraint is explicitly imposed *a priori* on *uv* assignment. Instead, the space of possible texel assignments (given an existing *uv* map and texture sheet) are explored around seams for solutions in which seams become invisible, such that bilinearly interpolated texture values along the two sides of the seams match perfectly (Figure 2). Among the solutions that satisfy this constraint, one is sought that minimizes the discrepancy from the initial configuration. An exact (i.e. analytic) fulfillment of the constraint would leave only a very small space of solutions (not necessarily including usable ones), but a numerical solution can be found, implicitly tolerating small approximation errors [LFJG17]. The advantage of this is that a much larger space of (almost) invisible seams is made available. A simpler strategy is to just reduce the signal discontinuities at the seams [Syl15], minimizing it in the least-square sense, by tweaking surrounding texel values. A drawback is that both these approaches (as opposed to the original invisible seams method [RNLL10]) also imposes limitations on the texture data (as sampled in the texel values). Though not directly useful for texture mapping purposes, the concept of seamless *uv* assignment can be generalized to generic 2D affine transformations [APL15].

Table 2: *Invisible Seams*

Applicability	
Polygonal Meshes	Yes
Point Clouds	No
Implicit Surfaces	No
Shape/Topology Limits	Depends on parameterization
Subdivisions	Yes
Tessellation Independence	If seams are preserved
Usability	
Automated Mapping	Limited
Mapping Customizability	Constraints need to be preserved
Model Editing after Painting	Problematic
Resolution Readjustment	Problematic
Texture Repetition	Yes
2D Image Representation	Poor
Filtering Quality	
Magnification Filtering	Yes
Minification Filtering	Yes (stricter constraints)
Anisotropic Filtering	Yes (stricter constraints)
Performance	
Vertex Data Duplication	Yes
Storage Overhead	2D mapping and wasted texture space
Access Overhead	None
Computation Overhead	None
Hardware Filtering	Yes
Implementation	
Asset Production	Automated mapping
Rendering	Standard pipeline

These prior schemes attempt to remove the seams as seen through a standard texture lookup. Other approaches modify the interpolation to hide seams during rendering [GP09]. Here, the local geometry across chart boundaries is stored into the texels as small triangle lists. This allows to interpolate across charts without visible discontinuities. The main drawback is an increase in shader complexity, memory accesses, and storage; however the authoring pipeline and input *uv* maps remain unaffected.

As a minor technical note, making seams invisible (with any method) may require using texture resolutions that are powers of 2. Our experiments on various GPUs revealed that seams can still be visible when using arbitrary texture resolutions (probably due to numerical errors introduced by the GPU).

3.2. Seamless Toroidal/Cylindrical Textures

In most cases, the traditional texture mapping approach only allows seams at mesh edges, and requires duplication of vertices along such edges. This first restriction can be dropped in the special case of *cylindrical* and *toroidal* maps [Tar12], which are useful for texturing models with approximately cylindrical or toroidal shapes (Figure 3). In cylindrical maps, a rectangular texture is wrapped around the side area of a cylinder, and a single seam line runs along

Table 3: *Seamless Toroidal/Cylindrical Textures*

Applicability	
Polygonal Meshes	Yes
Point Clouds	No
Implicit Surfaces	No
Shape/Topology Limits	Toroidal/cylindrical only
Subdivisions	Yes
Tessellation Independence	Yes
Usability	
Automated Mapping	Usually easy for applicable shapes
Mapping Customizability	Yes
Model Editing after Painting	No
Resolution Readjustment	Problematic
Texture Repetition	Yes
2D Image Representation	Yes
Filtering Quality	
Magnification Filtering	Yes
Minification Filtering	Yes
Anisotropic Filtering	Yes
Performance	
Vertex Data Duplication	No
Storage Overhead	2D mapping only
Access Overhead	None
Computation Overhead	Minor
Hardware Filtering	Yes
Implementation	
Asset Production	Simple automated mapping
Rendering	Simple UV manipulation

the cylinder. In a toroidal map, there are two orthogonal, closed seams loop, meeting in a point. The seams are made invisible by simply wrapping the texture.

This approach also permits avoiding vertex data duplication along seams. This is achieved by a simple and resource undemanding render-time technique that produces multiple interpolations of *uv* coordinates specified on the vertices: one of the directly interpolates the *uv* values, others wrap around the edges of the texture. This guarantees that that at least one interpolation is the "correct" one. At a fragment level, the binary choice is driven by a comparison of the screen-space derivatives of the alternative interpolations.

Figure 3: ***Toroidal/Cylindrical Textures:*** *Traditional texture mapping (shown on the left side) reveals the seams. Toroidal/cylindrical mapping [Tar12] (shown on the right) eliminates the seams. Circular and square insets show close-ups.*

This method only targets to two relatively uncommon classes of mappings: topologically toroidal or cylindrical maps. These classes already present natural built-in advantages: seams need not be represented by mesh edges, and seams are invisible. The adoption of this algorithm further removes the need for vertex duplications. This can be particularly helpful to simplify procedural generation approaches where vertices (and their *uv* coordinates) are generated on the fly [MCT16].

3.3. Seamless Texture Atlases

Filtering artifacts of traditional texture mapping along seams can be easily eliminated, if the mapping produces quadrilateral pieces on the 2D texture image. The seamless texture atlases method [PCK04] begins with splitting the 3D model into pieces that can be flattened onto square-shaped *charts* that are mapped to different places on the 2D texture image (Figure 4). The seams (i.e. the edges of the square charts) are placed axis-aligned and exactly between texels. The seams are hidden in filtering by adding borders around charts. The texture data along these border texels are assigned from the texture data of the neighboring charts (on the 3D model). Thus, even when neighboring charts are placed separately on the 2D texture image, bilinear filtering (including the chart borders) produces the same values on either side of the seams, thereby hiding them.

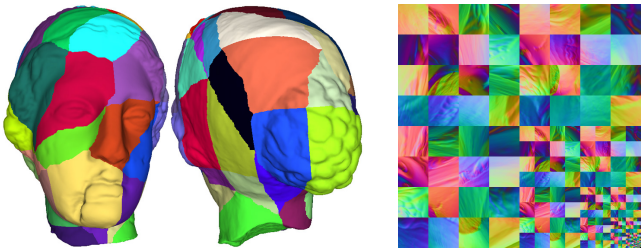


Figure 4: ***Seamless Texture Atlases:*** *An example model split into quadrilateral charts and the corresponding texture in the form of a flat seamless atlas [PCK04].*

Minification filtering using mipmaps is supported by adding a texel-wide border (i.e. padding) around each mipmap level of each chart. While conceptually simple, mipmapping becomes slightly complicated, because the size (i.e. thickness) of the borders must be maintained at each mipmap level. Therefore, the *uv* coordinates used for the highest resolution mipmap level cannot be directly used for the other mipmap levels. A simple solution is provided by storing normalized *uv* coordinates per chart, such that each *uv* pair within $[0,1]^2$ corresponds to a position within a chart. The locations of each chart on the 2D texture image are stored in a lookup table. The *uv* coordinates are scaled and shifted differently for each mipmap level to account for the border texels.

Table 4: *Seamless Texture Atlases*

Applicability	
Polygonal Meshes	Yes
Point Clouds	No
Implicit Surfaces	No
Shape/Topology Limits	None
Subdivisions	Yes
Tessellation Independence	No
Usability	
Automated Mapping	Limited
Mapping Customizability	Problematic
Model Editing after Painting	Problematic
Resolution Readjustment	Per patch only
Texture Repetition	Per patch only
2D Image Representation	Poor
Filtering Quality	
Magnification Filtering	Yes
Minification Filtering	Yes, with custom mipmaps
Anisotropic Filtering	Yes, with seams artifacts
Performance	
Vertex Data Duplication	Yes
Storage Overhead	2D mapping & indirection table
Access Overhead	Single indirection
Computation Overhead	Indirection
Hardware Filtering	Yes
Implementation	
Asset Production	Quadrangulation, automated mapping, 3D painting
Rendering	Simple UV manipulation & indirection

The seamless texture atlases method allows both storing all mipmap levels together within a single 2D image or storing them as separate images. Hardware-accelerated bilinear filtering can be utilized, but the mipmap level must be determined in software and the corresponding *uv* coordinates for the mipmap level must be computed before they are used for texture lookup operations. This process also includes accessing a lookup table that stores the location of the chart.

4. Connectivity-based Representations

Connectivity-based representations use the inherent parameterization of the model, instead of defining a separate parameterization for mapping. The topology of the mesh model is used directly for defining the texture data on the model primitives. Therefore, these approaches completely eliminate the need for specifying a mapping. As a result they avoid the difficulties and limitations of the mapping process. Since the texture data is directly associated with the model primitives, they permit operations like model editing after texture painting and local resolution readjustment. All of these qualities make these approaches especially powerful for authoring. However, since they define the texture data directly on a 3D model surface, they require 3D authoring tools, such as 3D painting.

4.1. Ptex

Ptex [BL08] is probably the most commonly known alternative to texture mapping, used and promoted by Disney Animation and Pixar studios. The concept of Ptex can be summarized as per-face-texture, meaning each face of a model is practically assigned a separate texture of its own (Figure 5). This treatment eliminates the need for specifying a mapping, since each quad-shaped face can be trivially mapped onto a rectangular texture. Triangular faces are handled using textures with triangle-shaped texels.

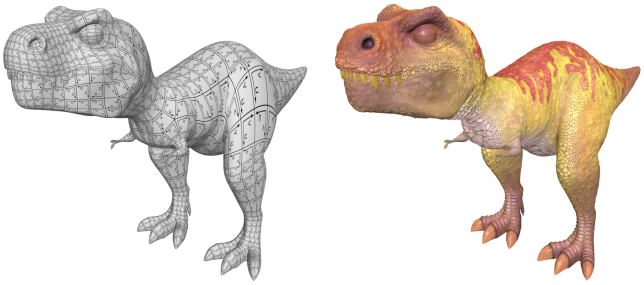


Figure 5: ***Ptex:*** *Per-face textures shown on a model and the resulting texture [BL08] (©Walt Disney Animation Studios).*

Ptex can easily handle model editing operations that can modify the model geometry and topology. Since each face effectively gets its own texture, it is not affected by geometrical deformations of the model. Topological operations that remove faces or add new faces can be trivially handled by adding and removing corresponding face textures. The texture data for the rest of the model remains unchanged. Subdivision operations that split faces into smaller ones can be efficiently handled by storing textures per groups of subdivided faces that are generated from the same face of the lower-resolution model version. Therefore, in a sense, Ptex can be defined on a low-resolution model and the texture data can be directly used by any subdivision of its faces. However, Ptex does not provide tessellation independence, so a different remeshing of the model cannot directly use the original Ptex data.

Having a separate texture per face makes it easy to handle texture filtering operations within a face. However, near the edges of the faces, texture filtering requires accessing the colors on the neighboring faces. This is facilitated by introducing an adjacency list data structure. Ptex stores four face indices per face, indicating the four neighbors of each quad face. In addition, four edge indices are stored per face to represent which edge of each neighboring face is the shared edge. This adjacency data is used during texture filtering for accessing the texture data on neighboring faces.

One important advantage of Ptex is that the texture resolution of each face can be adjusted independently. This permits having higher resolution only where texture data requires. It also allows setting the texture resolution of a face based on its size. However, texture filtering near edges becomes difficult when the neighboring face has a different resolution. More importantly, mismatched resolutions between neighboring faces can lead to inconsistencies in texture filtering on either side of the shared edges, thereby revealing edges (similar to the seam artifacts of traditional texture mapping, but along every edge where face resolutions are different).

Table 5: *Ptex*

Applicability	
Polygonal Meshes	Quad-dominant semi-regular meshes
Point Clouds	No
Implicit Surfaces	No
Shape/Topology Limits	Problems with extraordinary vertices
Subdivisions	Yes
Tessellation Independence	No
Usability	
Automated Mapping	Yes
Mapping Customizability	No
Model Editing after Painting	Yes
Resolution Readjustment	Yes
Texture Repetition	Only identical topology
2D Image Representation	Poor
Filtering Quality	
Magnification Filtering	Yes (seams at singularities)
Minification Filtering	Yes, up to single color per quad
Anisotropic Filtering	Yes, with custom filtering
Performance	
Vertex Data Duplication	N/A
Storage Overhead	Adjacency list & face resolutions
Access Overhead	Indirections
Computation Overhead	Custom filtering
Hardware Filtering	No
Implementation	
Asset Production	3D painting
Rendering	Custom filtering

When filtering near vertices of a face, the filter kernel is split into four pieces: the first piece corresponds to the face that is being processed, the next two correspond to the two neighbors around the vertex, and the last one corresponds to the face on the opposing side of the face that is being processed. Accessing this last face involves finding the neighbor of a neighboring face using the adjacency data. This operation works well for valance-4 vertices that are shared by four faces, but extraordinary vertecies that are shared by fewer or more than four faces (i.e. singularities) lead to filtering discontinuities. Yet, such discontinuities are of little practical concern when using models with relatively few extraordinary vertices.

Both magnification and minification filtering are supported. Mipmaps can be used up to a single color value per face. Anisotropic filtering can be implemented as well. Yet, none of these filtering operations can directly utilize hardware-accelerated filtering on current GPUs, so they must be implemented in software. The software implementation of texture filtering with Ptex can be relatively complicated. The storage overhead includes the adjacency list per face and accessing texture data involves one or two indirections using the adjacency list. Nonetheless, high-quality filtering can be achieved using custom filtering operations implemented in software.

An alternative implementation of Ptex can directly use hardware texture filtering on the GPU with array textures [Tot13]. Using texture borders, where the alpha values are set to zero, the need for crossing edges for texture filtering is avoided and the adjacency list is not used. This, however, prevents having varying face texture resolutions and introduces discontinuities along edges.

The main advantage of Ptex is that it substantially simplifies the texture authoring process, as compared to traditional texture mapping. Since no mapping is involved, a given model can be painted directly using a 3D painting interface. Quick GPU-based updates of the Ptex structure are also possible [SKNS15].

4.2. Mesh Colors

Mesh colors [YKH10] are closely related to Ptex. They were introduced around the same time and similarly have been used in production [Lam15]. Mesh colors can be considered as an extension of the concept of vertex colors. Indeed, the lowest-resolution mesh colors are vertex colors, which store a single texture value per vertex. In fact, a high-resolution mesh storing millions of vertices can be textured using vertex colors alone, though vertex colors do not provide a good mechanism for minification or anisotropic filtering. Higher-resolution mesh colors also include regularly-spaced colors (or any type of texture data) along the edges and on the faces of a model as well. These additional colors for higher-resolutions are called *edge colors* and *face colors*. The 3D positions of colors correspond to vertices on a regular tessellation of the faces. In terms of color placement topologies, Ptex and mesh colors can be considered dual pairs, since the color locations of mesh colors are in-between the color locations of Ptex with comparable resolution (and vice versa), as shown in Figure 6.

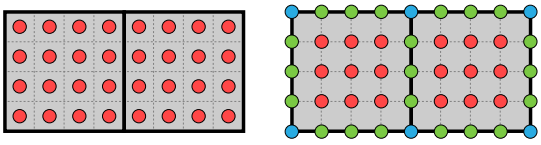


Figure 6: *Color locations of Ptex and mesh colors on two quad faces of a 3D model.*

Mesh colors (Figure 7) provide the same authoring advantages of Ptex. Using the inherent parameterization of the model, no mapping is needed for mesh colors. Model geometry can be freely modified and altering model topology only impacts the texture data on the modified faces. The texture resolution of each face can be specified independently. The resolution of an edge is automatically defined as the highest resolution of the two faces sharing the edge.

For filtering, however, mesh colors offer advantages over Ptex. Using colors on vertices and along edges completely eliminates filtering discontinuities, including extraordinary vertices. The texture value at any point on a face can be computed using the face, edge, and vertex colors of the face, without accessing the colors of neighboring faces. Yet, since edge and vertex colors are shared between neighboring faces, they are stored separately, which introduces some complexity in filtering operations. This process can be simplified by duplicating the edge and vertex colors at a cost of additional storage. Both magnification and minification filtering are

Table 6: *Mesh Colors*

Applicability	
Polygonal Meshes	Quads & triangles
Point Clouds	Single color per point
Implicit Surfaces	No
Shape/Topology Limits	None
Subdivisions	Yes
Tessellation Independence	No
Usability	
Automated Mapping	Yes
Mapping Customizability	No
Model Editing after Painting	Yes
Resolution Readjustment	Yes
Texture Repetition	Only identical topology
2D Image Representation	No
Filtering Quality	
Magnification Filtering	Yes
Minification Filtering	Yes, up to vertex colors
Anisotropic Filtering	Yes, with custom filtering
Performance	
Vertex Data Duplication	N/A
Storage Overhead	Edge indices and face resolutions
Access Overhead	None
Computation Overhead	Custom filtering
Hardware Filtering	No
Implementation	
Asset Production	3D painting
Rendering	Custom filtering

Table 7: *Mesh Color Textures*

Applicability	
Polygonal Meshes	Quads & triangles
Point Clouds	Single color per point
Implicit Surfaces	No
Shape/Topology Limits	None
Subdivisions	Yes
Tessellation Independence	No
Usability	
Automated Mapping	Yes
Mapping Customizability	No
Model Editing after Painting	Yes
Resolution Readjustment	Yes
Texture Repetition	Only identical topology
2D Image Representation	Poor
Filtering Quality	
Magnification Filtering	Yes
Minification Filtering	Yes
Anisotropic Filtering	Yes, with seam artifacts
Performance	
Vertex Data Duplication	Yes
Storage Overhead	4D mapping & wasted space
Access Overhead	None
Computation Overhead	UV calculation
Hardware Filtering	Yes
Implementation	
Asset Production	3D painting
Rendering	Simple UV calculation

supported and mipmaps can be used up to vertex colors (i.e. the resolution of mipmaps cannot be below vertex colors).

Defining mesh colors on a mesh requires assigning edge indices so that colors along edges can be uniquely specified. These indices are used during texture filtering. Separately accessing face, edge, and vertex colors requires custom filtering operations, so mesh colors cannot be directly used with hardware texture filtering on current GPUs. Filtering operations must be implemented in software.

A variant of this method uses the GPU hardware tessellation pattern [SPM*13], instead of the linear tessellation pattern of mesh colors. This prevents the need for performing any filtering operations when used for displacement mapping with GPU tessellation.

Figure 7: ***Mesh Colors:*** *An example model textured using mesh colors [YKH10]. The colors of this particular example are automatically converted from a 2D texture.*

4.3. Mesh Color Textures

Mesh color textures [Yuk16] provide a GPU-friendly version of mesh colors. The goal of mesh color textures is to convert mesh colors into a form that can be used with hardware texture filtering on current GPUs.

Since both Ptex and mesh colors require custom texture filtering operations, they must be implemented in software. However, hardware-accelerated texture filtering can be an order of magnitude faster than software implementation. Therefore, custom texture filtering cannot compete with the performance of traditional 2D textures on the GPU. This is particularly important for real-time rendering applications.

Mesh color textures effectively convert mesh colors into a 2D textures by carefully copying the mesh color values (Figure 8). This process duplicates edge and vertex colors and it also uses extra padding for triangles, so it leads to a minor storage overhead. Mipmapping is supported by storing each mipmap level on a separate texture. The texture coordinates for a mipmap level are easily computed from a custom 4D texture coordinate representation. As a result, mesh color textures can achieve the performance of traditional texture mapping on the GPU.

Therefore, mesh color textures offer the texture authoring benefits of mesh colors to applications that require real-time rendering performance. Furthermore, using mesh color textures, as op-

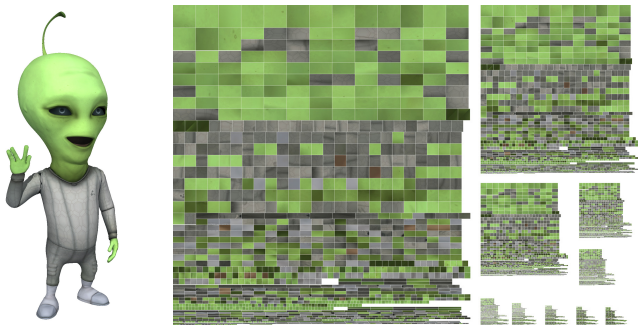


Figure 8: ***Mesh Color Textures:*** *An example model and its mesh color texture, showing all mipmap levels [Yuk16].*

posed to traditional texture mapping, eliminates the seam artifacts in minification filtering with mipmaps. Mesh color textures support anisotropic filtering as well, but on current GPU hardware anisotropic filtering leads to seam artifacts, similar to traditional texture mapping.

5. Sparse Volumetric Representations

A straightforward approach to encode texture data along surface (colors, normals, etc.) is to store them in a voxel grid covering the object. Given a surface point the texture data is directly retrieved from the grid, using 3D coordinates, potentially interpolating in-between grid nodes. This entirely by-passes the need for optimizing and storing a planar parameterization, and also permits texturing various surface representations, such as point clouds and implicit surfaces. A naïve implementation of this approach would require a large space, which would be cubic with the resolution of the sampling, making the approach impractical. Several countermeasures are adopted to avoid this problem, using sparse volumetric data structures, as illustrated in Figure 9.

Overall, a key advantage of these approaches is their conceptual simplicity and ease of implementation. By exploiting a more complex texture lookup, they entirely remove the need for a planar parameterization. Most of these structures can be built efficiently, even on-the-fly, thus allowing dynamic allocation of textures along evolving surfaces.

Using volume textures to store color information, however, presents a number of drawbacks. A first issue is that thin surfaces get the same color on both sides. This can be alleviated by storing normals alongside colors [BD02], at the expense of a further increase in storage and lookup cost. Another drawback relates to the cost of trilinear interpolation. If implemented in shaders, trilinear interpolation requires up to eight full lookups through the data structure. This cost is somewhat mitigated by hardware caches, but this is generally at least an order of magnitude slower than a direct texture lookup. This drawback is reduced when storing bricks (i.e. dense blocks of voxels) instead of individual voxels, in which case the hardware trilinear interpolation can be directly used when sampling within a brick. This can, however, significantly increase the memory requirements, especially when using larger bricks and requiring duplicate samples along brick boundaries. Anisotropic filtering can be difficult with these approaches. Since the texture data

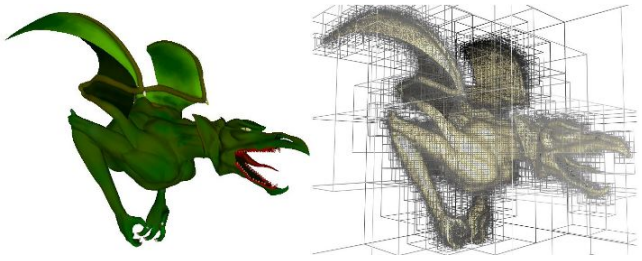


Figure 9: ***Octree Textures:*** *A hierarchical data-structure storing colors around an object surface [LHN05].*

is stored in 3D, these approaches typically require 3D painting, like the connectivity-based approaches discussed in the previous section. Animated (i.e. deforming) models can be easily handled by simply storing the rest positions of the vertices, which is analogous to storing 3D texture coordinates. Note that when the animation method already stores the rest shape (such as blend shapes), no extra storage is needed for accessing the texture data. The methods in this category are excellent choices when the overheads in computation and memory are acceptable. They are versatile, simple to implement, and robust. Yet, more elaborate methods in this category can provide improved efficiency in storage and texture access.

We discussed these methods in the context of surface texturing. Yet, similar approaches can be used for volume rendering [CNLE09], exploiting the fact that density fields are often sparse, with varying quantities of details.

5.1. Adaptive Texture Maps

Adaptive texture maps [KE02] are one of the earliest form of alternative texture mapping directly running on the GPUs. At the time it was published it showed how GPU functionalities, which were quite limited by then, could be used in novel ways to create data structures for encoding textures more efficiently.

The motivation for adaptive texture maps is that the texture content can have uneven frequency: some parts can have constant colors, while others contain detailed patterns. With traditional texture mapping, the entire texture is encoded using the same resolution, thus wasting memory where the content has low frequency. Similarly, in the case of volumetric representations for surface textures, only a small percentage of voxels contain data.

To address this problem, adaptive texture maps divide the 2D traditional texture in square tiles. The adaptive map is represented in memory by two separate textures (Figure 10). The first one is the *index grid*, which is a low resolution texture covering the original texture image by storing one entry per tile. Each entry is an index into the second texture, the *tile map*. The tile map holds the actual texture data. This representation is similar to the vector quantization scheme [BAC96], which was introduced much earlier for texture compression. This allows entirely skipping the storage cost of empty tiles, replacing them with a background color. It is also possible to repeat some tiles by indexing them multiple times and to store tiles at different resolutions, in which case the index grid stores an additional scaling factor.

In 2D, adaptive texture maps are mostly used as traditional

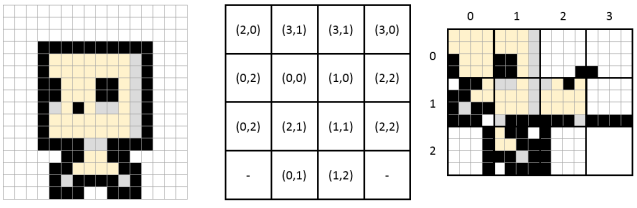
Table 8: *Adaptive Texture Maps*

Applicability	
Polygonal Meshes	Yes
Point Clouds	No
Implicit Surfaces	No
Shape/Topology Limits	None
Subdivisions	Yes
Tessellation Independence	If seams are preserved
Usability	
Automated Mapping	Limited
Mapping Customizability	Yes
Model Editing after Painting	Problematic
Resolution Readjustment	Yes (local resolution changes)
Texture Repetition	Yes (with instancing)
2D Image Representation	Poor
Filtering Quality	
Magnification Filtering	Yes
Minification Filtering	Yes, up to tile size
Anisotropic Filtering	Yes, with seam artifacts
Performance	
Vertex Data Duplication	Yes
Storage Overhead	2D mapping & indirection map
Access Overhead	Single indirection
Computation Overhead	Indirection
Hardware Filtering	Within tiles, not across boundaries
Implementation	
Asset Production	Automated tile construction
Rendering	Indirection

Table 9: *Octrees, N^3-trees, and Brickmaps*

Applicability	
Polygonal Meshes	Yes
Point Clouds	Yes
Implicit Surfaces	Yes
Shape/Topology Limits	Thin parts get same color
Subdivisions	Yes, with varying spatial resolution
Tessellation Independence	Yes, if shape is preserved
Usability	
Automated Mapping	Yes
Mapping Customizability	No
Model Editing after Painting	Yes (fast local reconstruction)
Resolution Readjustment	Yes (local subdivision)
Texture Repetition	No
2D Image Representation	No
Filtering Quality	
Magnification Filtering	Yes
Minification Filtering	Yes
Anisotropic Filtering	No
Performance	
Vertex Data Duplication	No
Storage Overhead	Hierarchy & rest-pose
Access Overhead	Hierarchy & indirections
Computation Overhead	Hierarchy & indirection
Hardware Filtering	No
Implementation	
Asset Production	3D painting
Rendering	Custom access & filtering

texture maps, with the ability to skip empty space and instantiate content. Ultimately they can be used to implement a paging system, where only tiles required for rendering are stored in the tile map, while the index grid serves as a virtual address table [LDN04, OvWS12]. In 3D, it allows volume textures to become practical for encoding surface properties. The rendering cost is relatively low, with a single indirection. However, special care must be taken to exploit native hardware interpolation, which is typically achieved by duplicating texture data around the tile boundaries. Mipmapping can be challenging. It works properly only for a limited number of levels and becomes discontinuous beyond a certain level. Yet, this problem can be addressed at the expense of additional texture lookups [Lef08].



(a) Original Texture (b) Index Grid (c) Tile Map

Figure 10: ***Adaptive Texture Maps:*** *(a) the texture is divided in square tiles and (c) non-empty tiles are stored in a compact tile map, while (b) an index grid covers the initial texture area and points into the tile map. Repeated tiles are stored only once.*

5.2. Octree Textures

Octree textures [BD02, DGPR02, LHN05, KLS*05] generalize the approach of adaptive texture maps to a full hierarchy. The hierarchy is encoded into a texture divided in small blocks of size $2 \times 2 \times 2$ for an octree (Figure 9). Each entry of each block stores either the texture data or encodes a pointer towards a child block.

Octree textures are used for encoding a volumetric texture around the object surface; thus, applying a texture without having to pre-compute a global planar parameterization. Thanks to the hierarchy, the structure remains compact in memory. The lookup, however, is more expensive by the nested dependent texture accesses. The trade-off between memory compactness and lookup cost can be alleviated by using higher resolution internal nodes (instead of $2 \times 2 \times 2$). This generalization is referred to as N^3-trees [LHN05], where the value of N may also vary at each level of the hierarchy.

Filtering becomes more complicated due to the hierarchical nature of the data structure. The hierarchy naturally encodes colors at all resolutions, storing the average color of subtrees within each parent node. However, trilinear interpolation requires multiple hierarchical lookups that have to be performed in software with a specialized shader code. Additional samples also have to be stored around the surface to properly define the interpolation of voxels. Regarding this point, it is interesting to note that primal trees afford for a more efficient interpolation, with fewer lookups [LH07].

5.3. Brick Maps

Brick maps [CB04] were developed for storing precomputed global illumination in arbitrary scenes. They also rely on an octree hierarchy, but each node stores a brick: a block of K^3 voxels. The child nodes also store bricks of size K^3; thus, they multiply the resolution by two at each octree level (Figure 11). This is different from N^3-trees where the subdivision along each axis is N at each step, with N^3 child nodes, thus offering a different trade-off.

The data structure is geared towards caching irradiance data along surfaces, allowing to render extremely large scenes by loading and unloading voxel bricks as required. An implementation is included in the RenderMan software.

Similarly to N^3-trees, the shallowness of the hierarchy reduces texture access cost, but increases the total memory consumption.

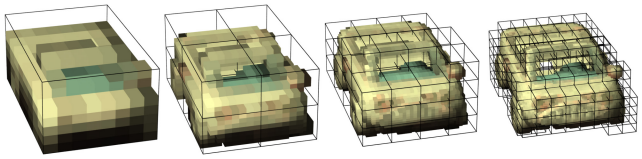


Figure 11: ***Brick Maps:*** *Different levels of brick maps representing the colors on an example model surface [CB04].*

5.4. Perfect Spatial Hashing

Perfect spatial hashing [LH06] encodes the texture data in a sparse voxel grid around the surface. However, instead of using a hierarchy, this method is based on spatial hashing (Figure 12). A hash function $H(x,y,z)$ directly computes the position where the data of a voxel at a given 3D position (x,y,z). The hahs function itself is

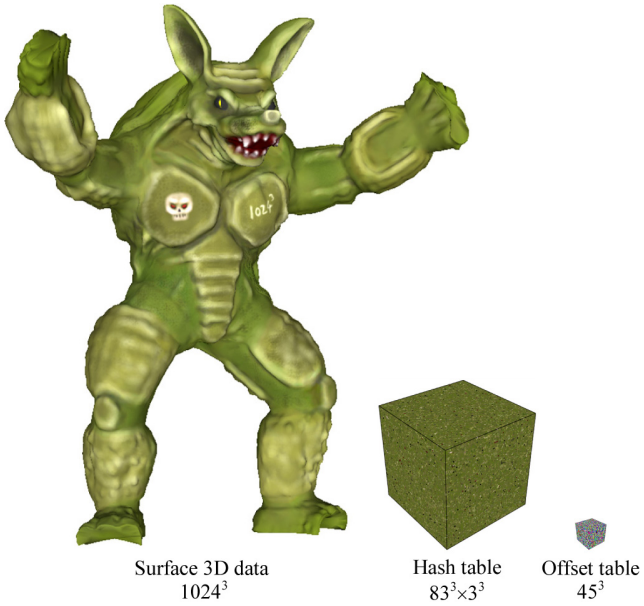


Figure 12: ***Perfect Spatial Hashing:*** *An example model with its hash table that contains the color data and its offset table [LH06].*

Table 10: *Perfect Spatial Hashing*

Applicability	
Polygonal Meshes	Yes
Point Clouds	Yes
Implicit Surfaces	Yes
Shape/Topology Limits	Thin parts get same color
Subdivisions	Yes, with varying spatial resolution
Tessellation Independence	Yes, if shape is preserved
Usability	
Automated Mapping	Yes, but slow
Mapping Customizability	No
Model Editing after Painting	No
Resolution Readjustment	No
Texture Repetition	No
2D Image Representation	No
Filtering Quality	
Magnification Filtering	Yes
Minification Filtering	Yes
Anisotropic Filtering	No
Performance	
Vertex Data Duplication	No
Storage Overhead	~4-bits per entry, borders (with blocking), & rest-pose
Access Overhead	Single indirection
Computation Overhead	Hash precomputation, indirection
Hardware Filtering	No
Implementation	
Asset Production	3D painting
Rendering	Indirection & custom filtering (with blocking)

stored in a compact hash table. The hash function H is precomputed to exhibit two important properties. First, no collision occurs (i.e. *perfect* hash), that is $H(\mathbf{p}_1) = H(\mathbf{p}_2) \Rightarrow \mathbf{p}_1 = \mathbf{p}_2$, where $\mathbf{p}_1$ and $\mathbf{p}_2$ are positions on the 3D model. This exploits the fact that, in the context of texturing, only voxels containing texture data are accessed and stored. Second, the hash table is *minimal* and it is just large enough to contain the voxel data. The construction of the hash function is time consuming, but it is automatic, though later approaches have explored fast construction [ASA*09, GLHL11]. The spatial overhead is due to the encoding of H, and requires on average 4 bits per voxel.

This results in a very efficient method where the sparse data is accessed through only two texture lookups (a single indirection). However, the loss of cache coherency, due to the randomization of the access by the hash, penalizes access performance.

6. Volume-based Parametrizations

Volume-based parameterizations define a mapping from the surface to the 2D texture space as a function of the volume embedding the object. These approaches encode the texture as a traditional 2D texture map: a *global* planar mapping is defined exactly as with the standard texture mapping approach. The difference, however, lies in how the mapping is defined, which is usually performed from

the surface point coordinates (and possibly their normal [LD07]), often by accessing auxiliary volume data-structures.

These methods can be considered as splitting the texture computation into two components: the first one computes the *uv* coordinates by providing a mapping from the original model to an intermediate shape, and the second one maps the intermediate shape to a 2D texture that stores the texture data. The first component provides a continuous mapping and cuts are introduced as a part of the second component.

The properties of volume-based parameterizations have some similarities to sparse volumetric representations (Sec. 5). There is no need to store *uv* coordinates and these methods apply to other geometric representations (implicit surfaces, point clouds, etc.). This independence from the underlying mesh topology allows the same texture to be reused across mesh resolutions for LoD, because the necessary cuts do not have to be along the edges of the mesh. Animated (deforming) models require storing the rest positions, which may already be available with some animation methods (such as blend shapes). The data structures are also generally more compact than sparse volumetric representations, as they encode the mapping (which is usually smooth except for cuts) rather the texture data itself. Combined with hardware acceleration in the final 2D texture lookups, these approaches generally result in significantly lighter per-fragment processing and more compact structures, compared to sparse volumetric representations.

The core difficulty, however, lies in how to encode the mapping efficiently and how to precompute it robustly. Also, thin geometric features pose a potential threat, as opposite parts of the surface must be mapped to different texture positions, in spite of their geometric proximity in 3D. Each method in this category addresses these challenges differently.

6.1. TileTrees

A tiletree [LD07] stores 2D square texture tiles on the faces of the cubic nodes of an octree (Figure 13). Conceptually, the square tiles are positioned in front of the corresponding surface pieces. By storing 2D tiles alongside the surface, the tiletree strongly reduces the depth of the volume data structure, as compared to octree textures (Sec. 5.2). The texture tiles are accessed using the surface point coordinates and normals. The tiletree is specially constructed for a given surface, determining the required set of tiles to achieve a full, injective coverage. After the hierarchy is built, the tiles are packed in a 2D texture. The entire process is automatic, robust, and fast.

Tiletrees offer many of the advantages of volume mappings, while providing a simple and automatic construction process, that does not require a global mapping optimization. The use of normals in the lookup helps disambiguating thin features (i.e. two sides of a thin sheet project to two different tiles). However, tiletrees have several limitations. The per-fragment access cost is relatively high, compared to other volume-based parameterizations; this is due to the traversal of the tree hierarchy and the explicit interpolation that requires multiple lookups. Tiletrees suffer from the same distortions as cube-maps. The mapping incurs a large number of seams. While the seams are not visible, they prevent using the 2D texture map directly as a painting canvas. Similarly, the seams limit mipmapping capabilities and cause overhead due to texel replication to ensure correct texture data interpolation. The reliance of normals can prevent reusing the same texture for different tessellations, if the normals differ significantly.

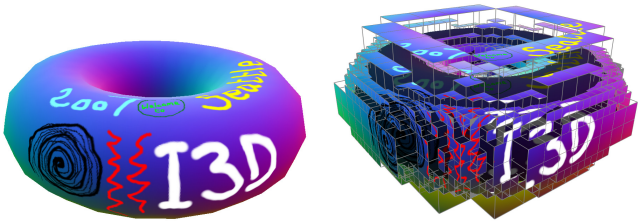


Figure 13: ***TileTrees:*** *An example model and its tiletree [LD07].*

Table 11: *TileTrees*

Applicability	
Polygonal Meshes	Yes
Point Clouds	Yes (if normal available)
Implicit Surfaces	Yes
Shape/Topology Limits	Limited local complexity
Subdivisions	Yes
Tessellation Independence	Yes, if shape is preserved
Usability	
Automated Mapping	Yes
Mapping Customizability	No
Model Editing after Painting	No
Resolution Readjustment	Yes (per tile)
Texture Repetition	No
2D Image Representation	Poor
Filtering Quality	
Magnification Filtering	Yes
Minification Filtering	Yes (up to tile size)
Anisotropic Filtering	No
Performance	
Vertex Data Duplication	No
Storage Overhead	Shallow tree & rest-pose
Access Overhead	Few indirections
Computation Overhead	Indirections
Hardware Filtering	No
Implementation	
Asset Production	3D painting
Rendering	Custom access & filtering

6.2. PolyCube-Maps

Polycube-maps can be considered an extension of cube-maps [Gre86]. Cube-maps are traditionally employed for *directional* texturing only, such as computing reflections (environment maps). However, they may also be used to texture approximately cubic or spherical objects. In this case, the mesh vertices are assigned 3D texture coordinates, that refer to points on the surface of the cube. The assignment can be precomputed and stored at vertices or, for more spherical objects, be procedural and left entirely to the vertex shader: the trivial gnomonic function can be employed for this

Table 12: *PolyCube-Maps*

Applicability	
Polygonal Meshes	Yes
Point Clouds	Single color per point
Implicit Surfaces	No
Shape/Topology Limits	Topology must be reproduced with low-resolution polycube
Subdivisions	Yes
Tessellation Independence	Yes
Usability	
Automated Mapping	Polycube construction needed
Mapping Customizability	Yes
Model Editing after Painting	No
Resolution Readjustment	Possible by adding cubes
Texture Repetition	Possible by using Wang tiles
2D Image Representation	Poor
Filtering Quality	
Magnification Filtering	Yes
Minification Filtering	Yes (up to cube size)
Anisotropic Filtering	No
Performance	
Vertex Data Duplication	No
Storage Overhead	3D texture coordinates
Access Overhead	Single indirection
Computation Overhead	Indirections & 5 cases
Hardware Filtering	Yes
Implementation	
Asset Production	3D painting & polycube generation
Rendering	Custom access

purpose, but other less distorted closed-form alternatives have been proposed [WWL07].

Polycube-maps [THCM04] generalize the concept of cube-maps by substituting the cube the with an arbitrary *polycube* (a geometric shape defined as assemblage of cubes attached face to face) as the intermediate volumetric representation (Figure 14). A set of square tiles are assigned to the faces of the polycube and the tiles are then packed on a 2D texture sheet. The *uv* map of a mesh is then defined by explicitly assigning to each vertex of the mesh a 3D parametric position (u,v,s) on the surface of the polycube. Texture lookup first accesses a tiny volumetric structure representing the polycube and the packing of the tiles, then the texture data is sampled from the 2D texture sheet, analogous to a cube-map.

The task of automatically or semi-automatically constructing a polycube-map received considerable attention [LJFW08, WJH*08, HWFQ09, WYZ*11, YZWL14, HJS*14, HZ16]. These methods aim to optimize different (and potentially conflicting metrics), such as limiting the number of cubes, limiting the number of irregular (non-valency 4) vertices, matching the topology of the input surface, and maximizing shape similarity between the two shapes. Polycube-map based surface parametrizations have also been used for problems other than texture mapping, such as construction of higher-order surface representations [WHL*08, LJFW08], reverse subdivision [XGH*11, GXH*13],

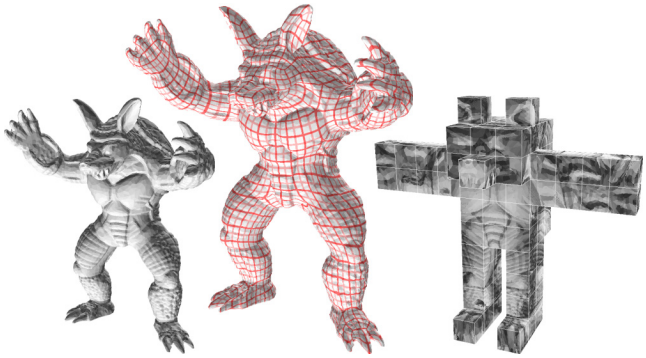


Figure 14: ***PolyCube-Maps:*** *An example model, its polycube-map parameterization, and the corresponding polycube [THCM04].*

volumetric modelling of thin shells [HXH10], shape analysis and design [XGH*11, GXH*13], cross-parametrization and morphing [FJFS05, WYZ*11] and, especially, semiregular hexahedral volume remeshing [GSZ11, FXBH16, YZWL14, HZ16] intended for physical simulations.

Polycube-maps share the advantages of tiletrees (Sec. 6.1), but they incur a much smaller texture access overhead. This is because polycube-maps do not require a hierarchical data structure and the texture data can be accessed using a single indirection. They also produce fewer seams than tiletrees, all of which are *invisible* (i.e. free from interpolation artifacts). Texture repetition can be achieved using Wang tiles [FL05, CL*10].

As compared to traditional texture mapping, they replace the problem of defining a *uv* mapping with the problem of constructing a polycube. Furthermore, polycube-maps are not as general as traditional texture mapping, because small features of the model (such as small handles or tunnels) may lead to excessively complex polycubes. Also, polycube-maps require storing the parameterization as a 3D mapping (as opposed to 2D), but do not require vertex duplications for handling seams.

6.3. Volume-Encoded UV-Maps

Volume-encoded uv-maps generate *uv* coordinates for points on the model surface based on their positions using a volumetric function defined in the bounding box of the model. This volumetric function is represented using a low-resolution 3D lattice and stored as a standard 3D texture. The texture data is stored as a standard 2D texture (Figure 15). The seams are defined by the volumetric function and they do not necessarily align with the model edges. In fact, the topology of the model is not used for *uv* mapping; therefore, this approach provides complete tessellation independence and it can handle any surface representation.

Accessing the texture data requires first computing the *uv* coordinates, which is achieved by sampling the 3D texture that stores the volumetric function. Then, the *uv* coordinates are used for reading the texture data. Hardware accelerated trilinear filtering can be used for sampling both textures, resulting in low computational overhead with a single indirection. Anisotropic filtering is not supported.

Table 13: *Volume-Encoded UV-Maps*

Applicability	
Polygonal Meshes	Quads & triangles
Point Clouds	Yes
Implicit Surfaces	Yes
Shape/Topology Limits	Thin parts get the same color
Subdivisions	Yes
Tessellation Independence	Yes
Usability	
Automated Mapping	Limited
Mapping Customizability	Yes
Model Editing after Painting	No
Resolution Readjustment	Problematic
Texture Repetition	Yes
2D Image Representation	Yes
Filtering Quality	
Magnification Filtering	Yes, with seam artifacts
Minification Filtering	Yes, with seam artifacts
Anisotropic Filtering	No
Performance	
Vertex Data Duplication	No
Storage Overhead	Can be much better or much worse
Access Overhead	Single indirection
Computation Overhead	Indirection
Hardware Filtering	Yes
Implementation	
Asset Production	Most existing tools can be used
Rendering	Simple indirection

As compared to standard 2D textures, volume-encoded uv-maps replace the problem of defining a *uv* mapping with the problem of defining a volume-encoded parameterization. This task consists of selecting a proper lattice resolution, selecting the block size (which controls the density of potential cuts), and assigning *uv* coordinates to lattice vertices. The volumetric function needs to be optimized to achieve a parameterization with low distortion, fewer cuts, desirable cut positions, and an efficient usage of the 2D texture layout. Constraining the volumetric function to be locally as constant as possible along the surface normal direction, allows sharing the same function with geometrically similar model, making this approach highly suitable for LoD representations.

For many models, this technique is capable of expressing a *uv* mapping that is qualitatively similar to (and as customizable as) traditional texture mapping. For example, it can represent an atlas-based mapping (where the surface is divided into islands, each island mapped to a 2D chart, and each chart is efficiently packed in the final 2D texture); different cuts can be designated to open the surface and cuts can be added to reduce mapping distortions; texture symmetries can be exploited by mapping different parts of the models to the same part of the texture; and cut positions can be tailored according to user needs (for avoiding cuts in semantically important areas). On the other hand, given that the volumetric resolution is constant, this technique can fail to produce injective mappings in the presence of small geometric features.

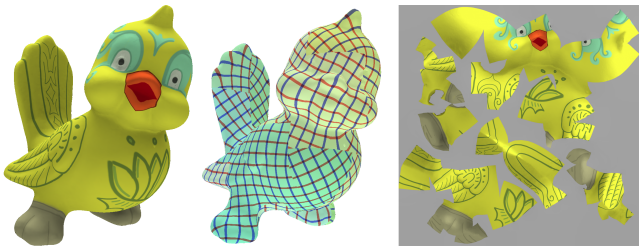


Figure 15: ***Volume-Encoded UV-Maps:*** *An example model, its parameterization, and the corresponding 2D texture [Tar16].*

The computational overhead at render time is much smaller than other alternatives in this category (except simple cube-maps) and it is also less expensive in terms of memory consumption. Since a hierarchical structure (such as an octree) is not used, texture accesses include only a single indirection. This approach is compatible with tangent-space normal mapping.

7. Conclusions

We have provided an overview of various techniques that rethink the concept and the process of texture mapping that is used ubiquitously in computer graphics applications to enrich surfaces with high-frequency signals. Since virtually all existing graphics tools and hardware have been designed around traditional texture mapping, the adaptation of its alternatives has been slow in the graphics community. As the intrinsic problems of traditional texture mapping continues to hinder the efficiency of content creation process and the quality of the rendered images, the graphics community now appears to be open to rethinking texture mapping.

All methods we discussed above have various advantages and limitations. Therefore, we believe that the most beneficial way of rethinking texture mapping depends on the properties of the given application, and we hope that the detailed evaluations provided above can be helpful in identifying the desirable alternatives. We also hope that the discussions and evaluations above indicate the open problems in this domain for stimulating future research.

References

[APL15] AIGERMAN N., PORANNE R., LIPMAN Y.: Seamless Surface Mappings. *ACM Trans. Graph. 34*, 4 (July 2015), 72:1–72:13. URL: http://doi.acm.org/10.1145/2766921, doi:10.1145/2766921. 5

[ASA*09] ALCANTARA D. A., SHARF A., ABBASINEJAD F., SENGUPTA S., MITZENMACHER M., OWENS J. D., AMENTA N.: Real-time parallel hashing on the gpu. In *ACM SIGGRAPH Asia 2009 Papers* (2009), SIGGRAPH Asia '09, pp. 154:1–154:9. 12

[BAC96] BEERS A. C., AGRAWALA M., CHADDHA N.: Rendering from Compressed Textures. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1996), SIGGRAPH '96, ACM, pp. 373–378. URL: http://doi.acm.org/10.1145/237170.237276, doi:10.1145/237170.237276. 10

[BD02] BENSON D., DAVIS J.: Octree textures. *ACM Transactions on Graphics 21*, 3 (2002), 785–790. 2, 10, 11

[BL08] BURLEY B., LACEWELL D.: Ptex: Per-face texture mapping for production rendering. In *Eurographics Symp. on Rendering* (2008), EGSR'08, pp. 1155–1164. 1, 2, 7

[BZK09] BOMMES D., ZIMMER H., KOBBELT L.: Mixed-integer quadrangulation. *ACM Trans. Graph. 28*, 3 (July 2009), 77:1–77:10. URL: http://doi.acm.org/10.1145/1531326.1531383, doi:10.1145/1531326.1531383. 5

[CB04] CHRISTENSEN P. H., BATALI D.: An irradiance atlas for global illumination in complex production scenes. In *Proc. of Rendering Techniques* (2004), EGSR'04, pp. 133–141. 1, 2, 12

[CL*10] CHANG C.-C., LIN C.-Y., ET AL.: Texture tiling on 3d models using automatic polycube-maps and wang tiles. *Journal of Information Science and Engineering 26*, 1 (2010), 291–305. 14

[CNLE09] CRASSIN C., NEYRET F., LEFEBVRE S., EISEMANN E.: Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of Symposium on Interactive 3D Graphics and Games* (2009), I3D '09, ACM, pp. 15–22. 1, 10

[DGM18] DE GOES F. F., MEYER M.: Generating uv maps for modified meshes, Mar. 29 2018. US Patent App. 15/279,252. 4

[DGPR02] DEBRY D. G., GIBBS J., PETTY D. D., ROBINS N.: Painting and rendering textures on unparameterized models. *ACM Transactions on Graphics 21*, 3 (2002), 763–768. 11

[FJFS05] FAN Z., JIN X., FENG J., SUN H.: Mesh morphing using polycube-based cross-parameterization. *Computer Animation and Virtual Worlds 16*, 3-4 (2005), 499–508. URL: http://dx.doi.org/10.1002/cav.92, doi:10.1002/cav.92. 14

[FL05] FU C.-W., LEUNG M.-K.: Texture tiling on arbitrary topological surfaces using wang tiles. In *Proceedings of the Sixteenth Eurographics Conference on Rendering Techniques* (Aire-la-Ville, Switzerland, Switzerland, 2005), EGSR '05, Eurographics Association, pp. 99–104. URL: http://dx.doi.org/10.2312/EGWR/EGSR05/099-104, doi:10.2312/EGWR/EGSR05/099-104. 14

[FXBH16] FANG X., XU W., BAO H., HUANG J.: All-hex meshing using closed-form induced polycube. *ACM Trans. Graph. 35*, 4 (July 2016), 124:1–124:9. URL: http://doi.acm.org/10.1145/2897824.2925957, doi:10.1145/2897824.2925957. 14

[GLHL11] GARCÍA I., LEFEBVRE S., HORNUS S., LASRAM A.: Coherent parallel hashing. *ACM Trans. Graph. 30*, 6 (2011). 2, 12

[GP09] GONZÁLEZ F., PATOW G.: Continuity Mapping for Multi-chart Textures. *ACM Trans. Graph. 28*, 5 (2009), 109:1–109:8. 5

[Gre86] GREENE N.: Environment mapping and other applications of world projections. *IEEE Comput. Graph. Appl. 6*, 11 (Nov. 1986), 21–29. URL: http://dx.doi.org/10.1109/MCG.1986.276658, doi:10.1109/MCG.1986.276658. 13

[GSZ11] GREGSON J., SHEFFER A., ZHANG E.: All-hex mesh generation via volumetric polycube deformation. *Computer Graphics Forum 30*, 5 (2011), 1407–1416. URL: http://dx.doi.org/10.1111/j.1467-8659.2011.02015.x, doi:10.1111/j.1467-8659.2011.02015.x. 14

[GXH*13] GARCIA I., XIA J., HE Y., XIN S., PATOW G.: Interactive Applications for Sketch-Based Editable Polycube Map. *IEEE Transactions on Visualization and Computer Graphics 19*, 7 (July 2013), 1158–1171. doi:10.1109/TVCG.2012.308. 14

[HJS*14] HUANG J., JIANG T., SHI Z., TONG Y., BAO H., DESBRUN M.: l_1-based construction of polycube maps from complex shapes. *ACM Trans. Graph. 33*, 3 (June 2014), 25:1–25:11. URL: http://doi.acm.org/10.1145/2602141, doi:10.1145/2602141. 14

[HWFQ09] HE Y., WANG H., FU C.-W., QIN H.: A divide-and-conquer approach for automatic polycube map construction. *Computers & Graphics 33*, 3 (2009), 369–380. 14

[HXH10] HAN S., XIA J., HE Y.: Hexahedral shell mesh construction via volumetric polycube map. In *Proceedings of the 14th ACM Symposium on Solid and Physical Modeling* (New York, NY, USA, 2010), SPM '10, ACM, pp. 127–136. URL: http://doi.acm.org/10.1145/1839778.1839796, doi:10.1145/1839778.1839796. 14

[HZ16] HU K., ZHANG Y. J.: Centroidal voronoi tessellation based polycube construction for adaptive all-hexahedral mesh generation. *Computer Methods in Applied Mechanics and Engineering 305* (2016), 405 – 421. URL: http://www.sciencedirect.com/science/article/pii/S0045782516301037, doi:https://doi.org/10.1016/j.cma.2016.03.021. 14

[JTPSH15] JAKOB W., TARINI M., PANOZZO D., SORKINE-HORNUNG O.: Instant field-aligned meshes. *ACM Trans. Graph. 34*, 6 (Nov. 2015). doi:10.1145/2816795.2818078. 5

[KE02] KRAUS M., ERTL T.: Adaptive texture maps. In *Proceedings of Conference on Graphics Hardware* (2002), HWWS '02, pp. 7–15. 2, 10

[KLS*05] KNISS J., LEFOHN A., STRZODKA R., SENGUPTA S., OWENS J. D.: Octree textures on graphics hardware. In *ACM SIGGRAPH 2005 Sketches* (2005), SIGGRAPH 2005. 11

[Lam15] LAMBERT T.: From 2d to 3d painting with mesh colors. In *ACM SIGGRAPH 2015 Talks* (New York, NY, USA, 2015), SIGGRAPH '15, ACM, pp. 72:1–72:1. 8

[LD07] LEFEBVRE S., DACHSBACHER C.: Tiletrees. In *Proceedings of the Symposium on Interactive 3D Graphics and Games* (2007), ACM, pp. 25–31. 1, 2, 13

[LDN04] LEFEBVRE S., DARBON J., NEYRET F.: *Unified Texture Management for Arbitrary Meshes*. Tech. Rep. RR5210-, INRIA, may 2004. URL: http://www-evasion.imag.fr/Publications/2004/LDN04. 11

[Lef08] LEFEBVRE S.: Filtered Tilemaps (in Shader X6). In *Shader X6: Advanced Rendering Techniques*, Engel W., (Ed.), Shader X6. Charles River Media, 2008, pp. 63–72. 11

[LFJG17] LIU S., FERGUSON Z., JACOBSON A., GINGOLD Y.: Seamless: Seam erasure and seam-aware decoupling of shape from mesh resolution. *ACM Trans. Graph. 36*, 6 (Nov. 2017), 216:1–216:15. URL: http://doi.acm.org/10.1145/3130800.3130897, doi:10.1145/3130800.3130897. 2, 4, 5

[LH06] LEFEBVRE S., HOPPE H.: Perfect spatial hashing. In *ACM Transactions on Graphics* (2006), vol. 25, ACM, pp. 579–588. 1, 2, 12

[LH07] LEFEBVRE S., HOPPE H.: Compressed random-access trees for spatially coherent data. In *Rendering Techniques (Proceedings of the Eurographics Symposium on Rendering)* (2007), Eurographics. URL: http://www-sop.inria.fr/reves/Basilic/2007/LH07. 11

[LHN05] LEFEBVRE S., HORNUS S., NEYRET F.: Octree textures on the gpu. *GPU gems 2* (2005), 595–613. 1, 2, 10, 11

[LJFW08] LIN J., JIN X., FAN Z., WANG C. C. L.: Automatic polycube-maps. In *Advances in Geometric Modeling and Processing* (Berlin, Heidelberg, 2008), Chen F., Jüttler B., (Eds.), Springer Berlin Heidelberg, pp. 3–16. 14

[MCT16] MAGRI D., CIGNONI P., TARINI M.: Anisotropic matcap: easy capture and reproduction of anisotropic materials. In *Proceedings of the Conference on Smart Tools and Applications in Computer Graphics* (2016), Eurographics Association, pp. 71–78. 6

[OvWS12] OBERT J., VAN WAVEREN J. M. P., SELLERS G.: Virtual texturing in software and hardware. In *ACM SIGGRAPH 2012 Courses* (New York, NY, USA, 2012), SIGGRAPH '12, ACM, pp. 5:1–5:29. URL: http://doi.acm.org/10.1145/2343483.2343488, doi:10.1145/2343483.2343488. 11

[PCK04] PURNOMO B., COHEN J. D., KUMAR S.: Seamless texture atlases. In *Proceedings of Symposium on Geometry Processing* (2004), ACM, pp. 65–74. 2, 6

[PKCH18] PRADA F., KAZHDAN M., CHUANG M., HOPPE H.: Gradient-domain Processing Within a Texture Atlas. *ACM Trans. Graph. 37*, 4 (July 2018), 154:1–154:14. URL: http://doi.acm.

org/10.1145/3197517.3201317, doi:10.1145/3197517.3201317. 4

[RLL*06] RAY N., LI W. C., LÉVY B., SHEFFER A., ALLIEZ P.: Periodic global parameterization. *ACM Trans. Graph. 25*, 4 (Oct. 2006), 1460–1485. URL: http://doi.acm.org/10.1145/1183287.1183297, doi:10.1145/1183287.1183297. 5

[RNLL10] RAY N., NIVOLIERS V., LEFEBVRE S., LEVY B.: Invisible Seams. *Computer Graphics Forum* (2010). 1, 2, 4, 5

[SKNS15] SCHÄFER H., KEINERT B., NIEßNER M., STAMMINGER M.: Local Painting and Deformation of Meshes on the GPU. *Comput. Graph. Forum 34*, 1 (Feb. 2015), 26–35. URL: http://dx.doi.org/10.1111/cgf.12456, doi:10.1111/cgf.12456. 8

[SPGT18] SCHERTLER N., PANOZZO D., GUMHOLD S., TARINI M.: Generalized Motorcycle Graphs for Imperfect Quad-dominant Meshes. *ACM Trans. Graph. 37*, 4 (July 2018), 155:1–155:16. URL: http://doi.acm.org/10.1145/3197517.3201389, doi:10.1145/3197517.3201389. 5

[SPM*13] SCHAEFER H., PRUS M., MEYER Q., SÜEβMUTH J., STAMMINGER M.: Multiresolution Attributes for Hardware Tessellated Objects. *IEEE Transactions on Visualization and Computer Graphics 19*, 9 (Sep. 2013), 1488–1498. doi:10.1109/TVCG.2013.44. 9

[SSGH01] SANDER P. V., SNYDER J., GORTLER S. J., HOPPE H.: Texture Mapping Progressive Meshes. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 2001), SIGGRAPH '01, ACM, pp. 409–416. URL: http://doi.acm.org/10.1145/383259.383307, doi:10.1145/383259.383307. 4

[Syl15] SYLVAN S.: Fixing Texture Seams With Linear Least-Squares, Nov. 2015. URL: https://www.sebastiansylvan.com/post/LeastSquaresTextureSeams/. 5

[Tar12] TARINI M.: Cylindrical and toroidal parameterizations without vertex seams. *Journal of Graphics Tools 16*, 3 (2012), 144–150. 2, 5, 6

[Tar16] TARINI M.: Volume-encoded uv-maps. *ACM Transactions on Graphics 35*, 4 (July 2016), 107:1–107:13. URL: http://doi.acm.org/10.1145/2897824.2925898, doi:10.1145/2897824.2925898. 1, 2, 15

[THCM04] TARINI M., HORMANN K., CIGNONI P., MONTANI C.: Polycube-maps. *ACM Trans. Graph. 23* (2004), 853–860. 1, 2, 5, 14

[Tot13] TOTH R.: Avoiding Texture Seams by Discarding Filter Taps. *Journal of Computer Graphics Techniques (JCGT) 2*, 2 (December 2013), 91–104. URL: http://jcgt.org/published/0002/02/07/. 8

[WHL*08] WANG H., HE Y., LI X., GU X., QIN H.: Polycube splines. *Computer-Aided Design 40*, 6 (2008), 721 – 733. Selected Papers from the ACM Solid and Physical Modeling Symposium 2007. URL: http://www.sciencedirect.com/science/article/pii/S0010448508000389, doi:https://doi.org/10.1016/j.cad.2008.01.012. 14

[WJH*08] WANG H., JIN M., HE Y., GU X., QIN H.: User-controllable polycube map for manifold spline construction. In *Proceedings of the 2008 ACM Symposium on Solid and Physical Modeling* (New York, NY, USA, 2008), SPM '08, ACM, pp. 397–404. URL: http://doi.acm.org/10.1145/1364901.1364958, doi:10.1145/1364901.1364958. 14

[WWL07] WAN L., WONG T.-T., LEUNG C.-S.: Isocube: Exploiting the cubemap hardware. *IEEE Transactions on Visualization and Computer Graphics 13*, 4 (2007), 720–731. 14

[WYZ*11] WAN S., YIN Z., ZHANG K., ZHANG H., LI X.: A topology-preserving optimization algorithm for polycube mapping. *Computers & Graphics 35*, 3 (2011), 639 – 649. Shape Modeling Interna tional (SMI) Conference 2011. URL: http://www.sciencedirect.com/science/article/pii/S0097849311000574, doi:https://doi.org/10.1016/j.cag.2011.03.018. 14

[XGH*11] XIA J., GARCIA I., HE Y., XIN S.-Q., PATOW G.: Editable polycube map for gpu-based subdivision surfaces. In *Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2011), I3D '11, ACM, pp. 151–158. URL: http://doi.acm.org/10.1145/1944745.1944771, doi:10.1145/1944745.1944771. 14

[YKH10] YUKSEL C., KEYSER J., HOUSE D. H.: Mesh colors. *ACM Transactions on Grapics 29*, 2 (2010), 15:1–15:11. 1, 2, 8, 9

[Yuk16] YUKSEL C.: Mesh colors with hardware texture filtering. In *ACM SIGGRAPH 2016 Talks* (New York, NY, USA, 2016), SIGGRAPH '16, ACM. 2, 9, 10

[YZWL14] YU W., ZHANG K., WAN S., LI X.: Optimizing polycube domain construction for hexahedral remeshing. *Computer-Aided Design 46* (2014), 58 – 68. 2013 SIAM Conference on Geometric and Physical Modeling. URL: http://www.sciencedirect.com/science/article/pii/S0010448513001589, doi:https://doi.org/10.1016/j.cad.2013.08.018. 14