# PolyCube-Maps

Marco Tarini Kai Hormann Paolo Cignoni Claudio Montani
Visual Computing Lab, ISTI / CNR, Pisa*

## Abstract

Standard texture mapping of real-world meshes suffers from the presence of seams that need to be introduced in order to avoid excessive distortions and to make the topology of the mesh compatible to the one of the texture domain. In contrast, cube maps provide a mechanism that could be used for seamless texture mapping with low distortion, but only if the object roughly resembles a cube. We extend this concept to arbitrary meshes by using as texture domain the surface of a *polycube* whose shape is similar to that of the given mesh. Our approach leads to a seamless texture mapping method that is simple enough to be implemented in currently available graphics hardware.

**CR Categories:** I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, Shading, Shadowing, and Texture; I.3.1 [Computer Graphics]: Hardware Architecture—Graphics Processors

**Keywords:** texture mapping, *u*-*v*-mapping, atlas generation, surface parameterization, cube maps

**URL:** http://vcg.isti.cnr.it/polycubemaps

## 1 Introduction

The task of texture mapping a 3D surface with colour information from an image or with some other signal from a 2D domain is relevant for many applications in computer graphics and the quality of the result depends heavily on the quality of the underlying parameterization. Ideally, the parameterization should be conformal and area-preserving so as to avoid any signal distortion, but it is well-known that such an isometric mapping exists only for developable surfaces like cones and cylinders.

The standard approach for triangle meshes therefore is to cut the surface into several disk-like patches each of which can be parameterized with low distortion. This *multi-chart* or *atlas* approach inevitably produces *seams*, in other words, the boundaries of the patches need to be replicated so that there will be vertices that have the same position in 3D but different texture coordinates in 2D.

Besides the problem of segmenting a given mesh such that the seams become least visible, the process of creating a good overall mapping also requires to define suitable boundaries of the charts in the 2D domain, to compute the parameterizations for each individual patch, and to finally pack the texture patches efficiently into a rectangular shape. Artists who manually design *u*-*v*-mappings of their models with great care, as well as automatic or semi-automatic algorithms, strive to find the best possible solution for each of these steps and we have seen extremely good results even for very complex meshes in the past.

*e-mail: {tarini,hormann,cignoni,montani}@isti.cnr.it

However, we are so used to the maxim that texture mapping of triangle meshes requires a segmentation and the existence of seams, that we tend to forget that they are a most limiting factor. In fact, it is well-known that seams cause:

- *Mesh dependence*: The different levels-of-detail (LOD) of a multi-resolution model usually require an individual parameterization and texture image, unless care is taken that the patch boundaries coincide for all levels.
- *Inadequate filtering*: Mip-mapping and bilinear interpolation both require the texels to be contiguous, a property that is not satisfied at the patch boundaries. As a consequence seams are visible.
- *Wasted texture memory*: Even the best packing algorithms cannot avoid that some parts of the texture domain are not covered by a texture patch. The uncovered texels may be used to partially prevent the filtering artefact mentioned above by adding a certain number of texels to enlarge the chart boundaries, but in general they do not store any information and must be considered wasted.

In this paper we introduce PolyCube-Maps, a new mechanism for superior texture mapping, that avoids the first two drawbacks and wastes almost no texture memory. It can be seen as a generalization of the well known cube map mechanism (see Section 2).

### 1.1 Related work

Most of the work on texture mapping follows the multi-chart approach with varying focuses on the different aspects of a good atlas generation, namely partitioning, parameterization, and packing.

In order to enable parameterizations without any distortion, Cignoni et al. [1999] and Carr and Hart [2002] propose to let the patches be composed of a single triangle or pairs of triangles, but this results in a highly fragmented texture space and introduces seams all over the mesh. Other approaches consider larger patches and try to make the parameterization per patch as conformal or area-preserving as possible [Maillot et al. 1993; Lévy et al. 2002; Grimm 2002; Sorkine et al. 2002; Sander et al. 2003]. For more detailed information on parameterizations in general we refer the interested reader to the recent survey by Floater and Hormann [2004].

Multi-chart methods suffer from the fact that they produce seams with all the drawbacks mentioned above. Several authors therefore suggested to cut the surface where the seam is least visible [Piponi and Borshukov 2000; Lévy et al. 2002; Sheffer and Hart 2002]. Moreover, seams heavily constrain the underlying geometric representation of the mesh because triangles are not allowed to cross the patch boundaries. This is a severe limitation, for example, if one single texture shall be used for all LODs of a multi-resolution model. Cohen et al. [1998] addressed this problem by constructing the multi-resolution model with a constrained simplification sequence that handles the patch boundaries appropriately and an improvement of this method was presented by Praun et al. [2000].

The only way to avoid having seams is to choose a texture domain that has both the same topology as the given mesh and a similar shape. There exist several methods that construct a seamless parameterization of a mesh over a triangulated base complex [Eck et al. 1995; Lee et al. 1998; Khodakovsky et al. 2003; Praun and Hoppe 2003] and use it for remeshing and mesh compression.

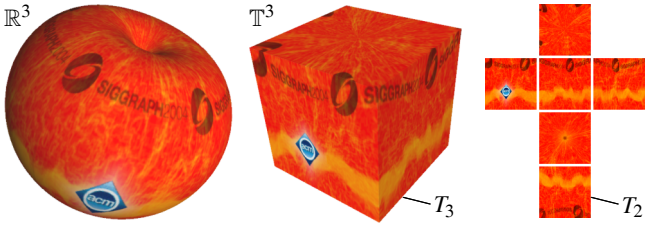


Figure 1: Cube maps can be used to seamlessly texture map an apple (left). In this case, the 3D texture domain $T_3$ is the surface of a single cube that is immersed in the 3D texture space $\mathbb{T}^3$ (middle) and corresponds to a 2D texture domain $T_2$ that consists of six square images (right).

### 1.2 Overview

In principle, these seamless parameterization methods could also be used for texture mapping. Colour information could be defined on the domain triangles of the base complex and the parameter values of the mesh vertices could be used to linearly map the colour to the mesh triangles. However, difficulties arise when the vertices of a mesh triangle have parameter values on different domain triangles and their linear interpolation is a *secant triangle* that falls outside the surface on which the colour information is defined.

Our approach is similar to the idea that we just sketched, but instead of using a triangulated base complex we use the surface of a polycube as texture domain. The special structure of this surface not only allows to efficiently store and access the colour information in a standard 2D texture, but also to handle the problem of secant triangles by simply projecting them onto the texture domain. Both this projection and the colour access are simple enough to be implemented in currently available graphics hardware. As a result we have a new seamless texture mapping technique. But let us start by explaining the basic idea behind our PolyCube-Maps which stems from the concept of cube maps.

## 2 PolyCube-Maps

Cube maps are commonly used for environment mapping, but they can also be used to define a seamless texture mapping for, say, an apple (see Figure 1). In fact, all we need to do is to assign to each vertex of such a 3D model a 3D texture position (which can differ from the vertex position). We call the space of possible texture positions the *3D texture space* and denote it by $\mathbb{T}^3$ to distinguish it from the object space $\mathbb{R}^3$ that contains the vertex positions. The cube map mechanism will then use a simple central projection to project the texture position of every rendered fragment onto the surface of a unitary cube with its centre at the origin. Let us call the surface of this cube the *3D texture domain* and denote it by $T_3$ with $T_3 \subset \mathbb{T}^3$. The cube map mechanism will further associate each point of $T_3$ with a position in a 2D texture space, which in this case is a collection of six planar square texture images, one for each face of the cube. We denote this *2D texture domain* by $T_2$. The resulting mapping will be seamless and will avoid all the drawbacks that we sketched in the introduction. However, this use of cube maps is fairly uncommon because it works only for quasi-spheres and our main idea is to extend this concept to more general shapes.

For our PolyCube-Maps we use as 3D texture domain $T_3$ the surface of a *polycube* rather than a single cube. A polycube is a shape composed of axis-aligned unit cubes that are attached face to face (see Figure 2, left). In order to get the best results, the used polycube should very roughly resemble the shape of the given mesh and capture the large scale features.

Once the polycube is defined, we proceed as follows. First we assign to each vertex $v$ of the mesh a unique 3D texture position

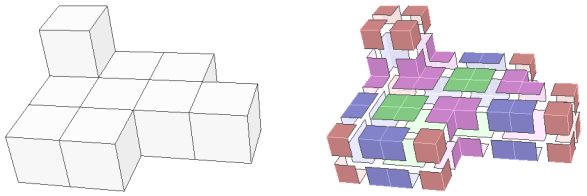

Figure 2: A polycube that consists of 10 cubes (left) and the partition of its surface into cells as explained in Section 3 (right).

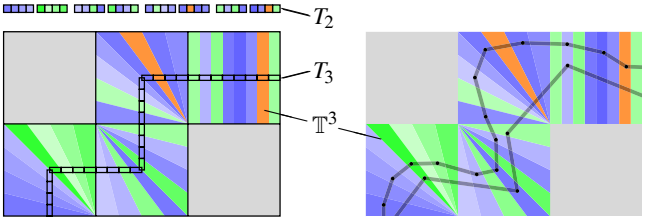


Figure 3: The 2D analogue of our method: the projection $\mathscr{P}$ maps each point (or fragment) in $\mathbb{T}^3$ onto the 3D texture domain $T_3$ (left). The mapping $\mathscr{M}$ can then be used to look up the texture information from the 2D texture domain $T_2$. PolyCube-Maps are not tied to the mesh structure and work for different mesh representations (right).

$v_{T3} = (v_r, v_s, v_t) \in \mathbb{T}^3$. At rendering time, the vertices and their 3D texture positions are fed to the graphics pipeline and the rasterizer interpolates the latter ones to get a 3D texture position $f_{I3}$ for every produced fragment $f$ and passes it on to the fragment shader. Even if all 3D texture positions $v_{T3}$ lie on $T_3$, this is not necessarily the case for the interpolated 3D texture position $f_{I3}$. Therefore, the fragment shader applies a projection $\mathscr{P} : \mathbb{T}^3 \to T_3$ to map $f_{I3}$ to a point $f_{T3}$ in the 3D texture domain. It further uses a second mapping $\mathscr{M} : T_3 \to T_2$ to determine the colour information at $f_{T3}$ that is stored in the 2D texture domain $T_2$ (see Figure 3 for the 2D analogue). In our case, $T_2$ is one single rectangular texture image with a packing of several square patches.

The most important feature of PolyCube-Maps is that the 3D texture coordinates vary continuously over the surface of the object and therefore enable a seamless texture mapping even though the texture information itself is stored as a collection of square images.

## 3 How PolyCube-Maps Work

Let us now explain in detail how we define the functions $\mathscr{P}$ and $\mathscr{M}$. Remember that we want to use PolyCube-Maps for the purpose of texture mapping and both $\mathscr{P}$ and $\mathscr{M}$ must be computed in the fragment shader which is the tighter sub-loop of the graphics pipeline. Therefore their implementation must be as simple and quick as possible. To achieve this, we define both mappings piecewise over an adequate partition of $\mathbb{T}^3$.

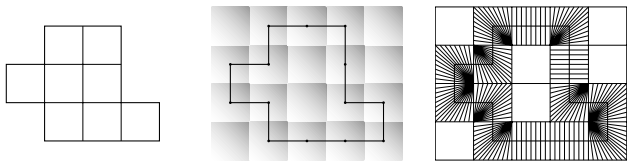

Figure 4: Another 2D analogue: we roughly approximate the object surface with a polycube (left), consider the dual space of unit cubes centered in the corners of the polycube (middle), and finally have for each non-empty cube a projection function that assigns each point inside a cube to the polycube surface (right).

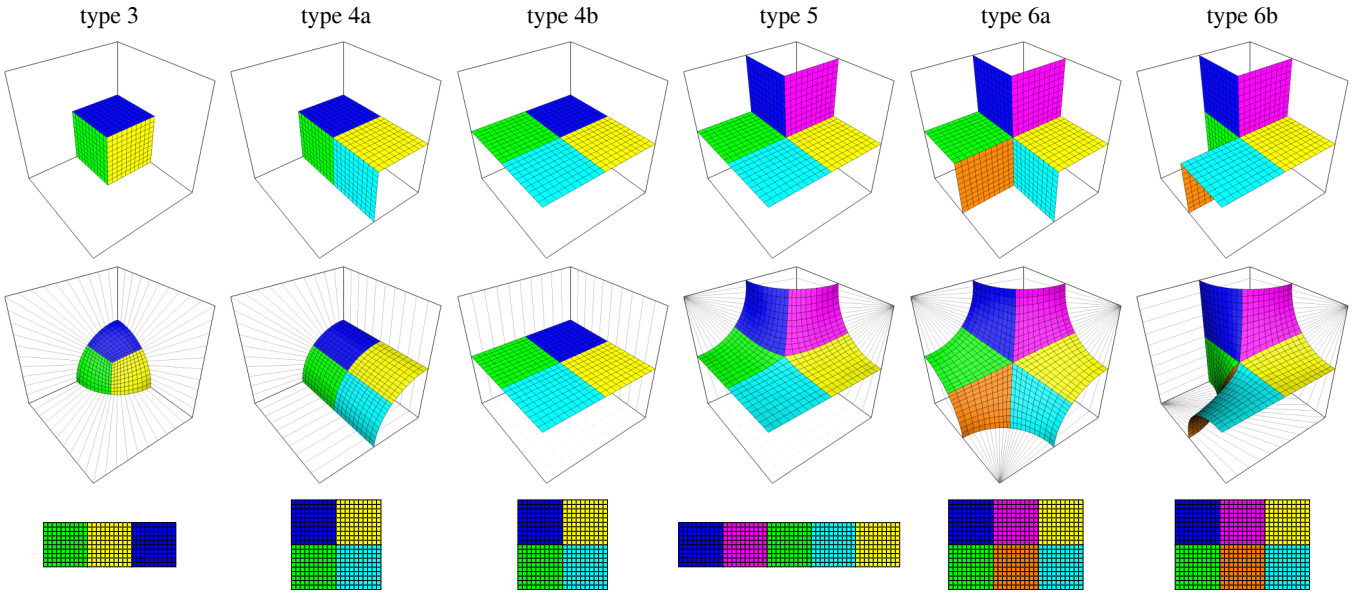


Figure 5: Six basic configurations can occur (up to rotations and reflections) in a non-empty cell. Top: sub-part of the polycube surface $T_3$ inside the cell with different colours for the individual facelets. Centre: projection lines of $\mathscr{P}$ inside the cell; they are orthogonal to the shown surfaces. Bottom: packing of squarelets into patches; the colour of the squarelets corresponds to the colour of the associated facelet.

We subdivide the 3D texture space $\mathbb{T}^3$ into cubic cells that are equal in size and orientation to the cubes that the polycube is composed of, but we offset these cells by 0.5 in each direction such that the vertices of the polycube lie at their centres (see Figure 4 for the 2D analogue). We chose this *dual* partition because of the following advantages:

1. it is still easy to determine in which cell a point in $\mathbb{T}^3$ lies,
2. the number of different configurations that can occur in the cells which intersect with the surface of the polycube is limited to a small number,
3. for each of these configurations it is possible to define the functions $\mathscr{P}$ and $\mathscr{M}$ piecewise inside each cell and still make them continuous at the faces of the cells,
4. these functions $\mathscr{P}$ and $\mathscr{M}$ are simple to compute.

## 3.1 Cell configurations

We now consider the intersection of the cells with the polycube surface $T_3$ and remark that it naturally subdivides the faces of the polycube into four *facelets*. There are 63 different configurations of facelets that can occur inside a cell which intersects with the polycube since we only consider polycubes with a two-manifold surface. These configurations can be further reduced down to the six basic configurations in Figure 5 if we take out rotational and reflectional similarities. Each cell contains between three and six facelets. For example, the polycube that consists of a single cube is decomposed into eight cells containing three facelets each (see Figure 6). More complex examples are shown in Figures 2 and 11 where the colour coding refers to the basic configuration and the facelets inside each cell are separated by thin white lines.

## 3.2 The projection $\mathscr{P}$

For each of the basic configurations we can now define the projection $\mathscr{P}$ that maps points in $\mathbb{T}^3$ onto $T_3$ by specifying the projection direction at any point $v_{T3} = (v_r, v_s, v_t)$ inside the cell. If we assume without loss of generality that the coordinates are between $(0,0,0)$ and $(1,1,1)$ then the projection direction of $v_{T3}$ for the first four configurations is given by

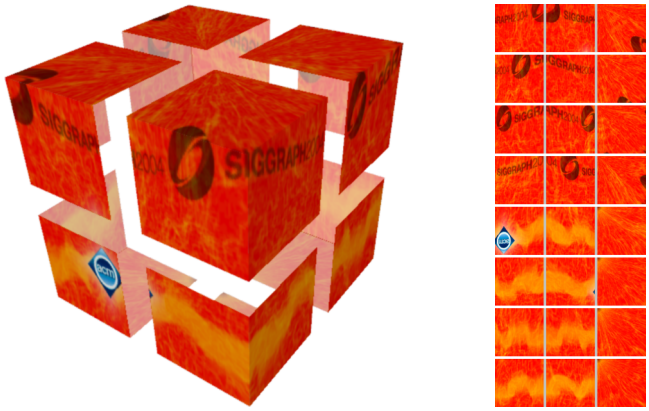

Figure 6: A simple PolyCube-Map example for the apple model of Figure 1. The polycube is subdivided into eight cells of type 3 but each with a different rotation. Each cell contains three facelets that are stored as a set of three texture squarelets.

| type 3 | type 4a | type 4b | type 5 |
|---|---|---|---|
| $(r,s,t)$ | $(r,0,t)$ | $(0,0,1)$ | $(\frac{r}{s}-r, 1-s, 1-t)$, if $s \geq r$<br>$(1-r, \frac{s}{r}-s, 1-t)$, if $s < r$ |

The formulas for the other two cases are slightly more complicated but since we did not use them in our implementation (see Section 5.2) we omit them for the sake of brevity.

A nice property of the so-defined projections is that they partition the interior of the cell into (infinitely many) non-intersecting lines such that all points on each of these lines are projected to the same point on $T_3$. Furthermore, the projection lines of two neighbouring cells are identical on the common face and hence the projection is globally continuous. The central row in Figure 5 gives an idea how the projections for the six basic configurations look like and an example of the 2D analogue is shown on the right of Figure 4.

### 3.3 The mapping $\mathscr{M}$

Conceptually, all colour information that we use for texture mapping is defined on the 3D texture domain $T_3$, i.e. on the facelets of the polycube. We could store the colour information as a 3D texture, but since most of the voxels would be empty, this option is by far too wasteful. Instead, we use a technique similar to the one that is used by cube maps.

Each facelet in $T_3$ is a square region and the mapping $\mathscr{M}$ maps this region to a corresponding *squarelet* in the 2D texture domain $T_2$. Our squarelets consist of $S \times S$ texels where the size $S$ is a user-specified parameter that must be chosen as a power of two.

All the squarelets for the facelets in one cell are packed together in a larger rectangular *texture patch* as shown in the bottom row of Figure 5. Due to the shape of the texture patches, they can easily be packed into $T_2$ and the position of, for example, the upper left corner can be stored as a *global* 2D offset for each particular cell. The *local* offsets to the individual squarelets inside the texture patch do not need to be stored as they are fixed for each of the six basic configurations.

Thus, applying the mapping $\mathscr{M}$ to a point $p \in T_3$ inside a cell consists of three steps. First, the relative position of $p$ inside the containing facelet is determined and multiplied with $S$ to give the relative position in the corresponding squarelet. Then the result is offset according to the fixed packing of the squarelets inside the texture patch and finally, the global offset of that patch in $T_2$ is added.

Both the local and the global offset are a multiple of $S$. This means that subsequent mip-map levels will merge together only texels coming from the same squarelet and that each squarelet is represented by a single pixel on the coarsest mip-map level $\log_2(S)$.

## 4 The 3D Look-up Table

The polycube must be adapted to the shape of the mesh that is to be textured and therefore we need a flexible way for the application that uses PolyCube-Maps to specify the polycube. Our solution is to store the cell-structure of $\mathbb{T}^3$ in a 3D look-up table $T_3^{\mathrm{LUT}}$ and define each cell through a set of parameters in the corresponding entry of $T_3^{\mathrm{LUT}}$. This look-up table must be kept in texture memory so that it can be accessed by the fragment shader. It can either be stored as a small 3D texture or, in order to reduce the number of rendering-time state variables, it can be serialized and physically kept in a subpart of $T_2$.

When processing a fragment $f$, the fragment shader first determines the cell that contains the fragment's interpolated 3D texture position $f_{I3}$. A single texture access to $T_3^{\mathrm{LUT}}$ at the corresponding entry returns all the parameters needed to compute $\mathscr{P}$ and $\mathscr{M}$ which are then used to find the final 2D texture position $f_{T2} \in T_2$.

Each entry of $T_3^{\mathrm{LUT}}$ is packed in one $(r,g,b)$-texel, so that it can be fetched with a single texture read instruction. More precisely, an entry $e$ is composed of three parts: $C$, $R$, and $O$, where

- $e.C$ is the index of one of the six basic cell configurations,
- $e.R$ is the index of one of the 24 axis-to-axis rotations,
- $e.O$ is the global offset of the patch corresponding to $e$ in $T_2$.

The rotation $e.R$ maps each axis into another axis in the positive or negative direction and is used to transform the given cell into the default orientation of the configuration $e.C$ shown in Figure 5. If there are several different rotations that achieve this, any of them can be chosen.

While the values $e.C$ and $e.R$ are packed together in a single byte, $e.O$ requires two bytes, one for each coordinate. These coordinates are expressed in multiples of $S$ so that 8 bits are sufficient to store them.

To make $T_3^{\mathrm{LUT}}$ a random access table, we also include all the empty cells in the bounding box of the polycube. However, the look-up table is still very small as a polycube typically consists of only a small number of cubes. In the examples that we show in this paper, $T_3^{\mathrm{LUT}}$ is always smaller than one Kilo texel, whereas the final texture $T_2$ can be several Mega texels.

If a model has multiple associated textures (e.g. a colour map, a normal map, and a specular coefficient map) then the textures can all share the same PolyCube-Map and they can be accessed through the same $T_3^{\mathrm{LUT}}$.

## 5 Fragment Shader Program

Each fragment that enters the fragment shader with a 3D texture position $f_{I3} \in \mathbb{T}^3$ will undergo the fragment program that is described by the following pseudo-code:

1. compute the 3D index $i \in \mathbb{N}^3$ of the cell that contains $f_{I3}$ by $i = \lfloor f_{I3} + (0.5, 0.5, 0.5) \rfloor$ and the subcell position $f_s = f_{I3} - i$ with $f_s \in [-0.5, +0.5)^3$,
2. fetch the entry $e$ from the texture $T_3^{\mathrm{LUT}}$ at index $i$,
3. rotate $f_s$ around the origin by $e.R$ (see Section 5.1),
4. apply the projection $\mathscr{P}$ and the mapping $\mathscr{M}$ (without the global offset) as defined for case $e.C$ (see Section 5.2),
5. add the global offset $e.O$,
6. use the result as an index to access the final texel value in the 2D texture $T_2$ (see Section 5.3).

Note that the index $i$ must first be serialized in the second step if the look-up table $T_3^{\mathrm{LUT}}$ is kept in a tiny subpart of the 2D texture $T_2$. Furthermore, if a model has multiple associated textures, then the last access is repeated for each texture using the same coordinate.

A similar texturing approach that utilizes a 3D look-up table in order to access information relative to a sub-volume of the space has also been used in the totally different context of compressing volumetric datasets [Schneider and Westermann 2003].

An optimized implementation of this algorithm in *OpenGL* ARB Fragment Program language is just 54 instructions long, leaving enough resources to implement other effects (e.g. shading). If a complex fragment program needs to access several textures in order to compute the final colour values, then the 54 instructions overhead is paid only once since all textures share the same PolyCube-Map.

### 5.1 Storing and applying rotations

We store the rotation $e.R$ that is applied in the third step of the algorithm above in a special way that is not only space-efficient but also allows to unpack and apply the rotation with few operations.

Any of the 24 possible axis-to-axis rotations can be coded in a series of 5 bits where each bit decides whether the corresponding operation of the following ordered list is to be performed or not:

1. $(r,s,t) \rightarrow (-t,-s,-r)$
2. $(r,s,t) \rightarrow (s,t,r)$
3. $(r,s,t) \rightarrow (s,t,r)$
4. $(r,s,t) \rightarrow (-r,-s,t)$
5. $(r,s,t) \rightarrow (r,-s,-t)$

Each operation can be implemented with a single *extended swizzle* command plus a conditional assignment that stores or discards the result (according to the value of the corresponding bit in $e.R$).

The current fragment shader languages do not support bit-wise operations, but the value of the $i$-th bit of $e.R$ can be extracted by

a sequence of supported operations: multiply by $2^{-(i+1)}$, take the fractional part, subtract 0.5, and finally check positivity. Vector operations can be used to recover four boolean values in parallel.

### 5.2 Applying the basic cases

After the rotation, the fragment shader computes the projection $\mathscr{P}$ and the mapping $\mathscr{M}$, both at one go. Although these functions are defined differently for the basic cases and it would be sufficient to execute only the code that implements the case $e.C$, we cannot do this because the shading languages lack a general branching construct. Instead we compute all cases in sequence and use a conditional assignment at the end of each case to record only the result of case $e.C$ in a register.

Since all cases are computed anyway, it is profitable to identify common subexpressions that are shared by different branches so as to reduce the number of overall instructions. Actually, we took care of maximizing the number of common subexpressions, when we decided on the default orientation of the basic cases and the packing of squarelets into patches.

A side effect of this non-branched architecture is that the cost of processing a fragment depends on the total number of instructions needed to cover all basic cases. Moreover, the cases 6a and 6b are the most complex and the least beneficial, as they hardly ever occur in useful polycubes. And since their implementation in the fragment shader would have burdened the execution for all the other cases as well, we decided to leave them out of our implementation. This choice implies a limitation on the polycube layout, but when we constructed polycubes for real-world meshes (see Section 6) we found the practical effect of this limitation to be negligible.

### 5.3 Filtering and mip-mapping

When the final texel value is fetched from $T_2$ in the last step of the algorithm, the mip-mapping mechanism still works (including the linear interpolation between different mip-map levels), because the size and the positions of squarelets are both powers of 2. The only difference with respect to the default is that the mip-map level selection must be set so that it is based on the speed of the texture positions $f_{I3}$ in $\mathbb{T}^3$, rather than the final texture position in $T_2$.

In contrast, bilinear interpolation cannot be performed as usual, because the interpolation would mix texels that belong to different squarelets whenever the border of a squarelet is accessed. However, we can still run the code multiple times and manually interpolate the fetched texels in the fragment shader. In this way, bilinear interpolation can be performed without adding any texels at squarelet borders, because the fragment shader "knows" about the patch boundaries. This method requires to turn off the automatic bilinear interpolation, which can be done in current fragment languages as they explicitly allow this kind of "do-it-yourself" interpolation.

The complete scheme costs 4·54 instructions for computing the 2D texture positions plus 4 for the texture fetches and 3 for the bilinear interpolation itself. But it is possible to compromise between quality and efficiency by using an anti-aliasing schemes that accesses and interpolates a smaller number of texels. Of course, each texture fetch can be mip-mapped automatically.

## 6 Construction of a PolyCube-Map

So far we described the *mechanism* of PolyCube-Maps and we will now sketch a method that can be used for the *construction* of a PolyCube-Map for a given triangle mesh. We used this semi-automatic technique to produce all the examples that are shown in this paper.

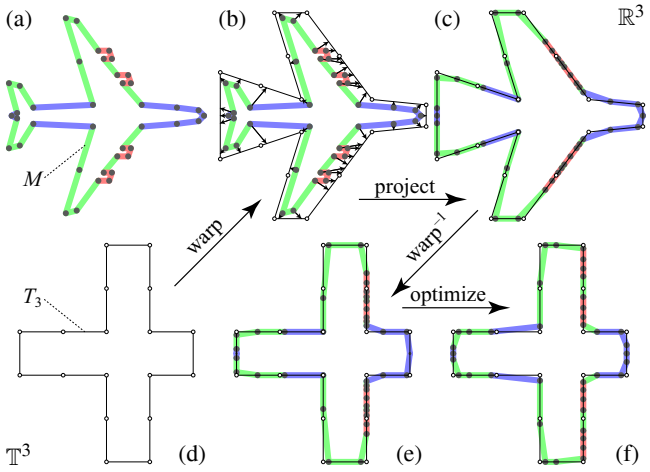


Figure 7: The 2D analogue of our technique to assign 3D texture positions to the vertices of a mesh (a): we first warp the polycube (d) close to the mesh (b), then we project the vertices in the normal direction onto the warped polycube surface (c), warp the result back (e), and finally optimize the texture positions. The meshes in the top row are in $\mathbb{R}^3$ while those in the bottom row are in $\mathbb{T}^3$.

### 6.1 Construction of the poly-cubic parameterization

The first step is to assign to every vertex $v$ of the given mesh $M$ a 3D texture position $v_{T3} \in T_3$. As we have seen in Section 2, this can be done by a simple central projection if the mesh has a sphere-like shape and $T_3$ is a surrounding cube. For more complex meshes we propose the following procedure that is illustrated in Figure 7.

We start by defining a polycube that has roughly the same shape as $M$ and captures all the large scale features. For example, the polycube for the bunny has two stacks of 4 cubes that resemble the ears, a $3\times3\times4$ block for the head, and so on (see Figure 10).

Next we warp the surface $T_3$ of the polycube from its axis-aligned position in the 3D texture space $\mathbb{T}^3$ to the object space $\mathbb{R}^3$. We manually move its vertices close to the mesh and take care that the large scale features of the warped polycube surface and those of the mesh are roughly aligned. For some meshes a simple scaling, rotation, and translation of the polycube surface can serve as a warp function. For example, this was the case for the Laurana and the 3-holes object (see Figure 10).

Then we establish a correspondence between both surfaces by moving every vertex $v$ of $M$ along the surface normal direction onto the deformed polycube. This projection may generate fold-overs, mostly in regions with small scale features, but before we attend to this matter, we apply the inverse warp function to the projected vertices and map them to $T_3$.

These initial 3D texture positions $v_{T3}$ usually do not define a good parameterization, in other words, the piecewise linear function that maps each triangle of $M$ to the corresponding parameter triangle in $\mathbb{T}^3$ deforms the shape of the triangles considerably and may not even be one-to-one in some parts. We therefore implemented a simple iterative procedure to optimize the texture positions and to minimize the overall distortion of the parameterization.

For each vertex $v$ we consider the local mapping between the one-ring of $v$ in $M$ and the one-ring of $v_{T3}$ in $\mathbb{T}^3$. Then we compute the gradient of the deformation energy of the local mapping with respect to $v_{T3}$ and a simple one-dimensional line-search along this direction gives us a new position $v'_{T3}$. If the one-ring of $v_{T3}$ is not flat, then $v'_{T3}$ may not lie on $T_3$ and we use the projection $\mathscr{P}$ to map it back onto $T_3$ in that case. By iterating over the vertices and applying these *local* optimizations, we successively improve the quality of the poly-cubic parameterization *globally*.

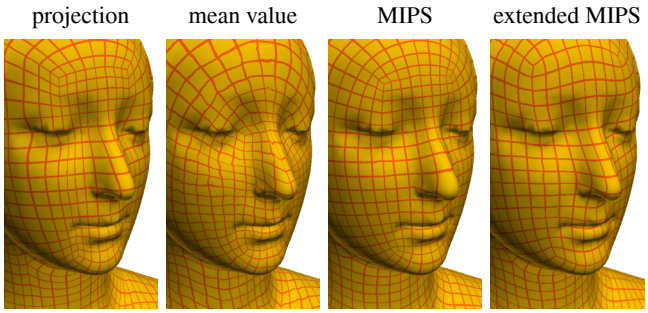


Figure 8: Optimizing the initial parameterization that we created by projection (left) using different techniques: the mean value coordinates and the MIPS method (middle) tend to produce conformal maps at the expense of global area distortion, while the extended MIPS method (right) nicely balances angle and area deformations.

In our experiments we tested three different deformation energies and Figure 8 shows a comparison of the results. The first one is based on the *mean value coordinates* [Floater 2003] and has the advantage of leading to a linear local optimization problem. This method has also been used in [Khodakovsky et al. 2003] and tends to give conformal parameterizations. The same holds for the non-linear MIPS energy [Hormann and Greiner 2000] but both results are not well-suited for our purposes as the area deformation can be quite large. We therefore prefer to use an extension of the MIPS method that was presented by Degener et al. [2003]. It allows to mediate between the conformality and the area-preservation of the parameterization by choosing a weighting parameter $\theta$ and we found $\theta = 3$ to give very good results in all our examples. For more detailed information on parameterizations we refer the interested reader to the recent survey by Floater and Hormann [2004].

### 6.2 Construction of the look-up table

Once the polycube has been specified, the construction of $T_3^{\mathrm{LUT}}$ is simple. For each cell that contains a vertex of the polycube we assign the basic case $e.C$ and a rotation $e.R$ to the corresponding entry $e$ in $T_3^{\mathrm{LUT}}$. Both are fully determined by the arrangement of the eight cubes incident to that vertex and we can easily precompute the 63 possible configurations that can occur around the vertices of a two-manifold polycube surface.

We further assign the global offset $e.O$ for all non-empty cells, thus defining the global packing of the texture patches inside $T_2$. Even very simple heuristics deal well with this simple packing. For example, we can iteratively assign patches to the first available place, scanning $T_2$ from left to right and from top to bottom (see Figures 9 and 12).

Of course, we cannot guarantee to cover the entire $2^N \times 2^M$ texture space $T_2$, but this is a minor problem. Whenever we need to keep textures for multiple models in the texture memory, they can be packed in the same global texture map so that only a small fraction of texture space will be left unused in the end.

## 7 Experimental Results

To test the potential of PolyCube-Maps, we produced a few examples (see Figures 10 and 12) with the method described in Section 6. The texel values of the texture map $T_2$ have been filled using either a regular pattern or the shading of the mesh at highest resolution as in [Cignoni et al. 1999]. Note that our method is not limited to closed surfaces. In fact, the Laurana model is open at the bottom and we can still use PolyCube-Maps as long as the polycube surface is open at the bottom, too (see Figure 11).

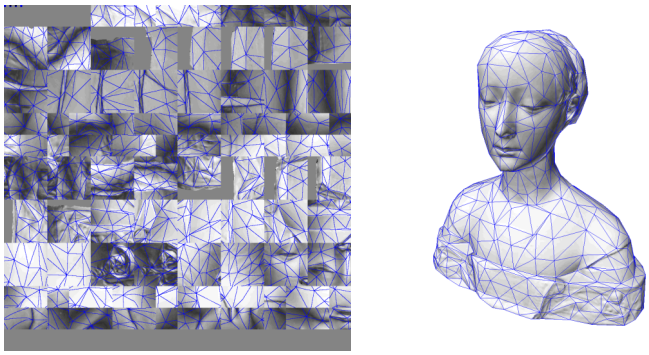

Figure 9: The packing of texture patches is almost perfect and it can be seen that triangles are allowed to span across multiple squarelets.

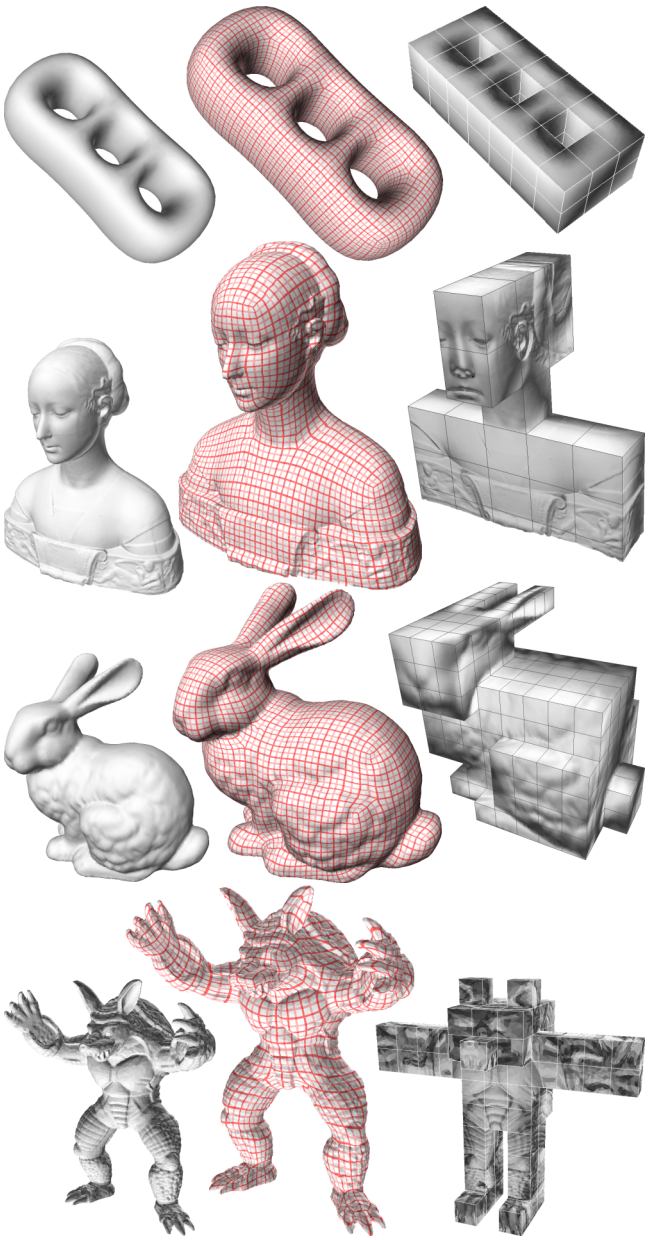

Figure 10: Examples of models with poly-cubic parameterizations: the original model (left), using the PolyCube-Map to texture it with a regular grid (middle), and shaded parameterization of the mesh over the polycube surface (right).

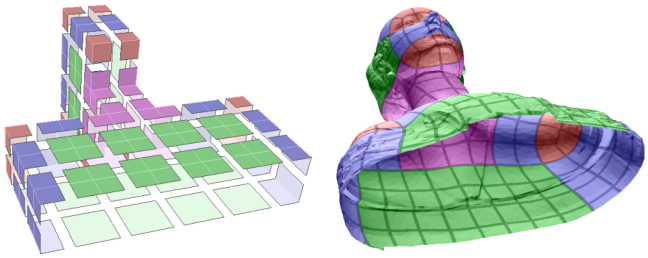

Figure 11: Our method also allows to handle open meshes as long as the polycube captures this feature and is open, too. The facelets of the polycube and the corresponding parts on the mesh are coloured according to the cell configuration.

| | distortion | | stretch efficiency | | | |
|---|---|---|---|---|---|---|
| | area | angle | $T_3 \rightarrow M$ | $M \rightarrow T_3$ | [Gu] | [Praun] |
| 3-holes | 1.003 | 1.011 | 0.986 | 0.986 | | |
| Laurana | 1.141 | 1.101 | 0.795 | 0.745 | | |
| bunny | 1.034 | 1.069 | 0.892 | 0.913 | 0.639 | 0.703 |
| armadillo | 1.224 | 1.318 | 0.616 | 0.577 | 0.607 | 0.465 |

Table 1: Distortion of the poly-cubic parameterizations $\phi$ from Figure 10. Area and angle distortions are measured by integrating and normalizing the values $\sigma_1\sigma_2 + 1/(\sigma_1\sigma_2)$ and $\sigma_1/\sigma_2 + \sigma_2/\sigma_1$, respectively, where $\sigma_1$ and $\sigma_2$ are the singular values of the Jacobian matrix $J_\phi$ (see [Degener et al. 2003] and [Floater and Hormann 2004] for details). The stretch efficiency is computed as in [Praun and Hoppe 2003]. For all measures, the optimal value is 1.

Table 1 lists several distortion measures that document the quality of the underlying poly-cubic parameterizations and compares it with the results of Gu et al. [2002] and Praun and Hoppe [2003]. It can be seen that using a polycube surface as parameterization domain instead of a flat domain (Gu et al. use a square) or simple spherical shapes (Praun and Hoppe use platonic solids) helps to reduce the overall distortion. This is not surprising because the polycube has a shape similar to that of the mesh.

**Mesh independence.** We can also apply a single PolyCube-Map to several simplified versions of a given mesh as shown in Figure 12. A key property of our method is that the simplification of the original mesh does not have to take the PolyCube-Map into account and vice versa. In other words, none of the atomic simplification operations (e.g. edge collapse) has to be forbidden because of the texture parameterization, and on the other hand, the definition of the texture $T_2$ and the look-up table $T_3^{\text{LUT}}$ are both independent of the simplification process.

Apart from such simplified versions, the same PolyCube-Map would also work with other models that share the general shape of the original one, including multi-resolution structures and remeshes (even quadrilateral ones), as long as an appropriate 3D texture position is defined for every vertex (see Figure 3). In our example we assigned to each vertex of the simplified meshes the 3D texture position of the closest point (in the normal direction) on the surface of the original model which in turn is defined by linear interpolation of the texture coordinates at the corners of the containing triangle.

**Rendering performance.** Texture mapping with PolyCube-Maps can slow down a fill-limited application because of the longer fragment shader required. Still we never experienced the frame rate to drop below 30 fps with mip-mapping and without bilinear interpolation or 10 fps with both turned on, even when most of the screen was covered. Tests were performed with an *nVIDIA GeForce FX 5600* and an *ATI Radeon 9800 Pro* on a *Pentium 4* with 2.4 Ghz.

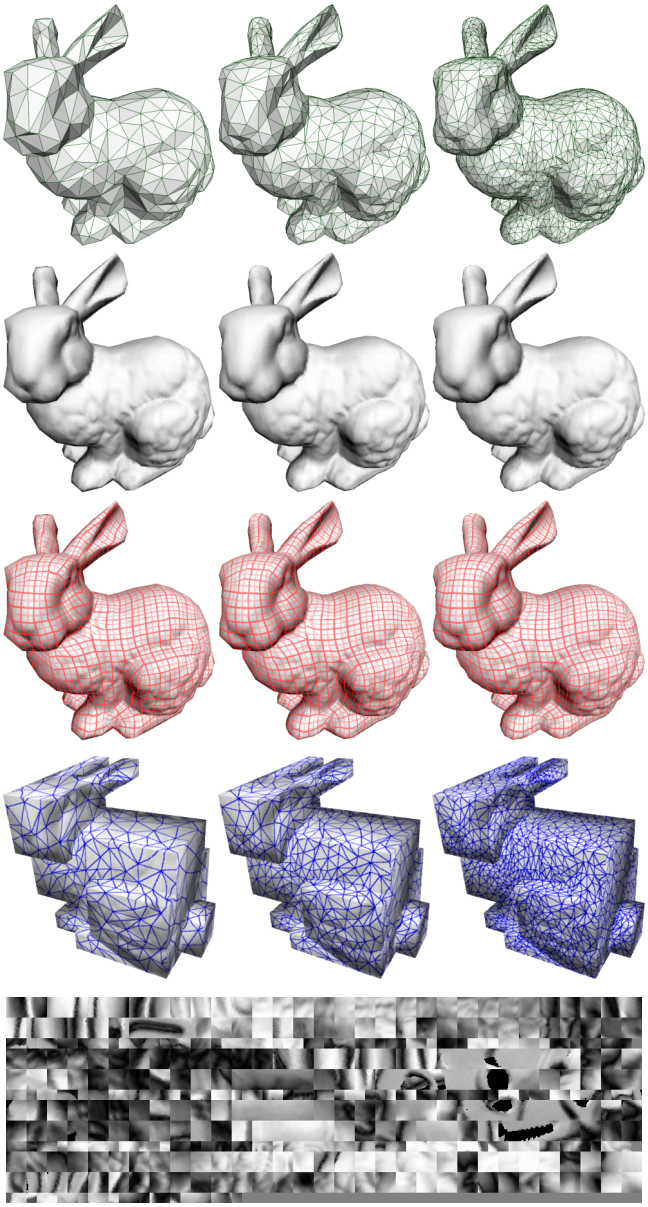

Figure 12: A single PolyCube-Map can be used to texture map different simplified versions (top) with the shading of the given mesh or a regular pattern (middle). The corresponding poly-cubic parameterizations and the texture $T_2$ are shown at the bottom.

## 8 Conclusions and Discussion

PolyCube-Maps provide a new mechanism for texture mapping general 3D meshes. Once an appropriate polycube is chosen and a poly-cubic parameterization of the given mesh over the polycube surface is computed, it can be used to seamlessly map colour or other signals onto that mesh and other models with a similar shape.

If each squarelet is considered a chart, then a PolyCube-Map can be seen as a multi-chart method, but with one important difference. Although the final 2D texture coordinates, if seen as a function that is defined over the mesh surface, are discontinuous at the chart boundaries, these discontinuities are not visible because the underlying 3D texture coordinates are continuous. And since they are dealt with on a per-fragment basis that is hidden from the user, PolyCube-Maps have all the advantages of a seamless mapping that

were discussed in the introduction, including mip-mapping. Moreover, PolyCube-Maps are mesh independent and the packing of texture patches is trivial because of their simple rectangular shape and it causes almost no wasted texture space.

### 8.1 Limits of PolyCube-Maps

Obviously, PolyCube-Maps also have limits in the scope of their applicability. If the geometry or topology of a mesh is too complex or has features at very different scales, then an appropriate polycube would consist of so many cubes that the size of the corresponding $T_3^{\mathrm{LUT}}$ would soon exceed the texture memory. An extreme example would be a model of a tree with trunk, branches, and leaves.

In Section 7 we showed that a single PolyCube-Map can be used for texture mapping different representations of the same object. However, if such a representation deviates too much from the original mesh for which the PolyCube-Map was constructed, then it can happen that the texture position of a produced fragment falls in an empty cell, which results in a visible rendering artefact.

### 8.2 Geometry images

PolyCube-Maps represent a special type of parameterization that has been designed for texture mapping, but it can also be used for remeshing and storing purposes, becoming a variant of *geometry images* [Gu et al. 2002]. Each final texel in $T_2$ can be used to store a sample of the mesh by mapping the coordinates $(x,y,z)$ to the $(r,g,b)$ channels. The pair of textures $T_2$ and $T_3^{\mathrm{LUT}}$ can then be seen as a stand-alone representation of the original model.

In particular, each squarelet in $T_2$ would encode a subpart of the encoded surface with a trivially defined connectivity (each group of four adjacent texels forms a quad). The information that is stored in $T_3^{\mathrm{LUT}}$ would then be used to zipper the subparts into a single coherent mesh. This would be easy because the sides of neighbouring subparts have the same number of points.

### 8.3 Future work

**Extensions.** As soon as the GPU programmability has advanced so that the number of fragment program instructions is less critical, it will be possible to include the cell cases that we left out of the current implementation. Any two-manifold polycube surface can then be used as 3D texture domain $T_3$. Another useful extension would be some hierarchical approach (e.g. with octrees) for the subdivision of the texture space $\mathbb{T}^3$, taking care not to require too many additional accesses to the 3D look-up table $T_3^{\mathrm{LUT}}$.

**Tiled textures.** PolyCube-Maps have the potential for a new type of tiled textures. For each cell configuration we could create and store 2D texture patches with matching boundaries, and the 3D look-up table would then be used to seamlessly map them onto the polycube surface in a sort of two-manifold Wang Tiles scheme [Cohen et al. 2003]. The result would be similar to lapped textures [Praun et al. 2000] but without the need to use a resampled texture atlas or to render triangles multiple times with alpha blending.

**Automatic parameterization.** We believe that this work opens the way to a new category of surface parameterization methods that parameterize a mesh over a polycube surface instead of a flat domain or a coarse simplicial complex. The most challenging part will be to determine appropriate polycubes with minimal or no user intervention, but also adapting existing parameterization methods to this new kind of domain as well as speeding them up with hierarchical methods will be a worthwhile task.


### Acknowledgements

This work was supported by the projects *ViHAP3D* (EU IST-2001-32641) and *MACROGeo* (FIRB-MIUR RBAU01MZJ5) and by the Deutsche Forschungsgemeinschaft (DFG HO 2457/1-1).


## References


CARR, N. A., AND HART, J. C. 2002. Meshed atlases for real-time procedural solid texturing. *ACM Transactions on Graphics 21*, 2, 106–131.

CIGNONI, P., MONTANI, C., ROCCHINI, C., SCOPIGNO, R., AND TARINI, M. 1999. Preserving attribute values on simplified meshes by resampling detail textures. *The Visual Computer 15*, 10, 519–539.

COHEN, J., OLANO, M., AND MANOCHA, D. 1998. Appearance-preserving simplification. In *Proc. of ACM SIGGRAPH 98*, 115–122.

COHEN, M. F., SHADE, J., HILLER, S., AND DEUSSEN, O. 2003. Wang Tiles for image and texture generation. *ACM Transactions on Graphics 22*, 3, 287–294.

DEGENER, P., MESETH, J., AND KLEIN, R. 2003. An adaptable surface parameterization method. In *Proc. of the 12th International Meshing Roundtable*, 201–213.

ECK, M., DEROSE, T., DUCHAMP, T., HOPPE, H., LOUNSBERY, M., AND STUETZLE, W. 1995. Multiresolution analysis of arbitrary meshes. In *Proc. of ACM SIGGRAPH 95*, 173–182.

FLOATER, M. S., AND HORMANN, K. 2004. Surface parameterization: a tutorial and survey. In *Advances in Multiresolution for Geometric Modelling*, N. A. Dodgson, M. S. Floater, and M. A. Sabin, Eds. Springer, 259–284.

FLOATER, M. S. 2003. Mean value coordinates. *Computer Aided Geometric Design 20*, 1, 19–27.

GRIMM, C. M. 2002. Simple manifolds for surface modeling and parameterization. In *Proc. of Shape Modeling International 2002*, 237–244.

GU, X., GORTLER, S. J., AND HOPPE, H. 2002. Geometry images. *ACM Transactions on Graphics 21*, 3, 355–361.

HORMANN, K., AND GREINER, G. 2000. MIPS: An efficient global parametrization method. In *Curve and Surface Design: Saint-Malo 1999*, P.-J. Laurent, P. Sablonnière, and L. L. Schumaker, Eds. Vanderbilt University Press, 153–162.

KHODAKOVSKY, A., LITKE, N., AND SCHRÖDER, P. 2003. Globally smooth parameterizations with low distortion. *ACM Transactions on Graphics 22*, 3, 350–357.

LEE, A. W. F., SWELDENS, W., SCHRÖDER, P., COWSAR, L., AND DOBKIN, D. 1998. MAPS: multiresolution adaptive parameterization of surfaces. In *Proc. of ACM SIGGRAPH 98*, 95–104.

LÉVY, B., PETITJEAN, S., RAY, N., AND MAILLOT, J. 2002. Least squares conformal maps for automatic texture atlas generation. *ACM Transactions on Graphics 21*, 3, 362–371.

MAILLOT, J., YAHIA, H., AND VERROUST, A. 1993. Interactive texture mapping. In *Proc. of ACM SIGGRAPH 93*, 27–34.

PIPONI, D., AND BORSHUKOV, G. 2000. Seamless texture mapping of subdivision surfaces by model pelting and texture blending. In *Proc. of ACM SIGGRAPH 2000*, 471–478.

PRAUN, E., AND HOPPE, H. 2003. Spherical parametrization and remeshing. *ACM Transactions on Graphics 22*, 3, 340–349.

PRAUN, E., FINKELSTEIN, A., AND HOPPE, H. 2000. Lapped textures. In *Proc. of ACM SIGGRAPH 2000*, 465–470.

SANDER, P., WOOD, Z., GORTLER, S. J., SNYDER, J., AND HOPPE, H. 2003. Multi-chart geometry images. In *Proc. of the Symposium on Geometry Processing 2003*, 146–155.

SCHNEIDER, J., AND WESTERMANN, R. 2003. Compression domain volume rendering. In *Proc. of Visualization 2003*, 293–300.

SHEFFER, A., AND HART, J. C. 2002. Seamster: inconspicuous low-distortion texture seam layout. In *Proc. of Visualization 2002*, 291–298.

SORKINE, O., COHEN-OR, D., GOLDENTHAL, R., AND LISCHINSKI, D. 2002. Bounded-distortion piecewise mesh parameterization. In *Proc. of Visualization 2002*, 355–362.