Eurographics Symposium on Geometry Processing (2004)
R. Scopigno, D. Zorin, (Editors)

Seamless Texture Atlases

Budirijanto Purnomo, Jonathan D. Cohen and Subodh Kumar

Department of Computer Science, Johns Hopkins University, USA

Abstract

Texture atlas parameterization provides an effective way to map a variety of color and data attributes from 2D texture domains onto polygonal surface meshes. However, the individual charts of such atlases are typically plagued by noticeable seams. We describe a new type of atlas which is seamless by construction. Our seamless atlas comprises all quadrilateral charts, and permits seamless texturing, as well as per-fragment down-sampling on rendering hardware and polygon simplification. We demonstrate the use of this atlas for capturing appearance attributes and producing seamless renderings.

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Curve, Surface, Solid and Object Representations , I.3.7 [Computer Graphics]: Color, Shading, Shadowing and Texture

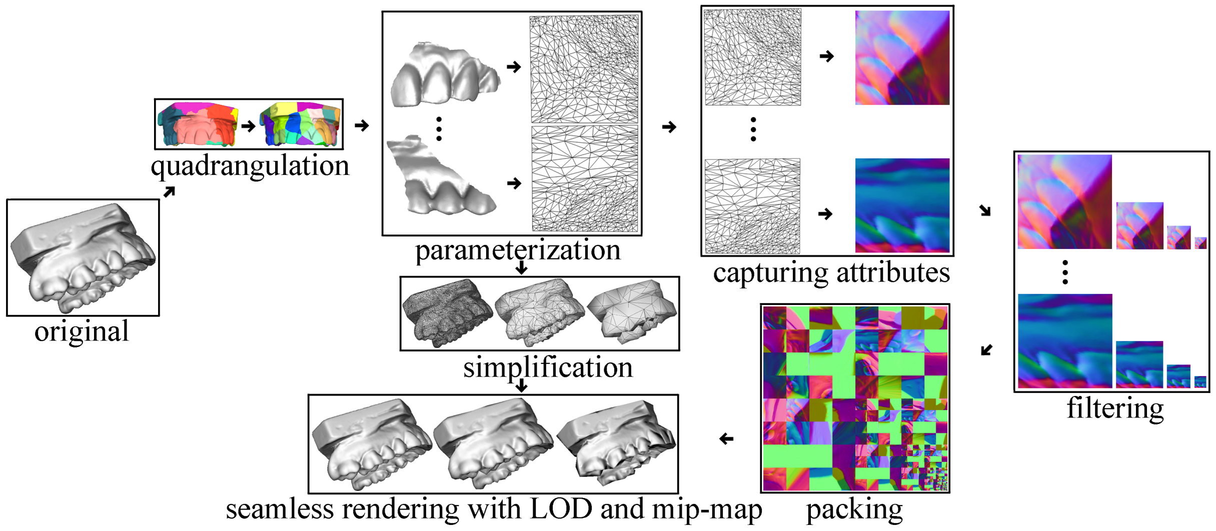

Figure 1: *Seamless atlas pipeline*

1. Introduction

The application of color image maps to 3D geometry via texture mapping has been used to enhance the appearance of rendered images for three decades. Over that time period, the importance and ubiquity of texture mapping has continually increased. Fast texture mapping has become a mainstay of graphics hardware, and texture images now convey a variety of data to procedural shaders executing on fragment processors.

Unfortunately, establishing a parameterization of a general 3D surface into a 2D texture domain remains a challenging problem. An increasingly popular technique is to create an atlas. This atlas comprises a set of charts, each of which maps a connected part of the 3D surface (a patch) onto a piece of the 2D texture domain. This piecewise approach allows a reasonable degree of local control in constructing the mappings. However, there are several opportunities here for the introduction of seams (0-order discontinuities) between the neighboring patches.

We take the position that the best way to eliminate the appearance of seams on textured surfaces is to design the atlas structure with seam avoidance in mind from the very start. To achieve this end, we introduce the notion of a seamless atlas. The seamless atlas is inspired by OpenGL texture borders [WNDS99] and the seamless bricking techniques used by applications performing 3D texturing of large data [LHJ99]. The atlas comprises a number of quadrilateral patches, mapped to square (or rectangular) charts in texture space. These charts may be trivially packed into a single texture map in a variety of ways.

In this paper we demonstrate that this relatively simple structure of our seamless atlas permits all of the following:

- **Seamless texturing:** Rendering using our new atlas produces no seam artifacts.
- **Downsampling:** Seamless atlas textures may be easily downsampled, producing lower resolution texture that is still seamless as well as stationary.
- **Mip-mapping:** Texture resolutions may be selected or blended on a per-fragment basis.
- **Geometry simplification:** The simple structure of the seamless atlas makes it straightforward to downsample the geometry while still retaining the seamless property.

In this paper, we describe algorithms for creating and rendering using a seamless atlas. We demonstrate the use of seamless atlases for resampling vertex colors, vertex normals, and per-triangle textures and producing properly filtered renderings. Figure 1 shows our system pipeline.

2. Related Work

One way to generate an atlas is to make a simple chart for each triangle. Often, the mesh is first simplified, and then these base triangles are used to construct the charts [CMR*99, LSS*98, SGR96]. The simplified triangles are then packed into texture space and sampled to generate texture maps. When such texture maps are rendered, seams may appear between triangles due to bilinear interpolation between adjacently packed triangle charts and do not easily allow per-fragment mip-mapping.

Alternatively, triangles may be clustered into patches which are then parameterized as charts [EDD*95, SSGH01, SWG*03]. By parameterizing into convex polygons with small numbers of sides, it is possible to simplify the patches without disturbing the parameterizations along the boundaries [COM98, SSGH01]. However, if the matching boundary edges differ in length or orientation in the texture domain, it is still difficult to eliminate subtle seams along the boundaries (even if a one texel padding is applied just outside the charts).

We find that eliminating seams is straightforward if the atlas comprises quadrilateral patches mapped to square or rectangular charts. Such atlases have been designed using manual intervention for decorating implicit surfaces [Ped95] and for converting meshes to spline patches [KL96]. A quadrilateral atlas can be created automatically by merging adjacent triangular charts [EH96].

Recognizing that seams are an important problem with atlases, various approaches have been developed to minimize their effect. For example, the seams may be forced into regions of high negative curvature [LPRM02, SH02] and thus made less apparent. As an alternative, an image fidelity metric [ZMT04] can be used to minimize the visual effect of seams, along with other error sources.

Atlases used for geometry images, either single-chart [GGH02] or multi-chart [SWG*03] are interesting cases, because these geometry images are water tight (i.e. seamless) by design. However, although it is possible to downsample these images to lower resolutions, special border treatments at each resolution make the mappings between boundary texels at various resolutions challenging. Hence, it is difficult to mip-map these structures. The elegant boundary structure of the more specialized spherically parameterized geometry images [PH03] may be more conducive to per-fragment mip-mapping.

Recent work in the domain of procedural solid texturing has produced a multi-resolution texture atlas [CH02], which uses standard mip-mapping on graphics hardware. This texture atlas has several desirable properties, including excellent control of the sampling rate across the surface and efficient use of the entire texture space. However, the packing and mip-mapping scheme used still generates seams between charts except at the highest-resolution mip-map level.

Thus the appearance of seams in general atlas parameterizations has remained a persistent, open problem that affects both model representations and rendering. We address this problem in the sections that follow, beginning with the design of the seamless atlas.

3. Seamless Atlas

The seamless atlas comprises a set of quadrilateral patches that are mapped to rectangles in texture space. We want to sample data into these texture rectangles and also downsample the data into mip-maps in such a way that each mip-map level may be rendered seamlessly on the polygonal surface. Texture seams have two sources:

- interpolation between boundary texels of two unrelated patches
- discontinuity in the texel colors between two adjacent patches on the surface

In addition to being seamless, the texture should remain stationary on the surface across mip-map levels. This means that we must ensure that a given texture coordinate refers to precisely the same point on the chart at all resolutions. As it turns out, OpenGL (and the hardware it abstracts) is designed to accomplish just this, so we begin with a look at the approach probably intended by the OpenGL architects.

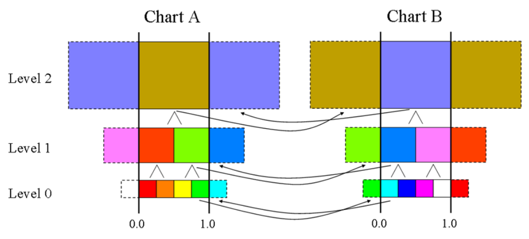

Figure 2: *Two charts represented as OpenGL texture maps with borders (depicted for 1D textures).*

3.1. OpenGL Seamless Texturing - Separate Style

In OpenGL, the data value stored at a texel is associated with the center of that texel. Consider the example of a 1D texture in Figure 2. When using linear texture filtering (bilinear for 2D), the red color at level 0 of chart A is the sample at the texture coordinate 1/8, the center of the leftmost texel. To eliminate a seam between chart A and chart B (which lie next to each other on some surface), the value at texture coordinate 1.0 on chart A must exactly match that at texture coordinate 0.0 on chart B. But these values must be reconstructed on each chart. For example, on chart B, the value is constructed by interpolating between the first texel and some other value deemed to be at sampled at coordinate -1/8.

This can be accomplished in OpenGL by binding a different texture for each chart of our seamless atlas and using the texture border mechanism to provide the correct texel value at -1/8. The texture border provides storage for exactly one additional texel just outside the [0,1] range of the texture domain at every mip-map level. The border texel controls the interpolation at all fragments with texture coordinates contained in the extremal half texel of the [0,1] range (e.g. from 0 to 1/8 and from 7/8 to 1 in our 4-texel image). In our example, a cyan texel would be placed at the right border of chart A and a green texel will be placed at the left border of chart B (marked with dashed lines). Thus, the boundary between chart A and chart B at level 0 will interpolate to the average of green and cyan on both charts. Each mip-map level in this example is generated from the previous using a 2-texel-wide box filter and border texels are again replicated from the neighboring chart at each level.

A seamless atlas created using our algorithms (see Section 4) may be stored in this form and rendered in this fashion, a style we call *separate* (because the patch textures are stored and loaded separately). In addition to being the simplest seamless atlas style (see Figure 3), it also easily accommodates using a different maximum resolution for each texture and provides access to any advanced filters (e.g. anisotropic filtering on each mip-map level, trilinear filtering across mip-map levels, etc.) supported by the hardware and driver. This can be useful if charts have very different areas on the surface or if their actual data have different spatial frequencies.

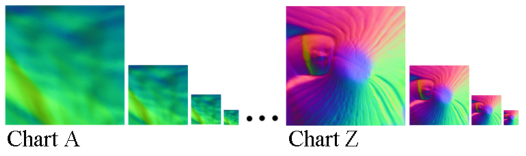

Figure 3: *Separate seamless atlas. Each chart is shown with its mip-map.*

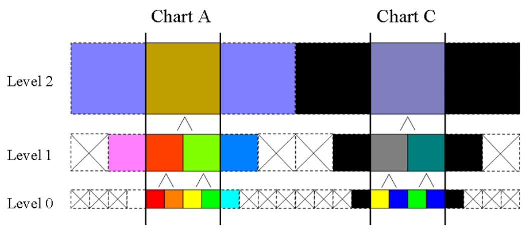

Figure 4: *Two charts naively arranged into a single texture. Wasted texels are marked with 'x'.*

However, separate style can have negative performance implications. Binding a new texture generally causes a flush of the fragment pipeline, and each patch must be issued as a separate draw call. Managing multiple textures is also sometimes an unnecessary burden for the application programmer and may cause fragmentation of texture memory. In addition, some drivers may not be optimized for texture borders, and the use of separate textures typically restricts the texture size to a power of two.

3.2. Combining Charts - Stacked Style

Although it is often desirable to combine all levels of all charts into a single master texture, it is non-trivial if we wish to maintain both the seamless and stationary properties. Consider arranging chart A next to some chart C in the texture and assume that patch A does not neighbor patch C on the surface.

As before, both charts still require padding by "border" texels to guarantee seamless interpolation with their neighboring charts on the surface. However, to leave a chart's texture coordinates stationary across levels, we may never reuse the texture space occupied by any border texel (i.e., at any level). At the lowest resolution of 1x1, a texel occupies space equal to the whole chart (shown at level 2 in Figure 4). In 2D, this corresponds to wastage of about 75% of the texels including all levels of the texture map (shown with X through them in the figure). Here we do not have the benefit of the convenient, single-texel padding provided by the hardware around the border of entire texture map.

Instead, we suggest a different sampling and filtering scheme, inspired by strategies used for "bricking" and filtering of large 3D texture data [LHJ99, LCCK02]. Rather than keeping perfect alignment in texture space across texture levels, we will condense the data more and more in suc-

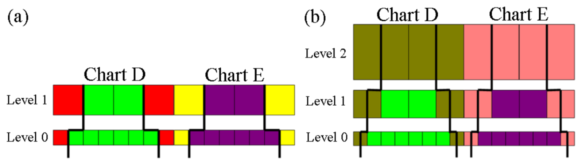

Figure 5: *(a) Two charts arranged in a single texture. Red and yellow texels are replicated from appropriated chart neighbors. A chart's texture coordinates now depend on the mip-map level. (b) Two charts packed with only 1/2 texel of border.*

cessive levels (see Figure 5a). In this case we will keep the texture stationary on the surface by automatically adjusting the texture coordinates according to the mip-map level. In this figure, we have only one border texel (of the appropriate resolution) next to each chart at every level. So, for example, the texture coordinates for level 0 of chart D are scaled to lie in [1/8, 7/8], whereas level 1 lies in [1/4, 3/4]. Just like the separate style, this wastes half of each border texel (with the other half used for bilinear interpolation). We can eliminate even that waste by effectively using only half a texel of border on each side as shown in Figure 5b.

This arrangement implies a somewhat simpler sampling scheme as well, in which we generate the boundary texel by sampling a patch exactly on its boundary. (Notice that the modified sampling pattern has changed the outermost texel values in chart D to the average of red and green, and the values in chart E to the average of yellow and purple.) Chart D's range on level 0 using this alternative sampling scheme is [1/16, 15/16] and on level 1 is [1/8, 7/8]. An example of stacked seamless atlas is shown in Figure 6.

To use this style, we first parameterize a patch on the [0,1] domain, as in the separate style, but at rasterization time transform the texture coordinates of each fragment appropriately for the desired mip-map level. However, it is essential that the hardware's texture lookup then use the mip-map level expected after the texture coordinate adjustment. Fortunately, this apparently recursive equation has a closed form solution, which we present in Section 6.

Ideally, the graphics hardware would natively compute the mip-map texture value based on our equation, making this

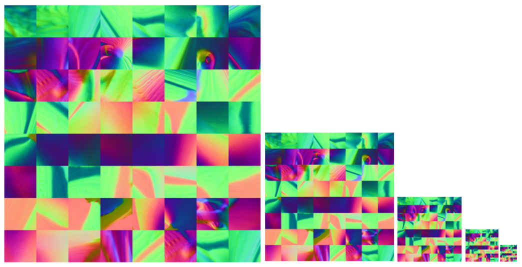

Figure 6: *Stacked seamless atlas for Isis model.*

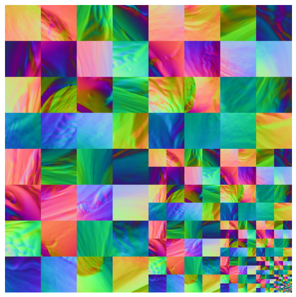

Figure 7: *Flat seamless atlas for Igea model.*

style appropriate. Meanwhile, we can implement this in a fragment program if we know the vendor's mip-map level computation formula. In order to avoid that requirement, we propose the next style.

3.3. Combining Levels - Flat Style

The texture sampling in flat style is identical to that in stacked style. The primary difference here is that we ourselves arrange all the resolutions of all the charts in a single texture image, avoiding the ambiguities inherent in guessing the details of a particular hardware vendor's mip-map level computation.

We lay out all the 2D charts into the texture in a standard mip-map pattern, with the lower-resolution images packed in the bottom quadrant of the image. Given texture coordinates for the level 0 image, it is straightforward to compute in the fragment program the proper texture coordinates for the higher mip-map levels. Figure 7 shows an example of a flat seamless atlas.

3.4. Varying Chart Resolution - Packed Style

One limitation of the stacked and flat styles is that they both make use of the fact that all charts have the same maximum resolution to compute the texture coordinates for a particular level given the texture coordinates at level 0. However, there are many circumstances when we might like to keep some charts at higher resolutions than others. For example, depending on the chart creation process used, some charts may map to a significantly larger area than other charts. It's also possible that certain charts map to regions with more detail as measured by some metric. The packed style trades some simplicity for the space savings of a more adaptive sampling rate. It also allows rectangular charts.

To arrange the charts into a texture, we can use a simple packing algorithm [SGR96]. A texture width is chosen to be a power of 2. We fill this texture from left to right, inserting the charts in order of decreasing size. When all the charts have been placed, we choose the texture height to be just large enough to contain these textures. Because we do not require hardware mip-mapping for this layout, the height does not have to be a power of 2. Figure 8 shows an example of such a layout. In this example, we kept only one third

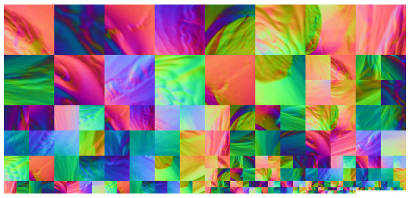

Figure 8: *Packed seamless atlas for Igea model.*

	Separate	Stacked	Flat	Packed
Texture binds	Many	Single	Single	Single
Draw calls	Many	Single	Single	Single
Chart locations	None	Implicit	Implicit	Lookup tbl
Advanced texture filtering	Built-in	Built-in	Fragment program	Fragment program
Mip-map level	Built-in	Fragment program	Fragment program	Fragment program
Texture space	Optimal (tex border)	Possible wastage	Possible wastage	Efficient (rect tex)

Table 1: *Summarizing the different layout styles of using a seamless atlas.*

of the textures at the highest resolution, reducing the texture requirements to one half that for a stacked or flat texture with the same maximum resolution. The locations of the various resolution charts are stored in a lookup table, indexed by chart number and mip-map level. Depending on the number of charts, this table may be stored in either the uniform parameters to the fragment program or as another texture map. To match the colors along boundaries between charts of different maximum resolutions, the higher-resolution boundary is effectively downsampled to match the interpolated colors of the lower-resolution boundary. Table 1 summarizes the different trade-offs in the four layout styles of using a seamless atlas.

4. Creating a Seamless Atlas

We create the atlas by first clustering polygons into arbitrary-sided patches. Next, we subdivide each such patch into a set of quadrilateral, as illustrated in Figure 9. Such quadrilateral patches usually have low distortion when parameterized on a [0,1]x[0,1] domain. Once the patches are determined, we sample its attributes into a texture, and perform filtering to produce several mip-map levels.

4.1. Creating Polygonal Patches

Clustering triangles into patches is a well-known problem. We use a technique which ensures that the patches do not deviate much from a plane [GWH01]. We follow a bottom-up construction, by starting with each triangle as a patch. The patches are merged greedily minimizing the following metric $k_1 Q + k_2 P$, where Q is the quadric error, P is the ratio of the area and the perimeter of a patch.

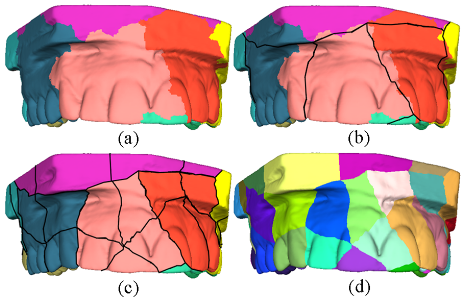

Figure 9: *Teeth model showing the patch creation. (a) Creating patches (Section 4.1). (b) Straightening polygonal patch boundaries. (c) Patch quadrangulation (Section 4.2). (d) The resulting quadrilateral patches.*

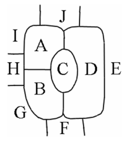

Figure 10: *Merging clusters. The following pair merges are disallowed: A-B, A-D, and B-D. Any other pair could merge without violating our topological rules.*

We extend this work to follow two topological rules during all clustering stages:

1. Every patch is homeomorphic to a disc.
2. Any pair of patches is adjacent at a single vertex, along a single common boundary (homeomorphic to a line), or not at all.

The effect of these rules on applicable merges is shown in Figure 10.

At the end of patch formation process, we straighten patch boundaries [SSGH01] by choosing locally shorter paths (of triangle edges) between patch corners using Djikstra's shortest path algorithm. The edges' weights are equal to their geometric lengths scaled by their distance from the closest point on the original patch boundaries. This ensures that the boundaries do not move too far from their original positions.

4.2. Patch Quadrangulation

We employ a Catmull-Clark-inspired subdivision scheme [CC78] to partition n-sided patches into quadrilateral ones (Figure 11). The algorithm consists of two steps. First, we compute a center point, c, in the polygonal patch. Then, this point is connected by a path to the median of each edge of the polygonal patches. Although this subdivision may be performed on the domain after the

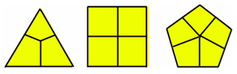

Figure 11: *Simple planar polygon quadrangulation.*

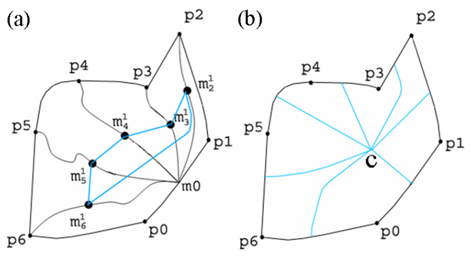

Figure 12: *Quadrangulating a polygonal patch. (a) Finding the center point. The blue polygon lines represent the new polygonal patch after one iteration of the algorithm. The number of polygonal sides reduces by two. (b) The computed center point is connected to median of each boundary curve.*

polygonal patch is parameterized, we choose to directly subdivide the patch and then parameterize each. This turns out to be more efficient and results in somewhat smoother parameterizations.

Center point computation: We find the center point by successive bisection. Consider the patch in Figure 12. Let us choose one of the patch boundaries, say $E_0 = p_0 p_1$. We first find its median m_0. We locate it by summing the lengths of triangle edges on E_0 and choosing the median vertex. We next find the shortest surface path from m_0 to each corner p_i of the patch, $i > 1$. We again use Dijkstra's shortest path algorithm on the surface graph, with the restriction that a path may not exit the patch. We start by assigning a weight of infinity to each edge except those that are incident on the source vertex but are not along the patch boundary. During the path construction, each edge incident to the path found so far is released by assigning to it a weight equal to its edge length. Edges incident on a boundary vertex are not released, thus the paths may not merge.

We perform the following iteration until only one or two vertices remain and hence no more patches can be formed (see Figure 12a):

- Find the shortest paths P_i from m_0 to each p_i
- Find the median vertex m_i^1 along each path P_i
- Connect each pair (circularly) $m_i^1 - m_{i+1}^1$ by the shortest path
- Generate a new patch with two fewer vertices.

If at the end of the iterations two vertices remain, we choose their median (along the shortest path) as c', otherwise we choose the remaining vertex. This procedure converges to the actual centroid for planar polygons, but is geometrically near the center for non-planar patches. In practice, we choose c from the neighborhood of c'. We choose the highest degree vertex in the neighborhood. High degree vertices are usually near surface features and aid the following quadrangulation procedure, thus they form good patch corners.

Subdivision: Once we have found the center c, we join it to the median of each boundary curve of the polygonal patch in the manner shown in Figure 12b. The connecting curve is

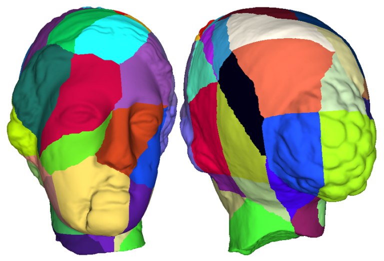

Figure 13: *Igea model partitioned into quadrilateral patches.*

again the shortest path, with a modification. We do not allow these paths to intersect, or even be near. Close paths result in narrow regions on the resulting quadrilaterals resulting in parameterization artifacts. We again use Dijkstra's shortest path algorithm to find these paths in a greedy fashion. We start by finding the shortest path, P_0, from the median of edge E_0 to the center point c as described before. We next find the shortest path, $P_{n/2}$, from the median of the next edge $E_{n/2}$ to c, with the weights of edges scaled by their distance from path P_0. This results in a path that stays far from P_0. We similarly use binary subdivision on each side to determine the order in which to find all paths, P_i. Each path stays far from those found before it. It is possible in some cases for no paths to exist from the center to one of the edges as two paths may have only one route through a vertex v and paths are not allowed to intersect. We subdivide the triangle adjacent to v to create new paths.

Paths P_i subdivide the original n-sided patch into n quads as shown in Figure 12b, with the property that each quad-patch edge has exactly two quads adjacent to it and each quad-patch corner has a ring homeomorphic to a disk. Figure 13 shows the resulting quadrilateral patches on Igea model.

4.3. Parameterization

Since our patches are rather simple, we use an inexpensive parameterization scheme [SSGH01] for each quadrilateral patch. We first distribute the vertices on the four boundary curves of the patch on the $u = 0$, $u = 1$, $v = 0$ and $v = 1$ texture boundaries. Each boundary vertex is placed at a texture coordinate proportional to the ratio of its path length from the corner and the total boundary curve length. This is an arc-length parameterization of the boundary curve. For the internal points, we perform uniform edge-spring parameterization followed by geometric stretch minimization [SSGH01]. Other parameterization algorithms may also be employed here [DMA02, Flo97, ZMT04], and considering multiple patches at a time can permit parameterizations which are smooth across patch boundaries [KLS03].

4.4. Chart Generation

Recall that we generate the "border" texture samples from patch boundaries. We use graphics hardware to create these

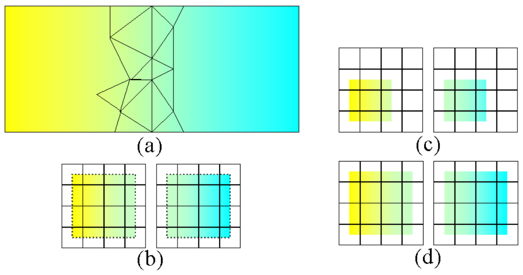

Figure 14: *Sampling by rendering: The grid shows the pixels and the colored polygon is being sampled. (a) Triangles at the boundary of two adjacent patches. (b) The desired samples at the centers of pixels. (c) Obtained samples due to rasterization rules. (d) All samples generated by using lines.*

samples when needed. We draw each triangle using its texture coordinate as geometry. Figure 14 shows the rasterization of a patch. At level 0, the boundary texels are sampled precisely from the patch boundaries. Because pixels are sampled at their centers (Figure 14b), we render the triangles with the coordinates scaled half a pixel in. This ensures that the corner pixel sample lies precisely at the patch boundary.

Due to the rasterization rules [FvDFH90], this causes two window boundaries to not be drawn (Figure 14c). We redraw these boundaries by drawing the triangle edges on the patch boundary also as lines, again shifted half a pixel in. This ensures that the boundary pixels on the window contain samples precisely at the patch boundaries.

4.5. Filtering

Once the highest resolution texture is computed, we filter it down to generate all levels of the mip-map. Since a change of texture coordinates is required for each mip-map level, the filtering has to be performed carefully as well.

We must ensure that the boundary texels at each level of a patch agrees with the corresponding level of the adjacent patch and represents the colors precisely on the boundary. We also must ensure that the samples are taken from the correct place on the triangle with the modified location of the patch boundary. It is possible to simply re-render the patches in the parametric space using a smaller window. However, any filter better than "sub-sampling" requires a careful con-

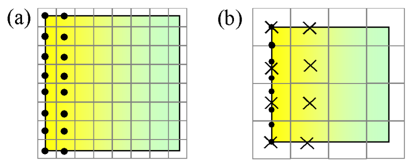

Figure 15: *Texture Filtering: Grid shows the texels; color comes from rendered samples. (a) The original 8x8 samples. (b) From 8x8 samples to 4x4 samples. The dot represents the old 8x8 samples. The cross represents the new 4x4 samples.*

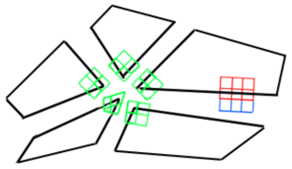

Figure 16: *Convolving on the corner and the boundary of the charts.*

struction. If we wish to generate a 4x4 texture from an 8x8 texture (Figure 15), after moving the patch boundary half a texel inside, the same boundary needs to be sampled four times instead of eight (marked by crosses in Figure 15b). The new locations are not a subset of the old locations. In general the locations of the sample are at $1/(2^l - 1)$ distance apart for texture level l. In order to avoid seams at all levels, the values must be the same on both patches adjacent to a boundary. Hence, these samples must be generated by using a filter that is symmetric about the texture boundary.

We use a two-step filter: first convolve each 3x3 set of samples at level l with a Gaussian filter to obtain samples at the texel centers for level l. Then linearly interpolate these samples to obtain the samples at the texels for level $l+1$. When the Gaussian kernel is applied at a boundary, we gather the missing samples (e.g. the blue samples in Figure 16) from the adjacent patch. Special care is required at the corners, which may connect an arbitrary number of patches. In this case, all neighboring samples are gathered and arranged uniformly around the corner sample for filtering, as shown in Figure 16. This ensures that the corresponding boundary texels contain the same values for adjacent patches and their values are generated from values across the boundary for a smooth filtering.

5. Geometric Simplification

Given our regular atlas structure, performing the geometric simplification is straightforward. We use a priority queue driven, half-edge-collapse based scheme. We impose a few restrictions to maintain the seamless parameterization. Interior vertices may collapse to chart boundary vertices, but not vice versa. Similarly, boundary vertices may collapse to corner vertices, but not vice versa. Corner vertices are never moved. Finally, edge collapses along chart boundaries must be applied to both sides of the boundary. These restrictions are similar to those employed by the appearance-preserving simplification algorithm [COM98], which operated on atlases having similar properties to ours.

6. Rendering With a Seamless Atlas

The rendering requirements for a seamless atlas depend on the chosen style (Section 3). In the separate style, for example, the patches are rendered individually, with each requiring a texture bind and draw call to issue its vertices. The texture coordinates of each patch's vertices lie in [0,1], and no special fragment program is required (except to perform standard normal mapping, etc.).

In contrast, the flat style requires a single texture bind and a single draw call to issue all the vertices of all the patches. In principle, the texture coordinates of each patch could fill the [0,1] range, so long as a patch number is also issued with each vertex. In practice, at the expense of a few bits of texture coordinate precision, we encode the patch number in the texture coordinate, extracting it using div/mod operations in the fragment program. The texture coordinates issued with each vertex are thus the coordinates appropriate for mapping the particular patch at mip-map level 0, so the patch boundary coordinates lie in the center of its outermost texels at that highest resolution. The fragment program then does the following:

1. Scale and bias the texture coordinates so the particular patch fills the entire [0,1] domain
2. Compute the appropriate mip-map level, i, for this fragment
3. Scale and bias the texture coordinates to shrink the boundaries by half a texel in each direction at this mip-map resolution
4. Bias the texture coordinates to lie in the appropriate patch of the appropriate resolution in the full [0,1] texture space

The scale and bias operations themselves are all straightforward. The challenge, then, lies in computing the appropriate mip-map level.

Given a patch with its texture coordinates (u, v) between 0 and 1, the formula for computing the default mip-map level l is

$$l = log\,MAX\left[\sqrt{\left(\frac{\partial u}{\partial x}\right)^2+\left(\frac{\partial v}{\partial x}\right)^2},\sqrt{\left(\frac{\partial u}{\partial y}\right)^2+\left(\frac{\partial v}{\partial y}\right)^2}\right] \quad (1)$$

where (x, y) is the screen coordinates.

If N is the original texture resolution, the texel size n_i at mip-map level i is $2^i/N$. For level i, the modified texture coordinates (s, t) should lie in the range ($n_i/2$, $1-n_i/2$) to avoid a seam. In general $s=(1-n_i)u+n_i/2$ and $t=(1-n_i)v+n_i/2$. This means the derivative $(\partial s/\partial x)$ for the modified coordinate is $(\partial u/\partial x)*(\partial s/\partial u)=(\partial u/\partial x)*(1-n_i)$. All four partial derivatives are similarly scaled by $(1-n_i)$. Hence, the new desired mip-map level i for the modified coordinates is

$$i = log\,MAX\left[\sqrt{\left(\frac{\partial s}{\partial x}\right)^2+\left(\frac{\partial t}{\partial x}\right)^2},\sqrt{\left(\frac{\partial s}{\partial y}\right)^2+\left(\frac{\partial t}{\partial y}\right)^2}\right]$$

i.e., $i = l + log(1-n_i) = l + log(1-2^i/N)$. Solving this equation, we get

$$i = l + log\left(1+\frac{2^l}{N}\right) \quad (2)$$

Thus, the scaled texture coordinates, based on the new mip-map level i, can be directly computed from the default mip-map level l (based on the vertices' original texture coordinates). Trilinear filtering is applied manually if desired by using the integer mip-map levels surrounding i, computing two sets of texture coordinates, doing two bilinear texture lookups, and interpolating.

Although the stacked style may be implemented in a fragment program, it is really best suited for a custom texture lookup mode supported by the driver and texture unit. To implement in a fragment program on current hardware, we modify step 4 to account for the fact that we fill the entire texture domain with only the highest resolution texture, using the hardware mip-map storage for the other levels. We give the texture unit the modified texture coordinates as well as a set of derivatives to attempt to force it to use the desired mip-map level. As mentioned previously, this can be difficult, because the hardware implementation may not implement equation 1 exactly. For example the square root may be missing. Equation 2 can usually be modified accordingly, allowing us to determine appropriate derivatives to pass to the texture lookup. Ideally, in a driver-supported setting, the stacked style could allow a single texture lookup to perform filters such as trilinear, anisotropic, etc.

7. Results

We have implemented the algorithms described in this paper and tested them on several models. We generated surface patches and parameterized them. We captured per-vertex colors, per-vertex normals and per-triangle textures into seamless atlases.

Table 2 provides the experimental results for separate (Column A-D) and flat (Column E-G) atlases. These results are performed with a Pentium4 2.8 GHz PC, 1 GB RAM and an NVIDIA Quadro FX 3000 graphics card. It is running Windows XP with NVIDIA Cg 1.2 compiler and fp30 fragment shader profile. These results are rendered with per-pixel lighting and normal map textures. We use the VBO (Vertex Buffer Object) extension for fast triangle rendering. Column A and E show results without mip-mapping. Column B shows result for both bilinear filtering and trilinear filtering. We perform rendering with anisotropic 4x and 8x in Column C and D. Column F shows result with bilinear filtering for flat atlas. Result with trilinear filtering for flat style is shown in Column G. Columns A-E use 12 fragment instructions. Columns F and G use 57 and 67 fragment instructions, respectively. These results show that we can display complex models in interactive rates.

It is informative to compare Column A with Column E. It seems that for our current set of tests, the per-chart texture binds and draw calls do not cause much penalty for Column A as compared to Column E, which has a single bind and draw call. In other tests, where we use coarser geometric levels of detail of these models, we have seen Column E perform as much as 20% faster than Column A. Thus it seems worthwhile to pack charts into a single texture on cur-

Model	**Tris** K	**A** fps	**B** fps	**C** fps	**D** fps	**E** fps	**F** fps	**G** fps
Knee	76	108	121	97	85	110	32	21
Bunny	76	93	113	88	74	83	30	19
Club	105	77	90	72	63	80	24	16
Teeth	234	40	44	38	34	41	12	8
Igea	269	35	39	33	30	35	10	7
Isis	377	26	29	27	23	26	8	6
Cuneiform	1079	11	11	11	10	11	3	2

Table 2: *Timing results for seamless rendering*

rent hardware only when the number of triangles per chart is quite small. When this is beneficial, it is also possible in some scenes to pack the geometry and textures of multiple objects together, further reducing bind and draw overheads. The texture component of this process is straightforward in the context of our atlases.

It is also informative to compare Column A to Column B. We find that standard trilinear mip-mapping and anisotropic filtering achieves similar performance, and both outperform non-mip-mapped rendering (presumably due to improved texel caching). With hardware support for our atlas lookups, we should expect a similar speedup from Column E to Columns F and G. However, the number of instructions required to implement these lookups in a fragment program apparently outweighs the gain in cache performance.

Figure 17 demonstrates the effectiveness of correct sampling at the patch boundaries of all mip-map levels. Naïve approaches may interpolate between correct texels and those of adjacently packed charts, resulting in the visible seam artifacts shown here.

Figure 18 shows the normal-mapped tooth at varying levels of geometric detail. Figure 19 shows a texture-mapped Bunny model generated from a model which had per-triangle textures. A close-up at a part of Bunny verifies that we do not introduce seams on the patch boundaries.

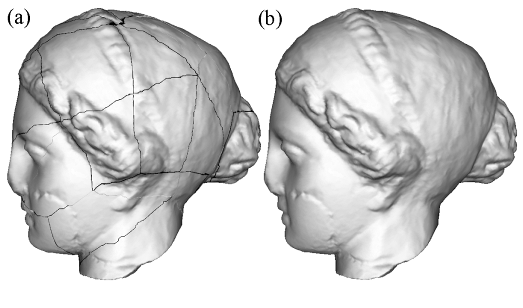

Figure 17: *Rendering normal-mapped Igea model. (a) Seams introduced because of interpolation across adjacently-packed chart boundaries. (b) Our correct boundary filtering and lookup eliminates these seams.*

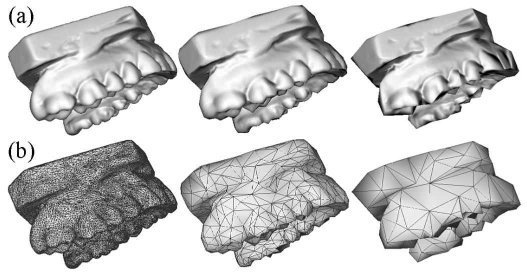

Figure 18: *Normal-mapped Teeth model with varying level of detail. (a) Seamless rendering with LOD. (b) Rendering the corresponding mesh without normal map.*

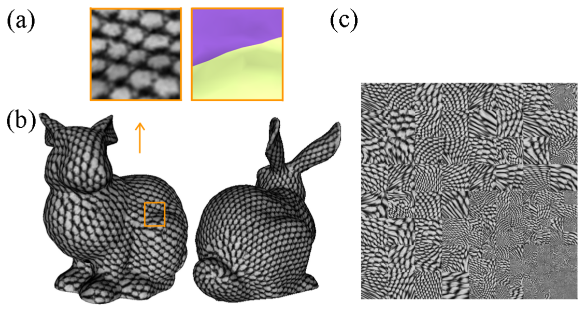

Figure 19: *Texture-mapped Bunny model and its flat seamless atlas. (a) A close-up on a part of Bunny with a patch boundary (with the patches shown next to it). (b) Seamless texture-mapped Bunny. (c) The flat atlas.*

Figure 20 (see color plates in printed proceedings) shows a Cuneiform tablet model [KSD*03] with one million triangles, rendered using individual mip-map levels of its color texture. As we increase the texture filtering, the rendering remains seamless.

A mip-mapped Club model is shown in Figure 21.The result of applying various texture filtering modes to the Club model is shown in Figure 22.

8. Conclusion

We have presented an algorithm for constructing seamless atlases, which seamlessly texture polygonal meshes and allow mip-mapping as well as geometric simplification. These new texture atlases take advantage of fragment processor programmability. Our approach adapts methods of seamless texturing available in OpenGL to the requirements for large batch sizes of geometry to provide maximal performance on modern graphics hardware. While it is possible to adjust the texture coordinates at each level of mip-mapping in a fragment program, better results could be achieved with a custom texture lookup function. The seamless atlas could thus become a new and useful custum texture format.

There remain several important avenues for future research. Metrics for construction of variable precision charts for the packed style bear further investigation. We also believe that combining patch quadrangulation directly with parameterization would yield better patches. In addition, out of core parameterization and texture caching would be useful for larger models.

9. Acknowledgments

We thank Jatin Chhugani, Yuan Chen, and Marc Olano for insightful discussions. Models appear courtesy of the Canada National Research Council, Cyberware, Stanford Computer Graphics Lab, and Greg Turk. This research was supported in part by NSF Medium ITR IIS-0205586 and a DOE Early Career Award (the views expressed in this work are not necessarily those of our sponsors).

References

[CC78] CATMULL E., CLARK J.: Recursively generated b-spline surfaces on arbitrary topological meshes. *Computer-aided Design 10*, 6 (1978), 350–355.

[CH02] CARR N., HART J.: Meshed atlases for real-time procedural solid texturing. *ACM Transactions on Graphics 21*, 2 (2002), 106–131.

[CMR*99] CIGNONI P., MONTANI C., ROCCHINI C., SCOPIGNO R., TARINI M.: Preserving attribute values on simplified meshes by resampling detail textures. *The Visual Computer 15*, 10 (1999), 519–539.

[COM98] COHEN J., OLANO M., MANOCHA D.: Appearance-preserving simplification. In *Proceedings of SIGGRAPH* (1998), pp. 115–122.

[DMA02] DESBRUN M., MEYER M., ALLIEZ P.: Intrinsic parameterizations of surface meshes. *Computer Graphics Forum 21* (2002), 209–218.

[EDD*95] ECK M., DEROSE T., DUCHAMP T., HOPPE H., LOUNSBERY M., STUETZLE W.: Multiresolution analysis of arbitrary meshes. In *Proceedings of SIGGRAPH* (1995), pp. 173–182.

[EH96] ECK M., HOPPE H.: Automatic reconstruction of b-spline surfaces of arbitrary topological type. In *Proceedings of SIGGRAPH* (1996), pp. 325–334.

[Flo97] FLOATER M.: Parameterization and smooth approximation of surface triangulation. *Computer Aided Geometric Design 14* (1997), 231–250.

[FvDFH90] FOLEY J., VAN DAM A., FEINER S., HUGHES J.: *Computer Graphics: Principles and Practice. The Systems Programming Series. 2nd edition.* Addison-Wesley, Reading, MA, 1990.

[GGH02] GU X., GORTLER S., HOPPE H.: Geometry images. In *Proceedings of SIGGRAPH* (2002), pp. 355–361.

[GWH01] GARLAND M., WILLMOTT A., HECKBERT P.: Hierarchical face clustering on polygonal surfaces. In *Proceedings of Symposium on Interactive 3D Graphics* (2001), pp. 49–58.

[KL96] KRISHNAMURTHY V., LEVOY M.: Fitting smooth surfaces to dense polygon meshes. In *Proceedings of SIGGRAPH* (1996), pp. 313–324.

[KLS03] KHODAKOVSKY A., LITKE N., SCHRÖDER P.: Globally smooth parameterizations with low distortion. In *Proceedings of ACM SIGGRAPH* (2003), vol. 22(3) of *ACM Transactions on Graphics*, pp. 350–357.

[KSD*03] KUMAR S., SNYDER D., DUNCAN D., COHEN J., COOPER J.: Digital preservation of ancient cunieform tablets using 3d scanning. In *Proceedings of Fourth International Conference on 3-D Digital Imaging and Modeling* (2003), pp. 326–333.

[LCCK02] LEVEN J., CORSO J., COHEN J. D., KUMAR S.: Interactive visualization of unstructured grids using hierarchical 3d textures. In *Proceedings of IEEE Symposium on Volume Visualization and Graphics* (2002), pp. 37–44.

[LHJ99] LAMAR E., HAMANN B., JOY K. I.: Multiresolution techniques for interactive texture-based volume visualization. In *Proceedings of IEEE Visualization* (1999), pp. 355–361.

[LPRM02] LÉVY B., PETITJEAN S., RAY N., MAILLOT J.: Least squares conformal maps for automatic texture atlas generation. In *Proceedings of SIGGRAPH* (2002), pp. 362–371.

[LSS*98] LEE A., SWELDENS W., SCHRÖDER P., COWSAR L., DOBKIN D.: Maps: multiresolution adaptive parameterization of surfaces. In *Proceedings of SIGGRAPH* (1998), pp. 95–104.

[Ped95] PEDERSEN H. K.: Decorating implicit surfaces. In *Proceedings of SIGGRAPH* (1995), pp. 291–300.

[PH03] PRAUN E., HOPPE H.: Spherical parametrization and remeshing. *ACM Transactions on Graphics 22*, 3 (2003), 340–349.

[SGR96] SOUCY M., GODIN G., RIOUX M.: A texture-mapping approach for the compression of colored 3d triangulations. *The Visual Computer 12* (1996), 503–514.

[SH02] SHEFFER A., HART J.: Seamster: inconspicuous low-distortion texture seam layout. In *Proceedings of IEEE Visualization* (2002), pp. 291–298.

[SSGH01] SANDER P., SNYDER J., GORTLER S., HOPPE H.: Texture mapping progressive meshes. In *Proceedings of SIGGRAPH* (2001), pp. 409–416.

[SWG*03] SANDER P., WOOD Z., GORTLER S., SNYDER J., HOPPE H.: Multi-chart geometry images. In *Proceedings of the Eurographics/ACM SIGGRAPH Symposium on Geometry Processing* (2003), pp. 146–155.

[WNDS99] WOO M., NEIDER J., DAVIS T., SHREINER: *OpenGL Programming Guide*. Addison Wesley, 1999.

[ZMT04] ZHANG E., MISCHAIKOW K., TURK G.: Feature-based surface parameterization and texture mapping. *ACM Transactions on Graphics* (2004).

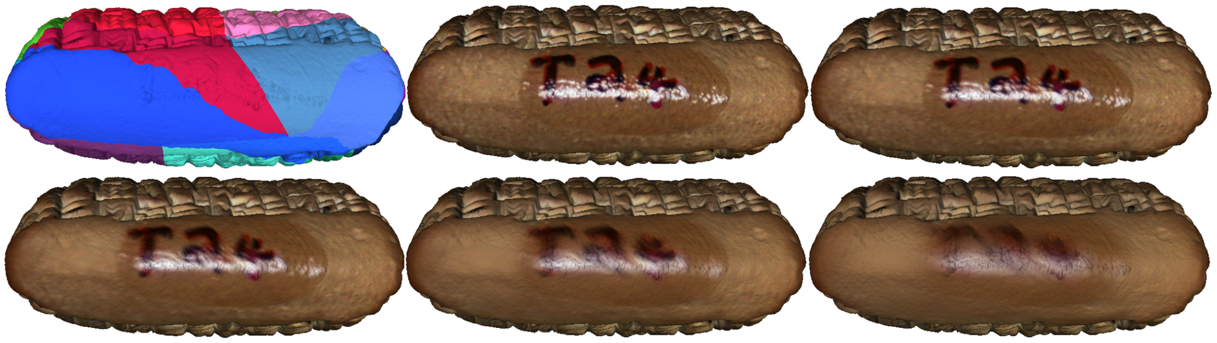

Figure 20: *One-million-triangle Cuneiform tablet model, rendered at several of its individual mip-map levels to illustrate the seamless filtering.*

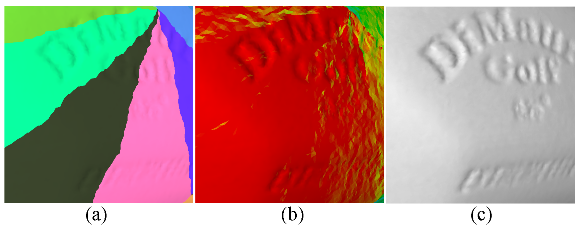

Figure 21: *Club model with seamless rendering with mip-mapped normal. (a) Patches shown. (b) Color indicates mip-map level. (c) Seamless mip-mapped rendering.*

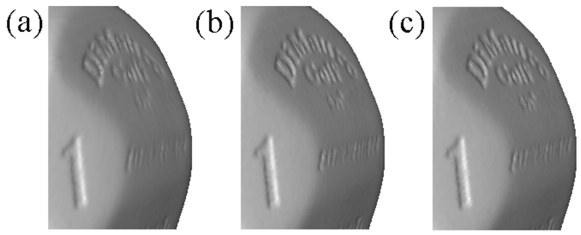

Figure 22: *Club model with various advanced texture filtering modes. (a) Trilinear filtering. (b) Anisotropic 4x. (c) Anisotropic 8x.*