Vision, Modeling, and Visualization (2010)

Hardware Accelerated 3D Mesh Painting

Randolf Schärfig and Kai Hormann

Faculty of Informatics, University of Lugano, Switzerland

Abstract

In this paper we present a new algorithm for interactively painting onto 3D meshes that exploits recent advances of GPU technology. As the user moves a brush over the 3D mesh, its paint pattern is projected onto the 3D geometry at the current viewing angle and copied to the corresponding region in the object's texture atlas. Both operations are realized on the GPU, with the advantage that all data resides in the fast GPU memory, which in turn leads to high frame rates. A main feature of our approach is the handling of seams. Whenever the brush overlaps two or more patches, this situation is detected and the paint pattern is copied correctly to the corresponding texture charts. In this way the operation of the projection into the texture atlas is completely reduced to a single texture lookup. The performance is independent of the resolution of both the brush and the texture atlas as well as the number of mesh triangles.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Bitmap and framebuffer operations I.3.7 [Computer Graphics]: Color, shading, shadowing, and texture

1. Introduction

A common way of storing data given on the surface of a 3D mesh is to write it to the object's texture atlas [Low01]. And as the texture resolution dictates the sampling density, current techniques usually fill the texture by sampling the surface values on the mesh once for each texel. For large textures this can be very expensive and therefore undesirable, in particular if the data on the mesh surface changes frequently.

An example of the latter is mesh painting, where the user wants to draw with a brush onto the surface of a mesh and so its texture atlas must be updated correctly and at interactive speed. In this situation, a more natural approach is to work the other way around, that is, to project and copy the brush pattern into the relevant texels.

The aim of this paper is to describe a general technique for realizing this mapping $\psi\colon \mathcal{B} \to \mathcal{T}$ from the brush $\mathcal{B}$ (given as a bitmap) into the 2D texture atlas $\mathcal{T}$ of a 3D mesh $\mathcal{M}$ efficiently on the graphics card. In a nutshell, we decompose ψ into a projection $\pi\colon \mathcal{B} \to \mathcal{M}$ that maps the brush onto the 3D mesh (see Section 3.3) and the parameterization $\varphi\colon \mathcal{M} \to \mathcal{T}$ of $\mathcal{M}$ which further maps into the texture atlas, where φ is computed with any standard method [LPRM02, LZX*08]. Combining both mappings then yields $\psi = \varphi \circ \pi$.

The main difficulty of this approach is to correctly deal with texture atlases that contain more than one texture chart. In this situation, the 3D mesh is split into several patches, each with its own texture chart, which is generally unavoidable for complex meshes, both for topological and practical reasons regarding parametric distortion [LPRM02]. Now, if the projection $\mathcal{R} = \pi(\mathcal{B}) \subset \mathcal{M}$ of the brush is a contiguous region on the mesh surface that spans across $k \geq 2$ patches, its image $\varphi(\mathcal{R})$ in the texture atlas is no longer contiguous as it lies in different texture charts (see Fig. 4). We handle this situation by mapping the brush into each of the separate texture charts that correspond to the k patches which intersect with $\mathcal{R}$. In order to realize this idea, we must enlarge the charts by computing additional *virtual texture coordinates* for the mesh vertices near the patch boundaries (see Section 3.5). A nice consequence of using enlarged charts is that the texture data is replicated in corresponding regions near the chart boundaries in the texture atlas, which in turn helps to avoid texture bleeding if bilinear texture filtering and mipmapping is turned on. Figs. 1 and 7 show some examples of our approach.

Our technique is designed to nicely follow the flow of the graphics pipeline and exploits the capabilities of every unit: vertex processor, geometry processor, rasterizer, and fragment processor. Once fed with the relevant data, it computes the mapping ψ purely on the GPU, without needing to read

DOI: 10.2312/PE/VMV/VMV10/211-218

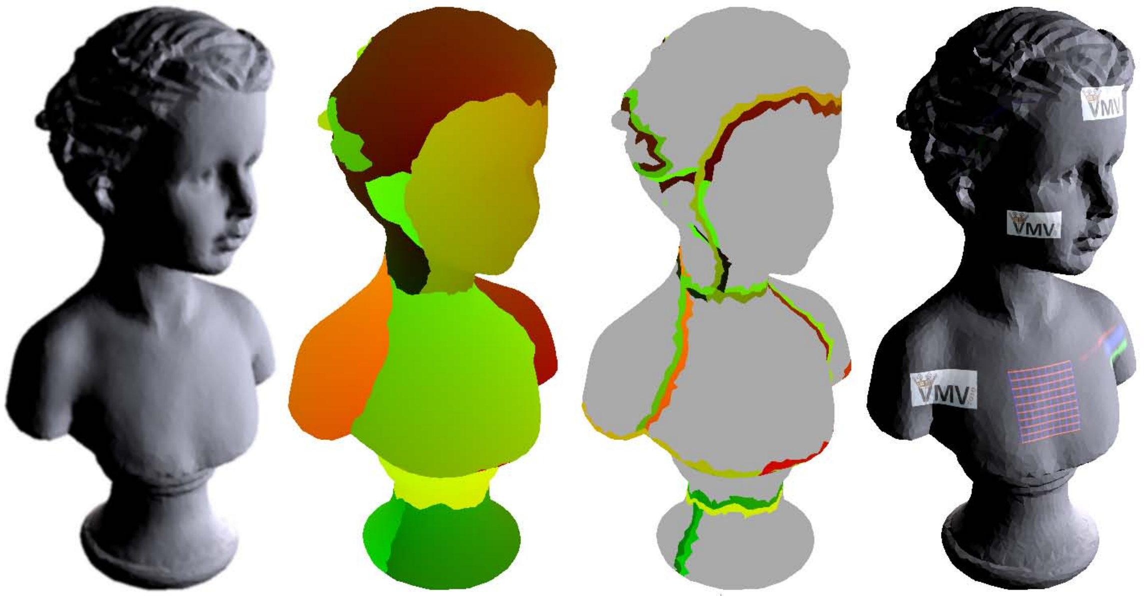

Figure 1: *Overview of our method (from left to right): The first image shows the 3D mesh that the user wants to paint onto. Note that it is segmented into several patches as becomes clear from the next image, which shows the content of the TexBuffer, i.e. the interpolated texture coordinates for the mesh, colour-coded in the red and green components. The third image shows the virtual texture coordinates of triangles near the seams. The colour coding is as for the second image and it can be observed that they continue the texture coordinates of each patch across the seams into the neighbouring patches. The rightmost image finally shows the result of a painting session; the corresponding texture atlas is shown in Fig. 2.*

any data back from the graphics card, and it requires only a single drawing step for any painting that is done.

A notable feature of our technique is that it treats the brush as a contiguous object and does not simply project each single pixel individually into the texture atlas. The latter approach would create holes in the texture if the texture has a higher resolution than the brush, but our approach does not suffer from such sampling artefacts. Moreover, the runtime is independent of the mesh complexity.

1.1. Contributions

The main contributions of this paper are:

- We introduce the concept of a *TexBuffer* that contains the 2D texture coordinates of the 3D mesh in screen coordinates and is used to realize the mapping ψ by telling each point in the brush B to which position in the texture atlas it corresponds.
- We describe how to compute suitable *virtual texture coordinates* for enlarging the texture charts.
- We use both concepts to ensure that the brush is projected correctly into the texture atlas, even in the case when its projection onto the 3D mesh overlaps one or more seams, hence the brush must be copied to more than one chart in the texture atlas.

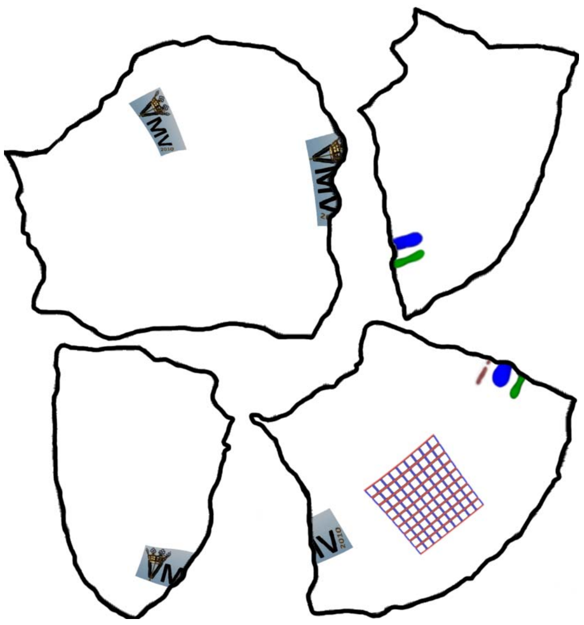

Figure 2: *Part of the texture atlas used for the rightmost image in Fig. 1.*

2. Related Work

The first approach to mesh painting was presented by Hanrahan and Haeberli [HH90]. They simply sample the brush once for each vertex of the mesh and store the sampled values as vertex colours. This technique was introduced before the rise of textures and has the disadvantage that the mesh has to be very densely tessellated for visibly appealing results. A similar approach was later presented by Agrawala et al. [ABL95].

The recent paper by Fu and Chen [FC08] proposes to draw directly to the mesh triangles and even sub-sample the mesh if it is not sufficiently tessellated for good results. This approach does not follow the design of modern graphics hardware, which provides methods for reading such detail from a texture. Other GPU-based techniques sample the whole texture area, which is very expensive. Although this might work at interactive rates with a small texture atlas, it becomes slower with increasing texture sizes or more than one image per texture atlas if the mesh is very complex.

Other approaches simply process all texels of the whole texture atlas by drawing all mesh triangles into the texture atlas and sampling the corresponding screen triangles. If for some texel the corresponding screen pixel is overlain by the current brush, then the pixel is set to the colour of the corresponding brush pixel, otherwise the texel is discarded. This is completely done within the fragment program and therefore quite expensive since it operates on many pixels that are actually not get drawn to.

Igarashi and Cosgrove [IC01] follow this idea but introduce an intermediate step for storing the colour from a painting session in a frame buffer object that covers the whole screen. This is sufficient as long as the camera does not change and appears to the user as if the paint had already been copied into the texture atlas and texture-mapped back on the mesh surface. But the colour is actually only copied into the texture atlas of the object (by the method described above) whenever the camera moves. The main drawback of this approach is that it is bound to screen resolution. When the brush resolution exceeds the screen resolution, it can therefore not be copied without loss into the texture atlas, even if the resolution of the latter allows for the brush to be stored in full resolution.

Another paper that uses the GPU for rendering was presented by Ritschel et al. [RBM06]. They propose to store geometry images in a texture atlas which is then used to render the object and allows for interactive surface changes by painting on the mesh. But this paper is restricted to Catmull–Clark subdivision surfaces, because it relies on the specific connectivity information that is induced by the hierarchy of these surfaces.

The technique described by Lefebvre et al [LHN05] is designed to draw on meshes that do not possess a parameterization. Therefore the paint information is not stored in a texture atlas but instead in a 3D texture. For painting as well as for rasterization this method uses an octree which is handled entirely on the GPU to guarantee fast access to the texture. The advantage is that it does not require any precomputed parameterization into a texture atlas. On the other, a 3D texture requires a lot of space in the graphics card memory. Another drawback of this approach is the limitation of the maximum texture resolution. This limitation does not apply to our method because it could easily be extended to work with multiple 2D textures per model, and then the overall texture resolution would be virtually unlimited.

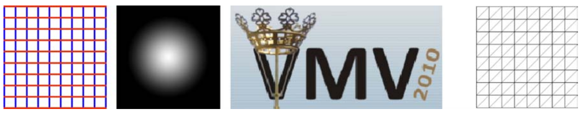

Figure 3: *A small collection of brushes used for the example in Fig. 1 and brush geometry for* $n = 8$ *(rightmost image).*

3. The Algorithm

Let us start by fixing the notation used in the description of our algorithm. The given 3D mesh $\mathcal{M}$ consists of vertices $\mathbf{P} \subset \mathbb{R}^3$ and triangles $\mathbf{T}$, and usually each vertex $P \in \mathbf{P}$ has a unique associated texture coordinate $p \in \mathcal{T}$ in the 2D texture atlas $\mathcal{T} \subset \mathbb{R}^2$. These texture coordinates can be computed with any standard parameterization method (see [HLS07] for an overview) and we assume them to be given. In our examples, we used the method of Lévy et al. [LPRM02]. They induce the parameterization $\varphi\colon \mathcal{M} \to \mathcal{T}$, which linearly maps from each mesh triangle $T = [P_0, P_1, P_2] \in \mathbf{T}$ to the corresponding texture triangle $t = [p_0, p_1, p_2] \subset \mathcal{T}$.

If the mesh is split into $m > 1$ patches $\mathcal{M} = \mathcal{M}_1 \cup \cdots \cup \mathcal{M}_m$, each with its own texture chart $\mathcal{T}_i$, then we occasionally add the chart index i to texture coordinates and triangles in order to emphasize to which chart they belong (e.g., $p^i \in \mathcal{T}_i$ or $t^j \subset \mathcal{T}_j$). Moreover, any mesh vertex P on the common boundary of a pair of neighbouring patches $\mathcal{M}_i$ and $\mathcal{M}_j$ has two texture coordinates, p^i and p^j, one in each corresponding chart, and likewise for the few vertices where three (or even more) patches meet.

The brush $\mathcal{B}$ that is used for painting onto the mesh consists of the *brush texture* (a general 2D image) and the *brush geometry*, which is a simple regular 2D mesh with $(n+1)^2$ *brush vertices* $\mathbf{Q}$ and $2n^2$ *brush triangles* $\mathbf{S}$; see Fig. 3. The brush resolution n depends on the distance between the camera and the object and is chosen such that size of the brush triangles is similar to the size of the mesh triangles in screen space (see Sec. 3.4 for details). More formally, we let the brush be the unit square $\mathcal{B} = [0,1] \times [0,1]$ and consider the brush texture to be a mapping that specifies a colour value $\mathcal{I}(b)$ for any point $b \in \mathcal{B}$. Moreover, the brush vertices are distributed on a regular grid over $\mathcal{B}$, i.e. $\mathbf{Q} = \{(i/n, j/n) : 0 \le i, j \le n\}$, and so the whole brush is covered by brush triangles, $\mathcal{B} = \bigcup_{S \in \mathbf{S}} S$.

Now the main goal of our method is to efficiently implement the mapping $\psi\colon \mathcal{B} \to \mathcal{T}$ on the graphics card and use it to copy the brush texture $\mathcal{I}(\mathcal{B})$ into the texture atlas $\mathcal{T}$. This is done by first projecting each brush vertex $Q \in \mathbf{Q}$ onto the mesh $\mathcal{M}$ and then using the given parameterization φ to determine the associated texture coordinate $q = \psi(Q) = (\varphi \circ \pi)(Q)$ of the projected point $\pi(Q) \in \mathcal{M}$. Finally, we extend the mapping ψ from the brush vertices to the brush triangles, i.e. for each brush triangle $S = [Q_0, Q_1, Q_2] \in \mathbf{S}$ we linearly map the brush texture $\mathcal{I}(S)$ to the corresponding triangle $s = [q_0, q_1, q_2]$ in $\mathcal{T}$.

3.1. Overview

We distinguish two possible user interactions: either the user changes the camera position or the viewing angle, or uses the brush to draw onto the mesh. In the first case, we

- draw the mesh on screen with texturing and lighting turned on, so that the user sees what he is interacting with;
- draw the object again into the *TexBuffer*, which is a FrameBufferObject (FBO) that stores the interpolated texture coordinates of the mesh (see Section 3.2).

On the other hand, when drawing onto the mesh, the user moves a textured brush (see Fig. 3) with the mouse over the screen and this image needs to be copied (or alpha-blended) into the corresponding regions of the texture atlas, either continuously (painting) or when the mouse button is pressed (stamping). In order to do so, we

- draw the brush triangles on screen and texture them with the currently chosen brush texture to show the user where the brush is;
- read the corresponding texture coordinates for the brush vertices from the TexBuffer and draw the brush triangles again, this time into the texture atlas, using the texture coordinates retrieved in the previous step (see Section 3.3);
- draw the mesh on screen with texturing and lighting turned on (as above) with the new texture information created in the previous step, so as to give the user feedback of his drawing action.

Note that all steps can be done at interactive rates and depend neither on the complexity of the mesh (number of vertices and triangles) nor on the resolution of both the brush texture and the texture atlas.

3.2. Initialization

At the start of the program, both the mesh $\mathcal{M}$ (including texture coordinates) and its texture atlas $\mathcal{T}$ are loaded. Since we need to have write access to the texture on the graphics card, we store $\mathcal{T}$ as an FBO. The texture can either be an existing texture image or just an empty bitmap, set to a user-specified background colour. An important feature of using an FBO as texture is that FBOs allow to write negative values, and hence support additive and subtractive image manipulation.

We further instantiate the *TexBuffer* (short for texture-buffer) as a second FBO, whose size is identical to the window in which the 3D mesh is displayed. This TexBuffer is used to store the interpolated texture coordinates of the mesh and it is set up in a second rendering step whenever the user has changed the camera settings. For each mesh triangle $T = [P_0, P_1, P_2]$ that is rendered into the TexBuffer, we let the rasterizer linearly interpolate the texture coordinates p_0, p_1, p_2 of its vertices, and let the fragment program write the interpolated texture coordinate $(u, v) \in \mathcal{T}$ into the red and green component of the TexBuffer pixels (see Fig. 1). If the mesh consists of two or more patches, then we further use the blue component to store the index of the chart that each triangle belongs to (see Sec. 3.4 for details).

Thus, the TexBuffer provides an efficient evaluation of the parameterization $\varphi\colon \mathcal{M} \to \mathcal{T}$ on the GPU: for any surface point $M \in \mathcal{M}$ that is visible from the current camera position and hence has an associated screen coordinate $(x, y) \in \mathbb{N}^2$, we can simply get its texture coordinate $\varphi(M)$ by reading it from the TexBuffer at the coordinates (x, y).

3.3. Painting

During a painting session, the goal is to quickly transfer the brush texture into the texture atlas, and this essential part of the program must be as fast as possible, because it is carried out for every paint stroke. We implement this operation on the GPU by rendering the brush geometry in the following way.

Whenever a paint event is evoked, each brush vertex $Q \in \mathbf{Q}$ has a certain screen coordinate (x, y) that depends on the current position, orientation, and size of the brush. By associating with Q the surface point $M \in \mathcal{M}$ that is visible at pixel (x, y) in the screen buffer, we effectively define a projection $\pi\colon \mathcal{B} \to \mathcal{M}$ for each brush vertex with $\pi(Q) = M$. Note that this merely describes the underlying concept, but does not involve any computations. The real action happens in the vertex program, where we read the TexBuffer at (x, y) to retrieve the texture coordinate $\varphi(M)$ of the surface point M and replace the coordinate (x, y) of Q by $\varphi(M)$ before sending this brush vertex down the rest of the graphics pipeline, along with its brush texture coordinate $(i/n, j/n) \in \mathcal{B}$.

As $\varphi(M) = \varphi(\pi(Q)) = \psi(Q)$, this modification essentially converts each brush triangle $S = [Q_0, Q_1, Q_2]$ into the corresponding texture triangle $s = [q_0, q_1, q_2]$, where $q_i = \psi(Q_i)$. By now rendering this triangle into the FBO that contains the texture atlas $\mathcal{T}$, with texturing turned on, we effectively copy the brush texture $\mathcal{I}(S)$ that is given for each brush triangle S into the correct portion of the texture atlas.

We should emphasize here that the brush triangles form a contiguous cover of the brush, and so this way of implementing the (piecewise linear) map $\psi\colon \mathcal{B} \to \mathcal{T}$ is guaranteed

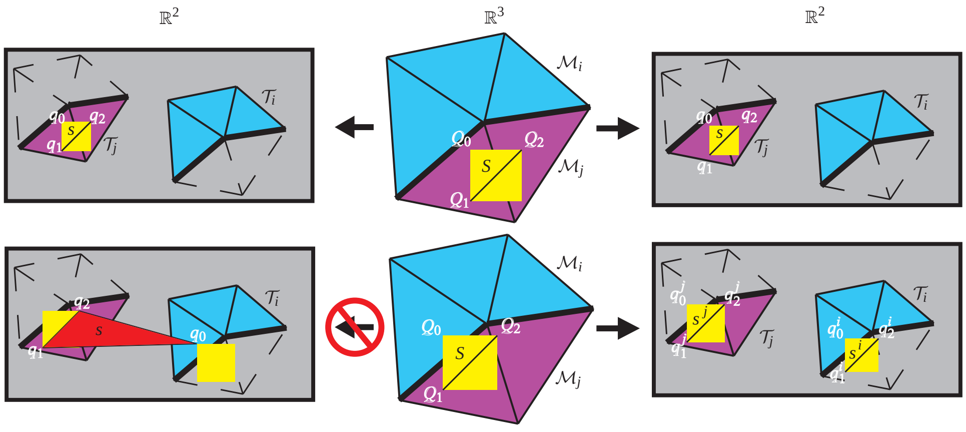

Figure 4: *If the brush geometry (yellow square) is projected completely into one chart, then we simply map its brush vertices into the corresponding texture chart (top row). However, if it overlaps a seam (bottom row), then we cannot proceed in this way, as this would result in the wrong texture region to be filled with the brush texture (left). Instead, we generate two instances of the brush triangles and map them to both corresponding texture charts, utilizing VTC (right).*

to create a contiguous copy of the brush texture in the texture atlas. I.e., even if the resolution of the texture atlas is much higher than that of the brush texture, it does not leave any relevant pixels in $\mathcal{T}$ unpainted, as it could happen if we would only splat the individual brush texture pixels into $\mathcal{T}$.

3.4. Seams

While the method explained in the previous section works well if the mesh consists of a single patch, a slightly more involved process is required for handling multiple patches, which is the usual situation for any non-trivial mesh. It can then happen that the brush overlaps two or more patches and so the brush texture must be split and copied into two or more disjoint regions in the texture atlas. We resolve this problem on the level of brush triangles.

First of all, as mentioned in Sec. 3.2, we use the TexBuffer to also store the chart indices of the visible mesh triangles. So when looking up the texture coordinate q for some brush vertex Q in the vertex program, we also get the more detailed information that it belongs to some chart $\mathcal{T}_i$, i.e. $q = q^i$. After the primitive assembly, we then use a geometry program to check if the current triangle $s = [q_0^i, q_1^j, q_2^k]$ is contained in a single patch or spans across a seam by comparing the chart indices i, j, k. If they are all the same then we just proceed as usual (Fig. 4, top), but if two of them differ, say $i \neq j = k$, then rasterizing s as it is would copy the brush texture for this triangle to the wrong region of the texture atlas (Fig. 4, bottom left).

Instead, the correct solution in this situation is to create two instances $s^i = [q_0^i, q_1^i, q_2^i]$ and $s^j = [q_0^j, q_1^j, q_2^j]$ of s and to render them into the charts $\mathcal{T}_i$ and $\mathcal{T}_j$, respectively (Fig. 4, bottom right). But how do we get the missing texture coordinates q_0^j, q_1^i, q_2^i for setting up s^i and s^j?

A straightforward solution is to simply enlarge each patch $\mathcal{M}_i$ by adding a ring of triangles to its boundary before computing the corresponding chart $\mathcal{T}_i$. Thus, if $\mathcal{M}_i$ and $\mathcal{M}_j$ are neighbouring patches and T is a triangle in $\mathcal{M}_i$ with at least one vertex on the seam, then it gets two corresponding texture triangles $t^i \subset \mathcal{T}_i$ and $t^j \subset \mathcal{T}_j$. While only the primary texture triangle t^i is used for texturing T when the mesh $\mathcal{M}$ is displayed, we need the secondary t^j to provide the missing texture coordinates above. More precisely, when initializing the TexBuffer, we store the interpolated texture coordinates from t^i and the index i as RGB values as described in Sec. 3.2, but additionally store the interpolated texture coordinates from t^j and the index j as RGB values in a second colour attachment of the TexBuffer-FBO.

Then, if the brush triangle vertex Q_0 is projected into this triangle, $\pi(Q_0) \in T$, we first fetch its primary texture coordinate q_0^i from the TexBuffer in the vertex program as in Sec. 3.2. If the geometry program later detects a seam overlap, we look up the secondary texture coordinate q_0^j in the same way from the second colour attachment of the TexBuffer in order to correctly set up the triangle s^j. Of course, the same strategy is applied to get the secondary texture coordinates q_1^i and q_2^i of the other two brush triangle vertices which are needed to specify s^i. Note that all this can be realized in the geometry program with just a single conditional branch and is thus very efficient.

Near a mesh vertex P where three mesh patches meet, it can even happen that the vertices of a single brush triangle are mapped into three different texture charts. In this case,

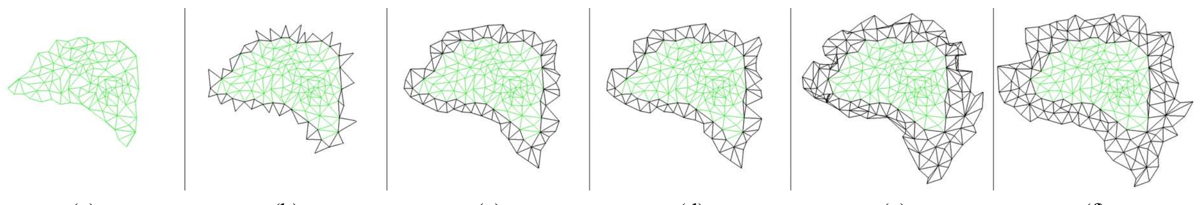

Figure 5: *From left to right: chart of a mesh patch (a); VTC of neighbouring triangles with one seam edge (b); initial VTC of remaining vertices in the one-ring (c); result of optimizing the one-ring (d); initial VTC of vertices in the two-ring (e); result of optimizing the two-ring (f). Note how the optimization untangles the triangles and reduces the distortion.*

we need to create three instances of s, which in turn requires to store another secondary set of texture coordinates plus index for all mesh triangles in the one-ring around P, and we simply use a third colour attachment to the TexBuffer for storing and accessing this data.

Even with this method of texture chart enlargement, it may occur that the secondary texture coordinates, which are needed to specify the several instances of a seam-overlapping brush triangle, are not available in the TexBuffer. For example, this can happen when the brush triangles are relatively big compared to the mesh triangles (in screen space) and then one of its vertices may end up being projected into a mesh triangle that is not adjacent to the seam and hence has no secondary texture coordinates in the neighbouring patch (compare Fig. 1).

Our solution to this problem is twofold: first, we enlarge the patches not only by a single ring of triangles, but rather add two or even three rings around the boundary of each patch; second, we adapt the brush resolution n so that brush triangles and mesh triangles are of similar size. E.g., if the mesh is far from the camera, we need a high resolution of the brush, while a brush with two triangles is sufficient if the user has zoomed very close to the mesh.

3.5. Virtual Texture Coordinates

Although the idea of providing secondary texture coordinates by enlarging and parameterizing the mesh patches as described in the previous section works conceptually, it has two major disadvantages:

1. It is often the case that a parameterization is given and can or should not be changed, for example, when a user wants to modify an already existing texture atlas.
2. By parameterizing the enlarged patches individually, it can happen that distortion across a seam edge between patches $\mathcal{M}_i$ and $\mathcal{M}_j$ is different in the corresponding charts $\mathcal{T}_i$ and $\mathcal{T}_j$, and this can yield a severe texture mismatch on both sides of the seam when mapping the texture back to the mesh as shown in Fig. 6.

We overcome both disadvantages by enlarging not the patches, but rather the charts of a given parameterization (without modifying them) and by taking care of maintaining the same parametric distortion around corresponding chart boundaries. In order to distinguish the secondary texture coordinates computed by our method from the ones obtained by simply parameterizing enlarged patches, we call them *virtual texture coordinates* (VTC). While computed differently, VTC are utilized by our method exactly as explained in the previous section.

Suppose $\mathcal{M}_i$ and $\mathcal{M}_j$ are neighbouring patches and that $[P_1, P_2]$ is one of the seam edges on their common boundary. Adjacent to this edge are the two triangles $T_1 = [P_0, P_1, P_2] \subset \mathcal{M}_i$ and $T_2 = [P_3, P_2, P_1] \subset \mathcal{M}_j$, with corresponding texture triangles $t_1^i = [p_0^i, p_1^i, p_2^i] \subset \mathcal{T}_i$ and $t_2^i = [p_3^j, p_2^j, p_1^j] \subset \mathcal{T}_j$, according to the given parameterization. In order to compute the VTC p_0^j of P_0 in $\mathcal{T}_j$ we

1. rotate T_1 about the common edge $[P_1, P_2]$ so that the rotated vertex $\tilde{P}_0$ and T_2 lie in the same plane;
2. determine the barycentric coordinates of $\tilde{P}_0$ with respect to T_2, i.e. we compute $\lambda_1, \lambda_2, \lambda_3$ such that $\tilde{P}_0 = \sum_{k=1}^{3} \lambda_k P_k$ and $\sum_{k=1}^{3} \lambda_k = 1$;
3. set $p_0^j = \sum_{k=1}^{3} \lambda_k p_k^j$.

In this way, the quadrilateral $\Diamond_j = [p_0^j, p_1^j, p_3^j, p_2^j]$ is an affine image of the quadrilateral $[\tilde{P}_0, P_1, P_3, P_2]$, and by computing the VTC p_3^i of P_3 in $\mathcal{T}_i$ analogously, we guarantee that the parametric distortions across the two corresponding chart boundaries $[p_1^i, p_2^i]$ and $[p_1^j, p_2^j]$ are compatible. That is, if we copy the brush texture into both quadrilaterals $\Diamond_i = [p_0^i, p_1^i, p_3^i, p_2^i]$ and $\Diamond_j$ as described in Sec. 3.4 and illustrated in Fig. 4, and texture the mesh triangles T_1 and T_2 with the texture information stored in t_1^i and t_2^j, then these fit perfectly together along the common edge $[P_1, P_2]$, because $\Diamond_i$ and $\Diamond_j$ are just affine images of each other (see Fig. 6).

While this fixes the VTC for the vertices of all triangles with one edge on a seam (see Fig. 5 b), there usually remain a few more vertices in the one-ring around each patch boundary for which we still need to specify a VTC. For example, if $[P_0, P_1]$ and $[P_1, P_2]$ are two successive seam edges with adjacent triangles $[P_0, P_1, P_3]$ and $[P_1, P_2, P_4]$ and two triangles $[P_1, P_5, P_3]$, $[P_1, P_4, P_5]$ in between (all in the same patch and on the same side of the two edges), then the previous

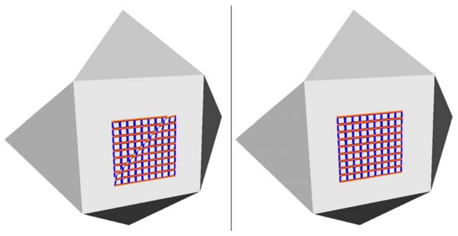

Figure 6: *It is important to calculate the VTC very carefully. If the parametric distortion to both sides of the seam is not compatible, then the painted pattern gets strongly distorted when mapped back to the mesh (left), otherwise it works out nicely (right).*

algorithms determines VTC p_3 and p_4 for P_3 and P_4, but not for P_5.

We first initialize this missing VTC by a simple linear interpolation $p_5 = (p_3 + p_4)/2$, and similarly if there should be more missing VTC between p_3 and p_4 (see Fig. 5 c). We then minimize the parametric distortion for the affected triangles $[P_1, P_5, P_3]$ and $[P_1, P_4, P_5]$ by applying a few iterations (10 to 15) of the ARAP method [LZX*08] to get an optimized VTC p_5 (see Fig. 5 d).

The whole procedure can be repeated to add more rings of VTC to each chart (see Fig. 5 e,f), with the only difference, that from the second ring on, all added VTC can be optimized by the ARAP method, as only the VTC that were computed in the very first step above are constrained to remain unmodified so as to guarantee compatible distortion across the seam edges. The only case in which we deviate from this condition is when some of the initially computed virtual texture triangles overlap each other, which happens very rarely (less than 1%). Then we include the VTC that cause the overlap into the ARAP optimization so as to get rid of the overlap.

4. Discussion

Our mesh painting algorithm stands out for three reasons: first, the underlying brush geometry guarantees that the brush texture is copied correctly into the texture atlas, regardless of the resolution of both the brush and the atlas. If instead we would project the individual texels of the brush texture into the texture atlas, then two errors are likely to occur:

- If the brush texture has a higher resolution than the texture atlas, then several brush pixels might get projected onto the same texture pixel, resulting in an "overdraw" of that pixel, so that the texture will look irregular. This cannot happen when using brush triangles, because the graphics card discards triangles that cover only a single texture pixel.
- If the brush texture has a lower resolution, then the projected pixels may lie far away from each other in the texture atlas, thus creating a grid pattern with gaps in between. Again, this cannot happen when using brush triangles, because the brush is always copied contiguously into the texture atlas this way.

Second, our method nicely handles the problem of seam-overlapping and provides a simple way of projecting a contiguous area from screen space correctly into the texture atlas, even if this area spans across several patches and thus needs to be mapped to several charts at separate locations in the texture atlas.

Third, all steps of our technique are implemented exclusively on the GPU, including all data access, which avoids expensive read back operations of data into the RAM. It exploits the natural flow of the graphics pipeline (vertex $\rightarrow$ geometry $\rightarrow$ fragment program) and needs to write only into those pixels of the texture atlas that are affected in each paint event, instead of testing for all texels whether the corresponding surface point is below the brush or not. All this leads to interactive frame rates (more than 60 fps) on a modest Nvidia 9600 GT Mobile graphics card and is basically independent of the mesh complexity (number of vertices and triangles) and both the size of the brush texture and the texture atlas.

A nice feature of our method is that we can also handle the case where the user wants to draw onto more than one mesh, each with its own texture atlas. Using the *MultiDrawBuffer*-extension of FBOs, we can easily have the texture FBO contain more than one colour buffer, and the fragment program that manages the copying of the brush texture into the texture atlas can decide into which texture to map, depending on the mesh which is currently being painted. It receives this information from the geometry program.

4.1. Limitations

Despite the advantages of our method, it also has two limitations. So far, we do not handle the situation where more than three mesh patches meet in a common mesh vertex. This could in principle be handled by adding further colour attachments to the TexBuffer, but would require a much more complex (and slower) geometry program for distinguishing all the different situations that can occur in the case that a brush triangle overlaps this common mesh vertex. Moreover, our method of choosing the brush resolution adaptively may fail, if the mesh triangles are very non-uniform in size, so that a cluster of very small triangles resides next to a very large triangle on opposite sides of a seam. Then, adding any fixed number of VTC rings around the chart boundaries might not be enough to ensure that every brush vertex of a seam-overlapping brush triangle can read the required VTC from the TexBuffer.

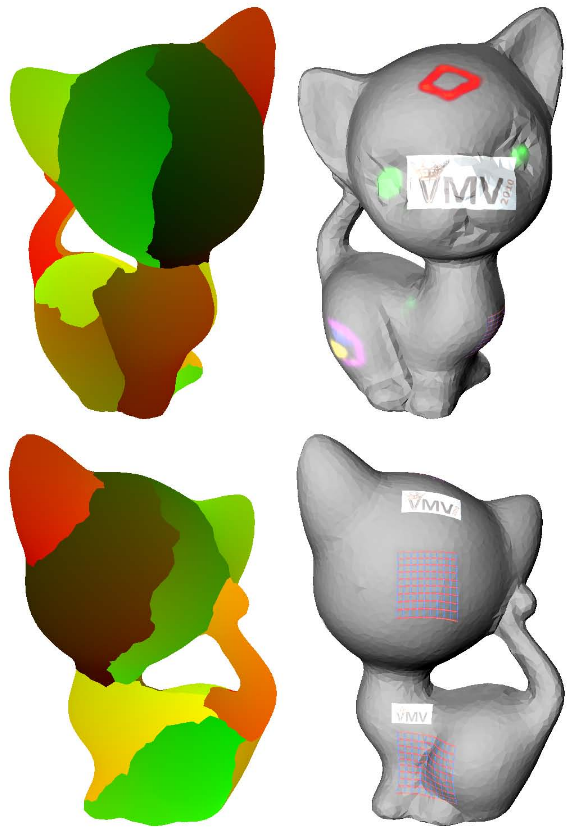

Figure 7: *Two views (top and bottom row) of a mesh that has been painted with our method. The left side shows the content of the first layer of the TexBuffer, the right side shows the textured model as displayed to the user.*

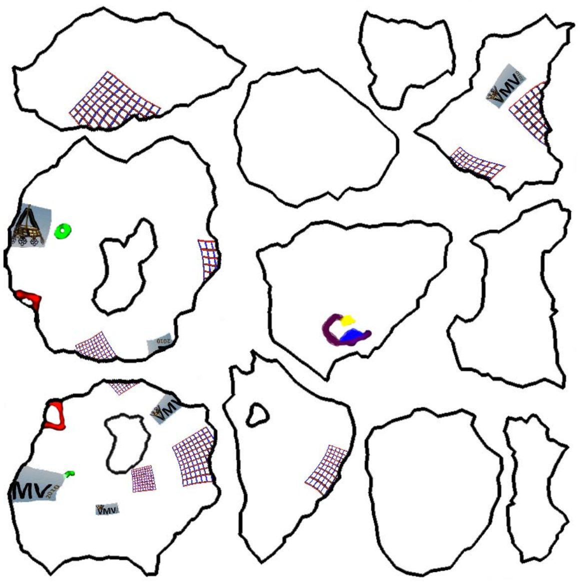

Figure 8: *Texture atlas for the mesh in Fig. 7.*

4.2. Future Work

We would like to extend our technique for "surface aware" painting, in which the geometry program would not project the brush triangles onto the mesh but rather unfold them onto the real mesh surface. This would add a naturally looking distortion to the painted texture and might be very useful in certain drawing situations, for example when painting on a surface that models a folded cloth.

References

[ABL95] AGRAWALA M., BEERS A. C., LEVOY M.: 3D painting on scanned surfaces. In *I3D '95: Proceedings of the 1995 Symposium on Interactive 3D graphics* (1995), pp. 145–150. 3

[FC08] FU Y., CHEN Y.: Haptic 3D-mesh painting based on dynamic subdivision. *Computer-Aided Design and Applications 5* (2008), 131–141. 3

[HH90] HANRAHAN P., HAEBERLI P.: Direct WYSIWYG painting and texturing on 3D shapes. *SIGGRAPH Computer Graphics 24*, 4 (1990), 215–223. 3

[HLS07] HORMANN K., LÉVY B., SHEFFER A.: Mesh parameterization: Theory and practice. In *SIGGRAPH 2007 Course Notes* (San Diego, CA, Aug. 2007), no. 2, ACM Press, pp. vi+115. 3

[IC01] IGARASHI T., COSGROVE D.: Adaptive unwrapping for interactive texture painting. In *I3D '01: Proceedings of the 2001 Symposium on Interactive 3D graphics* (2001), pp. 209–216. 3

[LHN05] LEFEBVRE S., HORNUS S., NEYRET F.: Octree textures on the GPU. In *GPU Gems 2*, Pharr M., (Ed.). Addison-Wesley, 2005, ch. 37, pp. 595–613. 3

[Low01] LOW K.-L.: *Simulated 3D painting*. Tech. Rep. TR01-022, Department of Computer Science, University of North Carolina at Chapel Hill, June 2001. 1

[LPRM02] LÉVY B., PETITJEAN S., RAY N., MAILLOT J.: Least squares conformal maps for automatic texture atlas generation. *ACM Transactions on Graphics 21*, 3 (2002), 362–371. 1, 3

[LZX*08] LIU L., ZHANG L., XU Y., GOTSMAN C., GORTLER S. J.: A local/global approach to mesh parameterization. *Computer Graphics Forum 27*, 5 (2008), 1495–1504. 1, 7

[RBM06] RITSCHEL T., BOTSCH M., MÜLLER S.: Multiresolution GPU mesh painting. In *Eurographics 2006 Short Papers* (Sept. 2006), pp. 17–20. 3