A Framework for Interactive Texturing on Curved Surfaces

Hans Køhling Pedersen
Computer Science Department
Stanford University[*]

Abstract

Existing techniques for texturing curved surfaces are either only applicable for a limited subset of surface representations (3d painting of parametric patches or dense polygonal meshes for example), or do not lend themselves naturally to interactive texture editing (e.g. procedural and solid textures). Although such methods have been used to produce stunning effects, there is a lack of flexible and general purpose tools, such as those provided by 2d image processing applications. This work argues that interactive texturing could benefit from a more cohesive paradigm built around a kernel of powerful and general operations. Using an analogy to the evolution of 2d painting algorithms, the paper motivates a framework for interactive texturing operations on curved surfaces and describes an approach for translating, rotating, and warping regions of texture (*patchinos*) on a surface. These ideas have been implemented for parametric and implicit surfaces. As an interesting side effect, this more unified framework also opens the door to a number of new interactive 3d texturing techniques that have no natural counterparts in two dimensions.

1 Introduction

In the past decade, 2d painting systems have revolutionized the field of desktop publishing. The success of these products has stimulated an intense research interest in interactive image processing tools and a diverse range of applications based on highly specialized operations has been absorbed into this thriving market, such as "Adobe PhotoShop", "Fractal Design Painter" and "Kai's Power Tools". However, underneath this diversity is an underlying framework consisting of a few general and powerful algorithms, most prominently the concepts of *digital compositing* [1] and *copy and paste* [2].

The idea behind compositing is to reduce the complexity of rendering by separating the image into a number of layers, generate an image for each layer, and subsequently synthesize the layers into one composite image using *mattes* and *alpha blending* [3]. Copy and paste operations allow portions of an image to be extracted, moved, warped and repositioned interactively. The two ideas supplement each other well as they share the underlying philosophy of looking at an image as a combination of layers that can be combined using high level operators. The two methods will henceforth be referred to under the common term of *image compositing*.

[*] I can be reached at Laboratory for Computer Science, Massachusetts Institute of Technology.

Today, image compositing forms the foundation for all commercial painting packages, a success which can be attributed to a convenient and intuitive user interface, simplicity and elegance of the underlying theoretical model, and computational efficiency that makes the idea practical.

1.1 Interactive 3d texturing

Compared to traditional framebuffer systems, which date back to the early 1970s [4], 3d painting is a new discipline [5]. Recently, however, the market for these products has experienced an explosive growth that rivals that of 2d systems a decade ago. Unfortunately, the literature on 3d painting algorithms exhibits a tendency to focus on one particular type of surface representation, such as parametric patches [5], scanned polygonal meshes [6], parameterized meshes [7], or implicit surfaces [8], thus polarizing the spectrum of painting algorithms rather than moving towards a unifying standard. Although the fundamental differences between the underlying surface representations make certain operations better suited for one particular class of surfaces than another (cubic patches, for example, are ideal for patterns utilizing trim curves, while polygonal meshes often provide a more compact representation for low frequency textures), it would be desirable to identify a nucleus of operations, a lowest common denominator, that could serve as a general framework for all painting systems. In order to design such an architecture, we need to identify a suitable core of generic operations. Fleisher et al. [9] introduced a novel framework for *cellular texturing* compatible with all standard surface representations. However, since our focus is on more general operations supported by an intuitive interface, the proven concept of image compositing forms an even better source of inspiration. Furthermore, Daily and Kiss's [10] recent study of users of painting systems concluded that artists are more likely to embrace a 3d application if its interface resembles that of the well known 2d packages. Motivated by these observations, this paper proposes a framework for interactive texturing of general smooth surfaces that extends the power of the interactive image compositing paradigm to three dimensions.

1.2 Overview

Section 2 presents an architecture for texturing operations on smooth surfaces, followed by a generalization of the basic concepts of image compositing to surfaces in section 3. Section 4 discusses more advanced high level operations, followed by a conclusion and ideas for future work.

2 Architecture

Before moving on to texture compositing on surfaces, we will start by developing an architecture that will allow such operations to be applied on general smooth models. Pedersen [8] took a step towards this goal with an approach for placing images arbitrarily on implicit surfaces in lieu of extending the method to other types of surfaces. In this section, a more elaborate architecture will be presented.

2.1 Surfaces

For some operations, such as dragging curves smoothly across a surface, it is most convenient to work with a representation that offers continuous derivatives, but for others a polyhedral approximation is more practical (estimating coordinate transformations between a surface and a patch on it, for example). To get the best of both worlds, surfaces are represented by a differentiable function as well as a parameterized polygonal mesh. For our study, we will assume that both representations are readily available. This assumption is reasonable because such techniques exist for widely used surface representations, such as spline patches (the trivial case), implicit surfaces [8], polyhedral surfaces [7, 11] and dense uniform point clouds [12].

```
Class Vector2    Real $u$,$v$
Class Vector3    Real $x$,$y$,$z$

Class Surface
  private:
    DifferentiableSurface $smooth$
              ; Differentiable function
    Mesh $mesh$    ; Parameterized polygonal mesh
  public:
    ComputeNormal(Sample $p$) $\mapsto$ Vector3
    Slide(Sample $p$,Vector3 $v$)
    ComputeCurve(Sample $p_1$,Sample $p_2$) $\mapsto$ Curve
    GetTextureCoordinates(Sample $p$) $\mapsto$ ($u$,$v$,$patch\_id$)

Class DifferentiableSurface
  virtual Vector3 ComputeNormal(Sample $p$)
  virtual Slide(Sample $p$,Vector3 $v$)

Class Parametric : public DifferentiableSurface
  private:
    Patch patches[MAX_SIZE]
        ; Parameterizations for individual patches
  public:
    ComputeNormal(Sample $p$) $\mapsto$ Vector3
    Slide(Sample $p$,Vector3 $v$)

Class Implicit : public DifferentiableSurface
  private:
    Gradient(Vector3 $x$) $\mapsto$ Vector3
    AttractPoint(Vector3 $x$) $\mapsto$ Vector3
              ; Prevents points from drifting away.
              ; See [13] for details.
  public:
    ComputeNormal(Sample $p$) $\mapsto$ Vector3
    Slide(Sample $p$,Vector3 $v$)
```

Figure 1: Representation of surfaces. See section 2.1 for further comments.

More specifically, surfaces are represented by the data structure shown in figure 1. *Slide* moves a sample point p with velocity v constrained to the surface[1], *Surface : ComputeCurve* computes a curve between two arbitrary points on the surface, and *GetTextureCoordinates* determines the surface texture coordinates corresponding to a point on the surface. Internally, *ComputeNormal* and *Move* are implemented using only the differentiable function, while the two latter procedures utilize both the smooth and polygonal representations.

Any high level texturing algorithm that can be implemented in terms of these primitives can be applied for any class for surfaces for which the above library is available.

2.2 Differentiable surfaces

Class *DifferentiableSurface* is used for operations that require a continuous derivative. In principle, it could have any number of sub-classes corresponding to subsets of surfaces with different mathematical representations, but we will focus on the important cases of implicit and parametric surfaces. Because different types of surfaces possess inherently different internal representations, surface samples are also represented as derived classes. The corresponding data structures are shown in figure 2.

```
Class Sample
  Vector3 $x$    ; Position

Class ParametricSample : public Sample
  Real $u$,$v$    ; ($u$,$v$) coordinates
  Integer $patch\_id$    ; Index of surface patch

Implicit: ComputeNormal(Sample $p$) $\mapsto$ Vector3
  $g$ = Gradient($p.x$)
  return $\frac{g}{|g|}$

Implicit: Slide(Sample $p$,Vector3 $v$)
  $v_{proj} = p.$ProjectToTangentPlane($v$)
  $p.x = p.x + v_{proj}$
  $p.x$ = AttractPoint($p.x$)

Parametric: ComputeNormal(Sample $p$) $\mapsto$ Vector3
  $\phi_u$ = patches[p.patch_id].U_Derivative($p.u, p.v$)
  $\phi_v$ = patches[p.patch_id].V_Derivative($p.u, p.v$)
  return $\frac{\phi_u \times \phi_v}{|\phi_u \times \phi_v|}$

Parametric: Slide(Sample $p$,Vector3 $v$)
  $v\_proj = p.$ProjectToTangentPlane($v$)
  $\langle\langle$ Update ($p.u$,$p.v$,$p$.patch_id) by $v\_proj$ $\rangle\rangle$
    ; See section 2.2 for detailed comments
```

Figure 2: Implementation of smooth surfaces.

In *Parametric:Slide*, the velocity vector v is projected to the tangent plane and the partial derivatives are used to estimate a uv-offset, which is added to ($p.u$,$p.v$). Care must be taken when a sample moves across a patch boundary: first, the point is moved

[1]For simplicity, it is assumed that $|v|$ is small enough for the differentiable surface to be approximated by its tangent plane within this distance. Larger velocity vectors should be subdivided into a number of smaller steps.

to the boundary of the patch, then ($p.u$,$p.v$,$p.patch\_id$) is set to the corresponding triple in the adjacent patch's coordinates and, finally, a recursive call to *Slide* completes the operation.

Notice that although procedures like *ComputeNormal* and *Slide* may be implemented very differently for parametric and implicit surfaces, we do not have to worry about such low level details when we move on to design higher level operations. Thus, from now on we will no longer distinguish between different types of surfaces but formulate all algorithms in terms of the methods of the *Surface* superclass.

3 Texture compositing

Whereas digital images can be represented by a simple uniform grid of samples, the corresponding problem of sampling a texture on a 3d surface currently has no equally convenient solution. We choose to sample the texture signal in texture maps corresponding to parameterized polygonal patches (see [8] for a motivation of this representation).

Similar to image compositing, our approach consists of three simple steps:

1. Copy a region of texture.
2. Move it.
3. Paste it back.

This section will describe how each step can be generalized to surfaces.

3.1 Copying and pasting

2d painting systems offer a variety of interactive and automatic tools for outlining regions of an image. After a region has been selected, a rectangular bounding region is copied along with an alpha channel specifying the opaqueness of each pixel. Whereas computing bounding boxes and copying pixels is uncomplicated for images, the corresponding problems for curved surfaces are a little less straightforward.

In our system, the region to be extracted is outlined by one or more closed polygonal curves on the surface using the *ComputeCurve* operation. Ideally, it would be convenient to compute a bounding box for the region automatically, but we leave this problem to future work and currently draw a rectangular bounding patch interactively. From now on, such a patch will be referred to as a *patchino* to distinguish it from the patches that constitute the polygonal mesh. After the patchino has been parameterized (see section 4 of [8] for details), the closed curves are projected to it ([8], section 5.1) and a matte is computed by performing an inside-outside test for each texel (in practice, we tessellate the closed regions and scan-convert the resulting triangles directly into the alpha channel). Finally, a coordinate transformation between the surface patches and the patchino is computed and used to copy the texels from the surface into the texture space of the latter. In section 5.2.1 of [8], an algorithm for *pasting* textures from a patchino to a surface was outlined (see also figure 9a), and the corresponding *cut* operation can be implemented similarly using the *GetTextureCoordinates* primitive (figure 9b). Although these algorithms are somewhat more time consuming to implement for surface textures than for images, the procedures are completely analogous and the simple and intuitive user interface is preserved.

3.2 Moving textures - overview

In two dimensions, regions of an image can be translated, rotated and scaled using simple affine transformations. Unfortunately, the attractive simplicity of planar geometry does not generalize directly to curved surfaces. In differential geometry, the literature on curves and surfaces in surfaces present various approaches to the problem of describing regions of curved surfaces independently of the surface representation. In some cases, such as cubic spline patches on low degree implicit surfaces [14], it is possible to derive expressions for regions of a surface analytically, but, unfortunately, the range of mathematical tools for analyzing this problem is limited, and existing results are too special case to be practical for our problem: to slide patches freely across general surfaces reliably and at interactive speed.

Due to these shortcomings of existing analytical tools, we choose to make some simplifying assumptions:

1. A patchino is approximated by a mesh of coupled springs connecting a grid of sample points.
2. As the patchino moves across the surface, only the sample points are constrained to remain on the surface.

Given these assumptions, moving a patchino can be formulated as a constrained optimization problem, namely that of minimizing metric distortion relative to some *rest shape* subject to the constraint that all samples in the grid must remain on the surface. Although parts of the patchino thus may not lie exactly on the surface, the accuracy with which it approximates the surface geometry can be chosen arbitrarily by increasing or decreasing the density of the samples in the mesh. Since most smooth surfaces are eventually rendered as a set of polygons instead of a differentiable function, it is not unreasonable to use a polygonal approximation to the patches in the texturing stage as well. We will return with a further discussion of the implications of these assumptions in section 6.

3.3 Moving textures - theory

Let $\phi : U \mapsto S$ be a parameterization of a regular patch, R, on smooth surface, S, that has a normal vector field with continuous derivative (notice that S is not required to have a parameterization).

Finding a reparameterization such that the metric distortion

$$E(U) = \int\int_U Error(\phi_u, \phi_v) du dv$$

is minimal, where ϕ_u and ϕ_v denote the partial derivatives and $Error$ is some objective function measuring the distortion within the patch, is a standard problem in graphics research [15, 7, 16, 11]. Various functionals have been proposed, weighting the preservation of angles and distances in different ways. The Green-Lagrange deformation tensor is a simple example:

$$E_{GL}(U) = \int\int_U ||G_\phi(u,v) - I||^2 du dv$$

$$= \int\int_U ({\phi_u}^2 - 1)^2 + 2(\phi_u \phi_v)^2 + ({\phi_v}^2 - 1)^2 du dv$$

where G_ϕ denotes the metric tensor of ϕ.

Our problem, however, is slightly different: instead of minimizing distortion relative to a plane, it is minimized relative to a curved rest shape parameterization θ:

$$\mathcal{E}(U) = \int\int_U ||G_\phi(u,v) - G_\theta(u,v)||^2 du dv$$

$$= \int\int_U ({\phi_u}^2 - {\theta_u}^2)^2 + 2(\phi_u\phi_v - \theta_u\theta_v)^2 + ({\phi_v}^2 - {\theta_v}^2)^2 du dv,$$

subject to the constraint that $\phi(U) \subseteq S$. Just like E_{GL} penalizes the deviation between the metric tensor to ϕ and the identity matrix, $\mathcal{E}$ measures the difference between the metric tensor of ϕ and that of θ. Naturally, there is a tradeoff between minimizing metric distortion and the "stick-to-surface" constraint, and, aside from a few simple surfaces, it is impossible to avoid some degree of distortion.

Although conceptually simple, the problem of implementing these ideas feasibly in an interactive system is challenging. To make the approach practical, we will have to replace $\mathcal{E}$ with a slightly different functional.

3.3.1 Discretization

Given a patch on a surface, its parameterization, θ, is said to be the rest shape of any other patch (with parameterization ϕ) for which $\mathcal{E}(U) = 0$. As patchinos are stored as a discrete grid of samples rather than a continuous function, the rest shape is represented as a list of spring coordinates, (k_u, k_v), for each node, p_i, in the grid:

$$q_{ij}.x_{proj} = p_i.x + k_u^{ij} p_i.\theta_u + k_v^{ij} p_i.\theta_v$$

where $q_{ij}.x_{proj}$ denotes the projection of p_i's j'th neighbor onto the tangent plane at $p_i.x$ (see also *Sample* in figure 7), and (k_u^{ij}, k_v^{ij}) are the rest coordinates of spring j emanating from node i. Each (k_u, k_v) pair thus represents the coordinates of an adjacent node in the tangent plane through $p.x$ and spanned by the basis $p.\theta_u, p.\theta_v$ (see figure 3).

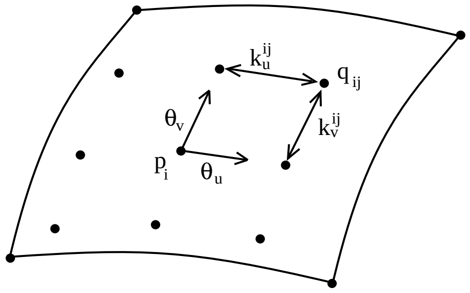

Figure 3: The local coordinates of the rest lengths of the springs are measured in the (θ_u, θ_v) coordinate frame at p.

The reason for measuring the rest lengths of the springs in local coordinates rather than in absolute distances is that the partial derivatives inevitably will change as the patchino moves across the surface, and fighting this distortion by trying to keep the metric tensors identical everywhere (see $\mathcal{E}$ above) is a losing battle that will quickly result in the patchino folding onto itself. Our solution to the tradeoff between minimizing $\mathcal{E}$ and preserving the structural integrity of the grid is to assign a priority to each sample point, indicating the relative importance of minimizing distortion in this particular region. Since the base surface was assumed to be relatively smooth, it is reasonable to allow a greater amount of distortion near the edges of a patchino than at its center, and this can be accomplished using local coordinates as described above and controlling the order in which the nodes are updated in each iteration. More specifically, the algorithm, which will be described in detail in section 3.4, proceeds in a spiral pattern emanating from the center as shown in figure 4.

The discretization of the modified $\mathcal{E}$ is

$$\mathcal{E}^*(U) = \sum_i \sum_j (k_{u_\phi}^{ij} - k_{u_\theta}^{ij})^2 + (k_{v_\phi}^{ij} - k_{v_\theta}^{ij})^2$$

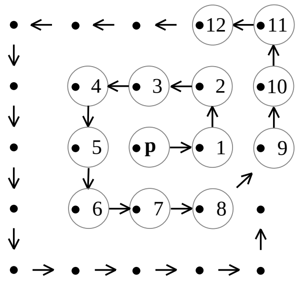

Figure 4: The order in which the nodes are visited forms a spiral emanating from the center node.

where the greek subscript denotes the parameterization for which the derivatives define the basis vectors where the spring coordinates are measured. In essence, the procedure adapts the stiffness of the springs to the geometry of the surface, but still preserves the shape of the patch in the sense that no matter how far it moves from the location where its rest shape was specified, it will reattain its original shape when it returns.

To summarize, the algorithm works by first estimating the partial derivatives at each sample point using finite differences, then minimizing $\mathcal{E}^*$ by updating the positions of the samples in a specific order subject to the constraint that all the samples must remain on the surface. These two steps are repeated continously, only interrupted by external forces exerted on a patchino by user interaction. We will return with more details on the implementation in a moment.

3.3.2 Moving patchinos

Motion can be integrated into the optimization procedure by repeating two steps. First, each sample point $p_i.x$ is moved with a velocity vector $p_i.v$, perturbing $\mathcal{E}^*$ slightly from a local minimum, and the objective function is then pulled back towards a minimum as described in the previous section.

As with any optimization technique, the robustness depends on the step size, and the trick is to pick the sample velocities so that the updated positions will remain in the proximity of a local minimum for $\mathcal{E}^*$.

3.4 Moving textures - implementation

Figure 7 outlines an implementation of the most basic operations needed for interacting with patchinos. In the following, each step will be described in more detail.

3.4.1 Freezing the rest shape

Before a patchino can be moved, its rest shape must be initialized to the parameterization of a patchino. The algorithm for *freezing* the current shape of a patch to its rest shape is sketched as *Patchino : Freeze* in figure 7. Given the resulting data structure and the partial derivatives at each node, the mesh can be optimized towards the rest shape as shown in procedure *Pathino : ReduceDistortion*.

3.4.2 Translation

In our interface, the user selects a patch by clicking on a control point. As the projection of this point is dragged in screen space, the patch moves constrained to the surface so that the control point on the patch follows the cursor. Per default, the control point is taken to be the center vertex, but it could be any fixed point on the patch. Thus, the input to the translation algorithm is simply a three-dimensional vector.

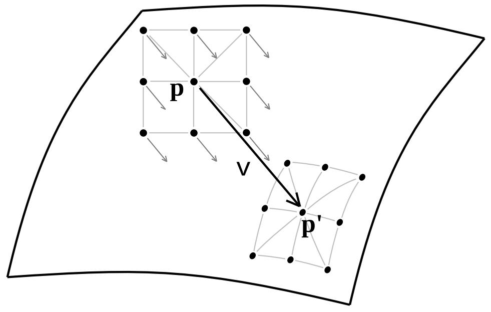

Figure 5: Propagating transformation velocities through a patch. When a transformation is applied to the center point in the grid, similar transformations are applied to its neighbors.

If this velocity vector only was applied to the center node, it might take many iterations for the operation to propagate to the entire patchino grid, and this lag makes the idea infeasible in an interactive system. A naive alternative would be to add the same velocity vector to every node, but this idea fails because it violates the stick-to-surface constraint in regions of high curvature. Instead, the velocity vector estimates are propagated outwards from the control point in a spiral order (see figure 4): First, the location of the control sample, $p.x$, is moved with velocity v constrained to the surface using *Surface : Slide*, and the tangent plane component of the difference between the new and the old position is stored in $p.v$. Second, for each immediate neighbor, q, to the control point, $p.v$ is projected to the tangent plane at $q.x$ and restored to its original length, and the resulting velocity vector is used to update $q.x$ constrained to the surface. Finally, the velocity vector $q.v$ is computed as described for $p.v$. This spiral continues until every vertex has been visited.

3.4.3 Rotation

Rotations are performed by pulling a *handle vector* (see figure 9c) emanating from the center node and constrained to the tangent plane. As the user moves the cursor, the patchino is updated so that the center node remains fixed in screen space and the projection of the handle vector always points towards the cursor. The input to the rotation operation is thus an angular offset in the tangent plane to the center node.

Guessing a set of velocity vectors for a rotation is a little more tricky than the similar problem for translations, as even a tiny rotation angle could potentially result in large displacements at nodes further from the center, thus causing numerical difficulties. Our solution is to propagate the rotation velocity vectors using the same maximum step size for every node in the grid. This yields a faster angular velocity for nodes near the center of a patchino than for nodes at its border, bounding the maximum step size at the expense of introducing a delay for nodes further away from the center. In practice, this heuristic has performed well, and it provides a sensible solution to the tradeoff between interactive feedback and robustness. The details of the algorithm are outlined as *Patchino : Rotate* on figure 7.

3.4.4 Warping

While scaling and shearing are standard features in traditional painting systems, it appears that these operations may not be as useful on curved surfaces. Instead, more general non-linear warping operations have proven to be convenient.

Patches are not restricted to remain the same shape while they are being translated and rotated on the surface. The user can interactively deform a patch by manipulating control points. This type

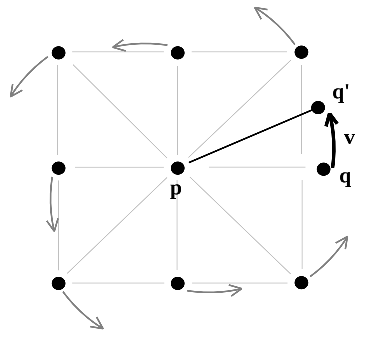

Figure 6: Propagating rotations to the samples in a patch. Given a rotation angle ω in the tangent plane, the neighbors to p try to "rotate" around this point, while remaining constrained to the surface. For example, sample q moves to q'. The samples are updated in the same order as shown on figure 4.

of warping is simple to implement given the framework described for the above operations:all that needs to be done is to modify the rest shape of the patch as it is being deformed, and the optimization process described in section 3.3.1 will automatically make the patch return to its new rest shape.

It is best to use a separate mesh of *texture springs* to store the warped patch coordinates. The original mesh of *geometry springs* should remain relatively undistorted, as the translation and rotation operations require this mesh to be as regular as possible to assure maximum robustness. Therefore, the warped coordinate system is specified independently of the geometry coordinates (see figure 8) with the exception that geometry nodes at patchino boundaries can be repositioned freely (texture and geometry nodes coincide at the boundaries). At any time, the user can map the warped texture into the texture space of the patch using the *Surface : GetTextureCoordinates* operation.

In our current implementation, the interface lets the user click on a number of feature points inside the patch and move these freely. The key points can be constrained to remain at fixed positions in the patch, allowing detailed feature alignment between texture and geometry (see figure 10). These basic operations could be extended to a more sophisticated library of warping tools, allowing points, curves and regions inside a patch to be dragged and constrained.

In our experience, surface warping is a natural and useful extension of the translation and rotation operations, and it forms the last element in our interactive texturing framework.

4 Higher level operations

Just as there are 2d image processing methods that have no meaning on curved surfaces, there are interactive texturing algorithms unique to surfaces. An interesting aspect of the architecture described in the previous sections is that it has served as inspiration for a number of new texturing operations that have no natural counterparts in traditional 2d painting systems.

4.1 Cylindrical patchinos

As many interesting shapes have parts that are relatively cylindrical, a cylindrical patchino feature has been implemented. Cylindrical patchinos are defined by three boundary curves: two cyclic geodesic curvature minimizing interpolants and one geodesic arranged in the shape of an oldfashioned pair of eye-glasses (see figures 9d and 11) and parameterized using a straightforward extension of the techniques described in [8] and [11]. The optimization

```
Class Sample
    Vector3 $x, n, v$          ; Position, normal and velocity
    Vector3 $\phi_u$,$\phi_v$   ; Derivatives
    Sample $neighbors$[8]   ; Spring coordinates
    Vector2 $k$[8]
```

```
Class Patchino
  private:
    DifferentiableSurface surface          ; Pointer to surface
    Sample grid[DIM_U][DIM_V]   ; Grid of samples
  public:
    Freeze()                       ; Set $\theta = \phi$
    Translate(Vector3 $v$)   ; Slide in direction $v$
    Rotate(Real $\omega$)   ;Rotate by angle $\omega$ around center
    ReduceDistortion() ; Minimize $\mathcal{E}^*(U)$
```

```
Patchino : Freeze()
    for $p_i \in grid$
      for $q_{ij} \in p_i.neighbors$
        ⟨⟨ $(p_i.k[j].u, p_i.k[j].v)$ = coordinates of
          $q_{ij}.x - p_i.x$ projected to tangent plane at $q_{ij}.x$
          as a linear combination as $q_{ij}.\phi_u$ and $q_{ij}.\phi_u$ ⟩⟩

Patchino : ReduceDistortion()
    $new\_p.x.x = new\_p.x.y = new\_p.x.z = 0$
    for $p \in grid$ in spiral order
      for $q \in p.neighbors$
        $new\_p.x = new\_p.x + q.x +$
          $q.k[j].u \cdot q.\phi_u + q.k[j].v \cdot q.\phi_v$
              ; Here, $q.neighbors[j] = p$
    for $p \in grid$
      $p.x = new\_p.x / \#p.neighbors$
    ⟨⟨ Recompute $n$, $\phi.u$ and $\phi.v$ at each sample ⟩⟩

Patchino : Translate(Vector3 $v$)
    $center = grid$[DIM_U/2][DIM_V/2]
    $surface$.Slide($center$,$v$)
    $center.v$ = ⟨⟨ $center.x - center.x_{old}$
      projected to tangent plane at $center.x$ ⟩⟩
    for $p \in grid/\{$center$\}$ in spiral order   ; See figure 4
      $q = p.neighbors[``interior'']$   ; See section 3.4.2
      $surface$.Slide($p, q.v$)
      $p.v$ = ⟨⟨ $p.x - p.x_{old}$ projected
            to tangent plane at $p.x$ ⟩⟩
    ⟨⟨ Recompute $n$, $\phi.u$ and $\phi.v$ at each sample ⟩⟩

Patchino : Rotate(Real $\omega$)
    $center$ =grid[DIM_U/2][DIM_V/2]
    ⟨⟨ Rotate $center.\phi_u$ and $center.\phi_v$
      by $\omega$ in tangent plane at $center.x$ ⟩⟩
    for $p \in$ grid/{center} in spiral order
      $q = p.neighbors[``interior'']$   ; See text
      $v$ = ⟨⟨ $p.x$ rotated by $\omega$ around $q.x$ in
        tangent plane at $q.x$ ⟩⟩ - $p.x$
      $surface$.Slide($p$,$v$)
      ⟨⟨ Estimate $p.\phi_u$ and $p.\phi_v$ based on $p.x$ and $q.x$ ⟩⟩
    ⟨⟨ Recompute $n$, $\phi.u$ and $\phi.v$ at each sample ⟩⟩
```

Figure 7: Implementation of patchinos.

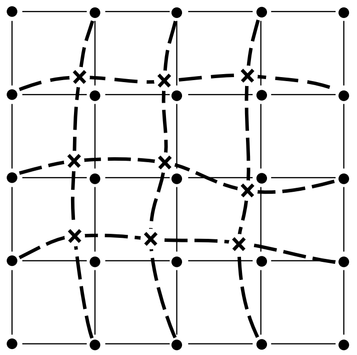

Figure 8: Texture warping. The regular grid illustrates the geometric coordinates of the patch, which should remain as undistorted as possible at all times. The curved lines shows the separate texture coordinate grid for the patch.

procedure from rectangular patchinos generalizes almost directly, except that the order in which the nodes are visited is specified by circular curves, emanating from middle of the cylinder and proceeding up and down, rather than a spiral.

Experiments have shown that compositing using cylindrical patchinos is a highly useful addition to the lower level operations from section 3. Moreover, the general idea of working with patches of different topologies present an interesting new set of sub-problems, such as how to apply transformations between rectangular and cylindrical patches interactively and how to deal with rotations. Potential applications of spherical and Mobius strip patchinos are left as a thought experiment for the interested reader.

4.2 Multi-layered compositing

In our system, any number of texture patches can exist on the surface at any time, and the user is free to translate, rotate, scale and deform these by clicking on them. Motivated by the image compositing paradigm, the patches reside in different layers and can be lowered or raised similar to the way windows can be manipulated on graphically oriented operating systems.

At any time, textures can be copy and pasted between any patch and the surface. Aside from the special case of mappings between a texture patch and the surface as described in section 3.1, textures can also be mapped between arbitrary combinations of patchinos and the surface (see figure 9e). This facilitates general image processing operations between sets of patches equivalent to [3] and thus the entire digital compositing paradigm to be applied on curved surfaces. Current texture mapping hardware supports alpha blending and allows the various textures to be rendered at interactive rates, making the approach practical on widely used high-end graphics workstations.

5 Future work

We will conclude with a few ideas for other new and potentially useful texturing operations.

Geometric patterns. A look at almost any intriguing 3d surface will reveal a correlation between geometry and textures. Tools allowing artists to take advantage of symmetries and constrain the position and extent of patchinos relative to each other would probably be very useful.

Swiss cheese patchinos. Some surfaces have regions that are relatively flat except for a hole or a branch extending outwards. In such cases, it would be convenient to operate with patchinos that would ignore the highly irregular regions. This might be accomplished by drawing feature curves around the base of a branch and constraining a surrounding patchino not to move within these.

Copy and paste of surface geometry. Recent progress in 3d data acquisition and surface fitting techniques ([17] and [11], [12]), present an interesting challenge in how to texture models of a hitherto unseen level of complexity. This problem might be alleviated by a new range of interactive applications between traditional modeling and painting systems. Operations supported by such systems could include features for dragging free-form-deformation lattices constrained to a surface while minimizing volumetric distortion subject to appropriate constraints, and copy and paste of actual geometric features extending from the surface.

6 Limitations

The underlying assumption of this work is that the base surface has to be "smooth". If the user tries to drag a patchino over a sharp spike or across any region containing frequencies that are high compared to the sampling density of the spring mesh, the patchino mesh may start to fold onto itself. In this case, the user currently looses the rest shape and has to reparameterize the interior of the patchino. Although improved robustness would obviously be desirable, the prototype implementation of these ideas is capable of dealing with a sufficiently general range of surfaces to make the approach feasible (see figures 9f, 12 and 13). In a nutshell, the more irregular the surface, the less favorable texture compositing is going to be compared to brush painting, and just like copy and paste has not replaced hand painting in 2d systems, the same is unlikely to happen for 3d surfaces. However, severe distortion is unavoidable on highly irregular surfaces, and as severely distorted texture mappings rarely look attractive, it is questionable whether this capability would even be desirable. Instead, it appears that texture compositing of relatively smooth surfaces could be a much more feasible endeavor.

7 Conclusion

Considering the demand for interactive texturing algorithms for 3d painting applications, there is a need for a more cohesive methodology for texturing of 3d surfaces. This work has proposed a kernel of powerful and general operations in the hope that this will serve as a starting point for a process that could eventually lead to more friendly 3d painting applications. Furthermore, a stronger underlying framework for texturing of general surfaces could potentially help point towards new interesting directions of research and thus accelerate the development of exciting new interactive tools.

8 Acknowledgments

Thanks to Pat Hanrahan for a fun year in California, to Venkat Krishnamurthy for many helpful and enjoyable discussions and for allowing me to use his excellent parameterization package [11], to Julie Dorsey for lots of inspiration and cool ideas, to Tamara Munzner for kindly helping with an important video tape in the last minutes before a challenging deadline, to Brian Curless for allowing me to use his friendly 3d scanning software [17], and to all of the reviewers for their much appreciated advise. Finally, I would like to thank everyone in the Stanford graphics group who helped make my visit as pleasant as it was, i særdeleshed et stort tak til Bill Lorensen for being the best sort-of-Danish officemate I have ever had.

References

[1] E. Catmull. A hidden-surface algorithm with anti-aliasing. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, volume 12, pages 6–11, August 1978.

[2] Richard Shoup. *SuperPaint*. Xerox PARC, 1974.

[3] Thomas Porter and Tom Duff. Compositing digital images. In Hank Christiansen, editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 253–259, July 1984.

[4] 1990 computer graphics achievement award. *Computer Graphics (SIGGRAPH '90 Proceedings)*, 24:17–18, August 1990.

[5] Pat Hanrahan and Paul E. Haeberli. Direct WYSIWYG painting and texturing on 3D shapes. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 215–223, August 1990.

[6] Maneesh Agrawala, Andrew C. Beers, and Marc Levoy. 3d painting on scanned surfaces. In *Proceedings 1995 Symposium on Interactive 3D Graphics (Monterey, California, April 9–12, 1995)*, pages 145–152.

[7] Jérôme Maillot, Hussein Yahia, and Anne Verroust. Interactive texture mapping. In James T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 27–34, August 1993.

[8] Hans K. Pedersen. Decorating implicit surfaces. In Robert Cook, editor, *Proceedings of SIGGRAPH '95 (Los Angeles, California, August 6–11, 1995)*, Computer Graphics Proceedings, Annual Conference Series, pages 291–300. ACM SIGGRAPH, ACM Press, August 1995.

[9] Kurt Fleisher, David H. Laidlaw, Bena L. Currin, and Alan H. Barr. Cellular texture generation. In Robert Cook, editor, *Proceedings of SIGGRAPH '95 (Los Angeles, California, August 6–11, 1995)*, Computer Graphics Proceedings, Annual Conference Series, pages 239–248. ACM SIGGRAPH, ACM Press, August 1995.

[10] Julie Daily and Kenneth Kiss. 3d painting: Paradigms for painting in a new dimension, chi '95 conference proceedings(denver colorado, may 7–11, 1995).

[11] Venkat Krishnamurthy and Marc Levoy. Fitting smooth surfaces to dense polygonal meshes for computer animation. In *Proceedings of SIGGRAPH '96 (New Orleans, Louisiana, August 4–9, 1996)*, august 1996.

[12] Matthias Eck and Hugues Hoppe. Automatic reconstruction of b-spline surfaces of arbitrary topological type. In *Proceedings of SIGGRAPH '96 (New Orleans, Louisiana, August 4–9, 1996)*, august 1996.

[13] Andrew P. Witkin and Paul S. Heckbert. Using particles to sample and control implicit surfaces. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 269–278. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[14] R. Dietz, J. Hoschek, and B. Jüttler. An algebraic approach to curves and surfaces on the sphere and on other quadrics. *Computer Aided Geometric Design*, 10(3):211–230, August 1993.

[15] Chakib Bennis, Jean-Marc Vézien, Gérard Iglésias, and André Gagalowicz. Piecewise surface flattening for non-distorted texture mapping. In Thomas W. Sederberg, editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 237–246, July 1991.

[16] Matthias Eck et al. Multiresolution analysis of arbitrary meshes. In Robert Cook, editor, *Proceedings of SIGGRAPH '95 (Los Angeles, California, August 6–11, 1995)*, Computer Graphics Proceedings, Annual Conference Series, pages 173–182. ACM SIGGRAPH, ACM Press, August 1995.

[17] Brian Curless and Marc Levoy. A volumetric method for building complex models from range images. In *Proceedings of SIGGRAPH '96 (New Orleans, Louisiana, August 4–9, 1996)*, august 1996.

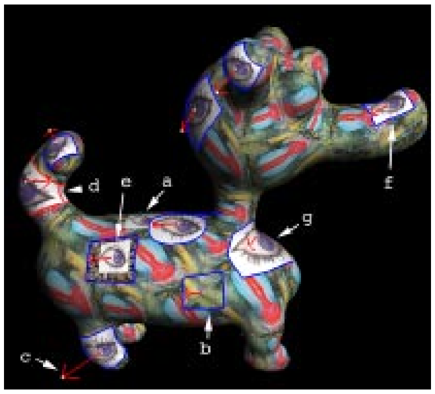

Figure 9: Patchinos on an implicit surface. a) Pasted texture. b) Copied texture. c) Interactive handle for rotations. d) Cylindrical patchino. e) Layered operations: texture mapped from one patchino to another. f) This and the other patchinos were dragged from the back of the dog in less than 10 seconds. g) Warped texture.

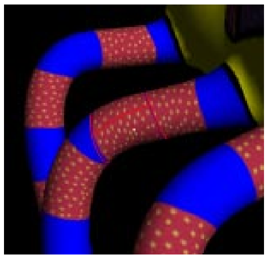

Figure 10: Copy and paste with cylindrical patchinos.

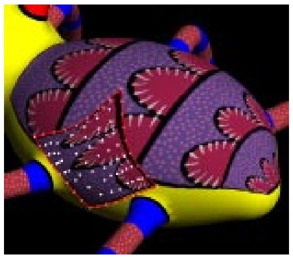

Figure 11: Warping for feature alignment. The same patchino has been pasted onto the surface at four different locations.

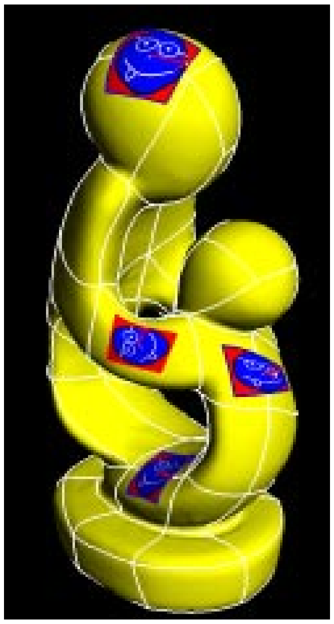

Figure 12: Patchinos on cubic spline patches.

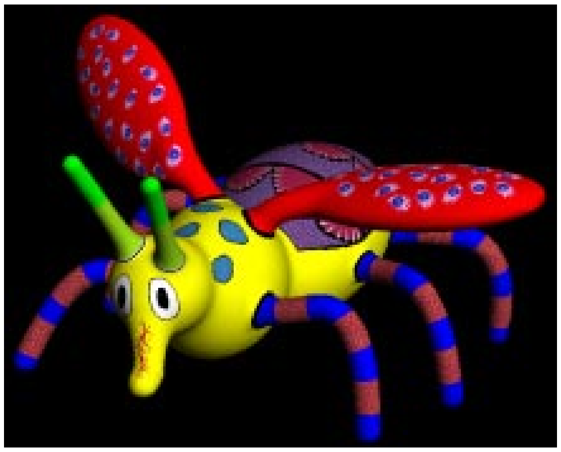

Figure 13: Copy and paste on an implicit insect.