Implicit Decals: Interactive Editing of Repetitive Patterns on Surfaces

Erwin de Groot\textsuperscript{1,2}, Brian Wyvill\textsuperscript{3,4}, Loïc Barthe\textsuperscript{5}, Ahmad Nasri\textsuperscript{6}, Paul Lalonde\textsuperscript{7}

\textsuperscript{1}University of Calgary (Canada), \textsuperscript{2}ASML, \textsuperscript{3}University of Victoria (Canada), \textsuperscript{4}University of Bath (UK), \textsuperscript{5}IRIT, Université de Toulouse (France), \textsuperscript{6}American University of Beirut (Lebanon), \textsuperscript{7}Microsoft Canada

Abstract

Texture mapping is an essential component for creating 3D models and is widely used in both the game and the movie industries. Creating texture maps has always been a complex task and existing methods carefully balance flexibility with ease of use. One difficulty in using texturing is the repeated placement of individual textures over larger areas. In this paper we propose a method which uses decals to place images onto a model. Our method allows the decals to compete for space and to deform as they are being pushed by other decals. A spherical field function is used to determine the position and the size of each decal and the deformation applied to fit the decals. The decals may span multiple objects with heterogeneous representations. Our method does not require an explicit parameterization of the model. As such, varieties of patterns including repeated patterns like rocks, tiles, and scales can be mapped. We have implemented the method using the GPU where placement, size, and orientation of thousands of decals are manipulated in real time.


1. Introduction

For many computer graphics applications, such as interactive computer games and animation, 3D objects appearance is defined by 2D textures [BN76, HH90, YKH10]. Most texturing tools require the determination of an appropriate parameterization and atlas for polygonal or implicit surfaces where no natural parameterization exists [Lév01, ZPKG02, KSG03, ZWT\textsuperscript{*}05]. Creating such a parameterization by hand is very time consuming and automatic creation suffers restrictions or compromises the quality of the result, espe-
cially in the presence of large distortions. As a consequence, the editing and the placement of textures on arbitrary surfaces remains a tedious task requiring artists to spend much time to achieve a desired result.

While it is difficult to limit distortions in the parametrization for large textures, it becomes easier when they are composed of repetitive patterns such as dragon scales or giraffe freckles (see Teaser). Large textures are then decomposed in a set of small tiles, often called decals [Ped95], that are individually positioned and mapped with a local parametrization on the surface. These small surface elements can be parametrized with very low distortions using exponential maps [SGW06].

However, to be practical for the user, a very large number of decals must be positioned and displaced interactively, which is not feasible with current techniques. A second useful functionality would be the ability to define textures across multiple objects which may in addition use different representations (such as meshes, point-set-surfaces and implicit surfaces).

To this end, we propose the use of a decaling interface whose models are free of global parametrization, as suggested by Schmidt et al. [SGW06]. Our method for defining a local parametrization is not bound to the underlying geometry, typically meshes or parametric representations. We use a particle system to automatically place cellular pattern elements (decals) over the surface of one or more objects. Subsequently, real-time interaction is possible to fine tune each element’s deformation and placement. Fast local parametrization is obtained under the assumption that for small decals, fine distortion control is not required. The use of the Euclidean distance between a surface point and the associated decal centre (i.e. the particle) is then sufficient, thus avoiding the far more expensive computation of the geodesic distance along the surface. This statement is validated by several examples illustrated in the Teaser and in Figures 13 and 9. Very fast parametrization computation is then performed via the use of spherical field functions, centred on particles. This has also the advantage of enabling the application of field deformations and implicit composition operators in order to control the shape of decals and the way they cover the surface area. These field deformations allow the automatic adjustment of the textures when decals overlap such as the eyes of the dragon and the giraffe in the teaser, or when they compete for space as is the case for the dragon scales and the giraffe freckles. This is especially important in real-time interactive graphics applications, where decals representing pattern elements are mostly required to be similar but not identical, as shown in our examples.

The main contributions of our method are:

- The very fast computation of local parameterizations based on the Euclidean distance over a model. This local parametrization can be computed at arbitrary resolution and is independent of the underlying geometric representation.
- The technique is simple enough to implement in a pixel shader, without modifying the graphics pipeline nor limiting the use of other shaders, and it allows thousands of decals to be placed and edited interactively.
- Our decals can compete for space and deform when they interact with nearby decals.
- Surface connectivity is not required thus a decal can be placed across multiple objects or across gaps in an object without changing the object representation.

This paper is organized as follows. After presenting the related works on texturing with decals (section 2), we explain how decals can be distributed over surfaces and their position edited (section 3). We then present our local parametrization system with its deformations when it populates a surface (section 4), before detailing implementation (section 5) and discussing our results (sections 7 and 8).

2. Previous work

A standard way of defining object appearance, or material information, is the creation of a single texture or a set of large textures. Textures are in general 2D or 3D. When 3D textures are used, the material is defined for all points in the 3D space in which the object is embedded. Each point of a surface is directly parametrized by its coordinates and while textures can be defined by repeated features, as done by Du et al. [DHM13], the direct control of the texture appearance on the surface remains very difficult. We rather focus on 2D textures that are directly defined over the surface.

There are several approaches to texture design but interactive painting tools [HH90, YKH10] have steered texture design interfaces. In these approaches the parameter space is either already explicit in the object representation, or is piece-wise approximated as necessary to maintain detail in the image as it is painted incrementally.

Solid procedural textures [Per85, Pea85, Wor96] are generated using noise functions or other texture basis functions. They can create many patterns, but it can be time consuming to find the right parameters and it is not possible to manipulate local features.

Another type of texture mapping interface is constrained parameterization [Lév01, KSG03]. Here a set of constraints are manually specified between the desired texture image and the surface. Global optimization algorithms are then applied to map the image onto the surface such as to satisfy the constraints and minimize a given distortion metric [GY03]. Recent advances support point sets [ZPKG02], and atlas generation from multiple images [ZWT’05]. These systems do not address the general problem of texture design, as the desired 2D images are assumed to already exist.

A visually related area of work, though very different in
implementation, is the field of texture synthesis. Wei and 
Levoy’s work exemplifies this approach [WL01]. Starting
from exemplar textures, they synthesize a new texture over
a given object by searching for the best-pixel fit in local
neighbourhoods. Similar techniques are presented by
Turk [Tur01], Efros and Freeman [EF01] and Lefebvre and
Hoppe [LH06]. These techniques must all maintain a lo-
cal or global parameterization for the object as well as the
full-size generated texture, making the techniques applicable
to pre-processing more than interactive editing. In fairness,
these methods are expected to be used when a regular tiling
deals is inadequate to capture the desired variation in the
semi-tiled textures. Turk [Tur91] proposes a technique
synthesizing a texture with repeated patterns from points
regularly sampled over the surface and a reaction-diffusion
mechanism. Even though no parametrization is required, this
approach does not support interactive editing and it is limited
in the variety of patterns it can produce.

The direct manipulation over the surface we are looking for
is efficiently done using a texturing interface introduced by
Pedersen [Ped95], called a ‘decal interface’, which com-
bines aspects of both painting and constraint tools. In this
approach the metaphor is that of 2D images affixed to the
surface. Pedersen dubs these ‘patchinos’, but they are now
more usually called decals. Decals are treated as independent
scene elements which are constrained to lie on a surface, but
may otherwise be interactively manipulated. Because a sim-
ple mapping exists between the image and the surface, 2D
image processing tools can be trivially implemented. Decals
are composited in real-time, mimicking 2D image compos-
ing [PD84] and vector graphics interfaces. This approach al-
 lows artists to interact with surface texture directly, using fa-
miliar 2D methods and tools. One of the biggest benefits of
decaling is that it allows for easy re-use of 2D images in tex-
ture design. When combined with a digital camera or image
database, realistic textures can be created very quickly. Con-
strained parameterization can also be used to apply decals,
however the human interface of [Ped95] is difficult to imple-
ment because of the problem of simultaneously moving all
of the constraints across the surface.

An interactive decaling system based on hardware-accelerated octree textures is described in [LHN05]. Ba-
sic interactive positioning and blending composition is sup-
ported. Like 3D painting, decals are applied using planar
projection. In this method image sprites can be combined
to produce blended sprites (decals). In our work we use the
implicit field to apply more complex operations such as de-
forming decals so that, for example, a snake’s scales are not
uniform but compete for space. Similarly, Autodesk Alias
products also supports application of decals using planar
projection, as well as conformal decals that rely on the sur-
face geometry [SGW06].

Schmidt et al. [SGW06] build on the decaling idea and ad-
dress many of the problems of Pedersen’s interface. In this
work a local exponential map parameterization is generated
from a single point and geodesic radius, that serves to sim-
plify the user interface and support automatic creation of de-
cals. The system can be applied to any point set, and pro-
vides a nice tool for texturing animated implicit surfaces.
It can preserve texturing even in the presence of topologi-
cal changes. Our ‘implicit decal’ approach described here is
similar in spirit but instead of deriving the local parameteri-
ization from an exponential map that is based on the geo-
metry of the surface, we introduce an implicit support surface.
This gives us most of the properties of Schmidt’s system plus
the advantages listed above.

Tiletrees [LD07] solve the problem of texturing onto arbi-
trary surfaces but the octree must be regenerated if there is a
small change in the model. In our system if a small change
is made to the model, decals can be projected back onto the
surface and may change position but the general appearance
of the decal will not change.

3. Decal placement

Decal placement consists of positioning particles over the
surface whose centre point will serve as centre for the field
functions \( f_i : \mathbb{R}^3 \rightarrow [0, 1] \) from which a local parametriza-
tion is derived for the decal. Two strategies can be used:
manual or automatic placement. Manual placement of par-
ticles is done by selecting positions on the surface of the
model (see left image of Figure 1). Even though this an
easy task, it takes a long time to texture complex surfaces
or surfaces that require a large number of decals. A tool to
automatically texture the whole surface greatly reduces the
modelling time. With our texturing method such tools are
easily created. All our method needs for input is a list of
particles (position, size and orientation). In our system this
input is generated by scattering particles [WH94] onto the
surface of the model and have them repel each other. The ra-

radius \( s_t \) of each particle is calculated by taking the maximum
distance to neighbouring particles in the local Delaunay tri-
angulation. These local triangulations can be simply computed
from the local Voronoi regions surrounding the par-
ticle centres [Ben75, Tur91]. The orientation, i.e. local 2D
frame \((u_i, v_i) \in [0, 1]^2\) tangent to the surface and of origin
the particle centre \( p_i \), is either random or aligned to some
surface field such as maximum or maximum curvature direc-
tion fields [ACSD’03, FJW’05] or as done by Turk [Tur01].

Fleischer et al.’s Cellular Texture Generation work [FLCB95]
offer a method to generate seed locations for our particles, evi-

ducing a system of partial differential equations to generate seed points and surface-
based field values on object surfaces. We implemented a
simplified version of this work, and instead of placing a
generic primitive at each particle, we place one of our
crels.

The right image of Figure 1 shows 1000 decals distributed
over the surface of a model using a particle system. It should be noted that the user can interactively edit the decals after the initial placement.

We use a particle system because it has the double advantage of being both easy to implement and flexible for integrating decal editing operations. For example, after automatically positioning decals on a model, the user can interactively edit a group of decals while the particle system repositions surrounding particles. That way, textures can be translated over the surface, rotated and scaled. Instead of a particle system, remeshing algorithms [BPR‘06] could be used to generate a coarse mesh. The positions of the vertices of this mesh would be the positions of the particles.

Once particles cover the surface, they are likely to overlap and in that case, a strategy has to be chosen in order to define how the textures coming from each overlapping decal are combined. Different solutions can be adopted. One decal can be dominant and only its texture is applied, resulting in an overlapping feature (as the eye of the dragon in the teaser). Textures can be blended (blurred) as presented by Schmidt et al. [SGW06]. A new behavior, presented in the following sections, is deformation in contact, which replaces overlapping texture by texture deformations so that they are in contact but do not overlap. This generates useful results as those illustrated in the teaser and in Figure 9.

4. Local parametrization

Once particles are distributed over the surface, we have as input their centre \( p_i \), radius \( s_i \) and local frame \((u_i,v_i)\) illustrated in Figure 2 and computed as explained in Section 3. Particles centre \( p_i \) and radius \( s_i \) are used to compute an isotropic spherical field functions \( f_i : \mathbb{R}^3 \to [0,1] \) which together with the local frame \((u_i,v_i)\) allows us to derive the local parametrization (section 4.1). Deformations produced by contact between neighbouring decals are presented in section 4.2 and the way field functions can be modified to produce parametrization adapted to decals of different shapes is explained in section 4.3.
From equation 1 for its high smoothness and its suitability for contact deformations (see section 4.2).

From field function \(f_i\) and the local frame \((u_i, v_i)\) we derive the local texture coordinates as a polar 2D coordinate system \((r(q), \theta(q))\) as follows. To calculate the radius \(r(q)\) at a surface point \(q\), we first evaluate the field value \(f_i(q)\) to which we apply the inverse of \(g\). We then scale it by \(g^{-1}(1/2)\) to get the radius \(r(q) \in [0, 1]\):

\[
r(q) = \frac{g^{-1}(F(q))}{g^{-1}(\frac{1}{2})},
\]

(3)

where \(F = f_i\) if only a single field function covers the point \(q\). We use equation 3 instead of the simpler scaled distance \(r(q) = \|q-p_i\|^\frac{1}{2}\) in order to support the different definition of \(F\) presented in Section 4.2 where decals are deformed when they are in contact as illustrated in Figure 5.

\[
\theta(q) = \arctan\left(\frac{v_i \cdot (q-p_i)}{u_i \cdot (q-p_i)}\right).
\]

(4)

Figure 4 shows an example of a our local parametrization on a plane and on a sphere. The vectors \(u_i\) and \(v_i\) are calculated from the surface normal at \(p_i\) (using cross product) and a user defined orientation angle.

4.2. Contact deformations

When decals are in collision, overlapping is avoided and contact decal deformations are performed as illustrated in Figure 5. As our local parametrization is directly derived from field functions \(f_i: \mathbb{R}^3 \rightarrow [0, 1]\), they support the set operators produced for combining implicit surfaces defined by compactly supported field functions [Bio97].

We need a deformation modelling contact between \(n\) field functions, avoiding gaps and discontinuities while maintaining \(r\) in \([0, 1]\) with \(r(q) = 1\) for all surface point \(q\) in the deformed field function boundary (see Figure 5). Following the procedure suggested by Cani [Can93], we compute a field function \(F\) as a deformation of the function \(f_k\) having the highest field value at a point \(q\):

\[
F(q) = \frac{1}{2} + \left(f_k(q) - \frac{1}{2}\right) \prod_{j \neq k} h(f_j(q), f_k(q)),
\]

(5)

where

\[
h(s, t) = \begin{cases} 
1 - \left(\frac{s+t-1}{2s-1}\right)^{1/2} & \text{if } s + t \geq 1 \\
1 & \text{if } s + t < 1
\end{cases}
\]

In this formulation, when a point \(q\) only lies in the field of one field function, the product part of equation 5 yields 1 and the whole equation equals \(f_i\). When other field functions overlap in \(q\), equation 5 adapts \(f_i(q)\) to make the decal touch but not overlap the nearby decals. This behavior is obtained when function \(F\) reproduces the graph presented in Figure 7, illustrating the combination of two field functions (including contact). This graph has been constructed following the properties and the procedures presented by Barthe et al. [BWdG04], Bernhardt et al. [BBCW10] and Gourmel et al. [GBC13]. Even though several equations matching this graph could be proposed, our formulation has the advantage of being both \(n\)-ary, i.e. it is able to combine any number of decals at once, and efficient to compute.
the fields are not overlapping each other and no deformation is needed. In situation B and C, the fields are overlapping but the parametrized areas are not, so again, no deformation is needed. Only in situation D where the parametrized areas overlap, is deformation required.

The different areas numbered from I to IV in which the final function $F$ is computed with a specific expected result are illustrated in Figure 7. In this figure, the values of function $f_1$ are taken as abscissa and those of function $f_0$ as ordinate. On the upper-left part of the diagonal line $f_0 = f_1$, the field function with the highest value is $f_0$ and $F$ is the result of the deformation of $f_0$ while in the lower-right part of this line, the highest value is the one of $f_1$ and $F$ is the result of its deformation. As we expect the same deformation behavior on both decals, the one parametrized from $f_0$ and the one parametrized from $f_1$, these two parts are symmetric and we propose to only focus on the upper-left part. The black lines in Figure 7 are the iso-lines of the resulting decal for $F = \frac{1}{2}$, $F = \frac{2}{3}$, etc. Where these iso-lines are horizontal, it means that they reproduce the values of $f_0$. $F = f_0$ and no deformation is performed. Otherwise, they represent the way the iso-values of $f_0$ are deformed by $F$, i.e. the way the decal is deformed. The contact is where $f_0 = f_1$ and we see in red area of Figure 7 how the function $F$ deforms the iso-curves at its vicinity. We refer to the implicit extrusion fields [BGC01] for a detailed explanation of how to link the graph of the composition operator (here function $F$) with the deformed objects (here the decals).

In other words, no deformation is to be performed in situations A, B and C (Figure 6), which is guaranteed as in areas I and II, $F = f_0$. In fact, deformation must hold in areas that only exist in situation D, which correspond to areas III and IV. These areas are where contact and field deformations are thus to be performed. Formally, they correspond to field values where $f_0(q) + f_1(q) > 1$. In area IV, contact deformations at parametrization boundaries are performed by ensuring that if $f_0 = f_1$, then $F = \frac{1}{2}$. The rest of area IV and area III are used to build a function $F$ performing smooth field deformations between the contact at boundaries and areas I and II where no deformation are required.

The resulting deformation in contact performed on our parametrization when this definition of $F$ is used in equation 3 is illustrated with two and three field functions in Figure 5.

### 4.3. Anisotropic parametrization

Non-circular decals are easily supported by slightly adapting the technique presented in [BS95]. In the computation of the field function in equation 1, rather than scaling the Euclidean distance between a point $q$ and the particle centre $p_i$ with a constant radius $s$, this distance is scaled by the distance between $p_i$ and the point $b_i(q)$ of intersection of a ray launched from $p_i$ and the decal boundary (see Figure 8-left). This scaling is done so that for all point $q$ lying on the decal boundary, $f_i(q) = \frac{1}{2}$.

$$f_i(q) = g\left(\frac{\|q - p_i\|}{b_i(q)}\right) g^{-1}\left(\frac{1}{2}\right). \tag{6}$$

**Figure 7:** Graph of our function $F$ when two field functions are involved. Field functions $f_1$ and $f_0$ give the values for respectively the abscissa and the ordinate.

The puzzle piece used in Figure 8 is mapped on the bunny in Figure 9 along with some other non-circular decals.

**Figure 8:** Storing the field function $f_i$ in a 2D texture for the puzzle piece decal. Left: computing $b_i$. Right: the field values in the texture (the yellow line marks the $\frac{1}{2}$ contour).

In case, function $f_1$ can be directly precomputed and stored in a 2D texture (Figure 8-right) or computed from an analytical function $b_i : \mathbb{R}^2 \rightarrow \mathbb{R}^+$ giving the radial distance between the decal centre and its boundary for all point $q$.

The puzzle piece used in Figure 8 is mapped on the bunny in Figure 9 along with some other non-circular decals.

### 5. Implementation

Implicit decals require texture mapping on a per-pixel basis which can be done in pixel shaders. The definition of decals by 3D field functions allows a computation of the pixel colour using only its 3D coordinates as input (as for 3D textures). The requirement for an efficient evaluation is the fast access to decals enclosing the pixel. Once this is done, contact deformation, overlapping or any invertible transformation can be applied to derive the texture coordinates and compute the final material information. In our system the
pixel shader of the GPU is used, because it can achieve interactive frame rates. To render a number of implicit decals on the surface of a model, the decal information (position, size and tangent vectors) is loaded into the graphics card memory. The pixel shader program queries the decal information to calculate texture coordinates. To increase performance and allow a large number of decals to be rendered, we implemented an octree structure similar to the one described in [LHN05]. An octree data structure as described in [LSK'06] could also be used.

Before the decals are loaded into the GPU memory, an octree is constructed containing the decals. Each non empty voxel contains either 8 references to other voxels or a maximum of 8 references to decals that intersect with the voxel. A voxel is subdivided when it overlaps with more than 8 decals. The list of voxels is loaded into GPU memory together with the list of decals. Our implementation of the pixel shader contains an octree lookup function which typically uses between 4 and 10 repetitions to traverse the octree and retrieve the needed voxel from the octree. With this implementation the pixel shader performs a maximum of 10 octree lookup repetitions and includes a maximum of 8 decals in the calculation of the final texture coordinates. Without an octree, all decals would have to be included in the calculation of the final texture coordinates. Rebuilding the octree will only be necessary when the user starts editing (adding, removing, moving) a different set of decals. After the rebuild (which usually is nearly instant, but can take up to a second when thousands of decals are used), editing operations can be performed interactively.

6. Filtering

Decals can be removed from the octree by removing them from the decal list in the GPU memory. Adding and moving decals however, would normally require a new construction of the octree. To prevent rebuilding the octree while the user is editing a small set of decals, the pixel shader implementation allows for a small number of decals to be added in addition to the ones in the octree. These decals will always be included in the calculation of the final texture coordinates. Rebuilding the octree will only be necessary when the user starts editing (adding, removing, moving) a different set of decals. After the rebuild (which usually is nearly instant, but can take up to a second when thousands of decals are used), editing operations can be performed interactively.

In general, existing techniques like trilinear or anisotropic
filtering with Mipmapping can be used. However, problems can occur on the borders of the decals. Some of these problems can be avoided by using a uniform colour for the borders of the decal texture images. This creates a smooth transition between touching decals. When the decals do not cover the whole surface, the border colour can be chosen to be transparent to create a smooth transition with the surface colour. Still some problems remain as can be seen in the top row of Figure 10:

1. When the decals cover no more than a few pixels (first two columns of Figure 10) standard Mipmap generation produces incorrect colours. This is caused by the use of low resolution Mipmaps which contain colour information from all pixels in the texture and not just the pixels used for the decal.

2. When the decal sizes approach pixel level (first column of Figure 10), the decal colours do not necessary converge towards the same solid colour when different texture images are used. This can result in seemingly random pixel colours and flickering of the pixels in animation.

3. There are artifacts in between the decals (last two columns of Figure 10). In the situation where two filtering samples cover different decals, the sample texture coordinates are close to the edge of the circular texture image (see Figure 11). The texture coordinates should result in the same colour (because the edge of each decal has a uniform colour) even though the texture coordinates are very different. With standard graphics hardware the difference between the texture coordinates will result in the use of a very low resolution Mipmap giving in the wrong colour.

The first problem can simply be solved by using a filter to create the Mipmaps which determines which pixels should be used. To solve the second problem, the low resolution Mipmaps need to be adjusted so that they converge towards the same colour. To do this, for each resolution a new image is constructed (the target image) which is the average of all Mipmaps of that resolution. If some textures are used for the decal.

The third problem can be solved in two steps. First, the pixel shader detects whether two filtering samples lie in different decals. This is be done by comparing the positions of the decals used in each sample. Figure 11 depicts the situation when these positions are different. In texture space the texture coordinates $A$ of one sample and the texture coordinates $B$ of the other lie farther apart than what one would expect looking at the positions in model space. This causes the filtering technique to select a Mipmap with the wrong resolution. Instead of using the distance between $A$ and $B$ in texture space to determine the pixel resolution and select the appropriate Mipmap, the distance between $A$ and $B'$ is used. As can be seen in the right image of Figure 11 $B'$ lies on the other side of the decal edge from $A$ at the same distance from the edge as $B$ (this distance is indicated by the blue curly bracket). The distance between $A$ and $B'$ in texture space is a far better representation of the actual distance between $A$ and $B$ in model space.

The combined solutions for the 3 problems above result in the images shown in the bottom row of Figure 10.

7. Results

Figure 12 shows how our decals handle sharp edges, holes in the geometry and bumps on the surface. Figure 13 shows how a single implicit decal can texture more than one surface at the same time. This can be useful in games, where stains or gunshot holes need to be applied in real time.

The dragon model shown on the Teaser-left has been textured using our method. The green scales were automatically placed using a particle system. The user manually added the (non-deforming) eye decals, and gave the decals on the ball, the tongue, and the teeth a solid colour. Some decals were slightly moved to fit the boundaries of the coloured parts of the model (ball, teeth and tongue). The whole texturing process took about 15 minutes.

The left image of Figure 9 shows a closeup of the knot
Each blood stain is a decal which can cover different models. This image shows that our method supports high resolution textures: the texture resolution of the decals in the front is the same as the resolution of the decals in the back.

The middle image of Figure 9 shows the interaction of decals with different sizes and even shows the special case of Figure 6D where a decal is placed completely inside a larger decal. The right image demonstrates several non-circular decals.

Table 1: Rendering times and memory consumption of our method. The “Voxels” column shows the number of voxels that was needed to construct a suitable octree. The “Octree” column shows the amount of time needed to construct the octree on the CPU. The “ms/frame” column shows the rendering time of one frame in milliseconds and the “Mem” column indicates the amount of video memory allocated for storing the decals and the octree (this does not include the texture images).

<table>
<thead>
<tr>
<th>Model</th>
<th>Decals</th>
<th>Voxels</th>
<th>Octree</th>
<th>ms/frame</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dragon</td>
<td>100</td>
<td>697</td>
<td>0.02s</td>
<td>25</td>
<td>12KB</td>
</tr>
<tr>
<td>Dragon</td>
<td>1K</td>
<td>5753</td>
<td>0.06s</td>
<td>30</td>
<td>116KB</td>
</tr>
<tr>
<td>Dragon</td>
<td>10K</td>
<td>58905</td>
<td>0.78s</td>
<td>32</td>
<td>1.14MB</td>
</tr>
<tr>
<td>Knot</td>
<td>100</td>
<td>897</td>
<td>0.02s</td>
<td>25</td>
<td>14KB</td>
</tr>
<tr>
<td>Knot</td>
<td>1K</td>
<td>8657</td>
<td>0.09s</td>
<td>25</td>
<td>127KB</td>
</tr>
<tr>
<td>Knot</td>
<td>10K</td>
<td>56217</td>
<td>0.80s</td>
<td>26</td>
<td>1.13MB</td>
</tr>
<tr>
<td>Horse</td>
<td>100</td>
<td>48121</td>
<td>0.02s</td>
<td>25</td>
<td>197KB</td>
</tr>
<tr>
<td>Horse</td>
<td>1K</td>
<td>130833</td>
<td>1.23s</td>
<td>25</td>
<td>605KB</td>
</tr>
<tr>
<td>Horse</td>
<td>10K</td>
<td>261561</td>
<td>2.58s</td>
<td>26</td>
<td>1.91MB</td>
</tr>
<tr>
<td>Spheres</td>
<td>100</td>
<td>921</td>
<td>0.02s</td>
<td>25</td>
<td>13KB</td>
</tr>
<tr>
<td>Spheres</td>
<td>1K</td>
<td>7393</td>
<td>0.16s</td>
<td>26</td>
<td>123KB</td>
</tr>
<tr>
<td>Spheres</td>
<td>10K</td>
<td>69889</td>
<td>1.28s</td>
<td>26</td>
<td>1.18MB</td>
</tr>
</tbody>
</table>

The results in table 1 show that our method supports interactive visualization and editing as the frame rates are between 30 and 40 frames per second even for more complex models with thousands of decals at a high window resolution. Note that these frame rates apply to our test cases where the model covers a large part of the screen. In practice Implicit Decals would only be used on some parts of the whole scene which would result in higher frame rates. Editing large groups of decals is still possible, but then the octree construction time becomes the bottleneck. To reduce the octree construction time in future implementations, a multi threaded algorithm could be implemented.

8. Discussion

Since the decal is mapped using a 3D field function, any part of the surface intersecting this field function where it is greater than \( \frac{1}{2} \) is textured. This is the desired effect when the whole surface is textured as our field function contact deformations guarantee the local influence of each decal. This naturally avoids the case shown in Figure 14-middle where a decal “bleeds” through a thin part of the model. This situation can only happen if some parts of the surface are not textured and it is thus easily avoided by using an “empty” decal defined by transparent texture images. This user interaction can also be avoided by using thinner ellipsoid field functions \( f_i \).

In general our parametrization behaves well when placed on bumps, ripples or other high curvature features. But in areas with large distortions, some parts of the texture image could be mapped onto the surface more than once which results in the texture folding over itself (see Figure 14-right). This problem is actually inherent to our parametrization as no specific treatment is done to minimize distortions. The natural solution is to use more smaller decals in these areas but if this is not desired, the field function \( f_i \) will have to be computed from a more sophisticated procedure taking into account the local surface distortions.

Our technique is mainly prescribed for a lot of small decals, that are interactively placed and manipulated over the surface. When decals become large, depending on the pattern to be repeated and on the nature of the object itself, it is more likely that undesired distortions appear. In this case, implicit...
decals may fail to provide a satisfactory result and it would be better to use a more elaborate technique such as the decals proposed by Schmidt et al. [SGW06].

It is also difficult to maintain a consistent surface covering if it is animated. Decals may change of shape and gaps may appear. The texturing of dynamic objects with implicit decals remains an open problem.

Current graphics hardware still limits the number of decals that can be edited simultaneously to approximately 20. When small groups of decals are being edited, they will be removed from the octree and added to the list of editable decals. The octree needs to be recomputed whenever the number of decals in this list exceeds the hardware limit. When ten thousands of decals are used, this can result in an occasional delay of approximately one second (see table 1). Editing operations that involve large groups of decals, always require the octree to be rebuild and will be significantly slower.

9. Conclusion and Future Work

The main contribution of this research is an efficient method for placing and editing surface decals that can interact with one another. The surface does not require an existing parameterization. Since the method lends itself to implementation on the GPU, thousands of decals can be placed and edited. Traditional techniques would require excessively large textures to achieve a reasonable resolution, making these techniques infeasible for real-time applications.

An earlier decaling method [SGW06] uses an arc length approximation (exponential map) to find a local parameterization on the surface, whereas our method uses an implicit field that is independent of the geometry. Our implicit technique has several advantages demonstrated in section 7:

- Implicit decals are independent of the underlying geometry making them applicable to multi-resolution meshes.
- The bulk of the computations can be done in the pixel shader making interactive editing of thousands of decals possible.
- Contact deformation can be used to make the decals appear to compete for space.
- An implicit decal can be made to span several objects without duplicating the decal or merging the objects.

Our method is similar to Cellular Texture Generation [FLCB95], in which a geometric primitive is placed at each particle, whereas we place a texture primitive instead. In this way many similar effects can be achieved, but at low-cost in a pixel shader. As future work we could couple our implicit decals with a GPU-shader based displacement map method, to allow for a large set of Fleischer et al.’s images to be generated in hardware.

Section 4.2 shows how contact deformation can be used to shape the decals. Future work includes investigating how other implicit operations like blending can be used to achieve different effects. Also, different distance metrics like Manhattan or anisotropic distance could be used to produce a wider variety of results.

A larger number of patterns could also be generated by creating two or more layers of decals. The final pattern is created by combining the layers, for example by using transparency to uncover deeper layers or some other function which combines the colours of each layer. Future work can also include designing functions to combine layers; how should layers be positioned in relation to each other to create interesting patterns.

To incorporate implicit decals more easily into existing modeling systems, a conversion between an implicit decal texture and more conventional textures can be written. When an atlas is in place, the model space positions of each texel of the atlas can simply be evaluated to get the corresponding colours. Future research can be done to create converters for other texture systems and incorporate better filtering.

Other future work includes better tools for decal positioning and editing. If a Poisson-disk sampling method [DH06] could be adapted to curved closed surfaces of any topology, this would significantly reduce the computation time of an initial decal distribution. Also, the system would greatly benefit from editing tools like a decal spray paint tool and brush tools that change the size or orientation of the decals. Automatic determination of the orientation of the decals could be done by using the gradient field or other properties of the model. Finally, decals with different behaviours could be added. For example decals which deform other decals, but do not deform themselves.

Acknowledgments

This work has been partially funded by NSERC (the Natural Science and Engineering Research Council of Canada), the Royal Society, Uk, Intel (Canada) and the IM&M project (ANR-11-JS02-007). We are grateful for the support of the American University of Beirut for Ahmad Nasri’s sabbatical leave to the University of Calgary.

References
