Real-Time Path-Based Surface Detail

Carles Bosch, Gustavo Patow

Research Report IIiA09-01-RR

Institut d'Informàtica i Aplicacions
Universitat de Girona

Abstract

We present a GPU algorithm to render in real-time surface detail features based on paths, such as scratches and grooves. Our method efficiently models these features using a continuous representation that is stored in two textures. The first texture is a grid that specifies the positions of the features and provides pointers to the second texture containing the paths, profiles and material information. This data is evaluated by a fragment shader on the GPU, resulting in an accurate and fast rendering of the surface detail. Such kind of detail is rendered without using additional geometry, thus considerably reducing the size of the geometry to be processed, allowing real-time computations. Intersections between features and other special cases are robustly handled by a CSG analogy. Antialiasing, non-height field features, and visibility computations are also taken into account. This method allows application of path-based features onto general surfaces just like traditional textures.

1 Introduction

Up to now, real-time visualization of surface detail has been limited to the generation and usage of geometry or sampled data structures as in bump mapping [Bli78], displacement mapping [WWT+03], or relief mapping [POC05, Tat06], which show aliasing problems for close views and do not provide a correct solution for filtering in far-views. On the other hand, vector textures are gaining popularity [QMK08, NH08], but are limited to flat 2D representations without encoding other 3D information besides normal map perturbation techniques [PRZ05]. Visibility and occlusion issues in these representations have never been treated in the context of vector-based representations.

This paper presents a feature-based per-pixel displacement mapping technique. It builds upon previous vector texture representations and per-pixel displacement mapping techniques, and makes a step forward to achieve a robust and flexible real-time vector based displacement mapping algorithm. The presented method is able to visualize geometry details like scratches, cracks, grooves and extremely sharp edged features like bricks or edges on manufactured objects. Also, our method allows accurate visualization of these path-based features in a single pass algorithm by performing a single write per pixel.

Approach: In this paper, we describe a new real-time method to compute geometric detail like scratches and grooves in texture-space by using a continuous representation that is stored in two textures, without relying on additional geometry (albeit with an increase in computational cost). We use techniques derived from the usage of vector textures in the GPU to store the geometry and properties of path profiles, and evaluate them in real-time. The first texture is a grid providing pointers to the second texture which contains the feature paths, profiles and material information. A fragment shader at the GPU evaluates this data, and generates an accurate and fast rendering by using a Constructive Solid Geometry (CSG) analogy.

Contributions: We show that the CSG analogy is a very flexible and powerful one, and that the method presented here allows accurate visualization of path-based features. This is done in a single pass and by performing only one write per pixel. As a consequence, we have a low-bandwidth coherent memory access, which is advantageous for many-core architectures. Also, it has efficient approximate anti-aliasing which allows the rendering of the features from close to distant views. We use two main approximate filtering techniques, called line-sampling and supersampling. Both techniques are used in combination to solve both visibility and shadowing anti-aliasing issues. Moreover, the shader can spatially adapt the number of samples needed according to the local geometric complexity.

Limitations: Our path-based feature representation shares a few limitations with other vector-based representations [NH08]. For example, it assumes a static layout of features, as a dynamic situation would require re-encoding features at each time step, which is very fast but is not capable of real-time results. Also, a feature segment can be replicated in many cells it overlaps, but in our experience there is almost no storage overhead. Also, each cell may have a different number of features, thus requiring an indirection scheme to avoid data sparseness. Our representation is limited to path-based features only, not allowing generic detail to be added. Also, we require the features and the object surface where they are applied to have low curvature.

2 Previous work

The method we present in this paper is closely related to surface detail techniques, real-time vector texture representations, and scratches and grooves modeling and rendering.

In general, macro-geometric models use general techniques that allow the simulation of different kinds of surface details, such as bump mapping [Bli78], displacement mapping [WWT+03], or relief mapping [POC05, Tat06], among others. For an in-depth survey on displacement mapping techniques on the GPU, please refer to [LKU08]. These macro-geometry models suffer from resolution problems and are not able to simulate high frequency or close details. The method presented here addresses these issues in an efficient and natural way.

Our method is also related to real-time vector graphics, which always have had a great appeal because of their seamless scaling capabilities. In [LB05], they require a heavy preprocessing that includes segmenting the contour and embedding each segment in a triangle. Other schemes present limitations in the number of primitives allowed for each cell: a few line segments [Sen04, TC04, LH06], an implicit bilinear curve [TC05,Lov05], two quadratic segments [PRZ05], or a fixed number of corner features [QMK06]. Also, Parilov et al. [PRZ05] presented a method for rendering normal maps with discontinuities, which was restricted to path patterns with no "T" junctions, no occlusion computations and with a unique profile for all features. All these methods share a drawback of limiting the number of allowed primitives, which is bad for areas which require high detail. One solution would be to use a finer lattice, but this would greatly increase storage needs. We use a variable-length cell representation that allows for graphics of arbitrary complexity, having none of the above mentioned restrictions. Our approach stores feature paths in a way similar to [QMK08, NH08], but here it is used to store a 3D structure, not a 2D one as in the mentioned methods.

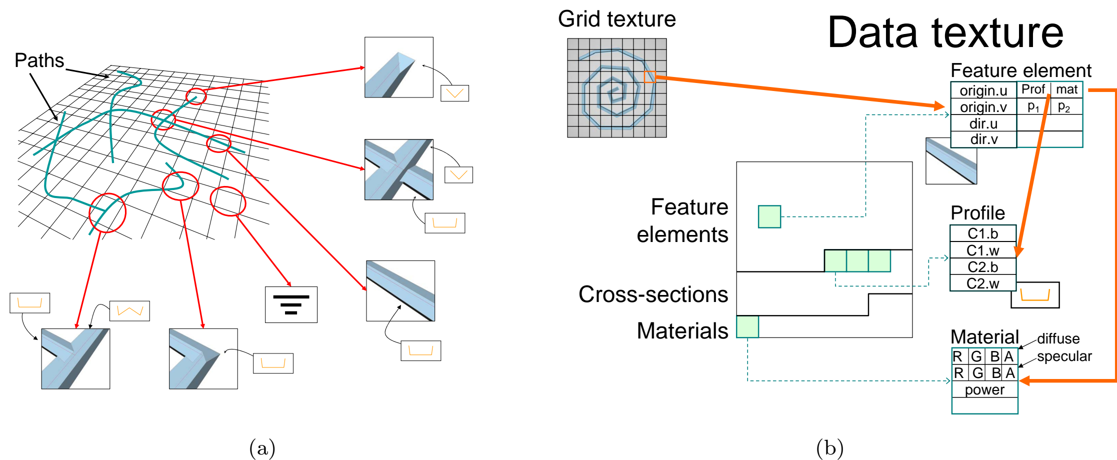
Scratch models simulate small isolated grooves that are individually visible, but not their geometry yet. These models combine a 2D texture with an anisotropic BRDF model. The texture specifies the position and path of each feature over the surface, and is represented by an image with the features painted on it. Examples are the works by Mérillou et al. [MDG01] and Bosch et al. [BPMG04]. Bosch et al. [BPMG08] propose a method to encode and render grooves similar to ours, but their method is much slower than the one presented here and not feasible to be used in real-time applications. Porumbescu et al. [PBFJ05] introduced shell maps, which allow to add arbitrary small-scale surface detail to a triangulated object, but not at interactive rates. Later, [JMW07] introduced techniques that allowed the obtention of interactive frame-rates. The technique presented here is not as general as these, but allows real-time frame-rates to be obtained.

Naturally, details like grooves can also be included in the geometry model of the objects. Such an approach is usually taken for interactive sculpting or editing, and several works can be found dealing with subdivision surfaces or volumetric models. Clearly, our method avoids the fine discretization required by those methods by transferring those evaluations to the pixel shader, and thus lowering bandwidth needs without performing scattered updates to the framebuffer, as would happen with geometry-based approaches.

3 Representing the surface detail

In our approach, details are modeled using a representation based on paths and cross-sections in texture space, similar to the one described by Bosch et al. [BPMG04]. Such a representation is very compact and can be easily applied to any surface having a texture parametrization, without the need of reprojecting the paths between different surfaces. In addition, paths in texture space can easily be defined and evaluated in 2D, thus allowing the application of path-based features onto general objects as traditional texturing techniques.

Here we will assume that the local geometry inside a pixel can be approximated by a set of planar facets, forming a cross-section that is point-wise perpendicular to the main path direction, following the path's local or tangent frame. As long as the assumption of low cross-section variability holds, this approximation is valid. Also, all of our features are defined to lie *below* the object surface. We define another surface, the *base* surface, to be the effective surface the viewer sees. The path is specified as lying on the UV texture plane. If we additionally assume that the curvature of the paths and the surface are locally smooth, we can safely approximate the path by a 2D polyline. See Figure 1(a). The user can specify paths directly in texture space using a piecewise-linear approximation of a more general curve, or in 3D world space by defining the paths onto the object surface and then transforming these onto texture space. Each segment of the polyline representing the path is called a *feature element*. Cross-sections are also modeled using a 2D polyline, and these can penetrate the base surface, protrude from it, or both, but never protrude from the object surface. Our method also handles non-height field features, as shown in Figure 7. Finally, for each feature, the user can also choose specific material properties, which can be necessary when these



(a) (b)

Figure 1: The data structures for representing features (a) The grid texture is used to spatially index each type of possible feature content. (b) Each non-empty entry in the grid texture points to a list of paths elements, which in turn refer to entries in the list of cross-sections and materials.

are different from the ones of the base surface.

3.1 Data structure definitions

In order to determine in run-time which features will be contained in a projected pixel, we first precompute a uniform space subdivision of the UV texture plane. This forms a grid of equally sized cells, where each cell contains a reference to the list of crossing paths, the number of entries in that list, and the maximum height of the features contained within. If no feature crosses the current cell, a null reference is stored. To avoid rendering problems, grid resolution should be chosen so that the cell size is similar to the average feature width. We use a texture for storing this grid, and a second texture for the associated data. See Figure 1. Actually, for correct rendering, each grid cell contains all features that cover an *extended* cell region. This way, when evaluating a feature path that is close to a cell border, we do not need to evaluate nearby cells.

In the data texture, feature elements and their properties are sequentially stored (Figure 1(b)). For each feature element, we store in one texture texel its origin coordinates and the vector to its end in texture coordinates together. In another texel for faster texture access, we store the references to the associated cross-section and material properties, packed on two floating point values. We also store path priorities that will be used for the evaluation of special cases like isolated path endings, intersection ends or corners, see Section 4.4. This way, for each path we need two contiguous texels, which takes advantage of texture cache coherence. The two empty channels of the second texel are used for profile perturbations, as explained in Section 4.5. After all the paths, the data texture contains the cross-sections stored as lists of 2D points, and the material properties. For each material we use up to four floating-point values, usually containing the diffuse and specular colors (each packed as a single floating-point value), the specular power, etc. Notice that the cross-sections and materials are stored separately to avoid duplicates among features, since most of them tend to share these properties.

3.2 Pre-processing steps

The generation of the needed data structures is performed in an off-line pre-processing stage. As a first step we convert every feature element to a quad representing in texture space the feature element extension. The feature element is longitudinal and coincides with the center of the quad, whose width is equal to the feature width. In a second step, the quads are rendered and the lists of feature element identifiers are created by rendering the quads representing feature elements using

the depth-peeling technique [Eve99]. Then the result is retrieved to the CPU for lists creation. The cross-section profiles and the materials are added to the textures at the end of this pre-processing stage. The whole process only takes a few seconds even for the most sophisticated examples we tried.

4 Rendering of path-based features

Here, we present the method to render path-based features, implemented on programmable graphics hardware in order to achieve real-time frame-rates. After explaining the basic setup, we will explain a generalization for profile variations along the paths and antialiasing.

4.1 Finding the features

At rendering time, in the fragment shader units, our algorithm performs a search in texture space for the intersection between the viewing ray (transformed to tangent coordinates) and the features. For that, we narrow down the search space by inspecting only the cells that lie on the 2D projection of the path traveled by the ray, in a similar way to relief mapping [POC05].

The search algorithm we implemented is a simplification of the algorithm explained in [SvG06]. This algorithm needs a start position and a direction. The starting point is the point where the view ray intersects the actual geometry surface, that is, the current UV texture coordinates, and the direction is the viewing ray in tangent space. The algorithm uses a cursor that points to the current position along the view ray. Initially this cursor is at the starting point. Our goal is to advance the cursor as far as possible in each iteration until the intersection with the surface is found. If the ray does not intersect the plane associated with the maximum cell height, the cursor is advanced to the texel boundary, since the intersection is not inside this cell. Then, the algorithm continues directly in the adjacent cell. If the height of the cursor is lower than the stored height, the features are then evaluated for intersection. The process finishes when the ray intersects the geometry in the current cell. If the intersection with a profile happens outside the current cell, it is ignored and the next cell is evaluated.

In the case where the object surface is not flat, the algorithm should be modified to take into account the local curvature, which can be simply done by using the technique described by [OP05] for correct renderings of geometrically-detailed surfaces and silhouettes.

4.2 Evaluating features: simple case

For each cell where the contained geometry must be evaluated for intersection, the process starts by retrieving the corresponding data entry from the grid texture at the current cursor, and looking if that cell contains a feature, more than one or none at all. In the latter case, the ray simply intersects the base surface, while for the other two cases, the features are retrieved from the data texture and evaluated. In any case, if the intersection does not lie in the current cell, the search continues as explained in the previous section. To simplify the following explanations, we will start with the simplest case of only one feature in a cell (a very usual setting for surfaces with only a few features).

When no special case is present, the local geometry at the current cell can be approximated using a 2D cross-section. This greatly simplifies the computations by removing one dimension to the problem, and can be done due to the assumptions stated in Section 3. Once the feature is retrieved and no other features are in the cell, the computations for an isolated feature begin. The profile is projected along the viewing direction onto the horizontal line representing the base surface. See Figure 2. We use these projected points onto the base line, ordered according to the viewing direction, to determine the visible segments from the viewpoint and keeping track of facets that are not occluded in the projected pixel footprint. For point sampling, retrieving the intersection point is a trivial task. For our approximate antialiasing strategy, this information about projected faces is used as explained in Section 4.6.

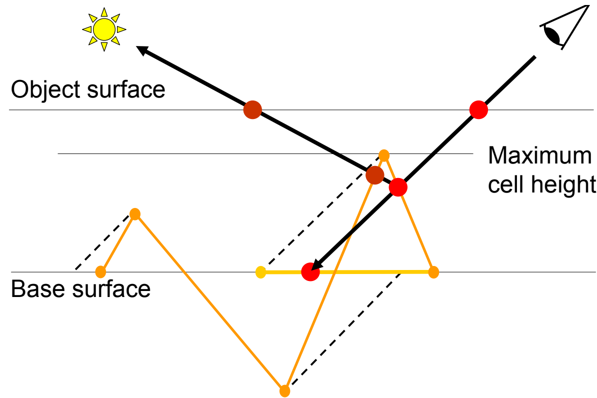


Figure 2: Projection of the feature profile to get the visible facet

Finally, the shadowing computations are performed. The lighting step of these computations repeats the previous steps, but this time with respect to the light source direction. In the case of point sampling, if there is a blocker, i.e. the illuminated facet is different from the visible one, the point is in shadow. Otherwise, it is illuminated and the material at that point is retrieved, the normal is computed from the visible facet coordinates, and shading is finally computed. For the case when an antialiasing approach is used, please refer to Section 4.6.

4.3 Evaluating features: intersecting features

In the case where more than one feature is present in the same cell, computations must be performed to determine the actual intersection. In this section, we will explain the case of intersecting features, leaving for the next section the other cases, like intersected ends, isolated ends, or corners. In order to perform these computations, we use a Constructive Solid Geometry (CSG) analogy, which takes as inputs the intersections for the isolated grooves as described in the previous section, and combines them to find the final intersection result. This method is much more elegant, simpler and faster to compute than the one presented in [BPMG08].

The first thing to do is to independently compute the intersections of the ray with each profile. For every profile, this will generate an odd number of intersections. As the profiles are known in advance, the maximum number of intersections can be pre-computed and used to define the size of the vectors used in the fragment shader to evaluate the features. So, the key idea of our algorithm is to compute the intersections of a ray with each individual profile, and combine the results in order to get the final intersection point.

In this algorithm, we start evaluating the first profile, and, in an iterative process, we evaluate a new profile and combine the result with the result of the evaluation so far.

If we look at Figure 3, we can see an example with two simple features intersecting. If the features have no protruding parts, we can think that the features were built by removing material from the base surface. This is like building the features with Constructive Solid Geometry (CSG), in the sense that we consider we have a CSG tree: from a flat, solid surface, we subtract the volume of each of the features in turn, resulting in holes that can be ray-traced [FvDFH90].

But subtracting features is not sufficient if we can have features with protruding parts, as those paths can not be modeled by subtracting material. Following our CSG analogy, we can think about adding (union) the material for those peaks, and then subtracting the parts of the feature that are below the flat base surface, as can be seen in Figure 4(a). It is important to mention that we must subtract not only the part of the feature that is strictly under the base surface, but we must subtract a whole wedge, from below the surface extending above it. Furthermore, all the additions must be performed before any subtraction to obtain the desired result. In practice, this can be achieved by taking into account that the intersections of a ray and a profile are already sorted by the intersection process itself. With this classification, a ray that intersects a feature is partitioned

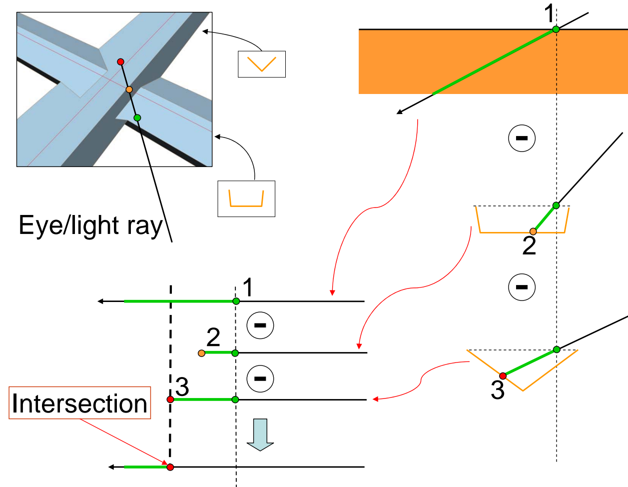


Figure 3: The crossing of two features is considered as the subtraction of the two volumes from the base flat surface (left, above). When ray tracing, we compute its spans through the feature body (right), and we subtract them from the ray path, resulting in the final intersection point (below).

into segments that come from the base surface or addition of material, or segments that come from the subtraction of material. See Figure 4(b). Now, we combine the segments using a special operation that performs a subtraction when either segment is subtracting, otherwise it performs an addition. This procedure is repeated for every groove present in the current cell, subsequently combining their ray segments. At the end, the visible point is the first addition point found.

4.4 Special geometries

For the special cases such as intersected ends (T-shaped endings), isolated ends, or corners, we propose a variation of the previous approach that is based on the use of "priority" flags. In an intersected end, for example, some facets of the non-ending feature (the added one) are prioritized with respect to the feature that ends. These facets are the ones that remain on the opposite side of the end, that is, the non-intersected facets (see left of Figure 5). When evaluating the ray intersections, the ray segments starting at a prioritized facet will always take precedence over the non-prioritized ones, independently if they are adding or removing material. This will produce the complete visibility of the prioritized facets and the simulation of the ending for the other feature.

Concerning isolated ends, these are modeled as a kind of intersected end too: we add an extra feature at the end to represent the facets of the final part of the groove. In this case, we give priority to all the facets of the ending feature, as shown in the middle of Figure 5. Although this also holds for corners, the main difference is that, at corners, the prioritized facets depend on the side where they lie, as can be seen in the right of Figure 5. If facets lie on the external side of the corner they are labeled with priority, independently of the feature where they lie.

All these different priority flags are stored in the data texture, as stated in Section 3. Since priorities always affect one side or another of the feature, we do not store one flag per facet, but a single value that identifies which sides of the feature have priority. According to this value, the cross-section facets are then labeled depending on the side where they lie. For any given feature, the priority refers to the next feature in the current cell list, so that we can handle more than one special case in the same cell.

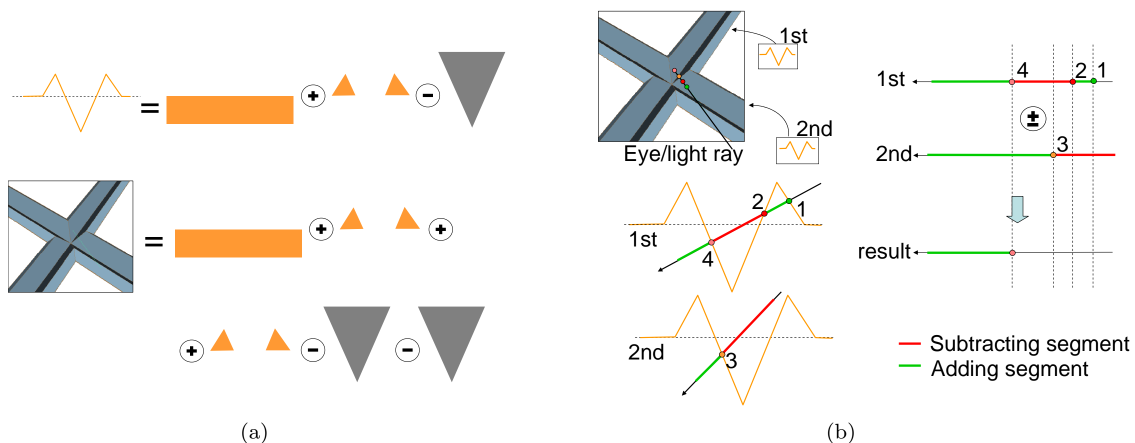


Figure 4: (a) When computing the intersection of features, CSG operations must be ordered and all additions must be performed before any subtraction. (b) When tracing a ray, it is partitioned in segments coming from an addition (red segments) and from a subtraction (green segments). Then, tracing a ray becomes a regular CSG ray-tracing operation.

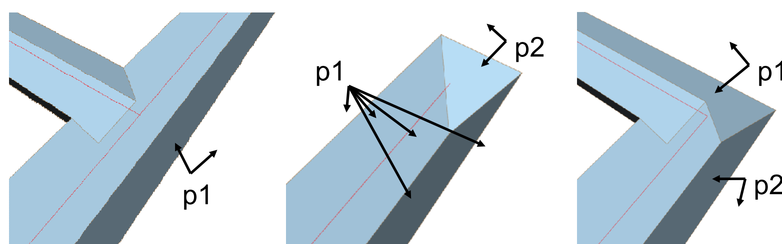


Figure 5: Special situations. From left to right: intersected end, isolated end, and corner. In the figure, the p_i represent the priority of path i.

4.5 Profile perturbations

With a method like the one presented here it is very easy to perform a variation of the path profile along its length by means of a perturbation function. As explained in Section 3, we left two texture channels of the second texel for their usage for this purpose. One interesting and flexible way of doing this is to store the values of a 2D global parametrization of the path into these two entries. This way, for every feature element, we would know the value of this parameter for both ends of the segment. For instance, if the path is parametrized in a way such that the parameter has value 1 in one end and 0 in the other, the path can be reduced in size from full width to zero along its way, as can bee seen in the cracks of Figure 7. This variation can also be associated with a functional expression, like a sinus (top left of Figure 7) or a polynomial one that could be stored in the texture and then be evaluated in rendering time.

4.6 Approximate antialiasing

The method presented so far is intended for point-sampling, which generates aliasing artifacts. We can add an efficient antialiasing, without resorting to A-buffer fragment lists and without extra memory consumption, at the expense of added computational cost.

In order to compute an anti-aliased version of the shader, the first step is to determine the footprint of the pixel in texture space, just as in anisotropic texture filtering. This footprint may

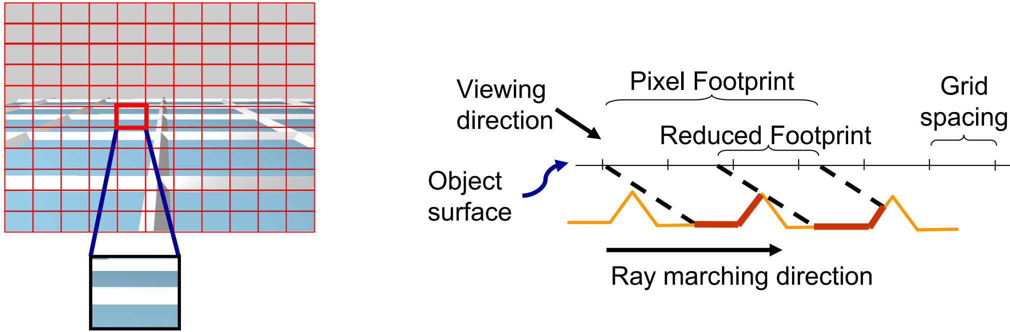


Figure 6: Pixel content is determined in a front to back order, iteratively visiting each 2D profile in turn. Each time an intersection is found, the pixel footprint is reduced proportionally to the covered area and the process continues until the footprint is fully covered.

overlap several cells, which would require the evaluation of multiple texels. The exact solution for this problem was presented in [BPMG08], but their solution is unfeasible for real-time rendering. We decided to implement an approximation by evaluating only those texels lying on the projection onto the object surface of the line defined by tracing a single ray through the viewing pixel. We do this in front to back order, reducing the footprint each time an intersection is computed, see Figure 6. As our cells were extended with nearby information, and we evaluate a reduced number of cells, in our experiments this approximation resulted in renderings without noticeable artifacts for short up to medium distances, or not very grazing angles.

Now, depending on the situation we are in, we take two main approaches we call line-sampling, for isolated grooves, and supersampling for the case of intersecting features (Section 4.3) and the special geometries (Section 4.4). Finally, we also consider the case of mixed situations, where we can see through a pixel both isolated grooves and, for instance, intersecting features.

- **Line-sampling:** As explained in Section 4.2, the simplest case of an isolated groove can be solved in 2D. As explained, we can identify the parts of the feature segments that are visible form the viewpoint. Computing an anti-aliased final color for the pixel footprint is just a sum over all visible feature segments, weighting the color computed for each segment by the relative size of the projected segment. Computing a segment color is as simple as retrieving the corresponding material from the *data texture* and evaluating its BRDF with the normal of the segment in 3D.

- **Supersampling:** Because grid texels encode the list of relevant paths, we can evaluate and combine colors at multiple subpixel samples, without any added bandwidth. Intersecting grooves and special cases require a different approach. In this case, our system evaluates n samples within a parallelogram footprint, and blends them using a weighting filter. In all cases, we evaluate the texel just once, updating all samples as each path element is decoded. This requires allocating a few temporary registers per sample (e.g. accumulated color).

- **Mixed cases:** As we mentioned before, we evaluate only the cells lying on the projection onto the surface of the ray from the eye trough the pixel. For each cell, depending on its content, we apply iteratively one of the two previous techniques, namely line-sampling or supersampling. If we apply line-sampling, we reduce the footprint area proportionally to the ratio of the length of the projected visible segments to the total one-dimensional footprint length. Again, this approximation has proved to be effective, as no noticeable artifacts can be seen in our experiments. If we apply supersampling, the footprint reduction ratio is the number of samples that missed any feature in the current cell (and continued beyond the current cell) to the total sample number.

4.6.1 Antialiased shadowing

Shadowing anti-aliasing is performed much in the same way as mentioned above. From the visibility step we know the parts of the features that are visible. Now, depending on that information, different cases should be taken into account:

- **Line-sampling only:** As mentioned above, when rendering an anti-aliased isolated grove the approach of line-sampling is used. To compute the shadowing of each illuminated segment, we compute the same steps as in Section 4.2 but taking as projection direction the light direction. As we already know which parts of the grooves are seen from the observer, we treat each one of the facets in these parts iteratively to compute the hard shadowed areas and the illuminated ones.

- **Supersampling only:** When treating a case where there is an intersection or a special case, and as the intersections for these cases are computed with a point sampling strategy, shadows are computed exactly the same way: for each sample taken for visibility, we shoot a ray from the light source and check whether there is an intersection between the point we want to shade and the light source. If not, the point is illuminated. Otherwise it is in shadows.

- **Mixed cases:** This is the case of isolated grooves with shadows coming from an intersection or special case, or the other way round. In both cases, we switch to a point-sampling scheme, where a ray is traced between the intersection point (or the extremes of the back-projected lines from line-sampling). It is clear that this sampling scheme represents an approximate estimate, specially for the isolated grooves where a 2D scheme is back-projected to 3D at the center of the pixel footprint. The accuracy of this approximation can be ameliorated by taking more samples at positions obtained by sliding the points along the feature path, and combining the results adequately.

5 Experimental results

Our method has been implemented as a fragment shader using Cg and the OpenGL API. The rendering times for the images are included in Table 1, and correspond to the shader running on a GeForce 8800. As can be seen, this table also includes the memory consumption due to our different textures as well as their resolution.

In Figure 7, we can see some results of the possibilities that this method opens: features with curved paths, features with perturbations along their path, like a sinusoidal variation and a linear one for cracks, one-sided profiles that allow modeling of depressed or protruding surfaces, carved features like text, and features over curved surfaces.

Observe that the images show masking and shadowing effects and different special situations like groove intersections, ends, or corners. Our method allows the correct rendering of all these effects at real-time frame rates. Furthermore, our textures require a low memory consumption: the grid textures used in these examples are very small, going from 25x25 to 125x125 (see Table 1),

Figure	fps	Mem	Text res
7 top left	245.0	49	75x75 / 42x42
7 top middle	56.6	12	25x25 / 24x24
7 top right	156.4	1	2x2 / 8x8
7 bottom left	193.2	64	100x100 / 39x39
7 bottom middle	36.8	50	75x75 / 42x42
7 bottom right	425.6	8	25x25 / 16x16
11 left	127	159	100x100 / 88x88
11 middle	73.7	159	100x100 / 88x88
11 right	169.1	159	100x100 / 88x88

Table 1: Performance of our method for each figure. In this case, rendering times are in frames per second and memory in Kb. The resolution of the two textures is also included.

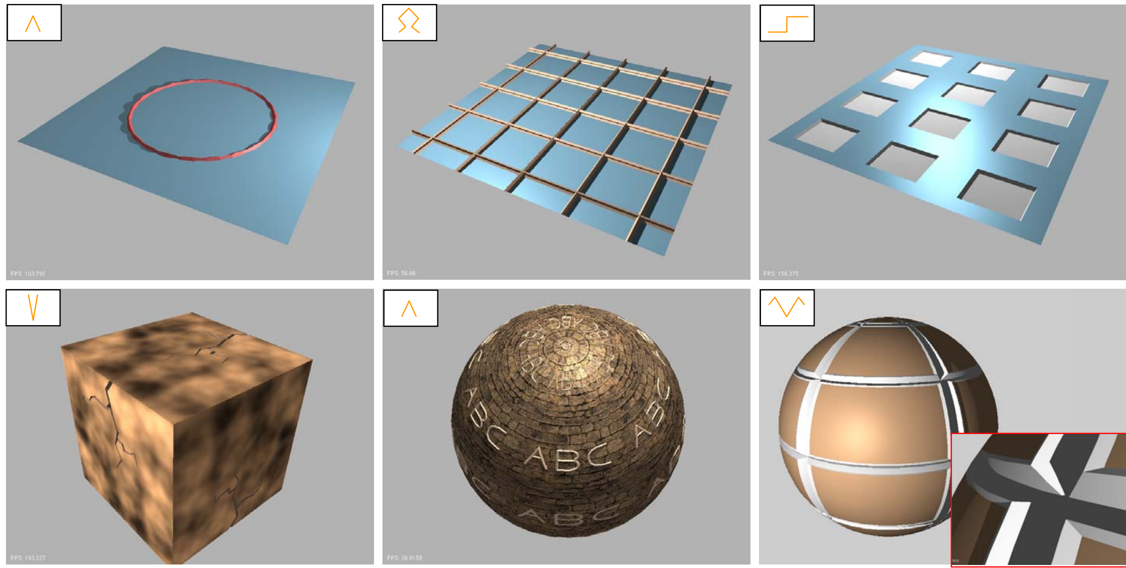


Figure 7: Some examples of path-based features. From top left to bottom right: a curved path with a sinusoidal profile variation, a non-height field profile, a one-sided profile (with a square path), cracks, grooves and complex-profile scratches onto spherical shapes.

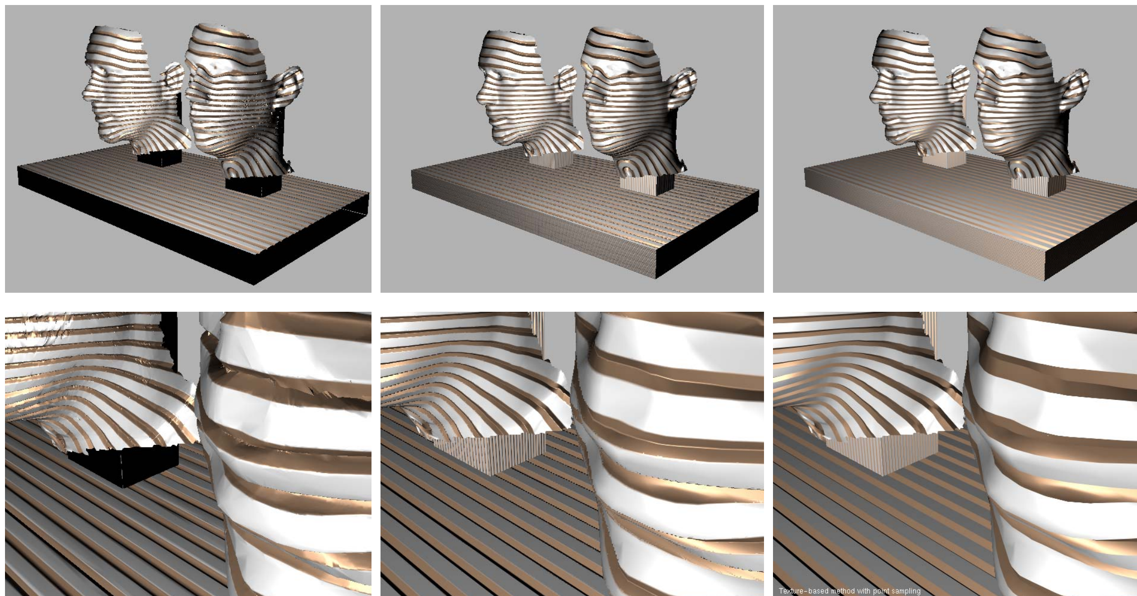


Figure 8: Comparison between geometry (left, 49/50 fps), relief mapping (RM) (middle, 181/599 fps) and our method (right, 198/315 fps). RM uses a 512x512 texture. Geometry: 698107 triangles (Maya feature-based displacement mapping). The other two: 14948 triangles. Timings are for the far(upper)/close(lower) images.

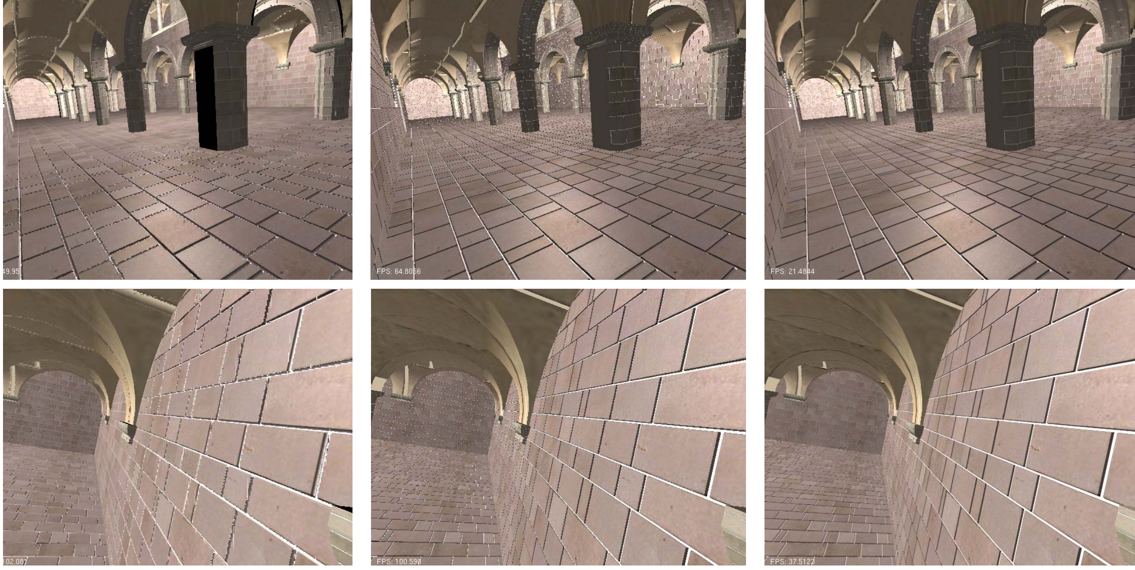


Figure 9: Antialiasing at the Sponza Atrium scene: from left to right, relief mapping (49 fps), 1 sample (65 fps) and our approximate antialiasing scheme (22 fps).

and data textures are even lower. The resolution of the grid texture only determines our efficiency when finding if the current point contains a feature or not: the less features are contained in the cells, the faster becomes the shader. Texture resolution depends on the resolution of the grid, the number of groves and on the pattern or properties of the features. Image-based techniques such as relief mapping would require high resolution textures in order to obtain a similar quality. For example, in Figure 8, we can see the same feature rendered with displacement-mapped geometry, relief mapping and our technique. The amount of geometry needed to get details clearly shows the advantages of texture-based methods. Also, when comparing with relief mapping, even with highly detailed textures, such kind of sharp detail can not be correctly simulated without using a feature-based approach like the one presented here. This can also be seen in Figure 9. Also, observe the good performance of our method in comparison with the other two approaches.

In Figure 9, we can see the Sponza Atrium scene rendered with relief mapping, with $n = 1$ sample per pixel and with our approximate anti-aliasing scheme. Observe how the mip-mapping scheme of relief mapping [POC05] erases the details for medium distances. When comparing with supersampling alone for all cases, we have observed the frame-rate decreases with the number of samples, as the computation has a cost that scales as $O(n)$. It must be mentioned that it is performed entirely on local data, and is therefore amenable to additional parallelism.

Finally, Figures 10 and 11 show two examples of more complex scenes composed of several grooved surfaces, using a square profile (Figure 10 also shows variation in profiles along the feature paths). To simulate the bricks, some of the facets use the material properties of the base surface (bricks) and others the properties of the grooves (mortar). This is specially noticeable in the right image of Figure 11, which represents a close-up of the region shown in the middle image of the same figure. Since several groove surfaces must be processed and these contain many grooves, the frame rates are much lower in this case (see Table 1). Nevertheless, the timings are fast enough and the interactivity is not lost.

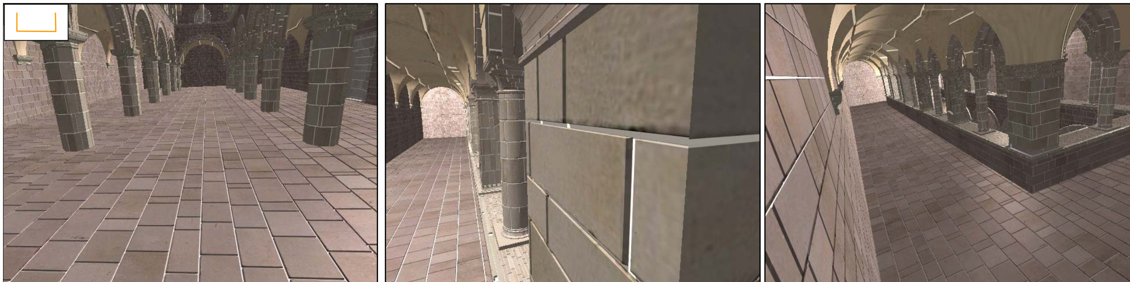


Figure 10: Sponza Atrium rendered from different viewpoints using our approach to simulate the bricks. Frame-rates are, from left to right: 25, 42.4, 39.9 fps (with approximate antialiasing).

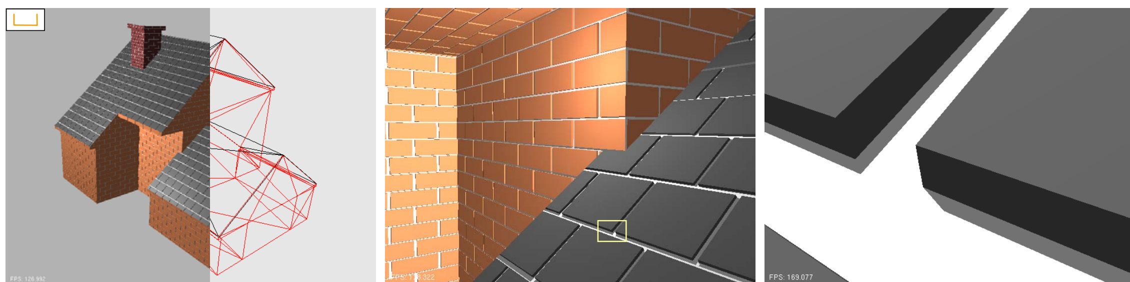


Figure 11: House rendered in real-time from different viewpoints using our approach to simulate the bricks. The underlying mesh is shown on the left.

6 Discussion and limitations

As seen above, the complexity of each grid cell depends linearly on the number of features crossing that cell, so complexity can be arbitrary and can be introduced only where strictly needed. Also, the time needed by each fragment shader depends on this number of features, so empty cells are really quick to evaluate.

The anti-aliasing strategy described in Section 4.6 relies on simplification assumptions that break down when the full pixel footprint grows larger than a few texels. The ideal solution would be, then, to resort to some sort of mipmapping strategy. Unfortunately, and to the best of our knowledge, up to now there is no closed solution for the problem of filtering with local occlusion, shadowing and masking [HSRG07]. Research in this problem clearly is beyond the scope of this paper. Nevertheless, it is possible to switch to some sort of normal or BRDF distributions and use the solutions presented in [HSRG07] and [IT08], although these solutions would ignore occlusions and thus would result in an unrealistic result. The same would happen if we switch to a Relief Mapping solution and use the mipmapping described in [POC05].

It is important to analyze the influence of the grid size in the requirements and performance of this method. As expected, as the grid increases resolution, lists of paths for each texel will get shorter in average, but at an increased total memory cost. For certain texels, however, this length has a lower bound, as a texel of any size that covers the intersection of, let's say, 4 scratches, will have (at least) a length of 4 entries. Although reducing list average length means fewer evaluations at each grid cell, the increase in grid resolution also implies more cells to evaluate in the search (specially for more grazing angles), and this behavior dominates over the reduction in list sizes. This can be seen in the Figure 12. Other search strategies can be implemented, like the ones described in [LKU08].

Also, when a pixel footprint grows larger than the overlap region, we could switch to a texture

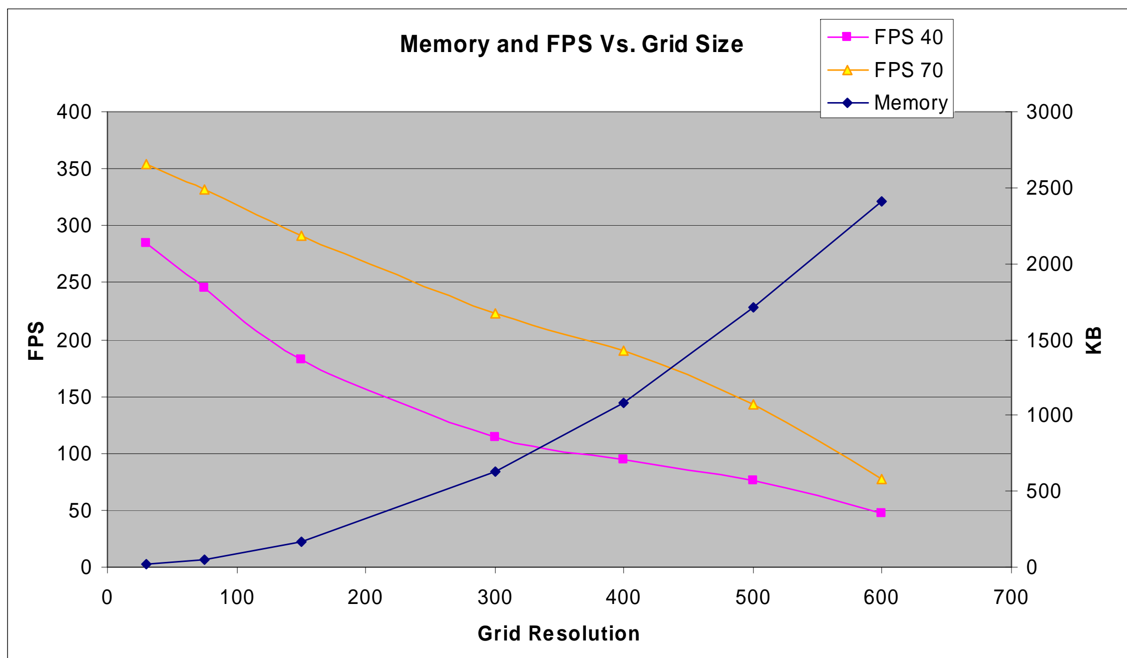


Figure 12: Memory requirements and framerate as a function of grid resolution for two viewing angles (40 and 70 degrees) for the top left image in Figure 7.

that stores the parameters of a BRDF based on an anisotropic model [KS00]. At points with features, they would store anisotropic parameters according to the current path. At the other points, they would store isotropic parameters corresponding to the surface reflection. These parameters could then be used to compute the BRDF during the rendering stage.

Framerates are clearly dependent on the number and complexity of the cells with intersecting features. The bigger the number of cells with intersecting features, or the more features intersect in those cells, the slower the evaluation would be. However, as can be seen above, our algorithm provides high framerates even for really complex layouts.

One important limitation to mention is that the textures used by our method need to be precomputed, which greatly difficult the editing of the features or their properties. Thus, it is clear that having animated features would be almost impossible without changing dynamically both textures.

7 Conclusions

In this paper, we present a method to compute path-based features that can model scratches, features and other kind of similar phenomena. We have successfully generated several patterns, including non-height field patterns, bricks, cracks, flat depressions and patterns with perturbations along their lengths. The method can be successfully applied even onto curved surfaces without modifications. Also, it has efficient antialiasing, and does not present extra memory consumption, at the expense of added computational cost. Even with many samples, framerates remain interactive. We can conclude that the method is both robust and accurate, providing surface detail with a quality only possible with geometry. It behaves much better in terms of quality than other techniques like Relief Mapping.

8 Acknowledgements

The Sponza Atrium is courtesy of Marko Dabrovic. This work was founded by grant TIN2007-67120 from the Ministerio de Educación y Ciencia, Spain.

References

[Bli78] James F. Blinn. Simulation of wrinkled surfaces. In *Computer Graphics (Proceedings of SIGGRAPH 78)*, volume 12, pages 286–292, August 1978.

[BPMG04] Carles Bosch, Xavier Pueyo, Stéphane Mérillou, and Djamchid Ghazanfarpour. A physically-based model for rendering realistic scratches. *Computer Graphics Forum*, 23(3):361–370, September 2004.

[BPMG08] Carles Bosch, Xavier Pueyo, Stéphane Mérillou, and Djamchid Ghazanfarpour. A resolution independent approach for the accurate rendering of grooved surfaces. *Computer Graphics Forum (Pacific Graphics 2008)*, 27(7), 2008.

[Eve99] C. Everitt. Interactive order-independent transparency, 1999. White paper, NVIDIA Corporation.

[FvDFH90] James D. Foley, Andries van Dam, Stephen K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, 2nd edition, 1990.

[HSRG07] Charles Han, Bo Sun, Ravi Ramamoorthi, and Eitan Grinspun. Frequency domain normal map filtering. In *SIGGRAPH '07: ACM SIGGRAPH 2007 papers*, page 28, 2007.

[IT08] R. Brooks Van Horn III and Greg Turk. Antialiasing procedural shaders with reduction maps. *IEEE Transactions on Visualization and Computer Graphics*, 14(3):539–550, 2008.

[JMW07] Stefan Jeschke, Stephan Mantler, and Michael Wimmer. Interactive smooth and curved shell mapping. In *Rendering Techniques 2007 (Proceedings Eurographics Symposium on Rendering)*, pages 351–360, 6 2007.

[KS00] Jan Kautz and Hans-Peter Seidel. Towards interactive bump mapping with anisotropic shift-variant BRDFs. In *2000 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 51–58, August 2000.

[LB05] Charles Loop and Jim Blinn. Resolution independent curve rendering using programmable graphics hardware. *ACM Trans. Graph.*, 24(3):1000–1009, 2005.

[LH06] Sylvain Lefebvre and Hugues Hoppe. Perfect spatial hashing. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 579–588, 2006.

[LKU08] LzlSzirmay-Kalos and Tam Umenhoffer. Displacement mapping on the gpu - state of the art. *Computer Graphics Forum*, 27(1), 2008.

[Lov05] Jörn Loviscach. Efficient magnification of bi-level textures. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Sketches*, page 131, 2005.

[MDG01] Stéphane Mérillou, Jean-Michel Dischler, and Djamchid Ghazanfarpour. Surface scratches: Measuring, modeling and rendering. *The Visual Computer*, 17(1):30–45, 2001.

[NH08] D. Nehab and H. Hoppe. Random-access rendering of general vector graphics. *ACM Transactions on Graphics*, pages 2008.

[OP05] Manuel M. Oliveira and Fábio Policarpo. An efficient representation for surface details. In *UFRGS Technical Report RP-351*, 2005.

[PBFJ05] Serban D. Porumbescu, Brian Budge, Louis Feng, and Kenneth I. Joy. Shell maps. *ACM Trans. Graph.*, 24(3):626–633, 2005.

[POC05] Fábio Policarpo, Manuel M. Oliveira, and João L. D. Comba. Real-time relief mapping on arbitrary polygonal surfaces. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, pages 155–162, 2005.

[PRZ05] E. Parilov, I. Rosenberg, and D. Zorin. Real-time rendering of normal maps with discontinuities. Technical Report TR2005-872, NYU, 2005.

[QMK06] Zheng Qin, Michael D. McCool, and Craig S. Kaplan. Real-time texture-mapped vector glyphs. In *SI3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 125–132, New York, NY, USA, 2006. ACM Press.

[QMK08] Zheng Qin, Michael D. McCool, and Craig Kaplan. Precise vector textures for real-time 3d rendering. In *SI3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, pages 199–206, New York, NY, USA, 2008. ACM.

[Sen04] Pradeep Sen. Silhouette maps for improved texture magnification. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 65–73, 2004.

[SvG06] M. F. A. Schroders and R. v. Gulik. Quadtree relief mapping. In *GH '06: Proceedings of the 21st ACM SIGGRAPH/Eurographics symposium on Graphics hardware*, pages 61–66, 2006.

[Tat06] Natalya Tatarchuk. Dynamic parallax occlusion mapping with approximate soft shadows. In *SI3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 63–69, 2006.

[TC04] Jack Tumblin and Prason Choudhury. Bixels: Picture samples with sharp embedded boundaries. In *Rendering Techniques 2004: 15th Eurographics Workshop on Rendering*, pages 255–264, June 2004.

[TC05] Marco Tarini and Paolo Cignoni. Pinchmaps: textures with customizable discontinuities. *Comput. Graph. Forum*, 24(3):557–568, 2005.

[WWT+03] Lifeng Wang, Xi Wang, Xin Tong, Stephen Lin, Shimin Hu, Baining Guo, and Heung-Yeung Shum. View-dependent displacement mapping. *ACM Transactions on Graphics*, 22(3):334–339, July 2003.