Real-Time Path-Based Surface Detail

Carles Bosch, Gustavo Patow

Grup de Geometria i Gràfics
Universitat de Girona
E-17071 Girona, Spain

Abstract

We present a GPU algorithm to render path-based 3D surface detail in real-time. Our method models these features using a vector representation that is efficiently stored in two textures. First texture is used to specify the position of the features, while the second texture contains their paths, profiles and material information. A fragment shader is then proposed to evaluate this data on the GPU by performing an accurate and fast rendering of the details, including visibility computations and antialiasing. Some of our main contributions include a CSG approach to efficiently deal with intersections and similar cases, and an efficient antialiasing method for the GPU. This technique allows application of path-based features such as grooves and similar details just like traditional textures, thus can be used onto general surfaces.

Key words: surface detail, real-time rendering, vector graphics

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 [1], displacement mapping [2], or relief mapping [3, 4], 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 [5] [6], but are limited to flat 2D representations without encoding other 3D information besides normal map perturbation techniques [7]. Visibility

Email addresses: cbosch@ima.udg.edu (Carles Bosch), dagush@ima.udg.edu (Gustavo Patow)

Figure 1: The Knight Champion rendered from different viewpoints and distances using our approach (left images, 59 $\sim$ 78 fps) and relief mapping (right images, 57 $\sim$ 81 fps) to simulate the armor engravings. Observe how our anti-aliasing strategy correctly reproduces the grooves visibility while relief mapping produces noticeable artifacts (insets).

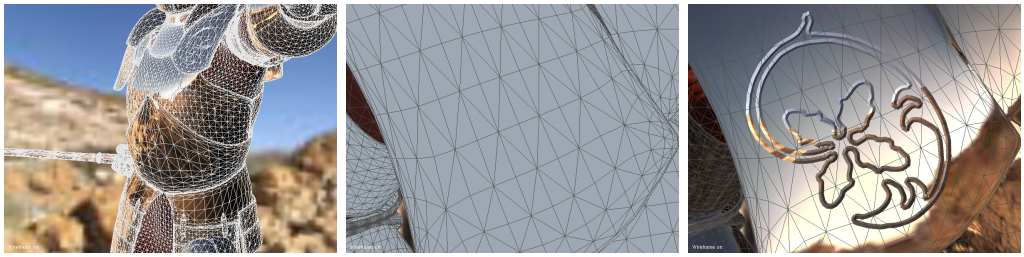


Figure 2: The presented method is capable to visualize geometry details like scratches, cracks, grooves and extremely sharp edged features without the amount of geometry needed to get the details. This clearly shows the advantages of texture-based methods. Left: the full model. Middle: the base geometry. Right: the generated detail.

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 (See figure 1). The presented method is capable 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: Our real-time method computes 3D geometric detail 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). See Figure 2. We use techniques derived from the usage of vector textures in the GPU to store the geometry and properties of the features, and evaluate them in real-time. The first texture is a grid that specifies the positions of the features, 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: The new method presented here is the first real-time approach to present 3D vector-based surface detail other than flat textures. In particular, it allows accurate visualization of path-based features, although this could be extended to other features as well. We also introduce a CSG analogy that is both flexible and powerful. The visualization 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 region-sampling and supersampling. Both techniques are used in combination to solve both visibility and shadowing anti-aliasing issues.

Limitations: Our path-based feature representation shares a few limitations with other vector-based representations [6]. 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 texels it overlaps, but in our experience there is almost no storage overhead. Also, each texel may have a different number of features, thus requiring an indirection scheme to avoid data sparseness. Finally, we require the features and the object surface where they are applied to have low curvature, in order to obtain correct visibility computations.

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 [1], dis-

placement mapping [2], relief mapping [3] or parallax mapping [8, 4], among others. For an in-depth survey on displacement mapping techniques on the GPU, please refer to [9]. These macro-geometry models suffer from resolution problems and are not able to correctly simulate high frequency or very close details. Compared to these approaches, the method presented here addresses these issues in an efficient and natural way, as can be seen in Figure 1.

Our method is also related to real-time vector graphics, which always have had a great appealing because of their seamless scaling capabilities. In [10], 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 texel: a few line segments [11] [12] [13], an implicit bilinear curve [14], two quadratic segments [7], or a fixed number of corner features [5]. Also, Parilov et al. [7] 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 texel representation that allows for patterns of arbitrary complexity, having none of the above mentioned restrictions. Our approach stores feature paths in a way similar to [5, 6], 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 visible but where their geometry is imperceptible. These models combine a 2D texture with an anisotropic BRDF model. The texture specifies the position of each scratch, while the BRDF is used to compute their reflection. Examples are the works by Merillou et al. [15] and Bosch et al. [16]. Recently, an extension has been proposed to deal with more general path-based features, which removes the limitations on the size of the features or their geometric cases (e.g. intersections) [17]. Our method is based on a similar idea, but our rendering techniques are GPU-friendly, which results in real-time frame rates for a similar rendering quality.

Porumbescu et al. [18] introduced shell maps, which allow to add arbitrary small-scale surface detail to a triangulated object, but not at interactive rates. Later, [19] 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 approach is usually taken for interactive sculpting or editing. 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. Overview

In this work we represent 3D geometric detail with a continuous representation based on paths and cross-sections in texture space. This information is stored in two textures: the first one is a grid overlaid on the surface features, where each cell provides the positions of the features themselves and references the second texture. This second texture contains the geometry and properties of the feature path profiles, and material information. As mentioned, the proposed method does not need additional geometry to represent these surface details.

At runtime, our algorithm performs a search in texture space for the intersection between the viewing ray (transformed to tangent coordinates) and the features, sequentially evaluating the contents of each texel along the projected ray. In order to perform an accurate and fast rendering of the features, we use a Constructive Solid Geometry (CSG) analogy to compute the intersection between each viewing ray and the features in each texel. Evaluation of profile perturbations are taken into account, and shadowing is computed by tracing rays in texture space from the light source to the intersection point of the viewing ray and the features. To improve rendering quality, two antialiasing approaches are also proposed: a region sampling strategy for isolated features and a supersampling approach for multiple cases.

4. Representing the Surface Detail

4.1. Detail Representations

In our approach, details are modelled using a representation based on paths and cross-sections in texture space, similar to the one described by Bosch et al. [17]. Such a representation is very compact and can be easily applied to any surface having a texture parametrization, without the need of reprojecting the features 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.

In our implementation, we support paths formed both by linear and/or quadratic segments. These paths are specified as lying on the UV texture plane. See Figure 3. The user can specify paths directly in texture space by providing a sequence of 2D points or in 3D world space by defining the paths onto the object surface and then transforming these onto texture space. The obtained set of points define a sequence of linear and/or quadratic segments (called *feature elements*). Another alternative is to generate the engravings directly by vectorizing an artist-provided displacement map. See Figure 1 and Section 6.

All of our features are defined to lay strictly *below* the object surface, so we implicitly define another surface, the *base* surface, to be the effective surface the viewer sees. This base surface lays below the object surface. See Figure 4. Here we assume that cross-sections may be approximated by a set of planar facets, forming a profile that is point-wise perpendicular to the main path direction and following the path's tangent frame. Cross-sections can thus be modelled using a 2D polyline, and these can penetrate the base surface and/or protrude from it. This kind of representation can easily handle non-height field features as well. See Figure 8. Finally, for each feature, the user can also choose specific material properties, which can be necessary when these are different from the ones of the base surface. Also, in Figure 8 (middle and right in the bottom row) we can see that our algorithm correctly handles cases where the object surface is not flat by tracking an approximate curvature along the ray [20]. For the situation where the ray crosses a chart boundary, ray tracing can continue as described in [21].

4.2. Data Structure Definitions

In order to store all the previous information for its efficient evaluation into the GPU, we use two textures, as mentioned before. The first one specifies the location of the different features and serves as a uniform grid, where each texel contains a reference to the list of crossing features, the number of entries in that list, and the maximum height of the features contained within. If no feature crosses the current texel, a null reference is stored. These lists are then stored in the second texture called *data texture*, as explained below. See Figure 3. Actually, for correct rendering, features are extended prior to determine which texels are covered. This way, when evaluating a feature that

is close to a texel border, we do not need to evaluate nearby texels, especially during antialiasing, as explained in Section 5.7.

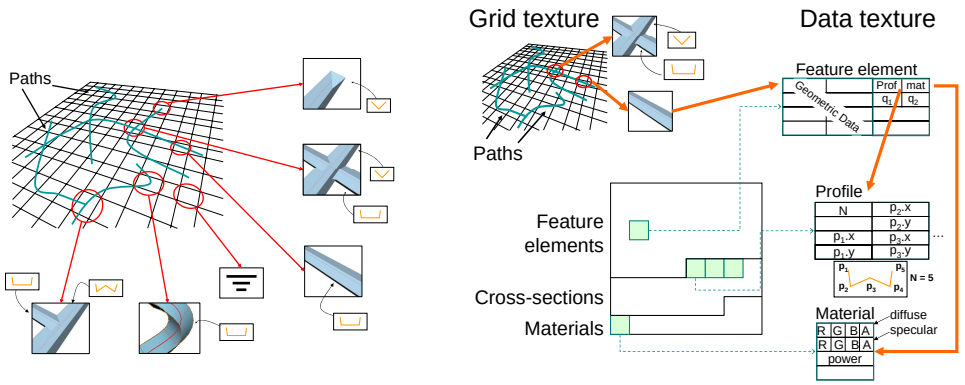


Figure 3: Data structures for the features: the grid texture is used to spatially index each type of possible feature content. Each non-empty entry points to a list of feature elements, which in turn refer to entries in the list of cross sections and materials.

In the data texture, the lists of features and their properties are sequentially stored (Figure 3). At this point, each feature element (segment) is considered as an independent feature, so in fact, we only need to store feature elements in the lists. For each feature element, depending if its linear or quadratic (needing a 1-bit flag to choose), we store different information. For linear elements we store together in one texture texel its origin and the vector to its end coordinates, while for quadratic elements we store its three defining vertices (plus weights, if needed) in two texels [6]. In another following texel, we store the references to the associated cross-section and material properties, packed on a single floating point value. We also store path priorities that will be used for the evaluation of special cases like feature ends or intersections, see Section 5.4. This way, for each path we need only two or three contiguous texels, which takes advantage of texture cache coherence. The two empty channels of the last texel may be used for profile perturbations, as explained in Section 5.5.

After all this data, the data texture contains the cross-sections and the material properties. Cross-sections are stored as lists of 2D points, with the number of points included at the beginning of each list (see Figure 3(b)). 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.

To avoid rendering problems, grid resolution should be chosen so that the texel size is similar to the average feature width, see Section 6. In our experiments, texture resolutions for the grid texture ranged from 2^2 to 250^2 for the more complex examples (the Knight Champion in Figure 1). See Table 1 for more details.

4.3. 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 some representative geometry. For linear feature elements, we use a quad representing in texture space the feature element extension. In this case, the feature element is longitudinal and coincides with the center of the quad, whose width is equal to the (extended) feature width. In the case of quadratic arcs, for each curve we generate a quad that bounds the curve and we render it in texture space using the method described in [6].

In a second step, the quads associated with the feature elements are rendered and the lists of feature elements are created by rendering these quads using the depth-peeling technique [22]. 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. In any case, this behavior is linear in the number of feature elements, so the pre-processing cost is reasonably small.

Notice that the previous data is always stored in blocks of four values, which represents a single RGBA texel in the texture. Our objective is to use as few texels as possible, so that fewer texture accesses are needed at runtime, one of the expensive operations in a GPU program.

5. 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 as well as our antialiasing extensions.

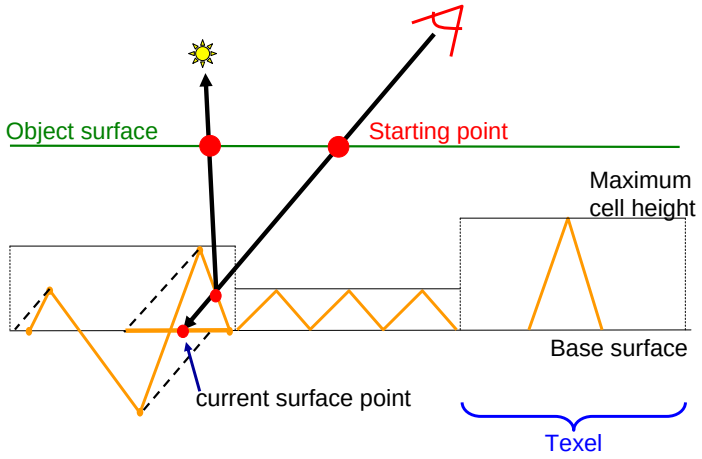


Figure 4: Projection of the feature profile to get the visible facet

5.1. Finding the Features

At rendering time, in the fragment shader units, our algorithm starts by performing 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 texels that the ray actually follows in the first texture: The texels laying on the 2D projection of the path travelled by the ray, in a similar way to relief mapping [3], but without skipping any texel.

The search algorithm we implemented is a simplification of the algorithm explained in [23]. The algorithm needs a start position and a direction. The starting point is the point where the view ray intersects the object surface (i.e., the current UV texture coordinates), and the direction is the direction of the viewing ray in tangent space. See Figure 4. 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 base surface or a feature is found. So, if the ray does not intersect the plane associated with the maximum texel height, the cursor is advanced to the texel boundary, since the intersection is not inside this node. Then, the algorithm continues directly in the adjacent node. If the height of the cursor is lower than the maximum texel height, the features inside that texel are then evaluated for intersection. The process finishes when the ray intersects the geometry in

the current texel. Of course, if the intersection happens outside the current texel boundaries, it is ignored and the next texel is evaluated. We can see the pseudo-code for this operation at the Algorithm 1, where $featureHeights$ is the value into the grid texture that stores the maximum height of the features for each texel, $intersection(ray, features(cursor))$ is the algorithm for the intersection computations between the ray and the features, which we will describe in the following sections, and $outOfBounds(cursor)$ is a function that evaluates whether the ray is outside the grid texture limits or not.

Algorithm 1 Feature computations

```
1: cursor = (u,v)
2: while not outOfBounds(cursor) do
3:   if intersection(ray, next_texel).height ≥ featureHeights(cursor)
     then
4:     Advance cursor to next texel
5:   else
6:     if intersection(ray, features(cursor)) then
7:       return cursor
8:     else
9:       Advance cursor to next texel
10:    end if
11:  end if
12: end while
13: return no intersection found
```

In the case where the object surface is not flat, the algorithm should be modified for correct renderings. To take into account the local curvature, it can be simply done by modifying the direction of the viewing ray at each visited texel by taking into account the shape of the object. For instance, this can be done by simple tracking an approximate curvature along the ray, as described by [20], or by also taking into account the stretching introduced by the parameterizations, as done in [24].

5.2. Evaluating Simple Cases

For each texel 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 texel contains features or not (i.e. the entry is valid or not). If no features are found, the ray

simply is verified for intersection with the base surface, while for the former the features are retrieved from the data texture and evaluated. In any case, if the intersection does not lay in the current texel, the search continues as explained in the previous section. See pseudocode at Algorithm 2, where *mat* stands for material and N for the surface normal.

Algorithm 2 Intersection computations (ray, features)

```
1: Retrieve feature attributes from grid
2: if There are features at cursor then
3:    if Isolated feature then
4:       ProcessIsolatedFeature(cursor, features) → mat,N
5:    else
6:       ProcessIntOfFeatures(cursor, features) → mat,N
7:    end if
8: else
9:    Compute ray/base-plane intersection → mat,N
10: end if
11: Compute shading(mat,N) → color
```

To simplify the following explanations, we will start with the simplest case of only one feature in a texel (a very usual setting for surfaces with only a few features). In the next section we will extend this explanation for the case where more feature elements are present in the same texel.

In this simpler case, the local geometry at the current texel can be approximated using a 2D cross-section. This greatly simplifies the computations by removing one dimension to the problem, and does not introduce significant errors as long as the local feature curvature or the cross-section's perturbation is not high [17]. Once the feature element is retrieved along with its cross-section, the computations for the isolated feature begin. For intersecting a single ray with the 2D profile of a feature element, we should project the ray onto the local cross-section plane and intersect it with each profile segment. This, however, can be simplified by using a point-sampling adaptation of the algorithm explained in [17]. Using this approach, we simply project each profile facet onto the base surface according to the ray direction, and evaluate each segment against the current surface point, which can be done with simple 1D operations. The surface point is here represented by the distance between the intersection of the ray with the base surface and

the current path direction, as depicted in Figure 4. By sequentially evaluating the facets according to the ray direction, notice that we automatically deal with any possible occlusion between the facets. For our approximate antialiasing strategy, this information about projected facets is then used as explained in Section 5.7. For the case of a curved feature, visibility is computed in the same way by locally approximating the feature as a straight one. For this, we simply need to determine its current tangent direction by computing the closest point on curve to that point with the algorithm described in [6].

5.3. Evaluating Intersecting Features

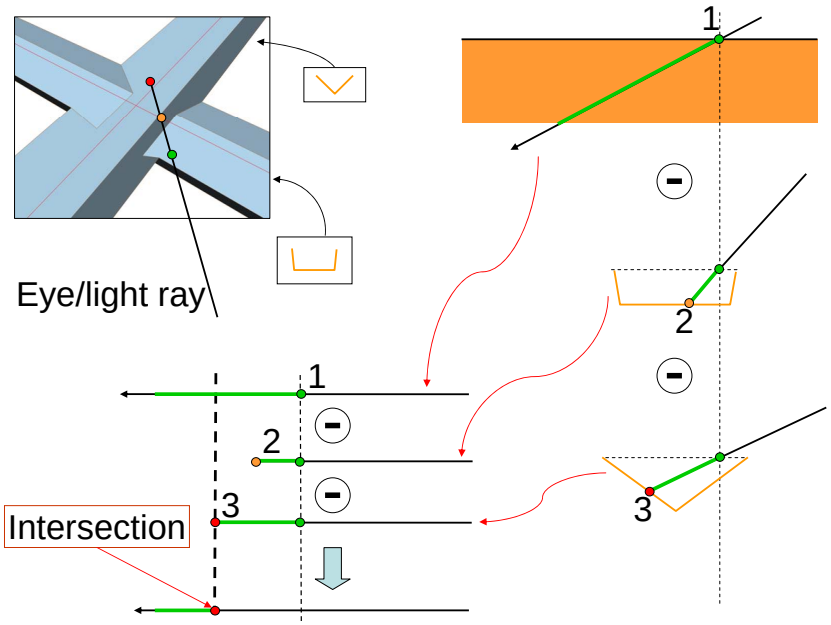


Figure 5: Crossing of two features is considered as the subtraction of the two volumes from the basic flat surface (left, above). When ray tracing, we compute its spans through the feature body (right), and we subtract them from the ray path, resulting in the final intersection point (below).

In the case where more than one feature is present in the same texel, computations must be performed to determine their actual intersection, if any. In this section we will explain the case of multiple features (parallel or

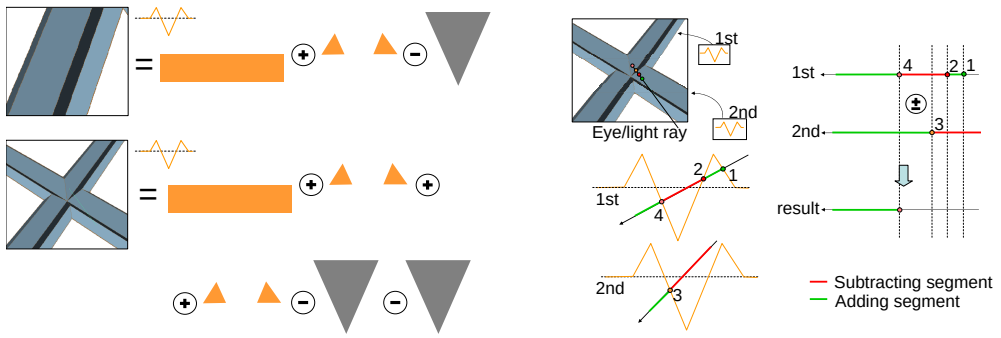
intersecting), leaving for the next section other cases like intersected ends, isolated ends, or corners. In order to perform these computations we use a Constructive Solid Geometry (CSG) analogy, which takes as input the intersections between the ray and each individual feature profile, and combines them to find the final intersection result. This method is much more elegant, simpler and faster to compute than the one presented in [17], see Section 5.2. If we look at Figure 5, we can see an example with two simple intersecting features. 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 [25].

The first thing to do is to compute the intersections of the ray with each profile independently of each other, using the algorithm described in the previous section. In this case, however, we need to compute all the intersection points, not only the first one. 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. We start evaluating every profile in an iterative process, combining the intersections of the ray with the current profile with the result so far (according to the CSG operation). This is shown in Algorithm 3.

Algorithm 3 ProcessIntOfFeatures($cursor, features$)

```
1: Get data and profile for first feature
2: Project 1st profile and find all the intersection points → intSegs
3: while there are features to evaluate do
4:    Get data and profile for next feature
5:    Project next profile and find all intersection points → intSegs'
6:    Combine(intSegs, intSegs') → intSegs
7: end while
8: return firstPoint(intSegs)
```

But subtracting material is not sufficient if we can have protruding parts. See Figure 6. In those case, and following our CSG analogy, we can think about adding (union) the material for peaks, and then subtracting the parts of the feature that are below the base surface. It is important to mention that we must subtract not only the part of the feature that is strictly under



(a) (b)

Figure 6: (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 additions (red segments) and subtractions (green segments). Ray tracing becomes a regular CSG operation.

the base surface, but subtract a whole wedge, from below the surface and extending above it. See Figure 6(a). 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. We simply need to sequentially classify the ray segments as adding or removing material after each intersection. Initial segments only need to be classified if intersecting an internal profile facet, which is done as subtractions to simulate the wedge described before. See Figure 6. After that, we propose to combine the segments using a special CSG 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 texel, subsequently combining their ray segments. At the end, the visible point is the first addition point found. In our current implementation, the assignment of internal and external faces is manually done, but it is not difficult to do it automatically by starting on both extremes and classify every face towards the center as external, until the normal of the feature face changes its orientation with respect to the direction of classification.

5.4. Special Geometries

Other situations such as the ones depicted in Figure 7 can be evaluated in a similar way by considering more CSG operations. Intersected ends,

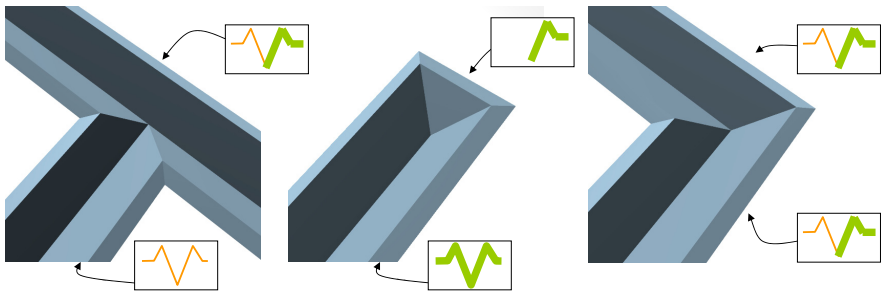


Figure 7: Special situations. From left to right: intersected end, isolated end, and corner. In the figure, green parts have a priority assigned.

for example, can be seen as a regular intersection between features followed by two extra CSG operations, in order to recover the added/removed parts beyond the end (see left image of Figure 7).

When evaluating intersections using our approach of combining ray segments, note that each segment has a priority related to the order of the corresponding CSG operation. Hence, material subtractions have priority over additions because these are performed later. Using the same idea, intersected ends are simulated by simply giving higher priority to the ray segments belonging to the recovered portion (right side of the feature in Figure 7 left). During ray-profile intersections, this means that ray segments need to be classified as additions or subtractions as before, but giving more priority to the ones intersecting the prioritized side of the profile, if any. During the ray combination, priorities will be used in the same way, but considering an extra priority level. Regarding isolated ends (Figure 7 middle), we first need to consider an extra feature perpendicular to the main one in order to process this as an intersection. This feature only uses half the original profile, and both profiles are prioritized so that we can obtain the desired result. A similar procedure can be applied for corners, as shown in Figure 7 right.

The different priorities associated to the features are previously stored in the data texture, as stated in Section 4. During a pre-processing step, we first determine which special cases are contained in each cell, and then assign the corresponding priorities to each feature. Since priorities always affect one side or another of a feature profile, we only need to specify which side of the feature has priority, i.e. which profile portion needs to be recovered. Priorities can thus be efficiently stored as a single tag along with the

feature element data, stating if its profile will have left, right, both-sides or no priority. In order to handle different situations in the same cell (e.g. a feature forming a corner and an isolated end after that), we actually store two priorities for each feature element: one related to the previous feature in the ordered feature list and another with the next feature (see q_1 and q_2 in Figure 3(b)). These priorities will be properly used when combining the ray segments from two consecutive features.

5.5. Profile Perturbations

With a method like the one presented here it is very easy to perform a variation of the feature profile along its path by means of a perturbation function. As explained in Section 4, we left two texture channels for this purpose. One interesting and flexible way of doing this is to store the values of a 2D global parameterization 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 parameterized 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 8. This variation can also be associated with a functional expression, like a sinus (top left of Figure 8) or a polynomial one that could be stored in the texture and then be evaluated in rendering time.

5.6. Shadowing

Once the intersection of the ray and the path-based detail is found, shadowing computations are performed. The lighting step of these computations repeats the previous steps, but this time with respect to the light source direction. For this, the visible point is first re-projected onto the object surface according to the light direction. If after repeating the process there is a blocker, i.e. the illuminated facet is different from the visible one, the point is in shadow. If not, 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.

5.7. Approximate Antialiasing

The method presented so far is intended for point-sampling, which generates aliasing artifacts. In this section we explain how to include an efficient antialiasing both for the direct visualization and for shadowing. This method

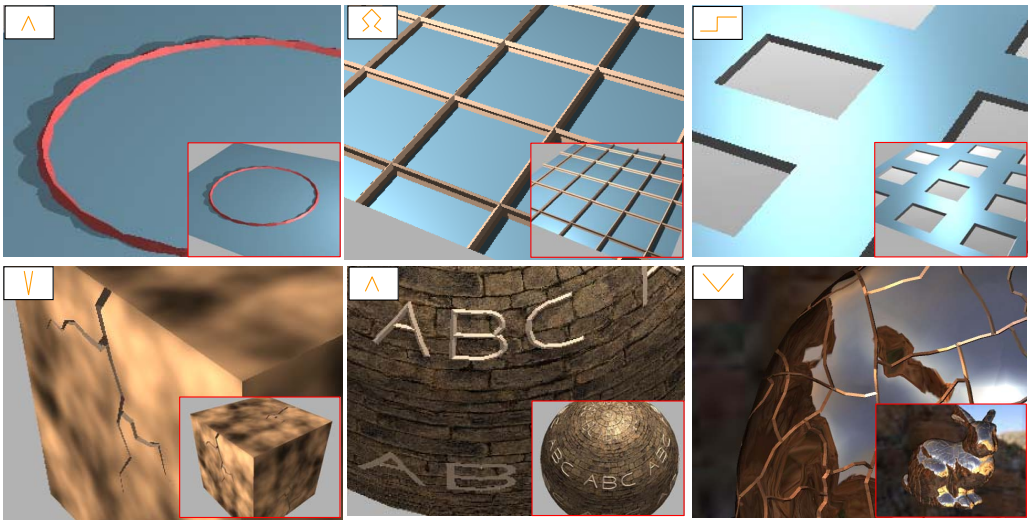


Figure 8: Some examples of path-based features rendered with our point sampling approach. From top left to bottom right: a curved path with a sinusoidal profile variation, a non-height field profile, a one-sided profile (with a square path), cracks, protruding letters and complex-profile scratches onto spherical shapes.

works 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 done in anisotropic texture filtering. This footprint, however, may overlap several texels, which would require the evaluation of the multiple features contained within. The exact solution for this problem was presented in [17], but their solution is unfeasible for real-time rendering. We decided to implement an approximation by evaluating only those texels traversed by the current ray, as before. However, the intersection with the features is now done by taking into account the pixel footprint. Each time an intersection is computed, the footprint is subsequently reduced and the traversal continues until the entire footprint has been processed, as explained below, and depicted in Figure 9. As our texels were extended with nearby information (See Section 4.2), in our experiments this approximation resulted in renderings without noticeable artifacts for short up to medium distances, and not very grazing angles.

Now, depending on the situation we are in, we take two main approaches to intersect the footprint with the features. One we call it region-sampling,

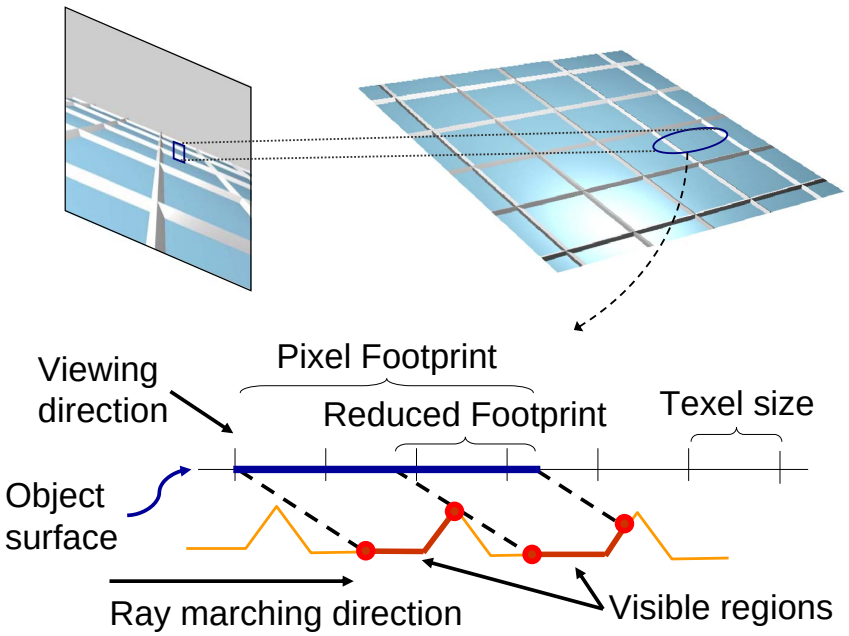


Figure 9: Pixel content is determined in a front to back order, iteratively visiting each 2D profile. Each time an intersection is found, the pixel footprint is reduced proportionally to the covered area and the process continues until the footprint is fully covered.

for isolated features, and then supersampling for the case of intersecting features (Section 5.3) and other special geometries (Section 5.4). Finally, we also consider the case of mixed situations, where we can see through a pixel both isolated grooves and, for instance, intersecting features.

- **Region-Sampling:** As explained in Section 5.2, the simplest case of an isolated groove can be solved in 2D by means of its profile. The main difference is that now we need to consider all the visible profile facets contained in the footprint, not only the one intersecting with the viewing ray. This will require the evaluation of the different visible segments, which can be easily accomplished in 1D by projecting the profile onto the base surface, as done during point sampling (see Section 5.2). Computing an anti-aliased final color for the pixel footprint is then just a sum over all visible feature segments, weighting the color computed for each segment by the relative size of the projected segment. Note that during the shadowing step, the shadowed portions of these facets should be removed in order to obtain a correct result. Since storing each previous visible segment for its later shadowing test would be costly, we decided to only store the points delimiting each visible profile region (see Figure 9). By reprojecting these points during the shadowing step, we can then easily determine which facets are both visible and illuminated for the final color computation. This algorithm is somehow similar to the line sampling method proposed in [17], but with this last part being more suitable for a GPU implementation.

- **Supersampling:** For feature intersections and ends, our CSG method can not be easily extended with region sampling. In those cases, we better resort to a supersampling strategy, where the footprint is evaluated using different samples and blended by means of a weighting filter. In order to compute these samples, the texture is only traversed once, and in case where a texel contains a special case, the samples are evaluated as the features are decoded. This is acceptable as the grid texture is computed using extended features, as explained above, and requires less computations than a full supersampling strategy. In our examples, we use 4 samples taken halfway between the pixel center and its corners and average them using a simple box filter, which tends to give acceptable results. In [17], antialiasing on groove intersections and ends where analytically evaluated using a polygonal footprint projected on each feature facet, but this would be clearly unfeasible for the GPU.

Figure	fps	Mem (kb)	Texture resolution
1 left	57.7	535	250x250 / 137x137
1 middle	59.5	535	250x250 / 137x137
1 right	78.1	535	250x250 / 137x137
8 top left	245.0	30.6	50x50 / 42x42
8 top middle	56.6	12	25x25 / 24x24
8 top right	156.4	1	2x2 / 8x8
8 bottom left	193.2	64	100x100 / 39x39
8 bottom middle	36.8	50	75x75 / 42x42
8 bottom right	84.4	198	100x100 / 101x101

Table 1: Performance of our method for different figures. Rendering times are in frames per second and memory in kilobytes. The resolution of the two textures is also included.

- **Mixed cases:** During a ray traversal, texels laying on the ray's path may require the use of region sampling or supersampling according to the case. If we apply region sampling, the footprint area is proportionally reduced to the ratio of the length of the projected visible regions to the total footprint length. If we apply supersampling, the footprint reduction ratio is the number of samples that missed any feature in the current texel (and continued beyond the current texel) to the total number of samples. Dealing with mixed cases, however, may become difficult during the shadowing step, especially when mixing visible regions with points. In those cases, its simpler to consider visible regions as point samples and evaluate them using supersampling as well. These cases only happen in some especial situations, and will result in a similar quality than the one obtained at intersections and ends. Note that region sampling is far more accurate and fast than using supersampling, so whenever is possible, is better to use this technique.

6. 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 8 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 pro-

files that allow modelling of depreciations or protruding surfaces, protruding features like text, and features over curved surfaces.

These images where rendered using our point-sampling method. Observe that the images show masking and shadowing effects and different special situations like feature intersections, ends, and 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 100x100, and data textures are even lower. See Table 1.

In Figure 1, an example of a more complex scene is shown, this time rendered using our antialiasing scheme. The engravings in the Knight Champion armor were generated directly from the displacement map the artist provided with the model. This was done by vectorizing the image (with Inkscape) and drawing the segments in texture space for the generation of the textures (with a Maya plug-in). The obtained textures require a higher resolution due to the complexity of the pattern (see Table 1). This, however, should be compared with the requirements of relief mapping [3], which required textures of 2048^2 to properly represent this kind of features. Even with highly detailed textures, note that such kind of sharp detail can not be correctly simulated without using a feature-based approach like the one presented here (see mid and close images in Figure 1).

This can also be seen in the Sponza Atrium scene of Figure 10. In this case, we compared different antialiasing options fot both our method and relief mapping. Observe how the mip-mapping scheme of relief mapping erases the details for medium distances, which is also clearly visible in Figure 1. Also observe the quality of our method with and without applying antialiasing. Our antialiasing scheme requires more computations but the results are clearly better. Using point sampling everywhere instead of region sampling, the quality would be worst and the framerate would be lower (17.24 fps with 4 samples per pixel). In Figure 11 we can see the Spoza Atrium scene with $n = 1$, 2 and 4 samples per pixel. With supersampling, the frame-rate decreases with the number of samples, as the computation has a cost that scales as $O(n)$, being n the number of samples used. It must be mentioned, however, that it is performed entirely on local data, and is therefore amenable to additional parallelism. With region sampling, supersampling is only required at intersections and ends, thus improving the general performance.

In Figure 12, our method is also compared against both an explicit geometric representation of the features and xNormal's instanced tessellation im-

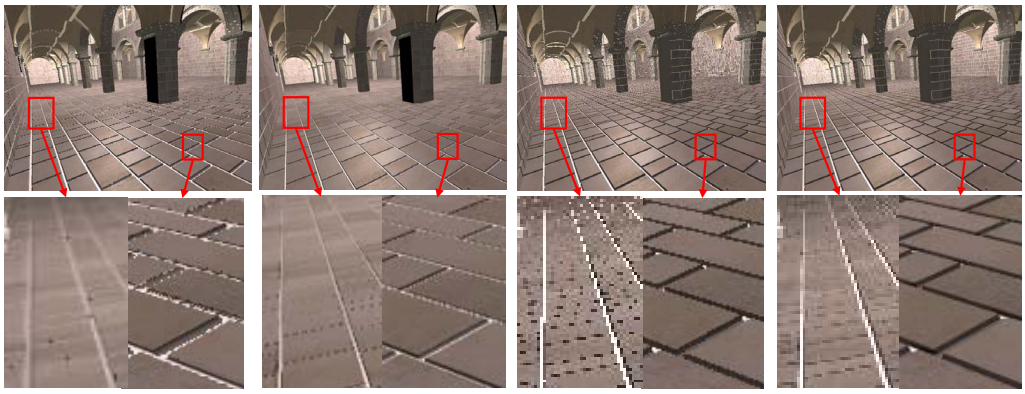


Figure 10: Comparison of techniques for the Sponza Atrium: from left to right, relief mapping (resolution=1024^2, 51.8 fps), relief mapping with 4× Full Screen Anti-Aliasing (512^2, 33.1 fps), 1 sample (512^2, 64.3 fps) and our approximate antialiasing scheme (512^2, 33.6 fps).

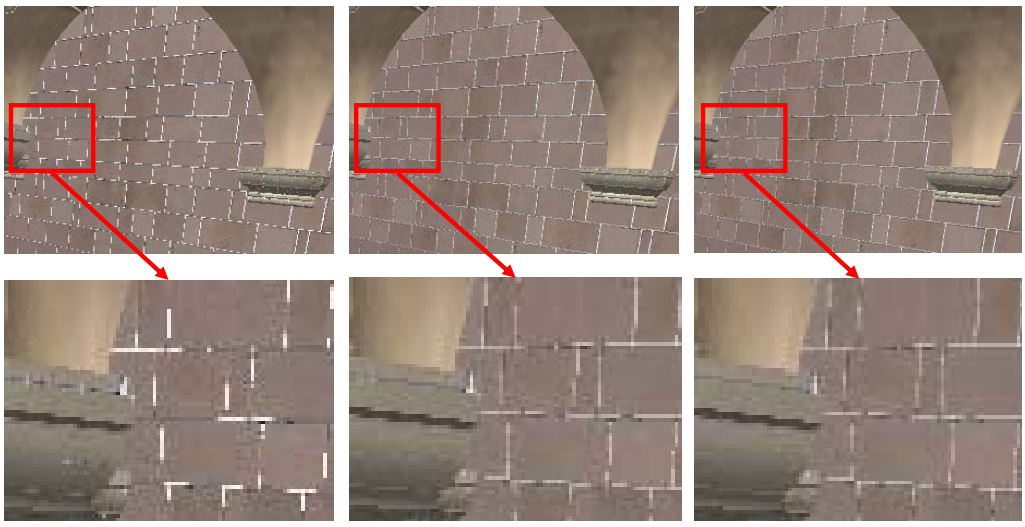


Figure 11: Antialiasing at the Sponza Atrium scene: from left to right, 1 sample (65.3 fps), 2 samples (29.0 fps) and 4 samples per pixel (15.9 fps) respectively.

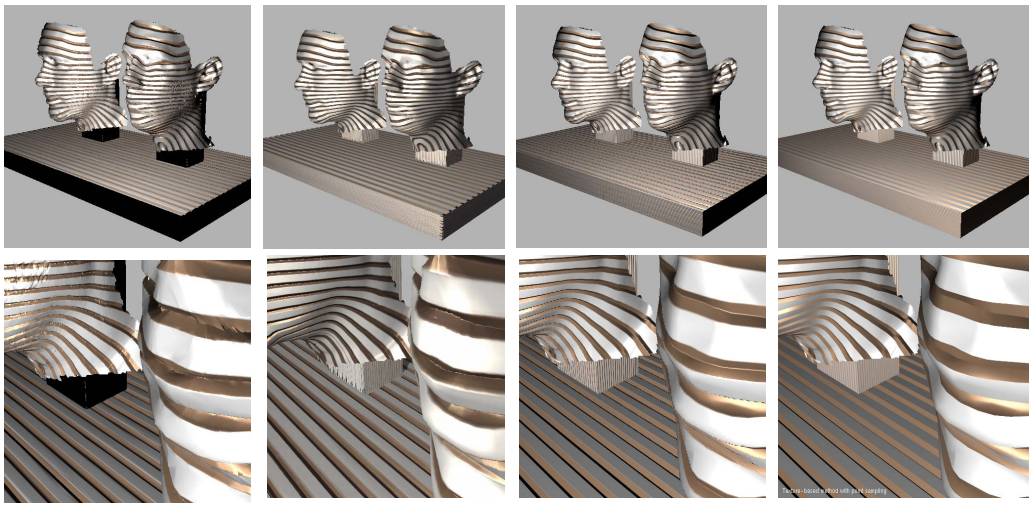


Figure 12: Comparison between geometry (left, 49/50 fps), instanced tessellation (second column, 60/57 fps), relief mapping (RM) (third column, 181/599 fps) and our method (right, 198/315 fps). RM uses a 512x512 texture. Geometry: 698107 triangles (Maya feature-based displacement mapping), tessellation in the geometry shader (second column). The other two: 14948 triangles. Timings in parenthesis are for the far and the close images, respectively.

plementation [26]. Observe the quality and good performance of our method in comparison with the other approaches, specially because of the sparseness of the features represented. Geometric representations greatly depend on the quality and subdivision of the original model. Furthermore, the number of triangles tend to rapidly grow, especially for curved features or surfaces. Relief mapping-based techniques, in general, tend to be faster for representing surface detail than our method, but our method is able to represent such detail much more accurately and at a fraction of the storage cost, while maintaining reasonable frame-rates. Finally, in Figure 13, we can see that our method provides similar results to those presented in [17], but at a much higher frame-rate: the method presented here is more than 1500 times faster than the original one. The most notorious visual differences are at intersections and other isolated cases, but the differences are negligible when taking into account the frame-rates. Also, the detail is preserved even with a low resolution texture, so the memory cost remains small.

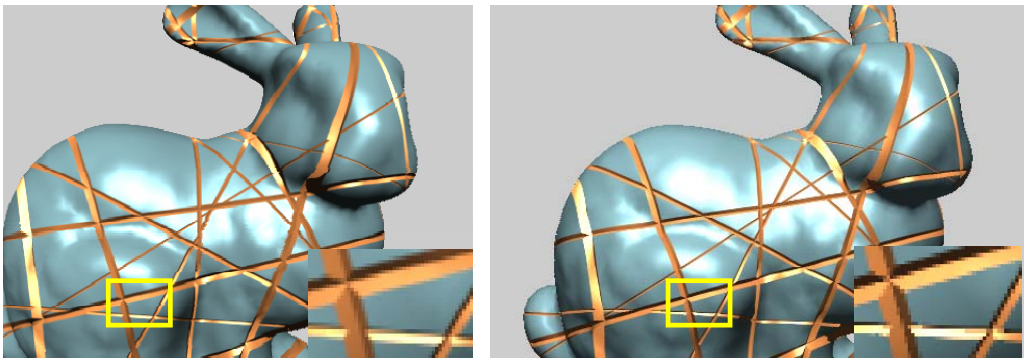


Figure 13: Comparison between the method proposed in [17], at 0.03 fps (right) and our method running at 47 fps (left). We can see that the proposed method outperforms the original one with similar quality results.

7. Discussion and Limitations

As seen above, the complexity for evaluating each grid texel depends linearly on the number of features crossing that texel, so complexity can be arbitrary and it is possible to control to introduce it only where strictly needed. Also, the time needed by each fragment shader depends on this number of features, so empty texels are really quick to evaluate. As in [17], isolated features are faster to evaluate than multiple cases with feature intersections or ends, but the latter tend to be very localized in most common patterns. Our approaches, however, are faster to evaluate and may be applied to any feature represented by a CSG tree.

The anti-aliasing strategy described in Section 5.7 relies on simplification assumptions that may 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 [27]. 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 [27] and [28], or a solution based on an anisotropic model [29], 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 approach described in [3].

It is important to analyze the influence of the grid size in the requirements and performance of this method. 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 texels, the faster becomes the shader. Texture resolution depends on the resolution of the grid, the number of features and on the pattern or properties of the features. As expected, as the grid increases resolution, the lists of features 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 features, will have (at least) a length of 4 entries. Although reducing list average length means fewer evaluations at each texel, the increase in grid resolution also implies more texels to evaluate in the search (specially for more grazing angles), and this behavior dominates over the reduction in list sizes for large grid resolutions. On the other extreme, a low resolution grid means many features to be verified at each texel, thus also lowering performance. Obviously, there is an equilibrium point that maximizes performance, as can be seen in Figure 14. As we mentioned earlier, this equilibrium point is when the grid size is similar to the sizes of the represented features.

One important limitation to mention is that the textures used by our method need to be precomputed, which greatly difficults edition or animation of the features or their properties. Thus, it is clear that having animated features would be almost impossible without changing dynamically both textures, but this limitation is common to vector-based representations [6].

From Figure 8 (middle and right in the bottom row) we can see that our algorithm behaves correctly in the case where the object surface is not flat, by simple tracking an approximate curvature along the ray, as described by [20]. Other solutions like taking into account the stretching introduced by the parameterizations, as done in [24], could also be used.

8. Conclusions

In this paper we have presented a method to compute path-based features that can model many different surface details. We have successfully generated several patterns, including non-height field patterns, bricks, cracks, flat depreciations and patterns with perturbations along their path lengths. The method can be successfully applied even onto curved surfaces. Also, it has efficient antialiasing, and does not present extra memory consumption. Even

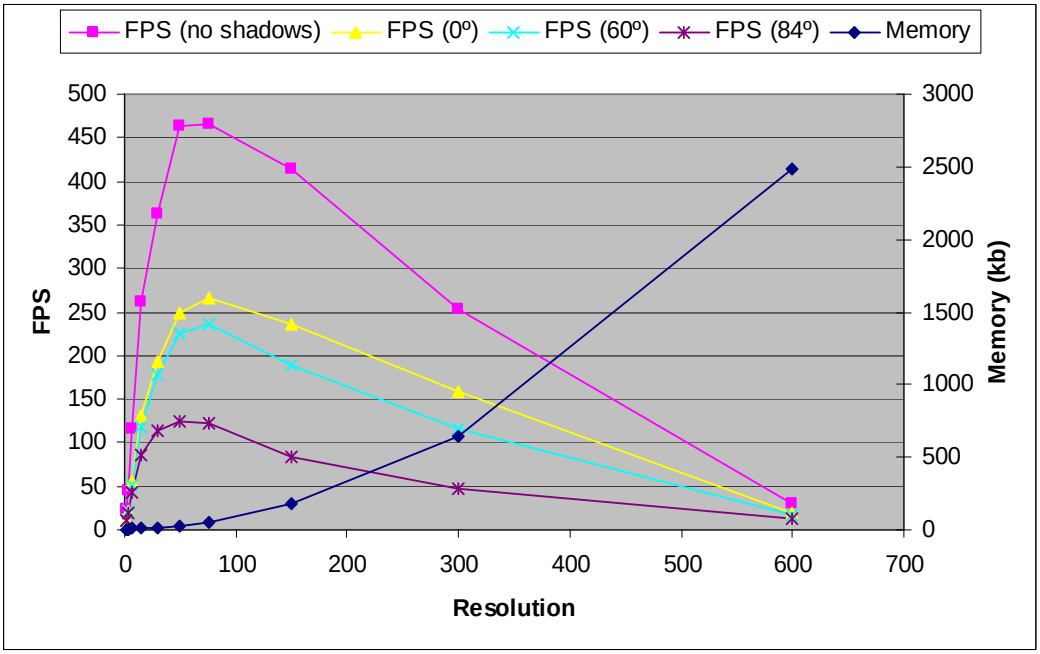


Figure 14: Memory requirements (in kilobytes) and framerate (in frames per second) as a function of grid resolution for different angles for the top left image in figure 8.

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 but without its overhead. In addition, it behaves much better in terms of quality than other techniques like Relief Mapping.

9. Acknowledgements

Sponza Atrium is courtesy of Marko Dabrovic, Knight Champion model by DAZ 3D (www.daz3d.com). This work was funded with grant TIN2007-67120 from the Ministerio de Educación y Ciencia, Spain.

References

References

[1] J. F. Blinn, Simulation of wrinkled surfaces, in: Computer Graphics (Proceedings of SIGGRAPH 78), Vol. 12, 1978, pp. 286–292.

[2] L. Wang, X. Wang, X. Tong, S. Lin, S. Hu, B. Guo, H.-Y. Shum, View-dependent displacement mapping, ACM Transactions on Graphics 22 (3) (2003) 334–339.

[3] F. Policarpo, M. M. Oliveira, J. L. D. Comba, Real-time relief mapping on arbitrary polygonal surfaces, in: Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games, 2005, pp. 155–162.

[4] N. Tatarchuk, Dynamic parallax occlusion mapping with approximate soft shadows, in: SI3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games, 2006, pp. 63–69.

[5] Z. Qin, M. D. McCool, C. Kaplan, Precise vector textures for real-time 3d rendering, in: SI3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games, 2008, pp. 199–206.

[6] D. Nehab, H. Hoppe., Random-access rendering of general vector graphics, ACM Transactions on Graphics (2008)

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

[8] T. Kaneko, T. Takahei, M. Inami, N. Kawakami, Y. Yanagida, T. Maeda, S. Tachi, Detailed shape representation with parallax mapping, in: In Proceedings of the ICAT 2001, 2001, pp. 205–208.

[9] L. Szirmay-Kalos, T. Umenhoffer, Displacement mapping on the gpu - state of the art, Comput. Graph. Forum 27 (1).

[10] C. Loop, J. Blinn, Resolution independent curve rendering using programmable graphics hardware, ACM Trans. Graph. 24 (3) (2005) 1000–1009.

[11] P. Sen, Silhouette maps for improved texture magnification, in: HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, 2004, pp. 65–73.

[12] J. Tumblin, P. Choudhury, Bixels: Picture samples with sharp embedded boundaries, in: Rendering Techniques 2004: 15th Eurographics Workshop on Rendering, 2004, pp. 255–264.

[13] S. Lefebvre, H. Hoppe, Perfect spatial hashing, in: SIGGRAPH '06: ACM SIGGRAPH 2006 Papers, 2006, pp. 579–588.

[14] M. Tarini, P. Cignoni, Pinchmaps: textures with customizable discontinuities, Comput. Graph. Forum 24 (3) (2005) 557–568.

[15] S. Mérillou, J.-M. Dischler, D. Ghazanfarpour, Surface scratches: Measuring, modeling and rendering, The Visual Computer 17 (1) (2001) 30–45.

[16] C. Bosch, X. Pueyo, S. Mérillou, D. Ghazanfarpour, A physically-based model for rendering realistic scratches, Computer Graphics Forum 23 (3) (2004) 361–370.

[17] C. Bosch, X. Pueyo, S. Mérillou, D. Ghazanfarpour, A resolution independent approach for the accurate rendering of grooved surfaces, Computer Graphics Forum (Pacific Graphics 2008) 27 (7).

[18] S. D. Porumbescu, B. Budge, L. Feng, K. I. Joy, Shell maps, ACM Trans. Graph. 24 (3) (2005) 626–633.

[19] S. Jeschke, S. Mantler, M. Wimmer, Interactive smooth and curved shell mapping, in: Rendering Techniques 2007 (Proceedings Eurographics Symposium on Rendering), 2007, pp. 351–360.

[20] M. M. Oliveira, F. Policarpo, An efficient representation for surface details, in: UFRGS Technical Report RP-351, 2005.

[21] F. González, G. Patow, Continuity mapping for multi-chart textures, ACM Transactions on Graphics 28 (5) (2009) 109.

[22] C. Everitt, Interactive order-independent transparency, white paper, NVIDIA Corporation (1999).

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

[24] Y.-C. Chen, C.-F. Chang, A prism-free method for silhouette rendering in inverse displacement mapping, Comput. Graph. Forum 27 (7) (2008) 1929–1936.

[25] J. D. Foley, A. van Dam, S. K. Feiner, J. F. Hughes, Computer Graphics: Principles and Practice, 2nd Edition, Addison-Wesley, 1990.

[26] S. Orgaz, xnormal, http://www.xnormal.net/ (2007).

[27] C. Han, B. Sun, R. Ramamoorthi, E. Grinspun, Frequency domain normal map filtering, in: SIGGRAPH '07: ACM SIGGRAPH 2007 papers, 2007, p. 28.

[28] R. B. Van Horn III, G. Turk, Antialiasing procedural shaders with reduction maps, IEEE Transactions on Visualization and Computer Graphics 14 (3) (2008) 539–550.

[29] J. Kautz, H.-P. Seidel, Towards interactive bump mapping with anisotropic shift-variant BRDFs, in: 2000 SIGGRAPH / Eurographics Workshop on Graphics Hardware, 2000, pp. 51–58.