Real Time Scratches and Grooves

Carles Bosch, Gustavo Patow
Grup de gràfics de Girona
Universitat de Girona
17071 Girona
[cbosch,dagush]@ima.udg.es

Abstract

We present a GPU-accelerated algorithm to render surface detail features like scratches and grooves in real time. Our method efficiently models these features using a continuous representation that is stored in two textures. The first texture plays the role of a hash and provides pointers to the second texture containing the scratches paths, profiles and material information. This data is evaluated by a fragment shader at 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 and accelerating computations.

1 Introduction

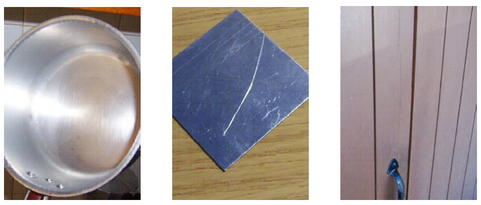


Figure 1: Pictures of real surfaces exhibiting a different type of groove. From left to right: a polished pan with lots of micro-grooves, a scratched surface with isolated scratches, and a door with big grooves between the wooden planks.

Grooves and scratches are common surface features that are present in lots of real world objects, especially on those obtained by the human process or manufacture. See Figure 1. They can be typically found, for example, on engraved objects, polished metals, or assembled/tiled surfaces. The main characteristic of a groove is its shape, which is often represented by means of a cross-section and a path over the surface [MDG01a, CGF04]. From now on, we will use the words scratch and groove in an interchangeable manner.

Up to now, real time visualization of grooves and scratches has been limited to the generation and usage of geometry or sampled data structures as in bump mapping [Bli78], displacement mapping [Coo84, WWT^{+}03], or relief mapping [POC05, Tat06], which show aliasing problems for close views.

In this paper, we describe a new, fast method to compute geometric detail like scratches and grooves in real time, by using a continuous representation that is stored in two textures, without relying on additional geometry. The first texture plays the role of spatial hash, providing pointers to the second texture which contains the scratches 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.

This paper is organized as follows. The next Section presents the Previous work, and in Section 3 the new method for representing real time scratches and grooves is presented. Finally, in Section 4 our results are presented and reviewed, and in Section 5 we show some insights into promising future work.

2 Previous Work

Concerning scratches and grooves, previous methods can be classified into three categories, depending on the type or size of the groove that is simulated: anisotropic BRDF models, scratch models, and macro-geometric models.

Anisotropic models simulate very small grooves distributed along the overall surface, which are not individually perceptible but provide an homogeneous anisotropic aspect to the surface, as on polished surfaces. Most of them are empirical models that are chosen when the micro-geometry is unknown [War92, Ban94]. Physically-based models are also available, but only for certain types of micro-geometry [PF90, Sta99]. For general types of periodic micro-geometry, brute force methods have been also proposed [Kaj85, WAT92].

Scratch models, on the other hand, 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 scratch over the surface, and is represented by an image with the scratches painted on it. The BRDF then models the light reflection on each scratch point. To compute this reflection, Becket and Badler [BB90] simply apply a random intensity to each scratch, while Merillou et al. [MDG01b] use a physically-based BRDF that takes into account the cross-section of each scratch, but they are limited to specific shapes. This problem was addressed in [BPMG04], which introduced algorithms to take into account different microgeometries and variations of the profiles along the paths.

Finally, macro-geometric models use general techniques that allow the simulation of different kinds of surface details, such as bump mapping [Bli78], displacement mapping [Coo84, WWT$^+$03], or relief mapping [POC05, Tat06], among others. However, these macro-geometry models are rarely suitable for small or distant details, requiring good antialiasing or filtering methods, which can be very time consuming.

To account for grooves of all sizes in a same image, a common solution is to perform smooth transitions between different representations of the surface features, one for each scale or distance. Becker and Max [BM93] address the transition among displacement mapping, bump mapping, and BRDF. Another solution is to use a single representation based on normal distribution maps [Fou92, KHS01]. These methods, however, do not consider all the geometric effects that can appear on or due to grooves, such as masking, shadowing, or inter-reflection. These effects must be precomputed using other approaches [Max88, HDKS00]. The work presented in [BPMG06] removed many of the previous limitations on geometry, size or distribution over the surface, in a resolution independent manner, using the same representation as in [BPMG04].

3 Interactive Rendering of Grooves using Graphics Hardware

The interactive rendering of grooved surfaces can be useful in many applications, like in a walk-through or as a pre-visualization tool before a complex rendering setup. Although the previous general method presented in [BPMG06] can render accurate grooved surfaces faster than using geometry, each image can take several seconds to render, which is far from being interactive. To overcome this, we have decided to adapt this method for its implementation on programmable graphics hardware, which can outperform CPU-based algorithms. Since the graphics processor (GPU) is not as general as the CPU, the algorithm needs to be modified if we want to run it on this platform. The main points that have required our attention are explained next.

3.1 Representing the grooves

Grooves are modelled using a representation based on paths and cross-sections [BPMG04]. In our case, grooves are modelled using the same representation in texture space. Such a representation is very compact and can be

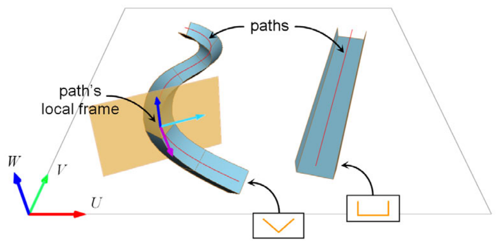


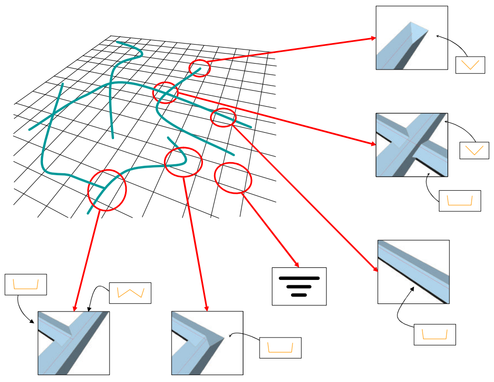
Figure 2: Grooves are represented in texture space by means of paths (in red) and cross-sections (in orange).

easily applied to any surface having a texture parametrization, without the need of reprojecting the paths between different surfaces. In addition, texture space paths can be easily defined and evaluated in 2D.

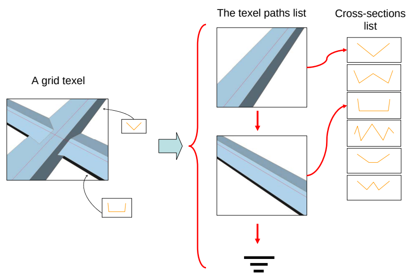
First, the geometry of each groove is described by a unique path and cross-section (see Figure 2): the path is specified as lying on the UV texture plane and the cross-section as being perpendicular to it, following the path's local or tangent frame. Paths are modeled by means of 2D polylines, that the user can specify directly in texture space or in 3D world space, by defining the paths onto the object surface and then transforming these onto texture space. Cross-sections are modeled using a 2D piecewise line (polyline), because they allow easy and quick computations of the light reflection. These cross-sections can penetrate the surface, protrude from it, or both.

For each groove, apart from its path and cross-section, the user can also choose specific reflection and transmission properties. This is necessary only when these are different from the ones of the base surface.

During the rendering stage, in order to quickly determine which grooves could be contained in a projected pixel, we use a precomputed uniform space subdivision of the UV texture plane. This forms a grid of equally sized cells, where each cell contains a list of its crossing paths (see [BPMG06] for details). Each of these elements represents a reference to the path geometry stored in main memory, that also contains a reference to the associated cross-section, reflection properties, and so on.



(a)



(b)

Figure 3: The data structures for representing grooves (a) The hash texture is used to spatially index each type of possible groove content (line, ends and intersections). (b) Each non-empty entry in the hash texture points to a list of paths information, which in turn refers to entries in the list of cross sections for all paths in the texture.

To transfer all this information to the GPU, we must store the data in textures, which will be loaded on the graphics memory and accessed by our program. We use two textures for this, one for the grid and another for the grooves' data. See Figure 3. However, the grid texture does not contain a list of all of its crossing grooves, but only a reference to the first one, whose properties are stored at the data texture. This only requires a single value for the grid texture: a null reference if no groove crosses the current cell, or a reference to the first crossing path otherwise. In the data texture, grooves are sequentially stored along with a flag indicating if the next block belongs to a groove on the same cell or not (Figure 3(b)). For each groove, we store its implicit line equation using three floating-point values, and the references to the associated cross-section and material properties packed on a fourth value, which also includes the mentioned flag. After 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 packet as a single value), the specular power, etc.

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. Also notice that the cross-sections and materials are stored separately to avoid duplicates among grooves, since most of them tend to share these properties.

3.2 Finding the grooves

The implementation presented in [BPMG06] projected a 2D bounding box of each pixel onto the grid, and checked the covered cells for gooves. To ensure all candidate grooves were taken into account, this bounding box was enlarged by taking into account parameters as the dimensions of the grooves and the view or light directions. An example is when the pixel does not project onto a groove but the contained surface is shadowed by a peak belonging to a close groove.

Unfortunately, this algorithm can not be ported to the GPU because of the inherent high computational cost in terms of texture lookups. The solution is to be able to gather all candidate grooves by sampling only at the pixel center, which can be done if we also store all grooves which might potentially overlap with the cell. In the case of grooves with protruding facets, we must detect if one of these facets shadows or is visible far from the bounds (width) of the groove. In such cases, we proceed as before and use a different version of the grid texture according to the view and light angle. Another solution could be the traversal of the grid texture in the given view or light direction, if only few cells must be visited.

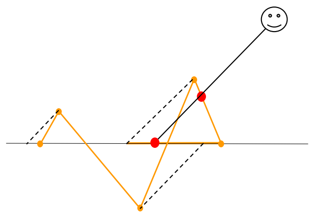


Figure 4: Projection of the scratch/groove profile to get the visible facet (See [BPMG06])

3.3 Evaluating the grooves

The evaluation of the scratches and grooves is completely done inside Fragment Shader units at the graphics hardware. The process starts by retrieving the corresponding entry from the hash texture at the current texture coords, and evaluating if that cell contains a groove, more than one or none at all. In the latter case, a traditional evaluation of the surface BRDF is performed. See Algorithm 1. For the other two cases, the scratches are retrieved from the data texture and evaluated. To simplify explanations, we will start with the simplest case of only one scratch in a cell (a very usual setting for surfaces with only a few scratches)

In the following code, mat stands for material, N for the surface normal, pos_path for an entry in the data texture, pos_prof for the cross-section identifier, and pos_mat for the material identifier.

```
Using surface attributes → mat, N
Read cell from hash tex(UV) → pos_path
if ( pos_path != -1 )
    Get scratch data for 1st scratch
        from data texture → path,
        pos_mat, pos_prof, etc.
    Read profile points from data
        texture (at pos_prof)
    if ( next_flag == 0 )
        ProcessIsolatedScratch→mat,N
    else
        ProcessIntOfScratches→mat,N
    endif
endif
Compute shading(mat,N) → color
```

Algorithm 1: Scratches computations.

Once the first scratch is retrieved and the flag for the next scratch is null, meaning no other scratches are in the cell, the computations for an isolated scratch begin. The pseudocode for those computations can be found in the Algorithm 2. The computations in this case are divided in two main parts: the visibility and the shadowing computations. In the first one, the profile is projected in two dimensions onto the horizontal line representing the base surface [BPMG06]. See Figure 4. Then the shadowing computations are performed. The lighting step of these computations start by projecting the profile again, but this time with respect to the light source direction. If there is a blocker, i.e. the illuminated facet is different from the visible one, the point is in shadows. If not, it is illuminated and the material at that point is retrieved, the normal is computed from the visible facet coordinates, and both stored for their later use.

```
// Visibility
Project profile in view direction
    and find the facet containing
    current point (visible facet)→f
Check if visible facet belongs to the
    scratch or to the surface
    ( f>0 && f<prof_points )→inside

// Shadowing
if ( inside )
    Project current point from the
        visible facet to the surface
        (in the light dir)
else
    Use the same point
endif

Project profile in light direction
    and find the facet containing
    current point → f2
Check if current point is
    shadowed (f != f2) → shadowed
// Set new material and normal
if ( shadowed )
    mat = 0 // black material
else if ( inside_scratch )
    Read material from data texture
        (at pos_mat) → mat
    Compute normal of visible facet
        and transform to 3D
        texture/tangent space → N
endif
```

Algorithm 2: Isolated scratch case.

3.4 Intersecting grooves

In the case where more than one scratch is present in the same cell, intersection computations must be performed to determine the actual intersected scratch. The first thing to do is to independently compute the intersections of the eye with the different profiles, as can be seen in the Algorithm 3. For every profile, this will generate an odd number of intersection points. As the profiles are known in advance, the maximum number of intersections can be pre-computed and used to define the maximum size of the vectors used in the fragment shader to evaluate the grooves. As before, the process is repeated to compute the right shadowing at the intersection point. So, the basic idea of our algorithm is to compute the intersections of a ray with each individual scratch profile, and combine them 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.

```
// Visibility
Project 1st profile and find all
    the intersection points
    → intSegs
While there are grooves to evaluate
    Get scratch data for next scratch
```

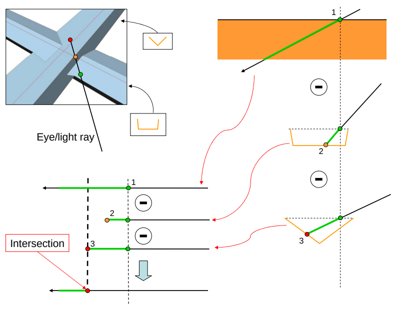


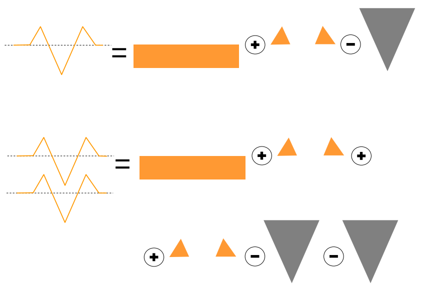
Figure 5: Considering the crossing of two scratches as the subtraction of the two volumes from the basic flat surface (left, above). When tracing each ray, we compute its spans through the groove body (right), and we subtract these spans from the ray path, resulting in the final intersection point (left, below).

```
        Read profile points from data tex
        Project next profile → int
        Combine(intSegs, int) → intSegs
return firstPoint(intSegs)
```

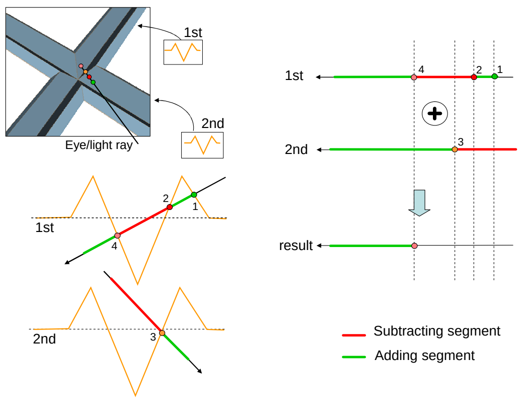
Algorithm 3: Intersection of scratches case.

If we look at Figure 5, we can see a basic example with two simple grooves intersecting. If the grooves have no protruding parts, we can think that the grooves were built by removing material from the base surface. This is like building the grooves with Constructive solid Geometry (CSG). We can consider that we have a CSG tree: from a flat, solid surface, we subtract the volume of each of the grooves in turn, resulting in holes that can be ray-traced as described in [FvDFH90].

But subtracting grooves is not sufficient if we can have scratches with protruding parts, as those paths can not be modelled 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 scratch that are below the base flat surface, as can be seen in Figure 6(a). It is important to mention that we must subtract not only the part of the scratch that is strictly under the base surface, but we must subtract a



(a)



(b)

Figure 6: (a) When computing the intersection of grooves, CSG operations must be ordered and all additions must be performed before any subtraction. (b) When tracing rays, a ray is partitioned in segments coming from an addition (red segments) and segments coming from a subtraction (green segments). Then, tracing a ray becomes a regular CSG ray-tracing operation.

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, as rays always progress from top to bottom in the texture local coordinate system. In order to correctly handle this, the different faces of a scratch profile must be classified according to their origin: "external" if they come from the addition of material, and "internal" if they are part of a subtracting material. With this classification, a ray that intersects a scratch is partitioned into segments that come from the base surface or addition of material, or segments that come from the subtraction of material. See Figure 6(b). In our current implementation, this 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 scratch face changes its orientation with respect to the direction of classification.

3.5 Special geometries

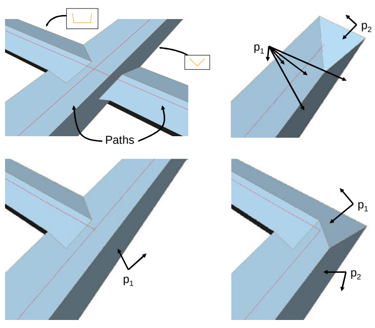


Figure 7: Special situations. Clockwise from top left: Intersection, intersected end, isolated end, and corner.

Although intersections are handled in a natural way with the previously explained method, in the case of T-junctions, ends or corners, a slightly different approach has to be used. In the mentioned cases, a "priority" flag is manually associated for some relevant segments of the profiles of the scratch that forces the ending of the other one. As an example, in a "T" junction, the opposite segments of the scratch that does not finish are labelled as having "priority" with respect to the scratch that ends. This way, when evaluating the intersections, the intersections from a prioritized segment take precedence over the non-prioritized ones, allowing to simulate the situation where one of the scratches ends abruptly when colliding with the other. Endings are modelled like a "T" junction, but without visualizing the perpendicular scratch. This is achieved by giving priority to all the facets of the ending groove, too. In a corner, carefully assigning the priorities to both scratches guarantees the correct visualization.

Figure	FPS	Mem	Text res
8 top left	895.5	72	60x60 / 61x61
8 top middle	621.7	158	150x150 / 67x67
8 bottom right	318.7	131	150x150 / 53x53
9 bottom left	484.5	8	25x25 / 16x16
9 bottom right	109.4	8	25x25 / 16x16
10 bottom left	352.5	52	60x60 / 49x49
10 bottom right	425.6	8	25x25 / 16x16
11 top left	127	100	100x100 / 62x62
11 top middle	73.7	100	100x100 / 62x62
11 bottom right	169.1	100	100x100 / 62x62

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

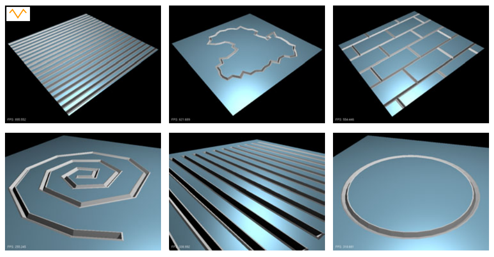


Figure 8: Flat plane with different groove patterns.

4 Results and Conclusions

Our method has been implemented as a fragment shader using Cg and the OpenGL API. The rendering times for the next 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.

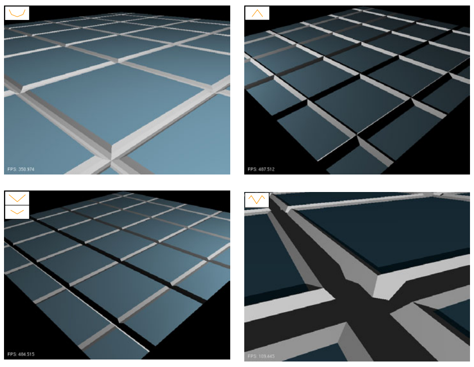


Figure 9: Flat plane with different cross-sections for a set of intersecting grooves.

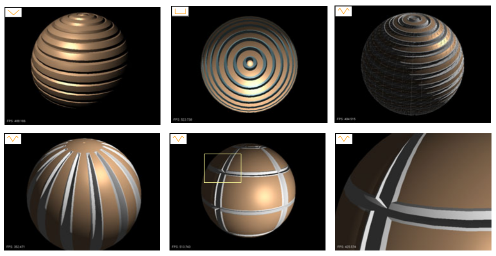


Figure 10: Curved surface rendered using our GPU program under different views and lighting conditions, with grooves using different materials, cross-sections (top) and patterns (bottom). The underlying mesh is included in the top right.

In Figure 8 we can see some results of our method for a flat plane consisting of different groove patterns with the same cross-section.

Figure 9 shows a similar flat plane with a groove pattern consisting of a set of intersecting grooves. In this case, different cross-sections have been used for each image, which are included in their top left.

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 150x150 (see Table 1), and data textures are even lower. The resolution of the grid texture only determines our efficiency when finding if the current point contains a certain groove or not: the less grooves 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 grooves. Image-based techniques such as relief mapping would require high resolution textures in order to obtain a similar quality. Even with highly detailed textures, such kind of sharp detail can not be correctly simulated without using a geometry-based approach.

In Figure 10, we can see some other similar patterns applied to a non-flat surface, here represented by a sphere. In the top row, we have used a set of parallel grooves and then applied different cross-sections and material properties on them. The middle image has been rendered from a viewpoint pointing towards the top of the sphere, and the right image shows the underlying mesh. With our method, the visibility and shadowing of the different grooves can be properly simulated without modifying the surface geometry, as can be observed. In the bottom row, we have applied two different groove patterns that include ends and intersections. In this case, the image on the right shows the same sphere used in the middle image but rendered from a closer viewpoint. This close-up corresponds to the selected region in the middle image.

Finally, Figure 11 shows an example of a more complex scene composed of several grooved surfaces, using the same groove pattern as in the top right image of Figure 8, properly tiled. 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 bottom right image, which represents a close-up of the region shown in the bottom middle image too. 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

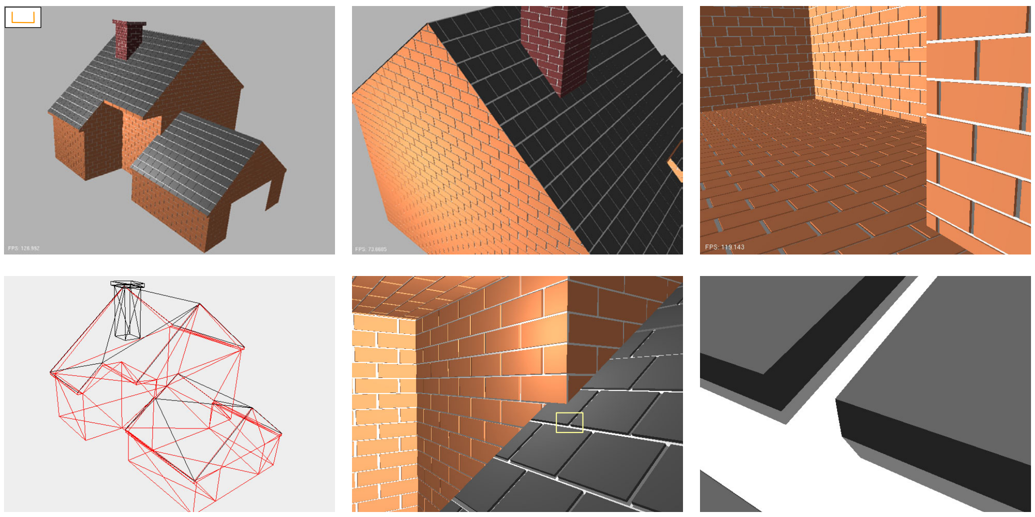


Figure 11: House rendered in real-time from different viewpoints using our approach to simulate the bricks. The underlying mesh is shown in the bottom left.

timings are fast enough and the interactivity is not lost.

5 Future work

In our previous approach, the use of textures for the storage of the geometry and properties of the grooves introduces some important limitations. First limitation is that textures need to be precomputed, which greatly difficults the editing of the grooves or their properties. Another limitation is that the textures must later be accessed at run time in order to recover the data, and this introduces a considerable decrease in the performance of our shader.

One promising line of research is to draw scratches as textured quads, each containing a scratch or part of a scratch. Then, a pixel shader would compute the scratch appearance as described above. This would allow editing scratches in real time, allowing to view the final scratches as we move them on the surface we want the scratch on. Unfortunately, intersecting scratches are more difficult to model, as we expect them to be passed in any order to the graphics hardware. This would require multiple passes to resolve the overlapping scratches, each pass adding a scratch on top of the previous one, if needed. In this scenario, scratch data is passed as properties of the quads (using TEXCOORD registers), with intensive usage of render-to-texture and multiple render targets. This method, compared to the one described in this paper, has some advantages like being able to edit in real time the shape and position of the scratches' paths, and it is not difficult to foresee it to be faster than the previous one. Unfortunately, it presents some disadvantages, too, like not being easily applicable to curved objects.

6 Acknowledgements

This work was founded with grant TIN2004-07672-C03-01 from the Ministerio de Educación y Ciencia, Spain.

References

[Ban94] David C. Banks. Illumination in diverse codimensions. In *Proceedings of SIGGRAPH 94*, Computer Graphics Proceedings, Annual Conference Series, pages 327–334, July 1994.

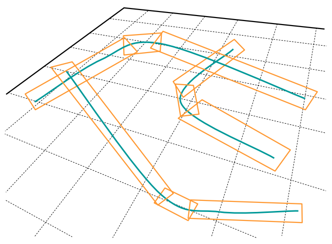


Figure 12: A representation of the scratches as textured quads.

[BB90] Welton Becket and Norman I. Badler. Imperfection for realistic image synthesis. *Journal of Visualization and Computer Animation*, 1(1):26–32, August 1990.

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

[BM93] Barry G. Becker and Nelson L. Max. Smooth transitions between bump rendering algorithms. In *Proceedings of SIGGRAPH 93*, Computer Graphics Proceedings, Annual Conference Series, pages 183–190, August 1993.

[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.

[BPMG06] Carles Bosch, Xavier Pueyo, Stéphane Mérillou, and Djamchid Ghazanfarpour. General rendering of grooved surfaces. Research Report IIiA06-10-RR, Institut d'Informàtica i Aplicacions, Universitat de Girona, 2006.

[CGF04] Chiara Eva Catalano, F. Giannini, and B. Falcidieno. Introducing sweep features in modeling with subdivision surfaces. *Journal of WSCG*, 12(1):81–88, 2004.

[Coo84] Robert L. Cook. Shade trees. In *Computer Graphics (Proceedings of SIGGRAPH 84)*, volume 18, pages 223–231, July 1984.

[Fou92] Alain Fournier. Normal distribution functions and multiple surfaces. In *Graphics Interface '92 Workshop on Local Illumination*, pages 45–52, May 1992.

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

[HDKS00] Wolfgang Heidrich, Katja Daubert, Jan Kautz, and Hans-Peter Seidel. Illuminating micro geometry based on precomputed visibility. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 455–464, July 2000.

[Kaj85] James T. Kajiya. Anisotropic reflection models. In *Computer Graphics (Proceedings of SIGGRAPH 85)*, volume 19, pages 15–21, July 1985.

[KHS01] Jan Kautz, Wolfgang Heidrich, and Hans-Peter Seidel. Real-time bump map synthesis. In *2001 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 109–114, August 2001.

[Max88] Nelson L. Max. Horizon mapping: shadows for bump-mapped surfaces. *The Visual Computer*, 4(2):109–117, July 1988.

[MDG01a] Stéphane Mérillou, Jean-Michel Dischler, and Djamchid Ghazanfarpour. Surface scratches: Measuring, modeling and rendering. *The Visual Computer*, 17(1):30–45, 2001.

[MDG01b] Stéphane Mérillou, Jean-Michel Dischler, and Djamchid Ghazanfarpour. Surface scratches: Measuring, modeling and rendering. *The Visual Computer*, 17(1):30–45, 2001.

[PF90] Pierre Poulin and Alain Fournier. A model for anisotropic reflection. In *Computer Graphics (Proceedings of SIGGRAPH 90)*, volume 24, pages 273–282, August 1990.

[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.

[Sta99] Jos Stam. Diffraction shaders. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pages 101–110, August 1999.

[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, New York, NY, USA, 2006. ACM Press.

[War92] Gregory J. Ward. Measuring and modeling anisotropic reflection. In *Computer Graphics (Proceedings of SIGGRAPH 92)*, volume 26, pages 265–272, July 1992.

[WAT92] Stephen H. Westin, James R. Arvo, and Kenneth E. Torrance. Predicting reflectance functions from complex surfaces. In *Computer Graphics (Proceedings of SIGGRAPH 92)*, volume 26, pages 255–264, July 1992.

[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.