Graphics Hardware (2005)
M. Meissner, B.- O. Schneider (Editors)

Generic Mesh Refinement on GPU

Tamy Boubekeur & Christophe Schlick
LaBRI INRIA CNRS University of Bordeaux, France

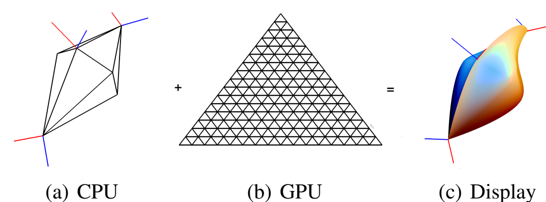
Abstract

Many recent publications have shown that a large variety of computation involved in computer graphics can be moved from the CPU to the GPU, by a clever use of vertex or fragment shaders. Nonetheless there is still one kind of algorithms that is hard to translate from CPU to GPU: mesh refinement techniques*. The main reason for this, is that vertex shaders available on current graphics hardware do not allow the generation of additional vertices on a mesh stored in graphics hardware. In this paper, we propose a general solution to generate mesh refinement on GPU. The main idea is to define a generic refinement pattern that will be used to virtually create additional inner vertices for a given polygon. These vertices are then translated according to some procedural displacement map defining the underlying geometry (similarly, the normal vectors may be transformed according to some procedural normal map). For illustration purpose, we use a tesselated triangular pattern, but many other refinement patterns may be employed. To show its flexibility, the technique has been applied on a large variety of refinement techniques: procedural displacement mapping, as well as more complex techniques such as curved PN-triangles or ST-meshes.*

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Line and curve generation I.3.3 [Computer Graphics]: Display algorithms

1. Introduction

One ubiquitous technique to generate complex geometric models in computer graphics, is to start from a coarse model and apply some *refinement techniques* to get the final enriched model. The many refinement techniques that have been proposed in the literature over the years, can be divided in two main families: *displacement mapping* that are usually employed to add some geometric details to a coarse model, and *subdivision surfaces* that are used to generate smooth surfaces from a small number of polygons. Trying to get a full GPU implementation of displacement mapping or subdivision surfaces is challenging because of one major limitation of current graphics hardware: the lack of geometry generation. Indeed, the rendering pipeline of current GPU has been designed to efficiently rasterize a huge list of indexed polygons, but it is not able to generate more polygons than those sent through the graphics bus, by the application running on the CPU. One major consequence is that in current high-end applications, the whole mesh refinement process is done on the CPU, and the huge set of refined triangles is then sent to the GPU. To avoid a bandwidth bottleneck on the graphics bus, a trade-off has to be found, between uploading detailed geometry or uploading detailed textures. This issue explains why subdivision surfaces or displacement mapping are mainly used in applications where the rendering is done



(a) CPU (b) GPU (c) Display

Figure 1: *Starting from a coarse mesh (a) provided by the CPU, the GPU uses a generic refinement pattern (b) to create an enriched mesh (c) in a standalone process, thus keeping the CPU resources totally free to perform other tasks, such as deforming or animating the coarse model.*

off-line (e.g. movie industry) and are seldom implemented in realtime 3D engines. Note that some specific tessellation units (that were able to split some high-level geometric patch into a set of tris or quads) have sometimes been included in specialized hardware (e.g. B-Spline and Bezier tessellation on SGI hardware, Curved PN-Triangle tessellation [VPBM01] on the ATI Radeon 8500), but no current standard hardware does include a generic tessellation unit, that would enable a GPU implementation of multi-purpose refinement techniques.

In this paper, we propose a general principle to generate mesh refinement on existing GPUs that only requires an

© The Eurographics Association 2005.

easy-to-implement vertex shader. The main idea is to define a generic refinement pattern that will be used to virtually create additional inner vertices for a given polygon. As a consequence, the footprint of the refined mesh is dramatically reduced, both on the CPU and the GPU memory, and the amount of data that is transmitted through the graphics bus becomes totally independent of the refinement depth used for the rendering (see Figure 1).

2. Previous Works

In the field of fast mesh refinement for real-time applications, a large among of work has been done to optimize evaluation of various subdivision surface schemes (see [ZS00] for a survey). But, despite the popularity of subdivision surfaces in high-end software environments, relatively few work has been published about hardware assisted implementation of subdivision schemes. Bishoff et al. [BKS00] have proposed to use the *forward differenciation* to compute the limit position of vertices for the Loop scheme [Loo87]. Recently, Bolz et Schroder [BS02] [BS03] have proposed an hardware evaluation of the Loop scheme, including the additional "shape factor" rules introduced by Biermann et al. [BLZ00]. The main problem with subdivision schemes such as Loop or Catmull-Clark, is that the refinement process by subdivision schemes is heavily based on dynamic adjacency information. Such information can be easily provided in software implementation by using clever multi-linked dynamic data structures, but is extremely difficult to translate into hardware-friendly terms. As an alternative to high quality subdivision methods, Vlachos et al. have proposed a fast mesh refinement method called *Curved PN Triangles* [VPBM01]. This method is based on the construction of a triangular Bezier patches [Far01] for each triangle, driven by its 3 vertices and associated normals. As shown by Chung and Kim [CK03], this refinement scheme is well adapted to hardware implementation, but it requires some specific tessellation unit (note that such a specific tessellation unit has actually been included in the ATI Radeon 8500 hardware in 2001). In contrast with our pure *vertex shading* approach, and for the case of subdivision surfaces, *fragment shading* techniques has been intensively used to render refined meshes, using for instance *geometry images* [LHSW03] or spiral-enumerated fragment meshes [SJP05].

3. Mesh refinement on GPUs

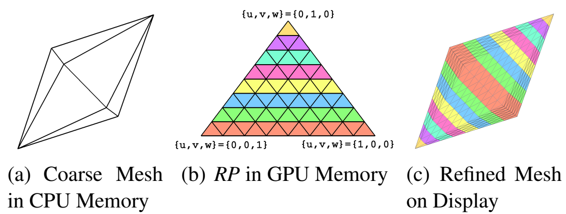
3.1. Overview

While GPU-friendly primitives, such as display lists or vertex buffer objects, dramatically reduce the CPU memory footprint and the traffic jam on the graphics bus (the model is loaded once for all on the GPU memory and is then canceled from the CPU memory), it also involves some strong limitations:

- The whole mesh has to fit into the GPU memory (this may be difficult for very detailed meshes).
- When real-time constraints require some LOD rendering, the mesh has to be stored n times, for n different levels of detail.
- When the mesh includes a dynamic behavior that cannot be expressed by vertex shaders, it has to be updated from the CPU at each frame.

When considering these limitations, it appears quite obvious that a desirable solution would be to use only a coarse mesh at the CPU level (such a coarse mesh can be efficiently updated and transmitted on a frame by frame basis, in the case of complex dynamic behaviors) and to add some on-the-fly mesh refinement process at the GPU level (which only generates the appropriate level-of-detail, according to the viewing parameters). This section presents the solution that we propose for this mesh refinement process.

3.2. Refinement Pattern



(a) Coarse Mesh in CPU Memory (b) *RP* in GPU Memory (c) Refined Mesh on Display

Figure 2: *Our refinement method only requires one single refinement pattern RP stored at full resolution as a vertex buffer on the GPU memory. In our example, RP is a tesselated triangle that is splitted into 8 different triangle strips (each strip has a different color)*

The main idea of our approach is to use *one single Refinement Pattern (RP)* for each polygonal shape that exists in the mesh. In the remainder of the paper, we only consider the usual case of triangular meshes (and thus the case of a triangular refinement pattern), but the approach is still valid for other polygonal shapes. This refinement pattern is transmitted once for all, from the CPU to the GPU, as a vertex buffer containing a few strips (see Figure 2). Let A_T be the set of attributes of a coarse triangle T. Typically A_T contains the 3 vertex position, 3 vertex normals, 3 vertex colors, and 3 texture coordinates. We propose to render the refined mesh with the following algorithm:

render (Mesh M)
    for each coarse triangle T of M **do**
        place A_T as uniform variables for the vertex shader;
        draw *RP*;

Actually, embedded within the draw function, there is a vertex shader which transforms the vertices of *RP* toward the position, rotation and scale of T (see at the end of this subsection for details). One major advantage of this algorithm, is that the amount of data transmitted on the graphic bus is

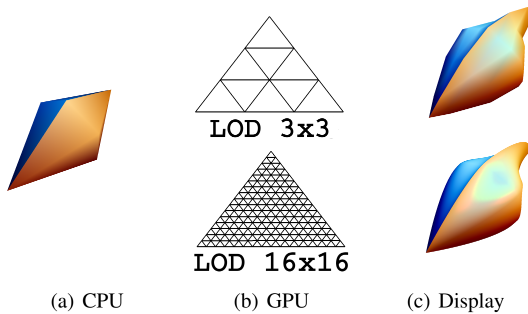
© The Eurographics Association 2005.

independent of the resolution of the refined mesh. The task of the CPU is thus reduced to the uploading of the coarse triangle attributes and eventually some additional properties that characterize the behavior of the refined mesh (for instance, a random value for a fractal noise). The uniform variables placed onto the GPU stay constant during all the drawing process of *RP*. Let now suppose that a functional $f_{A_T} : [0,1]^2 \rightarrow R^3$ can be constructed over A_T. To evaluate f_{A_T} at each vertex V of *RP*, we have to recover its parametrization $\{u,v\}$ onto T. Since *RP* is only used for the topological storage purpose, we propose to encode the parametrization at V as its position vector: $V_{xyz} := \{w,u,v\}$ where $w = 1-u-v$. Now, during the vertex shading pass, the GPU can clearly identify the parametrization $\{u,v\}$ for each vertex V of *RP*, and thus evaluate its functional value $f_{A_T}(u,v)$. Of course, each attribute in the set A_T may eventually be interpolated by a different functional. The most simple application of our principle is a tessellation of the mesh, combined with some height field texture. Let us consider the position attributes $\{P_0,P_1,P_2\}$ of the current coarse triangle drawn and the parametrization $\{1-u-v,u,v\}$ (encoded as the position of inner vertices) of each vertex V of *RP*. To perform a simple tessellation, we just have to interpolate between $\{P_0,P_1,P_2\}$ to obtain the output position V_{xyz} of V:

$$V_{xyz} := wP_0 + uP_1 + vP_2 = V_x P_0 + V_y P_1 + V_z P_2$$

Figure 2 shows the principle of this scheme. Now, since we have a regular parametrization of *RP*, it is straightforward to use it to interpolate between the texture coordinates of T. For instance, if a height map is stored on the GPU as a texture, by querying a height value at each inner vertex V, a displacement mapping generated at the resolution of the *RP* is straightforwardly obtained.

3.3. Levels-Of-Detail on GPU



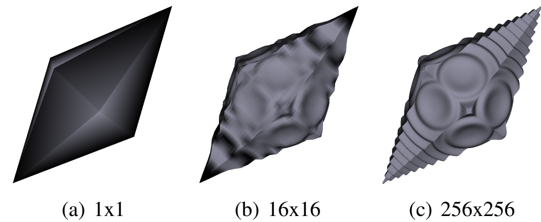
(a) CPU (b) GPU (c) Display

Figure 3: *Different LODs of a coarse model can be rendered by simply selecting the appropriate RP vertex buffer on the GPU. Here, the vertex shader smoothes the mesh while preserving the sharp creases specified at coarse level.*

A basic LOD scheme (depending on the distance of the object to the current point of view) can be generated by storing on the GPU a set of patterns $RP_0, RP_1 \ldots, RP_n$ computed at different resolutions and by selecting the desired resolution at rendering-time (see Figure 3).

4. Examples

We have shown that the displacement mapping by height field texture was straightforward to integrate with our method. But a large variety of other local mesh enhancements can be integrated in the vertex shader, and consistently applied on the refined coarse meshes. This section illustrates some of the results that we have obtained with our generic on-the-fly refinement process.



(a) 1x1 (b) 16x16 (c) 256x256

Figure 4: *Procedural displacement with high frequencies. The smoothness of the function appears at a very high tesselation rate. Our approach limits the GPU memory footprint to only* one *tesselated triangle, with an equivalent framerate to the complete tesselated object stored as a vertex buffer.*

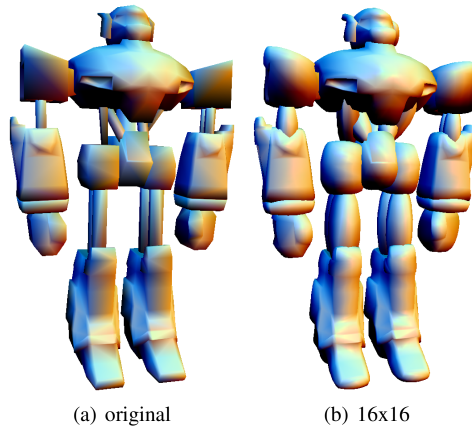
Procedural Displacement Mapping: The Shader Model 3 [NVi] has introduced the access to the texture memory from the vertex shading stage. One direct consequence of this extension is to allow GPU implementation of displacement mapping. Unfortunately, several strong limitations occur with such an implementation: First, texture look-up with random access is very expensive when writing vertex or fragment shaders, which means that the performance is dramatically reduced in that case. The only solution to get reasonable performance is to achieve coherent cache-friendly texture access, which is far from being always possible. Second, aliasing artefacts are quite common as textures are not filtered when accessed by the vertex shader. Third, this approach is not suitable for dynamic effects, like waves on the ocean, for instance.

Following the work of Perlin, it is widely admitted that many kinds of visual enrichments (especially dynamic visual enrichments) can be efficiently replaced by a continuous procedural function, rather than storing and interpolating a discrete texture. This is of special interest for displacement mappings which often have dynamic behaviors. In our framework, implementing procedural displacement mapping simply means that the uploaded attributes at each coarse triangle should embed all the mandatory data for the construction of the procedural displacement map on that triangle, which usually means a few number of floating point values.

The nice property of this approach is that the refinement pattern can be adapted to the level of refinement needed

© The Eurographics Association 2005.

to get correct antialiased effects. Figure 4 shows that, even with pathological examples that exhibit very high frequencies, the displacement process can be adapted to get a nice aliasing free rendering. The displacement process used in Figure 4 simply consists in an elevation of each inner vertex P along its normal vector N according to a sinusoidal function: $a\sin(f||P||)$, so the only additional data to be transmitted to the GPU are the amplitude a and the frequency f. The same principle is obviously applicable to more complex functions such as the ubiquitous Perlin Noise.



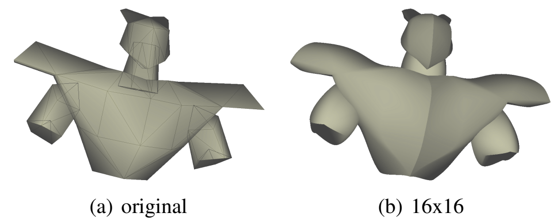
(a) original (b) 16x16

Figure 5: *Our refinement method adapted to PN-Triangles.*

Curved PN Triangles: The work of Vlachos et al. on *Curved PN-Triangles* [VPBM01] has demonstrated that a purely local interpolating refinement scheme can be used to obtain a fast mesh smoothing, providing a *visual smoothness*, despite the lack of high order geometric continuity of the underlying patches. The principle of the Curved PN-Triangle is quite simple: for each triangle of a mesh, a cubic Bezier surface is constructed with the 3 points P_k and their 3 associated normal vectors N_k. With such a loose definition, it is not possible to generate a continuous surface across edge boundaries of the initial mesh. So to get aesthetic results that offer some kind of *visual* continuity, Vlachos et al. have proposed to independently generate a smooth normal field over the mesh, by using a linear or a quadratic interpolation of normal vectors (the general process of PN-Triangles is briefly recalled in Appendix A).

Practically, Curved PN-Triangles offer a visual quality similar to the Modifed Butterfly subdivision scheme [DLG90], but are much more GPU friendly: they are local, require only a constant amount of data, and can be directly applied on an indexed face set representation, without any additional topological structure.

Unfortunately, even if Curved PN-Triangles have been integrated in the previous generation of ATI hardware (8500 family [ATI]), they have never been specified as a standard for all graphics cards. So, another contribution of the present paper is to provide a generic solution to the hardware implementation of the Curved PN-Triangles on all programmable GPUs. With our method, the CPU has just to precompute the 10 PN-Triangle coefficients (see Appendix A) associated to each coarse triangle, and transmit them with the set of color attributes, before the rendering call of the refinement pattern *RP*. The evaluation of the Bezier patch $b(u,v)$ is then performed on the GPU by the vertex shader according to this bunch of uniform coefficients. Note that these coefficients are the only information transmitted through the graphics bus, whatever the level of refinement generated by the *RP*; these coefficients may also be updated at a very low cost, in the case of dynamic models. Figure 5 presents an example of our hardware PN-Triangles refinement on a coarse mesh.



(a) original (b) 16x16

Figure 6: *Out refinement method applied to ST-Meshes. The per-vertex* tension *and* sharpness *shape parameters express sharp creases that are propagated to the refined mesh.*

Scalar Tagged mesh: With the machinery proposed by Vlachos et al., it becomes possible to propose a complete surface control system during the refinement. In particular, a set of additional per-vertex scalar attributes can help to manage other local differential geometry behaviors, such as sharp edges or cone-like vertices. We have recently introduced a generalization of PN-Triangles, called *Scalar Tagged Meshes* (ST-Meshes, for short, also called *Scalar Tagged PN Triangles* in [BRS05]), that allows the user to manipulated some additional scalar shape parameters, classified in two families:

- **vertex parameters**: for instance, a *tension* parameter controls the local curvature of the refined surface, at each vertex;
- **edge parameters**: for instance, a *sharpness* parameter controls the amount of tangent plane discontinuity at each edge, and a *bias* parameter controls the orientation of this discontinuity.

These shape parameters are then simply combined to modify the Bezier coefficients used in the PN-Triangle model to generate the refined geometry and the normal vector field. The reader may find the formulations for these modified coefficients and normal in Appendix B. Here again, the coefficient are transmitted as uniform variables to the vertex shader, and

© The Eurographics Association 2005.

the remainder of the process is totally similar as above. Figure 6 presents an example of our hardware refinement process applied on an ST-Mesh, where the scalar tags have been set to define two sharp creases.

5. Performance

We have analyzed the whole performance of our system, by comparing the rendering framerate and the memory footprint (number of stored triangles) of a given mesh stored at a given resolution, either by using a static vertex buffer object (VBO) storing the whole mesh as triangle strips, or by using our generic refinement technique where just one tessellated refinement pattern is actually stored. Our configuration is an Intel P4 at 3.0 GHz, 1GB RAM, with a NVIDIA Quadro FX 4400 (PCIe). All the tests have been performed under MS Windows using OpenGL Shading Language [KBR04]. The results are summarized in Figure 7.

Coarse Triangles	10	100	1000
Static VBO			
64x64 - FPS	973.10	107.46	11.94
64x64 - Vertices	42250	422500	4225000
128x128 - FPS	271.85	29.87	3.97
128x128 - Vertices	166410	1664100	16641000
Our approach			
64x64 - FPS	965.12	107.11	11.72
64x64 - Vertices	4225	4225	4225
128x128 - FPS	269.35	29.05	3.81
128x128 - Vertices	16641	16641	16641

Figure 7: *Rendering framerate and GPU footprint (in number of vertices) for the rendering of some tesselated models (without culling).*

The main result of our on-the-fly refinement technique is clearly visible: the framerate obtained is equivalent to the one obtained by storing the full refinement mesh, but the footprint is dramatically reduces, as expected. Globally, our approach divides the GPU footprint by about the number of input coarse triangles. Note that we have just given the number of vertices, but the GPU memory also handles the topology. After a strippification step, the static vertex buffer topology represents about 50% of additional data to manage in GPU memory, while the topology cost is totally negligible with our approach. In terms of framerate, the slight performance drop of our system is essentially due to the barycentric interpolation between the 3 original vertices, when evaluating the position and normal at each inner vertex of the *RP*. Actually, a simple tessellation is never used in practice and is usually combined with more advanced surface refinement procedures, like the ones presented in Section 4. In that case, the computation of the procedural function usually represents the most expensive part of the vertex shader, and the framerates obtained with static vertex-buffer objects and with our approach are totally equivalent. For instance, the ST-Mesh presented on Figure 6 is made of 162 triangles. We obtain the same framerate of 61 fps when it is refined to 331 938 triangles either statically (with a full vertex buffer object) or dynamically (with a *RP* at 64x64, which corresponds to a recursive subdivision at level 6, a quite usual refinement rate for CG models).

Limitations: The simple LOD scheme described in Section 3.3 is not adaptative, and produces popping effects [Hop96]. One of our current works is to try to resolve this constraint locally, at the *RP* level.

6. Conclusion

We have proposed an easy-to-implement method to obtain a virtual mesh generation on the GPU. The contributions of our approach are:

- A very simple method to obtain a low cost tessellation of meshes.
- A generic and economic hardware implementation of the Curved PN-Triangle on standard programmable GPU.
- A virtual '*on-the-fly GPU tesselator unit*, for the cost of a linear interpolation between 3 vertices by refined vertex.
- The possibility to render refined and enriched meshes without neither storing nor transmitting through the graphics bus, the high resolution *ready-to-render* mesh, that would virtually not fit in GPU memory for a very deep refinement.

Our method always involves a constant amount of data transmission to the GPU, independently of the target resolution. This approach is clearly interesting in the case of dynamic objects, for completely local refinement scheme and for procedural surfaces generated on simple shapes. Our approach is particulary fruitfully for very deep refinement and when the bottleneck of an application using intensively tesselation is identified on the CPU-GPU bus (AGP or PCI express for instance).

One of our main future work will be to propose a better LOD scheme, adaptive and geomorphic. We investigate also the rendering of true subdivision surfaces and high quality parametric surfaces, by reducing to a fixe size the per-triangle information required.

References

[ATI] ATI: Ati technologies. http://www.ati.com. 4

[BKS00] Bishoff S., Kobbelt L., Seidel H.-P.: Towards hardware implementation of loop subdivision. *Proc. ACM Workshop on Graphics Hardware* (2000). 2

[BLZ00] Biermann H., Levin A., Zorin D.: Piecewise smooth subdivision surfaces with normal control. *Proc. ACM SIGGRAPH* (2000). 2

© The Eurographics Association 2005.

[BRS05] BOUBEKEUR T., REUTER P., SCHLICK C.: Scalar tagged pn triangles. In *Eurographics 2005 (short papers) to appear* (august 2005). 4

[BS02] BOLZ J., SCHRODER P.: Rapid evaluation of catmull-clark subdivision surfaces. *Proc. 3D Web Technology* (2002). 2

[BS03] BOLZ J., SCHRODER P.: Evaluation of subdivision surfaces on programmable graphics hardware. *Preprint available at http://www.multires.caltech.edu/pubs/GPUSubD.pdf* (2003). 2

[CK03] CHUNG K., KIM L.-S.: A pn triangle generation unit for fast and simple tesselation hardware. In *IEEE International Symposium on Circuits and Systems* (2003), pp. 728–731. 2

[DLG90] DYN N., LEVINE D., GREGORY J.: A butterfly subdivision scheme for surface interpolation with tension control. *ACM Trans. Graphics 9*, 2 (1990). 4

[Far01] FARIN G.: *Curves and Surfaces for CAGD (Fifth Edition)*. Morgan Kaufman Inc., 2001. 2

[Hop96] HOPPE H.: Progressive meshes. *Computer Graphics 30*, Annual Conference Series (1996), 99–108. 5

[KBR04] KESSENICH J., BALDWIN D., ROST R.: The opengl shading language. http://www.opengl.org, 2004. 5

[LHSW03] LOSASSO F., HOPPE H., SCHAEFER S., WARREN J.: Smooth geometry images. In *Proceedings of SGP 2003* (2003), Eurographics Association, pp. 138–145. 2

[Loo87] LOOP C.: Smooth subdivisions surfaces based on triangles. *Master Thesis, University of Utah* (1987). 2

[NVi] NVIDIA: Shader model 3. http://www.nvidia.com, 2004. 3

[SJP05] SHIUE L.-J., JONES I., PETERS J.: A realtime gpu subdivision kernel. In *ACM Siggraph* (2005). 2

[VPBM01] VLACHOS A., PETERS J., BOYD C., MITCHELL J.: Curved PN triangles. *Proc. ACM I3D* (2001). 1, 2, 4

[ZS00] ZORIN D., SCHRODER P.: Subdivision for modeling and animation. *ACM SIGGRAPH Courses Notes* (2000). 2

Appendix A: Curved PN Triangles

The principle of PN-Triangles is to replace each triangle of the initial mesh by an enriched description that combines a displacement field and a normal field. The displacement field is used to improve the geometry, and is defined as a cubic triangular Bezier patch. The normal field is used to improve the shading, and is defined as a quadratic triangular Bezier patch.

Displacement Field: For a triangle defined by 3 vertices $V_i = (P_i, N_i)$, with P_i being the position and N_i being the normal, the displacement field $b(u,v)$ of a PN-Triangle is defined as follows:

$$b(u,v) = \sum_{i+j+k=3} b_{ijk} u^i v^j w^k \tag{1}$$

where $w = 1-u-v$ and $u,v,w \geq 0$. Note that (u,v,w) are the three barycentric coordinates relative to the vertices.

By defining $s_{ij} = (P_j - P_i).N_i$ where . represents the scalar product, the control points b_{ijk} of the Bezier patch are given by:

- $b_{300} = P_1$, $b_{030} = P_2$, $b_{003} = P_3$,
- $b_{210} = (2P_1 + P_2 - s_{12}N_1)/3$, $b_{120} = (2P_2 + P_1 - s_{21}N_2)/3$,
- $b_{021} = (2P_2 + P_3 - s_{23}N_2)/3$, $b_{012} = (2P_3 + P_2 - s_{32}N_3)/3$,
- $b_{102} = (2P_3 + P_1 - s_{31}N_3)/3$, $b_{210} = (2P_1 + P_3 - s_{13}N_1)/3$,
- $b_{111} = \frac{1}{4}(b_{210} + b_{120} + b_{021} + b_{012} + b_{102} + b_{201}) - \frac{1}{6}(b_{300} + b_{030} + b_{003})$.

So, for each point on the initial triangle, defined by its barycentric coordinates (u,v,w), one simply has to compute the displacement vector $b(u,v)$ to get its position on the final refined surface.

Normal Field: Similarly, the normal field $n(u,v)$ of a PN-Triangle is defined as follows:

$$n(u,v) = \sum_{i+j+k=2} n_{ijk} u^i v^j w^k \tag{2}$$

where (u,v,w) are the barycentric coordinates as previously. By defining $t_{ij} = 4\frac{(P_j-P_i).(N_i+N_j)}{(P_j-P_i).(P_j-P_i)}$, the control points n_{ijk} of the Bezier patch are given by: $n_{200} = N_1$, $n_{020} = N_2$, $n_{002} = N_3$, $n_{110} = N_1 + N_2 - t_{12}(P_2 - P_1)$, $n_{011} = N_2 + N_3 - t_{23}(P_3 - P_2)$, $n_{101} = N_3 + N_1 - t_{31}(P_1 - P_3)$. Note that in the original paper on PN-triangles, there was a confusion between the control points n_{110}, n_{011} and n_{101}, and the corresponding normal vector computed by $n(1/2,1/2,0)$, $n(0,1/2,1/2)$ and $n(1/2,0,1/2)$. The formulation given here is the correct one. Remember to normalize the vector to unit length to get the final normal vector: $N_{uv} = n(u,v)/\|n(u,v)\|$

Appendix B: Scalar Tagged Mesh

Scalar Tagged Mesh have been designed to model local surfaces features such as sharp creases with a set of per-vertex shape parameters placed on a coarse mesh. The mesh can then be refined with a modification of the original Curved PN-Triangle model that handles these additional shape parameters. Each vertex V_i of a ST-Mesh is defined by a position P_i, a normal N_i, a deviation vector Δ_i that indicates the direction of the crease passing through V_i (null for smooth areas) and 3 scalar tags that express the shape parameters: θ_i for the tension, σ_i for the sharpness at V_i if Δ_i non null, and β_i for the bias of this hypothetic sharp crease. All these intuitive tags can easily be set interactively. When a non-null vector Δ_i is set, this means that V_i has 2 normals, one for each side of the sharp crease. In this case, we consider the normal "sharp" N'_i used for a given triangle adjacent to V_i according to the side where this triangle is. This sharp normal is so either $N_i'^{+} = (N_i + \sigma_i\Delta_i)/\|(N_i + \sigma_i\Delta_i)\|$ or $N_i'^{-} = (N_i - \sigma_i\Delta_i)/\|(N_i - \sigma_i\Delta_i)\|$. This normal is used for the shading stage (procedural normal map) described in Appendix A.

Now, to express these local shape parameters for the refinement, we incorporate their scalar tags to the Bezier coefficients of Appendix A. So, we simply associate each tangent coefficient (b_{210}, b_{201}, b_{120}, b_{102}, b_{012}, b_{021}) to its nearest vertex. Let consider the tangent coefficient b_i associated to the vertex V_j, on the edge $\{V_j, V_k\}$. We construct the coefficient in the following way:

$$\begin{aligned}
b_i &= d_i + e_i \\
d_i &= P_j + \frac{(1-\theta_j)}{3}(P_k - P_j) \\
e_i &= (1-\delta_i)\Pi(P_j, X_j, d_i)X_j + \delta_i\Pi(P_j, N_j, d_i)Y_j \\
&\quad \text{where } \Pi(p,n,q) = -n.(q-p) \\
X_j &= \frac{(1-\sigma_j)N_j + N'_j}{\|(1-\sigma_j)N_j + N'_j\|} \\
Y_j &= \frac{N_j + \beta_j\Delta_j}{\|N_j + \beta_j\Delta_j\|}
\end{aligned} \tag{3}$$

Note that some restrictions are applied on the scalar tags to avoid ambiguous cases: at most 1 crease may pass through a given vertex and at most 2 vertices with non-null vector Δ may be defined for a given triangle.