SYNTHESIS OF MULTIRESOLUTION SCENES WITH GLOBAL ILLUMINATION ON A GPU

Raquel Concheiro[1], Margarita Amor[1], Montserrat Bóo[2], Iago Iglesias[1], Emilio J. Padrón[1] and Ramón Doallo[1]

[1]*Dept. Electronics and Systems, Univ. Coruña, E–15071 A Coruña, Spain*

[2]*Dept. Electronic and Comp. Eng., Univ. Santiago de Compostela, E–15782 Santiago de Compostela, Spain*

Keywords: Multiresolution, GPU, Radiosity, Subdivision Surface.

Abstract: The radiosity computation has the important feature of producing view independent results, but these results are mesh dependent and, in consequence, are attached to a specific level of detail in the input mesh. Therefore, rendering at iterative frame rates would benefit from the utilization of multiresolution models. In this paper we focus on the rendering stage of a solution for hierarchical radiosity for multiresolution systems. This method is based on the application of an enriched hierarchical radiosity algorithm to an input scene with low resolution objects (represented by coarse meshes), and the efficient data management of the resulting values. The proposed encoding makes it possible to apply the color values obtained for the coarse objects to detailed versions of these objects during the rendering phase. These finer meshes are obtained by a standard mesh subdivision strategy, such as the Loop subdivision scheme. Our solution performs the whole rendering stage of this multiresolution approach on the GPU, implementing it in the geometry shader using Microsoft HLSL. Results of our implementation show an important reduction in computational costs.

1 INTRODUCTION

Realistic scene walkthroughs are commonly performed in many applications such as virtual reality, simulations, animation, games or geographic information system. Global illumination models, such as the hierarchical radiosity algorithm (Hanrahan et al., 1991), are usually needed to achieve realistic illumination results. Radiosity is view independent, so the results obtained can be re-employed for camera walkthroughs of the scene. An important drawback, however, is the mesh dependence of the radiosity computation that, in consequence, results in obtaining illumination values attached to a specific level of detail in the input mesh. Due to this fact, the obvious approach to get optimum results could be using a very detailed input mesh but, unfortunately, that means high computational costs. Thus, real time rendering of complex scenes would benefit from the utilization of multiresolution models. This could imply radiosity recomputation after each change in the object resolution level when the camera position changes.

Most of the existing approaches to the use of multiresolution models in a radiosity engine (Garland et al., 2001; Gobbetti et al., 2003) are based on the re-computation of the radiosity values in terms of the selected resolution of the objects. A proposal that differs from this scheme is presented in (Padrón et al., 2010). In that work the illumination values are precomputed only once, using the hierarchical radiosity method, and the challenge lies in the application of the fixed colors computed to a multiresolution scene. The different multiresolution meshes are obtained by a standard mesh subdivision strategy, specifically the Loop subdivision scheme in this case (Loop, 1987).

Real time tessellation has been performed on Subdivision Surfaces on GPU (Shiue et al., 2005; Loop and Schaefer, 2008; Patney et al., 2009). Focussing on surface subdivision techniques related with our work, (Shiue et al., 2005) uses multiple textures in order to store the base mesh as a patch and makes use of a precomputed lookup table, so these proposals present a limitation on the subdivision level. In (Loop and Schaefer, 2008) tessellated surface is approximated and in general does not possess the same continuity as the limit surface. In (Patney et al., 2009) a data structure and breadth-first algorithm to perform recursive subdivision of Catmull-Clark on a GPU using CUDA, is presented.

In this paper we take the proposal in (Padrón et al.,

Concheiro R., Amor M., Bóo M., Iglesias I., J. Padrón E. and Doallo R..
SYNTHESIS OF MULTIRESOLUTION SCENES WITH GLOBAL ILLUMINATION ON A GPU.
DOI: 10.5220/0003853402740279
In *Proceedings of the International Conference on Computer Graphics Theory and Applications* (GRAPP-2012), pages 274-279
ISBN: 978-989-8565-02-0

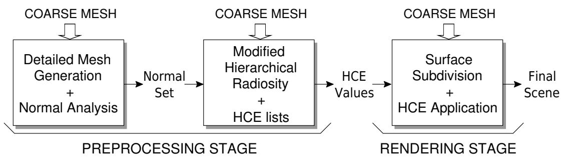
COARSE MESH
Detailed Mesh Generation + Normal Analysis
Normal Set
COARSE MESH
Modified Hierarchical Radiosity + HCE lists
HCE Values
COARSE MESH
Surface Subdivision + HCE Application
Final Scene
PREPROCESSING STAGE
RENDERING STAGE

Figure 1: Scheme of a multiresolution radiosity algorithm.

2010) as a starting point and present an implementation that performs the whole rendering stage of this multiresolution approach on the GPU. This phase basically consists of two steps, first a surface subdivision step and, subsequently, the mapping of colors obtained from coarse meshes of the objects in the scene to more detailed versions of those objects. Both steps are implemented in the geometry shader using the API DirectX 10, with the shaders programmed with Microsoft HLSL. A variant version of the original Loop subdivision scheme, based on a factored approach (Warren and Schaefer, 2004), has been used as it is more appropriate for an efficient implementation on a GPU. Additionally, an analysis of the different alternatives to access the necessary neighboring information is included in this work.

2 HIERARCHICAL RADIOSITY FOR MULTIRESOLUTION SYSTEMS

In (Padrón et al., 2010), the global illumination computation for a synthetic scene with multiresolution objects is performed using the coarse mesh of these objects. Thus, the colors are precomputed only once, using an enriched version of the hierarchical radiosity method to obtain a set of significant pairs normal vector-color per polygon. Subsequently, the main challenge consist in the application of the precomputed colors to the finer meshes of these multiresolution objects. The basic structure of this approach is outlined in Figure 1. The algorithm comprises three main stages, two of them associated with the radiosity computation (preprocessing), and the last one dealing with the final mesh representation (rendering).

The developed solution is based on the analysis of the normal vectors of the multiresolution mesh and the selection of a representative set of normals (*first preprocessing stage*). This set is employed to enrich the hierarchical radiosity algorithm, thus obtaining a set of radiosity values per triangle in the coarse mesh of a multiresolution object (*second preprocessing stage*). These view independent values are efficiently stored and selectively applied to the multiresolution mesh during the last step (*rendering stage*). The proposed encoding, named Hierarchical Color Encoding method (HCE), allows the direct assignment of colors to meshes with any resolution. Note that these computed energy values can be applied to the objects in the scene in any resolution configuration, so high quality scenes with very low computation requirements can be obtained.

The mesh detail level of the multiresolution objects in the final render depends on the camera position and computational cost constraints. This multiresolution scheme is based on a standard mesh subdivision strategy. Among all standard subdivision strategies, the *Loop* subdivision scheme (Loop, 1987) has been used, although the extension for working with other subdivision schemes is straightforward. Once the mesh is conveniently refined, the radiosity values are applied according to the HCE representation and a given normal selection criterion specified.

HCE is a representation method that defines a hierarchical color triangle structure that stores the values obtained from the hierarchical radiosity algorithm. The encoding is based on the following structure: $e_i\ [T_i\ L_i\ C_i\ R_i]$, where e_i is the triangle color, followed by four bits, enclosed in brackets in our notation. Each of these bits is associated with each one of the four child subtriangles (*Top*, *Left*, *Center*, *Right*). A value of 1 means that the subtriangle is in turn subdivided into four children, as a consequence of having been further subdivided during the hierarchical refinement procedure, also coded in the HCE structure. A value of 0 marks that child as a leaf element.

With this notation, the adaptive subdivision hierarchy for any coarse triangle can be represented in a list J, taking the information associated with each level in the hierarchy in a breadth-first way (each subdivision level is listed before starting the next level). To deal with the set of normal vectors per root triangle and the radiosity values associated with them obtained from the global illumination step, the resulting HCE list for a triangle i with normal $\vec{n}_i^0$ and an additional set of m normals ($N_i = \{\vec{n}_i^1, \vec{n}_i^2, \dots, \vec{n}_i^m\}$) would have the following structure:

$$J = s\,m\,N_i\,E_0\ [T_0\ L_0\ C_0\ R_0]\ \ E_1\ [T_1\ L_1\ C_1\ R_1]\dots$$

where E_i is the set of $m+1$ colors ($E_i = \{e_i^0, e_i^1, \dots, e_i^m\}$) computed for each triangle in the hierarchy for each normal vector. The starting bit, s, is added to mark the root triangle as a leaf ($s = 0$) when necessary ($s = 1$ means there are child triangles).

During the rendering stage, once the subdivision level of a multiresolution object is selected and the corresponding model generated, the assignment of colors is performed. A set of colors was computed per subtriangle and listed in the J list, each one associated with a representative normal. When rendering, the final color to be assigned to the triangle is based on a normal comparison criterion. Specifically, if $m+1$ colors were obtained with the radiosity procedure for a triangle i, the correct color ($e_i^k \in E_i, 0 \leq k \leq m$) assigned to a triangle j in the final mesh is the one with the closest normal vector n_i^k

$$\vec{n}_i^k \cdot \vec{n}_j^0 \leq \vec{n}_i^l \cdot \vec{n}_j^0, \ \ l \in \{0...m\}$$

where $\vec{n}_j^0$ is the normal vector of triangle j.

3 SURFACE SUBDIVISION AND COLOR APPLICATION ON A GPU USING THE GEOMETRY SHADER

This section describes the GPU implementation developed for the rendering step of the multiresolution method depicted in Section 2. This last stage of the method dynamically selects the geometric level of detail of each multiresolution object to be rendered as a function of the viewpoint and the associated computational costs. The refined mesh for the selected level of detail is computed by applying the Loop subdivision scheme to the coarse mesh. Afterwards, the appropriate colors obtained from the global illumination step are assigned.

We have implemented both the two tasks involved in this phase, surface subdivision and color mapping, in a kernel for the geometry shader. This kernel is spawned for each triangle in the input mesh and performs the whole computation (both tasks) for that triangle. Notice that our proposal is recomputation-based, so many redundant computations are made due to the geometry shader constraint that prevent the synchronization among tasks. The redundant computations allows multiple arbitrary asynchronous kernel calls and does not affect performance, however. The details of our proposal are detailed below.

3.1 Geometry Shader HLSL Implementation of Surface Subdivision with Loop

Our implementation of subdivision using Loop follows the factored approach introduced in (Warren and Schaefer, 2004). This approach deals with all the vertices in the mesh, the new ones introduced with the subdivision of the edges and the old ones, in a similar way. This feature simplifies the original Loop algorithm, avoiding computations and, more important, branch divergence, common performance killer for

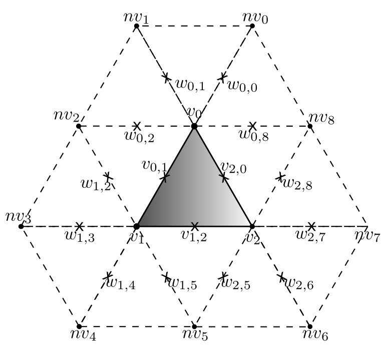

Figure 2: First step in kernel: *Linear Subdivision*.

computations on GPU kernels. To minimize the number of times the subdivision kernel is called in the geometry shader, and to keep the number of output triangles below the recommended 25 (Lorenz and Döllner, 2008) as well, we have unrolled a couple of iterations of the subdivision iterative process in our implementation. Of course, more iterations of the subdivision process may be computed by sending back the results to the pipeline through stream out.

Each input triangle to be subdivided by the kernel, being its three vertices $\{v_0, v_1, v_2\}$ (see Figure 2) needs information about the adjacent triangles sharing a vertex with it, that is, the rest of triangles with vertices $\{v_0, nv_0, v_1\}$, $\{v_1, nv_1, v_2\}$ and so on. A regular vertex is shared by six triangles (valence 6), so a regular case like the one in the figure needs the geometry data of 12 adjacent triangles. The notation used is v_i for the vertices in the triangle to be subdivided, nv_j for neighboring vertices and w_k for the additional newly created vertices needed for the computation.

Each iteration of the subdivision scheme is separated into three different steps: *Linear Subdivision*, *Averaging* and *Correction Factor*. The *Linear subdivision* step (see Figure 2) creates a new vertex as a result of the linear subdivision of every edge in the original input and the adjacent triangles. Therefore, the new vertices $\{v_{0,1}$, $v_{1,2}$, $v_{2,0}$, $w_{0,1}$, $w_{0,2}$, $w_{1,2}$, $w_{1,3}$, $w_{1,4}$, $w_{1,5}$, $w_{2,5}$, $w_{2,6}$, $w_{2,7}$, $w_{2,8}$, $w_{0,8}\}$ are obtained as a result. Only $v_{0,1}$, $v_{1,2}$ and $v_{2,0}$ are inserted in the mesh, though. The rest, vertices $w_{i,j}$, are only used in the next step, *Averaging*, and will be recomputed in other kernel executions for triangles $\{nv_i, v_i, v_j\}$. The *Averaging* step computes the new positions of the vertices v_0, v_1, v_2, $v_{0,1}$, $v_{0,2}$ and $v_{0,4}$, aiming at improving smoothness, this way:

$$v_i' = \frac{1}{4}v_i + \frac{3}{4n}\sum_{j=1}^{n} adjacent\_vertex_j,$$

where n is the valence of v_i.

Lastly, a correction factor, $cf(n)$, is applied to the

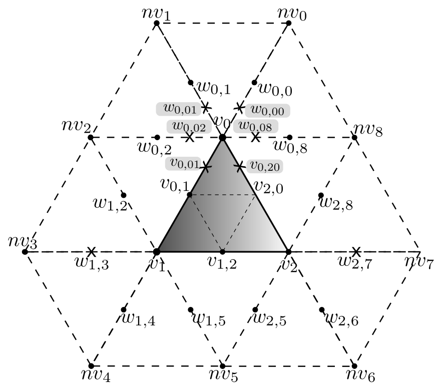

Figure 3: Second iteration in the subdivision process.

relevant vertices (v_0, v_1, v_2, $v_{0,1}$, $v_{0,2}$ and $v_{0,4}$, following with the example of Figure 2) to keep the continuity of the surface as in the original Loop algorithm (C^2 at regular vertices and C^1 at extraordinary vertices). The new location for these vertices will be:

$$v''_i = v_i + cf(n) \cdot \left(v'_i - v_i\right).$$

The correction term is a function of the valence, n:

$$cf(n) = \frac{5}{3} - \frac{8}{3}\left(\frac{3}{8} + \frac{1}{4}\cos\left(2\pi/n\right)\right)^2,$$

so it does not modify regular vertices, that is, $v''_i = v'_i$ when $n = 6$. The terms for each possible value of n are precomputed and stored in the constant memory. This read-only memory, optimized for the access to constants, provides a low latency, similar to register.

After the first iteration, 4 new triangles are created and make up the output of the geometry shader if no more iterations are needed. Otherwise, in the second iteration (as commented above, two iterations are unrolled in our subdivision kernel) 16 new triangles are obtained from those 4 triangles. This second iteration performs again the three steps with the same operations than the previous one. Notice, however, that it is necessary to compute the *Averaging* and *Correction Factor* steps of the first iteration for some vertices whose computation had not been completed during the first iteration (because it was not needed). An example is shown in Figure 3: the vertices $w_{i,j}$ ($w_{0,0}$, $w_{0,1}\ldots$) were obtained by the *Linear Subdivision* step at iter. 1, but the other two steps, *Averaging* and *Correction Factor*, were only applied on the vertices of the triangle being subdivided (v_0, $v_{0,1}\ldots$). Now, we need the final (smooth) position of these vertices to get some of the new vertices for the second iteration.

For example, following the example in Figure 3, the smooth position of $w_{0,1}$ is needed to get the vertex $w_{0,01}$ as the linear interpolation of $v_0 - w_{0,1}$. Exactly the same for the rest of the vertices $w_{i,j}$ in the figure so the *Averaging* and *Correction Factor* steps are applied to these vertices. These computations in turn imply the computation of additional linear interpolations between the vertices nv_j. Finally, after updating the positions of the vertices $w_{i,j}$ the three steps of the second iteration can be performed.

3.2 HLSL Implementation of Color Mapping with HCE

Once the input triangle has been subdivided up to the needed level, the resultant set of triangles has to be rendered. That requires to compute the proper colors for the vertices of these triangles. Since the color values provided by the global illumination phase have been coded for each coarse triangle using the HCE representation, the kernel running in the geometry shader must decode and map the corresponding radiosity values to the new triangles and compute the final color value in the vertices. Therefore, this is a two-step process:

1. First, for each generated triangle the more appropriate color has to be selected from the HCE representation. This color is the color associated with the more similar vector to the normal vector of the sub-triangle among a set of normal vectors associated with the original parent triangle.
2. Once the color of all the sub-triangles has been obtained, the color of every vertex is computed as a weighted average of the colors of the triangles it belongs to. Naturally, besides the colors of the sub-triangles obtained from the input triangle, the colors of the sub-triangles resulting from the neighboring triangles are needed as well, so they are recomputed as times as needed.

3.3 Implementation Details

As it has been commented, the geometric data of the neighboring triangles is needed to perform an accurate Loop subdivision of a triangle. Specifically, we need to access the position of the vertices in the adjacent triangles, 12 triangles and 10 vertices in the regular case, as seen in Figure 2 (vertices nv_0 to nv_9).

We have analyzed three different ways for accessing the information of the neighboring triangles in the kernel. The performance achieved with any of them is analyzed in Section 4. The three alternatives work with the same basic structure `VS_OUTPUT`, defined in Listing 1, that encapsulates the information of each of the input vertices that come from the vertex shader: position, normal vector, color and additional information needed in the kernel (the field named `STATUS`), Listing 1: Geometry shader input and vertex shader output.

```
struct VS_OUTPUT {
  float4 vPosition : POSITION0;
  float3 vNormal : NORMAL;
  float4 vColor : COLOR0;
  float4 ints : STATUS;
}
```

such as the vertex index, whether it is an inner vertex o not, whether it is part of a multiresolution object or not and its valence (number of triangles that vertex belongs to).

As it has been commented, there are recomputation in the three proposed alternatives as every input triangle is assigned to a computational core with no possibility of sharing information among cores.

Neighboring Information in Texture Memory (NITM). This first implementation gets all the neighboring information needed during the kernel computation from the texture memory. All the triangles are indexed using the `PrimitiveId` generated by the system. The signature of the function for the geometry shader is:

```
void GS(triangle VS_OUTPUT v[3], inout
       TriangleStream <GS_OUTPUT> TriStream,
        uint PrimitiveId : SV_PRIMITIVEID)
```

Regarding the function prototype, the first parameter sets an input of 3 vertices of type `VS_OUTPUT` for the geometry shader. With the primitive type `triangle`, this means a triangle $\{v[0], v[1], v[2]\}$.

Triangle with Adjacent Structure (TA). The information of the three adjacent triangles that share an edge with the triangle to be subdivided is passed directly to the kernel, using a triangle with an adjacent structure as input. This way, we take advantage of the `triangleadj` primitive of geometry shader, that makes this adjacent information directly accessible in the kernel. Rest of the neighboring data is still accessed through the texture memory. Now the function has this prototype:

```
void GS(triangleadj VS_OUTPUT v[6], inout
       TriangleStream <GS_OUTPUT> TriStream,
        uint PrimitiveId : SV_PRIMITIVEID)
```

Therefore, the first parameter sets now an input of 6 vertices. As the primitive type `triangleadj` indicates, these vertices are interpreted as a triangle $\{v[0], v[1], v[2]\}$ and three adjacent vertices.

Neighboring Information included in VS_OUTPUT (VS_OUTPUT). Instead of using the adjacency structure as input data, we modify the `VS_OUTPUT` structure for keeping the necessary neighboring information, avoiding the costly accesses to texture memory. The new `VS_OUTPUT` simply adds this field to the structure in Listing 1:

```
6   float4 neighbors[12] : NEIGHBORS;
```

The prototype of the kernel is the same that in the NITM case, passing just the three vertices of the triangle to be processed. Adding all the neighboring data needed for the computation to the three input vertices may seem quite redundant, but although we are replicating part of the neighboring information, this is completely compensated by the increase in the performance achieved by avoiding the texture memory accesses. As a matter of fact, there is only needed to access texture memory in the infrequent case when the number of adjacent triangles is greater than 12.

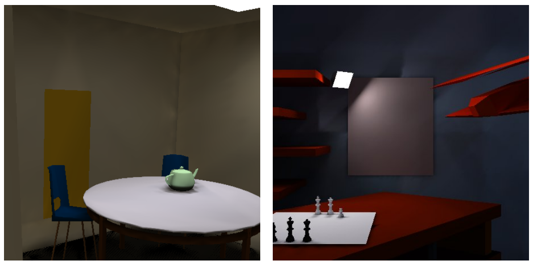
(a) (b)

Figure 4: Test scenes (a) *Teatime* (b) *Chesstime*.

4 EXPERIMENTAL RESULTS

In this section we discuss the performance of our GPU proposal for the rendering phase of the target multiresolution system. All the results have been obtained in a computer with processor Intel Core i7 920 2.67 GHz, 6 GB of RAM and a video card ATI Radeon 5870. The software platform was Microsoft Windows 7 Ultimate 64 bits as operative system and the Microsoft Visual C++ 2008 compiler with HLSL Shader Model 4.0 and DirectX 10.

In the following, we include the results obtained with two of the scenes employed in our tests: *Teatime* (see Figure 4(a)) and *Chesstime* (see Figure 4(b)). Among the different objects in these scenes, we have employed several multiresolution models: a teapot, a cup and a spoon for *Teatime* scene and several chess pieces for *Chesstime* (specifically two kings, two queens, one pawn and one bishop).

In Table 1 we show the results (frames per second) obtained for the surface subdivision of the multiresolution objects in the test scenes by the three implementations described in Section 3.3. As can be

Table 1: Comparison of the three proposed alternatives (fps for surface subdivision).

OBJECT	NITM	TA	VS_OUTPUT	OBJECT	NITM	TA	VS_OUTPUT
Pawn	1840	1966	4063	*Teapot*	1720	1873	3970
Bishop	1879	1989	4063	*Spoon*	1701	1810	3907
Queen	1850	1975	3998	*Cup*	1772	1920	3970
King	1820	1960	3975				

Table 2: Frames per second for rendering the multiresolution objects in the test scenes.

OBJECT	1^{st} it.	2^{nd} it.	OBJECT	1^{st} it.	2^{nd} it.
Pawn	2750	662	*Teapot*	2726	642
Bishop	2752	665	*Spoon*	2750	668
Queen	2735	660	*Cup*	2755	625
King	2742	659			

observed, all the results are more than enough for real time rendering. The worst results have been obtained by the NITM approach, due to the accesses to texture memory. There is an average improvement of 6.8% in the TA proposal as texture memory accesses to vertices included in the adjacency structure are prevented. The table shows that the VS_OUTPUT implementation gets an improvement of 120.8% and 106.6% regarding NITM and TA, respectively. This increase in the performance is produced by practically avoiding any access to texture memory. Since this option is obviously the best approach, the rest of the results shown in this sections refer exclusively to it.

Table 2 presents the rendering results (frames per second) for the whole rendering (surface subdivision + assignment of color) of the multiresolution objects for one and two levels of subdivision, whereas the results for the whole scenes for two iterations are 530 and 260 fps, respectively for *Teatime* and *Chesstime*. FPS have been measured for the worst case, that is, when all the multiresolution objects are rendered. As can be observed, real time rendering has been achieved in all the cases, and the differences in rendering time between the two scenes are due to the number of primitives to process in both cases.

5 CONCLUSIONS

An efficient GPU implementation of the Loop surface subdivision scheme has been presented in this work. This implementation has been integrated in a multiresolution system, mapping the color results obtained from global illumination in the subdivided objects. Among the different options analyzed for performing tessellation on a GPU, we have opted for using the geometry shader. Even so, our proposal is recomputation-based, so many redundant computations are made because of the geometry shader constraint that does not allow the synchronization among tasks. However, these redundant computations does not deteriorate the performance, as have been shown.

Furthermore, an analysis of three different options for accessing the necessary neighboring information in the geometry shader has been done. As a result, directly passing data though vertex buffer has got the best results since most of the accesses to texture memory are avoided. Our implementation achieves a great performance, rendering all the test scenes in real time.

REFERENCES

Garland, M., Willmot, A., and Heckbert, P. H. (2001). Hierarchical face clustering on polygonal surfaces. In *I3D '01: Proceedings of the 2001 Symposium on Interactive 3D Graphics*, pages 49–58, Research Triangle Park, NC. ACM, New York, NY, USA.

Gobbetti, E., Spanò, L., and Agus, M. (2003). Hierarchical higher order face cluster radiosity for global illumination walkthroughs of complex non-diffuse environments. *Computer Graphics Forum*, 22(3):563–572.

Hanrahan, P., Saltzman, D., and Aupperle, L. (1991). A rapid hierarchical radiosity algorithm. *SIGGRAPH Comput. Graph.*, 25(4):197–206.

Loop, C. (1987). Smooth subdivision surfaces based on triangles. Master's thesis, University of Utah, Dept. of Mathematics.

Loop, C. and Schaefer, S. (2008). Approximating catmull-clark subdivision surfaces with bicubic patches. *ACM Transactions on Graphics*, 1.

Lorenz, H. and Döllner, J. (2008). Dynamic mesh refinement on GPU using geometry shaders. In *Proc. of 16th Int. Conf. in Central Europe on Computer Graphics, Visualization and Computer (WSCG'08)*.

Padrón, E. J., Amor, M., Bóo, M., and Doallo, R. (2010). Hierarchical radiosity for multiresolution systems based on normal tests. *The Computer Journal*, 53(6):741–752.

Patney, A., Ebeida, M. S., and Owens, J. D. (2009). Parallel View-Dependent Tessellation of Catmull-Clark Subdivision Surfaces. In *Proc. of the Conference on High Performance Graphics 2009 (HPG'09)*, pages 99–108.

Shiue, L.-J., Jones, I., and Peters, J. (2005). A Real-time GPU Subdivision Kernel. *ACM Trans. Graph.*, 24(3):1010–1015.

Warren, J. and Schaefer, S. (2004). A factored approach to subdivision surfaces. *IEEE Computer Graphics and Applications*, 24(3):74–81.