Generating subdivision curves with L−systems on a GPU

Radomir Mech[1] and Przemyslaw Prusinkiewicz[2]
[1]SGI, Mountain View, California.
[2]University of Calgary, Calgary, Canada.

Abstract

The introduction of floating−point pixel shaders has initiated a trend of moving algorithms from CPUs to graphics cards. The first algorithms were in the rendering domain, but recently we have witnessed increased interest in modeling algorithms as well. In this paper we present techniques for generating subdivision curves on a modern Graphics Processing Unit (GPU). We use an existing method for generating subdivision curves with L−systems, we extend these L−systems to implement adaptive subdivision, and we show how these L−systems can be implemented on a GPU. We chose L−systems because they can express many modeling algorithms in a compact way and are parallel in nature, making them an attractive paradigm for programming a GPU.

Reference

Radomir Mech and Przemyslaw Prusinkiewicz: Generating subdivision curves with L−systems on a GPU. *SIGGRAPH 2003 Sketches and Applications*.

Generating subdivision curves with L-systems on a GPU[*]

Radomír Měch[†]
SGI

Przemyslaw Prusinkiewicz[†]
University of Calgary

Abstract

The introduction of floating-point pixel shaders has initiated a trend of moving algorithms from CPUs to graphics cards. The first algorithms were in the rendering domain, but recently we have witnessed increased interest in modeling algorithms as well.

In this paper we present techniques for generating subdivision curves on a modern Graphics Processing Unit (GPU). We use an existing method for generating subdivision curves with L-systems, we extend these L-systems to implement adaptive subdivision, and we show how these L-systems can be implemented on a GPU.

We chose L-systems because they can express many modeling algorithms in a compact way and are parallel in nature, making them an attractive paradigm for programming a GPU.

1 Introduction

In recent years subdivision curves became an important alternative to parametric curves in computer aided design. For a modeler they are very attractive because a complex curve can be defined using a small number of control points.

The new programmable graphics hardware with capabilities of executing a set of instructions during the vertex or fragment processing has proven to be capable of solving difficult processing tasks. Various algorithms have been implemented on these Graphics Processing Units (GPUs), ranging from ray-tracing [6] to solving differential equations [2]. Considering the parallel nature of subdivision algorithms the new graphics hardware is a suitable candidate for implementing them.

In this note we review L-systems that capture different subdivision scheme, we extend the L-systems presented in [5] with support for adaptive subdivision of curves, and we show how to implement these L-systems on a GPU[1].

GPUs can be programmed using assembler level languages or higher level languages, such as Cg [3] or Direct X 9.0 HLSL[2]. We chose to implement L-systems using the assembler level language.

[*]This is an extended version of a sketch to be presented at SIGGRAPH 2003.

[†]rmech@sgi.com, pwp@cpsc.ucalgary.ca

[1]Our implementation and description are based on the ATI Radeon 9700 card (http://www.ati.com/developer).

[2]http://msdn.microsoft.com/directx/

2 Generating subdivision curves

Subdivision curves can be described using context-sensitive parametric L-systems [5]. Control points of the curve are stored as symbols in the initial string, with parameters specifying point locations[3]. L-system productions are used to replace each point with new points according to a given subdivision scheme. For example, Chaikin subdivision of a closed curve is captured by a single production [5],

L-system 1:

$$P(\mathbf{v}_l) < P(\mathbf{v}) > P(\mathbf{v}_r) \rightarrow P(\tfrac{1}{4}\mathbf{v}_l + \tfrac{3}{4}\mathbf{v})P(\tfrac{3}{4}\mathbf{v} + \tfrac{1}{4}\mathbf{v}_r),$$

which replaces one point, the strict predecessor, with two new points, forming the successor. The location of each new point is an affine combination of the locations v, v_l and v_r of the predecessor point and its context (neighbors).

It is easy to modify L-system 1 to express different subdivision schemes. Each scheme is using different affine combination of the neighbors. Some schemes presented in [5] are using more than one neighbor on each side of the point, but not more than two. Thus we can combine these L-systems in a single scheme:

L-system 2:

$$P(\mathbf{v}_0)P(\mathbf{v}_1) < P(\mathbf{v}_2) > P(\mathbf{v}_3)P(\mathbf{v}_4) \rightarrow P(\textstyle\sum_{i=0}^{4} a[i].\mathbf{v}_i)P(\sum_{i=0}^{4} b[i].\mathbf{v}_i),$$

where arrays a and b store parameters of the affine combination for each new symbol. L-system 2 can express Chaikin subdivision scheme using values $a = \{0, \frac{1}{4}, \frac{3}{4}, 0, 0\}$ and $b = \{0, 0, \frac{3}{4}, \frac{1}{4}, 0\}$, cubic B-spline subdivision using $a = \{0, \frac{1}{8}, \frac{3}{4}, \frac{1}{8}, 0\}$ and $b = \{0, 0, \frac{1}{2}, \frac{1}{2}, 0\}$, and Dyn-Levin-Gregory (4-point) subdivision using $a = \{0, 0, 1, 0, 0\}$ and $b = \{0, -\frac{1}{16}, \frac{9}{16}, \frac{9}{16}, -\frac{1}{16}\}$.

In the case of open subdivision curves, end points of the curve do not change location and the rules for creating new points in their neighborhood are different from those operating farther from the endpoints. If we denote the endpoints by symbol E, we can expand L-system 1 to open curves as follows [5]:

L-system 3:

p_1: $E(\mathbf{v}_l) < P(\mathbf{v}) > P(\mathbf{v}_r) \rightarrow P(\frac{1}{2}\mathbf{v}_l + \frac{1}{2}\mathbf{v})P(\frac{3}{4}\mathbf{v} + \frac{1}{4}\mathbf{v}_r)$

p_2: $P(\mathbf{v}_l) < P(\mathbf{v}) > E(\mathbf{v}_r) \rightarrow P(\frac{1}{4}\mathbf{v}_l + \frac{3}{4}\mathbf{v})P(\frac{1}{2}\mathbf{v} + \frac{1}{2}\mathbf{v}_r)$

p_3: $P(\mathbf{v}_l) < P(\mathbf{v}) > P(\mathbf{v}_r) \rightarrow P(\frac{1}{4}\mathbf{v}_l + \frac{3}{4}\mathbf{v})P(\frac{3}{4}\mathbf{v} + \frac{1}{4}\mathbf{v}_r)$

p_4: $E(\mathbf{v}) \rightarrow E(\mathbf{v})$

L-system 3 can be generalized in a similar manner to L-system 1. To this end, we extend L-system 3 with two new productions, in which the symbol E is two symbols away

[3]We make here a distinction between the *location* of a point (three coordinates) and its *position* in the string (an index value).

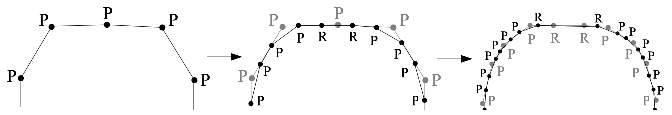

Figure 1: Operation of L-system 5. Points of type 0 are marked as R.

from the predecessor, and we define arrays a and b for each production:

L-system 4:

p_1: $E(\mathbf{v}_0)P(\mathbf{v}_1) < P(\mathbf{v}_2) > P(\mathbf{v}_3)P(\mathbf{v}_4)$
$\rightarrow P(\sum_{i=0}^{4} a[0][i].\mathbf{v}_i)P(\sum_{i=0}^{4} b[0][i].\mathbf{v}_i)$

p_2: $E(\mathbf{v}_1) < P(\mathbf{v}_2) > P(\mathbf{v}_3)P(\mathbf{v}_4)$
$\rightarrow P(\sum_{i=1}^{4} a[1][i].\mathbf{v}_i)P(\sum_{i=1}^{4} b[1][i].\mathbf{v}_i)$

p_3: $P(\mathbf{v}_0)P(\mathbf{v}_1) < P(\mathbf{v}_2) > P(\mathbf{v}_3)E(\mathbf{v}_4)$
$\rightarrow P(\sum_{i=0}^{4} a[2][i].\mathbf{v}_i)P(\sum_{i=0}^{4} b[2][i].\mathbf{v}_i)$

p_4: $P(\mathbf{v}_0)P(\mathbf{v}_1) < P(\mathbf{v}_2) > E(\mathbf{v}_3)$
$\rightarrow P(\sum_{i=0}^{3} a[3][i].\mathbf{v}_i)P(\sum_{i=0}^{3} b[3][i].\mathbf{v}_i)$

p_5: $P(\mathbf{v}_0)P(\mathbf{v}_1) < P(\mathbf{v}_2) > P(\mathbf{v}_3)P(\mathbf{v}_4)$
$\rightarrow P(\sum_{i=0}^{4} a[4][i].\mathbf{v}_i)P(\sum_{i=0}^{4} b[4][i].\mathbf{v}_i)$

p_6: $E(\mathbf{v}) \rightarrow E(\mathbf{v})$

L-systems provide a compact way of defining subdivision curves and they are easy to modify. For example, let us expand L-system 3 to support adaptive subdivision of open curves (see L-system 5). For this purpose, we add a second parameter t specifying the type of a point to each symbol P. This parameter is equal to 1 (the default) if the point is to be subdivided and 0 if it should not be subdivided any further. We also extend L-system 3 with three new productions. Productions p_1 and p_2 make sure that the point of type 0 next to a point of type 1 creates only one new point and not two. Production p_3 tests whether the point is close to the midpoint between its neighbors, in which case the newly created points are of type 0. Our approach is similar to the one described by Xu *et al.* [7]. Here is the resulting L-system:

L-system 5:

p_1: $P(\mathbf{v}_l, t_l) < P(\mathbf{v}, t)$: $t = 0 \;\&\; t_l = 1 \rightarrow P(\frac{1}{4}\mathbf{v}_l + \frac{3}{4}\mathbf{v}, 0)$

p_2: $P(\mathbf{v}, t) > P(\mathbf{v}_r, t_r)$: $t_r = 1 \;\&\; t = 0 \rightarrow P(\frac{3}{4}\mathbf{v} + \frac{1}{4}\mathbf{v}_r, 0)$

p_3: $P(\mathbf{v}_l, t_l) < P(\mathbf{v}, t) > P(\mathbf{v}_r, t_r)$: $|\mathbf{v} - \frac{\mathbf{v}_l + \mathbf{v}_r}{2}| < T$
$\rightarrow P(\frac{1}{4}\mathbf{v}_l + \frac{3}{4}\mathbf{v}, 0)P(\frac{3}{4}\mathbf{v} + \frac{1}{4}\mathbf{v}_r, 0)$

p_4: $E(\mathbf{v}_l) < P(\mathbf{v}, t) > P(\mathbf{v}_r, t_r)$
$\rightarrow P(\frac{1}{2}\mathbf{v}_l + \frac{1}{2}\mathbf{v}, t)P(\frac{3}{4}\mathbf{v} + \frac{1}{4}\mathbf{v}_r, t)$

p_5: $P(\mathbf{v}_l, t + l) < P(\mathbf{v}, t) > E(\mathbf{v}_r)$
$\rightarrow P(\frac{1}{4}\mathbf{v}_l + \frac{3}{4}\mathbf{v}, t)P(\frac{1}{2}\mathbf{v} + \frac{1}{2}\mathbf{v}_r, t)$

p_6: $P(\mathbf{v}_l, t_l) < P(\mathbf{v}, t) > P(\mathbf{v}_r, t_r)$
$\rightarrow P(\frac{1}{4}\mathbf{v}_l + \frac{3}{4}\mathbf{v}, 1)P(\frac{3}{4}\mathbf{v} + \frac{1}{4}\mathbf{v}_r, 1)$

p_7: $E(\mathbf{v}) \rightarrow E(\mathbf{v})$

In this L-system we are taking advantage of the assumption that if more than one production can be used to rewrite the predecessor, the one that appears first in the production list is chosen. For example, the third production is applied only to symbols to which the first or second production cannot be applied. Figure 1 illustrates the operation of L-system 5.

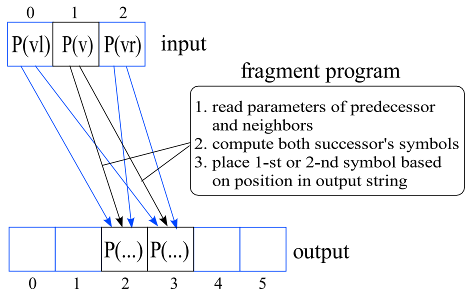
0
1
2
P(vl)
P(v)
P(vr)
input
fragment program
1. read parameters of predecessor and neighbors
2. compute both successor's symbols
3. place 1-st or 2-nd symbol based on position in output string
P(...)
P(...)
output
0
1
2
3
4
5

Figure 2: An L-system on GPU, algorithm 1: each symbol is replaced by two new symbols.

In the next section we implement these L-systems directly on a GPU.

3 L-systems on a GPU

Algorithm 1. An L-system in which each symbol is replaced by a constant number of k symbols (for example, L-system 1 or L-system 2) is easy to implement on graphics hardware that supports floating-point fragment programs (also known as pixel shaders) (Figure 2). We store the initial string in one line of a texture[4]. The letter symbol of each point is in the alpha channel, and the coordinates are in the RGB channels. Given an input string of length n, we draw a line of length kn into a P-buffer, off-screen memory located on the graphics card. A pixel of the line at position i represents the $i\%k$-th point of the successor of the i/k-th symbol in the input string. As the line is rendered, the fragment program reads texel values at positions $(i/k - 1)\%n$, $(i/k)\%n$ and $(i/k + 1)\%n$ (the left context, the strict predecessor, and the right context), and sets the value of pixel i as defined for the $i\%k$-th point of the production successor. The positions of the predecessor and neighbors are deduced from three sets of texture coordinates. The texture coordinates of neighbors are shifted to the left and right from the predecessor coordinates. The value of i used to determine the symbol of the successor is set using a 1D texture coordinate with values of 0 and kn assigned with the two vertices of the line.

Once the symbol of the successor is identified, the fragment program has to compute symbol's parameters. If the computations for all successor's symbols are similar, such as in case of L-system 2, they can be performed by a single fragment program. This program uses a set of local fragment program parameters or an input texture to specify different parameters for each computation (equivalent to arrays a and b in L-system 2). The correct set of parameters is selected based on the symbol's position i in the final string. If the computations vary significantly, they cannot be expressed by a single formula that uses different parameters for different symbols of the successor. In this case we can apply a fragment program that computes all symbols of the successor

[4]If one line is not enough, we modify the neighbor selection process in order to store the string in a 2D texture.

and selects the one identified by the position i. If these computations do not fit into a single fragment program, we can use a set of fragment programs applied one after another, each setting only a particular symbol of the successor. This will be less of an issue in the future, because the maximum length of a fragment program will be significantly larger.

In each subsequent iteration of the algorithm, we bind the P-buffer as the input texture and use another P-buffer as the output. Finally, we read the final string using glReadPixels, and render the vertices. In the near future, the drivers will support rendering into a vertex array, which will make it possible to avoid the readback.

Algorithm 2. If an L-system has more than one production, and they have successors of different length (for example, L-system 3) there are two issues: to find a production for each symbol, and to position the successor in the output string. There are two approaches to finding the production. If the productions are of a similar form and the coefficients used to compute the successor's parameters can be tabulated, such as in L-system 3, 4 or 5, two fragment programs can be used, one to find an applicable production and one to apply it. These programs use textures that specify the correspondence between a specific predecessor and its successor, given the predecessor's context (see below for more details). If L-system productions vary significantly, it is necessary to represent each production or a group of similar productions using a separate fragment program.

The first approach is more desirable because it is easy for a user to modify the L-system by changing texture data without any changes to fragment programs. All productions are specified using two textures, the *predecessor texture* and the *successor texture*. Each row of the predecessor texture stores information on the context of all productions with the same strict predecessor. The productions are specified one after another, each production is specified by its four neighbors, the successor length and the index of the first symbol of the successor in the successor texture (see Figure 3). Optionally, for each production, the row can also store coefficients used to evaluate the production's condition. Each column of the successor texture stores the symbols and affine combination coefficients for one successor symbol of one production.

Figure 3 illustrates the operation of an L-system using textures organized as described above. Fragment program 1 finds the matching production for each point in the predecessor string, and outputs the successor length l and the index s of the first symbol of the successor, stored in the successor texture. Since the program tests one set of neighbors at a time, this takes up to M passes, where M is the maximum number of productions with the same strict predecessor.

To determine the position of each successor in the output string, we simulate the scan-add operation defined as foolows [4]: if $y =$scan-add(x), then $y[i] = \sum_{j=0}^{i-1} x[j]$ (and $y[0] = 0$)[5]. Before the productions are applied we run fragment program 2, which sums the lengths of all successors to

[5]We use the version of the scan-add operation, in which we do not add the value at the given position to the sum.

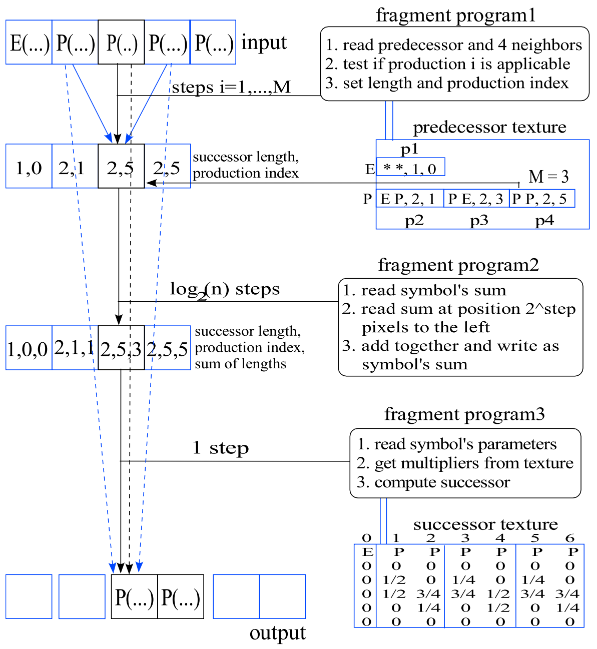

Figure 3: An L-system on GPU, algorithm 2: productions specified using textures, successor lengths vary. Texture data correspond to L-system 3.

the left of a given symbol. This can be done in $\lfloor log_2(n) \rfloor$ passes. These sums are read with glReadPixels and used to create a set of line segments on a CPU, each starting at the pixel given by a sum. Again, the readback can be avoided once rendering into vertex arrays is supported in drivers.

The 1D texture coordinates at vertices of each line segment are set to s and $s + l$. Fragment program 3, executed for each pixel of each line segment, accesses the successor texture column identified by the 1D texture coordinate. It retrieves the symbol and its affine combination coefficients from the texture, computes the affine combination of the predecessor point and its neighbors, and sets the new symbol and the computed value.

If we have a set of productions whose successors have the same length, the scan-add step can be skipped. A single line of length kn is drawn as in algorithm 1 and the position i is used to determine the symbol of the successor in fragment program 3. Sometimes we can determine the successor from the position i even if the productions have successors of different length. In L-system 3, for example, only the first and last symbol in the string produce one new symbol, all other symbols produce two, and therefore the position of each successor can be determined in advance.

In the subsequent iteration of the subdivision process, the P-buffer is used as a input texture for the fragment programs. The final string is read with glReadPixels, and the vertices are rendered as in the closed curve case.

4 Results

Figures 4 and 5 show sample subdivision curves generated using L-systems 2, 3 and 4 implemented on the ATI's

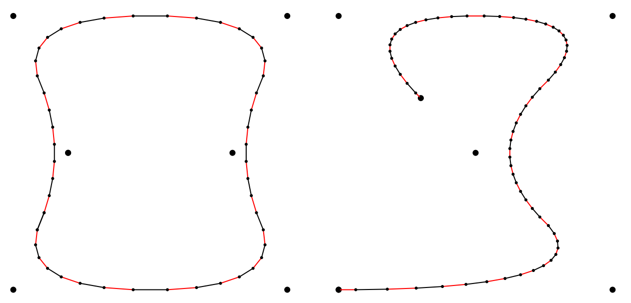

Figure 4: Closed and open subdivision curves generated in 3 steps using L-system 2 and L-system 3 implemented on a GPU.

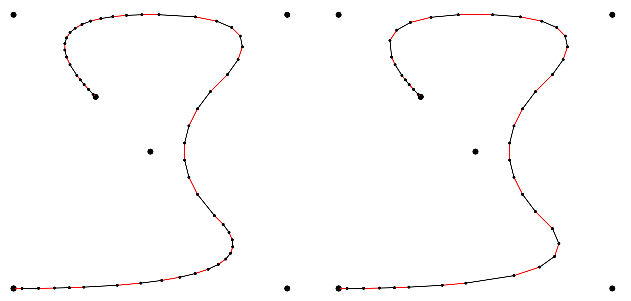

Figure 5: Adaptive subdivision curves generated in 5 steps using L-system 5 implemented on a GPU. $T = 0.025$ and 0.05.

Radeon 9700.

In the case of the closed curve in Figure 4, we used algorithm 1. A single fragment program generates both new points of the successor in a single rendering pass. The arrays a and b (L-system 2) are set using local parameters of the fragment program. The program has 15 instructions (12 arithmetic instructions and 3 texture reads). It took 0.4 ms to generate the closed curve in Figure 4, out of which 0.3 ms were spent in switching the rendering context from one P-buffer to another[6]. One context switch took about 0.1 ms, but the future drivers should significantly reduce this unnecessary overhead [1]. The overhead of context switches is also reduced if several curves are evaluated at once. Subdividing a curve defined by 4 control points 8 times (subdivision level 8) resulted in 1024 points and took (8*0.1 + 0.2) ms. These times do not include the final readback, which for 1024 points takes about 0.17ms.

Using a software implementation on a 2.4 GHz Pentium 4 CPU to generate three levels of subdivision took about the same time (0.1 ms), but at higher subdivision levels the GPU implementation became faster (if we ignore the context switch overhead). At subdivision level 8, the GPU was about twice as fast as the CPU.

In the case of open curves we used algorithm 2. The L-system is automatically converted into the predecessor texture and successor texture. Fragment program 1 has 45 instructions (35 arithmetic instructions + 10 texture reads), fragment program 2 has 24 instructions (16+8)[7], and fragment program 3 has 18 instructions (15+3). It took 2.1 ms to generate the open curve in Figure 4, out of which 1.35 ms were spent on 11 context switches and 0.3 ms on 3 readbacks after each scan-add operation. The overall time of 2.1 ms can be reduced by 0.9 ms (5 context switches + 0.4 ms) by skipping the scan-add operation, because in L-system 3 the position of each production successor can easily be determined (see Section 3). The timings for the two adaptive curves in Figure 5 are 3.8 ms (2.4 ms for 20 context switches, 0.5 ms for 5 scan-add readbacks) and 3.7 ms (2.3 ms for 19 context switches, 0.5 ms for 5 scan-add readbacks). In this case we cannot skip the scan-add step.

The software implementation of open subdivision curves is faster than the GPU implementation for a small number of control points. Subdividing a non-adaptive open curve from Figure 4 up to level 8 was 4 times faster in software (ignoring the cost of context switches). The GPU disadvantage is caused by having to perform several rendering passes to find a production, and several passes to perform scan-add operation, while dealing with a relatively small number of pixels. Once we increase the number of pixels by evaluating several curves in parallel the GPU algorithm becomes faster. Evaluating 16 non-adaptive open curves (8 subdivision levels) took about the same time on the CPU and the GPU, and for 32 curves the GPU was about 50% faster. Consequently, using the GPU for evaluating subdivision curves is better only if one needs to evaluate many of them at once.

5 Conclusions

We created a set of fragment programs on ATI's Radeon 9700 that implement L-systems capable of generating subdivision curves. We implemented not only selected basic schemes, but also an adaptive scheme.

We chose L-systems as a conceptual basis for our implementation because they compactly express many subdivision schemes and they can easily be modified by changing few parameters or adding a few new productions. Our approach is similar to the implementation of L-systems on the Connection Machine by Ortiz *et al.* [4]. In contrast to Ortiz *et al.*, however, our implementation supports parametric L-systems and uses different data structures, more suitable for fast access by a GPU.

As the results indicate, if we have to perform more than one rendering pass for a single subdivision step the GPU implementation becomes faster compared to a CPU implementation when many curves are evaluated at once. An additional problem is the fact that current drivers do not implement switches of rendering context very efficiently and that the API for rendering to vertex arrays has not been finalized yet and thus the drivers lack the support for this functionality.

[6]We have to alternate between two P-buffers because a single P-buffer cannot be used both as an input and output.

[7]We observed that it is faster to use a longer fragment program 2 that sums 8 values at once and reduce the number of passes needed for the scan-add operation than to use a shorter program in more passes (even if we ignore the cost of context switches).

An intriguing problem for further research is an extension of this work to subdivision surfaces, where the advantage of a GPU implementation is likely to be more significant, because we are dealing with larger numbers of points.

Acknowledgemens

We would like to thank Sylvain Lefebvre for helpful comments on this note.

References

[1] J. Bolz and P. Schröder. Evaluation of subdivision surfaces on programmable graphics hardware. http://www.multires.caltech.edu/pubs/GPUSubD.pdf. Submitted for publication.

[2] N. Goodnight, G. Lewin, D. Luebke, and K. Skadron. A multigrid solver for boundary-value problems using programmable graphics hardware. Technical Report CS-2003-02, Univ. of Virginia Dept. of Computer Science., January 2003.

[3] W. R. Mark, S. Glanville, and K. Akeley. Cg: A System for Programming Graphics Hardware in a C-like Lanuage. *ACM Transactions on Graphics*, 22(3), July 2003. To appear.

[4] L.F. Ortiz, R.Y. Pinter, and S.S. Pinter. An array language for data parallelism: Definition, compilation, and applications. *The Journal of Supercomputing*, (5):7–29, 1991.

[5] P. Prusinkiewicz, F. Samavati, C. Smith, and R. Karwowski. L-system description of subdivision curves. *International Journal of Shape Modeling*. To appear..

[6] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):703–712, July 2002.

[7] Z. Xu and K. Kondo. Local Subdivision Process with Doo-Sabin Subdivision Surfaces. *SMI 2002:International Conference on Shape Modelling and Applications*, May 2002.