Towards Hardware Implementation Of Loop Subdivision

Stephan Bischoff   Leif P. Kobbelt   Hans-Peter Seidel

Max-Planck-Institute for Computer Sciences[*]

Abstract

We present a novel algorithm to evaluate and render Loop subdivision surfaces. The algorithm exploits the fact that Loop subdivision surfaces are piecewise polynomial and uses the forward difference technique for efficiently computing uniform samples on the limit surface. The main advantage of our algorithm is that it only requires a small and constant amount of memory that does not depend on the subdivision depth. The simple structure of the algorithm enables a scalable degree of hardware implementation. By low-level parallelization of the computations, we can reduce the critical computation costs to a theoretical minimum of about one `float[3]`-operation per triangle.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture—Graphics processors I.3.3 [Computer Graphics]: Picture/Image Generation—Display algorithms

Keywords: subdivision surfaces, rendering, forward differences, hardware implementation

1 INTRODUCTION

Due to their flexibility to model smooth surfaces of arbitrary shape, subdivision surfaces are receiving more and more attention in the area of geometric modeling. After the analysis of subdivision schemes has been investigated over the last decade [15, 18], the related techniques are now entering all areas of applied computer graphics [5, 7]. Since subdivision meshes are considered to fill the gap between plain triangle meshes and sophisticated NURBS-representations, they are used whenever the efficiency of polygonal meshes has to be combined with the superior surface quality of NURBS.

In current implementations, subdivision surfaces are usually treated as high level representations of surface geometry. Before rendering a subdivision surface on the screen it is converted into an ordinary triangle mesh by refining the given control mesh sufficiently. A natural question that arises in this context is whether subdivision surfaces could be handled as a new rendering primitive, i.e., as a basic piece of geometry that is passed to the graphics sub-system directly. To achieve this, the evaluation procedure for subdivision surfaces has to move beyond the graphics API and ideally should be implemented in hardware.

[*]Im Stadtwald, 66123 Saarbrücken, Germany
Email: {bischoff,kobbelt,hpseidel}@mpi-sb.mpg.de

There are two major challenges: The first is to define a general interface through which arbitrary control meshes can be transferred to the graphics system and the second is to find an algorithm which is suitable for hardware implementation, the central requirements being: simple algorithmic structure (few special cases) and small (constant) memory requirements.

The most commonly used subdivision scheme in graphics is Loop's scheme since it works on arbitrary triangle meshes and generates curvature continuous surfaces in the limit [11, 18]. Hence, we are focussing on the evaluation of Loop subdivision surfaces throughout the paper. After discussing related work, we recollect some facts on Loop subdivision. In Section 3 we explain the forward difference technique which is the basic tool for our evaluation algorithm in Section 4. It's space, memory and precision requirements are discussed in Section 5. Finally we show some experimental results in Section 6. Additional implementation details are given explicitly in the appendix.

1.1 Previous work

Several authors have investigated this topic before [9, 12, 14]. The naive approach of first generating a refined mesh and then passing its triangles to the shading pipeline is not appropriate in most cases since the memory requirements increase exponentially with the refinement depth.

Consequently, Kohler and Müller [9] reduce the complexity of the data structure by only refining along a front which advances across the surface. Their general technique applies to any tensor-product subdivision scheme. Pulli and Segal [14] focus on Loop subdivision and further reduce the storage needs by processing local surface patches individually. However, in order to simplify the data management, they perform a preprocessing which clusters pairs of triangular patches to quadrilateral ones. Müller and Havemann [12] extend this approach by taking adaptive refinement strategies into account during the clustering.

In all proposed solutions the storage requirements still depend on the subdivision depth down to which the initial control mesh has to be refined. The underlying rationale is always the same: change the computation order from breadth-first (standard subdivision) to depth-first and maintain only the "active" part of the control mesh. Obviously this idea applies to non-polynomial subdivision schemes as well.

Our algorithm, in contrast, specifically exploits the knowledge about the piecewise polynomial structure of Loop subdivision surfaces (see also [17]). It can be modified to work for the Doo/Sabin [6] and the Catmull/Clark scheme [3] as well but it cannot be generalized to non-polynomial schemes. To rate the performance, we compare our scheme to Pulli/Segal since this can be considered as the current standard solution.

2 LOOP SUBDIVISION

Loop's subdivision scheme [11] generalizes the uniform refinement of quartic Box-spline surfaces to control meshes of arbitrary topology. It is one of the most thoroughly analyzed subdivision schemes. In this section we summarize some of its relevant properties.

2.1 Definitions

Let M_0 be an open or closed triangle mesh of arbitrary topology. The number of edges incident to a vertex is called the *valence* of this vertex. A vertex is called *regular* if it is an inner vertex of valence 6 or a boundary vertex of valence 4, otherwise it is called *irregular*. A triangle having 0, ≤ 1, or ≤ 3 irregular vertices is called *regular*, *semi-regular*, or *irregular* respectively.

2.2 Subdivision operator

A *(Loop) subdivision step* maps a triangle mesh M_i to a refined triangle mesh M_{i+1} by applying the following two operations:

- *Splitting*: Each triangle is split in four by inserting new vertices on the midpoints of the edges and connecting them pairwise (see Figure 1).
- *Averaging*: Each vertex is relocated by replacing its position with a weighted average of its neighbors. The corresponding masks are shown in Figure 2.

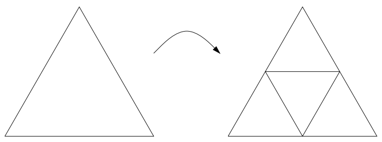

Figure 1: *Splitting: A triangle is split in four.*

The sequence $M_0, M_1, \ldots$ generated by iteratively applying this operator converges to a smooth limit surface M_∞.

Note that the boundary masks $B2$ and $B3$ are chosen such that the resulting boundary curve converges to a cubic B-spline and does not depend on the inner vertices of the mesh. This assures that control meshes with a common boundary polygon can be patched together without leaving gaps. Other choices for the boundary rules are possible [2] but our's is optimized for the evaluation procedure.

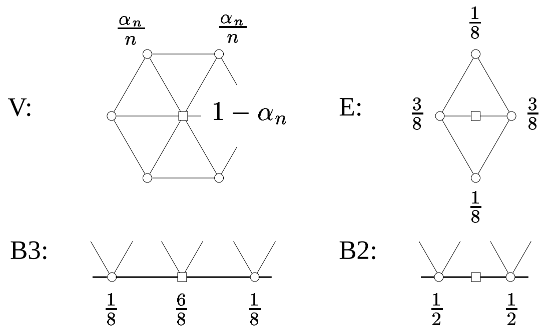

Figure 2: *Averaging: Masks for the interior vertices (upper row) and for the boundary vertices (lower row). Vertices from the coarser level (even vertices) are marked by circles, new vertices on the finer level (odd vertices) are marked by squares. Only the mask V does depend on the valence n of the center vertex:* $\alpha_n = \frac{1}{64}\left(40 - (3 + 2\cos(2\pi/n))^2\right)$.

Another approach to controlling the boundary of a Loop surface works as follows (see e.g. [13]): Instead of defining additional smoothing rules for the boundary vertices, we can simply discard the boundary vertices together with their incident triangles after each subdivision step. The meshes M_i will still converge to a well-defined smooth limit but the resulting surface will have shrunk along the boundary (see Figure 3). In order to avoid this effect an additional strip of triangles can be added along the boundary before the first subdivision step is executed (*shrink-to-fit*).

In this setup it is easy to recover the behaviour of our boundary masks $B2$ and $B3$ at the regular parts of the boundary: One just has to add a boundary strip by using the parallelogram rule (see Figure 4). This fact will be very useful in Section 4 since it preserves the polynomial patch structure of the limit surface at the boundaries and avoids the handling of special cases.

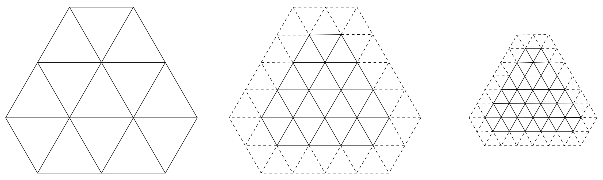

Figure 3: *Discarding the boundary triangles after each subdivision step results in a "shrinking" effect.*

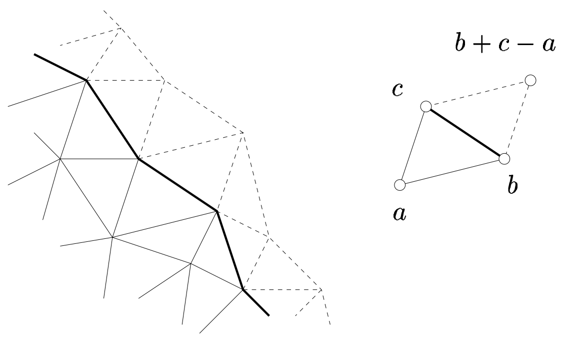

Figure 4: *Left: The original boundary is extended by a strip of triangles (left). The new points are computed by the parallelogram rule (right).*

2.3 Patch structure

Each triangle Δ of a mesh M_i converges to a triangular surface patch $\Phi_i(\Delta) \subset M_\infty$ which we call a *Loop patch*. The notion of (semi-,ir-)regularity is carried over from triangles to Loop patches. Due to the finite size of the subdivision masks, a Loop patch is defined by a finite part of M_i which we call the *control mesh* of the patch (see Figure 5). It consists of Δ itself and the directly adjacent triangles. For Loop subdivision it is known that the patch $\Phi_i(\Delta)$ corresponding to a *regular* triangle Δ can be expressed as a single polynomial patch of degree four (quartic Box-spline). As each subdivision step introduces only regular vertices, the regular regions of a mesh are constantly growing — yielding more and more polynomial patches. Therefore any Loop patch $\Phi_i(\Delta)$ consists of countably many polynomial patches which are arranged as depicted in Figure 6.

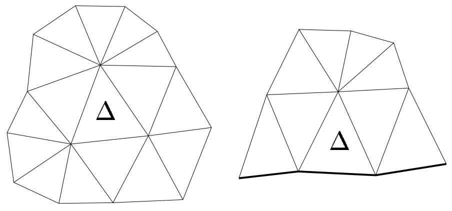

Figure 5: *Control mesh of a Loop patch $\Phi_i(\Delta)$ in the interior (left) and at the boundary (right).*

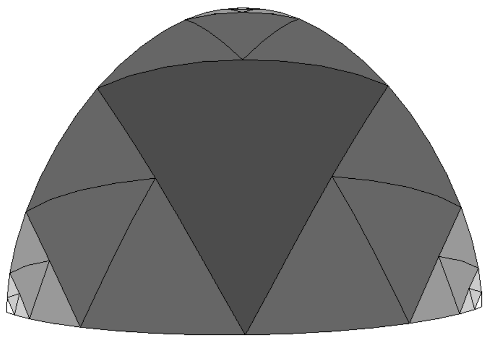

Figure 6: *Piecewise polynomial structure of a Loop patch. For irregular Loop patches every subdivision step adds another layer of three polynomial patches at each irregular corner (depicted by different shades of grey).*

2.4 Chordal triangle mesh

The *naive method* for approximating a Loop patch works like this: Given a target resolution r, subdivide the given mesh r times and project the vertices of the resulting mesh M_r to the limit surface M_∞. We call the final mesh the *chordal triangle mesh* of resolution r. Figure 7 depicts this procedure in the univariate setting. Figure 8 shows the projection masks for Loop subdivision.

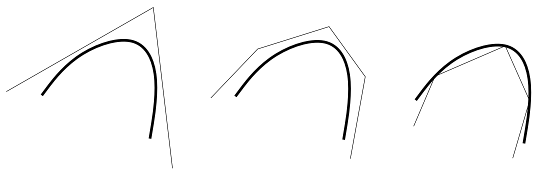

Figure 7: *Chordal triangle mesh: original mesh and limit curve (left), after subdivision (middle), after projection (right).*

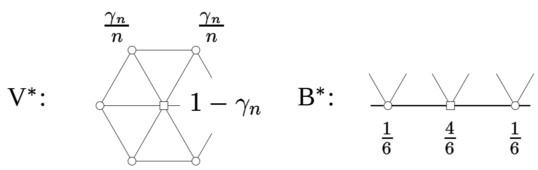

Figure 8: *Projection masks: These masks project the vertices of any control mesh M_i to the limit surface M_∞: $\gamma_n = 8\,\alpha_n/(3+8\,\alpha_n)$.*

3 EVALUATING POLYNOMIALS USING FORWARD DIFFERENCES

Forward differences are well known to be the most efficient technique to evaluate polynomials (cf. e.g. [8]). However, the bivariate case is hardly ever used to our knowledge. As this is the basic element of our algorithm, we recollect some of the relevant details here.

3.1 Univariate polynomials

Let $p(x)$ be a univariate polynomial of degree n. We denote by

$$\begin{aligned} \Delta_i^0 &= p(i) \\ \Delta_i^k &= \Delta_{i+1}^{k-1} - \Delta_i^{k-1} \end{aligned}$$

the *forward differences* of p at the integer parameter values i. Their connections to derivatives reveal that Δ_i^n is constant for all i. Exploiting this fact and rewriting the recurrence formula for the differences, we get the following simple scheme to evaluate the polynomial at the integers $p(i) = \Delta_i^0$, $i = 0, 1, \ldots$ from the set of forward differences $\Delta_0^0, \Delta_0^1, \ldots, \Delta_0^n$:

$$\begin{array}{ccc} \Delta_0^n & \ldots & \Delta_0^1 & \Delta_0^0 \\ \hline \Delta_1^n & \ldots & \Delta_1^1 & \Delta_1^0 \\ \Delta_2^n & \ldots & \Delta_2^1 & \Delta_2^0 \\ \vdots & & \vdots & \vdots \end{array} \qquad \begin{tikzcd} \Delta_i^{k+1} \arrow[dr] & \Delta_i^k \arrow[d] \\ & \Delta_{i+1}^k = \Delta_i^k + \Delta_i^{k+1} \end{tikzcd}$$

Except for the initial computation of $\Delta_0^0, \Delta_0^1, \ldots, \Delta_0^n$ this scheme only requires n additions per point. The scheme can be parallelized in a hardware implementation if we have n synchronous addition units available (see Figure 9). Since all n additions can be executed in parallel, every new sample point is computed in one clock cycle. This means that although the total computation costs are n additions, the *critical costs* (according to the depth of the data dependence graph) are just one operation per sample.

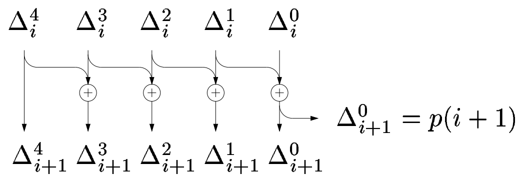

Figure 9: *Parallelizing the forward difference scheme in the case of quartic polynomials ($n = 4$).*

Note that $\Delta_0^0, \Delta_0^1, \ldots, \Delta_0^n$ are linear combinations of the coefficients of p and can therefore be computed from any other basis representation by a matrix multiplication. Consider for example the quadratic polynomial $p(x) = a + b\,x + c\,x^2$. If we want to evaluate p at the scaled integers $i\,h$ for a given step width h, we have to compute the forward differences over the points $p(0)$, $p(h)$, and $p(2\,h)$. The resulting matrix that maps the monomial coefficients to the differences is given by:

$$\begin{pmatrix} \Delta_0^0 \\ \Delta_0^1 \\ \Delta_0^2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & h & h^2 \\ 0 & 0 & 2\,h^2 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \end{pmatrix}.$$

3.2 Bivariate polynomials

The forward difference technique also applies in the bivariate setting. Let $p(x, y)$ be a bivariate polynomial of total degree n. We denote by

$$\begin{aligned} \Delta_{i,j}^{0,0} &= p(i,j) \\ \Delta_{i,j}^{k,l} &= \Delta_{i+1,j}^{k-1,l} - \Delta_{i,j}^{k-1,l} = \Delta_{i,j+1}^{k,l-1} - \Delta_{i,j}^{k,l-1} \end{aligned}$$

the *mixed forward differences* of p at the parameter values (i, j) with integer i, j. Similar to the univariate setting it follows that

$$\Delta_{i,j}^{k,l} = const. \qquad k + l = n$$

where the constant depends only on k and l. To compute the regular sample points $p(i, j)$, $i, j = 0, 1, \ldots$ we arrange the mixed forward differences in a triangular scheme

$$\begin{array}{ccccc}
\Delta_{i,0}^{n,0} & \Delta_{i,0}^{n-1,0} & \dots & \Delta_{i,0}^{1,0} & \Delta_{i,0}^{0,0} \\
 & \Delta_{i,0}^{n-1,1} & \dots & \Delta_{i,0}^{1,1} & \Delta_{i,0}^{0,1} \\
 & & \ddots & \vdots & \vdots \\
 & & & \Delta_{i,0}^{1,n-1} & \Delta_{i,0}^{0,n-1} \\
 & & & & \Delta_{i,0}^{0,n}
\end{array} \qquad (*)$$

and iterate the following two steps in a nested loop

- (*Inner loop*) Compute the points $p(i,j)$ for a fixed i and $j = 0, 1, \dots$ by applying the univariate scheme to a copy of the rightmost column of $(*)$. This is possible since the rightmost column of $(*)$ coincides with a univariate forward differencing scheme that can be used to compute the points in the ith column (cf. Figure 10).

- (*Outer loop*) Apply one step of the univariate scheme to each row of $(*)$. This shifts the whole configuration to the right $(i \to i+1)$ such that the next column can be computed in the inner loop. Notice that the kth row of $(*)$ corresponds to a degree $n-k$ polynomial.

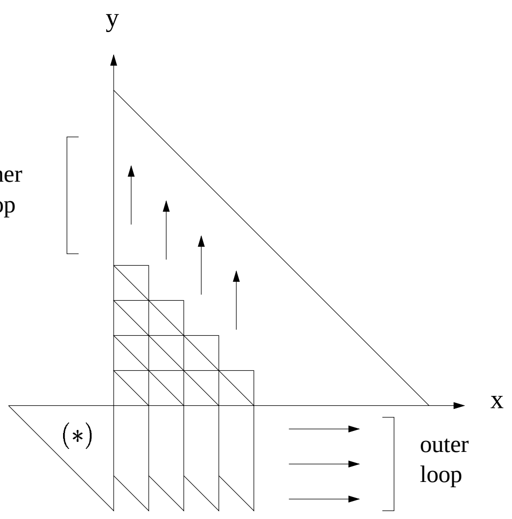

Figure 10: *Rendering a triangular polynomial patch using forward differences. The triangle strips are rendered upwards while the triangular scheme $(*)$ is "moved" from left to right.*

For the discussion of the numerical error in Section 5.3 the following observation will be helpful: For the evaluation of p at the parameter value (i,j) the forward differencing algorithm first performs i steps of the outer loop and then j steps of the inner loop. Each such step is called an *extrapolation step* and is the source of possible roundoff errors. (Counting a step of the outer loop only once is justified by the fact that the rows of $(*)$ are computed independently from each other.) For evaluating $p(i,j)$ at parameter values $(i,j), i+j \le K$ we therefore need at most K extrapolation steps per computed value.

3.3 Implementation

In this section we describe a routine which renders the chordal triangle mesh of a triangular polynomial patch of degree four using the aforementioned scheme. The input consists of the appropriate forward differences and the target resolution r. The mesh is then rendered in form of "vertical" triangle strips (see Figure 10). We use two sets of register variables `l0...l3` and `r0...r3` to simultaneously compute the points $p(i,j)$ and $p(i+1,j)$, $j = 0, 1, \dots$ The triangle scheme $(*)$ is stored in the variables $\Delta^{4,0} \dots \Delta^{0,4}$.

```
render_forward_differences( $r$,
      $\Delta^{4,0}$, $\Delta^{3,0}$, $\Delta^{2,0}$, $\Delta^{1,0}$, $\Delta^{0,0}$,
             $\Delta^{3,1}$, $\Delta^{2,1}$, $\Delta^{1,1}$, $\Delta^{0,1}$,
                    $\Delta^{2,2}$, $\Delta^{1,2}$, $\Delta^{0,2}$,
                           $\Delta^{1,3}$, $\Delta^{0,3}$,
                                  $\Delta^{0,4}$)
for (i = 0; i < $2^r$; i++)
  l0 = $\Delta^{0,0}$; l1 = $\Delta^{0,1}$; l2 = $\Delta^{0,2}$; l3 = $\Delta^{0,3}$;  ]

  $\Delta^{0,0}$+=$\Delta^{1,0}$;$\Delta^{1,0}$+=$\Delta^{2,0}$;$\Delta^{2,0}$+=$\Delta^{3,0}$;$\Delta^{3,0}$+=$\Delta^{4,0}$;  ]
  $\Delta^{0,1}$+=$\Delta^{1,1}$;$\Delta^{1,1}$+=$\Delta^{2,1}$;$\Delta^{2,1}$+=$\Delta^{3,1}$;            ]
  $\Delta^{0,2}$+=$\Delta^{1,2}$;$\Delta^{1,2}$+=$\Delta^{2,2}$;                        ]
  $\Delta^{0,3}$+=$\Delta^{1,3}$;                                    ]
  r0 = $\Delta^{0,0}$; r1 = $\Delta^{0,1}$; r2 = $\Delta^{0,2}$; r3 = $\Delta^{0,3}$;  ]

  glBegin(GL_TRIANGLE_STRIP);
  for (j = 0; j < $2^r$ - i; j++)
    glVertex3fv(l0); glVertex3fv(r0);
    l0 += l1; l1 += l2; l2 += l3; l3 += $\Delta^{0,4}$;  ]
    r0 += r1; r1 += r2; r2 += r3; r3 += $\Delta^{0,4}$;  ]
  glVertex3fv(l0);
  glEnd();
```

Note that this iterative procedure only needs 23 register variables and is therefore well suited for hardware implementation. Note also that the operations comprised by the brackets are independent and could be executed in parallel for maximum performance.

The above procedure obviously computes every vertex twice: once as `r0` and then as `l0` in the next column. We can reduce the number of operations by the factor 2 if those intermediate values are cached. However this would cause additional storage overhead.

Note that the target resolution r can be determined automatically — the farther away the patch from the observer the lower the its resolution. However, to avoid gaps between adjacent patches all patches of an object should be rendered at the same target resolution.

Instead of creating triangle strips one can also use adaptive forward differencing techniques to draw the patches pixel-wise (cf. [10]).

4 EVALUATING LOOP SURFACES

In this section we describe the basic idea of our algorithm. The implementation details and data structures are described in the appendix.

The algorithm consists of several routines that can be arranged in the following hierarchy:

```
4  render_patch
3  render_semi_patch
2  render_reg_patch
1  render_forward_differences,
   render_one_triangle,
   $S$, $P_m$, $P_0$, $P_1$, $P_2$, $Q_r$
```

The ordering is such that the higher procedures call the lower ones. From top to bottom the number of special cases that have to be handled within a procedure decreases while the number of arithmetic operations increases. For a specific rendering system, we can choose the extent to which the routines should be implemented in hardware by first considering level 1 and then going up the hierarchy. Higher levels could be implemented in software without significant loss of efficiency. However, the routine `render_forward_differences` should definitely be implemented in hardware since the main work is done there.

The highest level API consists only of the routine `render_patch`, which provides the new graphics primitive. Given the control mesh of an arbitrary Loop patch it renders the corresponding chordal triangle mesh with prescribed resolution.

The basic idea of our algorithm is as follows: The input control mesh is subdivided by `render_patch`. According to the patch structure described in Section 2 this gives rise to one regular and three semi-regular patches. Those are handed over to the routines `render_reg_patch` and `render_semi_patch` respectively. The routine `render_reg_patch` does some preprocessing and then renders the patch using `render_forward_differences`. The routine `render_semi_patch` is more complex. First, the control mesh is subdivided once more. As the original patch was semi-regular this results in one semi-regular and three regular patches. The regular patches are rendered as above, while the semi-regular mesh is further subdivided. This process is repeated with decreasing resolution until the chordal triangle mesh consists of merely one triangle, which then is rendered via `render_one_triangle`.

In the following we explain the details for the different levels:

4.1 Level 1

The routines S, P_m, P_0, P_1, P_2, and Q_r implement the following linear operators:

- The *subdivision operator* S maps the control mesh M of a Loop patch to the subdivided mesh M' (see Figure 11).
- The *selection operators* P_m, P_0, P_1, P_2 select parts of the mesh M' and orient it such that the irregular vertex is on top (cf. Figure 12 and 14).
- The *forward difference operator* Q_r maps the control mesh of a regular inner Loop patch to the forward differences of the corresponding polynomial. This operator depends on the target step width $h = 2^{-r}$. The matrix Q_r is given explicitly in the Appendix.

Note that these operators could also be described as matrices but the direct computation might be more efficient for hardware implementation. The actual procedural formulations are given in the Appendix.

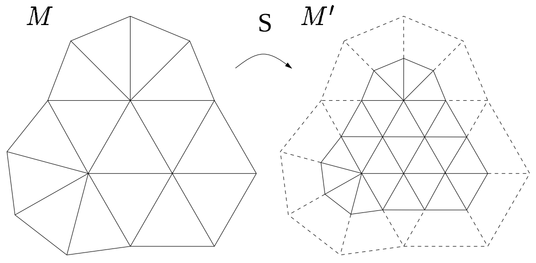

Figure 11: *The action of the subdivision operator S.*

The input of `render_one_triangle` consists of the control mesh of an arbitrary Loop patch. The procedure projects the three corner vertices to the limit surface by using the rules of Figure 8 and renders the triangle. The procedure `render_forward_differences` was explained in Section 3.

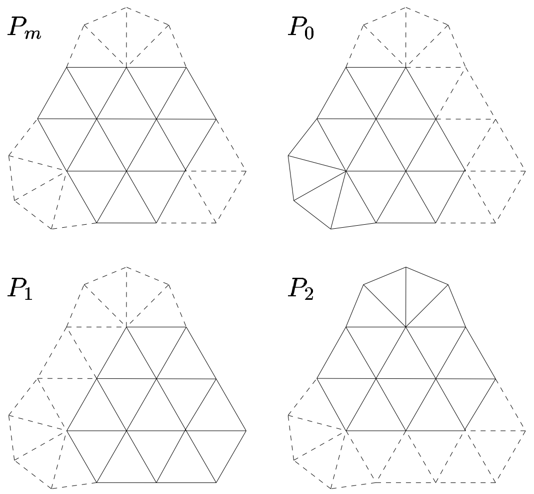

Figure 12: *The selection operators P_m, P_0, P_1, P_2. Note that the resulting mesh has to be oriented such that the remaining irregular vertex is on top (not shown here).*

4.2 Level 2

The routine `render_regular_patch` renders the chordal triangle mesh of a regular Loop patch. Figure 13 shows typical input meshes. As the patch may be a boundary patch, not all of the vertices (like the ones marked by circles) might be present. However we can construct the missing vertices by using the parallelogram rule as described in Section 2. Since this is equivalent to the application of the special boundary rules $B2$ and $B3$, we can forgo a special treatment of the boundary!

```
render_regular_patch($r$, $M$)
  construct potentially missing vertices by parallelogram rule
  $f$ = $Q_r(M)$;
  render_forward_differences($r$, $f$);
```

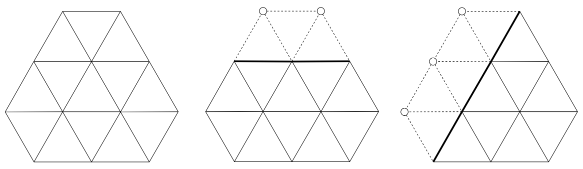

Figure 13: *Control mesh of a regular patch.*

4.3 Level 3

The routine `render_semi_patch` renders the chordal triangle mesh of a semi-regular Loop patch at resolution r. The input mesh is supposed to be oriented such that the irregular vertex is on top (see Figure 14).

```
render_semi_regular_patch($r$, $M$)
if ($r$ == 0)
  render_one_triangle($M$)
if ( top vertex is regular )
  render_regular_patch($r$, $M$)
```

```
else
  $M'$ = $S(M)$
  for (i = 0; i < $r$; i++)
    render_regular_patch($r$-i, $P_m(M')$)
    render_regular_patch($r$-i, $P_0(M')$)
    render_regular_patch($r$-i, $P_1(M')$)
    $M''$ = $P_2(M')$
    $M'$ = $S(M'')$
  render_one_triangle($M''$)
```

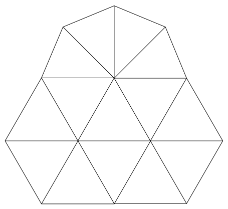

Figure 14: *Orientation of semi-regular control meshes: The irregular vertex is on top.*

4.4 Level 4

The routine render_patch covers the most general case, it renders the chordal triangle mesh of an arbitrary Loop patch at resolution r.

```
render_patch($r$, $M$)
  if ($r$ == 0)
    render_one_triangle($M$)
  else if ( all vertices are regular )
    render_regular_patch($r$, $M$)
  else
    $M'$ = $S(M)$
    render_regular_patch($r$-1, $P_m(M')$)
    render_semi_patch($r$-1, $P_0(M')$)
    render_semi_patch($r$-1, $P_1(M')$)
    render_semi_patch($r$-1, $P_2(M')$)
```

5 DISCUSSION

5.1 Memory requirements

The size of the data structure CMesh used for storing Loop control meshes is approximately three times the maximum valence of the vertices (see the Appendix). By carefully programming we only need three of those structures to run the evaluation procedure — one on each of the levels 2, 3, and 4. This makes the overall memory consumption constant and independent from the refinement depth which means that there is virtually no upper bound for the maximum refinement. Even more important is the fact that the central procedure render_forward_differences (level 1) uses only 23 register variables to store the necessary forward differences.

In contrast, the memory requirements for all the depth-first approaches is $O(2^r)$ [9, 14] if subdivision down to the rth level is computed. In addition our algorithm does not need to maintain lists of control vertices from various levels of refinement.

5.2 Time requirements

We define a basic *operation* to be a vector-vector-addition or a scalar-vector-multiplication. For large resolutions r the time for initializing the forward differences becomes neglegible compared to the work done in the inner loop of render_forward_differences. In that procedure one needs 8 vector additions to render 2 triangles. Hence, for large r the ratio operations/rendered triangle approaches 4 (without caching!). In a hardware implementation all these four operations could be executed in parallel (cf. Section 3).

Let us now compute the number of operations for the naive implementation: Let m be the number of vertices of the initial mesh. This implies that we have approximately $4^i m$ vertices on level i. To compute the vertices on the next level $i+1$ we have to “lift” the $4^i m$ old vertices using the mask V, which takes approximately 8 operations per vertex and compute the $3 \cdot 4^i m$ new vertices using mask E, which takes 5 operations per vertex. In total we get

$$\sum_{i=0}^{r-1} 4^i m(8+5\cdot 3)/4^r m \approx 8$$

operations per vertex on the rth refinement level which means 4 operations/triangle. If we want to project the resulting vertices eventually to the limit surface, we need another 8 operations per vertex (4 operations per triangle) adding up to 8 operations per triangle (however, the last step is omitted in many implementations).

This complexity estimate holds for the algorithms of Pulli/Segal-type as well since they merely change the computation order from breadth-first to depth-first. We conclude that both algorithm take $O(n)$ time where n is the number of rendered triangles.

Nevertheless, we believe that our algorithm will surely outperform the other ones for a variety of reasons:

- Our algorithm can be parallelized maximally, i.e. the inner loop could be implemented in such a way that it needs only one clock-cycle per triangle (cf. four synchronous adders in Figure 9).
- We do not need to calculate additional entities, like indices or addresses of memory locations.
- No memory management is necessary since we just have a fixed number of register variables.
- We do not need to do any preprocessing (patch clustering) of the data.

5.3 Precision requirements

One potential weak point of our algorithm could be numerical precision — forward differences tend to become numerically instable for extremely small step widths. The intrinsic difficulty with all difference schemes is numerical cancellation when two large values are subtracted yielding a result with small absolute value. However, in our implementation we used single precision float variables and did not experience severe problems up to reasonably large refinement depths r. Bartels [1] analyzed the error of univariate cubic forward differencing and concludes that one “is unlikely to have problems” when the number of forward differencing steps is $\leq 2^8$ (using floating point arithmetic). Now, as far as the error analysis is concerned, the bivariate extrapolation steps correspond exactly to the univariate forward differencing steps, therefore the error propagation of bivariate forward differencing is not worse than that of univariate forward differencing. Following the lines of thought in Section 3.2 we see that we will never perform more than 2^r extrapolation steps! Thus for $r \leq 8$ we are unlikely to run into problems.

Noticeable roundoff errors can occur if the lengths of the edges in the control mesh are several orders of magnitude smaller than their distance to the origin. A simple technique to increase numerical stability in such situations is to shift the center of gravity for each Loop patch to the origin, run the evaluation procedure and shift the surface samples back to their original position. This would cause a computational overhead of one operation per triangle (without caching).

Using double precision float variables reduces numerical errors by several orders of magnitude but also doubles the memory requirements. We used floating point arithmetic as this is less sensitive to errors, however an implementation based on integers is also possible (see e.g. [4]).

6 EXPERIMENTAL RESULTS

Experimental results are given for three control meshes (cf. Figure 15): A tetrahedron, which in some sense is a worst case example as all vertices are irregular, a torus, which in the same sense is optimal because all vertices are regular, and a reduced version of the well-known Stanford bunny (130 triangles). In order to avoid computational overhead for low refinement levels we implemented a special subroutine that renders the chordal mesh of regular patches at subdivision levels 1 and 2 by using explicit masks for the vertex positions.

Figure 16 shows the number of operations per triangle which asymptotically approaches four as we expected. While each of the patches of the torus can immediately be rendered by forward differencing the patches of the tetrahedron and the bunny contain irregular vertices and therefore show a patch structure like in Figure 6. This is the reason for the peak at subdivision levels 1 and 2. For refinement levels 6 and higher our software implementation renders about 500K triangles per second on an SGI O2.

Figure 17 shows the maximum relative deviation caused by roundoff. For this we evaluated the corresponding polynomials at the parameter values $(0,1)$, $(\frac{1}{2},\frac{1}{2})$ and $(1,0)$ once by using explicit masks and once by using forward differences. To obtain the relative error we divided the norm of the difference by the norm of the explicit solution and took the maximum.

Finally Figure 18 shows the reflection lines of the bunny at different subdivision levels. For high quality renderings of reflecting objects (e.g. a car body) these lines are very important. Because they are very sensitive to discontinuities of the surface (their continuity is in general one less than that of the surface, see [16]) they are often used to measure the quality of a surface.

The left bunny was rendered at subdivision level three and the vertex normals were not interpolated across the triangles. This is therefore a C^0 surface which can clearly be seen as the reflection lines are not even continuous. The middle bunny was also rendered at subdivision level three but this time the vertex normals were interpolated across the triangles (Phong shading). The resulting reflection lines are continuous but not smooth. The right bunny was rendered at subdivision level seven and the vertex normals were not interpolated. The smooth reflection lines indicate an approximate C^2-surface. We conclude that we can get rid of linear interpolation and its artifacts by using high subdivision levels.

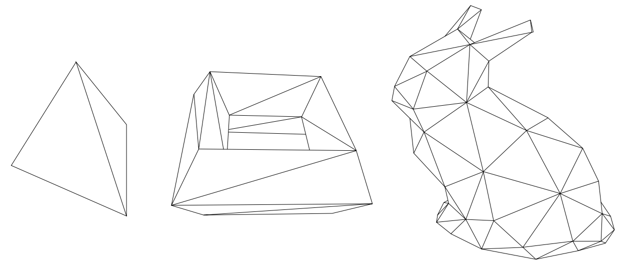

Figure 15: *Example meshes: tetrahedron, torus, bunny.*

7 CONCLUSION AND FUTURE WORK

We presented a new algorithm for evaluating and rendering Loop subdivision surfaces. Its main advantages are

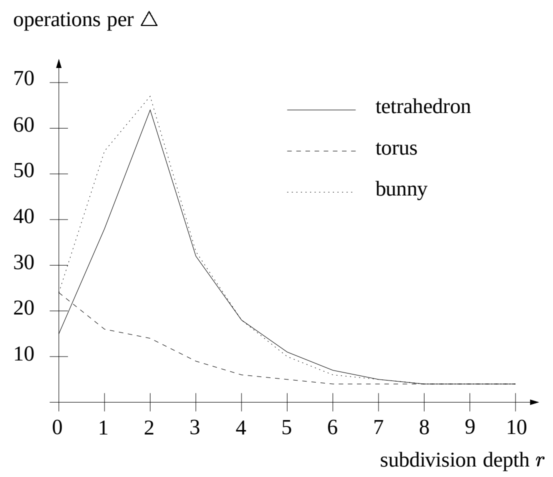

Figure 16: *Operation count per triangle for a given subdivision level.*

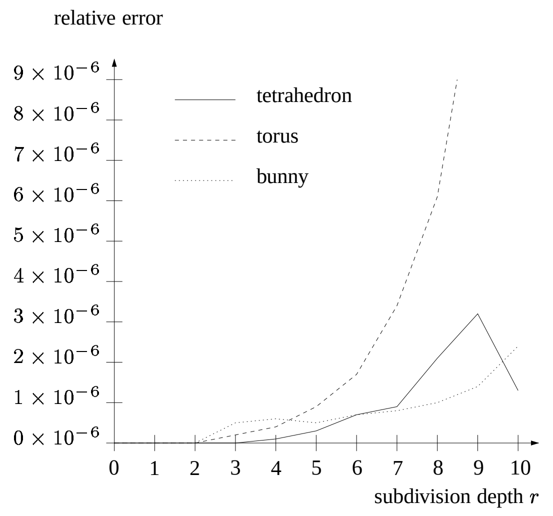

Figure 17: *Maximal relative error for a given subdivision level.*

- *Speed*: We only need four operations for each triangle. Additional speed-up can be obtained by executing these operations in parallel or by caching intermediate results.
- *Constant memory requirements*: The central procedure uses only (and exactly) 23 register variables.
- *Simplicity*: Both in the algorithmic structure and in the programming interface.

We achieve this by specializing to Loop subdivision. The scheme could be reformulated for the Doo/Sabin [6] and the Catmull/Clark [3] scheme since those schemes also generate piecewise polynomial limit surfaces. However, there is no way to generalize it to non-polynomial subdivision schemes.

In the future we are planning to also include the computation of the exact surface normals. Forward differences provide all the local information to compute partial derivatives. This extension would increase the number of register variables in the procedure `render_forward_differences` to 29. Another feature which could be included is to combine the evaluation algorithm with an adaptive refinement strategy.

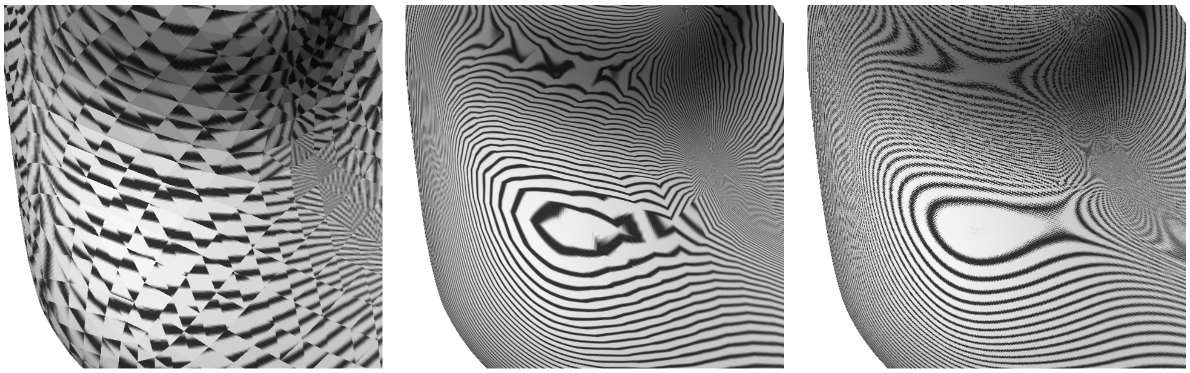
Figure 18: *Reflections lines for subdivision levels $r = 3$, without interpolation of vertex normals (left) $r = 3$, with interpolating vertex normals (middle), and $r = 7$, without interpolation of vertex normals (right).*

References

[1] R. H. Bartels, J. C. Beatty, and B. A. Barsky. *An Introduction to Splines for Use in Computer Graphics and Modeling*. Morgan Kaufmann, 1987.

[2] H. Biermann, A. Levin, and D. Zorin. Piecewise smooth subdivision surfaces with normal control. In *SIGGRAPH 2000 Conference Proceedings*.

[3] E. Catmull and J. Clark. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer-Aided Design*, 10:350–355, September 1978.

[4] Sheue-Ling Chang, Michael Shantz, and Robert Rocchetti. Rendering cubic curves and surfaces with integer adaptive forward differencing. In *Computer Graphics (SIGGRAPH 89 Conference Proceedings)*, pages 157–166. ACM Press, July 1989.

[5] Tony DeRose, Michael Kass, and Tien Truong. Subdivision surfaces in character animation. In *SIGGRAPH 98 Proceedings*, Annual Conference Series, pages 85–94, August 1998.

[6] D. Doo and M. Sabin. Behaviour of recursive division surfaces near extraordinary points. *Computer-Aided Design*, 10:356–360, September 1978.

[7] Denis Zorin et. al. Subdivision for modeling and animation. In *SIGGRAPH 99 Course Notes*, 1999.

[8] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics — Principles and Practice (2nd Ed.)*. Addison-Wesley, 1990.

[9] Markus Kohler and Heinrich Müller. Efficient calculation of subdivision surfaces for visualization. Technical Report 585, University of Dortmund, 1995.

[10] Sheue-Ling Lien, Michael Shantz, and Vaughan Pratt. Adaptive forward differencing for rendering curves and surfaces. In *Computer Graphics (SIGGRAPH 87 Conference Proceedings)*, pages 111–118. ACM Press, July 1987.

[11] Charles T. Loop. Smooth subdivision surfaces based on triangles. Master's thesis, University of Utah, Department of Mathematics, 1987.

[12] Kerstin Müller and Sven Havemann. Subdivision surface tesselation on the fly using a versatile mesh data strucure. Computer Graphics Forum, Eurographics 2000 issue.

[13] Ahmad H. Nasri. Polyhedral subdivision methods for free-form surfaces. *ACM Transactions on Graphics*, 6(1):29–73, January 1987.

[14] Kari Pulli and Mark Segal. Fast rendering of subdivision surfaces. In *Eurographics Rendering Workshop 1996*, pages 61–70, 1996.

[15] Ulrich Reif. A unified approach to subdivision algorithms near extraordinary vertices. *Computer Aided Geometric Design*, 12(2):153–174, 1995.

[16] Michael Spivak. *A comprehensive introduction to differential geometry*. Publish or Perish, Inc., 1979.

[17] Jos Stam. Exact evaluation of Catmull-Clark subdivision surfaces at arbitrary parameter values. *SIGGRAPH 98 Conference Proceedings*, pages 395–404, August 1998.

[18] Denis Zorin. *C^k Continuity of Subdivision Surfaces*. PhD thesis, California Institute of Technology, Department of Computer Sciences, 1996.

A IMPLEMENTATION

A.1 Data structures

We need to store control meshes for arbitrary configurations of Loop patches like the one in Figure 19 and control meshes that result from a subdivision step of a Loop patch like the one in Figure 21. Our data structure stores for each of the three corners 0,1,2 of such a mesh the center vertex, the surrounding vertices, the valence and the index of the boundary edges (if there are any):

```
struct { Vec3f c[3][MaxVal];
         int   n[3];
         int   h[3]; } CMesh;
```

The array `c[][]` holds the coordinate values for the center vertex and the surrounding vertices (numerated counterclockwise) as shown in Figure 19 and 21. Vertices that appear in more than one neighborhood are stored multiple times. This is intentional as it simplifies the algorithms. In particular we don't need to deal with special cases like vertices of valence 3.

The array `n[]` contains the vertex valences and the array `h[]` contains the indices of the boundary edges. Note that since we are dealing with manifolds, no more than two boundary edges can emanate from a single vertex. The boundary edges are coded by the following scheme: If `h[i]==-1` there are no boundary edges incident to center vertex `i`. Otherwise the edges (`c[i][h[i]-1]`, `c[i][n[i]]`) and (`c[i][n[i]]`, `c[i][h[i]]`) are boundary edges. In the example of Figure 19 and 21 we have `n[0] == 6`, `n[1] == 8`, `n[2] == 7` and `h[0] == 1`, `h[1] == 3`, `h[2] == 6`.

We now give the procedural formulation of the matrix operators:

A.2 The operator S

Because of the cyclical arrangement of the coordinates in our data structure the operator S can be implemented compactly:

```
L(N, C, S)    = (1-αN) * C + αN * S / N
E(A, B, C, D) = (3 * A + 3 * B + C + D) / 8
B2(A, B)      = (A + B) / 2
B3(A, B, C)   = (A + 6 * B + C) / 8

S(CMesh d, CMesh s)
  for (i = 0; i < 3; i++)
    n = d.n[i] = s.n[i];
    h = d.h[i] = s.h[i];
    if (h == -1)
      sum = s.c[i][0] + ... + s.c[i][n-1]
      d.c[i][n] = L(n, s.c[i][n], sum)
```

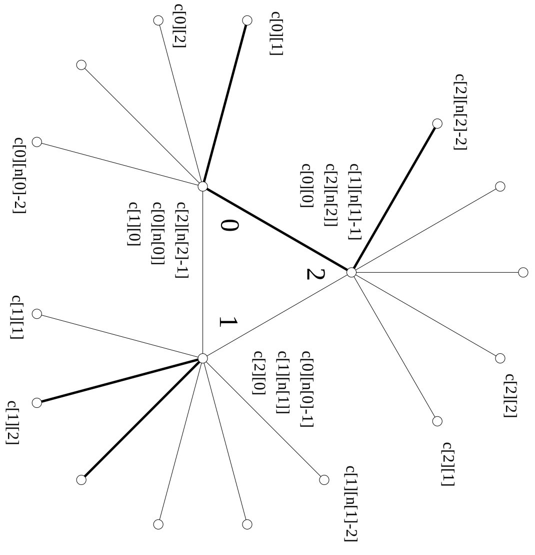

Figure 19: *Control mesh used as input for S, Q_r and as output of P_m, P_0, P_1, P_2.*

```
    else
      d.c[i][n] = B3(s.c[i][h-1], s.c[i][n],
                     s.c[i][h]);
    for (j = 0; j < n; j++)
      if (j == h || j == h-1)
        d.c[i][j] = B2(s.c[i][n],
                       s.c[i][j]);
      else
        d.c[i][j] = E(s.c[i][n], s.c[i][j],
                      s.c[i][(j+1) % n],
                      s.c[i][(j-1) % n]);
```

Note that the definitions of L, E, B3 and B2 correspond to the Loop masks in Figure 2.

A.3 The operators P_m, P_0, P_1, P_2

The operators P_m, P_0, P_1 and P_2 merely select some vertices of a CMesh and permute them such that the irregular vertex and its neighbors are stored in c[2][]. This can be done without any calculations and is not carried out here.

A.4 The operator Q_r

The operator Q_r maps a regular Loop control mesh to the corresponding mixed forward differences. This can be written as a matrix-vector-multiplication (see Figure 20). We use the following abbreviations:

$$a = 2^r, \quad b = 4^r, \quad c = 8^r, \quad d = 16^r$$

Note that the entries of the matrix Q_r depend on the refinement level r since the original forward difference scheme computes sample values at the integer parameter values $(i, j) \in \mathrm{Z}^2$. In order to evaluate at $(i, j) \in 2^{-r}\,\mathrm{Z}^2$ we have to reparameterize the surface patch.

As the tenth row of Q_r consists only of zero entries, the mixed forward difference $\Delta^{2,2}_{0,0}$ is constant zero, so we can save a register variable here.

$$
\begin{bmatrix} \Delta^{4,0} \\ \Delta^{3,0} \\ \Delta^{2,0} \\ \Delta^{1,0} \\ \Delta^{0,0} \\ \Delta^{3,1} \\ \Delta^{2,1} \\ \Delta^{1,1} \\ \Delta^{0,1} \\ \Delta^{2,2} \\ \Delta^{1,2} \\ \Delta^{0,2} \\ \Delta^{1,3} \\ \Delta^{0,3} \\ \Delta^{0,4} \end{bmatrix}
= \frac{1}{12d}
\begin{bmatrix}
-24 & -24 & 48 & -24 & 24 & -24 & 48 & -24 & 24 & -24 & 0 & 0 \\
48a-36 & -36-24a & 72-24a & 12a-36 & 36-24a & 12a-36 & 72-24a & 12a-36 & 36 & 12a-36 & 0 & 0 \\
48a-24b-14 & 12b-14-24a & 28-24a & 12a-14 & 14+12b-24a & 12a-14 & 28-24a & 12a-14 & 14 & 12a-14 & 0 & 0 \\
8a-12b-1 & 6b+4c-4a-1 & 2c+2-4a & 2a-2c-1 & 6b-4c-4a+1 & 2a-2c-1 & 2c+2-4a & 2a-1 & 1 & 2a-1 & 0 & 0 \\
6d & d & d & d & d & d & d & 0 & 0 & 0 & 0 & 0 \\
-12 & -12 & 24 & -12 & 12 & -12 & 24 & -12 & 12 & -12 & 0 & 0 \\
24a-12 & -12a-12 & 24-24a & 12a-12 & 12-12a & -12 & 24 & -12 & 12 & 12a-12 & 0 & 0 \\
24a-12b-4 & 2+6b-18a & 2+6b-18a & 2-6b+6a & 6b-6a & 6b-6a & 2-6b+6a & -2 & 2 & 12a-4 & 2 & -2 \\
8a-12b-1 & 2c+2-4a & 6b+4c-4a-1 & 2c+2-4a & 2a-2c-1 & 6b-4c-4a+1 & 2a-2c-1 & 0 & 0 & 2a-1 & 1 & 2a-1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
24a-12 & 24a-24a & -12a-12 & 24 & -12 & 12-12a & 12a-12 & 0 & 0 & 12a-12 & 12 & -12 \\
48a-24b-14 & 28-24a & 12b-24a-14 & 28-24a & 12a-14 & 14+12b-24a & 12a-14 & 0 & 0 & 12a-14 & 14 & 12a-14 \\
-12 & 24 & -12 & 24 & -12 & 12 & -12 & 0 & 0 & -12 & 12 & -12 \\
48a-36 & 72-24a & -36-24a & 72-24a & 12a-36 & 36-24a & 12a-36 & 0 & 0 & 12a-36 & 36 & 12a-36 \\
-24 & 48 & -24 & 48 & -24 & 24 & -24 & 0 & 0 & -24 & 24 & -24
\end{bmatrix}
\begin{bmatrix} \mathtt{c[0][n[0]]} \\ \mathtt{c[1][n[1]]} \\ \mathtt{c[2][n[2]]} \\ \mathtt{c[0][1]} \\ \mathtt{c[0][2]} \\ \mathtt{c[0][3]} \\ \mathtt{c[1][1]} \\ \mathtt{c[1][2]} \\ \mathtt{c[1][3]} \\ \mathtt{c[2][1]} \\ \mathtt{c[2][2]} \\ \mathtt{c[2][3]} \end{bmatrix}
$$

Figure 20: *Matrix Q_r.*

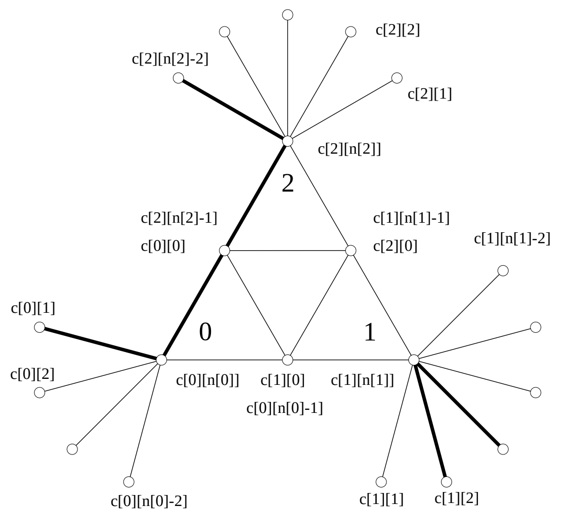

Figure 21: *Control mesh used as output of S and as input for P_m, P_0, P_1, P_2.*

A.5 The routine `render_one_triangle`

The routine `render_one_triangle` gets the control mesh of a Loop patch as in Figure 19 as input, projects the vertices `c[0][n[0]]`, `c[1][n[1]]`, `c[2][n[2]]` to the limit surface and renders the corresponding triangle.

```
render_one_triangle(CMesh m)
  for (i = 0; i < 3; i++)
    if (m.h[i] > -2)
      x[i] = (  m.c[i][m.h[i]] +
               4*m.c[i][m.n[i]] +
                 m.c[i][m.h[i]+1])/6
    else
      x[i] = (1-$\gamma_{\mathtt{m.n[i]}}$)*c[i][m.n[i]] +
               $\gamma_{\mathtt{m.n[i]}}$/m.n[i] * (m.c[i][0]+...+
                     m.c[i][m.n[i]-1])
  render the triangle  (x[0], x[1], x[2])
```