Efficient Storage and Processing of Adaptive Triangular Grids using Sierpinski Curves

Csaba Attila Vigh,
Dr. Michael Bader
Department of Informatics, TU München
JASS 2006, course 2:
Numerical Simulation: From Models to Visualizations

Abstract. In this paper an algorithm is presented to store and process fully adaptive computational grids requiring only a minimal amount of memory. The adaptive grid is specified by a recursive bisection of triangular grid cells. The cells are stored and processed in an order specified by the Sierpinski space-filling curve. A sophisticated system of stacks is used to ensure cache-efficient access to the unknowns.

1. Introduction

One of the most common approaches to modeling and simulation is based on PDEs and their numerical discretization with finite elements or similar methods. In the generation of the respective computational grids, there is often a demand for adaptive refinement. Introducing adaptive refinement leads to a trade-off between memory requirements and computing time. This is due to the need to obtain the neighbor relationships between grid cells both during grid generation and computation. Storing these relations explicitly allows arbitrary unstructured grids, but requires a considerable memory overhead. It could be even more than 1 kilobyte of memory used per unknown.

Now we want to address a situation where memory should be saved as far as possible, which requires the use of a strongly structured grid, and the neighbor relations must be computed instead. In this paper we will present grids resulting from recursive splitting of triangles. To efficiently process such a grid, we present a scheme that combines the use of space-filling curves and a system of stacks. The stack-like access leads to excellent cache-efficiency, while the parallelization strategies based on space-filling curves are readily available.

2. Space filling curves

In 1878, Cantor demonstrated that any two finite-dimensional manifolds, no matter what their dimensions, have the same cardinality. This implies, that the unit interval $[0,1]$ can be mapped bijectively onto the square $[0,1]^2$, or onto the cube $[0,1]^3$. The question arose immediately whether or not such a mapping can be continuous. In 1879 Netto showed that such a mapping is necessarily discontinuous.

Suppose the condition of bijectivity is dropped, is it possible to obtain a continuous surjective mapping? Since any continuous mapping from $[0,1]$ into the plane (or space) is called a curve, the question may be rephrased: Is there a curve that passes through every point of a two-dimensional region with positive Jordan content (area)?

Peano settled this question by constructing in 1890 the first such curve. Further examples by Hilbert, Sierpinski, and others followed.

2.1 Hilbert's Space-Filling Curve

Although Peano discovered the first space-filling curve, Hilbert was who recognized a general geometric generating procedure that allowed the construction of an entire class of space-filling curves: If the interval I ($[0,1]$) can be mapped continuously onto the square Q ($[0,1]^2$), then after partitioning I into four congruent subintervals and Q info four congruent subsquares, each subinterval can be mapped continuously onto one of the subsquares. Next, each subinterval is, in turn, partitioned into four congruent subintervals and each subsquare into four congruent subsquares, and the argument is repeated. If this is carried out infinitely many times, I and Q are partitioned into 2^{2n} congruent replicas. Hilbert has demonstrated that the subsquares can be arranged so that adjacent subintervals correspond to adjacent subsquares with an edge in common. Furthermore, the inclusion relationships are preserved: if a square corresponds to an interval, then its subsquares correspond to the subintervals of that interval (see Figure 1).

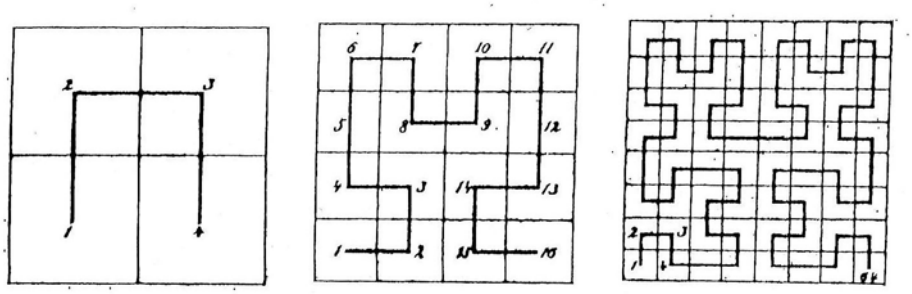

Figure 1: Generating Hilbert's Space-Filling Curve (from Sagan [2])

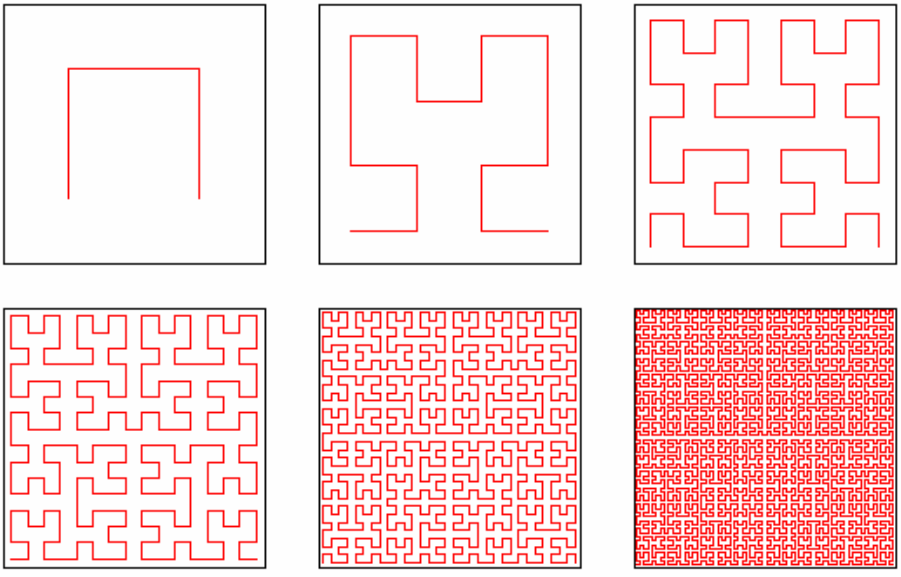

Figure 2: Six iterations of the Hilbert curve (from Wikipedia [4])

Hilbert's mapping is surjective and continuous, meaning it is a space-filling curve, and it is nowhere differentiable (see Sagan [2]). Hilbert's curve can be extended to three dimensions, the unit interval and the unit cube will be divided in 8 congruent subintervals and subcubes, with the proper ordering (see Sagan [2]).

2.2 Peano's Space-Filling Curve

In the generation of Peano's space-filling curve we partition the unit interval I into 9 congruent subintervals, and the unit square Q into 9 congruent subsquares. The subsquares will be arranged in the order indicated by Figure 3. If the partitioning is carried out n times, then we may obtain 3^{2n} subintervals mapped into 3^{2n} subsquares. Figure 4 shows the first three iterations of the Peano curve.

3	4	9
2	5	8
1	6	7

Figure 3: Peano's mapping

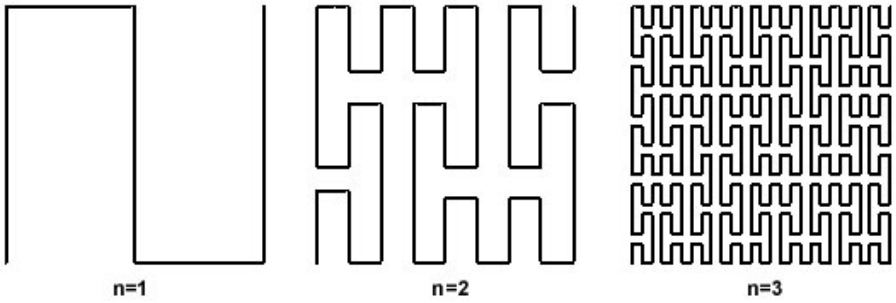

Figure 4: Three iterations of the Peano curve (from Wikipedia [4])

2.3 Sierpinski's Space-Filling Curve

In 1912 Sierpinski introduced another space-filling curve, of which the first four iterations are displayed on Figure 5.

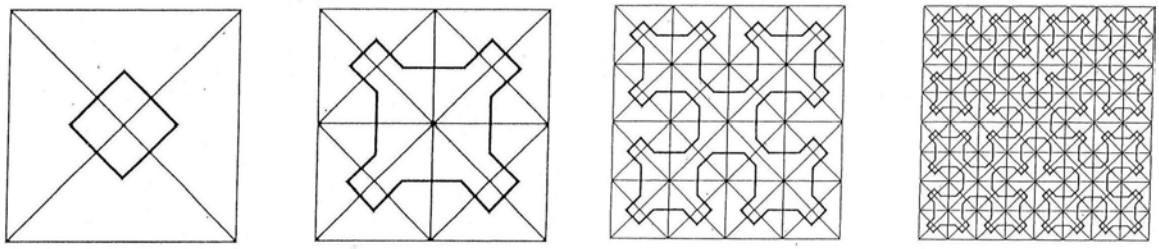

Figure 5: Four iterations of the Sierpinski curve (from Sagan [2])

Half of Sierpinski's curve lies on one, while the other half lies on the other right isosceles triangle that is obtained by slicing the square into half by its diagonal.

Therefore, we will view Sierpinski's curve as a map from the unit interval I onto a right isosceles triangle T with vertices at (0, 0), (2, 0) and (1, 1). Using Hilbert's generating principle, we partition I into two congruent subintervals and T into two congruent subtriangles. After n times we obtain 2^n subintervals mapped into 2^n subtriangles. The order in which subtriangles have to be arranged in order to satisfy the requirements that adjacent subintervals be mapped onto adjacent triangles with an edge in common and that each mapping preserve the preceding one, is shown in Figure 6.

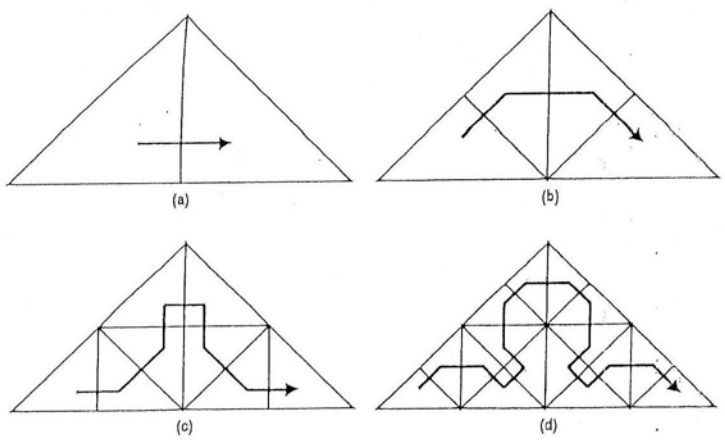

Figure 6: Generation of the Sierpinski Curve (from Sagan [2])

The curve starts from (0, 0) and ends at (2, 0). The requirement that the exit point from each subtriangle has to coincide with the entry point of the following one includes an orientation in each subtriangle, which is shown on Figure 7.

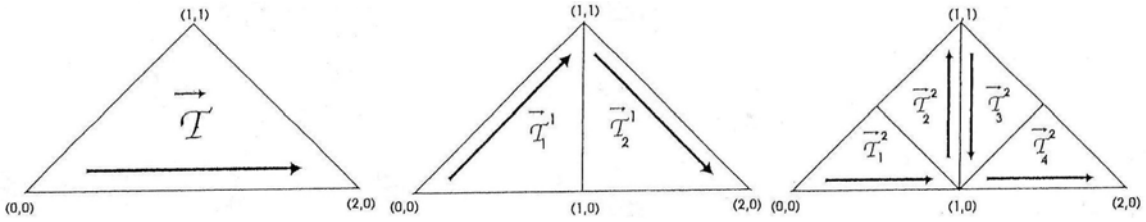

Figure 7: Mapping T onto its congruent parts (from Sagan [2])

3. Recursively Structured Triangular Grids and Sierpinski Curves

Starting from the simplest case of a computational domain, a right isosceles triangle acting as the starting cell, the computational grid is constructed in a recursive process. We recursively split each triangle cell into two congruent subcells. This splitting is repeated until the desired resolution of the grid has been reached. The grid may also be adaptive, as shown in Figure 8. The respective substructuring tree is shown next to it.

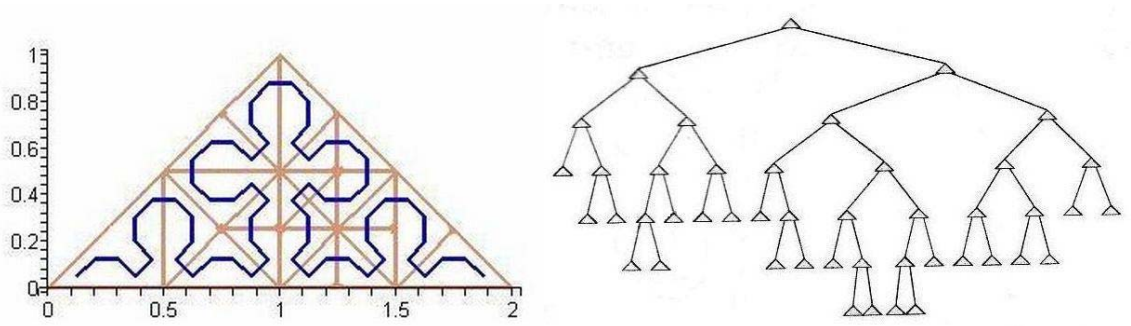

Figure 8: Recursive construction of the grid on a triangular domain (from Bader [1])

A respective uniformly refined recursive construction is used to define the Sierpinski curve, which is used to generate a linear order on the grid cells. This corresponds to a depth-first traversal of the substructuring tree. To store the grid structure therefore requires only one bit per cell to indicate whether a cell is a leave or whether it is adaptively refined.

There are a few extensions to this basic scheme, which offer more flexibility regarding complicated computational domains:

- instead of one initial triangle, a simple grid of several triangles may be used
- cells can be arbitrary triangles as long as the structure of the recursive subdivision is not changed: one leg of each triangular cell will be defined as the *tagged edge* and take the role of the hypotenuse
- subtriangles do not need to be real subsets of the parent triangle: the tagged edge (or hypotenuse) can be replaced by a linear interpolation of the boundary (see Figure 9)

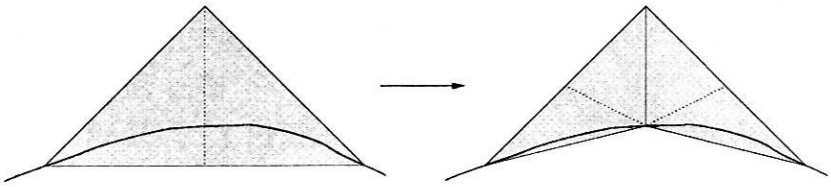
Figure 9: Subdividing triangles at boundaries (from Bader [1])

4. Discretization of the PDE

Consider, for example, a discretization using linear finite elements on the triangular grid cells. It will generate an element stiffness matrix and a right hand side for each cell. Accumulation of these local systems will lead to a global system of equations for the unknowns, which are placed on the nodes of the grid.

In our case we assume that storing the local or global system of equations is considered to be too memory-consuming. Instead, we assume that it is possible to compute the stiffness matrix on the fly or even hardcode it into the software. Then we only need a minimal amount of memory to store the recursive grid structure and the values of the unknowns. This is typical to iterative solvers, which contain the matrix-vector product between the stiffness matrix and the vector of unknowns.

In the classical node-oriented approach, this product would be evaluated line-by-line using a loop over the unknowns. This requires access to all neighboring nodes for each unknown. In a recursively structured grid this might be difficult: a neighbor may be part of an element that lies on an entirely different subtree. Therefore, the grid should be processed in a cell-oriented way.

5. Cache Efficient Processing of the Computational Grid

In such a cell-oriented processing, the problem is not to access all neighbors of the currently processed unknown, but to access all unknowns within the current cell. We will not store the indices of the unknowns for each element, but instead process the elements along the Sierpinski curve.

As we can see from Figure 10, the Sierpinski curve divides the unknowns into two halves, one lying on the left of the curve, the other half on the right. We can mark the respective nodes with two different colors: red (circles) and green (boxes). Processing the grid cells in the Sierpinski order, we recognize that the access to the unknowns is compatible with the access to a stack. Consider the unknowns 5 to 10: during processing the cells to the left of them, they are accessed in ascending order; during processing the cells to the right of them, they are accessed in descending order. In addition the unknowns 8, 9, 10 are in turn placed on top of the respective stack.

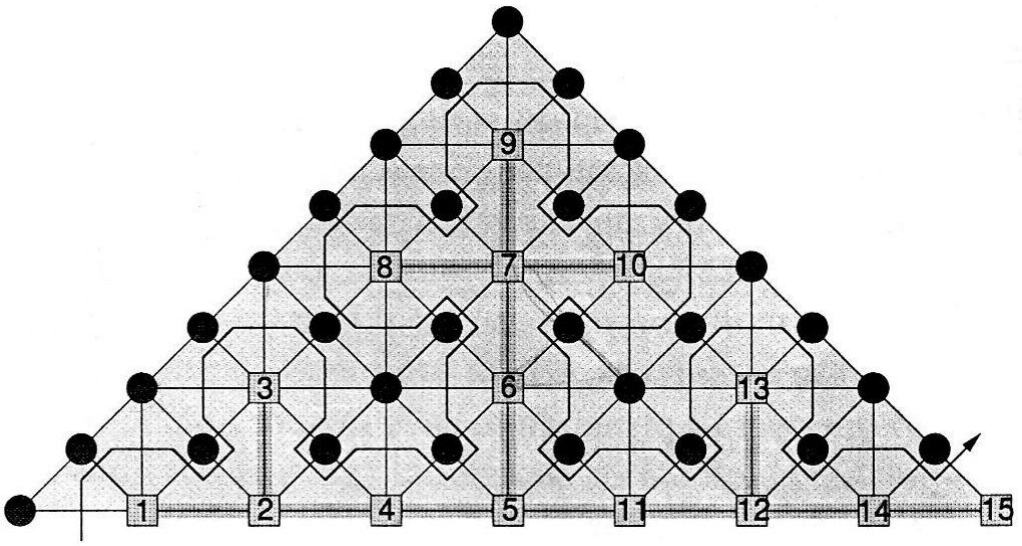

Figure 10: Marching through a grid of triangular elements in Sierpinski order. The nodes to the left and to the right of the curve are accessed in an order that motivates the use of a stack to store intermediate results (from Bader [1])

A system of four stacks is needed to organize the access to the unknowns:

- one read stack that holds the initial values of the unknowns;
- two helper stacks, a *green* and a *red* stack to hold intermediate values of the unknowns of the respective color;
- one write stack to store the updated values of the unknowns.

Whenever we move from a processed cell to the other, two unknowns can always be reused, those adjacent to the common edge. Therefore we only have to deal with the remaining two that are opposite to the common edge.

The remaining unknown in the exited cell will either be put onto the write stack (if its processing is complete) or onto the helper stack of the correct color (if it still has to be processed by other cells). To decide whether the processing completed or not, we can use a counter for the number of accesses. To determine the color of the unknown, we need to know whether it lies to the left or to the right of the curve. As the Sierpinski curve always enters and exits a cell at the two nodes adjacent to the hypotenuse, there are only three possible scenarios (see Figure 11):

- the curve enters through the hypotenuse – then it exits across the opposite leg (it will not go back to the cell where it came from);
- the curve enters through the adjacent leg and leaves through the hypotenuse;
- the curve enters and exits across the opposite legs of the triangle.

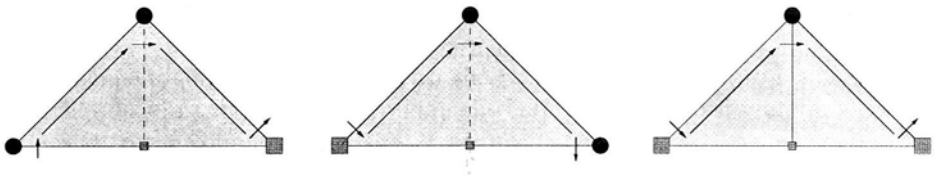

Figure 11: Three scenarios to determine the coloring of the nodes: to the left of the curve is red (circles), to the right if green (boxes) (from Bader [1])

The remaining unknown in the entered cell will either be taken from the read stack, if it has never been used before, or from the respective colored helper stack. This decision depends only whether the unknown has already been accessed before. Therefore, we consider whether the three adjacent triangle cells have already been processed or not. For two out of three this is known: the cell adjacent to the entering edge has already been processed; the cell adjacent to the exit edge has not. The third cell can be *old* (processed already) or *new* (not yet processed). Consequently, we split each of our existing three scenarios according to this additional criterion, and obtain six new scenarios (see also Figure 12):

- if at least one of the adjacent edges is marked as *old*, we have to take the unknown from the respective colored stack;
- if both adjacent edges are marked as *new*, we have to fetch the unknowns from the read stack.

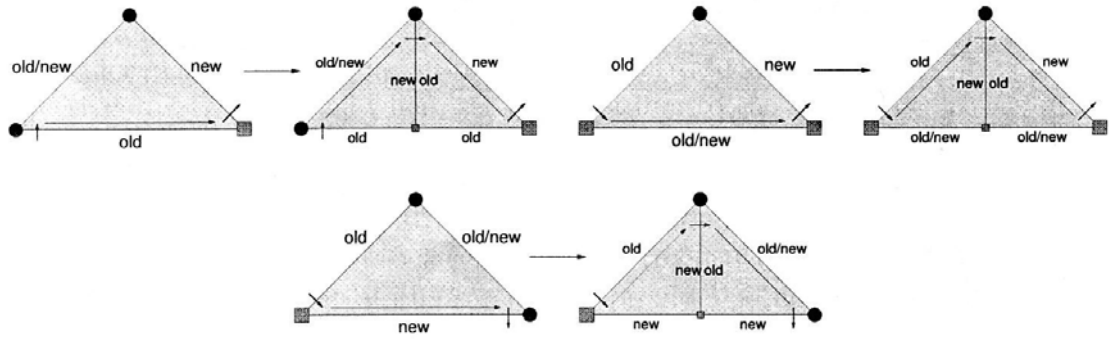

Figure 12: Determination and recursive propagation of edge parameters (from Bader [1])

Knowing the scenario of the cell, in case the cell is divided, we also know exactly the scenario for the two subcells as well. The processing of the grid can thus be managed by a set of six recursive procedures, which on the leaves perform the operations on the discretization level.

6. 3D Sierpinski Curves

From the 3D Sierpinski curve we expect to fill a tetrahedron. Therefore we define a *tetrahedron with a tagged edge* to be a 4-tuple $[x_1, x_2, x_3, x_4]$ with $x_1, x_2, x_3, x_4 \in \Re^3$ where the edge $\overline{x_1, x_2}$ is directed and takes the role of the tagged edge.

We then can bisect such a tetrahedron along the tagged edge into two sub-tetrahedrons in the following way:

$$[x_1, x_2, x_3, x_4] \to [x_1, x_3, x_4, x_5], [x_3, x_2, x_4, x_5], \text{ where } x_5 \in \overline{x_1, x_2}\,.$$

Using again Hilbert's geometric generating principle, we have the unit interval *I* subdivided into 2^n subintervals and the starting tetrahedron into 2^n sub-tetrahedrons. The Sierpinski curve is approximated by the polygonal line given by the tagged edges:

$$\overline{x_1, x_2} \to \overline{x_1, x_3}, \overline{x_3, x_2}$$

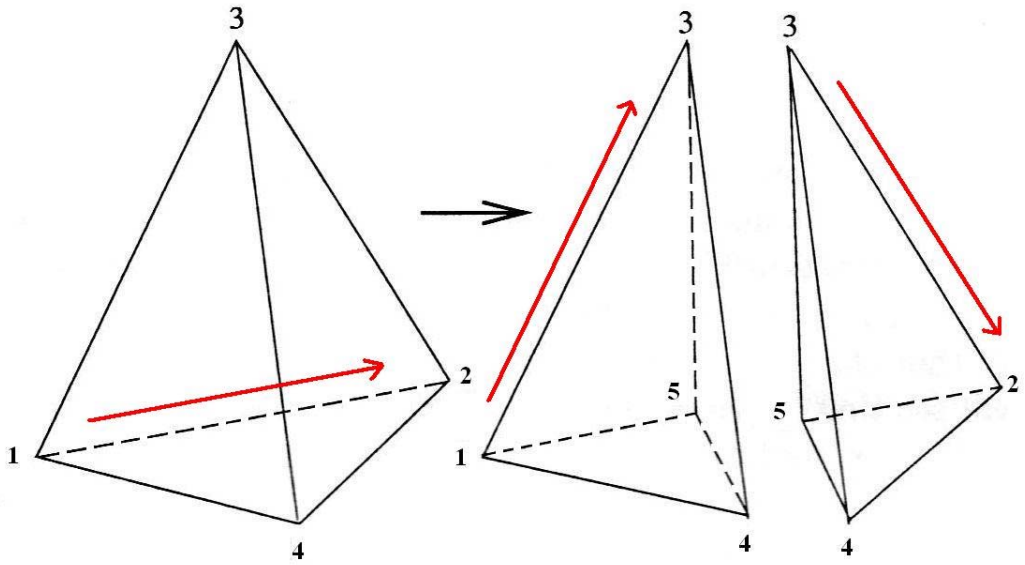

Figure 13: Bisection of a tagged tetrahedron. Sierpinski curve (red arrows): $\overline{x_1, x_2} \rightarrow \overline{x_1, x_3}, \overline{x_3, x_2}$.

7. Conclusion

We presented an algorithm to efficiently generate and process an adaptive triangular grid with minimal memory requirements. A result from Stevenson [3] states that in order to maintain *conformity* in any locally refined grid (triangles, tetrahedrons or any n-simplices) using recursive bisections, only finite number of additional bisections is needed. Furthermore, these additional bisections will maintain the locality of the refinement, meaning that the grid will not become globally uniformly refined.

It is hoped that, by implementing it, such a computational speed is achieved that is competitive with the algorithms based on regular grids. Extension of the scheme to the three-dimensional tetrahedron grid using the 3D version of Sierpinski's space-filling curve is currently subject to research.

References

1. M. Bader, Ch. Zenger. *Efficient storage and processing of adaptive triangular grids using Sierpinski curves*
2. H. Sagan. *Space-filling curves*
3. R. Stevenson. *The completion of locally refined simplical partitions created by bisection*
4. http://en.wikipedia.org