Computational Aesthetics in Graphics, Visualization, and Imaging (2010)
O. Deussen and P. Jepp (Editors)

Recursive scene graphs for art and design

Brian Wyvill[1] and Neil A. Dodgson[2]

[1]Department of Computer Science, University of Victoria, British Columbia, Canada
[2]The Computer Laboratory, University of Cambridge, Cambridge, UK

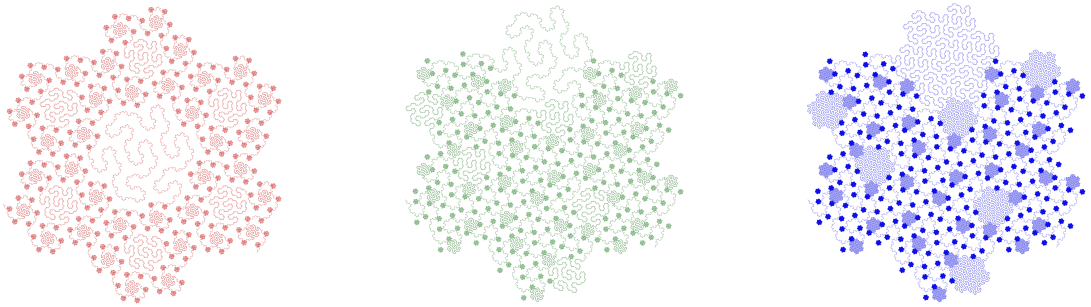


Figure 1: *Three Kochsnakes drawn using recursive scene graphs with both global and local limits on recursion.*

Abstract

Conventional scene graphs use directed acyclic graphs; conventional iterated function systems use infinitely recursive definitions. We investigate scene graphs with recursive cycles for defining graphical scenes. This permits both conventional scene graphs and iterated function systems within the same framework and opens the way for other definitions not possible with either. We explore several mechanisms for limiting the implied recursion in cyclic graphs, including both global and local limits. This approach permits a range of possibilities, including scenes with carefully controlled and locally varying recursive depth. It has applications in art and design.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—Line and curve generation I.3.6 [Computer Graphics]: Methodology and Techniques—Graphics data structures and data types

Acyclic scene graphs have long been a standard mechanism for representing the geometry in a scene [SC92]. Primitive objects, transformations, and material properties are all nodes in the graph. The graph is traversed from its root node, and the traversal is guaranteed to terminate, because the graph is finite and acyclic.

Iterated function systems [Hut81], by contrast, consist of recursive geometric definitions that require an external constraint to force termination. This can be as simple as a globally-imposed limit on the recursion depth.

We investigate the possibilities suggested by *recursive* scene graphs (Section 2). The immediate concern is to ensure that there is a mechanism for guaranteeing termination. Both global and local termination criteria are possible, giving a wider scope of expression than either acyclic graphs or iterated function systems. We compare our method to previous work (Section 1) and demonstrate examples (Section 3).

Our contributions are (1) a demonstration that recursive scene graphs, with guaranteed termination, produce a range of interesting effects; (2) the definition and testing of a range of mechanisms for limiting local and global recursion depth in recursive scene graphs; and (3) mechanisms for the correct handling of mutual recursion in such graphs.

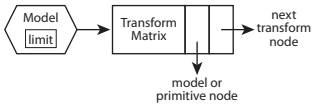


Figure 2: *The basic node types in a DCG: the model node contains an integer limit and a pointer to a list of transform nodes. The transform node contains a geometric transformation, other information that sets object properties, a pointer to the next transform node, and a pointer to either a model node or a primitive node.*

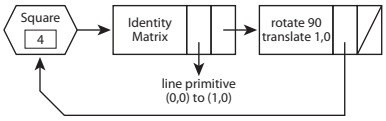


Figure 3: *An example DCG: this draws a square. A line primitive is drawn, a transform executed to move to the end of the line and rotate through 90°, then a recursive call to the model node. The counter in the model node decrements every time it is visited. When it reaches zero, the cycling ceases.*

1. Comparison to Previous Work

The basic concepts of directed cyclic scene graphs, on which we build, were outlined in the 1970s by Geoff and Brian Wyvill [Wyv75a, Wyv75b, **?**]. Object instancing in computer graphics goes back further, to Sutherland's *Sketchpad* [Sut63]. Iterated function systems have their origins over a century ago [vK06], were investigated by Mandelbrot and others in the 1970s [Man82], and set on a firm mathematical footing by Hutchinson in the early '80s [Hut81]. L-systems were devised, by Lindenmayer in the 1960s, as a mathematical theory to describe plant development [Lin68]; in computer graphics, they were developed further as a combination of rewriting rules and turtle graphics [PL90].

Ebert [EMP*03] uses the term *cyclic scene graph* to mean both the limit of an L-system and, equivalently, an iterated function system. In his cyclic graphs there are no primitives and the scene is defined purely by the geometric transforms. This requires non-standard rendering algorithms, such as the ray tracers described by Kajiya and Hart [Kaj83, HD91]. In our definition, by contrast, we explicitly ensure that there is termination and include primitives in the scene graph. These allow us to render the scene using a variant of the normal scene graph traversal algorithm.

Gervautz, Traxler, and Schmalstieg [GT96, SG97] used a similar concept to us, with counters that decrement on each cycle to ensure termination, and the possibility of using more complex parameterisations to change behaviour at each level of recursion. Their main objective was to save on memory usage and network bandwidth, having observed that directed cyclic graphs are a compact representation. Memory and bandwidth are now cheap, so our work considers instead whether such recursive scene graphs can be used with artistic intent. Though our work was developed independently of Gervautz et al., we can be considered to be extending their work by providing new mechanisms: local limits, choice nodes, and mutual recursion.

2. The Directed Cyclic Graph Mechanism

Our scene graphs are directed cyclic graphs that consist of three types of node: model nodes, transform nodes, and primitive nodes (Figure 2). Each model node heads a linked list of transform nodes. Each transform node points to either a model node or a primitive node. As with conventional scene graphs, the transform nodes can be implemented either as general transformations or as a range of more specialised versions, such as translation, rotation, texture, and colour nodes.

In our implementation, a model node and its associated list of transform nodes are all processed within a single function, with the list of transform nodes processed in a *while* loop. Descent into a child model node is handled using a recursive function call. Pseudo-code for the traversal algorithm is given in Appendix A.

2.1. Global Limits

Adding cycles to a scene graph requires careful management. Traversal of the graph must always terminate, regardless of complexity. Our solution is to enforce the rule that all cyclic references go via a model node and that model node traversal is managed to ensure termination.

The mechanism for this is a simple integer counter within each model node (see example in Figure 3). If the counter is zero (more strictly, if it is not positive), then its list of transform nodes is not processed. If it is positive, then the counter value is pushed onto a stack, the counter then decremented, and the model's transform nodes are processed.

Schmalstieg and Gervautz [SG97] used a similar mechanism of a counter that is decremented at each level. Their method also included general calculation that allowed more complex operations on the counter than simple decrement, though they do not show any uses of this.

In our mechanism, descent into a model node requires a recursive function call; on return from this call, the value of the counter is popped off the stack. This allows multiple instances of the node to occur within the recursive definition, as demonstrated in Figure 4. The use of the stack also permits local limits to be defined, with the guarantee that the original limit will be restored on exit from the scope of the local limit.

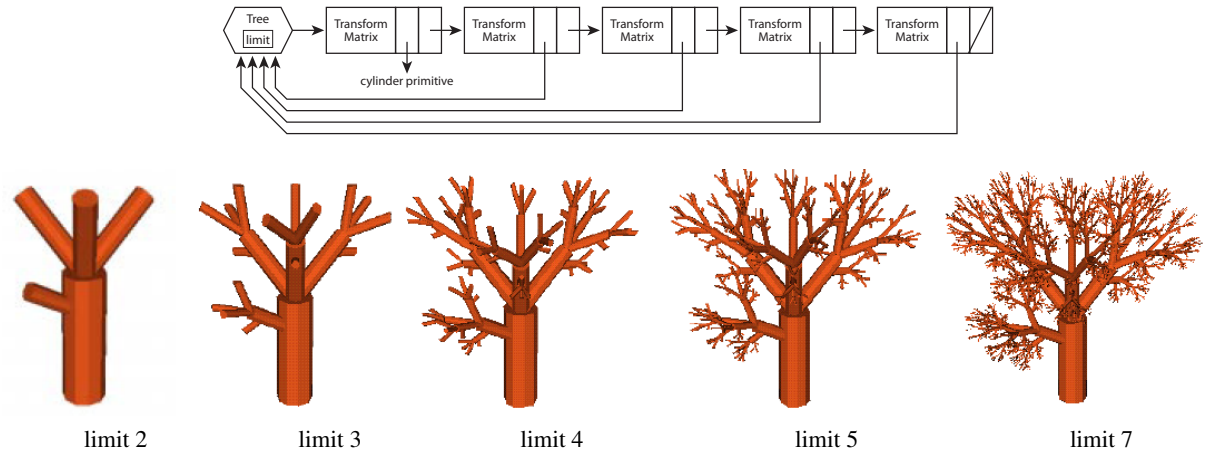


Figure 4: *Top: a multiply instanced DCG definition to draw a cylinder tree. Bottom: output from this with different limits.*

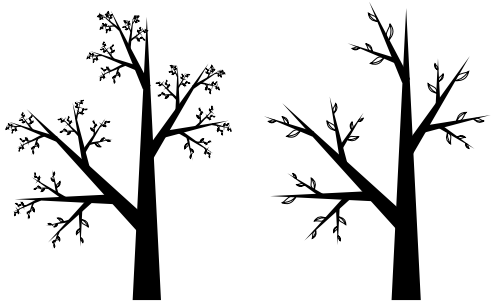


Figure 5: *Parameterised limit changes. (a) A thorn tree defined with a basic DCG, recursion limit five. (b) The same thorn tree with parameterised limits reducing the recursion depth on the second (middle) and third (highest) branches. Note that the branch pointing vertically down at the left hand side is at the same recursion depth (five) in both models, having been generated as the first branch off each parent branch.*

2.2. Local Limits

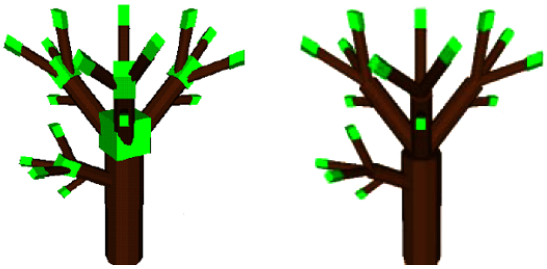
It is common practice, in systems that use instancing, for attributes such as colour to be passed down the hierarchy. In a similar fashion the transform node can override the global recursion limit of the model that is transformed, with a local limit stored in the transform. We have devised three ways in which a transform node is able to do this.

1. Set a *local limit*. This changes the limit for the child model node, but does so on this cycle only. Deeper recursive passes through the transform node are *not* allowed to set the local limit again because this would cause infinite recursion. This is implemented using a guard in the transform node. See Figure 10 for an example.
2. Set an *absolute limit*. This is implemented identically to the local limit but without setting the guard. Every time this transform node is traversed, the same absolute limit is set on the model node. This can easily cause infinite recursion and therefore must be used, with caution, within some outer cycle that has a model node with a normally-managed limit. See Figure 13 for an example.
3. Set a *parameterised limit*. This changes the child node's limit based on the current value of the parent node's limit. This type of limit is most sensibly used when the child and parent nodes are the same model. If the change can be to a value greater than the current limit then infinite recursion is a possibility, again requiring cautious management, as for absolute limits. If the change is guaranteed to be a value less than or equal to the current limit then termination is also guaranteed. Schmalstieg and Gervautz [SG97] mechanism is also able to handle this kind of limit.

An example use of parameterised limits is the tree in Figure 5(b). There are three branches. The first (lowest) branch recurses normally. Parameterised limits are attached to the second and third branches. On the second branch, the recursion limit is decremented by an extra one on each cycle. On the third branch, it is decremented by an extra two. This means that recursion terminates on these branches more quickly than on the first branch. By combining the parameterised limit changes with the different scale factors on the different branches, we get a result where the terminating leaves are more uniformly sized than in the non-parameterised case.

2.3. *Choice* Nodes

It is useful to have transform nodes that behave differently depending on the current recursion limit. We therefore implement a mechanism whereby a transform node descends



(a) (b)

Figure 6: *The tree definition from Figure 4 with a green cube added at the end of the cylinder. (a) The green cube is attached to a standard transform node. (b) The green cube is attached to a* choice *node with value set to zero; this means that the green cube is only drawn at the final recursive step.*

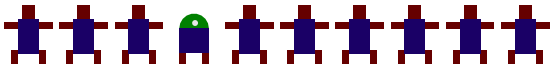


Figure 7: *A level 10 recursive definition of a row of robots. A* choice *node has been used to insert a different model when the recursive limit is equal to six.*

into different children depending on the current value of the recursion counter.

An example of the *choice* node is shown in Figure 6. A green cube added to the end of the cylinder in the tree definition of Figure 4. By attaching the green cube to a *choice* node, set to trigger when the counter hits zero, it is only drawn on the final round of recursion. This parallels Schmalstieg and Gervautz's [SG97] termination mechanism. Figure 7, by contrast, shows an example where the choice is made at a level other than the terminating one. A DCG is used to represent a row of identical models. A *choice* node is used to replace a single instance with a different model. Figure 8 shows a range of choices being made at different recursion levels. The idea of including a condition on each transform node so that it is only entered at *one specific* global recursion level was first implemented in the 1970s and used for defining space filling curves [Wyv75a, WW83]. Here we have extended that idea in three ways: to allow multiple different choices (e.g., Figure 8); to allow for there to be a default model, with the effect that the default model is *replaced* by a different model at some levels (e.g., Figures 7 and 8); and to allow for there to be no default model within the *choice* node, with the effect that an extra model is *added* to the scene at a particular level (e.g., Figure 6(b)).

3. Example Uses

An example system has been implemented and to simplify the construction of the data structure we defined a simple scripting language to illustrate the uses of the system. For

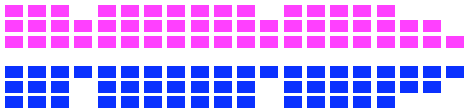


Figure 8: *Aircraft seating plan: an example of a* choice *node with several alternatives. There are special cases at recursion levels 0, 1, 2 (front of plane), 8, and 16 (emergency exit rows).*

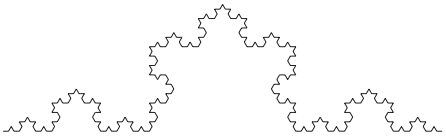


Figure 9: *The Koch Curve Level 5*

presentation purposes we provide the script, which describes each example rather than a diagram of the data structure that is generated. The script includes a number of key words that should be self explanatory to those familiar with computer graphics. The '%' symbol precedes comments where the code needs some explanation. The command 'a' introduces a new transform node. The model or primitive is named first followed by any translations, colour changes, or local limits that need to go in the transform node. The examples given are all 2D for simplicity.

```
% Definition of a square
primitive line 0 0  1 0 end
define square
 limit 4
 a line
 a square rotate 90 translate 1 0
end
```

The above definition builds the DCG in Figure 3. It has a model node called square with a global recursion limit of four. There are two transform nodes: one that calls a primitive line and the other that refers recursively to square.

3.1. Selective Recursion

Global recursion limits stored in a model node can be replaced during graph traversal by a local limit stored in the parent transform node (Section 2.2). To illustrate this consider the space filling Koch curve defined below. The *between* operator (`between` $x_1\,y_1\,x_2\,y_2\,x_3\,y_3\,x_4\,y_4$) is a simple syntactic mechanism that specifies the affine transform matrix (scale, rotation, translation) required to take an object defined by two points, $(x_1,y_1),(x_2,y_2)$, and place it at two other points, $(x_3,y_3),(x_4,y_4)$.

```
% Koch curve
primitive line 0 0   1 0 end
def koch
limit 5
```

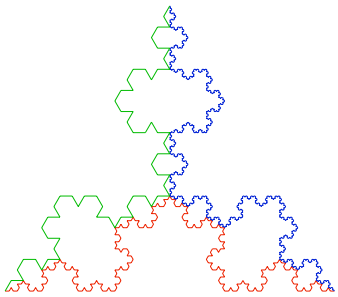


Figure 10: *A snowflake of Koch curves, at levels 4, 5 and 6*

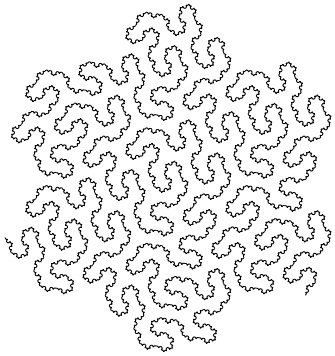


Figure 11: *Kochsnake: A level 4 Gosper Flowsnake [Gar76] with lines replaced by a Koch curve of level 5.*

```
 a line if 0
 a koch between 0 0  3 0  0   0      1   0
 a koch between 0 0  3 0  1   0      1.5 0.866
 a koch between 0 0  3 0  1.5 0.866  2   0
 a koch between 0 0  3 0  2   0      3   0
end
```

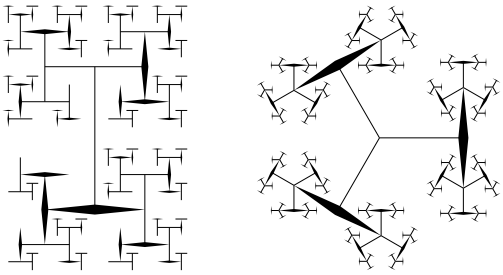
Figure 9 shows the result.

We continue the example by defining a version of the Koch *snowflake*, consisting of a triangle of Koch curves. We use local limits to make the three sides of different recursion limits, see Figure 10.

```
def snow
 a koch                    locallimit 5  col 0.9 0 0
 a koch rot 120  trans 3 0  locallimit 6  col 0 0 0.8
 a koch scale 1 -1  rot 60  locallimit 4  col 0 0.7 0
end
```

Our formulation makes it easy to mix space filling curves of a different type (Figure 11).

```
% Kochsnake: Gosper's Flowsnake using the
% Koch curve (defined above) as a primitive
def ksnk
 limit  4
 a koch between  0 0   3 0       -1.5 0.866  3   0   if 0
 a ksnk between -1.5 0.866 3 0   -1.5 0.866  0   0
 a ksnk between  3 0  -1.5 0.866  3   1.732  3.0 0
 a ksnk between  3 0  -1.5 0.866  3   1.732  1.5 2.598
 a ksnk between  3 0  -1.5 0.866  1.5 2.598  0   3.464
 a ksnk between  3 0  -1.5 0.866  1.5 0.866  0   1.732
 a ksnk between  3 0  -1.5 0.866  0   0      1.5 0.866
end
```



(a) (b)

Figure 12: *Simple mutual recursion. (a) Each model recurses on both itself and one other model. The lines have a recursion limit of five; the diamonds a recursion limit of three. (b) Each model recurses on one other model, which in turn recurses back on the first.*

Figure 1 demonstrates how the local limits used in Figure 10 can be applied recursively. We modify the definition of the Kochsnake, above. The data structure has seven recursive branches. The left and middle objects in Figure 1 are both Kochsnakes of level four; the difference between them being that the sixth and fourth branches, respectively, have a local limit set to three. The rightmost object, like the middle object, has its fourth branch modified. In this rightmost case, the fourth branch has its local limit set to four, rather than three. Remember that local limits are exchanged for global limits after traversal has started, so all of the fourth branches, at all levels of recursion, gain an extra level over those in the middle object.

3.2. Mutual Recursion

A basic example of mutual recursion is shown in Figure 12(a).

```
% the primitives line and diamond draw those shapes
% each is 2 units long, from -1 0 to 1 0

def linemodel
limit 5
 a line
 a linemodel  scale 0.7  rot  90  trans  1 0
 a diammodel  scale 0.7  rot -90  trans -1 0
end

def diammodel
limit 3
 a diamond
 a linemodel  scale 0.7  rot  90  trans  1 0
 a diammodel  scale 0.7  rot -90  trans -1 0
end
```

There are two types of model, the line and the slim diamond. Each model calls one instance of itself and one instance of the other. The line model has a recursion limit of five; the

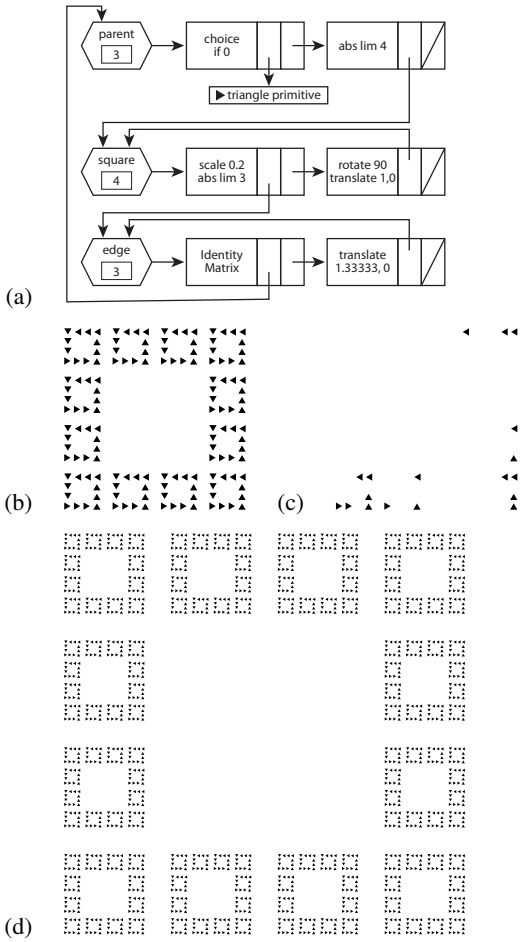


Figure 13: *Mutual recursion. A square is made up of four edges. An edge is made up of three squares. The termination primitive is a filled triangle, which is drawn rather than recursing to another square. (a) the DCG; (b) is generated by the DCG. (d) by increasing the limit on* parent *to four.(c) is generated by the DCG without the* absolute limits.

diamond model a recursion limit of three. If you start at the centre of the model and trace any path to a leaf node, you will find that there are at most five lines and at most three diamonds on the path. Figure 12(b), by contrast, shows mutual recursion where each model calls a number of instances of another model but does not call instances of itself.

To illustrate the power of mutual recursion, consider the border in Figure 13. The script below produces the DCG shown in Figure 13(a), which generates the border in Figure 13(b). A square consists of four edges. Each edge consists of three squares. This mutual recursion can continue arbitrarily deep, with termination drawing a primitive in place of a square.

```
def edge
 limit 3
 a parent
 a edge trans 1.33333 0
end

def square
 limit 4
 a edge    scale 0.2 absolutelimit 3
 a square  rot 90    trans  1 0
end

def parent
limit 3
  choice
    % choose square if not 0, triangle if 0
    a square absolutelimit 4
    % the primitive triangle draws the terminating shape
    a triangle if 0
  end choice
end
```

Figure 13(c) illustrates the same scene without absolute limits. The code is identical to that above with the two '`absolutelimit` *n*' commands removed. Without the absolute limits we do not produce the desired effect, because the limits get used up as recursion progresses. In particular, notice how the number of edges on each sub-square decreases by one on each subsequent edge of the main square, and how the number of primitive triangles drawn on an edge decreases by one on each subsequent square within any given edge.

The code above, this time unmodified, defines the scene graph that produces Figure 13(b). It uses an *absolute limit* to push fresh limits into the sub-squares and the sub-edges. This allows all of the squares and edges to be complete but requires there to be some parent model with a limit that is *not* overridden by an absolute limit. Incrementing the parent model's limit by one (to `limit 4`) gives the even more detailed version in Figure 13(d).

3.3. Further Examples

Figures 14–18 demonstrate a range of other scenes generated by our system. These have been principally driven by a desire to produce an artistically interesting effect.

4. Conclusion

For a long time the representation of hierarchical models in computer graphics systems has been based on the directed acyclic graph (DAG). Although the use of recursive cycles (DCG) was suggested in the early 1970s, little use has been made of this structure since that time. There are several reasons for this, including the fact that the use of recursion has been viewed as inefficient as well as the lack of good techniques for handling recursively defined models. The progress in hardware has taken care of the earlier view of efficiency and in this work we have proposed some mechanisms for handling, global and local recursion limits, and a method for dealing with mutual recursion.

We have demonstrated that recursion may be controlled and used in computer graphics systems and shown some example uses in the area of art and design. In particular, we have demonstrated examples where complex objects are generated by simple definitions.

It has not escaped our attention that any of our scripts could easily be compiled into a DAG rather than a DCG. One could justifiably ask what advantage obtains from using DCGs. Our observation, throughout our investigation, is that the DCG provides an interesting alternative way of thinking about and experimenting with geometric designs, with serendipitous artistic effects arising that the designer had not anticipated (e.g., Figures 14 and 15). They allow exploration of a space of design, using a scripting specification that is short and is straightforward for anyone familiar with computer programming. †

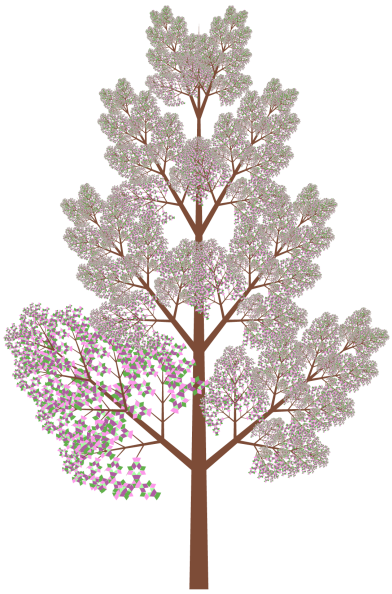


Figure 14: *Tree: using local limits (thanks to Gemma Bone for choosing the colours).*

References

[EMP*03] EBERT D. S., MUSGRAVE F. K., PEACHEY D., PERLIN K., WORLEY S.: *Texturing and Modeling: A Procedural Approach*. Morgan Kaufmann, San Francisco, 2003. 2

[Gar76] GARDNER M.: Mathematical games: In which "monster" curves force redefinition of the word "curve". *Scientific American 235* (1976), 124–133. 5

[GT96] GERVAUTZ M., TRAXLER C.: Representation and realistic rendering of natural phenomena with cyclic CSG graphs. *The Visual Computer 12*, 2 (1996), 62–74. 2

[HD91] HART J. C., DEFANTI T. A.: Efficient antialiased rendering of 3-D linear fractals. In *SIGGRAPH* (New York, 1991), ACM, pp. 91–100. 2

[Hut81] HUTCHINSON J.: Fractals and self-similarity. *Indiana Univ. Math. J 30*, 5 (1981), 713–747. 1, 2

[Kaj83] KAJIYA J. T.: New techniques for ray tracing procedurally defined objects. *ACM Trans. Graph. 2*, 3 (1983), 161–181. 2

[Lin68] LINDENMAYER A.: Mathematical models for cellular interactions in development. Part I: Filaments with one-sided inputs. Part II: Simple and branching filaments with two-sided inputs. *Journal of Theoretical Biology 18*, 3 (1968), 280–315. 2

[Man82] MANDELBROT B.: *The Fractal Geometry of Nature*. Freeman, 1982. 2

[PL90] PRUSINKIEWICZ P., LINDENMAYER A.: *The Algorithmic Beauty of Plants*. Springer-Verlag, New York, 1990. 2

[SC92] STRAUSS P. S., CAREY R.: An object-oriented 3D graphics toolkit. In *SIGGRAPH* (1992), ACM, pp. 341–349. 1

[SG97] SCHMALSTIEG D., GERVAUTZ M.: Modeling and rendering of outdoor scenes for distributed virtual environments. In *VRST '97: Proceedings of the ACM symposium on Virtual Reality Software and Technology* (1997), pp. 209–215. 2, 3, 4

[Sut63] SUTHERLAND I. E.: Sketch pad: a man-machine graphical communication system. In *Proc. Spring Joint Computer Conference* (1963), pp. 329–346. 2

[vK06] VON KOCH H.: Une méthode géométrique élémentaire pour l'étude de certaines questions de la théorie des courbes planes. *Acta Mathematica 30*, 1 (1906), 145–174. 2

[WW83] WITTEN I., WYVILL B.: On the generation and use of space filling curves. *Software Practice and Experience 13* (1983), 519–525. 4

[Wyv75a] WYVILL B.: *An interactive graphics language*. PhD thesis, University of Bradford, 1975. 2, 4

[Wyv75b] WYVILL G.: Pictorial Description Language II. *Proc. ONLINE 75, Brunel University, Uxbridge, UK* (1975). 2

Appendix A: Simplified Traversal Algorithm

The algorithm described in pseudo-code covers the basic three nodes described in Section 2.

```
traverse(model* md, Matrix m)
{
  if( md IS primitive ) output( md, m ) ;
        // multiply coordinates by matrix m
  else
  {
    Transform t;
    t = md->firstTransform;
    if( md->limit > 0 ) {
      push( md->limit ) ;
      md->limit-- ;
      while ( t != NULL ) {
        traverse( t->childModel, m * t->mat ) ;
        t = t->next ;
      } // while
      pop( md->limit ) ;
    } // if
  } // if
} // traverse
```

† This work was partially supported by the Natural Sciences and Engineering Research Council of Canada.

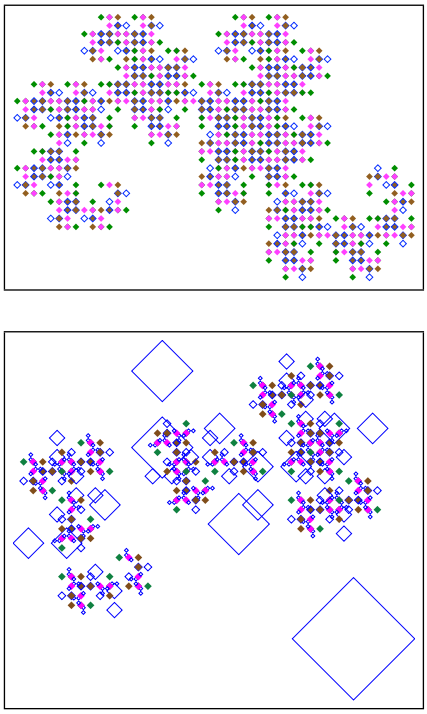


Figure 15: *Dragon curves. Top: a dragon curve of level six with each of the four branches terminating with a different coloured block. Bottom: the same dragon curve with local limits on one branch: note how this generates white squares that are larger and smaller than the coloured squares, because the local limit kicks in at different levels on different branches.*

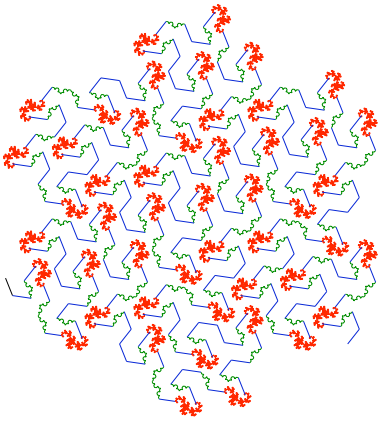


Figure 16: *Kochdragsnake: the Kochsnake with a choice node that allows some nodes to be substituted by a dragon curve and some by straight lines.*

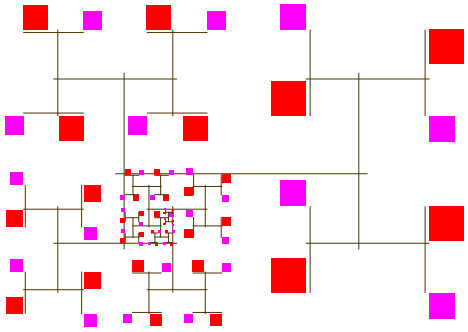


Figure 17: *Modified recursive H: an attempt to generate a pleasing effect using varying levels of recursion to balance the expected and the surprising.*

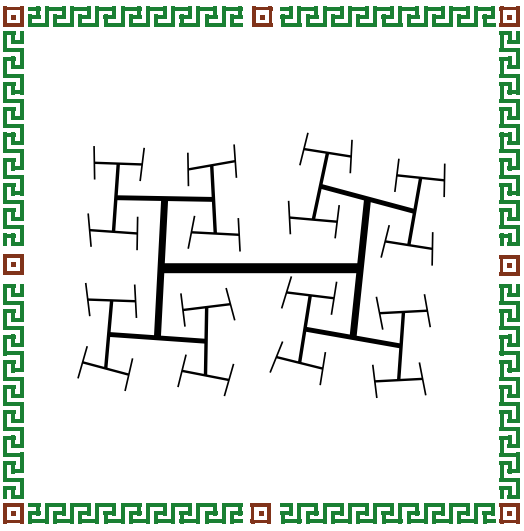


Figure 18: *Two examples. The central recursive H uses randomly parameterised rotation. The edges in the border are defined using a* choice *node. The script for the border model is shown below.*

```
def lineofblocks
  limit 20
  choice
    a block  % default choice
    a corner if  0
    a bright if  1
    a bleft  if  9
    a cpiece if 10
    a bright if 11
    a bleft  if 19
  end choice
  a lineofblocks trans 90 0
end
```