Procedural Geometry Synthesis on the GPU

Patrick Lacz  John C. Hart
University of Illinois Urbana-Champaign
June 4, 2004

Abstract

We present a framework for translating procedural production rules, such as parametric L-Systems based on scene graphs instead of turtle graphics, into a form that can iteratively grow geometry completely on graphics hardware. The formulation is extended to handle context-sensitive, conditional and non-deterministic rules. We also examine and enhance existing GPU-based techniques for bitonic merge sorting. The result is procedural geometry synthesized directly on the graphics card.

1 Introduction

Procedurally generated geometry remains an important tool in developing highly detailed, natural-looking models. Most prominently, L-systems [19] have been used in a great number of natural modeling applications, and have found extended applications in fractals, urban geometry synthesis, and subdivision surfaces to name a few. Along with extensions to L-systems, alternative geometry synthesis languages have been developed such as Procedural Geometric Instancing [13] and Grove [2]. Thus far, these important graphical tools have been limited to CPU implementations. We propose a framework for utilizing the GPU with these tools.

The programmable elements of the modern GPU were developed primarily to support advanced local illumination and shading models and the smoothing of polygonal meshes. However, the fast streaming processing capabilities of the GPU make it an attractive computational platform for accelerating other applications in computer graphics, including e.g. solving linear systems [4, 9], physical simulation [11], ray-tracing [7, 20], and global illumination [8, 21].

These applications make use of a fragment feedback loop where the rendered output is used as input in later shading passes, and the geometry sent to the GPU is often a simply a “screen-filling quad” whose rasterization activates all of the fragment programs. In OpenGL literature, this feedback loop is called “render-to-texture,” where the rendered off-screen output is used as a texture in subsequent passes, using a data structure called the pbuffter1.

While the fragment feedback loop is quite powerful, its application is designed to process images and textures, and its application to processing geometry is limited. For example, geometry images are 2D images corresponding to 3D vertex locations resulting from parameterizing meshes into the rectangular image domain [10]. Such geometry images are easily deformed using the GPU, however vertices may not be inserted or deleted from the mesh.

The application of the GPU to the task of procedural geometry synthesis requires more control over the creation and destruction of geometry elements than is permitted by fragment programs. These capabilities have become available in a recent GPU extension that uses the output of a rendering pass as input geometry. This feedback loop is often referred to as “render-to-vertex array/buffer” [3]. In this specification, the pixels of the framebuffer output are stored in a buffer and reinterpreted by the vertex shader as an indexed face set.

The goal of this paper is to reconfigure the vertex and pixel shading architecture of the GPU into a geometry processing pipeline capable of interpreting and instantiating procedural geometry, described for example by an L-system. To ease implementation and accelerate processing, the terminal symbols in our L-system are scene-graph nodes rather than the turtle graphics instructions commonly found in most L-system applications [19]. This enables the GPU to process procedurally-generated geometry in a context-independent randomly accessible manner which greatly improves rendering efficiency and more closely adheres to the scene graph data structure commonly used in computer graphics modeling systems.

1 An F-buffer data structure has also been proposed to support the same feedback that bypasses the texture map and supports overlapping triangles [17].
2 Previous Work

The previous work that relates to geometry synthesis on the GPU has thus far concentrated on the application of subdivision curves and surfaces. We present a more general technique that is not specific to one domain. Měch and Prusinkiewicz [18] have investigated generating subdivision curves using specialized L-system productions. Their work focused on subdivision curves and the possible extension to surfaces. Bolz and Schröder [5] have developed a subdivision surface implementation using a “render-to-vertex buffer” technique. Their algorithm performs Catmull-Clark subdivision by computing the limit surface for a tessellated patch of the input mesh. The tessellation is not driven by procedural rules and thus is unsuitable for the procedural instancing our research pursues.

Both our work and that of Měch and Prusinkiewicz use parametric L-systems as a model for the allowed production rules. These L-systems have symbols that can contain internal state; these symbols are called modules. Parametric L-systems have recently been used in describing and classifying subdivision surface patterns [22]. The L+C language [14] served as reference for current capabilities of parametric L-systems. Our algorithm provides a similar framework for creating executable procedural geometry using Cg as the implementation language rather than C.

The Měch paper is limited to performing a single linear combination of data from one level to the next. In comparison, we are able to apply arbitrary transformations that contain potentially complex dependencies. We expand on the amount of data able to be stored with a single symbol. They introduce a means of avoiding generating invalid modules by a production length-summation stage. Unfortunately, the number of modules this method can produce is limited by the maximum pbuffer width. We have developed a more general and powerful technique for interpreting or instancing the L-system as geometry.

A central component of our implementation consists of sorting the current set of modules. The bitonic mergesort algorithm we use has been described in detail by Ian Buck and Tim Purcell [6, 21]. Our work adds analysis on improving performance for smaller sets of elements.

3 Background

An L-system is a parallel graph grammar defined on an alphabet of (terminal and non-terminal) symbols [15]. The L-system consists of a collection of one or more productions which map each non-terminal symbol to one or more (terminal and/or nonterminal) symbols, and the defining action of an L-system is that productions are applied to all non-terminal symbols in a string simultaneously, in parallel, as opposed to e.g. a context-free grammar where productions are applied in a prescribed order. Finally, an L-system contains an axiom which serves as the initial string of symbols on which the L-system applies its first round of productions.

Figure 1 shows an example of a simple 2-D branching structure.
A string of symbols is converted to geometry by a left-to-right parsing and execution of the corresponding instructions to the turtle. At each step of this parsing, the turtle has a given state which in 2-D can be denoted by the quadruple \((x, y, \theta, \alpha)\) where \(x, y\) indicate the position of the turtle, \(\theta\) its orientation as an angle with respect to a predefined direction, and \(\alpha\) a global scale of the turtle steps and cylinders. The execution of the turtle instruction corresponding to each symbol in a string can potentially change this state. This situation leads to a rather inefficient serialization of the graphics generated by the turtle graphics paradigm, and one especially problematic for the parallel resources of the GPU.

The serial turtle state problem can be ameliorated by techniques developed previously for converting an L-system into a recurrent iterated function system \([19]\) or a scene graph \([12]\). These methods accumulate the state of the turtle before each non-terminal symbol, such that the recursion of an L-system can be replaced by the recursion of an iterated function system or of a cyclic instancing scene graph.

We thus translate the target of each (recursive) production of an L-system into a collection of targets, each consisting of a non-terminal prepended with the turtle commands leading to its state\(^2\). For example, the L-system production (1) becomes

\[
L \rightarrow \{ aL, af + L, aff - L, aff - L, aflL \}. \tag{2}
\]

In this form, the L-system production becomes more amenable to parallel implementation. Each of the elements of the set of targets on the right-hand-side of this production can be further recursed and evaluated without processing any of the other elements. We thus utilize this form in the mapping of L-system interpretation described in the next section.

\section{4 Synthesis on the GPU}

Our implementation of these concepts follows efforts to make executable L-systems, such as L+C. A translation is found from the production rules to a source language, in this case Cg. We concentrate on the difficulty of mapping the problem to graphics hardware and the techniques used to overcome its challenges.

\subsection{4.1 Framework}

Our framework is built on top of a Cg application. A procedural scene graph is represented as a collection of non-terminal symbols with state, productions, and an initial axiom. Each production is represented as in (2). The turtle commands of that example become operators, which transform the state of the source symbol to that of the target symbol. Operators are not stored explicitly. They correspond to snippets of Cg code with constant or symbolic references for the CPU to manage.

The state contained by the non-terminal symbols is stored as fragments in framebuffers on the GPU. Each framebuffer in our model is capable of holding a limited number of parameters, so for all but trivial cases, the data must be distributed between several framebuffers. To the set of non-terminal symbols, we add \(\emptyset\), a non-terminal symbol that has no state. Symbols of this type are removed from the string after an iteration. See Section 4.3 for details.

Once the state of the modules have been divided into buffers, the operators are split by the state being written, inserted into Cg programs, and compiled. We chose Cg for the ease of working in a re-targetable high-level language \([16]\). Our implementation provides the user with the ability to load an axiom, apply the production rules to the current string of symbols, and instance the current string.

Creating an automated implementation of this process remains for future work. Thus far, our Cg code has been generated by hand according to the guidelines of the framework.

\subsection{4.2 Algorithm Overview}

The algorithm consists of maintaining of the set of framebuffers representing the symbol string. Each iteration pass

\footnote{And also possibly other strings of terminal symbols not shown in this example.}
amplifies the number of symbols using fragment programs derived from the production rules. The results are then sorted to remove the Ø symbols that may have been generated by our application. We describe how this framework can support context-sensitive productions and non-deterministic operations.

Instancing passes generate geometry from the set of symbols. The CPU provides a mesh to use for instancing for each symbol. Using an additional sorting pass, we are able to perform a culling pass to instance a subset of the symbols in the string. The CPU can isolate the visible subset or a single symbol type without actually touching the data.

The basic outline for the algorithm is: apply production rules, sort by symbol type and remove Ø symbols, repeat desired number of iterations, instance geometry from the symbols.

4.3 Representation

The symbol string is stored and maintained in a set of framebuffers. Each pixel in a framebuffer holds a portion of the state of the symbol it corresponds to. The combination of all the pixels in the same position in the set of framebuffers comprises the state for that symbol.

If there are multiple non-terminal symbols, the symbols are enumerated and stored within the state in a consistent location. Thus, even if different symbols contain different data, it is possible to determine which symbol a pixel represents in a consistent fashion.

Each production is decomposed into a set of products as in the example from Section 3. Each product has a prefix of operators (terminal symbols that are never explicitly generated, but influence the state of the product), and the new target symbol. The operators contain code that must access state from the source symbol and store the result in the state for the new symbol. Once the data has been distributed on the frame buffers, the operators are divided by which framebuffer holds the data those operators write. These sets of operations are then decomposed into fragment programs. The structure of these programs is described in Figure 3.

4.4 Iteration

Figure 3 shows the algorithm used to perform a single iteration of the system. The algorithm assumes that the structures for the symbol state have been separated out into fields that will each fit into a single pixel on the framebuffer. The for all new symbols line is performed via rasterization on the graphics card. The contents of that loop are executed as a fragment program.

The matching production from the algorithm is found from the previous state. It could be a simple match, where a given symbol has only one production, or one that makes use of the state for that symbol, such as a different production once a certain global height is reached.

The matching rule is determined by the position of the symbol in the framebuffer. For the set of productions in an L-system, there is some maximum number of outputs from a single production. We call this value the degree of the system, \( d \). A block of \( dN \) fragments are drawn, where \( N \) is the number of modules resulting from the previous iteration. A fragment in the new string is able to determine its ancestor by computing \( \lfloor m/d \rfloor \) where \( m \) is the index of the \( m \)th new module. We then use the \( r \)th rule, where \( r \) is the remainder of \( m/d \).

If there is no matching rule or production, a Ø symbol is generated. All Ø symbols will be removed in the next stage of the algorithm, so adding these symbols will not change the context of neighboring symbols. Once the symbol string is sorted, the length of the string is truncated to remove all Ø symbols. The sorting is described in Section 4.5.

As an alternative to sorting out Ø symbols, an implementation could use the product length-summation from Méch and Prusinkiewicz [18] making use of an additional “render-to-vertex buffer” pass. If the symbol string becomes large, beyond the width allowed for a pbuffer, this technique will encounter major difficulty: it must find a way to pack the lines it generates into the rows of the framebuffer.

4.5 Sorting

Bitonic merge-sort is a sorting algorithm designed for parallel computation. The algorithm contains no comparison dependent operations, which makes it ideal for GPU im-
Figure 4: Sorting Network for N=8. Each stage is defined by a pair (Block-size, Sort-Direction-size). The arrows represent comparisons that occur and in which direction the two elements are ascending. The bars next to the arrows show the size of the groups: the bar to the left indicates the block of comparisons, and the bar on the right indicates the regions sorted in the same direction.

The algorithm has been adapted to work using fragment programs [6, 21]. Each stage of the algorithm is characterized by two variables, which will be inputs to the fragment program. We examine the potential of combining multiple stages into a single fragment program and show that this approach can give substantial execution time benefits on modern hardware.

The program is able to use the grouping data described in Figure 4 to determine the elements to compare and in which direction it should sort. The block-size parameter for a stage gives the size of each block of data being compared. The sort-direction parameter, likewise, gives the size of a block containing the same sort direction. Notice that the first block is always sorted ascending, thus the direction being sorted can be determined by the parity of $\left\lfloor \frac{m}{\text{sort\_direction}} \right\rfloor$. This feature, where an element’s dependencies are completely encoded in these values and its position, is what makes this sorting algorithm suitable for GPU implementation. It also means that there is no need to isolate each stage from its neighbors; the added dependencies are easily computed.

On the CPU, we generate the required stages for the sorting algorithm based on the size of the input. Figure 5 shows how the list of stages is created. From this construction, it is clear that the cost of the algorithm is $O((\log N)^2 \cdot \text{cost per stage})$. We can derive the inputs to the fragment programs by pulling from the front of the list; for the programs that perform multiple stages, we can pull multiple pairs of parameters off the list.

We recognized that the cost of switching framebuffers to perform the next pass can be a large constant cost per stage. If we try to perform multiple stages per pass, the number of framebuffer switches could be substantially reduced, and the cost of each switch could be divided over more stages.

Performing multiple stages in a single fragment program introduces some new inefficiencies. In our original formulation, each comparison was being performed twice: once for the fragment writing out the greater value and one for the lesser value. When we add an additional stage to the program, that inefficiency increases exponentially: with two stages, the comparisons on the first stage are performed four times. For three stages, the comparison is performed eight times and so on. However, since more comparisons were being made, greater parallelism could be exploited inside the program.

The results of our investigation in improving sorting times is presented in Section 5.2.

Our data representation provides some challenges to a sorting implementation. Because our state data is divided into several framebuffers, minimizing data movement during sorting is important. Instead, we create a framebuffer where each fragment contains the key data required to sort, the number representing the symbol for the product and a reference to the corresponding data being sorted. Sorting is performed on the references so that data is only moved once for each framebuffer at the end of the sorting process.

Once the sort is performed on buffer of references, we make a pass for each buffer containing the symbol data and copy it into its sorted position. This extra pass serves another purpose: we can identify Ø symbols and not copy those fragments. Using an occlusion query, the managing program will know how many valid symbols to continue processing. It is this ability that allows the framework to overcome many arbitrary limitations of earlier GPU-based procedural geometry algorithms.

4.6 Instancing and Interpretation

Attempting to implement the turtle-command based interpretation of L-systems is not practical on a pure GPU implementation. The interpretation stage of L-systems assumes a left-to-right reading of the operators, generating a path upon which instances of geometry are created. The $n$th symbol is dependent in this way on the $n-1$ sym-
create sort mapping of symbols stratifying by output for each state texture t:
  output mapped symbols from t
for each vertex attribute:
  for each type of output:
    push attributes to graphics card
    output vertices to graphics card memory
for each type of output:
  push triangles to graphics card
  output triangles to graphics card memory

Figure 6: Instancing Algorithm

symbols preceding it. This implied dependency on the "new-context" [14] of the symbol would require a costly $O(N)$ passes to propagate the data to all symbols.

Recognizing this limitation, the geometry we represent should be viewed as a procedural scene graph, where at each iteration, a cut of the graph is stored in the framebuffer. After $i$ iterations, the framebuffer contains all the graph nodes at depth $i$. Each node is considered to have a program based on the production rules that generates the next level of the scene graph. This form of procedural scene graph is implemented in PGI [13].

Our instancing operation uses a feedback loop from the fragment processor back to the vertex processor. In order to keep the algorithm purely GPU driven and avoid costly read backs, the fragment programs need to generate every part of the output mesh: the vertices, any vertex properties, and the index list for triangulation.

First, the list of symbols needs to be culled to only the set of desired symbols. This can be done easily by a fragment program that determines if the symbol should be instanced (for instance, performing a bounding volume intersection check with the view frustum). If the symbol passes the test, the input data is merely copied to the output; if not, a Ø symbol is output. A sorting pass is made over the data and the valid modules are packed at the beginning of the buffer, ready to be used for the next stage.

The symbols are sorted by type so that a single quad may be used to create the modified vertex values. Each fragment computes the symbol it is working from as $\lfloor p/N \rfloor$ where $p$ is the index of the fragment and $N$ is the size of the output per symbol (the number of vertices or triangles per instance, usually). The generation of each type of output is separated and no space in the output mesh wasted. The CPU uses occlusion queries to increment the destination offset with each pass, so outputs are appended to the generated data.

4.7 Context Sensitivity

If we modify our sort to preserve order rather than stratify the results based on symbol, we can allow the production rules to be context sensitive based on the context of the ancestor symbol. To modify the sort, we change the comparison operator to find all Ø symbols greater than all other symbols, but other symbols are sorted by their original indices in the framebuffer.

Once this modified sorting operator is used, the neighbor of an element is trivially found from its index by adding or subtracting one, followed by the same modular arithmetic used to find the ancestor symbol.

However, once the sort has been modified, the symbols are no longer in an order easily used by our instancing algorithm. We must perform an additional sort in this case to stratify the symbols by instance type.

4.8 Nondeterministic Effects

Random productions can be introduced by providing a framebuffer populated with random numbers or vectors for each pixel as input to the production program, or by using the Cg library function noise(). Conditional pruning, where logic determines if a symbol should be generated, can be supported by outputting a Ø symbol instead of a valid symbol if the symbol is discarded.

5 Results

We implemented this system using a NVIDIA GeForce 5900 Ultra FX using Cg code. It was essential for our implementation that the types of variables capable of being output by the framebuffer be able to map to the data types used for geometry representation. Thirty-two bit integer and floating point values are used for indices and vertex properties respectively, and so the video card must be capable of producing these values as outputs. The graphics card we used was capable of directly handling 32-bit floating point values; integers are supported using packing operations.

5.1 Procedural Geometry

The picture in Figure 1 is created with one symbol, only two-dimensional coordinates and no conditional pruning. This allowed us to keep the state in one four-component framebuffer. This setup allowed for interactive speeds when changing the parameters, such as the branching angle, of the system.
When we add enough data to allow for three-dimensional local coordinate systems, models such as the tree in Figure 7 are possible to generate. The application allows us to symbolically link the parameters of the operations to several slider bars; we can interactively manipulate the branching angles to generate the tree we wish.

Execution times are highly dependent on the size of the framebuffer chosen. For buffers of size 512x512, the tree takes approximately 115ms to perform an iteration and about 80ms to instance. Above 10000 instances, the iteration time because to increase rapidly. This corresponds to thresholds when the time spent sorting becomes scaled with input size. As mentioned in Section 6, these times should improve quite dramatically with true "render-to-vertex buffer" implementations.

5.2 Sorting

We use sorting to remove invalid elements and stratify outputs to perform instancing. It is a central component of the algorithm and we became very interested in improving its performance for the size of data we worked with. The primary algorithmic enhancement we investigated was performing multiple stages of the bitonic sort in a single fragment program. The source positions of all the data required for a given comparison can be derived from two parameters defining the stage and the index of the fragment being written.

We timed sorting data of different sizes to determine what performance gain there was, if any. We found that for small data sets, the overhead of setting up the required passes dominated the required time, thus there is a period of sub-linear time growth. As the data size grew, the application became fill-bound; the time spent processing fragments began to dominate. After this point, we found a nearly linear increase in time for data sets. Figure 8 shows the results for small data-set sizes. Note that the "x" scale of the graph is logarithmic. After the sizes listed, the sorting times approximately double for each doubling of the input size. Thus at $2^{16}$, the single stage takes approximately 167ms, the two-stage takes 210ms, and the three-stage program requires 555ms per sort. The bitonic sort algorithm has asymptotic running times of $O(N \log N)$.

We found that reducing the number of conditional assignments made in the sorting fragment programs lead to significant speed ups. By replacing the series of conditional assignments with an XOR operation we were able to reduce the number of conditional assignments to 1 per comparison and exploit greater instruction-level parallelism. This simple optimization lead to an approximate 30% reduction in execution time for larger data sets.
6 Conclusion

The “render-to-vertex buffer” feature is not completely supported in current drivers and hardware. Presently, returning the data to the vertex processor will involve a copy made back to the driver’s memory on the AGP bus or possibly to CPU memory. Also, the framebuffer memory and memory used for vertex data is not necessarily unified on all graphics cards, and may require a copy from the framebuffer to vertex memory even locally on the video card. These limitations have given this feedback loop the epithet “copy-to-vertex array.” We expect that this limitation is temporary and thus have written our algorithms with the expectation that “render-to-vertex buffer” will be available for future implementations of this work.

6.1 Future Work

Our implementation is largely a proof-of-concept application. The Cg we currently use is generated by hand. Creating a pre-compiler to automate this process will allow us to create more complicated procedures quickly.

We are investigating ways to resolving the packing problem of the length-summation technique used in [18]. For small L-systems it requires only $\Theta(\log_2 N)$ passes rather than the $\Theta(\log^2 N)$ passes required for a sort at every iteration. A sort will still be required to stratify symbols for instancing.

New capabilities are being introduced to graphics hardware that could make instancing or procedural geometry algorithms more efficient and practical. With faster read back speeds, hybrid CPU/GPU approaches could be used with offline rendering.

Lastly, since the work of Velho [22], it is possible that this framework could be applied to subdivision surfaces in some manner. Particle systems implemented on GPU platforms could be extended to apply instancing or further procedural effects.

References


