Predictive Lazy Amplification: Synthesis and Rendering of Massive Procedural Scenes in Real Time

Carlúcio Santos Cordeiro, Luiz Chaimowicz
Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
Belo Horizonte, Brasil
carlucio@gmail.com, chaimo@dcc.ufmg.br

***Abstract*—In this paper we propose a new paradigm for procedural modeling that enables the real time visualization of massive procedural scenes. For this, we use a combination of memory and task management with two well known procedural modeling paradigms: data amplification and lazy evaluation. Experimental results show that, in addition to obtaining performance gains through parallelism, the implemented system can generate and visualize a procedural scene far greater than the available memory in real time using only a single PC equipped with a GPU and a multicore processor.**

***Keywords*-Rendering ; Modeling and reconstruction**

I. Introduction

Procedural modeling is a very interesting approach to synthesize a large variety of textures and 3D models. In this approach, geometric data are synthesized by a set of algorithms (procedures). The use of procedural modeling techniques can help graphic artists to create complex scenes in less time than if they were completely shaped by hand using 3D modeling softwares. Procedural modeling is not intended to replace the work of artists, but to collaborate with their productivity, providing more elaborate modeling tools. The artists do not have to worry about the richness of details. They just have to deal with parameters that represent some "look and feel" of each model.

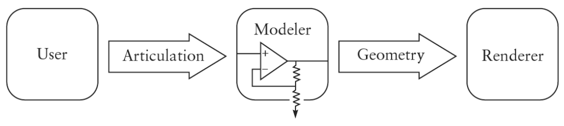
In the gaming industry, procedural modeling was extensively used in the first generations of personal computers and videogame consoles. Some classic games from mid-80s showed us that these techniques were an interesting way of dealing with the limitations of these old systems. Then, for some time, the use of procedural content generation in games was left behind, mainly due to the increased memory capacity of the gaming systems. Nevertheless, in the last few years, games are demanding massive environments with richness of details and procedural modeling is becoming an interesting approach again, not only to cope with memory constraints, but mainly to deal with the high demand for visual content in modern games.

This field presents at least three major research challenges. The first one is the main theme of procedural modeling: create procedures and algorithms that synthesize realistic models with richness details. The second challenge is controllability: with the abstraction provided by procedural models, users may lose control over certain details of the generated scenes [1]. The third challenge is managing the large amount of data that are generated by procedural models. This is a less studied problem and is the main focus of our research.

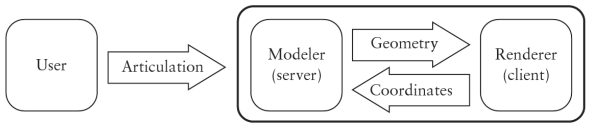
According to Hart [2], [3], applications that use procedural synthesis of geometry may be classified into two paradigms: data amplification and lazy evaluation. In a concise definition, data amplification consists in generating all the geometry of the scene in memory before the visualization process. On the other hand, in lazy evaluation, only the objects needed for a given view are generated. Data amplification can generate a lot of objects in an appropriate representation to be rendered in real time, but it is only applicable for scenes that, when amplified, fit in system memory. Lazy evaluation avoids memory problems by synthesizing data only when required, but there is no memory management and the synthesis must be done in each frame. Thus, lazy evaluation may be inadequate for real time applications since the cost of generating all the data for rendering each frame can be prohibitive. These two fundamental paradigms are discussed in more detail in sections II-A and II-B.

A simpler solution would be to generate the entire scene in an external file using data amplification and then use out-of-core visualization techniques to render the massive model. However, using this approach, the scene should be fully synthesized on a storage device before visualization. This method is less flexible and demands secondary memory which can become another obstacle. To overcome these problems we need more elaborated techniques.

In this paper we propose a paradigm for governing procedural synthesis of geometry that enables real time visualization of massive procedural scenes. We can briefly define it as a compromise between data amplification and lazy evaluation through memory and task management. The main idea is to do a "lazy data amplification" in a speculative way, predicting which models in the scene must be synthesized and maintained in memory, as well models that should be discarded. We call our new paradigm "Predictive Lazy



(a) Data amplification



(b) Lazy evaluation

Figure 1. Fundamental procedural modeling paradigms proposed by Hart [3].

Amplification". We also propose and implement a scene graph technology that employs this new approach.

This paper is organized as follows: in Section 2 we discuss relevant related work and some background in the field. Section 3 presents our proposed paradigm and scene graph technology. In section 4 we discuss some implementation issues and in section 5 we provide some experimental results that show a proof of concept of the proposed paradigm. Finally, in Section 6, we present the conclusions and possibilities for future work.

II. Background and Related Work

Most of the works in the procedural modeling field deal with the problem of proposing or improving some procedural models. Researchers have designed procedural models for various types of objects and natural phenomena. We can cite seminal works that focus on building textures [4], [5], [6], terrains [7], [8], plants and vegetation [9], [10], [11], cities and architecture elements [12], [13], [14] and seashells [15], [16]. Although our work is not directly related to any particular procedural model class, our technique allows the use of different procedural models, such as those just mentioned.

There are also works that enhance or improve fundamental procedural modeling techniques. Many of them explore the use of GPUs to speed up procedural synthesis, for example, procedural textures [17], terrains [18], parametric and subdivision surfaces [19], [20] and particle systems [21]. However, since GPUs are naturally well explored in real time rendering applications, we are not interested in using GPUs for procedural synthesis at first. Moreover, the complexity and variety of procedural modeling techniques make them hard to adapt and implement in current GPU technology, which is still more restrictive than the traditional CPU programming.

The rendering of massive models is discussed mainly in out-of-core visualization works, such as [22], [23], [24]. Most solutions use techniques such as visibility calculations, scene and memory management and parallel computing. Some systems show that interactive visualization of large volumes of data is possible using only a single home PC [25]. Our solution follow this thread. The main difference here is that geometric data is not stored and read from disk, but generated on demand by procedural modeling algorithms.

Few works discuss the data explosion problem in procedural modeling. Hart [2], [3] is one of the first to address this problem. He proposed the classification of procedural modeling systems in two fundamental paradigms that he called data amplification and lazy evaluation. He also proposed a scene graph technology called Procedural Geometric Instancing (PGI) that employs lazy evaluation. These concepts are fundamental in our work and will be discussed in the next sections.

A. Data Amplification

Applications that follow the data amplification paradigm synthesize all geometry before the rendering process. Smith [26] coined the term data amplification to explain how procedural models transform a small amount of parameters (input data) into a large amount of geometry (output data). Data amplification causes an explosion of intermediary data, due to the fact that the procedural model is converted into a geometric representation suitable for rendering. Figure 1(a) illustrates the data amplification paradigm. Observing the diagram, we can see the serial characteristic of data amplification. First the user articulates the procedural model setting its parameters. Then the modeler synthesize the intermediate representation. Finally the renderer receives the geometry from the modeler and draws the model.

B. Lazy Evaluation

The lazy evaluation paradigm avoids the storage problems that may occur in data amplification. For this, the procedural synthesis is performed on demand only when the system needs data. Unlike data amplification, which is a sequential approach, lazy evaluation follows an asynchronous architecture in a client-server fashion. The diagram in Figure 1(b) shows that the renderer maintains a direct communication with the modeler. Each time the renderer needs a model, it makes a request to the modeler that then synthesizes the data on demand.

C. Procedural Geometric Instancing

Scene graphs are spatial data structures commonly used in graphics applications and are a frequent research subject [27], [28]. Scene graphs can represent basic geometric instances, but, in general, a standard scene graph is not able to represent all types of procedural geometric instances. The

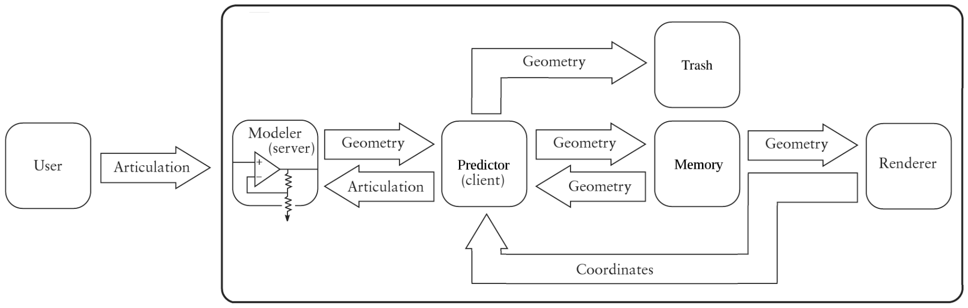


Figure 2. The Predictive Lazy Amplification paradigm proposed in this work.

scene graph technique called Procedural Geometric Instancing (PGI) is an extended type of scene graph that employs lazy evaluation to support more classes of procedural models [2]. Basically, PGI nodes are augmented with procedures that are executed whenever the model is requested, *i.e.*, whenever it is drawn on the screen.

III. Our Approach

A. Predictive Lazy Amplification

As mentioned, data amplification causes a geometric data explosion. The intermediate representation of a complex scene can become extremely large. For example, a poplar tree procedural model described by 16 KB, when evaluated yields 6.7 MB [11]. A small forest of ten thousand trees would result in about 60 GB and easily extrapolate the main memory of the best home PCs today. On the other hand, using solely lazy evaluation to synthesize, draw and drop all the models when required is an inadequate approach to be used in real time rendering. Generating a view from a complex scene can be very expensive to be done on every rendered frame.

A simple solution to visualize massive procedural scenes in real-time would be to use data amplification to generate the scene in a file and then use an out-of-core visualization algorithm for displaying it. However, this approach has a limitation imposed by the use of the secondary memory. Returning to the above example, if we want a forest with a million distinct poplar trees, the result would be a file with about 6 terabytes. To avoid these problems, we combine the two fundamental procedural modeling paradigms with memory and task management techniques in a single rendering pipeline. We call this new paradigm *Predictive Lazy Amplification*.

The central idea of predictive lazy amplification is to establish a compromise between the two well known procedural synthesis paradigms, data amplification and lazy evaluation. In a nutshell, this new paradigm consists in using a visibility prediction method to manage memory allocation and task management in the system. Models that are in the current field of view or that are likely to become visible in a near future are synthesized on demand and kept in memory as needed. Models that are in memory and have low probability of being seen are discarded and the memory used by them is freed.

Figure 2 illustrates the predictive lazy amplification paradigm. Observing the diagram, we can note that predictive lazy amplification is also asynchronous, with a client-server architecture similar to that used in lazy evaluation. But the client here is the predictor and not the renderer. During visualization, the predictor estimates what models are needed and requests them to the modeler. The modeler synthesizes data in a format suitable to be rendered. The predictor then maintains this data in memory as long as they are needed. With the required data in memory, the renderer can then render the visible scene directly. When the predictor decides that a model is no longer necessary, it discards intermediate data and liberates the memory that was being used.

B. Blowfish Scene Graph

In section II-C we briefly presented PGI (Procedural Geometric Instancing) that is an extended scene graph type capable of supporting procedural models. We also mentioned that PGI was designed with the lazy evaluation paradigm in mind. Likewise, we also propose a scene graph technology that employs our new paradigm. We call it Blowfish[1] Scene Graph (BSG). The BSG object model is ilustrated as a UML class diagram in Figure 3. All BSG components are classified as a *geometry*, a *attribute* or a *node*, and all scene graph traversals are implemented by *evaluators*. Two evaluators that are fundamental for system works are the *renderer* and the *predictor*. Next, we briefly describe the main BSG components and operations.

1) Main Operations: In PGI, an object is always instantiated when it is drawn because it uses lazy evaluation. The approach proposed here follows a different path. The instantiation is not related to rendering and runs in a separate operation. The main idea of the BSG is to decouple the

[1]The name Blowfish is inspired by the fish that has the property of inflating his body when threatened by a predator or other environmental factor.

procedural instantiation operation from the rendering and define two new operations to handle this process:

- ***inflate*:** The procedural instantiation is performed in this operation. When a scene component is “inflated”, it is synthesized from its parameters and an appropriate representation is generated for rendering. We can see this operation as applying data amplification in a scene component.
- ***shrink*:** When this operation is applied on a scene component, all data that was amplified by the inflate operation is discarded, turning the component back to its compact state. Because of its ability of inflating and shrinking we call this scene graph “Blowfish Scene Graph”.

2) Components: Another difference between PGI and BSG is that instead of a single abstract scene component, which is the PGI scene graph node, BSG defines three main abstract scene components:

- **Nodes:** Nodes are used to structure scenes. In our design, nodes are not shared among others. Shared components are geometries and attributes, which are attached to each node. Also, the node has other basic attributes like geometric transformations and a bounding volume.
- **Geometries:** Geometries are all components that can be drawn. Examples range from the most basic objects such as spheres, cylinders and cubes to more specific ones like a terrain block or a tree. For a geometry to be drawn, it must be in an amplified state (inflated), so that it will be in an appropriate representation to be rendered.
- **Attributes:** Surface details, reflection, refraction and shadows are some examples of visual effects that can be obtained using a variety of rendering techniques, including lighting models, textures, shaders, shadow algorithms and so on. Rendering techniques are implemented by attributes that can be attached to the nodes of the scene graph. When the scene graph is drawn, an attribute component present in a node indicates that the effect implemented by this attribute will be used to render this node and all its children.

IV. Implementation

To evaluate our approach, we implemented a real time rendering framework[2] employing the ideas described in the previous section. Due to performance and portability issues, we decided to implement the system using C++, OpenGL for rendering and SDL for threads and events. The main characteristics of our system are:

- **Extensibility:** Our architecture is plugin-based. New scene graph components and evaluators can be added at runtime. Each of the components mentioned above define an interface that must be implemented by each plugin. Developers can easily create new components simply by writing classes that inherits from the three abstract components in question. So far, we have developed only a minimal set of plugins, just enough to evaluate our approach. The main plugins that we have implemented are procedural terrains, rocks, trees and others like scene graph nodes, textures and shaders.
- **Parallelism:** We employ a multithreaded design. Our system uses at least three threads, one for the main loop, one for the predictor, and a third one for a modeler. Our system also allows the use of more than a single modeler, according to the number of processing cores available. The theory leads us to believe that if we have extra processing cores available, we can use them as modelers. This idea is based in using task parallelism through the manager-worker technique [29]. The predictor takes the role of manager and modelers act as workers. Thus we can accelerate the procedural synthesis by distributing the models that should be generated to the available modelers.

[2]Source code will be available at http://psygen.sourceforge.net/

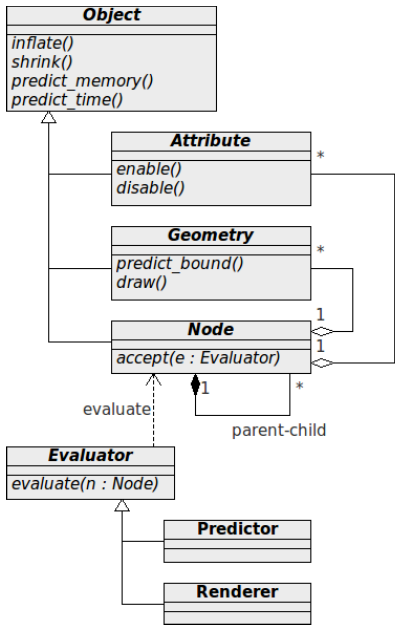


Figure 3. The Blowfish Scene Graph (BSG) object model.

We developed a quite simple visibility algorithm for the predictor. It basically consists in defining a sphere relatively larger than the view frustum, that is initially centered at the camera position. All components of the scene graph that are inside or overlaps the sphere must be amplified and all others should be shrunken. At each update the predictor determines whether the view frustum is near the limit defined by the visibility sphere. When this occurs, the sphere is again centered on the current camera position and the predictor thread starts running, traversing the scene graph and defining which components should be amplified or shrunk based on their position relative to the visibility sphere.

Finally we should emphasize that predictive lazy amplification is based on technology independent assumptions. For example, the renderer can be implemented using APIs such as OpenGL or Direct3D. The memory used to amplify models can be main memory (RAM) or video memory (GPU). Our approach also does not impose a visibility prediction method to be used by the predictor neither other techniques to be employed in the implementation.

V. Experimental Results

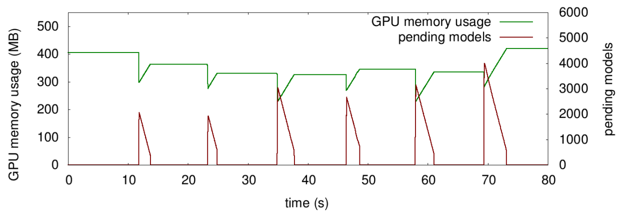
In this section we present the experimental results obtained using our implementation. This results provide an initial analysis of our paradigm and technology. For these experiments, we used a massive procedural scene composed of several terrain blocks, with trees and rocks distributed across each block. The detailed scene data is:

- Number of terrain blocks: 64.
- Number of trees across each block: random between 6000 and 7000.
- Number of rocks across each block: random between 1000 and 1500.
- Estimated scene size: 32 GB

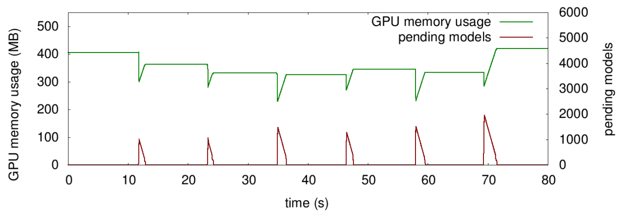
The tests consisted in collecting some data during virtual flights across the scene. In each test we varied only the number of modelers used. The scene and camera path were identical to maintain consistency among results. In each run we logged: (1) the framerate, (2) number of pending objects waiting to be amplified and (3) the used GPU memory. We used a Linux system composed of a Intel Core i7 2.666 GHz processor with 3GB of RAM and a NVIDIA GeForce 280 GTX GPU with 1 GB of video memory. Because this is a quad-core processor with SMT (Simultaneous MultiThreading), we varied the number of used modelers from one to eight.

Figure 4 shows the GPU memory usage and pending models waiting to be amplified during the execution using a different number of modelers. The main result is that the system never requires more than 500 MB of GPU memory at any time. If the scene was entirely amplified, we would need about 32 GB to store all the geometry. We may also notice that, when the number pending objects is greater than zero, there is a change in the amount of memory used, since objects were generated and discarded in those moments.

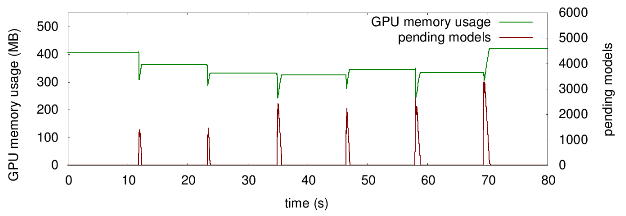
Figure 5 shows the frame rate behavior and pending models waiting to be amplified during the execution using a increasing number of modelers. In all situations, even with a large number of models waiting to be amplified, the frame rate never dropped below 30 Hz (tests were performed with vertical sync on, so the maximum obtained frame rate was 60 Hz). Also, looking at the graphs, we can see that when three or four modelers were used, the impact caused by secondary threads on frame rate was minimal. Particularly



(a) 1 modeler



(b) 4 modelers



(c) 7 modelers

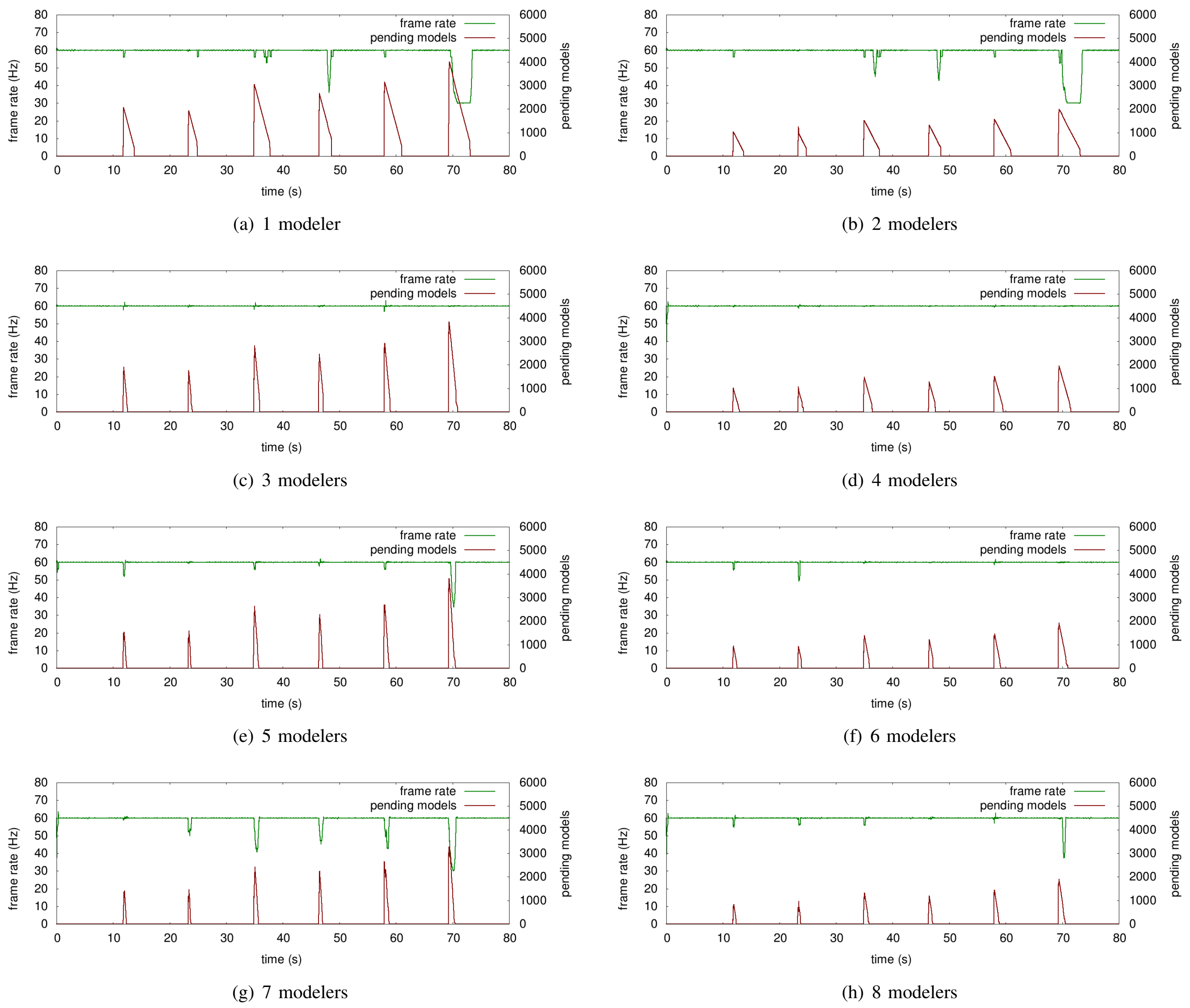
Figure 4. GPU memory used by the amplified models.

in our results, the frame rate was more stable when we used three or four modelers.

To estimate the gain due to parallelism, we measured the total time spent with amplification of models in each run. We did this by adding all the time intervals in which the number of pending models was greater than zero. Then we compared the total time for the run in which we used only one modeler with each run using more than one. This was done in a similar way to speedup computation, that refers to how much a parallel algorithm is faster than a corresponding sequential version. But in our implementation, even in the simplest case using only one modeler, the algorithm is also parallel, with three threads: the renderer, the predictor and one modeler. These results are shown in table I and in the graph of Figure 6.

Although we have not done a formal analysis of the limits of our system, we present here an informal discussion. Particularly, the limits are dictated mainly by the following factors:

- The visibility algorithm of the predictor, that establishes the set of objects that must be in inflated state.
- The size of the scene graph in shrunk state.
- The synthesis time of objects, that must be less than each predictor call.



(a) 1 modeler (b) 2 modelers (c) 3 modelers (d) 4 modelers (e) 5 modelers (f) 6 modelers (g) 7 modelers (h) 8 modelers

Figure 5. Frame rate and pending models.

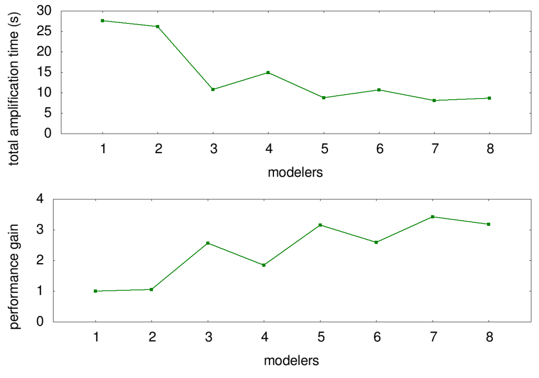


Figure 6. Total amplification time and performance gain with parallelism.

Finally, Figure 7 presents some screenshots of our system running. But we must emphasize again that the focus of our work is not the diversity of procedural models, but the performance with the massive scene is synthesized and rendered. Another important observation is that, although we use a terrain-based scene in tests, our approach allows any type of scene provided that it is structured using a BSG.

VI. Conclusion

In this paper we proposed a new procedural modeling paradigm that we called predictive lazy amplification. This paradigm can be defined as a compromise between data amplification and lazy evaluation. If we analyze memory usage, lazy evaluation can handle massive scenes while data amplification only allows the visualization of scenes that

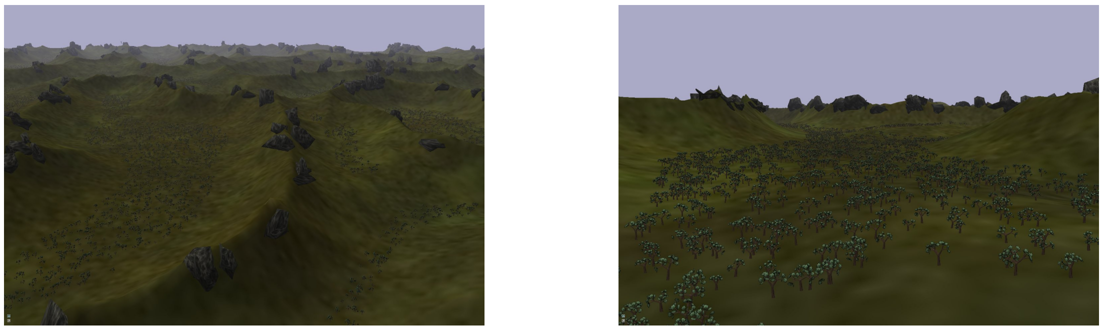


Figure 7. Screenshoots taken from our system viewing a massive procedural scene.

Table I
PERFORMANCE GAIN THROUGH MODELERS (THREADS) USED.

Number of modelers	Number of CPU threads used	total amplification time	Performance Gain
1	3	27.56	1.00
2	4	26.17	1.05
3	5	10.73	2.57
4	6	14.92	1.85
5	7	8.71	3.16
6	8	10.65	2.59
7	9	8.07	3.42
8	10	8.67	3.18

can be fully amplified in the available memory. But if we take into account the real-time rendering capability, lazy evaluation may not be an appropriate choice. Depending on the scene complexity, only the time spent on procedural synthesis can preclude a good frame rate for real time rendering.

Predictive lazy amplification can cover cases in which these two basic paradigms do not work well. We presented some tests as proof of concept. We used a scene with about 32 GB on a PC with 3 GB of RAM and 1 GB of video memory. Our implementation was able to visualize the used scene at 60 frames per second. This same scene could not be visualized using data amplification since it would not fit in available memory, or with lazy amplification, since depending on the camera position, only the synthesis of the models that appear in the field of vision would take about one second.

Our implementation makes good use of multicore processors. This approach can be interesting for applications designed to run on home PCs, as games for example. Dual-core and quad-core processors are already quite common and accessible to most users. Another detail is that we can not assume that using fewer threads, we will have better frame rate. As a result shown in this paper, the best overall performance was using five threads on a quad-core processor with SMT. One improvement that we want to do is to control the thread affinity. So we can guarantee a unique core for rendering. In particular we have implemented using SDL threads, which offers no function to control the thread affinity.

Another problem of our implementation is memory fragmentation. Whenever the models are amplified by the modelers, they allocate memory and when they are shrunk, they release the used memory. Because this is a frequent process, it causes fragmentation. We want to improve memory management and prevent fragmentation in a future work. Another paths for future work are:

- **Better visibility prediction algorithm:** The visibility prediction algorithm that we used is fairly simple. Occlusion culling and other camera data such as speed and acceleration can be used to optimize the algorithm. Another alternative is to experiment other algorithms in this category, such as PLP [25].
- **Load balancing and sorting:** As the time spent with the synthesis for each model can vary widely, the way they are distributed to the modelers can generate unbalanced load. With an estimated synthesis time of each model, we can achieve a better scheduling of the modelers. Another possibility is to sort the models to be synthesized relative to the camera.
- **Level of detail support:** Procedural models are usually multi-resolution. They can be generated at the level of detail that is desired. Then we can use more detailed models closer to the camera. The challenge here is to compute what level of detail is more appropriate for a given model and your distance to the camera.
- **Modelers running on GPU:** As we said earlier, in this first work we had no intent to exploit the GPU as a general purpose processor. Real time rendering applications are already exploiting the ability of graphics processors. However, an implementation that can detect whether the GPU is not being fully exploited and take advantage of it for modelers can be very interesting.

Acknowledgment

We would like to thank Wagner Toledo Corrêa, for all comments and suggestions.

References

[1] D. S. Ebert, K. F. Musgrave, D. Peachey, K. Perlin, and S. Worley, *Texturing & Modeling: A Procedural Approach, Third Edition*. Morgan Kaufmann, December 2002.

[2] J. C. Hart, "The object instancing paradigm for linear fractal modeling," in *Proceedings of the conference on Graphics interface '92*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992, pp. 224–231.

[3] ——, "Procedural synthesis of geometry," in *Texturing & Modeling: A Procedural Approach, Third Edition*. Morgan Kaufmann, 2002, ch. 11, pp. 305–334.

[4] K. Perlin, "An image synthesizer," in *SIGGRAPH*, P. Cole, R. Heilman, and B. A. Barsky, Eds. ACM, 1985, pp. 287–296.

[5] ——, "Improving noise," in *SIGGRAPH*, T. Appolloni, Ed. ACM, 2002, pp. 681–682.

[6] S. Worley, "A cellular texture basis function," in *SIGGRAPH*, 1996, pp. 291–294.

[7] F. K. Musgrave, C. E. Kolb, and R. S. Mace, "The synthesis and rendering of eroded fractal terrains," in *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM, 1989, pp. 41–50.

[8] K. Musgrave, "Building worlds in cyberspace," in *Computer Graphics International*, 1999, pp. 164–.

[9] P. Prusinkiewicz, "Applications of l-systems to computer imagery," in *Graph-Grammars and Their Application to Computer Science*, ser. Lecture Notes in Computer Science, H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, Eds., vol. 291. Springer, 1986, pp. 534–548.

[10] P. Prusinkiewicz and A. Lindenmayer, *The algorithmic beauty of plants*. New York, NY, USA: Springer-Verlag New York, Inc., 1990.

[11] O. Deussen, P. Hanrahan, B. Lintermann, R. Měch, M. Pharr, and P. Prusinkiewicz, "Realistic modeling and rendering of plant ecosystems," in *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM, 1998, pp. 275–286.

[12] Y. I. H. Parish and P. Mller, "Procedural modeling of cities," in *Proceedings of ACM SIGGRAPH 2001*, E. Fiume, Ed. New York, NY, USA: ACM Press, 2001, pp. 301–308.

[13] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. V. Gool, "Procedural modeling of buildings," vol. 25, no. 3, pp. 614–623, 2006.

[14] P. Müller, G. Zeng, P. Wonka, and L. V. Gool, "Image-based procedural modeling of facades," vol. 26, no. 3, 2007.

[15] D. R. Fowler, H. Meinhardt, and P. Prusinkiewicz, "Modeling seashells," in *SIGGRAPH*, J. J. Thomas, Ed. ACM, 1992, pp. 379–387.

[16] H. Meinhardt, *The algorithmic beauty of sea shells*. New York, NY, USA: Springer-Verlag New York, Inc., 1995.

[17] G. James, "Operations for hardware-accelerated procedural texture animation," in *Game Programming Gems 2*. Charles River Media, 2001, pp. 497–509.

[18] R. Geiss, "Generating complex procedural terrains using the gpu," in *GPU Gems 3*. Addison-Wesley Professional, 2007, chapter 1, pp. 7–37.

[19] C. Eisenacher, Q. Meyer, and C. Loop, "Real-time view-dependent rendering of parametric surfaces," in *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games*. New York, NY, USA: ACM, 2009, pp. 137–143.

[20] L.-J. Shiue, I. Jones, and J. Peters, "A realtime gpu subdivision kernel," in *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*. New York, NY, USA: ACM, 2005, pp. 1010–1015.

[21] P. Kipfer, M. Segal, and R. Westermann, "Uberflow: a gpu-based particle engine," in *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. New York, NY, USA: ACM, 2004, pp. 115–122.

[22] D. Kasik, "Course 4: State of the art in massive model visualization," in *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*. New York, NY, USA: ACM, 2007, p. 1.

[23] D. Kasik, A. Dietrich, E. Gobbetti, F. Marton, D. Manocha, P. Slusallek, A. Stephens, and S. E. Yoon, "Massive model visualization techniques: course notes," in *SIGGRAPH '08: ACM SIGGRAPH 2008 classes*. New York, NY, USA: ACM, 2008, pp. 1–188. [Online]. Available: http://dx.doi.org/10.1145/1401132.1401190

[24] S.-E. Yoon, P. Lindstrom, V. Pascucci, and D. Manocha, "Cache-oblivious mesh layouts," in *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*. New York, NY, USA: ACM, 2005, pp. 886–893.

[25] W. T. Corrêa, J. T. Klosowski, and C. T. Silva, "Visibility-based prefetching for interactive out-of-core rendering," in *Proceedings of PVG 2003 (6th IEEE Symposium on Parallel and Large-Data Visualization and Graphics)*, 2003, pp. 1–8.

[26] A. R. Smith, "Plants, fractals, and formal languages," *SIGGRAPH Comput. Graph.*, vol. 18, no. 3, pp. 1–10, 1984.

[27] H. Sowizral, "Scene graphs in the new millennium," *IEEE Computer Graphics and Applications*, vol. 20, no. 1, pp. 56–57, 2000.

[28] J. Döllner and K. Hinrichs, "A generalized scene graph," in *VMV*, B. Girod, G. Greiner, H. Niemann, and H.-P. Seidel, Eds. Aka GmbH, 2000, pp. 247–254.

[29] M. J. Quinn, *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group, 2003.