Formal Derivation of two Parallel Rendering Algorithms

Theoharis Theoharis
Department of Informatics
The University of Athens
Panepistimioupolis 15771, Athens, Greece
email: theotheo@di.uoa.gr

Ali E. Abdallah
The University of Reading
Department of Computer Science
Reading, RG6 6AY, U.K.
email: A.Abdallah@reading.ac.uk

Abstract *This paper presents the formal derivations of two parallel rendering algorithms from a high level specification. The initial specification of the problem is formulated as a functional program. A calculational approach is used to derive, from the original specification, two parallel algorithms expressed as networks of communicating processes in Hoare's* CSP. *Both algorithms exploit pipelined parallelism in order to achieve efficiency. The first algorithm is massively parallel but the second uses a fixed number of processing elements. The derivation is done in two stages. Firstly, a calculus of function decomposition is used in order to transform the specification into an instance of a generic parallel functional form. Secondly, the generic functional form is refined into a collection of communicating processes in* CSP *using a formal refinement framework based on previous work.*

Keywords: parallel rendering, functional programming, program transformation

1 Introduction

Computer graphics can be considered as the process which transforms a three dimensional model of a scene into a two dimensional array of pixel colours, known as the *synthetic image* or just *image*. A typical example is flight simulation, where the pilot flies through a predefined terrain and images representing the pilot's current view are rapidly generated. Another example is architectural walkthrough where the customer can explore a visual model of a proposed building. In this section we shall introduce the essential concepts of computer graphics and try to formally capture them.

Synthetic image generation (commonly called *computer graphics*) has been around for about 30 years. An increasing amount of image realism has been achieved in this period making increasing demands on the performance of the supporting computer systems. These increasing demands have even oustripped the dramatic rise in processor speeds; coupled with the need for real time performance in many graphics applications this has led to the use of parallel processing techniques, even from the very young days of the field. In fact it has been one of the first realistic applications of parallel processing, not only because of this need for computing power, but also because, as many researchers have put it, graphics offers "embarrassing opportunities for parallelism" [17, 15].

In this paper we will concentrate on the rendering stage of the computer graphics process. This stage is the most computationally expensive. We start by giving a functional formalism of the problem and then we systematically apply correctness preserving transformations rules to derive a new functional form which exhibits a high degree of implicit parallelism. Finally, the functional form is refined into a collection of communicating sequential processes described in Hoare's CSP notation. Through algebraic program transformations the final functional form can be implemented as processes with different physical configurations. Some of these are suitable for massively parallel machines, fixed pipes of processes, systolic designs and FPGAs.

2 Notation and Preliminaries

Throughout this paper, we use the functional notation and calculus developed by Bird and Meertens [6, 7] for specifying algorithmics and reasoning about them and will use the CSP notation and its calculus developed by Hoare [14, 1, 2] for specifying processes and reasoning about them. We give a brief summary of the notation and conventions used in this paper. The reader is advised to consult the above references for details.

Lists are finite sequences of values of the same type. The list concatenation operator is denoted by $+\!\!+$ and the list construction operator is denoted by $:$. The elements of an enumerated list are displayed between square brackets and separated by commas. Functions are usually defined using higher order functions or by sets of recursive equations. Function composition is denoted by $\circ$. The operator $*$ (pronounced "map") takes a function on the left and a list on the right and applies the function to each element of the list. Informally, we have:

$$f * [a_1, a_2, \cdots, a_2] = [f(a_1), f(a_2), \cdots, f(a_n)]$$

The operator / (pronounced "reduce") takes an associative binary operator on the left and a list on the right. It can be informally described as follows

$$(\oplus)/\,[a_1, a_2, \cdots, a_n] = a_1 \oplus a_2 \oplus \cdots \oplus a_n$$

2.1 Algebraic Laws

One important asset of the functional programming framework and, in particular, Bird-Meertens Formalish (BMF) is its richness in algebraic laws which allow the transformation of a program from one form to another while preserving its meaning. Here is a short list of frequently used algebraic rules which will be used later in this paper. Historically, the "promotion rules" are intended to express the idea that an operation on a compound structure can be "promoted" into its components.
map distributivity:

$$(f \circ g)* = (f*) \circ (g*)$$

map promotion:

$$f* \circ +\!\!+/ = +\!\!+/ \circ (f*)*$$

reduce promotion:

$$\oplus/ \circ +\!\!+/ = \oplus/ \circ (\oplus/)*$$

3 Formal Specification

3.1 Polygonal Model

A number of modeling schemes, such as polygonal modeling, bicubic patches and constructive solid geometry [13, 19], have been proposed over the years. We shall concentrate on the classic *polygon model* because of its widespread acceptance, simplicity and maturity. In this model objects are represented as sets of 3D polygons; each object is typically defined in its own *object coordinate system*. In order to define the polygonal model, we first need to capture some basic concepts. A colour is encoded as a number in some colour system such as RGB or CMY [13, 19]:

$$colour == num$$

Having defined a two dimensional (2D) and a three dimensional (3D) point as:

$$\begin{array}{lll} point2d & == & (num, num) \\ point3d & == & (num, num, num) \end{array}$$

we can build up the definition of a polygonal model:

$$\begin{array}{lll} vertex & == & (point3d, colour) \\ polygon & == & [vertex] \\ object & == & [polygon] \\ model & == & [object] \end{array}$$

A graphics model is thus a list of polygonal objects. Each polygon is a list of vertices and each vertex is a 3D point and an associated colour which has been determined by a shading model such as Gouraud or Phong [18, 19]. The above abstraction of a vertex coincides with the Gouraud model. The colour of each point in the teapot image is derived from the colour of the vertices of the polygon in which it lies in the polygonal model of the teapot. This is achieved through a given interpolation function *icolour*:

$$icolour :: polygon \rightarrow point2d \rightarrow colour$$

Similarly, the depth (from a viewing point) of each point in the teapot can be derived from the depth of the vertices of the polygon in which it lies.

$$idepth :: polygon \rightarrow point2d \rightarrow num$$

We use *point2d* in the definitions of *icolour* and *idepth* since the point is completely determined from the knowledge that it is coplanar with the polygon vertices. Figure 1 shows the polygonal model of a teapot and figure 2 shows the final image of the teapot from a certain viewpoint after going through the computer graphics process.

3.2 Rendering

The rendering stage can be seen as a function which takes a list of projected polygons and a background image, say *bkg*, as inputs and produces a new image, after rendering its input polygons, as output. An image is a set of pixels. Each pixel is defined as:

$$pixel == (point2d, colour, depth)$$

The initial image either contains the initial values (background colour, maximum depth) or is a partly rendered image; this definition allows us to decouple the rendering of one polygon from that of another, thus facilitating parallel processing. The *renderimage* function can be defined using a simpler function, *rendpix*, which solves the rendering problem for a single pixel:

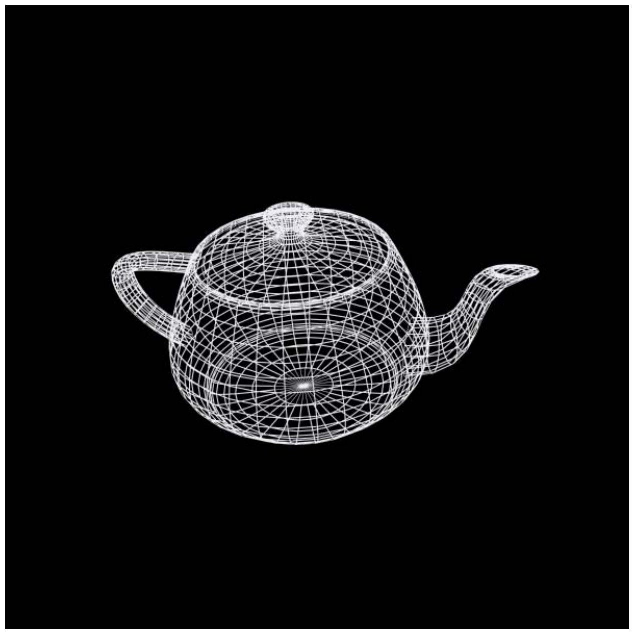

Figure 1: Polygonal model of a teapot

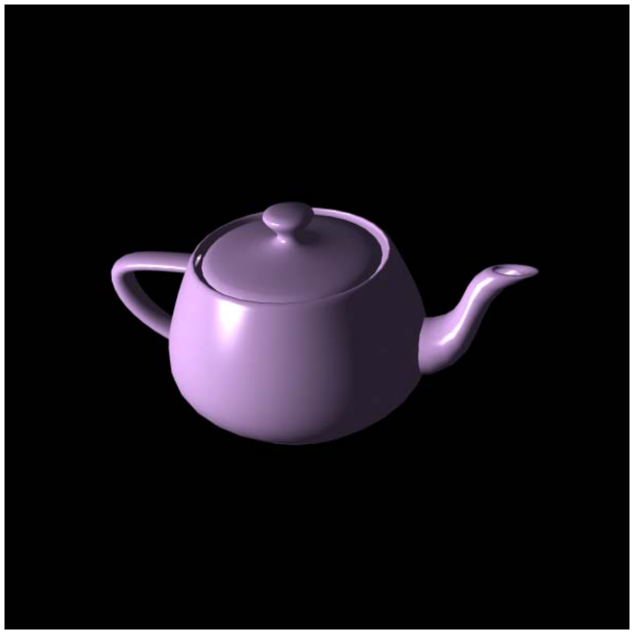

Figure 2: Rendered image of the teapot from a viewpoint

$$\begin{array}{l} renderimage :: model \to [pixel] \to [pixel]; \\ renderimage\ m\ bkg \quad = \quad (rendpix\ m) * bkg \end{array}$$

The *rendpix* function can be defined as follows:

$$\begin{array}{lll} rendpix :: model \to pixel \to pixel; && \\ rendpix\ []\ x & = & x \\ rendpix\ (g\colon gs)\ x = & & rendstep\ g\ (rendpix\ gs\ x) \end{array}$$

The above recursive definition leaves a pixel unchanged if the poygon list is empty (stopping case) otherwise it updates the value calculated for he tail of the polygon list by the head polygon using another function which can easily be defined as:

$$\begin{array}{l} rendstep :: polygon \to pixel \to pixel; \\ rendstep\ g\ (p, c, d) \\ = \quad (p, c, d),\ \text{if}\ \ p\ \text{outside}\ g\ \vee\ d < idepth\ g\ p \\ = \quad (p, icolour\ g\ p, idepth\ g\ p), \quad \text{otherwise} \end{array}$$

where *icolour* and *idepth* are functions which calculate by interpolation the colour and depth values of polygon g at pixel p respectively, from the colour and depth values at the vertices of the polygon:

4 Derivation of a Massively Parallel Solution

By applying the *tail recursion unrolling* rule, *rendpix* can be described as a composition of several functions; each of which is an instance of *rendstep* that deals with a particular polygon from the list m.

$$rendpix\ m = (\circ)/\,(rendstep * m)$$

In other words, assuming the model m consists of a list of, say n, polygons $[g_1, g_2, \ldots, g_n]$, then *rendpix m* can be expressed as a composition of n functions:

$$rendstep\ g_1 \circ rendstep\ g_2 \ldots \circ rendstep\ g_n$$

Now by applying the distributivity law of *map* over function composition, the rendering of a whole image, $(rendpix\ m)*$ can be derived as a composition of n functions:

$$(rendpix\ m)* = (\circ)/\,((map \circ rendstep) * m)$$

That is, $(rendpix\ [g_1, g_2, \ldots, g_n])*$ is:

$$(rendstep\, g_1) * \circ (rendstep\, g_2) * .. \circ (rendstep\, g_n) *$$

The composition of functions can be realized in CSP as piping of processes. Hence, the above form can be efficiently implemented as a pipe of n processes. Each process in the pipe, $MAP(rendstep\ g)$, deals with a particular polygon from the polygonal list m. It repeatedly inputs a pixel from its left neighbour, update the pixel value (colour and depth) by taking into account the polygon maintained by the process, and outputs the new pixel value to its right neighbour. The whole network is depicted in Fig. 3 and can be consisely expressed as:

$$(\gg)/\,((MAP \circ rendstep) * (reverse\ m))$$

For any function f, the pipe process $MAP(f)$ refines the function $f*$. By unfolding the CSP definition of the process MAP, the behaviour of each process in the pipe can be synthesized as follows:

$$\begin{array}{lll} (MAP \circ rendstep)\ (g) && \\ = \quad MAP(rendstep\ g) && \\ = \quad \mu Z \bullet \ (& ?\text{“}eot\text{”} \to !eot \to SKIP & \\ & \mid & \\ & ?x \to !(rendstep\ g\ x) \to Z) & \end{array}$$

The above solution effectively places the responsibility for rendering one polygon on each pipeline stage. Pixels flow through the pipeline and take their final value upon exit. This is a massively parallel algorithm. Assuming that the image to be rendered has k pixels and the model m has n polygons, the starting sequential algorithm requires $O(n \times k)$ computational steps but the pipelined version requires only $O(n + k)$ computational steps. We have thus arrived at the architecture proposed by Cohen and Demetrescu [8].

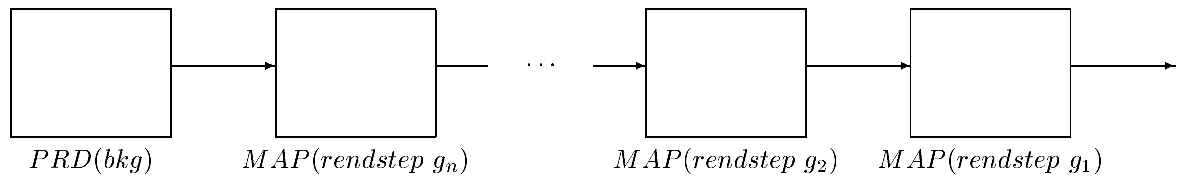

Figure 3: Rendering of the background image *bkimg* by successive polygons

5 Transformation to a fixed length pipe of processes

In practice, the number of processing elements in a parallel machine is much smaller than n, the number of polygons in the graphics model. We will show that using algebraic transformation, the massively parallel algorithm given in the previous Section can be transformed to an efficient pipelined algorithm with a given fixed length, say p. Consider a partition function *parts p* which partitions the list of polygons m into exactly p groups of consecutive elements. The function *parts* can be specified as follows:

$$\begin{array}{l} parts :: num \to [A] \to [[A]] \\ +\!\!+/\ (parts\ p\ m) = m \end{array}$$

That is: $parts\ p\ m = [m_1, m_2, .., m_p]$ and

$$m_1 +\!\!+ m_2 +\!\!+ .. +\!\!+ m_p = m$$

We have

$$rendpix\ m = (\circ)/\,(rendstep * m)$$

to transform this to a composition of p functions, we reason as follows

$$\begin{array}{ll} & rendpix\ m \\ & \quad \{\text{def. of} m\} \\ = & rendpix\ (m_1 +\!\!+ m_2 .. +\!\!+ m_p) \\ & \quad \{\text{def. of } rendpix\} \\ = & (\circ)/\,(rendstep * (m_1 +\!\!+ m_2 .. +\!\!+ m_p)) \\ & \quad \{\text{distributivity of } * \text{ over } +\!\!+\} \\ = & (\circ)/\,((rendstep * m_1) +\!\!+ .. +\!\!+ (rendstep * m_p)) \\ & \quad \{\text{reduction promotion}\} \\ = & ((\circ)/\,(rendstep * m_1)) \circ ..((\circ)/\,(rendstep * m_p)) \\ & \quad \{\text{def. of } rendpix\} \\ = & (rendpix\ m_1) \circ .. \circ (rendpix\ m_p) \\ & \quad \{\text{def. of } (\circ)/\} \\ = & (\circ)/\,[rendpix\ m_1, \ldots, rendpix\ m_p] \\ & \quad \{\text{def. of} *\} \\ = & (\circ)/\,(rendpix * [m_1, m_2, .., m_p]) \\ & \quad \{\text{def. of } parts\ p\} \\ = & (\circ)/\,(rendpix * (parts\ p\ m)) \end{array}$$

We can generalize this to *renderimage* by appealing to the distributivity law of *map* over function composition, hence, we reason as follows

$$\begin{array}{ll} & renderimage\ m \\ & \quad \{\text{def. of } renderimage\} \\ = & (rendpix\ m) * \\ & \quad \{\text{previous result of } rendpix\ m\} \\ = & ((\circ)/\,(rendpix * (parts\ p\ m))) * \\ & \quad \{\text{unfolding definitions}\} \\ = & ((rendpix\ m_1) \circ .. \circ (rendpix\ m_p)) * \\ & \quad \{\text{distributivity of } * \text{ over} \circ\} \\ = & ((rendpix\ m_1) *) \circ .. \circ ((rendpix\ m_p) *) \\ & \quad \{\text{def. of } renderimage\} \\ = & (renderimage\ m_1) \circ .. \circ (renderimage\ m_p) \\ & \quad \{\text{folding definitions}\} \\ = & (\circ)/\,(renderimage * (parts\ p\ m)) \end{array}$$

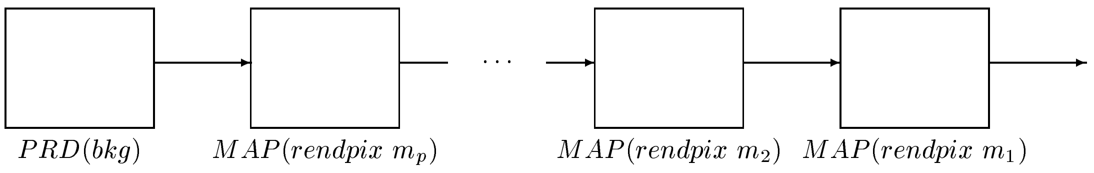

Figure 4: Fixed length pipeline for image rendering

this composition of functions can be systematically transformed into a pipelined network of p communicating processes as illustrated in Fig. 4; each stage in the pipeline is an instance of a single pipe process which refines the function $(rendpix\ gs)*$. In other words, this process takes a background image on its input channel, one pixel at a time, and produces the rendering of that image, one pixel at a time, according to the partition of the polygons it is holding. We have,

$$renderimage\ gs = (rendpix\ gs)*$$

as we have seen, this function can be refined into the process $MAP(rendpix\ gs)$. Therefore, the whole algorithmic expression can be transformed into the following pipe:

$$(\gg)/\,((MAP \circ rendpix) * (reverse\ (parts\ p\ m))$$

By unfolding the CSP definition of the process MAP, the behaviour of each process in the pipe can be synthesized as follows:

$$\begin{array}{ll} \multicolumn{2}{l}{(MAP \circ rendpix)\ (gs)} \\ = & MAP(rendpix\ gs) \\ = & \mu Z \bullet \ (\ \ ?\text{``}eot\text{''} \rightarrow !eot \rightarrow SKIP \\ & \qquad\qquad | \\ & \qquad\qquad ?x \rightarrow !(rendpix\ gs\ x) \rightarrow Z) \end{array}$$

6 Conclusion

Starting from a formal functional specification of the computationally expensive graphics rendering phase, we have derived using strict mathematical transformations, two parallel algorithms. Both algorithms exploit pipelined parallelism in order to achieve efficiency. The first algorithm is massively parallel but the second uses a fixed number of processing elements. Due to the nature of the transformations, we can ensure that the parallel implementations satisfy the original specification and we can also reason about them using well known mathematical properties. Apart from providing a correctness proof and semantics consolidation, this method is very useful as a concise and clear communication medium for algorithm designers and engineers. We plan to use the same techniques to derive other parallel implementations, with different physical process configurations such as trees and meshes, from the original specification of this problem.

The transformational approach used in this paper for deriving reconfigurable parallel algorithms is based on earlier work by the authors [1, 2, 3, 4]. It has benefited from related work on transformational programming and parallelization by several colleagues [6, 7, 9]. Related work on formal methods for describing a framework for the specification of modular graphics systems in Z appeared in [5, 11]. There is also a formal Z specification of a small example from GKS in [12]. The methods used are non-procedural but parallelism is not discussed. Parallel rendering algorithms are discussed in [8, 10, 15, 16, 17].

Acknowledgements

The authors would like to thank *The British Council* for financially supporting their joint research programme between The University of Reading and The University of Athens. The second author is supported by The University of Athens research grant 70/4/4151.

References

[1] A. E. Abdallah, Derivation of Parallel Algorithms from Functional Specifications to *CSP* Processes, in: Bernhard Möller, ed., *Mathematics of Program Construction*, LNCS **947**, (Springer Verlag, 1995) 67-96.

[2] A. E. Abdallah, Synthesis of Massively Pipelined Algorithms for List Manipulation, in L. Bouge and P. Fraigniaud and A. Mignotte and Y. Robert (eds), Proceedings of the *European Conference on Parallel Processing, EuroPar'96*, LNCS **1024**, (Springer Verlag, 1996), pp 911-920.

[3] A. E. Abdallah, and T. Theoharis, Synthesis of Massively Pipelined Algorithms from Recursive Functional Programs, in: K. Li, T.S. Abdelrahman, E. Luque, eds., Proceedings of the Eighth IASTED International Conference on Parallel and Distributed Computing and Systems, Chicago, USA, (IASTED/ACTA press, October 1996), 500-504.

[4] A. E. Abdallah, and T. Theoharis, Derivation of Efficient Parallel Algorithms on a Ring of Processors, in: E. Luque, eds., Proceedings of the European Conference on Parallel and Distributed Computing and Systems, Barcelona, Spain, (IASTED/ACTA press, June 1997), 118-125.

[5] D.B. Arnold, D.A. Duce, G.J. Reynolds, An Approach to the Formal Specification of Configurable Models of Graphics Systems, *Eurographics '87 Conference proceedings*, (North-Holland, 1987), pp. 439-463.

[6] R. S. Bird, An Introduction to the Theory of Lists, in M. Broy, ed., *Logic of Programming and Calculi of Discreet Design*, (Springer, Berlin, 1987) 3-42.

[7] R. S. Bird, and P. Wadler, *Introduction to Functional Programming*, (Prentice-Hall, 1988).

[8] D. Cohen, and S. Demetrescu, *A VLSI Approach to Computer Image Generation*, Information Sciences Institute, (University of Southern California, 1981).

[9] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation* (Pitman, 1989).

[10] M. Cox, and P. Hanrahan, A Distributed Snooping Algorithm for Pixel Merging, *IEEE Parallel and Distributed Technology* Summer 1994, 30-36.

[11] D.A. Duce, F. Paterno, A Formal Specification of a Graphics System in the Framework of the Computer Graphics Reference Model, *Computer Graphics Forum*, **12(1)**, (Springer, 1993), 3-20.

[12] D.A.Duce, E.V.C. Fielding and L.S. Marshall, Formal Specification of a Small Example from GKS, *ACM Transactions on Graphics*, **7(3)**, (ACM press 1988), 180-197.

[13] J. Foley et al, *Introduction to Computer Graphics* (Addison Wesley, 1994).

[14] C. A. R. Hoare, *Communicating Sequential Processes.* (Prentice-Hall, 1985).

[15] T.Y. Lee, et al, Image Composition Schemes for Sort-Last Polygon Rendering on 2D Mesh Multicomputers, *IEEE Transactions on Visualization and Computer Graphics* **2** (3) (1996) 202-217.

[16] S. Molnar, J. Eyles, J. Poulton, PixelFlow: High Speed Rendering Using Image Composition, *SIGGRAPH 1992*, (ACM Press, 1992), pp.231-248.

[17] T. Theoharis, *Algorithms for Parallel Polygon Rendering*, LNCS **373**, (Springer Verlag, 1989).

[18] A.Watt, and M. Watt, *Advanced Animation and Rendering Techniques*, (Addison Wesley, 1992).

[19] A. Watt, and F. Policarpo *The Computer Image*, (Addison Wesley, 1998).