A Functional View of Parallel Computer Graphics

Ali E. Abdallah
Centre of Applied Formal methods
School of computing
South Bank University
London, SE1 0AA, UK
A.Abdallah@sbu.ac.uk

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

Abstract

The main purpose of this paper is to present a functional view of the fundamentals of the computer graphics process based on the classic polygonal model. There are several advantages for adopting such an approach. Firstly, the functional view is a natural abstraction of the problem. Secondly, many well known computer graphics optimization techniques can be directly obtained from the original specification by applying general and well understood transformational programming algebraic laws on functional expressions. Thirdly, a number of highly parallel implementations suited for various parallel architectures can be derived from the initial specification by a systematic application of general transformation strategies for parallelizing functional programs.

1 Introduction

Synthetic image generation (commonly called *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.

Computer graphics has been studied for about 30 years and has lately witnessed a period of explossive growth. An increasing amount of image realism has been achieved in this period making increasing demands on the performance of the supporting computer systems. Its immediate relenance to the human computer interface as well as a large number of important applications, from the film industry to computer aided design, have fuelled a great influx of funds and new products into the field. The annual Siggraph conference is widely regarded as the largest conference in the world. However this dramatic growth has taken its tall by creating chaos in the terminology used and the semantics of computer graphics operations. It is an area which is in great need of formalism. Some attempts to this end have taken place in the past [5, 12] but they were mainly concerned with the graphics standards of the time which soon became obsolete and de-facto standards were established. Here we attempt to put a stone in the direction of formalising some core elements of these de-facto standard operations. We do dot claim to have a complete formal framework for computer graphics; far from it. But we do hope to point in the right direction for this large, but necessary, task.

The increasing demands on performance have constantly 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 [17, 15]. For this reason we

0-7695-1165-1/01 $10.00 © 2001 IEEE

pay special attention to the issue of parallelism which is at the heart of all high performance graphics systems.

Providing a functional programming framework for Computer Graphics has several attractions. First it provides a natural abstraction (clear, concise specification) to the problem. Second the functional specification provides an instant prototype. Third it is possible to apply gereral transformational programming rules to transform the functional specification into more efficient versions. In so doing many well known graphics optimisation techniques (e.g. clipping, culling) can be formally justified and understood. Fourth it is possible to derive parallel versions of these algorithms by applying well known functional programming parallelization methodologies (e.g. skeleton, annotation technique [9, 3]). Finally it is possible to formally link a family of sequential and parallel algorithms by understanding how they can be derived from a common functional specification.

In this paper we shall introduce the essential concepts of computer graphics and formally capture them in a functional notation [7]. 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 [14]. 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, 3] 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.

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, ?, ?, ?], 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, ?, ?, ?]:

$$colour == num$$

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

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

we can build up the definition of a polygonal model:

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

Some discussion is in order at this point. A polygon is defined as a list of vertices: it is necessary to impose the order of a list on the vertices because this order is used in the normal vector and the point-inside-polygon calculations. In the first case the order is required in order to produce a vector which points in the correct "outward" direction and in the second in order to unambiguously define the interior of the polygon. An object could be defined as a set of polygons because their order does not matter in this case and repetitions should not be allowed. Equally a model could be defined as set or bag of objects. An object may appear multiple times within a model (e.g. a car) but normally not at the same location. If we therefore include the transformation that places each instance of the object in the data structure then we can use a set since instances will be differentiated by this transformation (known as the modelling transformation in computer graphics); otherwise we must use a bag to allow for repetitions of an object within the model. However in this paper, in order to simplify the manipulation and the notation required, we shall use lists as specified above.

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 [?, ?]. 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 the point 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

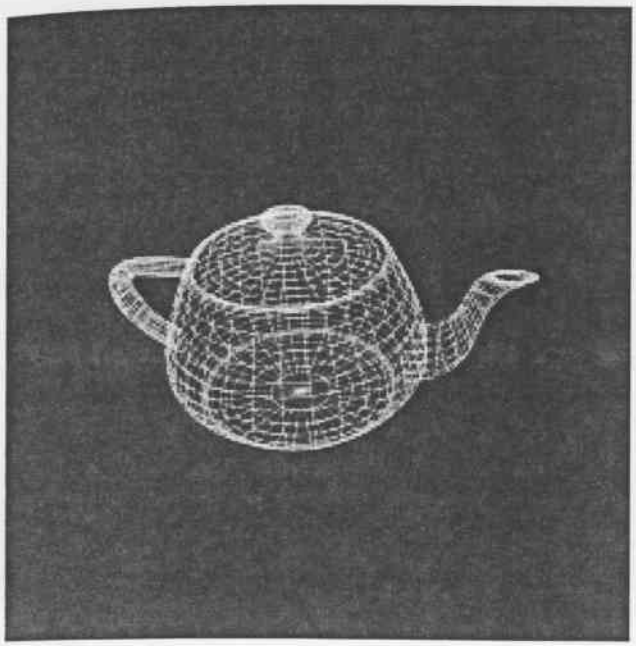

Figure 1: Polygonal model of a teapot

Figure 2: Rendered image of the teapot from a viewpoint

polygons, as output. An image can be defined as a set of pixels; we shall use a list structure here, however, for a concrete representation. 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:

$$\begin{aligned} &renderimage :: [polygon] \rightarrow [pixel] \rightarrow [pixel]; \\ &renderimage\ m\ bkg \quad = \quad (rendpix\ m) * bkg \end{aligned}$$

The *rendpix* function can be defined as follows:

$$\begin{aligned} &rendpix :: [polygon] \rightarrow pixel \rightarrow pixel; \\ &rendpix\ []\ x \qquad = \quad x \\ &rendpix\ (g{:}\ gs)\ x = \quad rendstep\ g\ (rendpix\ gs\ x) \end{aligned}$$

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:

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

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.

3.3 Animation

In practical graphics applications such as computer animation, the graphics processing pipeline is usually driven by successively

changing viewing parameters. To generate real time animation, the graphics application should be able to produce 30 consecutive *frames* (images) per second. To capture the whole animation process, we define the function *animate* as follows:

$$
\begin{array}{l}
frame == [pixel] \\
animate :: model \rightarrow [viewingparms] \rightarrow [frame] \\
mkframe :: model \rightarrow viewingparms \rightarrow frame \\
animate\ m \quad = \quad (mkframe\ m)* \\
mkframe\ m\ v \quad = \quad renderimage\ (geo\ v\ m)\ bkg
\end{array}
$$

Given a static model m as a list of polygonal objects, the function *animate* takes a list of viewing parameters (triplets consisting of viewing position, view direction and up vector) and produces a list of frames by applying the function $mkframe\ m$ for each viewing parameter. In turn, the function $mkframe\ m$ takes a viewing parameter, performs a geometric transformation on the model resulting in a list of polygons and renders these using *renderimage*.

The geometric transformation captured by the function *geo* first changes the coordinate system to one which is centered at the viewpoint and keeps from the list of objects those polygons that are "front faces" via the culling operation and lie within a "viewing pyramid" via the clipping operation.

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:

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

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].

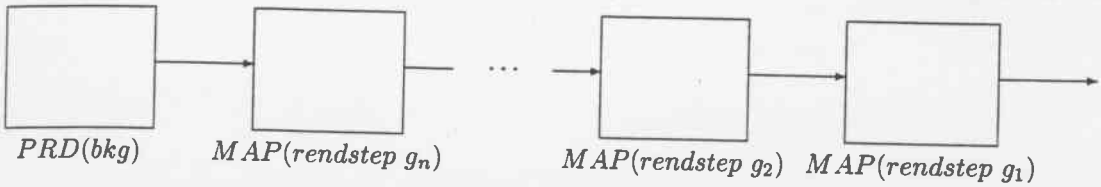

Figure 3: Rendering of the background image *bkimg* by successive polygons

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:

$$parts :: num \to [A] \to [[A]]$$
$$+\!\!+/\,(parts\ p\ m) = m$$

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

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

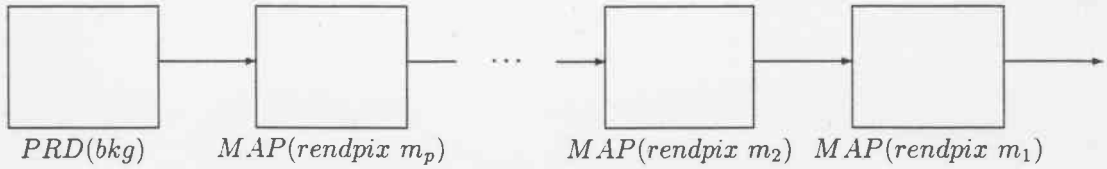

Figure 4: Fixed length pipeline for image rendering

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:

$$\begin{array}{ll} & (MAP \circ rendpix)\ (gs) \\ = & MAP(rendpix\ gs) \\ = & \mu Z \bullet (\ ?\text{“}eot\text{”} \rightarrow !eot \rightarrow SKIP \\ & \qquad\qquad \mid \\ & \qquad\qquad ?x \rightarrow !(rendpix\ gs\ x) \rightarrow Z) \end{array}$$

6 Related Work and Conclusion

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 researchers [3, 6, 7, 11, 9]. Related work on formal methods for describing a framework for the specification of modular graphics systems in Z appeared in [5, 12]. The methods used are non-procedural but parallelism is not discussed. Parallel rendering algorithms are discussed in [10, 15, 16, 17].

We provided an introductory functional description of the fundamental computer graphics operations that have become de-facto standards. Although incomplete, this can serve as a starting point for a formal framework for computer graphics. 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.

Acknowledgements

The authors would like to thank the cultural exchange programme by The British Council of Athens and the Greek Department of Education for financially supporting their joint research.

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, Functional Process Modelling. In K Hammond and G. Michealson (eds), *Research Directions in Parallel Functional Programming.* (Springer Verlag, October 1999). pp339-361.

[4] A. E. Abdallah, G. Simiakakis, and T. Theoharis, Formal development of a reconfigurable tool for parallel DNA matching. Proceedings of the 7th IEEE International Conference on Electronics Circuits and Systems, Jounieh, Lebanon (IEEE Computer Society Press, 2000), 268-272.

[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, *Introduction to Functional Programming using Haskell*, (Prentice-Hall, 1998).

[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] J. Darlington, A. J. Field, P.G. Harrison, and P.H.J. Kelly, Q. Wu, and R.L. While, Parallel Programming Using Skeletons Functions, in *PARLE93, Parallel Architectures and Languages Europe* LNCS , (Springer-Verlag, 93).

[12] 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.

[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).