3

Point-driven Generation of Images from a Hierarchical Data Structure

Dirk de Jong, Paul van Slobbe, and Marinus van Splunter

In this paper, a system is described which renders an image from a hierarchical data structure in a point-driven way. The data structure allows dynamic color mapping and arbitrary affine transformations of objects with respect to their parent coordinate system. The point driven method allows for easy VLSI implementation, efficient use of memory and exploitation of parallelism.

CR Categories and Subject Descriptors:
I.3.1 *[Computer Graphics]* : Hardware Architecture — *Raster display devices*
I.3.3 *[Computer Graphics]* : Picture/Image Generation — *Display Algorithms*
I.3.5 *[Computer Graphics]* : Computational Geometry and Object Modeling —
E.1 *[Data]* : Data Structures —
F.2.2 *[Analysis of Algorithms and Problem Complexity]* : Nonnumerical Algorithms and Problems — *Computations on discrete structures; Geometrical problems and computations*

1. Introduction

During the last 20 years, fast developments in the area of computer graphics have taken place, developments concerning more efficient and powerful algorithms and implementation of these in dedicated hardware. Although a standard [PHIGS] has been developed in which the use of hierarchically defined objects plays a major role, little effort has been spent on the development of dedicated hardware capable of exploiting this hierarchy. Exceptions are the VS8000 workstation from DEC and the PS300 from E&S.

Hierarchical modeling is a powerful and compact way of designing objects. With this idea in mind special hardware has been developed to generate a 2-dimensional image from a hierarchical data structure. In contrast with existing approaches, for each pixel independently a hierarchical data structure is scanned using the corresponding world coordinates. In this way the raster image is not built up object by object into the picture memory but generated point by point. The data structure comprises two object types:

primitive and composed. The primitive objects are defined by a list of Bézier curves. A point containment test for these primitives, and an implementation of this test in dedicated hardware was designed by Corthout [COJO]. The composed objects contain color information and transformation information, to relate the local coordinate system of the object to the viewing coordinate system. Furthermore the description contains a bounding box representing the smallest rectangular region inside which the object is contained and two pointers to define the relations between the objects

The point-driven approach which was followed in this design offers a number of advantages such as easy implementation in VLSI. The classical approach can be outlined as follows:

a transformation

b clipping

c preprocessing (linearization, sorting)

d filling of spans

The last three steps can be omitted by applying a point-driven approach. The point-driven approach holds that for each point independently it is investigated which object it is part of. The test is repeated for every point within the viewport. It is obvious that the clipping problem is omitted because all the points are in the viewport. The last step does not exist either in this approach since we don't have to generate the inside area from a contour, but only have to test if a point is inside a contour. For the same reason also step c becomes superfluous. Besides the computational simplicity (see also [CHAZ]) some additional advantages can be mentioned.

For interaction purposes it is often necessary to identify an object a user is pointing at with some device. With an object-driven method it can be very difficult or impossible to display a set of cyclically ordered objects. It is shown in Appendix B that the point-driven approach provides a very simple solution. A general way of anti-aliasing can be applied without extra memory usage. Two other circumstances in which the algorithm is more efficient than its object driven counterpart is dynamic color mapping and generating data in random order. Finally since no preprocessing is done, no intermediate memory is required for this stage. A disadvantage of the point-driven method is that it is slower than the object-driven method when implemented on a single general purpose processor. However, the algorithms are easier to implement in dedicated hardware and a further improvement of speed is possible by a virtually unlimited amount, through the application of image space parallelism.

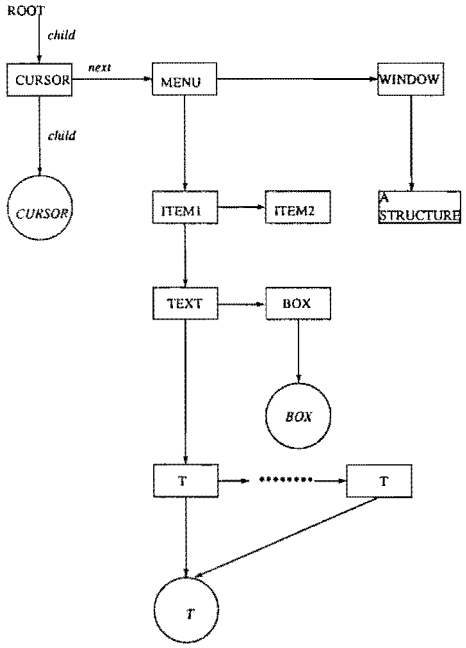

Figure 1: Example: Implementation of a window manager in the hierarchical data structure.

2. The Data Structure

The data structure will be explained with an example which may also be an application. In Figure 1 an example is shown of a data structure which could occur in a window manager environment. The structure consists of primitive- and composed objects. The primitive objects describe shapes, in this case Bézier shapes which are composed of a list of Bézier curves. The composed objects are connected to a group of more simple sub objects. The main structure is determined by the two pointers child and next.

Child is the pointer to the first sub-object and next points to the next object. The priority in which objects are displayed depends on the order in which they appear in the graph. In this example the object cursor has a higher priority than the object menu and the object window. In Figure 2 the contents of the composed objects are shown in some more detail. The data can be divided in four main blocks:

1 Structural information(the pointers child and next) explained above

2 Surface information (Color and dynamic color mapping)

3 Transformation information regarding the position and orientation of objects

4 Bounding Box information.

The color of an object belongs to the surface information. The color may be defined as an RGB value but also a dynamic color may be applied (specified by flag S). In this case the color is defined as a structure. In both cases the color of an object can be inherited by its

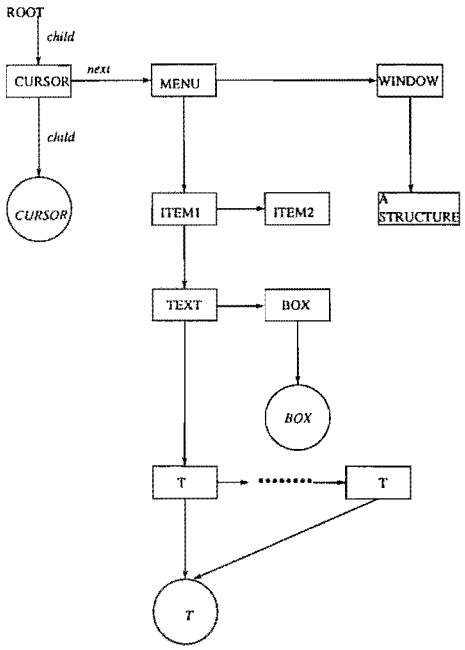

Figure 1: Example: Implementation of a window manager in the hierarchical data structure.

2. The Data Structure

The data structure will be explained with an example which may also be an application. In Figure 1 an example is shown of a data structure which could occur in a window manager environment. The structure consists of primitive- and composed objects. The primitive objects describe shapes, in this case Bézier shapes which are composed of a list of Bézier curves. The composed objects are connected to a group of more simple sub objects. The main structure is determined by the two pointers child and next.

Child is the pointer to the first sub-object and next points to the next object. The priority in which objects are displayed depends on the order in which they appear in the graph. In this example the object cursor has a higher priority than the object menu and the object window. In Figure 2 the contents of the composed objects are shown in some more detail. The data can be divided in four main blocks:

1 Structural information(the pointers child and next) explained above

2 Surface information (Color and dynamic color mapping)

3 Transformation information regarding the position and orientation of objects

4 Bounding Box information.

The color of an object belongs to the surface information. The color may be defined as an RGB value but also a dynamic color may be applied (specified by flag S). In this case the color is defined as a structure. In both cases the color of an object can be inherited by its

Structural:	Child Next
Color:	Color (RGB or a reference) Flags I,A,S
Transformation:	Translation {Tx,Ty} Matrix {Mxx,Mxy,Myx,Myy}
Bounding Box:	Xmin,Ymin Xmax,Ymax

Figure 2: Overview of data contained by a composed object.

subobjects, but it can also be overwritten. The behavior depends on two additional flags (I and A) and will be described in more detail in the next paragraph.

The transformation information describes the orientation and position of an object relative to its parent coordinate system. Currently affine transformations are allowed.

Some extra information is added to the data structure in order to speed-up the rendering process. The bounding box describes the smallest rectangle in which an object fits. The bounding box is described in the parent coordinate system.

3. A Description of the Algorithm

A compact description of the algorithm is shown in Appendix A. The function Com_Inside() determines if a point p is contained in a composed object o. Another function not further exposed here is Bez_Inside() which determines if a point p is contained in a Bézier shape which is pointed at by o->child. A coordinate transform is done by the function Transform(). In order to implement top down and bottom up coloring, the parameters c and i are passed to Com_Inside(). A color can be inherited from an object which is defined higher in the hierarchy, via parameter c. Whether the color is inherited or not, depends on the value of the flag i and the value of the local flag o->a.

Texture mapping is enabled with the parameters s and ps. If a texture is mapped on an object, the color of the object is interpreted as a reference to another object. The final color is determined by a recursive call to that object. With a slightly more complicated algorithm it is possible to apply a circular depth ordering. The algorithm outlined above is repeated for every point on the screen. A viewing transformation may take place between the display coordinates of a point and the coordinates with which the algorithm is called (See Figure 3).

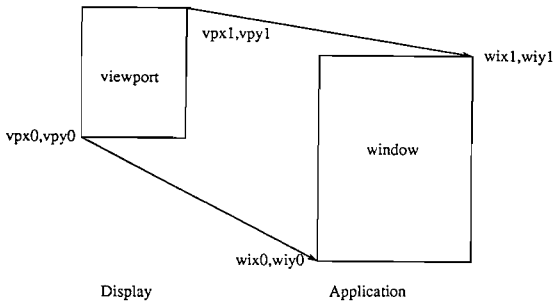

Figure 3: Relation between display- and application coordinates.

4. Hardware Implementation

4.1. An Overview

Each point processor consists of four main modules. These modules (Point Generator, Hierarchy Processor, Transformation Processor and Bounding Box Test) are shown in Figure 4 in relation with two other components (Display Device and Bézier Processor). In order to provide a clear overview, only the most important signals and busses are shown. The Point Generator mediates between the Display Device and the application. For every point on the display the Point Generator calculates the corresponding point in application coordinates. If the image has to be anti-aliased the Point Generator generates a set of application coordinates of which the corresponding colors are averaged. A request (Get-Point) is sent to the Hierarchy Processor to calculate the color while at the same time the application coordinate of the point is made available to the Transformation Processor.

The Hierarchy Processor determines to which object in the tree a certain query point belongs. For performing this task it uses the results of the Bounding Box Test and the Bézier Processor. Depending on the path through the tree, the Hierarchy Processor assigns a color to the query point. As soon as the Hierarchy processor is finished the signal HP-ready will be set to assert that the value on the color bus is valid. The Transformation Processor transforms the query point to the local coordinate system of each object. The Bounding Box Processor performs a global test to determine whether it makes sense to wait for the coordinate transform and evaluate an object still further, or that the next object available should be evaluated.

The result is communicated with the signal Out-box. The Bézier Processor determines if a point is contained in a primitive and replies with the signal BP-result. In order to decrease computation time the bounding box test and the transformation are implemented in parallel. The current design supports two interaction primitives which are frequently used in an interactive environment. A user can specify the address of an object (primitive or composed).

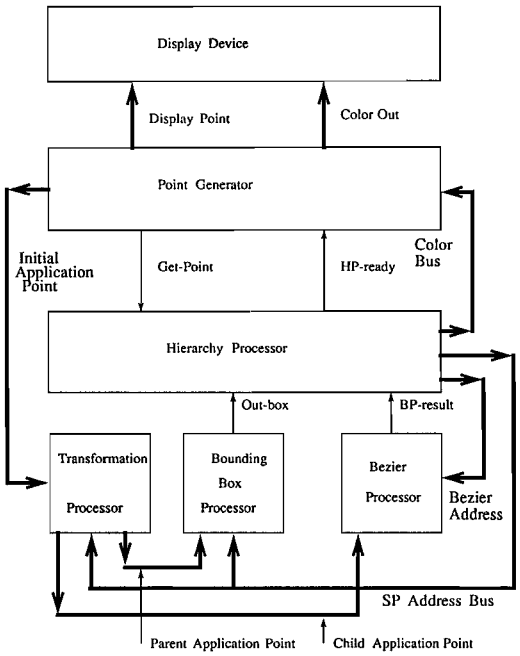

Figure 4: An overview of the structure processor and its environment.

When a pixel is rendered within this object a signal is generated. In the second case the user can specify a point with a pointing device. The corresponding path will be registered in an array which is accessible for the application.

In the following paragraphs we will give a more detailed discussion of the four modules

4.2. The Point Generator

The point generator (see Figure 5) is responsible for the viewing transformation as well as anti-aliasing. The point loop generates the coordinates in a sequential order, although this is not required by the structure processor. If an anti-aliased version of the image is to be rendered an offset is added to this coordinate by the anti-alias loop. The anti-alias loop traverses a sequence offsets once for each original coordinate. The loop may generate 4 or 16 offsets. Subsequently the composed coordinate is transformed to application coordinates. On the other side of the controller the part is shown which delivers the color to the display device. If an un-anti-aliased version of the image is rendered, the color which is returned by the hierarchy processor is directly sent to the display device. If however the image is to be anti-aliased, the colors which are returned by the hierarchy processor are accumulated until the anti-alias loop is finished and subsequently the sum is divided by 4 or 16.

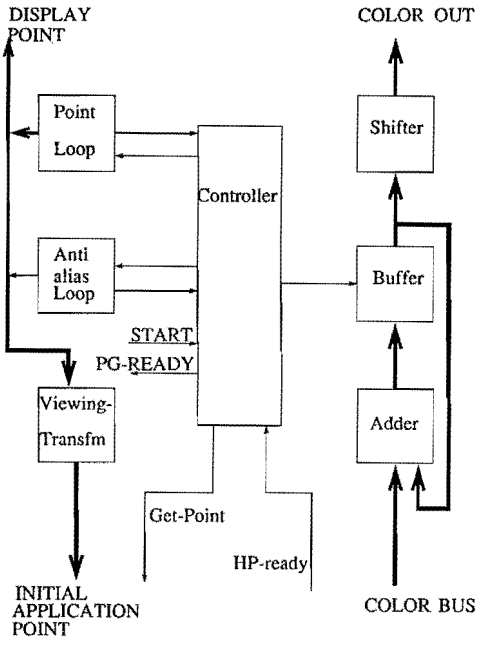

Figure 5: A functional overview of the point generator.

4.3. The Hierarchy Processor

The heart of the structure processor is the hierarchy processor. A task of the hierarchy processor is to interpret the data structure and to select primitives for the Bézier processor. Only those primitives are selected whose bounding box contains the query point. A second task is to assign a color to each point. We will give a more detailed explanation of the hierarchy processor guided by Figure 6.

For every query point the hierarchy processor starts with selecting the root object. The address of this object is contained in the root register. The current address is put on the address bus so that this is available for the peripheral processors (the transformation processor and the bounding box processor). The data belonging to this address is copied to registers. Depending on the result of the bounding box test, the type of object etc. the following address which is selected is the successor address, the child address, the color (interpreted as an address), or an address popped from the stack. If the hierarchy processor has found an object which refers to a primitive such that the query point is inside the bounding box of this object a query is sent to the Bézier processor for determining point containment in the curve which is referred to. If the Bézier processor determines that the query point is inside, the color which is assigned to the point is returned to the point generator.

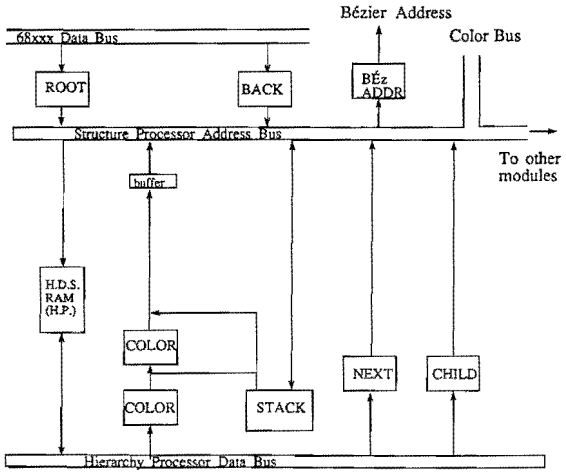

Figure 6: An overview of the hierarchy processor.

4.4. The Bounding Box Processor

The simplest module in the design (shown in Figure 7) is the bounding box test. This test enables a fast pruning of the tree for every query point. In order to perform the test as fast as possible, the four edges of the bounding box are compared to the query point in parallel.

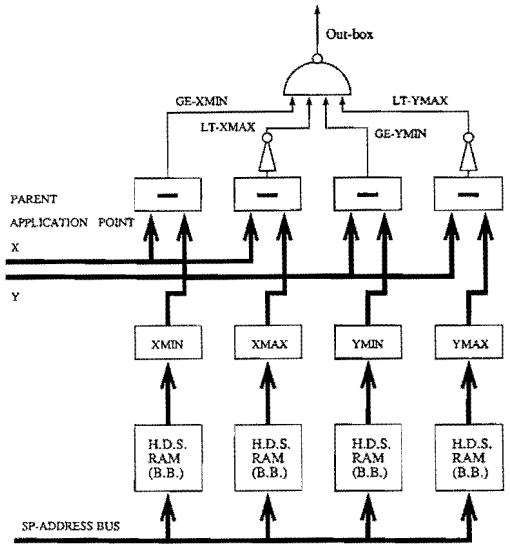

Figure 7: Overview of the bounding box processor.

4.5. The Transformation Processor

The transformation processor (see Figure 8) performs the coordinate transform of the query point from the parent coordinate system to the local coordinate system of the child object. In order to guarantee a transform which is as fast as possible depending on the type of transform used, the most efficient path is selected with two multiplexers. The part which performs the actual transformation is enclosed in the dashed box. Outside the box the stack is shown to save the context, registers for the value of the query point in parent- and child coordinate systems, the structure query point, a register with a query point for the Bézier processor and some buffers to prevent bus collisions.

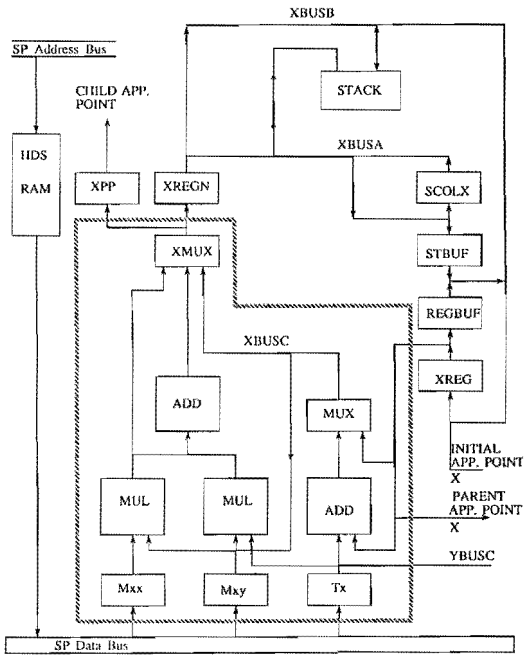

Figure 8: A functional overview of the part of the transformation processor which is responsible for computing the x-coordinate. The part that computes the y-coordinate is identical.

5. Parallelism

Implementation of an algorithm in dedicated hardware and exploiting parallelism will in general improve the performance. In this case four possible ways of parallelism exist, depending on the domain in which the division of tasks is applied. Image space parallelism is the approach where each processor renders a part of the screen. In a system where object space parallelism is applied, each processor will render a part out of the set of objects. But still other approaches are possible: Parallelism can be applied somewhere

on the path from object to image. This last approach will be referred to as in-between parallelism. In-between parallelism can be implemented in a sequential way (pipeline), or by a simultaneous execution of equivalent tasks. An overview of the four types of parallelism is shown in Table 1.

Image space	Fuchs pixel planes [FUCHS]
Object space	Dippè and Swenson [DIPPE]
In-between parallel	Bounding Box Test in this paper
In-between pipeline	Pipeline of geometry engines [CLARK]

Table 1: An overview of approaches for parallelism.

In general the first two approaches will require extra memory. If parallelism in image space is applied, each processor needs its own copy of the data structure. In the case of object space parallelism the processors only need that part of the data structure which they have to display, but they will need a full copy of the frame buffer, so that the processors don't have to wait for each other. The other two approaches do not require extra memory, but are limited in application. For instance the bounding box test can be improved with a factor four by testing the four edges of the box in parallel, but more than four times is not possible. The same applies to the pipeline. The optimal improvement of the speed depends of the number of independent modules.

In our approach, a combination of a number of methods mentioned previously can be used. First the bounding box test and the transformation are implemented in parallel. A parallel implementation in image space is very easy because each processor only computes one pixel at a time. But also parallelism in object space is possible if the processors are used in combination with a preprocessor that selects the parts of the data structure which are significant for each point processor. If the processors are used in that way, the performance may increase more then linearly, while at the same time the required amount of memory per processor decreases.

6. Conclusion

The approach described in this paper offers large advantages for a number of 2D applications. Although the implementation of the point-driven algorithm may be slower than its object driven counterpart when implemented on a sequential processor, the point-driven algorithms can be implemented in much simpler dedicated hardware and parallelism can be applied in a straightforward way. Further extensions are the implementation of primitive objects described by bitmaps and addition of an explicit priority, semi transparent objects etc. This hardware can be a powerful support for an environment in which text and graphics are integrated in a hierarchical framework.

References

[CHAZ] B. Chazelle, and D.Dobkin, "Detection is easier than computation.", In *Annual ACM Symposium on Theory of Computing*, Los Angeles (1980), pp 146-153

[CLARK] Clark J.H., "Structuring a VLSI System architecture", *Lambda*, second quarter 1980, pp 25-30

[COJO] M.E.A. Corthout, and H.B.M. Jonkers, "A New Point Containment Algorithm for B_Regions in the Discrete Plane", In *Theoretical Foundations of Computer Graphics and CAD* NATO ASI Report 1988, pp 279-306, Springer Verlag

[DIPPE] Dippe M., and Swenson J., "An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis", *ACM Siggraph* vol 18, 1984, pp 149-158

[FUCHS] Fuchs H. et al, "Fast Spheres, Shadows, Textures, Transparencies, and image enhancements in Pixel Planes", *ACM Siggraph*, vol 19, 1985, pp 111-120

[PHIGS] American National Standard for the functional specification of the Programmers Hierarchical Interactive Graphics Standard (PHIGS) X3H3/85-21 , X3H31/85-05

Appendix A: A Compact Description of the Algorithm

```
Com_Inside ( OBJECT o, POINT p , COLOR c , BOOL i , BOOL s , POINT ps) {
POINT P;                    /* Query Point */
BOOL I;             /* Flag, Inherit Color ? */
BOOL S              /* Flag  Structure Color */;
POINT PS;                   /* Query Point within Structure */
COLOR C;                    /* Color */
do {
        if ( Box_Inside ( o->box , p ) ) {
                /* Query Point is inside Box of Current Object */
                /* Do first transformation to local coordinate system */
                P = Transform ( o->trans , p );
                if ( i || o->a ) {
                        /* Overwrite inherited color and set of flags */
                        C = o->c; I = o->i, S = o->s;
                        if ( S ) {
                                PS = P;
                        }
                } else {
                        /* Keep inherited color and set of flags */
                        C = c; I = i; S = s;
                        if ( S ) {
                                PS = ps,
                        }
                }
                if ( type (o) == primitive ) {
                        if ( Bez_Inside ( o->child , P ) ) {
                                if ( S ) {
                                        C = Com_Inside ( C, PS, 0, 0, 0, 0 ),
                                }
                        } else {
                                C = -1;
                        }
                } else {
                        C = Com_Inside ( o->child, P, C, I, S, PS );
                }
        } else {
                /* Query Point is Outside Box of Current Object */
                C = -1;
        }
} while ( C == -1 && ( o = o->next ) != NIL )
return ( C );
}
```

Appendix B: Cyclic Depth Ordering

The following code will determine if a query point is inside a cyclically defined object.

```
Com_InsideC ( OBJECT o, POINT p , COLOR c , BOOL i , BOOL s , POINT ps) {
COLOR C,C1,                 /* Color */
        C = Com_Inside (o,p,c,i,s,ps);
        if ( C != -1 ) {
                if ( o->next == NIL ) return C,
                do {

                        if ( o->next == NIL ) {
                                C1 = Com_Inside (o, p, c, i, s, ps) ,
                                if ( C1 != -1)      return (C1);
                                else                return (C),
                        } else o = o->next,
                } while (1);
        } else return ( Com_Inside (o, p, c, i, s, ps) );
}
```