8318 COPY 2

EXPLOITING PARALLELISM in the GRAPHICS PIPELINE

by

Theoharis A. Theoharis

Technical Monograph PRG-54

September 1985
(published June 1986)

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD
England

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD
England

A dissertation submitted for the degree of
Master of Science in the University of Oxford
September 1985

The stages of a Graphics Output Pipeline are discussed, specified and implemented as an OCCAM pipeline. A comparison of the performance of the stages reveals (or rather confirms) the "bottleneck" stage. The Splitter Tree approach to improving this stage's performance is analysed, specified and implemented.

Acknowledgements

I would like to express my gratitude to Ian Page and Joe Stoy, my supervisors, for their help and advice, Ib Sorensen for his assistance with the Z specifications, Professor Hoare and Richard Bird for their kind interest in my financial problems and all the friends and colleagues in the Programming Research Group for making this a most constructive and enjoyable year. The financial support from the S.E.R.C. is acknowledged.

T.A. Theoharis.

To my parents Anthony and Danai.

The graphics pipeline transforms objects described in 3D world coordinates into a picture on a 2D graphics screen. In this thesis we have dealt with the following important stages of the graphics pipeline: clipping, Hidden Surface Elimination (HSE) and coordinate transformations. The viewing transformation stage performs a coordinate transformation from the 3D world coordinate system used to describe our objects, to another 3D coordinate system whose origin is at the point of observation and whose Z axis lies along the direction of view. The purpose of the viewing transformation is to make the calculations involved in the next stage, clipping, easier.

Clipping acts like a filter which only lets through those objects that are potentially visible from the point of observation. For example objects that lie behind the point of observation are invisible. The stages that follow clipping only have to deal with the objects that clipping lets through and these might only constitute a small portion of the original data base.

The next stage, the perspective transformation projects our 3D objects onto the plane of the screen. In doing that, depth information is not destroyed because it is essential for the following stage, HSE.

We have said above that clipping only lets through those objects that are "potentially" visible from the point of observation. This is because some objects might hide others. HSE determines the frontmost object for every pixel of the screen.

The main aim of this project has been the study of the stages of the graphics pipeline with the view of suggesting an architecture that provides a fast implementation of the pipeline.

Chapter 1 describes the stages of the graphics pipeline and chapter 2 formalises that description by mathematically specifying in Z the operation that the more complicated stages of the pipeline should implement.

Chapters 3 & 4 introduce two algorithms which implement the clipping and HSE operations respectively and give their CSP specifications.

Chapter 5 compares the perforance of the stages of the pipeline and identifies the "bottleneck" stage which leads to an investigation of a method for "widening" the bottleneck in chapter 6.

Chapter 7 mentions conclusions and suggests further work.

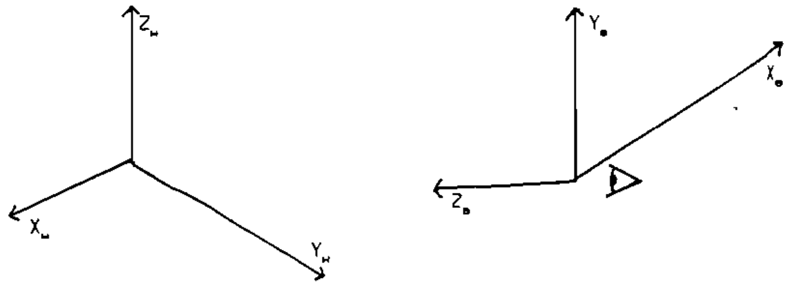
Chapter 1

Background : The Graphics Pipeline

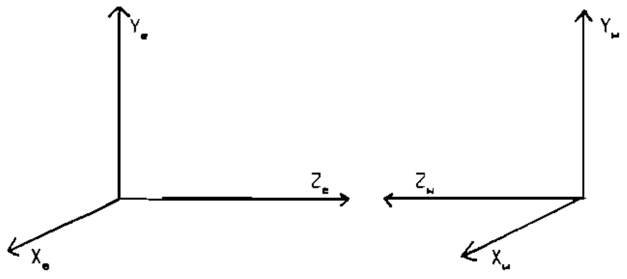
This chapter describes the purpose and function of the main stages of the graphics pipeline and the coordinate systems involved.

Assume a world whose objects are defined solely in terms of polygons in some 3D coordinate system called world coordinate system (WC). WC is assumed to be right handed since right handed coordinate systems are more common than left handed ones.

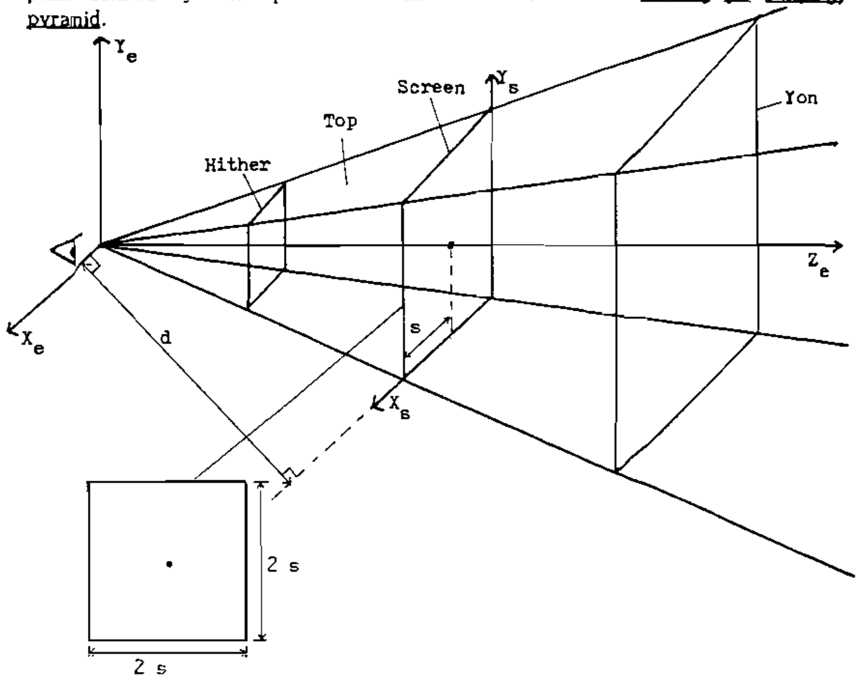
We would like to show on our graphics screen the view of an imaginary observer within our world. First of all we must transform the coordinates of our polygons into a left handed coordinate system whose origin is at the observer's eye and whose Z_e axis lies along the direction of view.



The new coordinate system is called eye coordinate system (EC). This transformation simplifies the calculations involved in later stages of the pipeline and is called viewing transformation. A special case of the viewing transformation is when the Z_w axis is collinear with the Z_e axis and the other axes are parallel, in other words we only transform from a right handed WC system to a left handed EC system.

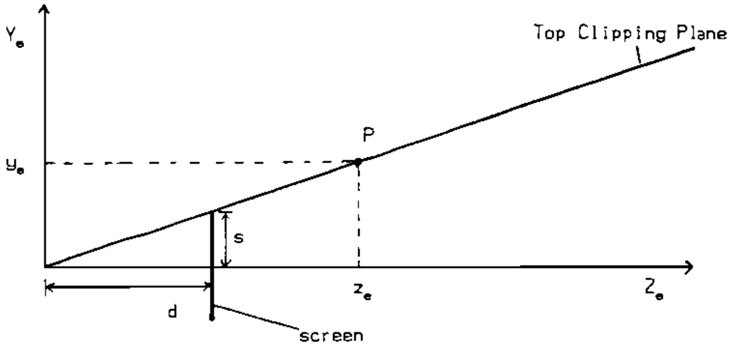


Next, we must determine what part of our world our observer can see. His view can be simulated by a pyramid whose apex is at the EC origin, is symmetrical about the Z_e axis and each of its four faces is perpendicular to the plane defined by some pair of EC axes. It is called the <u>viewing (or clipping) pyramid</u>.



The process of separating the objects, or parts of objects, that lie inside the pyramid from those that do not is called <u>clipping.</u> In addition to the four clipping planes defined by the four faces of the pyramid we usually have "hither" and "yon" clipping planes, which are perpendicular to the Z_e axis, in order to impose depth restrictions. Our 2D screen is assumed to be positioned so that its plane is perpendicular to the Z_e axis, its four corners coincide with the four edges of the viewing pyramid and its X_s and Y_s axes are parallel to and have the same direction as X_e and Y_e respectively.

We can express the clipping limits in terms of the distance of the screen from the point of observation, d, half the screen size, s and the Z_e coordinate of the point being clipped



Let's consider the top clipping plane shown above. For any point $P(x_e,y_e,z_e)$ which lies on it we can show by similar triangles that

$$d / s = z_e / y_e$$

or

$$y_e = z_e * (s / d) = w$$

If a point lies below (inside) the top clipping plane then

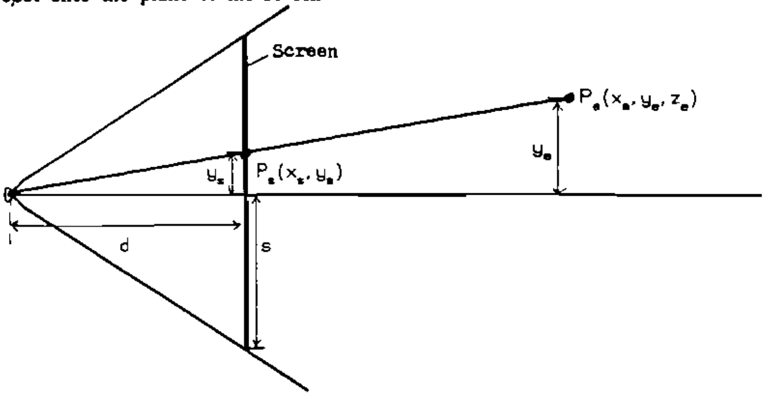
$$y_e < w$$

and if it lies above (outside) the top clipping plane then

$$y_e > w$$

The clipping limits for the other planes are given in appendix 2. Clipping reduces the number of polygons that have to be processed by later stages of the pipeline.

Having determined which objects lie within the viewing pyramid, we must next get a 2D description of them as the screen is a 2D device. We therefore project onto the plane of the screen



We calculate the 2D screen coordinates of P_s by similar triangles

$$y_s / d = y_e / z_e$$

therefore

$$y_s = (y_e / z_e) * d$$

similarly for x

$$x_s = (x_e / z_e) * d.$$

This transformation from the EC system to the screen coordinate system (SC) is called perspective transformation. x_s and y_s are expressed in the units that d is expressed in. Instead we could define them as dimensionless fractions by dividing by s

$$y_s = (y_e / z_e) * (d / s) = y_e / w$$

and

$$x_s = (x_e / z_e) * (d / s) = x_e / w$$

Note that

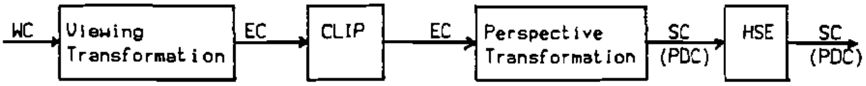
$$-1 < x_s, y_s < 1$$

since $-w < x_e, y_e < w$ after clipping. Therefore x_s and y_s can easily be scaled to any <u>physical device coordinate system (PDC)</u> i.e. the coordinate system used by a physical display device (e.g. 512 x 256). We could have transformed directly from EC to PDC but the intermediate dimensionless SC system enables us to use multiple devices with different PDC systems[1].

To add some realism to our picture we must not display those polygons, or parts of polygons, that are obscurred by others. The operation that determines which is the frontmost polygon at each point of our screen, and hence eliminates the hidden ones, is called Hidden Surface Elimination (HSE). It obviously requires depth information and this is lost in the perspective transformation. We need a <u>depth preserving</u> perspective transformation and an augmented SC (or PDC if we transform directly from EC into PDC) system that includes a third coordinate z_s. In calculating z_s we must make sure that planes in EC transform to planes in SC [Park85]. The interpretation of x_s and y_s is not changed.

But why didn't we perform the HSE operation in the 3D EC system in the first place ? The answer is that it is much harder [SuSp74]. In order to perform the HSE operation in EC we would have to consider "rays" leaving the observation point at various angles and compute which faces they intersect. Such trigonometric computations would be very costly. The depth preserving perspective transformation transforms an object A in EC into an object A' in SC such that A' viewed orthographically looks the same as A viewed in perspective. In other words, the perspective transformation moves the point of observation to infinity transforming the space enclosed by the EC clipping pyramid (truncated by the hither and yon clipping planes) into a SC cube. Hence overlap tests can be done simply by comparing the x_s and y_s coordinates of points.

Here are the stages of the graphics pipeline described in this chapter



---

1. In our OCCAM implementation the perspetive transformation transforms directly from EC to PDC in order to avoid the use of reals.

Chapter 2

Z Specification of two Major Graphics Operations

In this chapter we shall give the mathematical specification in Z [Z85] of Clipping and Hidden Surface Elimination (HSE) in order to make their meanings precise before proceeding to the description of algorithms to implement them.

2.1. Geometrical Definitions

A point in 3D is described by its cartesian coordinates

```
┌─POINT──────────────────────────────
│      $x : \mathbb{R}$
│      $y : \mathbb{R}$
│      $z : \mathbb{R}$
└────────────────────────────────────
```

A plane is the set of 3D points which satisfy a plane equation

```
┌─PLANE──────────────────────────────
│      $plane \quad : \mathbb{P}(POINT)$
│      $a, b, c, d : \mathbb{R}$
├──────────────────────
│      $plane = \{(x, y, z) : POINT \mid a*x + b*y + c*z + d = 0\}$
└────────────────────────────────────
```

A line is the intersection of two non parallel planes

```
┌─LINE───────────────────────────────
│      $line : \mathbb{P}(POINT)$
├──────────────────────
│      $\exists\ PLANE_1 ; PLANE_2 \mid plane_1 \cap plane_2 \neq \phi \bullet$
│              $line = plane_1 \cap plane_2$
└────────────────────────────────────
```

The unrestricted set of 3D points is called <u>space</u>

```
┌─SPACE──────────────────────────────
│      space : ℙ(POINT)
└────────────────────────────────
```

A plane divides space into two <u>halfspaces.</u> The coordinates of all points in one halfspace give a positive value when substituted into the plane equation, whereas the coordinates of the points in the other halfspace give a negative value

```
┌─HALFSPACE──────────────────────────────
│      halfspace : ℙ(POINT)
│      a,b,c,d   : ℝ
├──────────────────────
│      halfspace = {(x,y,z) : POINT | a*x + b*y + c*z + d > 0}
│      ∨
│      halfspace = {(x,y,z) : POINT | a*x + b*y + c*z + d < 0}
└────────────────────────────────
```

A <u>halfline</u> from a point p is a semi infinite line whose one and only end is at p. It is defined as the intersection of a line and a halfspace. The line must not be parallel to the plane defining the halfspace and p must lie on this plane. The halfline does not include p

```
┌─HALFLINE──────────────────────────────
│      p        : POINT
│      halfline : ℙ(POINT)
├──────────────────────
│      ∃ LINE ; HALFSPACE | line ∩ helfspace ≠ ∅•
│            halfline = line ∩ halfspace
│            p ∈ line
│            a*p.x + b*p.y +c*p.z + d = 0
└────────────────────────────────
```

A line segment consists of a starting point, an ending point and all the points between them which lie on the line defined by these two points. We define a line_segment as the intersection of two collinear halflines of opposite direction. It is convenient to include in the line_segment either the starting or the ending point, but not both

```
─LINE_SEGMENT───────────────────────────────
      start          : POINT
      end            : POINT
      line_segment   : $\mathbb{P}$(POINT)
─────────────────────
      $\exists$ HALFLINE$_1$ ; HALFLINE$_2$ |
            $p_1$ = start $\wedge$ $p_2$ = end
            start $\in$ halfline$_2$ $\wedge$ end $\in$ halfline$_1$$\bullet$
            line_segment = (halfline$_1$ $\cap$ halfline$_2$) $\cup$ {start}
────────────────────────────
```

Two line_segments are connected if the start point of one of them coincides with the end point of the other

```
─CONNECTED──────────────────────────────────
      connected : LINE_SEGMENT $\leftrightarrow$ LINE_SEGMENT
─────────────────────
      $\forall$ ls1, ls2 : LINE_SEGMENT | (ls1, ls2) $\in$ connected$\bullet$
            ls1.end = ls2.start
────────────────────────────
```

A sequence of connected line_segments is defined so that line_segments which are adjacent in the sequence are connected

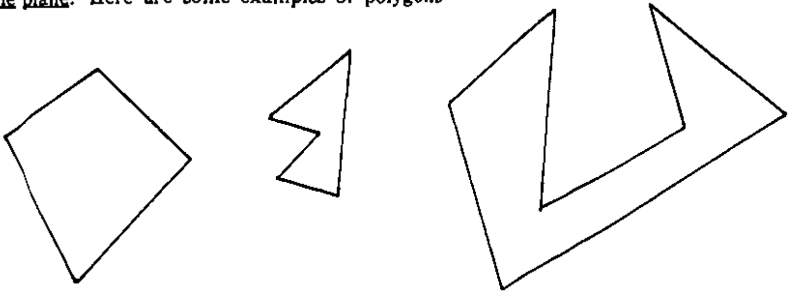
```
─CONNECTED_LINE_SEGMENTS────────────────────────────────
      connected_line_segments : $\mathbb{P}$(seq(LINE_SEGMENT))
─────────────────────
      $\forall$ lss : seq(LINE_SEGMENT)$\bullet$
            (lss $\in$ connected_line_segments $\Leftrightarrow$
             lss$^{-1}$;succ;lss $\subseteq$ CONNECTED
            )
────────────────────────────
```

We shall define a polygon to consist of

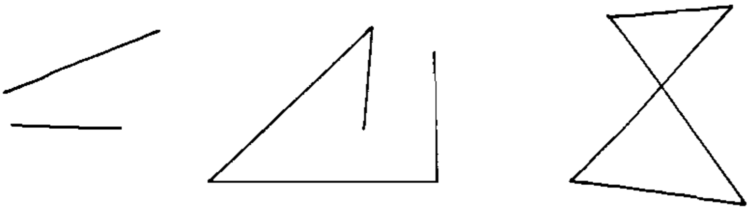
i. Its edges and

ii. Its contents i.e. all the points bounded by the edges.

The edges must be pairwise connected and the beginning of the first edge must coincide with the end of the last edge, in other words the polygon must be closed. It is not allowed for the edges to cross over each other and they must all lie in the same plane. Here are some examples of polygons

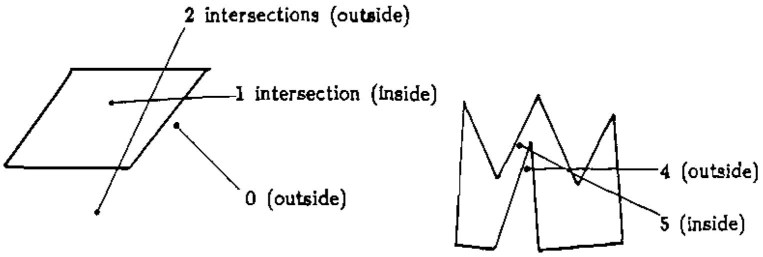


and some counter examples



The contents of the polygon are defined as those points from which there is "no escape" from the polygon. In other words any "escape route" is bound to meet an edge of the polygon.

A necessary and sufficient condition for a point p to be inside the polygon i.e. a point of no escape, is that the number of intersections of any halfline starting at p with the edges of the polygon be odd. A few examples will illustrate this



The following Z specification of a polygon encapsulates the above requirements

```
┌─POLYGON──────────────────────────────────────
│    contents : ℙ(POINT)
├──────────────────────────────────
│∃ edges : seq(LINE_SEGMENT)•
│   (edges ∈ CONNECTED_LINE_SEGMENTS
│    edges(1).start = edges(#edges).end
│    ∀ i,j : dom(edges) | i≠j •
│        (edges(i).line_segment) ∩ (edges(j).line_segment) = ∅
│    ∃ pl : PLANE•
│        ⋃ {i : 1..#edges • (edges(i).line_segment)} ⊆ pl.plane
│    contents ≙ {q : POINT | (∃ HALFLINE | q = p•
│        #(halfline ∩ (⋃ {i : 1..#edges • edges(i).line_segment})) ∈ odd)}
│   )
└──────────────────────────────────────
```

2.2. The Clipping Operation

The clipping operation restricts objects (or the polygons that define them) to those that lie within a certain region of space, the clipping region. The clipping region can be described as the generalised intersection of an appropriate sequence of halfspaces. The result of clipping a polygon is then the intersection of the points that lie within the polygon (its contents) with the clipping region

```
┌─CLIP──────────────────────────────────
│p?                : POLYGON
│clipping_region   : seq(HALFSPACE)
│p!                : POLYGON
├──────────────────────────────────
│p!.contents =
│   ⋂ {i : 1..#clipping_region • (clipping_region(i).halfspace)}
│      ∩ p?.contents
└──────────────────────────────────────
```

The clipping region usually takes the form of a truncated pyramid, the clipping pyramid, as described in chapter 1.

2.3. The HSE Operation

The following specifications are generic in terms of COLOUR

[COLOUR]

A picture contains 3D polygons of various colours. As each polygon can only have one colour, a picture can be described by the partial function

$$PIC : POLYGON \rightarrowtail COLOUR$$

The points that belong to some polygon of PIC are related to the colour of the polygon that they belong to. This is a relation since some points might belong to more than one polygon and therefore be associated with more than one colour

$$USP\_rel : POINT \leftrightarrow COLOUR$$
$$USP\_rel \mathrel{\widehat{=}} \{p : POINT ; poly : dom(PIC) \mid p \in poly.contents \bullet (p, PIC(poly))\}$$

USP_rel stands for Unhidden Surface Picture relation. We shall next derive from USP_rel a function, USP_fun, that associates a unique colour to every point. The colour that USP_fun associates with a point p must be one of the colour(s) that USP_rel associates with p, the choice being implementation dependent. For example our OCCAM implementation associates with p the colour of the first polygon, in the order of processing, that contains p

$$USP\_fun : POINT \rightarrowtail COLOUR$$
$$dom(USP\_fun) = dom(USP\_rel)$$
$$USP\_fun \subseteq USP\_rel$$

The result of performing the HSE operation will be a Hidden Surface Picture (HSP) that associates 2D coordinates to colours (our points so far have been 3D)

$$HSP : \mathbb{R} \times \mathbb{R} \rightarrowtail COLOUR$$

The colour of a 2D point (x,y) is the colour of the 3D point (x,y,z) which has the smallest z coordinate among all 3D points whose lateral and vertical coordinates are x and y respectively

```
HSP ≙ {p :  dom(USP_fun) |
                    (∀ p₁ : dom(USP_fun) |
                         p₁ ≠ p ∧
                         p₁.x = p.x ∧
                         p₁.y = p.y •     p₁.z > p.z) •
               ((p.x, p.y), USP_fun(p)) }
```

The direct comparison of the z coordinates of points with the same x and y is only valid if a (depth preserving) perspective transformation has preceeded it (see chapter 1). The HSE operation is then defined as follows

```
┌HSE─────────────────────────────────────────────
│      pic? : POLYGON ⇸ COLOUR
│      hsp! : ℝ × ℝ ⇸ COLOUR
├──────────────────────────────
│      ∃ usp_rel : POINT ↔ COLOUR; usp_fun : POINT ⇸ COLOUR •
│         (usp_rel = {p : POINT ; poly : dom(pic?) | p ∈ poly.contents •
│                        (p, pic?(poly))}
│          dom(usp_fun) = dom(usp_rel)
│          usp_fun ⊆ usp_rel
│          hsp! = {p : dom(usp_fun) |
│                        (∀ p₁ : dom(usp_fun) |
│                             p₁ ≠ p ∧
│                             p₁.x = p.x ∧
│                             p₁.y = p.y •     p₁.z > p.z) •
│                   ((p.x,p.y), usp_fun(p)) }
│
└────────────────────────────────────────
```

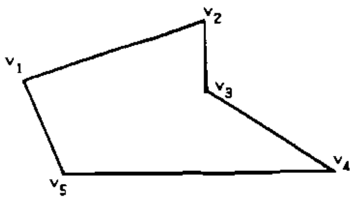
Chapter 3

A Parallel Clipping Algorithm

The objective of the clipping operation was described using the Z language in section 2.2. This chapter describes and gives a CSP trace specification of a parallel OCCAM [INMO84] implementation of the Sutherland-Hodgman polygon clipping algorithm [Suth74].

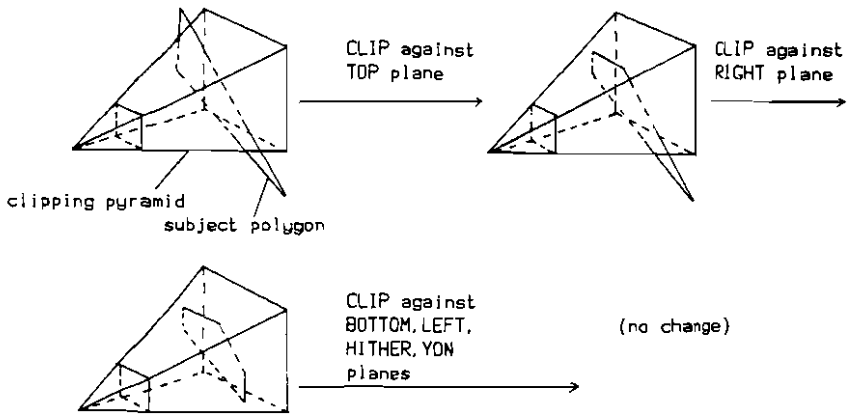
3.1. Description

The polygon to be clipped, the subject polygon, is represented as a sequence of vertices; the first and the last being the same[i]. The vertices occur in the order defined by a clockwise traversal around the polygon. For example



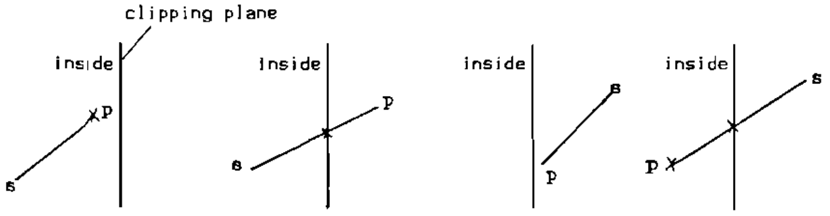
is represented as $\langle v_1, v_2, v_3, v_4, v_5, v_1 \rangle$

The algorithm clips the subject polygon against the first plane of the clipping pyramid and produces a new sequence of vertices which represent the subject polygon clipped against the first clipping plane. The process is repeated for each plane of the clipping pyramid. The sequence of vertices coming out of the last clipping stage represents the subject polygon clipped against the clipping pyramid. Here is an example of the algorithm at work



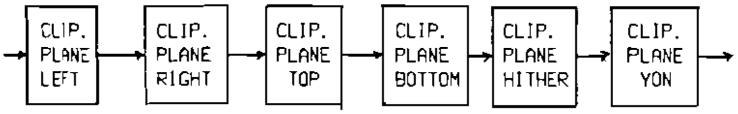
i. Repeating the first vertex as last in the representation of a polygon, makes consideration of its edges and hence reasoning about the polygon, easier. In our implementation we have avoided this duplication by remembering the first vertex (see Appendices 1 & 4).

But how is clipping against a plane performed? The vertices of the subject polygon are considered in pairs (s,p) in a clockwise traversal around it. For each such pair 0,1 or 2 vertices are output to the next stage depending on the relationship between the pair (s,p) and the clipping plane. There are four cases to be considered



X represent output vertices

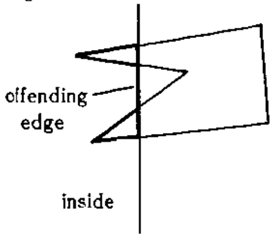
The obvious parallel implementation of the algorithm is as a 6-stage pipeline, where 6 is the number of planes in the clipping pyramid. A stream of vertices will pass through the pipeline. The input stream represents the subject polygon and the stream coming out of the ith stage represents the subject polygon after it has been clipped against planes 1..i of the clipping pyramid



Notice that, as far as the clipping algorithm is concerned, the clipping volume can be of any convenient shape defined by any number of planes.

A Problem with Concave Polygons

Concave polygons which result in two or more polygons after clipping, will give rise to an edge which connects the resulting polygons as shown below



The offending edge could be removed by a modest alteration to the clipping algorithm as described in [Suth74].

3.2. CSP Specification

Each stage, $\mathrm{CLIP.PLANE}_{plane}$, of the clipper must comply with the following specification

$$right \leq f_{plane}(left)$$

where left and right are the input and output channels of the clipping stage respectively and f_{plane} is defined as

$$
\begin{aligned}
f_{plane}(\langle\rangle) &= \langle\rangle \\
f_{plane}(\langle p\rangle) &= \langle\rangle \\
f_{plane}(\langle s,p\rangle \frown rest) &= \langle p\rangle \frown f_{plane}(\langle p\rangle \frown rest), \\
&\qquad inside^{i}(plane,s)\ \&\ inside(plane,p) \\
&= \langle\ intersection^{ii}(s,p,plane),\ p\ \rangle \frown f_{plane}(\langle p\rangle \frown rest), \\
&\qquad \sim inside(plane,s)\ \&\ inside(plane,p) \\
&= \langle\ intersection(s,p,plane)\ \rangle \frown f_{plane}(\langle p\rangle \frown rest), \\
&\qquad inside(plane,s)\ \&\ \sim inside(plane,p) \\
&= f_{plane}(\langle p\rangle \frown rest), \\
&\qquad \sim inside(plane,s)\ \&\ \sim inside(plane,p)
\end{aligned}
$$

f_{plane} specifies recursively the relationship that must hold between the input and the output vertices of a clipping stage.

Now assuming that each clipping stage satisfies its specification i.e.

$$\forall\ i \mid 1 \leq i \leq N^{iii} \bullet \mathrm{CLIP.PLANE}_{plane(i)}\ \ sat\ \ right \leq f_{plane(i)}(left)$$

---

i. inside(plane,p) delivers TRUE or FALSE depending on whether p is on the "inside" of the clipping plane or not. Its calculation is shown in appendix 2.

ii. intersection(s,p,plane) delivers the coordinates of the point of intersection of the line segment from s to p with the clipping plane. Its calculation is shown in appendix 2.

iii. N is the number of clipping stages. N = 6 in the case of the clipping pyramid with Hither and Yon planes.

we can deduce the following about their combination in a pipeline (by L1 of section 4.4.4. of [Hoar83])

$$\gg_{1 <= i <= N} \mathrm{CLIP.PLANE}_{plane(i)}$$

sat

$$\exists\, s_1 . s_2 . . s_{N-1} \bullet (\text{right} \leq f_{plane(N)}(s_{N-1})\ \&$$
$$s_{N-1} \leq f_{plane(N-1)}(s_{N-2})\ \&$$
$$\vdots$$
$$s_1 \leq f_{plane(1)}(\text{left})$$
$$) \qquad \ldots(A)$$

assuming CLIP.PLANE is left guarded.

lemma

$s \leq f(t)\ \&\ t \leq u \Rightarrow s \leq f(u)$

assuming $f(p) \leq f(p \frown q)$

proof

$t \leq u \rightarrow \exists\, v \bullet\ t \frown v = u$ (section 1.6.5. of [Hoar83])		...(1)
$s \leq f(t)$	(given)	...(2)
$f(t) \leq f(t \frown v)$	(assumption)	...(3)
$f(t) \leq f(u)$	(by (1) & (3))	...(4)
$s \leq f(u)$	(by (2), (4) and transitivity of $\leq$)	

By the above lemma and noting that $f_{plane}(p) \leq f_{plane}(p \frown q)$, (A) can be simplified to

$$\gg_{1 <= i <= N} \mathrm{CLIP.PLANE}_{plane(i)}$$

sat

$$\text{right} \leq f_{plane(N)}(f_{plane(N-1)} . . (f_{plane(1)}(\text{left})]$$

In other words we have proved that if the relationship specified by f_{plane} holds between the input and the output vertices of each clipping stage, then the relationship that holds between the input and the output vertices of the combination of all the clipping stages in a pipeline is given by the combination of the f_{plane} functions of all the stages. This obviously means that the output vertices lie on the "inside" of all the clipping planes, as desired.
This result can be instantiated to the case of the clipping pyramid

$$\gg_{i:\{LEFT,RIGHT,TOP,BOTTOM,HITHER,YON\}} CLIP.PLANE_{plane(i)}$$

sat

$$right \leqslant f_{YON}(f_{HITHER}(f_{BOTTOM}(f_{TOP}(f_{RIGHT}(f_{LEFT}(left))]$$

Chapter 4

Hidden Surface Elimination Algorithm

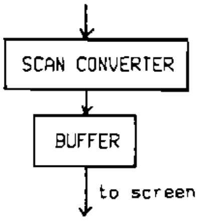
This chapter describes the Z-buffer Hidden Surface Elimination (HSE) algorithm and gives its CSP specification.

The Z-buffer HSE algorithm has been chosen among the wide variety of HSE algorithms for the following reasons

i. It is relatively simple to implement in a language like OCCAM that does not provide many data structures.

ii. It fits well into our notion of the pipeline of polygons (see appendix 1) as it does not require to examine all the polygons at once. Instead polygons are processed individually in the order they come down the pipeline. There is no explicit depth sorting step required.

The drawback of this algorithm is that it uses a large 2D array called Depth buffer (Z-buffer) on top of the usual Frame buffer (F-buffer) array that is used to store the colour of the pixels. The Z-buffer is used to store a depth value for each pixel of the screen, so its dimensions are Yresolution * Xresolution.
The algorithm consists of a Scan Converter and a Buffer Process' running in parallel



The scan converter receives polygons in "augmented" Physical Device Coordinates (that include a depth value) and determines the pixels that lie within each polygon. In addition to that it calculates the depth of the polygon at each pixel within it by making use of its plane equation (see Appendix 3) and transmits **(colour,x,y,depth)** quadruplets to the buffer process.
The buffer process receives such quadruplets and for each of them it takes the following action;

If the value of the Z-buffer at (**x**,y) is greater than **depth**,
 it updates this value to **depth** and also updates the (**x**,y) position of the F-buffer to **colour**,
otherwise it does nothing.

---

i. The name Buffer might be misleading here. It is not a buffer in the CSP sense but a process that controls the Z and F buffers (which are just 2D matrices).

This in effect means that if the previous polygon that included pixel (**x**,**y**) was further away than the current one at this pixel, then the current one hides the previous one at (**x**,**y**) and pixel (**x**,**y**) must take its colour.
Before processing a new frame, the Z-buffer is initialised to the maximum representable depth value and the F-buffer to the background colour.

The Scan Converter

First of all we have had to implement in OCCAM certain data structures along with specialised operations on them to support the scan conversion algorithm. These were a bucket organised Edge Table (ET) and an Active Edge Table (AET) organised as a list. The ET has one bucket for each scanline, containing information about the edges whose minimum y coordinate corresponds to that scanline. The AET contains information about the edges that the current scanline intersects. The implementation of these data structures is described in Appendix 3.
The scan conversion algorithm is an extension of the one described in [Fole82]; it also estimates the depth of the relevant polygon at each pixel scan converted. Here is its description

```
For each polygon
*Determine the plane equation coefficients a,b,c and d
*Clear the ET and AET data structures
*Construct the ET for the polygon's edges
*Let Y be the index of the first non-empty ET bucket
*While (AET ≠ empty) OR (Y < index of last non-empty ET bucket)
  **Move ET bucket Y into the AET maintaining AET sorted on x
  **For each pair of edges  e1,e2  in the AET
    ***Let  X1,X2  be the x intersections of e1 and e2 with scanline Y
    ***Compute the depth  Z  of the polygon's plane at  X1,Y
       (Z = -(d + a*X1 + b*Y) / c)
    ***Compute the depth increment $Z_{inc}$ = -a / c
    ***For X := X1 to X2
     ****Send <polygon colour,X,Y,Z> to the buffer process
     ****Z := Z + $Z_{inc}$
  **Update the AET by removing those edges whose $y_{max}$ is equal
    to Y and computing the x intercept of the remaining AET edges
    with scanline Y + 1
  **Bubblesort the AET, in case it became out of order by the update
  **Y := Y + 1
```

Before specifying the function of the Z-buffer HSE algorithm in CSP, we must define two auxilliary functions

KM returns a constant matrix of the value given to it as argument. The size of the matrix is equal to the resolution of the screen

$$KM : VAL \longrightarrow MATRIX$$
$$KM\ (v) = [v]_{i:1..Xresolution,\ j:1..Yresolution}$$

UPDATE updates a location of a matrix. The matrix to be updated, the location concerned and the new value are arguments of UPDATE

$$UPDATE : MATRIX \times VAL \times VAL \times VAL \longrightarrow MATRIX$$
$$UPDATE\ (M, x, y, v) = M \oplus \{\langle x, y\rangle \mapsto v\}$$

The scan converter process inputs polygons on channel b and outputs the pixels within each polygon along with their associated colour and depth on channel c. A special kind of polygon, NEXT.FRAME.POLY, separates the polygons of one frame from those of the next (see Appendix 1)

```
α(SCAN.CONVERTER) = {b,c}

SCAN.CONVERTER = b ? polygon
                 (  (c ! NEXT.FRAME.PIXEL →
                     SCAN.CONVERTER
                    )
                   ⊀ polygon = NEXT.FRAME.POLY ⊁
                    (For each pixel (x,y) inside polygon
                      (c ! colour(polygon)    →
                       c ! x                  →
                       c ! y                  →
                       c ! depth(polygon,x,y)
                      )
                    ) ;
                    SCAN.CONVERTER
                 )
```

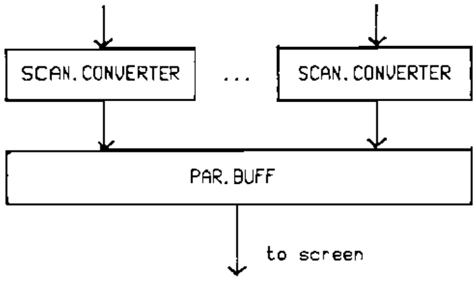
The buffer process, which is only called so because of the usual name of this HSE algorithm, receives pixels along with their associated colour and depth values from the SCAN.CONVERTER on channel c. A special kind of pixel, NEXT.FRAME.PIXEL, signals the start of a new frame. Upon receipt of this pixel the buffer process sends the F-buffer to the screen in order to be displayed and re-initialises the Z and F buffers

```
α(BUFFER) = {c, screen}

BUFFER_{Z.F} = c ? colour  →
             (  (screen ! F →
                 BUFFER_{KM(MAX.DEPTH), KM(BACK.GROUND.COLOUR)}
                )
               ⊀ colour = NEXT.FRAME.PIXEL ⊁
                (c ? x       →
                 c ? y       →
                 c ? depth   →
                 ( BUFFER_{UPDATE(Z,x,y,depth), UPDATE(F,x,y,colour)}

                  ⊀ Z(x,y) > depth ⊁

                   BUFFER_{Z.F}
                 )
                )
             )
```

The parallel combination of the SCAN.CONVERTER and the BUFFER is our HSE algorithm

$$ZHSE = (SCAN.CONVERTER \parallel BUFFER_{KM(MAX.DEPTH),\ KM(BACK.GROUND.COLOUR)}) \setminus \{c\}$$

Since the most complicated and time consuming part of the algorithm is the scan conversion, we could have many scan converters running in parallel by distributing the polygons amongst them (see section 6.3.3.)



We then need a buffer process, **PAR.BUFF**, that is capable of dealing with all the scan converters. It only sends the F-buffer to the screen if it has received a **NEXT.FRAME.PIXEL** from all the scan converters. It uses a set, S, to keep track of those scan converters (SCC) that have sent a **NEXT.FRAME.PIXEL**

$$\alpha(\mathrm{PAR.BUFF}) = \{\mathrm{screen}\} \cup \{c_i \mid i:1..\#\mathrm{SCC}\}$$

```
$\mathrm{PAR.BUFF}_{Z,F,S} = |_{i \in S}$ (  $c_i$ ? colour $\rightarrow$
                        ( ( (screen ! F $\rightarrow$
                              $\mathrm{PAR.BUFF}_{\mathrm{KM(MAX.DEPTH)},\ \mathrm{KM(BACK.GROUND.COLOUR)},\ 1..\#\mathrm{SCC}}$
                             )
                           $\not<$ S-{i} = $\phi$ $\not>$
                            $\mathrm{PAR.BUFF}_{Z,F,S-\{i\}}$
                           )
                          $\not<$ colour = NEXT.FRAME.PIXEL $\not>$
                          ($c_i$ ? x        $\rightarrow$
                           $c_i$ ? y        $\rightarrow$
                           $c_i$ ? depth    $\rightarrow$
                           ( $\mathrm{PAR.BUFF}_{\mathrm{UPDATE}(Z,x,y,\mathrm{depth}),\ \mathrm{UPDATE}(F,x,y,\mathrm{colour}),\ S}$

                             $\not<$ Z(x,y) > depth $\not>$

                             $\mathrm{PAR.BUFF}_{Z,F,S}$
                           )
                          )
                        ))
```

PAR.BUFF can be combined with the scan converters as follows

$$\mathrm{PAR.ZHSE} = ((\|_{i:1..\#\mathrm{SCC}} (c_i \underset{c}{\rightarrow} (b_i \underset{b}{\rightarrow} (\mathrm{SCAN.CONVERTER})))) \; \| \; \mathrm{PAR.BUFF}_{\mathrm{KM(MAX.DEPTH)},\ \mathrm{KM(BACK.GROUND.COLOUR)},\ 1..\#\mathrm{SCC}}) \setminus \{c_i \mid i:1..\#\mathrm{SCC}\}$$

$$\alpha(\mathrm{PAR.ZHSE}) = \{\mathrm{screen}\} \cup \{b_i \mid i:1..\#\mathrm{SCC}\}$$

Performance of the Graphics Pipeline : The Bottleneck

Two different models are used to compare the performance of the stages of the graphics pipeline. The first is based on an extension of a performance evaluation of ten Hidden Surface Elimination (HSE) algorithms by Sutherland et al [SuSp74]. The second is based on code timing figures derived from our OCCAM implementation of the pipeline using the INMOS Transputer *Estimator.*

5.1. Graphics Pipeline Performance (1)

In this section we compare the performance of the stages of the graphics pipeline based on the excellent performance evaluation of ten HSE algorithms by Sutherland et al [SuSp74].

[SuSp74] compare the ten HSE algorithms as follows. The operations that each algorithm has to perform (like sorting, searching, intersection calculations etc) are identified and assigned a complexity factor, cf_{op}, depending on a crude relative estimate of their time complexity. A complexity factor of 1 is assigned to very simple operations (like solving a plane equation), 10 to more costly operations (like computing the relationship between two segments in 2D) and 100 to very expensive operations (like computing the intersection between an edge and an object in 3D).

The number of times an algorithm has to perform each operation,
n_{op}, is expressed in terms of "Environment Statistics" (like the total number of edges in the environment, the number of relevant faces (after clipping), the resolution of the screen etc). The performance of each algorithm is then estimated as

$$\text{algorithm performance} = \Sigma_{op} \; n_{op} * cf_{op}$$

The ten algorithms are compared in three environments of different complexity by varying the values of the Environment Statistics.

We shall estimate the performance of the clipping and
coordinate transformation stages in terms of the same Environment Statistics that [SuSp74] used to evaluate the ten HSE algorithms.

The Sutherland Hodgman polygon clipping algorithm is described in Chapter 3. In our implementation we have structured it as a six stage pipeline, each stage clipping against one of the six clipping planes (see section 3.1.). A clipping stage considers the edges of each polygon (which are defined in terms of pairs of vertices (s, p)) and for each such edge it does the following

```
in_s  :=  inside(plane,s)          --determine which side of the
in_p  :=  inside(plane,p)          --clipping plane s & p lie on

case
      in_s  & in_p  : output p to next stage

     ~in_s  & in_p  : {i := intersection(s,p,plane)
                       output i to next stage
                       output p to next stage
                      }

      in_s & ~in_p  : {i := intersection(s,p,plane)
                       output i to next stage
                      }

      otherwise     :  donothing
endcase
```

The above code fragment is executed once for each edge in the environment. E_t of [SuSp74] is the environment statistic that stands for the total number of edges in the environment (before clipping). Since the first clipping stage will consider all the edges in the environment, E_t is the number of times the code fragment will be executed. Each execution requires 2 "inside" calculations and possibly one "intersection" calculation. Since these are both simple, each execution is assigned a complexity factor of 10. It therefore takes 6 * 10 units of time for the first vertex to pass through the 6-stage pipeline (if it isn't clipped out) and then the rest of the vertices are processed in E_t * 10 units of time. The time performance of the clipping algorithm (i.e. the time it takes to process all the edges) is

$6 * 10 + E_t * 10$ OR
$E_t * 10$ units of time since E_t is likely to be large

5.1.2. Performance of Coordinate Transformations

Coordinate transformations involve some arithmetic operations for each vertex hence a complexity factor of 10. The number of vertices in the environment is the same as the number of edges. However, the number of vertices that reach the perspective transformation stage is likely to be smaller than the original number of vertices, E_t, since some vertices will be filtered out by the clipper. [SuSp74] provide another statistic, E_r, (relevant number of edges) which stands for the number of edges (vertices) that survive the clipper. The performances of the viewing and perspective transformations are therefore

Viewing	: $E_t * 10$	units of time
Perspective	: $E_r * 10$	units of time

since the viewing and perspective transformations are performed before and after clipping respectively.

5.1.3. The Bottleneck

We shall compare the performance of clipping and the two coordinate transformations against the performance of the HSE algorithms in each of the three environments.

The values of E_t and E_r for each of the three environments are [SuSp74]

Environment	E_t	E_r
A	800	400
B	20K	10K
C	480K	240K

$K = 10^3$

Here is how the performances of the HSE algorithms, borrowed from [SuSp74] table 7, compare with those of clipping and the two coordinate transformations (we only show the best and worst HSE algorithm performance for each environment)

Environment	HSE(best)	HSE(worst)	Clipping	Viewing Transf.	Perspective Transf.
A	140K	2.4B	8K	8K	4K
B	1.4M	62B	200K	200K	100K
C	7.5M	1500B	4.8M	4.8M	2.4M

$K = 10^3$
$M = 10^6$
$B = 10^9$

Although the comparison is crude, it is evident that HSE is the bottleneck.

5.2. Graphics Pipeline Performance (2)

In this section we shall estimate the "rate of flow" of polygons through each of the stages of our OCCAM implementation of the graphics pipeline in order to verify the bottleneck and derive more accurate figures for our implementation.

The time taken by each stage of the pipeline to process a certain environment is estimated using the INMOS Transputer Estimator. This is a static estimator i.e. it does not consider the execution of the program. As a result we encountered difficulties with the following constructs

i. WHILE loops
(the estimator considers a single execution of the loop)

ii. IF statements
(the estimator considers the most expensive alternative)

The first problem was solved by estimating the number of times a loop is executed in terms of environment parameters like the number of polygons, the resolution of the screen etc and multiplying that by the cost of a single execution of the loop.

The solution to the second problem would involve estimating the probabilities for each path of an IF statement, multiplying them by the cost of the path and summing up the products. The complexity of the solution coupled with the observation that most IF statements in our code are quite evenly balanced, led us to ignore this problem.

The following estmation of the timing of the stages of the graphics pipeline assumes knowledge of the algorithms involved and their implementation, details of which are given in chapters 3 & 4 and appendix 3.

The environment parameters we need in order to estimate the number of loop executions are the following

1. Vertical screen resolution	Yres
2. Horizontal screen resolution	Xres
3. Depth complexity	Dc
4. Total number of polygons	P_t
5. Number of relevant polygons	P_r
6. Average polygon width in pixels	W
7. Average polygon height in pixels (or scanlines)	H
8. Average number of edges per polygon	E
9. Average number of edges per bucket in final Edge Table	E_{etf}
10. Average number of edges per bucket in Edge Table being constructed	E_{etc}
11. Average number of edges in the Active Edge Table	E_{aet}

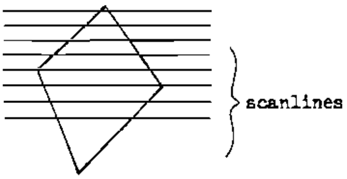
Let's assume that our environment contains 1024 four-sided polygons (P_L=1024, E=4) and that half of them are clipped out[1] (P_r=512). Let's also say that our screen's resolution is 500 × 500 (Yres=500, Xres=500) and that we have a depth complexity of 1 (Dc=1). Depth complexity is the average number of polygons that cover a pixel or, equivalently, the average number of times that a pixel is output from the scan converter.

Then the average number of pixels covered by a polygon is
$(Yres * Xres * Dc) / P_r$ and therefore
$W = H = \sqrt{(Yres * Xres * Dc) / P_r}$
assuming no particular shape for a polygon.

Now since the edges of only one polygon occupy the Edge Table (ET) at a time, E_{etf}=E/H, as we only consider the ET buckets that correspond to scanlines that our polygon intersects (there are H of them). E_{etf} denotes the number of edges per ET bucket once the ET has been constructed. This is different from the average number of edges per ET bucket while the ET is being constructed, E_{etc}. Before inserting the first edge into the ET, the average number of edges per relevant ET bucket is 0, before the second edge it is 1/H, before the third it is 2/H and before the fourth it is 3/H (maintaining our assumption that E=4). Therefore

$$E_{etc} = (0 + 1/H + 2/H + 3/H) / 4$$
$$= 3 / (H * 2).$$

E_{aet}, the average number of edges in the Active Edge Table (AET) (or the average number of polygon edges that a scanline intersects,provided it intersects some), can be made equal to 2 if we assume that the majority of polygons used for building pictures are convex



1.The assumption that half of the polygons are clipped out is borrowed from [SuSp74].

Here are the environment parameter values that result from our instantiation

$Yres = 500$
$Xres = 500$
$Dc = 1$
$P_t = 1024$
$P_r = 512$
$W = \sqrt{(Yres * Xres * Dc) / P_r} = \sqrt{(500 * 500 * 1) / 512} = 22$
$H = W = 22$
$E = 4$
$E_{etf} = E / H = 4 / 22 = .18$
$E_{etc} = 3 / (H * 2) = .07$
$E_{set} = 2$

We shall next use these parameters to estimate the number of transputer cycles, hence the amount of time, that each of the stages of the pipeline would take to process our environment (called the stage's timing). In what follows multiplications arise from loops. The cost of the loop (in transputer cycles) given by the Transputer Estimator is multiplied by the estimate of the number of times the loop will be executed (which is expressed in terms of the environment parameters); both figures are given on the program listing in appendix 4.

5.2.1. Viewing Transformation Timing

The viewing transformation's timing can be expressed as

$T_v = P_t * (138 + E * 399)$
$= 1.8M$ transputer cycles
which would take .09 sec on INMOS T424-20 [INTR84].

5.2.2. Perspective Transformation Timing

Its timing is

$T_p = P_r * (138 + E * 439)$
$= 1M$ transputer cycles
which would take .05 sec on T424-20.

Note that since the perspective transformation is performed after clipping, the expression used for the number of polygons is P_r.

5.2.3. Clip Timing

As we are only interested in the rate of flow through the clipping pipeline, we should consider the timing of the first clipping stage which deals with the most complex environment. The timing of this stage is

$$T_C = P_t * (1213 + (E-1) * 979)$$
$$= 4.2M \text{ transputer cycles}$$
which would take .21 sec on T424-20.

5.2.4. HSE Timing

Before determining the timing of the scan converter, which is the main routine of our OCCAM implementation of the Z-buffer algorithm, we estimated the amount of time taken by each of the auxilliary procedures it uses by means of procedure calls

CLEAR	takes	$T_{CL} = 13686$ transputer cycles (tc)
INSERT.ET.EDGE	..	$T_{IN} = 192 + 307 + (E_{etc} / 2)*49 = 501$ tc
MOVE.ET.BUCKET.TO.AET	..	$T_{MO} = 36 + E_{etf}*195 = 71$ tc
UPDATE.AET	..	$T_{UP} = 34 + E_{aet}*103 = 240$ tc
BUBBLESORT[i]	..	$T_{SO} = 505$ tc
UPDATE.MIN.MAX.ET.BUCKET	..	$T_{MM} = 54$ tc

The cost of the scan converter is then given by

$$T_{SC} = P_r * (1034 + T_{CL} + (E - 2) * (139 + T_{IN} + T_{MM}) + 2 * T_{IN} + 2 * T_{MM} +$$
$$+ H * (44 + T_{MO} + E_{aet} * (414 + W * 152) + T_{UP} + T_{SO}))$$
$$= 103M \text{ transputer cycles}$$
which would take 5.2 sec on T424-20.

So, for our particular environment instantiation, the rate of flow through the HSE stage is $T_{SC} / T_C = 25$ times smaller than the next smallest rate of flow among the other stages of the pipeline.

i. The WHILE loops of BUBBLESORT are likely to be executed once only since the AET will rarely be out of order.

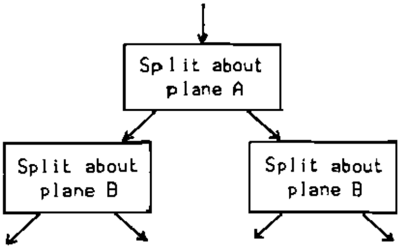
Chapter 6

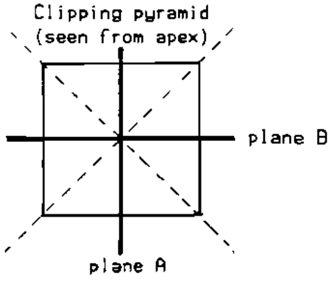
Alleviating the Bottleneck by a Splitter Tree

This chapter describes the Splitter Tree approach [Park80] to alleviating the bottleneck of the graphics pipeline imposed by HSE. A CSP specification is given and the optimal depth of the Splitter Tree is estimated.

6.1. The Splitter Tree

The performance of the bottleneck HSE stage of the graphics pipeline can be improved by splitting up the polygon stream coming down the pipeline into several substreams using a splitter tree [Park80] before the HSE stage. A number of HSE processes can then process separate data in parallel





Each node of the splitter tree is not much more than a clipper of the Sutherland-Hodgman type [Suth74], except that instead of discarding the polygons or parts of polygons that lie on the "outside" of the splitting plane, it uses the splitting plane to separate those polygons that lie on one side of it from those that lie on the other.

To achieve 2^n-way splitting, so that 2^n HSE processors can run in parallel, we need 2^n-1 splitter nodes arranged in a binary tree of depth n. For example to achieve 8-way splitting we need 2^3-1 = 7 splitter nodes arranged in 3 levels.

Since the splitting algorithm is a small modification of the clipping one, the rate of flow through splitter nodes should be much the same as the rate of flow through clipper stages (see sections 5.2.3. and 6.4.) and therefore the splitter tree should not impose any timing overhead (apart from an initial delay of the polygon stream by an amount of time proportional to the number of splitter tree levels).

Assumptions

The following conditions must hold if the splitter tree is to achieve its purpose

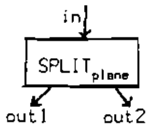
i. The polygons (or whatever our primitive objects) must be evenly distributed about the splitting planes.

ii. A new bottleneck must not be created when the outputs of the HSE processes are put back together in order to be displayed (see diagram of section 6.3.3.).

6.2. The Splitter Node

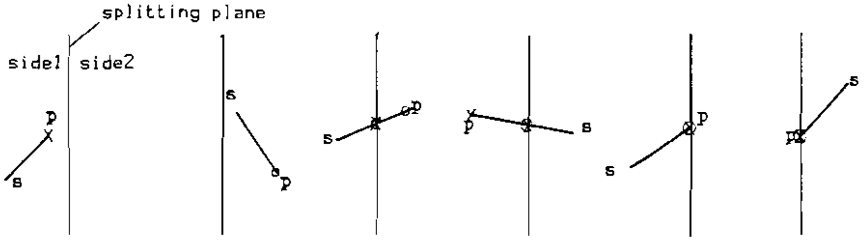
6.2.1. Description

A splitter node splits its input stream (which consists of polygon vertices) into two output streams; the decision as to which output stream a vertex goes to depending on which side of the node's splitting plane the vertex lies in



A Splitter Node

Several splitter nodes connected together in a binary tree fashion, constitute a splitter tree. Here is how each splitter node works. The vertices of the subject polygon are considered in pairs (s,p) in a clockwise traversal around the polygon, just as for clipping. There are 6 cases to be considered



× represent output vertices to channel out1
○ represent output vertices to channel out2

6.2.2. CSP specification

Each splitter node must satisfy

$$\begin{aligned}
&out1 \leq f^1_{plane}\ (in)\ \land \\
&out2 \leq f^2_{plane}\ (in)
\end{aligned}$$

where

$$\begin{aligned}
f^1_{plane}\ (\langle\rangle) &= \langle\rangle \\
f^1_{plane}\ (\langle p\rangle) &= \langle\rangle \\
f^1_{plane}\ (\langle s, p\rangle \frown rest) &= \langle p\rangle \frown f^1_{plane}(\langle p\rangle \frown rest), \\
&\qquad\qquad (\neg side2[i](plane, s)\ \&\ side1[i](plane, p) \\
&\qquad\qquad \lor\ onplane[ii](plane, p)) \\
&= \langle intersection(s, p, plane),\ p\rangle \frown f^1_{plane}(p \frown rest), \\
&\qquad\qquad side2(plane, s)\ \&\ side1(plane, p) \\
&= \langle intersection(s, p, plane)\rangle \frown f^1_{plane}(\langle p\rangle \frown rest), \\
&\qquad\qquad side1(plane, s)\ \&\ side2(plane, p) \\
&= f^1_{plane}(\langle p\rangle \frown rest), \\
&\qquad\qquad TRUE
\end{aligned}$$

and

$$\begin{aligned}
f^2_{plane}\ (\langle\rangle) &= \langle\rangle \\
f^2_{plane}\ (\langle p\rangle) &= \langle\rangle \\
f^2_{plane}\ (\langle s, p\rangle \frown rest) &= \langle p\rangle \frown f^2_{plane}(\langle p\rangle \frown rest), \\
&\qquad\qquad (\neg side1(plane, s)\ \&\ side2(plane, p) \\
&\qquad\qquad \lor\ onplane(plane, p)) \\
&= \langle intersection(s, p, plane),\ p\rangle \frown f^2_{plane}(p \frown rest), \\
&\qquad\qquad side1(plane, s)\ \&\ side2(plane, p) \\
&= \langle intersection(s, p, plane)\rangle \frown f^2_{plane}(\langle p\rangle \frown rest), \\
&\qquad\qquad side2(plane, s)\ \&\ side1(plane, p) \\
&= f^2_{plane}(\langle p\rangle \frown rest), \\
&\qquad\qquad TRUE
\end{aligned}$$

f^1_{plane} and f^2_{plane} recursively specify the relationship that must hold between the input and each of the two outputs of a splitter node.

---

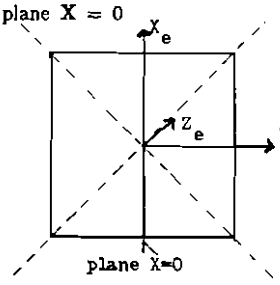
i. side1(plane,s) is a function identical to the function inside(plane,s) of clipping; but in splitting both sides of the splitting plane are treated equally and the name of this function is supposed to signify that. side1 delivers false if s is on the splitting plane. side2 is a similar function for the other side of the splitting plane.

ii. onplane(plane,p) is a function which tests if a point p is on the splitting plane by checking if p satisfies the splitting plane equation.

6.3. Combining the Splitter Tree with the Clipping pipeline

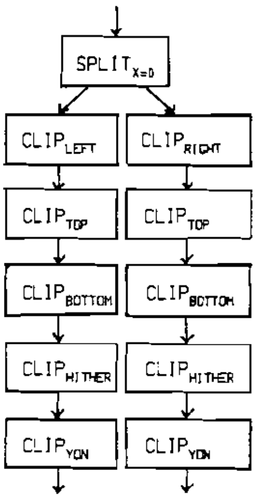
6.3.1. Description

Let's instantiate our splitter tree by assuming that it consists of only one level which splits about the plane **X** = 0



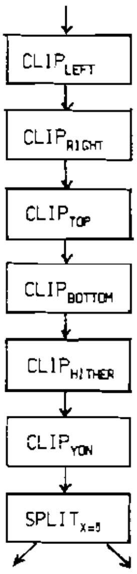
Clipping Pyramid (apex view)

There are two possible ways of combining our one level splitter tree with the clipping pipeline; either we split and then clip or we clip and then split



$\text{SPLIT}_{X=0}$
$\text{CLIP}_{\text{LEFT}}$
$\text{CLIP}_{\text{RIGHT}}$
CLIP_{TOP}
CLIP_{TOP}
$\text{CLIP}_{\text{BOTTOM}}$
$\text{CLIP}_{\text{BOTTOM}}$
$\text{CLIP}_{\text{HITHER}}$
$\text{CLIP}_{\text{HITHER}}$
CLIP_{YON}
CLIP_{YON}

SPLITTER/CLIPPER



$\text{CLIP}_{\text{LEFT}}$
$\text{CLIP}_{\text{RIGHT}}$
CLIP_{TOP}
$\text{CLIP}_{\text{BOTTOM}}$
$\text{CLIP}_{\text{HITHER}}$
CLIP_{YON}
$\text{SPLIT}_{X=0}$

CLIPPER/SPLITTER

At first the SPLITTER/CLIPPER seems faster; each polygon has to pass through only six stages compared to seven in the case of the CLIPPER/SPLITTER. However the CLIPPER/SPLITTER is preferable for two reasons

i. It uses fewer clipping processes; in this case six compared to ten of the SPLITTER/CLIPPER.

ii. The SPLITTER/CLIPPER is not really much faster. The rate at which the SPLITTER/CLIPPER processes polygons is determined by the speed of its splitter which deals with the most complex environment. Both designs process polygons at about the same rate; the only difference being that the SPLITTER/CLIPPER outputs its first polygon earlier by the amount of time it takes to clip one polygon against a plane. And this is insignificant if the polygon stream is long.

6.3.2. CSP Specification

Our clipping pipeline can be modelled by the CSP process

$$\mathrm{CLIP.PIPE} = \mathrm{CLIP}_{\mathrm{LEFT}} \gg \mathrm{CLIP}_{\mathrm{RIGHT}} \gg \mathrm{CLIP}_{\mathrm{TOP}} \gg \mathrm{CLIP}_{\mathrm{BOTTOM}} \gg \mathrm{CLIP}_{\mathrm{HITHER}} \gg \mathrm{CLIP}_{\mathrm{YON}}$$

$$\alpha(\mathrm{CLIP.PIPE}) = \{\mathrm{left}, \mathrm{right}\}$$

(see section 3.2.).

The one level splitter tree can be modelled by the process

$$\mathrm{SPLIT.TREE} = \mathrm{SPLIT}_{X=0}$$

$$\alpha(\mathrm{SPLIT.TREE}) = \{\mathrm{in},\ \mathrm{out1},\ \mathrm{out2}\}$$

We shall now combine them in a CLIPPER/SPLITTER fashion by renaming the <u>right</u> channel of CLIP.PIPE to <u>in</u> and then hiding it as it is an internal communication channel between the two processes

$$\mathrm{CLIP/SPLIT} = (\mathrm{in} \xrightarrow[\mathrm{right}]{} \mathrm{CLIP.PIPE} \parallel \mathrm{SPLIT.TREE}) \setminus \{\mathrm{in}\}$$

$$\alpha(\mathrm{CLIP/SPLIT}) = \{\mathrm{left},\ \mathrm{out1},\ \mathrm{out2}\}$$

From section 3.2. we know that

$$\mathrm{CLIP.PIPE\ sat\ right} \leqslant f_{\mathrm{CLIP}}(\mathrm{left})$$

where $f_{\mathrm{CLIP}} = f_{\mathrm{YON}} \cdot f_{\mathrm{HITHER}} \cdot f_{\mathrm{BOTTOM}} \cdot f_{\mathrm{TOP}} \cdot f_{\mathrm{RIGHT}} \cdot f_{\mathrm{LEFT}}$

Therefore

$$\mathrm{in} \xrightarrow[\mathrm{right}]{} \mathrm{CLIP.PIPE\ sat\ in} \leqslant f_{\mathrm{CLIP}}(\mathrm{left})$$

Also, from section 6.2.2. we know that

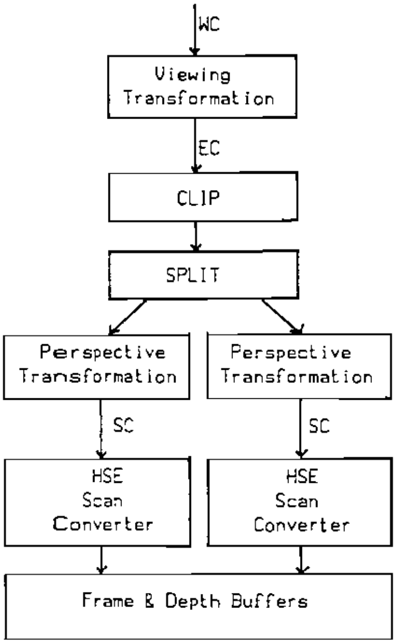
$$\begin{array}{l} \quad \mathrm{SPLIT.TREE} \\ \mathrm{sat} \\ \quad \mathrm{out1} \leqslant f^{1}_{X=0}(\mathrm{in}) \quad \wedge \\ \quad \mathrm{out2} \leqslant f^{2}_{X=0}(\mathrm{in}) \end{array}$$

It must therefore be the case that

$$\begin{array}{l} \quad (\mathrm{in} \xrightarrow[\mathrm{right}]{} \mathrm{CLIP.PIPE} \parallel \mathrm{SPLIT.TREE}) \setminus \{\mathrm{in}\} \\ \mathrm{sat} \\ \quad \mathrm{out1} \leqslant f^{1}_{X=0}(f_{\mathrm{CLIP}}(\mathrm{left})) \quad \wedge \\ \quad \mathrm{out2} \leqslant f^{2}_{X=0}(f_{\mathrm{CLIP}}(\mathrm{left})) \end{array}$$

since <u>in</u> is the only common channel of $\mathrm{in} \xrightarrow[\mathrm{right}]{}$ CLIP.PIPE and SPLIT.TREE.

Here is how our pipeline looks after the introduction of the splitter tree. A splitter tree of depth 1 is shown



This corresponds exactly to the structure of our OCCAM program.

6.4. Optimal Depth of the Splitter Tree

In order to determine the optimal depth of the splitter tree we shall extend the second model we used to compare the stages of the graphics pipeline (see section 5.2.). First of all we give the relationship between the environment parameters before and after a split. Since the environment is divided into two equal halves by a split, the area of the screen corresponding to each half of the object space (Yres * Xres) and the number of relevant polygons (P_r) reduce by half (assuming that the majority of the polygons are not cut by the splitting plane). The rest of the environment parameters are not affected by a split.

Here is how the environment parameters after a split (shown primed) relate to the ones before the split

$$\begin{aligned}
&Yres' = Yres / \gamma \qquad\qquad \text{where } 1<\gamma<2\\
&Xres' = (Xres * \gamma) / 2\\
&\qquad\qquad (\text{so that } Yres * Xres = 2 * (Yres' * Xres'))\\
&Dc' = Dc\\
&P_r' = P_r / 2\\
&W' = \sqrt{(Yres' * Xres' * Dc')/P_r'} = \sqrt{(((Yres * Xres)/2) * Dc) / (P_r/2)} = W\\
&H' = H \qquad \text{similarly}\\
&E' = E\\
&E_{str}' = E' / H' = E / H = E_{str}\\
&E_{stc}' = 3 / (H' * 2) = 3 / (H * 2) = E_{stc}\\
&E_{sst}' = E_{sst}
\end{aligned}$$

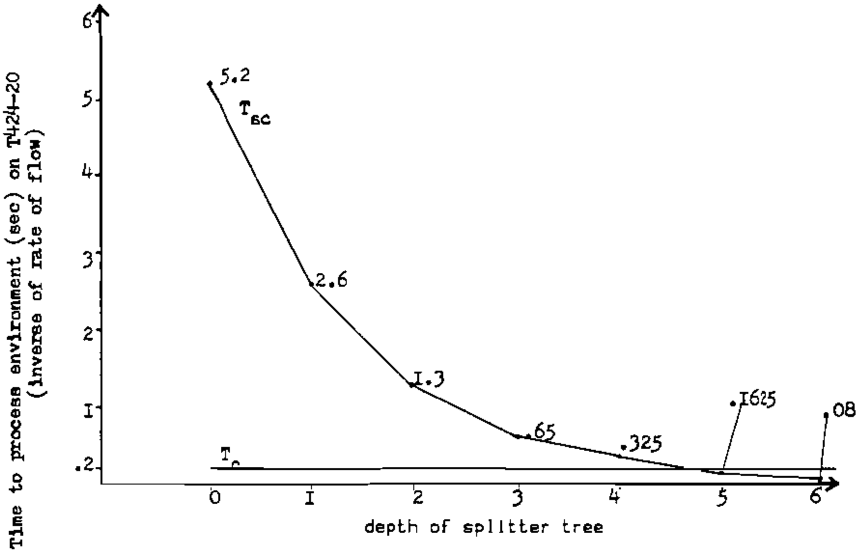
Note that P_t, the number of polygons before clipping, is irrelevant since splitting is performed after clipping.

To estimate the optimal depth of the splitter tree we reason as follows. For each new layer we add to the splitter tree, the number of its leaves (the HSE processors) is doubled (assuming a binary splitter tree). Hence the rate of flow through the HSE layer is also doubled since the environment is evenly distributed about the splitting planes.

What's the limit to how fast we can make the pipeline as a whole? The rate of flow through a pipeline is only as large as the smallest rate of flow over all its stages. Hence we should only increase the depth of the splitter tree until the rate of flow through the HSE layer is equal to the smallest rate of flow over all the other stages. We must consider therefore the rate of flow through the transformation stages, clipping and the root node of the splitter tree which deals with the most complex environment. We have derived timings (the time taken to process our environment i.e. the inverse of the rate of flow) for the viewing and perspective transformations as well as for clipping (T_V, T_P and T_C) in section 5.2. The timing for the first splitting node is

$$T_{SP} = P_r * (1430 + (E-1) * 1131)$$
$$= 2.5M \text{ transputer cycles}$$
which would take .12 sec on T424-20.

Now $\max(T_V,T_P,T_C,T_{SP}) = T_C$. Hence clipping has the smallest rate of flow over all the stages of our pipeline except HSE. The following graph shows the relationship between the depth of the splitter tree and the timings of HSE (T_{SC}) and clip (T_C).



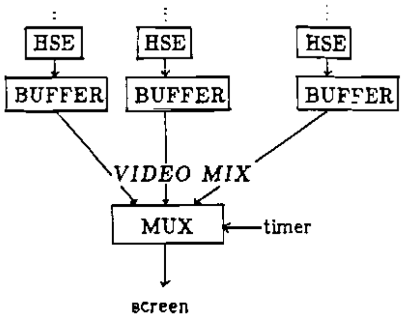
Our splitter tree should therefore be of depth 5. This means 32-way splitting requiring 32 HSE processors.

6.5. Is the Transputer Link Data Rate Adequate ?

As in any system of parallel processors, we must ensure that the amount of information that must be communicated between the processors can be handled by the communication links.

Let's assume that each of our processes is running on a separate transputer, using on-chip memory only and that we are processing an environment of the complexity described in section 5.2. The restriction on the rate of flow of polygons through our pipeline imposed by the processing speed of the stages is about 5000 (pre clipping) polygons / sec (implied in section 6.4). Does the available link data rate allow this rate of flow or does it impose a stricter limit ?

The highest communication rate is likely to be required either on the link going into the first stage of the clipper or on the links coming out of each of the HSE Scan converters (see figure of section 6.3.3). This is because the clipper is likely to reduce the number of polygons going down the pipeline, hence the amount of information that has to be communicated, but the amount of information is increased again by the HSE Scan Converters which convert the polygon descriptions into pixels. One might ask : And how will the massive pixel outputs of all the 32 HSE Scan Converter processes be put back together in order to be displayed ? The answer offered by today's technology is Time Multiplexed Video Mixing of the video outputs of the Frame Buffers (which are Dual Ported Video RAMs). In other words the image is put back together in video, the very last step before being diaplayed. Of course the Frame Buffers must have some intelligence in order to deal with the pixel descriptions they receive from the HSE Scan Converters as dictated by the Z-buffer algorithm. A microprocessor and the Depth and Frame buffers would probably be placed where BUFFER is shown below



Let's estimate the data rate required on the two links mentioned above

A. Clipper Input Link.
We have assumed that each polygon has an average of 4 vertices (E = 4), each described by 3 coordinates occupying a total of 48 bytes (assuming that each coordinate is a 4 byte integer). In addition each polygon has 1 byte to describe its colour and 1 control byte, making a total of 50 bytes / polygon. At 5000 polygons / sec, the required data rate is 250 Kbytes / sec.

B. HSE Scan Converter Output Link.
Maintaining our assumption that the clipper halves the number of polygons (see section 5.2), the 32 HSE Scan Converter processes have 2500 polygons / sec to deal with or about 80 polygons / sec each, since we have assumed an even distribution of polygons about the splitting planes. With an average polygon area of 484 pixels (W = H = 22 pixels, section 5.2), each HSE Scan Converter has to output about 40,000 pixel descriptions / sec. Each pixel description consists of the x and y screen coordinates of the pixel occupying 2 bytes each, the depth of the relevant polygon occupying 2 bytes¹ and the colour of the polygon occupying 1 byte; that makes 7 bytes / pixel description. A data rate of 280 Kbytes / sec is thus required. (With appropriate coding we can avoid the transmission of redundant information. The colour value need only be transmitted once per polygon for example. We could use data reduction if the data rate of this link was inadequate; the extra computation needed should be taken into account in the timing estimate for the HSE Scan Converter).

---

1. HSE takes place in image space (after the Perspective Transformation has been performed). It therefore uses either Physical Device Coordinates or Normalised Device Coordinates (which will be transformed into the Physical Device Coordinate systems of several devices). The useful range of such coordinates is limited by the resolution of the screen and 2 bytes / coordinate is more than enough to address even the highest resolution screens. The accuracy of the z-coordinate is also assumed to be reduced to 2 bytes by the (depth preserving) Perspective Transformation in order to decrease the size of the Z-buffer.

The factors that can limit a link's data rate are

1. Transmission Time (1.1 μsec / (byte + control bits))

2. Scheduling Overhead (6 cycles / communication)

3. Memory Contention between Links / Processor

4. Rendezvous delay

The average time taken by a communication is given by the estimator as 26 cycles, allowing a maximum data rate of about 770 Kbytes / sec for a 50 nsec cycle (taking into account the worst case of single byte communications). This estimate takes into account limiting factors 1. and 2. and the maximum data rate it allows would be sufficient for our purposes if those were the only limiting factors.

Memory Contention between links and processor is irrelevant if we only have one process / transputer as either the process is executing and all the memory cycles are available to it, or the process has been descheduled in order for a communication to take place in which case the link concerned has all the memory cycles available to it[i].

If one of the two processes taking part in a communication arrives late at the Rendezvous then the communication takes more than 26 cycles for the process that arrived first. However the slowest stages of our pipeline (HSE and CLIP) are balanced and the (fast) stages between them essentially act as buffers. It is therefore likely that the slow stages will make the fast stages wait for them at the Rendezvous with the effect of lengthening their communication time (so that the faster stages will run at the pace of the slowest ones) and there should be no overall delay.

From the above discussion we can conclude that the estimator's 26 cycle communication time takes into account the effective factors that limit the data rate and therefore the available link data rate should be sufficient to handle the communication between the stages of our pipeline under the assumptions we made. Furthermore it seems that there is scope for optimising our code in order to increase its processing capability and take advantage of the spare link data rate.

---

i. If we put more than one process per transputer - for example several clipping stages per transputer, excluding the "bottleneck" first clipping stage of course - then memory contention between links and processor must be taken into account before deciding on the allocation of processes to transputers. One can start at the fact that if all links are working flat out, they request 1 memory cycle every 325 nsec. This corresponds to 15 % of the total number of memory cycles for a cycle time of 50 nsec. In other words there will be a memory contention for 15 % of the memory references made by the processor.

Chapter 7

Conclusions and Further Work

From Operation Specifications to Algorithm Specifications

We gained a clear idea of the clipping and hidden surface elimination operations by specifying them in Z in chapter 2.

We then proceeded to specify algorithms which would implement the above operations in CSP (chapters 3 & 4) and finally we coded these algorithms in OCCAM.

Since Z and CSP are formal specification notations, we could formally relate the specifications of the operations to the specifications of the algorithms by the rules of data refinement, but this was outside the scope and time limits of this project.

Even Distribution of Object Space Primitives

It is an essential assumption of the splitter tree that the polygons (or other primitives) are evenly distributed about the splitting planes, else some of the HSE processors will be idle.

Instead of assigning a contiguous area of object space to each HSE processor we could assign to it arbitrary non contiguous areas by appropriate splitting. The workload is then likely to be more evenly spread among the HSE processors. But would the extra splitting that this implies as well as the cost of reconstructing the image at the other end be cost-effective ?

Real Time ?

From the graph of section 6.4. it is evident that we could not hope to achieve a rate of flow through our current pipeline of more than one frame (of the complexity described in chapter 5) per .2 seconds since that is the value of the clipping overhead T_C. For our particular environment therefore, we have not achieved real time performance i.e. a rate of flow of at least 25 frames per second. Our bottleneck has moved further up the pipeline to the first stage of the clipper. If we want to further improve our timing we have to divide the clipper's task. One possibility would be to divide our 3D polygon data base amongst several pipelines like the one introduced in this project.

Transputer Implementation

In order to decide how to allocate the processes that constitute the stages of the pipeline amongst transputers we must take into account their speed and code size. Here are some hints (refer to diagram of section 6.3.3.). For a system with a splitter tree of optimal depth, we expect that the rate of flow through the HSE layer will be equal to that through the first clipping stage and that these will be the stages with the smallest rate of flow of our pipeline (see chapter 6). Hence each HSE stage as well as the first clipping stage should reside on their own transputers to avoid any timing penalty. The pair of perspective transformation processes that emerge from each leaf of the splitter tree could be incorporated in the transputer of the splitter tree leaf since the leaf splitting node will be dealing with an environment that is much simpler than the original (16 times simpler for a splitter tree of depth 5).

We should also consider the possibility of a tertiary splitting tree so that all the channels of a transputer are utilised by a splitter tree node. This implies that each splitter tree node performs 3-way splitting about 2 planes. The root node should remain a binary splitting node to avoid any timing penalty.

Suitability of OCCAM

Its features were handy in expressing the parallel combination of our algorithms and the non-determinism involved in the "buffer" process of the HSE which has no means of knowing which scan converter process to expect the next input from (see diagram of section 6.3.3 and OCCAM code in appendix 4).

However OCCAM's lack of data structures meant that we have had to implement buckets and lists using 1D arrays (see appendix 3).

The use of real numbers has been avoided for two reasons

i. speed
ii. unavailability

Scaling has been used instead.

[Fole82] Foley J.D. and A. Van Dam,
"Fundamentals of Interactive Computer Graphics",
Addison Wesley, 1982.

[Hoar83] Hoare C.A.R.,
"Notes on Communicating Sequential Processes",
Technical Monograph PRG-33, August 1983.

[INMO84] INMOS Limited,
"OCCAM Programming Manual",
Prentice Hall International, 1984.

[INTR84] INMOS Limited,
"IMS T424 Transputer Preliminary Reference Manual",
August 1984.

[Newm79] Newman W.M. and R.F.Sproul,
"Principles of Interactive Computer Graphics",
McGraw Hill, 1979.

[Park85] Park C.S.,
"Interactive Microcomputer Graphics",
Addison Wesley, 1985.

[Park80] Parke F.I.,
"Simulation and Expected Performance Analysis of
Multiple Processor Z-buffer Systems",
Computer Graphics 14 (ACM-SIGGRAPH),
N° 3 (July 1980), pp. 48-56.

[Suth74] Sutherland I.E. and G.W. Hodgman,
"Reentrant Polygon Clipping",
Communications of the ACM, 17(1),
Jannary 1974, pp. 32-42.

[SuSp74] Sutherland I.E., Sproul R.F. and R.A. Schumacker,
"A Characterization of Ten Hidden Surface Algorithms",
Computing Surveys, March 1974, pp. 1-55.

[Z85] Notes for a Z Handbook, Draft 1,
Programming Research Group, 1985.

Appendix 1

Picture Format

This appendix describes the syntax used in our OCCAM implementation to define pictures in terms of polygons.
 Each static picture, called frame, consists of a set of objects
described in terms of polygons. All polygons go through the same pipeline, so a way of separating their coordinates is needed; for this reason the special value NEXT.POLYGON is inserted between the coordinates of successive polygons

```
<polygon> ::= <colour> {,<x>,<y>,<z>}*

<frame>   ::= <polygon> {,NEXT.POLYGON, <polygon>}*
```

Notice that a polygon can be empty (i.e. consist of a colour only), or consist of only one or two vertices. Such edgy forms of polygon can result from extreme cases in the splitter process or incorrect input and are eventually discarded.
 A sequence of frames can be used for animation. A movie is
a sequence of frames separated by the special value NEXT.FRAME

```
<movie> ::= <frame> {,NEXT.FRAME, <frame>}* END
```

The special values are used to reset the appropriate data structures before processing the next frame/polygon. For example the scan converter process uses NEXT.POLYGON as a signal to clear the Edge Table and the Active Edge Table before processing the next polygon. The NEXT.FRAME value can be used by the display controller to clear the screen.

A.2.1. Determining whether a Point is on the "Inside" of a Plane

From chapter 1 we know that we can determine whether a point is on the "inside" of a clipping plane by comparing the appropriate coordinate of the point with w (= z_e * (s/d)) which can be calculated as soon as the eye coordinates of a point are known. In our OCCAM implementation w is calculated once and for all in the viewing transformation stage and kept as the fourth component of the coordinates of a point $[x_e, y_e, z_e, w]$.

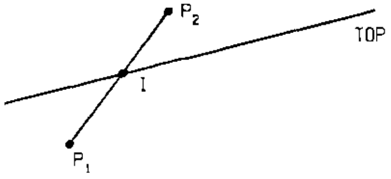
For the hither and yon clipping planes we don't need to compare against w as these planes are perpendicular to the Z_e axis. Here is a summary of the conditions that a point must satisfy in order to be on the "inside" of each of the six clipping planes

$P(x_e, y_e, z_e)$

Is Inside	If	
LEFT	$x_e > -w$	
RIGHT	$x_e < w$	
TOP	$y_e < w$	
BOTTOM	$y_e > -w$	
HITHER	$z_e > k_1$	$z_e = k_1$ is the hither clipping plane
YON	$z_e < k_2$	$z_e = k_2$ is the yon clipping plane

A.2.2. Calculating the Intersection of a Line Segment and a Plane

This only need be calculated if the line segment actually crosses the plane. The method used is the one suggested in [Suth74]. Consider the top clipping plane and a pair of points $P_1(x_1,y_1,z_1)$ and $P_2(x_2,y_2,z_2)$ on either side of it



Let α be the ratio $|P_1I| / |P_1P_2|$. Then the coordinates of I can be computed as

$$\overleftarrow{I} = \overleftarrow{P_1} + \alpha(\overleftarrow{P_2} - \overleftarrow{P_1})$$

by noting that $\overleftarrow{P_1}I = \alpha(\overleftarrow{P_2} - \overleftarrow{P_1})$.

In order to estimate the ratio α for the top clipping plane, we need a measure of the distance of P_1 and P_2 from that plane. $(y - w)$ is a suitable measure. Since this has opposite sign for P_1 and P_2, the ratio α is given by

$$\alpha = (y_1 - w) / ((y_1 - w) - (y_2 - w)).$$

α can be calculated similarly for the other clipping planes. In the case of the hither and yon planes, the calculation of α is simpler since the difference between the z-coordinates of P_1 and P_2 and the value of z at the plane can be used as the distance measure.

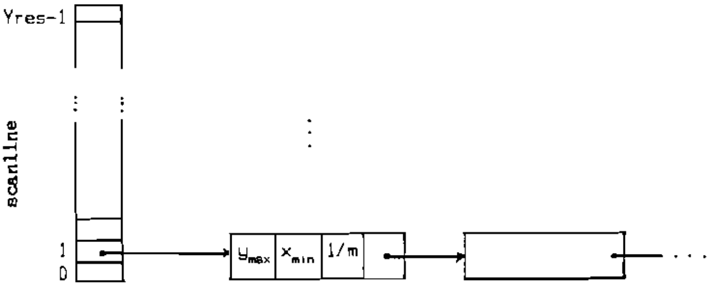
α can take values between 0 and 1 but in order to avoid the use of reals in our OCCAM program, we multiply the dividend in the expression for α by a "scale factor".

Notice that the divisor in the above expression is guaranteed to be non-zero by the fact that P_1 and P_2 lie on opposite sides of the plane.

A.3.1. Data Structures

The following data structures have had to be implemented in OCCAM in order to be used by the scan converter of the HSE algorithm [Fole82]

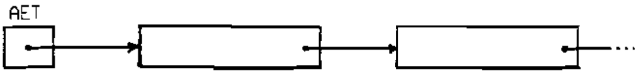
Edge Table (ET), organised as an array of buckets (one per scanline), to contain the edges of the polygon to be scan converted



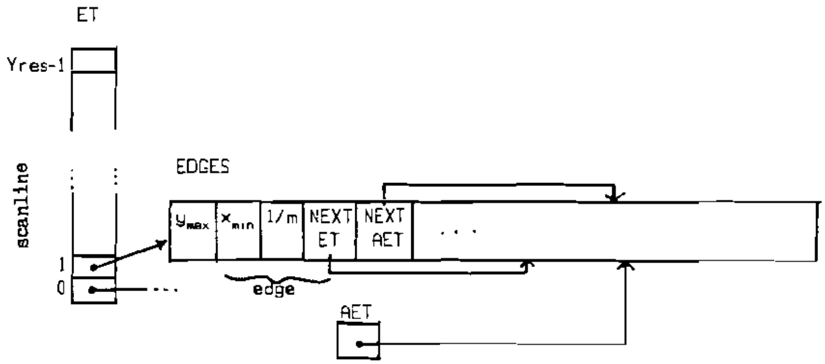
Information about each edge is kept in the bucket that corresponds to the scanline of its minimum y coordinate. The information kept for each edge is

- its maximum y coordinate (y_{max})
- the x coordinate corresponding to its minimum y coordinate (x_{min})
- its inverse slope (1/m)

Active Edge Table (AET), organised as a simple list, to contain the edges that the current scanline intersects



The ET and AET are implemented in OCCAM using a large 1D array to store the information about the edges. Another array acts as the bucket pointers for the ET. Edges belonging to the same bncket are linked together. An integer variable points to the first edge of the AET and the edges of the AET are also linked together



A.3.2. Operations

The following (specialised) operations on the above data structures were implemented in order to be used by the scan converter

i. CLEAR
{Initialise the ET and the AET to empty }

ii INSERT.ET.EDGE
{Insert an edge into the appropriate bucket of the ET }

iii. MOVE.ET.BUCKET.TO.AET
{Remove a bucket of edges from the ET and
insert them into the AET without destroying the ordering of the AET (on x_{min}) }

iv. UPDATE.AET
{Update the edges of the AET before processing the
next scanline. In other words remove from the AET those edges whose y_{max} is equal to the last scanline processed and update the x-intercept of the rest of the edges (x_{min}) for the next scanline }

v. BUBBLESORT
{Used to sort the AET in case it became out of order
during updating. The AET is likely to be sorted and in that case bubblesort performs well }

The following incremental calculations were used in order to save time during scan conversion

i. X-intercept of an Edge with the Next Scanline

If an edge of slope m intercepts scanline i at x_i $(y=i)$, it must intercept scanline i+1 at $x_i + 1/m$. This calculation is used in UPDATE.AET.

ii. Depth of a Polygon at the Next Pixel

The Z-buffer HSE algorithm requires that the depth of a polygon be estimated at each of the pixels within it. This can be done by solving the equation of the polygon's plane

```
a*x + b*y + c*z + d = 0
```

for z. But this calculation requires 1 division, 2 multiplications and 2 subtractions per pixel. Instead we observe that if the depth of a polygon at pixel (x,y) is z, then its depth at the next pixel on the current scanline (x+1,y) is

```
z + (((-d -a*(x+1) -b*y) / c) - ((-d -a*x -b*y) / c)) =

z - (a / c)
```

A.3.4. Calculation of the Plane Equation

The plane equation of a polygon is determined using the method suggested by Martin Newell and described in [SuSp74].

The coefficients a,b and c of the plane equation are determined as follows

$$a = \Sigma\ (y_i - y_j) * (z_i + z_j)$$
$$b = \Sigma\ (z_i - z_j) * (x_i + x_j)$$
$$c = \Sigma\ (x_i - x_j) * (y_i + y_j)$$

where

(x_i, y_i, z_i) is the i^{th} polygon vertex

$i = 1..\#(\text{vertices in polygon})$

$j = (i+1 \text{ if } i < \#(\text{vertices in polygon}) \text{ else } 1)$

In our implementation we require that the vertices of a polygon be coplanar and only take into account 3 of the vertices in determining the plane coefficients.

Having determined a,b and c, the d coefficient is found using the coordinates of a vertex to solve the plane equation for d.

Appendix 4

OCCAM Implementation

```
-- Constant Declarations
DEF SCREEN.HEIGHT = 21,    --must be odd
    SCREEN.WIDTH  = 21,
    HALF.SH       = 10,    --(SCREEN.HEIGHT - 1) / 2
    HALF.SW       = 10,    --(SCREEN.WIDTH  - 1) / 2
    MAX.DEPTH     = max.int,
    BACK.GND.COLOUR = '*s',
    MAX.EXPECTED  = 200,
    SCALE.FACTOR  = 1024,
    s             = 10,     --Screen Size / 2
    d             = 16,     --distance from E.C. origin to screen plane
    D             = 20,     --Distance from E.C. origin to W.C. origin
    K1            = 1,      --Z=K1 is the Hither clipping plane (E.C.)
    K2            = 30,     --Z=K2 is the Yon    clipping plane (E.C.)
    NEXT.POLYGON = min.int,
    NEXT.FRAME   = min.int + 1,
    END          = min.int + 2,
    NIL          = min.int + 3:
CHAN screen AT Screen.Index:
```

```
-- Inputter
PROC INPUTTER (CHAN OUT)=
  SEQ
    OUT ! 'O': -8;-4;0;  0;7;0;  8;-4;0;  NEXT.POLYGON
    OUT ! 'I': -9;5;4;  -7;7;4;  7;-7;-4;  5;-9;-4;  END:
```

```
--  Viewing Transformation
--Transform from World to Eye Coordinates
--and scale them up * SCALE.FACTOR to
--avoid use of reals.

PROC VIEWING.TRANSFORMATION( CHAN in, out,
                             VALUE s,
                                   d,
                                   D )=

  VAR x, y, z, colour, t:
  SEQ
    t := (s * SCALE.FACTOR) / d
    x := 0
    WHILE x <> END
      SEQ
        in ? colour
        out ! colour
        in ? x
        WHILE x > (min.int + 2)      --While not a control value
          SEQ
            in ? y; z
            out !  x * SCALE.FACTOR             --Xe * SCALE.FACTOR
            out !  y * SCALE.FACTOR             --Ye * SCALE.FACTOR
            out !  (D - z) * SCALE.FACTOR       --Ze * SCALE.FACTOR
            out !  t * (D - z)                  --W  * SCALE.FACTOR
            in ? x
        out ! x:
```

(NEX)number of times executed: P_t
(CEX)cost per exec.: 138 transp. t_{cycles}

NEX: 8
CEX: 399 tc

```
--  Intersection

--calculate the intersection of a plane and edge.
--alpha = ( (dist. from  point P1 to plane)/
--          (dist. from P1 to P2) )* SCALE.FACTOR
--(xi,yi,zi) are the intersection coordinates

PROC INTERSECTION( VALUE x1, y1, z1,
                         x2, y2, z2,
                         alpha,
                   VAR xi, yi, zi)=

  SEQ
    xi := x1 + (alpha * ((x2 - x1) / SCALE.FACTOR))
    yi := y1 + (alpha * ((y2 - y1) / SCALE.FACTOR))
    zi := z1 + (alpha * ((z2 - z1) / SCALE.FACTOR)):
```

```
--  Left    Clipper
--All incoming coordinates are * SCALE.FACTOR

PROC CLIP.LEFT (CHAN left, right,
                VALUE s, d)=

  VAR xfirst,yfirst,zfirst,   --first polygon vertex
      wfirst,
      xs,ys,zs,               --beginning of each edge
      xp,yp,zp,               --end of each edge
      xi,yi,zi,               --intersection coordinates
      ws,wp,wi,               -- w = (s / d) * z
      FIRST.POINT.INSIDE,
      SECOND.POINT.INSIDE,
      alpha,
      colour,
      t:                      -- t = s / d

  SEQ
    t := (s * SCALE.FACTOR) / d          --calculate s/d * SCALE.FACTOR
    xp := 0
    WHILE xp <> END
      SEQ
        left ? colour
        right ! colour
        left ? xs
        IF
          xs <= (min.int + 2)     --a control value; the polygon is null
            SEQ
              right ! xs
              xp := xs         --to terminate outer loop if xs = END
          TRUE
            SEQ
              left ? ys; zs; ws
              xfirst := xs
              yfirst := ys
              zfirst := zs
              wfirst := ws
              IF
                (xs + ws) >= 0
                  FIRST.POINT.INSIDE := TRUE
                TRUE
                  FIRST.POINT.INSIDE := FALSE
              left ? xp
              IF
                xp > (min.int + 2)
                  left ? yp; zp; wp
                TRUE
                  SKIP
              WHILE xp > (min.int + 2)          --While xp ~in {NEXT.POLYGON,
                SEQ                             --NEXT.FRAME, END}
```

NEX;P+ CEX;1213

NEX;E-I CEX;979

```
    IF
      (xp + wp) >= 0
        SECOND.POINT.INSIDE := TRUE
      TRUE
        SECOND.POINT.INSIDE := FALSE
    IF
      FIRST.POINT.INSIDE AND SECOND.POINT.INSIDE
        right ! xp; yp; zp; wp
      FIRST.POINT.INSIDE AND (NOT SECOND.POINT.INSIDE)
        SEQ
          alpha := ((xs + ws) * SCALE.FACTOR)
          / ((xs + ws) - (xp+ wp))

          INTERSECTION(xs,ys,zs, xp,yp,zp, alpha, xi,yi,zi)
          wi := t * (zi / SCALE.FACTOR)
          right ! xi; yi; zi; wi
      (NOT FIRST.POINT.INSIDE) AND SECOND.POINT.INSIDE
        SEQ
          alpha := ((xs + ws) * SCALE.FACTOR)
          / ((xs + ws) - (xp+ wp))

          INTERSECTION(xs,ys,zs, xp,yp,zp, alpha, xi,yi,zi)
          wi := t * (zi / SCALE.FACTOR)
          right ! xi; yi; zi; wi
          right ! xp; yp; zp; wp
      TRUE
        SKIP
    xs := xp
    ys := yp
    zs := zp
    ws := wp
    FIRST.POINT.INSIDE := SECOND.POINT.INSIDE
    left ? xp
    IF
      xp > (min.int + 2)     --not a control value
        left ? yp; zp; wp
      TRUE
        SKIP
--process last edge using saved vertex
IF
  (xfirst + wfirst) >= 0
    SECOND.POINT.INSIDE := TRUE
  TRUE
    SECOND.POINT.INSIDE := FALSE
IF
  FIRST.POINT.INSIDE AND SECOND.POINT.INSIDE
    right ! xfirst; yfirst; zfirst; wfirst
  FIRST.POINT.INSIDE AND (NOT SECOND.POINT.INSIDE)
    SEQ
      alpha := ((xs + ws) * SCALE.FACTOR) /
      ((xs + ws) - (xfirst + wfirst))
      INTERSECTION(xs,ys,zs, xfirst,yfirst,zfirst,
      alpha, xi,yi,zi)

      wi := t * (zi / SCALE.FACTOR)
      right ! xi; yi; zi; wi
  (NOT FIRST.POINT.INSIDE) AND SECOND.POINT.INSIDE
    SEQ
      alpha := ((xs + ws) * SCALE.FACTOR) /
      ((xs + ws) - (xfirst + wfirst))

      INTERSECTION(xs,ys,zs, xfirst,yfirst,zfirst,
      alpha, xi,yi,zi)

      wi := t * (zi / SCALE.FACTOR)
      right ! xi; yi; zi; wi
      right ! xfirst; yfirst; zfirst; wfirst
  TRUE
    SKIP
right ! xp:
```

```
-- Right  Clipper
--All incoming coordinates are * SCALE.FACTOR

PROC CLIP.RIGHT (CHAN left, right,
                 VALUE s, d)=

  VAR xfirst,yfirst,zfirst,  --first polygon vertex
      wfirst,
      xs,ys,zs,              --beginning of each edge
      xp,yp,zp,              --end of each edge
      xi,yi,zi,              --intersection coordinates
      ws,wp,wi,              -- w = (s / d) * z
      FIRST.POINT.INSIDE,
      SECOND.POINT.INSIDE,
      alpha,
      colour,
      t:                     -- t = s / d

  SEQ
    t := (s * SCALE.FACTOR) / d          --calculate s/d * SCALE.FACTOR
    xp := 0
    WHILE xp <> END
      SEQ
        left ? colour
        right ! colour
        left ? xs
        IF
          xs <= (min.int + 2)     --a control value; the polygon is null
            SEQ
              right ! xs
              xp  := xs        --to terminate outer loop if xs = END
          TRUE
            SEQ
              left ? ys; zs; ws
              xfirst := xs
              yfirst := ys
              zfirst := zs
              wfirst := ws
              IF
                (xs - ws) <= 0
                  FIRST.POINT.INSIDE := TRUE
                TRUE
                  FIRST.POINT.INSIDE := FALSE
              left ? xp
              IF
                xp > (min.int + 2)
                  left ? yp; zp; wp
                TRUE
                  SKIP
              WHILE xp > (min.int + 2)          --While xp ~in {NEXT.POLYGON,
                SEQ                             --NEXT.FRAME, END}
        IF
          (xp - wp) <= 0
            SECOND.POINT.INSIDE := TRUE
          TRUE
            SECOND.POINT.INSIDE := FALSE
        IF
          FIRST.POINT.INSIDE AND SECOND.POINT.INSIDE
            right ! xp; yp; zp; wp
          FIRST.POINT.INSIDE AND (NOT SECOND.POINT.INSIDE)
            SEQ
              alpha := ((xs - ws) * SCALE.FACTOR)
              / ((xs - ws) - (xp- wp))

              INTERSECTION(xs,ys,zs, xp,yp,zp, alpha, xi,yi,zi)
              wi := t * (zi / SCALE.FACTOR)
              right ! xi; yi; zi; wi
          (NOT FIRST.POINT.INSIDE) AND SECOND.POINT.INSIDE
            SEQ
              alpha := ((xs - ws) * SCALE.FACTOR)
              / ((xs - ws) - (xp- wp))

              INTERSECTION(xs,ys,zs, xp,yp,zp, alpha, xi,yi,zi)
              wi := t * (zi / SCALE.FACTOR)
              right ! xi; yi; zi; wi
              right ! xp; yp; zp; wp
          TRUE
            SKIP
        xs := xp
        ys := yp
        zs := zp
        ws := wp
        FIRST.POINT.INSIDE := SECOND.POINT.INSIDE
        left ? xp
        IF
          xp > (min.int + 2)     --not a control value
            left ? yp; zp; wp
          TRUE
            SKIP
    --process last edge using saved vertex
    IF
      (xfirst - wfirst) <= 0
        SECOND.POINT.INSIDE := TRUE
      TRUE
        SECOND.POINT.INSIDE := FALSE
    IF
      FIRST.POINT.INSIDE AND SECOND.POINT.INSIDE
        right ! xfirst; yfirst; zfirst; wfirst
      FIRST.POINT.INSIDE AND (NOT SECOND.POINT.INSIDE)
        SEQ
          alpha := ((xs - ws) * SCALE.FACTOR) /
          ((xs - ws) - (xfirst - wfirst))
        INTERSECTION(xs,ys,zs, xfirst,yfirst,zfirst,
        alpha, xi,yi,zi)

        wi := t * (zi / SCALE.FACTOR)
        right ! xi; yi; zi; wi
    (NOT FIRST.POINT.INSIDE) AND SECOND.POINT.INSIDE
      SEQ
        alpha := ((xs - ws) * SCALE.FACTOR) /
        ((xs - ws) - (xfirst - wfirst))

        INTERSECTION(xs,ys,zs, xfirst,yfirst,zfirst,
        alpha, xi,yi,zi)

        wi := t * (zi / SCALE.FACTOR)
        right ! xi; yi; zi; wi
        right ! xfirst; yfirst; zfirst; wfirst
    TRUE
      SKIP
right ! xp;
```

```
-- Top     Clipper
--All incoming coordinates are * SCALE.FACTOR

PROC CLIP.TOP (CHAN left, right,
               VALUE s, d)=

  VAR xfirst,yfirst,zfirst,  --first polygon vertex
      wfirst,
      xs,ys,zs,              --beginning of each edge
      xp,yp,zp,              --end of each edge
      xi,yi,zi,              --intersection coordinates
      ws,wp,wi,              -- w = (s / d) * z
      FIRST.POINT.INSIDE,
      SECOND.POINT.INSIDE,
      alpha,
      colour,
      t:                     -- t = s / d

  SEQ
    t := (s*SCALE.FACTOR) / d          --calculate s/d * SCALE.FACTOR
    xp := 0
    WHILE xp <> END
      SEQ
        left ? colour
        right ! colour
        left ? xs
        IF
          xs <= (min.int + 2)      --a control value; the polygon is null
            SEQ
              right ! xs
              xp := xs         --to terminate outer loop if xs = END
          TRUE
            SEQ
              left ? ys; zs; ws
              xfirst := xs
              yfirst := ys
              zfirst := zs
              wfirst := ws
              IF
                (ys - ws) <= 0
                  FIRST.POINT.INSIDE := TRUE
                TRUE
                  FIRST.POINT.INSIDE := FALSE
              left ? xp
              IF
                xp > (min.int + 2)
                  left ? yp; zp; wp
                TRUE
                  SKIP
              WHILE xp > (min.int + 2)        --While xp ~in {NEXT.POLYGON,
                SEQ                           --NEXT.FRAME, END}
    IF
      (yp - wp) <= 0
        SECOND.POINT.INSIDE := TRUE
      TRUE
        SECOND.POINT.INSIDE := FALSE
    IF
      FIRST.POINT.INSIDE AND SECOND.POINT.INSIDE
        right ! xp; yp; zp; wp
      FIRST.POINT.INSIDE AND (NOT SECOND.POINT.INSIDE)
        SEQ
          alpha := ((ys - ws) * SCALE.FACTOR)
          / ((ys - ws) - (yp - wp))

          INTERSECTION(xs,ys,zs, xp,yp,zp, alpha. xi,yi.zi)
          wi := t * (zi / SCALE.FACTOR)
          right ! xi; yi; zi; wi
      (NOT FIRST.POINT.INSIDE) AND SECOND.POINT.INSIDE
        SEQ
          alpha := ((ys - ws) * SCALE.FACTOR)
          / ((ys - ws) - (yp- wp))

          INTERSECTION(xs,ys,zs, xp,yp,zp, alpha, xi,yi,zi)
          wi := t * (zi / SCALE.FACTOR)

          right ! xi; yi; zi; wi
          right ! xp; yp; zp; wp
      TRUE
        SKIP
    xs := xp
    ys := yp
    zs := zp
    ws := wp
    FIRST.POINT.INSIDE := SECOND.POINT.INSIDE
    left ? xp
    IF
      xp > (min.int + 2)     --not a control value
        left ? yp; zp; wp
      TRUE
        SKIP
--process last edga using saved vertex
IF
  (yfirst - wfirst) <= 0
    SECOND.POINT.INSIDE := TRUE
  TRUE
    SECOND.POINT.INSIDE := FALSE
IF
  FIRST.POINT.INSIDE AND SECOND.POINT.INSIDE
    right ! xfirst; yfirst; zfirst; wfirst
  FIRST.POINT.INSIDE AND (NOT SECOND.POINT.INSIDE)
    SEQ
      alpha := ((ys - ws) * SCALE.FACTOR) /
      ((ys - ws) - (yfirst - wfirst))
```

```
        INTERSECTION(xs,ys,zs, xfirst,yfirst,zfirst,
        alpha, xi,yi,zi)

        wi := t * (zi / SCALE.FACTOR)
        right ! xi; yi; zi; wi
    (NOT FIRST.POINT.INSIDE) AND SECOND.POINT.INSIDE
      SEQ
        alpha := ((ys - ws) * SCALE.FACTOR) /
        ((ys - ws) - (yfirst - wfirst))

        INTERSECTION(xs,ys,zs, xfirst,yfirst,zfirst,
        alpha, xi,yi,zi)

        wi := t * (zi / SCALE.FACTOR)
        right ! xi; yi; zi; wi
        right ! xfirst; yfirst; zfirst; wfirst
    TRUE
      SKIP
  right ! xp:
```

```
— Bottom Clipper
—All incoming coordinates are * SCALE.FACTOR

PROC CLIP.BOTTOM (CHAN left, right,
               VALUE s, d)=

  VAR xfirst,yfirst,zfirst,  --first polygon vertex
      wfirst,
      xs,ys,zs,              --beginning of each edge
      xp,yp,zp,              --end of each edge
      xi,yi,zi,              --intersection coordinates
      ws,wp,wi,              -- w = (s / d) * z
      FIRST.POINT.INSIDE,
      SECOND.POINT.INSIDE,
      alpha,
      colour,
      t:                     -- t = s / d

  SEQ
    t := (s*SCALE.FACTOR) / d          --calculate s/d * SCALE.FACTOR
    xp := 0
    WHILE xp <> END
      SEQ
        left ? colour
        right ! colour
        left ? xs
        IF
          xs <= (min.int + 2)      --a control value; the polygon is null
            SEQ
              right ! xs
              xp := xs         --to terminate outer loop if xs = END
          TRUE
            SEQ
              left ? ys; zs; ws
              xfirst := xs
              yfirst := ys
              zfirst := zs
              wfirst := ws
              IF
                (ys + ws) >= 0
                  FIRST.POINT.INSIDE := TRUE
                TRUE
                  FIRST.POINT.INSIDE := FALSE
              left ? xp
              IF
                xp > (min.int + 2)
                  left ? yp; zp; wp
                TRUE
                  SKIP
              WHILE xp > (min.int + 2)        --While xp 'in {NEXT.POLYGON,
                SEQ                           --NEXT.FRAME, END}
      IF
        (yp + wp) >= 0
          SECOND.POINT.INSIDE := TRUE
        TRUE
          SECOND.POINT.INSIDE := FALSE
      IF
        FIRST.POINT.INSIDE AND SECOND.POINT.INSIDE
          right ! xp; yp; zp; wp
        FIRST.POINT.INSIDE AND (NOT SECOND.POINT.INSIDE)
          SEQ
            alpha := ((ys + ws) * SCALE.FACTOR)
            / ((ys + ws) - (wp+ wp))

            INTERSECTION(xs,ys,zs, xp,yp,zp, alpha, xi,yi,zi)
            wi := t * (zi / SCALE.FACTOR)
            right ! xi; yi; zi; wi
        (NOT FIRST.POINT.INSIDE) AND SECOND.POINT.INSIDE
          SEQ
            alpha := ((ys + ws) * SCALE.FACTOR)
            / ((ys + ws) - (yp + wp))

            INTERSECTION(xs,ys,zs, xp,yp,zp, alpha, xi,yi,zi)
            wi := t * (zi / SCALE.FACTOR)
            right ! xi; yi; zi; wi
            right ! xp; yp; zp; wp
        TRUE
          SKIP
      xs := xp
      ys := yp
      zs := zp
      ws := wp
      FIRST.POINT.INSIDE := SECOND.POINT.INSIDE
      left ? xp
      IF
        xp > (min.int + 2)     --not a control value
          left ? yp; zp; wp
        TRUE
          SKIP
  --process last edge using saved vertex
  IF
    (yfirst + wfirst) >= 0
      SECOND.POINT.INSIDE := TRUE
    TRUE
      SECOND.POINT.INSIDE := FALSE
  IF
    FIRST.POINT.INSIDE AND SECOND.POINT.INSIDE
      right ! xfirst; yfirst; zfirst; wfirst
    FIRST.POINT.INSIDE AND (NOT SECOND.POINT.INSIDE)
      SEQ
        alpha := ((ys + ws) * SCALE.FACTOR) /
        ((ys + ws) - (yfirst + wfirst))
        INTERSECTION(xs,ys,zs, xfirst,yfirst,zfirst,
        alpha, xi,yi,zi)

        wi := t * (zi / SCALE.FACTOR)
        right ! xi; yi; zi; wi
    (NOT FIRST.POINT.INSIDE) AND SECOND.POINT.INSIDE
      SEQ
        alpha := ((ys + ws) * SCALE.FACTOR) /
        ((ys + ws) - (yfirst + wfirst))

        INTERSECTION(xs,ys,zs, xfirst,yfirst,zfirst,
        alpha, xi,yi,zi)

        wi := t * (zi / SCALE.FACTOR)
        right ! xi; yi; zi; wi
        right ! xfirst; yfirst; zfirst; wfirst
    TRUE
      SKIP
  right ! xp:
```

```
--  Hither  Clipper
--All incoming coordinates are * SCALE.FACTOR

PROC CLIP.HITHER (CHAN left, right,
                  VALUE  s,d,
                         K )=      --z=K is the hither clipping plane

  VAR xfirst,yfirst,zfirst,  --first polygon vertex
      wfirst,
      xs,ys,zs,              --beginning of each edge
      xp,yp,zp,              --end of each edge
      xi,yi,zi,              --intersection coordinates
      ws,wp,wi,
      FIRST.POINT.INSIDE,
      SECOND.POINT.INSIDE,
      alpha,
      colour,
      t,
      k:

  SEQ
    k := K * SCALE.FACTOR       --All coordinates are * SCALE.FACTOR; so scale k too
    t := (s * SCALE.FACTOR) / d
    xp := 0
    WHILE xp <> END
      SEQ
        left ? colour
        right ! colour
        left ? xs
        IF
          xs <= (min.int + 2)      --a control value; the polygon is null
            SEQ
              right ! xs
              xp := xs          --to terminate outer loop if xs = END
          TRUE
            SEQ
              left ? ys; zs; ws
              xfirst := xs
              yfirst := ys
              zfirst := zs
              wfirst := ws
              IF
                (zs - k) >= 0
                  FIRST.POINT.INSIDE := TRUE
                TRUE
                  FIRST.POINT.INSIDE := FALSE
              left ? xp
              IF
                xp > (min.int + 2)
                  left ? yp; zp, wp
                TRUE
    SKIP
WHILE xp > (min.int + 2)           --While xp ~in {NEXT.POLYGON,
  SEQ                              --NEXT.FRAME, END}
    IF
      (zp - k) >= 0
        SECOND.POINT.INSIDE := TRUE
      TRUE
        SECOND.POINT.INSIDE := FALSE
    IF
      FIRST.POINT.INSIDE AND SECOND.POINT.INSIDE
        right ! xp; yp; zp; wp
      FIRST.POINT.INSIDE AND (NOT SECOND.POINT.INSIDE)
        SEQ
          alpha  = ((zs - k) * SCALE.FACTOR)
          / ((zs - k) - (zp - k))

          INTERSECTION(xs,ys,zs, xp,yp,zp, alpha, xi,yi,zi)
          wi := t * (zi / SCALE.FACTOR)
          right ! xi; yi; zi, wi
      (NOT FIRST.POINT.INSIDE) AND SECOND.POINT.INSIDE
        SEQ
          alpha := ((zs - k) * SCALE.FACTOR)
          / ((zs - k) - (zp - k))

          INTERSECTION(xs,ys,zs, xp,yp,zp, alpha, xi,yi,zi)
          wi := t * (zi / SCALE.FACTOR)
          right ! xi; yi; zi; wi
          right ! xp; yp; zp; wp
      TRUE
        SKIP
    xs := xp
    ys := yp
    zs := zp
    wp := ws
    FIRST.POINT.INSIDE := SECOND.POINT.INSIDE
    left ? xp
    IF
      xp > (min.int + 2)     --not a control value
        left ? yp, zp; wp
      TRUE
        SKIP
--process last edge using saved vertex
IF
  (zfirst - k) >= 0
    SECOND.POINT.INSIDE := TRUE
  TRUE
    SECOND.POINT.INSIDE := FALSE
IF
  FIRST.POINT.INSIDE AND SECOND.POINT.INSIDE
    right ! xfirst; yfirst; zfirst; wfirst
  FIRST.POINT.INSIDE AND (NOT SECOND.POINT.INSIDE)
    SEQ
```

```
      alpha := ((zs - k) * SCALE.FACTOR)
      / ((zs - k) - (zfirst -k))

      INTERSECTION(xs,ys,zs, xfirst,yfirst,zfirst,
      alpha, xi,yi,zi)

      wi := t * (zi / SCALE.FACTOR)
      right ! xi; yi; zi; wi
  (NOT FIRST.POINT.INSIDE) AND SECOND.POINT.INSIDE
    SEQ
      alpha := ((zs - k) * SCALE.FACTOR)
      / ((zs - k) - (zfirst -k))

      INTERSECTION(xs,ys,zs, xfirst,yfirst,zfirst,
      alpha, xi,yi,zi)

      wi := t * (zi / SCALE.FACTOR)
      right ! xi; yi; zi; wi
      right ! xfirst; yfirst; zfirst; wfirst
  TRUE
    SKIP
right ! xp;
```

```
--  Yon     Clipper
--All incoming coordinates are * SCALE.FACTOR

PROC CLIP.YON (CHAN left, right,
               VALUE s, d,
                     K )=      --z=K is the yon clipping plane

  VAR xfirst,yfirst,zfirst,  --first polygon vertex
      wfirst,
      xs,ys,zs,              --beginning of each edge
      xp,yp,zp,              --end of each edge
      xi,yi,zi,              --intersection coordinates
      ws,wp,wi,
      FIRST.POINT.INSIDE,
      SECOND.POINT.INSIDE,
      alpha,
      colour,
      t,
      k:

  SEQ
    k := K * SCALE.FACTOR       --All coordinates are * SCALE.FACTOR; so scale k too
    t := (s * SCALE.FACTOR) / d
    xp := 0
    WHILE xp <> END
      SEQ
        left ? colour
        right ! colour
        left ? xs
        IF
          xs <= (min.int + 2)      --a control value; the polygon is null
            SEQ
              right ! xs
              xp := xs          --to terminate outer loop if xs = END
          TRUE
            SEQ
              left ? ys; zs; ws
              xfirst := xs
              yfirst := ys
              zfirst := zs
              wfirst := ws
              IF
                (zs - k) <= 0
                  FIRST.POINT.INSIDE := TRUE
                TRUE
                  FIRST.POINT.INSIDE := FALSE
              left ? xp
              IF
                xp > (min.int + 2)
                  left ? yp; zp; wp
                TRUE
    SKIP
WHILE xp > (min.int + 2)          --While xp ~in {NEXT.POLYGON,
  SEQ                             --NEXT.FRAME, END}
    IF
      (zp - k) <= 0
        SECOND.POINT.INSIDE := TRUE
      TRUE
        SECOND.POINT.INSIDE := FALSE
    IF
      FIRST.POINT.INSIDE AND SECOND.POINT.INSIDE
        right ! xp; yp; zp; wp
      FIRST.POINT.INSIDE AND (NOT SECOND.POINT.INSIDE)
        SEQ
          alpha := ((zs - k) * SCALE.FACTOR)
          / ((zs - k) - (zp - k))

          INTERSECTION(xs,ys,zs, xp,yp,zp, alpha, xi,yi,zi)
          wi := t * (zi / SCALE.FACTOR)
          right ! xi; yi; zi; wi
      (NOT FIRST.POINT.INSIDE) AND SECOND.POINT.INSIDE
        SEQ
          alpha := ((zs - k) * SCALE.FACTOR)
          / ((zs - k) - (zp -k))

          INTERSECTION(xs,ys,zs, xp,yp,zp, alpha, xi,yi,zi)
          wi := t * (zi / SCALE.FACTOR)
          right ! xi; yi; zi; wi
          right ! xp; yp; zp; wp
      TRUE
        SKIP
    xs := xp
    ys := yp
    zs := zp
    ws := wp
    FIRST.POINT.INSIDE := SECOND.POINT.INSIDE
    left ? xp
    IF
      xp > (min.int + 2)     --not a control value
        left ? yp; zp; wp
      TRUE
        SKIP
--process last edge using saved vertex
IF
  (zfirst - k) <= 0
    SECOND.POINT.INSIDE := TRUE
  TRUE
    SECOND.POINT.INSIDE := FALSE
IF
  FIRST.POINT.INSIDE AND SECOND.POINT.INSIDE
    right ! xfirst; yfirst; zfirst; wfirst
  FIRST.POINT.INSIDE AND (NOT SECOND.POINT.INSIDE)
    SEQ
```

```
        alpha := ((zs - k) * SCALE.FACTOR)
        / ((zs - k) - (zfirst - k))

        INTERSECTION(xs,ys,zs, xfirst,yfirst,zfirst,
        alpha, xi,yi,zi)

        wi := t * (zi / SCALE.FACTOR)
        right ! xi; yi; zi; wi
  (NOT FIRST.POINT.INSIDE) AND SECOND.POINT.INSIDE
     SEQ
        alpha := ((zs - k) * SCALE.FACTOR)
        / ((zs - k) - (zfirst -k))

        INTERSECTION(xs,ys,zs, xfirst,yfirst,zfirst,
        alpha, xi,yi,zi)

        wi := t * (zi / SCALE.FACTOR)
        right ! xi; yi; zi; wi
        right ! xfirst; yfirst; zfirst; wfirst
  TRUE
     SKIP
right ! xp:
```

```
-- Splitter    (Splits about plane X = 0)
--All incoming coordinates are *SCALE.FACTOR

PROC X.SPLITTER( CHAN in,out1,out2,
                 VALUE s ,d   )=

  VAR xfirst,yfirst,zfirst,   --first polygon vertex
      wfirst,
      xs,ys,zs,               --beginning of each edge
      xp,yp,zp,               --end of each edge
      xi,yi,zi,               --intersection coordinates
      ws,wp,wi,
      FIRST.POINT.LEFT,
      FIRST.POINT.ON.PLANE,
      SECOND.POINT.LEFT,
      SECOND.POINT.ON.PLANE,
      alpha,
      colour,
      t:

  SEQ
    t := (s * SCALE.FACTOR) / d
    xp := 0
    WHILE xp <> END
      SEQ
        in ? colour
        out1 ! colour
        out2 ! colour
        in ? xs
        IF
          xs <= (min.int + 2)      --a control value: the polygon is null
            SEQ
              out1 ! xs
              out2 ! xs
              xp := xs          --to terminate outer loop if xs = END
          TRUE
            SEQ
              in ? ys; zs; ws
              xfirst := xs
              yfirst := ys
              zfirst := zs
              wfirst := ws
              IF
                xs < 0
                  SEQ
                    FIRST.POINT.LEFT  = TRUE
                    FIRST.POINT.ON.PLANE := FALSE
                xs > 0
```

NEX;P_r CEX;I430

NEX;B-I
CEX;II31

```
            SEQ
              FIRST.POINT.LEFT := FALSE
              FIRST.POINT.ON.PLANE := FALSE
          TRUE
            SEQ
              FIRST.POINT.ON.PLANE := TRUE
              FIRST.POINT.LEFT := FALSE
        in ? xp
        IF
          xp > (min.int + 2)   --not a control value
            in ? yp; zp; wp
          TRUE
            SKIP
        WHILE xp > (min.int + 2)        --While xp ~in {NEXT.POLYGON,
          SEQ                           --NEXT.FRAME, END}
            IF
              xp < 0
                SEQ
                  SECOND.POINT.LEFT := TRUE
                  SECOND.POINT.ON.PLANE := FALSE
              xp > 0
                SEQ
                  SECOND.POINT.LEFT := FALSE
                  SECOND.POINT.ON.PLANE := FALSE
              TRUE
                SEQ
                  SECOND.POINT.ON.PLANE := TRUE
                  SECOND.POINT.LEFT := FALSE
            IF
              SECOND.POINT.ON.PLANE
                SEQ
                  out1 ! xp; yp; zp; wp
                  out2 ! xp; yp; zp; wp
              SECOND.POINT.LEFT AND (FIRST.POINT.ON.PLANE OR
              FIRST.POINT.LEFT)
                out1 ! xp; yp; zp; wp
              (NOT FIRST.POINT.LEFT) AND (NOT SECOND.POINT.LEFT)
                out2 ! xp; yp; zp; wp
              FIRST.POINT.LEFT AND (NOT SECOND.POINT.LEFT)
                SEQ
                  alpha := ((-xs) * SCALE.FACTOR) / (xp - xs)
                  INTERSECTION(xs,ys,zs, xp,yp,zp, alpha, xi,yi,zi)
                  wi := t * (zi / SCALE.FACTOR)
                  out1 ! xi; yi; zi; wi
                  out2 ! xi; yi; zi; wi
                  out2 ! xp; yp; zp; wp
              (NOT FIRST.POINT.LEFT) AND SECOND.POINT.LEFT
                SEQ
                  alpha := (xs * SCALE.FACTOR) / (xs - xp)
                  INTERSECTION(xs,ys,zs, xp,yp,zp, alpha, xi,yi,zi)
                  wi  = t * (zi / SCALE.FACTOR)
                  out1 ! xi, yi; zi; wi
              out2 ! xi; yi; zi; wi
              out1 ! xp; yp; zp; wp
      xs := xp
      ys := yp
      zs := zp
      ws := wp
      FIRST.POINT.LEFT := SECOND.POINT.LEFT
      FIRST.POINT.ON.PLANE := SECOND.POINT.ON.PLANE
      in ? xp
      IF
        xp > (min.int + 2)    --not a control value
          in ? yp; zp; wp
        TRUE
          SKIP
--process last edge using saved vertex
IF
  xfirst < 0
    SEQ
      SECOND.POINT.LEFT := TRUE
      SECOND.POINT.ON.PLANE := FALSE
  xfirst > 0
    SEQ
      SECOND.POINT.LEFT := FALSE
      SECOND.POINT.ON.PLANE := FALSE
  TRUE
    SEQ
      SECOND.POINT.ON.PLANE := TRUE
      SECOND.POINT.LEFT := FALSE

IF
  SECOND.POINT.ON.PLANE
    SEQ
      out1 ! xfirst; yfirst; zfirst; wfirst
      out2 ! xfirst; yfirst; zfirst; wfirst

  SECOND.POINT.LEFT AND (FIRST.POINT.ON.PLANE OR
  FIRST.POINT.LEFT)
    out1 ! xfirst; yfirst; zfirst; wfirst

  (NOT FIRST.POINT.LEFT) AND (NOT SECOND.POINT.LEFT)
    out2 ! xfirst; yfirst; zfirst; wfirst

  FIRST.POINT.LEFT AND (NOT SECOND.POINT.LEFT)
    SEQ
      alpha := (.-xs) * SCALE.FACTOR) / (xfirst - xs)
      INTERSECTION(xs,ys,zs, xfirst,yfirst,zfirst,
      alpha, xi,yi,zi)

      wi := t * (zi / SCALE.FACTOR)
      out1 ! xi, yi; zi; wi
      out2 ! xi; yi; zi; wi
```

```
        out2 ! xfirst; yfirst; zfirst; wfirst

    (NOT FIRST.POINT.LEFT) AND SECOND.POINT.LEFT
      SEQ
        alpha := (xs * SCALE.FACTOR) / (xs - xfirst)
        INTERSECTION(xs,ys,zs, xfirst,yfirst,zfirst,
        alpha, xi,yi,zi)

        wi := t * (zi / SCALE.FACTOR)
        out1 ! xi; yi; zi; wi
        out2 ! xi; yi; zi; wi
        out1 ! xfirst; yfirst; zfirst; wfirst
  out1 ! xp
  out2 ! xp:
```

```
--  Perspective Transformation
--Transform from Eye to Screen Coordinates,
--preserving depth information
--and Descale Coordinates / SCALE.FACTOR
--(w includes SCALE.FACTOR)

PROC PERSPECTIVE.TRANSFORMATION( CHAN in,out)=
  VAR x, y, z, w, colour:
  SEQ
    x := 0
    WHILE x <> END
      SEQ
        in ? colour
        out ! colour
        in ? x
        WHILE x > (min.int + 2)
          SEQ
            in ? y; z; w
            out ! ((x * HALF.SW) / w) + HALF.SW     --Xs
            out ! ((y * HALF.SH) / w) + HALF.SH     --Ys
            out ! z / SCALE.FACTOR                  --Zs
            in ? x
        out ! x
```

NEX:P_T CEX:138

NEX:E
CEX:439

```
--  Data Structures (ET,AET) and Operations
  -- used by the Scan Converter of the HSE algorithm


--  Reset Data Structure for next polygon
PROC CLEAR(VAR ET[],
               AET,
               firstfree)=

  SEQ
    --Clear Edge Table
    SEQ i=[0 FOR SCREEN.HEIGHT]
      ET[i] := NIL
    --Clear Active Edge Table
    AET := NIL
    --Reset Edge Pointer
    firstfree := 0
```

cost:13686 tc (incl. call
for SCREEN.HEIGHT=500)

```
--Copy edge data into EDGES array
--Auxilliary of  INSERT.ET

PROC PUT (VAR EDGES[],
              firstfree,
          VALUE ymax,xmin,m )=

  SEQ
    EDGES[firstfree] := ymax
    firstfree := firstfree + 1
    EDGES[firstfree] := xmin
    firstfree := firstfree + 1
    EDGES[firstfree] := m
    firstfree := firstfree + 1
    EDGES[firstfree] := NIL
    firstfree := firstfree + 1
    EDGES[firstfree] := NIL
    firstfree := firstfree + 1:
```

```
-- --Insert an edge into the appropriate bucket of the edge table
     --maintaining order on (x,slope) within bucket.
     --Auxilliary of INSERT.ET.EDGE

PROC INSERT.ET (VAR ET[],
                    EDGES[],
                    firstfree,
              VALUE scanline,         --indicates appropriate bucket
                    ymax,xmin,m )=    --edge to be inserted

  VAR ET.PTR, PREV.ET.PTR, lastfree:
  SEQ
    PREV.ET.PTR :=NIL
    ET.PTR := ET[scanline]     --beginning of bucket for this scanline
    IF
      ET.PTR = NIL
        --Special case bucket is empty
        SEQ
          ET[scanline] := firstfree
          PUT(EDGES,firstfree,ymax,xmin,m)
      TRUE
        SEQ
          --Find appropriate place for insertion
          WHILE (EDGES[ET.PTR + 1] < xmin) AND (EDGES[ET.PTR + 3] <> NIL)
            SEQ
              PREV.ET.PTR := ET.PTR
              ET.PTR := EDGES[ET.PTR + 3]
          lastfree := firstfree
          PUT(EDGES,firstfree,ymax,xmin,m)
          IF
            (EDGES[ET.PTR + 1] < xmin) OR
            ((EDGES[ET.PTR + 1] = xmin) AND (EDGES[ET.PTR + 2] < m))
              --insert after the current bucket edge
              SEQ
                EDGES[lastfree + 3] := EDGES[ET.PTR + 3]
                EDGES[ET.PTR + 3] := lastfree
            TRUE
              --Insert before current bucket edge
              SEQ
                EDGES[lastfree + 3] := ET.PTR
                IF
                  PREV.ET.PTR = NIL
                    --becomes first bucket edge; special case
                    ET[scanline] := lastfree
                  TRUE
                    EDGES[PREV.ET.PTR + 3] := lastfree:
```

cost: 307 tc (incl. call)

$NEX_1E_{etc}/2$

CEX: 49

```
— --Determine slope of edge, shorten if required and insert into ET

PROC INSERT.ET.EDGE( VALUE x1,y1,        --x coordinates are *SCALE.FACTOR
                           x2,y2,
                           x3,y3,
                       VAR ET[],
                           EDGES[],
                           firstfree)=

  VAR xmin,ymin,xmax,ymax,m:
  IF
    y1 = y2
      --Horizontal edge, needs no processing
      SKIP
    TRUE
      SEQ
        IF
          y1 > y2
            SEQ
              ymin := y2
              xmin := x2
              ymax := y1
              xmax := x1
          y1 < y2
            SEQ
              ymin := y1
              xmin := x1
              ymax := y2
              xmax := x2
        --Calculate (1 / slope) * SCALE.FACTOR in m
        m := (xmax - xmin) / (ymax - ymin)
        IF
          (y1 <  y2) AND (y2 < y3)
            --Shorten edge: (x2,y2) is not a local maximum/minimum
            ymax := ymax - 1
          (y1 >  y2) AND (y2 > y3)
            --Shorten edge
            SEQ
              ymin := ymin + 1
              xmin := xmin + m
          TRUE
            --Don't shorten edge (x2,y2) is a local minimum/maximum
            SKIP
        INSERT.ET(ET,EDGES,firstfree,ymin,ymax,xmin,m):
```

cost: I92 tc (incl. call)

```
—  —Insert an edge into the AET maintaining order on x
     --Auxilliary of MOVE.ET.BUCKET.TO.AET
PROC INSERT.AET (VAR AET,
                     EDGES[],
               VALUE EDGE )=

  VAR x, AET.PTR, PREV.AET.PTR:
  SEQ
    IF
      AET = NIL
        —First edge of AET; special case
        SEQ
          AET := EDGE
          EDGES[EDGE + 4] := NIL
      TRUE
        SEQ
          x := EDGES[EDGE + 1]
          AET.PTR := AET
          PREV.AET.PTR := NIL
          WHILE ((EDGES[AET.PTR + 1] < x)
          AND (EDGES[AET.PTR + 4] <> NIL))

            SEQ
              PREV.AET.PTR := AET.PTR
              AET.PTR := EDGES[AET.PTR + 4]
          IF
            EDGES[AET.PTR + 1] < x
              --Insert after current AET edge
              SEQ
                EDGES[EDGE + 4] := EDGES[AET.PTR + 4]
                EDGES[AET.PTR + 4] := EDGE
            TRUE
              --Insert before current AET edge
              SEQ
                EDGES[EDGE + 4] := AET.PTR
                IF
                  PREV.AET.PTR = NIL
                    --Becomes firt edge of AET; special case
                    AET := EDGE
                  TRUE
                    EDGES[PREV.AET.PTR + 4] := EDGE:
```

NEX, E_{aet}-1

```
--  Move a bucket of edges from the ET to the AET,
     --Keeping the AET sorted on x.

PROC MOVE.ET.BUCKET.TO.AET (VAR EDGES[],
                                AET,
                                BEGINNING.OF.BUCKET )=

  VAR EDGE:
  SEQ
    EDGE := BEGINNING.OF.BUCKET
    WHILE EDGE <> NIL
      SEQ
        INSERT.AET(AET.EDGES.EDGE)
        EDGE := EDGES[EDGE + 3]
    BEGINNING.OF.BUCKET := NIL:     --remove bucket from ET
```

cost:36 tc (incl. call)

NEX:E_{etf}

CEX:195

```
--  --Remove an edge from the AET
     --Auxilliary of UPDATE.AET

PROC REMOVE.AET (VAR AET,
                     EDGES[],
                     PREV.EDGE,
                 VALUE EDGE )=

  IF
    PREV.EDGE = NIL
      AET := EDGES[EDGE + 4]
    TRUE
      EDGES[PREV.EDGE + 4] := EDGES[EDGE + 4]:
```

```
-- Update the AET
   --by removing those edges for which ymax = scanline and
   --calculating the x intercept of the rest of the edges
   --for the next scanline.

cost;34 tc (incl. call)
PROC UPDATE.AET (VAR AET,
                     EDGES[],
               VALUE scanline )=

  VAR EDGE, PREV.EDGE:
  SEQ
    PREV.EDGE := NIL
    EDGE := AET
    WHILE EDGE <> NIL
      SEQ
        IF
          EDGES[EDGE + 0] = scanline
            --remove this edge from the AET
            REMOVE.AET(AET,EDGES,PREV.EDGE,EDGE)
NEX:E
CEX:10tc
          TRUE
            SEQ
              --update the x intercept of this edge
              EDGES[EDGE + 1] := EDGES[EDGE + 1] + EDGES[EDGE + 2]
              PREV.EDGE := EDGE
        EDGE := EDGES[EDGE + 4]:
```

```
—  —Swap two edges. Auxilliary of BUBBLESORT.
PROC SWAP.EDGES (VAR EDGES[],
                 VALUE EDGE1,
                       EDGE2 )=

  VAR TEMP[3]:
  SEQ
    SEQ i=[0 FOR 3]
      TEMP[i]  := EDGES[EDGE1 + i]
    SEQ i=[0 FOR 3]
      EDGES[EDGE1 + i] := EDGES[EDGE2 + i]
    SEQ i=[0 FOR 3]
      EDGES[EDGE2 + i] := TEMP[i]:




--  Sort AET on x, using bubblesort

PROC BUBBLESORT (VAR AET,
                     EDGES[] )=

  VAR UNSORTED,EDGE1,EDGE2:
  SEQ
    IF
      ((AET = NIL) OR (EDGES[AET + 4] = NIL))
        --Trivially sorted
        SKIP
      TRUE
        SEQ
          UNSORTED := TRUE
          WHILE UNSORTED
            SEQ
              UNSORTED := FALSE
              EDGE1 := AET
              EDGE2 := EDGES[AET + 4]
              WHILE EDGE2 <> NIL
                SEQ
                  IF
                    EDGES[EDGE1 + 1] > EDGES[EDGE2 + 1]
                      SEQ
                        SWAP.EDGES(EDGES.EDGE1,EDGE2)
                        UNSORTED := TRUE
                    TRUE
                      SKIP
                  EDGE1 := EDGE2
                  EDGE2 := EDGES[EDGE2 + 4]:
```

cost:505 tc (incl. call)

cost: 54 tc (incl. call)

```
-- --Update indices of first and last non-empty ET buckets
PROC UPDATE.MIN.MAX.ET.Y (VAR MIN.ET.Y, MAX.ET.Y,
                          VALUE Y)=

  IF
    Y < MIN.ET.Y
      MIN.ET.Y := Y
    Y > MAX.ET.Y
      MAX.ET.Y := Y
    TRUE
      SKIP :
```

```
—  —The Scan Converter Process
     —Assumes v1,v2...vn representation of polygons(vi = ith vertex)

PROC SCAN.CONVERTER (CHAN IN,
                          TO.BUFFER,
                     VAR  ET[],
                          AET,
                          EDGES[],
                          firstfree )=

  VAR colour,                --polygon colour
      x1,y1,z1,              --polygon vertices
      x2,y2,z2,
      x3,y3,z3,
      keepx1,keepy1,keepz1,
      keepx2,keepy2,keepz2,
      a,b,c,d,               --plane coefficients
      MIN.ET.Y, MAX.ET.Y,    --first/last non-empty bucket in ET
      Y,                     --current scanline
      EDGE,                  --used to traverse AET
      X.START, X.FINISH,     --used in scan conversion
      Z, Z.INC :             --depth and depth increment

  SEQ
    CLEAR(ET,AET,firstfree)
    x3 := 0
    WHILE (x1 <> END) AND (x2 <> END) AND (x3 <> END)
      SEQ
        MIN.ET.Y := 0
        MAX.ET.Y := SCREEN.HEIGHT - 1
        IN ? colour
        IN ? x1
        IF
          x1 < (min.int + 3)
            TO.BUFFER ! x1              --Empty Polygon; ignore
          TRUE
            SEQ
              IN ? y1; z1
              IN ? x2
              IF
                x2 < (min.int + 3)
                  TO.BUFFER ! x2          --One vertex polygon; ignore
                TRUE
                  SEQ
                    IN ? y2; z2
                    IN ? x3
                    IF
                      x3 < (min.int + 3)
                        TO.BUFFER ! x3      --Two vertex polygon; ignore
                      TRUE
                        SEQ
                          IN ? y3; z3
```

NEX:P_r GEX:I034

NEX:E-2 GEX:I39

```
keepx1 := x1
keepy1 := y1
keepz1 := z1
keepx2 := x2
keepy2 := y2
keepz2 := z2
--determine plane equation from first 3 vertices;
--assume they are not collinear
a := (((y1-y2)*(z1+z2)) + ((y2-y3)*(z2+z3))
+ ((y3-y1)*(z3+z1)))

b := (((z1-z2)*(x1+x2)) + ((z2-z3)*(x2+x3))
+ ((z3-z1)*(x3+x1)))

c := (((x1-x2)*(y1+y2)) + ((x2-x3)*(y2+y3))
+ ((x3-x1)*(y3+y1)))

d := (( ((-a)*x1) - (b*y1)) - (c*z1))

--construct Edge Table
WHILE x3 > (min.int + 2)
          --Terminates when a control value is met
          --at the end of a polygon
  SEQ

    INSERT.ET.EDGE(x1*SCALE.FACTOR,y1,
    x2*SCALE.FACTOR,y2,x3*SCALE.FACTOR,y3,
    ET.EDGES,firstfree)

    UPDATE.MIN.MAX.ET.Y (MIN.ET.Y, MAX.ET.Y, y2)

    x1 := x2
    y1 := y2
    z1 := z2
    x2 := x3
    y2 := y3
    z2 := z3
    IN ? x3
    IF
      x3 >= (min.int + 3)  --If not a control value
        SEQ
          IN ? y3
          IN ? z3
      TRUE
        SKIP
INSERT.ET.EDGE(x1*SCALE.FACTOR,y1,
x2*SCALE.FACTOR,y2,keapx1*SCALE.FACTOR,keepy1,
ET.EDGES,firstfree)

UPDATE.MIN.MAX.ET.Y (MIN.ET.Y, MAX.ET.Y, y2)
```

```
          INSERT.ET.EDGE(x2*SCALE.FACTOR,y2,
          keepx1*SCALE.FACTOR,keepy1,
          keepx2*SCALE.FACTOR,keepy2,
          ET.EDGES,firstfree)

          UPDATE.MIN.MAX.ET.Y (MIN.ET.Y, MAX.ET.Y, keepy1)

          Y := MIN.ET.Y
          WHILE  (Y <= MAX.ET.Y) OR (AET <> NIL)
            SEQ
              MOVE.ET.BUCKET.TO.AET(EDGES, AET, ET[Y])
              --  --use AET to process this scanline
              EDGE := AET

              --assume c <> 0, if c = 0 the polygon
              --is parallel to the Z-axis and appears
              --as a line; such polygons are not processed

              WHILE ((EDGE <> NIL) AND (c <> 0))
                SEQ
                  X.START := EDGES[EDGE + 1] / SCALE.FACTOR
                  EDGE := EDGES[EDGE + 4]
                  X.FINISH := EDGES[EDGE + 1] / SCALE.FACTOR
                  EDGE := EDGES[EDGE + 4]
                  --calculate depth*SCALE.FACTOR at (X.START,Y)
                  Z := (( ((-d) - (a*X.START)) - (b*Y) )
                  * SCALE.FACTOR) / c

                  --calculate (depth increment)*SCALE.FACTOR
                  Z.INC := ((-a) * SCALE.FACTOR) / c
                  WHILE X.START <= X.FINISH
                    SEQ
                      TO.BUFFER ! colour; X.START; Y; Z
                      X.START := X.START + 1
                      Z := Z + Z.INC

              UPDATE.AET(AET,EDGES,Y)
              BUBBLESORT(AET,EDGES)
              Y := Y + 1
          IF
            x3 = NEXT.POLYGON
              SEQ
                TO.BUFFER ! NEXT.POLYGON
                CLEAR(ET,AET,firstfree)
            x3 = NEXT.FRAME
              SEQ
                TO.BUFFER ! NEXT.FRAME  --clear Z and F Buffers
                CLEAR(ET,AET,firstfree)
            TRUE     --x3 = END
              --terminate Buffer Process
              TO.BUFFER ! END;
```

NEX:H CEX:44

NEX:E_{aet} CEX:414

NEX:W CEX:152

```
—   —The Buffer Process & auxilliaries



--   --clear Z and F buffers prior to processing next frame
      —auxilliary of the Buffer Process

PROC CLEAR.BUFFERS (VAR Z.BUFFER[],
                        F.BUFFER[] )=

  VAR i, j:
  SEQ i=[0 FOR SCREEN.HEIGHT]
    SEQ j=[0 FOR SCREEN.WIDTH]
      SEQ
        Z.BUFFER[(i*SCREEN.WIDTH) + j] := MAX.DEPTH
        F.BUFFER[(i*SCREEN.WIDTH) + j] := BACK.GND.COLOUR:




--   --display the scene stored in the F-buffer
      —auxilliary of the Buffer Process

PROC DISPLAY (VAR F.BUFFER[])=     --VAR in order to save copying time
  VAR t:
  SEQ i=[0 FOR SCREEN.HEIGHT]
    SEQ
      t := ((SCREEN.HEIGHT - 1) - i) * SCREEN.WIDTH
      SEQ j=[0 FOR SCREEN.WIDTH]
        screen ! F.BUFFER[t + j]
      screen ! '*N'; '*C'; EndBuffer:
```

```
PROC BUFFER (CHAN IN1, IN2,
             VAR Z.BUFFER[],
                 F.BUFFER[] )=

  VAR colour1,  colour2, x, y, z:
  SEQ
    CLEAR.BUFFERS(Z.BUFFER, F.BUFFER)
    IN1 ? colour1
    IN2 ? colour2
    WHILE (colour1 <> END) OR (colour2 <> END)

      ALT
        (colour1 > (min.int + 2)) & IN1 ? x; y; z
          SEQ
            IF
              Z.BUFFER[(y*SCREEN.WIDTH) + x] > z
                PAR
                  Z.BUFFER[(y*SCREEN.WIDTH) + x] := z
                  F.BUFFER[(y*SCREEN.WIDTH) + x] := colour1
              TRUE
                SKIP
            IN1  ? colour1

        (colour2 > (min.int + 2)) & IN2 ? x; y; z
          SEQ
            IF
              Z.BUFFER[(y*SCREEN.WIDTH) + x] > z
                PAR
                  Z.BUFFER[(y*SCREEN.WIDTH) + x] := z
                  F.BUFFER[(y*SCREEN.WIDTH) + x] := colour2
              TRUE
                SKIP
            IN2  ? colour2

        (colour1 = NEXT.POLYGON) AND (colour2 = NEXT.POLYGON) & SKIP
          SEQ
            IN1  ? colour1
            IN2  ? colour2

        (colour1  = NEXT.FRAME) AND (colour2 = NEXT.FRAME) & SKIP
          SEQ
            DISPLAY(F.BUFFER)
            CLEAR.BUFFERS(Z.BUFFER, F.BUFFER)
            IN1  ? colour1
            IN2  ? colour2

        (colour1  = END) AND (colour2 = END) & SKIP
          SKIP       --Terminate
    DISPLAY(F.BUFFER):
```

```
VAR ET1[SCREEN.HEIGHT],
    ET2[SCREEN.HEIGHT],
    EDGES1[MAX.EXPECTED],
    EDGES2[MAX.EXPECTED],
    firstfree1,
    firstfree2,
    AET1,
    AET2,
    Z.BUFFER[SCREEN.HEIGHT * SCREEN.WIDTH],
    F.BUFFER[SCREEN.HEIGHT * SCREEN.WIDTH]:

CHAN c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11, c12, c13, c14:


     --THE   PIPELINE


PAR
  INPUTTER(c1)

  VIEWING.TRANSFORMATION(c1,c2,s,d,D)

  CLIP.LEFT(c2,c3,s,d)

  CLIP.RIGHT(c3,c4,s,d)

  CLIP.TOP(c4,c5,s,d)

  CLIP.BOTTOM(c5,c6,s,d)

  CLIP.HITHER(c6,c7,s,d,K1)

  CLIP.YON(c7,c8,s,d,K2)

  X.SPLITTER(c8,c9,c10,s,d)

  PERSPECTIVE.TRANSFORMATION(c9,c11)

  PERSPECTIVE.TRANSFORMATION(c10,c12)

  SCAN.CONVERTER(c11,c13,ET1,AET1,EDGES1,firstfree1)

  SCAN.CONVERTER(c12,c14,ET2,AET2,EDGES2,firstfree2)

  BUFFER(c13,c14,Z.BUFFER,F.BUFFER)
```

SAMPLE OUTPUT
PENETRATING POLYGONS

```
  I
 III
IIIII     0
 IIIII   00
  IIII I0000
   III II0000
    II II10000
     I II10000
    00II000000
   000OO0000000
   000OO00000000
          IIII
           II
            I
```