Language extensions to study data structures for raster graphics

LANGUAGE EXTENSIONS TO STUDY DATA STRUCTURES FOR RASTER GRAPHICS

BY

J. KALISVAART, L.R.A. KESSENER, W.J.M. LEMMENS, M.L.P. VAN LIEROP, F.J. PETERS, H.M.M. VAN DE WETERING

87/09

april 1987

2.

COMPUTING SCIENCE NOTES

This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science of Eindhoven University of Technology.
Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review.
Copies of these notes are available from the author or the editor.

Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands

editor: F.A.J. van Neerven

LANGUAGE EXTENSIONS TO STUDY DATA STRUCTURES FOR RASTER GRAPHICS

L.R.A. Kessener, M.L.P. van Lierop, J. Kalisvaart

Eindhoven University of Technology

F.J. Peters

Philips, Corporate ISA / CAD Centre

0. Introduction

Raster graphics images are made up from discrete elements, called pixels, arranged in a two-dimensional rectangular region. In programs images of that kind might be represented internally by one- or two-dimensional arrays consisting of as many components as there are pixels. However, better representations might be thought of, saving memory space and/or computation time. Examples are chain or boundary codes [2] and quadtrees [9]. Of the latter many variants exist, for an overview see [13].

Which representation is most suited depends upon the application one has in mind and on the computing machinery one has at his disposal. Moreover, it depends upon the programming environment and the programming language used, more specifically the way in which images are specified and the kind of operations that are allowed. A programming environment is needed to investigate and compare the different representations and data. Therefore we had to choose or to design a programming language to serve interactive graphics applications that show pictures of three-dimensional objects on a two-dimensional raster graphics screen.

Such a graphics programming language should be minimal (no unnecessary constructs, no frills) though complete enough to be a tool for investigation. Because of these demands PHIGS [6] and 3D-GKS [5] are not completely satisfactory. (For instance, of the eight output primitives in PHIGS only cell array is typical for raster graphics. Moreover the defining PHIGS document counts over a thousand pages and it seems rather difficult to select a PHIGS subset that suits our purposes. 3D-GKS does not allow the definition of hierarchically structured images.) Hence we decided to design a graphics language ourselves. It would serve no useful purpose to design a completely new language, for a graphics language needs all the capabilities of a general purpose language. Therefore the best thing to do is to use an existing general purpose language as the base. Because the computing system we have at our disposal is provided with a UNIX operating system, for which the C-compiler yields the most efficient code, we have chosen to extend C. That extension of C with graphical capabilities is termed CG. For a description of C see [8], [3] or [14].

A preprocessor has been developed to translate CG programs into plain C. The implementation of the preprocessor is described in [10], a user guide will be found in [15]. To construct the preprocessor Yacc has been used; the result is a portable preprocessor written in C. (Yacc is described in [7].)

In the next section an overview is given of the geometric types that form part of CG. These types together with the operations defined on them, are described in more detail in sections 2 (on output) and 3 (on interaction). Section 4 contains a complete example of an interactive program written in CG.

1. Types of CG

Besides the basic types **char, int, float** and **double** of C, CG is provided with the types **point, polygon, link** and **linklist.**

Values of type **point** represent geometrical points in three-dimensional space. From points polygons may be formed. Polygons are simple, colored, two- dimensional surfaces in three-dimensional space.

Polygons are used to form scenes. Scenes may be displayed on a raster graphic screen.

In [11] it is stated that the most common way of structuring pictures in graphics programming systems is in a hierarchy, so that a picture is made up of subparts that are pictures themselves. Moreover graphical transformations can be included within picture hierarchies to modify pictures according to their structure. In PHIGS for instance hierarchy plays an important role. In CG the type **link** is introduced to describe hierarchies between scenes. A scene consists of a bag of polygons and references to other (named) scenes.

Formally a scene is a 3-tuple consisting of a name (of type string), a bag of polygons and a bag of links. A value of type **link** is -again- a 3-tuple consisting of the name of a scene, a real 4 x 4 matrix and a list of attributes. Thus a scene may contain references to other scenes. Each reference is joined with a 4 x 4 transformation matrix and a list of attributes. If scene S contains a reference to T with matrix M and attribute list A, that is to say, if the 3-tuple (T, M, A) occurs in the link bag of S, then the polygons of T transformed by premultiplying their coordinates with M form part of S and the attributes in A are associated with all those polygons. Possible attributes are blinking, non-blinking, detectable, non-detectable, highlighted and so on.

S may contain more than one reference to T, each reference with its own transformation matrix and its own attribute list. T may be referenced by other scenes than S as well and moreover T may in its turn reference still other scenes. It is not allowed that T contains a reference to S. By conceiving each reference as a directed edge from the scene containing the reference to the scene referenced, a directed graph is obtained. In this graph no cycles are allowed. Names of scenes may be passed as parameters to functions operating on scenes. How hierarchically structured scenes are defined and manipulated is explained in chapter 2.

Because we aimed at interactive graphics applications, CG had to be provided with the capability to handle graphical input. To that end we introduced the type **linklist.** A discussion of its set of values and operators is better postponed till chapter 3, where interaction is discussed.

2. Scenes

The types **point, polygon** are called geometric types.

Geometric types are allowed in other type definitions; for instance arrays and functions of those types may be defined. Values and variables of geometric types are allowed in assignment statements. Geometric expressions are defined according to the following grammar rules: (For the grammar rules we use the syntax notation of [8] : alternatives are listed on separate lines.)

point-expression:
    point-identifier
    point-expression @+ *point-expression*
    point-expression @- *point-expression*
    float @* *point-expression*
    (@ float-expression, float-expression, float-expression @)

In the above syntax rules @+, @- and @* are special CG operators for coordinate wise addition, subtraction and multiplication. (The character @ has been introduced in CG expressions for reasons of convenience for the implementation of the preprocessor.)

Indexed variables (components of arrays or structs of points) are allowed as point-identifier.

Example 1:

After the declarations

point p1, p2, p3, p4;
float e;

the following assignment statements are allowed

p1 = (@ 0.0, 0.0, 1.0 @);
p2 = p1;
p3 = (@ 0.0, e, 2*e @);
p4 = p1 @+ 3.5 @* (@ 0.0, e, 2*e @);

(End of example.)

Polygon-expressions are defined by the rule

polygon-expression:
polygon-identifier
(@ point-expression-list @: color-index @)
extend (polygon-identifier, point-expression-list)
chcol (polygon-identifier, color-index)

where point-expression-list is defined by

point-expression-list:
point-expression
point-expression-list, point-expression

and color-index stands for an expression with an **int** value.

Let p1, ..., pn be point-expressions and c an int value. For the polygon-expression

(@ p1, ..., pn @: c @)

to be valid, it is necessary that the points p1, ..., pn all lie in the same two-dimensional plane; moreover it is required that the points p1, ..., pn list the vertices of a simply connected polygon, whose boundary consists of the edges (p1, p2), (p2, p3), ..., (pn, p1). The polygon-expression then represents that polygon; the color of that polygon is determined by the value c. That value acts as an index in a color table. Size and contents of that color table are implementation dependent.

The coordinate system is assumed to be right handed.

Assuming P to be a polygon identifier with point-expression-list Pp1, ..., Ppm and color-index c then the expression

extend(P, p1)

stands for the **polygon** with point-expression-list Pp1,...,Ppm, p1 and color-index c; the expression

chcol(P, d)

stands for the polygon with color-index d and the vertices of P.

Example 2:

Assume 3 and 5 are the indices in the color table associated with red and green respectively. Consider the following declarations and assignments:

```
int red, green;
point p1, p2, p3, p4, p5, p6;
polygon pg1, pg2;

red = 3; green = 5;
p1 = (@ 0.0, 0.0, 0.0 @);
p2 = (@ 1.0, 0.0, 0.0 @);
p3 = (@ 0.0, 1.0, 0.0 @);
p4 = (@ 1.0, 1.0, 0.0 @);
p5 = (@ 0.0, 0.0, 1.0 @);
p6 = (@ 1.0, 0.0, 1.0 @);

pg1 = (@ p1, p2, p4, p3 @: red @);
pg2 = (@ p1, p2, p6, p5 @: green @);
```

After these assignments pg1 represents a red unit square in the X,Y plane, whereas pg2 represents a green unit square in the X,Z plane.

(End of example.)

Although 'procedure' is not a C concept, we will use it to denote a function that does not return a value (unless via parameters).

To create scenes CG knows the procedure

```
create(nm)
string nm;
```

A call to this procedure results in the creation of a scene with name nm, provided that nm does not yet occur as name of a scene. (String is a predefined type, only the first eight characters of a string are significant.) The scene thus created consists of an empty bag of polygons and an empty bag of links.

To add polygons and links to a scene we have the procedures

```
addpol (nm, p)
string nm; polygon p;

addlink(nm, l)
string nm; link l;
```

A call to these procedures effects the addition of the polygon p and the link l respectively to the polygon bag and link bag respectively of the scene with the name nm (provided that a scene with name nm exists).

Polygon expressions being described above, we will now discuss link expressions. A link expression may either be a link identifier or a tuple between CG-like angle brackets. It may formally be described as:

link-expression:
- *link-identifier*
- <@ *string* @>
- <@ *string, matrix-identifier* @>
- <@ *string, matrix-identifier, att-list-identifier* @>

String stands for the name of a scene; matrix-identifier for a variable with a value of the predefined type matrix44, which is a two-dimensional float array: **float[4][4]**, and att-list-identifier for a variable

with a value of the predefined type **atlist,** which is an integer array:

index 0: blinking: (0=non-blinking, 1=blinking)
index 1: detectability: (0=undetectable, 1=detectable)
index 2: highlighting: (0=non-highlighting, 1=highlighting)

A link-expression without matrix-identifier denotes a link with the matrix

```
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
```

associated with it; a link-expression without att-list-identifier denotes a link with default values for all attributes. Default values are: non-blinking, detectable and non-highlighting.

In general the matrix associated with a matrix identifier has the following form:

```
  A     T
0 0 0   1
```

with A specifying a lineair transformation of the 3-dimensional space and T=(Tx,Ty,Tz) representing a translation.

Adding a link to a scene S creates a reference from S to the scene T whose name occurs in the link. Denoting that reference by S --> T , it is not allowed to introduce cycles of the form

```
--> S1 --> S2 -->  ...  --> Sn --
|                                |
|                                |
-----------------------------------
```

where S1, S2, ..., Sn denote scenes.

Example 3:

Let pg1 and pg2 be defined as in example 2. Then after execution of the statements

```
create("wing");
addpol("wing", pg1);
addpol("wing", pg2);
```

there exists a scene with name wing, an empty link bag and a polygon bag with the two polygons pg1 and pg2.

(End of example.)

Example 4:

Let pg1 be defined as in example 2 and let the matrix-identifier mt have the value

```
1 0 0 0
0 0 1 0
0 1 0 0
0 0 0 1
```

then after execution of the statements

```
create("plane");  addpol("plane", pg1)
create("wing");
addlink("wing", <@ "plane" @>);
addlink("wing", <@ "plane", mt @>);
```

the scene wing consists of an empty polygon bag and a link bag with two links, both to the same scene consisting of the only polygon pg1. One of the links has matrix mt associated with it. As it will be shown in the sequel, the display of the scene wing will show precisely the same polygons as a display of the scene wing in the preceding example; there is however one difference, both the polygons from this example are colored red, whereas in the preceding example one is red and the other is green.

(End of example.)

To remove scenes CG knows the procedure

```
remove(nm)
string nm;
```

A call to this procedure results in the removal of the scene with name nm (provided that such a scene exists); moreover all links containing the scene name nm are deleted from all link bags as well.

Note, CG does not know the type scene and, hence, there are no variables of type scene. Scenes may be referenced by names adhered to them; every scene is labeled with a unique name.

To display a scene on a screen CG knows the procedure

```
display(nm, vs)
string nm; viewspec vs;
```

The effect of a call to that procedure is that the scene with name nm is displayed on the screen. To transform the three-dimensional coordinates in which the scene is defined into the two-dimensional coordinates of the screen, viewing transformations, as specified by the parameter vs, are applied.

The predefined type **viewspec** is defined as follows:

```
typedef struct { /* viewing specification */

    short  proj;                  /* projection type :PARALLEL or PERSPECTIVE */
    point  VRP;                   /* view-reference point (user coordinates) */
    point  VPN;                   /* viewplane normal (user coordinates)     */
    point  VUP;                   /* view-up vector (user coordinates)       */
    float  ppd;                   /* projection plane distance (VRC) */
    point  PRP;                   /* if projection is perspective: eye-point
                                     else the vector connecting PRP and the centre of the
                                     window defines the direction of the parallel
                                     projection (WC relative to VRP) */
    float Umin,Umax,Vmin,Vmax;    /* window in projection plane (VRC) */
    float Nmin,Nmax;              /* hither and yon (VRC) */

} viewspec;
```

VRC means View Reference Coordinates; these coordinates are defined by the U, V and N axes. These axes specify the viewing space of the scene.

The display of a scene S not only results in the display of the polygons from the polygon bag of S, but also all the polygons from the scenes referenced by S are displayed. The coordinates of those polygons however are premultiplied with the associated matrices and the associated attributes are applied. It should be clear that also the polygons from scenes referenced by the scenes referenced by ... referenced by S are displayed.

As a matter of fact hidden surfaces are not shown.

3. Interaction

Interaction is an important aspect of many computer graphics applications. Interaction is a form of input/output. Data is exchanged between a (human) user and the program. Output of the program can be in the form of text and pictures. Input can be entered through several devices with different characteristics. In this section we will only be concerned with graphical interaction via the screen. The user points to something already displayed on the screen, that is to say, a location on the screen is selected. Because we are concerned with raster graphics only, we assume that always a pixel is selected. It is irrelevant which input device is used; it does not matter whether it is a joystick, stylus, mouse, fingertip or other. What matters only is that in some way or another a pixel is selected. For interaction via the screen CG is provided with the procedure **getpath** and the type **linklist.**

With the introduction of **getpath** we aim at achieving the following effect. Upon calling, **getpath** eventually yields a polygon and a list of links. Which polygon and which list of links is returned depends upon the screen contents and the pixel pointed at by the user of the program. The screen contents is fully determined by the latest call of **display.** The effect of a call to **getpath** is only defined if the pixel pointed at has got a value (color) due to the display of a polygon.

The procedure **getpath** has four arguments:

```
getpath(nm, p, ll, n)
string nm; polygon *p; linklist *ll; int *n;
```

Due to a call the variables nm, *p, *ll and *n will get the following values:

nm: the name of the scene passed as an argument to the most recent call of display;

*p: the polygon due to which the pixel pointed at has got its value;

*ll and *n: a list of *n links; if *p belongs to the polygon bag of nm then *n = 0 otherwise the first element of ll is a link of the link bag of nm such that the pixel pointed at has got its value due to the display of the referenced scene. The same applies to the other elements of ll. More formally it may be described as follows. Denoting by ll[i] the i-th element from ll and nm[i] being the scene occurring in ll[i] then:

if *n = 0 then *p belongs to the polygon bag of nm otherwise

(i) ll[1] belongs to the link bag of nm such that the pixel pointed at has got its value due to the display of the polygons of nm[1];

(ii) ll[i] is a link from the link bag of nm[i-1] (for 1 < i <= n) such that the pixel pointed at has got its value due to the display of the polygons of nm[i];

(iii) *p belongs to the polygon bag of nm[n].

Example 5.

Let the scenes wing and plane be defined as in example 4. If after execution of

```
display("wing", vs)
```

the pixel corresponding with the coordinates (0.5, 0, 0.5) is pointed at, then the call

```
getpath(nm, p, ll, n)
```

yields the following values:

```
nm  = "wing"
*p  = pg1
*ll = <@ "plane", mt @>
*n  = 1
```

(End of example.)

In order to process the information obtained by a call to getpath CG is provided with the following function having linklist as an argument:

```
link getlink(lol, i)
linklist *lol; int i;
```

The call

```
getlink(lol, i)
```

yields the i-th link of the linklist lol.

(A typical statement sequence is

```
display("scene");
getpath(nm, p, ll, n);
getlink(ll, 1)      .
```

)

Note that **getlink(lol,i)** is a link-expression and therefore may occur as an argument in a call to **addlink.** In order to obtain the flexibility required to write interactive applications, CG is also provided with the procedure

```
dellink(nm, l)
string nm; link l;
```

A call to **dellink** effects the removal of the link l from the link bag of the scene with name nm.

Similarly, CG knows the procedure

```
delpol(nm, p)
string nm; polygon *p;
```

to remove polygons from a polygon bag.

Results of a call to **getlink** may be passed as arguments to **dellink.** In the same way the polygon which results from a call to getpath may be passed as an argument to **delpol.**

For the sake of completeness we mention here three other forms of a link-expression:

```
chnm (l,nm)
chmat(l, m)
chatt(l, a)
```

where l, nm, m and a denote a link-expression, a name of a scene, a matrix-identifier and a att-list-identifier respectively. The expressions denote the links obtained from the link l by replacing the name

of the scene, the matrix-identifier and the att-list-identifier by nm, m and a respectively.

To change names of scenes CG is provided with the procedure

rename(nm1, nm2)
string nm1, nm2;

Provided that a scene with name nm1 exists, a call to this procedure has the following effect. If a scene with name nm2 does not exist it is created. Anyway the polygon bag and link bag of nm2 are made equal to those of nm1, next both the polygon and the link bag of nm1 are set to empty.

To open the workstation on which the scenes are to be displayed the procedure

openws()

has to be called before the first call of display. The workstation remains open for output and input until the procedure

closews()

closes it.

4. Example

As an example we consider the simplest interactive graphical application one can think of: a number of objects are displayed on the screen and the user of the program may -by pointing at a pixel- randomly select an object to be removed.

In CG programs the normal include mechanism as in C programs may be used. An example is shown in line 4. The file *typesvs.h* is supposed to contain the definition of the struct **viewspec** as defined at the description of the procedure **display** above.

The file *defaultvs.c* (line 16) contains an initialization of the viewing specification. This file is listed below the program.

In the program, as it is presented here, the objects are chosen the be quadrangles. The user of the program is requested -via an appropriate message- to type in the coordinates of the four vertices of a quadrangle (see lines 19-25) and he may specify a number of transformation matrices by typing in the sixteen coefficients of each matrix (see lines 37-41).

After this a scene consisting of the original quadrangle and the quadrangles obtained by applying each transformation to the original quadrangle is displayed on the screen (see lines 18, 27-28, 41, 46).

Color is defined in lines 8 and 15 and the viewing specification in lines 1. These viewing parameters specify a parallel view of the unit cube.

Next the user may select a number of times a quadrangle to be removed from the screen (see lines 50-58). The quadrangle is selected by pointing at a pixel of the screen (see line 56). After the removal of a link (see line 57), the scene is displayed again (line 58).

```
1   main()
2   {
3   #define RED 200
4   #include "typesvs.h"
5         float c1, c2, c3; matrix44 mt;
6         point p1, p2, p3, p4;
7         char c;
8         int i, j, color, *n;
9         viewspec defaults;
10        linklist ll;
11        polygon *p;
12
13        wsopen();
14
15        color = RED;
16  #include "defaultvs.c"
17
18        create("example1"); create("quad");
19        printf("type coordinates of the 4 vertices of a quadrangle\n",
20        "(in clockwise order; three coordinates for each vertex)\n");
21
22        scanf("%f %f %f", &c1, &c2, &c3); p1= (@ c1, c2, c3 @);
23        scanf("%f %f %f", &c1, &c2, &c3); p2= (@ c1, c2, c3 @);
24        scanf("%f %f %f", &c1, &c2, &c3); p3= (@ c1, c2, c3 @);
25        scanf("%f %f %f", &c1, &c2, &c3); p4= (@ c1, c2, c3 @);
26
27        addpol("quad", (@p1,p2,p3,p4 @: color @));
28        addlink("example1", <@ "quad" @>);
29
30        do {
31           c=' ';
32           while ((c!='y') && (c!='n')) {
33               printf("add a transformed quadrangle? (y/n) \n");
34               c=getchar();
35           }
36           if (c=='y') {
37               printf("type in row wise order 4x4 entries of the \n");
38               printf( "transformation matrix\n");
39               for (i=0; i<4; i++)
40               for (j=0; j<4; j++) scanf("%f", &mt[i][j]);
41                     addlink("example1", <@ "quad", mt @>);
42               }
43           }
44        while (c!='n');
45
46        display("example1", defaults);
47
48        do {
49           c=' ';
50           while ((c!='y') && (c!='n')) {
51                 printf("remove a quadrangle? (y/n)\n");
52                 c=getchar();
53           }
54           if (c=='y') {
```

```
55                  printf("select a quadrangle \n");
56                  getpath("example1", &p, &ll, n);
57                  dellink("example1", getlink(ll, 1));
58                  display("example1", defaults);
59                  }
60              }
61          while (c!='n');
62
63          wsclose();
64      }
```

A possible initialization of the viewing specification is:

```
defaults.proj = PERSPECTIVE;
defaults.VRP  = (@ 0., 0., 0. @);
defaults.VPN  = (@ 0., 0., 1. @);
defaults.VUP  = (@ 0., 1., 0. @);
defaults.ppd  = 0.;
defaults.PRP  = (@ 0.0, 0.0, 2. @);
defaults.Umin = -1.;
defaults.Umax = 1.;
defaults.Vmin = -1.;
defaults.Vmax = 1.;
defaults.Nmin = -1.;
defaults.Nmax = 1.;
```

5. References

[1] **Bourne, S.R., The Unix System** (International Computer Science Series; 6) Addison-Wesley, Reading, Massachusetts, 1983

[2] **Freeman, H., Computer Processing of Line-drawing Images** , ACM Comp. Surv., Vol. 6, 1974, pp. 57-97

[3] **Harbison, S.P. and Steele, G.L., C: A Reference Manual** , Prentice Hall, Englewood Cliffs, N.Y. 1984

[4] **ISO, Graphical Kernel System (GKS)** , Functional Description, ISO 7942, 1984

[5] **ISO TC97/SC21/WG5-2 N277 Rev. Graphical Kernel System for Three Dimensions (GKS-3D),** Functional Description, 1985

[6] **ISO TC97/SC21/WG2 N819, Programmers Hierarchical Interactive Graphical System, (PHIGS)** , 1985

[7] **Johnson, S.C., Yacc: Yet Another Compiler-Compiler** , Unix Programmer's Manual, Seventh Edition, Volume 2b, Bell Telephone Laboratories Inc. Murray Hill, N.J. 1978

[8] **Kernighan, B.W. and Ritchie, D.M., The C Programming Language** , Prentice-Hall, Englewood Cliffs, N.J. 1978

[9] **Klinger, A., Pattern and Search Statistics** , in: Optimizing Methods in Statistics (J.S. Rustagi ed.), Academic Press, New York, 1971, pp 303-337

[10] **Lemmens, W.J.M., The CG Preprocessor** , THE Internal Report, 1986

[11] **Mallgren, W.R., Formal Specification of Interactive Graphics Programming Languages** , The MIT Press, Cambridge, Massachusetts, 1983

[12] **Ritchie, D.M. and Thomson K., The Unix Time Sharing System** , Comm. ACM Vol. 17, 1974, pp. 365-375

[13] **Samet, H., The Quadtree and Related Hierarchical Data Structures** , ACM Comp. Surveys, Vol. 16, 1984, pp. 187-260

[14] **Unix Programmers' Manual** , Seventh Edition, Bell Telephone Laboratories Inc. Murray Hill, N.J. 1979

[15] **Kalisvaart, J., CG User Guide,** THE Internal Report, 1986

Appendix 1: CG syntax rules

For the grammar rules we use the syntax notation of [8] : alternatives are listed on separate lines.

point-expression:
point-identifier
point-expression @+ point-expression
point-expression @- point-expression
float @ point-expression*
(@ float-expression, float-expression, float-expression @)

polygon-expression:
polygon-identifier
(@ point-expression-list @: color-index @)
extend (polygon-identifier, point-expression-list)
chcol (polygon-identifier, color-index)

point-expression-list:
point-expression
point-expression-list, point-expression

link-expression:
link-identifier
<@ string @>
<@ string, matrix-identifier @>
<@ string, matrix-identifier, att-list-identifier @>

Appendix 2: CG functions and procedures

name	returns	ref	description
addlink(nm,lnk)	-	2	add lnk to scene with name nm
addpol(nm,plg)	-	2	add plg to scene with name nm
chatt(lnk,att)	link	3	change attributes of link to att
chcol(plg,i)	polygon	2	change color index of plg to i
chmat(lnk,mt)	link	3	change matrix of link to mt
chnm(lnk,nm)	link	3	change scene name of link to nm
create(nm)	-	2	create scene with name nm
dellink(nm,lnk)	-	3	delete lnk from scene with name nm
delpol(nm,plg)	-	3	delete plg from scene with name nm
display(nm,vs)	-	2	display scene with name nm applying vs
extend(plg,pnt)	polygon	2	extend the pointlist of plg with pnt
getlink(*ll,i)	link	3	get i-th link of ll
getpath(nm,*plg,*ll,*i)	-	3	get plg, ll of length i and scene of indicated pixel
remove(nm)	-	2	remove scene with name nm
rename(nm1,nm2)	-	3	make scene with name nm2 equal to scene nm1
wsclose()	-	3	close workstation
wsopen()	-	3	open workstation

abbreviation	meaning
att	variable of type atlist
i	variable of type int
ll	variable of type linklist
lnk	variable of type link
mt	variable of type float[4][4]
nm	variable of type string
plg	variable of type polygon
pnt	variable of type point
ref	chapter containing the definition
vs	variable of type viewspec

THE CG PREPROCESSOR

W.J.M. Lemmens

Eindhoven University of Technology

0. Introduction: Language extension by preprocessor.

A preprocessor is a program that processes some text in some way before it is actually used. The preprocessor's task is often the conversion of extensions to a specific programming language to standard language constructs. It is to be applied to programs written in such an extended language before compilation. Ideally a preprocessor should process only non-standard language elements and leave the rest of the source text unchanged. But this is only possible if the added elements have been set apart from the rest of the text, e.g. by providing them with a unique mark that does not occur in the rest of the program, and if they do not contain standard language elements. Mostly, however, there is a certain duplication of functions among the preprocessor and the compiler that is to be applied successively. A preprocessor could in principle have all the functions of a compiler, like syntax analysis, type checking and code generation. Preprocessors are made in order to save time and effort, as compared to creating a new compiler, so they should be as small as possible. Moreover, the penalty of added processor time for the translation of source code written in the extended language should be kept within reasonable bounds.

CG is an extended form of the C language, especially prepared for raster graphics applications [2]. It adds a number of keywords and syntax rules, but allows the extensions to be fully mixed with C expressions. This makes it necessary to duplicate some parts of the compiler functions. The CG programs are not fully parsed, however. Parsing is done only as far as needed for detection and analysis of the constructs and data types that have been added to the C language. In order to keep the preprocessor as small as possible, several measures have been taken:

- A number of new symbols have been added that allow their meaning to be deduced independent of context.
- Parsing is done in two stages: a eight-state preparser and a LALR(1)-parser produced by yacc [1].
- Data type information is used in the (context-independent) parsing process.
- The symbol set has been reduced: several different keywords are mapped to the same token.

In this way we avoid the use of a parser that will fully analyse the input text. The most notable differences to a compiler are the absence of evaluation of control structure, as far as **if, while-do, switch, case, goto** statements are concerned, and a minimalisation of the evaluation of parts of expressions that do not contain CG-types or -operators. A C-expression is seen as a mere string of C-tokens, interspersed with '(', ')' and ',' -characters.

The preprocessor is built in a number of layers: The lowest level is the operating system level, at which files are opened and characters are received from a terminal, for example. Next are the preprocessor-routines that use these functions to serve other preprocessor functions, and so on. The preprocessor may also be subdivided into functional blocks, straight across the levels. The main blocks are the input text evaluation and the output text generation. In turn, these blocks may again be subdivided into respectively text analysis and symbol table maintenance, and parse tree building and code generation.

The parser has been built with the help of Yacc [1], a program that builds parsers from specifications. The rest of the preprocessor has been written in C.

Table 1: preprocessor structure.

Level	text analysis routines	code generation routines
1	*nxtchar*	*bufupd*
2	*getname, cmpstr, digest*	
3	*findnode, addnode*	
4	*srchscp, dealloc*	
5	*explexan*	
6	*yylex*	*prefop, infop, varname, extend, pnode, code, pass*
7	parser - analysis part	parser - actions part

1. Description of preprocessor parts.

Table 2: short description of the preprocessor functions.

Level	Function	Remarks:
1	*bufupd*	Adds next character to circular buffer.
	nxtchar	Reads input stream, character by character, outputs any C-preprocessor directives immediately, skips comments and passes remaining characters to the lexical analyser and the character buffer.
2	*getname*	Accumulates alpha, digit and '_' characters into an name variable.
	cmpstr	Compares two strings to determine their lexical order.
	digest	Starting with an opening character, like '(', "eats" characters until the matching closing character is found.
3	*findnode*	Symbol table lookup.
	addnode	Add new symbol to symbol table.
4	*srchscp*	Search whole scope for 'name'-symbol.
	dealloc	"Peel off" one level of the symbol table.
5	*explexan*	Translate characters from nxtchar into tokens.
6	*yylex*	Preparser. Processes token strings before passing them to the parser.
	prefop,	Prefix operator (Prefop, infop, varname, extend, pnode are routines that create and attach new nodes to parse tree.)
	infop,	Postfix operator
	varname,	String (identifier)
	extend,	Extends string
	pnode	Function call
	code	Produces code from parse tree.
	pass	Passes text unchanged.
7	*yyparse*	Parser, applies syntax rules to token stream, builds parse tree, instigates output text generation.
	main	Initialisation and invocation of parser.

1.1. The input routine *nxtchar*.

The main purpose of the routine is to extract characters from the input stream and pass them tot the lexical analyser. Not all characters are passed, however. nxtchar first scans the input stream for C-preprocessor directives (lines starting with #) and comments (character strings enclosed in /* and */ -symbols). The C-preprocessor directives are copied into a buffer, but they are not passed to the lexical analyser. The comments are discarded altogether. The characters that remain are presented to the lexical analyser and they are also put into the circular buffer for later use in the output stream generation.

Every time a character is to be saved the routine first checks whether the buffer is full. Should this be the case, it first empties it into the output stream, after which the character is inserted into the then empty buffer. The user is informed of this occurrence by a warning that is sent to the terminal, while it is also announced by a comment in the output stream.

1.2. The lexical analyser *explexan*.

The lexical analyser is a relatively simple function that assigns a provisional token to every part of the input stream. It recognises keywords, CG-identifiers and certain other characters or character combinations. Spaces, tabs or carriage returns are skipped. The output of the lexical analyser is fed to the eight state preparser that will further process the tokens.

Table 3: tokens produced by the lexical analyser.

input string	output token	remark
"*"?	DECLKW, SUEKW, POINT,	1, 2
[a-zA-Z]	POLYGON, LINK, LINKLIST,	
[a-zA-Z0-9_]*	TRLKW, UNOP,	
	CG_ID IDENTIFIER	
@*	CGMUL	3
@)	CGCLOSE	3
@>	LKCLOSE	3
@+	CGPLUS	3
@-	CGMIN	3
@:	CGDEL	3
(@	CGOPEN	3
<@	LKOPEN	3
,	','	
:	':'	
;	';'	
(	'('	
)	')'	
{	'{'	
}	'}'	
[	'['	
]	']'	
=	'='	
EOF	EOF	
\"[\"]*\"	STRING	1
\[" "\t\n]	--	1
default	C	

Remarks:

1. Strings described by regular expressions as defined in: [3]

2. The keywords and the corresponding tokens they produce, are listed in table 4.
3. These are special CG-operators.

Table 4: keywords and tokens.

keyword	token	keyword	token
auto	DECLKW	**int**	DECLKW
break	CTRLKW	**link**	LINK
case	CTRLKW	**linklist**	LINKLIST
char	DECLKW	**long**	DECLKW
continue	CTRLKW	**point**	POINT
default	CTRLKW	**polygon**	POLYGON
do	CTRLKW	**register**	DECLKW
double	DECLKW	**return**	CTRLKW
else	CTRLKW	**short**	DECLKW
entry	CTRLKW	**sizeof**	UNOP
enum	SUEKW	**static**	DECLKW
extern	DECLKW	**struct**	SUEKW
float	DECLKW	**switch**	CTRLKW
for	CTRLKW	**typedef**	DECLKW
goto	CTRLKW	**union**	SUEKW
if	CTRLKW	**unsigned**	DECLKW
		while	CTRLKW

A large part of all input character combinations will default to the C-token. The diversity of the output tokens is thus reduced as compared to a full compiler. This results in an important reduction of the software that has to deal with the output of *explexan*.

1.3. The preparser *yylex*.

The preparser will receive a stream of tokens from the lexical analyser and process it before it is passed to the parser proper. In the process, some tokens will be replaced by others and some will be discarded altogether, resulting in a further reduction of the number and the diversity of the tokens that are offered to the parser. In certain states the preparser will be "insensitive" to certain tokens. They will not be passed to the parser at all. In some states a whole class of tokens is replaced by the same default token.

The actions taken by the preparser are dependent upon the state in which it operates. Switching between the possible states is effected by the parser or by the preparser itself. In the latter case the transition depends upon the input token or the lookahead character (the character following the input string that is identified by the current token). A transition may be accompanied by the presentation of a specific token, or it may occur directly, without generating any output.

There are eight states, which may be divided into two groups of associated states and a departure state, INIM and INID, from which one of both is chosen. The first group, consisting of DECM, TAIL and DINIT, handles declarations. The second, EXPM, CEXP and SUBSC, statements.

When in state INIM an IDENTIFIER is found, state INID is entered, effectively deferring the choice of on of the groups to the next symbol.

DECM processes declarations up to an identifier, after which TAIL takes over. TAIL tries to identify any parameter lists (generating token PARLIST) or array subscripts. If a = -character occurs, indicating an initialisation of a variable, DINIT is called, which will replace all of the initialisation with at most one token.

Statements will be processed by EXPM, until a CG identifier has been found, after which CEXP is entered. If CEXP finds a subscript, it switches to SUBSC for further processing, but if the following symbol does not indicate a subscript or an assignment, control is passed directly to EXPM again, except for an assignment symbol = without generating any output ot reading any new input symbol. Due to

this construction, subscripts and assignments are only evaluated after CG variables.

yylex uses another function, *digest*, to further reduce the stream of output tokens. *digest* is called with two different characters as parameters. These represent the beginning (which has already been detected by *yylex*) and end of a string that should be "digested", that is, put into the buffer, but not analysed by *yylex* or *explexan*. Any time *digest* detects the starting symbol in the character stream it recursively calls itself, so only an unpaired end symbol will finish digestion. So in fact *digest* provides extra states, or substates to the main *yylex* states.

Figure 1. illustrates the structure of the preparser with its states, whereas table 5 gives a detailed overview of the individual states.

Table 5. Overview of preparser states.

state: INIM

input symbol	next character	output symbol	next state	remarks
IDENTIFIER	--	IDENTIFIER	INID	
CG_ID	--	CG_ID	CEXP	
DECLKW	--	DECLKW	DECM	
SUEKW	--	SUEKW	DECM	
POINT	--	POINT	DECM	
POLYGON	--	POLYGON	DECM	
LINK	--	LINK	DECM	
LINKLIST	--	LINKLIST	DECM	
';'	--	';'	EXPM	
'{'	--	'{'	EXPM	
'}'	--	'}'	EXPM	
STRING	--	STRING	EXPM	
default	--	C	EXPM	1.

state: INID

input symbol	next character	output symbol	next state	remarks
IDENTIFIER	--	IDENTIFIER	TAIL	
';'	--	';'	INIM	
'(', ')', ','	--	'(', ')', ','	EXPM	
default	--	C	EXPM	

state: DECM

input symbol	next character	output symbol	next state	remarks
IDENTIFIER	--	IDENTIFIER	TAIL	
CG_ID	--	CG_ID	TAIL	
DECLKW	--	DECLKW	DECM	
SUEKW	--	SUEKW	DECM	
POINT	--	POINT	DECM	
POLYGON	--	POLYGON	DECM	
LINK	--	LINK	DECM	
LINKLIST	--	LINKLIST	DECM	
CTRLKW	--	CTRLKW	EXPM	
UNOP	--	UNOP	EXPM	
';'	--	';'	INIM	
'{', '}', ','	--	'{', '}', ','	DECM	
EOF	--	EOF	DECM	
default	--	--	DECM	

state: TAIL

input symbol	next character	output symbol	next state	remarks
IDENTIFIER	--	IDENTIFIER	TAIL	
'('	--	PARLIST	DECM	2.
','	--	','	DECM	
'{'	--	'{'	DECM	
';'	--	';'	INIM	
'='	--	'='	DINIT	
'['	--	--	TAIL	2.
default	--	--	TAIL	

state: DINIT

input symbol	next character	output symbol	next state	remarks
'{'	--	--	DINIT	2.
--	','	C	DECM	
--	';'	C	INIM	
--	default	--	DINIT	

state: EXPM

input symbol	next character	output symbol	next state	remarks
CG_ID	--	CG_ID	CEXP	
CGOPEN	--	CGOPEN	EXPM	
CGCLOSE	--	CGCLOSE	EXPM	
LKOPEN	--	LKOPEN	EXPM	
LKCLOSE	--	LKCLOSE	EXPM	
CGDEL	--	CGDEL	EXPM	
CGMUL	--	CGMUL	EXPM	
CGPLUS	--	CGPLUS	EXPM	
CGMIN	--	CGMIN	EXPM	
STRING	--	STRING	EXPM	
'('	--	'('	EXPM	
')'	--	')'	EXPM	
','	--	','	EXPM	
';'	--	';'	INIM	
'{'	--	'{'	INIM	
'}'	--	'}'	INIM	
default	--	C	EXPM	

state: CEXP

input symbol	next character	output symbol	next state	remarks
'['	--	'['	SUBSC	
']'	--	']'	CEXP	
'='	--	'='	EXPM	
default	--	--	EXPM	3.

state: SUBSC

input symbol	next character	output symbol	next state	remarks

','	--	','	EXPM	
')'	--	')'	EXPM	
';'	--	';'	INIM	
']'	--	']'	CEXP	
default	',', ';', ')'	C	EXPM	4.
default	']'	C	CEXP	4.

Remarks:

1. default = any symbol,except those mentioned above.

   -- = Don't care for input symbol and next character columns, no output for output symbol column.

2. Input text *digest* -ed up to and including unmatched closing symbol, e.g. { {1,2,3}, {4,5,6}, {7,8,9} } will generate no token.

3. Immediate switch to next state. No output is generated, no new symbol is read.

4. No output generated for all characters up to those listed under 'next character'.

1.4. The parser.

Using only 104 rules, the preprocessor parser is quite simple as compared to a compiler parser. Noticeable differences to a compiler are the lack of control structure evaluation and the far going simplification of C expressions. CG operations, however, have to be analysed in depth, as they should be replaced by function calls. In order to make this possible, the parser should be able to identify CG operators and CG operands. Dependent upon the operation, the latter could be C expressions or **point, polygon, link, linklist** variables. The CG variables are identified by use of the symbol table, which will be described later on.

while, if, switch -clauses need not be analysed as they will always appear to the left side of an assignment statement and never inside a parameter list. At the left of a = -symbol there will never be need for the insertion of a function call by the preprocessor, everything that occurs there may be considered as an undivided whole, so it is of no interest where the expression associated with the **if** or **while** ends and the CG-assignment begins.

Typechecking is done by comparing the types of the operands in an operation (=, @+, @-, @*), using rules that are contained in the parser actions. The preprocessor distinguishes only five data types: *C_type*, **point** type, **polygon** type, **link** type and **linklist** type. Typechecking is performed fully independent of the syntax checking, so type errors will not influence the parsing process, although they are signaled on the terminal. The variable GXT will at any moment have the value of the type of the (sub)expression that is currently being processed by the parser. It is updated every time a CG_ID token is passed to the parser or when a CG specification (expression enclosed in (@ and @)) occurs. Type checking is performed by testing the value of GXT at certain points.

Definition of parser:

Possible tokens:

C, STRING, DECLKW, SUEKW, CTRLKW, UNOP, PARLIST POINT, POLYGON, LINK, LINKLIST, LKOPEN, LKCLOSE CGMUL, CGPLUS, CGMIN, CGOPEN, CGCLOSE, CGDEL CG_ID, IDENTIFIER

Production rules:

```
start:
program: subprogram
        | program subprogram
        ;
subprogram: declaration ';'
                {pass(); mode=DECM; GXT=0;}
        | function_specifier
                {pass(); GXT=0; stpt=(stack *)malloc(sizeof(stack));
                 stpt->lroot=NULL; stpt->prev=TOS; TOS=stpt;}
         opt_declarations
         block
                {pass(); mode=DECM; GXT=0;}
        ;
block: '{'
                {pass(); mode=INIM;}
          body '}'
                {stpt=TOS;
                 if (TOS->lroot!=NULL)
                    {dealloc(TOS->lroot);
                    }
                 TOS=TOS->prev; free(stpt);
                }
        ;
function_specifier: function_declarator
        | typed_function_spec
        ;
function_declarator: declarator PARLIST
        ;
typed_function_spec: CG_keyword function_declarator
        | SUE_tag function_declarator
        | declarator function_declarator
        | DECLKW function_declarator
        | DECLKW typed_function_spec
        ;
opt_declarations:
        | declarations
        ;
body:   opt_progr_part
        | declaration ';'
                {pass(); GXT=0;}
          body
        ;
opt_progr_part:
        | progr_part
        ;
progr_part: progr_element
        | progr_part progr_element
        ;
progr_element: statement
                {pass();}
        ;
statement: CG_assignment ';'
                {code($1);}
        |
```

```
        {stpt=(stack *)malloc(sizeof(stack));
                stpt->lroot=NULL; stpt->prev=TOS; TOS=stpt;}
        block
      | C_expr ';'
                {code($1);}
      | C_expr
                {code($1); stpt=(stack *)malloc(sizeof(stack));
                 stpt->lroot=NULL; stpt->prev=TOS; TOS=stpt;}
        block
      | C_string '(' C_expr ';' C_expr ';' C_expr ')'
                {code(pnode($1, infop( ";", $3, infop( ";", $5, $7))));}
        statement
      | ';'
      ;
declarations: declaration ';'
                {pass(); GXT=0;}
      | declarations declaration ';'
                {pass(); GXT=0;}
      ;
declaration: CG_specifier
      | SUE_specifier
      | C_specifier
      | DECLKW declaration
      ;
SUE_specifier: SUE_keyword SUE_decl opt_decl_list
      ;
SUE_decl: '{' declarations
                {mode=DECM;}
        '}'
      | SUE_tag '{' declarations
                {mode=DECM;}
        '}'
      | SUE_tag
      ;
opt_decl_list:
      | decl_list
      ;
CG_specifier: CG_keyword decl_list
      ;
decl_list: init_declarator
      | decl_list ',' init_declarator
      ;
init_declarator: declarator opt_init
      | function_declarator
      ;
opt_init:
      | '=' C_expr
      ;
SUE_tag: SUE_keyword IDENTIFIER
      ;
SUE_keyword: SUEKW
                {GXT=CTYP;}
      ;
CG_keyword: POINT
```

```
                {GXT=PTTYP;}
        | POLYGON
                {GXT=PGTYP;}
        | LINK
                {GXT=LKTYP;}
        | LINKLIST
                {GXT=LLTYP;}
        ;
C_specifier: DECLKW decl_list
        | declarator decl_list
                {free($1);}
        ;
declarator: IDENTIFIER
                {int cond;
                 if (mode==TAIL)
                 {if (GXT>CTYP)
                    {cond=findnode(TOS->lroot);
                     if (cond!=0)
                     {cnode= addnode(cnode,cond);
                      cnode->idnr=$1<0? -GXT: GXT;
                      if (cond<0) TOS->lroot=cnode;
                     }
                     else fprintf(stderr,"Identifier declared twice. line %d\n",LNR);
                 }}else
                  $$=varname();}
        ;
CG_assignment: variable '='
                {HGXT=GXT;}
         CG_expression
                {$$=infop("=", $1, $4);}
        ;
mark_CG_ID: CG_ID
                {GXT=$1; $$=varname();}
        | C_expr CG_ID
                {GXT=$2; $$=extend($1);}
        ;
variable: mark_CG_ID
                {$$=$1;}
        | variable '[' opt_C_expr ']'
                {$$=extend(infop("[", $1, $3));}
        ;
expression: CG_expression
                {$$=$1;}
        | C_expr
                {$$=$1;}
        ;
mark_STRING: STRING
                {$$=varname();}
        ;
expr_list: expression
                {$$=$1;}
        | expr_list ',' expression
                {$$=infop( ",", $1, $3);}
        ;
```

```
CG_expression: term
                {$$=$1;}
        | CG_expression plusminus term
                {switch($2)
                        {case 0: {$$=prefop("addpt",$1,$3); break;}
                         case 1: {$$=prefop("subtpt",$1,$3); break;}
                        }
                }
        ;
factor: variable
                {$$=$1;}
        | mark_CG_ID '(' opt_expr_list ')'
                {$$=pnode($1, $3);}
        | spec_list
                {$$=$1;}
        | error
                {errmsg="in factor ";}
        ;
term:   factor
                {$$=$1;}
        | C_expr CGMUL multiplicant
                {$$=prefop("mulpt",$1,$3);}
        ;
plusminus: CGPLUS
                {$$=0;}
        | CGMIN
                {$$=1;}
        ;
multiplicant: factor
                {$$=$1;}
        | '(' CG_expression ')'
                {$$=$2;}
        ;
spec_list: CGOPEN C_expr ',' C_expr ',' C_expr CGCLOSE
                {GXT=PTTYP; $$=prefop( "ptspec", infop( ",", $2, infop( ",", $4, $6)));}
        | CGOPEN CG_expr_list CGDEL C_expr CGCLOSE
                {GXT=PGTYP; $$=prefop( "pgspec", $2, $4);}
        | LKOPEN C_expr LKCLOSE
                {GXT=LKTYP; $$=prefop( "crealnk", $2, NULL);}
        | LKOPEN C_expr ',' C_expr LKCLOSE
                {GXT=LKTYP; $$=prefop( "chmat", prefop( "crealnk", $2, NULL), $4);}
        | LKOPEN C_expr ',' C_expr ',' C_expr LKCLOSE
                {GXT=LKTYP; $$=prefop( "chatt", prefop( "chmat", prefop( "crealnk", $2, NULL), $4), $6);}
        ;
CG_expr_list: CG_expression
                {$$=prefop( "creaptl", $1, NULL);}
        | CG_expr_list ',' CG_expression
                {$$=prefop( "extptl", $1, $3);}
        ;
opt_expr_list:
                {$$=NULL;}
        | expr_list
                {$$=$1;}
        ;
```

```
opt_C_expr:
                {$$=NULL;}
        | C_expr
                {$$=$1;}
        ;
C_expr: C_tail
                {$$=$1;}
        | '(' C_expr_list ')' C_tail
                {$$=infop( "", pnode(NULL, $2), $4);}
        | mark_STRING
                {$$=$1;}
        ;
C_tail: C_string
                {$$=$1;}
        | C_string mark_STRING
                {$$=extend($1);}
        | C_subexpr
                {$$=$1;}
        | C_subexpr C_string
                {$$=infop( "", $1, $2);}
        ;
C_subexpr: C_term
                {$$=$1;}
        | C_subexpr C_term
                {$$=infop( "", $1, $2);}
        ;
C_term: C_string '(' opt_expr_list ')'
                {$$=pnode($1, $3);}
        ;
C_expr_list: C_expr
                {$$=$1;}
        | C_expr_list ',' C_expr
                {$$=infop( ",", $1, $3);}
        ;
C_string: C
                {$$=varname();}
        | declarator
                {$$=$1;}
        | C_string C
                {$$=extend($1);}
        ;
```

The syntax has been formulated in accordance with yacc conventions [1].

1.5. The symbol table.

The preprocessor should be able to detect CG-variables and so it maintains a symbol table, in which all the variables that are within the scope at any moment can be found. The symbol table has been built as a stack of binary trees, whereby the variables that are declared within the block that has last been entered will appear on top of the stack. An entry to the symbol table contains the name of the variable, its type and two pointers, to the left and right subtree, respectively. Every time the parser finds a declaration of a CG-type variable (**point, polygon, link, linklist**) it prepares a new entry, that is inserted into the top tree, using *addnode*, in such a way that it can easily be found by name. After leaving a block, the corresponding tree is popped from the stack and added to the free pool by *dealloc*.

If the lexical analyser encounters an alphabetic character it calls *getname* to have the associated string stored in a variable called *name*. Then it searches the keyword table, using *findnode* to discover if this name refers to a reserved word. If that is not the case, *srchscp* is called, which tries to find *name* in the symbol table by searching the tree stack, level after level, down from the top. If the name does not appear in the symbol table, the token IDENTIFIER is generated, otherwise it will be CG_ID, while the type, from the type field of the symbol table entry, is separately passed to the parser.

1.6. Code generation.

Most of the text that is offered to the preprocessor will have to be passed to the output unchanged. Especially declarations will only be used for information and not be moulded into something new. So after each declaration the parser calls *pass* to transfer the contents of the buffer to the output.

Expressions are treated differently, however, because the added CG-symbols have to be replaced by something else. Therefore the parser builds a parse tree in which CG-operations or -specifications are replaced by indications of the corresponding function calls. The nodes of the tree are generated by the functions *infop*, for an infix operator, *prefop*, for an operation that has to be converted to a function call (prefix operator), *pnode* for a function call, and *varname* for an operand whose representation is to be found in the buffer. The function *extend* may be used to add succeeding parts of the buffer to such a buffer string. *varname* and *extend* together create the "leaf"-nodes of the parse tree. These contain two buffer pointers that indicate the start and the end of the associated string in the buffer.

At the end of the expression the parser calls the *code* function, that converts the tree to linear code that will include the new function calls. Some of the contents of the buffer will never be found in the output code, mainly the new CG symbols. These parts are not indicated in any node of the parse tree, so they will not be put into the output stream during code generation. After every statement and declaration, *pass* is called to transfer the part remaining in the buffer (e.g. containing the closing ;) to the output.

This part may however also contain unwanted symbols, like @). These are prevented from occurring in the output by setting the variable NFPB (nr. of free places in the buffer) to *buflength* (buffer empty), each time after they are detected. The function *code*, as it removes pieces of text from the buffer, should also update NFPB. It does so only, however, if this would result in a higher NFPB value than the current one, thereby avoiding the situation in which an unwanted symbol would still be output, despite a previous setting of NFPB.

The number of free places in the buffer is further updated by *bufupd* and by *pass*.

2.1. Restrictions of CG with respect to C.

Due to the structure of the preprocessor there are a number of restrictions to the names and constructs a programmer may use:

1. Of course CG keywords (**link, linklist, point, polygon**) may not be used for variable or function identifiers.

2. CG-expressions may only occur as assignments or inside brackets (e.g. as part of a parameter list). So **return (@ c1, c2, c3 @);** is not allowed. The statement **return ((@ c1, c2, c3 @));** , however, is.
3. CG-type specifiers (**point, polygon, link, linklist**) may not be renamed (e.g. by a typedef).

2.2. Error Recovery.

Due to its rather involved structure, error recovery of the preprocessor is poor. Many syntax errors will leave the preparser in a false state, generating the wrong tokens, adding to the errors that have already occurred. The result will often be total disruption of the parsing process, leading to buffer overflow and consequent purging of the buffer.

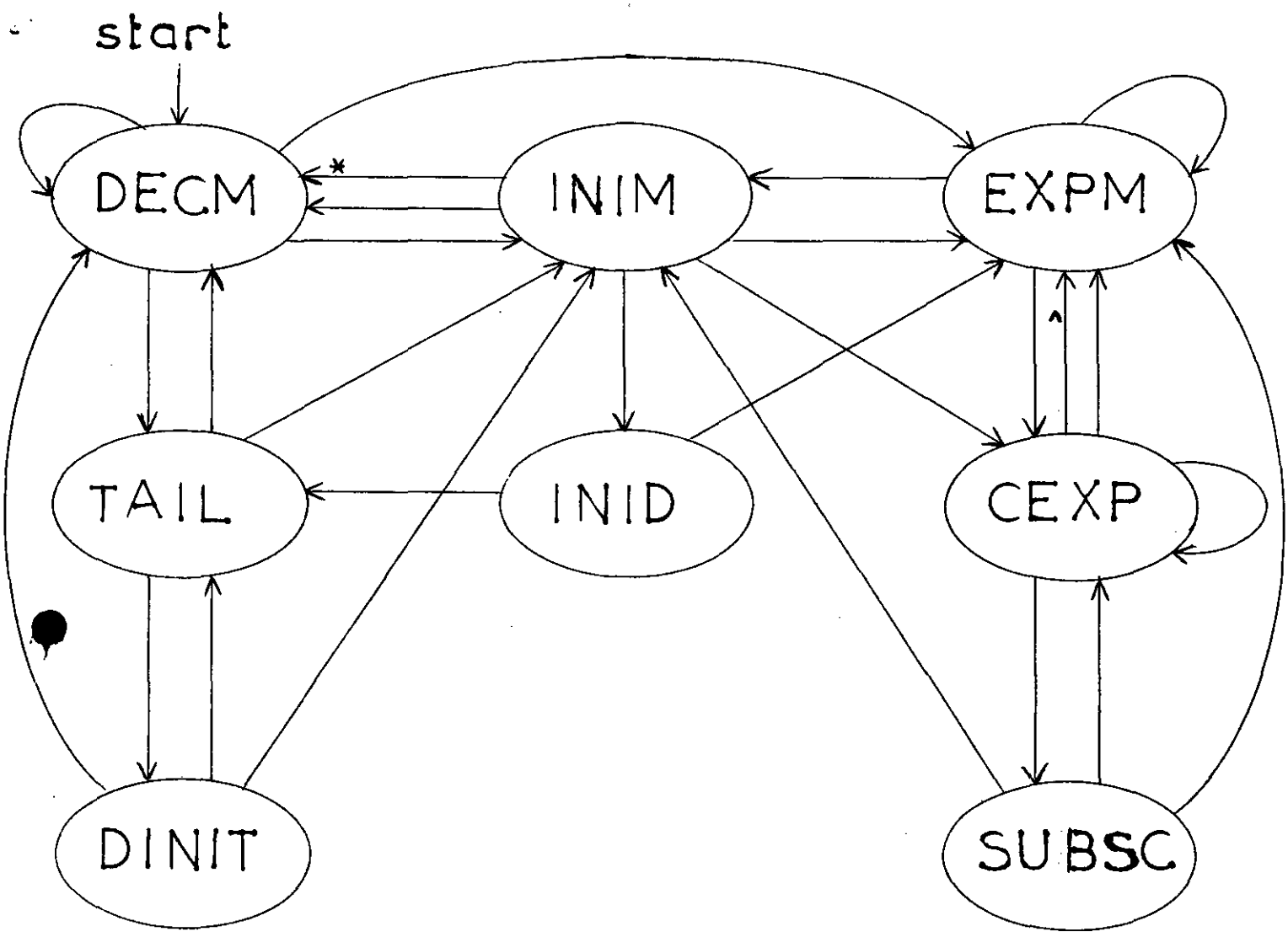
So, many errors will be notified by an error message on the terminal, followed by buffer overflow messages until all the input has been read. In order to get the preprocessor "back on the rails" as fast as possible after the occurrence of an error, special precautions should be taken, which have so far been neglected through lack of time.

3. References.

[1] **Johnson, S.C., Yacc: Yet Another Compiler-Compiler,** Unix Programmers' Manual, Seventh Edition, Volume 2b, Bell Telephone Laboratories Inc., Murray Hill, N.Y., 1978.

[2] **Kessener L.R.A., van Lierop M.L.P., Kalisvaart J., Peters F.J., Language Extensions to study Data Structures for Raster Graphics,** THE Internal Report, 1986.

[3] **Lesk, M.E. and Schmidt E., Lex - A Lexical Analyser Generator,** Unix Programmers' Manual, Seventh Edition, Volume 2b, Bell Telephone Laboratories Inc., Murray Hill, N.Y., 1979.



* transition forced by parser

^ immediate transition (no output)

Figure 1: states and transitions of the preparser yylex.

CG USER GUIDE

J. Kalisvaart

Eindhoven University of Technology

NAME

cg - CG preprocessor and compilation

SYNOPSIS

cg [option] ... file ...

DESCRIPTION

Cg preprocesses source files written in the graphical language CG and passes the preprocessed files to the C compiler cc. CG is an extension of the C programming language.

Arguments whose names end with '.cg' are taken to be cg source files; all other arguments are interpreted by cc.

The cg command is executed in two sequential steps. The first step is a process that produces a file for every '.cg' file with the same name but with the '.cg' extension substituted by the '.c' extension. This process converts typical CG language constructs into C.

If a syntax error occurs in a '.cg' file, then the preprocessing of the file will be aborted in many cases before the entire file has been preprocessed. Only the first error message is a reliable indication of the syntax error, others may be introduced by earlier detected errors.

Only if the first step terminates successfully, i.e. in none of the '.cg' files a syntax error has been detected, the second step, the invocation of cc, is entered.

Because the first step only checks CG language constructs, many possible syntax errors can be detected by cc only. The error messages of cc will refer to the '.c' files, the output files of the first step.

Although the output of the first step is a C source file and thus can be read and edited, it is not recommended to modify it. This would make the '.cg' file obsolete just like any source file becomes obsolete after modifying its object file.

All options of cc are accepted by cg. Options don't control the first step; they are just passed to cc.

To specify the **viewspec** type it is recommended to include the predefined file containing the viewspec definition (see FILES). This file contains CG types and will be expanded first before it will be preprocessed together with the CG source.

A CG program is constructed combining the CG and C syntax rules, but the following restrictions take effect:

The predefined CG constants **FALSE, TRUE, PARALLEL, PERSPECTIVE, NATT** and the predefined CG types **point, polygon, link, linklist, atlist, matrix44, string, viewspec** may not be redefined by a **#define** or by a **typedef** nor used for variable and function identifiers.

CG-expressions may only occur as assignments or inside brackets (e.g. as part of a parameter list). So **return (@ c1, c2, c3 @);** is not allowed. The statement **return ((@ c1, c2, c3 @));** , however, is.

Names identifying different CG type variables within one scope must be different, even if these variables are part of different data structures. So, e.g., **struct{point p; int i}pstruc; point p,p1;** is not allowed as **p** occurs twice in a declaration.

The functions appearing in the CG user interface or defined for internal use may not be redefined. See *Some notes on the implementation* for a list.

The auxiliary functions **addpt, crealnk, creaptl, extptl, mulpt, pgspec, ptspec, subtpt** may appear in a preprocessed .c file. The specifications of these functions are:

```
{true}
point addpt(p1,p2)
point p1,p2
{addpt = p1 @+ p2}
```

```
{true}
link crealnk(nm)
string nm;
{crealnk = <@ nm @>}

{true}
POINT3LIST creaptl(p)
point p;
{creaptl = (p)}

{ptl = (p1,...,pn)}
POINT3LIST extptl(ptl,p)
POINT3LIST ptl; point pnt;
{ptl = (p,p1,...,pn)}

{true}
point mulpt(a,p)
float a; point p;
{p = a @* p}

{ptl = (p1,...pn)}
polygon pgspec(ptl,col)
POINT3LIST ptl; int col;
{pgspec = (@ p1,...,pn @: col @)}

{true}
point ptspec(cx,cy,cz)
float cx,cy,cz;
{ptspec = (@ cx,cy,cz @)}

{true}
point subtpt(p1,p2)
point p1,p2;
{subtpt = p1 @- p2}
```

where **POINT3LIST** defines a linked list of points.

FILES

/usr/local/bin/cg	cg command
file.cg	input file
file.c	preprocessed file
file.o	object file
temp.h	predefined CG types
/usr/include/cg/typescg.h	predefined CG types
/usr/include/cg/types.h	internal types
/usr/include/cg/typesvs.h	definition of viewing specification
/usr/local/lib/cg/CGPRE/cgprep	library of cg functions
/usr/local/lib/cg/cglib.a	library of cg functions
/usr/local/lib/cg/DEPTHS/hsr.a	hidden surface removal library

SEE ALSO

Kessener L.R.A., van Lierop M.L.P., Kalisvaart J., Peters F.J., *Language Extensions to study Data Structures for Raster Graphics*, THE Internal Report, 1986.

Lemmens W.J.M., *The CG preprocessor*, THE internal report, 1986

Unix Programmer's Manual - CC(1), Bell Telephone Laboratories, Inc., 1979

Kalisvaart J., Van de Wetering H.M.M., *Some notes on the implementation of CG*, TUE internal report, 1986.

AUTHOR

J. Kalisvaart, Eindhoven University of Technology, Department of Mathematics and Computing Science

DIAGNOSTICS

The start of the first part of the cg command is announced by a line 'cg file' for every '.cg' file. The second part (compilation and linking) is indicated by a line starting with 'cc'.

The diagnostics produced by the CG preprocessor and the C compiler are intended to be self-explanatory.

Some notes on the implementation of CG

J. Kalisvaart, H.M.M. van de Wetering

Eindhoven University of Technology

0. Introduction

The CG implementation consists of a preprocessor, user functions, auxiliary functions depending on the visible surface calculation and auxiliary functions independent of the visible surface calculation.

CG runs on a Geminix computer under unix. Special hardware needed for CG is a colour monitor and some input tool to indicate a point on the screen, e.g. a mouse.

To generate output on the colour monitor CG uses the CWI implementation of GKS: C-GKS [4]. The functions *open gks, open workstation, activate workstation, set colour representation, select normalisation transformation, set window, set fill area interior style, set fill area colour index, set aspect source flags, set polyline colour index, polyline, fill area, deactivate workstation, close workstation, close gks* are used. Notice that some overhead is introduced by using the GKS layer under CG, because only a small subset of the functionality of GKS is needed.

The compilation of a CG source program takes place in the following steps:

- cc preprocessor to expand include files and defines [3].
- CG preprocessor to convert CG language constructs into C.
- cc compiler to compile the output of the CG preprocessor.

1. The CG preprocessor

directory of sources: */usr/local/src/cg/CGPRE*

directory of objects: */usr/local/lib/cg/CGPRE*

documentation: [1]

2. User functions

functions names: **addlink, addpol, chatt, chcol, chmat, chnm, create, dellink, delpol, display, extend, getlink, getpath, remove, rename, wsclose, wsopen.**

directory of sources: */usr/local/src/cg*

directory of objects: */usr/local/lib/cg*

documentation: for users: [2], implementation: sources.

remarks:

All types which may uccur in a CG program without definition reside in the file */usr/include/cg/typescg.h*. This file contains the CG types defined in the C language; they are made available for cc by the preprocessor by inserting a line **#include "temp.h"** at the top of the preprocessed file. The **cg** command script links *temp.h* to */usr/include/cg/typescg.h*.

The **viewspec** type definition resides in the file */usr/include/cg/typesvs.h*. This is a standard CG type which can be defined in a CG program by including this file. It is separated from the other CG types because the definition has to be preprocessed as it contains the CG type **point.**

The most comprehensive user routine is *display*. It performs the following actions to display a scene and its subscenes:

- compute the view matrix
- initialise the list of polygons sorted to decreasing Z values.
- initialise the list of polygons as given in the CG scene graph.
- traverse the CG graph (polygons and zero or more links to other scenes etc.) building a list of polygons which all are candidates to be displayed.
- Compute visible surfaces and finally address the colour monitor to display the picture.

The visible surface calculation method may be altered by reconsidering the last step of display. This also affects the function *getpath*.

3. Auxiliary functions depending on the visible surface calculation:

function names:

apdispl,
body,
depthsort,
finddisplelmt,
hidesurfs,
initdispl, initlst, intersection, inslst,
oneside,
parplane, proj,
relpostn, remdispl, remlst,
splt, splitbb,
wrlist.

directory of sources: */usr/local/src/cg/DEPTHS*

directory of objects: */usr/local/lib/cg/DEPTHS*

documentation: sources.

remarks: This directory contains an implementation of *depthsort*. For the time being only convex polygons can be processed because of the limitations of the function *proj*. This functions computes whether two polygons overlap or not.

4. Auxiliary functions independent of the visible surface calculation:

function names:

addpt, apddispl, aptopl,
clippol, cmpstr, compare, crealnk, creaptl, cross, cycle,
del, dele, del_baltree, detbb, detnorm, displww, dnorm, drawcursor, dr_polyg,
extptl,
findscene, find_dispelmt,
getpixel,
initdispl, initpl, inprod, ins, ins_baltree,
Lbalance,
matdet, mlpnt3, mlpnt4, mulmat, mulpt,
norm,
parmetdet, permatdet, perstrans, pgspec, planeq, planeq2, planefit, plotln,
poinpol, poltrans, ptspec,
Rbalance, recrem, remdispl, remple, rotamat, rterr,
scalemat, setmat, shearzmat, snijpar, snijper, subtpt,
translmat, traverse,
updtlist,
vsblpar, vsblper.

directory of sources: */usr/local/src/cg*

directory of objects: */usr/local/lib/cg*

documentation: sources.

remarks:Because of the finite machine precision, plane equations, although computed in double precision by the function *planeq*, in general are not exact. To check whether a given point lies in a plane or not, the function *planefit* computes a formula that theoretically should be zero. It is accepted if the value is less then the constant EPS. EPS is taken about 10 x the machine precision.

Planeq computes plane equations according to the Martin Newell method[5]. This method does not work for self intersecting polygons with equal size of "positive" and "negative" areas of the interior. Because of the limitations of the function *proj*, these cases may never occur. The function *planeq* however, is able to process every polygon. If necessary, the plane equation is computed by the function *planeq2* by considering only n-1 vertices, where the original polygon is assumed to contain n vertices.

References

[1] **Lemmens, W.J.M., The CG preprocessor,** THE Internal Report

[2] **Kessener, L.R.A., Van Lierop, M.L.P., Kalisvaart, J., Peters, F.J., Language Extensions to study Data Structures for Raster Graphics,** THE Internal Report

[3] **CC(1), Unix Programmers Manual,** Volume 1

[4] **C-GKS, the C implementation of GKS, the Graphical Kernel System.** User Guide. CWI Amsterdam 1986.

[5] **Newman, W.M., Sproull, R.F., Principles of Interactive Computer Graphics,** p 499, McGraw Hill, 1981.

COMPUTING SCIENCE NOTES

In this series appeared :

No.	Author(s)	Title
85/01	R.H. Mak	The formal specification and derivation of CMOS-circuits
85/02	W.M.C.J. van Overveld	On arithmetic operations with M-out-of-N-codes
85/03	W.J.M. Lemmens	Use of a computer for evaluation of flow films
85/04	T. Verhoeff H.M.J.L. Schols	Delay insensitive directed trace structures satisfy the foam rubber wrapper postulate
86/01	R. Koymans	Specifying message passing and real-time systems
86/02	G.A. Bussing K.M. van Hee M. Voorhoeve	ELISA, A language for formal specifications of information systems
86/03	Rob Hoogerwoord	Some reflections on the implementation of trace structures
86/04	G.J. Houben J. Paredaens K.M. van Hee	The partition of an information system in several parallel systems
86/05	Jan L.G. Dietz Kees M. van Hee	A framework for the conceptual modeling of discrete dynamic systems
86/06	Tom Verhoeff	Nondeterminism and divergence created by concealment in CSP
86/07	R. Gerth L. Shira	On proving communication closedness of distributed layers

86/08	R. Koymans R.K. Shyamasundar W.P. de Roever R. Gerth S. Arun Kumar	Compositional semantics for real-time distributed computing (Inf.&Control 1987)
86/09	C. Huizing R. Gerth W.P. de Roever	Full abstraction of a real-time denotational semantics for an OCCAM-like language
86/10	J. Hooman	A compositional proof theory for real-time distributed message passing
86/11	W.P. de Roever	Questions to Robin Milner - A responder's commentary (IFIP86)
86/12	A. Boucher R. Gerth	A timed failures model for extended communicating processes
86/13	R. Gerth W.P. de Roever	Proving monitors revisited: a first step towards verifying object oriented systems (Fund. Informatica IX-4)
86/14	R. Koymans	Specifying passing systems requires extending temporal logic
87/01	R. Gerth	On the existence of sound and complete axiomatizations of the monitor concept
87/02	Simon J. Klaver Chris F.M. Verberne	Federatieve Databases
87/03	G.J. Houben J.Paredaens	A formal approach to distributed information systems
87/04	T.Verhoeff	Delay-insensitive codes - An overview

87/05	R.Kuiper	Enforcing non-determinism via linear time temporal logic specification.
87/06	R.Koymans	Temporele logica specificatie van message passing en real-time systemen (in Dutch).
87/07	R.Koymans	Specifying message passing and real-time systems with real-time temporal logic.
87/08	H.M.J.L. Schols	The maximum number of states after projection.
87/09	J. Kalisvaart L.R.A. Kessener W.J.M. Lemmens M.L.P. van Lierop F.J. Peters H.M.M. van de Wetering	Language extensions to study structures for raster graphics.