Spring 1997

Appendices to the Quadtree Assignment

CMSC 420
David Traum

Appendix A Drawing quadtrees

A limited set of drawing primitives has been defined for drawing quadtrees. When the drawing procedures are invoked in your programs, they will produce ASCII output, that can then be used as input to a display or a printing program, or simply inspected for debugging purposes. Your programs can also produce other text, which will be printed along with the quadtree drawings if you feed the output to the printing program (but will not be shown with the display program).

The prototypes of Pascal drawing routines are provided in the file `drawing.h`, which is to be `#include`'d in your programs. The routines are implemented in `drawing.p` and are documented below.

The prototypes for C drawing routines are provided in the file `drawing_c.h`, which is to be `#include`'d in your programs. The routines are implemented in `drawing.c` and are documented below.

The source for the printing and display programs is in the files `printquad.c` and `showquad.c`. The display program, `showquad`, reads output files generated by your programs and displays the quadtrees, one by one, in an X window. To use it, you need to be working at a workstation that supports the X window system, such as SPARCstation or NeXTstation (using co-Xist). The printing program, `printquad`, translates the output from your programs into PostScript code, that can then be fed to a PostScript compatible printer. As an example of the use of `printquad`, you could do

```
myquadprog | printquad | qpr -q pg2-ps
```

to invoke your program, `myquadprog`, filter the output through `printquad`, and print it with `qpr` on the LaserWriter printer in the Parking Garage 2 WAM Lab.

The display and printing programs are written in C, and you are free to modify them and the drawing routines if you want to get other effects than those that are provided. To build the programs, execute `make` from the command shell.

There is no constraint put on the text that you generate along with the quadtree drawings, except that the character sequence `$$$$` must be avoided. That sequence marks the beginning of a block of drawing commands.

A.1 Files

The files listed in this section are available in the directory `~dt01/util`.

`drawing.p`	Pascal routines to be included in your program
`drawing.h`	Prototypes of Pascal routines
`drawing.c`	C routines to be included in your program
`drawing_c.h`	Prototypes of C routines
`test.pm1`	example files to test `showquad` and `printquad`
`Makefile`	build rules for `showquad` and `printquad`
`printquad.c`	code for `printquad`
`showquad.c`	code for `showquad`

A.2 Drawing Routines

To use these routines, the file `drawing.h` or `drawing_c.h` needs to be included into your source file. To do so, incorporate the following line in your program before their use:

`#include "drawing.h"` for Pascal

or

`#include "drawing_c.h"` for C/C++

`procedure StartPicture(lx, ly: real);`
Starts a new picture. This routine must be called before any other drawing routines are called. The size of the picture is assumed to be `lx * ly`, with the origin (0,0) in the lower left corner. You can draw outside of the boundary of the picture area, but it may not be displayed or printed.

`procedure EndPicture;`
Marks the end of the current picture. After this call, no drawing routine other than StartPicture may be called.

`procedure SetLineDash(black, white: integer);`
This routine is used to control whether subsequently drawn lines and rectangles are drawn dashed (i.e., broken) or not. If a line is dashed, there are pieces of the line that are drawn, and pieces that are not drawn. The length of the drawn pieces is determined by `black`, and the length of the pieces that are not drawn is determined by `white`. If `white` is 0, the line is not dashed. The unit of the parameters depends on the display program (same as line width, unless otherwise defined).

`procedure DrawLine(x1, y1, x2, y2: real);`
Draws a line with end points (`x1, y1`) and (`x2, y2`). The current line dash is used.

`procedure DrawRect(x1, y1, x2, y2: real);`
Draws a rectangle with top left corner at (`x1, y1`) and bottom right corner at (`x2, y2`). The current line dash is used.

`procedure DrawDot(x, y: real; r: integer);`
A dot centered at (`x, y`) with radius `r` is drawn. The unit of the radius is the line width, unless the output program defines a dot unit.

`procedure DrawChar(c: char; x, y: real);`
Draws character `c`, with the left side and base line of coordinate `x` and `y`, respectively. Do not make assumptions about the width of characters.

`procedure DrawName(n: nameptr; x, y: real);`
Draws a name, with the left side and base line of the leftmost character of coordinate `x` and `y`, respectively. Do not make assumptions about the width of characters. The type `nameptr` is assumed to be a pointer to a record with fields `LEN` and `STR`, as in the data definition that you will be given. If you store your names differently, you need to change this routine.

A.3 printquad

`printquad` generates PostScript code for printing the output from your programs. You can adjust a number of printing parameters, but some are fixed. Among the ones that are fixed are the margins and the font used. The adjustable printing parameters are provided on the command line as the program is invoked. This is how it should be invoked:

```
printquad [-d n][-f n][-h s][-l n][-s n] [file...  | < file]
```

where `n` is a real number and `s` is a string. Note that you can either provide a list of files to print or let `printquad` read from standard input. The options and their defaults are as follows:

`-d` [.5]	dash width and dot unit
`-f` [.5]	size of the picture font relative to text font
`-h` [date]	header
`-l` [.1]	line width
`-s` [2]	size of pixels (width and height)

The unit used for -d, -l and -s is 1/72", but the unit used for -f is the size of the text font (i.e., relative to Courier 10).

A.4 showquad

showquad interprets the output from your programs, and displays the quadtrees in a window on an X workstation. To let the program continue (i.e., to display the next quadtree, or to quit if there is none), click the mouse somewhere in the window. Some display parameters may be modified. showquad is run like this:

```
showquad [-f n][-l n][-s n] [file...  | < file]
```

where n is an integer. Note that you can either provide a list of files to display from or let showquad read from standard input.

The options and their defaults are as follows:

-f [8]	size of the picture font
-l [1]	line width
-s [4]	size of pixels (width and height)

The unit used for -l and -s is a screen pixel. For best results, you should choose an even number for -s. Legal values for -f should include 8, 10, 12, 14, 18 and 24 (the font used is Courier).

If you are not experienced in using the X window system, the following step by step description should get you going in using showquad:

1. Log onto the workstation.
2. If you are on a NeXT workstation, fire up co-Xist by double clicking on the X icon in the icon menu on the right side of the screen.
3. Open up an xterm window, and make sure that you have a window manager running (if you don't know what I'm talking about, then just execute twm & at the command line).
4. Make sure that you give access privileges for the remote host that you will be working on by using xhost. It's simplest just to execute xhost +, but if you are paranoid, you may put the host name after the plus, e.g., xhost +linus.umd.edu.
5. Execute the command hostname to get the name of the workstation and make a note of it.
6. Log onto the remote host (i.e., a machine where your class accounts are), e.g., rlogin linus.
7. When you have logged on, you must set up the DISPLAY variable (unless it is automatically set by your .login file), using the workstation name that you obtained in step 5. If you are using the C-shell (most likely), the command will look like: setenv DISPLAY <workstationname>:0, where <workstationname> is the name above. If you are using the Bourne or the Korn shell, the command sequence is DISPLAY=...; export DISPLAY
8. Now, you are ready to run showquad.

Appendix B Various System Programming Issues

B.1 The Make Facility

To automate the building of your programs, it is convenient to use make. You should include a make file with the name makefile in your electronic submissions. A makefile is a series of dependencies and actions. The syntax for a dependency and associated actions is

```
<filename>:  <filenamelist>
  <command1>
```

```
  ...
  <commandn>
```

The dependency states that the target file <filename> needs the files in the <filenamelist>, and that the commands <command1> through <commandn> should be executed to build <filename>. When make reads a makefile, it tries to build the target file. The initial target file may be listed on the command line when make is invoked (e.g., make printquad), or if no target is specified, the initial target is the filename of the first dependency. For any dependency with the target file in the head of the dependency, the make program checks recursively whether the files in the <filenamelist> need to be built. The associated commands are executed next, in case the target file doesn't exist, or has a less recent modification date than one of the files in the <filenamelist>.

This description may sound confusing, so let's look at an example. Let's say that your program resides in the file myquad.p and that you include drawing.i in your program. The rule for that will be:

```
myquad:  myquad.p drawing.i
  pc -o myquad myquad.p
```

With this rule, if myquad doesn't exist, the command will always be executed. Also, if myquad.p or drawing.i have been changed from the time myquad was last built, the command is executed. A more complex makefile is shown in the next section.

Note, that you must use the tab character in front of the actions. Also, there must be an empty line between each rule. That means you cannot have an empty line in the command list.

B.2 Separate Compilation

With the BSD UNIX Pascal compiler, it is possible to divide programs into modules that can then be compiled separately. Each module sits in its own .p file, but only one of them may contain a program statement. To be able to refer to routines defined in other modules than the current one, header files must be used. Header files contain declarations for routines that are can be called from modules other than the module they are defined in. Typically, there is one header file for each module. The routine declarations are analogous to forward declarations, but instead of the keyword forward being put at the end, the keyword external is used instead. As with forward declarations, the routine definition itself may not contain a parameter list or a function result. An example external declaration is in subsection B.2.3 below.

As an example, let's say that you break your program into two modules, prog.p and quad.p. prog.p contains the command decoder and other stuff, while quad.p contains the quadtree routines. Since we know that the decoder must call the quadtree routines, we let prog.p contain the program statement. It thus contains the begin ... end. statement that is executed when the program is started. quad.p, on the other hand, doesn't contain a program statement. Below we show partially how prog.p, quad.p, quad.h and a makefile to build the program might look like:

B.2.1 prog.p

```
program prog(input, output);

#include "quad.h"

...

begin
    ...
    display(qt);
    ...
end.
```

B.2.2 quad.p

```
#include "quad.h"

...
procedure display(qt: node);

    begin
        ...
    end;
...
```

Note, that the parameter list for `display` is commented out with braces. As `display` is declared in the header file `quad.h`, the parameter list cannot be repeated here.

B.2.3 quad.h

```
...
procedure display(qt:node); external;
...
```

B.2.4 makefile

```
all:    prog

prog.o: prog.p quad.h
    pc -c prog.p

quad.o: quad.p quad.h
    pc -c prog.p

prog:   prog.o quad.o
    pc -o prog prog.o quad.o
```

B.3 Electronic Submissions

A program will be set up for you to electronically turn in projects. To use it, add the following line to your `.cshrc` file (assuming your login shell is csh):

```
alias submit ~dt02/Bin/submit
```

The program takes two arguments, and is invoked like this:

```
submit <assignmentno> <filename>
```

Your submission is taken from the file `<filename>` and is timestamped. If you submit for the same assignment number more than once, only the last submission will be retained.

As you can see above, you can submit only a single file with the program. To submit a group of files, you need to group them together. A convenient way is to use `tar`. As an example, let's say we want to combine files `first` and `second` into `file.tar`. That can be done by issuing the command

```
tar -cf file.tar first second
```

Now, you can submit `first` and `second` by submitting `file.tar`, e.g.

```
submit 3 file.tar
```