Software Project Fall 2004

Gnome Graphics

Overview

The objective of this project is to write an interpreter for simple graphical programs. The programs control the behavior of a small creature – the gnome – that moves on a white canvas and holds a paintbrush in his hands. When the gnome walks and holds the paintbrush down, black spots are drawn on the canvas, and a picture is obtained. Your task is to write an interpreter that reads (=parses) a program for the gnome, runs it and outputs the resulting image.

The canvas on which the gnome walks is made of *n*-by-*m* pixels and is white in the beginning of execution (the size of the canvas is defined by the program). The width of the gnome's paintbrush is 1 pixel. At each moment, the gnome has a certain position on the canvas and a heading direction. These can be altered and queried by the program.

Programs for the gnome are written in a language that is described in Section 1.1. The programs initialize some parameters, such as the dimensions of the canvas, initial position of the gnome, his initial heading direction, etc., and give the gnome commands what to do (move forward *x* steps, turn *x* degrees, ...). Your project is composed of two major parts (the two parts might be of different programming complexity): one part understands a gnome program and the other executes it and outputs the image.

1 Gnome programming language

1.1 Language Overview

The gnome programming language is comprised of built-in commands that tell the gnome what to do. Moreover, the gnome programmer can use variables, routines and loops. A program has access to *built-in* variables that describe the current state of the gnome, such as his current position (a full list is given below).

A gnome program has the following properties:

- Every line holds at most one command (or it may be empty, or contain a comment).
- Tabs and spaces are ignored. Many spaces and tabs are the equivalent to one space.
- Any character after a '%' is ignored till the end of the line, i.e. the '%' is used for remarks, similar to // in C++.
- The language is case insensitive, this means that upper and lower cases are ignored. The following strings are equivalent in the language: "Hello", "hello", "HELLO" and "HelLo".

Programmers can define variables. A variable holds real values (i.e. float or double). Variables are local to their scope (meaning that a variable defined in a routine is only recognized within that routine). Global variables are not allowed. A variable is initialized when it is first used (i.e., no special declaration is needed, unlike in C language, for example). In addition, there are several built-in variables, for example "$PositionX" holds the current *x* coordinate of the gnome. Built-in variables are read-only, they cannot be changed by the gnome programmer directly.

Routine names are of the following format: [A–Z]([A–Z]—[0–9])* like Compute
Variable names are of the following format: $[A–Z]([A–Z]—[0–9])* like $TurnRadius

It is not allowed to name routines or variables by the names of built-in commands and built-in variables (these names are reserved). Routines do not return a value. They can have any number of parameters (zero or more). Parameters are passed by value, like in C. The parameters are local variables of the routine. The syntax of defining a routine is as follows:

```
ROUTINE RoutineName($param_name1, $param_name2, ..., $param_nameN)
{
  % Here comes the code of the routine
  ....
  ....
}
```

One routine can call another routine using the following syntax: "RoutineName(param1, param2, ...)". The parameters cannot be expressions, they can only be either constants (17, 5.5, ...) or variables.
Here is a sample routine:

```
%
% Draw a line of length $Length
%
ROUTINE DrawSegment($Length)
{
    Forward($Length)
}
```

Routines can be defined anywhere in the body of the program, but not inside other routines. It is allowed to call a routine that is defined later in the text of the program. Recursion is allowed.

Every program begins execution with a special routine called "Main", with zero parameters. Every program must have a Main routine. The first line of Main (excluding remark lines) defines the size of the canvas by calling built-in command CanvasSize. If this call is not made, default canvas size is used, as defined in the project header file. Execution stops after the last line of Main.

1.2 Sample program

```
%
% This program draws a square
%
ROUTINE main()
{
    CanvasSize(100, 100)
    $len = 10
    DrawSegment($len)
    TurnRight(90)
    DrawSegment($len)
    TurnRight(90)
    DrawSegment($len)
    TurnRight(90)
    DrawSegment($len)
}

ROUTINE DrawSegment($Length)
{
    Forward($Length)
}
```

1.3 Built-In Variables

Variable	Description
$PositionX	*x* coordinate of the gnome's current position
$PositionY	*y* coordinate of the gnome's current position
$CanvasWidth	The width of the canvas in pixels
$CanvasHeight	The height of the canvas in pixels
$Heading	Gnome's current heading direction in degrees. $Heading is always between 0 and 360 and measures deviation from up direction in clockwise manner. This means: when $Heading equals zero, the gnome is headed up (north); when $Heading is 90, the gnome is headed right (east), etc.
$IsBrushDown	Equals 1 if the paintbrush is down and 0 otherwise.
$false	constant 0
$true	constant 1

1.4 Built-in commands

Command	Description
CanvasSize(width, height)	Initializes the dimensions of the canvas. width and height are assumed to be positive integers, so the values passed to CanvasSize should be truncated. width and height should be constants (not variables). The call to this command can only appear as the first line of Main routine. If it appears someplace else, you generate a compilation error. If there's no call to CanvasSize, default size is used.
Forward(length)	Make the gnome go length distance forward in its current heading.
Back(length)	Make the gnome go length distance backwards with respect to his current heading.
TurnRight(angle)	Turn the heading of the gnome clockwise angle degrees. The gnome doesn't move, only changes heading!
TurnLeft(angle)	Turn the heading of the gnome counter-clockwise angle degrees. The gnome doesn't move, only changes heading!
BrushUp()	Make the gnome pick up his paintbrush, so that he won't draw anything even if he moves.
BrushDown()	Make the gnome put his paintbrush on the canvas, so that he will draw something when he moves.
ClearCanvas()	Makes the canvas all white.
SetPosition(x, y)	Place the gnome at coordinates (x,y) on the canvas. This command doesn't produce any drawing on the canvas.
SetHeading(angle)	Sets $Heading to be angle degrees.
Fill(x, y, color)	Fills an area starting from x,y with color black(1) or white(0).

if (<value>)
else
endif
	Control the flow of a program. <value> is either a constant or a value of a variable. A 0 value means false, any other value is true. The *else* is optional.

while (<value>) endwhile	Continue to execute the statements between the while and endwhile as long as value != 0.
<variable>=<exp> i.e. $<name>=<exp>	Assign a value to a variable. For example: $x=$y+5

NOTES:

1. Any parameter to the above functions can be a constant value or a value of a variable. Expressions are not supported. This means that the following are legal:
   TurnRight(30)
   TurnLeft($Heading)
   But TurnRight($angle+34) is not legal.
2. All angles are in degrees in the range of $[0,360)$. If the programmer's input included a number outside that range, the number should be normalized to the $[0,360)$ range.

1.5 Expressions

Numbers in the language are floating point of the form:
$[-]N[.M]$, that is: an optional '-' sign, followed by an integer, followed by an optional .integer. All of the following are legal: 1, -2, 3.1, -543.123
Formally, expressions are of the form:

exp : Value | BinOp | Func | Func2	
Value : <number> | <variable>	number, as defined above. <variable> is the value of the variable.
BinOp : <Value> <op> <Value>	<op> is any of the legal operators, i.e. +, -, *, etc.
Func : <Function>(<Value>)	function is any of the functions defined bellow, such as sin, cos, etc.
Func2 : <Function>(<Value>, <Value>)	A function of two variables, like random and atan2

The legal operators are: +, -, *, /, <, >, <=, >=, ==, !=
Supported functions: sin, cos, tan, asin, acos, atan, atan2, sqrt, round and random. In case of a parameter illegal value program exits.
random(low, high) returns a floating point random number in the range [low, high].
atan2 is like the function with the same name in the standard library.
For example the following are legal expressions:

```
3.45
$v2
$v1 + $v2
$angle*4.5
sqrt(3)
sqrt(-3)
```

But these are not legal expressions:

```
.76
1+2+3
```

1.6 Language syntax

Formally, the language syntax is as follows:

Program $\longrightarrow$	Routine { commands }	
commands $\longrightarrow$	oneline commands	
oneline $\longrightarrow$	singlecommand NEWLINE |	NEWLINE is a mark of end of line from the lexical analyzer (explained in class)
	NEWLINE	comments are ignored by the lexical analyzer
singlecommand $\longrightarrow$	$VarName = exp |	as defined in Section 1.5
	BuiltInCommand(parameters) |	
	RoutineName(parameters)|	Function call to RoutineName
	...	

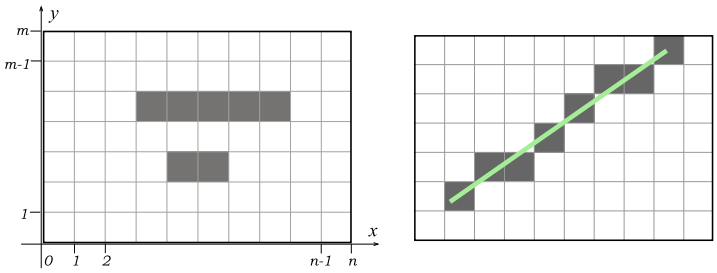
Routine names are of the following format: [A–Z]([A–Z]—[0–9])* like Compute22
Variable names are of the following format: $[A–Z]([A–Z]—[0–9])* like $TurningAngle

2 Canvas and graphics

Here is an explanation about the execution part of the project - how the drawing is performed.

2.1 Coordinate system

The canvas on which the gnome walks and draws is a rectangle of some size, let's say width=n and height=m. The origin $(0,0)$ of the coordinate system on the canvas is in the lower left corner, so that only non-negative coordinates' values are considered (see Figure 1(a)). The output image should be n-by-m pixels in size. Therefore, the valid range of coordinates is $[0,n)$ for x (horizontal) and $[0,m)$ for y (vertical). Note that the position of the gnome is **not** restricted to integer coordinates only, so for example $(5.5,17.06)$ could be a perfectly valid position. However, the output image is a discreet bitmap (consisting of discreet pixels), so you need to *rasterize* what the gnome draws.



(a) (b)

Figure 1: (a) The coordinate system on the canvas. (b) Rasterization of a line. See details on the Midpoint algorithm in the appendix.

2.2 Rasterization

Rasterization means coloring the appropriate pixels so that the resulting image resembles the lines that the gnome drew (see Figure 1(b)). If the gnome draws a horizontal or vertical line, it is easy - we just "turn on" some part of a row/column of the canvas pixels. When rasterizing lines with general slopes, we need a special algorithm that tells us which pixels to turn on. In this project you will use the Midpoint algorithm from basic Computer Graphics, described in the appendix.

2.3 Filling

Filling, means coloring a closed area with a specific color. The colors that can be used are black (1) and white (0). The filling starts from an initial point (x,y) on the canvas and advances until it hits the boundary of the area. If the area is not closed the filling should fail. In this project the FloodFill algorithm described in the appendix is used to perform area color filling.

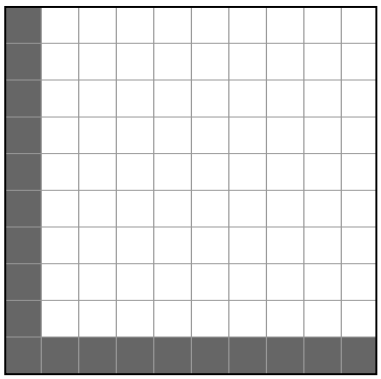
2.4 Canvas data structure

You should use a bitmap data structure to represent the canvas. Since the canvas is black-and-white, you need one bit per pixel to represent the drawn image. The bitmap structure is defined in the project header file and it looks like this:

```
typedef struct {
    int width;
    int height;
    unsigned char *bits; /* pointer to the bits array*/
} GnomeBitmap;
```

```
unsigned char bottom_left_corner[] =
        {
        0xff, 0xa0,  /* 1111111111000000 */
        0x80, 0x0,   /* 1000000000000000 */
        0x80, 0x0,   /* 1000000000000000 */
        0x80, 0x0,   /* 1000000000000000 */
        0x80, 0x0,   /* 1000000000000000 */
        0x80, 0x0,   /* 1000000000000000 */
        0x80, 0x0,   /* 1000000000000000 */
        0x80, 0x0,   /* 1000000000000000 */
        0x80, 0x0,   /* 1000000000000000 */
        0x80, 0x0    /* 1000000000000000 */
       };

GnomeBitmap canvas;
canvas.width = 10;
canvas.height = 10;
canvas.bits = bottom_left_corner;
```



bits is a pointer to an array of bits representing the image. The bits of the canvas bitmap are stored by rows, starting from the bottom row and up. For example, the first bit of the array (the most significant bit of bits[0]) represents the bottom left corner pixel of the canvas; the *n*th bit represents the bottom right corner. The rows are scanned from left to right. Each row is stored in integer amount of bytes (unsigned chars). If the number of pixels per row (the width of the image) is not some multiplication of 8, then the last bits are ignored. For example, if the number of pixels per row is 5, then each row will occupy a full byte and the last 3 bits will be ignored and must be set to zero.

Here is and example of a 10-by-10 bitmap: the last six bits in each row are ignored.

The GnomeBitmap structure is passed to the graphics library GnomeGR that will display it on screen in single step mode (explained in Section 3).

2.5 Initialization

The default parameters for the gnome's initial position and heading, as well as the default canvas size, are defined in the project header file available on the webpage.

2.6 Rules of restricted walking

The gnome is not allowed to step out of the canvas. It is up to the gnome's programmer to make sure he doesn't. Whenever the gnome is attempting to walk outside of the canvas (i.e. outside of the legal range of coordinates), the execution stops, prints an appropriate error message, containing the line number of the program that caused the error, and the previous position of the gnome (his position before the execution of the last command). However, your program should output the resulting image created so far.

3 Input and output

- The output of each program execution is an image file that represents what the gnome drew, unless the gnome program contained syntax errors.
- You can open a graphical window using the GnomeGR library, can be found on the web–page.
- The command line for the gnome program is:
  `Gnome04 [-s] <mygnome.prg> [-b] <output_image.pgm>`

  Gnome program files end with the extension '.prg'. The output image is written to specified filename with extension '.pgm'. The image should be in PGM format (see description in the appendix).
  - -s (optional) Single step mode (for debugging). Print each command and display the current canvas on screen (using GnomeGR library). Then wait for the user to press <enter>, and go to the next command, and so on. If -s is not specified, no graphical window is opened.
  - -b The output PGM file is in binary format. If -b is not specified, the output is in ASCII format.

  The order of the arguments should be strictly as written above.
  For example: "Gnome04 square.prg -b square.pgm"

Run-time output

- In case of an error (during parsing of the gnome program) output the line number where the error occurred and description of the error and exit the program. Use the following to do so:

  `printf("Parsing error in line line:%d, Description:%s\n", line_number, reason);`
- In *Single Step* mode, print the line number in the source and the line from the source that is being executed. Also print the current parameters of the gnome: his (x,y) position, his current heading and whether the brush is up or down.
- If an illegal operation occurs, such as division by zero or non-positive value passed to CanvasSize, print what happened and exit.
- If illegal arguments are passed to the program (like prg file that doesn't exist), print what's wrong and exit.

4 Requirements

The project grade is made of these parts:

1. Initial design document. 5% (not checked)
2. Working makefile 5%
3. Working gnome project. 50%
4. Documentation (described bellow) 30%
5. Three (3) non-trivial gnome programs 10%

To pass the course, the project must work.

5 Programming Guidelines

The coding quality is an important part of this project. Following are some general guidelines for correct coding. Try to follow these guidelines, as well as employing any additional knowledge you have in writing software.

- Modularity. Related functions should reside in the same file, while unrelated functions are to reside in different files.
- Functionality. Each function should perform one *thing*. This *thing* should be the function name. Functions should be short and clear, function length should not exceed one page. Any repeated code parts should be concentrated in functions.
- The maximum line length in your code should not exceed 80 characters.
- Documentation. The code should be clear. Therefore, short comments should appear in various places, such as preceding functions or complicated blocks, when defining variables, etc. If there are assumptions regarding the input / output of the functions, they should appear too.
- Naming conventions. A significant part of the documentation resides in the code itself. Proper name choosing is one way of doing it. The names of the variables and functions should imply their role. Each group of names should have the same convention. For example, function names can have capital letters leading each word in the name, variables can use lowercase characters with underscores between words, and constants can use all capital letters: GetNextLine (line_buffer, LINE_LEN);
- Simple data-structures, algorithms and implementations are preferred.
- It is recommended to handle data structures only via dedicated functions. For example, if there is a linked list used in some places, and some algorithms go serially over the list, it is a good idea to have functions such as GetFirst and GetNext. The lists' elements should be accessed only via these functions.
- No compilation errors or warnings are permitted.
- Obviously, no run time errors are acceptable.
- A working project is more important than a fancy project. Make sure the basic features that are required here are working before adding anything to the project. Also, a not so efficient working project is much better than a very efficient non-working project.

6 Documentation

In addition to the code, you should also write a short description of the project. The documentation file should describe the entire project. It should be clear and built of nicely separated sections. Its length should be about 10 two-sided pages.

The idea is that when someone else (beside the author of the project) reads the documentation, she will be able to understand, modify and extend your project.

The documentation should include:

- Header - including the same information as in the partners (see directory structure section bellow) file.
- A list of all the files in the project, and a short (one line) description of each one of them.
- The modules of the project and their relationships.
- A general description of the project logic and the application flow.
- A list of all the data structures in use, together with a short explanation why they were chosen. Include a description of the alternatives to the decision, and list the pros and cons of each alternative.
- For each data structure, a short time complexity analysis for each of the implemented actions.
- Comments and suggestions about the project.
- The documentation should be printed in English.

The documentation is not part of the code.

7 Administrative Information and Rules

- You should write the project in "C", or if you wish, "C++". The project should run on UNIX machines in class Schreiber-04.
- Due: Thursday 22-January-2004, 10am
- You **must** work in groups of **two** people.
- On the second class (one week after the semester begins), submit an initial design document of the project (1-2 pages). This document should include the participants' names, main modules and relations between them, and data structures that will be used.
- **You should check the web page at least once a week for any updates on the project. The webpage is:**

  `http://www.cs.tau.ac.il/~sorkine/courses/proj04/`

  Some of the definitions of the project may change. Pay special attention to the forum. The questions and answers that are posted in the forum complement this document.
- After you submit your project, check your email regularly (at least once a week), you should expect a message with your run-time grade.

7.1 Directory Structure

Following all the guidelines is absolutely critical. Wrong directory structure will cause the run-time test to fail and will result in zero runt-time grade.

- There should be one directory under your account, named "GnomeProject04" (case is important!) with no sub-directories.
- The name of the executable is `Gnome04`.
- The directory should contain a file named "partners" that includes:
  One line for each team member with his/her login name, ID number, and full name separated with white spaces.
  The files of the project should reside in the directory of the team member in the first line. Other member should have the same directory that includes only the partners file.
- All of the files (source, header, makefile) should be under that same directory. There should be no object files.
- There should be a working makefile named "Makefile". The makefile should generate the executable of the project.
  In addition, there should be a "clean" target in the makefile that will remove the object files from the directory.
- You must set permission for the GnomeProject04 directory and all the files under this directory. In the command prompt, type:

```
> cd
> chmod a+x ~
> chmod 755 GnomeProject04
> chmod 644 GnomeProject04/*
```

7.2 Before you submit, make sure

- There is no debug output to the project, unless it is for debug purposes, disabled by appropriate #if clauses.
- Your project works on the department's UNIX systems, i.e. UNIX machines in the lab.
- Your documentation includes the general description, as explained in the documentation section.
- Your code is well commented, as explained in the Programming Guidelines section.
- The project does not depend on any specific information in your account. A good idea is to copy the GnomeProject04 directory temporarily to another account and compile it there.

7.3 What to submit

- A hardcopy of the documentation.
- **No hardcopy of the source files please!** The checker will looks at your code online.
- A hardcopy of the partners file.
- The electronic version of your work should reside in the proper location, untouched after the due date.

8 Grading Criteria

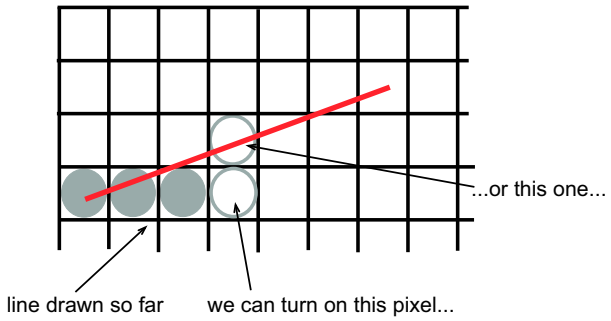
- Project submitted on time. Late submission rules are on the website. But it is highly recommended to submit on time!
- Fully functional according to the specifications.
- The project compiles flawlessly.
- Code quality.
- Documentation.
- Reasonable efficiency, make a tradeoff between implementation time and efficiency, but know that you have. When you do, write your considerations in the documentation.
- Initial design submitted on time.
- It is OK to share ideas with other groups, but it is forbidden to share code.

Some remarks on getting a good grade:

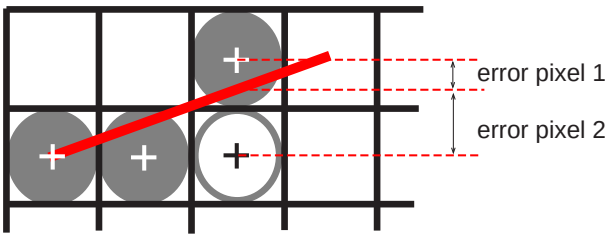
- Start working on the first day. The end of the semester is more stressed than the beginning.
- If something is not clear, ask.
- Do not leave things to the last minute. Make sure you have a running project ahead of time and test it.
- The documentation is an important part of the project. Write it when you finished coding and not the night before you submit the project.
- Work with a team member you trust, check that both of you are on the right track every once in a while.

A Rasterizing a line - Bresenham's Midpoint Algorithm

Midpoint algorithm is an efficient way of rasterizing a line on a discrete pixel grid. Suppose we want to draw a line that starts at (x_1, y_1) and ends at (x_2, y_2). For simplicity, let's assume for a moment that $x_1 < x_2$ and $y_1 < y_2$ and that the slope of the line is less than 45 degrees. We rasterize the line from left to right. At each step, we move right by 1 unit in horizontal direction. We need to decide whether to move also one unit upwards or not:



The decision is made by calculating the error that would be caused by each of the choices, i.e. by looking at the vertical distance between the candidate pixels' centers and the real line. The pixel with the smaller error is chosen. Note that the sum of the two errors is exactly one, so the pixel whose error is less than 0.5 will be chosen.



Here is a pseudocode for the Midpoint algorithm:

Assumptions:
- The points (x_1, y_1) and (x_2, y_2) are assumed not equal
- $\Delta x = x_2 - x1 > 0$ and $\Delta y = y_2 - y_1 > 0$. If this is not the case, the coordinate that has negative delta should be decreased by one instead of increased (for instance, if $\Delta y < 0$, replace "$j+=1$" by "$j-=1$" in the pseudocode).
- $\Delta x > \Delta y$. If not, replace the roles of x and y axes.

begin
$\Delta x = x_2 - x_1$
$\Delta y = y_2 - y_1$
$m = |\frac{\Delta y}{\Delta x}|$
$i_1 = \lfloor x_1 \rfloor$
$j = \lfloor y_1 \rfloor$
$i_2 = \lfloor x_2 \rfloor$
$\varepsilon = -(1 - (y_1 - j) - m(1 - (x_1 - i_1)))$

for $i = i_1$ to i_2
    TurnOnPixel(i, j)
    if $(\varepsilon \geq 0)$
        $j+=1$
        $\varepsilon-=1.0$
    end if
    $i+=1$
    $\varepsilon+=m$
end for
end

B PGM image file format

PGM stands for **p**ortable **g**ray**m**ap file format. The definition of this image format in ASCII form is as follows:

- A "magic number" for identifying the file type. A pgm file's magic number is the string "P2" for ASCII form.
- Whitespace (blanks, TABs, CRs, LFs).
- A width, formatted as ASCII characters in decimal.
- Whitespace.
- A height, again in ASCII decimal.
- Whitespace.
- The maximum gray value, again in ASCII decimal.
- Whitespace.
- Width * height gray values, each in ASCII decimal, between 0 and the specified maximum value, separated by whitespace, starting at the top-left corner of the graymap, proceeding in normal English reading order. A value of 0 means black, and the maximum value means white.
- Characters from a "#" to the next end-of-line are ignored (comments).
- No line should be longer than 70 characters.

Here is an example of a small graymap in this format:

```
P2
# feep.pgm
24 7
15
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 3 3 3 3 0 0 7 7 7 7 0 0 11 11 11 11 0 0 15 15 15 15 0
0 3 0 0 0 0 0 7 0 0 0 0 0 11 0 0 0 0 0 15 0 0 15 0
0 3 3 3 0 0 0 7 7 7 0 0 0 11 11 11 0 0 0 15 15 15 15
0 0 3 0 0 0 0 0 7 0 0 0 0 0 11 0 0 0 0 0 15 0 0 0
0 0 3 0 0 0 0 0 7 7 7 7 0 0 11 11 11 11 0 0 15 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

The binary form of PGM is similar to ASCII form. Here are the differences:

- The"magic number" is "P5" instead of "P2".
- The gray values are stored as plain bytes, instead of ASCII decimal.
- No whitespace is allowed in the grays section, and only a single character of whitespace (typically a newline) is allowed after the maxval.
- The files are smaller and many times faster to read and write.

Note that this raw format can only be used for maxvals less than or equal to 255.

Examples of PGM files in both ASCII and binary form can be found on the project webpage. Note that since you will write only black-and-white images, your maxval should always be 255, write 0 for back and 255 for white. The PGM files can be opened using xv viewer under UNIX/Linux machines and using Irfanview under Windows (see link on the webpage to obtain Irfanview).

C FloodFill algorithm

The purpose of FloodFill algorithm is to fill an arbitrary bounded space on the screen with a given color (we will be using black or white). You are to implement a FloodFill algorithm for the Fill command. The input to the algorithm is the starting pixel (x,y) and the new color. The algorithm performs by advancing from the initial pixel to adjacent pixels and changing their color to the new color. The algorithm stops when no new pixels are met or if it reaches the canvas boundary. If the boundary is reached, the algorithm should roll back and cancel the filling attempt. Here is a recursive pseudocode of the algorithm:

```
FloodFill (int x, int y, int newColor)
{
    color = readPoint(x,y);
    if (x,y) is on canvas boundary
        roll back;
    if ((x,y) not on boundary and color != newColor)
    {
        setPixel(newColor, x, y);
        floodFill(x+1, y, newColor);  /* right neighbor */
        floodFill(x, y+1, newColor);  /* up neighbor    */
        floodFill(x-1, y, newColor);  /* left neighbor  */
        floodFill(x, y-1, newColor);  /* down neighbor  */
    }
}
```

You are requested to implement the algorithm without using recursion.

D Changes

When	Where	What
5/11/03	Page 7, Section 7.3	You are no longer required to submit the hardcopy of the code.
5/11/03	Page 3, Section 1.4	Parameters of CanvasSize can only be constants.
22/12/03	Section 1.5	Illegal parameters to built-in functions cause program exit (see also the forum for a related question)
19/01/04	Grading criteria	Late submission rules can be found on the project website.