Software project
Gnome Graphics

Olga Sorkine
sorkine@tau.ac.il

Andrei Scharf
asotzio@tau.ac.il

Office: Schreiber 002, 03-6405360

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

Course outline

- Two classes at the beginning of the semester
- Work in groups of TWO.
- Address questions to me in person, in the forum or by email (better the forum, so everybody can benefit from it).

- Handout next week:
  - Initial design
  - Sample UNIX program

Outline cont.

- The project definition is on the web-page
- Get updated at the web page **Every Week**
- Programming language: C or C++
- Operating system: linux in classroom 04

- 22-January-2004, the final project.

Grading

- Written material: more details are in the project definition

- Runtime:
  - Run tests and give a pass/fail to each test

Project overview

• Interpreter for graphical program language: GNOME

• Program controls the behavior of a small creature – the gnome – that walks and draws on a white canvas.

Overview cont.

- Canvas:
  - *n*-by-*m* pixels
  - white in the beginning of execution

- Gnome:
  - Width of gnome's paintbrush is 1 pixel
  - Anytime: the gnome has certain position on the canvas and a heading direction

Project goal

Write parser that reads GNOME language:
- GNOME programming language

Implement GNOME programming language in C/C++ to execute various GNOME programs:
- GNOME graphics

GNOME programming language

- Built-in commands (Forward, Back…)
- Built-in variables ($PositionX, …)
- User defined routines (ROUTINE Square)
- User-defined variables ($len=10)
- Operators (+, -, …)
- Supported functions (sin, cos, …)
- Supported commands (if, else…)

Sample program

```
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)
}
```

Variables and Expressions

Built-in Variables:

- $PositionX
- $PositionY

User defined Variables:

- $len

Expressions are of the form:

- <Value>
- <Value> <op> <Value>
- Function(value)
- Function(value, value)

```
$x = 5
$y = 17
$x2 = $x * $x
$y2 = $y * $y
$len2 = $x2 + $y2
$len = sqrt($len2)
```

Routines

- Routines receive zero or more parameters
- Can be recursive
- Do not return any value

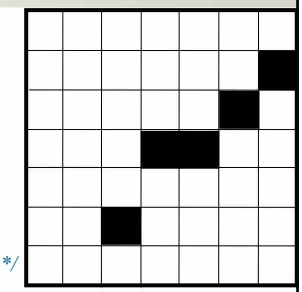
```
ROUTINE Square($len) {
        DrawSegment($len)
        TurnRight(90)
        DrawSegment($len)
        TurnRight(90)
        DrawSegment($len)
        TurnRight(90)
        DrawSegment($len)
}

ROUTINE Recurse($face, $grad) {
        if ($face>=0)
          DrawSegment(10)
          TurnRight($grad)
          $face = $face-1
          Recurse($face, $grad)
        endif
}
```

GNOME Graphics

- Canvas: rectangle of size *n x m*
- Coordinate system: origin (0,0) is on the canvas in the lower left corner
- Position of gnome is in floating point (5.5, 17.06).
- Canvas is a discrete pixel grid
- Mapping gnome drawing to canvas: *Rasterization*

Canvas



- Bitmap with 1bit per pixel (black/white 1/0)

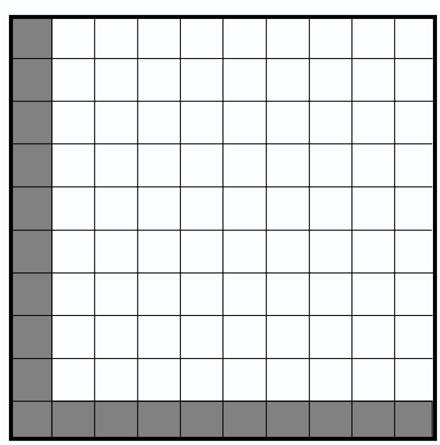
- Data structure:

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

- Each row is stored in integer amount of bytes (if the number of pixels per row is not some multiplication of 8, last bits are ignored)

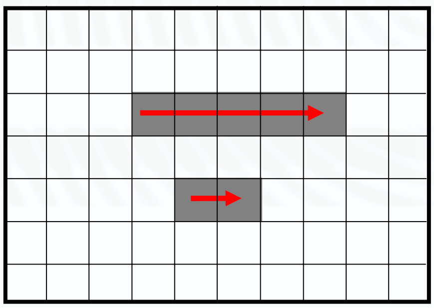
Black/White Bitmap

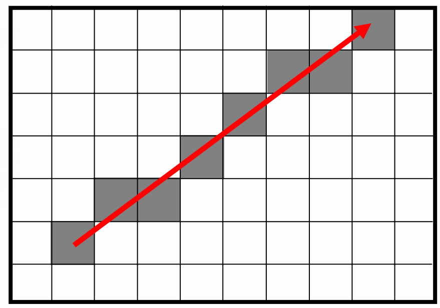
```
unsigned char bottom_left_corner[] =
{
0xff, 0xa0, /* 111111111000000 */
0x80, 0x0,  /* 100000000000000 */
0x80, 0x0,  /* 100000000000000 */
0x80, 0x0,  /* 100000000000000 */
0x80, 0x0,  /* 100000000000000 */
0x80, 0x0,  /* 100000000000000 */
0x80, 0x0,  /* 100000000000000 */
0x80, 0x0,  /* 100000000000000 */
0x80, 0x0,  /* 100000000000000 */
0x80, 0x0   /* 100000000000000 */
};
GnomeBitmap canvas;
canvas.width = 10;
canvas.height = 10;
canvas.bits = bottom_left_corner;
```



Rasterization

Raster: coloring the appropriate pixels so that the resulting image resembles the lines that the gnome drew





Constraints

- Initialization:
  - gnome's initial position/heading
  - canvas size
- Walking:
  - The gnome is not allowed to step out of the canvas
  - If the gnome is attempting to walk outside of the canvas:
    - execution stops,
    - output the resulting image created so far.

PGM file format

PGM (**p**ortable **g**ray**m**ap)
- “magic number” **P2**
- Whitespace (blanks, TABs, CRs, LFs).
- Width Height
- Maximum gray value
- Width * Height gray values, ASCII decimal, between 0 and maximum
- #Comment
- Max line length = 70 characters.

```
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 0
```

Bresenham’s Midpoint Algorithm

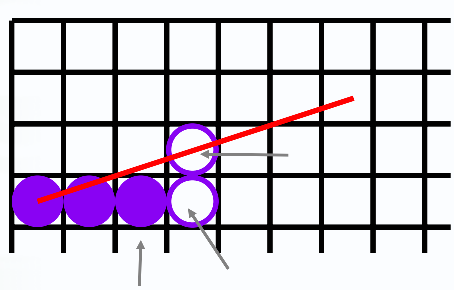
Goal: Draw the line between (x_1,y_1) and (x_2,y_2), m=dy/dy onto the discrete canvas.

Assume: m<1, $x_1<x_2$ and $y_1<y_2$

At each step:

$x = x+1$

$y = ?$



Decision:

- Calculate error between candidate pixels’ centers and real line (by looking at the vertical distance)
- Choose pixel with the smaller error

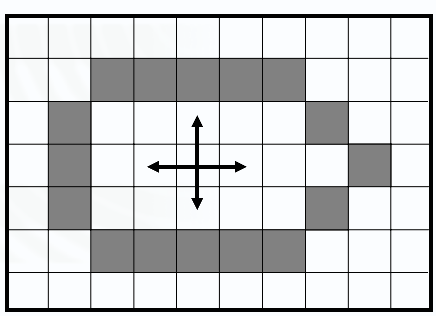
PseudoCode

$m = (y_2 - y_1)/(x_2 - x_1)$
$i_1 = \text{floor}(x_1)$
$j = \text{floor}(y_1)$
$i_2 = \text{floor}(x_2)$
$e = -(1-(y_1 - j) - m(1-(x_1 - i_1)))$

```
for i = i1 to i2
    TurnOnPixel(i, j)
    if (e >= 0)
        j += 1
        e -= 1.0
    end if
    i += 1
    e+= m
end for
```

Flood Fill algorithm

Problem: given a 2D closed polygon, fill its interior on a graphic display.



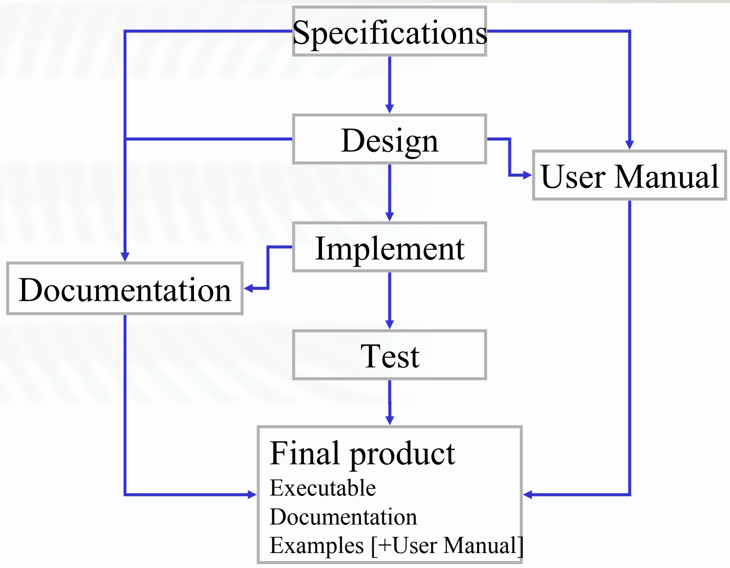
Pseudo Code

```
void 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);
          floodFill(x, y+1, newColor);
          floodFill(x-1, y, newColor);
          floodFill(x, y-1, newColor);
    }
}
```

Requirements

- A working project
- Documentation
- Examples

Developing software



Design – Modules (modularity…)

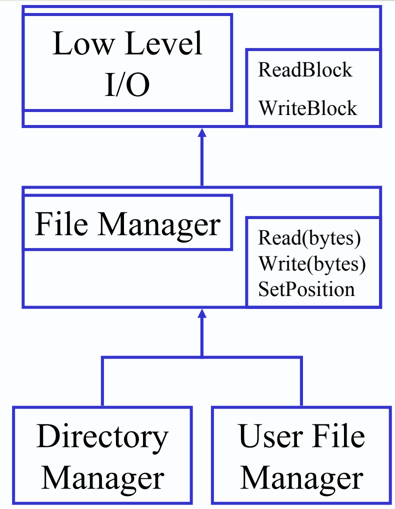
- Manage a single independent entity/ one responsibility
- Hold a set of functionalities
- Modules interact through an interface

List module
- create
- insert
- delete
- find
- ~~ComputeIntervals~~

Modules

You know your modules are OK if:

- You can **name** your modules and their **responsibility**
- You can define the **interaction**
- You can specify the **services** modules require from one another



Design

Read the project definition carefully and then:

- Modules diagram
- Description of modules
- Data structures

Next week

- More about the project
- Software engineering
- Questions

Makefile

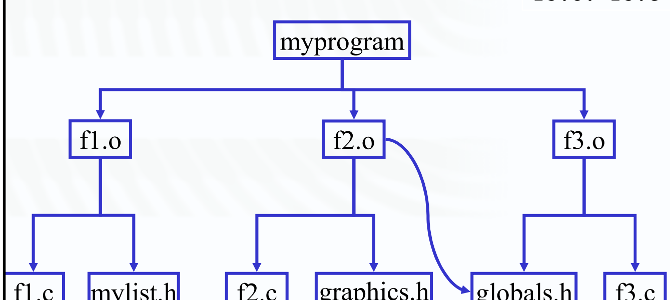
- **Dependency tree**
  - DAG actually

```
myprogram : f1.o f2.o f3.o

f1.o: f1.c myslist.h

f2.o: f2.c graphics.h globals.h

f3.o: f3.c globals.h
```



Makefile

- Dependency tree
- **Commands**

```
myprogram : f1.o f2.o f3.o
<TAB>   gcc -o myprogram f1.o
f2.o f3.o

f1.o: f1.c myslist.h
<TAB>   gcc -c -Wall f1.c

f2.o: f2.c graphics.h globals.h
<TAB>   gcc -c -Wall f2.c

f3.o: f3.c globals.h
<TAB>   gcc -c -Wall f3.c
```

Makefile

- Dependency tree
- Commands
- **Automatic variables**
  - $@ what stands before the colon (:)
  - $^ everything that stands after the colon
  - $< the first thing that stands after the colon

```
myprogram : f1.o f2.o f3.o
<TAB>   gcc -o $@ $^

f1.o: f1.c myslist.h
<TAB>   gcc -c -Wall $<

f2.o: f2.c graphics.h globals.h
<TAB>   gcc -c -Wall $<

f3.o: f3.c globals.h
<TAB>   gcc -c -Wall $<
```

Makefile

- Dependency tree
- Commands
- Automatic variables
- **Variables**

```
CFLAGS = -c -g -Wall
LIBS = -lm

myprogram : f1.o f2.o f3.o
<TAB>   gcc -o $@ $^ $(LIBS)

f1.o: f1.c myslist.h
<TAB>   gcc -c $(CFLAGS) $<

f2.o: f2.c graphics.h globals.h
<TAB>   gcc -c $(CFLAGS) $<

f3.o: f3.c globals.h
<TAB>   gcc -c $(CFLAGS) $<
```

Makefile

- Dependency tree
- Commands
- Automatic variables
- Variables
- **Implicit rules**
- **Multiple targets**
  - **Default: make all**
  - **“make clean”**
- **More information**
  - **NOVA: “tkinfo make”**

```
CFLAGS = -g -Wall
LIBS = -lm

.c.o:
<TAB>   $(CC) -c $(CFLAGS) $<

all: myprogram

myprogram : f1.o f2.o f3.o
<TAB>   gcc -o $@ $^ $(LIBS)

f1.o: f1.c myslist.h

f2.o: f2.c graphics.h globals.h

f3.o: f3.c globals.h

clean:
<TAB>   rm -f *.o *~
```

Unix program

- Built of three .c files and appropriate .h files
  - main.c
  - strrdup.c
  - Another.c
  - Makefile

Debugging in Unix (gdb)

Compile using “gcc -g”

gdb myprogram

l main - list the function main, **l misc.c:foo** - list foo() in misc.c

b 52 - set a break point at line 52

help

where prints the stack

up,down move in the stack, to inspect variables of calling routines.

run the program

n,s step over and step into

ddd – gdb + graphical interface (a bit more convenient)

Resources

Online help while using GDB.

Quick reference card, download from the web-page

> “info GDB” command on the unix