Laboratory 6 – Vector Graphics

Vector graphics is a technique in computer graphics where information on visible entities (shapes, text, colours, etc.) is stored in vectors. Vector graphics is an alternative to bitmap, where information about colour/luminosity is maintained for each point of the image (pixel). In scientific research, VG images are preferred to bitmap to represent data/diagrams because they store only the essential information and provide higher rendering quality when stretched horizontally or vertically. Examples of vector graphics formats are SVG (Scalable Vector graphics), Postscript, and WMF (Windows Meta File).

In this laboratory, we consider a simple vector graphics format with three coloured shapes:

- **segment**: A straight line between (x0, y0) and (x1, y1).
- **rectangle**: A rectangle with sides aligned with the axes, defined by two opposite corners (x0, y0) and (x1, y1)
- **circle**: A circle defined by the centre point (x0, y0) and radius "r"

The shapes are contained in a multiline text file, where each line represents a shape. The text file *devil.dat* contains an image with three circles (lines starting with "circle"), a rectangle (lines starting with "rectangle"), and two segments (starting with "segment"):

```
circle    320 200 180   0
circle    390 250  20   1
circle    250 250  20   1
rectangle 400 140 240 100   6
segment   330 250 380 300   2
segment   310 250 260 300   2
```

For segments and rectangles, the first four numbers after the string represent the coordinates of the two points (x0, y0) and (x1, y1). For circles, the first two numbers represent the coordinate of the centre, and the third number represents the radius.

For all shapes, the last number represents the colour according to the following encoding (this should not be copy-pasted as already available in the header file *bitpmap.h* provided):

```
#define BLACK   0
#define RED     1
#define BLUE    2
#define GREEN   3
#define CYAN    4
#define YELLOW  5
#define PURPLE  6
#define PINK    7
```

In order to complete Part 1 and Part 2 of this laboratory, a library consisting of a header file bitmap.h and an implementation file bitmap.c provided in the lab folder should be copied on the computer into the same folder as the project file (where also the main file resides). Also, the two files should be added to the list of project files so that they can be compiled together with the main file. In Dev-C++, files can be added to the project via the menu "Project -> Add To Project … ".

The main file should import the header library using the statement:

```
#include "bitmap.h"
```

Remember that a header file contains only the prototypes of the "Application Program Interface" (API functions), not their definition. The API definition is in the implementation file together with other working functions that should not be a concern for the user and should remain hidden.

Part 1 – Creating an internal representation of the drawing using a list

Our first goal is to load all the graphical elements into a "linked list" defined by the following typedef in order to provide an internal representation of the drawing:

```
typedef struct shape {
    point* pvec;
    int len;
    int closed;
    int color;
    struct shape* next;
} shape;
```

Each shape is a vector of points ("point" is defined in bitmap.h). The field *len* represents the number of points in the shape, the field *closed* determines if the shape forms a closed loop (i.e. if a line between the last and first point is present), the field *color* maintains the colour encoding according to the given palette.

Finally, the pointer *pvec* points an array whose elements are "*struct point"*. As a new entry is added to the list, memory should be allocated also for the array and the field *len* should be initialised consistently. The type *point* is defined in *bitmap.h* as follow:

```
typedef struct point {
    int x, y;
} point;
```

When a shape is defined, the number of points should reflect the number of vertices of the shape. For example, a rectangle should have four points corresponding to the four corners. Each point should be set to the correct coordinates x and y. In our implementation we approximate a circle with a closed regular polygon of 100 sides.

The table below shows the four functions to implement. All functions receive the list as a "double reference" (*shape ** plist*). This means that a pointer to *head* is given as reference instead of "*head*". Thus, dereferencing the pointer once allow to change the head when the new element is added at the beginning. The list should contain the elements in the same order as they appear.

Function Prototype	Description
`void rectangle(point p1, point p2, shape **slist, int col)`	The function rectangle adds a new entry with 4 points to the list modifying the head if needed. The field "color" is initialised too. Memory should be allocated for the vectors of points too.
`void circle(point c, double r, shape** slist, int col);`	The function circle adds a new closed shape with 100 points to the list modifying the head if needed and allocating memory. The function *cos()* and *sin()* from math.h should be used to calculate the position of the points.

`void segment(point p1, point p2, shape** slist, int col);`	The function segment adds two points to the list updating head and allocating memory.
`void load_shapes(char* fname, shape** slist);`	The text file "fname" provided as argument as is open for reading. If the file cannot be opened an error message is generated and the function terminates. The file is iterated until EOF. For each line, a string is read. If the string character is "rectangle" (check with *strcmp()*), the other parameters should be read using *fscanf()* and the function rectangle() should be called to update the list with points and colour of the rectangle. Similarly, if the string is "circle" or "segments" the functions circle() or segment() should be called with appropriate parameters.

Part 2 – Converting the drawing into a bitmap

After creating the list, we convert the internal representation into a bitmap file. The file should be saved with extension .bmp and opened using a bitmap view (the preview app in Windows or Mac is able to visualise .bmp files).

The following function from bitmap.h should be used to convert the internal representation into a bitmap

- `canvas* make_canvas()`
  - The function creates a canvas and returns its handler (ie. a pointer). The canvas consists of a 640x400 pixels matrix initially blank (all points are white). The other library functions can be used to draw point and lines on the canvas by changing the colour of pixels. The origin is at the lower-left corner and the coordinates grow positive upwards and towards right.

- `void draw_point(canvas *cp, point p, int col)`
  - The function draws the specified *point p* in colour *col* on the canvas referenced by *cp*. If the point is outside the canvas, no change is made.

- `void draw_line(canvas *cp, point p1, point p2, int col)`
  - The function draws a line from *point p1* to *point p2* in colour *col* on the canvas referenced by cp. Parts of the segment outside the canvas are ignored.

- `void save_bitmap(canvas *cp, char* filename)`
  - The function saves the canvas refenced by *cp* on the file specified by *filename*. The extension to filename should be .bmp. If the file cannot be opened for writing, an error message is produced.

Implement the function convert() that draws a list of shapes onto a canvas. The prototype is the following:

```
void convert(shape* sp, canvas* cp);
```

The first argument is a list of shapes, the second is the canvas handler. The function should scan the list *sp* and for every entry draw the set of lines that make up the shape. The function should also close the shape if the *close* field is set.

The function should be called from the main() so to create the bitmap file from “devil.dat”:

1. Load a list of shapes from the file devil.dat
2. Create a new canvas and store the handler
3. Translate the shape into the bitmap by calling convert()
4. Save the bitmap into the file “devil.bmp”