French Azerbaijani University
Arash Habibi
Second year
2019-2020

Data Structure and Algorithms 1

LineDraw project

1 Statement of problem

The aim of this project is to design a drawing tool capable of producing drawings composed of *piecewise linear 2D curves*. In this document, such curves are called *line strips*. Figure 1 represents one line strip. Figure 2 is a drawing composed of several line strips. We want our program (let us call it *LineDraw*) to be able to help the user to make such drawings.

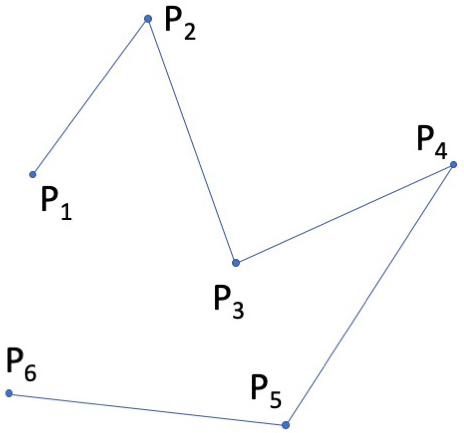

FIGURE 1: A piecewise linear 2D curve also called a *line strip* composed of 6 vertices : P_1 to P_6 and 5 edges linking these vertices.

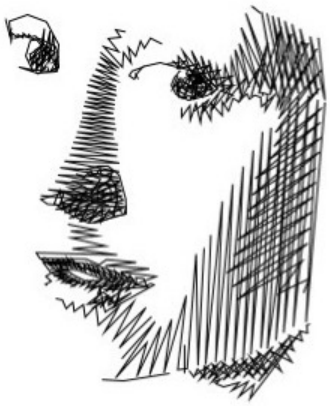

FIGURE 2: A drawing composed of several line strips.

The realisation of this project consists of several stages. As you go through these stages, your program will gain new features and your competence level will increase.

2 Stage 1 : exploring the tools : beginner level

In this stage, you will discover :

— The *OpenGL* graphic library which will enable you to draw line strips ;

— The *GLUT* library which will enable you to open a graphic window and to react to graphic events (mouse click, key press etc).

These libraries were formerly used for real-time graphic projects. Today more powerful libraries are preferred, but the learning of these tools is much more tedious. However the structure of these programs are often very similar.

2.1 OpenGL (Graphic library)

Here are a minimal set of *OpenGL* functions that you need to know in order to work your way in this project :

— `glViewport` defines a *viewport* which can roughly be described as a graphic window. The following call initializes the window by indicating its size.

```
glViewport(0, 0, window_width, window_height);
```

— `glClearColor` defines the erase color, in other words, the background color. The following call specifies that the background color should be white.

```
glClearColor(255,255,255,0);
```

— The previous two functions should be called only once in the beginning of your program. The following functions should be called each time you refresh the drawing. `glOrtho` defines the coordinates of the viewport. The following call :

```
glOrtho(0,window_width,window_height,0,-1,1);
```

means that :

- the x coordinates of the window are between 0 (left) and `window_width` (right).
- the y coordinates of the window are between 0 (top) and `window_height` (bottom).

— `glClear` actually erases everything and floods the window with the clear color (defined by `glClearColor`).

```
glClear(GL_COLOR_BUFFER_BIT);
```

— `glCcolor3ub` defines the current color. Everything drawn after this instruction will be drawn in the current color. The following call specifies that the current color should be red.

```
glColor3ub(255,0,0);
```

— `glBegin` and `glEnd()` define the beginning and the end of a drawing sequence. *OpenGL* can draw many shape primitives. We will only draw line strips. `glVertex2i` defines the actual position of the line strip's vertices. In the following sequence, we draw a line strips composed of three vertices : (100,100), (100,400) and (450,200).

```
glBegin(GL_LINE_STRIP);
glVertex2i( 100, 100);
glVertex2i( 100, 400);
glVertex2i( 450, 200);
glEnd();
```

— Calling `glFlush` tells *OpenGL* that you have finished defining your drawing and that it can proceed to execute your orders.

```
glFlush();
```

— There are other function calls which correspond to *GLUT* functions that we will see further.

On the *Moodle* platform :

1. Download the `drawing.tgz` archive ;
2. Uncompress it (`tar -xzvf drawing.tgz`);
3. Enter the directory ;
4. Type `make` to compile the program ;
5. Run the program (`./linedraw 500 500`)

For the time being, the only way to terminate the program is `Ctrl + C`.

Your job is to :

— modify the arguments of `glVertex2i` and `glColor3ub` to change the position of the vertices and the color of the line strip.

— How could you add new vertices ?

— Draw a red cross (one horizontal red line and one vertical red line of length = height = 10 pixels).

2.2 Graphic Library Utility Toolkit (GLUT)

You may think that, since we know how to make line strips, we have finished the project. But the aim of our program is not to draw ONE drawing but to enable the user to draw any drawing each time he runs the program. So we need to interact with the user.

Until now, the programs that you have written are composed of a set of function calls. Each time you run the program, the functions are called in the same order. In this project, you will discover *event-based programming*. This type of programming is connected with a certain number of possible *events*. In our case : pressed keys, mouse clicks, mouse movements, resized windows etc. In *event-based programming*, there are roughly two stages :

— One initialization stage in which, among other steps, each graphical event is associated with a function (called *call-back* functions) ;

— One infinite loop waiting for events. At each event, the associated function is called.

— and that is all.

This means that the order in which the functions are called depends plainly on the user's behaviour. The functions will probably never be called in the same order twice. On the *Moodle* platform :

1. Download the `events.tgz` archive ;
2. Uncompress it (`tar -xzvf events.tgz`);
3. Enter the directory ;
4. Type `make` to compile the program ;
5. Run the program (`./linedraw 500 500`);
6. See how the program reacts to mouse clicks and keyboard events (specifically to the Escape key) ;

This new program is very similar to the previous one, but it contains several function definitions with a `_CB` suffix, which stands for *callback*. Here are a few examples : `_display_CB` (already seen in the previous program) `_reshape_CB`, `_keyboard_CB`, `_spcial_CB`, `_mouse_CB` and `_mouse_move_CB`. In the `main` function, a series of function calls associate each of these functions with a special type of event :

```
glutDisplayFunc(_display_CB);
// specifies that each time we have to draw the scene, _display_CB should be called

glutReshapeFunc(_reshape_CB);
// specifies that each time the window is resized, _reshape_CB should be called

glutKeyboardFunc(_keyboard_CB);
// specifies that each time a key (characterized with an ascii code) is pressed,
// _keyboard_CB should be called ;

glutSpecialFunc(_special_CB);
// specifies that each time other keys (arrows, function keys, etc) are pressed,
// _special_CB should be called ;

glutMouseFunc(_mouse_CB);
// specifies that each time a mouse button is pressed or released,
// _mouse_CB should be called ;

glutMotionFunc(_mouse_move_CB);
// specifies that each time the mouse moves when clicked,
// _mouse_move_CB should be called.
```

Your job is to modify the previous program in such a way that :

— more keyboard events are recognized : at least 'a', 'e', 'f', 'r' RETURN, Backspace etc. Choose yourself how you want your program to react to these keys ;

— left clicking prints `hello !`. Other mouse buttons have no effect. Releasing mouse buttons should not print anything either ;

2.3 Global variables

The arguments of these `...Func` functions must have a very specific prototype. You can choose any name for those argument functions, but you cannot change the number and type of those functions' parameters. This is the reason why, if these functions need other data than those provided by their parameters, you will have to use global variables. This is a good occasion for you to see the advantages and cons of using such global variables.

Moving the first vertex

— Define two integer global variables `px=100` and `py=100` which represent the position of the line strip's first vertex. In order to draw this vertex, you will have to make the following call : `glVertex2i(px,py);` instead of `glVertex2i(100,100);`.

— Change the program so that pressing the left arrow key shifts the first vertex to the right by 5 pixels (likewise for the other arrow keys).

Now, we want to create different modes. Right now when you click, your program says `hello`. But :

— after we press 'r', we would like the mouse clicks to say hello in Russian ;

— after we press 'a', we would like the mouse clicks to say hello in Azeri ;

— after we press 'e', we would like the mouse clicks to say hello in English ;

— after we press 'f', we would like the mouse clicks to say hello in French.

In order to do so :

— Define a new integer global variable called `language=0`.

— Change the program (in `_keyboard_CB`) so that pressing 'a', 'e', 'f' and 'r' sets the value of `language` to different values ;

— Change the program (in `_mouse_CB`) so that the reaction of the mouse click depends on the value of `language`.

The exploration period comes to an end. Now you master your tools enough to achieve the project.

3 Stage 2 : Vectors and distances : Intermediate level

In this stage, you will write, from A to Z a header file and corresponding source file. But the function prototypes are clearly defined. In the next stages, you will have to analyse the needs and to build your own function prototypes.

In stage 4 (section 6) we will need to *select* vertices and edges. *Selecting* a vertex or an edge is done by clicking *"near"* that vertex or that edge. In our case, *"near"* means less than 5 pixels away. Therefore we need to be able to determine the distance between any vertex and the mouse click. In the same way, we need to determine the distance between any edge (line segment defined by two points) and a mouse click. The distance between a line segment $[AB]$ and a point P is the smallest distance between P and any point of $[AB]$. In other words :

$$dist(P,[AB]) = min_{M\in[AB]} dist(P,M) \tag{1}$$

This means that we have to represent the mouse click and all vertices by a specific data structure capable of representing vectors and points. The aim of this section is to enable you to calculate the distance between two points, and also the distance between a point and a line segment.

3.1 Vectors

Create a new header and source file called respectively `Vector.h` and `Vector.c`. Vectors can be defined by two floating point numbers (`x` and `y`). We need at least these functions, but feel free to write any intermediate function if needed.

— `Vector V_new(float x, float y);`
// which returns a new vector whose coordinates are `x` and `y` ;

— `void V_show(Vector v, char *label);`
// which prints the coordinates of `v` on the terminal, with possibly an identifying label ;

— `float V_PtPtDistance(Vector P, Vector A);`
// which returns the distance between point P and point A ;

— `float V_PtSegmentDistance(Vector P, Vector A, Vector B);`
// which returns the distance between point P and the $[AB]$ line segment ;

I believe the first two functions will be straightforward for you. For the other functions, if you have an idea, go ahead and implement it. Otherwise, you can also seek inspiration in the *Geometry* document on *Moodle*.

In order to test `V_PtPtDistance`, define a point A in your program and represent it graphically with a red cross. At each mouse click, put the position of the mouse click in a vector P and print the AP distance on the terminal. The printed number should be all the smaller than your clicks are near the red cross. I suggest that you test the `V_PtSegmentDistance` in a similar manner.

If you fail to find something for the fourth function, you may ask your teacher for help. Even if you do not find anything, you can still work on the next sections. We will have to find a turnaround solution in order to select edges.

> **Beware !** The only source file in which global variables are tolerated is the main source file containing the *GLUT* call-back functions. In particular, the `Vector.c` and `Vector.h` must not contain any global variables.

4 General procedure for each stage

The three next sections correspond to three stages. At each stage, as soon as you have achieved something satisfactory, something working, even partially, I strongly recommend that you do not continue modifying the same program. Do not touch the (partially or perfectly) working program ! Change the access permissions so that nobody can change them. Then make a copy of the whole program of this working version and modify the copy. In this way, even when you are working on some current stage, you can always show the results of the previous stage.

At each of the three following stages, you will have to implement increasingly complex algorithms. For each algorithm, you will have to design different data structures for the line strip. This data structure will be described in the `lineStrip.h` header file and the `lineStrip.c` source file. In order to write these files, at each stage, you will have to answer a series of questions (roughly the same questions at each stage). Here are the questions. I will not repeat them at each stage. I ask you to answer to these questions in the form of comments in the beginning of `lineStrip.h` header file.

1. Given the algorithm, enumerate the required interactions with the data structure (whatever this structure may be). These interactions must be independant of the graphical interface. Here are a few examples : add a new vertex, remove the latest vertex, remove the oldest vertex, remove a specific vertex, insert vertex after a given vertex, insert vertex at a specific index, move vertex index i, move vertex specified by a pointer etc. What are the interactions implied by the algorithm ?
2. Enumerate the function prototypes that should enable these interactions ;
3. Given this list of interactions, what abstract data type is most adapted to represent a line strip ? stack of vectors ? queue of vectors ? list of vectors ? others ?
4. We want the program to be able to add as many vertices to the line strip as the user needs. The only limit should be the computer's memory. Given this constraint, what concrete data structure should we use ? array ? linked list ?

The aim is NOT to imagine thousand possible interactions at random, choose the most complex data structure for all stages (like a doubly-linked circular list), write all the thousand functions and, in the end, pick those who may turn out to be handy for your specific algorithm. The aim is to find the minimal set of interactions and the simplest data structure suitable for those interactions. This means that, for each algorithm, you have a different set of interactions, different header files and different data structures.

After answering the above questions :

— Write the function prototypes in `LineStrip.h` and implement them in the source file ;

— I suggest that you first tested these functions with a classical (non-graphical) main function ;

— Finally, integrate these functions in the graphical main source file, in such a manner that the user can build any line strip as described in the algorithm.

5 Stage 3 : Creating and removing vertices : Expert level

Here is the algorithm, in other words, the way we want our program to work :

— When the program is launched, we only have a white window with no vertices and no edges ;

— Each time the user makes a left mouse click, a new vertex is added at the position of the click ;

— Each time the user presses the Backspace key, the last vertex is removed ;

— Pressing the Escape key exits the program.

6 Stage 4 : Editing the curves : Guru level

Here is how we want our program to work at this new stage. Our system is always in one of two modes : *create* or *edit*. Moreover, the line strip is characterized by one selected vertex or one selected edge (not both).

— In create mode :
  — everything works as in the previous stage ;
  — the selected vertex is always the most recent one.
  — Pressing the ENTER key switches to *edit* mode ;

— In edit mode :
  — clicking near a vertex selects that vertex and unselects everything else ;
  — clicking near an edge selects that edge and unselects everything else ;
  — when a vertex is selected, the arrow keys move the selected vertex in the corresponding direction ;
  — when a vertex is selected, the backspace key removes the selected vertex. The new selected vertex is one of the neighbors of the removed vertex (you may choose which one).

- when an edge is selected, pressing 'i' (as in *insert*) inserts a new vertex at the middle of the selected edge. The new selection is the newly created vertex.
- pressing 'n' (as in *new*) switches back to create mode.
- if the number of vertices is smaller than 2, then the system automatically switches to create mode.

— Pressing the Escape key exits the program.

The selected vertex should be displayed by a red cross. The selected edge should be displayed in another color. The strips should be represented in different colors depending on the mode (for example red in create mode and green in edit mode).

7 Stage 5 : Several curves : Genius mutant level

In the previous stages, we only dealt with one line strip. At this stage, we wish to deal with as many line strips as needed by the user. The system is characterized by three modes. The edit mode has been split into two modes. The three modes are : *create*, *object* and *component*. The system is always characterized by one selected line strip. The distance between a point P and a line strip S is the smallest distance between P and any of the edges of S.

— In create mode :
- everything works as in the previous stages ;
- Pressing the ENTER key switches the system to *object* mode ;

— In object mode :
- clicking near a line strip selects that line strip ;
- pressing the backspace key removes the selected line strip ;
- pressing 'n' key switches to *create* mode in order to create a new strip ;
- pressing ENTER enters the *component* mode ;

— In component mode :
- The component mode works as the edit mode in the previous stage.
- Pressing the ENTER key switches to *object* mode.
- Pressing the 'n' key switches to *create* mode in order to add vertices to the same strip.

— Pressing the Escape key exits the program.

8 General conditions

This project will not be graded, therefore there is no deadline. You may work on this project at the university or at home, alone or with other students or with the help of any person you wish to work with. However, I strongly recommend that you do not copy/paste any code but understand the proposed code and write your own version of the project. Otherwise, it will be very difficult (almost impossible) for you to do the very last stage of this project (stage 6, not described in this document) which will take place in the University in limited time.

This last stage, the practical exam, will be graded. It will require that you have reached at least the expert level. A part of the exam can be done even if you have not done the project at all. Another part will consist of modifying your project. Thus if you are able to modify features introduced in the last stage, your grade will be better than if you only modify features of stage 3. If you have any questions, feel free to ask me or your other teachers. My email address is `ahabibi@unistra.fr`.