Fundamentals of Programming 1

Introduction to 2D Graphics — Part One
The Allegro Library

Arkadiusz Chrobot

Department of Computer Science

June 3, 2020

Outline

Introduction

Contemporary computer systems use advanced graphics processing units to generate real-like images and three dimensional animations. However, this lecture is about two dimensional graphics. In the second part are presented simple, yet interesting programs that generate more or less complex images and animations. The first part of the lecture is about the Allegro library that provides means for writing software for processing graphics. The library is based on a similar one, which was available for Atari ST computers. It is mainly used for creating 2D and even 3D computer games. That's why it provides subroutines for processing graphics, sound, keyboard input, mouse input and even creating GUIs. In the lecture only the basic elements of the library are discussed. More information about it can be found on the following web page: http://liballeg.org/. The library is *open-source software* and can be downloaded from the that page.

Initialization and Finalization

Any program using the Allegro library has to include the `allegro.h` header file. If it needs to process the keyboard input then it also should include the `allegro/keyboard.h` header file (the notation means: the `keyboard.h` file in the `allegro` directory). The main subroutine that initializes the library is the `allegro_init` macro, which takes no arguments and returns an `int` number. If the number is different than zero, then it means that the initialization has failed. A string explaining the cause of the failure is stored in a global variable named `allegro_error`. It is an array of characters. The information can be displayed on the screen with the use of the `allegro_message()` function, which is invoked similarly to the `printf()` function. In a text mode it prints the message directly on the screen, while in the graphical mode it creates a window with the information. It can be used in the program only before the `set_gfx_mode()` function is invoked.

Initialization and Finalization

The `set_color_depth()` function allows the programmer to define the colour depth, i.e. the number of bits allocated in computer memory for storing information about a single pixel colour. The function takes an `int` number as its argument, but only following values are valid: `8`, `15`, `16`, `24` and `32`. The last value is used in the example programs. It means that the information about a colour occupies four bytes in the memory. The first one stores the level of red, the second one stores the level of green and the third one stores the level of blue. Those are primary colours, which means that any other colour can be expressed as a combination of the four. The last byte stores so-called alpha channel and it describes how opaque the pixel is. This colour description format is called the RGBA *colour space*. The `set_color_depth()` function returns no value.

Initialization and Finalization

The `set_gfx_mode()` function turns on a specific graphical mode. It takes five arguments. The first one is one of the following constants:

`GFX_AUTODETECT` – the function tries to turn on a graphical mode of a specified resolution, in a window or in a full screen,

`GFX_AUTODETECT_FULLSCREEN` – the function tries to turn on a graphical mode of a specified resolution in a full screen,

`GFX_AUTODETECT_WINDOWED` – the function tries to turn on a graphical mode of a specified resolution in a window,

`GFX_SAFE` – the function tries to turn on any graphical mode that will work,

`GFX_TEXT` – the function tries to turn on any text mode that will work.

Initialization and Finalization

Two next arguments of the `set_gfx_mode()` function are of `int` type and they describe the resolution of the graphical mode. The first one describes the number of pixels in the horizontal direction, the second one defines the number of pixels in the vertical direction. Two last arguments are also of the `int` type and define the resolution of so-called virtual screen, which is used for animations. The resolution of this screen is based on the resolution of the real screen. If a program doesn't create animations, then those arguments should be zero. If the `set_gfx_mode()` function returns value different than zero then turning on the graphical mode has failed. A program should call `allegro_exit()` to finish using the Allegro library. This function returns no value. The `END_OF_MAIN` macro ought to be also used in the program. It has to be expanded *just after* the `main()` function definition. This macro takes no arguments and performs finalization operations specific to the operating system under which the program using Allegro library works.

Image Organization

In the graphical mode the screen is divided into points called *pixels*, which stands for picture elements. A program that utilizes the Allegro library can access the screen using the `screen` global variable. The data type of the variable is `BIMAP*` which means it is a pointer to a `BIMAP` structure. The structure represents a set of pixels called a *bitmap*. The number of pixels in the screen is determined by two constants:

`SCREEN_H` – the number of pixels in the vertical direction,

`SCREEN_W` – the number of pixels in the horizontal direction.

Each pixel has coordinates that are a pair of natural numbers. The origin of the coordinate system is in the top left corner of the screen. The values of abscissa grow from the left to the right, and the values of ordinate grow from the top to the bottom. The maximal value of the ordinate is `SCREEN_H-1`. For the abscissa it is `SCREEN_W-1`. The origin has coordinates of $(0, 0)$.

Drawing Primitives

A simple element of an image is called a *primitive* in the computer graphics. The most primary primitive is a pixel. The `putpixel()` function sets a colour of a single pixel. The first argument of the function is a pointer to a bitmap structure (`BITMAP *`), where the colour for the pixel should be changed. Often the `screen` variable is used as this argument. Two next arguments are of the `int` type and they are coordinates of the pixel. The last argument is also of the `int` type and it is the code of the colour. It can be acquired in several different ways. One of them is using the `makecol()` function which takes three `int` numbers as arguments. Those numbers are levels of the red, green and blue. Despite their types the values of those arguments should be ranging from `0` to `255`. The value returned by the function is of the `int` type and it is the colour code. Other way of acquiring a colour code is to use predefined colour codes stored in the `palette_color` array, which has 256 elements.

Drawing Primitives

A line segment can be drawn in an image with the use of the `line()` function, which takes six arguments. The first one is the pointer to a bitmap structure. Two next, of the `int` type are the coordinates of the starting point of the segment, while other two of the same type are the coordinates of the ending point. The last argument is the code of the colour of the segment. A circle can be drawn with the use of `circle()` function, which takes five arguments. The first one is a pointer to a bitmap structure. All other arguments are of the `int` type. The second and third argument are the coordinates of the center of the circle. The forth argument is the length of the circle radius. The last one is the code of the circle colour. A rectangle, including a square, can be drawn with the use of the `rect()` function. It takes six arguments. The first one is a pointer to a bitmap structure. The rest of the arguments is of the `int` type. The second and third argument are the coordinates of the top left corner of the rectangle. The forth and fifth are the coordinates of the top right corner. The sixth argument is the code of the rectangle colour. All introduced functions for drawing primitives return no values.

Keyboard Handling

The keyboard input handling is initialized by the `install_keyboard()` function, which takes no arguments and returns zero on success or a nonzero number on failure. The information about pressed keys is not immediately available to software, but it is first stored in a dedicated RAM area called the *keyboard buffer*. The `keypressed()` function check if there are any data about pressed keys in the keyboard buffer. It returns an `int` number and takes no arguments. If the keyboard buffer is empty the `keypressed()` function returns zero, otherwise a nonzero value. Information about a single pressed key is returned by the `readkey()` function that takes no arguments and returns an `int` value. The least significant byte of the value is the ASCII code of the character associated with the key and the second byte stores the so-called scan code of the key. Often the function is used for holing the program until the user presses any key. In that case it is worth to know, that some keys store more then two bytes of informations in the keyboard buffer, so the function doesn't always stop and wait for new data about pressed keys.

Keyboard Handling

To avoid the issue described in the previous slide the `clear_keybuf()` function should always be called before the `readkey()` function. The former takes no arguments and returns no value, but empties the keyboard buffer. The state of every key can be monitored with the help of the `key` array. Each element of the array is of `char` type and if its value is different from zero, then the key to which it corresponds is currently pressed, otherwise it is not. There are defined constants that can be used as indices for the array. The names of those constants start with the `KEY_` prefix and correspond to the keys on the keyboard. For example the `KEY_ESC` constant indicates the element of the `key` array that describes the state of the *Escape* key and the `KEY_Q` constant indicates the element associated with the *q* key. Description of the rest of those constants can be found in a manual published on the web page which URL is in the first slide.

Animation

The Allegro library provides means for quickly drawing subsequent frames of an animation. The animation should be displayed with the frequency of at least 24 frames per second, to be perceived by a human brain as fluent. The effect can be achieved with the help of pages implemented in the Allegro library. A page is a bitmap on which a single frame of an animation is drawn. The Allegro library enables fast switching of pages. If two pages are available, the content of the first one can be displayed on the screen, and in the meanwhile the next frame of animation can be drawn on the second page. After the new frame is finished the pages switch places. The second one is displayed and the first one is used for drawing the next frame. The process continues as long as the animation lasts.

Animation

There are a few functions in the Allegro library that handle the pages. The `create_bitmap()` function creates a single bitmap. It takes two arguments of the `int` type, which define the resolution of the bitmap. The first one is the number of pixels in horizontal direction, the second one is the number of pixels in the vertical direction. The function returns an address of a newly created bitmap or `NULL` if it fails. The bitmap has to be destroyed before the program stops running or changes a display mode. This can be done with the help of the `destroy_bitmap()` function, which takes the pointer to the bitmap structure as its argument and returns no value. The `clear_bitmap()` function clears a bitmap by setting all its pixels to the default, black color. The function takes a pointer to the bitmap structure and returns no value. The `show_video_bitmap()` function displays a bitmap on the screen. It takes a pointer to the bitmap structure as an argument and returns an `int` number. If the value is not a zero then it means the function failed to complete its task.

Text Displaying

The Allegro library also provides functions for displaying text on the screen in a graphical mode. In this lecture only two of them are described. The `textout_centre_ex()` function takes seven arguments. The first one is a pointer to a bitmap structure. The second one is a pointer to a structure describing a font for the text. The pointer is of the `FONT *` type. The third argument is a pointer of the `char *` type to a text. The forth and fifth arguments are of the `int` type and they are coordinates on the bitmap of the middle of the text. The last two arguments are also of the `int` type and they are, respectively, the code of the colour of the text and the code of the colour of the text background. If the value of the last one is -1 then the background will be transparent. The `font` variable can be passed as the second argument of the function. It is a global variable that points to a structure describing the default font used by the Allegro library functions. The `textout_centre_ex()` returns no value.

Text Displaying

The `textprintf_ex()` function is another one that places a text on a bitmap. Similarly to the function described in the previous slide, it takes as its first argument a pointer to a structure describing the bitmap and as it second argument the pointer to a structure describing a font. The next two arguments are of the `int` type and they are coordinates of the pixel on the bitmap from which the function should start displaying text. Another pair of arguments is also of the `int` type and they are, respectively, the code of the text colour and the code of its background colour. If the latter has a value of `-1` then the background will be transparent. The rest of the arguments is the same as for the `printf()` function. The `textprintf_ex()` function returns no value.

Example — a Simple Graphics

In the previous slides are described only those elements of the Allegro library that are applied in the example programs presented in the lecture. Although there are not many of them, they can be useful for creating interesting 2D graphics. As an example in this part of the lecture is presented a program that draws a random circles in a full screen mode until the user presses any key on the keyboard. The PRNG is used in the program for randomly choosing the length of the radius, the coordinates of the middle point and the colour of each of the circles.

Example — a Simple Graphics

```
#include<allegro.h>
#include<allegro/keyboard.h>
#include<stdlib.h>
#include<time.h>

#define WIDTH 1366
#define HEIGHT 768
```

Example — a Comment

In the beginning of the program four header files are included. The first two have been already described in the lecture. The `stdlib.h` and `time.h` header files are included because the program is using the PRNG. The `WIDTH` and `HEIGHT` constants define, respectively, the number of pixels in horizontal and vertical directions on the screen. In other words they define the screen resolution in a full screen mode.

Example — a Simple Graphics

```
int initialize(int card, int width, int height)
{
    srand(time(NULL));
    if(allegro_init()) {
        allegro_message("allegro_init(): %s\n",allegro_error);
        return -1;
    }
    if(install_keyboard()) {
        allegro_message("install_keyboard(): %s\n",allegro_error);
        allegro_exit();
        return -1;
    }
    set_color_depth(32);
    if(set_gfx_mode(card,width,height,0,0)) {
        allegro_message("set_gfx_mode(): %s\n",allegro_error);
        allegro_exit();
        return -1;
    }
    return 0;
}
```

Example — a Comment

The `initialize()` function initializes the PRNG, the Allegro library and the keyboard input handling. It also defines the colour depth (32 bits) and switches the monitor to a selected graphics mode. It achieves those goals by using macros and functions described in the previous slides. The `initialize()` function has three parameters of the `int` type. The first one is a value determining the graphics card driver. By the parameter is passed one of the constants mentioned in the description of the `set_gfx_mode()` function. Values of the two other parameters are defining the resolution of the image. Two last arguments of the invocation of the `set_gfx_mode()` function are zeros, because no animation support is needed in the program. If one of the Allegro library functions used in the `initialize()` function fails the user will be informed about that by the `allegro_message()` function, that will display the description of the exception. Then a function finalizing the Allegro library will be invoked and the `initialize()` function will return `-1` and exit. If all aforementioned functions successfully complete their tasks, the `initialize()` function will return zero.

Example — a Simple Graphics

```
void draw_random_circles(BITMAP* bitmap)
{
    clear_keybuf();
    while(!keypressed()) {
        int colour = makecol(rand()%256,rand()%256,rand()%256);
        circle(bitmap,rand()%SCREEN_W,rand()%SCREEN_H,rand()%50,colour);
    }
}
```

Example — a Comment

The `draw_random_circles()` function, as its name suggests, is responsible for the core activity in the program. It returns no value, but it has one parameter by which a pointer to a bitmap structure is passed. The circles are drawn on the bitmap. In the function body, before the `while` loop, the function responsible for clearing the keyboard buffer is called. Inside the loop the following steps take place. The colour of a circle is established by generating values for each of the primary colours. The values ranging from `0` to `255` are chosen randomly and passed as arguments to the `makecol()` function. The function returns the colour code which is assigned to the `colour` variable. A circle is draw with the use of the `circle()` function. Its first argument is the parameter of the `draw_random_circles()` function, which is a pointer to a bitmap structure. The coordinates of the middle of the circle are chosen randomly in the range from `0` to `SCREEN_W-1` (abscissa) and from `0` to `SCREEN_H-1` (ordinate). The length of the radius is chosen randomly in the range from `0` to `49`.

Example — a Comment

This means that some of the circles are drawn partially “outside“ the screen, due to the coordinates of their centers or are not drawn at all because the length of their radii is zero.
The `while` loop stops after the user presses any key on the keyboard.

Example — a Simple Graphics

```
int main(void)
{
    if(initialize(GFX_AUTODETECT_FULLSCREEN,WIDTH,HEIGHT))
        return -1;
    draw_random_circles(screen);
    allegro_exit();
    return 0;
}
END_OF_MAIN()
```

Example — a Comment

In the `main()` function first the `initialize()` function is called. Its first argument is the `GFX_AUTODETECT_FULLSCREEN` constant, which means that the `set_gfx_mode()` function, invoked inside the `initialize()` function, will try to turn on the full screen mode, which resolution is defined by the `WIDTH` and `HEIGHT` constants that are passed as the second and the third argument of the `initialize()` function. Next, the `draw_random_circles()` function is invoked. The `screen` variable is passed as its only argument. It means that all graphical operations inside the function are mapped directly to the screen. After the `draw_random_circles()` exits the `allegro_exit()` function in called, which finalizes the Allegro library. Please notice, that the `END_OF_MAIN` macro is expanded after the end of the `main()` function definition.

Questions

?

The End

Thank You for Your Attention!