Fundamentals of Programming 1

Introduction to 2D Graphics — Part Two
The Allegro Library

Arkadiusz Chrobot

Department of Computer Science

June 3, 2020

Outline

Introduction

In the second part of the lecture example programs are presented that apply the elements of the Allegro library, described in the first part, to create 2D images. Two of those programs use the animation support offered by the library.

Fractals

Fractals can be described as a complex geometrical objects. Mathematics provides many definitions of the concept of a fractal as well as many algorithms for creating fractals. The most characteristic property of fractals is their repetitive structure. In this lecture three examples of fractals, generated by different algorithms are presented.

Fractals — IFS

Fractals can be created with the help of an *Iterated Function System* (IFS). The algorithm computes coordinates of new points belonging to a fractal by using affine mappings of the following form:

$$\begin{cases} x' = a \cdot x + b \cdot y + c \\ y' = d \cdot x + e \cdot y + f \end{cases}$$

The x' and y' are coordinates of a new point of the fractal and the x and y are coordinates of a fractal point that is already known. In the IFS algorithm several affine mappings are defined and one of them is randomly chosen for calculating the coordinates of a next point of the fractal. The more points are calculated, the more detailed is the image of the fractal.

Fractals — IFS

The program that demonstrates how a fractal is created with the help of the IFS uses four affine mappings with the following a, b, c, d, e and f coefficients:

	a	**b**	**c**	**d**	**e**	**f**
1	-0,67	-0,02	0	-0,18	0,81	10
2	0,4	0,4	0	-1	0,4	0
3	-0,4	-0,4	0	-0,1	0,4	0
4	-0,1	0	0	0,44	0,44	-2

The coefficients are taken from the book „Fraktale i chaos“ by Jerzy Kudrewicz available only in Polish.

Fractals — IFS

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

#define WIDTH 1366

#define HEIGHT 768

#define SCALE 15
```

Fractals — IFS

Comment

Aside from the header files associated with the Allegro library there are also included in the program the header files needed for using the PRNG. The `WIDTH` and the `HEIGHT` constants describe the width and the height of the screen in pixels. The `SCALE` constant is the scaling factor for the image, which has to be zoomed, otherwise it would be relatively small.

Fractals — IFS

```
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("%s\n",allegro_error);
                allegro_exit();
                return -1;
        }
        return 0;
}
```

Fractals — IFS

Comment

The `initialize()` function is responsible for initializing the Allegro library and the PRNG. It is implemented in the same or similar fashion in the other example programs. Initialization of the PRNG is the first action performed inside the functions body. The task is accomplished with the invocation of the `srand()` function. Next, the Allegro library is initialized with the use of `allegro_init` macro. Then the keyboard handling is activated with the use of `install_keyboard()` function. After that the colour depth is set by calling the `set_color_depth()` function. The RGBA colour model is applied in the program. The last task performed by the `initialize()` function is setting a specified graphical mode by invoking the `set_gfx_mode()` function. The said function takes as its first three arguments the parameters of the `initialize()` function. The value of the two last arguments is zero, because the animation support offered by the Allegro library is not used in the program.

Fractals — IFS

Comment

If any of the subroutines used in the `initialize()` function fails then a message is printed with the use of `allegro_message()` function, the `allegro_exit()` function is called to finalize the Allegro library and a proper exception code is returned.

Fractals — IFS

```
void draw_with_ifs(double x, double y)
{
        const int GREEN_COLOUR = makecol(0,255,0);
        while(!(key[KEY_Q]||key[KEY_ESC])) {
                switch(rand()%4) {
                        case 0:
                                x=-0.678*x-0.02*y;
                                y=-0.18*x+0.81*y+10;
                                break;
                        case 1:
                                x=0.4*x+0.4*y;
                                y=-0.1*x+0.4*y;
                                break;
                        case 2:
                                x=-0.4*x-0.4*y;
                                y=-0.1*x+0.4*y;
                                break;
                        case 3:
                                x=-0.1*x;
                                y=0.44*x+0.44*y-2;
                                break;
                }
                putpixel(screen,(SCREEN_W>>1)-(SCALE*x),(SCREEN_H-35)-(SCALE*y),GREEN_COLOUR);
        }
}
```

Fractals — IFS

Comment

The `draw_with_ifs()` function is responsible for drawing a fractal on the screen. The coordinates of the first point belonging to the fractal (which is not drawn) are passed to the function by parameters. At the beginning of its body is defined the `GREEN_COLOUR` constant, which stores the code of the green colour returned by the `makecol()` function. In the `while` loop the coordinates of subsequent points belonging to the fractal are computed and the fractal is drawn. The loop stops when the user presses the `q` key. Inside the loop one on the affine mappings is chosen randomly and then it is applied for calculating the coordinates of a new point belonging to the fractal. Next, those coordinates are converted to the coordinates of a pixel belonging to the fractal image. Those numbers are passed as a second and third arguments of the `putpixel()` function that changes the colour of the pixel. The first argument of this function is the `screen` variable, which is a pointer to the bitmap associated with the screen. The last argument of the `putpixel()` function is the colour code.

Fractals — IFS

Comment

Because the coordinates of a point are floating point numbers relative to the origin of the "regular" Cartesian coordinate system, they have to be converted to the coordinates of a pixel on the screen. The horizontal coordinate of the pixel is calculated by scaling the value of the abscissa of the point and subtracting from the result the half of the width of the screen. The resulting data is indirectly casted to the `int` type. The vertical coordinate is calculated similarly. The value of the ordinate of the point is scaled and then subtracted from the hight of the screen reduced by 35 pixels. The last value was chosen experimentally. Thanks to those conversions the fractal image is displayed centered and is not inverted.

Fractals — IFS

```
int main(void) {
        if(initialize(GFX_AUTODETECT_FULLSCREEN,WIDTH,HEIGHT)<0)
                return -1;
        draw_with_ifs(0.0,0.0);
        allegro_exit();
        return 0;
}
END_OF_MAIN()
```

Fractals — IFS

Comment

In the `main()` function the `initialize()` function is invoked first. As its first argument the `GFX_AUTODETECT_FULLSCREEN` constant is given. It means that the program should use a full screen graphical mode, provided its initialization is successful. The constants that define the width and height of the screen are the two other arguments of the function. After the Allegro library is initialized, the `draw_with_ifs()` function is called. Its arguments are the coordinated of the starting point of the fractal. After this function exits the `allegro_exit()` function is invoked to finalize the Allegro library.

Fractals — IFS

Summary

The presented program creates a fractal called a Christmas Tree. With the help of Iterated Function Systems it is possible to create other shapes of that type, like for example Barnsley's Fern. In the case of the Barnsley's fractal not only a different set of coefficients for the system of affine mappings is needed but also a slightly different way of choosing the systems randomly have to be applied. Plant-like looking fractals can be also drawn with the use of formal languages called L–Systems, which were invented by a Hungarian biologist and botanists Aristid Lidenmayer. In computer graphics they were popularized by a Polish computer scientist working in Canada, Przemysław Prusinkiewicz. Fractals are used for example in in computer games for creating a background that looks like a realistic landscape.

Fractals — Mandelbrot Set

The Mandelbrot Set is a set of points on a plane for which a sequence given by a following recursive (self–repeating) equation does not diverge:

$$\begin{cases} z_0 = 0 \\ z_{n+1} = z_n^2 + c \end{cases}$$

In this equation z is a complex variable and c is a complex constant. The set was discovered by a French mathematician who was born in Warsaw in 1924. The image of the fractal is created by scaling the coordinates of a large number of points on a complex plane in such a way that their real and imaginary parts (abscissa and ordinate) belong, respectively, to the following intervals: $(-2.5, 1)$ and $(-1, 1)$ and then substituting them for c in the equation and calculating a number of initial terms of the sequence. If the modulus of every calculated term is less than 2 ($|z_n| \leq 2$) then the point c belongs the set.

Fractals — Mandelbrot Set

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

#define WIDTH 1366

#define HEIGHT 768

#define MAXITER 8000
```

Fractals — Mandelbrot Set

Comment

The beginning of the program that generates the image of Mandelbrot Set is similar to the beginning of the program that creates the Christmas Tree fractal using the IFS. However, instead of the constant for scaling, the `MAXITER` constant is defined that describes how many of the initial terms of the sequence defined in the previous slide should be calculated by the program. Also the header files necessary for using the PRNG are not included.

Fractals — Mandelbrot Set

```
int initialize(int card, int width, int height)
{
        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("%s\n",allegro_error);
                allegro_exit();
                return -1;
        }
        return 0;
}
```

Fractals — Mandelbrot Set

Comment

The initializing function is also similar to the one used in the previous program. The only difference is that this one doesn't initialize the PRNG.

Fractals — Mandelbrot Set

```
double scale_x0(int x0)
{
        return 3.5*(((double)x0)/(SCREEN_W-1))-2.5;
}
```

Fractals — Mandelbrot Set

```
double scale_y0(int y0)
{
        return 2.0*(((double)y0)/(SCREEN_H-1))-1.0;
}
```

Fractals — Mandelbrot Set

Comment

The `scale_x0()` and `scale_y0()` functions are responsible for converting the coordinates of a pixel to the coordinates of a point that lies on a $(-2,5;1) \times (-1;1)$ plane.

Fractals — Mandelbrot Set

```
unsigned int calculate_mandelbrot(int xp, int yp)
{
        double x,y,x2,y2;
        unsigned int iteration = 0;
        double x0 = scale_x0(xp);
        double y0 = scale_y0(yp);

        x=y=x2=y2=0.0;

        while(x2+y2<=4.0 && iteration<MAXITER) {
                double tmp = x2-y2+x0;
                y=2.0*x*y+y0;
                x=tmp;
                iteration++;
                x2=x*x;
                y2=y*y;
        }

        return iteration==MAXITER?0:iteration;
}
```

Fractals — Mandelbrot Set

Comment

The function `calculate_mandelbrot()` calculates successive terms of the sequence for a pixel which coordinates are passed to it by parameters. First, these coordinates are converted (scaled) with the use of `scale_x0()` and `scale_y0()` functions. Next, in the `while` loop the successive terms of the sequence are calculated until the iteration counter reaches the value of the `MAXITER` constant or the modulus of the current term is greater or equal two. The latter condition is expressed as: `x2+y2<=4.0`. That expression can be derived from the formula: $|z_n \leq 2|$ as follows: $|z_n \leq 2| \Rightarrow |x_n + i \cdot y_n| \leq 2 \Rightarrow \sqrt{x_n^2 + y_n^2} \leq 2 \Rightarrow x_n^2 + y_n^2 \leq 4$. In the loop body the successive terms of the sequence are calculated and the number of the loop iterations is counted. The expressions used for calculating the terms can be derived assuming that $z = x + i \cdot y$, and $c = x_0 + i \cdot y_0$. So, the real part of the next sequence term equals $x^2 - y^2 + x_0$ and the imaginary part $2 \cdot x \cdot y + y_0$. The function returns the number of iterations of the loop for a given pixel or zero, if the number reached the value of `MAXITER` constant.

Fractals — Mandelbrot Set

```
int main(void) {
        if(initialize(GFX_AUTODETECT_FULLSCREEN,WIDTH,HEIGHT)<0)
                return -1;
        unsigned int x,y;
        unsigned char color;
        for(y=0;y<SCREEN_H;y++)
                for(x=0;x<SCREEN_W;x++) {
                        color=calculate_mandelbrot(x,y);
                        putpixel(screen,x,y,palette_color[color]);
                }
        while(!(key[KEY_Q]||key[KEY_ESC]))
                ;
        allegro_exit();
        return 0;
}
END_OF_MAIN()
```

Fractals — Mandelbrot Set

Comment

In the `main()` function aside from Allegro library initialization and finalization the Mandelbrot Set image is drawn. In the `for` loops for each of the pixel are calculated successive terms of the sequence with the help of the `calculate_mandelbrot()` function. The function returns the number of iterations after which the `while` loop stopped or zero. The returned value is used for calculating the code of the pixel colour. In the program a predefined palette of colours is applied which is stored in the `palette_color` array. The array has `256` elements, so the number of iterations has to be converted into a natural number ranging from `0` to `255`. This is accomplished by storing the result of `calculate_mandelbrot()` function in a variable of the `unsigned char` type. After the image is generated the program waits int the `while` loop for the user to press the `Esc` or `q` key. Please note, that no other activities take place in the loop aside from checking the state of the aforementioned keys.

Fractals — Mandelbrot Set 2

Creating the Mandelbrot Set image involves performing calculations with the use of complex numbers. The ISO C99 standard introduces to the C language elements that make implementation of such calculations easier. They are collected in the `complex.h` header file. Some of them are applied in the next program, which also draws Mandelbrot Set. Those elements are: the `complex` macro for creating the `double complex` type, the `I` constant of the $\sqrt{-1}$ value and the `cabs()` function that calculates the modulus of a complex number. For basic arithmetic operations on the `double complex` type variables the same operators can be applied as for variables of other numerical data types. Next slides present the program that creates the Mandelbrot Set image using complex number operations. Due to the similarity of the program with the previous one, only the differences between them are described.

Fractals — Mandelbrot Set 2

```
#include<allegro.h>
#include<allegro/keyboard.h>
#include<complex.h>

#define WIDTH 1366

#define HEIGHT 768

#define MAXITER 8000
```

Fractals — Mandelbrot Set 2

Comment

The code presented in the previous slide includes to the program the `complex.h` header file, which contains elements supporting complex number operations.

Fractals — Mandelbrot Set 2

```
int initialize(int card, int width, int height)
{
    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("%s\n",allegro_error);
        allegro_exit();
        return -1;
    }
    return 0;
}
```

Fractals — Mandelbrot Set 2

```
double scale_x0(int x0)
{
    return 3.5*(((double)x0)/(SCREEN_W-1))-2.5;
}
```

Fractals — Mandelbrot Set 2

```
double scale_y0(int y0)
{
    return 2.0*(((double)y0)/(SCREEN_H-1))-1.0;
}
```

Fractals — Mandelbrot Set 2

```
unsigned int calculate_mandelbrot(int xp, int yp)
{
    double complex z;
    unsigned int iteration = 0;
    double complex c = scale_x0(xp) + I*scale_y0(yp);

    z=0.0+0.0*I;

    while(cabs(z)<=2.0 && iteration<MAXITER) {
        z = z*z+c;
        iteration++;
    }

    return iteration==MAXITER?0:iteration;
}
```

Fractals — Mandelbrot Set 2

Comment

The `calculate_mandelbrot()` function applies the `double complex` type variables to calculate successive terms of the sequence. First, from the coordinates of a pixel is created the value of the `c` constant. Next, in the `z` variable is stored the value of the first term of the sequence. Then in the `while` loop the successive terms of the sequence are calculated and the number of the loop iterations is counted. The function code is more legible than its equivalent from the previous program, but its performance is worse.

Fractals — Mandelbrot Set 2

```
int main(void)
{
    if(initialize(GFX_AUTODETECT_FULLSCREEN,WIDTH,HEIGHT)<0)
        return -1;
    unsigned int x,y;
    unsigned char color;
    for(y=0; y<SCREEN_H; y++)
        for(x=0; x<SCREEN_W; x++) {
            color=calculate_mandelbrot(x,y);
            putpixel(screen,x,y,palette_color[color]);
        }
    while(!(key[KEY_Q]||key[KEY_ESC]))
        ;
    allegro_exit();
    return 0;
}
END_OF_MAIN()
```

Fractals — Mandelbrot Set 2

Comment

Other ways of creating the Mandelbrot Set and drawing fractals are described on the following web page: https://lodev.org/cgtutor/. The theoretical description of the Mandelbrot Set is partially taken from the Polish and English pages of Wikipedia.

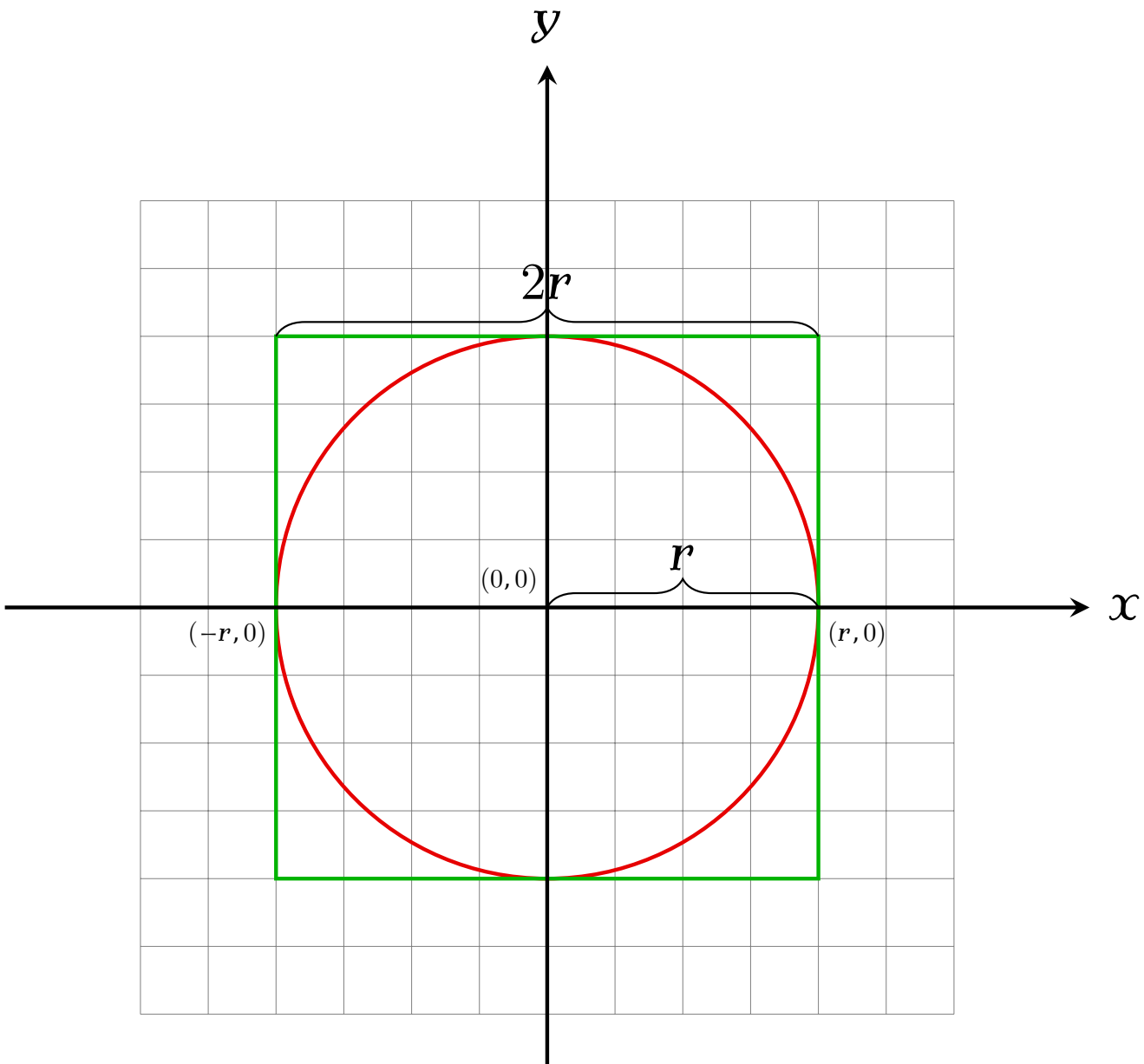
Calculation of π

The π number is one of the most frequently appearing constants in mathematics. Although its definition is simple — it is the ratio of the circle's circumference to its diameter — calculating its approximated value is not easy. There are many algorithms for that problem. The next program calculates the value of π using one of the statistical methods which are collectively know as Monte Carlo Methods. The name was coined by Polish mathematician Stanisław Ulam who also invented some of them.

Calculation of π

The method that is applied in this program is not the most efficient one as its steps needs to be repeated multiple times to achieve at least rough approximation of the value of the π. However, it can be demonstrated in an interesting way. In this method a disc inscribed in a square is given. The length of the disc's radius is r. The centre of the disc is in the origin of the coordinate system. Every side of the square has a length of $2r$ (please refer to the next slide). The ratio of the square area (P_{sq}) to the area of the disc (P_{dc}) is $\frac{P_{sq}}{P_{dc}} = \frac{(2 \cdot r)^2}{\pi \cdot r^2}$. If areas of the square and the disc were known then the value of the π could be calculated with the use of the following formula: $\pi = \frac{4 \cdot P_{dc}}{P_{sq}}$. It is possible to approximate the two missing values by choosing randomly points inside the square and verifying if they also belong to the disc. The square area is the total number of chosen points, and the disc area is the number of the chosen square points that belong also to this figure. More about this method can be found on the web page of Eve Astrid Andersson (http://www.eveandersson.com/pi/).

Calculation of π



Calculation of π

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

#define WIDTH 1366
#define HEIGHT 768
```

Calculation of π

Comment

The beginning of the program is similar to the beginnings of previously presented programs. Aside from the header files required by Allegro library also the header files necessary for using the PRNG are included, together with header files that contain declarations of needed mathematical functions and the definition of the `bool` data type.

Calculation of π

```
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;
}
```

Calculation of π

Comment

The definition of the `initialize()` function is the same as in the first program presented in this lecture.

Calculation of π

```
bool is_in_disc(double x, double y, const double radius)
{
    return sqrt(pow(x,2)+pow(y,2))<=radius;
}
```

Calculation of π

Comment

The `is_in_disc()` function verifies if a point, which coordinates are passed by the first two parameters, is inside a disc of a radius, which length is passed by the last parameter. According to the definition of a disc, all points that belong to such a figure lie within the distance measured from the disc's centre that is less than or equal to the radius. Because the centre of the disc, in the discussed method, is in the origin of the coordinate system, the distance from the disc centre to a point of the coordinates (x, y) can be calculated with the use of the following Euclidean formula: $\sqrt{x^2 + y^2}$. If the resulting value is less than or equal to the length of the radius then the point belongs to the disc, otherwise it doesn't. The `pow()` function used in the program is the exponentiation function. It takes two numbers of the `double` type as arguments and returns the value of the first raised to the power of the second. The result is also a `double` type number.

Calculation of π

```
void draw_pi(void)
{
    const int GREEN_COLOUR = makecol(0,255,0),
              RED_COLOUR = makecol(255,0,0),
              WHITE_COLOUR = makecol(255,255,255),
              SCREEN_MIDDLE_X = SCREEN_W>>1,
              SCREEN_MIDDLE_Y = SCREEN_H>>1;
    const double RADIUS = 300.0;
    unsigned long int in_square=0, in_disc=0;
    while(!keypressed()) {
        double x=-RADIUS+rand()%600+rand()/(1.0+RAND_MAX);
        double y=-RADIUS+rand()%600+rand()/(1.0+RAND_MAX);
        in_square++;
        if(is_in_circle(x,y,RADIUS)) {
            putpixel(screen,x+SCREEN_MIDDLE_X,y+SCREEN_MIDDLE_Y,RED_COLOUR);
            in_disc++;
        } else
            putpixel(screen,x+SCREEN_MIDDLE_X,y+SCREEN_MIDDLE_Y,GREEN_COLOUR);
        if(!(in_square%100000))
            textprintf_ex(screen,font,550,50,WHITE_COLOUR,0,"The PI number is: %1.20lf",
                (4.0*in_disc)/in_square);
    }
}
```

Calculation of π

Comment

The `draw_pi()` function makes the calculation of the π visible and also prints the current approximation of the number on the screen. In the function body three constants describing the codes of green, red and white colours are defined. The next two constants define the coordinates of the middle of the screen. They are used for converting coordinates of points to coordinates of pixels in such a way, that the resulting image is displayed in the centre of the screen. The last constant defined in the function describes the length of the radius. It is `300` points. In the function are also declared two variables which store the number of points belonging to the square (`in_square`) and to the disc (`in_disc`). Inside the `while` loop, which is performed until the user presses any key, the coordinates of a point in the square are randomly chosen and the value of the variable that counts them is incremented by one. The conditional statement checks if the point also belongs to the circle.

Calculation of π

Comment

If it is the case then the colour of the pixel corresponding to that point is set to red and the value of the `in_disc` variable, which is the counter of the points in the disc, is incremented by one. Otherwise, the pixel colour is set to green. Please notice, that the coordinates of the middle of the screen are added to the coordinates of the point. Also the coordinates of the point, which are of the `double` type are indirectly casted to the `int` type, which means that many of the randomly chosen points are mapped into the same pixel. The calculated value of the π number is displayed each `100.000th` iteration of the loop with the use of the `textprintf_ex()` function. The value is calculated using the formula described in previous slides. The approximated area of the square (P_{sq}) is stored in the `in_square` variable and the approximated area of the disc (P_{dc}) is stored in the `in_disc` variable.

Calculation of π

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

Calculation of π

Comment

In the `main()` function are invoked the other functions defined in the program.

Sine Wave

The next program plots a sine wave. This example serves the purpose of describing the general method of plotting a function graph on the computer screen. It is not enough to calculate the coordinates of points belonging to the graph and to set colour of pixels that correspond to those points. That way only a discrete function graph can be created — one that consists of many separate points. The correct way of plotting a function consists in creating the graph from very small line segments joined together.

Sine Wave

```
#include<allegro.h>
#include<allegro/keyboard.h>
#include<math.h>

#define WIDTH 1366
#define HEIGHT 768
```

Sine Wave

Comment

The program source code begins similarly to the code of other programs presented in this lecture. The `math.h` header file is necessary for using the sine function.

Sine Wave

```
int initialize(int card, int width, int height)
{
    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;
}
```

Sine Wave

Comment

The definition of the `initialize()` function is the same as in other example programs that don't require using the PRNG.

Sine Wave

```
void draw_sinus(BITMAP* bitmap)
{
    const double DEGREE_TO_RADIAN = M_PI/180.0, HALF_OF_SCREEN = SCREEN_H>>1, X_RATIO = SCREEN_W/360.0;
    double x_start=0.0,y_start=0.0,x_end,y_end;
    const int GREEN_COLOUR = makecol(0,255,0);
    for(x_end=1;x_end<=360;x_end++) {
            y_end = HALF_OF_SCREEN*sin(x_end*DEGREE_TO_RADIAN);
            line(bitmap,x_start*X_RATIO,HALF_OF_SCREEN-y_start,X_RATIO*x_end,HALF_OF_SCREEN-y_end,
                GREEN_COLOUR);
            x_start = x_end;
            y_start = y_end;
    }
}
```

Sine Wave

Comment

The `draw_sinus()` function plots the sine wave on the screen. At the beginning of its body three constants are declared. The `DEEGRE_TO_RADIAN` is used for converting degrees to radians. The `sin()` function requires as an argument an angle measured in radians. The `HALF_OF_SCREEN` constant[1] has a value that represents the half of the screen hight in pixels. The `X_RATIO` constant is the ratio of the screen width to the number of degrees in the full angle. In other words this constant expresses the number of pixels per one degree. Finally, the last constant is the green colour code. This is the colour of the function graph. The `x_start` and `y_start` variables are used for storing the coordinates of the starting point of currently plotted segment. The `x_end` and `y_end` variables store the coordinates of the ending point of that segment.

[1] Its type can be actually changed to `int`.

Sine Wave

Comment

The function is plotted in the `for` loop. The starting point of the first segment has the coordinates of the $(0, 0)$. The coordinates of the ending point are calculated during the first iteration of the loop. Plotting the sine wave directly would result in a "slightly jagged" line appearing on the screen, since the values of the sine function belong to the $[-1, 1]$ interval. This is why the ordinate of every point is multiplied by the value of `HALF_OF_SCREEN` constant. This causes the graph to be plotted across the full hight of the screen. The argument of the `sin()` function is an angle measured in radians. Hence the program multiplies the angle measured in degrees by the value of the `DEEGRE_TO_RADIAN` constant and passes the result to the function. After the coordinates of the ending point are calculated the segment is drawn on the screen with the use of the `line()` function.

Sine Wave

Comment

The abscissas of the starting and the ending pixel of the segment are obtained by multiplying the angle measured in degrees by the `X_RATIO` constant, so the graph is plotted across the full width of the screen. The ordinates are also subtracted from the half of the hight of the screen. This results in the graph being not inverted and displayed in the centre of the screen. In the next iteration of the `for` loop the coordinates of the ending point of the next segment of the graph are calculated. The coordinates of the ending point of the previous segment become the coordinates of the starting point of the next segment. Those steps are repeated until the whole graph is finished.

Sine Wave

```
void wait_for_any_key(void)
{
    clear_keybuf();
    while(!keypressed())
        ;
}
```

Sine Wave

Comment

The `wait_for_any_key()` function stops the program until the user presses any key on the keyboard. The `clear_keybuf()` function is invoked to clean the keyboard buffer and thus to make sure that the `wait_for_any_key()` function will work correctly. Next, the latter function waits in the `while` loop for the user to press any key. No other actions are performed in the loop.

Sine Wave

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

Sine Wave

Comment

In the `main()` function all the other functions defined in the program are called.

Animation — Rectangle

The next program shows an animation of a rectangle that moves from the left side of the screen to the right side and it repeats that movement until the user presses `q` or `Esc` key.

Animation — Rectangle

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

#define WIDTH 1366
#define HEIGHT 768
#define NUMBER_OF_PAGES 4

BITMAP *pages[NUMBER_OF_PAGES];
```

Animation — Rectangle

Comment

Unlike the program plotting a sine wave, this one doesn't include the header file with mathematical functions definitions. Instead, a constant is defined that specifies the number of pages, or in other words bitmaps, used by the animation support in the Allegro library. The constant is used in the definition of an array which elements are pointers to bitmap structures.

Animation — Rectangle

```
int initialize(int card, int width, int height)
{
    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,NUMBER_OF_PAGES*height)) {
        allegro_message("set_gfx_mode(): %s\n",allegro_error);
        allegro_exit();
        return -1;
    }
    return 0;
}
```

Animation — Rectangle

Comment

The definition of the `initialize()` function is similar to its equivalents in presented programs that do not use the PRNG. However, as the two last arguments for the `set_gfx_mode()` function are passed the dimensions of the virtual screen. They are necessary for enabling the animation support and for creating new bitmaps. Please notice, that it is enough if only one of those arguments is not zero.

Animation — Rectangle

```
int create_pages_array(BITMAP *pages[NUMBER_OF_PAGES])
{
    int i;
    for(i=0; i<NUMBER_OF_PAGES; i++) {
        pages[i] = create_video_bitmap(SCREEN_W,SCREEN_H);
        if(pages[i]==NULL)
            return -1;
    }
    return 0;
}
```

Animation — Rectangle

Comment

The `create_pages_array()` function creates bitmaps in a loop and assigns their addresses to the elements of the array of pointers to bitmap structures. If it fails to create any of them it returns a value specifying the exception and exits.

Animation — Rectangle

```
void destroy_pages_array(BITMAP *pages[NUMBER_OF_PAGES])
{
    int i;
    for(i=0; i<NUMBER_OF_PAGES; i++)
        destroy_bitmap(pages[i]);
}
```

Animation — Rectangle

Comment

The `destroy_bitmap_array()` function deletes the bitmaps created by the `create_bitmap_array()` function.

Animation — Rectangle

```
void animate_rectangle(BITMAP *pages[NUMBER_OF_PAGES], int speed, int rectangle_width, int rectangle_height)
{
    int page_number = 0;
    int x = 0;
    clear_keybuf();
    const int YELLOW_COLOUR = makecol(255,255,0);
    while(key[KEY_ESC]==0&&key[KEY_Q]==0) {
        BITMAP *active_page = pages[page_number];
        clear_bitmap(active_page);
        rect(active_page,x,SCREEN_H>>1,rectangle_width+x,rectangle_height+(SCREEN_H>>1),YELLOW_COLOUR);
        x=(x+speed)%SCREEN_W;
        if(show_video_bitmap(active_page))
            return;
        page_number = (page_number+1)%NUMBER_OF_PAGES;
    }
}
```

Animation — Rectangle

Comment

The `animate_rectangle()` function displays a rectangle that moves from the left edge of the screen to the right. The first argument of this function is the array of pointers to pages. The value of the second one specifies the speed of the animation (how quickly the rectangle moves). Values of the last two arguments specify dimensions of the rectangle. The `page_number` variable is used for indexing the array and also for specifying which of the pages is currently active. The `x` variable stores the abscissa of the top-left corner of the rectangle. The `YELLOW_COLOUR` constant stores the code of the yellow colour. This is the colour of the rectangle. The `clear_keybuf()` function is invoked before the `while` loop to clear the keyboard buffer. Inside the loop the subsequent frames of the animation are created until the user presses the `Esc` or the `q` key.

Animation — Rectangle

Comment

Inside the `while` loop the `page_number` variable specifies the element of the array that stores the address of the active page. This address is assigned to the `active_page` pointer and the page is cleared (the colour of all its pixels is set to black). Next the rectangle is drawn to the bitmap (the active page). Its top edge is at the half screen hight. Then the abscissa of the rectangle top-left corner location in the next frame is calculated with the use of the modular arithmetic. This allows the rectangle to "show up" at the left edge of the screen every time it "passes through" the right edge. After the abscissa is calculated the content of the active page is displayed on the screen. Then the function calculates a new value of the `page_number` variable to determine the next active page. This time it also uses the modular arithmetic, so each bitmap becomes the active page once every `NUMBER_OF_PAGES` iterations of the loop.

Animation — Rectangle

```
int main(void)
{
    if(initialize(GFX_AUTODETECT_FULLSCREEN,WIDTH,HEIGHT)<0)
        return -1;
    if(create_pages_array(pages))
        return -1;
    animate_rectangle(pages,1,100,50);
    destroy_pages_array(pages);
    allegro_exit();
    return 0;
}
END_OF_MAIN()
```

Animation — Rectangle

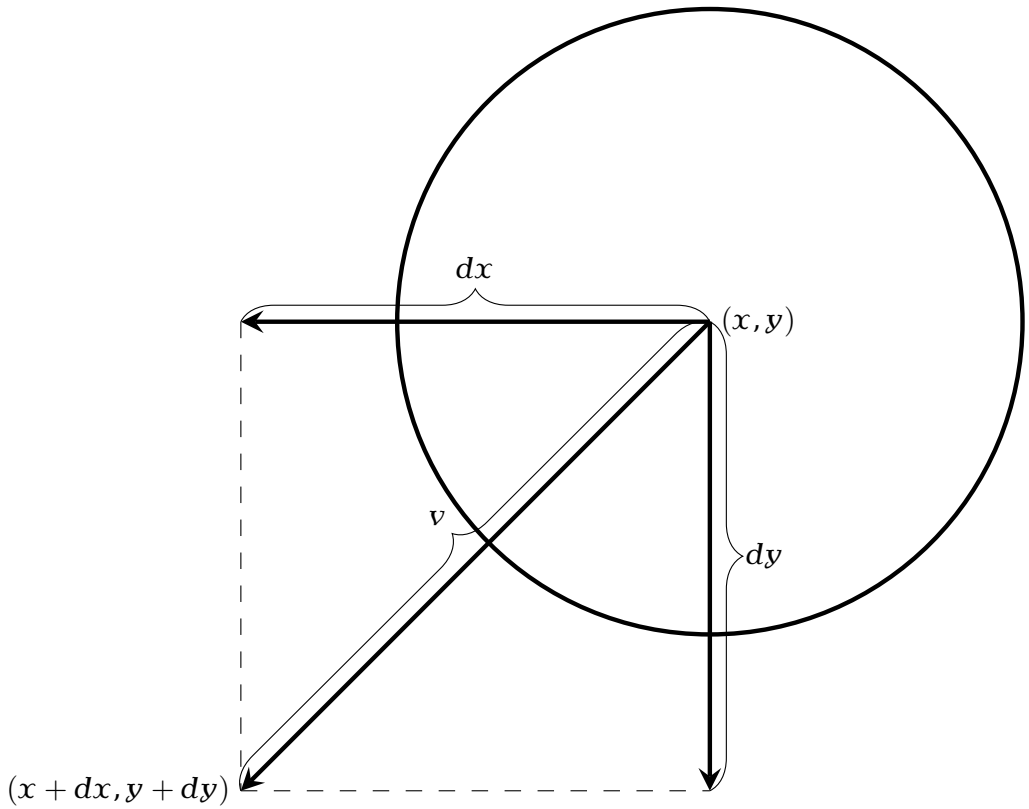
Comment

In the `main()` function the functions defined earlier in the program are called. Please note the order of invocations of the `create_pages_array()` and `destroy_pages_array()` functions.

Animation — Ball

The next example program also creates an animation, but this time it is an animation of a ball that bounces off the edges of the screen according to the law of reflection, which applies to the light rays: the angle of reflection is equal the angle of incidence. To simulate the movement of the ball, which is drawn as a circle, the program has to know the coordinates of the ball centre (x, y), its velocity vector (dx, dy) and the length of its radius (r). Those data are necessary for calculating the location of the middle of the ball, its direction (angle) and velocity in the next frame of the animation. Dependencies between the first two of those data items are shown in the figure in the next slide.

Animation — Ball

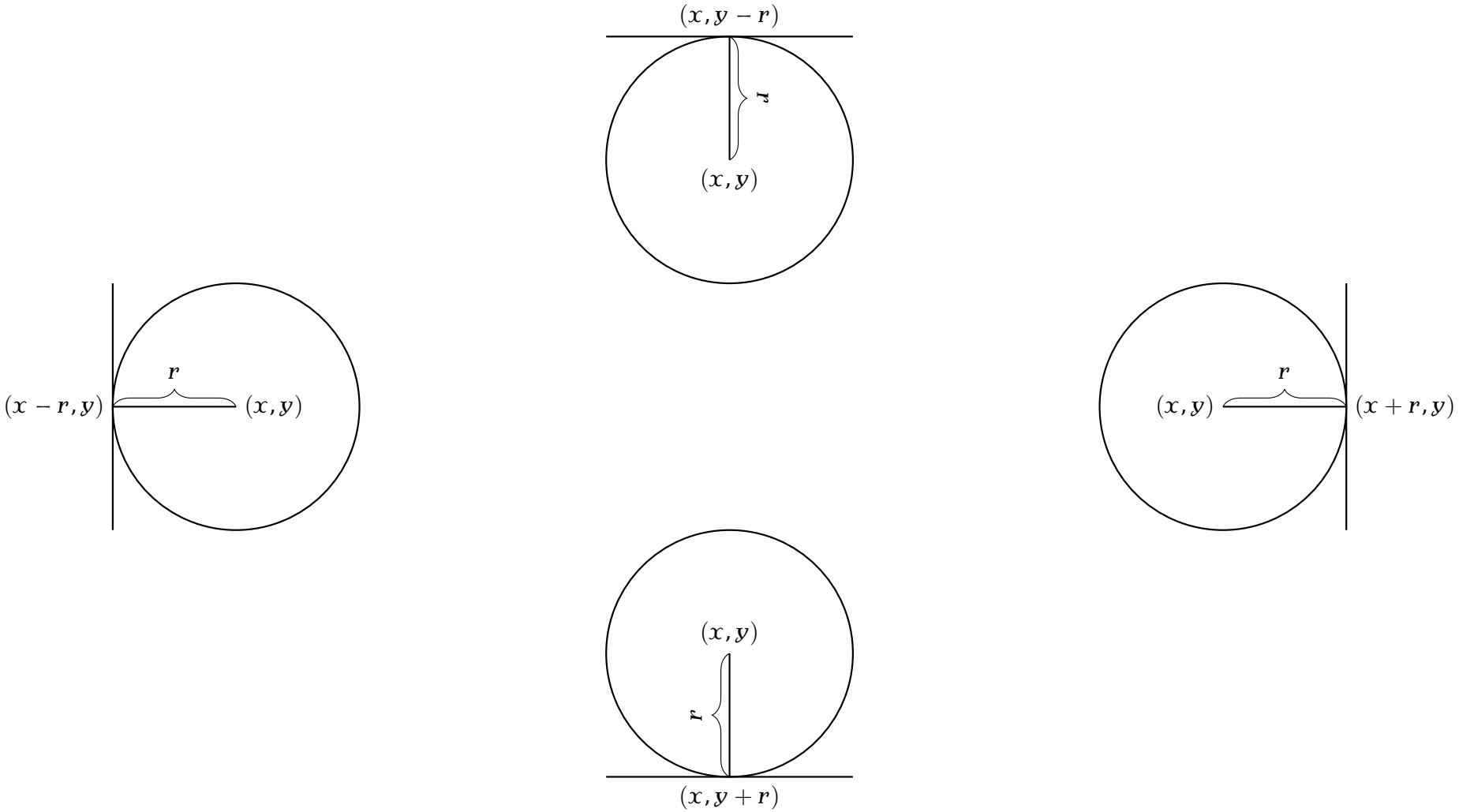


Animation — Ball

Hitting any of the screen edges forces the ball to change its movement direction. The collision is detected by comparing the current coordinates of the ball centre with abscissas or ordinates of horizontal or vertical edges. For example, verifying if the ball "touches" the left edge of the screen requires subtracting the ball radius length from the current abscissa of its middle and comparing the result with zero. If it is less than or equal zero then the ball has a contact with the edge. Similarly, in case of the right edge of the screen, the radius length is added to the current abscissa of the ball centre and the result is compared with the value of the `SCREEN_W` constant[2]. To detect collision of the ball with any of the horizontal edges, it is necessary to compare the value of similar expressions with zero or the value of the `SCREEN_H` constant. The next slide presents a figure that illustrates the overall concept of collision detection in the discussed program.

[2]For a more accurate collision detection, the result should be compared with the value of the constant minus one. However, such an accuracy is not required.

Animation — Ball



Animation — Ball

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

#define WIDTH 1366
#define HEIGHT 768
#define NUMBER_OF_PAGES 4
```

Animation — Ball

Comment

The beginning of this program is similar to the beginning of the previous one, but the header files required for using the PRNG are included.

Animation — Ball

```
struct ball_data {
    int x,y,dx,dy;
    unsigned char radius;

} ball;

BITMAP *pages[NUMBER_OF_PAGES];
```

Animation — Ball

Comment

In the program, aside from the array of pointers to the bitmap structures, the `ball` variable, which is a structure of the `ball_data` type, is declared. The members of this structure store data about the current coordinates of the ball centre, its velocity vector and the length of its radius.

Animation — Ball

```
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,NUMBER_OF_PAGES*height)) {
        allegro_message("set_gfx_mode(): %s\n",allegro_error);
        allegro_exit();
        return -1;
    }
    return 0;
}
```

Animation — Ball

Comment

The definition of `initialize()` function is similar to its equivalent from the previously presented program, but it additionally invokes functions that initialize the PRNG.

Animation — Ball

```
void set_ball(struct ball_data *ball)
{
    ball->radius = 20;
    ball->x = rand()%(SCREEN_W>>1)+ball->radius;
    ball->y = rand()%(SCREEN_H>>1)+ball->radius;
    ball->dx = 10;
    ball->dy = -10;
}
```

Animation — Ball

Comment

The `set_ball()` function initializes the structure that stores data about the ball. This variable is passed to the function by a pointer. The length of the ball radius is set to 20 pixels. The components of the velocity vector are set to 10 and −10 pixels respectively. Since the absolute values of the components are the same, the ball incidence angle is always 45°. Because of the signs of these values, the ball will initially move in the direction of the top-right corner of the screen. The coordinates of the centre of the ball are chosen randomly in such a way, that the ball is initially drawn entirely in the top-left quarter of the screen.

Animation — Ball

```
int create_pages_array(BITMAP *pages[NUMBER_OF_PAGES])
{
    int i;
    for(i=0; i<NUMBER_OF_PAGES; i++) {
        pages[i] = create_video_bitmap(SCREEN_W,SCREEN_H);
        if(pages[i]==NULL)
            return -1;
    }
    return 0;
}
```

Animation — Ball

```
void destroy_pages_array(BITMAP *pages[NUMBER_OF_PAGES])
{
    int i;
    for(i=0; i<NUMBER_OF_PAGES; i++)
        destroy_bitmap(pages[i]);
}
```

Animation — Ball

Comment

The functions presented in the two earlier slides are described in the previously discussed program.

Animation — Ball

```
void draw_ball(BITMAP *page, struct ball_data ball)
{
    clear_bitmap(page);
    circle(page,ball.x,ball.y,ball.radius,makecol(255,255,0));
}
```

Animation — Ball

Comment

The `draw_ball()` function draws a yellow circle to a bitmap, using the data from the structure passed by its second parameter. By the first parameter is passed the pointer to this bitmap. Before the circle is drawn, the colour of each bitmap pixel is set to black.

Animation — Ball

```
void update_ball_position(struct ball_data *ball)
{
    ball->x += ball->dx;
    ball->y += ball->dy;
    if(ball->x-ball->radius<=0||ball->x+ball->radius>=SCREEN_W)
        ball->dx = -ball->dx;
    if(ball->y-ball->radius<=0||ball->y+ball->radius>=SCREEN_H)
        ball->dy = -ball->dy;

}
```

Animation — Ball

Comment

The `update_ball_position()` function calculates the coordinates of the ball centre in the next frame of the animation. It uses data from the structure passed by its parameter. Additionally the function checks if the ball has hit any of the screen edges. If so, the ball has to bounce off the edge, which means that it has to change the direction of its movement. To this end, the following rules are applied:

- If the ball has a contact with any of the horizontal edges, the sign of the second component of the velocity vector will be reversed.
- If the ball has a contact with any of the vertical edges, the sign of the first component of the velocity vector will be reversed.

Animation — Ball

```
void animate_ball(struct ball_data ball, BITMAP *pages[NUMBER_OF_PAGES])
{
    int page_number = 0;
    clear_keybuf();
    while(key[KEY_ESC]==0&&key[KEY_Q]==0) {
        BITMAP *active_page = pages[page_number];
        draw_ball(active_page,ball);
        if(show_video_bitmap(active_page))
            return;
        update_ball_position(&ball);
        page_number = (page_number+1)%NUMBER_OF_PAGES;
    }
}
```

Animation — Ball

Comment

The `animate_ball()` function and the `animate_rectangle()` function from the previous program are very similar. The `while` loops inside these functions stop for the same reason. The difference is, that the `animate_ball()` function draws, with the help of the `draw_ball()` function, a circle instead of a rectangle. Moreover, the location of the circle in the next frame of the animation is calculated with the use of the `update_ball_position()` function.

Animation — Ball

```
int main(void)
{
    if(initialize(GFX_AUTODETECT_FULLSCREEN,WIDTH,HEIGHT)<0)
        return -1;
    if(create_pages_array(pages))
        return -1;
    set_ball(&ball);
    animate_ball(ball,pages);
    destroy_pages_array(pages);
    allegro_exit();
    return 0;
}
END_OF_MAIN()
```

Animation — Ball

Comment

Inside the `main()` function the functions defined earlier in the program are invoked. Before the function that creates the animation is called, the `set_ball()` function is invoked. It initializes the structure that stores information about the circle that represents the ball in the program.

Simple Textures

A pattern that covers a surface is called *a texture* in Computer Graphics. It turns out that complex, intriguing textures can be created with the help of simple mathematical expressions. The next program demonstrates some of these textures.

Simple Textures

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

#define WIDTH 1366
#define HEIGHT 768
```

Simple Textures

Comment

The program starts with preprocessor directives that include header files necessary for using the basic functions of the Allegro library and the keyboard support. They are followed by definitions of constants that define the computer display resolution.

Simple Textures

```
int initialize(int card, int width, int height)
{
    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;
}
```

Simple Textures

Comment

The definition of the `initialize()` function is the same as in the other presented programs that don't use the PRNG or the animation support.

Simple Textures

```
void wait_for_any_key(void)
{
    clear_keybuf();
    while(!keypressed())
        ;
}
```

Simple Textures

Comment

The `wait_for_key()` function has been described in the part of the lecture where the sine wave plotting program has been discussed.

Simple Textures

```
void draw_text(BITMAP *bitmap, FONT *font, const char *message)
{
    int red = makecol(255,0,0);
    textout_centre_ex(bitmap,font,message,SCREEN_W>>1,SCREEN_H>>1,red,-1);
}
```

Simple Textures

Comment

The `draw_text()` function displays in the middle of the screen a message in red. The content of this message is passed to the function by its last parameter.

Simple Textures

```
void or_draw(BITMAP* bitmap)
{
    int x,y;
    for(x=0; x<SCREEN_W; x++)
        for(y=0; y<SCREEN_H; y++) {
            int expression = (x|y)&255;
            int colour = makecol(expression,expression,expression);
            putpixel(bitmap,x,y,colour);
        }
}
```

Simple Textures

Comment

The `or_draw()` function draws on the screen a grayscale image consisting of recurring "tiles". Colours of the majority of the image pixels are bright. It is a consequence of the way the colour codes of these pixels are calculated. In this operation the bitwise *or* operator — `|` — is used. Its arguments are coordinates of a pixel in the image. The result is scaled-down and its final value ranges from 0 to 255. The scaling can be performed with the help of the remainder operator, but to speed up the calculations the bitwise *and* operator — `&` — is used instead. The second argument of this operator is the number 255. The scaled-down result is then used as the value of arguments passed to the `makecol()` function that returns the code of the pixel colour.

Simple Textures

```
void and_draw(BITMAP* bitmap)
{
    int x,y;
    for(x=0; x<SCREEN_W; x++)
        for(y=0; y<SCREEN_H; y++) {
            int expression = (x&y)&255;
            int colour = makecol(expression,expression,expression);
            putpixel(bitmap,x,y,colour);
        }
}
```

Simple Textures

Comment

The `and_draw()` function draws an image similar to the one created by the `or_draw()` function, but instead of the bitwise *or* it uses the bitwise *and* operator to calculate the colour of each pixel. The resulting image is darker then the previous one.

Simple Textures

```
void xor_draw(BITMAP* bitmap)
{
    int x,y;
    for(x=0; x<SCREEN_W; x++)
        for(y=0; y<SCREEN_H; y++) {
            int expression = (x^y)&255;
            int colour = makecol(expression,expression,expression);
            putpixel(bitmap,x,y,colour);
        }
}
```

Simple Textures

Comment

The `xor_draw()` function uses the bitwise *exclusive or* operator — `^` — to calculate the colour of each pixel. The resulting image is brighter than the one created with the use of the bitwise *and* operator, but darker than the one created with the use of bitwise *or* operator.

Simple Textures

```
void multiply_draw(BITMAP* bitmap)
{
    int x,y;
    for(x=0; x<SCREEN_W; x++)
        for(y=0; y<SCREEN_H; y++) {
            int expression = (x*y)&255;
            int colour = makecol(expression,expression,expression);
            putpixel(bitmap,x,y,colour);
        }
}
```

Simple Textures

Comment

The `multiply_draw()` function uses the multiplication operator to calculate the colour of each pixel on the screen. The resulting image is different than those generated by previously described functions. It is called the *moiré pattern*.

Simple Textures

```
void draw_sierpinski_triangle(BITMAP* bitmap)
{
    int x,y;
    int white = makecol(255,255,255);
    for(x=0; x<SCREEN_W; x++)
        for(y=0; y<SCREEN_H; y++) {
            if((x&y)==0)
                putpixel(bitmap,x,y,white);
        }
}
```

Simple Textures

Comment

The last function defined in the program (aside from the `main()` function) draws a fractal image called a Sierpiński Triangle. Its name originates from the surname of its discoverer, a Polish mathematician Wacław Sierpiński. There are many algorithms for drawing this fractal, but the one applied in this function is one of the simplest. If the output of bitwise *and* operator, which arguments are a pixel coordinates, is zero then the pixel colour will be set to white. Otherwise it will stay black. This rule is applied to all pixels on the screen.

Simple Textures

```
int main(void)
{
    if(initialize(GFX_AUTODETECT_FULLSCREEN,WIDTH,HEIGHT))
        return -1;
    or_draw(screen);
    draw_text(screen,font,"or");
    wait_for_any_key();
    and_draw(screen);
    draw_text(screen,font,"and");
    wait_for_any_key();
    xor_draw(screen);
    draw_text(screen,font,"xor");
    wait_for_any_key();
    multiply_draw(screen);
    draw_text(screen,font,"*");
    wait_for_any_key();
    clear(screen);
    draw_sierpinski_triangle(screen);
    wait_for_any_key();
    allegro_exit();
    return 0;
}
END_OF_MAIN()
```

Simple Textures

Comment

In the `main()` function all the other functions defined in the program are called. After each of them exits (with the exception of the last one) the name or the symbol of the operator used for drawing the image is displayed on the screen with the help of the `draw_text()` function. Then the program waits until the user presses any key. Before the function that draws the Sierpiński Triangle is invoked the screen is cleared with the use of the `clear()` function. After the fractal is drawn the program once again waits for the user to press any key. When the user does it the Allegro library is finalized and the program ends.

Questions

?

The End

Thank You for Your Attention!