Kielce University of Technology
Faculty of Electrical Engineering, Automatic Control and Computer Science

Basics of the computer graphics 1

Laboratory instruction 2

”Multiple buffering
Drawing primitives
Incremental algorithm for drawing a segment”

Authors:
Paweł Pięta, MSc
Daniel Kaczmarski, MSc

Translated by:
Konrad Sich, MSc

Kielce, 2025

1 Introduction

The aim of the classes is to discuss the multiple buffering technique and implement it in the game engine, as well as to implement drawing primitives, also using the incremental segment drawing algorithm presented in the instruction.

2 Multiple buffering

Multiple buffering is a technique of storing data using more than one data buffer. In the *producer-consumer* design pattern, it ensures that the consumer always sees complete (albeit possibly outdated) data, while at the same time the producer creates an updated version of it. Failure to use multiple caching would result in the consumer being in a situation where he or she would be working with partly old and partly updated data.

In computer graphics, the multiple buffering technique is used when displaying graphics on a computer screen. It involves using several image buffers of the same size. The buffers are displayed on the screen alternately, and the next animation frame is always drawn on an invisible buffer. Thanks to this, it is possible to avoid or significantly reduce image problems such as flickering, clipping or tearing, which, as a result, dramatically improves the smoothness of the displayed image.

The most commonly used versions of the multiple buffering technique in the context of computer graphics are:

- double buffering,
- triple buffering.

Double buffering, as the name suggests, uses two image buffers, while triple buffering uses three. Triple buffering can provide better performance because with double buffering, the program must wait until the current frame is copied to the graphics card's VRAM or replaced as the active screen buffer (*page flipping*) before drawing the next frame of the animation. This waiting period may last several milliseconds and during this time it is not possible to operate on any of the image buffers. With triple buffering, this delay can be eliminated because the program uses one active buffer, the so-called front buffer and two back buffers, which allows you to immediately start drawing the next frame of the animation.

3 Primitives

Primitives (*geometric primitives*) are simple geometric figures from which it is possible to build other, complex objects and more complicated structures. The simplest figures of this type in two-dimensional graphics are a point and a line segment. They also include:

curves, circles, triangles, rectangles and other polygons. Primitives can be drawn empty inside, or they can be filled with a given color or gradient.

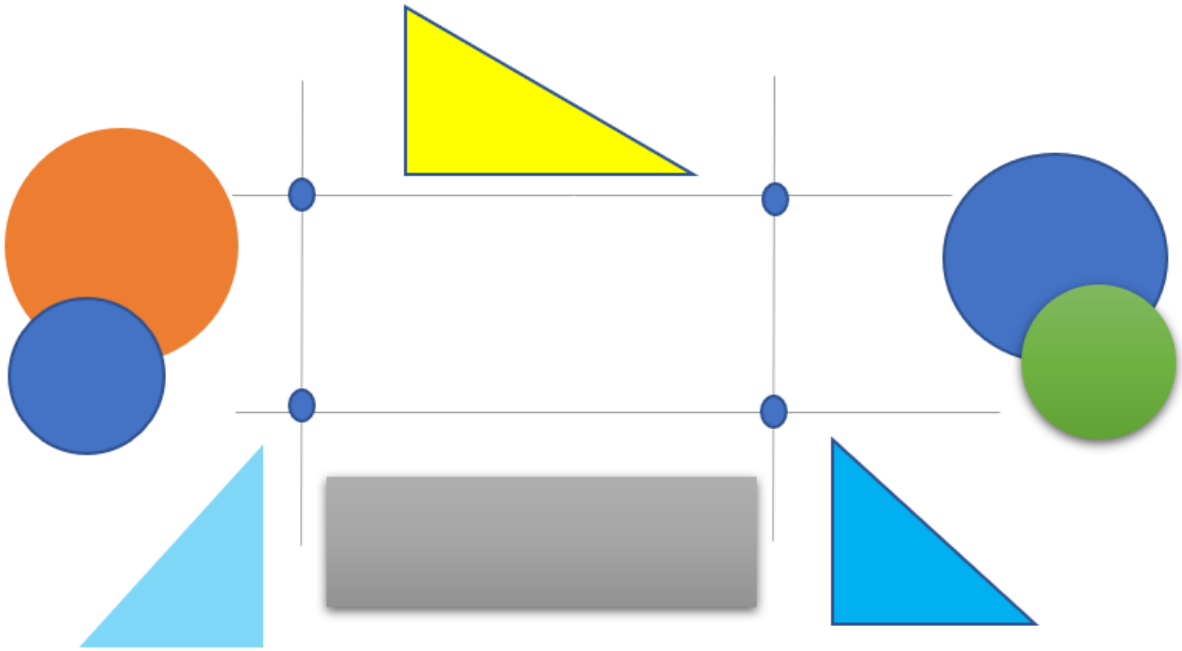

Figure 1: Primitives

For less complex video games, it is a good idea to equip the game engine with the ability to draw primitives. Such functionality can be implemented by an additional class, e.g. called `PrimitiveRenderer`, whose individual methods will be responsible for drawing various primitives. It is also worth providing this class with methods that will accept an array of coordinates as a parameter, which will enable more efficient drawing of more primitives in the loop.

4 Incremental algorithm for drawing a segment

The algorithm for drawing a segment using the incremental method can be presented in the following steps:

1. calculation of the slope of the section $m = \Delta y / \Delta x$,
2. incrementing `x` with step 1 starting from the left,
3. calculation of $y_i = m(x_i - x_0) + y_0$ for every x_i,
4. displaying a pixel at a point $(x_i, \text{Round}(y_i))$.

There are two problems with this algorithm:

- a large number of addition, subtraction and multiplication operations,
- incorrect drawing of the section for $|m| > 1$.

The first problem can be solved by significantly simplifying the equation, including eliminating multiplication. Due to the fact that:

$$x_{i+1} = x_i + 1$$
$$y_i = m(x_i - x_0) + y_0$$

the value of y_{i+1} can be calculated as follows:

$$y_{i+1} = m(x_{i+1} - x_0) + y_0 = m(x_i + 1 - x_0) + y_0 = m + m(x_i - x_0) + y_0 = m + y_i$$

The solution to the second problem is to replace all `x` $|m| > 1$ `y` values in the entire algorithm for $|m| > 1$:

1. recalculation of the slope of the section $m = \Delta x/\Delta y$,
2. increasing `y` with step 1 starting from the top,
3. calculation of $x_i = m(y_i - y_0) + x_0$ for every y_i,
4. displaying a pixel at a point (Round(x_i), y_i).

4.1 An example of how the algorithm works

The following coordinates of the beginning and end of the segment are given:

$$x_0 = 0 \ , \ y_0 = 2$$
$$x_1 = 10 \ , \ y_1 = 4$$

For the above coordinates, the values of Δx, Δy and m are:

$$\Delta x = x_1 - x_0 = 10 - 0 = 10$$
$$\Delta y = y_1 - y_0 = 4 - 2 = 2$$
$$m = \Delta y/\Delta x = 2/10 = 0.2$$

The above section drawn with the incremental algorithm is shown in the figure 2. The 1 table contains the real coordinates of individual pixels determined by the algorithm, while the 2 table contains the discrete coordinates , on the basis of which the episode was drawn.

x	0	1	2	3	4	5	6	7	8	9	10
y	2	2.2	2.4	2.6	2.8	3	3.2	3.4	3.6	3.8	4

Table 1: Example of operation of the incremental segment drawing algorithm – actual coordinates of individual pixels

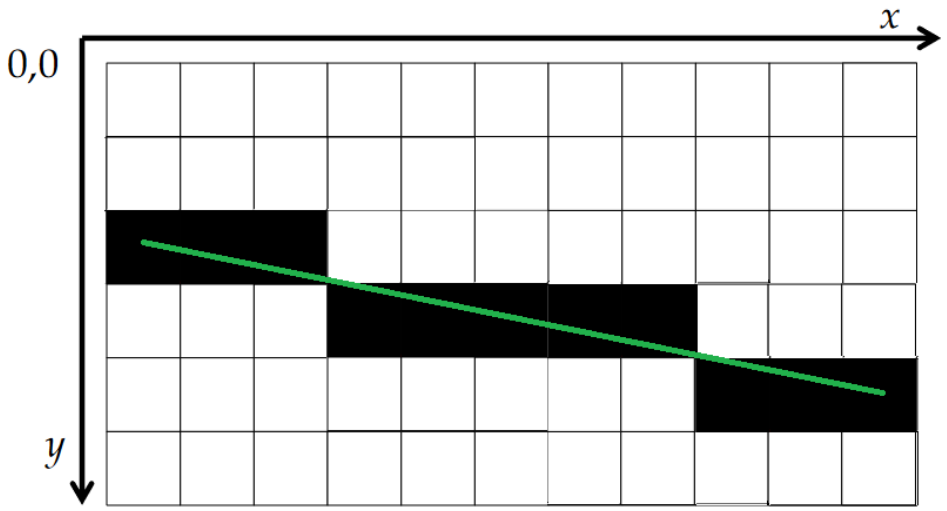

Figure 2: An example of the incremental segment drawing algorithm

x	0	1	2	3	4	5	6	7	8	9	10
y	2	2	2	3	3	3	3	3	4	4	4

Table 2: An example of the incremental segment drawing algorithm – discrete coordinates of individual pixels

5 Tasks

1. Make sure that the main game loop implemented in the `Engine` class uses a multi-buffering technique (double or triple buffering). If the graphics library you choose directly supports this technique, the scope of changes should be limited to extending the library's initialization method with an appropriate parameter enabling multiple buffering. Otherwise, it will be necessary to modify the main game loop so that it manages two or three image frame buffers on its own.

2. Implement the `PrimitiveRenderer`class, which will provide primitive drawing functionality. To do this, use the capabilities of the graphics library of your choice.

3. Extend the functionality of the `PrimitiveRenderer` class with another method that draws a line segment, which will use the incremental algorithm presented in the instruction. Verify the correct operation of the method for various cases of section inclination. Compare your results with those obtained using the default algorithm built into the graphics library.

4. Implement the `Point2D` class representing the coordinates of a point in 2D space. This class should have methods that enable:

   - reading the coordinates of a point,
   - modifying the coordinates of a point,

- drawing a point (use the `PrimitiveRenderer` class here).

5. Implement the `LineSegment` class to represent the line. The ends of a segment can be stored as two `Point2D` objects. This class should have methods that enable:

   - reading the coordinates of individual ends of the section,
   - modifying the coordinates of individual ends of the section,
   - drawing a segment (with a choice of drawing algorithm: default or incremental; use the `PrimitiveRenderer` class here).

6. Extend the functionality of the `PrimitiveRenderer` class with a method (or several methods) that allow you to draw an open or closed polyline. This method should accept as a parameter a set of points of the `Point2D` type or a set of segments of the `LineSegment` type (e.g. as an object of the `vector` class).