

Victor Ioan BÂCU

**ELEMENTS OF COMPUTER
ASSISTED GRAPHICS**

Laboratory works



UTPRESS
Cluj-Napoca, 2019
ISBN 978-606-737-377-6

Victor Ioan BĂCU

ELEMENTS OF COMPUTER ASSISTED GRAPHICS

Laboratory works



Editura UTPRESS
Cluj-Napoca, 2019
ISBN 978-606-737-377-6



Editura U.T.PRESS
Str. Observatorului nr. 34
C.P. 42, O.P. 2, 400775 Cluj-Napoca
Tel.:0264-401.999
e-mail: utpress@biblio.utcluj.ro
<http://biblioteca.utcluj.ro/editura>

Director: Ing. Călin D. Câmpean

Recenzia: Prof. dr. ing. Dorian GORGAN
Și. dr. ing. Teodor Traian ȘTEFĂNUȚ

Copyright © 2019 Editura U.T.PRESS

Reproducerea integrală sau parțială a textului sau ilustrațiilor din această carte este posibilă numai cu acordul prealabil scris al editurii U.T.PRESS.

ISBN 978-606-737-377-6

Preface

This book contains 12 laboratory works related to the computer graphics domain. Its main focus are the students from the second year of Computer Science department from the Computer Science and Automation faculty, Technical University of Cluj-Napoca, but it can be used by any engineer interested in this domain.

The content follows the structure of the Elements of Computer Assisted Graphics course taught at Technical University of Cluj-Napoca.

Each laboratory work is structured into three main sections. The first section presents the objectives and what is supposed to be learnt by students, the second section offers an overview of the theoretical background supporting the presented material. The last sections contains some assignments.

Cluj-Napoca,

04.06.2019

Author

Table of content

- Laboratory work 1 – SDL Introduction4**
 - 1 Objectives4
 - 2 Theoretical background4
 - 3 Assignments.....7
- Laboratory work 2 - Vectors8**
 - 1 Objectives8
 - 2 Theoretical background8
 - 3 Assignments.....10
- Laboratory work 3 - Matrices11**
 - 1 Objectives11
 - 2 Theoretical background11
 - 3 Assignments.....15
- Laboratory work 4 - Transformations16**
 - 1 Objectives16
 - 2 Theoretical background16
 - 3 Assignments.....20
- Laboratory work 5 – Applied transformations21**
 - 1 Objectives21
 - 2 Theoretical background21
 - 3 Assignments.....22
- Laboratory work 6 – Bresenham algorithm23**
 - 1 Objectives23
 - 2 Theoretical background23
 - 3 Assignments.....30
- Laboratory work 7 – Line clipping algorithms31**
 - 1 Objectives31
 - 2 Theoretical background31
 - 3 Assignments.....35

Laboratory work 8 – Rasterization pipeline	36
1 Objectives	36
2 Theoretical background	36
3 Assignments.....	41
Laboratory work 9 – Triangle rasterization algorithm.....	42
1 Objectives	42
2 Theoretical background	42
3 Assignments.....	44
Laboratory work 10 – Z-buffer algorithm.....	45
1 Objectives	45
2 Theoretical background	45
3 Assignments.....	48
Laboratory work 11 – Polygon clipping algorithms	49
1 Objectives	49
3 Assignment	54
Laboratory work 12 – Bezier curves	56
1 Objectives	56
2 Theoretical background	56
3 Assignments.....	57
References	58

Laboratory work 1 – SDL Introduction

1 Objectives

The objective of this laboratory is to describe briefly the SDL library and to exemplify the development of a basic SDL based application.

2 Theoretical background

2.1 Create a window using SDL

SDL (Simple DirectMedia Layer) library manages the access to graphics hardware (via OpenGL and Direct3D libraries) and also to audio, keyboard and mouse (independent on the underlying operating system). It supports Windows, Mac OS X and Linux and various other platforms. During this laboratory we will be using the C++ programming language but there are available bindings to other languages (such as C# or Python).

In order to create an SDL window the following steps are required:

1. Initialize the SDL library by calling the [SDL_Init\(\)](#) function with `SDL_INIT_VIDEO` as argument because we are using only the video subsystem of the SDL.

```
SDL_Init(SDL_INIT_VIDEO);
```

2. Create the window using the [SDL_CreateWindow\(\)](#) function. The arguments are the window title, position, width, height and some flags (for example to create a fullscreen window or a resizable window).

```
SDL_CreateWindow("SDL Hello World Example", SDL_WINDOWPOS_UNDEFINED,  
SDL_WINDOWPOS_UNDEFINED, WINDOW_WIDTH, WINDOW_HEIGHT,  
SDL_WINDOW_SHOWN | SDL_WINDOW_ALLOW_HIGHDPI);
```

Before closing the application we need to deallocate all the resources we created:

1. Destroy the window by calling the [SDL_DestroyWindow\(\)](#) function and passing as the argument the pointer to the window.

```
SDL_DestroyWindow(window);
```

2. Call the [SDL_Quit\(\)](#) function that is responsible to clean up all initialized subsystems (for our example we are using only the video subsystem).

```
SDL_Quit();
```

2.2 SDL Surface






In order to draw something onto the screen we need a canvas. In SDL the canvas can be represented by **SDL_Surface** or **SDL_Texture**. **SDL_Surface** is used in software rendering, instead **SDL_Texture** is used in hardware rendering, the main difference being the location of data (pixel) buffers. In this laboratory we will be using **SDL_Surface** because we will implement the graphics pipeline from scratch.

In SDL a surface is a structure containing a collection of pixels. Each window has attached such a surface and in addition we can create new surfaces and apply some operations on them (for instance we can load an image in a surface and copy the content to the window). The [SDL Surface](#) stores the format of the pixels, the dimension (width and height), a pointer to the actual pixel data and other relevant information. We will use a 32-bit pixel representation, in this situation we store 1 byte per channel (red, green, blue and alpha).

If we want to draw a rectangle we need to specify the starting position of the rectangle (the x and y coordinates) and the dimension (width and height). In order to specify the color we are using the **SDL_MapRGB()** function using the pixel format of the surface and the Red, Green and Blue values. The **SDL_FillRect()** function will update the surface with the specified rectangle and color.

```
SDL_Rect rectangleCoordinates = {100, 100, 200, 200};  
Uint32 rectagleColor = SDL_MapRGB(windowSurface->format, 255, 0, 0);  
SDL_FillRect(windowSurface, &rectangleCoordinates, rectagleColor);
```

For each color channel we specify the values between 0 and 255. Look at the following table for some color channel values, by combining the channel values we get different colors.

Color name	Red channel	Green channel	Blue channel	Color
White	255	255	255	
Red	255	0	0	
Green	0	255	0	
Blue	0	0	255	
Yellow	255	255	0	
Black	0	0	0	

2.3 Process events

In SDL the events could be things like pressing a keyboard key or mouse motion. All the events are stored in a queue in the order in which they occurred. By using the [SDL WaitEvent\(\)](#) function we get the next event from the queue.

2.3.1 Mouse pressed event

We can check if the left mouse button is pressed using the following piece of code, first we check the event type to be `SDL_MOUSEBUTTONDOWN` and the pressed button to be `SDL_BUTTON_LEFT`. The `SDL_GetMouseState()` function gets the coordinates of the mouse current position (x and y).

```
if(currentEvent.type == SDL_MOUSEBUTTONDOWN)
    if(currentEvent.button.button == SDL_BUTTON_LEFT)
        SDL_GetMouseState(&mouseX, &mouseY);
```

2.3.2 Mouse move event

We can check if the left mouse button is pressed while the mouse is moving using the following piece of code, first we check the event type to be `SDL_MOUSEMOTION` and the pressed button to be `SDL_BUTTON_LEFT`.

```
if(currentEvent.type == SDL_MOUSEMOTION)
    if(currentEvent.button.button == SDL_BUTTON_LEFT)
        SDL_GetMouseState(&mouseX, &mouseY);
```

2.3.3 Keyboard event

We can get the key pressed by using the following piece of code, first we check the event type to be `SDL_KEYDOWN` and then we process the desired keys.

```
if(currentEvent.type == SDL_KEYDOWN)
    switch(currentEvent.key.keysym.sym)
    {
        case SDLK_UP:
            //process UP key
            break;

        case SDLK_r:
            //process R key
            break;
        ...
        default:
            //default process
            break;
    }
```

2.4 Further reading

- Setting up SDL on various platforms – http://lazyfoo.net/SDL_tutorials/lesson01/index.php

- SDL Tutorials - http://lazyfoo.net/SDL_tutorials/

3 Assignments

Download and run the application from the laboratory website. Try to understand the basic example and then extend the application with the following functionality:

- Change the rectangle color by selecting the color channel using the R (for red), G (for green) and B (for blue) keys and by selecting the channel value by pressing the UP and DOWN keys.
- Interactively display a rectangle, the first set of coordinates is retrieved at left mouse button down event and the second set of coordinate is retrieved at every mouse move event (but only if the left button is still pressed).

Laboratory work 2 - Vectors

1 Objectives

The objective of this laboratory is to implement specific C++ classes for handling vectors.

2 Theoretical background

2.1 Vectors

In the field of computer graphics vectors are used to determine the angle between edges, the orientation of surfaces, the relative position of a point to a surface, in the computation of lighting models, and many other things.

We represent vectors by a list of numbers and graphically in a Cartesian coordinate system. We represent vectors as arrows and we name them by using bold letters. Geometrically a vector is described by direction and length. In 2D, a vector can be written as a combination of two not parallel vectors (with a length different from 0). A vector \mathbf{v} is represented by $\mathbf{v} = v_x\mathbf{x} + v_y\mathbf{y}$, where v_x, v_y are the Cartesian coordinates of the vector. We can write vectors horizontally and call them **row vectors** or we can write them vertically and call them **column vectors**. For the previous example we can write: $\mathbf{v} = \begin{bmatrix} v_x \\ v_y \end{bmatrix}$, or $\mathbf{v}^T = [v_x \quad v_y]$.

In computer graphics we are interested in 2D, 3D and 4D vectors and we refer to vector elements by:

- x, y (in 2D)
- x, y, z (in 3D)
- x, y, z, w (in 4D)

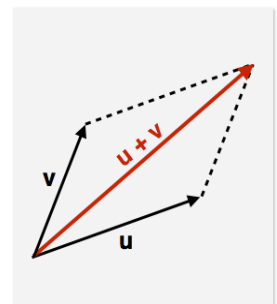
Vectors can be used to store displacement (the offset between two points) or locations (represented as displacement from a well known origin). Note however that locations are not vectors (we cannot add “Cluj” to “Bucharest”).

The difference between two points is a vector ($\mathbf{v} = Q - P$), and the sum between a point and a vector is a point ($\mathbf{v} + P = Q$).

2.2 Operations

2.2.1 Basic operations

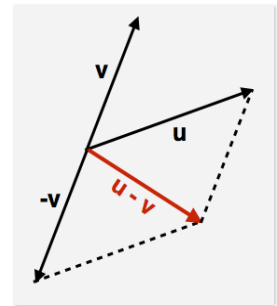
If two vectors have the same length and direction than they are equals.



The **length** is denoted by $\|\mathbf{v}\|$ and equals the square root of the sum of the square of vector elements. In the 2D case the length is $\|\mathbf{v}\| = \sqrt{v_x^2 + v_y^2}$.

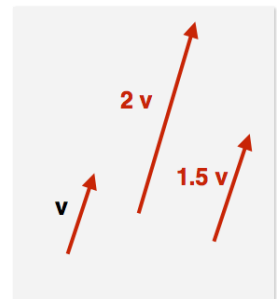
A **unit vector** is a vector with length equal to 1. The **zero vector** has the length equal to 0 (in this case the direction is undefined). We can **normalize** any nonzero vector by dividing the vector by its length (the new length equals 1).

We can **sum** two vectors by the parallelogram rule. This operation is commutative: $\mathbf{u} + \mathbf{v} = \mathbf{v} + \mathbf{u}$



We can represent also the **subtraction** of two vectors by a parallelogram.

The scalar multiplication is another operation which can be performed on vectors.

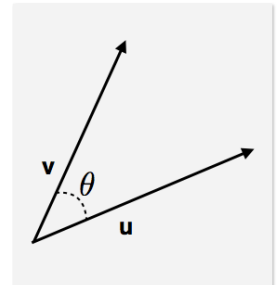


2.2.2 Dot product

The dot product of two vectors returns a scalar value related to the vectors' length and the angle between them.

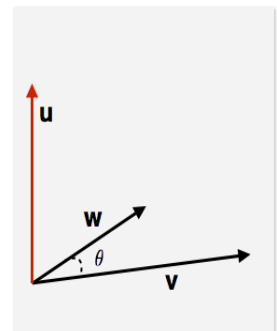
$$\mathbf{u} \cdot \mathbf{v} = \|\mathbf{u}\| \|\mathbf{v}\| \cos\theta$$

If two vectors \mathbf{u} and \mathbf{v} are represented in Cartesian coordinate then $\mathbf{u} \cdot \mathbf{v} = u_x v_x + u_y v_y$. Similar in 3D the dot product is $\mathbf{u} \cdot \mathbf{v} = u_x v_x + u_y v_y + u_z v_z$.



2.2.3 Cross product

In computer graphics we are mainly using this product only on 3D vectors, but the product can be generalized. The result is a vector perpendicular to the two vector. The length of the resulting vector is $\|\mathbf{v} \times \mathbf{w}\| = \|\mathbf{v}\| \|\mathbf{w}\| \sin\theta$. The direction of the vector generates a right-handed coordinate system.



$$\mathbf{v} \times \mathbf{w} = \begin{bmatrix} v_y w_z - v_z w_y \\ v_z w_x - v_x w_z \\ v_x w_y - v_y w_x \end{bmatrix}$$

3 Assignments

Download the source code from the web repository. You have to implement the methods inside the source files (vec2.cpp, vec3.cpp, and vec4.cpp). The header files contain the definition of classes and the methods that should be implemented.

Laboratory work 3 - Matrices

1 Objectives

The objective of this laboratory is to implement specific C++ classes for handling matrices.

2 Theoretical background

2.1 Matrix

A matrix is a collection of numbers arranged into rows and columns. In computer graphics we are using matrices to represent spatial transformations. The number of rows and columns are equal in this case, denoting a square matrix. In this laboratory we are using 3x3 and 4x4 matrices.

2.2 Identity matrix

The identity matrix is the matrix that has 1 on the diagonal and 0 elsewhere. For example, the 3x3 identity matrix is:

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

2.3 Matrix operations

2.3.1 Multiplication with a scalar

By multiplying a matrix \mathbf{M} with a scalar k we multiply each element of the matrix with the scalar value.

$$k\mathbf{M} = k \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} = \begin{bmatrix} km_{11} & km_{12} & km_{13} \\ km_{21} & km_{22} & km_{23} \\ km_{31} & km_{32} & km_{33} \end{bmatrix}$$

2.3.2 Addition of matrices

The matrices addition is done element by element, like in this example:

$$\begin{aligned} \mathbf{M} + \mathbf{N} &= \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} + \begin{bmatrix} n_{11} & n_{12} & n_{13} \\ n_{21} & n_{22} & n_{23} \\ n_{31} & n_{32} & n_{33} \end{bmatrix} \\ &= \begin{bmatrix} m_{11} + n_{11} & m_{12} + n_{12} & m_{13} + n_{13} \\ m_{21} + n_{21} & m_{22} + n_{22} & m_{23} + n_{23} \\ m_{31} + n_{31} & m_{32} + n_{32} & m_{33} + n_{33} \end{bmatrix} \end{aligned}$$

2.3.3 Multiplication with another matrix

In order to multiply two matrices, the number of columns in the first matrix must be the same with the number of rows of the second matrix.

$$\begin{bmatrix} a_{11} & \dots & a_{1m} \\ \vdots & & \vdots \\ a_{i1} & \dots & a_{im} \\ \vdots & & \vdots \\ a_{r1} & \dots & a_{rm} \end{bmatrix} \begin{bmatrix} b_{11} & \dots & b_{1j} & \dots & b_{1c} \\ \vdots & & \vdots & & \vdots \\ b_{m1} & \dots & b_{mj} & \dots & b_{mc} \end{bmatrix} = \begin{bmatrix} p_{11} & \dots & p_{1j} & \dots & p_{1c} \\ \vdots & & \vdots & & \vdots \\ p_{i1} & \dots & p_{ij} & \dots & p_{ic} \\ \vdots & & \vdots & & \vdots \\ p_{r1} & \dots & p_{rj} & \dots & p_{rc} \end{bmatrix}$$

The element p_{ij} is equal to:

$$p_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \dots + a_{im}b_{mj}$$

Matrix multiplication is not commutative ($\mathbf{M}_1\mathbf{M}_2 \neq \mathbf{M}_2\mathbf{M}_1$), but is associative and distributive:

$$(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$$

$$(\mathbf{A} + \mathbf{B})\mathbf{C} = \mathbf{AC} + \mathbf{BC}$$

2.3.4 Multiplication with a column vector

This operation is a particular case of the multiplication of two matrices operation.

$$\begin{bmatrix} a_{11} & \dots & a_{1m} \\ \vdots & & \vdots \\ a_{i1} & \dots & a_{im} \\ \vdots & & \vdots \\ a_{r1} & \dots & a_{rm} \end{bmatrix} \begin{bmatrix} b_{11} \\ \vdots \\ b_{j1} \\ \vdots \\ b_{m1} \end{bmatrix} = \begin{bmatrix} p_{11} \\ \vdots \\ p_{i1} \\ \vdots \\ p_{r1} \end{bmatrix}$$

The element p_{i1} is equal to:

$$p_{i1} = a_{i1}b_{11} + a_{i2}b_{21} + \dots + a_{im}b_{m1}$$

2.3.5 Transposition

The transpose of a matrix (denoted as \mathbf{M}^T) is the matrix where the columns and rows are switched. For example, for a 3x3 matrix:

$$\begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix}^T = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

The transpose of two matrices product is $(\mathbf{AB})^T = \mathbf{B}^T \mathbf{A}^T$.

2.3.6 Determinant of a matrix

For a 2x2 matrix the determinant is:

$$|\mathbf{M}| = \begin{vmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{vmatrix} = m_{11}m_{22} - m_{12}m_{21}$$

In order to compute the determinant, we have to find the cofactors of the matrix elements. The cofactor of each element of a square matrix is the determinant of a matrix (obtained by removing from the original matrix the row and column in which the element is in) multiplied by minus one in some cases. The sign of the cofactor can be determined by the following pattern:

$$\begin{bmatrix} + & - & + & \dots \\ - & + & - & \dots \\ + & - & + & \dots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

For a 4x4 matrix:

$$\mathbf{M} = \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{bmatrix}$$

the cofactor for the first row are

$$m_{11}^c = \begin{vmatrix} m_{22} & m_{23} & m_{24} \\ m_{32} & m_{33} & m_{34} \\ m_{42} & m_{43} & m_{44} \end{vmatrix}$$

$$m_{12}^c = - \begin{vmatrix} m_{21} & m_{23} & m_{24} \\ m_{31} & m_{33} & m_{34} \\ m_{41} & m_{43} & m_{44} \end{vmatrix}$$

$$m_{13}^c = \begin{vmatrix} m_{21} & m_{22} & m_{24} \\ m_{31} & m_{32} & m_{34} \\ m_{41} & m_{42} & m_{44} \end{vmatrix}$$

$$m_{14}^c = - \begin{vmatrix} m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \\ m_{41} & m_{42} & m_{43} \end{vmatrix}$$

The determinant equals the sum of the products of the elements (of any row or column) with their cofactors.

For a 3x3 matrix the determinant is:

$$\begin{aligned}
 |\mathbf{M}| &= \begin{vmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{vmatrix} \\
 &= m_{11} \begin{vmatrix} m_{22} & m_{23} \\ m_{32} & m_{33} \end{vmatrix} - m_{12} \begin{vmatrix} m_{21} & m_{23} \\ m_{31} & m_{33} \end{vmatrix} + m_{13} \begin{vmatrix} m_{21} & m_{22} \\ m_{31} & m_{32} \end{vmatrix}
 \end{aligned}$$

Similar, the determinant for a 4x4 matrix is:

$$\begin{aligned}
 |\mathbf{M}| &= \begin{vmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{vmatrix} = m_{11} \begin{vmatrix} m_{22} & m_{23} & m_{24} \\ m_{32} & m_{33} & m_{34} \\ m_{42} & m_{43} & m_{44} \end{vmatrix} - \\
 & m_{12} \begin{vmatrix} m_{21} & m_{23} & m_{24} \\ m_{31} & m_{33} & m_{34} \\ m_{41} & m_{43} & m_{44} \end{vmatrix} + m_{13} \begin{vmatrix} m_{21} & m_{22} & m_{24} \\ m_{31} & m_{32} & m_{34} \\ m_{41} & m_{42} & m_{44} \end{vmatrix} - \\
 & m_{14} \begin{vmatrix} m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \\ m_{41} & m_{42} & m_{43} \end{vmatrix}
 \end{aligned}$$

2.3.7 Inverse of a matrix

The inverse matrix (denoted as \mathbf{A}^{-1}) is the matrix for which $\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$. The inverse of two matrices product is $(\mathbf{A}\mathbf{B})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}$.

For a 4x4 matrix the inverse is:

$$\mathbf{M}^{-1} = \frac{1}{|\mathbf{M}|} \begin{bmatrix} m_{11}^c & m_{21}^c & m_{31}^c & m_{41}^c \\ m_{12}^c & m_{22}^c & m_{32}^c & m_{42}^c \\ m_{13}^c & m_{23}^c & m_{33}^c & m_{43}^c \\ m_{14}^c & m_{24}^c & m_{34}^c & m_{44}^c \end{bmatrix}$$

3 Assignments

Download the source code from the web repository. You have to implement the methods inside the source files (mat3.cpp and mat4.cpp). The header files contain the definition of classes and the methods that should be implemented.

Laboratory work 4 - Transformations

1 Objectives

This laboratory presents the key notions on 2D and 3D transformations (translation, scale, rotation).

2 Theoretical background

2.1 Defining 2D and 3D points

A 2D point is defined in a homogenous coordinate system by (x^*w, y^*w, w) . For simplicity in bi-dimensional systems the w parameter is set to 1. Therefore, the point definition is $(x, y, 1)$.

Another representation for the point is the following: $P = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$.

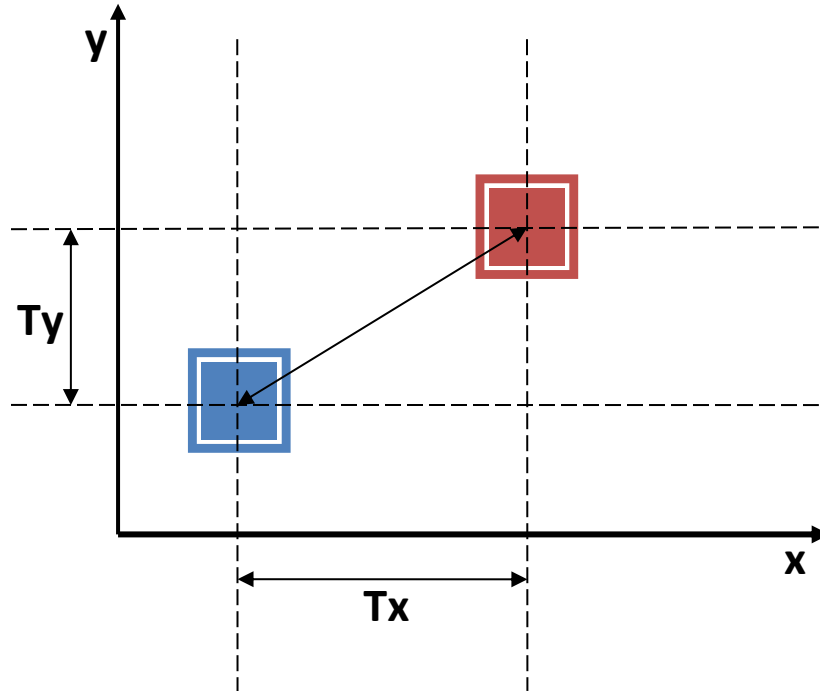
Similar in 3D we define points as (x^*w, y^*w, z^*w, w) and we represent the point as a column vector:

$$P = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

2.2 Transformations

2.2.1 2D Translation

The translation transformation is used to move an object (point) by a given amount.



The matrix for the translation operation is the following:

$$T = \begin{bmatrix} 1 & 0 & T_x \\ 0 & 1 & T_y \\ 0 & 0 & 1 \end{bmatrix}$$

where T_x and T_y represent the translation factors on x and y axes. If we apply the transformation to the 2D point, $P' = T * P$, we obtain the new coordinates for that point:

$$x' = x + T_x$$

$$y' = y + T_y$$

The matrix for the inverse transformation is the following:

$$T = \begin{bmatrix} 1 & 0 & -T_x \\ 0 & 1 & -T_y \\ 0 & 0 & 1 \end{bmatrix}$$

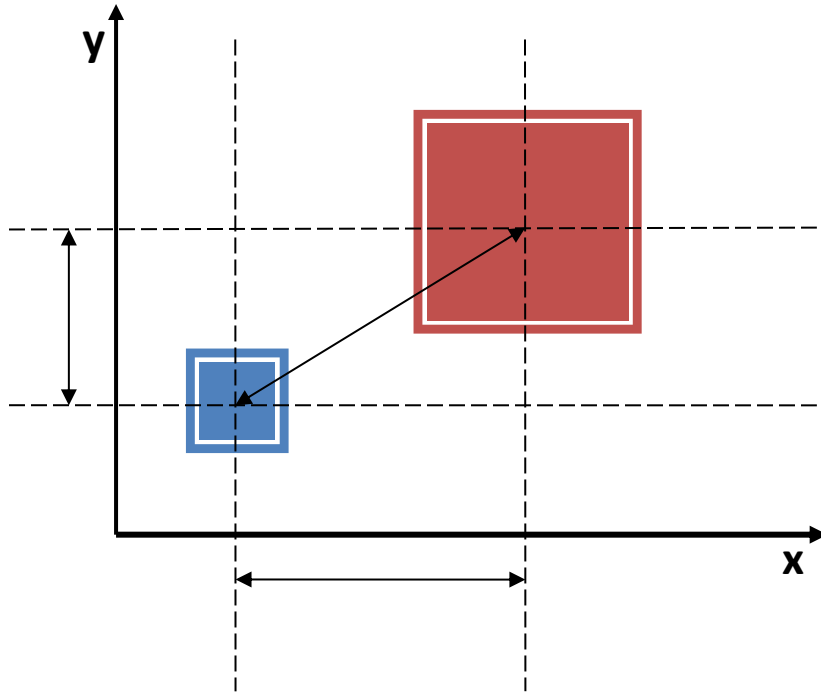
2.2.2 3D Translation

The only difference from the 2D case is that here we have one more coordinate and the transformation matrix will be 4x4.

$$T = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2.2.3 2D Scale

The scale transformation enlarges or reduces an object. The transformation is *relative to the origin*.



The matrix for the scale operation is the following:

$$S = \begin{bmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

where Sx and Sy represent the scale factors on x and y axes. If the Sx and Sy factors are equal then the scaling transformation is uniform. If the Sx and Sy factors are not equal then the scaling transformation is non-uniform. If we apply the transformation to the 2D point, $P' = S * P$, we obtain the new coordinates for that point:

$$x' = x * Sx$$

$$y' = y * Sy$$

If you set the scaling factors to ± 1 then you can reflect the original shape.

The matrix for the inverse transformation is the following:

$$S = \begin{bmatrix} 1/Sx & 0 & 0 \\ 0 & 1/Sy & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

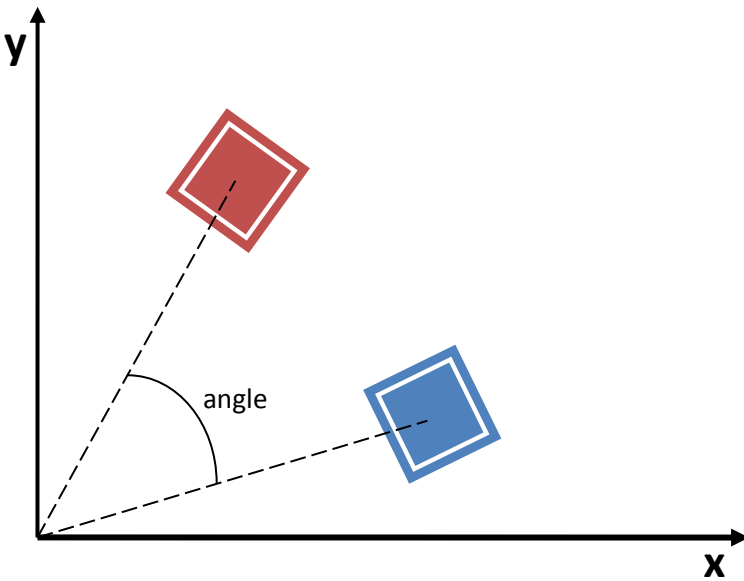
2.2.4 3D Scale

The 3D matrix representation of the scale transformation is the following:

$$S = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2.2.5 2D Rotation

This transformation rotates an object with a given angle. This transformation is also relative to the origin.



The matrix for the rotation operation is the following:

$$R = \begin{bmatrix} \cos \alpha & -\sin \alpha & 0 \\ \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

where α represents the rotation angle. If we apply the transformation to the 2D point, $P' = P * R$, we obtain the new coordinates for that point:

$$x' = x * \cos \alpha - y * \sin \alpha$$

$$y' = x * \sin \alpha + y * \cos \alpha$$

The matrix for the inverse transformation is the following:

$$R = \begin{bmatrix} \cos \alpha & \sin \alpha & 0 \\ -\sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

2.2.6 3D Rotation

We specify rotation in the 3D space independently on the x, y, and z axis. Rotation around the z axis is similar to the rotation in 2D (the z coordinate remains unchanged).

$$R_z = \begin{bmatrix} \cos \alpha & -\sin \alpha & 0 & 0 \\ \sin \alpha & \cos \alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \alpha & -\sin \alpha & 0 \\ 0 & \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y = \begin{bmatrix} \cos \alpha & 0 & \sin \alpha & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \alpha & 0 & \cos \alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3 Assignments

Download the source code from the web repository. You have to implement the methods inside the source file (transform.cpp). The header file (transform.h) contains the definition of methods that should be implemented.

Laboratory work 5 – Applied transformations

1 Objectives

The objective of this laboratory is to use the vectors and matrixes operations defined in previous laboratories, together with the transformation matrixes, in order to perform different transformations over a 2D figure.

2 Theoretical background

2.1 Using SDL application

In order to exemplify graphically the 2D transformations, we will use an application very similar to the one from Laboratory 1, based on the SDL (Simple DirectMedia Layer) library.

2.2 SDL Renderer

SDL_Renderer is a struct that handles all rendering. It is tied to a **SDL_Window** so it can only render within that **SDL_Window**. It also keeps track of the settings related to the rendering.

In order to create a renderer related to the application window, we will use the following code:

```
SDL_Renderer *windowRenderer;  
windowRenderer = SDL_CreateRenderer(window, -1, SDL_RENDERER_ACCELERATED);
```

There are several important functions tied to the **SDL_Renderer**:

1. **SDL_SetRenderDrawColor** sets the color that will be used in all drawing operations, until another call to the function is performed.

```
SDL_SetRenderDrawColor(renderer, r, g, b, a);
```

2. **SDL_RenderClear** clears the entire window display area using the current active color, previously set with **SDL_SetRenderDrawColor** function.

```
SDL_RenderClear(renderer);
```

3. **SDL_RenderPresent** will display everything drawn in the renderer to the screen. Until this function is called, all the drawing takes place in a hidden buffer that is not visible to

the user. The call to `SDL_RenderPresent` should be made only once, after all the drawing functions have been called.

```
SDL_RenderPresent(renderer);
```

2.3 Drawing a line

Drawing a line using SDL Renderer requires the following steps:

1. Define the color that will be used (example: blue)

```
SDL_SetRenderDrawColor(renderer, 0, 0, 255, 255);
```

2. Define the start and end points of the line. For this we will use `vec3` variables:

```
vec3 P1(100, 100, 1), P2(400, 100, 1);
```

3. Draw the line in the renderer

```
SDL_RenderDrawLine(windowRenderer, P1.x, P1.y, P2.x, P2.y);
```

4. Display the content of the renderer on the screen

```
SDL_RenderPresent(renderer);
```

3 Assignments

Download and run the application from the laboratory website. Try to understand the basic example and then extend the application with the following functionality:

- Include into the application your implementation files (.cpp).
- Define an initial rectangle with the top-left corner in $P_1(100, 100)$ and bottom-right corner in $P_2(400, 200)$.
- Rotate the rectangle around its center (diagonals intersection) by 10 degrees clockwise when `RIGHT_ARROW` is pressed and 10 degrees counterclockwise when `LEFT_ARROW` is pressed.
- Scale the rectangle having the top-left corner as a reference, using `UP_ARROW` and `DOWN_ARROW` keys.

Laboratory work 6 – Bresenham algorithm

1 Objectives

This laboratory highlights the Bresenham algorithms used for rendering some of the graphical primitives on a computer display. This paper begins by presenting some generic information about the algorithms and then exemplifies them for line and circle rasterization.

2 Theoretical background

The Bresenham algorithm for drawing lines onto a bi-dimensional space (like a computer display) is a fundamental method used in computer graphics discipline. The algorithm's efficiency makes it one of the most required methods for drawing continuous lines, circles or other graphical primitives. This process is called rasterization.

Each line, circle or other graphical primitives will be plotted pixel by pixel. Each pixel is described by a fixed (x, y) position in the bi-dimensional XOY space. The algorithm approximates the real line by computing each one of the line's pixel's position. Since the pixels are the smallest addressable screen elements in a display device, the algorithm approximation is good enough to "trick" the human eyes and to get the illusion of a real line. Figure 1a and Figure 1b shows the real line and the approximated line drawn over the pixel grid.

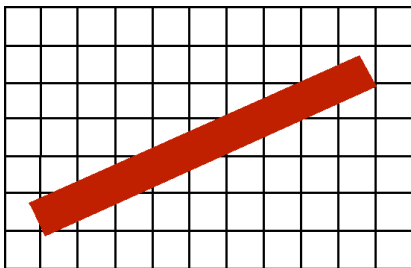


Figure 1a. Real line

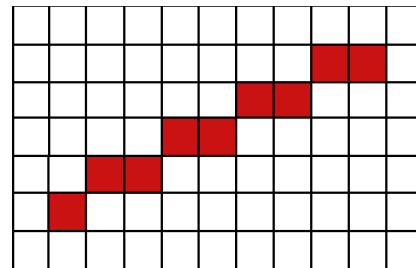


Figure 1b. Approximated line

Before moving on, it is worth to mention that both the line (1) and circle (2) can be mathematically described using the following equations:

$$y = m \cdot x + c \quad (1)$$

$$\text{with } m = \frac{y_2 - y_1}{x_2 - x_1}$$

$$(x - a)^2 + (y - b)^2 = R^2 \quad (2)$$

where:

- m is the line's slope;
- $(x_1, x_2), (y_1, y_2)$ are the two endpoints of the line segment;
- (a, b) represents the coordinates of the circle's center;

2.1 Bresenham's algorithm for line

For simplicity we will take into account a line segment with the slope from 0 to 1. Suppose the two endpoints of the line are $A(x_1, y_1)$ and $B(x_2, y_2)$. At this point we have to choose an initial point to start the algorithm. We can choose this point $(P(x_i, y_i))$ to be either A or B. Based on the starting position, we have eight possible choices to draw the next pixel of the line. This is due to the fact that each pixel is surrounded by 8 adjacent pixels.

Our example will consider only the case where we have two choice alternatives for the next pixel position (in other words this example will work only for the first octant of the trigonometric circle). For example, for the current point P we have the following drawing possibilities: $T(x_{i+1}, y_i)$ or $S(x_{i+1}, y_{i+1})$.

The decision criterion (Figure 2) for Bresenham's algorithm is based on the distance between the current point, P, and the real line segment. So the closest point (T or S) to the line segment will be chosen.

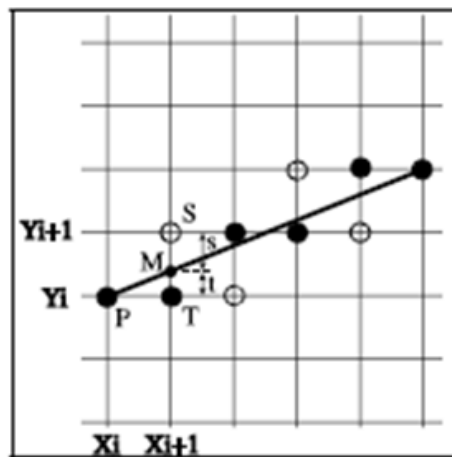


Figure 2. Decision criterion to choose the next line pixel that will plot on the screen display

The following paragraphs will describe the general steps of the Bresenham's algorithm in natural language rather than a programmatically one, because it is easier to understand.

a. Let us assume that we have to draw a line segment with the endpoints represented by $A(x_1, y_1)$ and $B(x_2, y_2)$. We translate the line segment with $(-x_1, -y_1)$ to place it on the XOY system origin.

b. Let $dx = x_2 - x_1$, $dy = y_2 - y_1$. The line that needs to be drawn can be described as

$$y = \frac{dx}{dy} x .$$

c. In this step we intend to compute the next line pixel, using the criterion mentioned above. From Figure 2, we can deduce that the closest point to the real line value is

$$T(x_{i+1}, y_i) . \text{ Based on this observation we could say that } M(x_{i+1}, x_{i+1} \cdot \frac{dy}{dx}) \quad (3).$$

$$\text{In other words } \begin{cases} t = y_m - y_i \\ s = y_{i+1} - y_m \end{cases} \Rightarrow t - s = 2 \cdot y_m - 2 \cdot y_i - 1 \quad (4).$$

Taking into account (3) and (4) we obtain $dx \cdot (t - s) = 2 \cdot dy \cdot x_{i+1} - 2 \cdot dx \cdot y_i - dx$. If the x coordinates of the line segment endpoints are in $x_1 < x_2$ relationship, then the sign of $t - s$ will coincide with the sign of $dx \cdot (t - s)$.

d. We can obtain the following recurrence relationship: $d_{i+1} = d_i + 2 \cdot dy - 2 \cdot dx \cdot (y_i - y_{i+1})$ if we consider that $dx \cdot (t - s) = d_{i+1}$.

The initial value of $d_i = 2 \cdot dy - dx$ is obtained for $x_0 = 0$ and $y_0 = 0$. We can conclude that:

- If $d_i \geq 0 \Rightarrow (t - s) \geq 0$ and the closest point to the real line segment is $S(x_{i+1}, y_{i+1})$.

Based on this observations we find that d_i value can be computed as

$$d_{i+1} = d_i + 2(dy - dx).$$

- If $d_i < 0 \Rightarrow (t - s) < 0$ and the closest point to the real line segment is $T(x_{i+1}, y_i)$.

Then the recurrence formula to compute d_i is $d_{i+1} = d_i + 2dy$.

The pseudo code for the Bresenham algorithm is described in the following paragraph, and it is based on the mathematical observations mentioned above.

```
//draw a line in the first octant
Algorithm Bresenham_line()
{
  //Initialize increments
  dx = abs(x2-x1);
  dy = abs(y2-y1);
  d = 2*dy-dx;
  inc1 = 2*dy;
  inc2 = 2*(dy-dx);
```

```

//Set the starting point, end point and current point
startX = x1;
startY = y1;
endX   = x2;
endY   = y2;
currentX = x1;
currentY = y1;

//Draw each pixel of the line
while (currentX < endX) {

//Draw the current pixel
DrawPixel(currentX, currentY);
increment currentX;

    if (d < 0) then {
        increment d using inc1;
    }
    else {
        increment currentY;
        increment d using inc2;
    }
}
}

```

2.2 Bresenham's algorithm for circle

Let's say we want to scan-convert a circle centered at (0,0) with an integer radius R (Figure 3). We'll see that the ideas we previously used for line scan-conversion can also be used for this task. First of all, notice that the interior of the circle is characterized by the inequality $D(x, y) : x^2 + y^2 - R^2 < 0$.

We'll use $D(x, y)$ to derive our decision variable.

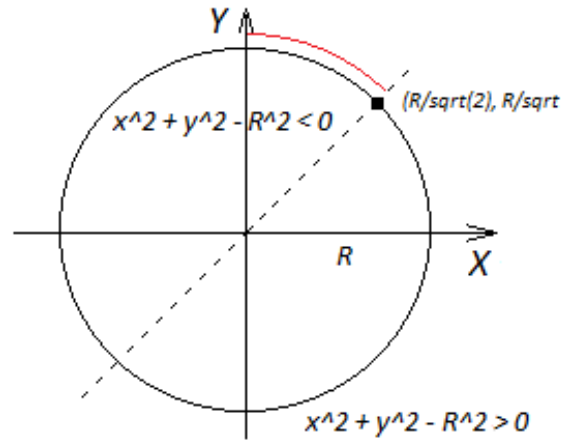


Figure 3. A circle and the description of its interior and exterior as two quadratic inequalities

Following the same approach as for the line segment representation, we'll first present the Bresenham's algorithm for the circle in natural language, describing for each step the general ideas behind it.

- a. First, let's think how to plot pixels close to the 1/8 of the circle marked red in Figure 3. The range of the x coordinate for such pixels is from 0 to $R\sqrt{2}$. We'll go over vertical scanlines through the centers of the pixels and, for each such scanline, compute the pixel on that line which is the closest to the scanline-circle intersection point (black dots in Figure 4). All such pixels will be plotted by our procedure.

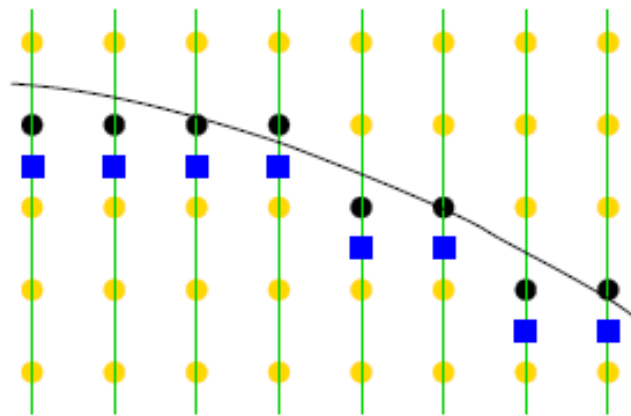


Figure 4. A circle and the description of its interior and exterior as two quadratic inequalities

- b. Notice that each time we move to the next scanline, the y -coordinate of the plotted point either stays the same or decreases by 1 (the slope of the circle is between -1 and 0). To decide what needs to be done, we'll use the decision variable, which will be the

value of $D(x, y)$ evaluated at the blue square (e.g. the midpoint between the plotted pixel and the pixel immediately below).

- c. The first pixel plotted is $(0, R)$ and therefore the initial value of the decision variable should be

$$D(0, R-0.5) = (R-0.5)^2 - R^2 = 0.25 - R$$

The y variable, holding the second coordinates of the plotted pixels, will be initialized to R . Let's think now at what happens after a point (x, y) is plotted. First, we'll pretend that we need to move the plotted point to the right (no change in y) and check if this keeps the decision variable negative (we don't want any blue squares outside the circle). If (x, y) is the last plotted point, the decision variable is $D(x, y-0.5)$. After we move to the right, it becomes $D(x+1, y-0.5)$. Simple arithmetic shows that it increases by $D(x+1, y-0.5) - D(x, y-0.5) = 2x+1$. If this increase makes it positive, we'd better move down by 1 pixel. This puts the blue square at $(x+1, y-1.5)$ and means that we need to increase the decision value by the previous $2x+1$ plus $D(x+1, y-1.5) - D(x+1, y-0.5) = 2-2y$.

- d. Clearly, to make the decision variable integer, we need to scale it by a factor of 4. Eight-way symmetry is used to go from 1/8-th of the circle to the full circle.

The pseudo code for the Bresenham's algorithm for circle is described below, based on the earlier made observations.

```
Algorithm Bresenham_circle ()
{
    currentY = R;
    d = 1/4 - R;

    //Go only one eighth of a circle
    for x = 0 to ceil(R/sqrt(2)) do {

        plot_points(currentX, currentY);
        increment the decision variable by 2x + 1;

        if (d > 0) then
        {
            increment the decision variable by 2 - 2y;
            decrement currentY;
        }
    }
}
```

You can find the **plot_points** function definition below:

```

Function plot_points (x, y)
{
    DrawPixel (x,y);
    DrawPixel (x,-y);
    DrawPixel (-x,y);
    DrawPixel (-x,-y);
    DrawPixel (y,x);
    DrawPixel (-y,x);
    DrawPixel (y,-x);
    DrawPixel (-y,-x);
}

```

2.3 Geometry rendering using SDL's API

The SDL 2.0 library provides a hardware accelerated rendering API for basic shapes such as rectangles, lines or points.

In order to use hardware accelerated rendering, we have to create a new **SDL_Renderer** for our SDL window.

```

//The window renderer
SDL_Renderer* renderer = NULL;
...
//Create renderer for window
renderer = SDL_CreateRenderer(window, -1, SDL_RENDERER_ACCELERATED);

```

To select a rendering color and to clear the window we can use **SDL_SetRenderDrawColor** and **SDL_RenderClear**.

```

//Clear screen
SDL_SetRenderDrawColor(renderer, 0xFF, 0xFF, 0xFF, 0xFF);
SDL_RenderClear(renderer);

```

Whenever a primitive is drawn, its color is the currently selected rendering color. To render points (pixels) we can use **SDL_RenderDrawPoint**.

```

//Draw current point
SDL_RenderDrawPoint(renderer, tmpCurrentX, tmpCurrentY);

```


3 Assignments

- Explore the implementation of Bresenham's algorithm for drawing lines provided in the laboratory's resources folder.
- Extend Bresenham's algorithm implementation for drawing lines to work in all octants of the trigonometric circle. The algorithm presented in this document, as well as the sample code cover only the first octant. **Note: Be careful when implementing Bresenham's algorithm. The SDL system of coordinates is different from the Cartesian system of coordinates.**
- Implement the function to draw circles using Bresenham's algorithm in the provided sample code.

Laboratory work 7 – Line clipping algorithms

1 Objectives

This laboratory presents the topic of clipping algorithms for graphical primitives. Two algorithms for line clipping are discussed, Cyrus-Beck and Cohen-Sutherland.

2 Theoretical background

2.1 Line clipping algorithms

Clipping algorithms are used to eliminate out-of-range values, meaning parts of segments or polygons which are outside the display area. In most cases the display area is defined as a rectangle and is called the clipping window. Relative to this clipping window the primitive (point, line or polygon) can be in the following relationship:

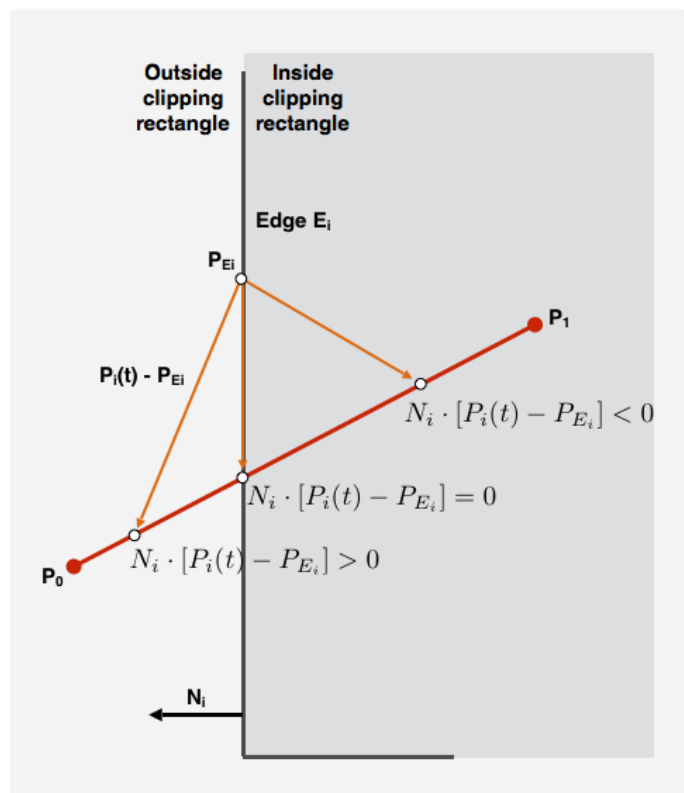
- Entirely inside the clipping window – no need to clip the primitive, continue to rasterize it.
- Entirely outside the clipping window – no need to clip the primitive, discard it.
- Intersects the clipping window – compute the intersection points and update the primitive, continue to rasterize it.

2.2 Cyrus-Beck algorithm

The Cyrus-Beck algorithm is used to clip a line segment against a convex polygon. The line segment is defined parametrically as:

$$P(t) = P_0 + (P_1 - P_0)t, \text{ where } t \in [0, 1]$$

For each edge E_i we compute the normal vector in such a way it points outward (see the figure on the right). From each edge we pick a point P_{Ei} . By computing the dot product $N_i \cdot (P(t) - P_{Ei})$ we can find the relative position of every point



on the line segment (defined for a particular value of the t parameter):

- if the value is negative the point lies in the inside halfplane
- if the value is positive the point lies in the outside halfplane
- if the value is zero the point is on the edge.

We are interested in the value of t for which $N_i \cdot (P(t) - P_{Ei}) = 0$.

First we substitute $P(t)$ and we regroup some terms:

$$N_i \cdot (P_0 + (P_1 - P_0)t - P_{Ei}) = 0$$

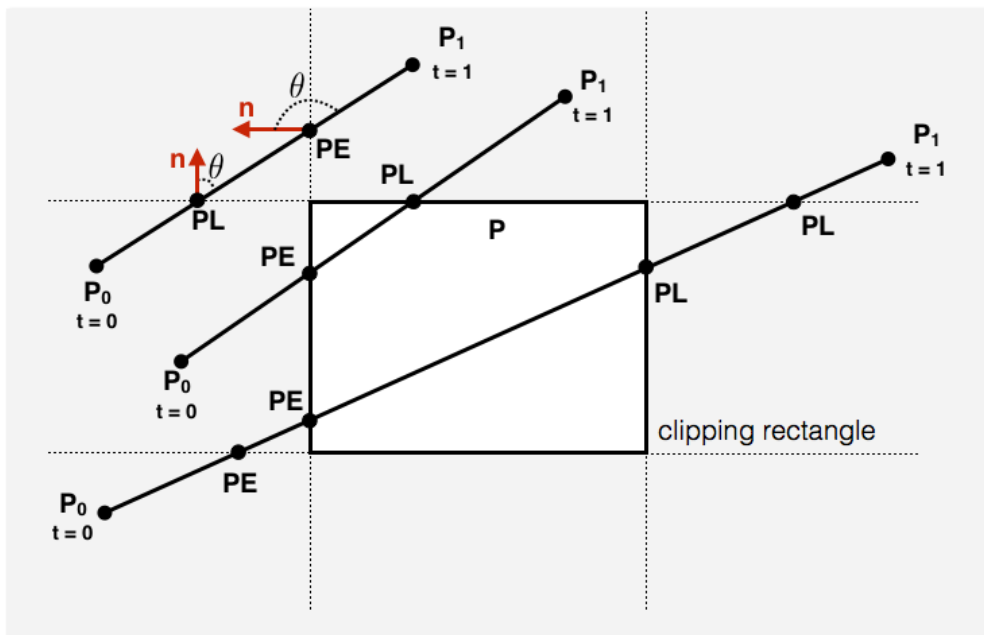
$$N_i \cdot (P_0 - P_{Ei}) + N_i \cdot (P_1 - P_0)t = 0$$

Let $D = P_1 - P_0$ be the vector from point P_0 to point P_1 . Then we can compute t as:

$$t = \frac{N_i \cdot (P_0 - P_{Ei})}{-N_i \cdot D}$$

We need to compute the value of t (for the intersection point) for each edge of the clipping window. The values of t outside the interval $[0, 1]$ are discarded. Each intersection is characterized by computing the angle between P_0P_1 and N_i as:

- “potentially entering” (PE) – the angle $> 90^\circ$
- “potentially leaving” (PL) – the angle $< 90^\circ$



2.2.1 Pseudocode

```
precalculate Ni and select a PEi for each edge;
if (P1 = P0)
    line degenerates to a point, so clip as a point;
else
    tE = 0; tL = 1;
    for (each candidate compute intersection with a clipping edge){
        if (Ni * D != 0){
            calculate t;
            use sign of Ni * D to categorize as PE (potentially entering) or PL (potentially leaving);
            if (PE)
                tE = max(tE, t);
            if (PL)
                tL = min(tL, t);
        }
    }
    if (tE > tL)
        return -1
    else
        return P(tE ) and P(tL ) as true clip intersections
```

2.3 Cohen-Sutherland clipping algorithm

2.3.1 Determine if a point P(x,y) is visible

Considering $P1(x_{min}, y_{min})$, and $P2(x_{max}, y_{max})$ the defining points of the visible area rectangle, the point $P(x,y)$ is visible only if the following conditions are met:

$$x_{min} \leq x \leq x_{max}$$

$$y_{min} \leq y \leq y_{max}$$

2.3.2 Determine if a segment is visible

In order to determine if a line segment is visible, we need slightly more complex algorithms. One idea would be to test the visibility of each point of the segment, before displaying it on the screen. But this method will require a lot of time and very many computations. The method can be easily improved by testing first the heads of the segment. If both these points are in the visible area, the entire segment will be visible. This case is called “simple acceptance”. On the same logic, if both points are outside and on the same side of the visible area, no part of the segment will be visible. This case is called “simple rejection”. In all the other cases we must use other algorithms to establish which part of the segment (if any) is visible.

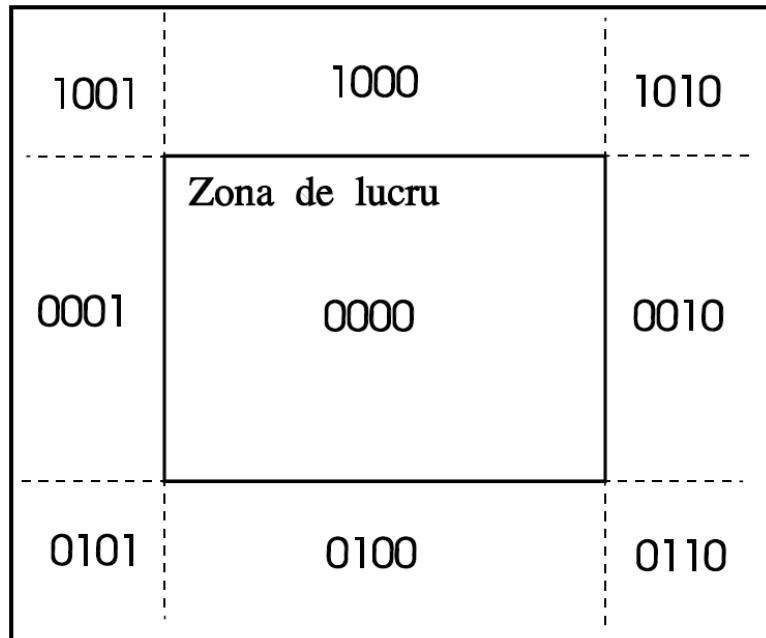
If a line segment cannot be included either in “simple acceptance” or “simple rejection” cases, then we have to compute its intersection points with the following lines:

$$y = y_{\max}, X = X_{\max}, y = y_{\min}, X = X_{\min}$$

and to eliminate the segments that are placed outside the visible area. As a result, we will obtain a new line segment. The algorithm is repeated until the resulted segment can be included in one of the “simple acceptance” or “simple rejection” cases.

The Cohen-Sutherland algorithm uses a four digits code to describe each head of the segment. The code has the following structure:

- the first digit is 1 if the point is above the visible area; otherwise is 0
- second digit is 1 if the point is under the visible area; otherwise is 0
- third digit is 1 if the point is on the right of the visible area; otherwise is 0
- fourth digit is 1 if the point is on the left side of the visible area; otherwise is 0



2.3.3 Pseudocode

```

repeat until FINISHED = TRUE
{
  COD1 = computeCScode(x1, y1) //compute the 4 digits code for P(x1, y1)
  COD2 = computeCScode(x2, y2) // compute the 4 digits code for P(x2, y2)
  RESPINS = SimpleRejection(COD1, COD2) //test for simple rejection case
  if RESPINS = TRUE
    FINISHED = TRUE
  else
    {
      DISPLAY = SimpleAcceptance(COD1, COD2) //test for simple acceptance case
      if DISPLAY = TRUE
        FINISHED = true
      else
        {
          if(P(x1, y1) is inside the display area)
            invert(x1,y1,x2,y2,COD1,COD2) //if P(x1, y1) is inside the display area, invert P(x1, y1) and P(x2,
            y2) together with their 4 digits CS codes

          if(COD1[1] = 1) and (y2 <> y1) //eliminate the segment above the display area
          {
            x1 = x1+(x2-x1)*(Ymax-y1)/(y2-y1)
            y1 = Ymax
          }
        }
    }
}

```


Laboratory work 8 – Rasterization pipeline

1 Objectives

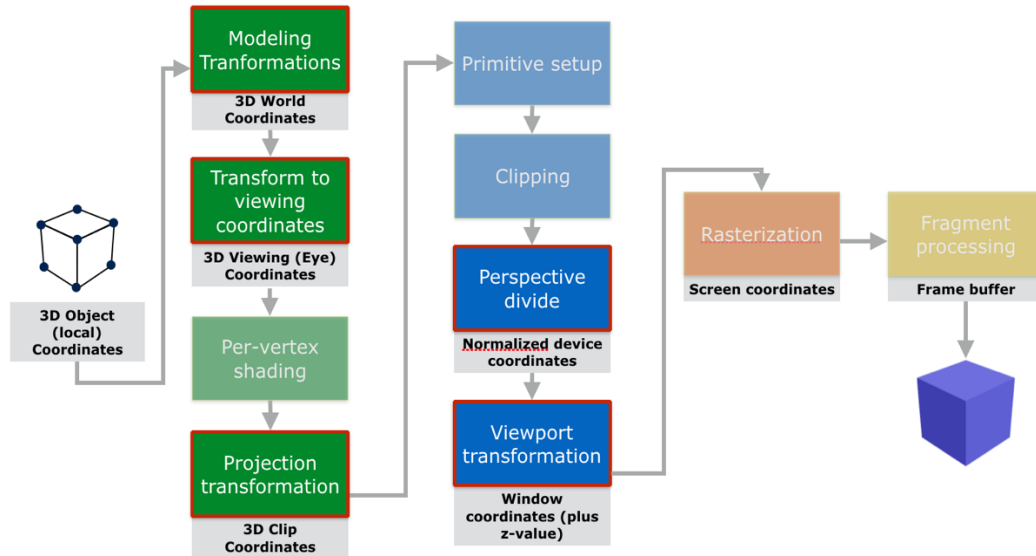
This laboratory presents the topic of viewing transformations used to map 3D locations (specified by x , y , and z coordinates) to 2D coordinates (specified by pixel coordinates).

2 Theoretical background

2.1 Visualization transformations

In order to map 3D coordinates to 2D coordinates we use a sequence of three transformations:

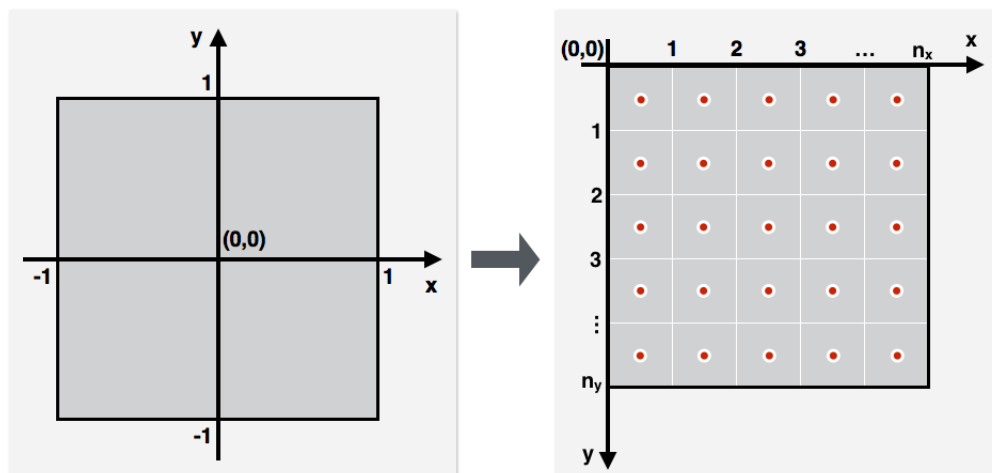
- **Camera transformation**
 - Used to place the camera at the origin and reposition all the other objects relative to the camera
 - This transformation depends only on the position and orientation of the camera
- **Projection transformation**
 - Used to project points from camera space
 - After the transformation all visible points will be in the range $[-1, 1]$
 - This transformation depends only on the type of projection (perspective or orthogonal)
- **Viewport transformation**
 - Used to map the unit image rectangle to the desired rectangle in pixel coordinates
 - This transformation depends only on the size and position of the output image



During these transformations we change the coordinate systems in which we specify the objects. Camera transformation changes coordinates from **world space** to **camera space**. The projection transformation moves points from **camera space** to the **canonical view volume** (here clipping is performed more efficiently). The last transformation, the viewport transformation maps the **canonical view volume** to **screen space**.

2.2 Viewport transformation

This transformation is used to map points from the canonical view volume (where the values are in the interval $[-1, 1]$) to the screen space (defined by the width and height of the resulting image). It is composed of several transformations, including translation, scale and reflection. The origin of the image is considered the top-left corner. For other specification of image space, the transformations could be different. The pixel center is considered to be at integer value plus 0.5 (both on x and y-axis).

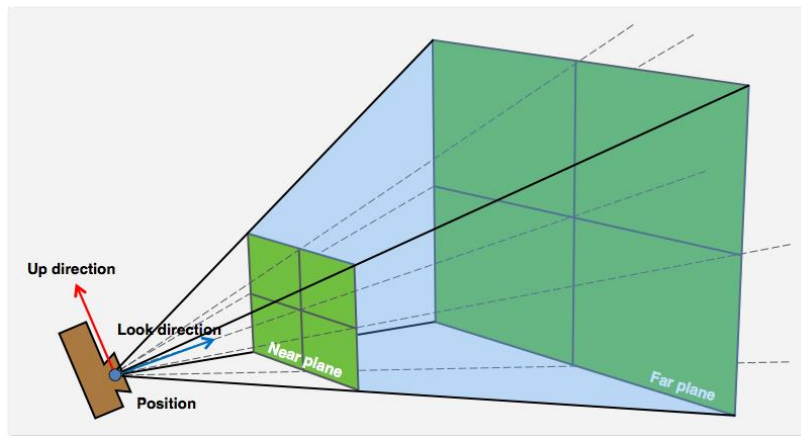


$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{S} = \begin{bmatrix} \frac{n_x}{2} & 0 & 0 & 0 \\ 0 & \frac{n_y}{2} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

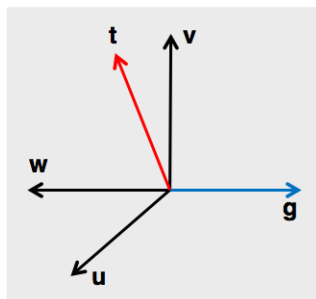
$$\mathbf{M}_{vp} = \mathbf{SMT} = \begin{bmatrix} \frac{n_x}{2} & 0 & 0 & \frac{n_x}{2} \\ 0 & -\frac{n_y}{2} & 0 & \frac{n_y}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2.3 Camera transformation

A virtual camera is defined by a set of attributes and parameters such as: Camera position, Orientation, Field of view, Type of projection (perspective or parallel projection), Depth of field, Focal distance, Tilt and offset of the camera lens relative to the camera body. For a basic camera we need to specify at least the position of the camera, its orientation and the field of view.



The camera is specified in world coordinates by the following parameters: eye position \mathbf{e} , gaze direction \mathbf{g} , view-up vector \mathbf{t} . From these parameters we need to define a coordinate system \mathbf{uvw} .



$$\mathbf{w} = -\frac{\mathbf{g}}{\|\mathbf{g}\|}$$

$$\mathbf{u} = \frac{\mathbf{t} \times \mathbf{w}}{\|\mathbf{t} \times \mathbf{w}\|}$$

$$\mathbf{v} = \mathbf{w} \times \mathbf{u}$$

We need to align the two different coordinate systems (u, v, w axes with the x, y, z axes) using the following transformation matrix:

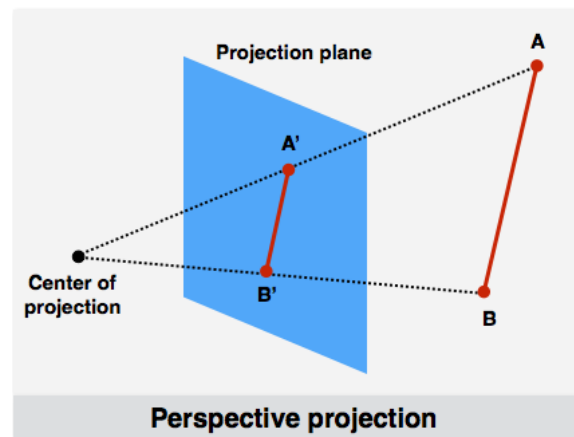
$$\mathbf{M}_{cam} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2.4 Projection transformation

Two types of projections are discussed in this document, perspective and parallel projections.

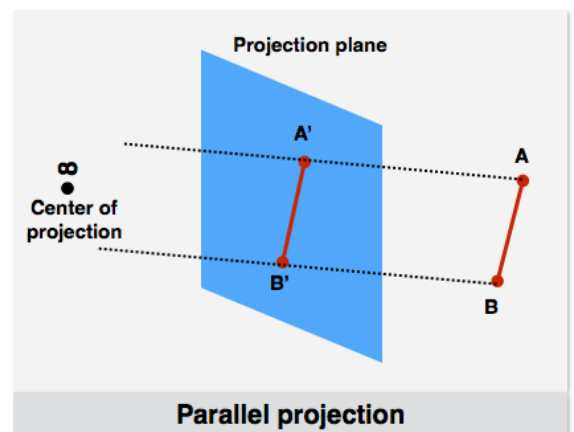
Perspective projection:

- Projection rays converge into the center of projection (location of the viewer)
- Objects appear smaller with the increase of distance from the center of projection (eye of observer)
- Lines parallel to the projection plane remain parallel
- Lines which are not parallel to the projection plane converge to a single point (vanishing point)



Parallel projection:

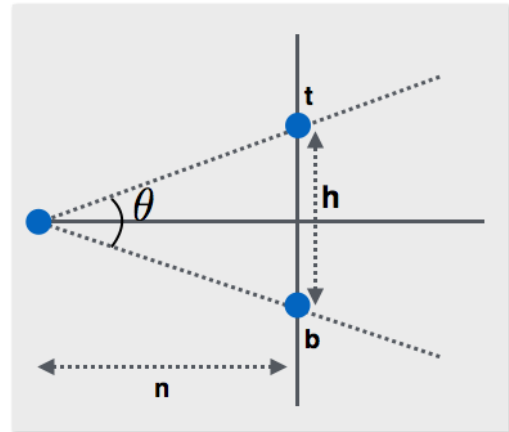
- Parallel projection rays
- Convergence point at infinity
- Viewer's position at infinity
- Parallel lines remain parallel



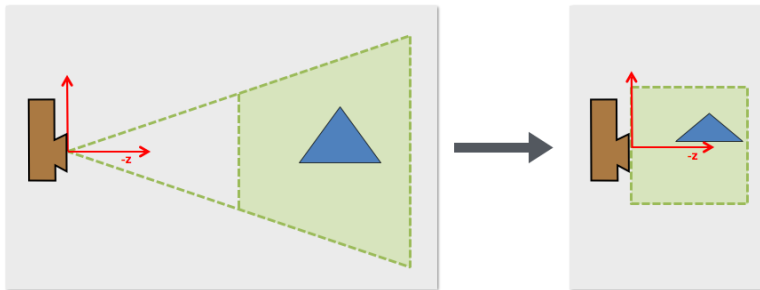
$$M_{per} = \begin{bmatrix} \frac{1}{aspect \tan \frac{\theta}{2}} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan \frac{\theta}{2}} & 0 & 0 \\ 0 & 0 & \frac{f+n}{n-f} & \frac{2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$\tan \frac{\theta}{2} = \frac{h}{2n}$$

$$aspect = \frac{width}{height} = \frac{r-l}{t-b}$$



Before displaying the vertices, we need to perform an operation called perspective divide.



The near and far planes are defined in camera coordinates. Because the camera is location in origin and is oriented in the negative z direction than the near and far values should be negative, with $n > f$.

$$P = M_p \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} xd \\ yd \\ zd \\ z \end{bmatrix} \xrightarrow{\text{After perspective divide}} \begin{bmatrix} \frac{d}{z}x \\ \frac{d}{z}y \\ d \\ 1 \end{bmatrix}$$

2.5 Combining transformations

```

construct Mvp
construct Mper
construct Mcam
M = Mvp * Mper * Mcam
for each line segment(ai,bi) do
    p = Mai
    q = Mbi
    drawline(xp/wp,yp/wp,xq/wq,yq/wq)

```

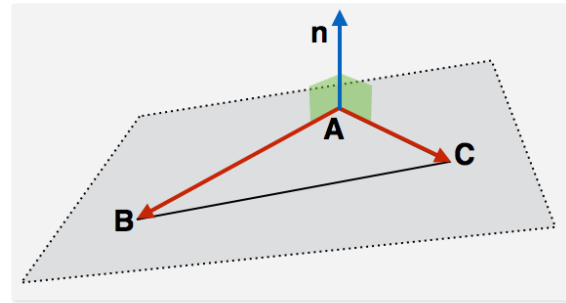
2.6 Compute and display the normal vector of a triangle

In order to compute the normal vector, we can use the following formula, based on the cross product between two vectors. The last step is to normalize the resulting vector in order to get a normalized vector.

$$\mathbf{n} = (B - A) \times (C - A)$$

The pseudocode for displaying the normal vector:

```
displayNormalVector()
  centerPoint = compute the center point of triangle
  normalVector = compute the normal vector
  secondPoint = centerPoint + normalVector * offset
  drawLine(centerPoint, secondPoint)
```

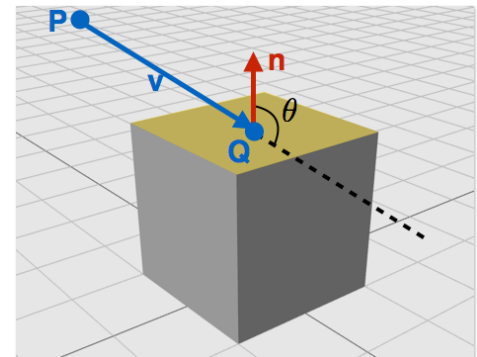


2.7 Back-face culling

Depending on the relative position of the camera to the object's triangles we can identify the visible triangles.

Relative position of a point **P** (camera position) against a plane (triangle):

- $\theta > 90^\circ$ then P is in front of the plane
- $\theta = 90^\circ$ then P is on the plane
- $\theta < 90^\circ$ then P is on back of the plane



The angle can be computed using the dot product between the two vectors, **v** and **n**.

2.8 Basic clipping in homogeneous coordinates

We can clip the points based on the following clipping planes:

$$\begin{aligned} -P.w &\leq P.x \leq P.w \\ -P.w &\leq P.y \leq P.w \\ -P.w &\leq P.z \leq P.w \end{aligned}$$

3 Assignments

- Download the source code and implement the specified methods in order to be able to display a 3D object, cull back-faces and display normal vector.

Laboratory work 9 – Triangle rasterization algorithm

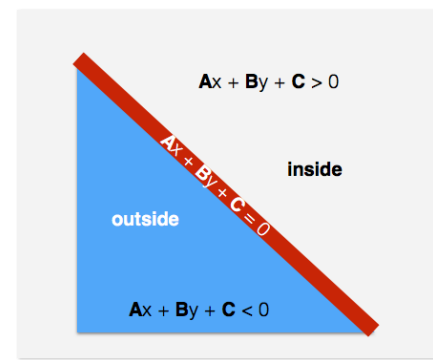
1 Objectives

This laboratory presents the topic of triangle rasterization using barycentric coordinates.

2 Theoretical background

2.1 Triangle definitions

We define each triangle by edges and we compute the edge equations such that the negative halfplane to be on the triangle's exterior. We start from the general implicit form of a line (in 2D) $Ax + By + C = 0$. The implicit form of a line passing through points $A(x_a, y_a)$ and $B(x_b, y_b)$ is: $(y_a - y_b)x + (x_b - x_a)y + x_a y_b - x_b y_a = 0$.

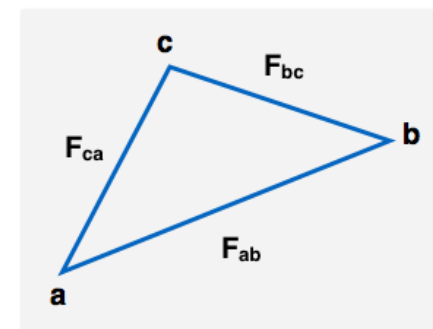


For a triangle define by vertices $a(x_a, y_a)$, $b(x_b, y_b)$ and $c(x_c, y_c)$ we have the following edge equations:

$$F_{ab} = (y_a - y_b)x + (x_b - x_a)y + x_a y_b - x_b y_a = 0$$

$$F_{bc} = (y_b - y_c)x + (x_c - x_b)y + x_b y_c - x_c y_b = 0$$

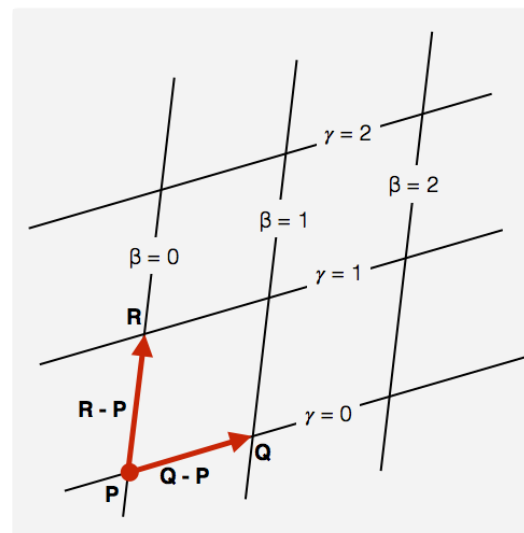
$$F_{ca} = (y_c - y_a)x + (x_a - x_c)y + x_c y_a - x_a y_c = 0$$



2.2 Barycentric coordinates

Any point p is a linear combination of points P , Q , and R :

$$\begin{aligned} p &= P + \beta(Q - P) + \gamma(R - P) \\ &= (1 - \beta - \gamma)P + \beta Q + \gamma R \\ &= \alpha P + \beta Q + \gamma R \end{aligned}$$



For triangles we need that:

$$\beta + \gamma \leq 1$$

$$\beta \geq 0$$

$$\gamma \geq 0$$

α, β, γ are called the **barycentric coordinates**.

Barycentric coordinates describe a point p as an affine combination of the triangle vertices:

$$p = \alpha P + \beta Q + \gamma R, \text{ where } \alpha + \beta + \gamma = 1$$

For any point p inside the triangle specified by vertices $a, b,$ and c :

$$0 < \alpha < 1$$

$$0 < \beta < 1$$

$$0 < \gamma < 1$$

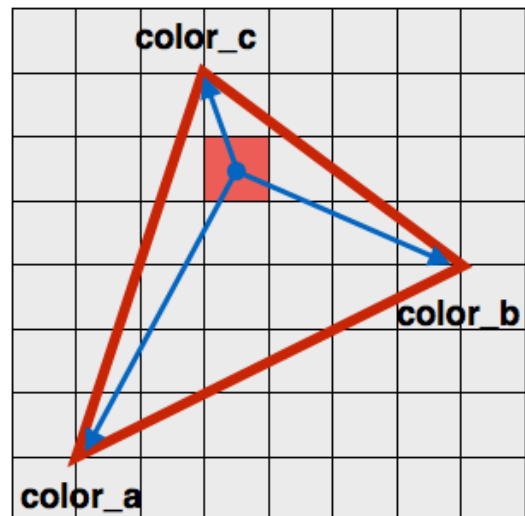
A point is on an edge if one barycentric coordinate is 0, and a point is on a vertex if two barycentric coordinates are 0.

For a triangle the barycentric coordinates for point p are:

$$\alpha = \frac{F_{bc}(x, y)}{F_{bc}(x_a, y_a)}$$

$$\beta = \frac{F_{ac}(x, y)}{F_{ac}(x_b, y_b)}$$

$$\gamma = 1 - \alpha - \beta$$



2.3 Triangle rasterization pseudocode

```
void triangleRasterization(vertices v[3]){
    bbox b = findBoundingBox(v);
    foreach pixel(x, y) in b
    {
        compute alpha, beta, gamma;
        if(0 < alpha < 1 and
```

```
    0 < beta < 1 and
    0 < gamma < 1)
  {
    color = color_a * alpha + color_b * beta + color_c * gamma;
    drawPixel(x, y) with color;
  }
}
```

3 Assignments

- Extend the implementation from the previous laboratory to add the functionality of displaying the 3D object filled with color.

Figure 1 – Frame-buffer and z-buffer for drawing a red triangle with $z = -5$

When a new primitive is rasterized, each of its resulting candidate pixels' depth is compared with the existing depth value in the z-buffer. The final pixel is updated only if the candidate pixel is closer to the center of projection than the value already in the buffer. If the pixel is updated, the existing depth value in the z-buffer is replaced by the new pixel's depth.

For example, when drawing a yellow triangle parallel with the Z axis and situated at a depth of $z = -7$, if there are no previously processed primitives, the resulting frame-buffer and z-buffer are the ones illustrated in Figure 2.

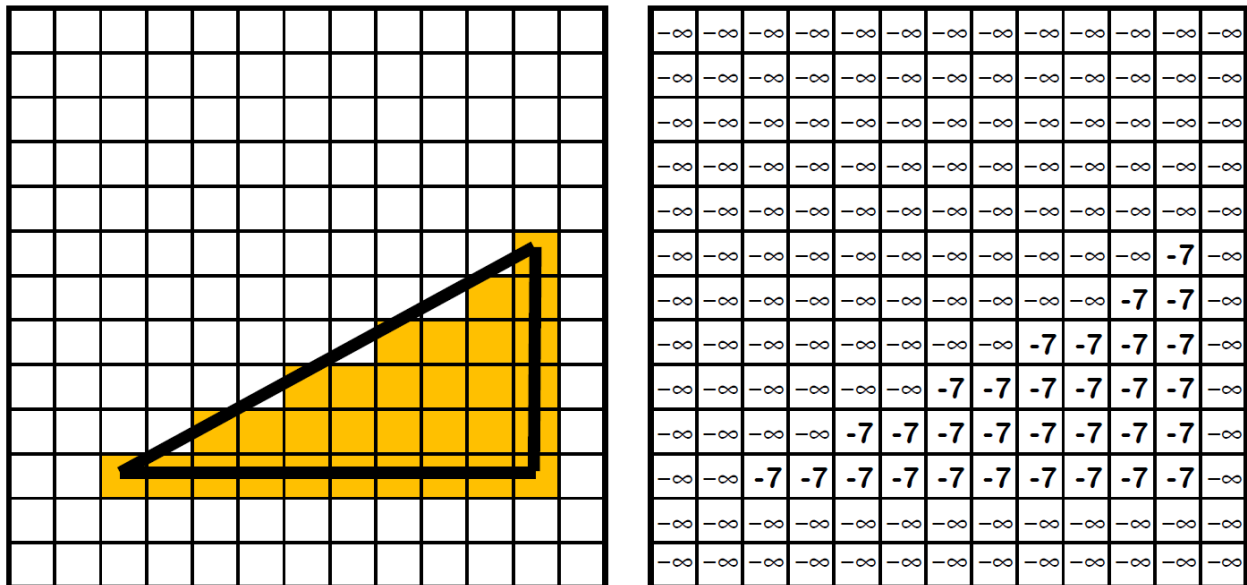


Figure 2 - Frame-buffer and z-buffer for drawing a yellow triangle with $z = -7$

However, if the yellow triangle is rasterized after the red triangle, then the contents of the frame-buffer and the z-buffer are as illustrated in Figure 3.

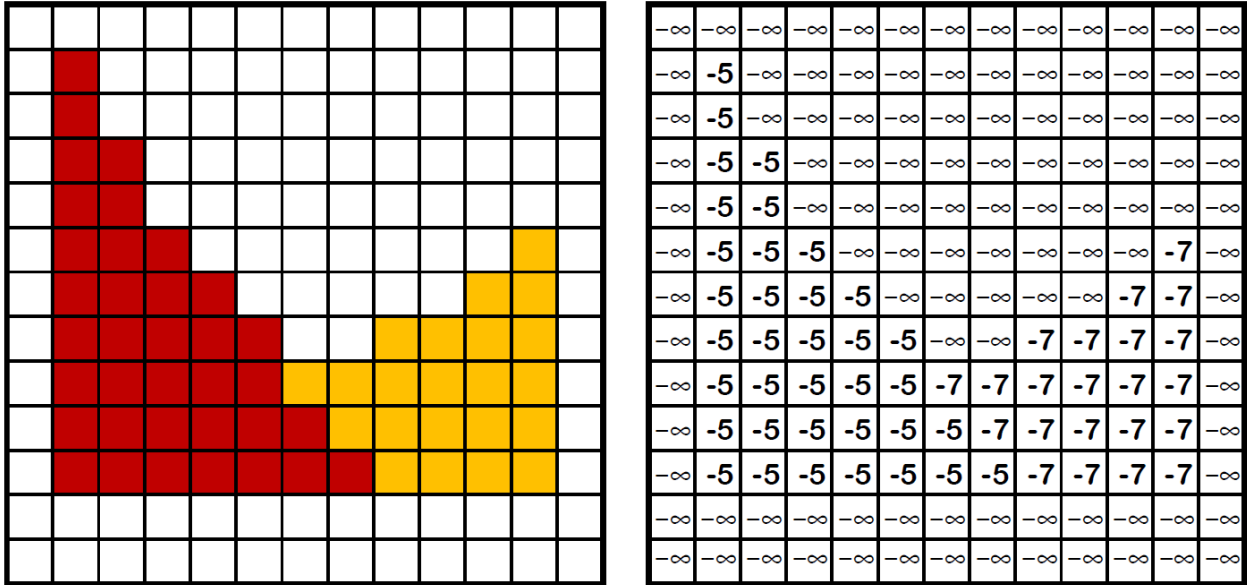


Figure 3 - Frame-buffer and z-buffer for drawing a red triangle with $z = -5$ and a yellow triangle with $z = -7$

The second processed primitive only updates the frame-buffer on the pixel positions for which the candidate depth is closer to the center of projection than the existing depth. The other positions remain unchanged.

2.1 Coloring the bunny based on depth

In order to obtain a better visual understanding of the importance of hidden surface removal, we will use a depth-based coloring scheme on the bunny from Laboratory work 9. The algorithm will compute each vertex' color based on its original Z coordinate (in object space) using the formula:

$$Color_{RGB} = (depthCoeff, depthCoeff, depthCoeff),$$

where

$$depthCoeff = 255 * \left(1 - \frac{vertex_z - min_z}{max_z - min_z}\right)$$

$Vertex_z$ is the current vertex' Z coordinate, max_z and min_z are the maximum and the minimum Z values among all of the bunny's vertices.

The result should resemble the image in Figure 4.

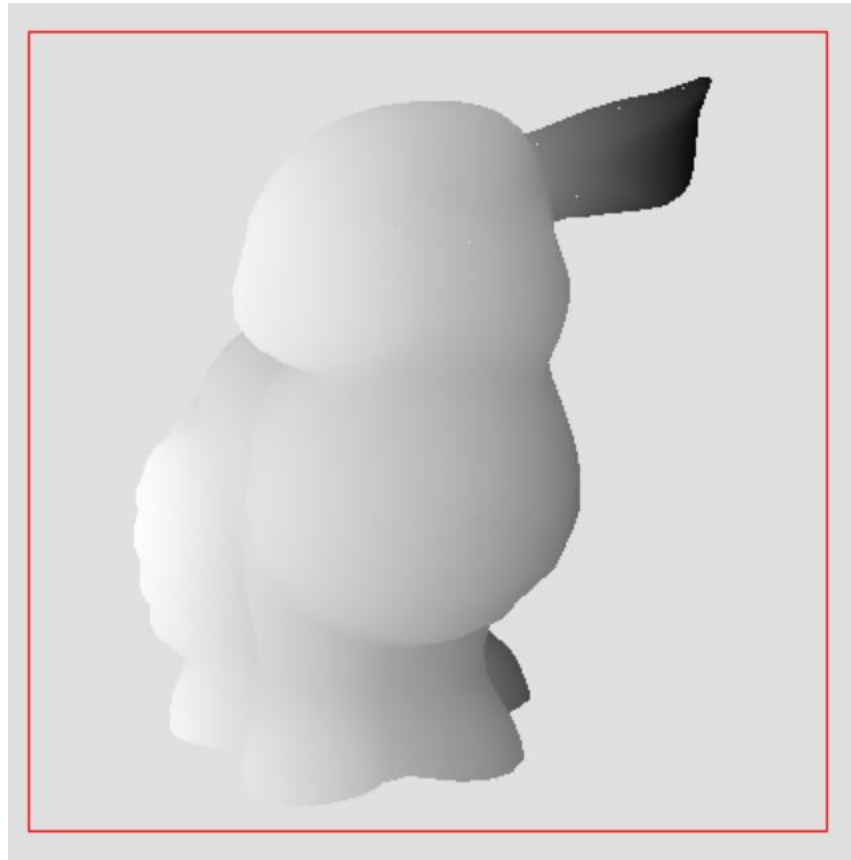


Figure 4 - Depth-based coloring

3 Assignments

- Apply the depth-based coloring scheme to the rasterized bunny from Laboratory work 10.
- Extend the triangle rasterization function from Laboratory work 9 to include a z-buffer hidden surface removal algorithm.
- Interactively switch between rasterizing with / without z-buffer, using the keyboard. To see the full impact of the algorithm, make sure to turn off Back Face Culling.

Laboratory work 11 – Polygon clipping algorithms

1 Objectives

Study, implement and evaluate the Sutherland-Hodgman and Weiler-Atherton clipping algorithms for polygons, in 2D object coordinate system.

2 Theoretical background

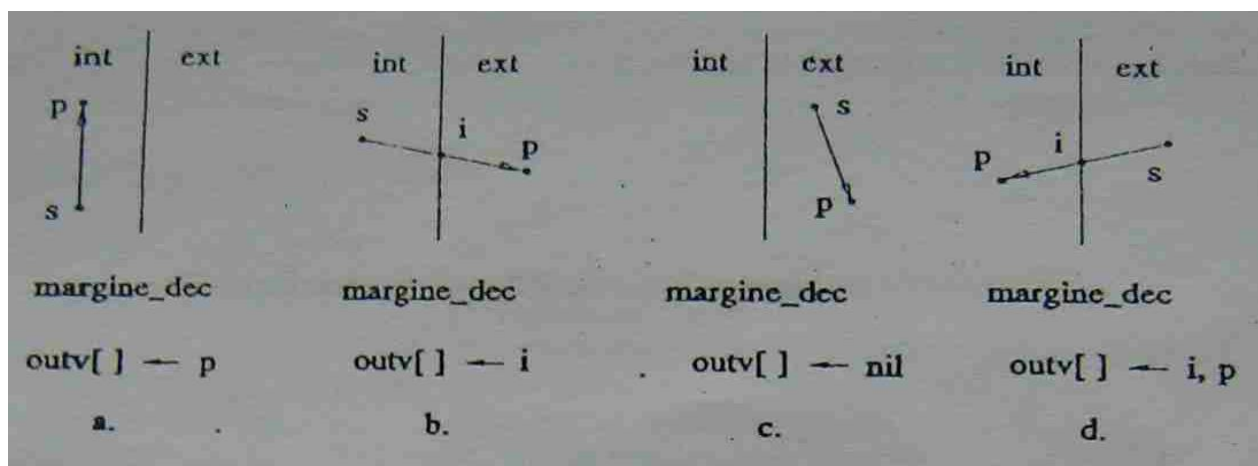
There are two main types of clipping algorithms against the margins of a display window, according to the coordinate space where we compute the operations:

- Raster algorithms, which operate in video memory. With these algorithms the clipping is computed for each pixel. Even if the algorithms themselves are pretty simple their implementation requires a large number of accesses to the video memory.
- Vectorial algorithms, which operate with the nodes describing the polygon. These clipping algorithms work directly with the data structure describing the polygons and will result in one or more new polygons described through a list of nodes.

The second types of algorithms involve more complex computations which are executed in the main system memory but their results is compatible with any graphical system as is described generally through a list of points. Two of the vectorial algorithms are very often used: Sutherland-Hodgman and Weiler-Atherton.

2.1 Sutherland-Hodgman clipping algorithm

Sutherland-Hodgman algorithm considers that the initial polygon is defined through a list of nodes $inv[] = \{v_1, v_2, \dots, v_n\}$. A conventional direction of nodes inspection is determined, for example: $v_1v_2, v_2v_3, \dots, v_nv_1$. The clipping of the polygon is finalized in four steps. At each step, all the edges of the polygon are clipped against one side of the working area. In the end, we will



obtain a list of points $outv[] = \{v1', v2', \dots, vp'\}$ that describe the clipped polygon.

Figure 1: Relationships between the margins of the display area and one edge of a polygon

We can identify four different relationships between the margins of the display area and any edge of a polygon (see Figure 1). We will consider s to be the initial node and p the final node of the edge. The four cases are:

- 1) Both nodes s and p are inside the display area. Node p will be added to the list of clipped nodes: $outv[] \leftarrow p$.
- 2) Node s is inside the display area while p is outside. We have to compute the intersection point i between the margin of the display area and the edge described by s and p . Node i will be added to the list of clipped nodes: $outv[] \leftarrow i$.
- 3) Both nodes s and p are outside the display area. Nothing will be added to the list of clipped nodes.
- 4) Node s is outside the display area while p is inside. We have to compute the intersection point between the margin of the display area and the edge described by s and p . We will add both p and i to the list of clipped nodes: $outv[] \leftarrow p, outv[] \leftarrow i$.

2.2 Sutherland-Hodgman algorithm pseudo-code description

```
Clipping SH(nodesList: inv, outv; displayAreaMargins: margine_dec[4])
{
  for j=0,4
  {
    ClipMarginSH(inv, outv, margine_dec[j]);
    inv = outv; //update the current nodes list
  }
}
```

```
ClipMarginSH( nodesList: inv, outv; displayAreaMargin: margine_dec)
{
  node i, p, s;
  s = last node from inv;
  for p=each node in inv
  {
    if(InDisplayArea(p, margine_dec)) //cases A and D
    {
      if(InDisplayArea(s, margine_dec)) //case A
      {
        add p to outv
      }
      else //case D
      {
        i = IntersectionPoint(s, p, margine_dec);
        add i to outv
        add p to outv
      }
    }
  }
}
```

```

else if(InDisplayArea(s, margine_dec)) //case B
{
    i = IntersectionPoint(s, p, margine_dec);
    add i to outv
}
s = p; //we do nothing for case C
//update the starting point
}
}

```

As you can see, the algorithm eliminates from the display list the parts of the polygons which are placed into the exterior half-plane determined by the display area margin.

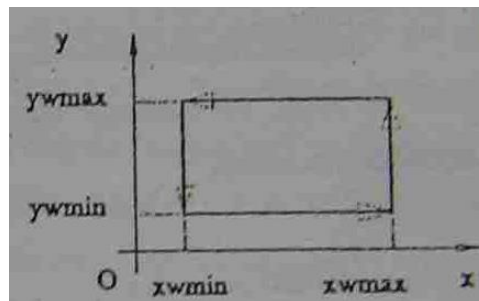


Figure 2: The checking order for the display area margins

The function `InDisplayArea(node: p; displayAreaMargin: margine_dec)` returns true if point `p` is in the same half-plane with the display area related to the line described by `margine_dec`. For simplicity we will consider a conventional direction for testing margins of the display area. Let this be the trigonometric direction. While we keep the same order into the display area margins list we can check the position of `p` according to Figure 2. This way, the function can be described:

```

InDisplayArea(node: p; displayAreaMargin: margine_dec)
{
    switch(margine_dec)
    {
        case right_margin: if(xp < xmargine_dec) return true; break;
        case top_margin: if(yp < ymargine_dec) return true; break;
        case left_margin: if(xp > xmargine_dec) return true; break;
        case bottom_margin: if(yp > ymargine_dec) return true; break;
    }
}

```

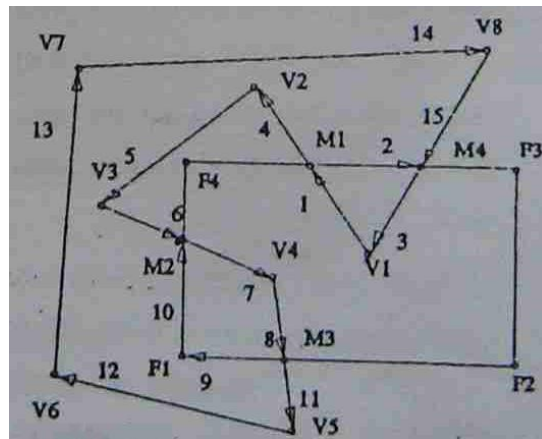
2.3 Weiler-Atherton clipping algorithm

Weiler-Atherton algorithm considers that the initial polygon is defined through a list of nodes $inv[] = \{v_1, v_2, \dots, v_n\}$. After the clipping algorithm is applied we will obtain zero, one or more polygons, each defined through a list of nodes $outvk[] = \{vk_1', vk_2', \dots, vk_p'\}$.

If the initial polygon intersects the margins of the display area, the result polygon will contain at least a portion of an edge of the initial polygon and portions from the display area margins. If the polygon is entirely outside of the display area, the result will be empty.

2.4 General description of the algorithm

The algorithm starts from one node of the polygon. Let us consider the polygon from Figure 3 and v_1 as the starting node. We will use i to count the resulting polygons. For the beginning, $i =$



1.

Figure 3: Weiler-Atherton - example of polygon clipping

1. We go through all the polygon's nodes in a conventional order, for example v_1v_2, v_2v_3, \dots . As v_1 is inside the display area we add it to the results nodes list for the first polygon $outv[1][i] \leftarrow v_1$.
2. We continue to check all the nodes of the initial polygon, in the previously established conventional order, until we get out of the display area. We add all these nodes to the $outv[1][i]$ list.
3. If we consider M_1 to be the exit point on the v_1v_2 edge, we add M_1 to $outv[1][i]$.
4. We will continue to check the margin of the display area which is inside the polygon, until we meet the first intersection with the initial polygon. We add this intersection point (M_4 in Figure 3) to the $outv[1][i]$ list, as it represents the entry point of the polygon into the display area.
5. We then continue to check the nodes of the initial polygon which are inside the display area until we get back again to v_1 or we get out again, in which case we go back to step 3 of the algorithm.
6. If v_1 has been reached, we will obtain the first result polygon. For our example: $outv[1][i] = v_1, M_1, M_4, (v_1)$.
7. We can go further to the next polygon: $i = i + 1$. Our new starting point will be M_1 .

8. We go through all the polygon nodes, in the conventional order, until we discover the first entry point (M2 in our example).
9. Starting with M2 we begin to construct a new polygon `outv[2][]`. In our example `outv[2][] = M2, v4, M3, F1 (, M2)`.

2.5 Weiler-Atherton algorithm implementation example

One possible implementation of the Weiler-Atherton algorithm could be:

1. Create a list (**lpp**) with the nodes of the initial polygon.
2. Create another list (**lpf**) that contains the corners of the display area.
3. Compute the intersection point of each polygon edge with the margins of the display area and add the resulting nodes to the **lpp** and **lpf** lists.
4. Add to the **lpf** the nodes of the polygon which reside on the margins of the display area.
5. Create the list of polygon's edges (**ILp**) which will keep for each edge a reference to two consecutive nodes of the polygon.
6. We check each edge from **ILp** to determine if it is:
 - a. Inside the display area
 - b. Outside the display area
 - c. On one of the margins of the display area
7. For each point in **lpp** we determine if it is placed:
 - a. Inside the display area
 - b. Outside the display area
 - c. On one of the margins of the display area
8. For each point in **lpf** we determine if it is placed:
 - a. Inside the polygon
 - b. Outside the polygon
 - c. On one of the edges of the polygon
9. We determine the computation order of the points in **lpf** list, keeping the result in var **sens**.
 - a. `sens = LEFT`, the next element is `->urm`
 - b. `sens = RIGHT`, the next element is `-> pred`
 - c. `sens = NEDEF`
10. if `sens = NEDEF` then


```

      if(polygon inside the window)           //q0 is the list of result polygons
          q0 = ILp                             //each polygon is represented by a list of
          edges
      else
          q0 = null
      else
      
```



```

for(each element of ILp) do
    Atherton(current_element of ILp);
    current_element = current_element -> urm;

```

11. Stop.

```

Atherton (latura: elem)
{
    if (elem is outside the window)
    {
        if(newp = true) secventa();
    }
    else if(elem is inside the window)
    {
        if(newp = false)
        {
            newp = true;
            add elem to q0;           //q0 is the list of result polygons
        }
        else
        {
            add elem to q0;
            if(elem->urm is inside the window) secventa();
        }
    }
    else                               //the point is on one margin of the display
    area
    {
        if(newp = true) secventa();
    }
}

```

The function **secventa()** should:

1. Locate in **lpf** the node that represents the beginning of the segment that is being analyzed.
2. Go through **lpf** in the conventional chosen direction until identifies a common point with **lpp**, adding to q0 each new edge from two consecutive points from **lpf**
3. if(the last point is the same with the starting point in q0)
 - then newp = false; //newp is true while we are building on the same polygon

3 Assignment

- Define a display area and mark it with a rectangle. Display a polygon and clip it against the display area previously defined using:
 - Sutherland-Hodgman algorithm

- Weiler-Atherton algorithm
- Extend the above request by allowing the user to define the display area using the mouse.

Laboratory work 12 – Bezier curves

1 Objectives

This laboratory presents the key notions on Bezier curves.

2 Theoretical background

2.1 Bezier curves

A Bezier curve is a parametric curve, used in computer graphics and other related fields, used to model smooth curves that can be scaled indefinitely. Another applicability of the Bezier curves is in animations where an object movement can be defined by using a Bezier curve, modifying in this way the velocity of the object.

2.2 Cubic Bezier curves

In order to define a cubic Bezier curve we need 4 points. The curve will pass through P_0 and P_3 , which are the starting point and the ending point of the curve. The curve will not pass through P_1 and P_2 which are control points and are used to provide directional information.

$$B(t) = (1 - t)^3 P_0 + 3(1 - t)^2 t P_1 + 3(1 - t) t^2 P_2 + t^3 P_3, t \in [0,1]$$

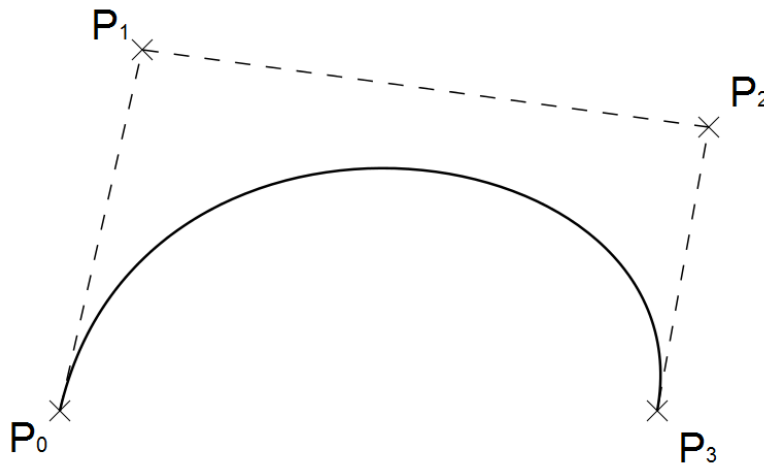


Fig. 1: Example of a Bezier curve with 4 points

2.3 Generalization

The Bezier curve of degree n can be generalized in this way:

$$B(t) = \sum_{i=0}^n \binom{n}{i} (1-t)^{n-i} t^i P_i$$

For example, for $n = 5$:

$$B(t) = (1-t)^5 P_0 + 5t(1-t)^4 P_1 + 10t^2(1-t)^3 P_2 + 10t^3(1-t)^2 P_3 + 5t^4(1-t) P_4 + t^5 P_5,$$

$$t \in [0,1]$$

Recursively the formula can be expressed as follows:

$$B(t) = B_{P_0 P_1 \dots P_n}(t) = (1-t)B_{P_0 P_1 \dots P_{n-1}}(t) + tB_{P_1 P_2 \dots P_n}(t)$$

3 Assignments

- Create an application to exemplify the Bezier curves.

References

1. Computer Graphics: Principles and Practice, John F. Hughes, Andries van Dam, James D. Foley, Morgan McGuire, Steven K. Feiner, David F. Sklar, Addison-Wesley, 2014
2. Fundamentals of Computer Graphics 4th Edition, Steve Marschner, Peter Shirley, A K Peters/CRC Press; 4 edition, 2015
3. Computer Graphics: Principles and Practice in C 2nd Edition, James D. Foley, Andries van Dam, Steven K. Feiner, John F. Hughes, Addison-Wesley Professional, 1995
4. Interactive Computer Graphics: A Top-Down Approach with Shader-Based OpenGL 6th Edition, Edward Angel and Dave Shreiner, Addison-Wesley, 2012
5. 3D Computer Graphics (3rd Edition), Allan Watt, Addison-Wesley, 2013
6. Mathematics for 3D Game Programming and Computer Graphics 3rd Edition, Eric Lengyel, Course Technology PTR, 2012
7. Foundations of 3D Computer Graphics, Steven J. Gortler, The MIT Press, 2012
8. Graphics and Visualization: Principles & Algorithms, T. Theoharis, A K Peters/CRC Press, 2008
9. Mathematics for Computer Graphics, John Vince, Springer; 5th ed., 2017