Visiting Mosque

2020 - 2021

Students:

- CĂPĂȚÎNĂ Răzvan Nicolae, 11th Grade, "Bogdan Petriceicu Hasdeu" National College, Buzău, Romania
- ENE Oana Maria, 11th Grade, "Bogdan Petriceicu Hasdeu" National College, Buzău, Romania
- NECHITA Maria, 11th Grade, "Bogdan Petriceicu Hasdeu" National College, Buzău, Romania

Teacher:

- NICOLAE Melania, "Bogdan Petriceicu Hasdeu" National College, Buzău, Romania

Researcher:

- BROUZET Robert, Université de Perpignan, France

THE SUBJECT:

Which are the points on the plane with integer coordinates 'visible' from the origin?
Are there any large invisible areas?

INTRODUCTION:

Each integer point on the plane is associated to one of the 856 columns of Mosque Cathedral.

The Great Mosque was constructed on the orders of Abd ar-Rahman I in 785 CE, when Córdoba was the capital of the Muslim-controlled region of Al-Andalus.

The Mosque Cathedral's hypostyle hall dates from the original mosque construction and originally served as the main prayer space for Muslims. The main hall of the mosque was used for a variety of purposes. It served as a central prayer hall for personal devotion; this is why we associated it to the origin: O (0,0).

The problem can therefore be rephrased as it follows:

Which are the columns of the Mosque Cathedral visible from the central prayer hall?

Are there any large invisible groups of columns?

THE SOLUTIONS FOUND:

To solve this problem we used two methods:

Method 1: COORDINATE PLANE STUDY

1.1. Graphic representation

Consider the 2 lines of sight denoted by the line segments emanating from the origin (the red bullet point) in the figure shown. In these two lines of sight, there are exactly two columns which are visible – one per each line of sight.

These visible columns are located at the blue bullet points. Obscured by them are three other black columns, which are not visible from the origin.

The column E(2,6) is obscured by the visible column D(1,3), while the columns C(6,3) and B(4,2) are obscured by the visible column A(2,1).

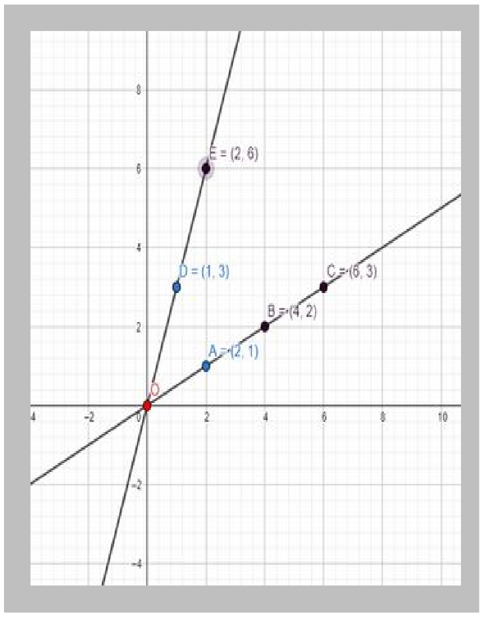


Figure 1: Graphic representation of method 1.1

1.2. Video simulation

We have created our own Mosque columns in our town:

! A video can also be found by accessing this link:

https://drive.google.com/file/d/1kxUbGHwvT1272Muoarhf3yPV1lF7sT_a/view

Video_simulation.MOV (2).mp4

Definition: In number theory, two integers a and b are coprime, relatively prime or mutually prime if the only positive integer that is a divisor of both of them is 1. Consequently, any prime number that divides one of a or b does not divide the other. This is equivalent to their greatest common divisor (gcd) being 1.

It turns out that the only visible points are the points X (a, b), where a and b are coprime.

$$\gcd(a, b) = 1$$

1.3. Are there large square areas of invisible points on the plane?

By creating a graphic representation, we found an example of a 2 X 2 hidden area of columns.

In this figure we can notice the specific four visible columns that obscure the other hidden ones:

E(20,14) – obscured by A(10,7)

F(21,14) – obscured by B(3,2)

G(20,15) – obscured by C(4,3)

H(21,15) – obscured by D(7,5)

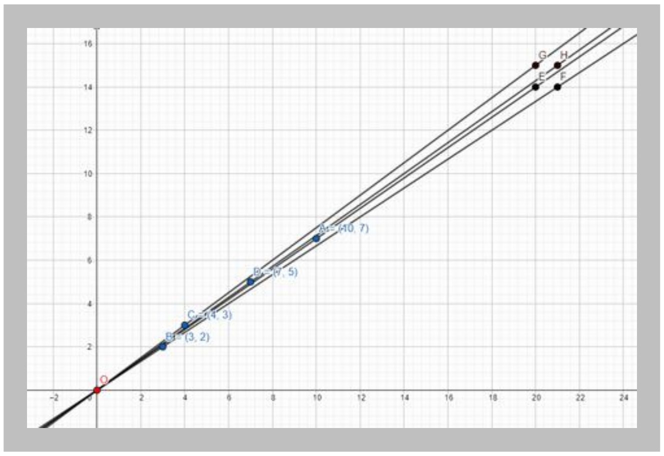


Figure 2: Graphic representation of method 1.3

Method 2: PROGRAMMING

In order to have an expanded view of all the points on the plane and to identify the large invisible areas, which are randomly placed very far away from the origin, we created a C++ program. For further explanation of the code, see the blue notes below:

```
#include <iostream>
#include <vector>

#include <glew.h>
#include <glfw3.h>
#include <glm.hpp>
#include
<gtc/matrix_transform.hpp>
#include <gtc/type_ptr.hpp>

using namespace std;

const double WINDOW_WIDTH =
1024.0;
const double WINDOW_HEIGHT =
1024.0;

const char* vertexShaderSource =
"#version 330 core \n"
"\n"
"layout (location = 0) in vec2
vertexPosition; \n"
"uniform mat4 ortho; \n"
"\n"
"void main() \n"
"{ \n"
"\n"
"   gl_Position = ortho *
vec4(vertexPosition.x,
vertexPosition.y, 0.0, 1.0); \n"
"\n"
"} \n"
"\0";

const char* fragmentShaderSource
=
"#version 330 core \n"
"\n"
"out vec4 vertexColour; \n"
"uniform vec3 colour; \n"
```

```
"\n"
"void main() \n"
"{ \n"
"\n"
"   vertexColour = vec4(colour, 1.0);
\n"
"\n"
"} \n"
"\0";

unsigned int VAO;
unsigned int VBO;

double currentTime;
double previousTime;
double deltaTime;

class Camera // here is a class
called "Camera" which stores all the
relevant pieces of information
regarding the camera, such as its X
and Y positions, but also its
movement speed and zoom speed
{
public:

    double x;
    double y;

    double movementSpeed;
    double zoomSpeed;

    double scale;
    const double MAX_SCALE = 1000.0;
    const double MIN_SCALE = 1.0;
```

```
Camera(double x, double y, double
movementSpeed, double zoomSpeed,
double scale) :
        x(x), y(y),
movementSpeed(movementSpeed),
zoomSpeed(zoomSpeed),
scale(scale) {}
    ~Camera() {};
private:
};
void updateDeltaTime() // this
function is used to smooth out
the movement, since the framerate
of the program may alter during
its run
{
    currentTime = glfwGetTime();
    deltaTime = currentTime -
previousTime;
    previousTime = currentTime;
}
void handleInput(GLFWwindow*
window, Camera* camera) // this
function handles the movement of
the camera onto the screen
```

```
void handleInput(GLFWwindow*
window, Camera* camera) // this
function handles the movement of
the camera onto the screen
{
    if (glfwGetKey(window,
GLFW_KEY_ESCAPE) == GLFW_PRESS) //
when we press escape the program
stops

glfwSetWindowShouldClose(window,
true);
    if (glfwGetKey(window,
GLFW_KEY_W) == GLFW_PRESS) // we
move upwards if we press 'w'
        camera->y += camera-
>movementSpeed * deltaTime;
    if (glfwGetKey(window,
GLFW_KEY_S) == GLFW_PRESS) // we
move downwards if we press 's'
        camera->y -= camera-
>movementSpeed * deltaTime;
    if (glfwGetKey(window,
GLFW_KEY_A) == GLFW_PRESS) // we
move to the left if we press 'a'
        camera->x -= camera-
>movementSpeed * deltaTime;
if (glfwGetKey(window, GLFW_KEY_D)
== GLFW_PRESS) // we move to the
right if we press 'd'
        camera->x += camera-
>movementSpeed * deltaTime;
if (glfwGetKey(window, GLFW_KEY_Q)
== GLFW_PRESS) // if we press 'q'
we will zoom in
    {
        camera->scale -= camera-
>zoomSpeed * camera->scale / 10.0 *
deltaTime;
        camera->scale = max(camera-
>scale, camera->MIN_SCALE);
    }
    if (glfwGetKey(window,
GLFW_KEY_E) == GLFW_PRESS) //
pressing 'e' will zoom out
    {
        camera->scale += camera-
>zoomSpeed * camera->scale / 10.0 *
deltaTime;
        camera->scale = min(camera-
>scale, camera->MAX_SCALE);
    }
}
```

```
bool isVisible(long long x, long
long y) // this function decides
whether a point at the
coordinates X and Y should be
displayed in white or in black
{
    x = abs(x); // we take the
absolute value of both
coordinates
    y = abs(y);
    if (x == 0 && y == 0) return
false; // if both of them are 0
then it is the origin and we
will return, since we want to
draw the origin in red
    if (x == y) return x ==
1; // if both of them are equal
then the point is visible only
if x == y == 1
    if (x == 0) return y ==
1; // if x is 0, then y must be
1 in order to draw the point
using white
    if (y == 0) return x ==
1; // if y is 0, then x must be
1 in order to draw the point
using white
// here we start the Euclid's
Algorithm for finding the
greates common divisor of X and
Y
    // we had the previous if
statements to eliminate some
special cases where this next
algorithm would not have worked

    long long rest;

    while (x % y)
    {
        rest = x % y;
        x = y;
        y = rest;
    }
return y == 1; // a point is
white / visible only if the
greatest common divisor of its
coordinates are 1 (the
coordinates are relative primes)
}

vector<double> vertices;
vector<double> origin;

void drawRectangle(double x,
double y, double width, double
height) // here we add a white
cell to the list of squares we
will draw onto the screen
{
    vertices.push_back(x);
    vertices.push_back(y);
    vertices.push_back(x +
width);
    vertices.push_back(y);
    vertices.push_back(x);
    vertices.push_back(y +
height);
```

```
vertices.push_back(x + width);
    vertices.push_back(y);

    vertices.push_back(x);
    vertices.push_back(y +
height);

    vertices.push_back(x +
width);
    vertices.push_back(y +
height);
}

void drawOrigin(double x,
double y, double width, double
height) // this function draws
the origin in red
{
    origin.push_back(x);
    origin.push_back(y);

    origin.push_back(x +
width);
    origin.push_back(y);

    origin.push_back(x);
    origin.push_back(y +
height);

    origin.push_back(x +
width);
    origin.push_back(y);

    origin.push_back(x);
    origin.push_back(y +
height);

    origin.push_back(x +
width);
    origin.push_back(y +
height);
}
}

void drawOrigin(double x,
double y, double width, double
height) // this function draws
the origin in red
{
    origin.push_back(x);
    origin.push_back(y);

    origin.push_back(x +
width);
    origin.push_back(y);

    origin.push_back(x);
    origin.push_back(y +
height);

    origin.push_back(x +
width);
    origin.push_back(y);

    origin.push_back(x);
    origin.push_back(y +
height);
origin.push_back(x + width);
    origin.push_back(y +
height);
}

int colourPath;
void draw(Camera* camera) //
this functions draws all the
squares (white, red and
black), respecting their
dimensions which is influenced
by how much we have zoomed in
or out
```

```
 {
    vertices.clear();
    origin.clear();

    double rectangleWidth =
WINDOW_WIDTH / (2.0 * camera-
>scale + 1.0);
    double rectangleHeight =
WINDOW_HEIGHT / (2.0 * camera-
>scale + 1.0);
for (long long i = camera->x -
camera->scale - 1; i <= camera-
>x + camera->scale + 1; i++)
    {
        for (long long j =
camera->y - camera->scale - 1;
j <= camera->y + camera->scale
+ 1; j++)
        {
            if (isVisible(i,
j))
            {
                drawRectangle(-
WINDOW_WIDTH / 2.0 +
rectangleWidth * (i - camera->x
+ camera->scale), -
WINDOW_HEIGHT / 2.0 +
rectangleHeight * (j - camera-
>y + camera->scale),
rectangleWidth,
rectangleHeight);
            }
        }
    }

    drawOrigin(-WINDOW_WIDTH /
2.0 + rectangleWidth * (-
camera->x + camera->scale), -
WINDOW_HEIGHT / 2.0 +
rectangleHeight * (-camera->y +
camera->scale), rectangleWidth,
rectangleHeight);
if (vertices.size() > 0){

glBufferData(GL_ARRAY_BUFFER,
sizeof(double) *
vertices.size(),
&(vertices.front()),
GL_DYNAMIC_DRAW);
        glUniform3f(colourPath,
1.0, 1.0, 1.0);

glDrawArrays(GL_TRIANGLES, 0,
vertices.size() / 2); }
glBufferData(GL_ARRAY_BUFFER,
sizeof(double) * origin.size(),
&(origin.front()),
GL_DYNAMIC_DRAW);
    glUniform3f(colourPath,
1.0, 0.0, 0.0);
    glDrawArrays(GL_TRIANGLES,
0, origin.size() / 2);
}
void displayCoordinates(Camera*
camera) // this function
display where is the camera
situated at the moment (this
function has been used for
debugging purposes)
{
    cout << "Current point
coordinates: " << camera->x <<
' ' << camera->y << '\n';
}

int main()
{
    glfwInit();


glfwWindowHint(GLFW_CONTEXT_VER
SION_MAJOR, 3);

glfwWindowHint(GLFW_CONTEXT_VER
SION_MINOR, 3);

glfwWindowHint(GLFW_OPENGL_PROF
ILE, GLFW_OPENGL_CORE_PROFILE);

    GLFWwindow* window =
glfwCreateWindow(WINDOW_WIDTH,
WINDOW_HEIGHT, "Visite a la
Mezquita", 0, 0);
    //glfwGetPrimaryMonitor();


glfwMakeContextCurrent(window);

    glewInit();

    //Camera* camera = new
Camera(129963314.0,
2546641254872348.0, 75.0, 5.0,
10.0); // by using these 2
coordinates instead of 0, 0 we
will see a 5x5 invisible zone
    Camera* camera = new
Camera(0.0, 0.0, 75.0, 5.0,
10.0); // here we set the
coordinates of the camera to
the origin 0, 0 (the first 2
parameters)

    unsigned int vertexShader =
glCreateShader(GL_VERTEX_SHADER
);

glShaderSource(vertexShader, 1,
&vertexShaderSource, NULL);

glCompileShader(vertexShader);

    unsigned int fragmentShader
=
glCreateShader(GL_FRAGMENT_SHAD
ER);

glShaderSource(fragmentShader,
1, &fragmentShaderSource,
NULL);

glCompileShader(fragmentShader)
;

    unsigned int shaderProgram
= glCreateProgram();

glAttachShader(shaderProgram,
vertexShader);

glAttachShader(shaderProgram,
fragmentShader);

glLinkProgram(shaderProgram);

glDeleteShader(vertexShader);

glDeleteShader(fragmentShader);

glUseProgram(shaderProgram);
    colourPath =
glGetUniformLocation(shaderProg
ram, "colour");
    int orthoPath =
glGetUniformLocation(shaderProg
ram, "ortho");
    glGenVertexArrays(1, &VAO);
    glGenBuffers(1, &VBO);
    glBindVertexArray(VAO);

glBindBuffer(GL_ARRAY_BUFFER,
VBO);
    glVertexAttribPointer(0, 2,
GL_DOUBLE, GL_FALSE, 2 *
sizeof(double), (void*)0);

glEnableVertexAttribArray(0);
    glm::mat4 ortho =
glm::ortho(-0.5 * WINDOW_WIDTH,
0.5 * WINDOW_WIDTH, -0.5 *
WINDOW_HEIGHT, 0.5 *
WINDOW_HEIGHT); // this
orthogonal matrix is used to
transform the absolute position
of a point into its screen
relative position

glUniformMatrix4fv(orthoPath,
1, GL_FALSE,
glm::value_ptr(ortho));
while (!
glfwWindowShouldClose(window))
// this while loop draws every
frame and clears the screen
after every drawn frame in
order to draw the next one,
until the program is stopped
using 'escape'
{
        updateDeltaTime();
        glClearColor(0.0, 0.0,
0.0, 1.0);

glClear(GL_COLOR_BUFFER_BIT);
        handleInput(window,
camera);
```

```
    draw(camera);


displayCoordinates(camera);

        glfwSwapBuffers(window);

        glfwPollEvents();

    }

  glDeleteBuffers(1, &VBO);

    glDeleteVertexArrays(1,
&VAO);

    glfwDestroyWindow(window);

    glfwTerminate();

         return 0;

}
```

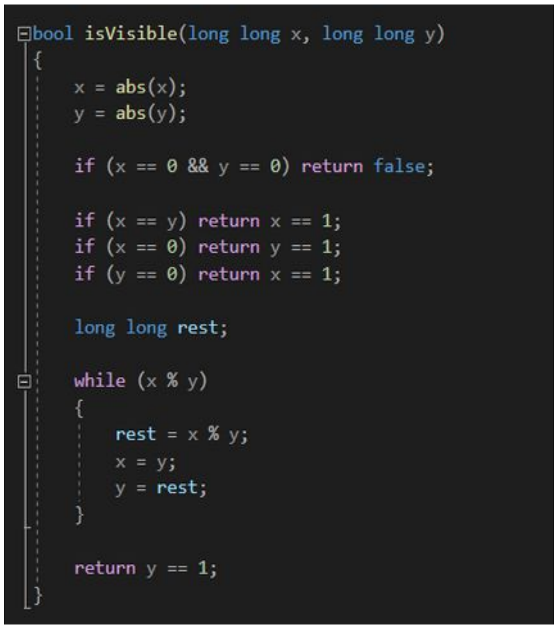
! The C++ program can also be found by accessing this link:

https://drive.google.com/drive/folders/1nBBqJucmq-oZaZEalVErL6oOO96nRPUe

Main.cpp

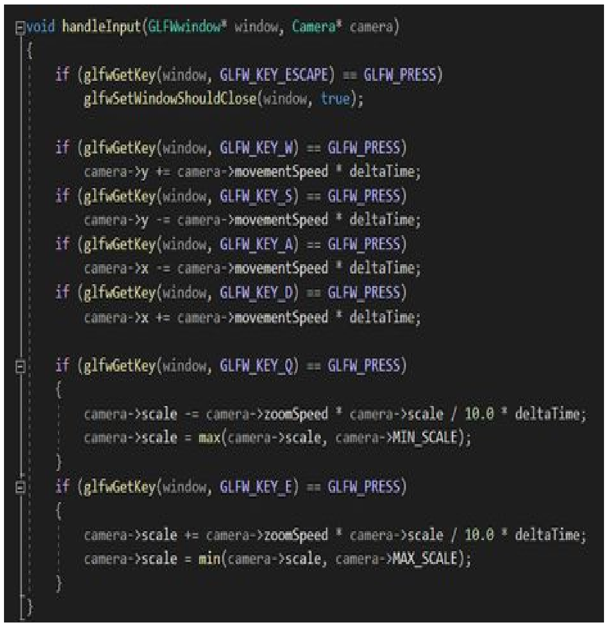
In order to try it, you must have the application Visual Studio installed on your computer.

Important parts of the code:



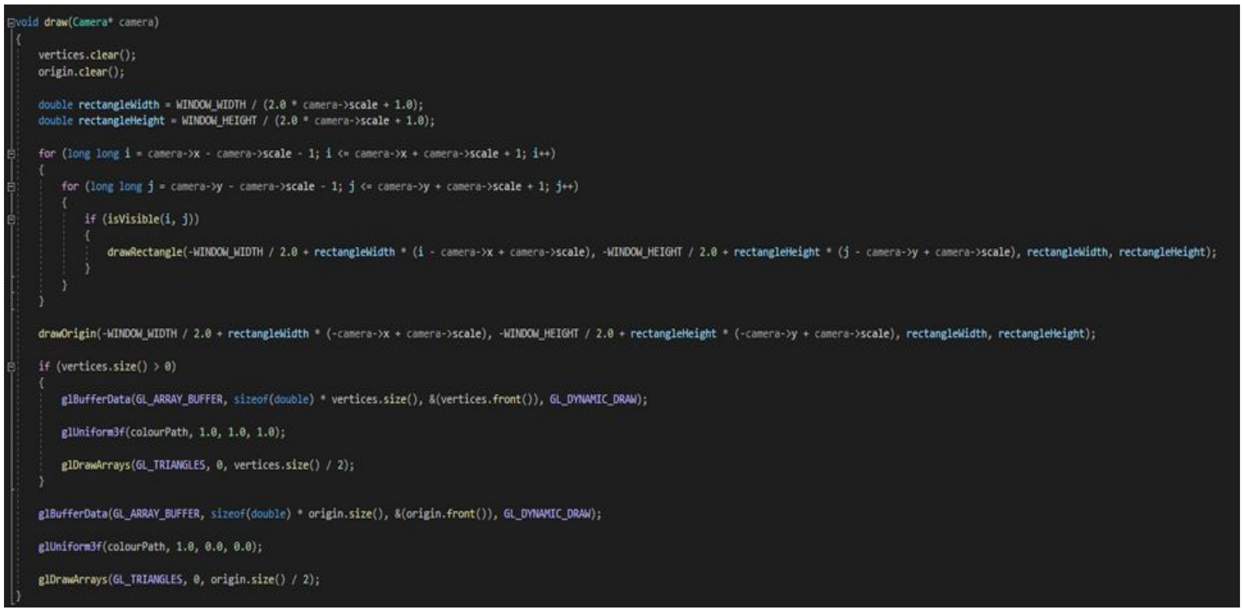
```
bool isVisible(long long x, long long y)
{
    x = abs(x);
    y = abs(y);

    if (x == 0 && y == 0) return false;

    if (x == y) return x == 1;
    if (x == 0) return y == 1;
    if (y == 0) return x == 1;

    long long rest;

    while (x % y)
    {
        rest = x % y;
        x = y;
        y = rest;
    }

    return y == 1;
}
```

Figure 3: The function which returns whether a point is visible or not by calculating the gcd of its coordinates



```
void handleInput(GLFWwindow* window, Camera* camera)
{
    if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);

    if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
        camera->y += camera->movementSpeed * deltaTime;
    if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
        camera->y -= camera->movementSpeed * deltaTime;
    if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
        camera->x -= camera->movementSpeed * deltaTime;
    if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
        camera->x += camera->movementSpeed * deltaTime;

    if (glfwGetKey(window, GLFW_KEY_Q) == GLFW_PRESS)
    {
        camera->scale -= camera->zoomSpeed * camera->scale / 10.0 * deltaTime;
        camera->scale = max(camera->scale, camera->MIN_SCALE);
    }
    if (glfwGetKey(window, GLFW_KEY_E) == GLFW_PRESS)
    {
        camera->scale += camera->zoomSpeed * camera->scale / 10.0 * deltaTime;
        camera->scale = min(camera->scale, camera->MAX_SCALE);
    }
}
```

Figure 4: The function which handles the movement of the camera in the graph.



```
void draw(Camera* camera)
{
    vertices.clear();
    origin.clear();

    double rectangleWidth = WINDOW_WIDTH / (2.0 * camera->scale + 1.0);
    double rectangleHeight = WINDOW_HEIGHT / (2.0 * camera->scale + 1.0);

    for (long long i = camera->x - camera->scale - 1; i <= camera->x + camera->scale + 1; i++)
    {
        for (long long j = camera->y - camera->scale - 1; j <= camera->y + camera->scale + 1; j++)
        {
            if (isVisible(i, j))
            {
                drawRectangle(-WINDOW_WIDTH / 2.0 + rectangleWidth * (i - camera->x + camera->scale), -WINDOW_HEIGHT / 2.0 + rectangleHeight * (j - camera->y + camera->scale), rectangleWidth, rectangleHeight);
            }
        }
    }

    drawOrigin(-WINDOW_WIDTH / 2.0 + rectangleWidth * (-camera->x + camera->scale), -WINDOW_HEIGHT / 2.0 + rectangleHeight * (-camera->y + camera->scale), rectangleWidth, rectangleHeight);

    if (vertices.size() > 0)
    {
        glBufferData(GL_ARRAY_BUFFER, sizeof(double) * vertices.size(), &(vertices.front()), GL_DYNAMIC_DRAW);

        glUniform3f(colourPath, 1.0, 1.0, 1.0);

        glDrawArrays(GL_TRIANGLES, 0, vertices.size() / 2);
    }

    glBufferData(GL_ARRAY_BUFFER, sizeof(double) * origin.size(), &(origin.front()), GL_DYNAMIC_DRAW);

    glUniform3f(colourPath, 1.0, 0.0, 0.0);

    glDrawArrays(GL_TRIANGLES, 0, origin.size() / 2);
}
```

Figure 5: The function which goes through all the points around the camera and verifies which of them are visible and which are not.

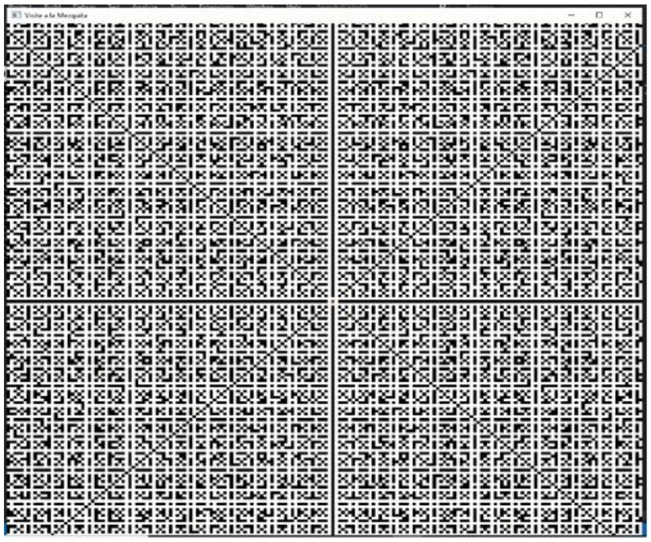


Figure 6: The center point (0, 0) is represented in red; the white points are visible, while the black points are not.

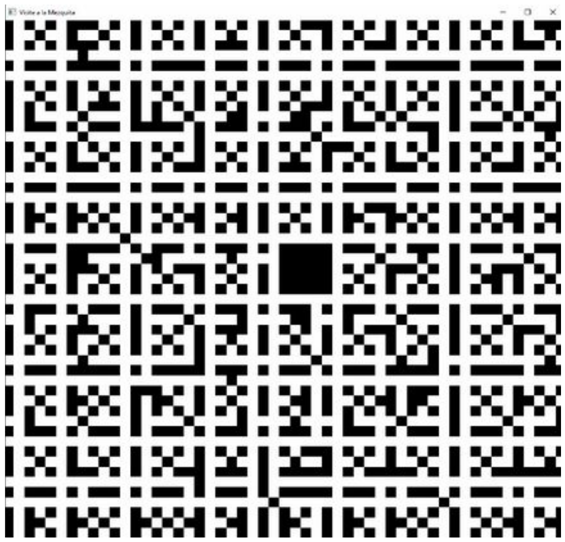


Figure 7: 5x5 invisible zone found by the C++ program at the coordinates: 129963314 and 2546641254872348

The points of integer coordinates (1,0), (0,1), (-1,0) and (0,-1) are the only visible points on the OX, OY axes from the origin. Obscured by them are all the other points on the axes, which are not visible from the origin. This results in the formation of 4 infinite areas of invisible points, in the form of 4 half lines emanating from the visible points mentioned above.

CONCLUSIONS

Conclusion 1

Let $X(a,b) \in \mathbb{Z}^2 \setminus \{(0,0)\}$
Then X(a,b) is visible if and only if **gcd(a,b)=1**.
Let: X(a,b) be a non-origin point in $\mathbb{Z}^2$
c = gcd(a,b); c > 1
$X'(\frac{a}{c}, \frac{b}{c})$ lies strictly between O(0,0) and X(a,b).
X can't therefore be seen from the origin, because X' blocks the sight line between O and X.

Conclusion 2

The second conclusion we have reached is that these large invisible areas required by the second half of our problem **do exist.**

The problem is that these areas are chaotically arranged. This means that we weren't able to find any pattern to follow in order to identify their exact positions. However, the larger the area is, the further we need to move away from the origin to find it.

There are four infinite regions of collinear points, which correspond to the coordinate axes of the plane. Another region of invisible points is the 5X5 area, which would have been impossible to find without the help of the program. We cannot say with certainty which is the largest area of points that is not visible from the origin. The computer is not able to operate with such big numbers, because they don't fit in the data type defined.

There is still some way to go before finding precise answers to all our questions and we concluded that this problem cannot be completely solved at the present time. We believe that technology will enable other better solutions in the future.

Notes d'édition

Aucune.