Title: To use Composite Transformations for Scaling and Translating a Triangle

Objective:

1. To implement a method for scaling and translating a triangle.
2. To evaluate how well the transformations affect the triangle's appearance.
3. To analyze the efficiency and speed of the transformation process.
4. To visualize and display the triangle with varying parameters.

Theory:

A triangle in a 2D plane is defined by its three vertices. In computer graphics, similar to circle drawing, we cannot directly render a perfect triangle due to the discrete nature of pixels. Instead, we approximate the triangle by determining and illuminating the closest pixels that best represent its edges. The vertices of a triangle can be represented as (x_1, y_1), (x_2, y_2), and (x_3, y_3).

To perform transformations, we will use composite transformations, where scaling is applied first, followed by translation. The translation matrix, scaling matrix and composite matrix can be defined as:

Translation Matrix: $\begin{bmatrix} 1 & 0 & Tx \\ 0 & 1 & Ty \\ 0 & 0 & 1 \end{bmatrix}$

Scaling Matrix: $\begin{bmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{bmatrix}$

Composite Matrix: $\begin{bmatrix} Sx & 0 & Tx \\ 0 & Sy & Ty \\ 0 & 0 & 1 \end{bmatrix}$

Algorithm:

`Step 1:` Start

`Step 2:` Declare variables for the triangle vertices (x_1, y_1), (x_2, y_2), (x_3, y_3), scaling factors s_x,s_y, and translation factors t_x, t_y.

`Step 3:` Read values of the triangle vertices, scaling factors, and translation factors.

`Step 4:` Apply scaling to each vertex :

$x^I = s_x * x$;

$y^I = s_y * y$;

`Step 5:` Apply translation to each vertex :

$x^{II} = x^I + t_x$;

$y^{II} = y^I + t_y$;

`Step 6:` Use line function to draw the edges of the triangle using the transformed vertices.

`Step 7:` Stop

Source Code:

```
#include <stdio.h>
#include <graphics.h>

void scaleTranslate(int *x, int *y, float sx, float sy, int tx, int ty) {
   *x = *x * sx + tx;
   *y *= *y * sy + ty;
}

void plotLines(int x1, int y1, int x2, int y2, int x3, int y3){
   line(x1, y1, x2, y2);
   line(x2, y2, x3, y3);
   line(x3, y3, x1, y1);
}
void plotScaledTranslatedLines(int x1, int y1, int x2, int y2, int x3, int y3, float sx, float sy, int tx,
int ty){
   line(x1*sx+tx, y1*sy+ty, x2*sx+tx, y2*sy+ty);
   line(x2*sx+tx, y2*sy+ty, x3*sx+tx, y3*sy+ty);
   line(x3*sx+tx, y3*sy+ty, x1*sx+tx, y1*sy+ty);
}

int main()
{
   int gd = DETECT, gm;
   initgraph(&gd, &gm, NULL);
   int x1 = 130, y1 = 80;
   int x2 = 80, y2 = 180;
   int x3 = 180, y3 = 180;
   int tx,ty;
   float sx,sy;
   plotLines(x1,y1,x2,y2,x3,y3);
   outtextxy(x1-70, y1-20, "Original Triangle");
   outtextxy(30, 10, "Kushal Shah");
   printf("Enter the translation factors = ");
   scanf("%d%d",&tx,&ty);
   printf("Enter the scaling factors = ");
   scanf("%f%f",&sx,&sy);
   plotScaledTranslatedLines(x1,y1,x2,y2,x3,y3,sx,sy,tx,ty);
   scaleTranslate(&x1, &y1, sx, sy, tx, ty);
   scaleTranslate(&x2, &y2, sx, sy, tx, ty);
   scaleTranslate(&x3, &y3, sx, sy, tx, ty);
   outtextxy(170, 380, "Scaled Translated Triangle");
   delay(50000);
   closegraph();
   return 0;
}
```

Output:

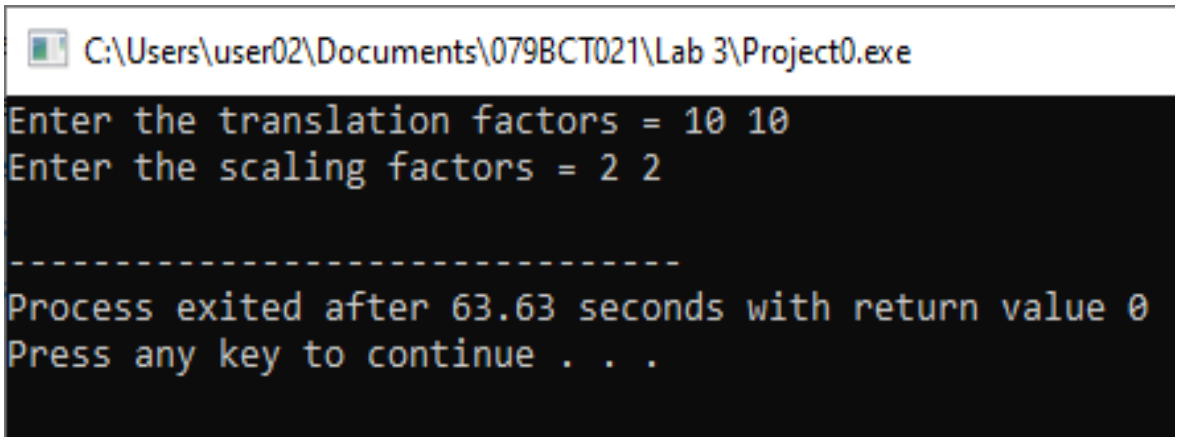

Figure 1: Inserting translation and scaling factors.

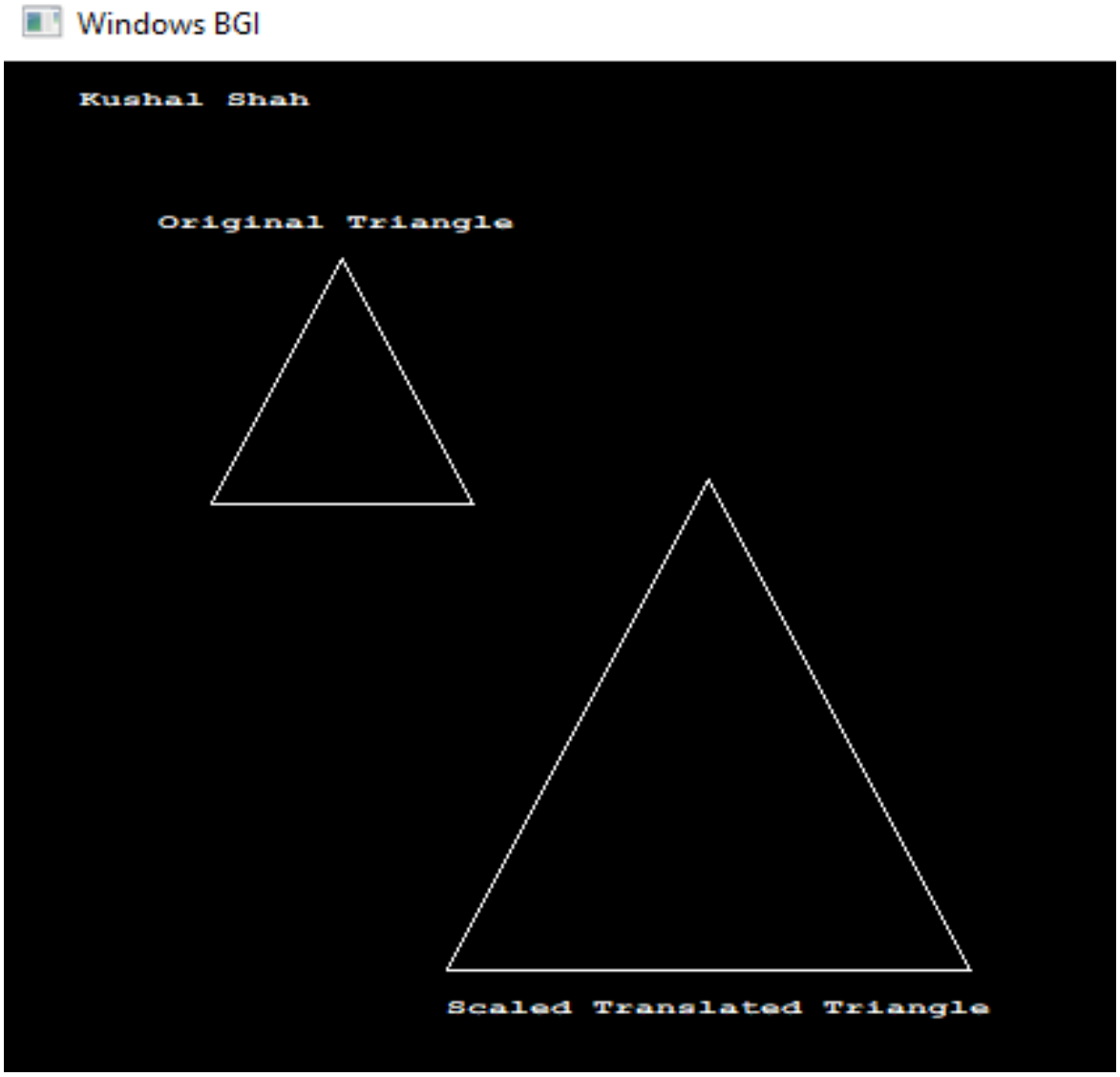

Figure 2: Drawing original and scaled translated triangle.

Discussion:

The method for scaling and translating a triangle using composite transformations is efficient and straightforward. By applying scaling first, we can adjust the size of the triangle, and then by translating, we can position it anywhere on the screen. This approach allows for flexible manipulation of graphical objects in computer graphics. The implementation leverages basic geometric transformations and can be easily extended to include rotation or other transformations as needed. The use of integer arithmetic in the drawing functions minimizes rounding errors and enhances performance.

Conclusion:

The composite transformation method for scaling and translating triangles is a fundamental technique in computer graphics. By applying transformations in a systematic manner, we can efficiently manipulate graphical objects while maintaining visual accuracy. This method is essential for various applications in graphics programming, providing a reliable way to generate and manipulate shapes across different digital platforms.