**Kathmandu University**

**Department of Computer Science and Engineering Dhulikhel, Kavre**



**Mini Report on**

**“Lab 4”**

**[Course Code: COMP 342]**

**(For partial fulfillment of III Year/ I Semester in Computer Science)**

**Submitted By**

**Nisham Ghimire (17)**

**Submitted To**

**Mr. Dhiraj Shrestha**

**Department of Computer Science and Engineering**

**Submission Date**

**14th January, 2024**

**LAB 4**

1. *Write a Program to implement:*
   1. *2D Translation*
   2. *2D Rotation*
   3. *2D Scaling*
   4. *2D Reflection*
   5. *2D Shearing*

*(For doing these Transformations consider any 2D shapes (Line, Triangle, Rectangle etc), and use Homogeneous coordinate Systems)*

1. **2D Translation:**

**Algorithm:**

* + 1. Create an empty set for the translated points ′*P*′.
    2. For each point in the original set:
       1. Take a Point *P*(*x*,*y*):

Calculate the translated x-coordinate *x*′=*x*+*dx*.Calculate the translated y- coordinate *y*′=*y*+*dy*.

* + - 1. Add the Translated Point to ′*P*′:

Add the point *P*′(*x*′,*y*′) to the set of translated points ′*P*′.

* + 1. Display the set of translated points ′*P*′.

Source Code:

import numpy as np

from typing import Tuple

import pygame as pg

from pygame.locals import \*

from OpenGL.GL import \*

from OpenGL.GLU import \*

from math import \*

Coordinate = Tuple[float, float]

def translate(point: Coordinate, translateX\_by: int, translateY\_by: int) -> Coordinate:

    x, y = point

    m = np.array([[x], [y], [1]])

    tx\_m = np.array([[1, 0, translateX\_by], [0, 1, translateY\_by], [0, 0, 1]])

    result = np.dot(tx\_m, m)

    return tuple(result[:2, 0])

def displayPaint():

    st\_point: Coordinate = (-4, -6)

    end\_point: Coordinate = (7, 3)

    tsl\_By = (5, 8)

    st\_tps = translate(st\_point, tsl\_By[0], tsl\_By[1])

    end\_tps = translate(end\_point, tsl\_By[0], tsl\_By[1])

    glBegin(GL\_LINES)

    glColor3f(1.0, 0.0, 1.0)

    glVertex2f(st\_point[0], st\_point[1])

    glVertex2f(end\_point[0], end\_point[1])

    glEnd()

    glBegin(GL\_LINES)

    glColor3f(1.0, 1.0, 1.0)

    glVertex2f(st\_tps[0], st\_tps[1])

    glVertex2f(end\_tps[0], end\_tps[1])

    glEnd()

def main():

    pg.init()

    pg.display.set\_mode((600, 600), DOUBLEBUF | OPENGL | GL\_RGB)

    pg.display.set\_caption("Translate - COMP342 Computer Graphics Lab")

    gluPerspective(150, 1, 1, 10)

    glTranslatef(0.0, 0.0, -10)

    while True:

        for ev in pg.event.get():

            if ev.type == pg.QUIT:

                pg.quit()

                quit()

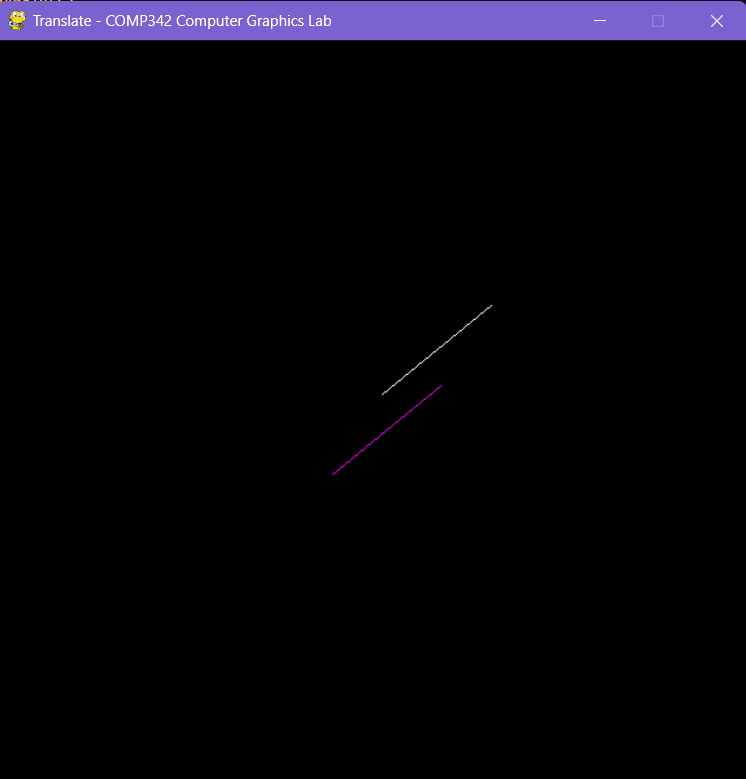
        displayPaint()

        pg.display.flip()

if \_\_name\_\_ == "\_\_main\_\_":

    main()

Output:



1. **2D Scaling:**

**Algorithm:**

1. Input the coordinates of the 2D point: (x, y).
2. Input the scaling factors for the x-axis (Sx) and y-axis (Sy).
3. Compute the scaled coordinates using the following formulas:
   * x' = x \* Sx
   * y' = y \* Sy
4. The new coordinates (x', y') represent the scaled point.

**Source Code:**

import numpy as np

from typing import Tuple

import pygame as pg

from pygame.locals import \*

from OpenGL.GL import \*

from OpenGL.GLU import \*

from math import \*

Coordinate = Tuple[float, float]

def scale(point: Coordinate, scaleX\_by: int, scaleY\_by: int) -> Coordinate:

    x, y = point

    m = np.array([[x], [y], [1]])

    tx\_m = np.array([[scaleX\_by, 0, 0], [0, scaleY\_by, 0], [0, 0, 1]])

    result = np.dot(tx\_m, m)

    return tuple(result[:2, 0])

def displayPaint():

    st\_point: Coordinate = (-4, -6)

    end\_point: Coordinate = (7, 3)

    scale\_By = (2, 2)

    st\_tps = scale(st\_point, scale\_By[0], scale\_By[1])

    end\_tps = scale(end\_point, scale\_By[0], scale\_By[1])

    glBegin(GL\_LINES)

    glColor3f(0.0, 1.0, 1.0)

    glVertex2f(st\_point[0], st\_point[1])

    glVertex2f(end\_point[0], end\_point[1])

    glEnd()

    glBegin(GL\_LINES)

    glColor3f(1.0, 1.0, 1.0)

    glVertex2f(st\_tps[0], st\_tps[1])

    glVertex2f(end\_tps[0], end\_tps[1])

    glEnd()

def main():

    pg.init()

    pg.display.set\_mode((600, 600), DOUBLEBUF | OPENGL | GL\_RGB)

    pg.display.set\_caption("Scaling - COMP342 Computer Graphics Lab")

    gluPerspective(150, 1, 1, 10)

    glTranslatef(0.0, 0.0, -10)

    while True:

        for ev in pg.event.get():

            if ev.type == pg.QUIT:

                pg.quit()

                quit()

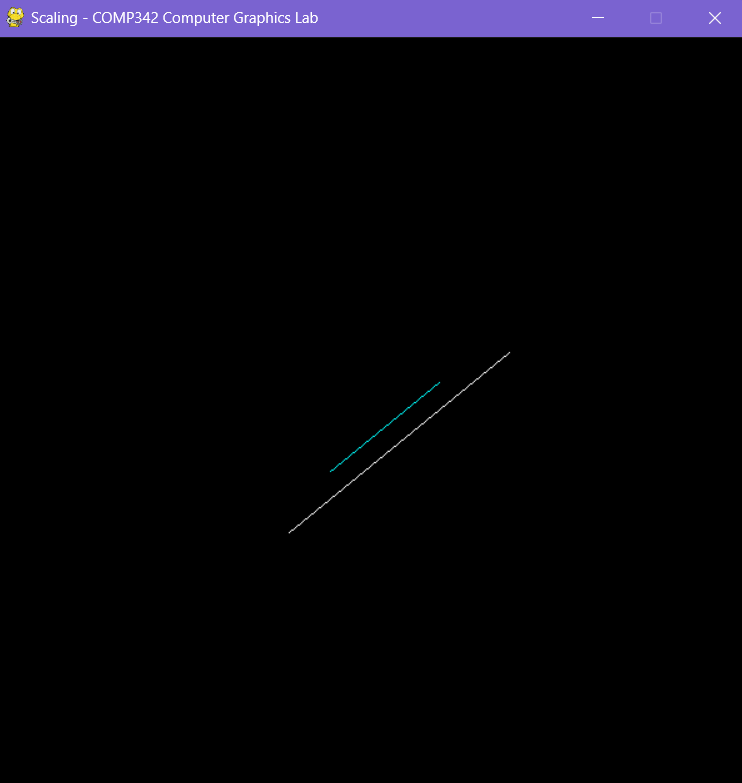
        displayPaint()

        pg.display.flip()

if \_\_name\_\_ == "\_\_main\_\_":

    main()

Output:



1. **2D Rotation:**

**Algorithm:**

1. Input:
   * Coordinates of the point to be rotated: (*x*,*y*)
   * Coordinates of the center of rotation: (*h*,*k*)
   * Rotation angle: *θ* (in degrees or radians)
2. Translate to Origin:
   * Translate the system so that the center of rotation becomes the origin: *x*′=*x*−*h, y*′=*y*−*k*
3. Rotate Around Origin:
   * Use the following formulas to calculate the new coordinates (*x*′′,*y*′′) after rotation aroundthe translated origin: *x*′′=*x*′⋅cos(*θ*)−*y*′⋅sin(*θ*), *y*′′=*x*′⋅sin(*θ*)+*y*′⋅cos(*θ*)
4. Translate Back:
   * Translate the system back to its original position: *x*′′′=*x*′′+*h, y*′′′=*y*′′+*k*
5. Output:
   * The new coordinates (*x*′′′,*y*′′′) represent the result of the rotation.

**Source Code:**

import numpy as np

from typing import Tuple

import pygame as pg

from pygame.locals import \*

from OpenGL.GL import \*

from OpenGL.GLU import \*

from math import \*

Coordinate = Tuple[float, float]

def rotate(point: Coordinate, rotateBy: float) -> Coordinate:

    x, y = point

    m = np.array([[x], [y], [1]])

    a = np.deg2rad(rotateBy)

    tx\_m = np.array([[np.cos(a), -np.sin(a), 0], [np.sin(a), np.cos(a), 0], [0, 0, 1]])

    result = np.dot(tx\_m, m)

    return tuple(result[:2, 0])

def displayPaint():

    st\_point: Coordinate = (2, 2)

    end\_point: Coordinate = (4, 6)

    rotateByAngle = 180

    st\_tps = rotate(st\_point, rotateByAngle)

    end\_tps = rotate(end\_point, rotateByAngle)

    glBegin(GL\_LINES)

    glColor3f(1.0, 1.0, 0.0)

    glVertex2f(st\_point[0], st\_point[1])

    glVertex2f(end\_point[0], end\_point[1])

    glEnd()

    glBegin(GL\_LINES)

    glColor3f(1.0, 1.0, 1.0)

    glVertex2f(st\_tps[0], st\_tps[1])

    glVertex2f(end\_tps[0], end\_tps[1])

    glEnd()

def main():

    pg.init()

    pg.display.set\_mode((600, 600), DOUBLEBUF | OPENGL | GL\_RGB)

    pg.display.set\_caption("Rotation - COMP342 Computer Graphics Lab")

    gluPerspective(260, 1, 1, 10)

    glTranslatef(0.0, 0.0, -10)

    while True:

        for ev in pg.event.get():

            if ev.type == pg.QUIT:

                pg.quit()

                quit()

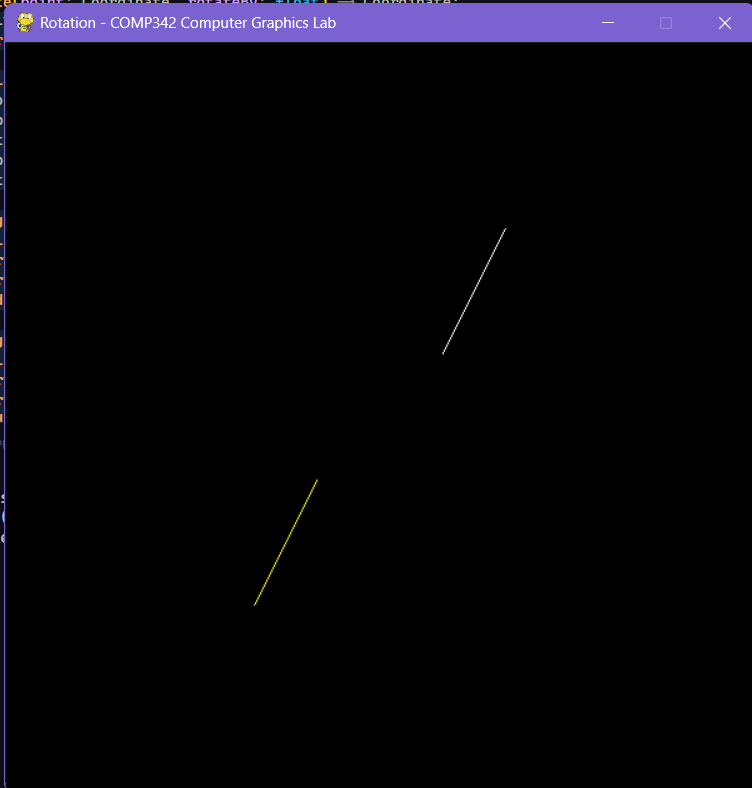
        displayPaint()

        pg.display.flip()

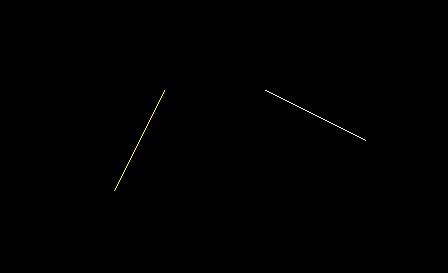
if \_\_name\_\_ == "\_\_main\_\_":

    main()

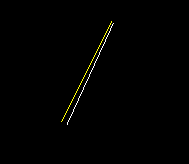
Output:(180 degree)



90 Degree



2 Degree



1. **Reflection**

**Algorithm:**

1. Input:
   * Read the set of 2D points.
2. Initialize:
   * Set up an empty list to store the transformed points.
3. Reflection Transformation:
   * For each point (x, y) in the original set:
     + Compute the reflected point (x', y') using the formula:
       - *x*′=*x*
       - *y*′=−*y*
4. Store Transformed Points:
   * Add the reflected point (x', y') to the list of transformed points.
5. Output:
   * The list of transformed points represents the 2D reflection along the x-axis.

**Source Code:**

import pygame

from pygame.locals import \*

from OpenGL.GL import \*

from math import cos, sin, radians

*# Initial coordinates of the line in homogeneous coordinates*

line = [[-0.5, -0.5, 1], [0.5, -0.5, 1]]

line1 = [[], []]

reflection\_axis = 'x'

def draw\_line(line, color):

    glBegin(GL\_LINES)

    glColor3f(color[0], color[1], color[2])  *# line color*

    glVertex2f(line[0][0] / line[0][2], line[0][1] / line[0][2])

    glVertex2f(line[1][0] / line[1][2], line[1][1] / line[1][2])

    glEnd()

def reflect(line):

    if reflection\_axis == 'x':

        reflection\_matrix = [

            [1, 0, 0],

            [0, -1, 0],

            [0, 0, 1]

        ]

    elif reflection\_axis == 'y':

        reflection\_matrix = [

            [-1, 0, 0],

            [0, 1, 0],

            [0, 0, 1]

        ]

    else:

        raise ValueError("Invalid reflection axis")

    new\_line = []

    for i in range(len(line)):

        x, y, w = line[i]

        new\_coords = [

            reflection\_matrix[0][0] \* x + reflection\_matrix[0][1] \* y +

            reflection\_matrix[0][2] \* w,

            reflection\_matrix[1][0] \* x + reflection\_matrix[1][1] \* y +

            reflection\_matrix[1][2] \* w,

            reflection\_matrix[2][0] \* x + reflection\_matrix[2][1] \* y +

            reflection\_matrix[2][2] \* w

        ]

        new\_line.append(new\_coords)

    return new\_line

def main():

    pygame.init()

    display = (800, 600)

    pygame.display.set\_mode(display, DOUBLEBUF | OPENGL)

    glMatrixMode(GL\_PROJECTION)

    glLoadIdentity()

    glOrtho(-1, 1, -1, 1, -1, 1)

    glMatrixMode(GL\_MODELVIEW)

    while True:

        for event in pygame.event.get():

            if event.type == pygame.QUIT:

                pygame.quit()

                quit()

        glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT)

        draw\_line(line, [1.0, 1.0, 1.0])  *# Original line in white*

        line1 = reflect(line)

        draw\_line(line1, [1.0, 0.0, 0.0])  *# Reflected line in red*

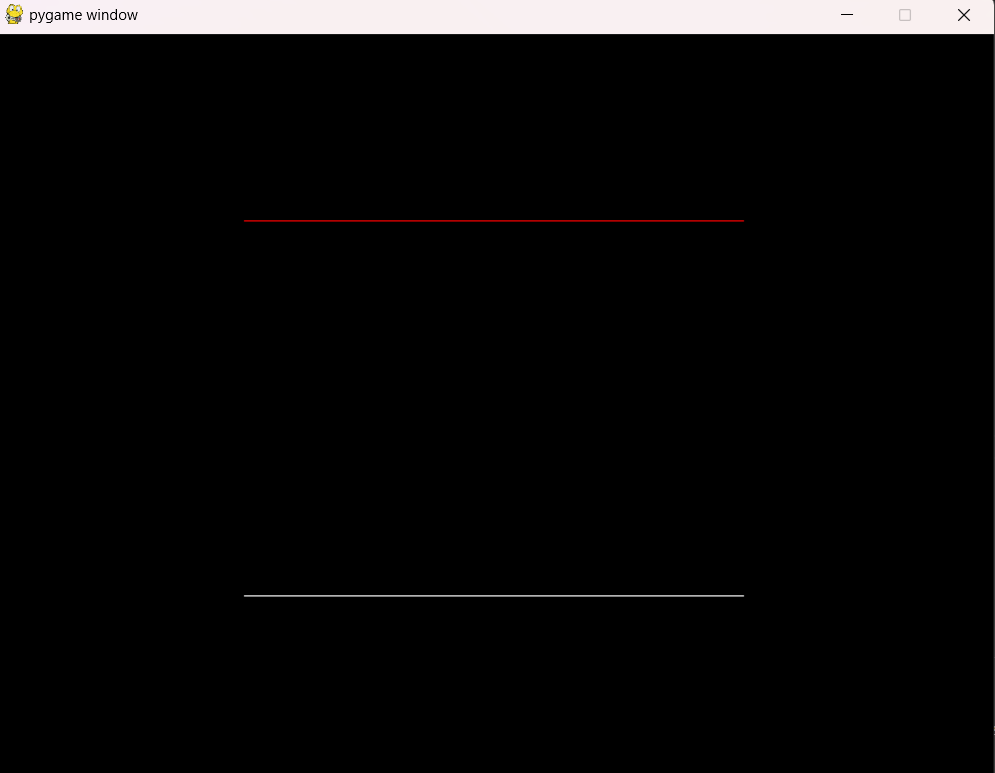
        pygame.display.flip()

        pygame.time.wait(10)

if \_\_name\_\_ == "\_\_main\_\_":

    main()

**Output:**



1. **Shearing**

**Algorithm:**

Horizontal Shear:

1. Input:
   * *x*,*y* coordinates of the point.
   * Shear factor *k*.
2. Calculate the new *x* coordinate in the sheared system: *x*′=*x*+*k*⋅*y*
3. Keep the *y* coordinate unchanged: *y*′=*y*
4. Output:
   * The new coordinates (*x*′,*y*′) represent the sheared point.

Vertical Shear:

1. Input:
   * *x*,*y* coordinates of the point.
   * Shear factor *k*.
2. Keep the *x* coordinate unchanged: *x*′=*x*
3. Calculate the new *y* coordinate in the sheared system: *y*′=y+*k*.*x*
4. Output:
   * The new coordinates (*x*′,*y*′) represent the sheared point.

**Source Code:**

import numpy as np

from typing import Tuple

import pygame as pg

from pygame.locals import \*

from OpenGL.GL import \*

from OpenGL.GLU import \*

from math import \*

Coordinate = Tuple[float, float]

def shear(point: Coordinate, shearX\_by: int, shearY\_by: int) -> Coordinate:

    x, y = point

    m = np.array([[x], [y], [1]])

    shearX\_m = np.array([[1, shearX\_by, 0], [0, 1, 0], [0, 0, 1]])

    shearY\_m = np.array([[1, 0, 0], [shearY\_by, 1, 0], [0, 0, 1]])

    composite\_m = np.dot(shearX\_m, shearY\_m)

    sheared\_m = np.dot(composite\_m, m)

    xT, yT, \_ = sheared\_m

    return (xT[0], yT[0])

def displayPaint():

    st\_point: Coordinate = (-4, -6)

    end\_point: Coordinate = (7, 3)

    shear\_By = (2, 1)

    st\_tps = shear(st\_point, shear\_By[0], shear\_By[1])

    end\_tps = shear(end\_point, shear\_By[0], shear\_By[1])

    glBegin(GL\_LINES)

    glColor3f(1.0, 1.0, 1.0)

    glVertex2f(st\_point[0], st\_point[1])

    glVertex2f(end\_point[0], end\_point[1])

    glEnd()

    glBegin(GL\_LINES)

    glColor3f(0.0, 1.0, 1.0)

    glVertex2f(st\_tps[0], st\_tps[1])

    glVertex2f(end\_tps[0], end\_tps[1])

    glEnd()

def main():

    pg.init()

    pg.display.set\_mode((600, 600), DOUBLEBUF | OPENGL | GL\_RGB)

    pg.display.set\_caption("Shearing - COMP342 Computer Graphics Lab")

    gluPerspective(150, 1, 1, 10)

    glTranslatef(0.0, 0.0, -10)

    while True:

        for ev in pg.event.get():

            if ev.type == pg.QUIT:

                pg.quit()

                quit()

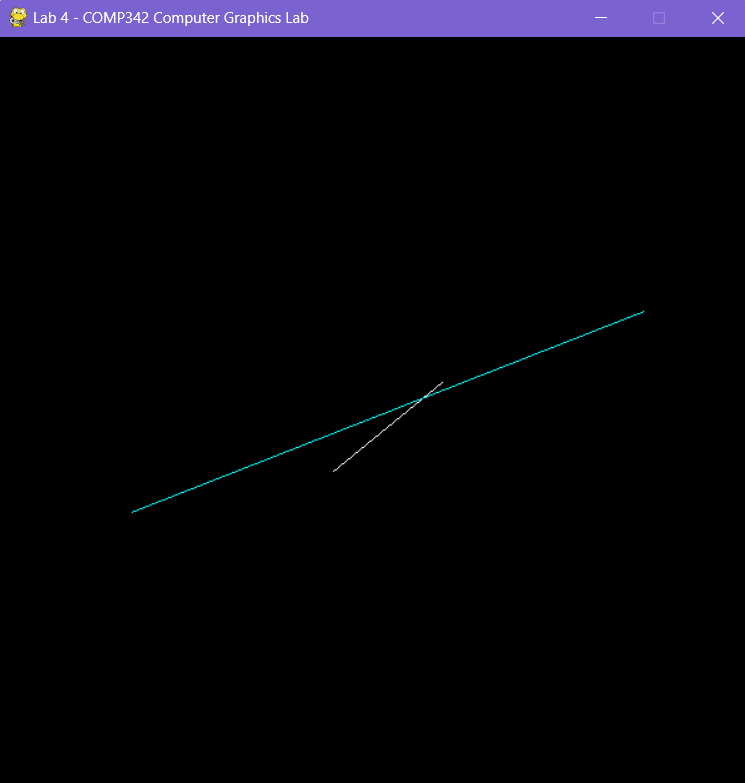
        displayPaint()

        pg.display.flip()

if \_\_name\_\_ == "\_\_main\_\_":

    main()

Output:



**Conclusion:**

In this lab, I delved into the world of 2D transformations using Python and OpenGL. I played around with translation, reflection, shearing, rotation, and scaling of geometric shapes in a 2D space. The hands-on work with transformation matrices and OpenGL's rendering tools emphasized the importance of grasping 2D transformation matrices in computer graphics. Connecting theory to application, I saw firsthand how these mathematical operations directly influenced graphical elements, showcasing their crucial role in crafting engaging images, particularly in the realm of digital art.