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

**Lab III**

**COMP 342**

**Submitted by:**

Manish Shivabhakti
Roll no: 63
CE, III year/ II Semester

**Date**: 29th May 2024

## Programming Language and Graphics Library

This project uses Python as the programming language. Graphics rendering and manipulation are facilitated by the Pygame, PyOpenGL, Math and Numpy for matrix operations libraries.

## Task 1: Midpoint Circle Drawing Algorithm

### Algorithm

- **Initialization:**
    - Start at the topmost point of the circle: x = 0, y = radius.
    - Calculate the initial decision parameter: d = 1 - radius.
- **Plot Points:**
    - While the y-coordinate is greater than or equal to the x-coordinate:
        - Plot points in all octants of the circle:
            - Plot points at (x, y), (y, x), (-x, y), (-y, x), (-x, -y), (-y, -x), (x, -y), and (y, -x).
        - If the decision parameter is less than or equal to 0:
            - Increment the decision parameter: d += 2 * x + 3.
        - Else:
            - Increment the decision parameter: d += 2 * (x - y) + 5.
            - Decrement the y-coordinate: y -= 1.
        - Increment the x-coordinate: x += 1.
- **Repeat:**
    - Repeat step 2 until the y-coordinate becomes less than the x-coordinate.

### Generating Points

```python
def drawCircleMidpoint(radius):
    x = 0
    y = radius
    d = 1 - radius  # Initial decision parameter

    # List to store the coordinates of points on the circle
    circle_points = []
    while y >= x:
        # Plot points in all octants
        circle_points.append((x, y))
        circle_points.append((y, x))
        circle_points.append((-x, y))
```

```
        circle_points.append((-y, x))
        circle_points.append((-x, -y))
        circle_points.append((-y, -x))
        circle_points.append((x, -y))
        circle_points.append((y, -x))
        if d <= 0:
            d += 2 * x + 3
        else:
            d += 2 * (x - y) + 5
            y -= 1
        x += 1
    return circle_points
```

## Plotting Points

```
def display(circle_points):
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)

    glColor3f(1.0, 0.0, 0.0)  # Red color
    glBegin(GL_POINTS)
    for x, y in circle_points:
        glVertex2f(x, y)
    glEnd()

    pygame.display.flip()
```

## Main Program

```
def main():
    pygame.init()
    pygame.display.set_mode((600, 600), DOUBLEBUF | OPENGL)
    pygame.display.set_caption('Circle Drawing using Midpoint
Algorithm')
    gluOrtho2D(-250, 250, -250, 250)

    glClearColor(1.0, 1.0, 1.0, 1.0)
    glColor3f(0.0, 0.0, 0.0)

    circle_points = drawCircleMidpoint(100)

    running = True
    while running:
        for event in pygame.event.get():
```

```
        if event.type == pygame.QUIT:
            running = False

    display(circle_points)
    pygame.time.wait(10)

pygame.quit()
```

## Output

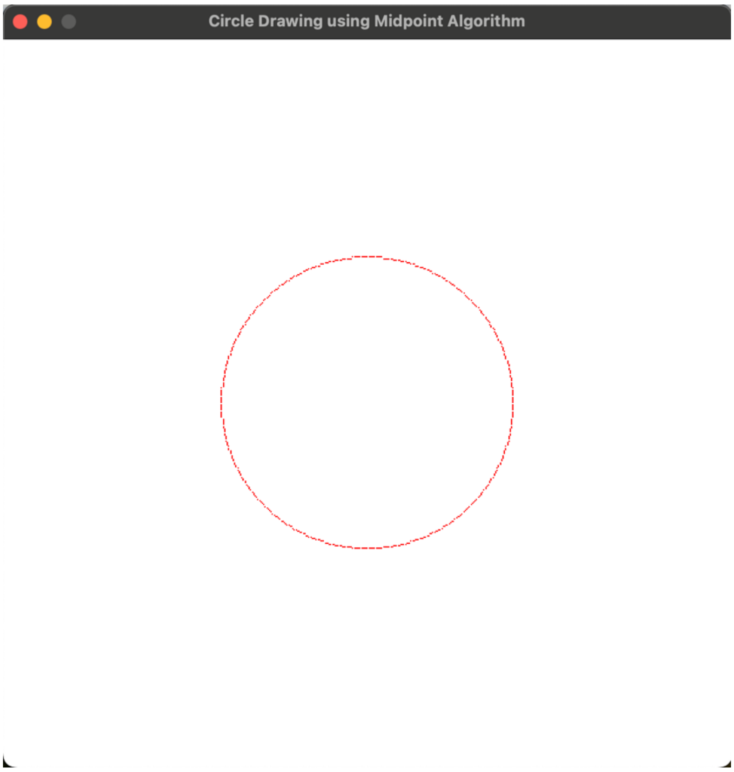


*Figure 1. Circle*

# Task 2: Midpoint Ellipse Drawing Algorithm

## Algorithm

- **Initialization:**

  - Start at the point (0, ry) on the ellipse, where rx and ry are the radii along the x and y axes, respectively.
  - Set initial values:
    - x = 0
    - y = ry
    - dx = 2 * ry^2 * x
    - dy = 2 * rx^2 * y
  - Calculate the initial decision parameter for region 1:
    - p1 = ry^2 - rx^2 * ry + 0.25 * rx^2

- **Plot Points in Region 1:**

  - While dx < dy:
    - Plot points at (x, y) and its symmetrical points in all four quadrants.
    - If the decision parameter p1 is non-negative:
      - Move to the next point by decrementing y and updating dy: y -= 1, dy -= 2 * rx^2.
    - Update dx and the decision parameter p1: x += 1, dx += 2 * ry^2, p1 += dx + ry^2.

- **Plot Points in Region 2:**

  - Calculate the initial decision parameter for region 2:
    - p2 = ry^2 * (x + 0.5)^2 + rx^2 * (y - 1)^2 - rx^2 * ry^2
  - While y >= 0:
    - Plot points at (x, y) and its symmetrical points in all four quadrants.
    - If the decision parameter p2 is positive or zero:
      - Move to the next point by incrementing x and updating dx: x += 1, dx += 2 * ry^2.
    - Decrement y and update dy: y -= 1, dy -= 2 * rx^2.
    - Update p2: p2 += dx - dy + rx^2.

- **Repeat:**

  - Repeat steps 2 and 3 until all points on the ellipse are plotted.

## Generating Points

```
def plotPoint(points, x, y, xc, yc):
    points.append((xc + x, yc + y))
    points.append((xc - x, yc + y))
```

```
    points.append((xc + x, yc - y))
    points.append((xc - x, yc - y))


def drawEllipseMidpoint(rx, ry, xc=0, yc=0):
    x = 0
    y = ry
    dx = 2 * ry * ry * x
    dy = 2 * rx * rx * y

    points = []

    p1 = ry * ry - rx * rx * ry + 0.25 * rx * rx

    while dx < dy:
        plotPoint(points, x, y, xc, yc)
        x += 1
        dx += 2 * ry * ry
        p1 += dx + ry * ry
        if p1 >= 0:
            y -= 1
            dy -= 2 * rx * rx
            p1 -= dy

    p2 = ry * ry * (x + 0.5) * (x + 0.5) + rx * rx * \
        (y - 1) * (y - 1) - rx * rx * ry * ry

    while y >= 0:
        plotPoint(points, x, y, xc, yc)
        y -= 1
        dy -= 2 * rx * rx
        p2 += rx * rx - dy
        if p2 <= 0:
            x += 1
            dx += 2 * ry * ry
            p2 += dx + rx * rx

    return points


def display(rx, ry, xc, yc):
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)

    points = drawEllipseMidpoint(rx, ry, xc, yc)
```

```
    glBegin(GL_POINTS)
    for point in points:
        glVertex2i(point[0], point[1])
    glEnd()

    pygame.display.flip()
```

## Main Program

```
def main():
    pygame.init()
    pygame.display.set_mode((600, 600), DOUBLEBUF | OPENGL)
    pygame.display.set_caption("Midpoint Ellipse Drawing Algorithm")
    gluOrtho2D(-100, 100, -100, 100)

    glClearColor(1.0, 1.0, 1.0, 1.0)
    glColor3f(0.0, 0.0, 0.0)

    running = True
    while running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False

        display(50, 30, 0, 0)
        pygame.time.wait(10)

    pygame.quit()
```

## Output

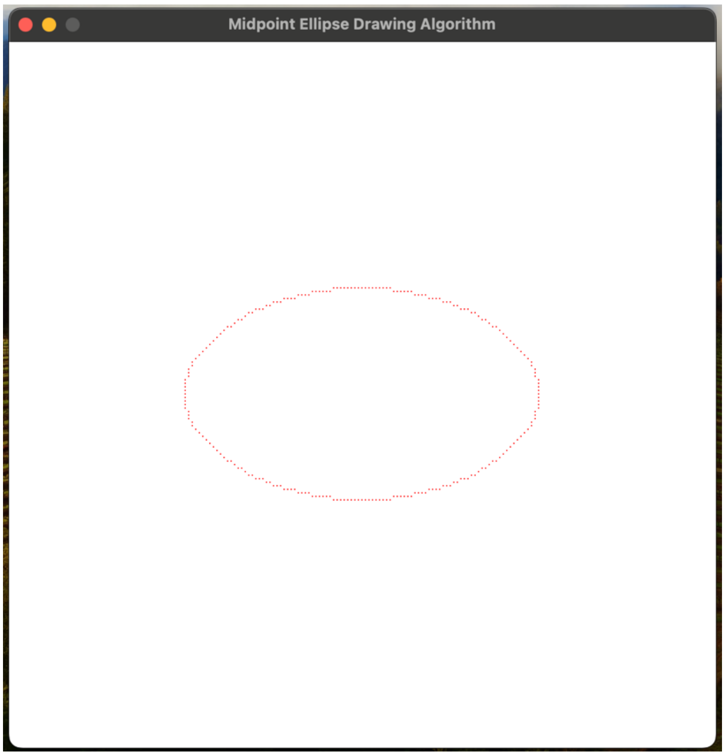


*Figure 2. Ellipse*

# Task 3.a: 2D Translation

## Algorithm

- **Initialization:**
    - Define the initial vertices of the object to be translated. These vertices represent the corners of a rectangle in this case.
- **Translation Matrix:**
    - Construct a translation matrix based on the translation distances in the x (tx) and y (ty) directions:

      ```
      | 1  0  tx |
      | 0  1  ty |
      | 0  0   1 |
      ```

    - Where tx and ty are the translation distances in the x and y directions, respectively.
- **Apply Translation:**
    - For each vertex of the object, apply the translation matrix to obtain the new translated coordinates:

      ```
      x_new = x + tx
      y_new = y + ty
      ```

    - Where (x, y) are the original coordinates of the vertex and (x_new, y_new) are the translated coordinates.
- **Update Display:**
    - Update the display with the translated vertices to visualize the translated object.
- **Repeat:**
    - Repeat steps 2 to 4 as necessary to perform further translations.

## Drawing Rectangle

```
def draw_rectangle():
    vertices = [
        [-0.5, -0.5, 1.0],
        [0.5, -0.5, 1.0],
        [0.5, 0.5, 1.0],
        [-0.5, 0.5, 1.0]
```

```
    ]

    glBegin(GL_QUADS)
    for vertex in vertices:
        glVertex2f(vertex[0], vertex[1])
    glEnd()
```

## Applying Translation

```
def apply_translation(vertices, tx, ty):
    translation_matrix = np.array([
        [1, 0, tx],
        [0, 1, ty],
        [0, 0, 1]
    ])

    translated_vertices = []
    for vertex in vertices:
        translated_vertex = np.dot(translation_matrix, vertex)
        translated_vertices.append(translated_vertex)

    return translated_vertices
```

## Display the vertices

```
def display(vertices):
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)

    glColor3f(1.0, 0.0, 0.0)  # Red color
    glBegin(GL_QUADS)
    for vertex in vertices:
        glVertex2f(vertex[0], vertex[1])
    glEnd()

    pygame.display.flip()
```

## Main Program

```
def main():
    pygame.init()
    pygame.display.set_mode((600, 600), DOUBLEBUF | OPENGL)
    pygame.display.set_caption('2D Translation')
    gluOrtho2D(-1, 1, -1, 1)

    glClearColor(1.0, 1.0, 1.0, 1.0)
```

```python
    glColor3f(0.0, 0.0, 0.0)

    vertices = [
        [-0.5, -0.5, 1.0],
        [0.5, -0.5, 1.0],
        [0.5, 0.5, 1.0],
        [-0.5, 0.5, 1.0]
    ]

    tx, ty = 0.0, 0.0  # Initial translation values

    running = True
    while running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False
            elif event.type == pygame.KEYDOWN:
                if event.key == pygame.K_LEFT:
                    tx -= 0.1
                elif event.key == pygame.K_RIGHT:
                    tx += 0.1
                elif event.key == pygame.K_UP:
                    ty += 0.1
                elif event.key == pygame.K_DOWN:
                    ty -= 0.1

        translated_vertices = apply_translation(vertices, tx, ty)
        display(translated_vertices)
        pygame.time.wait(10)

    pygame.quit()
```

## Output

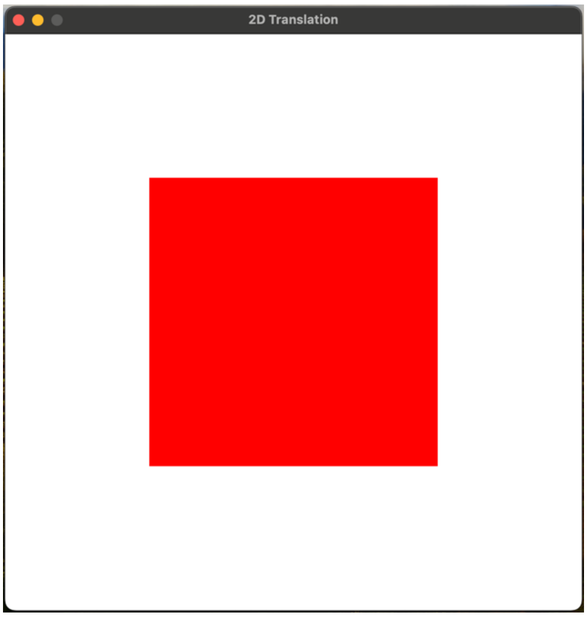


*Figure 3. Original*

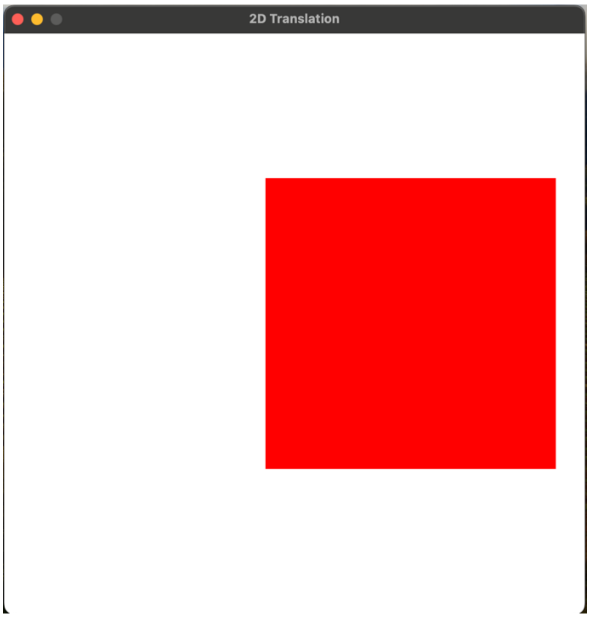


*Figure 4. Translated*

# Task 3.b: 2D Rotation

## Algorithm

- **Initialization:**
  - Define the initial vertices of the object to be rotated. These vertices represent the corners of a rectangle in this case.
- **Rotation Matrix:**
  - Construct a rotation matrix based on the rotation angle (theta) in radians:

```
| cos(theta)   -sin(theta)   0 |
| sin(theta)    cos(theta)   0 |
|     0             0        1 |
```

  - Where cos(theta) and sin(theta) are the cosine and sine of the rotation angle, respectively.
- **Apply Rotation:**
  - For each vertex of the object, apply the rotation matrix to obtain the new rotated coordinates:

```
x_new = x * cos(theta) - y * sin(theta)
y_new = x * sin(theta) + y * cos(theta)
```

  - Where (x, y) are the original coordinates of the vertex and (x_new, y_new) are the rotated coordinates.
- **Update Display:**
  - Update the display with the rotated vertices to visualize the rotated object.
- **Repeat:**
  - Repeat steps 2 to 4 as necessary to perform further rotations.

## Drawing Rectangle and Rotation function

```
def draw_rectangle():
    vertices = [
        [-0.5, -0.5, 1.0],
        [0.5, -0.5, 1.0],
        [0.5, 0.5, 1.0],
```

```python
        [-0.5, 0.5, 1.0]
    ]

    glBegin(GL_QUADS)
    for vertex in vertices:
        glVertex2f(vertex[0], vertex[1])
    glEnd()


def apply_rotation(vertices, angle):
    rotation_matrix = np.array([
        [math.cos(angle), -math.sin(angle), 0],
        [math.sin(angle), math.cos(angle), 0],
        [0, 0, 1]
    ])

    rotated_vertices = []
    for vertex in vertices:
        rotated_vertex = np.dot(rotation_matrix, vertex)
        rotated_vertices.append(rotated_vertex)

    return rotated_vertices
```

## Displaying the vertices

```python
def display(vertices):
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)

    glColor3f(1.0, 0.0, 0.0)  # Red color
    glBegin(GL_QUADS)
    for vertex in vertices:
        glVertex2f(vertex[0], vertex[1])
    glEnd()

    pygame.display.flip()
```

## Main Program

```python
def main():
    pygame.init()
    pygame.display.set_mode((600, 600), DOUBLEBUF | OPENGL)
    pygame.display.get_caption('2D Rotation')
    gluOrtho2D(-1, 1, -1, 1)

    glClearColor(1.0, 1.0, 1.0, 1.0)
```

```
    glColor3f(0.0, 0.0, 0.0)

    vertices = [
        [-0.5, -0.5, 1.0],
        [0.5, -0.5, 1.0],
        [0.5, 0.5, 1.0],
        [-0.5, 0.5, 1.0]
    ]

    angle = 0.0  # Initial rotation angle

    running = True
    while running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False
            elif event.type == pygame.KEYDOWN:
                if event.key == pygame.K_LEFT:
                    angle -= math.radians(10)
                elif event.key == pygame.K_RIGHT:
                    angle += math.radians(10)

        rotated_vertices = apply_rotation(vertices, angle)
        display(rotated_vertices)
        pygame.time.wait(10)

    pygame.quit()
```

## Output

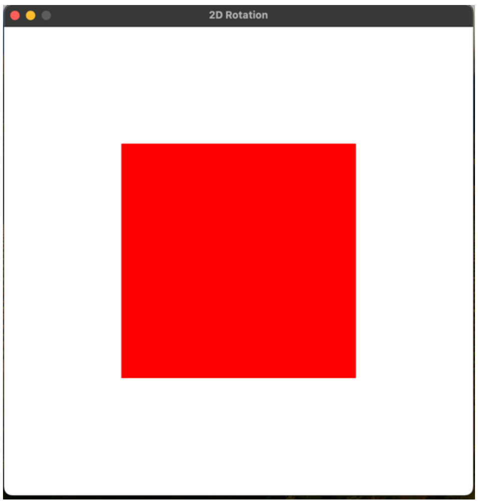


*Figure 5. Original*

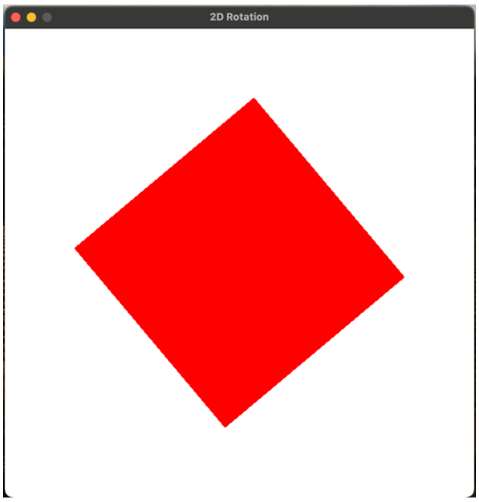


*Figure 6 Rotated*

# Task 3.c: 2D Scaling

## Algorithm

- **Initialization:**
  - Define the initial vertices of the object to be scaled. These vertices represent the corners of a rectangle in this case.
- **Scaling Matrix:**
  - Construct a scaling matrix based on the scaling factors (sx, sy) along the x and y axes:

    ```
    | sx   0   0 |
    | 0   sy   0 |
    | 0    0   1 |
    ```

  - Where sx and sy are the scaling factors along the x and y axes, respectively.
- **Apply Scaling:**
  - For each vertex of the object, apply the scaling matrix to obtain the new scaled coordinates:

    ```
    x_new = x * sx
    y_new = y * sy
    ```

  - Where (x, y) are the original coordinates of the vertex and (x_new, y_new) are the scaled coordinates.
- **Update Display:**
  - Update the display with the scaled vertices to visualize the scaled object.
- **Repeat:**
  - Repeat steps 2 to 4 as necessary to perform further scaling.

## Scaling Function

```
def apply_scaling(vertices, sx, sy):
    scaling_matrix = np.array([
        [sx, 0, 0],
        [0, sy, 0],
        [0, 0, 1]
    ])
```

```
    scaled_vertices = []
    for vertex in vertices:
        scaled_vertex = np.dot(scaling_matrix, vertex)
        scaled_vertices.append(scaled_vertex)

    return scaled_vertices
```

## Displaying the vertices

```
def display(vertices):
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)

    glColor3f(1.0, 0.0, 0.0)  # Red color
    glBegin(GL_QUADS)
    for vertex in vertices:
        glVertex2f(vertex[0], vertex[1])
    glEnd()

    pygame.display.flip()
```

## Main Program

```
def main():
    pygame.init()
    pygame.display.set_mode((800, 600), DOUBLEBUF | OPENGL)
    pygame.display.set_caption('2D Scaling')
    gluOrtho2D(-1, 1, -1, 1)

    glClearColor(1.0, 1.0, 1.0, 1.0)
    glColor3f(0.0, 0.0, 0.0)

    vertices = [
        [-0.5, -0.5, 1.0],
        [0.5, -0.5, 1.0],
        [0.5, 0.5, 1.0],
        [-0.5, 0.5, 1.0]
    ]

    sx, sy = 1.0, 1.0  # Initial scaling factors

    running = True
    while running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
```

```
                running = False
            elif event.type == pygame.KEYDOWN:
                if event.key == pygame.K_UP:
                    sx += 0.1
                    sy += 0.1
                elif event.key == pygame.K_DOWN:
                    sx -= 0.1
                    sy -= 0.1

        scaled_vertices = apply_scaling(vertices, sx, sy)
        display(scaled_vertices)
        pygame.time.wait(10)

    pygame.quit()
```

## Output

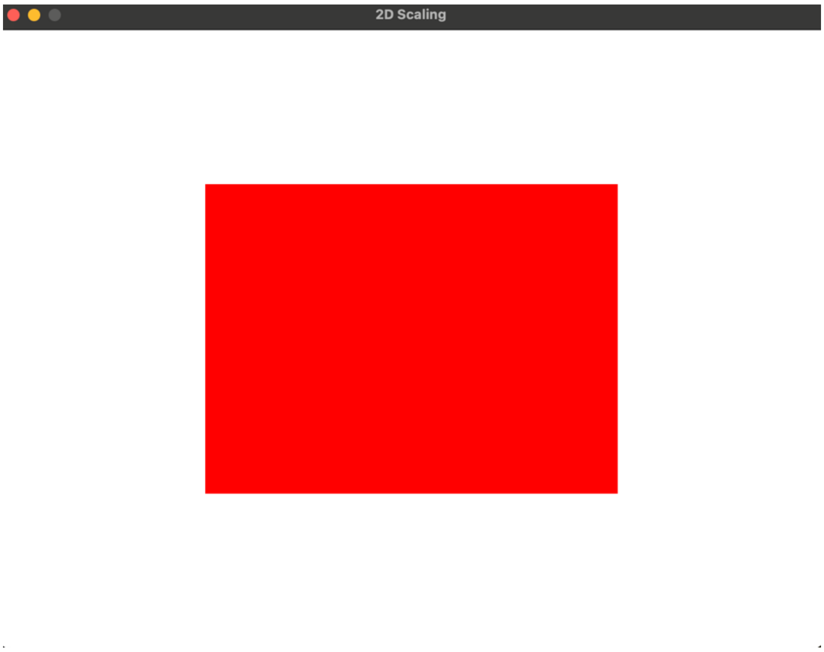


*Figure 7. Original Size*

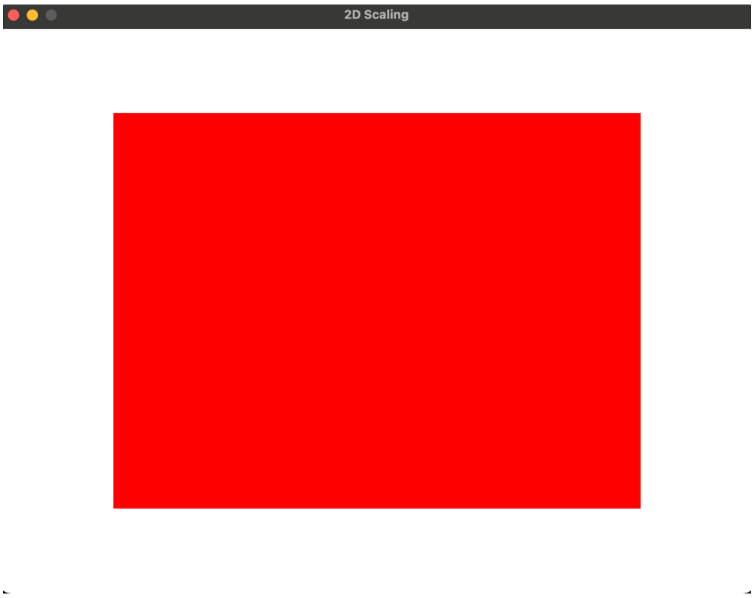


*Figure 8. Scaled up*

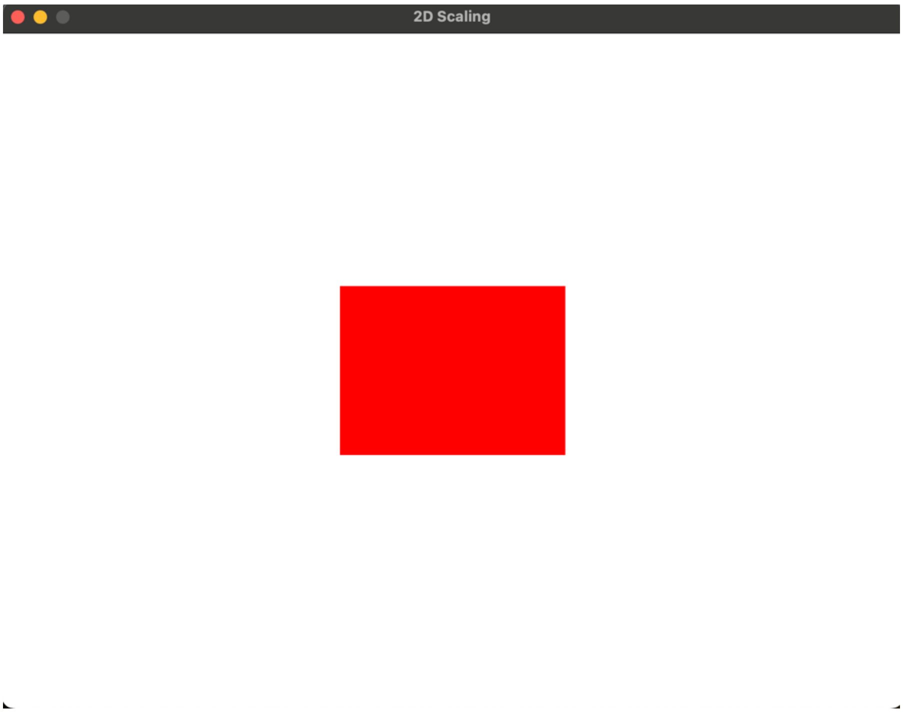


*Figure 9. Scaled Down*