ICS 2311: Group 15 - Ellipse Drawing, Transformation and Filling

Introduction

This document explains how we approached the ellipse drawing and transformation assignment for Group 15 in ICS 2311. We've implemented solutions using both C++ and Python with OpenGL and GLUT to handle:

- Drawing an ellipse from its mathematical equation
- Applying flood-fill algorithm with cyan color
- Implementing shear transformations
- Boundary filling the transformed ellipse with green color
- Anti-aliasing techniques for smooth elliptical boundaries

Code repository: [Here]

Understanding the Problem

The assignment asks us to:

1. Draw an ellipse with given by the equation

$$\frac{(x-2)^2}{36} + \frac{(y+1)^2}{25} = 1$$

2. Apply flood-fill algorithm to fill the ellipse with cyan color
3. Apply shear parameters (2 on X-axis, 2 on Y-axis) and find new coordinates
4. Boundary fill the transformed ellipse with green color
5. Develop anti-aliasing techniques for elliptical boundaries

Mathematical Foundation

Ellipse Equation

The standard form of an ellipse equation is: $(x-h)^2/a^2 + (y-k)^2/b^2 = 1$

Where:

- (h,k) is the center of the ellipse
- a is the semi-major axis length
- b is the semi-minor axis length

For our problem, the equation $(x-2)^2/36 + (y+1)^2/25 = 1$ gives us:

- Center at (h,k) = (2,-1)
- Semi-major axis a = 6 (sqrt(36))
- Semi-minor axis b = 5 (sqrt(25))

Parametric Form

To draw the ellipse, we use the parametric equation:

- $x = h + a * \cos(\theta)$
- $y = k + b * \sin(\theta)$

This allows us to generate points along the ellipse by varying θ from 0 to 2π.

Implementation Approach

1. Drawing the Original Ellipse

We begin by setting up the coordinate system and drawing the ellipse using the parametric equations:

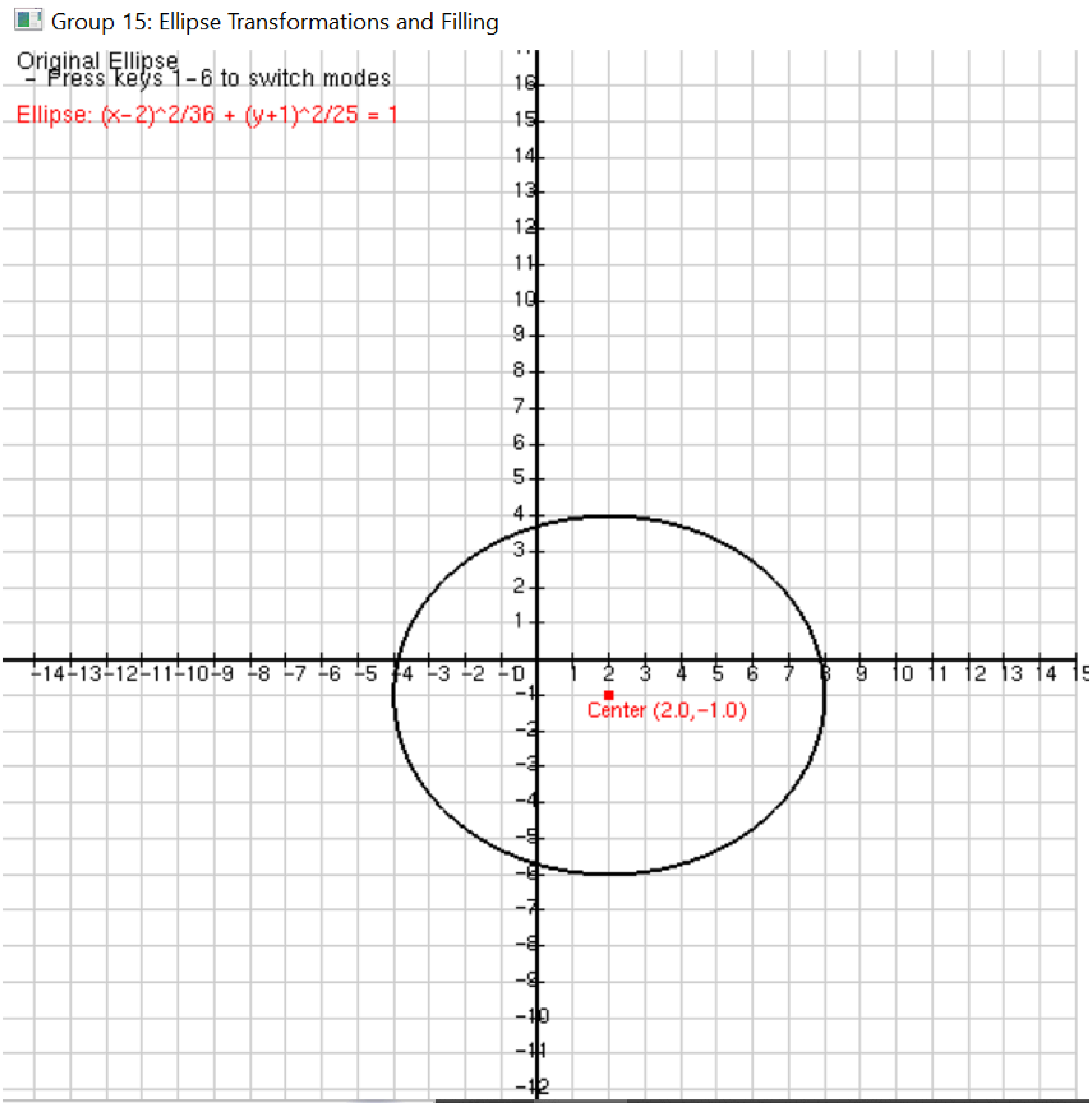
In Python:

```python
# Draw ellipse using parametric equation
glBegin(GL_LINE_LOOP)
for i in range(360):
    theta = i * math.pi / 180
    x = center_x + a * math.cos(theta)
    y = center_y + b * math.sin(theta)
    glVertex2f(x, y)
glEnd()
```

In C++:

```cpp
glBegin(GL_LINE_LOOP);
for (size_t i = 0; i < originalEllipsePoints.size(); i++) {
    glVertex2f(originalEllipsePoints[i].x, originalEllipsePoints[i].y);
}
glEnd();
```

Output:



2. Flood-Fill Algorithm (Cyan)

- **Seed Point**: Center of the original ellipse.
- **Fill Logic**: Iteratively check neighboring pixels against the ellipse equation
- **Efficiency**: Uses a stack-based approach to minimize redundant checks.

In C++:

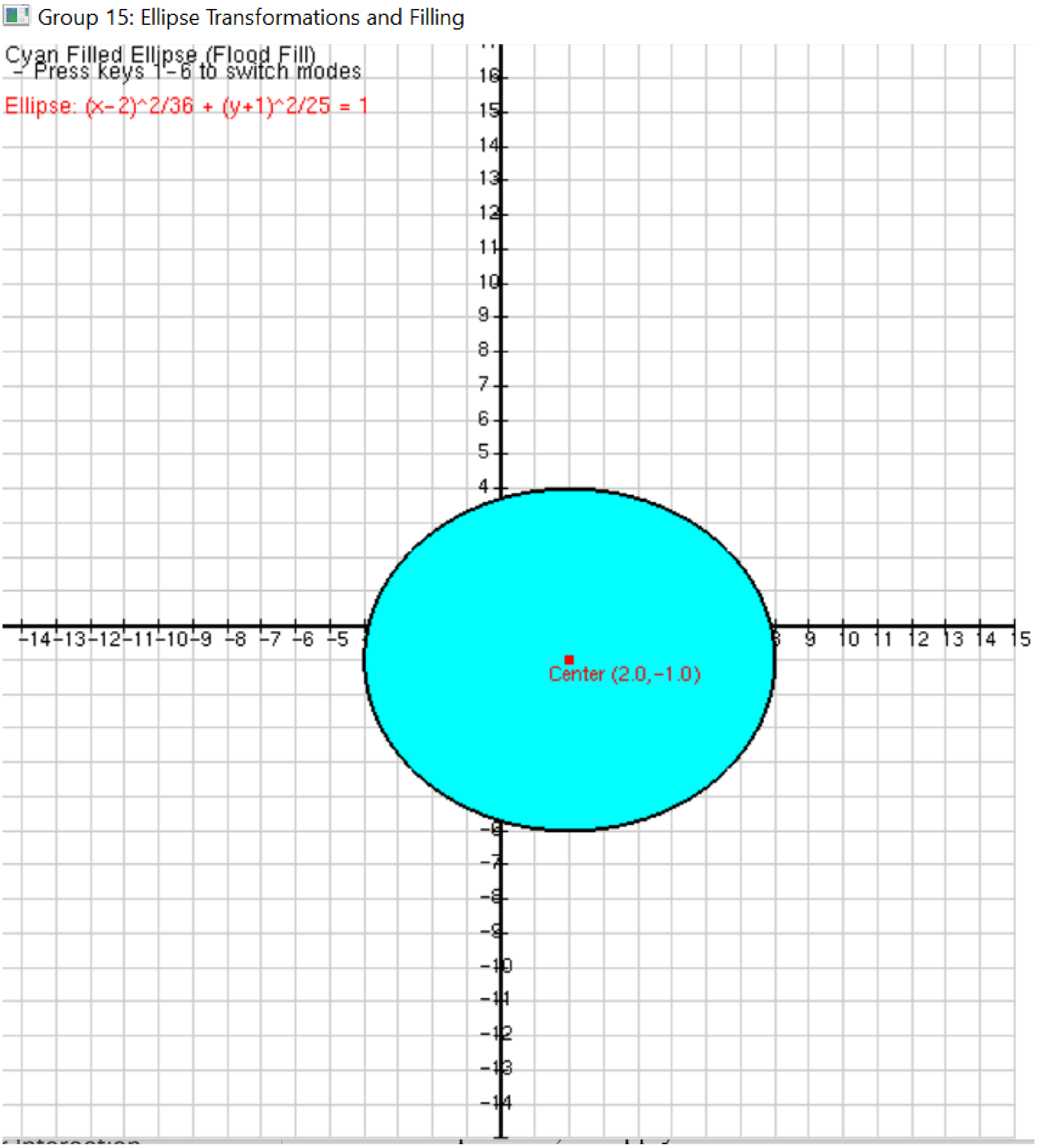
cpp

```cpp
// Flood-fill algorithm for the original ellipse (Cyan)
void floodFillOriginalEllipse() {
    // Cyan color
    float fillColor[3] = {0.0f, 1.0f, 1.0f}; // Cyan (RGB)

    // Start from the center of the ellipse (seed point)
    int seedX, seedY;
    worldToScreen(centerX, centerY, seedX, seedY);

    std::stack<Pixel> stack;
    stack.push(Pixel(seedX, seedY));

    // Create a buffer to track filled pixels
    bool** filled = new bool*[WINDOW_WIDTH];
    for (int i = 0; i < WINDOW_WIDTH; i++) {
        filled[i] = new bool[WINDOW_HEIGHT];
        for (int j = 0; j < WINDOW_HEIGHT; j++) {
            filled[i][j] = false;
        }
    }

    glPointSize(1.0f);
    glColor3f(fillColor[0], fillColor[1], fillColor[2]);
    glBegin(GL_POINTS);

    while (!stack.empty()) {
        Pixel p = stack.top();
        stack.pop();

        int x = p.x;
        int y = p.y;

        // Skip if already filled or outside window
        if (x < 0 || x >= WINDOW_WIDTH || y < 0 || y >= WINDOW_HEIGHT || filled[x][y]) {
            continue;
        }

        // Convert to world coordinates to check if inside ellipse
        float wx, wy;
        screenToWorld(x, y, wx, wy);

        if (isInsideOriginalEllipse(wx, wy)) {
            filled[x][y] = true;
            glVertex2f(wx, wy);

            // Push neighboring pixels
            stack.push(Pixel(x + 1, y));
            stack.push(Pixel(x - 1, y));
            stack.push(Pixel(x, y + 1));
            stack.push(Pixel(x, y - 1));
        }
    }

    glEnd();

    // Clean up
    for (int i = 0; i < WINDOW_WIDTH; i++) {
        delete[] filled[i];
    }
    delete[] filled;
}
```

In Python:

```python
def draw_filled_ellipse():
    """Draw the original ellipse filled with cyan color using flood-fill"""
    # Start flood-fill from the center
    seed_x, seed_y = world_to_screen(center_x, center_y)

    # Track filled pixels
    filled = [[False] * window_height for _ in range(window_width)]
    stack = [(seed_x, seed_y)]

    glPointSize(1.0)
    glColor3f(0, 1, 1)  # Cyan
    glBegin(GL_POINTS)

    while stack:
        x, y = stack.pop()

        if not (0 <= x < window_width and 0 <= y < window_height) or filled[x][y]:
            continue

        wx, wy = screen_to_world(x, y)
        if is_inside_original_ellipse(wx, wy):
            filled[x][y] = True
            glVertex2f(wx, wy)

            # Add neighboring pixels
            stack.append((x + 1, y))
            stack.append((x - 1, y))
            stack.append((x, y + 1))
            stack.append((x, y - 1))

    glEnd()
```

Output:



3. Shear Transformation

The shear transformation displaces points proportionally along both axes. For parameters **shearX = 2** and **shearY = 2**:

- **New Coordinates:**

$$x_{\text{new}} = x + 2y, \quad y_{\text{new}} = y + 2x$$

- **Inverse Transformation:**

  To check if a pixel lies inside the sheared ellipse, we reverse the shear using:

$$x_{\text{original}} = \frac{x - 2y}{-3}, \quad y_{\text{original}} = \frac{y - 2x}{-3}$$

  This ensures points are correctly mapped to the original ellipse's coordinate system.

  In c++:

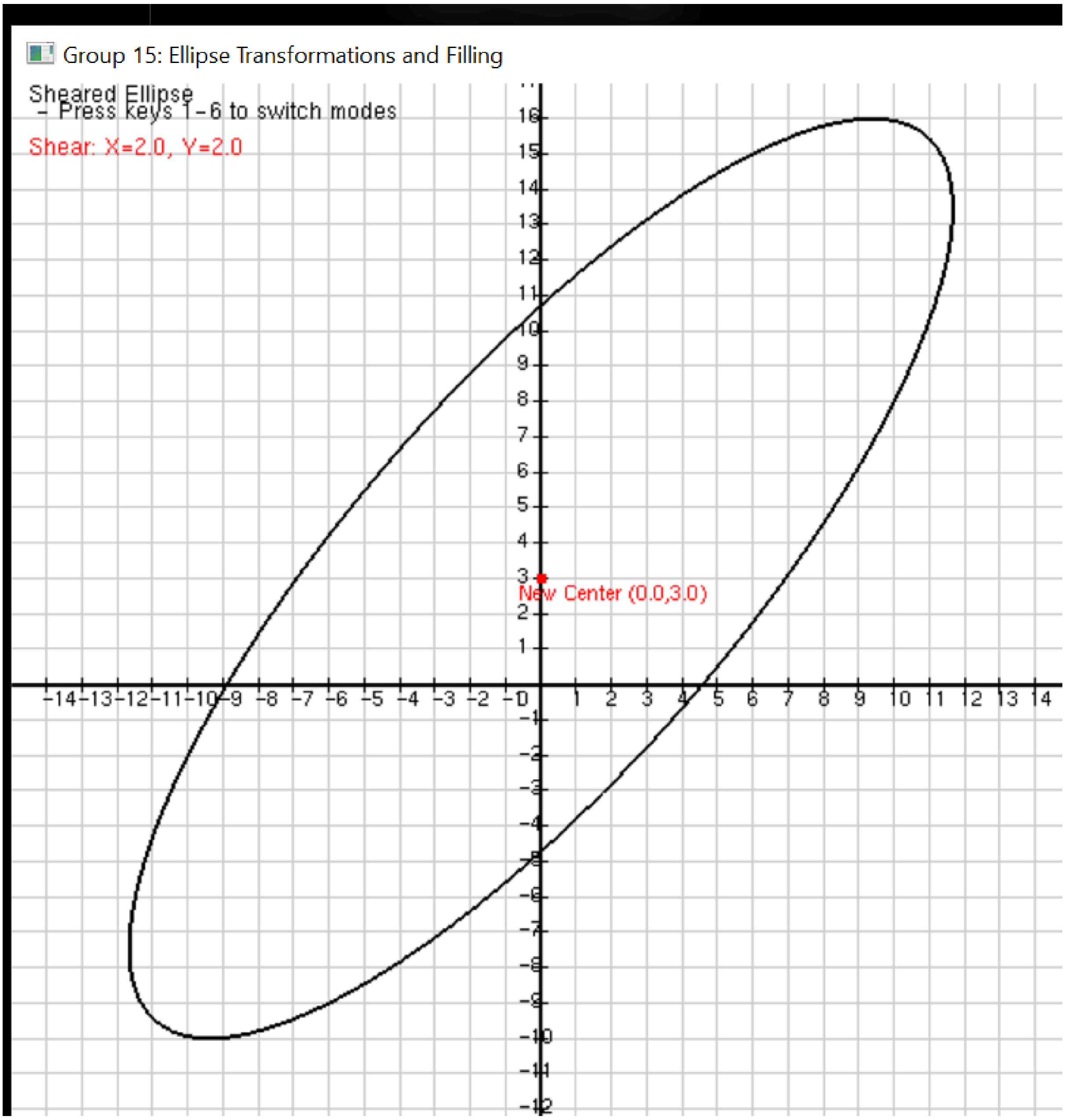
```cpp
// Apply shear transformation to ellipse points
void applyShearTransform() {
    shearedEllipsePoints.clear();

    for (size_t i = 0; i < originalEllipsePoints.size(); i++) {
        float x = originalEllipsePoints[i].x;
        float y = originalEllipsePoints[i].y;

        // Apply shear transformation
        float new_x = x + shearX * y;
        float new_y = y + shearY * x;

        shearedEllipsePoints.push_back(Point(new_x, new_y));
    }
}
```

  In Python:

python

```python
def draw_sheared_ellipse():
    """Draw the ellipse after applying shear transformation"""
    # Calculate points of the original ellipse and apply shear transformation
    sheared_points = []
    for i in range(360):
        theta = i * math.pi / 180
        x = center_x + a * math.cos(theta)
        y = center_y + b * math.sin(theta)

        # Apply shear transformation
        new_x = x + shear_x * y
        new_y = y + shear_y * x

        sheared_points.append((new_x, new_y))

    # Draw the sheared ellipse
    glColor3f(0, 0.5, 0)  # Dark green
    glLineWidth(2.0)
    glBegin(GL_LINE_LOOP)
    for x, y in sheared_points:
        glVertex2f(x, y)
    glEnd()
```

Output:



4. Boundary Fill Algorithm (Green)

How we implemented a boundary-fill algorithm :

Seed Point: Start at the sheared ellipse's center.

Inverse Shear: Transform screen coordinates to original ellipse space using the inverse shear.

Fill Check: Use the original ellipse equation to validate pixels.

Iterative Fill: Expand outward until the boundary is reached, ensuring the green fill aligns perfectly with the sheared shape.

In Python:

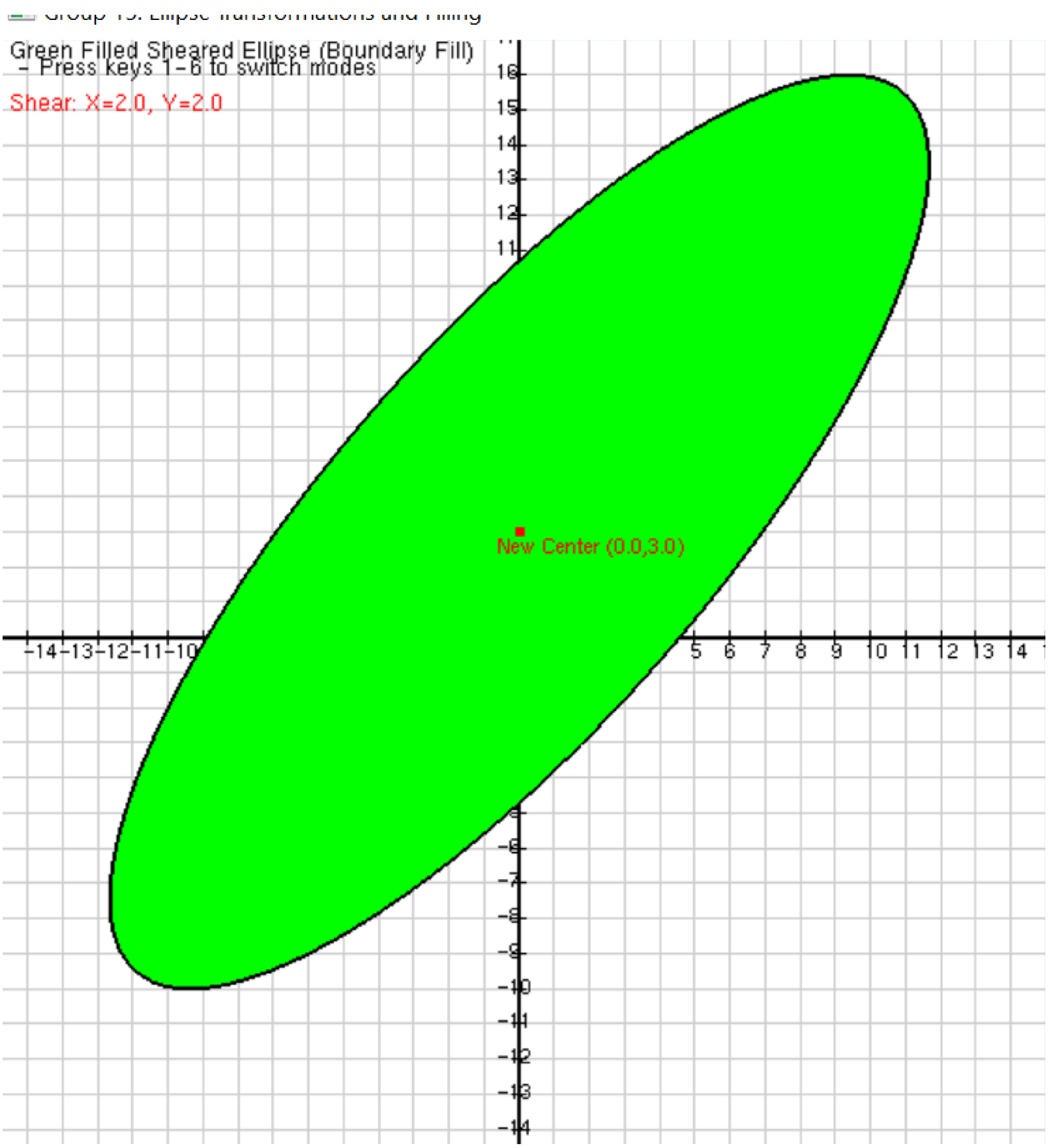
```python
def draw_filled_sheared_ellipse():
    """Draw the sheared ellipse filled with green color using boundary-fill"""
    # Calculate new center after shear
    new_center_x = center_x + shear_x * center_y
    new_center_y = center_y + shear_y * center_x

    # Start boundary-fill from the new center
    seed_x, seed_y = world_to_screen(new_center_x, new_center_y)

    # Track filled pixels
    filled = [[False] * window_height for _ in range(window_width)]

    # Use an iterative approach with a stack
    stack = [(seed_x, seed_y)]

    # Fill the ellipse first
    glPointSize(1.0)
    glColor3f(0, 1, 0)  # Green
    glBegin(GL_POINTS)

    while stack:
        x, y = stack.pop()

        # Skip if outside window or already filled
        if not (0 <= x < window_width and 0 <= y < window_height) or filled[x][y]:
            continue

        # Convert to world coordinates
        wx, wy = screen_to_world(x, y)

        # Check if inside the sheared ellipse
        if is_inside_sheared_ellipse(wx, wy):
            filled[x][y] = True
            glVertex2f(wx, wy)

            # Add neighboring pixels
            stack.append((x + 1, y))
            stack.append((x - 1, y))
            stack.append((x, y + 1))
            stack.append((x, y - 1))

    glEnd()

    # Draw the boundary after filling to ensure it's visible
    draw_sheared_ellipse()

    # Label the ellipse
    glColor3f(0, 0, 0)  # Black
    glRasterPos2f(-14.5, 16.0)
    for c in "Boundary-Fill Sheared Ellipse (Green)":
        glutBitmapCharacter(GLUT_BITMAP_9_BY_15, ord(c))
```

In C++:

cpp

```cpp
//  Boundary-fill algorithm for the sheared ellipse (Green)
void boundaryFillShearedEllipse() {
    // Green color for filling
    float fillColor[3] = {0.0f, 1.0f, 0.0f}; // Green (RGB)

    // Calculate transformed center
    float transformedCenterX = centerX + shearX * centerY;
    float transformedCenterY = centerY + shearY * centerX;

    // Use an iterative approach with a stack
    std::stack<Pixel> stack;
    int seedX, seedY;
    worldToScreen(transformedCenterX, transformedCenterY, seedX, seedY);
    stack.push(Pixel(seedX, seedY));

    // Create a buffer to track filled pixels
    bool** filled = new bool*[WINDOW_WIDTH];
    for (int i = 0; i < WINDOW_WIDTH; i++) {
        filled[i] = new bool[WINDOW_HEIGHT];
        for (int j = 0; j < WINDOW_HEIGHT; j++) {
            filled[i][j] = false;
        }
    }

    // Fill the ellipse first
    glPointSize(1.0f);
    glColor3f(fillColor[0], fillColor[1], fillColor[2]);
    glBegin(GL_POINTS);

    while (!stack.empty()) {
        Pixel p = stack.top();
        stack.pop();

        int x = p.x;
        int y = p.y;

        // Skip if outside window or already filled
        if (x < 0 || x >= WINDOW_WIDTH || y < 0 || y >= WINDOW_HEIGHT || filled[x][y]) {
            continue;
        }

        // Convert to world coordinates
        float wx, wy;
        screenToWorld(x, y, wx, wy);

        // Check if inside the sheared ellipse
        if (isInsideShearedEllipse(wx, wy)) {
            filled[x][y] = true;
            glVertex2f(wx, wy);

            // Push neighboring pixels
            stack.push(Pixel(x + 1, y));
            stack.push(Pixel(x - 1, y));
            stack.push(Pixel(x, y + 1));
            stack.push(Pixel(x, y - 1));
        }
    }

    glEnd();

    // Draw the boundary after filling to ensure it's visible
    drawShearedEllipse();

    // Clean up
    for (int i = 0; i < WINDOW_WIDTH; i++) {
        delete[] filled[i];
    }
    delete[] filled;
}
```

Output:



5. Anti-Aliasing Techniques

We implemented two anti-aliasing techniques:

a. OpenGL's Built-in Anti-aliasing:

We enable OpenGL's built-in line smoothing capabilities to achieve anti-aliasing.

In Python:

```python
glEnable(GL_LINE_SMOOTH)
glEnable(GL_BLEND)
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)
glHint(GL_LINE_SMOOTH_HINT, GL_NICEST)
```

In C++:

```cpp
glEnable(GL_LINE_SMOOTH);
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glHint(GL_LINE_SMOOTH_HINT, GL_NICEST);
```

b. Custom Anti-aliasing Implementation:

We also implemented a custom anti-aliasing technique that simulates the smoothing effect by drawing multiple versions of the ellipse with varying opacity levels.

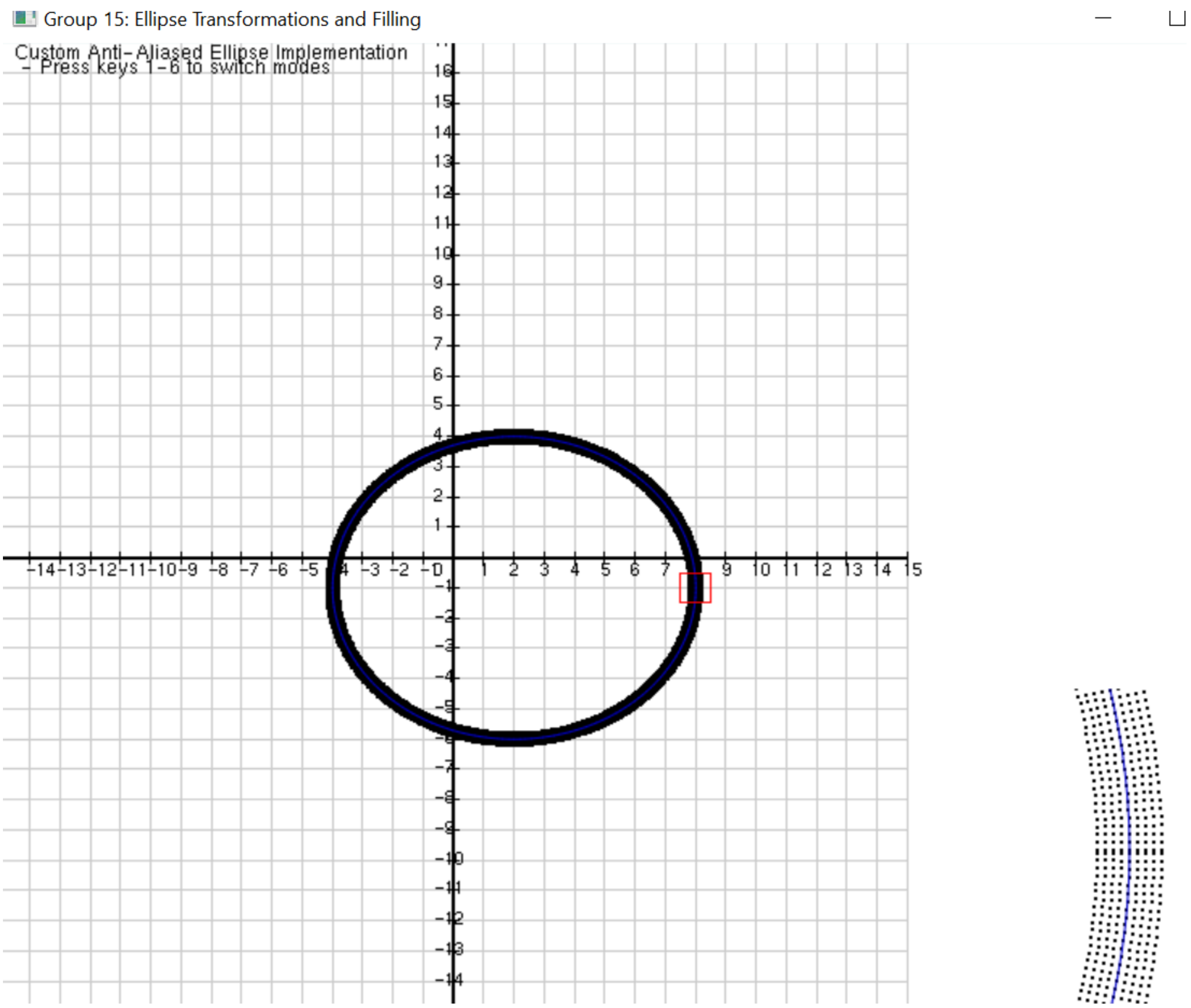
In Python:

```python
# Draw multiple ellipses with varying alpha values
for thickness in range(5, 0, -1):
    alpha = thickness / 5.0
    glColor4f(0, 0, 1, alpha)  # Blue with alpha
```

In C++:

```cpp
// Draw adjacent points with decreasing opacity to create anti-aliased effect
for (float offset = 0.05f; offset <= 0.2f; offset += 0.05f) {
    float alpha = 1.0f - offset * 5.0f; // Decreasing opacity

    // Inner points
    glColor4f(0.0f, 0.0f, 0.0f, alpha);
    glVertex2f(exactX - offset * cos(angle), exactY - offset * sin(angle));

    // Outer points
    glVertex2f(exactX + offset * cos(angle), exactY + offset * sin(angle));
}
```

Output:



Our custom implementation enhances the smoothness of the ellipse boundary by:

1. Drawing multiple concentric ellipses with decreasing opacity
2. Using higher density points to reduce the jagged appearance
3. Adding a zoomed view to highlight the anti-aliasing effect

6. User Interaction

We implemented keyboard interaction to allow switching between different views:

In Python:

```python
def keyboard(key, x, y):
    """Handle keyboard input"""
    global display_mode

    if key == b'1':
        display_mode = 1
    elif key == b'2':
        display_mode = 2
    # Additional modes.
```

..

In C++:

```cpp
void keyboard(unsigned char key, int x, int y) {
    if (key >= '1' && key <= '6') {
        currentMode = key - '1';
        glutPostRedisplay();
    } else if (key == 27) { // ESC key
        exit(0);
    }
}
```

Our program supports the following keys:

- Press 1: View original ellipse
- Press 2: View cyan-filled ellipse
- Press 3: View sheared ellipse
- Press 4: View green-filled sheared ellipse
- Press 5: View OpenGL anti-aliased ellipse
- Press 6: View custom anti-aliased ellipse
- Press ESC: Exit the program

Conclusion

This project successfully demonstrates multiple computer graphics concepts:

1. Mathematical representation of an ellipse using the parametric form to generate points

$$\frac{(x-2)^2}{36} + \frac{(y+1)^2}{25} = 1$$

2. Transformation techniques shown through shear transformation with parameters (2,2)
3. Fill algorithms applied through flood-fill and boundary-fill visualizations
4. Anti-aliasing techniques implemented both using OpenGL's built-in capabilities and a custom approach