By the end of this lab, students will be able to:

- Apply 2D geometric transformations (translation, scaling, rotation, reflection, shear).
- Represent transformations using matrices and homogeneous coordinates.
- Perform concatenation of transformations.
- Implement transformations programmatically (optional: C/OpenGL, Python, or any language taught).

Geometric Transformations

Changing the position, size, or orientation of objects in 2D or 3D space by modifying their coordinates.

Transforming Models/Objects

Transforming an object means applying transformations to all its points; polygonal models are transformed by modifying their vertices.

Basic Transformation Functions

Translation: shift;
Scaling: resize;
Rotation: rotate;

2D Translation

(x', y') = (x + dx, y + dy). Shifts an object without changing shape/size.

2D Scaling from the origin

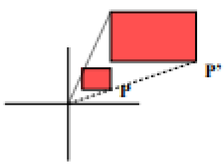
(x', y') = (sx * x, sy * y). Enlarges/shrinks about origin or fixed point.

Point P defined as $P(x, y)$,
Perform a scale (stretch) to Point $P'(x', y')$ by a factor s_x along the x axis,
and s_y along the y axis.

$$x' = s_x . x, \quad y' = s_y . y$$

Define the matrix

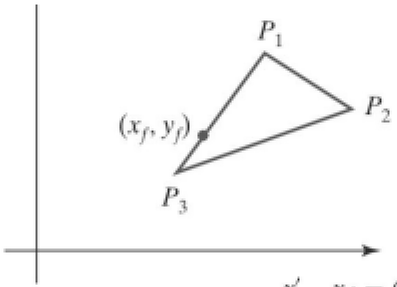
$$S = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$$



Now

$$P' = S \cdot P \quad \text{or} \quad \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

2D scaling with an unchanged point



$$x' - x_f = (x - x_f)s_x, \qquad y' - y_f = (y - y_f)s_y$$

$$x' = x \cdot s_x + x_f(1 - s_x)$$
$$y' = y \cdot s_y + y_f(1 - s_y)$$

where the additive terms $x_f(1 - s_x)$ and $y_f(1 - s_y)$ are constants for all points in the object.

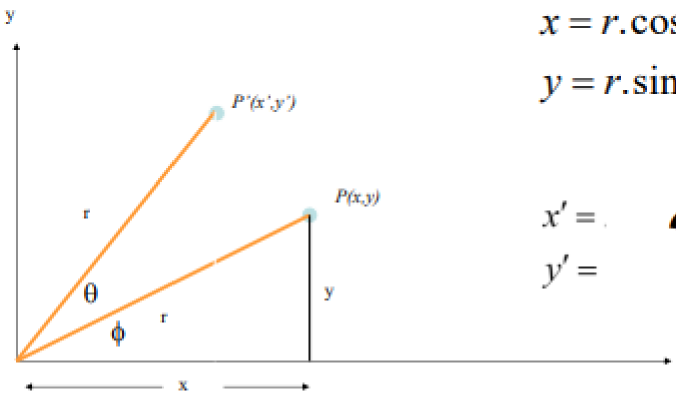
If you want to scale a point (x, y) **about a fixed point** (xf, yf) with scaling factors Sx and Sy, the combined formula is:

$$x' = Sx \cdot (x - xf) + xf$$

$$y' = Sy \cdot (y - yf) + yf$$

This combines **translation to origin**, **scaling**, and **translation back** into a single step.

2D Rotation about the origin



$$x = r.\cos\phi$$
$$y = r.\sin\phi$$

$x' =$

$y' =$?

$$x' = r.\cos(\theta + \phi) = r.\cos\phi.\cos\theta - r.\sin\phi.\sin\theta$$
$$y' = r.\sin(\theta + \phi) = r.\cos\phi.\sin\theta + r.\sin\phi.\cos\theta$$

Substituting for r :

$$x = r.\cos\phi$$
$$y = r.\sin\phi$$

Gives us :

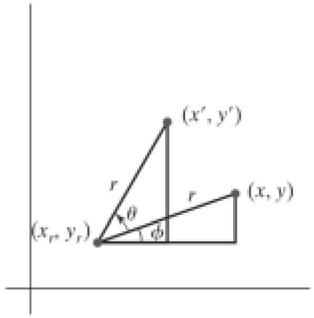
$$x' = x.\cos\theta - y.\sin\theta$$
$$y' = x.\sin\theta + y.\cos\theta$$

Rewriting in matrix form gives us :

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

Define the matrix $R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$, $P' = R \cdot P$

2D Rotation about an arbitrary point



$$x' = x_r + (x - x_r)\cos\theta - (y - y_r)\sin\theta$$
$$y' = y_r + (x - x_r)\sin\theta + (y - y_r)\cos\theta$$

Rule for matrix multiplication

Checking Compatibility: https://youtu.be/o6tGHLkZvVM?si=gkCh38DpMu-tpVZM
Multiplication of Matrices: https://youtu.be/RE-nDY2aWso?si=xyQ9xYWFcHSMOOMc

What are Homogeneous Coordinates?

- In simple terms, homogeneous coordinates are a way of representing N-dimensional coordinates with (N+1) numbers.
- For 2D graphics, we represent a point (x, y) as (x, y, w), where w is a scaling factor called the homogeneous component.
- The conventional 2D Cartesian coordinate is obtained by dividing by w:
  (x, y) in Cartesian space is represented as (x, y, 1) in homogeneous space.
  If you have a homogeneous point (x', y', w), you get the "real" position by calculating (x'/w, y'/w)

Translation (Homogeneous Coordinates)

Moves a point by a displacement vector (dx, dy):

$$(x', y') = (x + dx, y + dy)$$

Matrix Form (Homogeneous Coordinates):

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Scaling (Homogeneous Coordinates)

Scales a point relative to the origin by factors (s_x, s_y):

$$(x', y') = (s_x \cdot x, s_y \cdot y)$$

Matrix Form:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Rotation (Homogeneous Coordinates)

Rotates a point counterclockwise by angle θ about the origin:

$$x' = x \cdot \cos\theta - y \cdot \sin\theta$$
$$y' = x \cdot \sin\theta + y \cdot \cos\theta$$

Matrix Form:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Part A – Manual Exercises

Task 1: Translation

Question 1 : A quadrilateral has vertices at the points: P(0, 0), Q(6, 0), R(6, 4), and S(0, 4).
Apply a **translation** with the translation vector
Tx = 2 and Ty = -5.
What are the new coordinates of the quadrilateral after translation?

Question 2: On a map grid, a treasure chest is located at point T(150, 80). The instructions state: "Move 50 units East and 30 units North to find the key."
Assuming East is the positive x-direction and North is the positive y-direction, what are the coordinates of the key?

Question 3: After a translation, a triangle's vertices are at A'(0, -2), B'(3, 5), and C'(-1, 1). The translation vector used was Tx = -2, Ty = 4. What were the original coordinates of the triangle before the translation?

Task 2: Scaling

Question 4: A quadrilateral has vertices at P(2, 3), Q(5, 3), R(5, 6), and S(2, 6).
Apply scaling with Sx = 3 and Sy = 2 about the fixed point (4, 5).
Find the new coordinates of the quadrilateral.

Question 5: A triangle has vertices at A(4, 2), B(1, -1), and C(3, -3).

Apply scaling with Sx = 0.5 and Sy = 1.5 about the fixed point (0, 0).
Find the new coordinates of the triangle.

Task 3: Rotation

Question 6: A triangle has vertices at A(3, 2), B(5, 4), and C(6, 1).
Rotate the triangle by **180 degrees counterclockwise** about the origin (0, 0).
What are the new coordinates of the triangle after rotation?

Question 7: A triangle has vertices at A(2, 3), B(5, 3), and C(5, 6).
Rotate the triangle by **90 degrees counterclockwise** about the fixed point **(3, 3)**.
What are the new coordinates?

Part B – Coding Exercise

Task 4: Implement Basic Transformations

1. Implement each transformation function following the TODO comments
2. You have to,
    1. Draw a simple triangle
    2. Perform translation
    3. Perform scaling about origin
    4. Perform scaling about a fixed point
    5. Perform rotation about origin
    6. Perform rotation about a fixed point
3. Uncomment one transformation function at a time in main() to test it
4. Observe how each transformation affects the triangle

```
// Function to draw a simple  triangle
void drawTriangle(void) {
   // TODO: Set color to red (RGB values between 0.0 and 1.0)

   // TODO: Begin drawing triangles

   // TODO: Define three vertices to form a triangle
   // Bottom left vertex
   // Bottom right vertex
   // Top vertex

   // TODO: End drawing
}

// Translation transformation example
void Translate() {
   glClear(GL_COLOR_BUFFER_BIT);

   // TODO: Draw original triangle

   // TODO: Apply translation transformation
   // Save current transformation matrix
   // Translate by (0.5, 0.3, 0.0)
   // Draw translated triangle
   // Restore previous transformation matrix

   glutSwapBuffers();
```

```
}

// Scaling transformation about origin
void ScaleAboutOrigin() {
    glClear(GL_COLOR_BUFFER_BIT);

    // TODO: Draw the original triangle first for reference

    // TODO: Apply scaling transformation about origin
    // Scale by 2.0 in X, 1.5 in Y, 1.0 in Z
    // Draw scaled triangle

    glFlush();
}

// Scaling transformation about a fixed point
void ScaleAboutFixedPoint() {
    glClear(GL_COLOR_BUFFER_BIT);

    // TODO: Draw the original triangle first for reference

    // TODO: Implement scaling about a fixed point (0.25, 0.25)
    // Strategy: Move fixed point to origin, scale, then move back
    // Move fixed point to origin
    // Apply scaling
    // Move back to original position
    // Draw scaled triangle

    glFlush();
}

// Rotation transformation about origin
void RotationAboutOrigin() {
    glClear(GL_COLOR_BUFFER_BIT);

    // TODO: Draw the original triangle first for reference

    // TODO: Apply rotation transformation about origin
    // Rotate 45 degrees around Z-axis
    // Draw rotated triangle

    glFlush();
}

// Rotation about a fixed point (e.g., first vertex)
void RotationAboutFixedPoint() {
    glClear(GL_COLOR_BUFFER_BIT);

    // TODO: Draw the original triangle first for reference

    // TODO: Implement rotation about a fixed point (0.0, 0.0)
    // Strategy: Move fixed point to origin, rotate, then move back
    // Move fixed point to origin
    // Rotate 45 degrees around Z-axis
    // Move back to original position
    // Draw rotated triangle
```

```
    glFlush();
}

int main(int argc, char** argv) {
    // Initialize GLUT

    // TODO: Uncomment the function you want to test:
    // glutDisplayFunc(Translate);
    // glutDisplayFunc(ScaleAboutOrigin);
    // glutDisplayFunc(ScaleAboutFixedPoint);
    // glutDisplayFunc(RotationAboutOrigin);
    // glutDisplayFunc(RotationAboutFixedPoint);
    // Enter the main event loop
    glutMainLoop();

    return 0;
}
```

Task 5: Composite Transformation

1. Implement the solar system animation
2. Study how multiple transformations are combined
3. You will have to,
    1. Set rotation angles for animation
    2. Write a function to draw a circle
    3. Write a function to draw a coordinate grid
    4. In the display function create sun, earth and moon and perform transformations
4. Note - the use of glPushMatrix() and glPopMatrix()

How glPushMatrix() and glPopMatrix() work

1. **glPushMatrix()**: This function duplicates the matrix at the top of the current matrix stack (e.g., the modelview matrix) and places this duplicate on top of the stack.
2. **Transformations**: After calling glPushMatrix(), any subsequent transformation functions (glTranslate, glRotate, glScale, etc.) are applied to this newly pushed matrix.
3. **glPopMatrix()**: This function removes the matrix from the top of the stack. The matrix that was previously at the top (before the last glPushMatrix()) becomes the new current matrix.

Why they are used

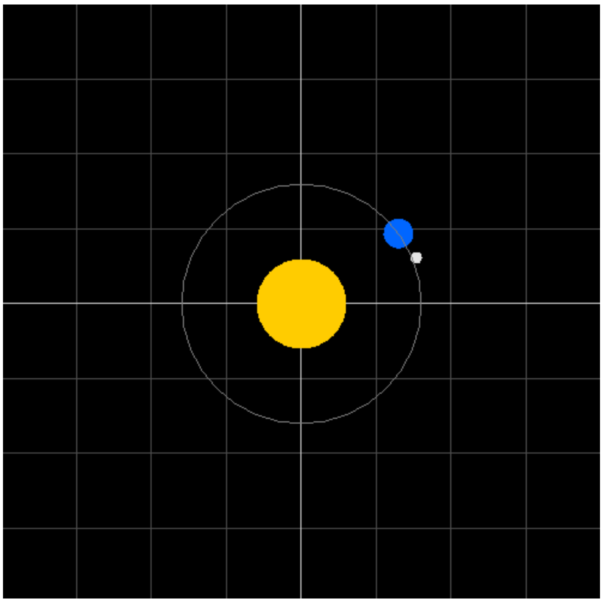
- **Hierarchical Modeling**: Think of a solar system where planets revolve around the sun.
    - You push the matrix to establish the sun's coordinate system.
    - You draw the sun.
    - You push again to establish the planet's coordinate system relative to the sun's.
    - You apply the planet's rotation and translation.
    - You draw the planet.
    - When you pop, you return to the sun's coordinate system, ready to draw the next planet without the first planet's transformations affecting it.
- **Controlling Transformation Scope**: You can apply a transformation to a single object or a group of objects and then easily revert to the previous transformation state without the changes affecting other parts of your scene.

```
// Rotation angles for animation
float earthOrbitAngle = 0.0f;
```

```
float earthRotationAngle = 0.0f;
float moonOrbitAngle = 0.0f;

// Function to draw a circle (used for Sun, Earth, Moon)
void drawCircle(float cx, float cy, float r, int num_segments, float red, float green, float blue) {
    // TODO: Set circle color

    // TODO: Begin drawing triangle fan

    // TODO: Define center of circle

    // TODO: Calculate and define circle vertices
    // For each segment, calculate vertex position using:
    // x = r * cos(theta) + cx
    // y = r * sin(theta) + cy

    // TODO: End drawing
}

// Function to draw a coordinate grid for reference
void drawGrid() {
    // TODO: Set grid color to gray

    // TODO: Begin drawing lines

    // TODO: Draw vertical lines from x = -2.0 to 2.0 in increments of 0.5

    // TODO: Draw horizontal lines from y = -2.0 to 2.0 in increments of 0.5

    // TODO: Draw main axes in a lighter color
    // X-axis from (-2.0, 0.0) to (2.0, 0.0)
    // Y-axis from (0.0, -2.0) to (0.0, 2.0)

    // TODO: End drawing
}

// Main display function for solar system animation
void display() {
    glClear(GL_COLOR_BUFFER_BIT);

    // Draw coordinate grid
    drawGrid();

    // Draw the Sun (fixed at origin, just scaled)
    // TODO: Save current matrix
    // TODO: Scale the sun to be larger (1.5 in both dimensions)
    // TODO: Draw yellow sun at origin with radius 0.2
    // TODO: Restore matrix

    // EARTH: Composite transformation example
    // TODO: Save current matrix state (identity)

    // TODO: Apply Earth's transformations:
    // 1. Orbit around Sun (rotation around origin)
    // 2. Translate to orbital distance (0.8, 0.0, 0.0)
    // 3. Earth's self-rotation
```

```
    // TODO: Draw Earth (blue) with radius 0.1

    // MOON: Nested composite transformation (relative to Earth)
    // TODO: Save Earth's transformation state

    // TODO: Apply Moon's transformations:
    // 1. Orbit around Earth (rotation around Earth's center)
    // 2. Translate to moon's orbital distance from Earth (0.2, 0.0, 0.0)

    // TODO: Draw Moon (white) with radius 0.04

    // TODO: Restore to Earth's transformation state
    // TODO: Restore to original matrix state (identity)

    // Draw orbital paths
    // TODO: Set orbit color to gray

    // TODO: Draw Earth's orbit (circle with radius 0.8)

    // Update angles for animation
    earthOrbitAngle += 0.2f;       // Earth orbits slowly
    earthRotationAngle += 1.0f;    // Earth rotates faster
    moonOrbitAngle += 0.5f;        // Moon orbits even faster

    glutSwapBuffers();
}

// Timer function for animation
void timer(int value) {
    // TODO: Trigger display function
    // TODO: Set up timer for ~60 FPS (1000ms/60 ≈ 16ms)
}

// Initialization function
void init() {
    // TODO: Set clear color to black

    // TODO: Set up orthographic projection
    // Set matrix mode to projection
    // Load identity matrix
    // Set up 2D orthographic projection from (-2,-2) to (2,2)
    // Set matrix mode to modelview
}

// Print instructions to console
void printInstructions() {
    printf("=== Composite Transformations Demo ===\n");
    printf("This program demonstrates:\n");
    printf("1. Scaling: The Sun is scaled to be larger\n");
    printf("2. Rotation: Earth orbits Sun, Moon orbits Earth\n");
    printf("3. Translation: Objects are positioned in their orbits\n");
    printf("4. Matrix Stack: glPushMatrix/glPopMatrix manage transformation states\n");
    printf("5. Nested Transformations: Moon's transform is relative to Earth's\n");
    printf("\nLab Tasks:\n");
    printf("1. Complete all TODO items in the transformation functions\n");
    printf("2. Experiment with different transformation parameters\n");
    printf("3. Add another planet to the solar system\n");
```

```
    printf("4. Modify the animation speeds to be more realistic\n");
}

int main(int argc, char** argv) {
    // Initialize GLUT
    // Initialize OpenGL settings
    init();

    // Print instructions to console
    printInstructions();

    // TODO: Uncomment the function you want to test:
    // glutDisplayFunc(Translate);
    // glutDisplayFunc(ScaleAboutOrigin);
    // glutDisplayFunc(ScaleAboutFixedPoint);
    // glutDisplayFunc(RotationAboutOrigin);
    // glutDisplayFunc(RotationAboutFixedPoint);
    glutDisplayFunc(display); // Solar system animation

    // Start animation timer
    glutTimerFunc(0, timer, 0);

    // Enter the main event loop
    glutMainLoop();

    return 0;
}
```



Submission: Answers for Part A and the screen shots of the code and relevant outputs for PartB