

# MODUL 4

KOMPUTER GRAFIK 2D  
TRANSFORMASI 2D (TRANSLASI, SCALE & BASIC ANIMASI)

D4 TEKNIK INFORMATIKA  
JURUSAN TEKNIK KOMPUTER DAN INFORMATIKA  
POLITEKNIK NEGERI BANDUNG



MUHAMMAD SAMUDERA BAGJA 058 | KOMPUTER GRAFIK |  
FEB, 05 2025

## **CONTENTS**

Konsep TRANSFORMASI.....	0
TRANSLASI.....	0
SCALING.....	2
ROTASI.....	3
HOMOGENEOUS MATRIKS.....	7
PENGUMPULAN.....	13

## KONSEP TRANSFORMASI

With the procedures for displaying output primitives and their attributes, we can create a variety of pictures and graphs. In many applications, there is also a need for altering or manipulating displays. Design applications and facility layouts are created by arranging the orientations and sizes of the component parts of the scene. And animations are produced by moving the "camera" or the objects in a scene along animation paths. Changes in orientation, size, and shape are accomplished with **geometric transformations** that alter the coordinate descriptions of objects. The basic geometric transformations are translation, rotation, and scaling. Other transformations that are often applied to objects include reflection and shear. We first discuss methods for performing geometric transformations and then consider how transformation functions can be incorporated into graphics packages.

### TRANSLASI

#### Translation

A **translation** is applied to an object by repositioning it along a straight-line path from one coordinate location to another. We translate a two-dimensional point by adding **translation distances**,  $t_x$  and  $t_y$ , to the original coordinate position  $(x, y)$  to move the point to a new position  $(x', y')$  (Fig. 5-1).

$$x' = x + t_x, \quad y' = y + t_y \quad (5.1)$$

The translation distance pair  $(t_x, t_y)$  is called a **translation vector** or **shift vector**.

We can express the translation equations 5-1 as a single matrix equation by using column vectors to represent coordinate positions and the translation vector:

$$\mathbf{P} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \quad \mathbf{P}' = \begin{bmatrix} x'_1 \\ x'_2 \end{bmatrix}, \quad \mathbf{T} = \begin{bmatrix} t_x \\ t_y \end{bmatrix} \quad (5-2)$$

This allows us to write the two-dimensional translation equations in the matrix form:

$$\mathbf{P}' = \mathbf{P} + \mathbf{T} \quad (5-3)$$

Sometimes matrix-transformation equations are expressed in terms of coordinate row vectors instead of column vectors. In this case, we would write the matrix representations as  $\mathbf{P} = [x \ y]$  and  $\mathbf{T} = [t_x \ t_y]$ . Since the column-vector representation for a point is standard mathematical notation, and since many graphics packages, for example, GKS and PHIGS, also use the column-vector representation, we will follow this convention.

Translation is a *rigid-body transformation* that moves objects without deforma-

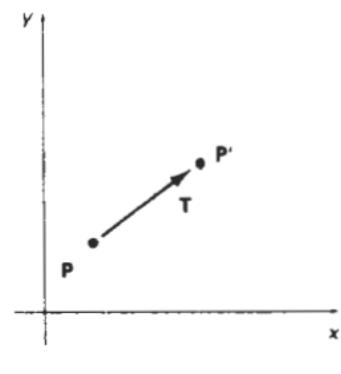


Figure 5-1  
Translating a point from position  $\mathbf{P}$  to position  $\mathbf{P}'$  with translation vector  $\mathbf{T}$ .

Similar methods are used to translate curved objects. To change the position of a circle or ellipse, we translate the center coordinates and redraw the figure in the new location. We translate other curves (for example, splines) by displacing the coordinate positions defining the objects, then we reconstruct the curve paths using the translated coordinate points.

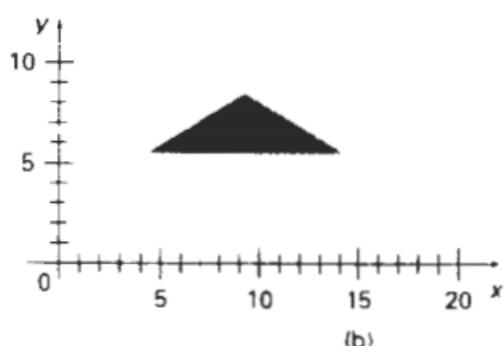
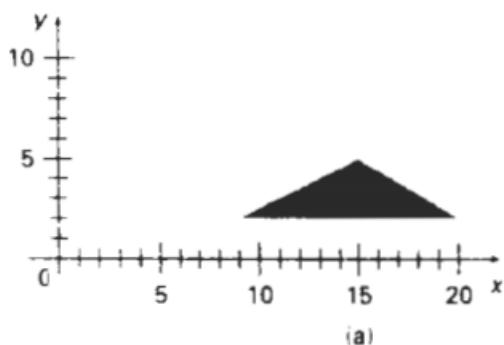


Figure 5-2  
Moving a polygon from position (a) to position (b) with the translation vector  $(-5.50, 3.75)$ .

## SCALING

A **scaling** transformation alters the size of an object. This operation can be carried out for polygons by multiplying the coordinate values ( $x, y$ ) of each vertex by **scaling factors**  $s_x$  and  $s_y$  to produce the transformed coordinates ( $x', y'$ ):

$$x' = x \cdot s_x, \quad y' = y \cdot s_y \quad (5-10)$$

Scaling factor  $s_x$  scales objects in the  $x$  direction, while  $s_y$  scales in the  $y$  direction. The transformation equations 5-10 can also be written in the matrix form:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} \quad (5-11)$$

or

$$\mathbf{P}' = \mathbf{S} \cdot \mathbf{P} \quad (5-12)$$

where  $\mathbf{S}$  is the 2 by 2 scaling matrix in Eq. 5-11.

Any positive numeric values can be assigned to the scaling factors  $s_x$  and  $s_y$ . Values less than 1 reduce the size of objects; values greater than 1 produce an enlargement. Specifying a value of 1 for both  $s_x$  and  $s_y$  leaves the size of objects unchanged. When  $s_x$  and  $s_y$  are assigned the same value, a **uniform scaling** is pro-

duced that maintains relative object proportions. Unequal values for  $s_x$  and  $s_y$  result in a **differential scaling** that is often used in design applications, where pictures are constructed from a few basic shapes that can be adjusted by scaling and positioning transformations (Fig. 5-6).

Objects transformed with Eq. 5-11 are both scaled and repositioned. Scaling factors with values less than 1 move objects closer to the coordinate origin, while values greater than 1 move coordinate positions farther from the origin. Figure 5-7 illustrates scaling a line by assigning the value 0.5 to both  $s_x$  and  $s_y$  in Eq. 5-11. Both the line length and the distance from the origin are reduced by a factor of 1/2.

We can rewrite these scaling transformations to separate the multiplicative and additive terms:

$$\begin{aligned}x' &= x \cdot s_x + x_i(1 - s_x) \\y' &= y \cdot s_y + y_i(1 - s_y)\end{aligned}\quad (5-14)$$

where the additive terms  $x_i(1 - s_x)$  and  $y_i(1 - s_y)$  are constant for all points in the object.

## ROTASI

### Rotation

A two-dimensional **rotation** is applied to an object by repositioning it along a circular path in the  $xy$  plane. To generate a rotation, we specify a **rotation angle**  $\theta$  and the position  $(x_r, y_r)$  of the **rotation point** (or **pivot point**) about which the object is to be rotated (Fig. 5-3). Positive values for the rotation angle define counterclockwise rotations about the pivot point, as in Fig. 5-3, and negative values rotate objects in the clockwise direction. This transformation can also be described as a rotation about a **rotation axis** that is perpendicular to the  $xy$  plane and passes through the pivot point.

We first determine the transformation equations for rotation of a point position  $P$  when the pivot point is at the coordinate origin. The angular and coordinate relationships of the original and transformed point positions are shown in Fig. 5-4. In this figure,  $r$  is the constant distance of the point from the origin, angle  $\phi$  is the original angular position of the point from the horizontal, and  $\theta$  is the rotation angle. Using standard trigonometric identities, we can express the transformed coordinates in terms of angles  $\theta$  and  $\phi$  as

$$\begin{aligned}x' &= r \cos(\phi + \theta) = r \cos \phi \cos \theta - r \sin \phi \sin \theta \\y' &= r \sin(\phi + \theta) = r \cos \phi \sin \theta + r \sin \phi \cos \theta\end{aligned}\quad (5-4)$$

The original coordinates of the point in polar coordinates are

$$x = r \cos \phi, \quad y = r \sin \phi \quad (5-5)$$

Substituting expressions 5-5 into 5-4, we obtain the transformation equations for rotating a point at position  $(x, y)$  through an angle  $\theta$  about the origin:

$$\begin{aligned} x' &= x \cos \theta - y \sin \theta \\ y' &= x \sin \theta + y \cos \theta \end{aligned} \quad (5-6)$$

With the column-vector representations 5-2 for coordinate positions, we can write the rotation equations in the matrix form:

$$\mathbf{P}' = \mathbf{R} \cdot \mathbf{P} \quad (5-7)$$

where the rotation matrix is

$$\mathbf{R} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \quad (5-8)$$

$$\begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}$$

When coordinate positions are represented as row vectors instead of column vectors, the matrix product in rotation equation 5-7 is transposed so that the transformed row coordinate vector  $[x' \ y']$  is calculated as

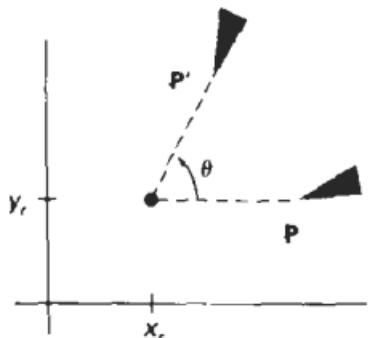
$$\begin{aligned} \mathbf{P}'^T &= (\mathbf{R} \cdot \mathbf{P})^T \\ &= \mathbf{P}^T \cdot \mathbf{R}^T \end{aligned}$$

where  $\mathbf{P}^T = [x \ y]$ , and the transpose  $\mathbf{R}^T$  of matrix  $\mathbf{R}$  is obtained by interchanging rows and columns. For a rotation matrix, the transpose is obtained by simply changing the sign of the sine terms.

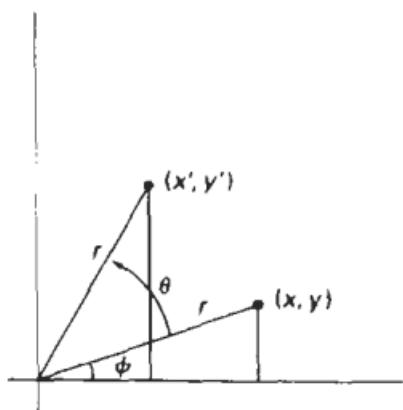
Rotation of a point about an arbitrary pivot position is illustrated in Fig. 5-5. Using the trigonometric relationships in this figure, we can generalize Eqs. 5-6 to obtain the transformation equations for rotation of a point about any specified rotation position  $(x_p, y_p)$ :

$$\begin{aligned}x' &= x_p + (x - x_p) \cos \theta - (y - y_p) \sin \theta \\y' &= y_p + (x - x_p) \sin \theta + (y - y_p) \cos \theta\end{aligned}\quad (5-9)$$

These general rotation equations differ from Eqs. 5-6 by the inclusion of additive terms, as well as the multiplicative factors on the coordinate values. Thus, the matrix expression 5-7 could be modified to include pivot coordinates by matrix addition of a column vector whose elements contain the additive (translational) terms in Eqs. 5-9. There are better ways, however, to formulate such matrix equations, and we discuss in Section 5-2 a more consistent scheme for representing the transformation equations.



**Figure 5-3**  
Rotation of an object through angle  $\theta$  about the pivot point  $(x_r, y_r)$ .

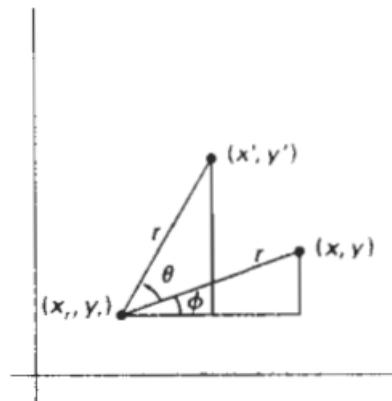


**Figure 5-4**  
Rotation of a point from position  $(x, y)$  to position  $(x', y')$  through an angle  $\theta$  relative to the coordinate origin. The original angular displacement of the point from the  $x$  axis is  $\phi$ .

## Section 5-1

---

### Basic Transformations



**Figure 5-5**  
Rotating a point from position  $(x, y)$  to position  $(x', y')$  through an angle  $\theta$  about rotation point  $(x_r, y_r)$ .

## HOMOGENEOUS MATRICKS

Many graphics applications involve sequences of geometric transformations. An animation, for example, might require an object to be translated and rotated at each increment of the motion. In design and picture construction applications,

we perform translations, rotations, and scalings to fit the picture components into their proper positions. Here we consider how the matrix representations discussed in the previous sections can be reformulated so that such transformation sequences can be efficiently processed.

We have seen in Section 5-1 that each of the basic transformations can be expressed in the general matrix form

$$\mathbf{P}' = \mathbf{M}_1 \cdot \mathbf{P} + \mathbf{M}_2 \quad (5-15)$$

with coordinate positions  $\mathbf{P}$  and  $\mathbf{P}'$  represented as column vectors. Matrix  $\mathbf{M}_1$  is a 2 by 2 array containing multiplicative factors, and  $\mathbf{M}_2$  is a two-element column matrix containing translational terms. For translation,  $\mathbf{M}_1$  is the identity matrix. For rotation or scaling,  $\mathbf{M}_2$  contains the translational terms associated with the pivot point or scaling fixed point. To produce a sequence of transformations with these equations, such as scaling followed by rotation then translation, we must calculate the transformed coordinates one step at a time. First, coordinate positions are scaled, then these scaled coordinates are rotated, and finally the rotated coordinates are translated. A more efficient approach would be to combine the transformations so that the final coordinate positions are obtained directly from the initial coordinates, thereby eliminating the calculation of intermediate coordinate values. To be able to do this, we need to reformulate Eq. 5-15 to eliminate the matrix addition associated with the translation terms in  $\mathbf{M}_2$ .

We can combine the multiplicative and translational terms for two-dimensional geometric transformations into a single matrix representation by expanding the 2 by 2 matrix representations to 3 by 3 matrices. This allows us to express all transformation equations as matrix multiplications, providing that we also expand the matrix representations for coordinate positions. To express any two-dimensional transformation as a matrix multiplication, we represent each Cartesian coordinate position  $(x, y)$  with the **homogeneous coordinate triple**  $(x_h, y_h, h)$ , where

$$x = \frac{x_h}{h}, \quad y = \frac{y_h}{h} \quad (5-16)$$

Thus, a general homogeneous coordinate representation can also be written as  $(h \cdot x, h \cdot y, h)$ . For two-dimensional geometric transformations, we can choose the homogeneous parameter  $h$  to be any nonzero value. Thus, there is an infinite number of equivalent homogeneous representations for each coordinate point  $(x, y)$ . A convenient choice is simply to set  $h = 1$ . Each two-dimensional position is then represented with homogeneous coordinates  $(x, y, 1)$ . Other values for parameter  $h$  are needed, for example, in matrix formulations of three-dimensional viewing transformations.

Thus, a general homogeneous coordinate representation can also be written as  $(h \cdot x, h \cdot y, h)$ . For two-dimensional geometric transformations, we can choose the homogeneous parameter  $h$  to be any nonzero value. Thus, there is an infinite number of equivalent homogeneous representations for each coordinate point  $(x, y)$ . A convenient choice is simply to set  $h = 1$ . Each two-dimensional position is then represented with homogeneous coordinates  $(x, y, 1)$ . Other values for parameter  $h$  are needed, for example, in matrix formulations of three-dimensional viewing transformations.

The term *homogeneous coordinates* is used in mathematics to refer to the effect of this representation on Cartesian equations. When a Cartesian point  $(x, y)$  is converted to a homogeneous representation  $(x_h, y_h, h)$ , equations containing  $x$  and  $y$ , such as  $f(x, y) = 0$ , become homogeneous equations in the three parameters  $x_h$ ,  $y_h$ , and  $h$ . This just means that if each of the three parameters is replaced by any value  $v$  times that parameter, the value  $v$  can be factored out of the equations.

represented with three-element column vectors, and transformation operations are written as 3 by 3 matrices. For translation, we have

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (5-17)$$

which we can write in the abbreviated form

$$\mathbf{P}' = \mathbf{T}(t_x, t_y) \cdot \mathbf{P} \quad (5-18)$$

Similarly, rotation transformation equations about the coordinate origin are now written as

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (5-19)$$

or as

$$\mathbf{P}' = \mathbf{R}(\theta) \cdot \mathbf{P} \quad (5-20)$$

Finally, a scaling transformation relative to the coordinate origin is now expressed as the matrix multiplication

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (5-21)$$

or

$$\mathbf{P}' = \mathbf{S}(s_x, s_y) \cdot \mathbf{P} \quad (5-22)$$

```
#include <math.h>
#include "graphics.h"

typedef float Matrix3x3[3][3];
Matrix3x3 theMatrix;

void matrix3x3SetIdentity (Matrix3x3 m)
{
    int i,j;

    for (i=0; i<3; i++) for (j=0; j<3; j++) m[i][j] = (i == j);

/* Multiplies matrix a times b, putting result in b */
void matrix3x3PreMultiply (Matrix3x3 a, Matrix3x3 b)
{
    int r,c;
    Matrix3x3 tmp;

    for (r = 0; r < 3; r++)
        for (c = 0; c < 3; c++)
            tmp[r][c] =
                a[r][0]*b[0][c] + a[r][1]*b[1][c] + a[r][2]*b[2][c];

    for (r = 0; r < 3; r++)
        for (c = 0; c < 3; c++)
            b[r][c] = tmp[r][c];
}

void translate2 (int tx, int ty)
{
    Matrix3x3 m;

    matrix3x3SetIdentity (m);
    m[0][2] = tx;
    m[1][2] = ty;
    matrix3x3PreMultiply (m, theMatrix);
```

```

void scale2 (float sx, float sy, wcPt2 refpt)
{
    Matrix3x3 m;

    matrix3x3SetIdentity (m);
    m[0][0] = sx;
    m[0][2] = (1 - sx) * refpt.x;
    m[1][1] = sy;
    m[1][2] = (1 - sy) * refpt.y;
    matrix3x3PreMultiply (m, theMatrix);
}

void rotate2 (float a, wcPt2 refPt)
{
    Matrix3x3 m;

    matrix3x3SetIdentity (m);
    a = pToRadians (a);
    m[0][0] = cosf (a);
    m[0][1] = -sinf (a);
    m[0][2] = refPt.x * (1 - cosf (a)) + refPt.y * sinf (a);
    m[1][0] = sinf (a);
    m[1][1] = cosf (a);
    m[1][2] = refPt.y * (1 - cosf (a)) - refPt.x * sinf (a);
    matrix3x3PreMultiply (m, theMatrix);
}

void transformPoints2 (int npts, wcPt2 *pts)
{
    int k;
    float tmp;

    for (k = 0; k < npts; k++) {
        tmp = theMatrix[0][0] * pts[k].x + theMatrix[0][1] *
              pts[k].y + theMatrix[0][2];
        pts[k].y = theMatrix[1][0] * pts[k].x + theMatrix[1][1] *
              pts[k].y + theMatrix[1][2];
        pts[k].x = tmp;
    }
}

```

```

void main (int argc, char ** argv)
{
    wcPt2 pts[3] = { 50.0, 50.0, 150.0, 50.0, 100.0, 150.0};
    wcPt2 refPt = {100.0, 100.0};
    long windowID = openGraphics (*argv, 200, 350);

    setBackground (WHITE);
    setColor (BLUE);
    pFillArea (3, pts);
    matrix3x3SetIdentity (theMatrix);
    scale2 (0.5, 0.5, refPt);
    rotate2 (90.0, refPt);
    translate2 (0, 150);
    transformPoints2 (3, pts);
    pFillArea (3, pts);
    sleep (10);
    closeGraphics (windowID);
}

```

## TASK 0: ANALISA TRANSFORMASI

**Tugas Task 0:** Buat Lesson Learnt untuk pertanyaan-pertanyaan berikut

1. Bedah code Transformasi.cs pada file template yang diberikan oleh Dosen
2. Adaptasi Transformasi.cs pada code praktikum modul 3 anda
3. Apa yang anda pahami tentang homogenous matriks / coordinates
4. Apa yang anda pahami dari fungsi transformPoints
5. Mana yang lebih efisien melakukan transformPoints untuk semua list vector sebuah bentuk, atau cukup melakukan transformPoints, titik-titik pembangun bentuk?

### Lesson Learnt

#### I. Bedah Code Transformasi.cs

File Transformasi.cs berisi implementasi transformasi geometri 2D menggunakan matriks 3x3. Berikut adalah penjelasan dari beberapa bagian penting dalam kode tersebut:

##### a. Matrix Operations

- **Matrix3x3Identity:** Membuat matriks identitas 3x3.
- **Matrix3x3Summation:** Menambahkan dua matriks 3x3.
- **Matrix3x3Subtraction:** Mengurangkan dua matriks 3x3.
- **Matrix3x3Multiplication:** Mengalikan dua matriks 3x3.

##### b. Geometric Transformations

- **Translation:** Melakukan translasi (pergeseran) pada titik dengan menambahkan nilai x dan y ke koordinat titik tersebut.
- **Scaling:** Melakukan scaling (penskalaan) pada titik dengan mengalikan koordinat titik dengan faktor skala x dan y.
- **RotationClockwise** dan **RotationCounterClockwise:** Melakukan rotasi searah jarum jam dan berlawanan arah jarum jam pada titik dengan sudut tertentu.
- **Shearing:** Melakukan shearing (geser) pada titik dengan faktor geser x dan y.
- **ReflectionToX**, **ReflectionToY**, dan **ReflectionToOrigin:** Melakukan refleksi terhadap sumbu X, sumbu Y, dan titik origin.

##### c. GetTransformPoint

- Fungsi ini mengambil daftar titik (`List<Vector2>`) dan mengaplikasikan transformasi matriks pada setiap titik tersebut. Hasilnya adalah daftar titik yang telah ditransformasi.

## 2. Adaptasi Transformasi.cs pada Code Praktikum Modul 3

Untuk mengadaptasi Transformasi.cs ke dalam code praktikum modul 3, perlu memastikan bahwa fungsi-fungsi transformasi yang ada di Transformasi.cs dapat digunakan untuk memanipulasi bentuk-bentuk geometri yang telah dibuat. Berikut adalah langkah-langkahnya:

1. **Inisialisasi Matriks Transformasi:**
  - o Buat matriks identitas untuk memulai transformasi.
2. **Aplikasikan Transformasi:**
  - o Gunakan fungsi-fungsi seperti Translation, Scaling, RotationClockwise, dll., untuk mengaplikasikan transformasi pada matriks.
3. **Transformasikan Titik-Titik:**
  - o Gunakan fungsi GetTransformPoint untuk mengaplikasikan matriks transformasi pada setiap titik yang membentuk bentuk geometri.
4. **Gambar Bentuk yang Telah Ditransformasi:**
  - o Setelah titik-titik ditransformasi, gunakan fungsi PutPixelAll dari GraphicsUtils untuk menggambar bentuk yang telah ditransformasi.

## 3. Homogenous Matriks / Coordinates

**Homogenous Coordinates** adalah sistem koordinat yang digunakan dalam grafik komputer untuk mempermudah operasi transformasi seperti translasi, rotasi, dan scaling. Dalam sistem ini, setiap titik dalam ruang 2D direpresentasikan sebagai vektor 3D ( $x, y, 1$ ). Dengan menggunakan homogenous coordinates, semua transformasi (translasi, rotasi, scaling) dapat direpresentasikan sebagai perkalian matriks, yang memungkinkan untuk menggabungkan beberapa transformasi menjadi satu matriks transformasi.

## 4. Fungsi GetTransformPoint

Fungsi GetTransformPoint digunakan untuk mengaplikasikan matriks transformasi pada setiap titik dalam daftar titik (`List<Vector2>`). Fungsi ini mengambil matriks transformasi dan daftar titik sebagai input, kemudian mengembalikan daftar titik yang telah ditransformasi.

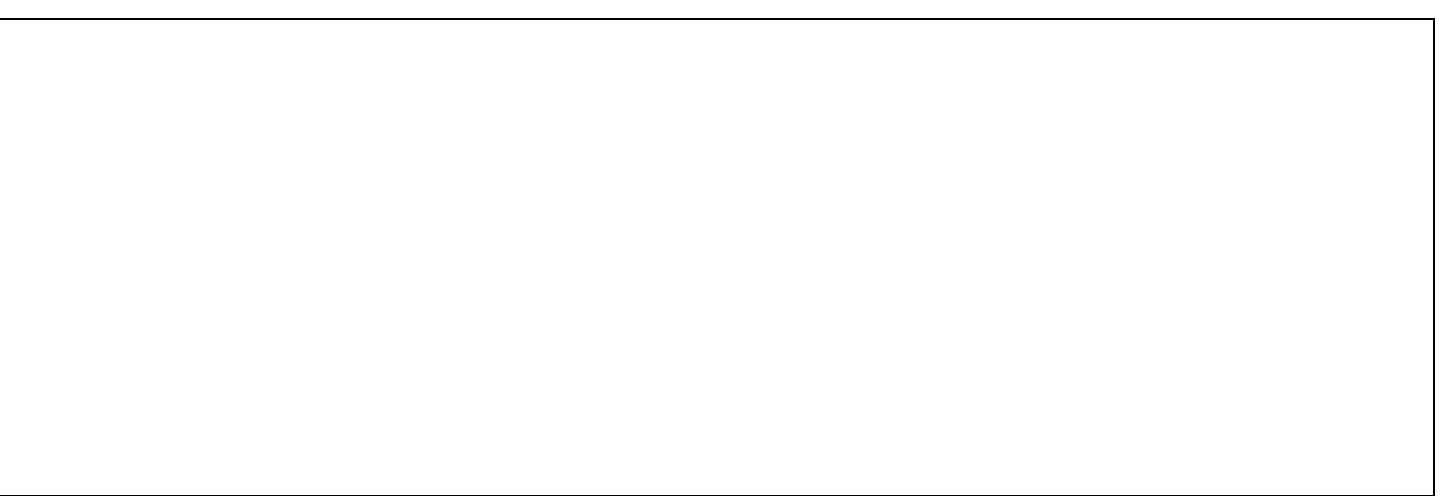
Proses yang dilakukan oleh fungsi ini adalah:

- Mengalikan setiap titik dengan matriks transformasi.
- Mengembalikan titik-titik yang telah ditransformasi.

## 5. Efisiensi Transformasi Titik

Lebih efisien melakukan transformasi pada titik-titik pembangun bentuk daripada melakukan transformasi pada semua titik dalam bentuk. Alasan utamanya adalah:

- **Jumlah Titik yang Lebih Sedikit:** Titik-titik pembangun bentuk (seperti titik sudut pada persegi atau segitiga) biasanya lebih sedikit daripada semua titik yang membentuk bentuk tersebut. Dengan mentransformasi titik-titik pembangun, kita hanya perlu melakukan transformasi pada beberapa titik, bukan pada ratusan atau ribuan titik.
- **Kemudahan dalam Menggambar:** Setelah titik-titik pembangun ditransformasi, kita dapat menggambar bentuk tersebut dengan menghubungkan titik-titik tersebut menggunakan algoritma garis atau bentuk lainnya. Ini lebih efisien daripada mentransformasi setiap piksel dalam bentuk.
- **Kinerja yang Lebih Baik:** Dengan mengurangi jumlah titik yang perlu ditransformasi, kita dapat meningkatkan kinerja aplikasi, terutama jika kita bekerja dengan bentuk yang kompleks atau dalam lingkungan real-time seperti game.

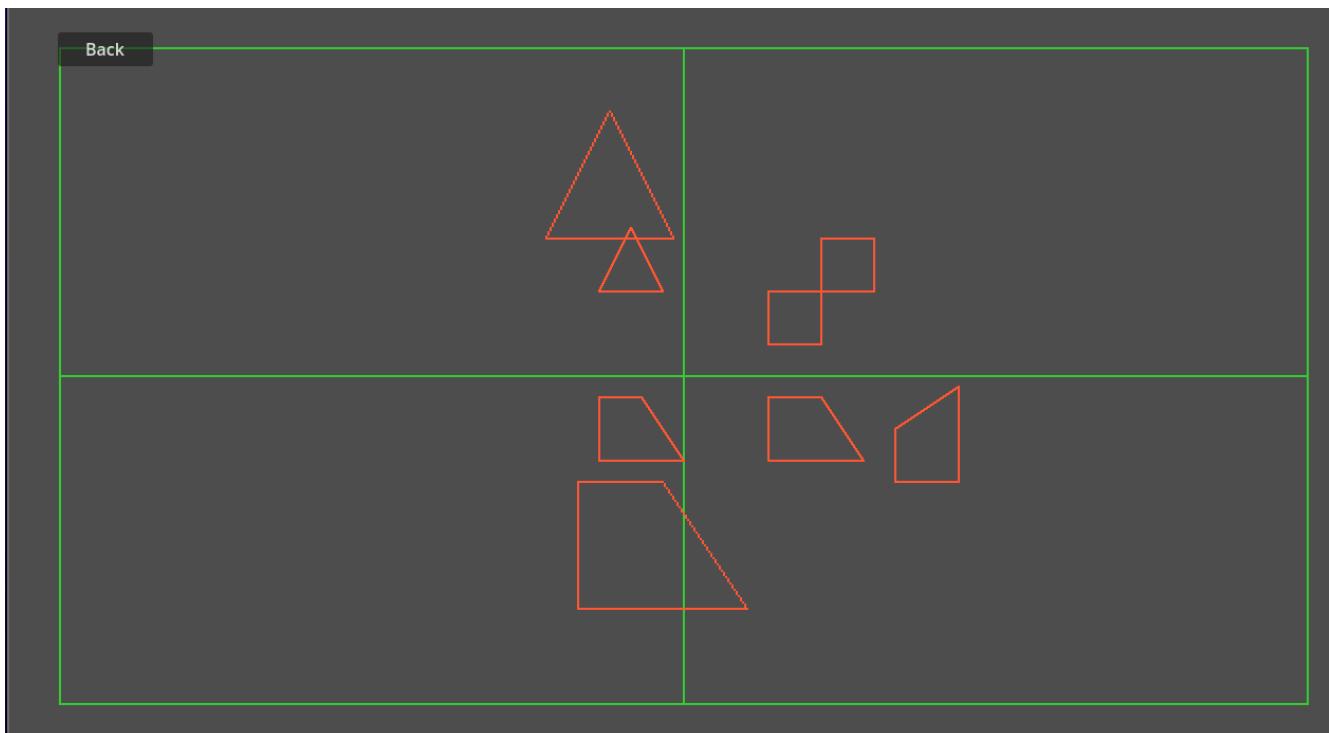


## TASK I: EKSPLORASI TRANSFORMASIX

### Tugas Task I (Karya I)

1. Copy Karya 3 pada Modul 3 ke Karya I Pada Modul 4
2. Persentasikan Bentuk Bentuk Dasar Transformasi (Translasi, Scaling, Rotasi, Translasi + Scaling, Scaling+Translasi)

#### Lesson Learnt



#### I. Memahami Transformasi dalam Grafika Komputer

Dalam pengembangan grafika komputer, transformasi adalah proses memanipulasi objek dengan cara tertentu, seperti translasi, rotasi, dan scaling. Kode ini menerapkan berbagai bentuk transformasi pada objek 2D menggunakan Godot dan C#.

#### 2. Implementasi Transformasi Dasar

##### a. Translasi

Translasi adalah pergeseran objek dalam ruang 2D tanpa mengubah bentuknya. Diterapkan dengan

metode `ApplyTranslation(x, y)` yang memodifikasi matriks transformasi dengan menambahkan offset tertentu.

```
private void ApplyTranslation(float x, float y)
```

```
{
```

```
    Vector2 coord = new Vector2(0, 0);  
    _transformasi.Translation(_transformMatrix, x, y, ref coord);
```

```
}
```

### b. Scaling

Scaling adalah perubahan ukuran objek. Diterapkan dengan `ApplyScaling(x, y)`, yang menggunakan titik pusat sebagai referensi.

```
private void ApplyScaling(float x, float y)
```

```
{
```

```
    Vector2 coord = new Vector2(WorldOriginX, WorldOriginY);  
    _transformasi.Scaling(_transformMatrix, x, y, coord);
```

```
}
```

### c. Rotasi

Rotasi mengubah orientasi objek searah atau berlawanan jarum jam berdasarkan sudut tertentu.

```
private void ApplyRotationClockwise(float angle)
```

```
{
```

```
    Vector2 coord = new Vector2(WorldOriginX, WorldOriginY);  
    _transformasi.RotationClockwise(_transformMatrix, angle, coord);
```

```
}
```

## 3. Penggunaan Transformasi Gabungan

Dalam kode, beberapa kombinasi transformasi diterapkan:

- **Translasi + Scaling:**
- `ApplyScaling(2f, 2f);`

```
ApplyTranslation(30, 30);
```

- **Scaling + Translasi:**
- `ApplyTranslation(30, 30);`

```
ApplyScaling(2f, 2f);
```

Urutan operasi transformasi berpengaruh terhadap hasil akhir.

## 4. Menggambar Objek dengan Transformasi

Setelah transformasi diterapkan, objek digambar dengan metode `PutPixelAll()`, memastikan bahwa perubahan terlihat di layar.

```
List<Vector2> transformedShape = _transformasi.GetTransformPoint(_transformMatrix, shape);
```

```
PutPixelAll(transformedShape, colorShape);
```

## 5. Penerapan dalam Kuadran Kartesian

- **Kuadran I:** Translasi positif pada X dan negatif pada Y.
- **Kuadran II:** Scaling lalu translasi.
- **Kuadran III:** Translasi lalu scaling.
- **Kuadran IV:** Rotasi lalu translasi.

## 6. Kesimpulan

- Transformasi adalah konsep fundamental dalam grafika komputer.
- Urutan operasi transformasi memengaruhi hasil akhir.
- Menggunakan matriks transformasi memungkinkan manipulasi bentuk secara efisien.

Dengan memahami dan mempraktikkan kode ini, kita dapat lebih mudah mengimplementasikan transformasi dalam pengembangan aplikasi grafis menggunakan Godot.

## TASK 2: KARYA 2D BUNGA TRANSFORMASI

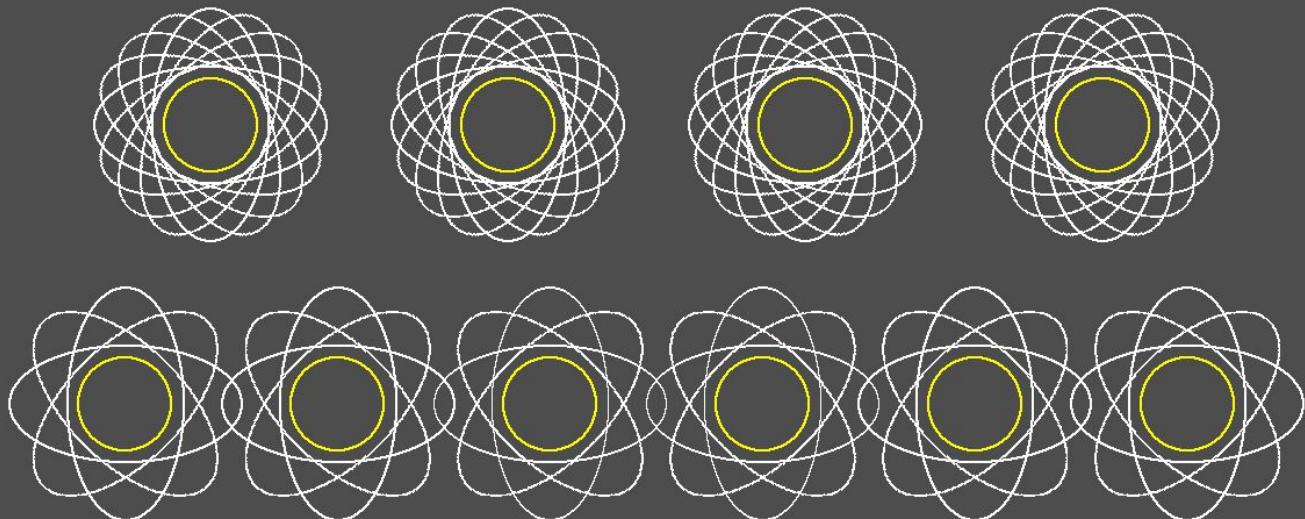
### Tugas Task 2 Karya 2

- I. Buatlah sebuah bunga yang terdiri dari lingkaran pusat dan 4 kelopak ellips (syarat buat satu kelopak terlebih dahulu lalu gunakan transformasi rotate untuk membuat kelopak lainnya)

- Buatlah sebuah bunga yang terdiri dari lingkaran pusat dan 8 kelopak ellips (syarat buat satu kelopak terlebih dahulu lalu gunakan transformasi rotate untuk membuat kelopak lainnya)
- Setiap bunga dibuat menjadi fungsi parametrik atau object (OOP) untuk memudahkan penduplikasian, duplikasikan bunga tersebut sebanyak 4 untuk bunga tipe 1, dan 6 untuk bunga tipe 2.

## Lesson Learnt

[Back](#)



## Lesson Learned - Membuat Bunga dengan Transformasi dan OOP di Godot (C#)

### 1. Membuat Bunga dengan 4 Kelopak Menggunakan Rotasi

Dalam proses pembuatan bunga, kita menggunakan **ellips** sebagai **kelopak** dan sebuah **lingkaran pusat** untuk bagian tengah bunga. Untuk membuat kelopak:

- Kita cukup membuat **satu ellips** terlebih dahulu.
- Kemudian, kita menerapkan **transformasi rotasi** untuk menggandakan kelopak tersebut di sekitar titik pusat bunga.
- Rotasi dilakukan pada sudut **0°, 45°, 90°, dan 135°** untuk mendapatkan total **4 kelopak**.

### Implementasi

- Menggunakan metode `RotatePoint()` untuk merotasi kelopak.
- Menempatkan hasil rotasi ke dalam loop agar kelopak dapat muncul di posisi yang sesuai.

### 2. Membuat Bunga dengan 8 Kelopak Menggunakan Rotasi

Konsepnya sama dengan bunga berkelopak 4, tetapi dengan jumlah rotasi lebih banyak.

- Kita tetap membuat **satu ellips terlebih dahulu**.
- Kemudian, kita menerapkan **rotasi ke sudut 0°, 22.5°, 45°, 67.5°, 90°, 112.5°, 135°, dan 157.5°**.
- Hasilnya adalah **8 kelopak** yang lebih rapat dan menyerupai bunga dengan bentuk lebih kompleks.

### 3. Menggunakan OOP untuk Memudahkan Duplikasi Bunga

Agar lebih mudah dalam duplikasi bunga, kita menggunakan konsep **Object-Oriented Programming (OOP)** dengan membuat kelas Bunga.

- Kelas ini memiliki atribut:
  - Pusat (koordinat pusat bunga)
  - Ukuran (ukuran bunga)
  - JumlahKelopak (jumlah kelopak, bisa diubah secara dinamis)

- Kelas ini memiliki metode Gambar(), yang akan menggambar bunga dengan jumlah kelopak sesuai parameter yang diberikan.

### Keuntungan OOP

- **Mudah diduplikasi:** Kita bisa membuat banyak objek bunga dengan mudah tanpa perlu menulis ulang kode.
- **Fleksibel:** Kita bisa mengubah jumlah kelopak dan posisi dengan mudah.
- **Lebih terstruktur:** Kode lebih rapi dan mudah dipahami.

### 4. Mengubah Jumlah Kelopak dengan Input Keyboard

Pada program ini, kita menambahkan fitur **input dari keyboard**:

- **Tekan tombol "1"** → Kembali ke jumlah kelopak awal (**8 untuk tipe 1, 4 untuk tipe 2**).
- **Tekan tombol "2"** → Semua bunga berubah menjadi **8 kelopak**.
- Menggunakan metode SetJumlahKelopak(int jumlah) untuk mengubah jumlah kelopak setiap bunga.
- Memanggil QueueRedraw() agar gambar diperbarui setiap kali tombol ditekan.

### Kesimpulan

1. **Rotasi** sangat berguna untuk menggandakan kelopak bunga tanpa perlu menggambar setiap kelopak secara manual.
2. **Menggunakan OOP** membuat kode lebih modular, mudah diperluas, dan lebih terstruktur.
3. **Interaksi dengan pengguna** (menggunakan input keyboard) memungkinkan modifikasi bunga secara dinamis tanpa harus menjalankan ulang program.
4. **Penerapan List** untuk menyimpan objek bunga memudahkan pengelolaan banyak bunga sekaligus.

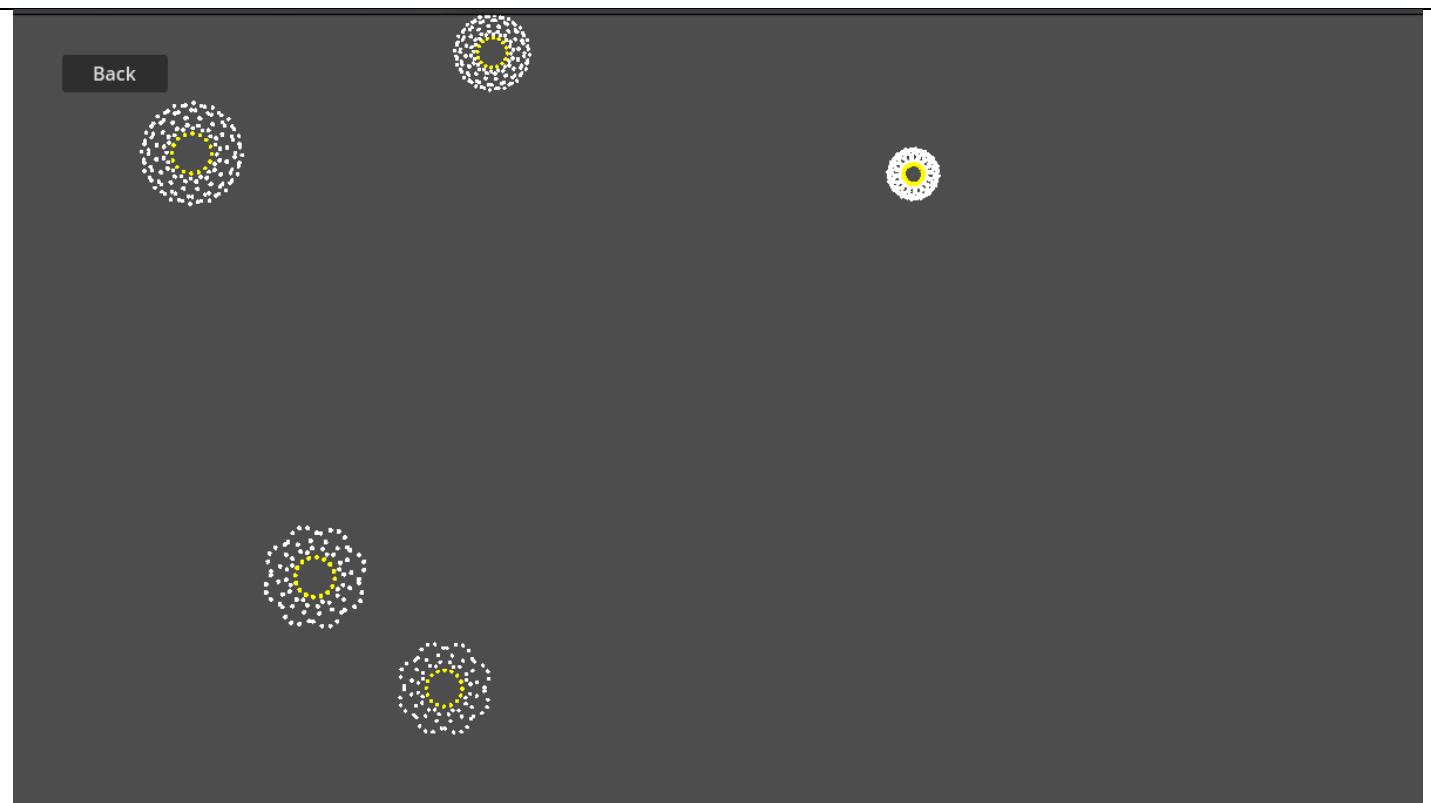
Dengan memahami konsep ini, kita dapat dengan mudah mengembangkan sistem grafik berbasis transformasi dan OOP untuk aplikasi lainnya, seperti animasi, simulasi objek berulang, atau sistem procedural generation di Godot.

## TASK 3: KARYA 2D BUNGA TRANSFORMASI ANIMASI

Tugas Task 3 Karya 3 (Animasinya dapat dilanjutkan minggu depan)

1. Semenarik mungkin tuangkan list ide-ide animasi untuk bunga tipe 1 dan tipe 2
2. Urutkan ide tersebut dari yang paling sederhana hingga yang paling kompleks. lalu pilih 2 ide animasi sederhana dan animasi kompleks.
3. Buatlah Design atau storyboard untuk animasi sederhana setiap bunga tipe 1 dan tipe 2 untuk (misal, untuk frame 1-60, bunga translasi +x dengan jarak 10 lalu berotasi 360 derajat lalu kembali ke tempat semula, dst).
4. Buatlah Design atau storyboard untuk animasi kompleks setiap bunga tipe 1 dan tipe 2 untuk (misal, untuk frame 1-60, bunga translasi +x dengan jarak 10 lalu berotasi 360 derajat lalu kembali ke tempat semula, dst).
5. Pada karya 3 implementasikan ide tersebut (untuk minggu depan).

Lesson Learnt



Disini saya menggunakan beberapa bunga yang saya modifikasi, pada tugas sebelumnya saya hanya menggunakan lingkaran dan elips, disini saya menggunakan variasi dengan menjadikannya sebagai titik titik dan menganimasikannya secara random, saya menganimasikannya dengan scaling, translasi,

- I. Struktur Animasi dan Pembuatan Bunga

**Posisi Awal:** Setiap bunga memiliki posisi acak dalam FieldSize.

• **Ukuran:** Ukuran bunga bervariasi antara MinFlowerSize dan MaxFlowerSize.

• **Jumlah Kelopak:** Setiap bunga memiliki 4 atau 8 kelopak secara acak.

• **Parameter Animasi:**

- **Sway (Goyangan):** Menggunakan swaySpeed dan swayAmount untuk efek angin.

- **Grow Pulse (Pulsasi Ukuran):** Bunga sedikit membesar dan mengecil dengan growPulseSpeed dan growPulseAmount.

- **Rotasi:** Bunga berputar searah atau berlawanan dengan rotationSpeed.

Kode ini secara keseluruhan sudah cukup terstruktur untuk mengelola animasi banyak bunga dengan variasi unik.

## 2. Teknik Animasi yang Digunakan

Animasi dalam kode ini dilakukan dengan cara:

### I. Perubahan Posisi (Swaying Effect)

- Menggunakan fungsi sinus untuk memberikan efek goyangan akibat angin:

```
float swayX = Mathf.Sin(time * _swaySpeed) * _swayAmount;  
float swayY = Mathf.Cos(time * _swaySpeed * 0.7f) * (_swayAmount * 0.5f);  
◦ Kombinasi Sin dan Cos menciptakan pergerakan yang lebih alami.
```

### 2. Pulsasi Ukuran (Growth Pulse)

- Menggunakan fungsi sinus untuk membuat bunga tampak membesar dan mengecil:

```
_currentScale = 1f + Mathf.Sin(time * _growPulseSpeed) * _growPulseAmount;  
◦ Efek ini menciptakan ilusi pernapasan pada bunga.
```

### 3. Rotasi

- Rotasi bunga dihitung dengan:  
`_currentRotation = (time * _rotationSpeed) % 360f;`
  - Ini menyebabkan bunga terus berputar seiring waktu.

### 4. Penerapan Transformasi

- Semua efek ini digabungkan dalam satu transformasi:

```
Transform2D transform = Transform2D.Identity  
.Scaled(new Vector2(_currentScale, _currentScale))  
.Rotated(Mathf.DegToRad(_currentRotation));
```

## 3. Kendala dalam Membuat Animasi Smooth

Masalah utama yang dihadapi adalah animasi yang belum terasa halus. Beberapa kemungkinan penyebabnya:

- **Delta Time Tidak Dipakai dengan Konsisten**

- Di `_Process(double delta)`, delta sudah diterapkan dengan benar untuk rotasi, tetapi bisa dicek ulang apakah Update juga mempertimbangkan delta dengan baik.

- **Kuantisasi Waktu yang Terlalu Kasar**

- Jika frame rate tidak stabil, gerakan berbasis `sin(time * speed)` bisa terasa patah-patah.
  - Bisa dicoba menggunakan delta lebih eksplisit:

```
float swayX = Mathf.Sin(_elapsedTime + delta) * _swaySpeed) * _swayAmount;
```

- **Transformasi Tidak Bertahap**

- Jika bunga langsung diperbarui dalam satu langkah besar tanpa interpolasi, hasilnya bisa terlihat tidak smooth.
  - **Solusi:** Gunakan Tween dari Godot untuk transisi yang lebih halus.

## 4. Optimasi Potensial

### 1. Gunakan Tween untuk Animasi yang Lebih Halus

- Godot memiliki sistem Tween untuk animasi interpolasi.
  - Daripada mengandalkan perhitungan manual sinus, bisa menggunakan:

```
var tween = create_tween()  
tween.tween_property(self, "rotation", 360, 2).set_trans(Tween.TRANS_SINE).set_ease(Tween.EASE_IN_OUT)  
○ Ini bisa membantu menghindari efek animasi yang patah-patah.
```

### 2. Gunakan Interpolasi Linear

- Saat menggambar, interpolasi linear bisa digunakan agar transisi lebih halus:

```
Vector2 smoothOffset = _currentOffset.Lerp(targetOffset, 0.1f);  
○ Ini memastikan bunga tidak langsung berpindah drastis.
```

### 3. Kurangi Frekuensi Pembaruan Animasi

- Jika animasi terasa terlalu cepat dan tidak smooth, bisa diperlambat dengan mengurangi nilai speed di `_swaySpeed` dan `_growPulseSpeed`.

# PENGUMPULAN

Ikuti Format yang diberikan di Google Classroom.