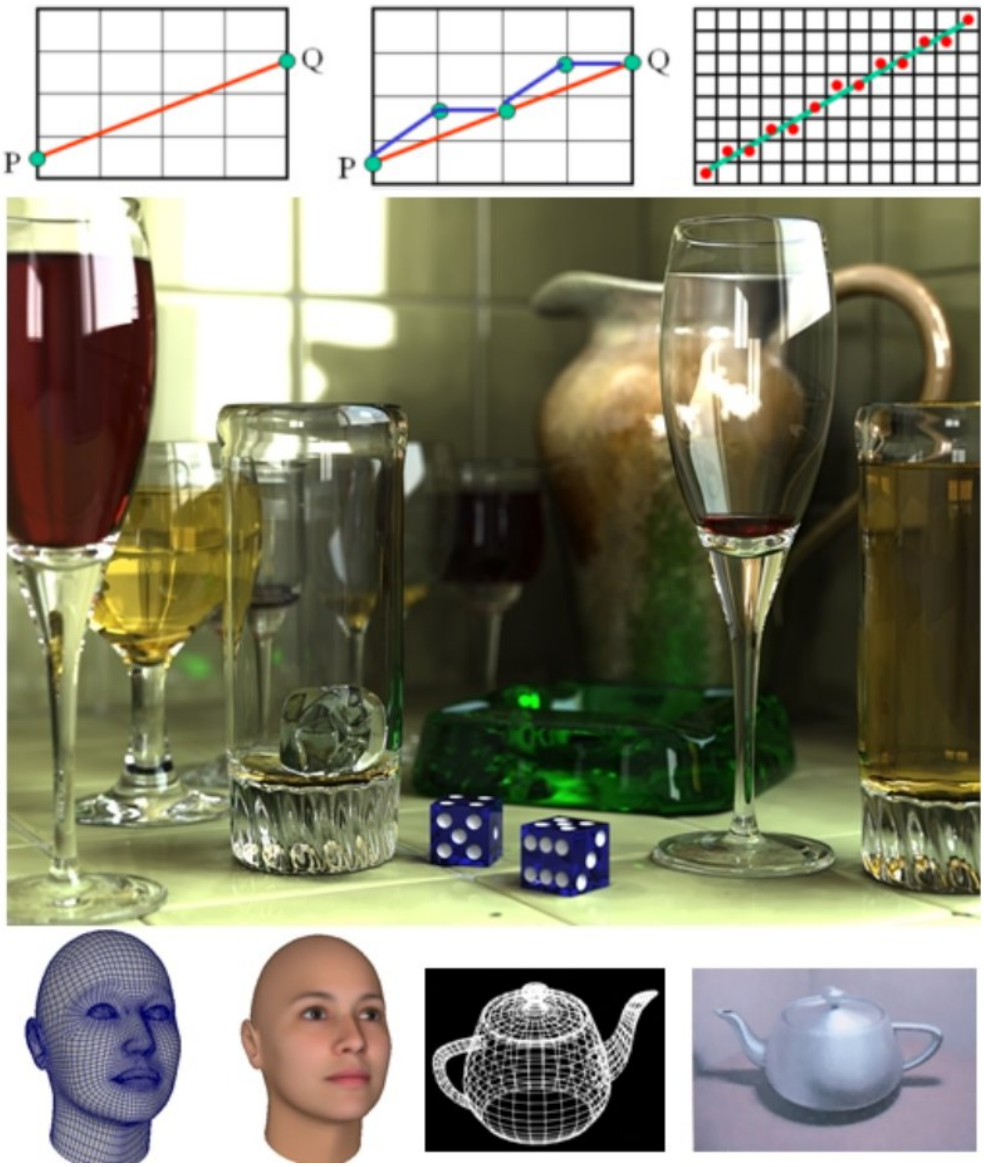
MODUL PRAKTIKUM GRAFIKA KOMPUTER

Dr. Setiawan Hadi, M.Sc.CS.
Prof. Anton Satria Prabuwono, Ph.D.



PROGRAM STUDI S-1 TEKNIK INFORMATIKA
FAKULTAS MATEMATIKA DAN ILMU PENGETAHUAN ALAM
UNIVERSITAS PADJADJARAN
2021

Daftar Isi

Kata Pengantar

Modul praktikum ini disusun sebagai pegangan bagi peserta mata kuliah Grafika Komputer dalam praktikum di laboratorium. Materi yang disajikan diusahakan sejalan dengan materi teori yang diberikan di dalam kelas. Dengan demikian penjelasan yang diberikan pada modul praktikum ini bersifat sebagai penyegar ingatan terhadap materi teoritis secara keseluruhan.

Mulai tahun 2021, materi praktikum mengalami penyesuaian dengan perkembangan teknologi informasi khususnya dengan perkembangan dalam bidang Grafika Komputer. Materi praktikum dikelompokkan menjadi tiga bagian. Bagian pertama berisi materi praktikum yang berkaitan konsep dasar grafika komputer, yaitu penggambaran objek primitif, windowing dan clipping, serta transformasi dua dimensi. Bagian kedua berisi materi praktikum yang menerapkan teknologi terkini dalam grafika komputer menggunakan bahasa pemrograman Python. Praktikum-praktikum yang disajikan berkaitan dengan aplikasi grafis dan animasi yang dengan mudah dapat dibangun menggunakan bahasa pemrograman Python. Bagian ketiga berisi materi praktikum OpenGL, sebuah *middleware* grafika komputer yang digunakan secara *multiplatform*, pada komputer *desktop* maupun *mobile environment*.

Setiap praktikum memiliki struktur (1) Teori (2) Latihan, dan (3) Praktikum. Teori berisi pemaparan materi substansial secara ringkas dan mendukung pemahaman peserta praktikum selain materi yang lebih lengkap dalam kelas dan buku pegangan. Latihan berisi tugas mengimplementasikan contoh-contoh yang diberikan dalam teori. Materi inti praktikum diberikan dalam bagian Praktikum, yang berisi tugas praktikum yang harus dilakukan peserta dan menyampaikan laporan hasil praktikumnya, baik ke asisten laboratorium, maupun ke LIVE!.

Penulis berharap modul praktikum ini bermanfaat meningkatkan pemahaman peserta mata kuliah Grafika Komputer. Selain itu, pengalaman praktis dalam pemrograman menggunakan bahasa komputer modern C++, C#, Python, Java, dan lain-lain dalam lingkungan yang modern dapat membuka dan menambah wawasan sekaligus peluang dalam mengembangkan aplikasi grafis yang lebih lanjut dan lebih realistis.

September 2021
S.H. A.S.P.

Catatan Edisi 2021

Pada edisi tahun 2021, telah dilakukan *re-design* struktur praktikum dalam bentuk (1) reorganisasi materi praktikum Grafika Komputer secara lengkap dan mengikuti perkembangan ilmu dan teknologi grafika komputer (2) reorganisasi pelaksanaan praktikum sehingga memiliki struktur yang baik (3) .... Namun karena keterbatasan waktu, maka pada edisi 2021, isi praktikum Grafika Komputer hanya lengkap pada Praktikum 1 hingga Praktikum 4. Pada waktu-waktu selanjutnya modul praktikum ini akan dilengkapi sehingga sempurna.

Secara keseluruhan modul praktikum Grafika Komputer ini terdiri dari 5 bagian yaitu Bagian A berisi praktikum-praktikum dasar Grafika Komputer, bagian B berisi praktikum pemanfaatan fungsi-fungsi internal GDI+ pada bahasa pemrograman .NET, bagian C berisi beriki praktikum Grafika dan Animasi komputer menggunakan Python, bagian D berisi Praktikum-praktikum OpenGL dan bagian E berisi praktikum-praktikom topik lanjutan Grafika Komputer.

Struktur setiap praktikum dibuat sedemikian rupa terdiri dari (i) Teori (ii) Latihan, dan (iii) Praktikum. Pada teori akan dijelaskan dasar-dasar teori dari praktikum yang akan dilakukan disertai dengan contoh. Pada Latihan diberikan tugas-tugas sebagai persiapan menuju raktikum yang sebenarnya. Pada Praktikum berisi tugas-tugas sebenarnya dari praktikum yang akan dilakukan. Pada bagain ini mahasiswa harus membuat Laporan Praktikum.

Petunjuk Pelaksanaan Praktikum

Dalam melaksanakan praktikum, praktikan melakukan langkah-langkah sebagai berikut yaitu :

Membaca Teori : yaitu membaca penjelasan tentang praktikum yang akan dilaksanakan berupa penjelasan teoritis disertai dengan contoh-contoh program.

Melaksanakan Latihan : yaitu mengimplementasikan program-program contoh yang diberikan pada bagian Teori.

Melaksanakan Praktikum : yaitu melaksanakan tugas-tugas yang diberikan pada bagian Praktikum. Dokumentas kegiatan praktikum harus dilakukan pada tahapan ini. Untuk lebih meningkatkan pemahaman, diharapkan praktikan tidak melakukan proses *copas*, tetapi membuat sendiri tugas praktikumnya. Dalam hal pemrograman diperlukan kreativitas dari praktikan dalam membuat program yang mudah dijalankan (*user friendly*), inovatif, dan informatif.

Membuat Laporan Praktikum : yaitu kegiatan membuat tulisan tentang kegiatan-kegiatan yang dilakukan dalam praktikum. Didalam laporan ini terdapat hal-hal rinci tentang dokumentasi program dan meta informasi yang berkaitan dengan praktikum yang dilakukan. Laporan ini harus mengacu pada struktur pelaporan ilmiah yang baku. Kegiatan ini juga sekaligus menjadi indikator penilaian praktikum.

Tabel di halaman berikut memberikan rangkuman isi praktikum dan kaitannya dengan capaian pembelajaran mata kuliah.

Part I

Dasar Grafika Komputer (C#)

Praktikum 1

Pemrograman Grafis dengan C#

1.1 Teori

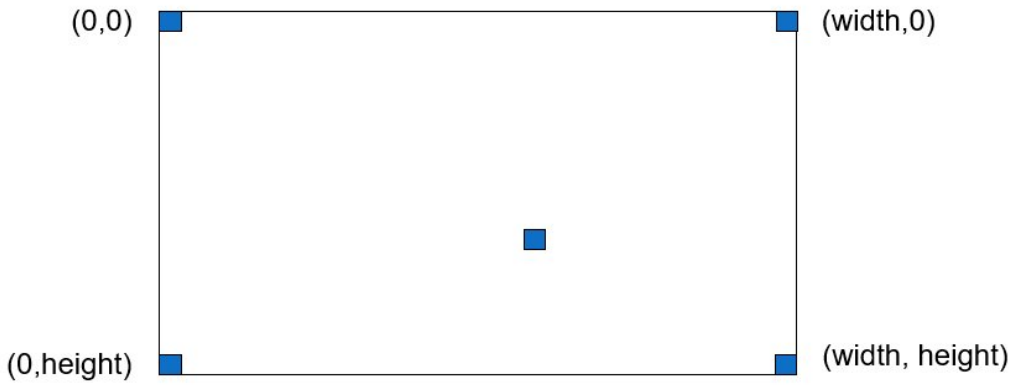
Fasilitas grafis sudah disediakan secara *default* pada bahasa pemgrograman C#. Artinya begitu anda memasang Visual Studio versi apapun di komputer, maka di dalamnya sudah tersedia fasilitas grafis. Fasilitas internal grafis pada Visual Studio disebut dengan GDI+.

GDI+ adalah pengembangan dari GDI. Dengan pengembangan ini, pemrograman grafis di Visual Studio menjadi lebih mudah karena masalah dasar sudah ditangani secara internal oleh Microsoft Visual Studio .NET.

GDI+ sebagai inti pemrograman grafis disimpan dalam *namespace* (*assembly*) `System.Drawing.dll`. Perintah GDI+ lainnya disimpan pada *namespace* `System.Text`, `System.Printing`, `System.Internal`, `System.Imaging`, `System.Drawing2D`, dan `System.Design`.

1.1.1 Sistem Koordinat

Pemrograman grafis pada Visual Studio mengandalkan sistem koordinat khusus yang berbeda dan sistem koordinat Kartesisus. Gambar berikut menjelaskan hal ini.



Perbedaan nyata antara sistem koordinat pada Visual Studio dengan sistem koordinat Kartesisus adalah titik origin. Pada sistem koordinat Visual Studio, titik

origin atau titik $(0, 0)$ berada pada lokasi **kiri atas**, sedangkan pada sistem koordinat Kartesius, titik origin pada umumnya ada pada lokasi kiri bawah.

1.1.2 Kelas Graphics

Kelas **Graphics** membungkus (*encapsulate*) perintah-perintah GDI+. Sebelum menggambar sebuah objek primitif, kita harus membuat sebuah *surface* atau kanvas grafis menggunakan kelas **Graphics**. Terdapat berbagai cara untuk membuat sebuah kanvas grafis pada Visual Studio. Salah satu caranya adalah sebagai berikut :

1. Jalan Visual Studio, buat sebuah proyek dalam *template* C# Windows Form App

2. Misalkan, **Form** akan dijadikan sebagai kanvas grafis, pilihlah `Properties` dari `Form`, lalu pilih `Events`, lalu pilih `Paint`, dan klik 2 kali.

3. Layar kode akan terbuka, ketikkan kode berikut :

```
1 Graphics g = CreateGraphics();
2 Pen p = new Pen(Color.Black);
3 g.DrawLine(p, 0, 0, 200, 200);
```

   Baris 1 mengubah Form menjadi kanvas grafis, Baris 2 mendefinisikan alat tulis `Pen` dengan warna hitam, baris 3 menggambarkan garis dari koordinat (0,0) ke koordinat (200,200).

Selain `DrawLine`, ada perintah-perintah lain yang ada dalam kelas **Graphics**, seperti pada tabel berikut ini.

`DrawArc`	*Draws an arc from the specified ellipse*
`DrawBezier`	*Draws a cubic bezier curve*
`DrawBeziers`	*Draws a series of cubic Bezier curves*
`DrawClosedCurve`	*Draws a closed curve defined by an array of points*
`DrawCurve`	*Draws a curve defined by an array of points*
`DrawEllipse`	*Draws an ellipse*
`DrawImage`	*Draws an image*
`DrawLine`	*Draws a line*
`DrawPath`	*Draws the lines and curves defined by a GraphicsPath*
`DrawPie`	*Draws the outline of a pie section*
`DrawPolygon`	*Draws the outline of a polygon*
`DrawRectangle`	*Draws the outline of a rectangle*
`DrawString`	*Draws a string*
`FillEllipse`	*Fills the interior of an ellipse defined by a bounding rectangle*
`FillPath`	*Fills the interior of a path*
`FillPie`	*Fills the interior of a pie section*
`FillPolygon`	*Fills the interior of a polygon defined by an array of points*
`FillRectangle`	*Fills the interior of a rectangle with a Brush*
`FillRectangles`	*Fills the interiors of a series of rectangles with a Brush*
`FillRegion`	*Fills the interior of a Region*

1.2 Latihan

1. Bacalah bagian teori tentang Sistem Koordinat dan Kelas **Graphics** dengan seksama dan cobalah contoh programnya, dan hasilnya seperti ini.

2. Cobalah semua perintah GDI+ yang ada pada tabel

1.3 Praktikum

1. Buatlah objek yang sederhana yang menerapkan perintah-perintah dalam kelas GDI+, minimal 3, maksimal 5 perintah GDI+

2. Buatlah objek yang kompleks bermakna yang menerapkan perintah-perintah dalam kelas GDI+, minimal 10

Tuliskan Laporan Praktikum, berikan deskripsi dan penjelasan secara rinci, termasuk *listing program* yang diberi komentar, kepada asisten praktikum di laboratorium.

Praktikum 2

Penggambaran Objek Primitif

2.1 Teori

Objek primitif diartikan sebagai objek dasar 2D yang membangun sebuah sistem grafika komputer. Secara umum objek primitif 2D dalam grafika komputer adalah garis dan lingkaran. Selain itu ada objek primitif yang menjadi dasar semuanya yaitu titik. Dua buah titik akan membangun sebuah garis, kumpulan garis akan membangun objek. Titik juga pada dasarnya akan membangun kurva, dan kurva akan membangun lingkaran atau juga elips. Pada praktikum ini yang disebut dengan objek primitif adalah garis dan lingkaran.

2.1.1 Garis

Untuk menggambar garis setidaknya terdapat 3 algoritma yaitu (i) algoritma Dasar, (ii) algoritma DDA, dan (3) algoritma Bresenham sebagaimana ditunjukkan pada Algoritma 2.1, Algoritma 2.2, dan Algoritma 2.3.

Algoritma 2.1 Penggambaran Garis Dasar

1: Inputkan x_1, y_1, dan x_2, y_2
2: Hitung $m = \frac{y_2 - y_1}{x_2 - x_1}$
3: **for** x_1 menuju x_2 dengan $\Delta x = 1$ **do**
4: hitung $y^* = y_1 + m$
5: $y_1 = y^*$
6: **end for**

Sebagaimana dijelaskan sebelumnya, elemen pembentuk garis adalah titik, minimal 2 titik. Namun untuk menggambar garis secara digital kita perlu menggambarkan titik-titik digital yang dilalui garis tersebut. Program di bawah ini memberikan contoh penggambaran titik berdasarkan koordinat yang dimasukkan. Dalam hal ini istilah titik diartikan sebagai piksel dalam layar komputer.

```
1         Graphics g;
2         int x, y;
3         Brush piksel = (Brush)Brushes.Black;
4         private void Form1_Load(object sender, EventArgs e)
```

Algoritma 2.2 Penggambaran Garis DDA

1: Inputkan (x_1, y_1) dan (x_2, y_2)
2: Hitung $dx = x_2 - x_1$ dan $dy = y_2 - y_1$
3: **if** $|dx| > |dy|$ **then**
4: $\quad s = |dx|$
5: **else**
6: $\quad s = |dy|$
7: **end if**
8: Hitung $\Delta x = \frac{dx}{s}$ dan $\Delta y = \frac{dy}{s}$
9: Hitung $x = x1$; $y = y1$
10: **repeat**
11: $\quad$ Hitung $x = x + \Delta x$; $y = y + \Delta y$
12: **until** s kali

```
5           {
6               g=canvas.CreateGraphics();
7           }
8           private void bt_Gambarkan_Click(object sender, EventArgs e)
9           {
10              x = Convert.ToInt16(pointX.Text);
11              y = Convert.ToInt16(pointY.Text);
12              g.FillRectangle(piksel, x, y, 1, 1);
13           }
14          private void bt_Bersihkan_Click(object sender, EventArgs e)
15          {
16              canvas.Refresh();
17          }
```

Baris 1-3 adalah pendeklarasikan identifier secara global sehingga dikenal di seluruh program

Baris 1 pendeklarasikan `g` sebagai *identifier* bertipe grafik

Baris 2 pendefinsian identifier `x` dan `y` bertipe *integer*, akan digunakan untuk menampung data masukan dari layar

Baris 3 pendeklarasian identifier piksel bertipe *brush* yang berwarna hitam

Baris 4 fungsi pada *event* `Load` untuk *form* yang digunakan

Baris 6 penugasan *identifier* `g` pada objek `canvas` sebagai grafik

Baris 8 fungsi yang dieksekusi saat klik *button* **Gambarkan**

Baris 10-11 `x` menjadi tempat menampung data dari isian *textbox* `pointX`, `y` menjadi tempat menampung data dari isian *textbox* `pointY`

Baris 12 menggambarkan piksel pada posisi `x`, `y` dengan ukuran 1×1

Baris 14 fungsi untuk membersihkan kanvas

Baris 16 perintah untuk membersihkan kanvas

Algoritma 2.3 Penggambaran Garis Bresenham

1: Inputkan (x_1, y_1) dan (x_2, y_2)
2: **if** $|dx| \geq |dy|$ **then**
3: $\Delta a = |dx|$, $\Delta b = |dy|$
4: **else**
5: $\Delta a = |dy|$, $\Delta b = |dx|$
6: **end if**
7: Hitung $\nabla = 2\Delta b - \Delta a$
8: **if** $|dx| \geq |dy|$ and $dx \geq 0$ **then**
9: $m1 = M3$
10: **else if** $|dx| < |dy|$ and $dx < 0$ **then**
11: $m1 = M5$
12: **else if** $|dx| \geq |dy|$ and $dx < 0$ **then**
13: $m1 = M7$
14: **else if** $|dx| < |dy|$ and $dx \geq 0$ **then**
15: $m1 = M1$
16: **end if**
17: **if** $dx \geq 0$ and $dy \geq 0$ **then**
18: $m2 = M2$
19: **else if** $dx \geq 0$ and $dy < 0$ **then**
20: $m2 = M4$
21: **else if** $dx < 0$ and $dy < 0$ **then**
22: $m2 = M6$
23: **else if** $dx < 0$ and $dy \geq 0$ **then**
24: $m2 = M8$
25: **end if**
26: **repeat**
27: **if** $\nabla \geq 0$ **then**
28: Jalankan $m2$
29: $\nabla = \nabla + 2\Delta b - 2\Delta a$
30: **else**
31: Jalankan $m1$
32: $\nabla = \nabla + 2\Delta b$
33: **end if**
34: **until** Mencapai titik (x_2, y_2)

2.1.2 Lingkaran

Untuk membangun lingkaran setidaknya terdapat 3 algoritma yaitu (i) algoritma Dasar, (ii) algoritma Polar, dan (3) algoritma Bresenham.

Algoritma dasar penggambaran lingkaran menggunakan pendekatan formulasi lingkaran $(x - xc)^2 + (y - yc)^2 = r^2$ yang ditransformasikan ke bentuk persamaan parametrik $y = yc \pm \sqrt{r^2 - (x - xc)^2}$. Algoritmanya ditunjukkan pada Algoritma 2.4.

Algoritma 2.4 Penggambaran Lingkaran Dasar

1: **for** $x = -r$ to r **do**
2: hitung $y = yc \pm \sqrt{r^2 - (x - xc)^2}$
3: **end for**

Algoritma penggambaran lingkaran polar, pada dasarnya sama seperti algoritma penggambaran lingkaran dasar, hanya pendekatannya dilakukan melalui karakteristik geometri dan kesimetrisan yaitu $x = xc + r.\cos\theta$ dan $y = yc + r\sin\theta$Algoritmanya ditunjukkan pada Algoritma 2.5.

Algoritma 2.5 Penggambaran Lingkaran Polar

1: **for** $\theta = 0$ to 2π **do**
2: hitung $x = rcos\theta$
3: hitung $y = rsin\theta$
4: **end for**

Sama seperti penggambaran garis, algoritma penggambaran lingkaran Bresenham menjadi algoritma inti dalam penggambaran lingkaran. Konsep pendekatan dilakukan dengan tidak melibatkan kalkulasi kompleks yang membutuhkan sumberdaya yang tinggi, tetapi didekati dengan pendekatan integer yang memanfaatkan konsep simetrisitas lingkaran. Algoritmanya ditunjukkan pada Algoritma 2.6 dan Algoritma 2.7

Algoritma 2.6 Penggambaran Lingkaran Bresenham

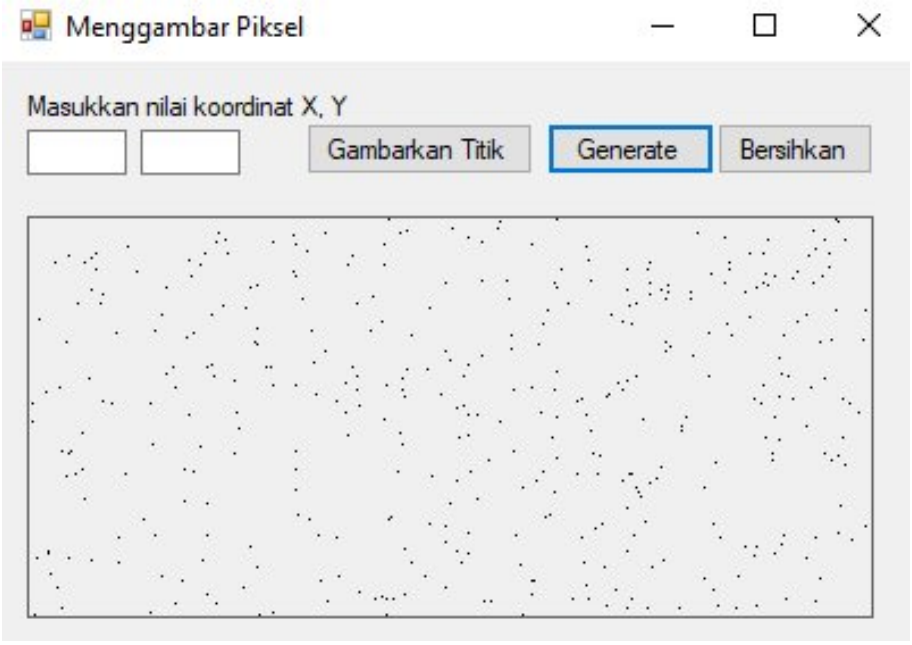
1: set (xc, yc) untuk pusat lingkaran
2: set (x, y) untuk salah satu titik dalam kurva lingkaran
3: tentukan $d = 3 - (2 * r)$, dimana r adalah jari-jari
4: panggil fungsi `drawCircle(int xc, int yc, int x, int y)`
5: **repeat**
6: *increment* nilai x
7: **if** $d < 0$ **then**
8: set $d = d + (4 * x) + 6$
9: **else**
10: set $d = d + 4 * (x - y) + 10$
11: *decrement* nilai y
12: **end if**
13: panggil fungsi `drawCircle(int xc, int yc, int x, int y)`
14: **until** $x <= y$

Algoritma 2.7 Fungsi drawCircle

```
1: putpixel (xc + x, yc + y)
2: putpixel (xc − x, yc + y)
3: putpixel (xc + x, yc − y)
4: putpixel (xc − x, yc − y)
5: putpixel (xc + y, yc + x)
6: putpixel (xc − y, yc + x)
7: putpixel (xc + y, yc − x)
8: putpixel (xc − y, yc − x)
```

2.2 Latihan

1. **Garis**
   (a) Lengkapi dan cobalah program contoh di atas
   (b) Buatlah sebuah fungsi untuk menggambar garis dimana parameter fungsinya adalah koordinat x, y
   (c) Cobalah fungsi yang anda buat dengan mengirimkan nilai x, y secara beruntun, anda bisa gunakan bilangan acak (Random) Contoh hasilnya seperti berikut ini.

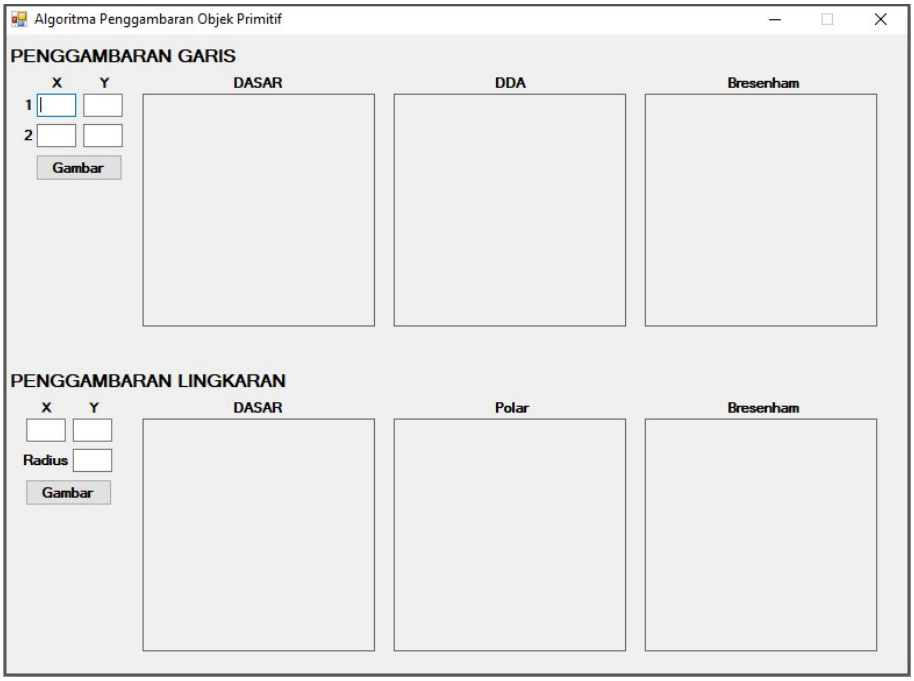


2. **Lingkaran**
   (a) Lihat Tugas Praktikum 6

2.3 Praktikum

1. Implementasikan algoritma penggambaran garis dasar

2. Implementasikan algoritma penggambaran garis DDA
3. Implementasikan algoritma penggambaran garis Bresenham
4. Implementasikan algoritma penggambaran lingkaran dasar
5. Implementasikan algoritma penggambaran lingkaran Polar
6. Implementasikan algoritma penggambaran lingkaran Bresenham
7. Integrasikan semua implementasi penggambaran objek promitif dalam sebuah program komputer, dan tampilkan hasilnya seperti contoh berikut.



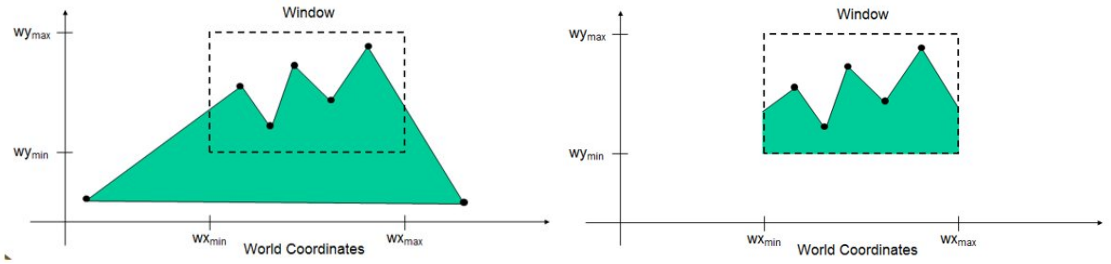
8. Dokumentasikan langkah-langkah yang dilakukan dari langkah 1 sampai dengan langkah 7 dan tuliskan dalam bentuk sebuah Laporan Praktikum dan di*submit* ke LMS LIVE! atau asisten laboratorium.

Praktikum 3

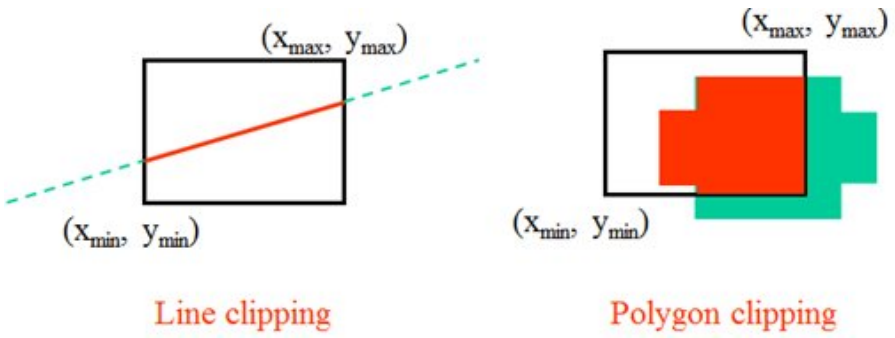
Windowing & Clipping

3.1 Teori

Windowing dan *Clipping* merupakan dua konsep umum yang selalu dijumpai dalam Grafika Komputer. Selain itu kedua istilah itu umum dilakukan dalam kegiatan yang berkaitan dengan grafis, tetapi dengan istilah yang berbeda. *windowing* ada sebuah konsep menempatkan tampilan objek nyata pada alat. Karena keterbatasan alat, maka terkadang hanya sebagian saja *view* yang dilihat secara nyata dapat ditampilkan pada alat atau pada dasarnya adalah layar komputer. Gambar berikut mengilustrasikan proses *windowing*.



Clipping adalah metode untuk memotong objek yang ada pada windows coordinate yang proses prosesnya dilakukan berdasarkan perhitungan posisi dan ukuran windows yang disesuaikan dengan kapasitas alat. Dengan teknik ini maka hanya objek yang berada pada area yang menjadi perhatian saja yang terlihat. Proses ini merupakan hal yang biasa dengan teknologi yang ada dewasa ini (*cropping, cutting*), namun proses internal pemrograman di dalamnya tidak sesederhana memakainya. Ilustrasi *clipping* bisa dilihat pada gambar di bawah ini.



Proses *clipping* dilakukan dengan menerapkan formula berikut.

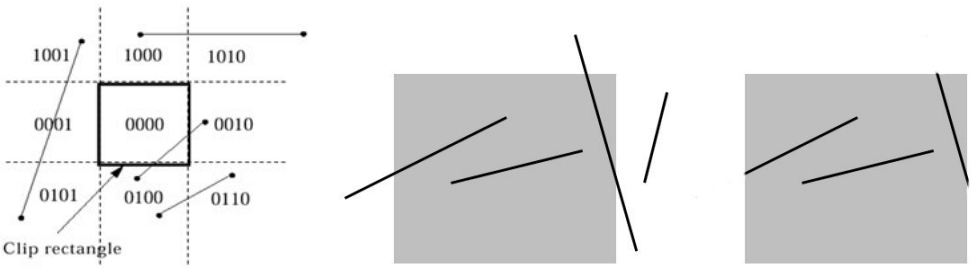
$$x_v = x_{v\,\min} + (x_w - x_{w\,\min}) \cdot \frac{x_{v\,\max} - x_{v\,\min}}{x_{w\,\max} - x_{w\,\min}}$$

$$y_v = y_{v\,\min} + (y_w - y_{w\,\min}) \cdot \frac{y_{v\,\max} - y_{v\,\min}}{y_{w\,\max} - y_{w\,\min}}$$

x_v dan y_v adalah koordinat objek dalam alat (*viewport*), x_w dan y_w adalah koordianat objek nyata dalam *windows*, $x_{v\,\min}$, $y_{v\,\min}$, $x_{v\,\max}$, dan $x_{v\,\max}$ adalah koordinat ujung kiri bawah dan kanan atas dari *viewport*, dan $x_{w\,\min}$, $y_{w\,\min}$, $x_{w\,\max}$, dan $x_{w\,\max}$ adalah koordinat ujung kiri bawah dan kanan atas dari *windows*. Jadi yang dicari adalah koordinat objek dalam *viewport* berdasarkan data dari koordinat objek dalam *windows*, ukuran *windows*, dan ukuran *viewport*.

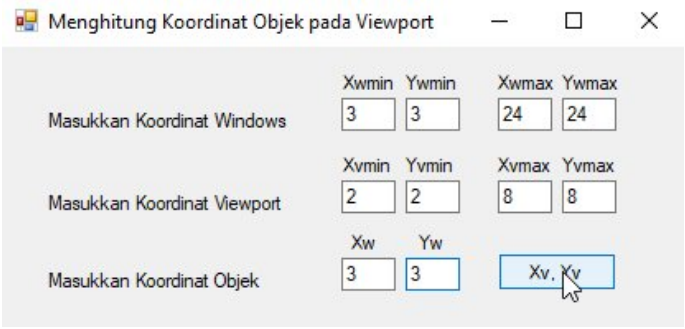
Misalnya koordinat *windows* adalah (3, 3, 24, 24), koordinat *viewport* adalah (2, 2, 8, 8), maka untuk sebuah objek pada *windows* dengan koordinat (3, 3) akan memiliki koordinat (2, 2) pada *viewport*.

Algortima yang umum dalam proses *windowing* dan *clipping* adalah algoritma Cohen-Sutherland. Selain itu juga ada algoritma lain yang dikembangkan oleh Liang-Barsky. Secara sederhana algoritma ini menerapkan konsep dimana objek yang ada di dalam *windows* akan kelihatan, dan objek di luar *windows* akan dipotong. Konsepnya menerapkan *bitcode* yang terdiri dari kode 4 bit seperti berikut ini.



3.2 Latihan

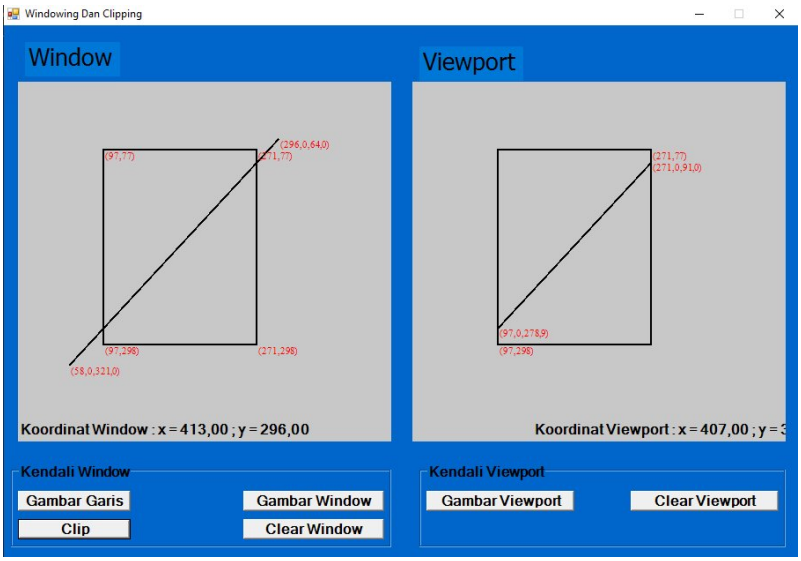
1. Buatlah program sederhana untuk menerapkan formula *clipping* yang dijelaskan pada bagian Teori di atas. Contoh tampilannya bisa seperti ini (silakan anda berkreasi sendiri).



3.3 Praktikum

Pada praktikum ini akan dilakukan implementasi dari teori dan konsep yang mendasari *windowing* dan *clipping* berdasarkan algoritma yang dibuat oleh Cohen-Sutherland dan Liang-Barsky.

1. Implementasikan salah satu algoritma *clipping* (Cohen-Sutherland atau Liang-Barsky, atau yang lain) pada komputer anda. Anda bisa membuat sendiri atau mencari modul yang sudah ada. Gambarannya bisa seperti ini.



2. Dokumentasikan langkah-langkah serta *source program* yang digunakan tuliskan dalam bentuk sebuah Laporan Praktikum dan di*submit* ke LMS LIVE! atau asisten laboratorium.

Praktikum 4

Transformasi Dua Dimensi

Praktikum Transformasi Dua Dimensi terdiri dari dua buah praktikum yaitu Praktikum Transformasi Dua Dimensi Dasar dan Praktikum Transformasi Dua Dimensi Lanjut.

4.1 Transformasi Dua Dimensi Dasar

4.1.1 Teori

Dalam pemodelan objek dua dimensi (2D), objek yang digambarkan dalam ruang dua dimensi dapat dimodifikasi. Pemodifikasian objek ini dapat dilakukan dengan melakukan berbagai operasi fungsi atau operasi transformasi geometri. Transformasi ini dapat berupa translasi, penskalaan, rotasi, dan refleksi. Selain itu ada juga transformasi shearing dan transformasi gabungan.

Proses transformasi dilakukan dengan mengalikan matriks objek dengan matriks transformasi, sehingga menghasilkan matriks baru yang berisi koordinat objek hasil transformasi. Secara umum formulasi transformasi dapat diformulasikan sebagai berikut: jika diketahui sebuah objek dua dimensi dengan kumpulan vektor posisi tertentu yang diwakili sebagai matriks A ditransformasikan dengan matriks transformasi T, maka operasinya dapat dituliskan sebagai:

$$A^* = A \otimes T$$

dimana $[A^*]$ adalah matriks objek baru hasil transformasi. Apabila A diwakili dengan elemen matriks $[x, y]$ dan $[T]$ diwakili dengan elemen $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$, maka hasil operasi transformasi matriksnya dapat dituliskan sebagai berikut :

$$A^* = A \otimes T = \begin{bmatrix} x & y \end{bmatrix} \otimes \begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} ax + cy & bx + dy \end{bmatrix}$$

Translasi adalah pergeseran, dalam hal ini objek digeser sehingga posisinya berpindah tanpa mengubah bentuk dari objek. Rotasi adalah perputaran objek, dimana objek diputar berdasarkan sudut tertentu baik secara CW (*clockwise*) maupun

CCW (*counterclockwise*) dengan pusat rotasi sumbu origin yaitu titik $(0, 0)$. Refleksi adalah pencerminan, dimana objek hasil transformasi seolah-olah berada pada cermin, dengan posisi cermin ada pada sumbu utama (horizontal, vertikal, diagonal). Rotasi dan refleksi, sama seperti translasi, tidak mengubah bentuk (*shape*) objek. Penskalaan memiliki karakteristik berbeda dimana objek hasil operasi berubah bentuk, dalam hal ini menjadi besar (*zoom-in*) atau menjadi kecil (*zoom-out*).

Translasi sebuah objek dengan nilai translasi T_x untuk translasi pada sumbu x dan T_y untuk translasi pada sumbu y merupakan operasi adisi dimana setiap elemen matrikx objek ditambahkan dengan nilai translasinya, yang diformulasikan sebagai berikut:

- $x^* = x + T_x$ untuk translasi pada sumbu x
- $y^* = y + T_y$ untuk translasi pada sumbu y

Rotasi objek dengan pusat origin dan sudut rotasi θ dilakukan operasi multiplikasi matriks berikut:

- $\begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$ untuk rotasi CCW dengan pusat *origin*, dan
- $\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$ untuk rotasi CW dengan pusat *origin*

Refleksi dilakukan pada sumbu utama dan diformulasikan sebagai berikut:

- pada sumbu x atau $y = 0$ adalah $\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
- pada sumbu y atau $x = 0$ adalah $\begin{bmatrix} -1 & 0 \\ 0 & 0 \end{bmatrix}$
- pada sumbu $y = x$ adalah $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
- pada sumbu $y = -x$ adalah $\begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix}$

Dilatasi atau penskalaan dilakukan dengan operasi perkalian objek dengan matriks transformasi umum $\begin{bmatrix} a & 0 \\ 0 & d \end{bmatrix}$ dimana a dan d adalah nilai dilatasi. Apabila nilai $a = d$ dan bernilai > 1 maka terjadi *enlargement*, sedangkan apabila nilai $a = d$ dan bernilai < 1 maka terjadi *compression*.

4.1.2 Latihan

Operasi transformasi sebagaian besar menggunakan operasi multiplikasi matriks. Dalam implementasinya pada program komputer, operasi perkalian matriks $C = A \otimes B$ dapat ditunjukkan dengan Algoritma 4.1 :

Algoritma 4.1 Perkalian matriks

1: Inputkan matriks A dan B
2: Deklarasikan matriks C dengan ukuran yang sesuai
3: **for** i dari 1 sampai n **do**
4: **for** j dari 1 sampai p **do**
5: $jum = 0$
6: **for** k dari 1 sampai m **do**
7: $jum \leftarrow jum + A_{ik} \times B_{kj}$
8: **end for**
9: $C_{ij} \leftarrow jum$
10: **end for**
11: **end for**
12: **return** C

1. Buatlah program perkalian matriks menggunakan bahasa C# dengan menerapkan algoritma di atas. Contoh *user-computer interaction* adalah sebagai berikut (apabila anda punya ide lain dipersilakan):

```
Masukkan matriks A
8
7
6
10
Matriks A adalah :
8       7
6       10
Masukkan matriks B
4
3
2
1
Matriks B adalah :
4       3
2       1
Perkalian Matriks A x B adalah :
46      31
44      28
```

4.1.3 Praktikum

1. Integrasikan semua operasi Transformasi Dua Dimensi dalam sebuah program komputer. Objek yang ditransformasikan bisa anda buat sendiri atau digenerate oleh komputer. Operasi transformasi meliputi Translasi, Rotasi dengan pusat (0,0), Refleksi, dan Dilatasi. Parameter yang dimasukkan harus fleksibel yaitu :

   (a) Nilai translasi T_x dan T_y

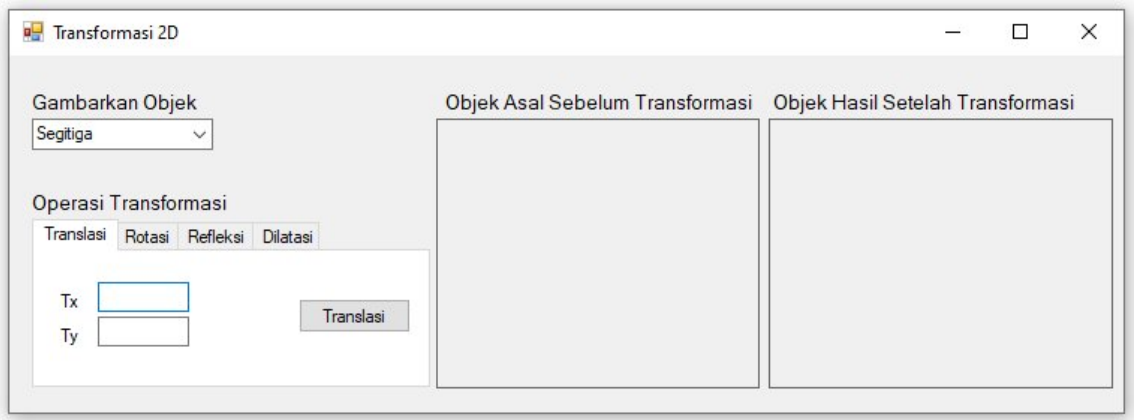
(b) Besaran sudut rotasi θ dan arah rotasi CW atau CCW

(c) Posisi cermin pada sumbu utama x, y, $y = x$, dan $y = -x$

(d) Besaran dilatasi

2. Dokumentasikan langkah-langkah yang dilakukan tuliskan dalam bentuk sebuah Laporan Praktikum dan di*submit* ke LMS LIVE! atau asisten laboratorium.

Contoh GUI nya kira-kira bisa seperti berikut ini.



4.2 Transformasi Dua Dimensi Lanjut

4.2.1 Teori

Pada hakekatnya transformasi dua dimensi dasar dan transformasi dua dimensi lanjut atau homogen hanya berbeda dalam penggunaan matriks transformasinya. Matriks Transformasi Umum atau MTU yang digunakan sebelumnya berukuran 2×2, namun pada Transformasi dua dimensi homogen digunakan MTU berukuran 3×3. Komposisi MTU Transformasi Dua Dimensi Homogen digambarkan sebagai berikut:

$$\begin{bmatrix} a & b & p \\ c & d & q \\ m & n & s \end{bmatrix}$$

a, b, c, d adalah elemen rotasi, refleksi dan penskalaan lokal; m, n adalah elemen translasi; p, q adalah elemen proyeksi; dan s adalah elemen penskalaan global.

Dengan menggunakan transformasi homogen maka terjadi fleksibilitas untuk operasi-operasi rotasi dan refleksi sehingga untuk rotasi tidak dibatasi oleh pusat rotasi harus pada origin dan untuk refleksi bisa dilakukan operasi pada sumbu yang sembarang.

Rotasi dengan Pusat Sembarang Jika sebuah objek dirotasikan sebesar θ dengan pusat rotasi (m, n), maka langkah-langkah yang harus dilakukan adalah

1. Translasikan pusat rotasi ke $(0, 0)$; karena yang kita ketahui hanyalah rumus rotasi pada origin
2. Lakukan rotasi sebesar yang diinginkan

3. Re-translasi pusat rotasi ke posisi semula

Releksi pada Sumbu Sembarang Jika sebuah objek direfleksikan pada sumbu $y = mx + c$ maka langkah-langkah yang harus dilakukan adalah

1. Translasikan sumbu sedemikian rupa sehingga menyentuh titik origin
2. Rotasikan cermin sehingga berimpit dengan salah satu sumbu utama
3. Refleksikan objek
4. Re-rotasi
5. Re-translasi

Hal lain yang membedakan dengan transformasi dua dimensi dasar adalah penambahan kolom ke-tiga dalam matriks objek sehingga ordo matriks menjadi $m \times 3$ dimana m adalah jumlah titik-titik dijital dari objek.

4.2.2 Latihan

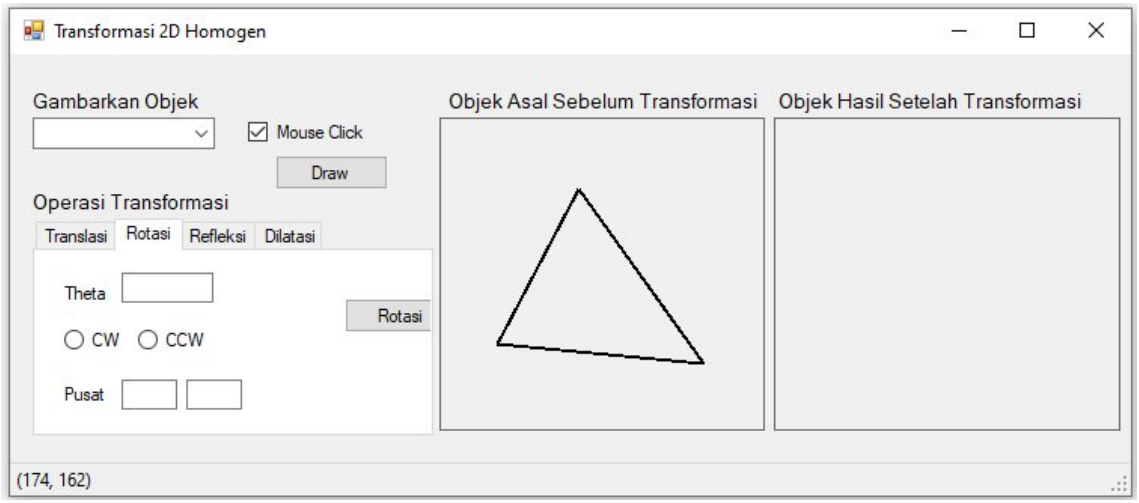
Tinjau kembali program operasi perkalian matriks yang anda buat, dan pastikan mampu mengatasi operasi perkalian matriks untuk transfromasi dua dimensi lanjut.

4.2.3 Praktikum

1. Modifikasi program yang anda buat untuk transformasi dua dimensi dasar sehingga mampu menangani operasi transformasi homogen.

2. *Enhance* program anda sehingga memiliki karekteristik visual-interaktif, misalnya menambahkan koordinat sumbu sehingga program lebih komunikatif, program mampu membuat objek berdasarkan *click* dari *mouse* dan lain sebagainya (silakan berinovasi).

   Contoh GUI nya kira-kira bisa seperti berikut ini.



Praktikum 5

Transformasi Tiga Dimensi

Praktikum 6

Proyeksi Paralel

Praktikum 7

Proyeksi Perspektif

Part II

Grafika Komputer Menggunakan GDI+ pada .NET

Praktikum 8

Sistem Koordinat

A transformation is a process that changes graphics objects from one state to another. Rotation, scaling, reflection, translation, and shearing are some examples of transformation. Transformations can be applied not only to graphics shapes, curves, and images, but even to image colors. In this chapter we will cover the following topics:

- The basics of transformation, including coordinate systems and matrices
- Global, local, and composite transformations
- Transformation functionality provided by the Graphics class
- Transformation concepts such as shearing, rotation, scaling, and translation
- The Matrix and ColorMatrix classes, and their role in transformation
- Matrix operations in image processing, including rotation, translation, shearing, and
- scaling
- Color transformation and recoloring
- Text transformation
- Composite transformations and the matrix order

Any drawing process involves a source and a destination. The source of a drawing is the application that created it, and the destination is a display or printer device. For example, the process of drawing a simple rectangle starts with a command telling GDI+ to draw on the screen, followed by GDI+ iterating through multiple steps before it finally renders a rectangle on the screen. In the same way, transformation involves some steps before it actually renders the transformed object on a device. These steps are shown in Figure 8.1, which shows that GDI+ is responsible for converting world coordinates to page coordinates and device coordinates before it can render a transformed object.

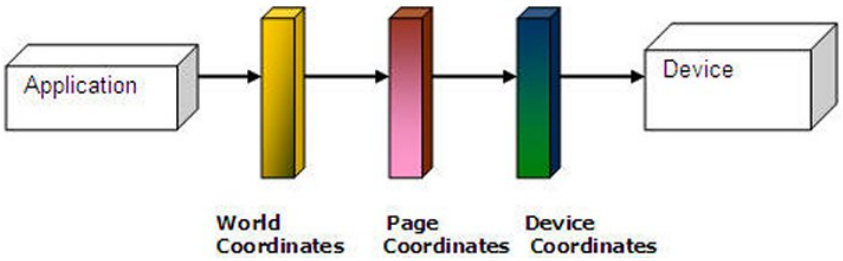


Figure 8.1 Langkah-langkah dalam Proses Transformasi

Before we discuss transformations, we need to understand coordinate systems. GDI+ defines three types of coordinate spaces: world, page, and device. When we ask GDI+ to draw a line from point $A(x_1, y_1)$ to point $B(x_2, y_2)$, these points are in the world coordinate system. Before GDI+ draws a graphics shape on a surface, the shape goes through a few transformation stages (conversions). The first stage converts world coordinates to page coordinates. Page coordinates may or may not be the same as world coordinates, depending on the transformation. The process of converting world coordinates to page coordinates is called world transformation.

The second stage converts page coordinates to device coordinates. Device coordinates represent how a graphics shape will be displayed on a device such as a monitor or printer. The process of converting page coordinates to device coordinates is called page transformation. Figure 8.2 shows the stages of conversion from world coordinates to device coordinates.

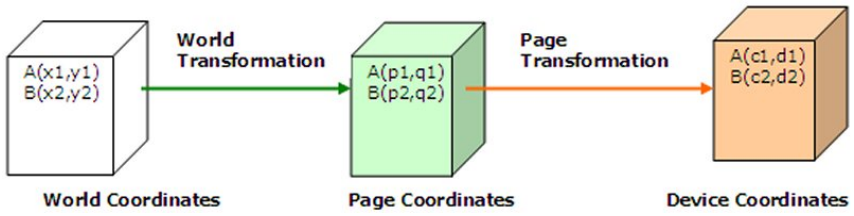


Figure 8.2 Tahapan-tahapan Transformasi

In GDI+, the default origin of all three coordinate systems is point $(0, 0)$, which is at the upper left corner of the client area. When we draw a line from point $A(0, 0)$ to point $B(120, 80)$, the line starts 0 pixels from the upper left corner in the x-direction and 0 pixels from the upper left corner in the y-direction, and it will end 120 pixels over in the x-direction and 80 pixels down in the y-direction. The line from point $A(0, 0)$ to point $B(120, 80)$ is shown in Figure 8.3.

Drawing this line programmatically is very simple. We must have a Graphics object associated with a surface (a form or a control). We can get a Graphics object in several ways. One way is to accept the implicit object provided by a form's paint event handler; another is to use the CreateGraphics method. Once we have a Graphics object, we call its draw and fill methods to draw and fill graphics objects. Listing 8.1 draws a line from starting point $A(0, 0)$ to ending point $B(120, 80)$. You can add this code to a form's paint event handler.

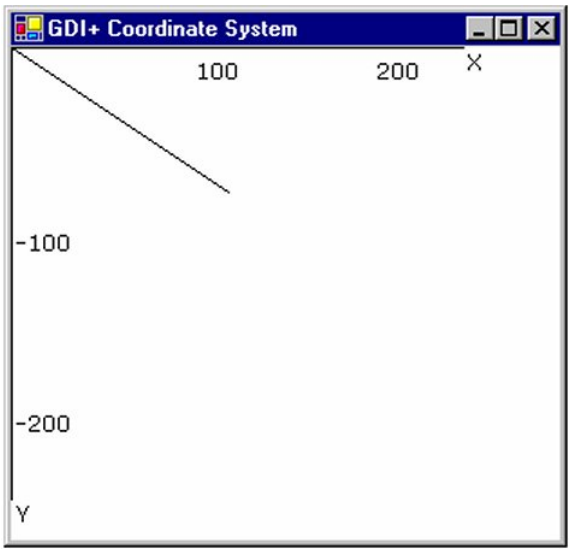


Figure 8.3 Menggambar Garis dari Titik (0,0) Ke (120,80)

Listing 8.1 Menggambar dari Titik (0, 0) ke Titik (120, 80)

```
1 Graphics g = e.Graphics;
2 Point A = new Point(0, 0);
3 Point B = new Point(120, 80);
4 g.DrawLine(Pens.Black, A, B);
```

Figure 8.3 shows the output from Listing 8.1. All three coordinate systems (world, page, and device) draw a line starting from point $(0, 0)$ in the upper left corner of the client area to point $(120, 80)$.

Now let's change to the page coordinate system. We draw a line from point $A(0, 0)$ to point $B(120, 80)$, but this time our origin is point $(50, 40)$ instead of the upper left corner. We shift the page coordinates from point $(0, 0)$ to point $(50, 40)$. The TranslateTransform method of the Graphics class does this for us. We will discuss this method in more detail in the discussion that follows. For now, let's try the code in Listing 8.2.

Listing 8.2 Menggambar Garis dari (0, 0) ke (120, 80) dengan Posisi Awal (50, 40)

```
1 Graphics g = e.Graphics;
2 g.TranslateTransform(50, 40);
3 Point A = new Point(0, 0);
4 Point B = new Point(120, 80);
5 g.DrawLine(Pens.Black, A, B);
```

Figure 8.4 shows the output from Listing 8.2. The page coordinate system now starts at point $(50, 40)$, so the line starts at point $(0, 0)$ and ends at point $(120, 80)$. The world coordinates in this case are still $(0, 0)$ and $(120, 80)$, but the page and device coordinates are $(50, 40)$ and $(170, 120)$. The device coordinates in this case are same as the page coordinates because the page unit is in the pixel (default) format.

What is the difference between page and device coordinates? Device coordinates determine what we actually see on the screen. They can be represented in many

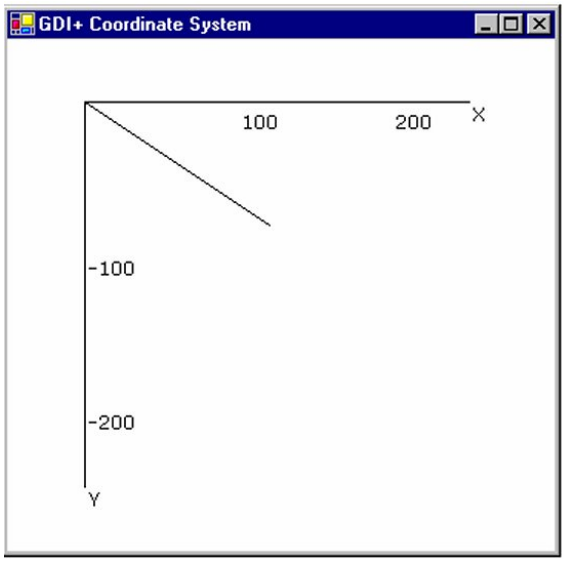


Figure 8.4 Menggambar Garis dari Titik (0,0) ke (120,80) Dimulai dari Posisi Awal (50,40)

formats, including pixels, millimeters, and inches. If the device coordinates are in pixel format, the page coordinates and device coordinates will be the same (this is typically true for monitors, but not for printers).

The `PageUnit` property of the `Graphics` class is of type `GraphicsUnit` enumeration. In Listing 8.3 we set the PageUnit property to inches. Now graphics objects will be measured in inches, so we need to pass inches instead of pixels. If we draw a line from point $(0, 0)$ to point $(2, 1)$, the line ends 2 inches from the left side and 1 inch from the top of the client area in the page coordinate system. In this case the starting and ending points are $(0, 0)$ and $(2, 1)$ in both world and page coordinates, but the device coordinate system converts them to inches. Hence the starting and ending points in the device coordinate system are $(0, 0)$ and $(192, 96)$, assuming a resolution of 96 dots per inch.

Listing 8.3 Mengeset Sistem Koordinat Alat ke Satuan Inci

```
1 g.PageUnit = GraphicsUnit.Inch;
2 g.DrawLine(Pens.Black, 0, 0, 2, 1);
```

Figure 8.5 shows the output from 8.5. The default width of the pen is 1 page unit, which in this case gives us a pen 1 inch wide.

Now let's create a new pen with a different width. Listing 8.4 creates a pen that's 1 pixel wide (it does so by dividing the number of pixels we want, in this case 1 by the page resolution, which is given by DpiX). We draw the line again, this time specifying a red color.

Listing 8.4 Menggunakan Opsi GraphicsUnit.Inches yang Disesuaikan dengan Ukuran Piksel

```
1 Pen redPen = new Pen(Color.Red, 1/g.DpiX);
2 g.PageUnit = GraphicsUnit.Inch;
3 g.DrawLine(Pens.Black, 0, 0, 2, 1);
```

Figure 8.6 shows the output from Listing 8.4.

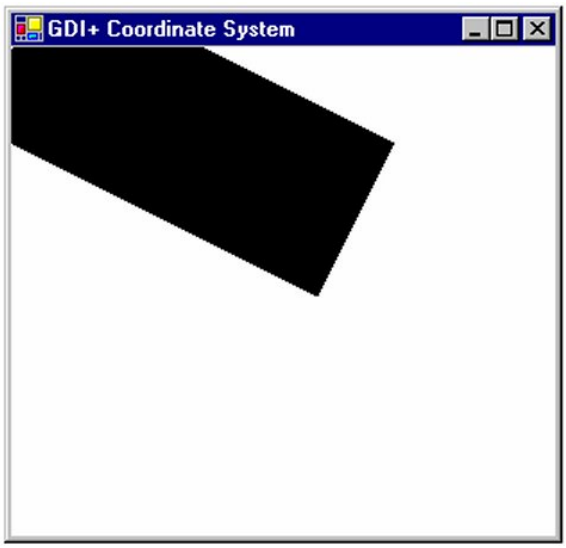


Figure 8.5 Menggambar dengan Opsi GraphicsUnit.Inches

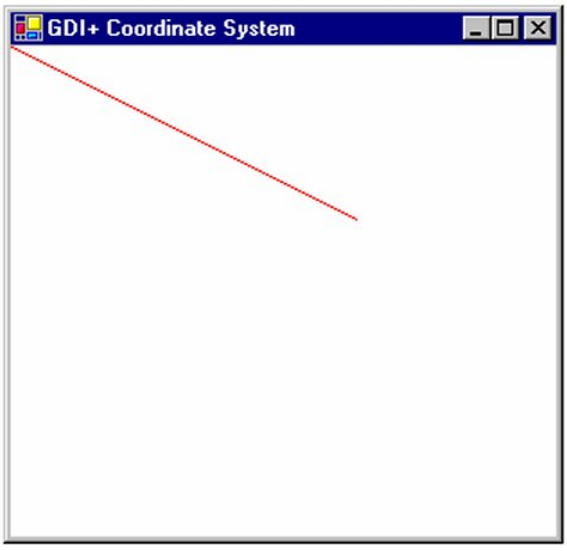


Figure 8.6 Menggunakan Opsi GraphicsUnit.Inches yang Disesuaikan dengan Ukuran Piksel

We can also combine the use of page and device coordinates. In Listing 8.5 we transform page coordinates to 1 inch from the left and 0.5 inch from the top of the upper left corner of the client area. Our new page coordinate system has starting and ending points of $(1, 0.5)$ and $(3, 1.5)$, but the device coordinate system converts them to pixels. Hence the starting and ending points in device coordinates are $(96, 48)$ and $(288, 144)$, assuming a resolution of 96 dots per inch.

Listing 8.5 Mengkombinasikan Koordinat Halaman dan Koordinat Alat

```
1 Pen redPen = new Pen(Color.Red, 1/g.DpiX);
2 g.TranslateTransform(1, 0.5f);
3 g.PageUnit = GraphicsUnit.Inch;
4 g.DrawLine(redPen, 0, 0, 2, 1);
```

Figure 8.7 shows the output from Listing 8.5.

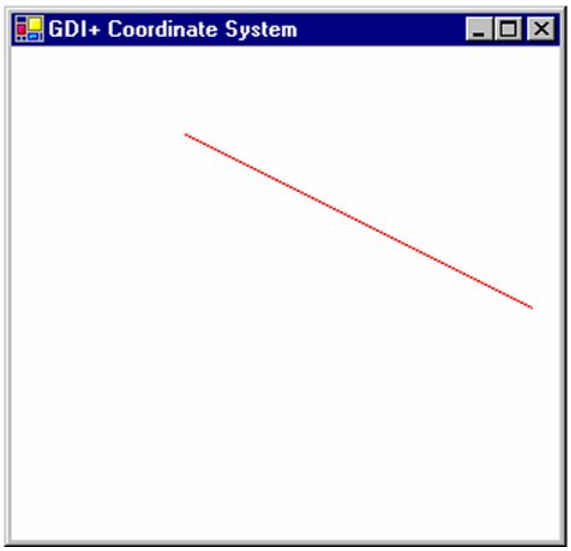


Figure 8.7 Mengkombinasikan Koordinat Halaman dan Koordinat Alat

Praktikum 9

Operasi Matriks

There are many types of transformations. Translation is a transformation of the xy plane that moves a graphics object toward or away from the origin of the surface in the x- or y-direction. For example, moving an object from point $A(x1, y1)$ to point $B(x2, y2)$ is a translation operation in which an object is being moved $(y2, y1)$ points in the y-direction.

Rotation moves an object around a fixed angle around the center of the plane. In the reflection transformation, an object moves to a position in the opposite direction from an axis, along a line perpendicular to the axis. The resulting object is the same distance from the axis as the original point, but in the opposite direction.

Simple transformations, including rotation, scaling, and reflection are called linear transformations. A linear transformation followed by translation is called an affine transformation.

The shearing transformation skews objects based on a shear factor. In the sample applications discussed throughout this chapter, will see how to use these transformations in GDI+. So far we've looked at only simple transformations. Now let's discuss some more complex transformation-related functionality defined in the .NET Framework library.

What Can You Transform?
You have just seen the basics of transforming lines. We can also transform graphics objects such as points, curves, shapes, images, text, colors, and textures, as well as colors and images used in pens and brushes.

The Matrix Class and Transformation Matrices play a vital role in the transformation process. A matrix is a multidimensional array of values in which each item in the array represents one value of the transformation operation, as we will see in the examples later in this chapter.

In GDI+, the Matrix class represents a 3×2 matrix that contains x, y, and w values in the first, second, and third columns, respectively.

NOTE: Before using the Matrix class in your applications, you need to add a reference to the `System.Drawing.Drawing2D namespace`.

We can create a Matrix object by using its overloaded constructors, which take an array of points (hold the matrix items) as arguments. The following code snippet

creates three Matrix objects from different overloaded constructors. The first Matrix object has no values for its items. The second and third objects have integer and floating point values, respectively, for the first six items of the matrix.

```
Matrix M1 = new Matrix();
Matrix M2 = new Matrix(2, 1, 3, 1, 0, 4);
Matrix M3 = new Matrix(0.0f, 1.0f, -1.0f, 0.0f, 0.0f, 0.0f);
```

The Matrix class provides properties for accessing and setting its member values. Table 9.1 describes these properties.

Table 9.1 Properti Kelas Matriks

Property	**Description**
Elements	Returns an array containing matrix elements
IsIdentity	Returns true if the matrix is an identity matrix (the value of all items equals 1 or 1.0); otherwise false
IsInvertible	Truens true if a matrix is invertible; otherwise false
OffsetX	Returns the x translation value of a matrix
OffsetY	Returns the y translation value of a matrix

The Matrix class provides methods to invert, rotate, scale, and transform matrices. The Invert method is used to reverse a matrix if it is invertible. This method takes no parameters.

NOTE: The Transform property of the Graphics class is used to apply a transformation in the form of a Matrix object. We will discuss this property in more detail in Section 10.4.

Listing 9.1 uses the Invert method to invert a matrix. We create a Matrix object and read its original values. Then we call the Invert method and read the new values.

Listing 9.1 Invers Matriks

```
1  private void InvertMenu_Click(object sender, System.EventArgs e)
2  {
3      string str = "Nilai Awal: ";
4      Matrix X = new Matrix(2, 1, 3, 1, 0, 4);
5      for(int i=0; i<X.Elements.Length; i++)
6      {
7          str += X.Elements[i].ToString();
8          str += ", ";
9      }
10     str += "\n";
11     str += "Nilai Invers: ";
12     X.Invert();
13     float[] pts = X.Elements;
14     for(int i=0; i<pts.Length; i++)
15     {
16         str += pts[i].ToString();
17         str += ", ";
18     }
19     MessageBox.Show(str);
20 }
```

The `Multiply` method multiplies a new matrix against an existing matrix and stores the result in the first matrix. Multiply takes two arguments. The first is the new matrix by which you want to multiply the existing matrix, and the second is an optional MatrixOrder argument that indicates the order of multiplication.

The `MatrixOrder` enumeration has two values: `Append` and `Prepend`. Append specifies that the new operation is applied after the preceding operation; Prepend specifies that the new operation is applied before the preceding operation during cumulative operations. Listing 9.2 multiplies two matrices. We create two Matrix objects and use the Multiply method to multiply the second matrix by the first. Then we read and display the resultant matrix.

Listing 9.2 Perkalian Dua Matriks

```
1  private void MultiplyMenu_Click(object sender, System.EventArgs e)
2  {
3      string str = null;
4      Matrix X = new Matrix(2.0f, 1.0f, 3.0f, 1.0f, 0.0f, 4.0f);
5      Matrix Y = new Matrix(0.0f, 1.0f, -1.0f, 0.0f, 0.0f, 0.0f);
6      X.Multiply(Y, MatrixOrder.Append);
7      for(int i=0; i<X.Elements.Length; i++)
8      {
9          str += X.Elements[i].ToString();
10         str += ", ";
11     }
12     MessageBox.Show(str);
13 }
```

The Reset method resets a matrix to the identity matrix (see Figure 10.21 for an example of an identity matrix). If we call the Reset method and then apply a matrix to transform an object, the result will be the original object.

Praktikum 10

Transformasi Objek

10.1 Rotasi

The `Rotate` and `RotateAt` methods are used to rotate a matrix. The `Rotate` method rotates a matrix at a specified angle. This method takes two arguments: a floating point value specifying the angle, and (optionally) the matrix order. The `RotateAt` method is useful when you need to change the center of the rotation. Its first parameter is the angle; the second parameter (of type float) specifies the center of rotation. The third (optional) parameter is the matrix order.

Listing 10.1 simply creates a Graphics object using the `CreateGraphics` method and calls `DrawLine` and `FillRectangle` to draw a line and fill a rectangle, respectively.

Listing 10.1 Menggambar Garis dan Segiempat Penuh

```
1 private void Rotate_Click(object sender, System.EventArgs e)
2 {
3     Graphics g = this.CreateGraphics();
4     g.Clear(this.BackColor);
5     g.DrawLine(new Pen(Color.Green, 3), new Point(120, 50), new Point
          (200, 50));
6     g.FillRectangle(Brushes.Blue, 200, 100, 100, 60);
7     g.Dispose();
8 }
```

Figure 10.1 shows the output from Listing 10.1.

Now let's rotate our graphics objects, using the Matrix object. In Listing 10.2 we create a Matrix object, call its Rotate method to rotate the matrix 45 degrees, and apply the Matrix object to the Graphics object by setting its Transform property.

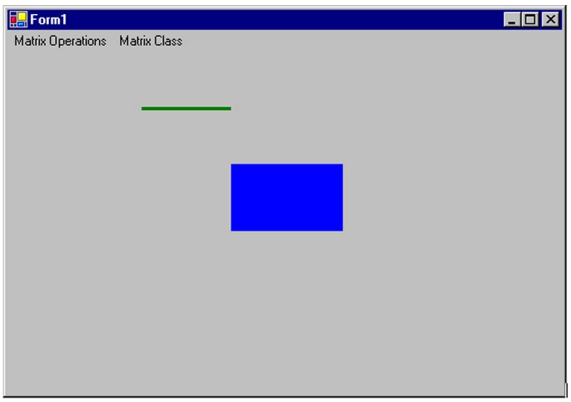


Figure 10.1 Menggambar Garis dan Segiempat Penuh

Listing 10.2 Memutar Objek Grafika

```
1  private void Rotate_Click(object sender, System.EventArgs e)
2  {
3      Graphics g = this.CreateGraphics();
4      g.Clear(this.BackColor);
5      Matrix X = new Matrix();
6      X.Rotate(45, MatrixOrder.Append);
7      g.Transform = X;
8      g.DrawLine(new Pen(Color.Green, 3), new Point(120, 50), new Point
           (200, 50));
9      g.FillRectangle(Brushes.Blue, 200, 100, 100, 60);
10     g.Dispose();
11 }
```

Figure 10.2 shows the new output. Both objects (line and rectangle) have been rotated 45 degrees.

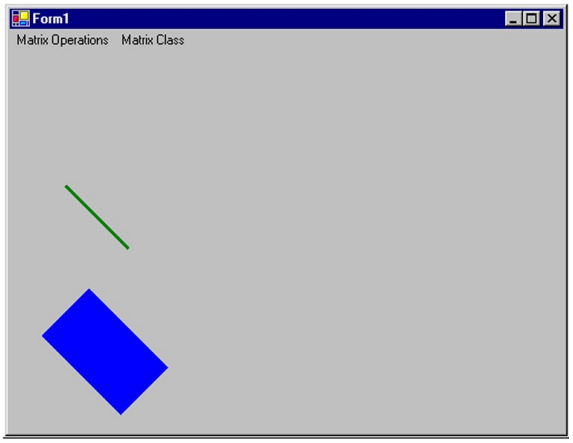


Figure 10.2 Memutar Objek Grafika

Now let's replace `Rotate` with `RotateAt`, as in Listing 10.3

Listing 10.3 Memutar Objek Grafika dengan Metode RotateAt

```
1  private void RotateAtMenu_Click(object sender, System.EventArgs e)
2  {
3      Graphics g = this.CreateGraphics();
4      g.Clear(this.BackColor);
5      Matrix X = new Matrix();
6      PointF pt = new PointF(180.0f, 50.0f);
7      X.RotateAt(45, pt, MatrixOrder.Append);
8      g.Transform = X;
9      g.DrawLine(new Pen(Color.Green, 3), new Point(120, 50), new Point
           (200, 50));
10     g.FillRectangle(Brushes.Blue, 200, 100, 100, 60);
11     g.Dispose();
12 }
```

This new code generates Figure 10.3.

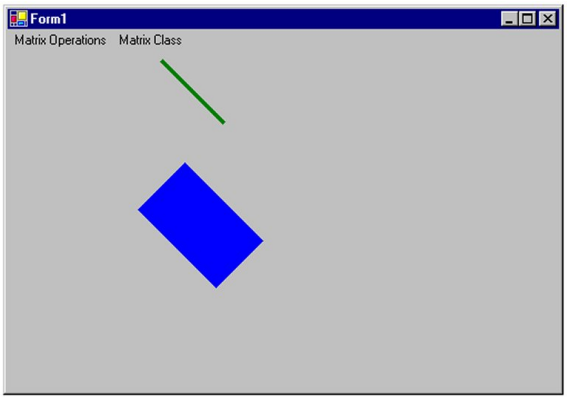


Figure 10.3 Memutar Objek Grafika dengan Metode RotateAt

If we call the `Reset` method in Listing 10.3 after `RotateAt` and before `g.Transform`, like this:

```
X.RotateAt(45, pt, MatrixOrder.Append);
X.Reset();
g.Transform = X;
```

The revised code generates Figure 10.4, which is the same as Figure 10.2. There is no rotation because the `Reset` method resets the transformation.

10.2 Penskalaan

The Scale method scales a matrix in the x- and y-directions. This method takes two floating values (scale factors), for the x- and y-axes, respectively. In Listing 10.11 we draw a rectangle with a width of 20 and a height of 30. Then we create a Matrix object and scale it by calling its Scale method with arguments 3 and 4 in the x- and y-directions, respectively.

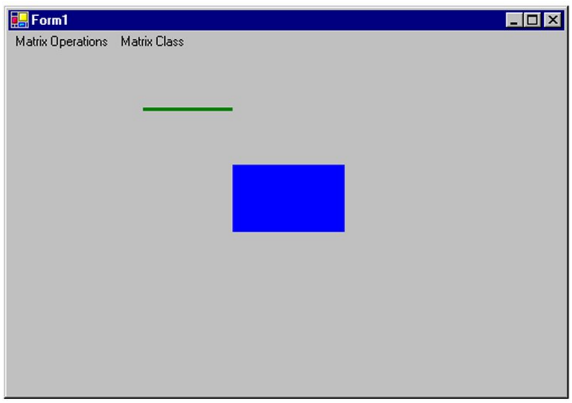


Figure 10.4 Mereset Transformasi menggunakan Metode Matrix.Reset()

Listing 10.4 Penskalaan Objek Grafika

```
1  private void Scale_Click(object sender, System.EventArgs e)
2  {
3      Graphics g = this.CreateGraphics();
4      g.Clear(this.BackColor);
5      g.FillRectangle(Brushes.Blue, 20, 20, 20, 30);
6      Matrix X = new Matrix();
7      X.Scale(3, 4, MatrixOrder.Append);
8      g.Transform = X;
9      g.FillRectangle(Brushes.Blue, 20, 20, 20, 30);
10     g.Dispose();
11 }
```

Figure 10.5 shows the output from Listing 10.4. The first rectangle is the original rectangle; the second rectangle is the scaled rectangle, in which the x position (and width) is scaled by 3, and the y position (and height) is scaled by 4.

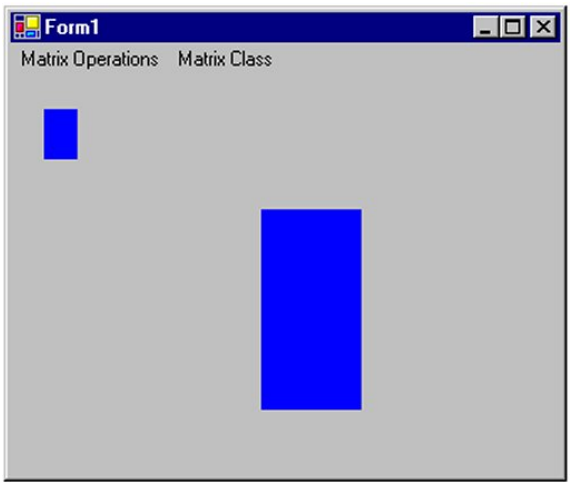


Figure 10.5 Penskalaan sebuah Segiempat

10.3 Shearing

The Shear method provides a shearing transformation and takes two floating point arguments, which represent the horizontal and vertical shear factors, respectively. In Listing 10.5 we draw a filled rectangle with a hatch brush. Then we call the Shear method to shear the matrix by 2 in the vertical direction, and we use Transform to apply the Matrix object.

Listing 10.5 Shearing Objek Grafika

```
1  private void Shear_Click(object sender, System.EventArgs e)
2  {
3      Graphics g = this.CreateGraphics();
4      g.Clear(this.BackColor);
5      HatchBrush hBrush = new HatchBrush(HatchStyle.DarkVertical, Color.
           Green, Color.Yellow);
6      g.FillRectangle(hBrush, 100, 50, 100, 60);
7      Matrix X = new Matrix();
8      X.Shear(2, 1);
9      g.Transform = X;
10     g.FillRectangle(hBrush, 10, 100, 100, 60);
11     hBrush.Dispose();
12     g.Dispose();
13 }
```

Figure 10.6 shows the output from Listing 10.5. The first rectangle in this figure is the original; the second is sheared.

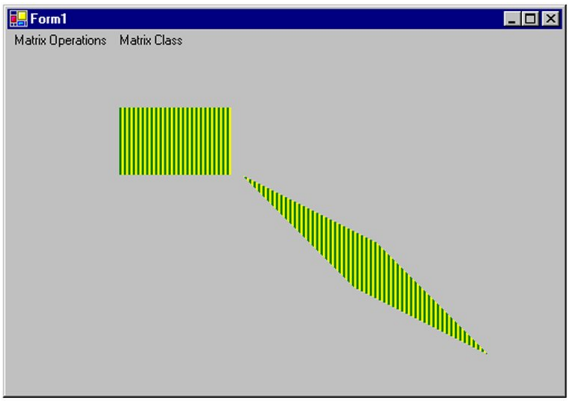


Figure 10.6 Shearing Segiempat menggunakan Metode Shear

10.4 Translasi

The Translate method translates objects by the specified value. This method takes two floating point arguments, which represent the x and y offsets. For example, Listing 10.6 translates the original rectangle by 100 pixels each in the x- and y-directions.

Listing 10.6 Mentranslasikan Objek Grafika

```
1  private void Translate_Click(object sender, System.EventArgs e)
2  {
3      Graphics g = this.CreateGraphics();
4      g.Clear(this.BackColor);
5      g.FillRectangle(Brushes.Blue, 50, 50, 100, 60);
6      Matrix X = new Matrix();
7      X.Translate(100, 100);
8      g.FillRectangle(Brushes.Blue, 50, 50, 100, 60);
9      g.Dispose();
10 }
```

Here we draw two rectangles with a width of 100 and a height of 60. Both rectangles start at (50, 50), but the code generates Figure 10.7. Even though the rectangles were drawn with the same size and location, the second rectangle after translation is now located 100 points away in the x- and y-directions from the first rectangle.

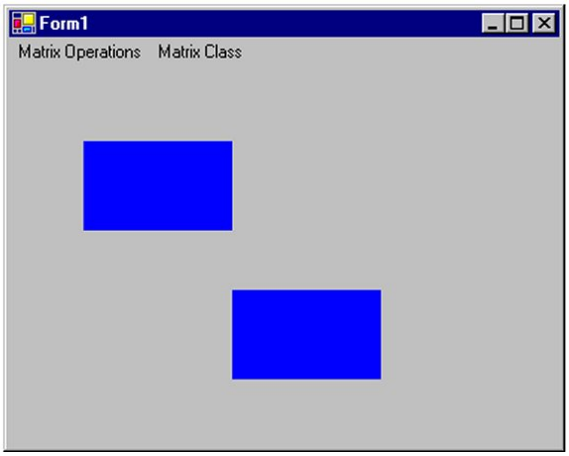


Figure 10.7 Pentranslasian sebuah Segiempat

Praktikum 11

Transformasi Komposit dan Graphics Path

In previous part we saw that the Graphics class provides some transformation-related members. Before we move to other transformation-related classes, let's review the transformation functionality defined in the Graphics class, as described in Table 11.1. We will see how to use these members in the examples throughout this chapter.

Table 11.1 Member yang Berkaitan dengan Transformasi dalam kelas Graphics

Member	**Description**	**Type**
Elements	Represents the world transformation for a Graphics object. Both get and set	Property
MultiplyTransform	Multiplies the world transformation of a Graphics object and a Matrix object. The Matrix object specifies the transformation action (scaling, rotation, or translation	Method
ResetTransform	Resets the world transformation matrix of a Graphics object to the identity matrix	Method
RotateTransform	Applies a specified rotation to the transformation matrix of a Graphics object	Method
ScaleTransform	Applies a specified scaling operation to the transformation matrix of a Graphics object by prepending it to the object's transformation matrix	Method
TransformPoints	Transform an array of points from one coordinate space to another using the current	Method
TranslateClips	Translates the clipping region of a Graphics object by specified amounts in the horizontal and vertical directions	Method
TranslateTransform	Prepends the specified translation to the tranaformation matrix of a Graphics object	Method

The Transform property of the Graphics class represents the world transformation of a Graphics object. It is applied to all items of the object. For example, if you have a rectangle, an ellipse, and a line and set the Transform property of the Graphics object, it will be applied to all three items. The Transform property is a Matrix object. The following code snippet creates a Matrix object and sets the Transform property:

```
Matrix X = new Matrix();
X.Scale(2, 2, MatrixOrder.Append);
g.Transform = X;
```

The transformation methods provided by the Graphics class are MultiplyTransform, ResetTransform, RotateTransform, ScaleTransform, TransformPoints, TranslateClip, and TranslateTransform. The MultiplyTransform method multiplies a transformation matrix by the world transformation coordinates of a Graphics object. It takes an argument of Matrix type. The second argument, which specifies the order of multiplication operation, is optional. The following code snippet creates a Matrix object with the Translate transformation. The MultiplyTransform method multiplies the Matrix object by the world coordinates of the Graphics object, translating all graphics items drawn by the Graphics object.

```
Matrix X = new Matrix();
X. Translate(200.0F, 100.0F);
g.MultiplyTransform(X, MatrixOrder.Append);
```

RotateTransform rotates the world transform by a specified angle. This method takes a floating point argument, which represents the rotation angle, and an optional second argument of MatrixOrder. The following code snippet rotates the world transformation of the Graphics object by 45 degrees:

```
g.RotateTransform(45.0F, MatrixOrder.Append);
```

The ScaleTransform method scales the world transformation in the specified x- and y-directions. The first and second arguments of this method are x- and y-direction scaling factors, and the third optional argument is MatrixOrder. The following code snippet scales the world transformation by 2 in the x-direction and by 3 in the y-direction:

```
g.ScaleTransform(2.0F, 3.0F, MatrixOrder.Append);
```

The TranslateClip method translates the clipping region in the horizontal and vertical directions. The first argument of this method represents the translation in the x-direction, and the second argument represents the translation in the y-direction:

```
e.Graphics.TranslateClip(20.0f, 10.0f);
```

The TranslateTransform method translates the world transformation by the specified x- and y-values and takes an optional third argument of MatrixOrder:

```
g.TranslateTransform(100.0F, 0.0F, MatrixOrder.Append);
```

We will use all of these methods in our examples.

11.1 Transformasi Global, Lokal, dan Komposit

Transformations can be divided into two categories based on their scope: global and local. In addition, there are composite transformations. A global transformation is

applicable to all items of a Graphics object. The Transform property of the Graphics class is used to set global transformations.

A composite transformation is a sequence of transformations. For example, scaling followed by translation and rotation is a composite translation. The MultiplyTransform, RotateTransform, ScaleTransform, and TranslateTransform methods are used to generate composite transformations.

Listing 11.1 draws two ellipses and a rectangle, then calls ScaleTransform, TranslateTransform, and RotateTransform (a composite transformation). The items are drawn again after the composite transformation.

Listing 11.1 Penerapan Transformasi Komposit

```
1  private void GlobalTransformation_Click(object sender, System.EventArgs
       e)
2  {
3      Graphics g = this.CreateGraphics();
4      g.Clear(this.BackColor);
5      Pen bluePen = new Pen(Color.Blue, 2);
6      Point pt1 = new Point(10, 10);
7      Point pt2 = new Point(20, 20);
8      Color [] lnColors = {Color.Black, Color.Red};
9      Rectangle rect1 = new Rectangle(10, 10, 15, 15);
10     LinearGradientBrush lgBrush1 = new LinearGradientBrush(rect1, Color.
           Blue, Color.Green, LinearGradientMode.BackwardDiagonal);
11     LinearGradientBrush lgBrush = new LinearGradientBrush(pt1, pt2,
           Color.Red, Color.Green);
12     lgBrush.LinearColors = lnColors;
13     lgBrush.GammaCorrection = true;
14     g.FillRectangle(lgBrush, 150, 0, 50, 100);
15     g.DrawEllipse(bluePen, 0, 0, 100, 50);
16     g.FillEllipse(lgBrush1, 300, 0, 100, 100);
17     g.ScaleTransform(1, 0.5f);
18     g.TranslateTransform(50, 0, MatrixOrder.Append);
19     g.RotateTransform(30.0f, MatrixOrder.Append);
20     g.FillEllipse(lgBrush1, 300, 0, 100, 100);
21     g.RotateTransform(15.0f, MatrixOrder.Append);
22     g.FillRectangle(lgBrush, 150, 0, 50, 100);
23     g.RotateTransform(15.0f, MatrixOrder.Append);
24     g.DrawEllipse(bluePen, 0, 0, 100, 50);
25     lgBrush1.Dispose();
26     lgBrush.Dispose();
27     bluePen.Dispose();
28     g.Dispose();
29 }
```

Figure 11.1 shows the output from Listing 11.1.

11.2 Graphics Path

A local transformation is applicable to only a specific item of a Graphics object. The best example of local transformation is transforming a graphics path. The Translate method of the GraphicsPath class translates only the items of a graphics

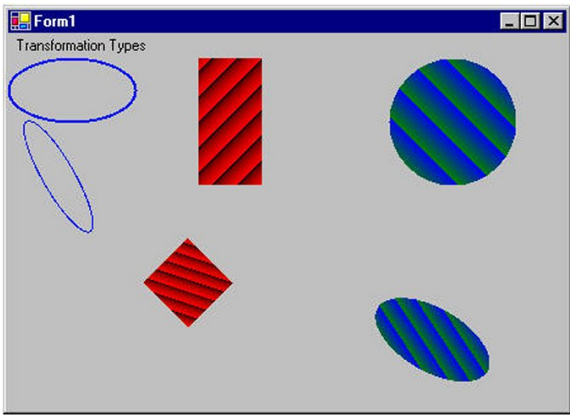


Figure 11.1 Transformasi Komposit

path. Listing 11.2 translates a graphics path. We create a Matrix object and apply rotate and translate transformations to it.

Listing 11.2 Translasi Item Graphics Path

```
1  private void LocalTransformation_Click(object sender, System.EventArgs
       e)
2  {
3      Graphics g = this.CreateGraphics();
4      g.Clear(this.BackColor);
5      GraphicsPath path = new GraphicsPath();
6      path.AddEllipse(50, 50, 100, 150);
7      path.AddLine(20, 20, 200, 20);
8      Pen bluePen = new Pen(Color.Blue, 2);
9      Matrix X = new Matrix();
10     X.Rotate(30);
11     X.Translate(50.0f, 0);
12     path.Transform(X);
13     g.DrawRectangle(Pens.Green, 200, 50, 100, 100);
14     g.DrawLine(Pens.Green, 30, 20, 200, 20);
15     g.DrawPath(bluePen, path);
16     bluePen.Dispose();
17     path.Dispose();
18     g.Dispose();
19 }
```

Figure 11.2 shows the output from Listing 11.2. The transformation affects only graphics path items (the ellipse and the blue [dark] line).

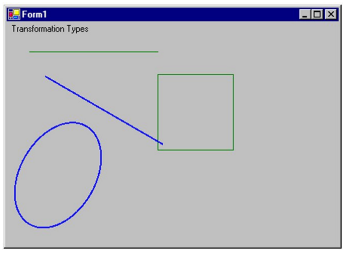


Figure 11.2 Transformasi Lokal

Praktikum 12

Transformasi Citra

Image transformation is exactly the same as any other transformation process. In this section we will see how to rotate, scale, translate, reflect, and shear images. We will create a Matrix object, set the transformation process by calling its methods, set the Matrix object as the Transform property or the transformation methods of the Graphics object, and call DrawImage.

Rotating images is similar to rotating other graphics. Listing 12.1 rotates an image. We create a Graphics object using the CreateGraphics method. Then we create a Bitmap object from a file and call the DrawImage method, which draws the image on the form. After that we create a Matrix object, call its Rotate method, rotate the image by 30 degrees, and apply the resulting matrix to the surface using the Transform property. Finally, we draw the image again using DrawImage.

Listing 12.1 Pemutaran Gambar

```
1  private void RotationMenu_Click(object sender, System.EventArgs e)
2  {
3      Graphics g = this.CreateGraphics();
4      g.Clear(this.BackColor);
5      Bitmap curBitmap = new Bitmap(@"roses.jpg");
6      g.DrawImage(curBitmap, 0, 0, 200, 200);
7      Matrix X = new Matrix();
8      X.Rotate(30);
9      g.Transform = X;
10     g.DrawImage(curBitmap, new Rectangle(205, 0, 200, 200), 0, 0,
           curBitmap.Width, curBitmap.Height, GraphicsUnit.Pixel) ;
11     curBitmap.Dispose();
12     g.Dispose();
13 }
```

Figure 12.1 shows the output from Listing 12.1. The first image is the original; the second image is rotated.

Now let's apply other transformations. Replacing the Rotate method in Listing 12.1 with the following line scales the image:

Listing 12.2 Scaling Images

```
1  X.Scale(2, 1, MatrixOrder.Append);
```

Figure 12.1 Pemutaran Gambar

Figure 12.2 Scaled Image

The scaled image is shown in Figure 12.2.

Replacing the Rotate method in Listing 12.1 with the following line translates the image with 100 offset in the x- and y-directions:

Listing 12.3 Pentranslasian Gambar

```
1 X.Translate(100, 100);
```

The new output is shown in Figure 12.3.

Figure 12.3 Pentranslasian Gambar

Replacing the Rotate method in Listing 12.1 with the following line shears the image:

Listing 12.4 Shearing Gambar

```
1 X.Shear(2, 1);
```

The new output is shown in Figure 12.4.

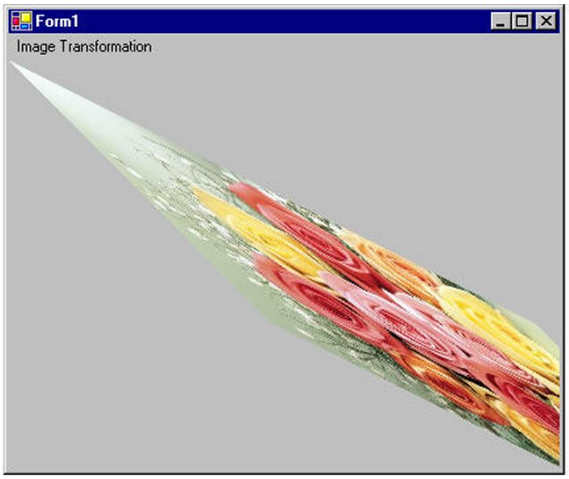


Figure 12.4 Shearing Gambar

Praktikum 13

Matriks dan Transformasi Warna

So far we've see transforming graphics shapes from one state to another but did you ever think about transforming colors? Why would you want to transform an image's colors? Suppose you want to provide gray scale effects, or need to reduce or increase the contrast, brightness, or redness of an image. That's when a color matrix is needed.

As we have discussed in earlier chapters, the color of each pixel of a GDI+ image or bitmap is represented by a 32-bit number where 8-bits are used for each of the red, green, blue, and alpha components. Each of the four components is a number from 0 to 255. For red, green, and blue components, 0 represents no intensity and 255 represents full intensity. For the alpha component, 0 represents fully transparent and 255 represents fully opaque. A color vector includes four items, i.e., (R, G, B, A). The minimum values for this vector are (0, 0, 0, 0) and the maximum values for this vector are (255, 255, 255, 255).

GDI+ allows the use of values between 0 and 1 where 0 represents the minimum intensity and 1 represents the maximum intensity. These values are used in a color matrix to represent the intensity and opacity of color components. For example, the color vector with the minimum values is $(0, 0, 0, 0)$ and the color vector with maximum values is $(1, 1, 1, 1)$.

In color transformation, we apply a color matrix on a color vector. This can be done by multiplying a $\times 4$ matrix. However a $\times 4$ matrix supports only linear transformations such as rotation, and scaling. To perform non-linear transformations such as translation, you need to use a 5×5 matrix. The element of the fifth row and the fifth column of the matrix must be 1 and all of the other entries in the five columns must be 0.

The elements of the matrix are identified using a zero-based index. Thus the first element of the matrix is $M[0][0]$ and the last element of a 5×5 matrix is $M[4][4]$. A 5×5 identity matrix is shown below. In this matrix, the elements $M[0][0]$, $M[1][1]$, $M[2][2]$ and $M[3][3]$ represent the red, blue, green, and alpha factors respectively.

The element $M[4][4]$ means nothing and it must always be 1.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Now suppose you want to double the intensity of the red component of a color: simply set $M[0][0] = 2$. For example, the matrix shown below doubles the intensity of the red component, decreases the intensity of the green component by half, triples the intensity of the blue component, and decreases the opacity of the color by half (semi-transparent).

$$\begin{bmatrix} 2 & 0 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

In the above matrix, we multiplied the intensity values. You can also add by using other matrix elements. For example, the matrix shown below will double the intensity of red component and add 0.2 to each of the red, green, and blue component intensities:

$$\begin{bmatrix} 2 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0.2 & 0.2 & 0.2 & 0 & 1 \end{bmatrix}$$

13.1 Kelas ColorMatrix

In this section, we will discuss the `ColorMatrix` class. As you can guess from its name, this class defines a matrix of colors. In the preceding sections, we discussed the Matrix class. The `ColorMatrix` class is not very different from the Matrix class. The Matrix class is used in general transformation to transform graphics shapes and images, while the `ColorMatrix` class is specifically designed to transform colors. Before we see practical use of the color transformation, we will discuss the `ColorMatrix` class, its properties and methods.

The `ColorMatrix` class constructor takes an array, which contains the values of matrix items. The Item property of this class represents a cell of the matrix and can be used to get and set cell values. Besides the Item property, the ColorMatrix class provides 25 `MatrixXY` properties, which represent items of the matrix at row (X+1) and column $(Y + 1)$. `MatrixXY` properties can be used to get and set an item's value. Listing 13.1 creates a `ColorMatrix` object with item $(4, 4)$ set to 0.5 (half opacity). After that it sets the value of item $(3, 4)$ to 0.8 and item $(1, 1)$ to 0.3.

Listing 13.1 Membuat Objek ColorMatrix

```
float[][] ptsArray ={
   new float[] {1, 0, 0, 0, 0},
   new float[] {0, 1, 0, 0, 0},
   new float[] {0, 0, 1, 0, 0},
   new float[] {0, 0, 0, 0.5f, 0},
   new float[] {0, 0, 0, 0, 1}};
ColorMatrix clrMatrix = new ColorMatrix(ptsArray);
if( clrMatrix.Matrix34 <= 0.5)
{
   clrMatrix.Matrix34 = 0.8f;
   clrMatrix.Matrix11 = 0.3f;
}
```

Now let's apply color matrices transform colors.

13.2 Operasi Matriks dalam Pengolahan Gambar

Re-coloring, the process of changing image colors, is a good example of color transformation. Re-coloring includes changing colors, intensity, contrast, and brightness of an image. It all can be done using the ImageAttribute class and its methods. The ColorMatrix can be applied on an Image using the SetColorMatrix method of the ImageAttribute class. The ImageAttribute object is used as a parameter when you call DrawImage.

13.2.1 Pentranslasian Warna

Translating colors increases or decreases color intensities by a set amount (not by multiplying them). Each color component (red, green, and blue) has 255 different intensity levels ranging from 0 to 255. For example, assume the current intensity level for the red component of a color is 100. Changing its intensity level to 150 would imply translating by 50.

In a color matrix representation, the intensity varies from 0 to 1. The last row's first four elements of a color matrix represent the translation of red, green, blue, and alpha components of a color, per Figure ???. Hence adding a value to these elements will transform a color. For example, $t1$, $t2$, $t3$, and $t4$ values in the following color matrix represent the red, green, blue and alpha component translations:

```
Color Matrix =
{1, 0, 0, 0, 0},
{0, 1, 0, 0, 0},
{0, 0, 1, 0, 0},
{0, 0, 0, 1, 0},
{t1, t2, t3, t4, 1}};
```

Listing 13.2 uses a `ColorMatrix` to translate colors. We change the current intensity of the red component to 0.90. First we create a Graphics object using the `CreateGraphics` method and create a Bitmap object from a file. After that we create an array of `ColorMatrix` elements and create a `ColorMatrix` from this array. Then we create an ImageAttributes object and set the color matrix using

SetColorMatrix, which takes the `ColorMatrix` object as its first parameter. After all that we draw two images. The first image has no effects, while the second image shows the result of our color matrix transformation. Finally, we dispose the objects.

Listing 13.2 Pentranslasian Warna menggunakan ColorMatrix

```
1  private void TranslationMenu_Click(object sender, System.EventArgs e)
2  {
3      Graphics g = this.CreateGraphics();
4      g.Clear(this.BackColor);
5      Bitmap curBitmap = new Bitmap("roses.jpg");
6      float[][] ptsArray =
7      {
8          new float[] {1, 0, 0, 0, 0},
9          new float[] {0, 1, 0, 0, 0},
10         new float[] {0, 0, 1, 0, 0},
11         new float[] {0, 0, 0, 1, 0},
12         new float[] {.90f, .0f, .0f, .0f, 1}
13     };
14     ColorMatrix clrMatrix = new ColorMatrix(ptsArray);
15     ImageAttributes imgAttribs = new ImageAttributes();
16     imgAttribs.SetColorMatrix(clrMatrix, ColorMatrixFlag.Default,
           ColorAdjustType.Default);
17     g.DrawImage(curBitmap, 0, 0, 200, 200);
18     g.DrawImage(curBitmap, new Rectangle(205, 0, 200, 200), 0, 0,
           curBitmap.Width, curBitmap.Height, GraphicsUnit.Pixel,
           imgAttribs) ;
19     curBitmap.Dispose();
20     g.Dispose();
21 }
```

Listing 13.2 generates Figure 13.1. The original image is on the left, while on the right we have the results of our color translation. If you change the values of other components (red, blue, and alpha) in the last row of the `ColorMatrix`, you'll see different results.

Figure 13.1 Pentranslasian Warna

13.2.2 Penskalaan Warna

Scaling color involves multiplying a color component value by a scaling factor. For example, the $t1$, $t2$, $t3$, and $t4$ values in the following color matrix represent the

red, green, blue and alpha components. If you change the value of $M[2][2]$ to 0.5, the transformation operation will multiply the green component by 0.5, cutting its intensity by half.

```
Color Matrix =
{t1, 0, 0, 0, 0},
{0, t2, 0, 0, 0},
{0, 0, t3, 0, 0},
{0, 0, 0, t4, 0},
{0, 0, 0, 0, 1}};
```

Listing 13.3 Penskalaan Warna

```
1  private void ScalingMenu_Click(object sender, System.EventArgs e)
2  {
3      Graphics g = this.CreateGraphics();
4      g.Clear(this.BackColor);
5      Bitmap curBitmap = new Bitmap("roses.jpg");
6      float[][] ptsArray =
7      {
8          new float[] {1, 0, 0, 0, 0},
9          new float[] {0, 0.8f, 0, 0, 0},
10         new float[] {0, 0, 0.5f, 0, 0},
11         new float[] {0, 0, 0, 0.5f, 0},
12         new float[] {0, 0, 0, 0, 1}
13     };
14     ColorMatrix clrMatrix = new ColorMatrix(ptsArray);
15     ImageAttributes imgAttribs = new ImageAttributes();
16     imgAttribs.SetColorMatrix(clrMatrix, ColorMatrixFlag.Default,
           ColorAdjustType.Default);
17     g.DrawImage(curBitmap, 0, 0, 200, 200);
18     g.DrawImage(curBitmap, new Rectangle(205, 0, 200, 200), 0, 0,
           curBitmap.Width, curBitmap.Height, GraphicsUnit.Pixel,
           imgAttribs) ;
19     curBitmap.Dispose();
20     g.Dispose();
21 }
```

Output from Listing 13.3 is shown in Figure 13.2. The original image is on the left, and on the right is the image after color scaling. If you change the values of $t1$, $t2$, $t3$, and $t4$, you will see different results.

13.2.3 Shearing Warna

Earlier in this chapter, we saw image shearing. It can be thought of as anchoring one corner of a rectangular region and stretching the opposite corner horizontally, vertically or in both directions. Shearing colors is basically the same process, but shearing color components rather than the image itself.

Color shearing increases or decreases a color component by an amount proportional to another color component. For example, consider the transformation where the red component is increased by one half the value of the blue component. Under such a transformation, the color $(0.2, 0.5, 1)$ would become $(0.7, 0.5, 1)$. The new

Figure 13.2 Penskalaan Warna menggunakan ColorMatrix

red component is $0.2 + (1/2)(1) = 0.7$. The following ColorMatrix is used to shear image colors.

```
float[][] ptsArray = {
new float[] {1, 0, 0, 0, 0},
new float[] {0, 1, 0, 0, 0},
new float[] {.50f, 0, 1, 0, 0},
new float[] {0, 0, 0, 1, 0},
new float[] {0, 0, 0, 0, 1}};
ColorMatrix clrMatrix = new ColorMatrix(ptsArray);
```

Now if you substitute this color matrix in Listing 13.3, the output will look like Figure 13.3.

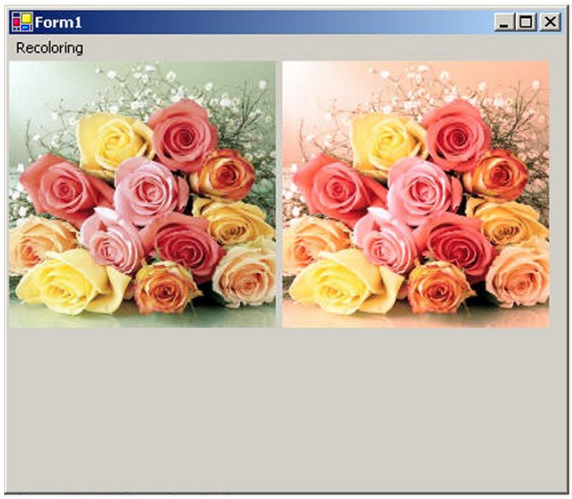


Figure 13.3 Shearing Warna

13.2.4 Perotasian Warna

Color, in GDI+ have four components: red, green, blue, and alpha. Rotating all four components in a four dimension space is hard to visualize. However it can be visualized in a three dimensional space. To do this, we drop the alpha component from the color structure and assume that there are only three colors: red, green, and

blue as shown in Figure 13.4. The three colors, red, green, and blue are perpendicular to each other, so the angle between any two primary colors is 90 degrees.

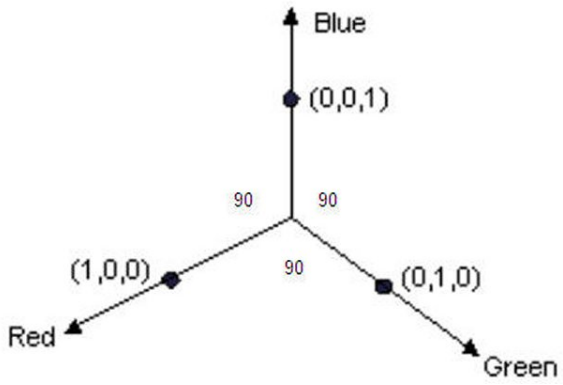


Figure 13.4 Ruang Rotasi RGB

Let's say the red, green, and blue colors are represented by points $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$ respectively. If you rotate a color with green component 1, and red and blue component 0 each by 90 degrees, the new value of the color would have red component 1, and green and blue component 0 each. If you rotate less than 90 degrees, the location would be in between green and red. Figure 13.5 shows how to initialize a color matrix to perform rotations about each of the three components: red, green, blue.

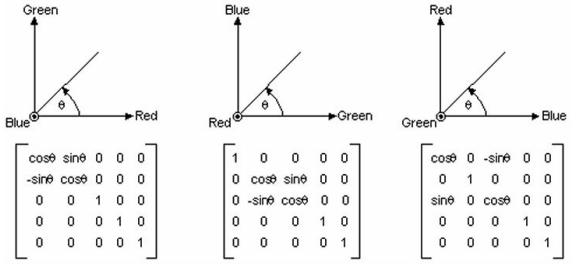


Figure 13.5 Inisialisasi RGB

Listing 13.4 rotates the colors by 45 degrees from the red component.

Listing 13.4 Rotasi Warna

```
1  private void RotationMenu_Click(object sender, System.EventArgs e)
2  {
3      float degrees = 45.0f;
4      double r = degrees*System.Math.PI/180;
5      Graphics g = this.CreateGraphics();
6      g.Clear(this.BackColor);
7      Bitmap curBitmap = new Bitmap("roses.jpg");
8      float[][] ptsArray =
9      {
10         new float[] {(float)System.Math.Cos(r), (float)System.Math.Sin(r)
               , 0, 0, 0}, new float[] {(float)-System.Math.Sin(r), (float)-
               System.Math.Cos(r), 0, 0, 0}, new float[] {.50f, 0, 1, 0, 0},
                new float[] {0, 0, 0, 1, 0}, new float[] {0, 0, 0, 0, 1} };
11     ColorMatrix clrMatrix = new ColorMatrix(ptsArray);
12     ImageAttributes imgAttribs = new ImageAttributes();
13     imgAttribs.SetColorMatrix(clrMatrix,
14     ColorMatrixFlag.Default,
15     ColorAdjustType.Default);
16     g.DrawImage(curBitmap, 0, 0, 200, 200);
17     g.DrawImage(curBitmap, new Rectangle(205, 0, 200, 200), 0, 0,
           curBitmap.Width, curBitmap.Height, GraphicsUnit.Pixel,
           imgAttribs) ;
18     curBitmap.Dispose();
19     g.Dispose();
20 }
```

Figure 13.6 slows output from Listing 13.4. On the left is the original image, and on the right is the image after color rotation.

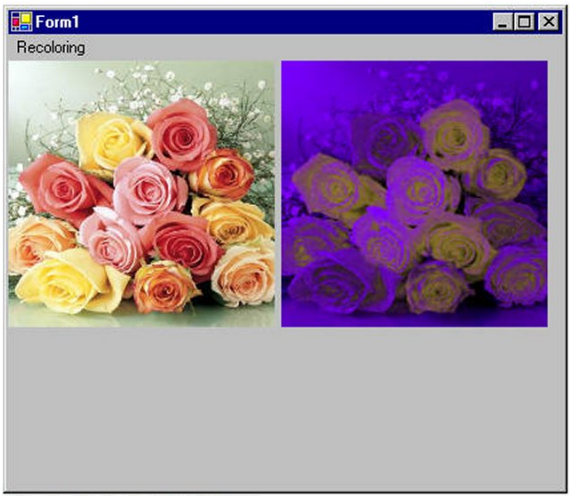


Figure 13.6 Rotasi Warna

13.3 Transformasi Teks

In previous explanation, we discussed how to use the ScaleTransform, RotateTransform, and TranslateTransform methods to transform text. We can also use a

transformation matrix to transform text.

We create a Matrix object with the transformation properties and apply it on the surface using the Transform property of the Graphics object. Listing 10-19 creates a Matrix object and sets it as the Transform property. We then call DrawString, which draws the text on the form. To test this code, add the code to a Form's paint event handler.

Listing 13.5 Contoh Transformasi Teks

```
Graphics g = e.Graphics;
string str =
    "Colors, fonts and text are common" +
    " elements of graphics programming." +
    "In this chapter, you learned " +
    " about the colors, fonts and text" +
    " representation in the "+
    ".NET framework class library. "+
    "You learned how to create "+
    "these elements and use in GDI+.";
Matrix M = new Matrix(1, 0, 0.5f, 1, 0, 0);
g.RotateTransform(45.0f, System.Drawing.Drawing2D.MatrixOrder.Prepend);
g.TranslateTransform(-20, -70);
g.Transform = M;
g.DrawString(str,
new Font("Verdana", 10),
new SolidBrush(Color.Blue),
new Rectangle(50,20,200,300) );
```

Listing 13.5 generates Figure 13.7.

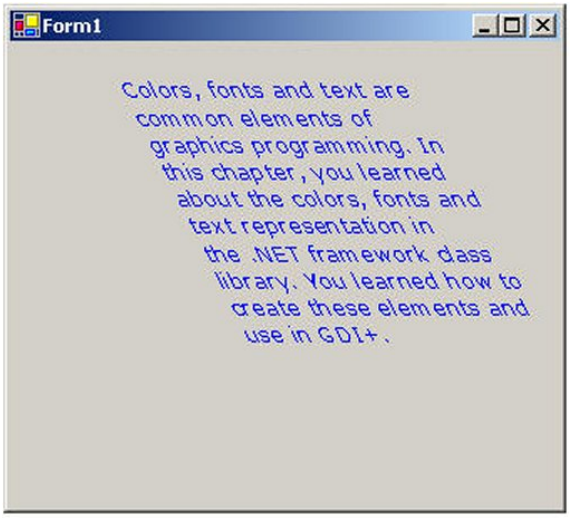


Figure 13.7 Transformasi Teks

You can apply shearing and other effects by changing the values of Matrix. For example, if you change Matrix as follows:

Listing 13.6 Shearing Teks

```
Matrix M = new Matrix(1, 0.5f, 0, 1, 0, 0);
```

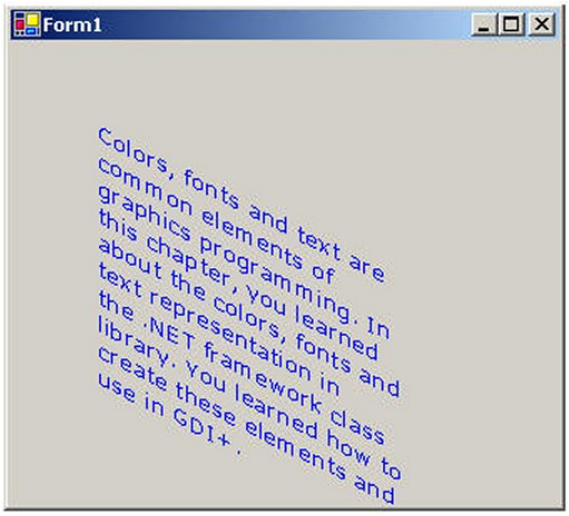


Figure 13.8 Shearing Teks

The new code will generate Figure 13.8.
We can reverse the text by just changing the value of the Matrix object as follows:

Listing 13.7 Pembalikan Teks

```
1 Matrix M = new Matrix(1, 1, 1, -1, 0, 0);
```

With results shown in Figure 13.9.

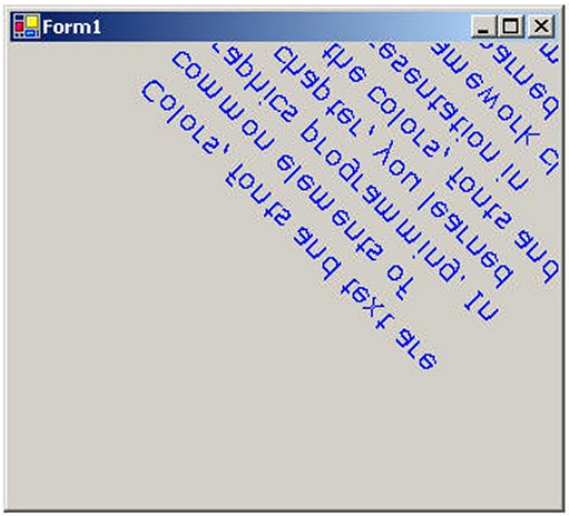


Figure 13.9 Pembalikan Teks menggunakan Matriks Transformasi

Praktikum 14

Urutan Transformasi

The Matrix object can store a single transformation or a sequence of transformations. A sequence of transformations is called a composite transformation, which is a result multiplying the matrices of the individual transformations.

In a composite transformation, the order of the individual transformations is very important. Matrix operations are not cumulative. For example, Graphics -> Rotate -> Translate -> Scale -> Graphics operation will generate a different result than the Graphics -> Scale -> Rotate -> Translate -> Graphics operation. The main reason order is significant is that transformations like rotation and scaling are done with respect to the origin of the coordinate system. Scaling an object that is centered at the origin produces a different result than scaling an object that has been moved away from the origin. Similarly, rotating an object that is centered at the origin produces a different result than rotating an object that has been moved away from the origin.

The MatrixOrder enumeration, which is an argument to the transformation methods, represents the transformation order. It has two values: Append and Prepend. The Append value indicates that the new operation will be applied after the preceding operation, while the Prepend value indicates that the new operation will be applied before the preceding operation.

Now let's write an application to see the transformation order. We create a Windows application and add a MainMenu control to the form. We add three menu items to the MainMenu. We also add areference to the System.Drawing.Drawing2D namespace because we will use the MatrixOrder enumeration, which is defined there. Listing 14.1 draws a rectangle before and after applying a scale, rotate, and translate sequence of transformations.

Listing 14.1 Skala, Rotasi, Translasi (SRT) dan Urutan Transformasinya

```
1  private void First_Click(object sender, System.EventArgs e)
2  {
3      Graphics g = this.CreateGraphics();
4      g.Clear(this.BackColor);
5      Rectangle rect = new Rectangle(20, 20, 100, 100);
6      SolidBrush brush = new SolidBrush(Color.Red);
7      g.FillRectangle(brush, rect);
8      g.ScaleTransform(1.75f, 0.5f);
9      g.RotateTransform(45.0f, MatrixOrder.Append);
10     g.TranslateTransform(150.0f, 50.0f, MatrixOrder.Append);
11     g.FillRectangle(brush, rect);
12     brush.Dispose();
13     g.Dispose();
14 }
```

Listing 14.1 generates Figure 14.1, where we see the original rectangle in the upper left, and on the lower right, we have the rectangle after composite transformation.

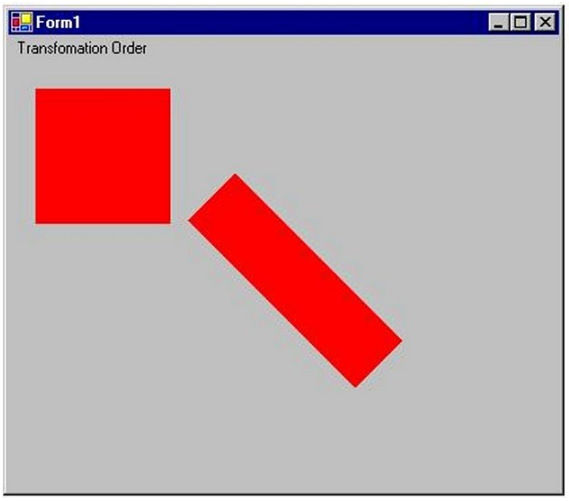


Figure 14.1 Urutan Transformasi Komposit

Now let's change the order of transformation to translate, rotate, and scale per Listing 14.1.

Listing 14.2 Translasi, Rotasi, Skala (TRS) Menggunakan Append

```
1  private void Second_Click(object sender, System.EventArgs e)
2  {
3      Graphics g = this.CreateGraphics();
4      g.Clear(this.BackColor);
5      Rectangle rect = new Rectangle(20, 20, 100, 100);
6      SolidBrush brush = new SolidBrush(Color.Red);
7      g.FillRectangle(brush, rect);
8      g.TranslateTransform(100.0f, 50.0f, MatrixOrder.Append);
9      g.ScaleTransform(1.75f, 0.5f);
10     g.RotateTransform(45.0f, MatrixOrder.Append);
11     g.FillRectangle(brush, rect);
12     brush.Dispose();
13     g.Dispose();
14 }
```

Listing 14.2 generates Figure 14.2. The original rectangle is in the save place, but the transformed rectangle has moved.

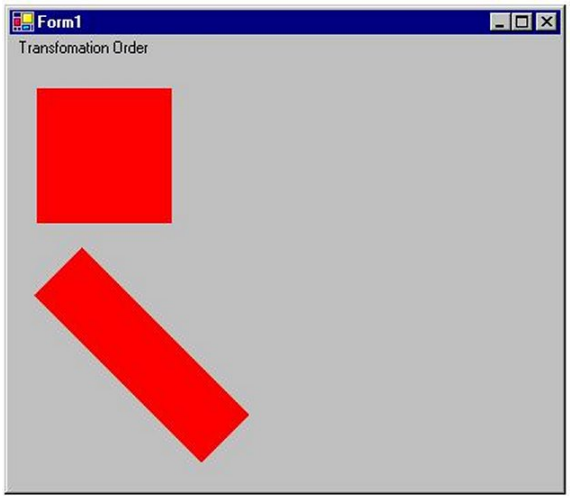


Figure 14.2 TRS Operation with Append

Let's keep the code from Listing 14.1 and change only the matrix transformation order from Append to Prepend per Listing 14.2.

Listing 14.3 TRS Transformation Order With Prepend

```
1  private void Third_Click(object sender, System.EventArgs e)
2  {
3      Graphics g = this.CreateGraphics();
4      g.Clear(this.BackColor);
5      Rectangle rect = new Rectangle(20, 20, 100, 100);
6      SolidBrush brush = new SolidBrush(Color.Red);
7      g.FillRectangle(brush, rect);
8      g.TranslateTransform(100.0f, 50.0f, MatrixOrder.Prepend);
9      g.RotateTransform(45.0f,MatrixOrder.Prepend);
10     g.ScaleTransform(1.75f, 0.5f);
11     g.FillRectangle(brush, rect);
12     brush.Dispose();
13     g.Dispose();
14 }
```

The new output is shown in Figure 14.3. As you can see, the matrix order affects the result.

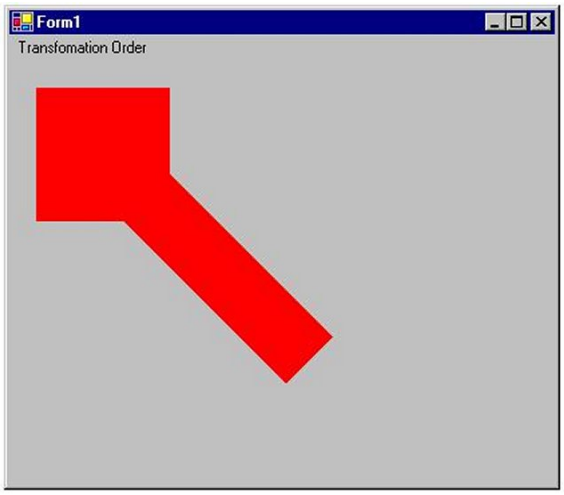


Figure 14.3 Transformasi TRS Menggunakan Prepend

Part III

Grafik dan Animasi Menggunakan Python

Part IV

Pemrograman OpenGL pada Desktop (C++)

Part V

Topik Lanjut dalam Grafika Komputer

Penutup

Pada akhir praktikum, praktikan hendaknya sudah memahami secara komprehensif dan praktis mengenai topik-topik dasar dan lanjut grafika komputer serta implementasinya dalam berbagai *environment*.

Modul praktikum ini belum lengkap dan belum sempurna. Dalam melaksanakan latihan dan tugas praktikum tidak mustahil para peserta mengalami kesulitan terutama dalam memahami apa yang diinginkan penulis dalam modul praktikum ini. Untuk itu perlu dilakukan revisi terus menerus baik dari sisi bahasa maupun struktur penulisan serta kejelasan materi yang diinginkan.

Walaupun demikian diharapkan buku ini bermanfaat dan dapat dijadikan pegangan tidak hanya bagi mahasiswa peserta kuliah Grafika Komputer di Program STudi S-1 Teknik Informatika FMIPA Universitas Padjajdaran, namun juga bagi masyarakat umum tang tertarik dan berminat memahami dan mendalami konsep dasar dalam grafika komputer. Implementasi pemrograman menggunakan bahasa pemrograman modern C#, Python, C++, serta bahasa-bahasa pemrograman lainya mudah-mudahan dapat menarik minat para mahasiswa untuk mengembangkan lebih lanjut.

Bagi yang ingin memperdalam lebih jauh lagi tentunya dapat dibaca buku-buku bacaan yang ditampilkan pada daftar bacaan modul praktikum ini. Penyempurnaan akan terus dilakukan pada waktu yang akan datang; kritik dan saran akan diterima dan dipertimbangkan dengan tangan dan hati terbuka.

Terimakasih sudah menggunakan modul praktikum ini.

setiawanhadi@unpad.ac.id

Bahan Bacaan

(1) Foley van Dam, Feiner, and Hughes, *Computer Graphics: Principles and Practice (2nd edition in C)*, Addison-Wesley publisher, 1997.

(2) David F. Rogers, Alan J. Adam, McGraw Hill, *Mathematical Elements for Computer Graphics*, 2007

(3) Donald Hearns, Pauline M. Baker, *Computer Graphics*, Prentice Hall, 2004

(4) Mahesh Chand, *Graphics Programming with GDI+*, Addison-Wesley, 2003

(5) John Vince, *Mathematics for Computer Graphics*, Second Edition, Springer, 2006

(6) Donald Hearn and M. Pauline Baker, *Computer Graphics with OpenGL, 3th edition*, Prentice Hall, 2004

(7) Fletcher Dunn and Ian Palberry, *3D Math Primer for Graphics and Game Development*, Wordware Publishing, 2002

Modul Praktikum Grafika Komputer (GDI+)

Dr. Setiawan Hadi

Edisi 1 September 2014
Edisi 1.1 September 2015
Edisi 1.2a September 2016
Edisi 2 September 2021