

**BHU-College of Informatics -Department of Computer Science
III-CS March-2020 - CoSc 3121- Computer Graphics – Handout III
[FINAL EXAM]**

TEXT AND CHARACTERS

settextstyle(): used to set the character style for the following text.

outtextxy(): used to display the characters/string in the specified location(x,y)

Method Prototype:

outtextxy(x,y, char*);

settextstyle(FONT,DIRECTION, CHAR SIZE);

FONT : 0 TO 4

DIRECTION: 0- HORIZONTAL

1- VERTICAL

CHAR SIZE: 0 TO 4

Example:

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

```
#include <stdlib.h>
```

```
#include <stdio.h>
```

```
#include <conio.h>
```

```
/* the names of the text styles supported */
```

```
/* Types of Font "DEFAULT font", "TRIPLEX font",
```

```
"SMALL font", "SANS SERIF font", "GOTHIC font"
```

```
*/
```

```
#include<iostream.h>
```

```
#include<conio.h>
```

```
#include<dos.h>
```

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

```
void main()
```

```
{
```

```
int gd= DETECT, gm;
```

```
initgraph(&gd, &gm, "c:\\turbo3\\bgi");
```

```
    settextstyle(0, HORIZ_DIR, 3);  
    outtextxy(100,100,"WELCOME TO BHU);
```

```
    settextstyle(1, VERT_DIR, 3);  
    outtextxy(100,200,"WELCOME TO BHU);
```

```
    settextstyle(2, HORIZ_DIR, 3);  
    outtextxy(100,300,"WELCOME TO BHU);
```

```
    settextstyle(3, VERT_DIR, 3);  
    outtextxy(100,400,"WELCOME TO BHU);
```

```
    settextstyle(4, HORIZ_DIR, 3);  
    outtextxy(100,450,"WELCOME TO BHU);  
    getch();
```

```
}
```

IMAGE CAPTURE AND DISPLAY:

getimage(): saves a bit image of the specified region in to memory.
putimage(): display the stored bit image on the screen.

Method Prototype:

```
getimage(x1,y1,x2,y2,buff); //void *buff;  
putimage(x,y,buff,1);
```

Example:

```
#include<iostream.h>  
#include<conio.h>  
#include<stdio.h>  
#include<graphics.h>  
#include<dos.h>  
#include<stdlib.h>  
void main()  
{
```

```

int gd=DETECT, gm=0;
initgraph(&gd, &gm, "c:\\turbo3\\BGI");
int area;
void *buff;
area=imagesize(0,0,100,100);
buff=malloc(area);

cleardevice();

circle(45,45,3);
circle(55,45,3);
circle(50,50,14);
circle(50,50,15);
ellipse(50,55,0,360,7,2);
getimage(0,0,100,100,buff);
cleardevice();
int x,y;
x=y=100;

char ch='1';
while(ch!='d')
{

cleardevice();
setcolor(random(15));
outtextxy(400,50,"III YEAR CS- STUDENTS");
putimage(x,y,buff,1);
delay(2);

ch=getch();
if(ch==72)y--;
if(ch==77)x++;
if(ch==80)y++;
if(ch==75)x--;

}
}

```

Transformation

Transformation means changing some graphics into something else by applying rules.

We can have various types of transformations such as translation, scaling up or down, rotation, etc.

When a transformation takes place on a 2D plane, it is called 2D transformation.

Transformations play an important role in computer graphics to reposition the graphics on the screen and change their size or orientation

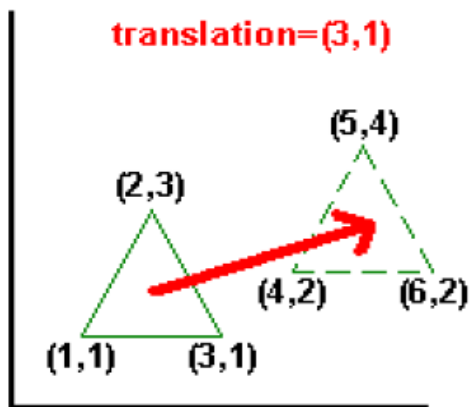


Figure - A 2-D Translation

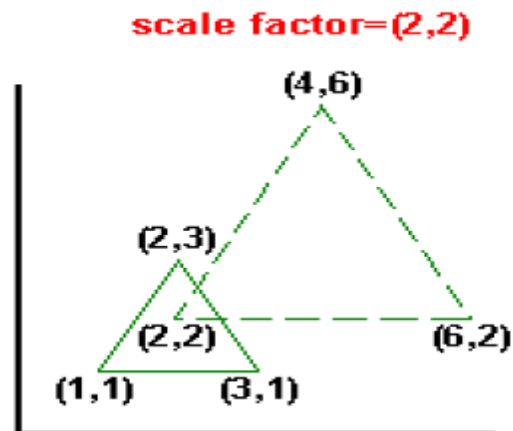


Figure - A 2-D Scaling

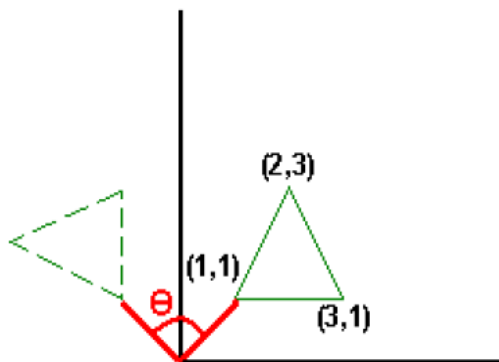


Figure - A 2-D Rotation about the Origin

2-D Translation

- The translation transformation **shifts all points by the same amount**.
- Therefore, in 2-D, we must define two translation parameters:
 - **The x-translation t_x**
 - **The y-translation t_y**
- To translate a point P to P' we add on a vector T :

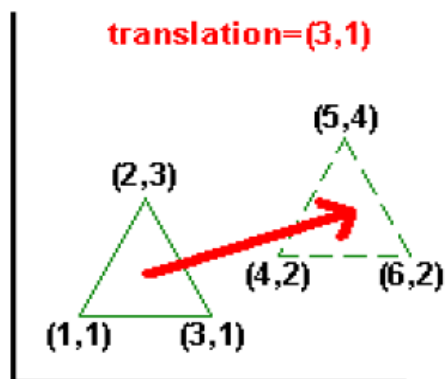
$$\begin{pmatrix} P'_x \\ P'_y \end{pmatrix} = \begin{pmatrix} P_x \\ P_y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix}$$

- Therefore, we can see that the relationship between points before and after the translation is:

$$P'_x = P_x + t_x$$

$$P'_y = P_y + t_y$$

The translation transformation **shifts all points by the same amount**.



2-D Scaling:

- The scaling transformation multiplies each coordinate of each point by a **scale factor**.
- The scale factor can be different for each coordinate (e.g. for the x and y coordinates).
 - If all **scale factors are equal** we call it **uniform scaling**,
 - whereas **if they are different** we call it **differential scaling**.
- To scale a point P by scale factors S_x and S_y we apply the scaling matrix S :

$$S = \begin{pmatrix} S_x & 0 \\ 0 & S_y \end{pmatrix} \quad \begin{pmatrix} P'_x \\ P'_y \end{pmatrix} = \begin{pmatrix} S_x & 0 \\ 0 & S_y \end{pmatrix} \begin{pmatrix} P_x \\ P_y \end{pmatrix}$$

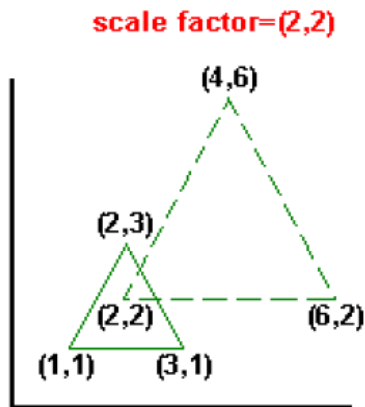
- Therefore, we can see that the relationship between points before and after the scaling is:

$$p'_x = S_x p_x$$

$$p'_y = S_y p_y$$

- **For example,**

The scaling transformation **multiplies each coordinate of each point by a scale factor**.



2-D Rotation:

- The rotation transformation rotates all points about a centre of rotation.
- The rotation transformation has a single parameter:
 - **The angle of rotation, θ .**
- To rotate a point P anti-clockwise by θ , we apply the rotation matrix R :

$$R = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix} \text{ at the } \begin{pmatrix} p'_x \\ p'_y \end{pmatrix} = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix} \begin{pmatrix} p_x \\ p_y \end{pmatrix} \text{ after}$$

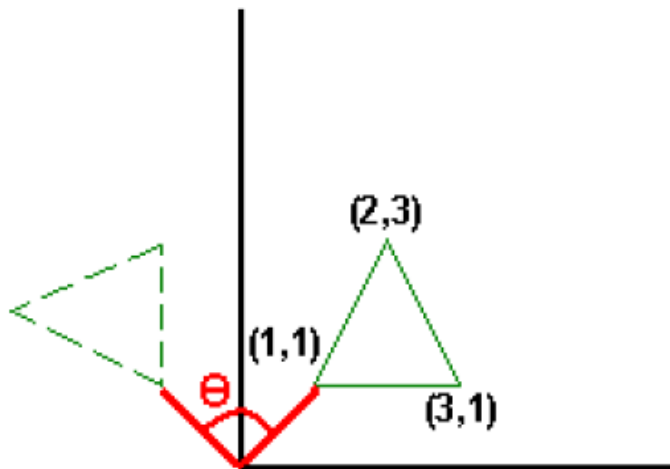
the rotation is:

$$p'_x = p_x \cos \theta - p_y \sin \theta$$

$$p'_y = p_y \cos \theta + p_x \sin \theta$$

- **For example,**

The rotation transformation **rotates all points about a centre of rotation**.



Homogeneous Coordinates

- Introduce an **extra homogeneous parameter** to standard Cartesian coordinates.
- The homogenous parameter normally has the value 1.
- With homogeneous coordinates we add an extra coordinate, the *homogenous parameter*, to each point in Cartesian coordinates.
- So 2-D points are stored as three values:
 - **The x-coordinate,**
 - **The y-coordinate and**
 - **The homogeneous parameter.**
- The relationship between homogeneous points and their corresponding Cartesian points is:

$$\text{Homogeneous point} = \begin{pmatrix} x \\ y \\ h \end{pmatrix}, \text{ Cartesian point} = \begin{pmatrix} x/h \\ y/h \\ 1 \end{pmatrix}$$

2-D Translation with Homogenous Coordinates

- Now we can express a translation transformation using a single matrix multiplication, as shown below.

$$P = \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix} \quad P' = \begin{pmatrix} p'_x \\ p'_y \\ 1 \end{pmatrix} \quad T = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} p'_x \\ p'_y \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix}$$

- Therefore 2-D Translation with Homogenous coordinates **exactly the same as before, but we used a matrix multiplication instead of an addition.**

$$p'_x = p_x + t_x, \quad p'_y = p_y + t_y,$$

Defining Transformations in OpenGL:

- To define matrix transformations in OpenGL we must introduce the concepts of ***premultiplying and postmultiplying***.
 - If we ***premultiply*** matrix ***A*** by matrix ***B*** the result is ***BA***.
 - If we ***postmultiply*** ***A*** by ***B*** the result is ***AB***.
- Whenever we apply a transformation in OpenGL it is applied to a ***current matrix***.
- **Almost all transformations in OpenGL *postmultiply* by this current matrix.**
- **OpenGL always uses a *right-handed* coordinate system.**
- OpenGL always postmultiplies by the current matrix. This means that the **sequence of transformations must be specified in the reverse order** to that which we want them to be applied.

Defining Transformations in OpenGL:

- Now let us look at OpenGL functions for defining transformations.
- In total, there are six functions that affect the current matrix.
- **The following routines can be used in OpenGL to define transformations:**
 - *glTranslate*(tx,ty,tz)*
 - *glRotate*(θ ,vx,vy,vz)*
 - *glScale*(Sx,Sy,Sz)*
 - *glLoadMatrix*(elements16)*
 - *glMultMatrix*(elements16)*
 - *glLoadIdentity(elements16)*

```

////////////////////////////////////
2-D transformation: translation, scaling ,rotation....
////////////////////////////////////
//Translation of rectangle
#include<iostream.h>
#include<conio.h>
#include<stdio.h>

```

```

#include<graphics.h>
#include<dos.h>
#include<stdlib.h>
void main()
{
int gd=DETECT,gm=0;
initgraph(&gd,&gm,"c:\\turbo3\\BGI");
int tx,ty;
cout<<"Enter the x and y";
cin>>tx>>ty;
//Original object
setcolor(3);
rectangle(100,100,200,200);

//Transformed Object
setcolor(RED);
rectangle(100+tx,100+ty,200+tx,200+ty);
getch();
}

```

Note: input in the range of 100,200..etc

//Scaling of a rectangle

```

#include<iostream.h>
#include<conio.h>
#include<stdio.h>
#include<graphics.h>
#include<dos.h>
#include<stdlib.h>
void main()
{
int gd=DETECT,gm=0;
initgraph(&gd,&gm,"c:\\turbo3\\BGI");
float sx,sy;
cout<<"Enter the x and y";
cin>>sx>>sy;
//Original object
setcolor(3);
rectangle(100,100,200,200);

```

```
//Transformed Object
setcolor(RED);
rectangle(100*sx,100*sy,200*sx,200*sy);
getch();
}
```

Note: input in the range of 0.5,1,1.5,2,etc..

//Rotation of a triangle

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
#include<graphics.h>
#include<dos.h>
#include<math.h>
float findPx(int x,inty,float ang)
{
float t=ang*3.14/180;
return 100+ (x*cos(t) - y*sin(t));
}
```

```
float findPy(int x,inty,float ang)
{
float t=ang*3.14/180;
return 100+(y*cos(t) + x*sin(t));
}
```

```
void main()
{
int gd=DETECT,gm=0;
initgraph(&gd,&gm,"c:\\turbo3\\BGI");
float ang;
int midx,midy;
midx=midy=150;
cout<<"Enter the angle";
cin>>ang;
//Original object  trianlge
setcolor(3);
```

```
line(100,100,200,200);  
line(200,200,100,200);  
line(100,100,100,200);
```

```
getch();  
//Transformed Object  
setcolor(RED);
```

```
int x1,y1,x2,y2;  
x1=findPx(100,100,ang);  
y1=findPy(100,100,ang);
```

```
x2=findPx(200,200,ang);  
y2=findPy(200,200,ang);
```

```
line(x1,y1,x2,y2);
```

```
x1=findPx(200,200,ang);  
y1=findPy(200,200,ang);
```

```
x2=findPx(100,200,ang);  
y2=findPy(100,200,ang);
```

```
line(x1,y1,x2,y2);
```

```
x1=findPx(100,100,ang);  
y1=findPy(100,100,ang);
```

```
x2=findPx(100,200,ang);  
y2=findPy(100,200,ang);
```

```
line(x1,y1,x2,y2);
```

```
getch();  
}
```

Note: input in the range of 0 to 360.

