2D Transformations

Transformation of objects aims to change the position, orientation and shape of the object. Transformation may be constant, linear or non-linear. In computer graphics packages, affine transformation (the one that combines constant and linear) is usually implemented for its simplicity. Non-linear transformation can be approximated to a set of affine transformations. The general form of affine transformation in 2D spaces is given by the following equation:

$$T\left(\begin{bmatrix}x\\y\end{bmatrix}\right)=\begin{pmatrix}a_{11} & a_{12}\\ a_{21} & a_{22}\end{pmatrix}\begin{bmatrix}x\\y\end{bmatrix}+\begin{bmatrix}b_1\\b_2\end{bmatrix}$$

The transform parameters are thus given by a matrix A=$\begin{pmatrix}a_{11} & a_{12}\\ a_{21} & a_{22}\end{pmatrix}$ and a vector b=$\begin{bmatrix}b_1\\b_2\end{bmatrix}$. It is usually preferable to have the parameters in a single matrix for easy manipulation. For this purpose, homogeneous coordinate systems are used for representation of affine transforms.

2D Homogeneous coordinate systems:

In such systems, points are represented by the triple (x, y, w) with w component added to the 2D coordinate system components (x, y). The homogeneous coordinate system is related to the standard 2D system through the relations H and S with H mapping from standard to homogeneous and S from homogeneous to standard:

$$H\colon (x,y)\rightarrow (x,y,1)$$

$$S\colon (x,y,w)\rightarrow \left(\frac{x}{w},\frac{y}{w}\right)$$

In homogenous coordinate system, the 2D affine transform can be equivalently written as:

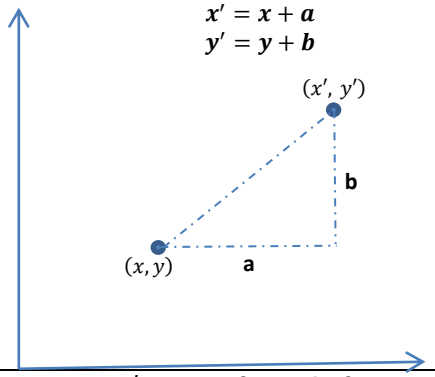
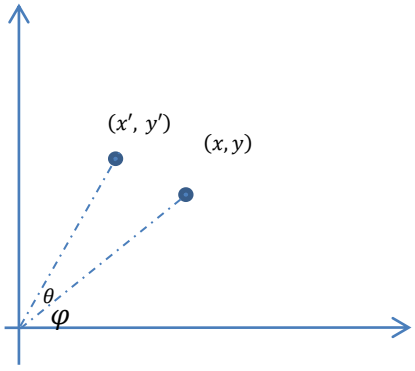
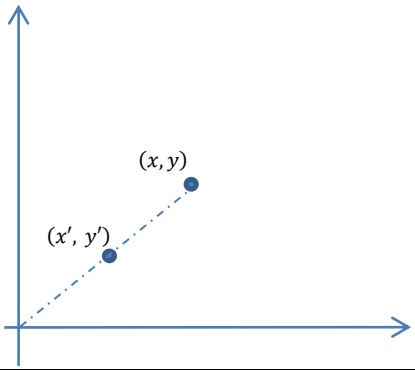
$$T\left(\begin{bmatrix}x\\y\\1\end{bmatrix}\right)=\begin{pmatrix}a_{11} & a_{12} & b_1\\ a_{21} & a_{22} & b_2\\ 0 & 0 & 1\end{pmatrix}\begin{bmatrix}x\\y\\1\end{bmatrix}$$

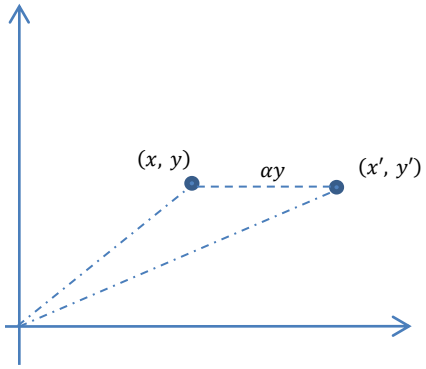
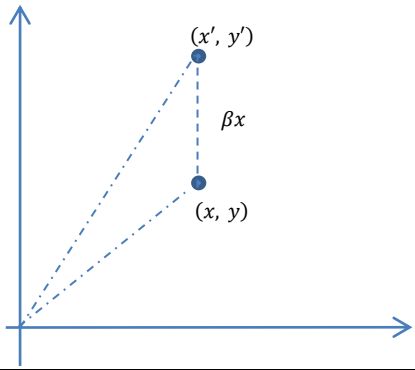
Note that the transform parameter is given by the matrix $\begin{pmatrix}a_{11} & a_{12} & b_1\\ a_{21} & a_{22} & b_2\\ 0 & 0 & 1\end{pmatrix}$ only.

Basic 2D transformations:

Many complex transformations can be easily factored to simple transformations. This leads to adopt the implementation of the basic transformations in computer graphics packages. In the following, we'll discuss the common basic transformations that include:

- Translation
- Rotation
- Scaling
- Shearing

Transformation	Equations	Matrix Form
Translation: **Translate(a, b)**	$x' = x + a$ $y' = y + b$ 	$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$
Rotation about O: **Rotate(θ)**	$x' = x\cos\theta - y\sin\theta$ $y' = x\sin\theta + y\cos\theta$  ***Proof:*** $x = R\cos\theta$ $y = R\sin\theta$ $x' = R\cos(\varphi + \theta)$ $= R\cos\varphi\cos\theta - R\sin\varphi\sin\theta$ $= x\cos\theta - y\sin\theta$ $y' = R\sin(\varphi + \theta)$ $= R\cos\varphi\sin\theta + R\sin\varphi\cos\theta$ $= x\sin\theta + y\cos\theta$	$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$
Scaling about O: **Scale(α, β)**	$x' = \alpha x$ $y' = \beta y$ 	$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} \alpha & 0 & 0 \\ 0 & \beta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$

Transformation	Equations	Matrix Form
Shearing in x-direction: X_Shear(α)	$x' = x + \alpha y$ $y' = y$ 	$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & \alpha & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$
Shearing in y-direction: Y_Shear(β)	$x' = x$ $y' = \beta x + y$ 	$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ \beta & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$

Note that the affine transformation of objects is accomplished by multiplying the transformation matrix by every point in the object in the homogeneous coordinate system.

Composite transformations:

Basic transformations can be used to synthesize composite transformations. As we'll see in the following examples, this composition is realized by matrix multiplication in the homogeneous coordinate system.

Example 1:

To rotate an object about a general center (a, b):

1. Translate the object such that (a, b) becomes the origin using the translation matrix:
$$T = \begin{bmatrix} 1 & 0 & -a \\ 0 & 1 & -b \\ 0 & 0 & 1 \end{bmatrix}$$
2. Rotate the object about the origin using the rotation matrix:
$$R = \begin{bmatrix} cos\,\theta & -sin\,\theta & 0 \\ sin\,\theta & cos\,\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$
3. Undo step 1; i.e. translate the origin to (a, b)
$$T^{-1} = \begin{bmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{bmatrix}$$

Note that: after step 1, a point $\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$ is transformed to $T\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$. After step 2, it becomes $RT\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$. Finally, it becomes $T^{-1}RT\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$. This means that object point should be multiplied by $T^{-1}RT$ which gives the composite transformation.

$$rotation(a,b,\theta) = \begin{bmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} cos\,\theta & -sin\,\theta & 0 \\ sin\,\theta & cos\,\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & -a \\ 0 & 1 & -b \\ 0 & 0 & 1 \end{bmatrix}$$

Example 2

To scale an object uniformly about its center (a, b) with a scaling factor of 0.5:

1- Translate (a, b) to O using

$$T = \begin{bmatrix} 1 & 0 & -a \\ 0 & 1 & -b \\ 0 & 0 & 1 \end{bmatrix}$$

2- Scale about O

$$S = \begin{bmatrix} 0.5 & 0 & 0 \\ 0 & 0.5 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

3- Undo step 1

$$T^{-1} = \begin{bmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{bmatrix}$$

So the composite scaling is given by:

$$T^{-1}S\,T = \begin{bmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 0.5 & 0 & 0 \\ 0 & 0.5 & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & -a \\ 0 & 1 & -b \\ 0 & 0 & 1 \end{bmatrix}$$

Implementation of 2D transformation utilities

Here we'll give an implementation for the set of utilities usually needed in 2D graphics systems. These include matrix and vector structures and operations, followed by the implementation of the basic transformations.

The following code defines the 'Vector2' and 'Matrix2' classes as represented in homogeneous coordinate systems. This is followed by the multiplication of matrix by vector utility and the multiplication of two matrices. The indexer (square bracket overload) is also defined to enable easy access to vector and matrix members.

```
class Vector2
{
        double v[3];
public:
        Vector2(){v[0]=v[1];v[2]=1;}
        Vector2(double x, double y){v[0]=x;v[1]=y;v[2]=1;}
        double& operator[](int n){return v[n];}
};
class Matrix2
{
        Vector2 A[3];
public:
        Vector2& operator[](int n){return A[n];}
```

```
        friend Matrix2 operator*(Matrix2& A,Matrix2& B)
        {
                Matrix2 C;
                for(int i=0;i<3;i++)
                        for(int j=0;j<3;j++)
                        {
                                C[i][j]=0;
                                for(int k=0;k<3;k++)C[i][j]+=A[i][k]*B[k][j];
                        }
                        return C;
        }
        friend Vector2 operator*(Matrix2& A,Vector2& v)
        {
                Vector2 r;
                for(int i=0;i<3;i++)
                {
                        r[i]=0;
                        for(int j=0;j<3;j++)r[i]+=A[i][j]*v[j];
                }
                return r;
        }
        Matrix2& operator*=(Matrix2& B)
        {
                *this=*this*B;
                return *this;
        }

};
```

The basic transformation utilities start with the identity transform (identity matrix) followed by translation, rotation, scaling and shearing transforms

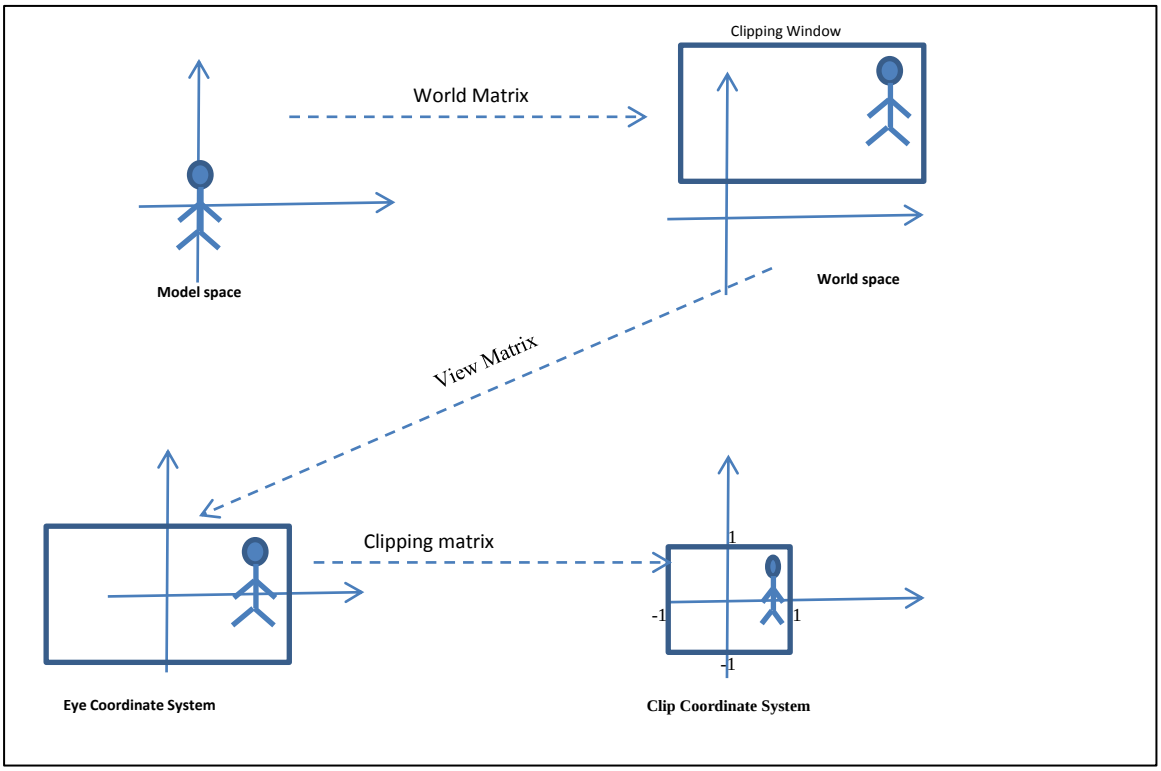
```
Matrix2 Identity()
{
        Matrix2 A;
        for(int i=0;i<3;i++)
                for(int j=0;j<3;j++)
                        A[i][j]= i==j ;
        return A;
}
Matrix2 translate(double dx, double dy)
{
        Matrix2 T=Identity();
        T[0][2]=dx;     T[1][2]=dy;
        return T;
}
Matrix2 rotate(double theta)
{
        Matrix2 R=Identity();
        R[0][0]=R[1][1]=cos(theta);
        R[1][0]=sin(theta);
        R[0][1]=-R[1][0];
        return R;
}
Matrix2 scale(double alpha, double beta)
{
        Matrix2 Sc=Identity();
        Sc[0][0]=alpha;
        Sc[1][1]=beta;
        return Sc;
}
Matrix2 shear_x(double alpha)
{
        Matrix2 Sh=Identity();
        Sh[0][1]=alpha;
        return Sh;
}
Matrix2 shear_y(double beta)
{
        Matrix2 Sh=Identity();
        Sh[1][0]=beta;
        return Sh;
}
```

2D Graphics Pipeline:

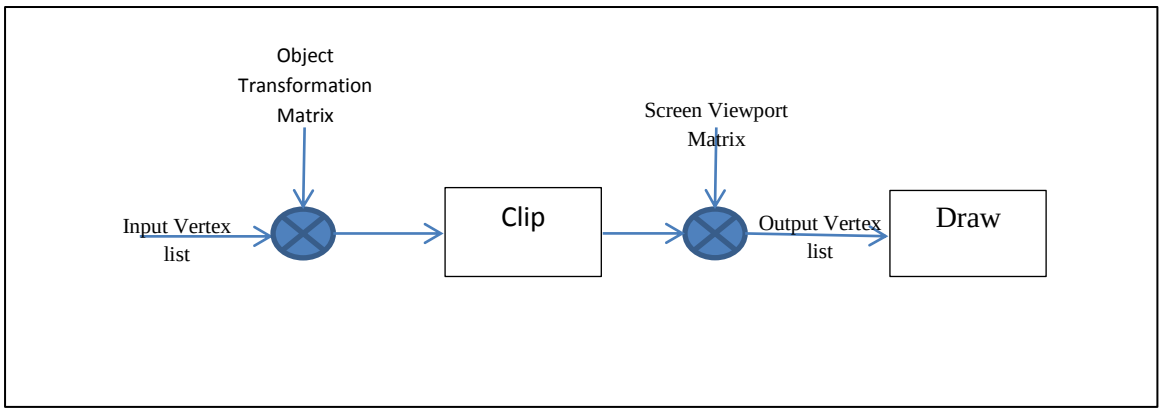
Any 2D object can be approximated by a set of primitive shapes (e.g. polygons, lines, splines, etc.). Each primitive shape contains a set of vertices. The object is thus associated with an input vertex list. Object designers usually define the coordinates of the input vertex list relative to some frame of reference called 'model space'. For easy design, the designers usually choose the origin of this frame of reference as some point on (or near to) the objet. When the object is placed in the world space in some position, its input vertex list must be transformed by some matrix called 'world matrix'. The object's world matrix could be a translation matrix that moves the origin of the object to the world position to where it would appear in. It might also contain other operations like rotation and scaling. Objects are then transformed from the world space to the view space (sometimes called camera space) where objects are represented relative to the camera (eye) coordinate system. For this purpose, the clipping window (defining the boundary and location of the camera view) is specified. The clipping window determines the objects and parts of objects that will appear on the screen. The center of the clipping window is the focus point of a camera (an eye) looking at the world and the boundaries of the window determines the clipping edges. The 'view matrix' is used to transform the center of the clipping window to the origin of the view space. The clipping window is then scaled to become a square of side length of 2 centered about the origin of the clipping space using the clipping matrix. The coordinates of the clipping window thus become (-1, -1, 1, 1). Any point having x or y coordinates greater than 1 in absolute value, is clipped out. The overall object transformation matrix contains three components:

- World matrix
- View matrix
- Clipping matrix

The following figure shows the effect of each matrix



Clipping is then applied to the resulting vertex list against the standard clipping window (-1, -1, 1, 1). The resulting vertex list is then multiplied by the viewport matrix that maps the vertex list from the standard window to a viewport on the screen. The viewport matrix is composed of scaling and translation operations. The output vertex list is used to draw the object on the screen.



To implement the above pipeline, we define the following set of classes:

Primitive Shape Class

We define an abstract class called 'Primitiveshape' which can be inherited to implement primitive shapes like polygons, triangles, etc. The 2D object class is a container class that contains a list of primitive shapes

```
class PrimitiveShape
{
protected:
        vector<Vector2> in_vert_list;
        vector<Vector2> out_vert_list;
public:
        virtual void ToClipSpace(Matrix2& object_mat)
        {
        out_vert_list.clear();
        for(int i=0;i<(int)in_vert_list.size();i++)
                        out_vert_list.push_back(object_mat*in_vert_list[i]);
        }
        virtual void Clip()=0;
        virtual void ToViewport(Matrix2& viewport_mat)
        {
        for(int i=0;i<(int)out_vert_list.size();i++)
                        out_vert_list[i]=viewport_mat*out_vert_list[i];
        }
        virtual void Draw(HDC hdc)=0;
};
```

Note that this class contains the abstract methods Clip and Draw that depend on the specific primitive shape driven from this class. The 'Transform' function is used to apply the overall object matrix to the primitive shape and the 'Map2Viewport' function is for the viewport mapping.

One can subclass the 'PrimitiveShape' class to define various primitive shapes (e.g. polygons, triangles, lines, splines and etc.). The following code shows the polygon and triangle primitive shapes:

```
Vector2 VIntersect(Vector2& v1,Vector2& v2,int x)
{
	Vector2 res;
	res[0]=x;
	res[1]=v1[1]+(x-v1[0])*(v2[1]-v1[1])/(v2[0]-v1[0]);
	return res;
}
Vector2 HIntersect(Vector2& v1,Vector2& v2,int y)
{
	Vector2 res;
	res[1]=y;
	res[0]=v1[0]+(y-v1[1])*(v2[0]-v1[0])/(v2[1]-v1[1]);
	return res;
}
```

```
class PolygonShape:public PrimitiveShape
{
private:
	void ClipLeft()
	{
		vector<Vector2> outlist;
		int n=(int)out_vert_list.size();
		Vector2 v1=out_vert_list[n-1];
		for(int i=0;i<n;i++)
		{
			Vector2 v2=out_vert_list[i];
			if(v1[0]<-1 && v2[0]>=-1)
			{
			outlist.push_back(VIntersect(v1,v2,-1)); outlist.push_back(v2);
			}
			else if(v1[0]>=-1 && v2[0]>=-1)outlist.push_back(v2);
			else if(v1[0]>=-1)outlist.push_back(VIntersect(v1,v2,-1));
			v1=v2;
		}
		out_vert_list.clear();
		out_vert_list=outlist;
	}
	void ClipBottom()
	{
		vector<Vector2> outlist;
		int n=(int)out_vert_list.size();
		Vector2 v1=out_vert_list[n-1];
		for(int i=0;i<n;i++)
		{
			Vector2 v2=out_vert_list[i];
			if(v1[1]<-1 && v2[1]>=-1)
			{
			outlist.push_back(HIntersect(v1,v2,-1)); outlist.push_back(v2);
			}
			else if(v1[1]>=-1 && v2[1]>=-1)outlist.push_back(v2);
			else if(v1[1]>=-1)outlist.push_back(HIntersect(v1,v2,-1));
			v1=v2;
		}
		out_vert_list.clear();
		out_vert_list=outlist;
	}
	void ClipRight()
	{
		vector<Vector2> outlist;
		int n=(int)out_vert_list.size();
		Vector2 v1=out_vert_list[n-1];
		for(int i=0;i<n;i++)
		{
			Vector2 v2=out_vert_list[i];
			if(v1[0]>1 && v2[0]<=1)
			{
			outlist.push_back(VIntersect(v1,v2,1));  outlist.push_back(v2);
			}
			else if(v1[0]<=1 && v2[0]<=1)outlist.push_back(v2);
			else if(v1[0]<=1)outlist.push_back(VIntersect(v1,v2,1));
			v1=v2;
		}
		out_vert_list.clear();
		out_vert_list=outlist;
	}
```

```
        void ClipTop()
        {
                vector<Vector2> outlist;
                int n=(int)out_vert_list.size();
                Vector2 v1=out_vert_list[n-1];
                for(int i=0;i<n;i++)
                {
                        Vector2 v2=out_vert_list[i];
                        if(v1[1]>1 && v2[1]<=1)
                        {
                        outlist.push_back(HIntersect(v1,v2,1));  outlist.push_back(v2);
                        }
                        else if(v1[1]<=1 && v2[1]<=1)outlist.push_back(v2);
                        else if(v1[1]<=1)outlist.push_back(HIntersect(v1,v2,1));
                        v1=v2;
                }
                out_vert_list.clear();
                out_vert_list=outlist;
        }
public:
        void AddVertex(Vector2& v)
        {
                in_vert_list.push_back(v);
        }
        void Clip()
        {
                ClipLeft();
                ClipBottom();
                ClipRight();
                ClipTop();
        }
        void Draw(HDC hdc)
        {
                int n=(int)out_vert_list.size();
                Vector2 v1=out_vert_list[n-1];
                MoveToEx(hdc,(int)v1[0],(int)v1[1],NULL);
                for(int i=0;i<n;i++)LineTo(hdc,(int)out_vert_list[i][0],(int)out_vert_list[i][1]);
        }
};
```

```
class TriangleShape:public PolygonShape
{
public:
        TriangleShape(Vector2& v1,Vector2& v2,Vector2& v3)
        {
                AddVertex(v1);
                AddVertex(v2);
                AddVertex(v3);
        }
};
```

```
class QuadShape:public PolygonShape
{
public:
        QuadShape(Vector2& v1,Vector2& v2,Vector2& v3,Vector2& v4)
        {
                AddVertex(v1);
                AddVertex(v2);
                AddVertex(v3);
                AddVertex(v4);
        }
};
```

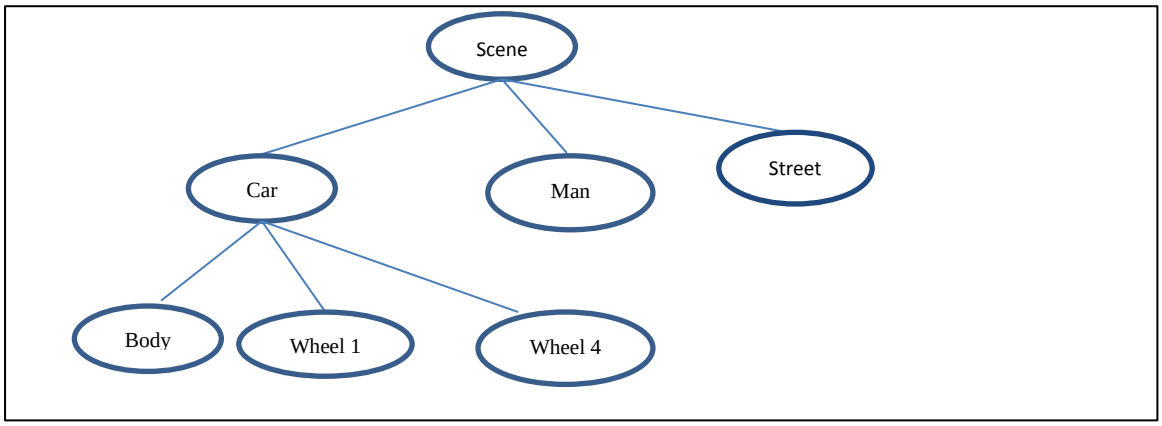
Simple 2D Objects

The simple 2D object is an object that is dealt with as one unit. The class is a container class that contains a list of primitive shapes. The class enables the construction of 2D objects through the 'AddPrimitive' function. The 'Draw' method implements the pipeline through the four calls per each primitive shape: 'ToClipSpace, 'Clip', 'ToViewport, and 'Draw'.

```
class Object2D
{
vector<PrimitiveShape *> shape_list;
public:
        void AddPrimitive(PrimitiveShape *p)
        {
                shape_list.push_back(p);
        }
        void Draw(HDC hdc,Matrix2& object_mat,Matrix2& viewport_mat)
        {
                for(int i=0;i<(int)shape_list.size();i++)
                {
                                shape_list[i]->ToClipSpace(object_mat);
                                shape_list[i]->Clip();
                                shape_list[i]->ToViewport(viewport_mat);
                                shape_list[i]->Draw(hdc);
                }
        }
};
```

Scene hierarchy classes:

For easy manipulation, a composite object is defined as a set of simple or composite objects defined relative to each other. For example, a human object is a composite object with arms, legs, and head are defined relative to the rest of the body. A scene can be viewed as a composite object with the elements of the scene being simple or composite objects defined relative to the scene. This suggests the definition of the 'scene tree' concept. A node in this tree may represent a simple or composite object relative to its parent node.



In our implementation each node contains:

- A pointer to a simple object (this pointer may be null in composite objects)
- A transformation matrix that define the node object relative to its parent node. If the node object is multiplied by this matrix, to will be transformed to the parent's coordinate system
- A list of child node pointers. This list is empty for root nodes

The flowing code shows the 'TreeNode' structure:

```
struct TreeNode
{
        Matrix2 Transform;
        Object2D *Object;
        vector<TreeNode *> Children;
        TreeNode(Matrix2 M1,Object2D *ob=NULL)
        {
                Object=ob; Transform=M1;
        }
        TreeNode *AddChild(Matrix2 M,Object2D *ob=NULL)
        {       TreeNode *res=new TreeNode(M,ob);
                Children.push_back(res);
                return res;
        }
        virtual ~TreeNode()
        {       for(int i=0;i<(int)Children.size();i++)delete Children[i];
                Children.clear();
        }
};
```

The following utility is a recursive function that draws a composite object (the scene) given its parent matrix (relative to the clipping space) and the viewport matrix. To bring the object to the clipping space, its matrix; being a relative matrix; is the multiplied by the parent matrix.

```
void Draw(HDC hdc,TreeNode *t,Matrix2 parent_mat,Matrix2& viewport_mat)
{
        if(t==NULL)return;
        Matrix2 mat=parent_mat*t->Transform;
        if(t->Object!=NULL)
                t->Object->Draw(hdc,mat,viewport_mat);
        for(int i=0;i<(int)t->Children.size();i++)Draw(hdc,t->Children[i],mat,viewport_mat);
}
```

The scene class is described in the next code:

```
class Scene
{
        TreeNode *root;
        Matrix2 viewport_mat;
        Matrix2 clip_mat;
        HDC hdc;
public:
        Scene()
        {
        InitClipView(-1,-1,1,1);
        SetViewport(0,0,800,600);
        }
        void InitClipView(double wleft,double wbottom,double wright,double wtop)
        {
        double cx=(wleft+wright)/2, cy=(wtop+wbottom)/2;
        if(root==NULL)root=new TreeNode(translate(-cx,-cy));else root->Transform=translate(-cx,-cy);
        clip_mat=scale(2/(wright-wleft),2/(wtop-wbottom));
        }
        void SetViewport(int vleft,int vtop,int vwidth,int vheight)
        {
        viewport_mat=translate(vleft+vwidth/2,vtop+vheight/2)*scale(vwidth/2,-vheight/2);
        }
        void Draw(HDC hdc)
        {
        if(root==NULL)return;
        ::Draw(hdc,root,clip_mat,viewport_mat);
        }
        TreeNode* AddNode(Matrix2 M,Object2D *obj)
        {
        root->AddChild(M,obj);
        return root->Children[(int)root->Children.size()-1];
        }
        TreeNode* GetRoot()
        {
                return root;
        }
};
```

As a demonstration to our 2D graphics package, you may try this code in the 'WndProc' function:

```
LONG WINAPI WndProc(HWND hWnd,UINT m,WPARAM wp,LPARAM lp)
{
HDC hdc;
PAINTSTRUCT ps;
static Scene s;
Object2D *stand;
Object2D *fan;
static TreeNode *stand_node,*fan_node;
static Matrix2 fan_rot=rotate(0.1);
switch(m)
{
case WM_CREATE:
        s.InitClipView(-100,-100,100,100);
        s.SetViewport(0,0,800,600);
        stand=new Object2D;
        stand->AddPrimitive(new QuadShape(Vector2(-20,5),Vector2(20,5),Vector2(20,0),Vector2(-20,0)));
        stand->AddPrimitive(new QuadShape(Vector2(-1,80),Vector2(1,80),Vector2(1,5),Vector2(-1,5)));
```

```
        stand_node=s.AddNode(translate(3,3),stand);
        fan=new Object2D;
        fan->AddPrimitive(new TriangleShape(Vector2(0,0),Vector2(15,2),Vector2(15,-2)));
        fan->AddPrimitive(new TriangleShape(Vector2(0,0),Vector2(-15,2),Vector2(-15,-2)));
        fan->AddPrimitive(new TriangleShape(Vector2(0,0),Vector2(-2,15),Vector2(2,15)));
        fan->AddPrimitive(new TriangleShape(Vector2(0,0),Vector2(-2,-15),Vector2(2,-15)));
        fan_node=stand_node->AddChild(translate(0,78),fan);
        SetTimer(hWnd,0,100,NULL);
        break;
case WM_TIMER:
        fan_node->Transform*=fan_rot;
        InvalidateRect(hWnd,NULL,TRUE);
        break;
case WM_PAINT:
        hdc=BeginPaint(hWnd,&ps);
        s.Draw(hdc);
        EndPaint(hWnd,&ps);
        break;
case WM_CLOSE:
        DestroyWindow(hWnd);
        break;
case WM_DESTROY:
        PostQuitMessage(0);
        break;
default: return DefWindowProc(hWnd,m,wp,lp);
}
return 0;
}
```

As you may note, the WM_CREATE message handler is used to setup the scene by:

- **Setting the clip window and viewport window:**

```
s.InitClipView(-100,-100,100,100);
s.SetViewport(0,0,800,600);
```

- **Creating the stand object add it to the root of the scene**

```
stand=new Object2D;
stand->AddPrimitive(new QuadShape(Vector2(-20,5),Vector2(20,5),Vector2(20,0),Vector2(-20,0)));
stand->AddPrimitive(new QuadShape(Vector2(-1,80),Vector2(1,80),Vector2(1,5),Vector2(-1,5)));
stand_node=s.AddNode(translate(3,3),stand);
```

- **Creating the fan object and add it as a child node to the stand**

```
fan=new Object2D;
fan->AddPrimitive(new TriangleShape(Vector2(0,0),Vector2(15,2),Vector2(15,-2)));
fan->AddPrimitive(new TriangleShape(Vector2(0,0),Vector2(-15,2),Vector2(-15,-2)));
fan->AddPrimitive(new TriangleShape(Vector2(0,0),Vector2(-2,15),Vector2(2,15)));
fan->AddPrimitive(new TriangleShape(Vector2(0,0),Vector2(-2,-15),Vector2(2,-15)));
fan_node=stand_node->AddChild(translate(0,78),fan);
```

- **Initializing the timer for animation:**

```
SetTimer(hWnd,0,100,NULL);
```

The WM_TIMER is used to animate the fan by multiplying its relative transform by a rotation matrix about the origin:

```
fan_node->Transform*=fan_rot;
InvalidateRect(hWnd,NULL,TRUE);
```

-