PROGRAMMERS HANDBOOK	
Author: M.Ali Caliskan	Revison : 1.0

1 Introduction

1.1 What Is PowerCAD?

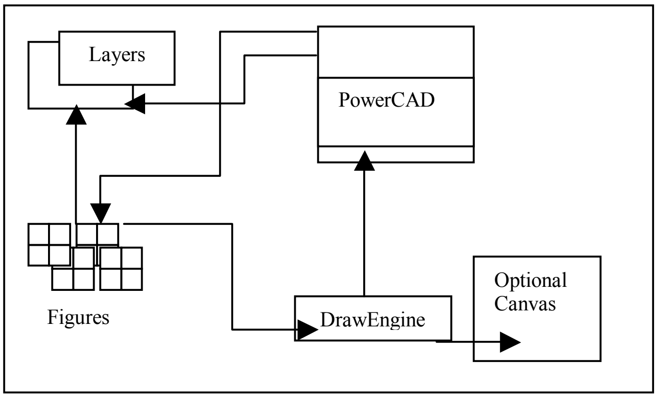
The PowerCAD is designed to provide an editor-like drawing area to handle drawings. The main idea behind the concept is the need for CAD modules in the software applications industry. Most engineering, multimedia or industrial applications need a CAD module embeded with in itself to provide the user a visual simulation. However, even to provide a satisfaction with a very basic and simple CAD environment, you still need a huge mathmetical CAD nucleus. So most applications do not include CAD modules, instead they suggest their customers to use famous (let's remember that they are very expensive) CAD tools in the market.

PowerCAD technology is developed to provide the application programmers a reuasable object library to develop CAD modules for their applications easily. With the PowerCAD components you can make a 2D CAD editor with almost no code. However if you take it more professional, by some more code you can make very flexible customization for your environment.

1.2 Installing PowerCAD

Installing PowerCAD is not different than installing any other component package. Just open the *.dpk file in Delphi and compile/install it. If you do not want to have PowerCAD components in a different package, than you should directly use pcadreg.pas file. This file is in all packages as open source and by modifying the RegisterComponent calls in the file, you can disable the installing of a specific component.

1.3 PowerCad Basics



The PowerCad technology can be used in two main ways. The first one is the user way. In this way the user do everything with mouse actions and control windows. The second way is the code way. In this way the drawings are made by making method calls to the component.

The PowerCAD component creates Layers and store the referances to them in a list. Each layer is a Tlayer class. The Layers in PowerCad are not container classes, they only provide the visibility control on the figures related with them. All drawing commands are handled as related with a Layer. For instance when you make a Line call to the PowerCAD or when you draw a line in the editor with mouse, the powercad records the referance of the proper layer in the figure data. *(The proper Layer is the Layer which of you pass the number in the call, or the active layer in the editor.)* The PowerCAD has a layer by default which has a name '*Base Layer*'. The additional layers are created by the user or the code.

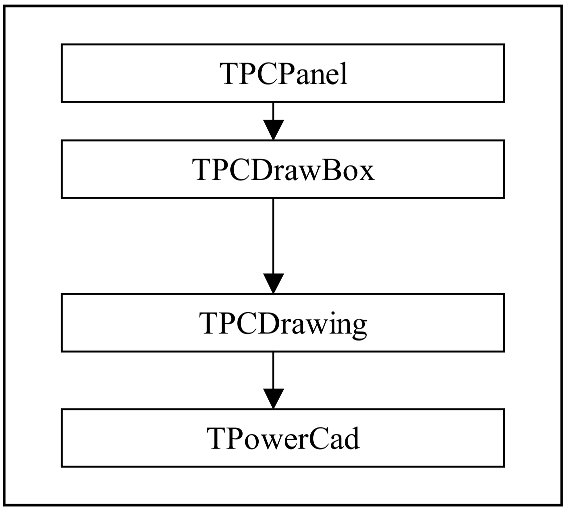
The *Base Layer* has a special behaving. The Layer Number is 0 and working in the Base Layer means being free of the restrictions of the Layer concept. You can reach any layer drawing from Base Layer. Working in the Base Layer simply means working in no layer or working in every layer.

Each drawing item in PowerCad is called *Figure*. A Figure object is a primitive of the PowerCAD tool library. A figure is the smallest entity of PowerCad. The figures are Line, Rectangle, PolyLine, Circle, Ellipse, Arc, Text, Vertex and Bitmap. The figures are stored all together in one memory database, even they are related with different layers.

The figures and the Layers together form the PowerCad Object database. This database is handled by PowerCAD and the drawing of the database is made by DrawEngine. When making redraw (screen refresh), PowerCAD uses the DrawEngine to render the object Database to its canvas.

Infact you can also perform the same way that PowerCAD does. You can create Layers without using PowerCAD and, you can make drawings through these layers. To render the created database you can directly use the DrawEngine. You should only make the DrawEngine informed about your Canvas. You can draw the shapes to any canvas by this way.

1.4 Component Hierarchy



TPCPanel
The PowerCad Panel

This class is an editor base which has measuring and zooming capabilities. It is not useful by itself always so you shouldn’t use it directly.

TPCDrawBox
The PowerCad Drawbox

This class is inherited from TPCPanel and is the first step to CAD concept. This class has a drawing page which the layout and dimensions can be arranged. Some general CAD concepts like grids, guides, snapping, coordinate system are included and implemented. The famous PowerCAD property *ToolIdx* is defined in this class. Also the major events that PowerCAD triggers are implemented in this class. It is not useful by itself always so you shouldn’t use it directly.

TPCDrawing
The PowerCAD Drawing

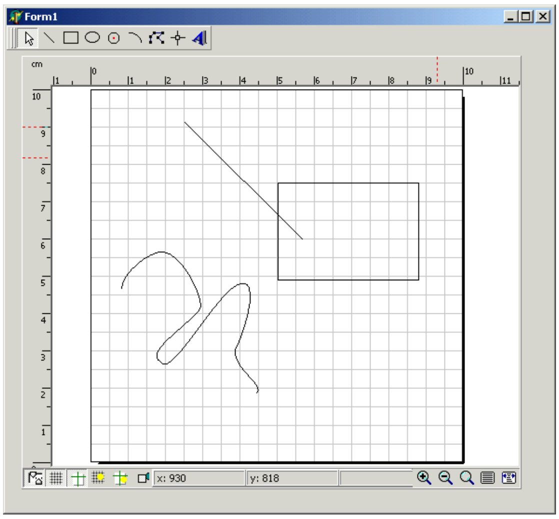
This class is inherited from TPCDrawBox. It includes all CAD calls for making drawings. The layer concept, blocks, figures, macros, plugins and all CAD related math concepts are implemented in this class. In fact this is the final step for the PowerCAD but the mouse actions in this class only causes events to be triggered. They are not interpreted as drawing actions according to the *ToolIdx*. To make drawings in this class you should always use the methods. If you want to make a custom CAD editor and so if you want to handle mouse actions by your code then use this class. It is more suitable to customize than the final class.

TPowerCad
The final PowerCad component

This class is inherited from TPCDrawing. It is the final class in the hierarchy. The only extension it has is the interpretion of the mouse actions. If the MouseActions property is set to True, the user's mouse actions is interpreted as drawing commands according to the ToolIdx in additon to the event triggering. So with this component the user can easily make CAD actions (drawing, scaling, rotating, etc.) with the mouse. This component is more usefull when you use it with the Toolbars and the Control windows(dialogs).

2 Using Components

2.1 Make a code-free CAD application.



TPowerCAD component is designed as to be used with no coding. To be able to use it, just put it on the form. Now you have got a CAD editor. But there is one case here. This editor should be invoked about the job it will handle. So we use the toolbars. We have got several toolbars but to have a simple functionality let us use noe the TPCToolBar. To use a toolbar on a form first we need a TPCDock. Put a TPCDock on the form, then a TPCToolsBars in it. Now we should set the CadControl property of the TPCToolsBar. Set it to the TPowerCad on the form.

OK! So simple, but at least it can draw anything. If you want to have more functionality than you will have to use more toolbars and also for tuning the PowerCAD editor you will need also some dialogs.

2.2 Cancel MouseActions: Take the control in your hand

The TPowerCad component has got a Boolean property called *MouseActions*. By default it is set to *True*. If you make it false the mouse actions will not be interpreted as drawing commands. In this case the mouse actions will be locked so the component will be just a Cad viewer or something else. But what is more here, you can develop a customized environment which gives developer defined reactions to the user mouse actions by implementing your own code in the mouse events.

```
        PowerCad1.MouseActions:= False;
```

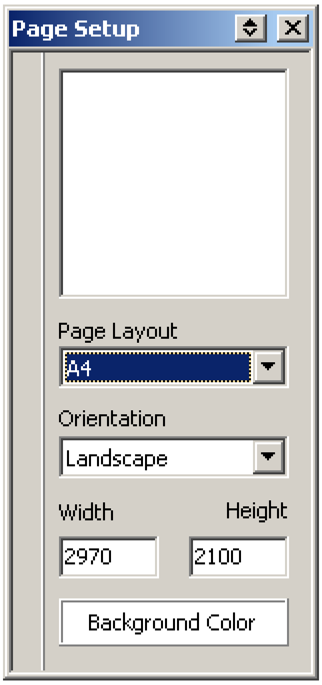
2.3 The Global PowerCad Unit : Deci Milimeter (dmm)

Most Cad Tools in the industry use milimeter as global unit. And also in these tools you can define decimal measures such as 2.4 mm. In PowerCad we didn't want to make a decimal measure unit because of the performance so first we decided to use an integer global unit. After this it was time to define the type of the unit, which means what will 1 (the smallest measure value) will express in PowerCad. If we look at the industry, we see that in technical drawings the smallest distance used is a half millimeter. So here we decided to use a global unit under millimeter. In the math books you can not see something as deci millimeter but for us it is one of the ten of a millimeter in PowerCad. So when you use 1 in PowerCad you mean 1/10 mm. For expressing a mm you will use 10, and for a cm you will use 100.

1 mm = 10 dmm

This is the measurement system in the background. On the ruler pages you will always see the mm, cm or a higher unit. Also if you want you can see the ruler as inch values. But never forget, in procedure calls or in measure value entering you will always use the PowerCad unit, **dmm**

2.4 Page Setup



Page Layout: A4

Orientation: Landscape

Width: 2970

Height: 2100

T*he page properties can be arranged from the* ***Page Setup*** *dialog assocaited with your PowerCad component.*

In PowerCad the drawing is made on an area called page. The page is the printable area of the drawing canvas and its size, layout, orientation or back color can be arranged before drawing. For page options PowerCad has a Page Setup dialog, however these arrangements are just some property settings so you can also make them manually by code.

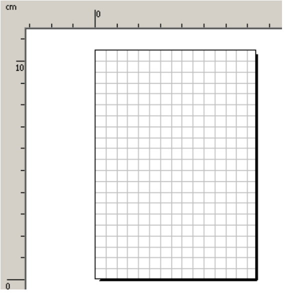
The page layout provides predefined industry paper sizes to the page. The page layouts with their sizes are as follows:

Layout	**Height(dmm)**	**Width(dmm)**
A0	11890	8410
A1	8410	5940
A2	5940	4210
A3	4210	2970
A4	2970	2100
A5	2100	1480
A6	1480	1050
B4	3530	2500
B5	2500	1760
Tabloid	4310	2790
Letter	2790	2150

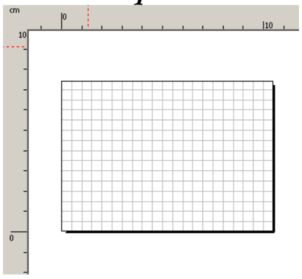
If you don't want to use any industry standart but your own page sizes then you can directly set the *WorkHeight* and *WorkWidth* property. In this case the **Page Layout** will get automatically *plCustom* value.

In addition to the page layout we should also speak about the page orientation. The page can be oriented as landscape or portrait. Above table shows the sizes when the page is portrait oriented. So the Heights are bigger than the widths. In case we orient it lanscape, the widths will be bigger than the heights.

Portrait Oriented



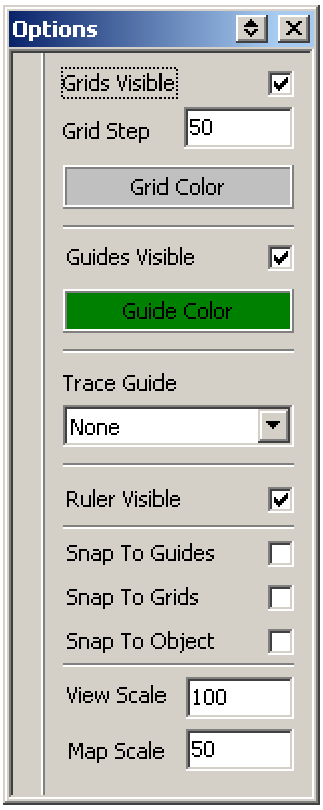
Landscape Oriented



You can arrange page options through ***Page Setup*** dialog or through setting related properties exampled as follows.

```
        PowerCad1.PageLayout := plA4
        PowerCad1. PageOrient:= poPortrait;
        // You can directly set the Page Sizes if you dont want
        // to use layout standarts.
        PowerCad1.WorkHeight := 1200;
        PowerCad1.WorkWidth := 1000;
```

2.5 General Options



Options

Grids Visible ☑

Grid Step 50

Guides Visible ☑

Trace Guide None

Ruler Visible ☑

Snap To Guides ☐

Snap To Grids ☐

Snap To Object ☐

View Scale 100

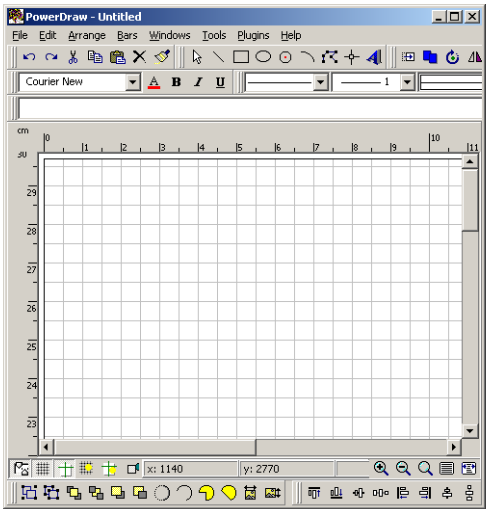
Map Scale 50

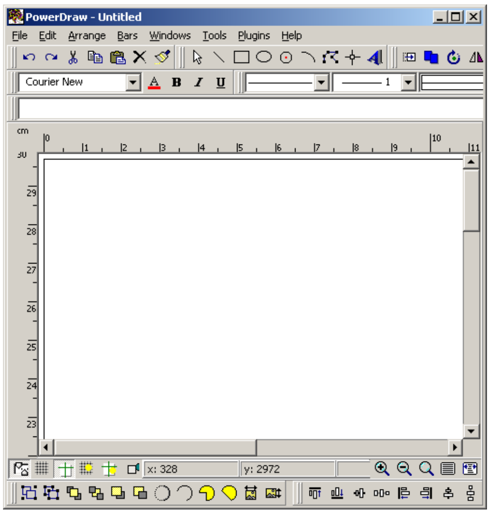
The environment in PowerCad can be tuned through the general options. There several options of PowerCad environment and most of them are accessable through Options dialog. (Others which are not in the Options Dialog will be included in the dialog) . The most basic options are grids, guides, trace guides, ruler, snapping and scaling.

Grids: In PowerCad, Grids are used to make drawings in a certain measured environment. Grids are drawn with their own color, horizontally and vertically, and provides also snapping. The grid step value; the distance between to grids is defined as dmm (1 mm = 10 dmm) by the developer or the user. The grid color and the grid step can be defined in the options dialog or from the proper properties.

```
PowerCad1.Grids := True;
PowerCad1.GridStep := 50;
PowerCad1.GridColor := clSilver;
```

In the below figure you can see the page ***with grids*** *and* ***without grids****.*





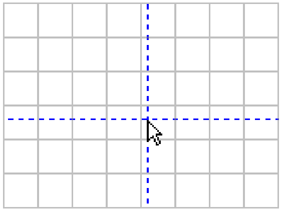
Guide Lines: In PowerCad, Guide Lines are used to aid user to organize objects in a vertical or horizontal line. Guide lines are drawn with their own color, horizontally and vertically, and provides also snapping. The guides can be added in runtime by starting a drag on the rulers and dropping it on the page. You can also drag the guide line to a different location than you have specified at the beginning. In the design time you can only set the visibility and color of the guide lines.

```
        PowerCad1.GuidesVisible:= True;
        PowerCad1.GruideColor := clGreen;
```

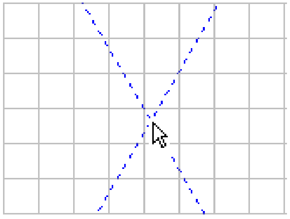
Trace Guides: In PowerCad, Trace Guides provides user some lines moving with mouse in different angles. Thus you can draw angled figures more easily. The trace guides can be in 90,60,45,30 angles. If you don't want to use the trace guides the GuideTrace property should be set to gtNone. In the below figure you can see the mouse moving with the different trace guides.

```
        PowerCad1.TraceGuides:= gtNinty;
```

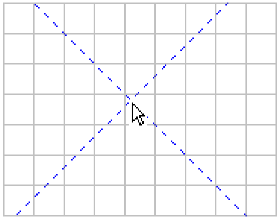
gtNinty



gtSixty



gtFortyFive



Rulers: In PowerCad, Rulers shows the measurements of the page. The x location of the cursor is highlighted on the horizontl (top) ruler and the y location of the cursor is highlighted on the vertical (left) ruler. The multiply of measure unit is arranged according to scale, so we can say that the ruler is always in autofit mode, so the tick distances is calculated automatically. The ruler sytem can be metric (mm based) or whitworth (inch based). You can arrange the visibility and unit system of the ruler through the Options dialog or setting the related properties.

```
       PowerCad1.RulerVisible:= True;
       PowerCad1.RulerSystem:= rsMetric;
```

Snapping : In PowerCad, to define object locations certainly, the mouse movements can be snapped to some references. These are the grids and the guide lines. When the mouse goes three pixel closer to the referance, it is snapped to the referance. PowerCad also provides object snapping, means the mouse is snapped to the most near point of an object when the SnapToNearPoint is true. Also in this case PowerCad fires the OnFigureSnap event to provide developer a custom snapping calculation.

```
       PowerCad1.SnapToGrid:= True;
       PowerCad1.SnapToGuides:= True;
       PowerCad1.SnapToNearPoint:= True;
```

Zooming: The view scale property used for zooming. When it is 100 the drawing is seemed in its actual size and when it is ,for instance, 50 the drawing is seemed in its half size. The bottom panel zoom buttons, ZoomOut and ZoomIn increases or decreases the zoom scale by 10. Also you can fit the drawing to the window area so that all the page can be seen. In this caase the viewscale property is automatically calculated. And last point in zooming is the Area Zooming.You can zoom the page as to fit a partial area of the drawing to the window. For zooming use the zoom buttons ( ) in the bottom panel or directly set below properties and use methods by code.

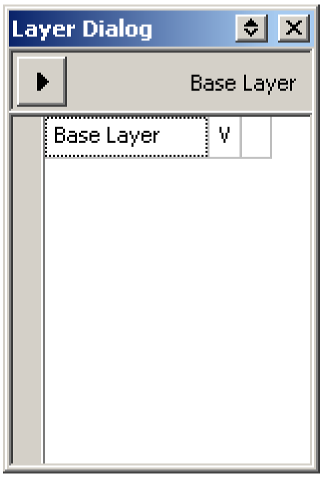
```
         PowerCad1.Scale:= 100;
         PowerCad1.FitToPage;
         PowerCad1.ZoomArea(ZoomRect);
```

Mapping : Map scale has the same functions as the scale values of the geographical maps. As declared before, in PowerCad the measurement unit is dmm. So to draw a 5 cm line you use the value 500. But this value (th 5 cm or 500 dmm) is the value on paper, so the question is to what does it correspond in real life. The map scale value gives this answer. When it equals to 100, this 5 cm long line will mean a 5 m long measurement in real life. (eg: it can be the width of a house room.) This value is used in dimension lines.

```
       PowerCad1.MapScale:= 100;
```

2.6 The "Layer" Concept in PowerCad

In PowerCad, the user works on the page. The page includes at least on Layer. To understand the Layer concept of PowerCad, (if you are not familiar with it from other tools) we can say that in fact a layer is separate part of the drawing. Think that you have a complex architectural drawing , and you draw the environmental objects (such as trees, ways) on a different paper, the building on a different paper and the electrical plan on a different paper. And let us consider that all these papers are transparent, and when you put them together, you have the complete drawing together. This separate drawings will provide you more effective maintainence of your plan. For instance if you change the electrical plan, you just throw out one paper and draw a new one. But note that each of these transparent papers has the same sizes. Now let us transfer this case to the PowerCad and call these transparent papers as "*Layer*".



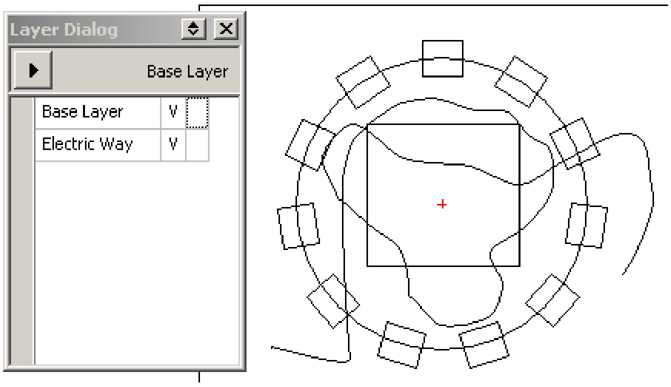
PowerCad has a default layer called “Base Layer”. You can not remove or rename the Base Layer. The base layer has got an index as 0. By default, what you draw is in this layer. If you want you can add new layers, you can delete them or you can merge them. And you can save a layer to a separate file.

In most of the method calls, you should send the layer number. The requested operation is handled on the given layer. In PowerCad editor all the operations are handled on the ActiveLayer. ActiveLayer is defined from the Layer dialog.

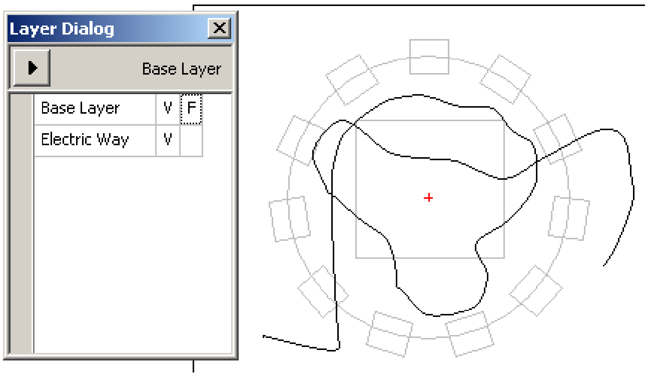
If the ActiveLayer is the Base Layer, the operations are handled through all layers. Because the Base Layer doesn’t got a limitation. If you want on a limited layer you should create your own and work on it.

When you work on a layer yu can make the others non visible or flue. A nonvisible layer will not be drawn and a flue layer will be draw as grayed. The layer dialog provides all layer operations with its grid and menu, but also you can do all layer operations through method calls and property settings.

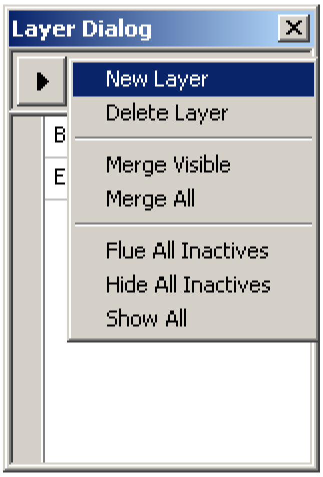
A visible layer will have a “V” letter in the second cell, and a flue layer will have a F on the third cell. For the opposite cases the letter will not be seen. To make then visible or nonvisibible, and flue or nonflue, just click on these letters in the layer grid.



On the left figure, you can see the layer dialog and the drawn objects. This is a pool drawing from birdlook. The base circle and rectangle, with small rectangles are the stone part of the pool, and the bezier curve in the drawing is the way of the electric cable.



The pool base and the electric cable way is drawn on different layers, so I can flue any of them to work easily on the other. Here I can also make invisible any of the layer.



Layer Operations:

By Clicking On the Layer Grid

Set Acive Layer: Click on the layer Row to make it ActivaLayer.

Show/Hide Layer: Click on the 2nd cell to show/hide the layer.

Flue/NonFlue Layer: Click on the 3rd cell to make the layer Flue/NonFlue.

By Clicking On the Layer Menu

New Layer: Creates a new Layer with the given name.

Delete Layer: Deletes the active layer with its contents.

Merge Visible: Merges the visible layers in the first visible layer.

Merge All: Merges all layers in the Base Layer.

Flue All InActives: Flues all layers accept the active layer.

Hide All InActives: Hides all layers accept the active layer.

Show All: Shows all layers.

You can also handle Layer operations by code.

```
PowerCad1.ActiveLayer := 0;
PowerCad1.NewLayer('MyLayer');
PowerCad1.DeleteLayer('MyLayer');
PowerCad1.DeleteLayerWithNbr(1);
PowerCad1.ShowLayer(1);
PowerCad1.HideLayer(1);
PowerCad1.FlueLayer(1);
PowerCad1.ExFlueLayer(0);
PowerCad1.ExHideLayer(0);
PowerCad1.ShowAllLayers;
PowerCad1.MergeAllLayers;
PowerCad1.MergeVisibleLayers;
LInfo := PowerCad1.GetLayerInfo(0);
```

3 Basic Tools

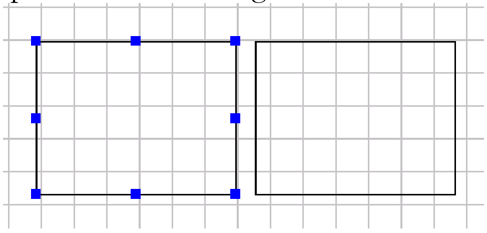
Every complex drawing is formed by basic figures. A CAD engine should provide these basic figures which will help user to make compound ones. In PowerCAD you have very basic figures all defined natively and can be created in several ways. Also you will need some operations to apply to the figures to have variations (such as rotating). PowerCad has also got these operations which are predifined and can be held in several ways. We call all these figures and operations PowerCAD basic tools. Later we will see how we can make our own customized figures or operations, but now let us see what he have got origionally and how we can use them.

You can use the tools by mouse clicks or method calls. To be able to use a tool (such as to draw a line) the ToolIdx and CurrentFigure properties should be properly set. You can set these properties to correct values by your code manually or by the ToolBars (provided in PowerCAD package) automatically. If you want to use any PowerCad toolbar (we call them "PowerBar") then you should set the CadControl property of the toolbar to the TPowerCAD component you are using on the form.

```
        Ex : ModifyBar1.CadControl := PowerCad1;
```

3.1 Select Tool

This is the most basic tool of PowerCad as other Cad engines. With the select tool you can make figures selected. A figure is drawn with selection points around when it is selected. And to select a figure means that the following operations and arrangments will be executed on this selected figure.

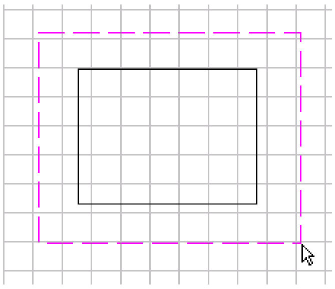


Two rectangles on the PowerCAD editor. The first one is selected and the other is not.

A figure can be selected in several ways.

You can select a figure by clicking on it or by drawing a selection rectangle around it. The ToolIdx should be set as fallows or the the *Select Button* (↖) should be down on the ToolBar associated with or TPowerCad component.

```
        PowerCad1.ToolIdx := toSelect
```



A figure is bounded with a temp slight rectangle to be selected with the select tool. You can also select a figure by just clicking on it.

You can also select a figure by using the methods SelectByPoint or SelectWithArea. The SelectByPoint method slects a figure if the sent point is on it. The SelectWithArea method selects one or more figures if they are in the sent rectangle.

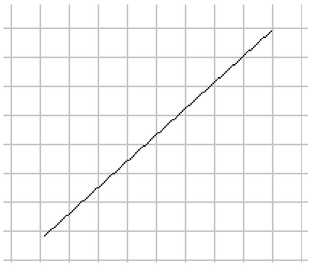
```
        PowerCad1.SelectByPoint(...);
        PowerCad1.SelectWithArea(...);
```

You can also directly set the selected property of the figure if you have got the handle of it.

```
        MyLine.Selected := True;
```

3.2 Line Tool

With the Line Tool you can create Line. A line has got two points that should be defined by the user (or the developer).

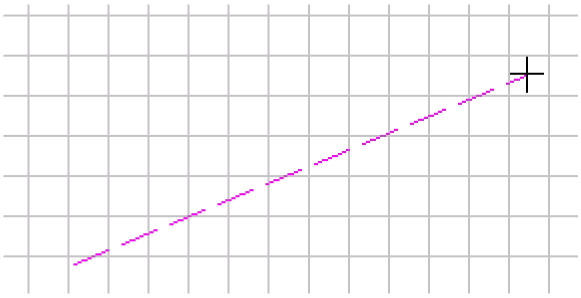


This is a line on the PowerCad editor.

A line can be created in following ways.

You can create a line by clicking on two different points on the editor. The ToolIdx should be set as fallows or the the *Line Button* (╲)should be down on the ToolBar associated with or TPowerCad component.

```
        PowerCad1.ToolIdx := toFigure
        PowerCad1.CurrentFigure := 'Tline';
```



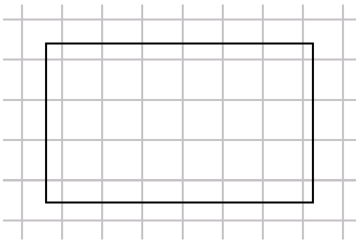
Creating a line on PowerCad Editor with mouse clicks.

You can also create a line by using the method *Line*. The *Line* method creates a line and returns its handle. In the later chapters we will se how we can use these returned handles.

```
        PowerCad1.Line(...);
```

3.3 Rectangle Tool

With the Rectangle Tool you can create Rectangle. A Rectangle has got two points that should be defined by the user (or the developer).

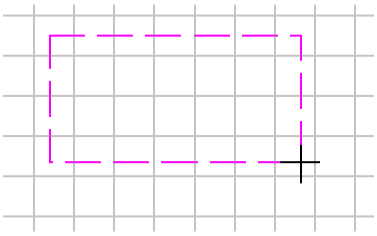


This is a Rectangle on the PowerCad editor.

A Rectangle can be created in following ways.

You can create a Rectangle by clicking on two different points on the editor. The ToolIdx should be set as fallows or the the *Rectangle Button* (□) should be down on the ToolBar associated with or TPowerCad component.

```
      PowerCad1.ToolIdx := toFigure
      PowerCad1.CurrentFigure := ‘TRectangle’;
```



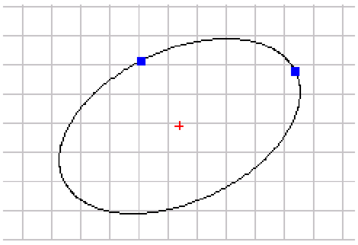
Creating a rectangle on PowerCad Editor with mouse clicks.

You can also create a Rectangle by using the method Rectangle. The Rectangle method creates a Rectangle and returns its handle. In the later chapters we will se how we can use these returned handles.

```
      PowerCad1.Rectangle(...);
```

3.4 Ellipse Tool

With the Ellipse Tool you can create Ellipse. An Ellipse has got three points that should be defined by the user (or the developer). The first one is the center and the other two are the control points that define the angle and radiuses of the ellipse.

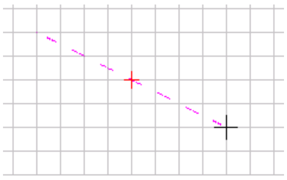


This is an Ellipse on the PowerCad editor.It is selected and the red cross is the center, the blue points are the control points.

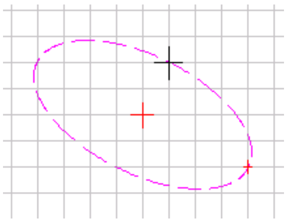
An ellipse can be created in following ways.

You can create an Ellipse by clicking on three different points on the editor. The ToolIdx should be set as fallows or the the *Ellipse Button* (⬭) should be down on the ToolBar associated with or TPowerCad component.

```
        PowerCad1.ToolIdx := toFigure;
        PowerCad1.CurrentFigure := 'TEllipse';
```



An Ellipse on the PowerCad editor is created with three mouse clicks. On the left figure, the user had clicked for the center point and now will click for the first control point for determining the angle of the ellipse and its first radius.



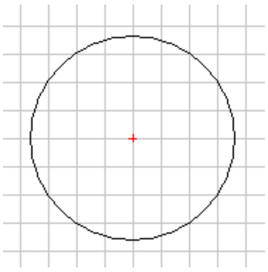
On the left figure, the center and the first control point has been selected. Now second control point will be selected for determining its second radius.

You can also create an Ellipse by using the method Ellipse or Ellipse3p. The Ellipse method creates an ellipse with the center point and radiuses provided in the parameters and returns its handle. The Ellipse3p method creates an ellipse with the center point and control points provided in the parameters and returns its handle. In the later chapters we will se how we can use these returned handles.

```
        PowerCad1.Ellipse(...);
        PowerCad1.Ellipse3p(...);
```

3.5 Circle Tool

With the Circle Tool you can create Circle. A Circle has got two points that should be defined by the user (or the developer). The first one is the center and the second one is any perimeter point that defines the radius.

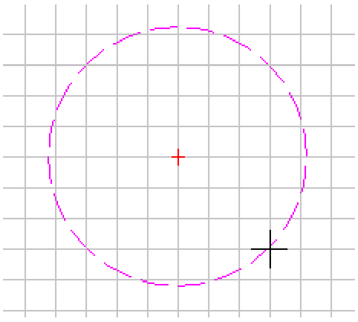


This is a Circle on the PowerCad editor. The red cross is the center.

A circle can be created in following ways.

You can create a circle by clicking on two different points on the editor. The ToolIdx should be set as fallows or the the *Circle Button* (⊙) should be down on the ToolBar associated with the TPowerCad component.

```
        PowerCad1.ToolIdx := toFigure;
        PowerCad1.CurrentFigure := 'TCircle';
```



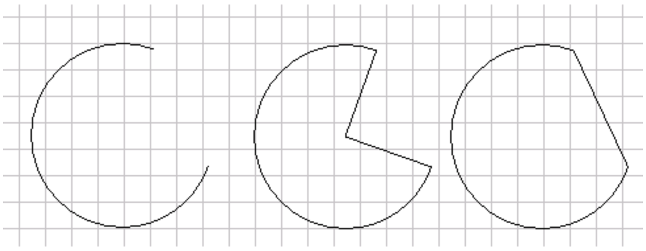
A Circle on the PowerCad editor is created with two mouse clicks. On the left figure, the user had clicked for the center point and now will click for a second point which will determine the radius of the circle.

You can also create a circle by using the method Circle. The Circle method creates an ellipse with the center point and radius provided in the parameters and returns its handle. In the later chapters we will se how we can use these returned handles.

```
      PowerCad1.Circle(...);
```

3.6 Arc Tool

With the Arc Tool you can create Arc. An Arc has got three points that should be defined by the user (or the developer). The first one is the center and the other two defines both the radius and start-end angle of the arc . An arc is also can be closed in two styles. You can close it by making a pie or with a chord connecting the ends and you can invert an arc to draw it in an opposite direction.

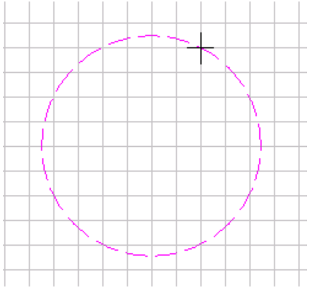


Three arcs closed in different styles. The fist one is open, second is closed as a pie, and the last one is closed with a chord.

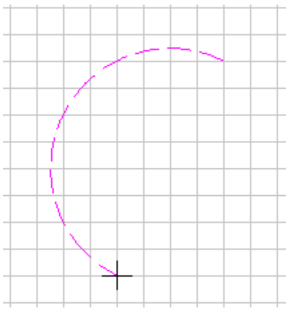
An arc can be created in following ways.

You can create an arc by clicking on three different points on the editor. The ToolIdx should be set as fallows or the the *Arc Button* (◜) should be down on the ToolBar associated with or TPowerCad component. To arrange the close style and to invert the direction of your arc you should use an TArrangeBar associated with your PowerCad control.

```
      PowerCad1.ToolIdx := toFigure;
      PowerCad1.CurrentFigure := 'TArc';
```



An Arc on the PowerCad editor is created with three mouse clicks. On the left figure, the user had clicked for the center point and now will click for the start point, This second click will also determine the radius of the arc.



On the left figure, the user had clicked for the center point and the start point.Now will click for the end point for determining the angle of the arc.Note that an arc in PowerCad is built couter-clockwise.

You can also create an arc by using the method Arc. The Arc method creates an arc with the center point,radius and start-end points provided in the parameters and returns its handle. In the later chapters we will see how we can use these returned handles. The close style of a selected arc can be arranged with the method *ArrangeArcStyleOfSelection* method. To invert a selected arc you should use the *InvertArcsOfSelection* method.

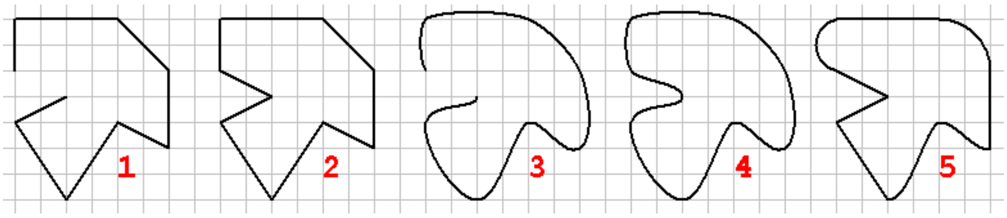
```
        PowerCad1.Arc(...);
        PowerCad1.ArrangeArcStyleOfSelection (...);
        PowerCad1.InvertArcsOfSelection (...);
```

If you have got the handle of an arc you make the arrangements directly on the arc object.

```
        MyArc.ArrangeStyle(...);
        MyArc.Invert(...);
```

3.7 Polyline Tool – (also for Polygon and Bezier)

With the Polyline Tool you can create polylines or closed polylines (polygons). The Line between two points of the polyline can be either a straight line , a curved line (bezier) or an arc in any angle.

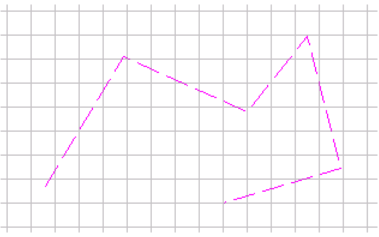


*In this figure all things you see are created with Polyline Tool. **1**. An open polyline **2**. A closed polyline (polygon) **3**. An open polyline that all segments are arranged as curve **4**. A closed polyline that all segments are arranged as curve. **5**. A closde polyline that the 1st and 3rd segments are arranged as arc (190 ,90) ,5th and 6th segments are arranged as curve.*

A polyline can be created in following ways.

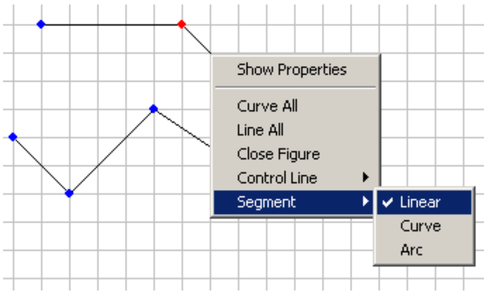
You can create a polyline by clicking on at least two (or more) different points on the editor. The ToolIdx should be set as fallows or the the *Polyline Button* ( ) should be down on the ToolBar associated with or TPowerCad component.

```
        PowerCad1.ToolIdx := toFigure;
        PowerCad1.CurrentFigure := ‘TPolyline’;
```

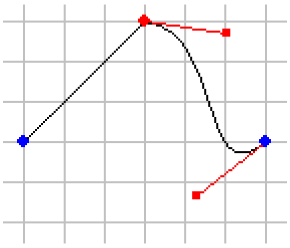


A Polyline on the PowerCad editor is created with unlimited mouse clicks.To finish the building you should click on the left mouse button..

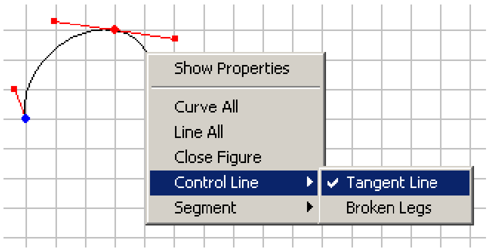
In PowerCad each click will be interpreted as a new point and to stop the building you should click on the left mouse button. Between two polyline point (knot) is a segment. A segment of a polyline can be a straight line, a curve or an arc. This can be arranged from the right-click menu. When you click on a segtment of a polyline, the clciked segment is the selected segment of the polyline, so the starting knot of the segment is the selected knot of the polyline. If you give any segment or knot command from the right-click menu, the command is applied to the selected segment/knot. If a segment is arranged as curve segment then it behaves lieke a bezier spline and as it is known; a bezier between two end knots requires 2 control points to arrange the curve. When a curve segment is selected the control points of the curve are also drawn so that you can arrange it by dragging the control points. An arc segment is an arc representation between two polyline knots. The degree of the arc is controlled by the arc center that is represented by a single control point. When the arc segment is the selected segment, this control point in the arc center is also drawn so that you can arrange the degree (circle size) of the arc segment by dragging the control point.



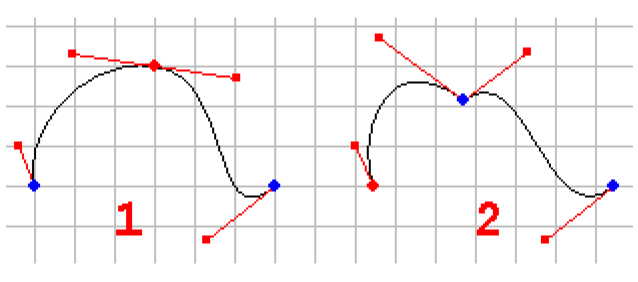
The right-click menu of a polyline provides the arrangement commands; the segment submenu includes three check commands to make the segment Linear, Curve or Arc. The 'Curve All' and 'Line All' commands make all the segments curve or line. The 'Close Figure' command draws the last segment between the last and the first knot.



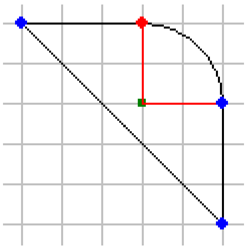
Controlling Curve Segments: A Curve segment of a polyline provides two control points when it is selected. When you drag these control points the curve is reshaped according to B-Spline geometry. If two consecutive segments are both curve segments then the control points arround the middle point (the second of the first segment and the first of the second segment) form a special behaviour together. Normally two consecutive bezier curves are handled together in CAD, so that to save the continuity of the curve the middle control points always form a tangent line. This can be arranged from the 'Control Line' submenu of the right-click menu. If the 'Tangent Line' command is checked the control points will form a tangent line together and when you move onbe of them the other will be also



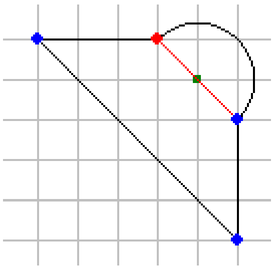
moved to save the tangent line. If the 'Broken Legs' command is checked then the middle control poinst will move independant from each other.

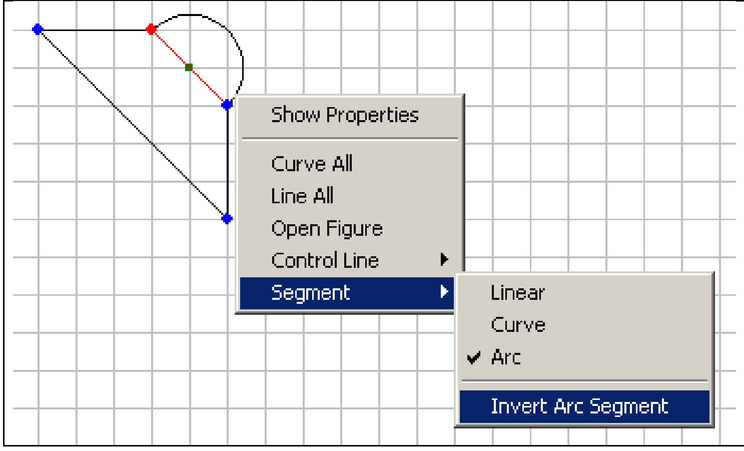


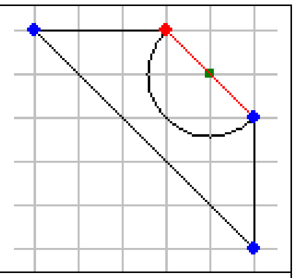
1. The Control Line of the middle knot is arraned as 'Tangent Line'. The control points move together always forming a tangent line.
2. The Control Line is arranged as 'Broken Legs'. The control points are independent from each other.



Controlling Arc Segments: When a polyline segment is arranged as arc segment, the radius of the arc is calculated according to the angle formed between the previous and next segment. In the left figure, the pre and next segments form a 90 angle so that the arc angle is arranged a 90 . The angle of the arc can be rearranged by its center. A single control point is provided as a green rectangle in arc center to control the radius so the angle of the arc. This control point can be moved on the mid-angular axis of the red lines connecting knots and the center. On the left figure the control point is moved to the center of the knots so that the arc angle is 180 . When the polyline segment is an arc, in the Segment submenu of the right-click menu a fourth command is provided; 'Invert Arc Segment'. When you click on this segment the arc segment is inverted, so that a convex to concave or the oppsote conversion is handled.







The concave arc segment is converted to convex arc from the 'Invert Arc Segment' command.

Using Polyline methods: You can create a polyline by using the method PolyLine. Also to be backwards compatible the method Polygon still exists. The Polyline method creates a polyline with points provided in the parameters and returns its handle. In the later chapters we will see how we can use these returned handles. By defining the *closed* parameter, you can make it either a polyline or a polygon. To make a polyline completely Bezier, you can use the *ConvertToBezier* method and to make any line segment bezier, you should use the *ArrangeSegment* method. And in case you want to make a polyline just a polyline with no bezier points you should use *ConvertToPolyline* method. Also if you want to set or get the control points of a Segment use the *GetControlPointsOfSegment* and *SetControlPointsOfSegment* methods of Polyline object. The control points can be handled also through the polyline knot with the method *GetControlPointsofKnot and SetControlPointsOfKnot.* The control line of a knot can be arranged with *TangentControlLine* or *BreakControlLine* methods. You can invert an arc segment with the *InvertSegment* method. You can get the type of a Segment by TypeOfSegment method. You van learn if a control line is tangent or not by the *IsKnotTangent* method.

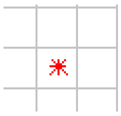
```
       PowerCad1.Polyline(...);
       PowerCad1.Polygon(...);
       // Polygon method is just for backwards compatibility.
       // Normally use Polyline method with the
       // closed parameter set as True to make a polygon.
       PowerCad1.ConvertToBezier(...);
       PowerCad1.ConvertToPolyline(...);
```

If you have got the handle of the polyline then you can use native calls as fallows.

```
       MyPolyLine.ConvertToBezier(...);
       MyPolyLine.ConvertToPolyLine(...);
       MyPolyLine.ArrangeSegment(...);
       MyPolyLine.ArrangeSelectedSegment(...);
       MyPolyLine.GetControlPointsOfSegment(...);
       MyPolyLine.SetControlPointsOfSegment (...);
       MyPolyLine.GetControlPointsOfKnot(...);
       MyPolyLine.SetControlPointsOfKnot (...);
       MyPolyLine.TangentConrolLine(...);
       MyPolyLine.BreakControlLine(...);
       MyPolyLine.TypeOfSegment(...);
       MyPolyLine.IsKnotTangent(...);
       MyPolyLine.Closed := True;
```

3.8 Point (Vertex) Tool

With the Point Tool you can create reference points.



This is a point on the PowerCad editor.

A polyline can be created in following ways.

You can create a point by clicking on one location on the editor. The ToolIdx should be set as fallows or the the *Point Button* (⌖) should be down on the ToolBar associated with or TPowerCad component.

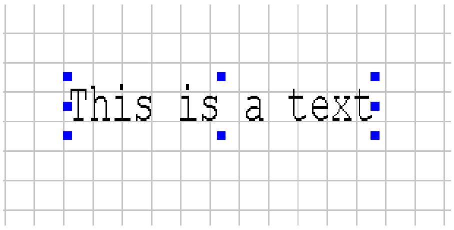
```
       PowerCad1.ToolIdx := toFigure;
       PowerCad1.CurrentFigure := ‘TVertex’;
```

You can also create a point by using the method Vertex. The Vertex method creates a point and returns its handle. In the later chapters we will se how we can use these returned handles.

```
        PowerCad1.Vertex(...);
```

3.9 Text Tool

With the Text Tool you can create Text in different fonts. A text has got a rectangular draw area and a text that should be defined by the user (or the developer).



This is a text

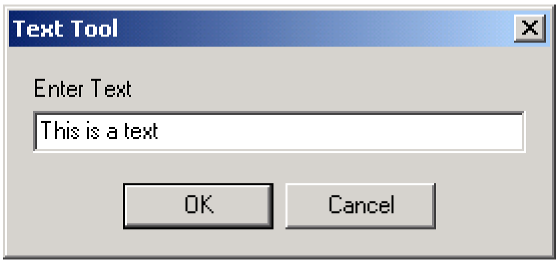
This is a text on the PowerCad editor.

A text can be created in following ways.

You can create a text by clicking on one location on the editor. The ToolIdx should be set as fallows or the the *Text Button* (A) should be down on the ToolBar associated with or TPowerCad component.

```
        PowerCad1.ToolIdx := toFigure
        PowerCad1.CurrentFigure := 'TText';
```

When you click on the editor with text tool a dialog will come up and ask you for the text.



Text Tool

Enter Text

This is a text

Creating a text on PowerCad Editor through the dialog. This dialog also comes up when you double click on the created text to edit it.

You can also create a text by using the method TextOut. The Text method creates a text and returns its handle. In the later chapters we will se how we can use these returned handles.

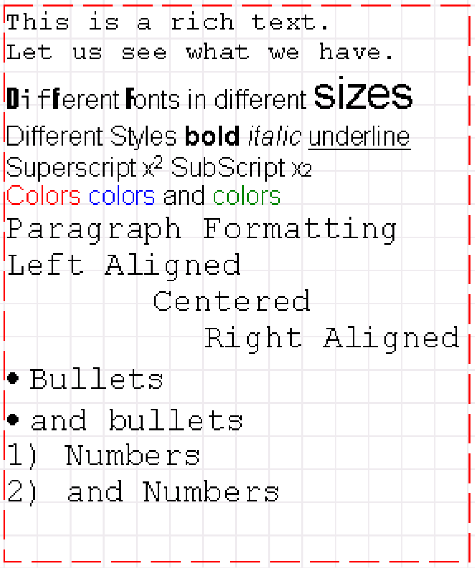
```
        PowerCad1.TextOut(...);
```

The Text Objects are dimensioned with the metric sytem. This means that you will not use typical windows font dimensioning with pixels. In spite of this, in PowerCad the height text object is defined and the width of one char is calculated with a special ratio defined by the user. These properties are all can be defined through dialogs or method parameters.

```
        Ex:   The height of the text is 100 dmm ( = 1 cm)
              The HWRatio is 0,6
              So the width of one character is 60 dmm ( = 6 mm)
```

3.10 RichText Tool

With the RichText Tool you can create Windows Rich Text . A rich text has got a rectangular draw area and a text that is in various font and styles.



This is a rich text.
Let us see what we have.
Different Fonts in different sizes
Different Styles **bold** *italic* underline
Superscript x^2 SubScript x_2
Colors colors and colors
Paragraph Formatting
Left Aligned
Centered
Right Aligned
- Bullets
- and bullets

1) Numbers
2) and Numbers

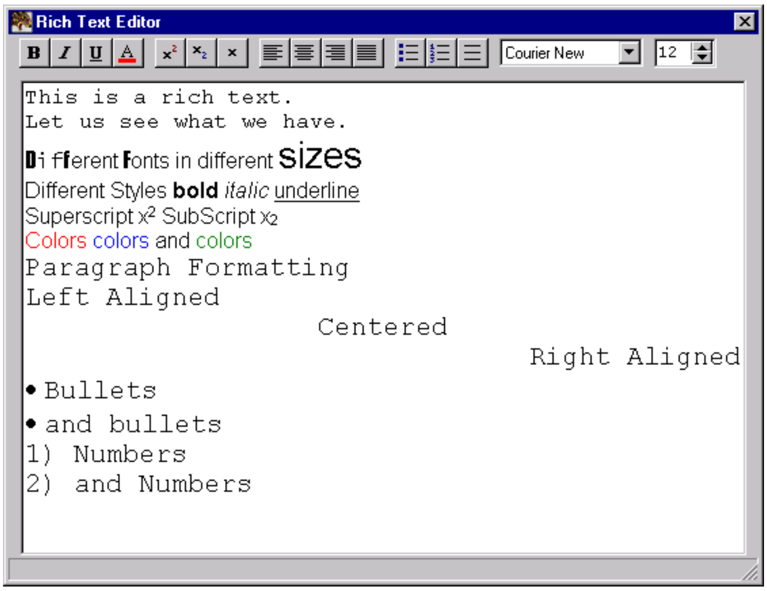
This is a rich text on the PowerCad editor.

A rich text can be created in following ways.

You can create a rich text by clicking on two different location on the editor. This locations will define the rectangukar area of the rich text. The ToolIdx should be set as fallows or the the *RichText Button* ( ) should be down on the ToolBar associated with TPowerCad component.

```
        PowerCad1.ToolIdx := toFigure
        PowerCad1.CurrentFigure := 'TRichText';
```

To edit the content of the rich text, you should double click on it to open the rich text editor.



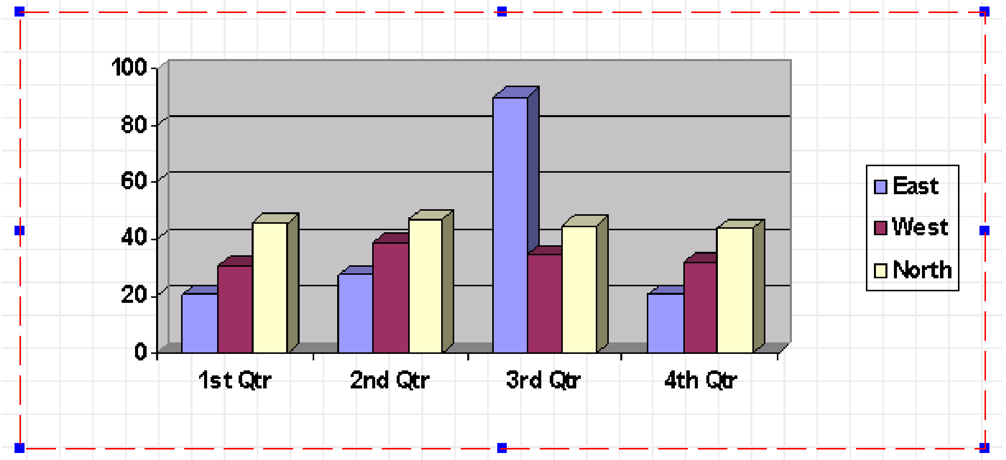
This is a rich text.
Let us see what we have.
Different Fonts in different sizes
Different Styles **bold** *italic* underline
Superscript x^2 SubScript x_2
Colors colors and colors
Paragraph Formatting
Left Aligned
Centered
Right Aligned
- Bullets
- and bullets

1) Numbers
2) and Numbers

The rich text editor.

3.11 OleObject Tool

With the OleObject Tool you can create Insert Ole Objects to your drawing . An ole object has got a rectangular draw area .



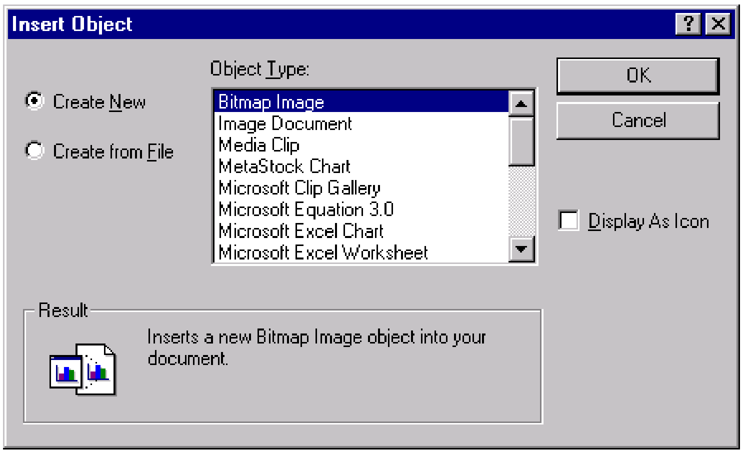
An ole object on PowerCad Editor. It is a ***Microsoft Graph.***

An ole object can be created in following ways.

You can create an ole object by clicking on two different location on the editor. This locations will define the rectangular area of the ole object. The ToolIdx should be set as fallows or the *OleObject Button* ( ) should be down on the ToolBar associated with TPowerCad component.

```
      PowerCad1.ToolIdx := toFigure
      PowerCad1.CurrentFigure := 'TOleObject';
```

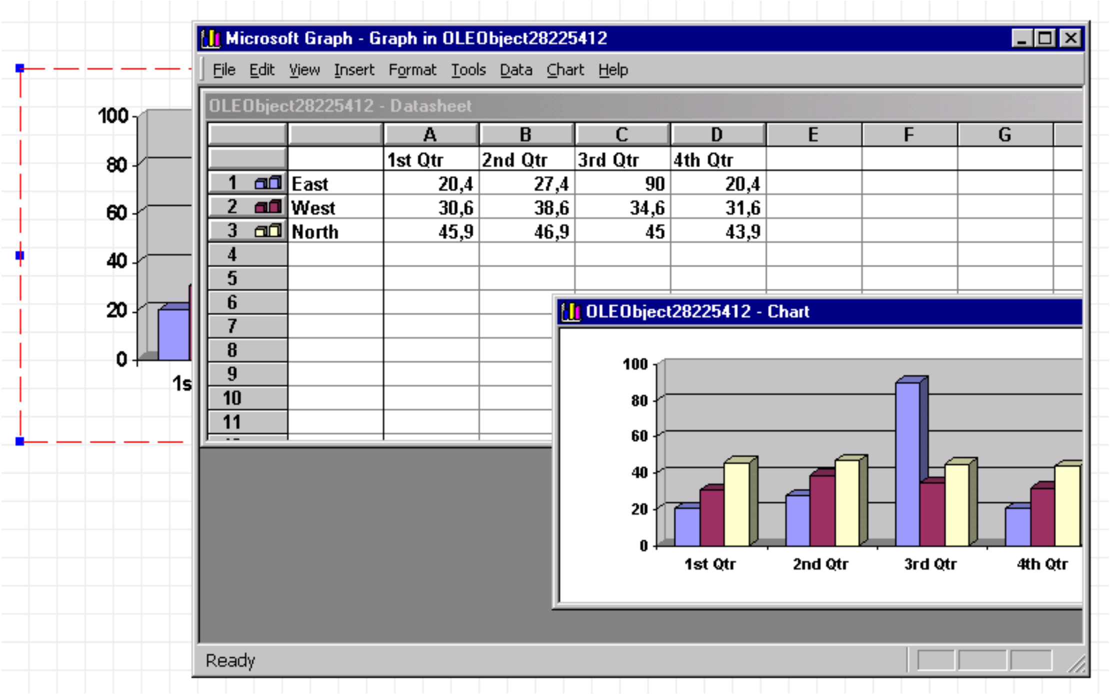
The created ole object will be blank by default. To insert an OLE in it, you should double click. With an ampty OleObject, double click will result an InsertObject Dialog, and with a non-empty OleObject, it will result the edito of the OLE object.



Inserts a new Bitmap Image object into your document.

The Insert Object Dialog of the Ole Object.

Editing ole object on PowerCad Editor. The native applicaiton is launched for editing the object.


Microsoft Graph - Graph in OLEObject28225412
File Edit View Insert Format Tools Data Chart Help
OLEObject28225412 - Datasheet
OLEObject28225412 - Chart
100
80
60
40
20
0
1st Qtr
2nd Qtr
3rd Qtr
4th Qtr
1s
Ready

		A	B	C	D	E	F	G
		1st Qtr	2nd Qtr	3rd Qtr	4th Qtr			
1	East	20,4	27,4	90	20,4			
2	West	30,6	38,6	34,6	31,6			
3	North	45,9	46,9	45	43,9			
4								
5								
6								
7								
8								
9								
10								
11								

3.12 Move Tool

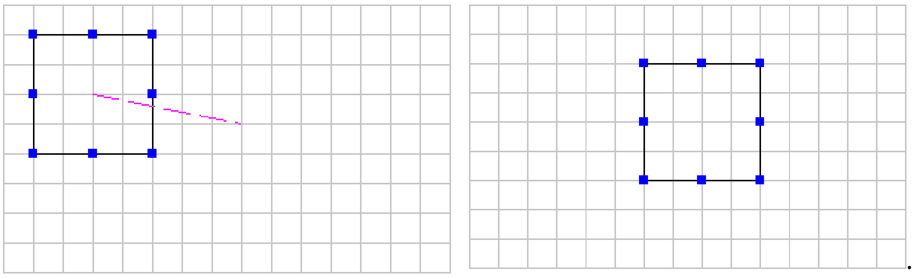
With the Move Tool you can move selected objects in horizontal and vertical directions. (Note that you can also move a figure by drag and drop with out the move tool. *See 'Drag and Drop Case in PowerCad'*)

An object can be moved in following ways.

You can move an object by clicking on two points on the editor. The ToolIdx should be set as fallows or the the *Move Button* (⊡) should be down on the ToolBar associated with or TPowerCad component.

```
        PowerCad1.ToolIdx  := toOpeartion
        PowerCad1.CurrentFigure  := 'TMove';
```

A move tool will need the two points to define the new location of the object(s) by calculating the delta x and delta y value. For example; if the first click is on the center of the object , then the center of the object will be on the second click afer the move has realized.



Moving a figure on the PowerCad Editor.

You can also move an object(s) by using the method MoveSelection. The MoveSelection method will move the selected object with the given delta x and delta y value.

```
        PowerCad1.MoveSelection(...);
```

If you have got the handle of the object(s) that you want to move then you can use native move call as fallows.

```
        MyLine.Move(...);
```

3.13 Duplicate Tool

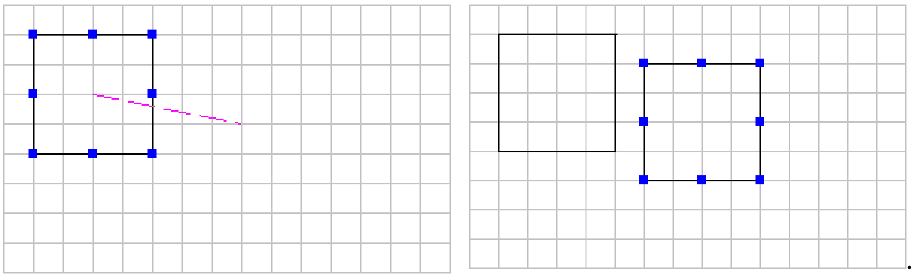
With the Duplicate Tool you can duplicate selected objects on a different location. (Note that you can also duplicate a figure by copying and pasting with out the duplicate tool. *See 'Clipboard in PowerCad'*)

An object can be duplicated in following ways.

You can duplicate an object by clicking on two points on the editor. The ToolIdx should be set as fallows or the the *Duplicate Button* (⧉) should be down on the ToolBar associated with or TPowerCad component.

```
        PowerCad1.ToolIdx := toOpeartion
        PowerCad1.CurrentFigure := ‘TDuplicate’;
```

A duplicate tool will need the two points to define the new location of the new object(s) by calculating the delta x and delta y value. For example; if the first click is on the center of the object , then the center of the new object will be on the second click afer the duplicate has realized.



Duplicating a figure on the PowerCad Editor.

You can also duplicate an object(s) by using the method *DuplicateSelection*. The *DuplicateSelection* method will duplicate the selected object in a defferent location with the given delta x and delta y value.

```
        PowerCad1.DuplicateSelection(...);
```

If you have got the handle of the object(s) that you want to duplicate then you can use native duplicate call as fallows.

```
        MyLine.Duplicate(...);
```

3.14 Rotate Tool

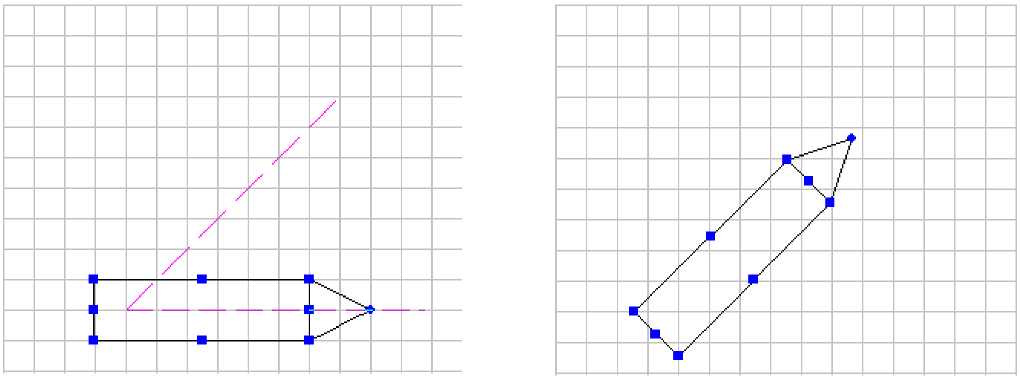
With the Rotate Tool you can rotate selected objects on a different location.

An object can be rotated in following ways.

You can rotate an object by clicking on three points on the editor. The ToolIdx should be set as fallows or the the *Rotate Button* ( ) should be down on the ToolBar associated with or TPowerCad component.

```
        PowerCad1.ToolIdx := toOpeartion
        PowerCad1.CurrentFigure := 'TRotate';
```

A rotate tool will need the three points to define the location of the new points of the object(s). The first point is the rotation center, so the selected object(s) will be rotated around this point. The second and the third point will form a rotation angle by the lines to the center point.



Rotating objects on PowerCad Editor. *The intersecting of the guide lines are the rotation center (first click). And the other two points for the lines to the rotation center for creating an angle. Any theree points on the editor will cause a rotation, but to be more close to the human sense, take the opeartion as selecting an axis on the object (the first two click) and then selecting the new layout of the pre-selected axis (the last click).*

You can also rotate an object(s) by using the method RotateSelection. The RotateSelection method will rotate the selected object(s) with the given angle and rotation center point..

```
        PCDrawing1.RotateSelection(...);
```

If you have got the handle of the object(s) that you want to roate then you can use native rotate call as fallows.

```
        MyLine.Rotate(...);
```

3.15 Mirror Tool

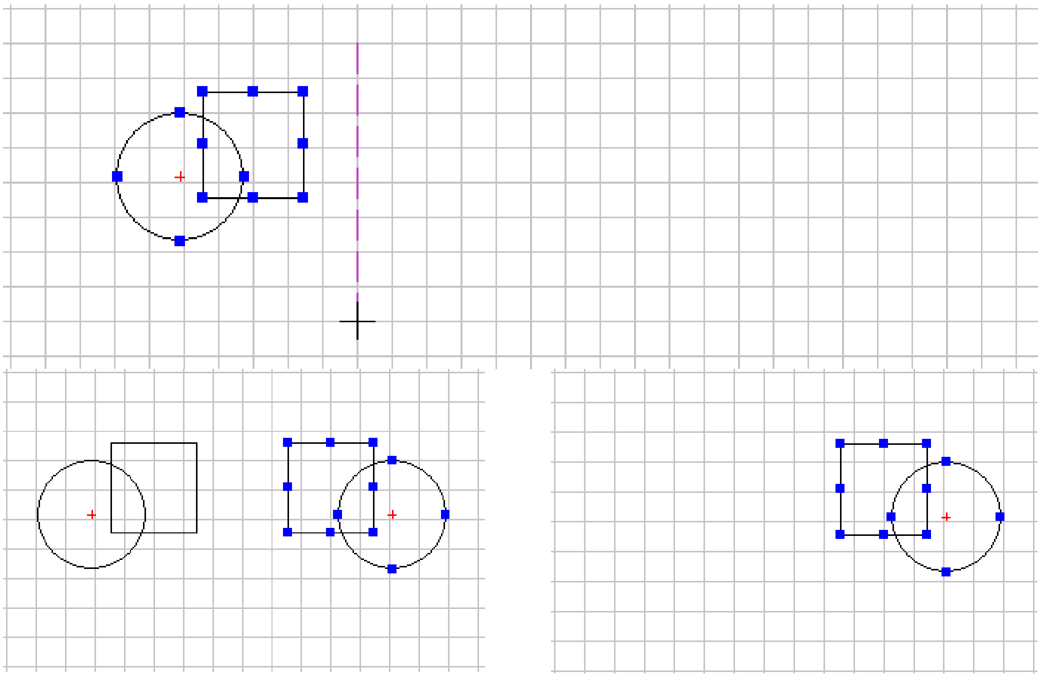
With the Mirror Tool you can have mirror selected objects by the help of a mirror axis. When making mirroring you can either save the old object(s) and create the mirrored one(s) as new, or just relocate the old object(s).

An object can be mirrored in following ways.

You can mirror an object by clicking on two points on the editor. The ToolIdx should be set as fallows or the the *Mirror Button* ( ) should be down on the ToolBar associated with or TPowerCad component.

```
        PowerCad1.ToolIdx := toOpeartion
        PowerCad1.CurrentFigure := 'TMirror';
```

A mirror tool will need the two points to define the location of the new points of the object(s). These two points together form an axis which represents the virtual mirror that will relocate the objects. In PowerCad editor if you want to save the old object(s) click on the shift button when mirroring.



Mirroring objects on PowerCad Editor. *In the first figure you see the user defining the mirro axis. The second figure is the mirrored objects with the duplicate option and the third figure is with out the duplicate option. The duplicate option is triggered with the shift tool pressed. Note that your mirror axis is not limited with vertical or horizontal lines. You can select any two points for defining the mirror axis.*

You can also mirror an object(s) by using the method *MirrorSelection*. The *MirrorSelection* method will mirror the selected object(s) with the given mirror axis and duplicate option.

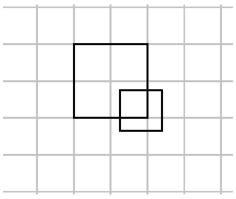
```
        PowerCad1.MirrorSelection(...);
```

If you have got the handle of the object(s) that you want to mirror then you can use native mirror call as fallows.

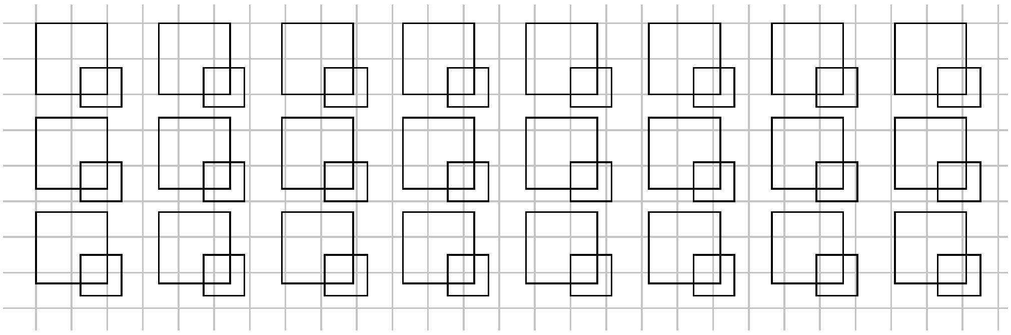
```
        MyLine.Mirror(...);
```

3.16 Rectangular Array Tool

With the Rectangular Array Tool you can multiply the selected object(s) in horizontal and vertical directions.



The objects before and after they are arrayed in rectangular form.

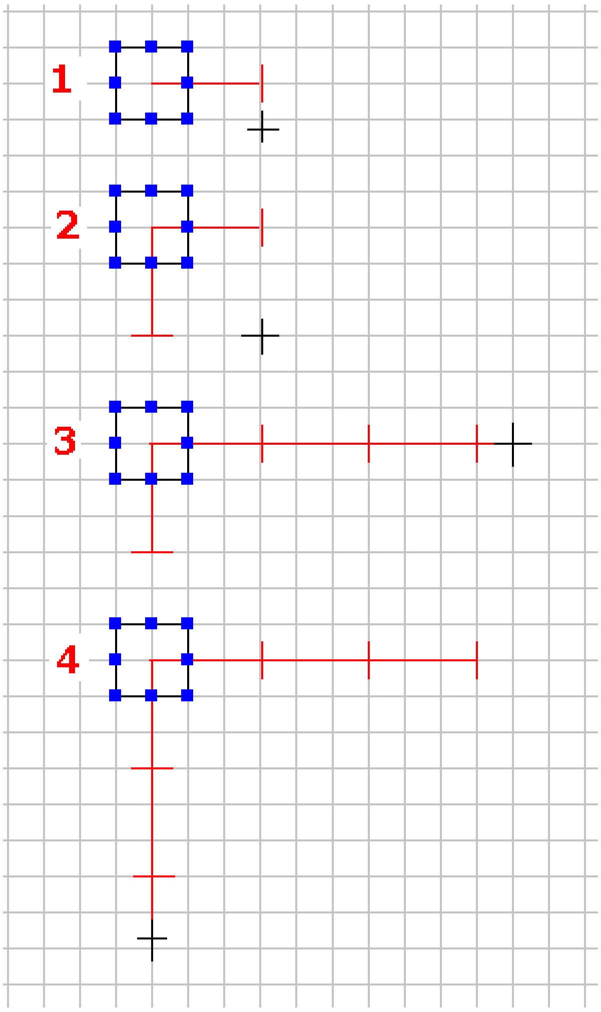


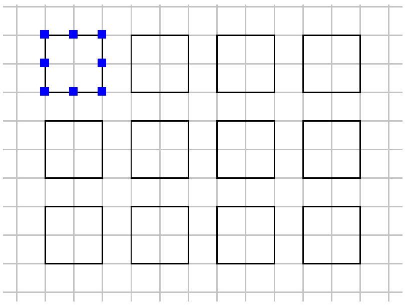
An object can be arrayed (in rectangular form) in following ways.

You can array an object by clicking on five points on the editor. The ToolIdx should be set as fallows or the the *Array Rect Button* (▦) should be down on the ToolBar associated with or TPowerCad component.

```
        PowerCad1.ToolIdx := toOperation
        PowerCad1.CurrentFigure := 'TArrayRect';
```

A rect array tool will need the five points to define the number of the repeating in horizontal and vertical directions and the distances between the repeatations. The first point is the reference point and the following two points will define the horizontal and vertical distances. After this three point, the next two point will define the number of repeatations. (the number of rows and columns).





Arraying an Object

***1 .** The referance point on the object is selected with the **first click** and the user is defining the x distance with the **second click**.*

***2.** The x distance is Ok, now the user is defining the y distance with the **third click**.*

***3.** The distances are Ok. Now the user is defining the number of horizontal repeatations by showing the PowerCad a big distance with a **fourth click**. The number is calcilated by the divisin of the big distance to the small distance.*

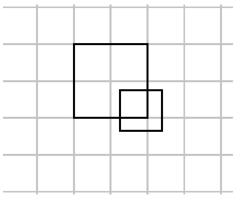
***4.** The horizontal number is OK, now with the **fifth click** the user is defining the number of vertical repeatations.*

You can also array an object(s) in rectangular form by using the method ArrayRectSelection. The ArrayRectSelection method will array the selected object(s) with the given distance and repeatations.

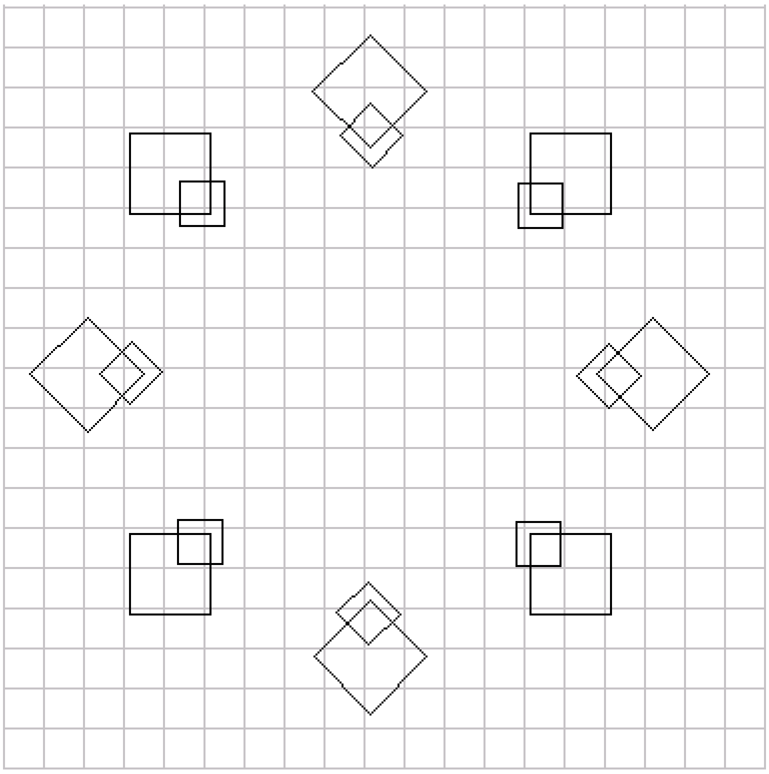
```
    PowerCad1.ArrayRectSelection(...);
```

3.17 Polar Array Tool

With the Polar Array Tool you can multiply the selected object(s) in a circular form.



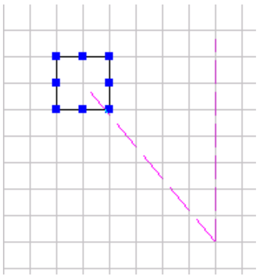
The objects before and after they are arrayed in polar form.



An object can be arrayed (in polar form) in following ways.

You can array an object by clicking on three points on the editor. The ToolIdx should be set as fallows or the the *Array Polar Button* ( ) should be down on the ToolBar associated with or TPowerCad component.

```
      PowerCad1.ToolIdx := toOpeartion
      PowerCad1.CurrentFigure := ‘TarrayRect’;
```



A Polar Array tool will need the three points to define the location of the new points of the repeated object(s). The first point is the rotation center, so the selected object(s) will be repeated around this point. The second and the third point will form a repeat rotation angle by the lines to the center point.

You can also array an object(s) in polar form by using the method ArrayPolartSelection. The ArrayPolarSelection method will repeat the selected object(s) with the given angle and rotation center point..

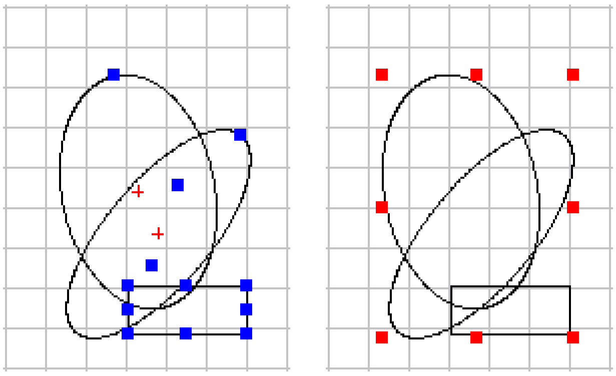
```
       PowerCad1.ArrayRectSelection(...);
```

4 Basic Arrangements

Arrangements are not tools, this means they do not need user specified points on the editor. You just select the object(s) and apply the arrangements. All arrangments are made by method calls and through a button click on the Arrange Toolbar.

4.1 Grouping / Ungrouping

Grouping objects make them behave tohether in the environment. When you select a gruop, all of the objects in the group is selected together.



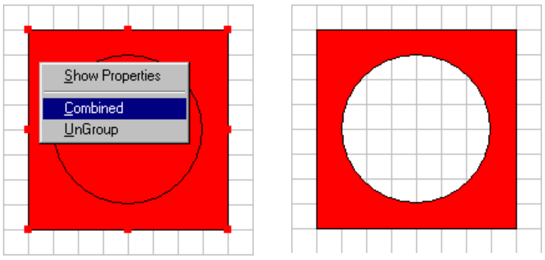
On the left, the first figure shows the objects before they are grouped and the second one shows them after gruping. An ungroping will breake the group into the previous objects again.

To group/ungroup the selected objects click on the *Group Button* ( ) /*UnGroup Button* ( ) on the Arrange toolbar assocaited with your TPowerCad or use the methods below.

```
       PowerCad1.GroupSelection(...);
       PowerCad1.UnGroupSelection(...);
```

You can also, group figures which are created in different layers. For the figures, being in different layers is not an objection for being a group.

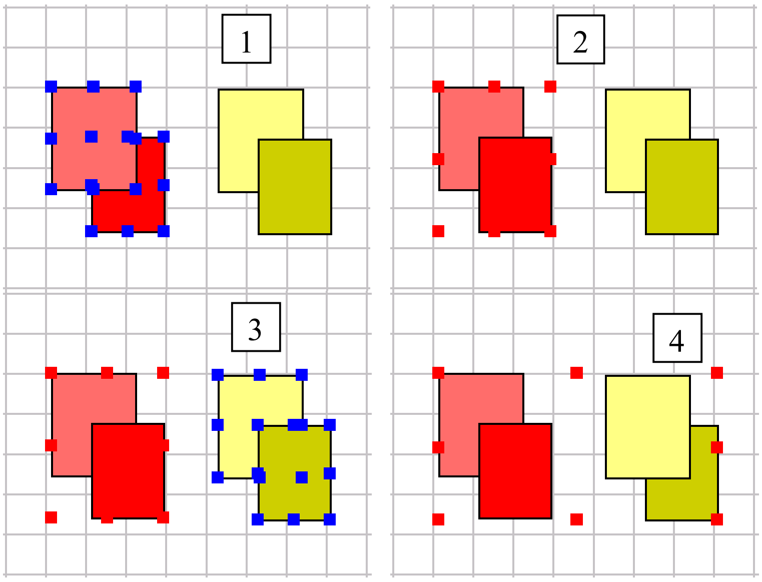
Combined Grouping



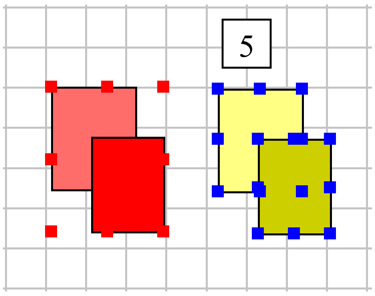
If you combine a group, the group will be painted in ALTERNATIVE style. So you can have holes in figures. To have combined group first group the filled objects and then from the popup menu make the group combined.

Nested Grouping

If you regroup any grouped object with other groups or indivuduals you make nested grouping. In this case when you ungroup, the final group will not break into all indivuduals, but the groups in the group will be saved again as a group.



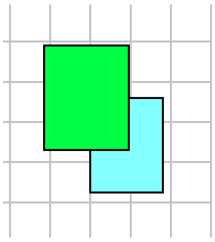
Example:
First we group the two red objects. Then with this group and aothr two yellow objects we make another grouping. ***This is nested grouping.***



When we ungrop the group in figure 4 above, we will have the result on the left.

4.2 Ordering

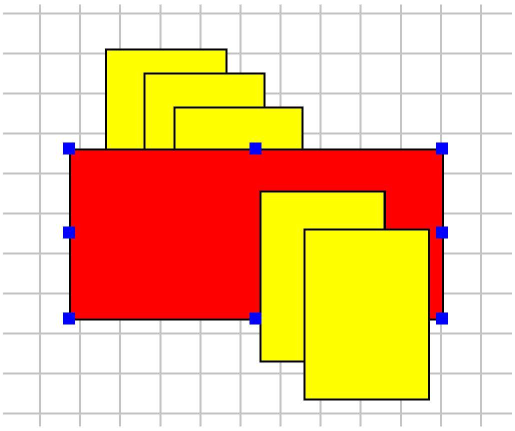
Ordering objects provides arranging the order of them in z direction. The one which is most front is drawn latest, so it is not covered by any other objects. The one which is most back is drawn first so, it is possible for other objects to cover it.



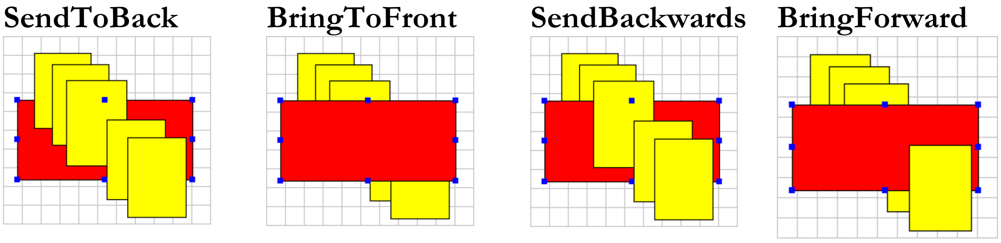
Z-Order : *Z-Order is the drawn order of the objects. On the left blue rectangle is drawn before the grennone is drawn. So the green one is more closer to the fron in z-order.*

There are for arrangements for ordering the objects. **1.**Bringing to Front **2.** Sending to Back **3.** Bringing Forward **4.** Sending Backwards

Bringing to Front, will bring the object to the most front, and sending it back will send it to the most back. Bringing forward will bring the object one step to the front and sending backwards will send it one step to back.



On the left figure the order of the objects is in their initial state The red object is selected and its order will be arranged as an example. If you apply each order arrangment below seperately from each other, you will have the results below the commands.

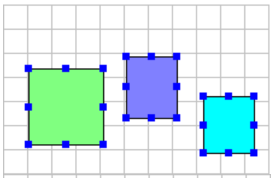


To order the selected objects click on the related *Order Button* ( ) on the Arrange toolbar assocaited with your TPowerCad or use the method below.

```
      PowerCad1.OrderSelection(...);
```

4.3 Aligning

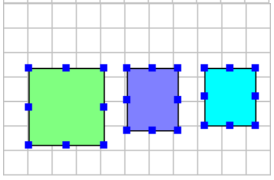
Aligning objects provides relocating them as their lefts, rights, tops, bottoms or their centers stand in one horizontal or vertical line. The firstly selected object is always referance for the aligning.

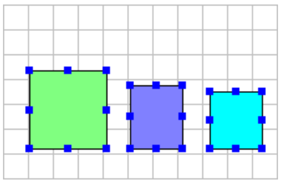


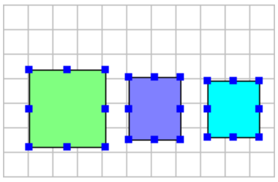
Horizontal Aligning

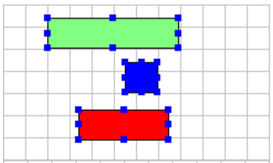
*In the **left** figure the objects before they are aligned.*

*In the **below** figures the objects after they are aligned. **1.**Aligning Tops **2.**Aligning Bottoms **3.**Aligning horizontal centers*





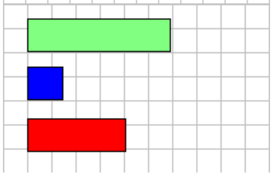


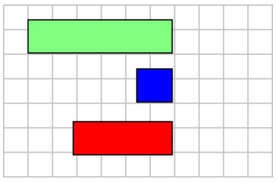


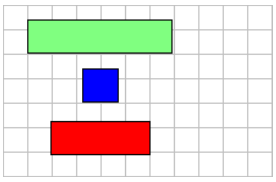
Vertical Aligning

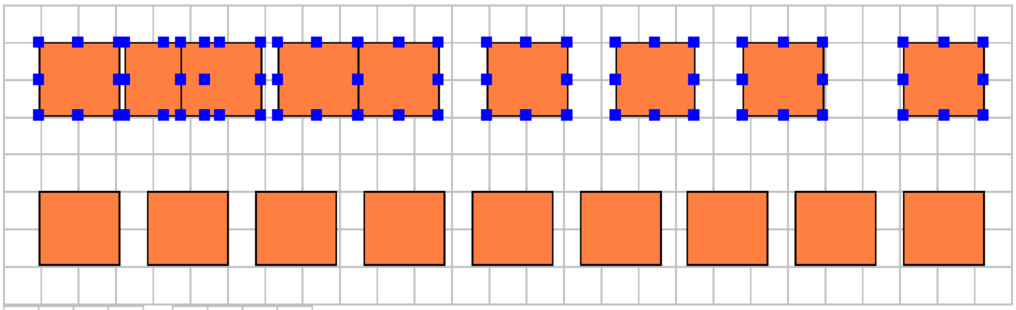
*In the **left** figure the objects before they are aligned.*

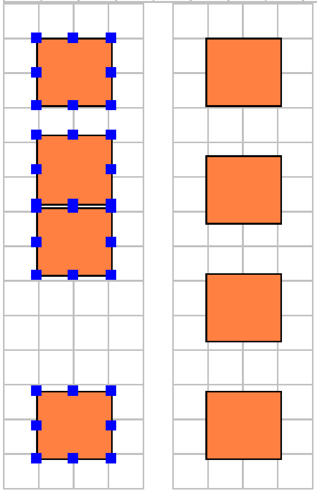
*In the **below** figures the objects after they are aligned. **1.**Aligning Lefts **2.** Aligning Rights **3.**Aligning vertical centers*











Distrubuting Centers:

*The Align Arrangement also includes distrubuting. With horizontal distrubuting (**above**), you can arrange the horizontal inter distances of the objects evenly and you can do the same for vertical distances with vertical distrubiting (**left**).*

To align the selected objects click on the related *Align Button* ( ) on the Arrange toolbar assocaited with your TPowerCad or use the method below.

```
      PowerCad1.AlignSelection(...);
```

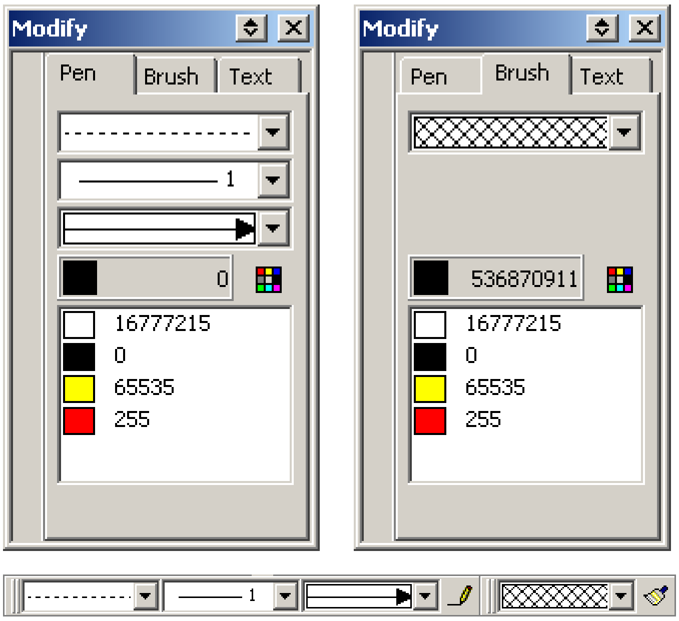
5 More about PowerCad

5.1 The "Pen" and the "Brush"

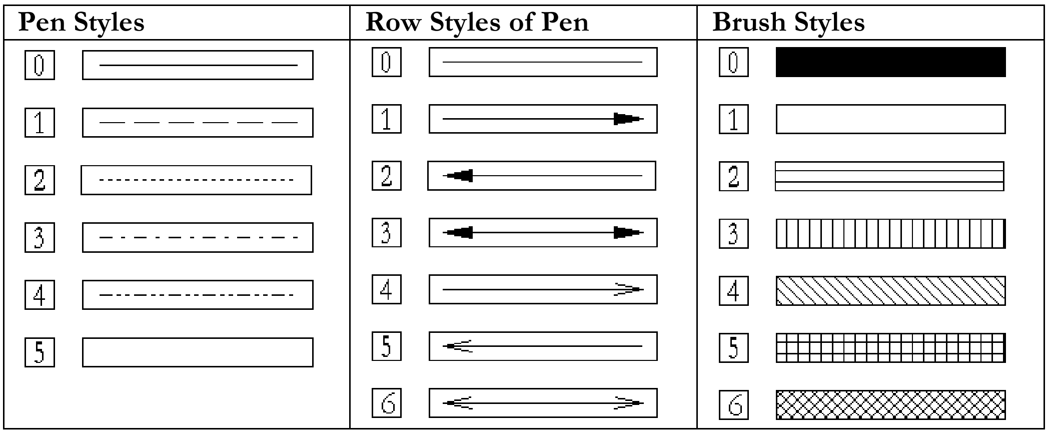
To draw something on a paper, you need a pen and to paint your objects you need a brush. This is not so different on digital drawings. PowerCad drawing system is based on Windows GDI (Graphical Device Interface) and before the 3rd release we do not want to go beyond the Windows GDI. So for now we are limited with windows' pen and brush.

In Windows GDI, a **pen** is a virtual actual record for a device context (screen, printer etc.) which stores the drawing style, the drawing width and the drawing color. So windows uses this record when making any drawing on the device context. And just like pen, a **brush** is also a virtual actual record for a device context but this time it stores the paint style and paint color of the drawings. So in one sentence the GDI draws the rectangle with the actual Pen and fills it with the actual Brush.

In PowerCad each figure has a pen and brush settings for itself. These settings are gathered from environmental default settings when first creating and then modified through the related dialogs. Also when creating figures through method calls you should know that some of the method parameters are for these pen and/or brush settings.



The Pen and Brush Settings can be modified through the ***Modify window*** *(left) associated with your control. The Pen settings include the style of the pen, the width of the pen, the row style of the pen (only in linear figures), and the color of the pen. The Brush settings include the style of the brush and the color of the brush. The pen and brush settings can also be modified through the* ***modify toolbar*** *(bottom) associated with you control.*



When creating figures through the method calls, the Pen and Brush styles are defined in the method parameters.

```
        PowerCad1.Rectangle (*,*,*,*,*,*,w,s,c,brs,brc,…);
        PowerCad1.Line(*,*,*,*,*,w,s,c,row,…);
        // w: pen width, s: pen style, c: pen color
        // brs: brush style, brc: brush color
        // row: row style
```

And to modify the pen and brush styles of the created figures through the method calls you should use the *ModifySelection* method.

```
        PowerCad1.ModifySelection(…)
```

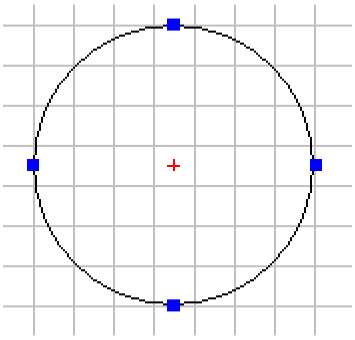
Also, if you have got the figure handles you can modify them directly through native calls.

```
        MyLine.Width := 1;
        MyLine.Color := clRed;
        MyLine.Style := 2; // or MyLine.Style := ord(psDot);
        MyLine.RowStyle := 0; // No rowed end
```

5.2 The Points of the Figures

When a figure is drawn on th PowerCad editor, you can see some points drawn with the figure when it is selected or not. These points help the user for modificating the figure and to guide user to show give some information about itself. To modify the figures we use the **Modification Points** and to be informed about them we look at the **Guide Points**.

The modification points are drawn when the figure is selected and by dragging and dropping this points the figure shape is modified or it is rescaled. The guide points are always drawn if the DrawGuidePoints of the figure is not set to false. These points give us location information such as the center of a circle or the focus of an ellipse. Note that the guide points or the modification points are not drawn when making **printing**. (They are not printed)



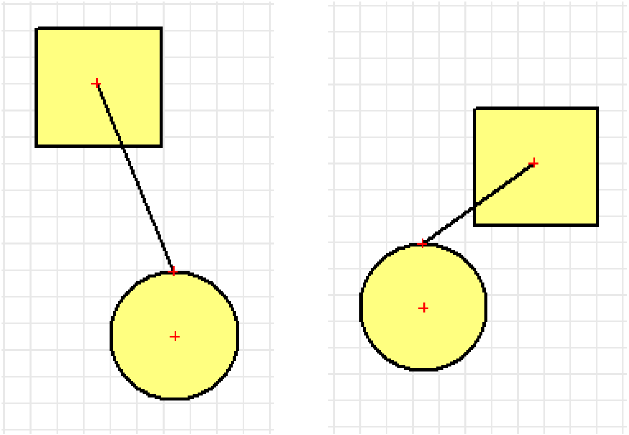
The blue rectangular points are the ***modification*** *points of the circle. By dragging and dropping these points, the radius is redefined. And the red cross point is the* ***guide*** *point of the circle which shows the center of the circle.*

Some figures in PowerCad such as polyline and ellipse , have got some points also for modification that are not located on the figure. These are the **control points** and in PowerCad, they also called modification points since they are used to reshape the figure.

5.3 The Joint Lines

In some drawings, some lines are used as bounds among two other figures. So when any of these figures move the end point of the line should also move to the same location of the figure. We call these lines *Joint Lines.*

In PowerCad a Line figure or a PolyLine figure can act as a joint line. You can join one of or both of the ends of a line/polyline to other figures. When a line end is joined to any figure a small red cross will be drawn in joined end of the line. (Note that these crosses are gu*ide points* of the line)



As you can see on the left figures, the joint line reacts to the figure movements. The line ends insist on saving their locations relative to the figures. Note that the joint ends are marked with the crosses.

To join a line/polyline end on PowerCad editor, drag the end of the line when it is selected, with your mouse (by also pressing *shift* and *ctrl* keys together), and drop it on the *selection area* of the figure you want to joint to. Here two points to be aware of:

1. The *selection area* of the figure is the area on which you press to select. For instance if you drop the line end inside a non-filled rectangle, it will not be joined. So to make easy joins first fill the figure or drop the line end on the edge of the figures.
2. Be free to drop it on any location of the figure. Because after you make the join, you can easily change the location of the join end. For instance, consider you have dropped the line end on the center of a circle when joining, and then you can redrag the join end to the edge (or even outside of it) of the circle. It will always remain joined.

To unjoin a line on the PowerCad editor you should use the line popup menu.

To control line joinings through the method calls please use the code below.

```
      PowerCad1.BoundLineToFigures(...);
      PowerCad1.BoundLinePoint(...);
      PowerCad1.UnBoundLine;
```

If you have got the handle of the object(s) that you want to join then you can use native method calls as fallows.

```
      MyLine.SetJFigure1(...);
      MyLine.SetJFigure2(...);
      MyLine.UnBound;
```

5.4 Handling Bitmaps

In PowerCad, bitmaps can be inserted from the disk, and they can be manipulated on the powercad editor. A bitmap data is normally hasn't got metric dimension info as all other picture formats. Almost every picture format is dimensioned with its pixel counts in width and height. So the applications which uses metric style of dimensioning, redimension these pictures according to an optimal resolution.

So now, the question is "What does resolution for a picture mean?". The resolution of a picture is the density of the pixels which defines how many of them will be drawn in one inch. Normally if you decrease the resolution the picture size in metrics will increase but the quality of it will decrease. And when you increase the resolution (so you will put more pixels to 1 inch size), your picture quality will increase when your picture dimensions decrease. But increasing the resolution will not always mean high quality , because after an optimal point the pixels in one inch will exceed the display adapter or printer possibilities and they will be drawn on eachother. So when we speak about the resolution we use the word optimal.

In PowerCad the optimal resolution for the bitmaps is 180 dpi. That is; if there are 1800 pixels in the width of the bitmap data, the width of the bitmap in PowerCad will be 10 inch (= 25,4 cm = 2540 dmm) by default. But after inserting the bitmap you can manage the sizes (width and height) by scaling it. In this case the resolution will decrease or increase of course.

To insert bitmap to the PowerCad editor you will just click on one point to show the topleft location of the picture. The ToolIdx should be set as fallows.

```
      PowerCad1.ToolIdx := toFigure;
      PowerCad1.CurrentFigure := 'TBmpObject';
```

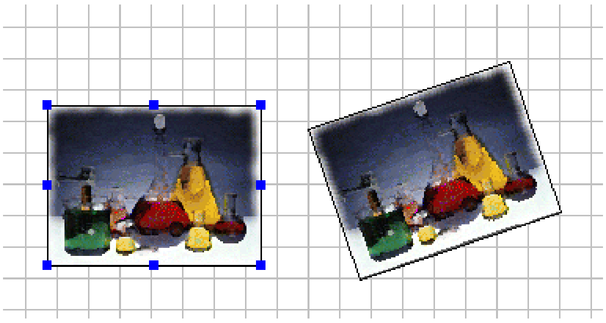
You can also insert a bitmap by using the method *InsertBitmap*. The *InsertBitmap* method creates a bitmap figure and returns its handle.

```
      PowerCad1.InsertBitmap(...);
```

You can do several operations on a bitmap inserted to the PowerCad editor like scaling, rotating, flipping and transparenting.

Scaling Bitmaps : Scaling bitmaps on the powercad editor is not so different than scaling any object. Only you should know that the resolution (so the quality) changes when you resize the bitmaps.

Rotating Bitmaps : Rotating bitmaps on the powercad editor is not so different than rotating any object. But to handle this operation PowerCad does a hard work in the background. To rotate a bitmap, powerdraw creates a new bitmap which stores the new locations of the scanlines (horizontal pixel set).

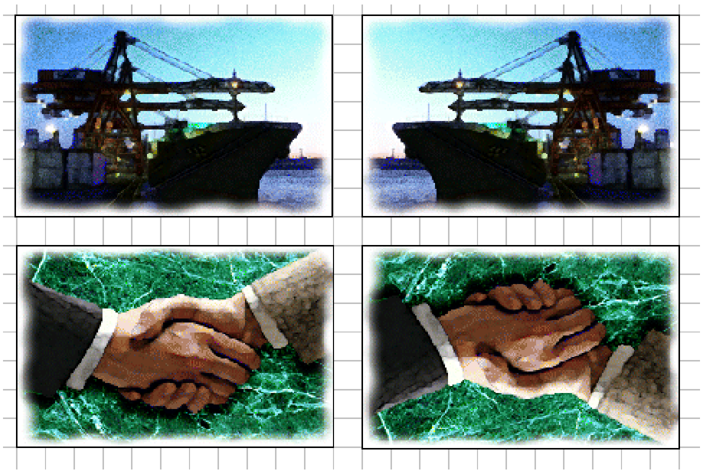


The bitmap with its rotated copy.

Flipping Bitmaps: Flipping is a specific arrangment for bitmaps. You can flip the bitmaps in **horizontal** or **vertical** direction. In case of flipping PowerCad inverts the location of the bitmap scanlines. So the one at the beginning goes to the end.

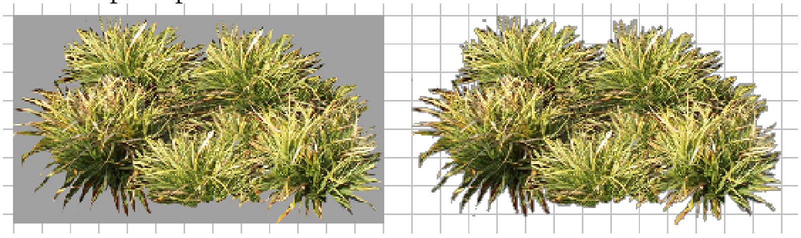
To flip the selected bitmap(s) click one of the the *Flip Buttons* ( ) on the Arrange toolbar assocaited with your TPowerCad or use the method below.

```
        PowerCad1.FlipImagesOfSelection(...);
```



The above pictures show the horizontal flipping and the below pictures show the vertical flipping.

Transparent Bitmaps: In fact for the bitmaps, there is nothing for the transparency in the file format. A **gif** can be transparent or not, this property of it is stored in its file format. However, for bitmaps, the transparency is defined on the application level by the users or the applications itself. To draw a bitmap as transparent on a canvas in windows means to redraw the background pixel instead of the bitmap pixel if it has a specific color (or the transparent color). So in Windows GDI to draw a bitmap transparent we should tell windows about the transparent color. In PowerCad this color is defined by the code itself by looking at the topleft pixel color.



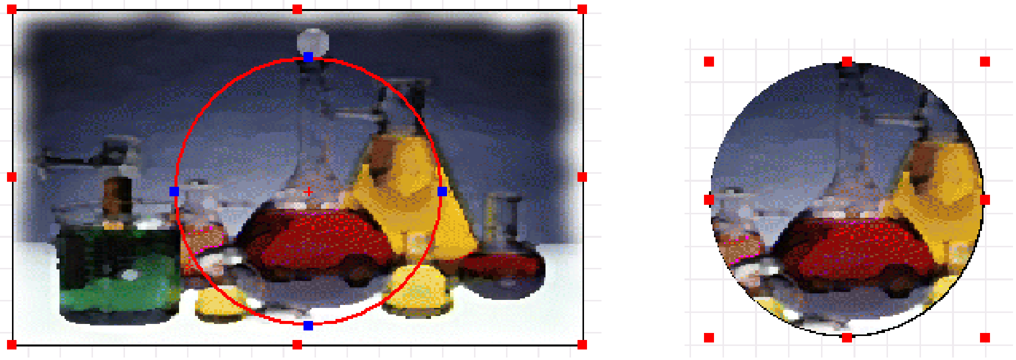
Transparenting bitmaps.

To make the selected bitmap(s) transparent use the method below.

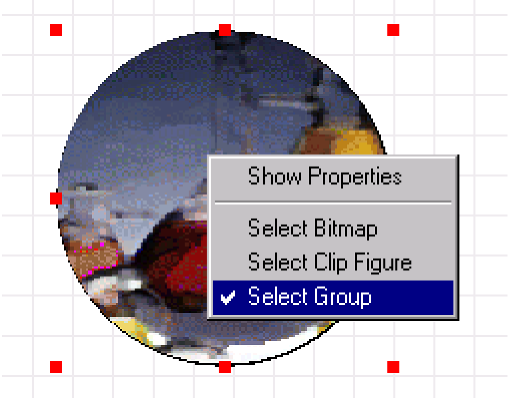
```
        PowerCad1.SetTransparentOfSelection (...);
```

Clipping Bitmaps: The bitmaps on the PowerCad editor can be clipped in a different close figure. To make this, the ClipSelBitmapToSelFigure method should be called when the bitmap and the clipping figure are selected together.

```
        PowerCad1.ClipSelBitmapToSelFigure (...);
```



A bitmap object cliped in a circle.



Selection of a Clipped Bitmap

The clipped bitmap, is selected as group by bitmap. This means the clip figure and the bitmap is selected together. More, you can select the bitmap, or you can select the clip figure alone from the pop up menu.

Storing Bitmaps: When a bitmap is inserted to PowerCad and the drawing saved to a file, the bitmap data is saved to the drawing file also. So when you make delivery of PowerCad drawings you shouldn't also deliver the bmp files with the drawing.

5.5 Handling Metafiles

In PowerCad, metafiles can be inserted or imported from the disk. If you insert a Windows Metafile (WMF) then a TWMFObject will be created and the inserted WMF will be behaved as it is a picture. The inserted WMF files are drawn to active device contexts by using Windows GDI directly. However, when you import a WMF it will be converted to PowerCad objects such as lines, rectangles, arcs etc. In this case the WMF file is analyzed and a convertion is handled. If you should edit the WMF then import it else it is always better to insert it as a picture.

To insert a metafile to the PowerCad editor you will just click on one point to show the topleft location of the picture. The ToolIdx should be set as fallows.

```
        PowerCad1.ToolIdx := toFigure;
        PowerCad1.CurrentFigure := 'TWmfObject';
```

You can also insert a bitmap by using the method *InsertWmf.* The *InsertWmf* method creates a WMF figure and returns its handle.

```
        PowerCad1.InsertWmf(...);
```

An inserted metafile can be rotated or mirrored only if the operating systen is NT based.

To import a metafile, you should use the ImportWMF method. This method analyzes the metafile and converts it to editable PowerCad objects.

```
        PowerCad1.ImportWmf(...);
```

5.6 Clipboard in PowerCad

When windows was first introduced, we all liked its Ctrl-C/Ctrl-V feature. Copying and pasting items is a standart in Windows applications, so any visual tool, should provide a copy/paste feature.

In PowerCad you can copy, cut and paste objects on the editor. These are just a few button clicks or method calls. But what is more important to declare here is the background process held in PowerCad trans clipboard.

Clipboard is an application common memory area managed by the Windows, to store temporary data which will be used later or exported to another application. When you select and copy a text from NotePad and then paste it to WordPad, what is done is this; Notepad writes the data(text) to the clipboard and Wordpad reads the data(text) from the clipboard.

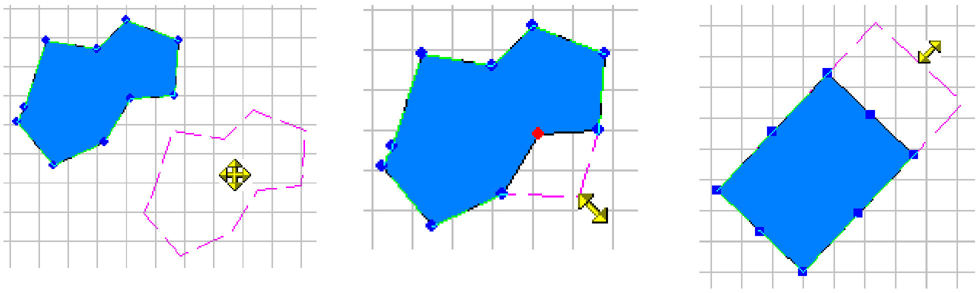
When an aplication copies its data to the ClipBoard it marks this data with a format indicator, so a different application that will share this data in the clipboard, should regard to the format of the data. Windows Clipboard API provides a standart interface to the applications, so that most aplications share their data by sending clipboard data in standart format. But since the limitations of the standart formats and the natural conversion rounding errors, an application sometimes sends the data to the clipboard with its own format in addition to a standart format.

PowerCad sends the figure data to the Clipboard in two formats: When the *CopySelection* and *CutSelection* methods are called, the figure data is written to the clipboard as PowerCad Streamed figure format and Windows Metafile Format. So the data in the Clipboard can be used by either PowerCad with no destruction in the numbers or by any application which can communicate with WMF.

When the *PasteFromClipboard* method is called, PowerCad checks Clipboard if any data is stored with its own format. If any PowerCad Data is found, PowerCad retreives this data and creates the figures stored in. And if PowerCad can not find any data in its own formta, then it looks if any WMF data exists. If WMF data is found then PowerCad creates figures from this WMF. So in anydrawing application, you can copy the drawing items to the Clipboard and paste ot to PowerCad drawing.

5.7 "Drag and Drop" Case in PowerCad

Famous Windows feature "Dragging and dropping" is active in PowerCad when you make locating. Locating in PowerCad is to change the location of a figure point, or the the figure itself. By dragging and dropping you can move the figure to a new location, or you can move the modification points of the figure, thus you make scaling or some other modifications on the figure.



5.8 File I/O in PowerCad

File formats of the applications are treated as they are secret. Most applications do not document their file formats to save their know-how in their organizations. The user created files of these applications are not accassable from other applications unless the file format is cracked.

PowerCad is not an application, it is a programming tool which contributes the creating of several different applications. So the problem here is about the file format. What will happen when we introduce a strict file format which is well documented? Yes, as you can guess each application created with PowerCad will have the same file format which is open to eachother and to other industry crackers. So PowerCad should provide something different and more than the file format.

Normally PowerCad has got two types of file formats. The first is the textual file format and the second is the binary file format. If you want to use them in standart ways you just use the standart IO method of PowerCad to save and load drawings.

If you do not want to use the standart IO, that is; you want to save drawings as not to be accessable from any other PowerCad application, you should customize the IO interface of PowerCad.

Below, you will see using the standart file IO and the custom file IO.

5.8.1 Using Standart File I/O

PowerCad file has got a binary format. The drawing is organized in sections, and eachsection is written to the file seperately. The file format provides an xml like communication, so that even the format changes, the eralier applications can load later files.

To save a drawing to a file and to load a drawing from the file use the methods below.

```
        PowerCad1.SaveToFile (fileName);
        PowerCad1.LoadFromFile (fileName);
```

The SaveToFile and LoadFromFile methods has got only parameter. The ***filename*** parameter specifes the path where the drawing will be saved to or load from.

PowerCad Binary File Structure

```
[FileStart]
Signature – 8 Bytes – (123,125,212,234,76,65,169,214)
Version – 4 Bytes
Section Count – 4 Bytes
[Sections]
Section Size – 4 Bytes
Section Name – Up to Null Char
Section Data – Up to Section Size
[File End]

Normally 4 sections are stored at the moment: Document Properties, Layers,
Figures,Line Joins
```

A section in PowerCad binary file is structured as fields and values. So version changes will not cause consistency problems. An earlier version is able to open a later version file. Just the earlier application will not show the newly added fileds. Also when new sections are added to the structure, again the new files will be opened in the earlier applications, just the new sections will be ignored.

5.8.2 Using the Custom I/O

The Custom I/O is designed for providing an application specific file format. For this you will get the data of the current drawing as text source or binary stream, then you will save this data to your file in the way you want.

More this custom interface provides you, to save the drawings with in a fiel you save something more about your application, in cases cad features are just a module of your application.

When uyu can get the data of the drawing as a whole stream, you can also get more specific data of the layers and figures separately. So you can save the data to the files in more customized structures.

For these purposes PowerCad has fallowing methods.

```
        PowerCad1.SaveToStream (...);
        PowerCad1.LoadFromStream(...);
```

The first method above, SaveToStream will save all the file data of the drawing as binary to the provided stream. To reload your file, call the *LoadFromStream* method with the stream provided by previous method.

In the following code , the drawing is saved to a bigger stream which includes various data. And also it is loaded from the stream again.

```
Procedure ApplicationSave;
Var
  MyStream: TfileStream;
Begin
  MyStream := TFileStream.Create(‘c:\file.ext’,fmCreate);
  OtherData1.SaveToStream(MyStream);
  PowerCad1.SaveToStream(MyStream);
  OtherData2.SaveToStream(MyStream);
  MyStream.Free;
End
```

```
Procedure ApplicationLoad;
Var
  MyStream: TfileStream;
Begin
  MyStream := TFileStream.Create(‘c:\file.ext’, fmOpenRead);
  OtherData1.LoadFromStream(MyStream);
  PowerCad1.LoadFromStream (MyStream);
  OtherData2.LoadFromStream (MyStream);
  MyStream.Free;
End
```

5.9 Printing Drawings

In PowerCad, the standart Print method, draws all objects directly to the printer canvas (device context). This style of printing is the most performant way and it doesn’t use too much resources from the memory. To use this standart printing use the method below.

```
      PowerCad1.Print (...);
```

The only problem with direct canvas printing that occures in some old printers, is the transparency fail. Some printer drivers can not handle raster operations. So if you have got a transparent bitmap in your drawing, PowerCad uses a temp memory bitmap to make the raster operations. And the rastered bitmap is then copied to the printer canvas. So in these cases you can feel a decrease in the performance.

Tiled Printing

If your printer page size is smaller than your drawing size, than you can print your drawing to more than one pages. This is called Tiled Printing. For tile printing you should give the sizes of the printer page, the drawing will be printed with margins on the pages. For tiled printing use the method below.

```
        PowerCad1.PrintByTiling (...);
```

5.10 Keyboard Commands

If PowerCad control in your form has got the focus, then the keyboard strokes is trapped by Powercad. These key strokes are first checked if they mathch the shortcut of a PowerCad Keyboard Command. The Powercad keyboard commands can only be active if the KeyCommands property is true.

```
        PowerCad1.KeyCommands := True;
```

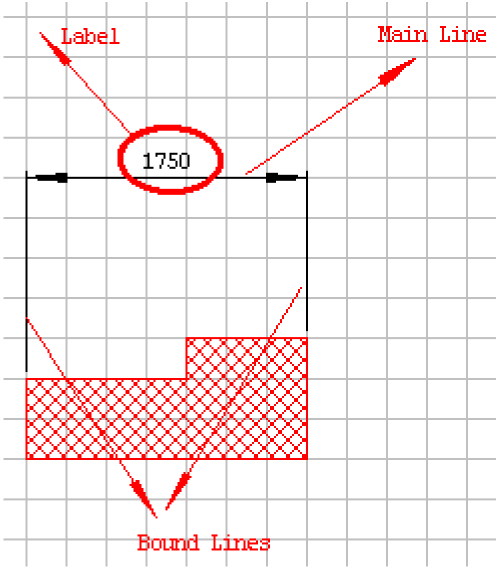
If the key stroke is any of the key commands, PowerCad exectes the matching command. The list of current keyboard commands are as below, in the later versions the list will be extended.

Down Arrow : Moves the selection 0,1 mm downside.
Up Arrow : Moves the selection 0,1 mm upside.
Left Arrow : Moves the selection 0,1 mm leftside.
Right Arrow : Moves the selection 0,1 mm rightside.

Ctrl+Down Arrow : Moves the selection 2 mm downside.
Ctrl+Up Arrow : Moves the selection 2 mm upside.
Ctrl+Left Arrow : Moves the selection 2 mm leftside.
Ctrl+Right Arrow : Moves the selection 2 mm rightside.

DEL : Deletes the selected figures.

Ctrl+A : Selects all figures.
Ctrl+C : Copies the selected figures to clipboard.
Ctrl+X : Cuts the selected figures to clipboard.
Ctrl+V : Pastes the figures from clipboard.
Ctrl+Z : Undoes the last action.
Ctrl+Y : Redoes the last undone action.
Ctrl+G : Groups the selected figures.
Ctrl+Shift+G : UnGroups the selected groups.
Ctrl+F : Brings the selected figures to front.
Ctrl+B : Sends the selected figures to back.
Ctrl+M : Makes the selected figures a block.
Ctrl+N : Prepares the control for a new drawing.
Ctrl+O : Prompts an Open Dialog for loading a drawing.
Ctrl+S : Prompts an Save Dialog for saving the drawing.
Ctrl+P : Prompts an Print Dialog for printing the drawing.



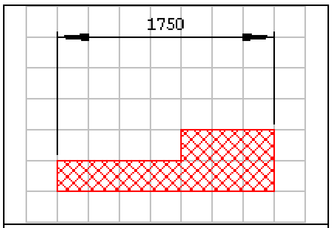
5.11 Dimensioning Tools

You can create dimension lines to show the size of figures and distances. The dimension lines can either show the dimension of pointed distance or any text entered by the user. In the current version of PowerCad, there are 3 types of dimension lines: Horizontal Dimension Line, Vertical Dimenison Line, Aligned Dimension Line.

A Diemension Line is formed by three elements as shown in the figure. The first is the **mainline**, secondly the **bound lines** and last the **Label**.

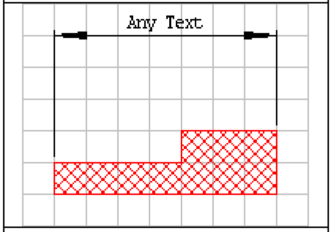
All dimension lines are controlled through their properties, these properties can be arranged from the Object Inspector . Also the right-click menu provides access to most of the dimension line properties.

AutoLabel: This is a Boolean property. When it is set to true, the distance that the dimension line is indicating is measured automatically and it is written in the label of the dimension after the distance is multiplied with the MapScale property of the drawing. If the AutoLabel property is false, then the text in the Label property of the Dimenison Line will be written in the label of the diemension Line.



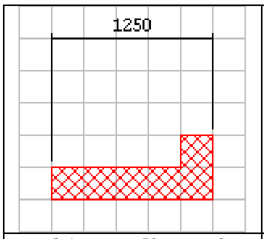
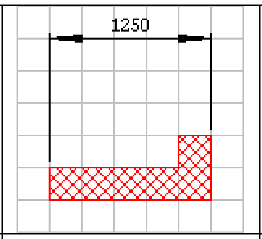
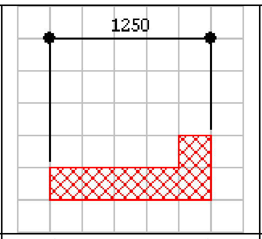
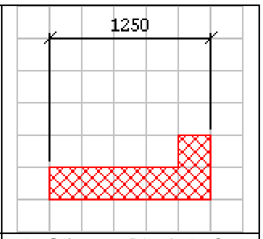
AutoLabel = True

Label: This property is a string property ; storing the text that will be drawen in the label of the dimension line. The text in this property will be written only when the AutoLabel property is set to false.

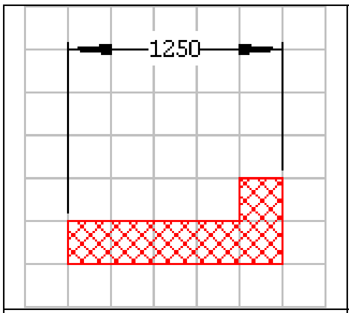
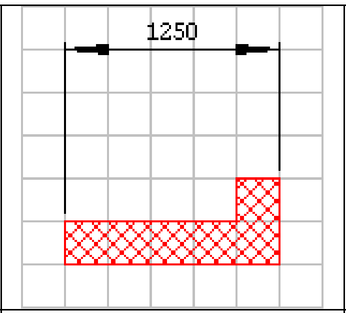
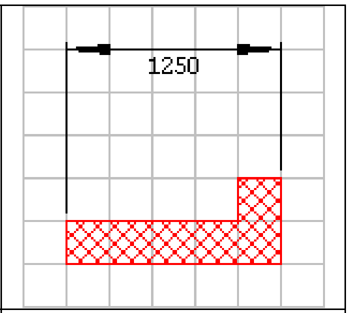


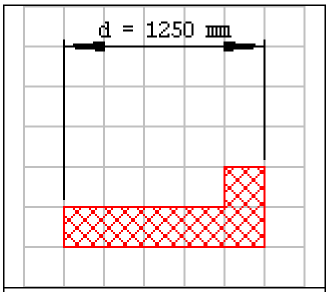
AutoLabel = False

EndType: The value of this property defines the connectings of the ends of the main line to the Bound Lines. This property can be ClearEnd, RowEnd, DotEnd, NickEnd.

			
EndType = ClearEnd	*EndType = RowEnd*	*EndType = DotEnd*	*EndType = NickEnd*

TextPosition: The value of this property defines the location of the label according to the main line. It can be OnLine, AboveLine or Below Line.

		
TextPosition = OnLine	*TextPosition = AboveLine*	*TextPosition = BelowLine*



LabelPrefix = 'd= '
LabelSuffix = ' mm'

LabelPrefix: This property is a text value. Use this property to add strict text before the size in the label. This is active when AutoLabel is True.

LabelSuffix: This property is a text value. Use this property to add strict text after the size in the label. This is active when AutoLabel is True.

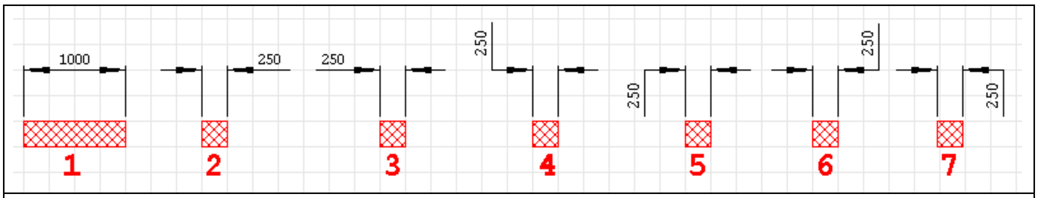
Labeling: The value this property shows the style of the main line with the label together. Normally each type Dimension Line provides different labeling styles, so we will see the details of this property for each type of dimension lines.

Horizontal Dimension Lines

Shows the horizontal distance between two points. The end of the bounding points may be in different y locations. The distance is always the horizontal projection of the points. To create a horizontal dimension line, the *Toolidx* and *CurrentFigure* property should be set as follows.

```
        PowerCad1.ToolIdx := toFigure
        PowerCad1.CurrentFigure := 'THDimLine';
```

To draw a horizontal dimension line PowerCad needs three points. The first two points are the points that defines the measured distance and the third point defines the location where main line will be drawn. Powercad draws two bounding lines 2 mm apart from the distance points to the y location of the third point. The main line is drawn according to the Labeling style of the dimension line. A horizontal dimension line can not be rotated or mirrored. This dimension line provides 7 styles of labeling as illustrated below.



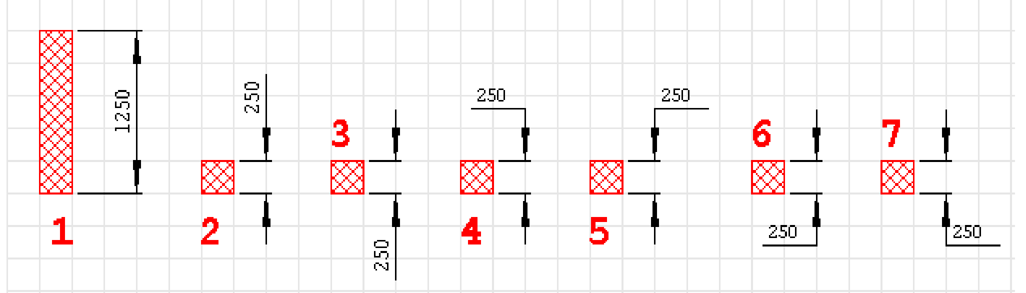
1. *Inside* **2.** *Right* **3.** *Left* **4.** *Left Top* **5.** *Left Bottom* **6.** *Right Top* **7.** *Right Bottom*

Vertical Dimension Lines

Shows the vertical distance between two points. The end of the bounding points may be in different x locations. The distance is always the vertical projection of the points. To create a vertical dimension line, the *Toolidx* and *CurrentFigure* property should be set as follows.

```
        PowerCad1.ToolIdx := toFigure
        PowerCad1.CurrentFigure := 'TVDimLine';
```

To draw a vertical dimension line PowerCad needs three points. The first two points are the points that defines the measured distance and the third point defines the location where main line will be drawn. Powercad draws two bounding lines 2 mm apart from the distance points to the x location of the third point. The main line is drawn according to the Labeling style of the dimension line. A vertical dimension line can not be rotated or mirrored. This dimension line provides 7 styles of labeling as illustrated below.



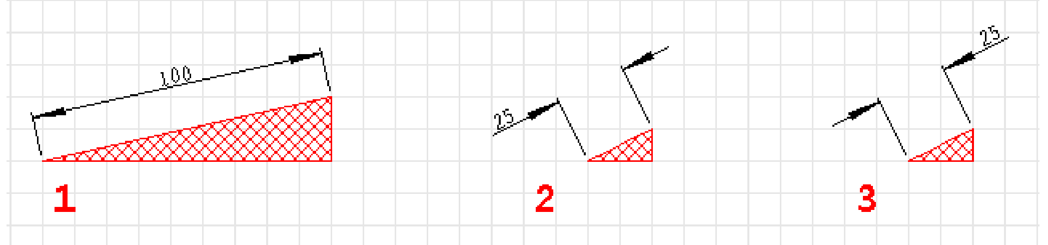
1. *Inside* **2.** *Top* **3.** *Bottom* **4.** *TopLeft* **5.** *TopRight* **6.** *BottomLeft* **7.** *BottomRight*

Aligned Dimension Lines

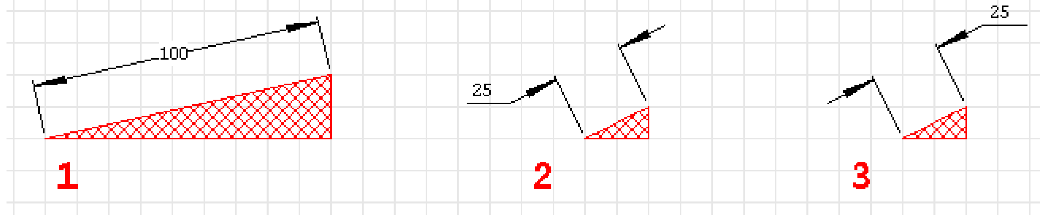
Shows the real distance between two points. To create an aligned dimension line, the *Toolidx* and *CurrentFigure* property should be set as follows.

```
      PowerCad1.ToolIdx := toFigure
      PowerCad1.CurrentFigure := 'TADimLine';
```

To draw an aligned dimension line PowerCad needs three points. The first two points are the points that defines the measured distance and the third point defines the location where main line will be drawn. Powercad draws two bounding lines 2 mm apart from the distance points to the location of the third point. The main line is drawn according to the Labeling style of the dimension line. An aligned dimension line **can be** rotated or mirrored. This dimension line provides 3 styles of labeling as illustrated below. But there is one more property AlwaysHorizontal that also defines the style of Labeling.



AlwaysHorizontal = *False* **Labeling = 1.** *Inside* **2.** *Left* **3.** *Right*



AlwaysHorizontal = *True* **Labeling = 1.** *Inside* **2.** *Left* **3.** *Right*

6 PowerCad Interface

In software engineering; the word ***Interface*** is used for anything which is in the middle of two things to provide the communication. You are familiar the word *"user interface"* which is used for the graphical controls that the user deals to communicate with the main software working in the background.

When we speak about the *interface of a class* we mean something same in function but different in form. The interface of the class is the members of the class that the programmer can access and make use of . These class members are properties and methods of the class. When you set a property of a class or make a call to a method, you make a change in the behaviour of the object or you make the object to behave in the way you want.

In PowerCad, the drawing is organized as Layers, and as the Figures that are related with these layers. So for the layer functionality, we have a class definition called "TLayer", and for the figures in these layers we have another class definition called "TFigure". In fact, TLayer and TFigure definitions have got factory privacy, and at the beginning, it is not intended to document these classes. But now, it is clear that using these classes directly will result more performance than using PowerCad methods.

6.1 Accessing Layers and Figures

You can access powercad database through the lists or by storing the handles returned by the creator methods. Also you can access the selected figures through the Selection list.

Using The Lists

The Layers are stored in the ***Layers*** list of the TPowerCad class. To get this list just use the statement below.

```
PowerCad1.Layers
```

You shouldn't make any modification on this list directly unless you are not a PowerCad expert. (This means don't add or remove items from this list). This list is just should be used for accassing the Layer objects. To get a layer from the list you should use the layer index. 0 is used fro the base layer and others so on.

```
PowerCad1.Layers[0]
```

But the above statement is just a pointer. To be able to use it as Layer you should typecast it.

```
Var
      LHandle: integer;
      MyLayer: Tlayer;
begin
      LHandle := PowerCad1.Layers[0];
      MyLayer := Tlayer(LHandle);
      MyLayer.Visible := seen;
      .
      .
 end;
```

The Figures are stored in the ***Figures*** list of the TPowerCad class. To get this list just use the statement below.

```
PowerCad1.Figures
```

You shouldn't make any modification on this list directly unless you are not a PowerCad expert. (This means don't add or remove items from this list). This list is just should be used for accassing the Figure objects. To get a figure from the list you should use the figure index.

```
PowerCad1.Figures[0]
```

But the above statement is just a pointer. To be able to use it as Figure you should typecast it.

```
Var
      FHandle: integer;
      MyFigure: TFigure;
begin
      FHandle := PowerCad1.Figures[0];
      MyFigure := TFigure(FHandle);
      MyFigure.Move(10,10);
      .
      .
      .
 end;
```

Using The Return Handles

All of the creator methods of the PowerCad returns the handle of the created object. If it is a Layer, the handle of the layer object is returned or if it is a figure the handle of the figure is returned.

You should also typecast the returning handle to its original class.

```
Var
      Lhandle,Fhandle: integer;
      MyLayer: Tlayer;
      MyFigure: TFigure;

Begin

      Lhandle :=  PowerCad1.NewLayer('MyLayer');
      MyLayer := Tlayer(Lhandle);

      FHandle := PowerCad1.Rectangle(…);
      MyFigure := TFigure(Fhandle);
      MyFigure.GetBounds(…);

End;
```

Using The Selection List

The selection list includes the handles of the selected figures. You can access the selected figures in this way.

```
Var
        MyFigure: TFigure;

Begin

        MyFigure := TFigure(PowerCad1.Selection[0]);
        MyFigure.GetBounds(…);

End;
```

6.2 "TFigure" Interface

TFigure is the base class for all figure classes. Normally through this class interface you can do all common operations. But figure specific operations (eg: To arrange the transparency of a Bitmap figure), should be handled through the interface of the class of that figure. In this section we will first see how to use the base class interface, and then how to use the derived classes' interfaces.

```
TFigure = class(TObject)
  Owner: TComponent;
  // The Owner is the PowerCad Object that involves the figure

  Name: String;
  // The name of the figure. You can set it as you want

  Handle: TFigHandle;
  // Handle is the pointer to the figure object itself. It is a
  // read only value.

  LayerHandle: LongInt;
  // The pointer to the layer object that the figure is related


  Width: integer;
  // The Pen Width

  Color: integer;
  // The Pen Color

  Style: integer;
  // The Pen Style

  RowStyle : Integer;
  // The row style. Applicable if the figure is a line or open
  // polyline.

  BrC,BrS : integer;
  // The Brush Color and Brush Style. Applicable if the figure
  // is a closed figure.

  PointCount: integer;
  // The number of the points used for drawing the figure
```

```
RegHandle : integer;
// The Handle to the Windows Region created for the figure.

Data: Pointer;
// A pointer to a user specified data for each figure.

Modified: Boolean;
// Used for internal purpose normally. Set this value to true
// if you want the figure to refresh itself.

Function Edit: Boolean;virtual;
// Calls the edit dialog of the figure if exists. Returns
// true if the user edits the figure.

Procedure GetBounds(var figMaxX,figMaxY,figMinX,figMinY:
                    integer);virtual;
// Gives the bounds of the figure.

Function GetBoundRect:TRect;virtual;
// Gives the bounds of the figure as a Trect.

Procedure Move(deltax, deltay: integer);virtual;
// Moves the figure in x and y direction

Procedure Rotate(aAngle: integer; cPoint: TPoint);virtual;
// Rotates the figure around cPoint with the angle aAngle.

Procedure Mirror(Point1,Point2: TPoint);virtual;
// Mirrors the figure relative to a line from point1 to
// point2.

Procedure Scale(percentx,percenty: integer;
                rPoint: Tpoint);virtual;
// Scales the figure with percentx in width and percenty in
// height according to the rPoint.

Function IsPointIn(x,y:integer): boolean; virtual;
// Returns true if the point(x,y) is in and/or on the figure.

Function CheckifInArea(area: TRect): boolean;
// Returns true if the figure is entirely in the given area.

Function Duplicate: TFigure; virtual;
// Duplicates the figure. The return value is the new figure.

Procedure Select;
// Selects the figure

Procedure Deselect;
// Deselects the figure

Procedure GetSourceText(var List: TStringList);virtual;
// Gives the source text of the figure, The list parameter
// should be created.

Procedure WriteToStream(Stream: TStream);virtual;
// Stores the figure data to the given stream.

Procedure Draw(DEngine:TPCDrawEngine;isFlue:Boolean);virtual;
// Draws the figure with the given Dengine. IsFlue specifies
// if it will be draw as grayed.
```

```
  Procedure DrawFigureGuides(DEngine: TPCDrawEngine);virtual;
  // Draws the guides of the figure. Such as a cross for the
  // center of a circle.

  Procedure DrawSelectionPoints(DEngine:TPCDrawEngine;
                              isFlue:Boolean);virtual;
  // Draws the selection points of the figure.

  Class Function CreateFromSource(lines: TStringList;
    LHandle:LongInt; aDrawStyle: TDrawStyle;
    aOwner: TComponent): TFigure;
  // Creates a figure with the given source text.

  Class Function CreateFromSStream(Stream: TStream;
    LHandle:LongInt; aDrawStyle: TDrawStyle;
    aOwner: TComponent): TFigure;
  // Creates a figure with the given stream.

  Property ControlPoints[Index : integer]:Tpoint;
  // Gives the control point with the Index. For the first one
  // use 1.Applicable in ellipse and bezier formed polyline.

  Property FigurePoints[Index : integer]:Tpoint;
  // Gives the figure point with the Index. For the first one
  // use 1.
end;
```

6.3 The Derived Figure Interfaces

Normally, almost all virtual functions of TFigure class are reimplemented in each derived figure class to provide figure specific behaviour. So normally the interface of a derived class is more crowded than the given interfaces below. The below interfaces are only the figure specific interface members, normally for a TLine class you can still use, for example, the Move method, but since they are explained above in the base class, below interfaces will not include the comman interface members.

6.3.1 "TLine" Interface

```
TLine = class(TFigure)
  Procedure SetJFigure1(jf:TFigure);
  // This method will join the first point of the line with the
  // given figure. If it is nill the existing bound will be
  // broken.

  Procedure SetJFigure2(jf:TFigure);
  // This method will join the second point of the line with
  // the given figure. If it is nill the existing bound will be
  // broken.

  Procedure UnBound;
  // This method will broken the bounds to the figures.

  constructor create( aX1,aY1,aX2,aY2,w,s,c: integer;
      row:integer;LHandle: LongInt; aDrawStyle: TDrawStyle;
      aOwner: TComponent);
   // The constructor:
   // aX1,aY1,aX2,aY2: The coordinates
   // w,s,c: pen width,style and color
   // row: The row style (0-6)
   // Lhandle: The handle to the layer it is related.
   // aDrawStyle: The DrawStyle. Use dsNormal for normal
   // drawings. dsTrace for creating temporary tracing figures.
   // aOwner: The Owner. Pass the PowerCad object.
end;
```

6.3.2 "TEllipse" Interface

```
TEllipse = class(TFigure)
  alen,blen : integer;
  // The vertical and horizontal radius of the ellipse

  constructor create(cX,cY,len1,len2,aAngle,w,s,c,abrs,abrc:
    integer; LHandle:LongInt; aDrawStyle: TDrawStyle;
    aOwner: TComponent);
  // The Constructor:
  // cX,cY: The Center point coordinates
  // len1,len2: the radius values
  // aAngle: the angle parameter in 1/10 degrees.
  // w,s,c: pen width,style and color
  // abrs,abrc : The brush style and color
  // Lhandle: The handle to the layer it is related.
  // aDrawStyle: The DrawStyle. Use dsNormal for normal
  // drawings. dsTrace for creating temporary tracing figures.
  // aOwner: The Owner. Pass the PowerCad object.
end;
```

6.3.3 "TCircle" Interface

```
TCircle = class(TFigure)
  radius : integer;
  // The radius of the circle

  constructor create(cX,cY,rad,w,s,c,abrs,abrc:
    integer; LHandle:LongInt; aDrawStyle: TDrawStyle;
    aOwner: TComponent);
  // The Constructor:
  // cX,cY: The Center point coordinates
  // rad: the radius
  // w,s,c: pen width,style and color
  // abrs,abrc : The brush style and color
  // Lhandle: The handle to the layer it is related.
  // aDrawStyle: The DrawStyle. Use dsNormal for normal
  // drawings. dsTrace for creating temporary tracing figures.
  // aOwner: The Owner. Pass the PowerCad object.
end;
```

6.3.4 "TArc" Interface

```
TArc = class(TFigure)
  radius : integer;
  // The radius of the arc circle

  ArcStyle: integer;
  // The style of the arc. 0:normal 1:Pie 2:Chord

  constructor create(cX,cY,rad,lx1,ly1,lx2,ly2,w,s,c,
        abrs,abrc,aArcSyle:integer;
        LHandle:LongInt; aDrawStyle: TDrawStyle;
        aOwner: TComponent);
  // The Constructor:
  // cX,cY: The Center point coordinates
  // rad: the radius
  // lx1,ly1,lx2,ly2: the line ends that forms the arc
  // w,s,c: pen width,style and color
  // abrs,abrc : The brush style and color
  // aArcStyle: The style of the arc
  // Lhandle: The handle to the layer it is related.
  // aDrawStyle: The DrawStyle. Use dsNormal for normal
  // drawings. dsTrace for creating temporary tracing figures.
  // aOwner: The Owner. Pass the PowerCad object.

  Procedure Invert;
  // Inverts the drawing direction of the arc.
end;
```

6.3.5 “TRectangle” Interface

```
TRectangle = class(TFigure)
  constructor create(aX1,aY1,ax2,ay2,w,s,c,
        abrs,abrc:integer;
        LHandle:LongInt; aDrawStyle: TDrawStyle;
        aOwner: TComponent);
  // The Constructor:
  // aX1,aY1,aX2,aY2: The corner point coordinates
  // w,s,c: pen width,style and color
  // abrs,abrc : The brush style and color
  // Lhandle: The handle to the layer it is related.
  // aDrawStyle: The DrawStyle. Use dsNormal for normal
  // drawings. dsTrace for creating temporary tracing figures.
  // aOwner: The Owner. Pass the PowerCad object.
end;
```

6.3.6 “TVertex” Interface

```
TVertex = class(TFigure)
  constructor create(aX,aY:integer;
        LHandle:LongInt; aDrawStyle: TDrawStyle;
        aOwner: TComponent);
  // The Constructor:
  // aX,aY: The point coordinates
  // Lhandle: The handle to the layer it is related.
  // aDrawStyle: The DrawStyle. Use dsNormal for normal
  // drawings. dsTrace for creating temporary tracing figures.
  // aOwner: The Owner. Pass the PowerCad object.
end;
```

6.3.7 “TPolyLine” Interface

```
TPolyline = class(TFigure)

  constructor create(Points: Array of TPoint;
    nbrPoint,w,s,c,abrs,abrc: integer;row:integer;
    aClosed: Boolean; LHandle: LongInt;
    aDrawStyle: TDrawStyle; aOwner: TComponent);

  // The Constructor:
```

```
  // Points: The point array that will be used to form the
  // polyline, the control points - that are used in bezier
  // forms - are not included in this array.
  // nbrPoint: the number of the points in the array
  // w,s,c: pen width,style and color
  // abrs,abrc : The brush style and color
  // row: the row style used at the ends (0-6)
  // aClosed: the closed property
  // Lhandle: The handle to the layer it is related.
  // aDrawStyle: The DrawStyle. Use dsNormal for normal
  // drawings. dsTrace for creating temporary tracing figures.
  // aOwner: The Owner. Pass the PowerCad object.

  Procedure SetJFigure1(jf:TFigure);
  // This method will join the first point of the line with the
  // given figure. If it is nill the existing bound will be
  // broken.

  Procedure SetJFigure2(jf:TFigure);
  // This method will join the second point of the line with
  // the given figure. If it is nill the existing bound will be
  // broken.

  Procedure UnBound;
  // This method will broken the bounds to the figures.

  Procedure ConvertToBezier;
  // This method converts all points to bezier points.The whole
  // polyline will behave in bezier form.

  Procedure ConvertToPolyLine;
  // This method converts all points to corner points.The whole
  // polyline will behave in linear form.

  Procedure ArrangePoint(ps: TBezierPoint; index:integer);
  // This method converts a specific point defined by index
  // parameter to the form defined by ps parameter.
  // psCorner: Linear corner
  // psCurve: Bezier form, the control line is tangent to the
  // curve.
  // psCurveCorner: Bezier form, the control line is cornered

  Procedure ArrangeSelPoint(ps: TBezierPoint);
  // This method converts the selected point of the
  // polyline to the form defined in ps parameter.

  Procedure GetControlPoints(index: integer;
            var cp1,cp2:TPoint);
  // This method gives the control points of a polyline point
  // if it is in bezier form.

  Procedure SetControlPoints(index: integer; cp1, cp2: TPoint);
  // This method sets the control points of a polyline point
  // if it is in bezier form

  Procedure SelectPoint(SeqNbr: Integer);
  // This method select the polyline point specified in SeqNbr
  // parameter.

  Property Closed: Boolean;
  // The polyline behaves like polygon when it is closed.
end;
```

6.3.8 "TBMPObject" Interface

```
TBMPObject = class (TFigure)
  Picture : TBitmap;
  // The Bitmap that is drawn. If you rset this field you
  // should set the modified property to true also.

  PictureName : string;
  // The file name of the bitmap when it is created from file.
  // You can set this field to any value you want

  ClipFigure: Tfigure;
  // If you want the bitmap to be clipped, set this field to
  // the clipping figure. To unclip, set this value to nil.

  constructor Create( x,y: integer; afName: string;
     LHandle:LongInt; aDrawStyle: TDrawStyle;
     aOwner: TComponent);

  // The Constructor:
  // x,y: The point coordinates that the bitmap will be located
  // afName: the file path to the bmp file on disk, that will
  // be inserted
  // Lhandle: The handle to the layer it is related.
  // aDrawStyle: The DrawStyle. Use dsNormal for normal
  // drawings. dsTrace for creating temporary tracing figures.
  // aOwner: The Owner. Pass the PowerCad object.

  constructor CreateEx( x,y: integer; aBitmap: TBitmap;
     LHandle:LongInt; aDrawStyle:TDrawStyle;
     aOwner: TComponent);
  // The Extended Constructor:
  // x,y: The point coordinates that the bitmap will be located
  // aBitmap: The bitmap object that will be inserted
  // Lhandle: The handle to the layer it is related.
  // aDrawStyle: The DrawStyle. Use dsNormal for normal
  // drawings. dsTrace for creating temporary tracing figures.
  // aOwner: The Owner. Pass the PowerCad object.

  Procedure FlipHorz;
  // Use this method to flip the bitmap horizontally.

  Procedure FlipVert;
  // Use this method to flip the bitmap vertically.

  Property Transparent: Boolean;
  // Use this property to control the transparency of the
  // bitmap.

end;
```

6.3.9 "TText" Interface

```
TText = class(TFigure)
  Color : TColor;
  // The color of the text

  Text: string;
  // The string that is drawn

  Font: TFont;
  // The font properties of the text. Not all fields active.
  // Use Font.Name, Font.Charset, Font.Style

  Height: integer;
  // The height of the font in PowerCad unit dmm (1/10 mm)

  wRatio: double;
  // The ratio of the width to the height. Default 0.8

  constructor Create( aX1,aY1,h:integer;ratio: double;
    atext: string; FontName:String; FontCharset: Byte;
    LHandle:LongInt; aDrawStyle: TDrawStyle;
    aOwner: TComponent);

  // The Constructor:
  // ax1,ay1: The coordinates that the text will be located
  // h: the height of the text in 1/10 mm unit.
  // ratio: the width ratio to the height of the text
  // aText: the text to be drawn
  // FontName: the name of the font of the text
  // FontCharset: The characterset of the font.
  // Lhandle: The handle to the layer it is related.
  // aDrawStyle: The DrawStyle. Use dsNormal for normal
  // drawings. dsTrace for creating temporary tracing figures.
  // aOwner: The Owner. Pass the PowerCad object.
end;
```

6.3.10 "TRichText" Interface

```
TRichText = class(TRectangle)
  re: TRichEdit98;
  // The Invisible RichEditControl 2.0. You can use this
  // control to access the rich text content

  MetaFile: TMetafile;
  // This is the metafile that the rich text is drawn on.

  constructor create(aX1,aY1,ax2,ay2,w,s,c,
        abrs,abrc:integer;
        LHandle:LongInt; aDrawStyle: TDrawStyle;
        aOwner: TComponent);
  // The Constructor:
  // aX1,aY1,aX2,aY2: The corner point coordinates
  // w,s,c: pen width,style and color
  // abrs,abrc : The brush style and color
  // Lhandle: The handle to the layer it is related.
  // aDrawStyle: The DrawStyle. Use dsNormal for normal
  // drawings. dsTrace for creating temporary tracing figures.
  // aOwner: The Owner. Pass the PowerCad object.
  // aDrawStyle: The DrawStyle. Use dsNormal for normal
  // drawings. dsTrace for creating temporary tracing figures.
  // aOwner: The Owner. Pass the PowerCad object.
end;
```

6.3.11 "TOLEObject" Interface

```
TOLEObject = class(TRectangle)
  ole: TOleContainer;
  // The Invisible Ole Container control. You can use this
  // control to access the ole object content

  MetaFile: TMetafile;
  // This is the metafile that the ole object is drawn on.

  constructor create(aX1,aY1,ax2,ay2,w,s,c,
        abrs,abrc:integer;
        LHandle:LongInt; aDrawStyle: TDrawStyle;
        aOwner: TComponent);
  // The Constructor:
  // aX1,aY1,aX2,aY2: The corner point coordinates
  // w,s,c: pen width,style and color
  // abrs,abrc : The brush style and color
  // Lhandle: The handle to the layer it is related.
  // aDrawStyle: The DrawStyle. Use dsNormal for normal
  // drawings. dsTrace for creating temporary tracing figures.
  // aOwner: The Owner. Pass the PowerCad object.
  // aDrawStyle: The DrawStyle. Use dsNormal for normal
  // drawings. dsTrace for creating temporary tracing figures.
  // aOwner: The Owner. Pass the PowerCad object.
end;
```

6.3.12 “TFigureGrp” Interface

```
TFigureGrp = class(TFigure)
  InFigures: TList;
  // The figures that are in the group

  Combined: Boolean;
  // The group will behave combined when this field is true

  Procedure UnGroup;
  // Use this method to ungroup a group

  Procedure AddFigure(fig: TFigure);
  // Add figure to the group

  constructor create(LHandle: LongInt;aOwner: TComponent);
  // The Constructor:
  // Lhandle: The handle to the layer it is related.
  // aOwner: The Owner. Pass the PowerCad object.

end;
```

6.3.13 “TBlock” Interface

```
TBlock = class(TFigureGrp)
  Blockname: String;
  // The block name of the block

  constructor create(LHandle: LongInt;aOwner: TComponent);
  // The Constructor:
  // Lhandle: The handle to the layer it is related.
  // aOwner: The Owner. Pass the PowerCad object.

end;
```

7 PowerCad in Background: Using DrawEngine Directly

In some applications, the Cad functionality is embedded in the applications modules and the behaviours are handled in the style of that application. So in these cases the developer doesn't need the PowerCad editor on his forms, but he needs the CAD functionality such as creating figures through code, making use of them in someway and finally drawing them on an applicaton defined Canvas (Device Context Wrapper Class in Delphi).

There are several ways to handle Cad in background but in any case you will need to create the object database (the figures that will be drawn). So firstly let us show the ways to create an object database.

7.1 Using the "Invisible" Way

The first way is using TPowerCad as an invisible Cad controller. In this case, the PowerCad control will be existing in the background but it will be used only an object database, and the figures will be drawn to a different deice context (dc).

Now let us remember that we should include some units in our uses clause.

```
Uses PowerCad, DrawObjects, DrawEngine, PCTypesUtils;
```

And, we should create the PowerCadControl which will be in the background.

```
Var
CadControl: TPowerCad;
.
.
CadControl := TPowerCad.Create(Form1);
CadControl.Visible := false;
```

Now, we have got the control in the background and we know that we have a Base Layer (LayerNumber = 0) in the control by default. If we want more layers we can create by layer methods as described before.

So now, we can create our figures by calling the proper methods of PowerCad.

```
CadControl.Rectangle(...);
CadControl.Circle(...);
CadControl.Ellipse(...);
CadControl.InsertPicture(...);
```

The final step is of course, drawing the figures on an application defined canvas. So let us remember that Coordinates in Cad-Drawing is a relatively interpreted case, this means; if you want draw something on somewhere you should give a method to the drawengine , a method that will show the way how the coordinates will be interpreted. Normally PowerCad uses a dmm based unit system, and user specified coordinate origin. But this is the system used by PowerCad when it handles the drawing by itself. So in custom canvas drawing you are not limited with PowerCad's coordinate system, you can use any of the systems unless you know what you do.

So as a result, one main rule about the Coordinate System: ***You should be consistent with the coordinates you give the figure drawing methods, when you specify the method of interpreting the coordinates.*** Of course the easiest way is to define the coordinates as pixel units of the windows (1 unit is one pixel, the origin is top left, the y value is positive to bottom), if your device context is a windows dc as a form canvas. In this case you shouldnt do anything extra in interpretter methods by leaving them blank.

So, how we will specify the interpreter methods. Everything is in DrawEngine object. You shuld create a drawEngine object and give all information about your canvas in this.

```
Var
MyEngine: TPCDrawEngine;
.
.
.
MyEngine := TPCDrawEngine.Create;

// First Let us give the Canvas
MyEngine.Canvas := Form1.Canvas;

// And the interpreter methods
MyEngine.ConvertPoint := MyPointConvertProc;
MyEngine.ConvertLen := MyLenConvertProc;
.
.

Procedure TForm1.MyPointConvertProc(var x,y:integer),
Begin
  // here you should write your conversion code for
  // points
  // the coming parameters are the coordinates that are
  // specified in the figure methods.
End;

Procedure TForm1.MyLenConvertProc(var Dim:integer),
Begin
  // here you should write your conversion code for
  // dimensions such as radius of a circle
  // the coming parameter is the dimension that are
  // specified in the figure methods.
End;
```

After the DrawEngine is specified, now you can draw the figures by calling their draw methods.

```
Var
MyFigure: TFigure;
i: integer;
Begin
  For i := 0 to CadControl.Figures.Count-1 do
  Begin
    MyFigure := TFigure(CadControl.Figures[ i ]);
    MyFigure.Draw(MyEngine,false (*not flue*));
  End;
End;
```

7.2 Using the "User" way

In this way, you shouldnt use the PowerCad as a background cad controller. Because as you see above, PowerCad is used just for storing the objects. So to have more performance, you can create the figure by your on, put them in your own user list, and draw them to your canvas with a draw engine. This way will be a little different then the first one, and so I will ony show the different steps.

First Create your list ,create the figures and store them in your list.

```
Var
MyFigures: TList;
MyFigure: Tfigure;
.
.
.
MyFigures := Tlist.Create;

// Create a rectangle and add it to the list

MyFigure := TRectangle.Create(…);
MyFigures.Add(MyFigure);

// Create a circle and add it to the list

MyFigure := TCircle.Create(…);
MyFigures.Add(MyFigure);
```

And now you should handle creating the DrawEngine as it is described in the first way. For drawing the figures we will use our own list now.

```
Var
MyFigure: TFigure;
i: integer;
Begin
  For i := 0 to MyFigures.Count-1 do
  Begin
    MyFigure := TFigure(MyFigures[ i ]);
    MyFigure.Draw(MyEngine,false(*not flue*));
  End;
End;
```

7.3 Two Examples for interpreter methods of DrawEngine

In this section we will show 2 examples for the coordinate system interpreting in DrawEngine object.

7.3.1 The First

In the first xample let us assume that, we want to use the original windows coordinate system. That is;

1 unit = 1 pixel
origin = top left
y direction = positive to bottom

You know this and use it when you make drawings on windows device context.

When creating our figures we will give the coordinates according to this system.

```
CadControl.Circle( 100,100,50,…);
```

The circle center is in Point (100,100) and the radius is 50.

The figure coordinates are specified in Windows default and so the interpreter methods will be left blank.

```
Procedure TForm1.MyPointConvertProc(var x,y:integer),
Begin
End;

Procedure TForm1.MyLenConvertProc(var Dim:integer),
Begin
End;
```

7.3.2 The Second

In the second example let us assume that, we want to use a metric measurement system. That is;

1 unit = 1 mm
origin = bottom left
y direction = positive to top

When creating our figures we will give the coordinates according to this metric system.

```
CadControl.Circle( 50,50,25,…);
```

The Circle center is in Point (50,50) (50 mm from bottom and 50 mm from left) and the radius is 25 mm .

The figure coordinates are specified in metric system and so we should decide how we will interpret 1 mm to pixel. Let us assume that for one millimeter we use 4 pixels.

```
Procedure TForm1.MyPointConvertProc(var x,y:integer),
Begin
  x := x * 4;
  y := y * 4;
  // the y value is considered to be positive to top and
  // the origin is bottom left. So the y value should be
  // converted to windows default origin
  y := form1.height - y;
End;

Procedure TForm1.MyLenConvertProc(var Dim:integer),
Begin
  Dim := Dim * 4;
End;
```

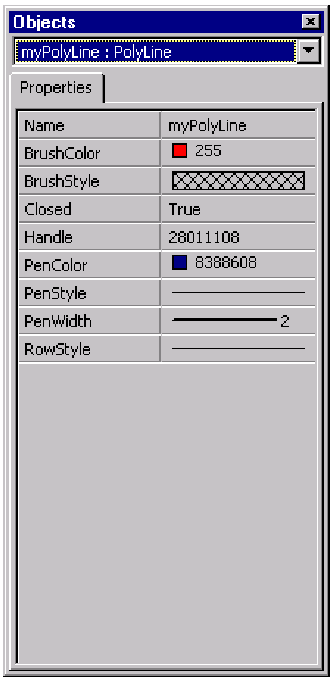
We can also use a zoom factor. So the value 4 is for the actual size of the drawing .It can be zoomed with a zoom factor. For a 50% view the zoom factor will be 0.5 and for a 200% view the zoom factor will be 2.

```
Procedure TForm1.MyPointConvertProc(var x,y:integer),
Begin
  x := x * 4 * ZoomFactor;
  y := y * 4 * ZoomFactor;
  y := form1.height - y;
End;

Procedure TForm1.MyLenConvertProc(var Dim:integer),
Begin
  Dim := Dim * 4 * ZoomFactor;
End;
```

8 PowerCad Skin

The PowerCad is designed to be used in every type of applications which need CAD capability. So each application has its own style of interface, and developers mostly want to hide the general well-known components in their own interface. This is because of the need of originality of an application. So with the use of standart interfaces, PowerCad also should provide developers a communicaton level with their own interface. This means PowerCad is able to communicate with the dialogs,toolbars or any other interface element of the application while it is also providing a standart interface. In this chapter you will see how the standart interface is used and next chapter , you will see customization of the Interface.

8.1 The Object Inspector (Properties Window)



myPolyLine : PolyLine

Name	myPolyLine
BrushColor	🟥 255
BrushStyle	▨
Closed	True
Handle	28011108
PenColor	🟦 8388608
PenStyle	
PenWidth	2
RowStyle	

The Object Inspector is designed to modify the properties of the drawen objects. The objects (with its properties) are registered to the Object Inspector in the runtime. Each property of an object has a Property Type; and according to this Type an editor is provided in the Object Inspector for that property. When a property is registered as ReadOnly, then it is locked to the user updates through the Inspector.

Each object (figure) in PowerCad has got a Name and Handle Property by default. The name property can be edited through the Inspector while the Handle property is read-only. The Handle property gives the handle of a figure and you have seen what can be done with a figue handle, int the previous chapters.

The property types in the object inspector can be listed as follows.

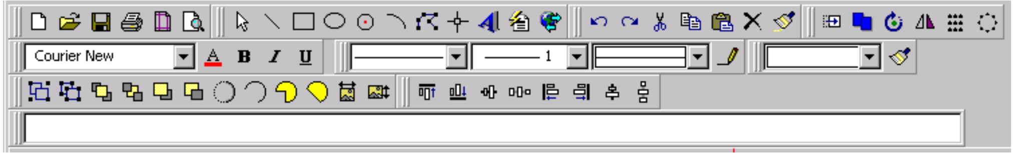
1. String Property
2. Integer Property
3. Extended (Real Numbers) Property
4. Boolean Property
5. Enum (Pick List) Property
6. Color Property
7. Font Property
8. Line Width Property
9. Line Style Property
10. Brush Style Property
11. FileName Property

For each of the below types, the Object Inspector provides a special editor which is sometimes an edit box, sometimes a combobox, sometimes a popup form.

8.2 ToolBars

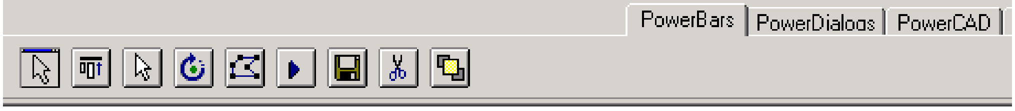
The toolbars in the PowerCad package are visual wrappers for PowerCad methods. Each button click results a method call in PowerCad. The toolbars are registered to the PowerCad so that they are aware of PowerCad changes. This means when something changes in PowerCad this toolbars are synchronized with PowerCad current status. *For example* when the *select* tool button is down in *Tools* Bar, if the user clicks on the *Rotate* button of the *Transform* Bar, the *Tools* Bar is notified by this change and so it makes the down(selected) button up.

The ToolBars Together



The PowerCad ToolBars , PowerBars, are dockable bars. They are placed on a TPCDock which can be aligned to top,bottom,left or right side of the form. So to use a PowerBar, there should be a TPCDock on the form. After you place the PowerBar on the PCDock, there is only one more thing to be able to use it. Set the CadControl Property of the ToolBar to the PowerCad control on your form. This will provide the communication between the Powerbar and powerCad control.

The PowerBars Tab on Delphi Component Bar

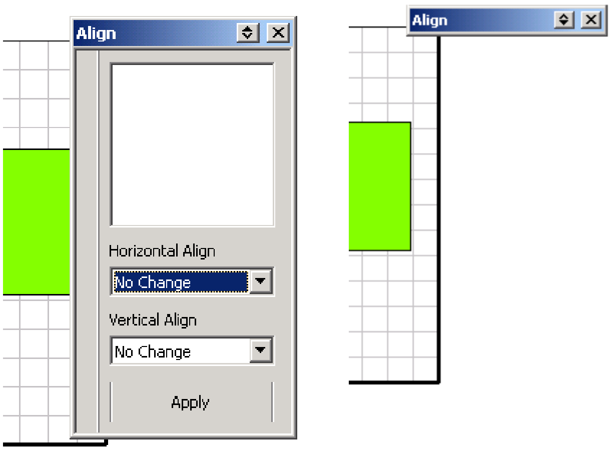


In Left to Right Order : PCDock, PCAlignBar, PCToolsBar, PCTransformBar, PCModifybar, PCCommandBar, PCFileBar, PCEditBar, PCArrangeBar.

8.3 Dialogs

Dialogs of PowerCad provides interfaces for arranging insider options of powercad that also can be doen through property assignments and method calls. The PowerCad dialogs are registered to PowerCad so that they are syncronized with the changes of Powercad.

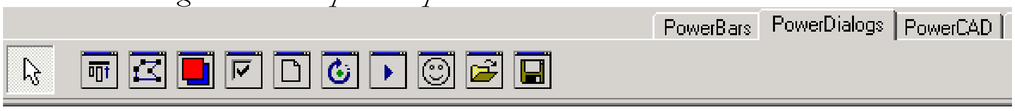
The PowerCad dialogs, PowerDialogs are invisible componets on the form, when their show methods are called, the form (interface) of the dialog is opened. Except two dialogs (PCSaveDialog and PCOpenDialog), the PowerDialogs are non-modal dialogs, this means, you can still do your work on the form when the dialog is open.



Horizontal Align
No Change
Vertical Align
No Change
Apply

The PowerDialogs can be visible (when the show method is called), invisible (when it is closed by its close button) or floating minimized (when it is minimized by its minimize button).

The PowerDialogs Tab on Delphi Component Bar



In Left to Right Order : PCAlignDlg, PCModifyDlg, PCLayerDlg, PCOptionsDlg, PCPageDlg, PCTransformDlg, PCMacroDialog, PCBlockDlg, PCOpenDialog, PCSaveDialog.

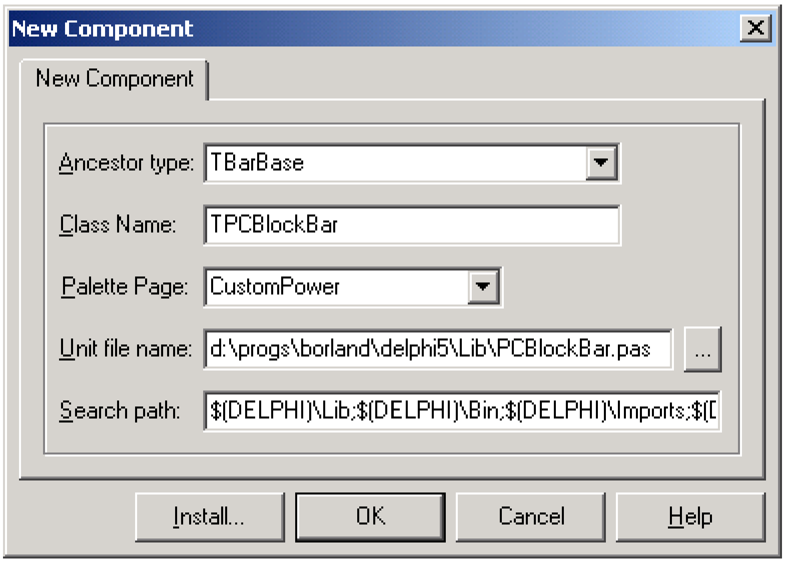
9 Make your own ToolBars and Dialogs

9.1 Making Custom ToolBars

As Declared before PowerBars are registered to the PowerCad Control when you set the CadControl property of any of them. This provides the communication of the Toolbar and the PowerCad. This communication is a two-way communication: The toolbar sends the user actions to the PowerCad so that PowerCad is effected, and PowerCad sends the change notifications to the toolbar so that the ToolBar is effected. The toolbar effects Powercad by setting its properties and calling its methods, and PowerCad effects the toolbar by calling toolbar's syncronize method so that the toolbar redesigns itself in this method. This method can be considered as an OnChangeEvent procedure.

To make a registered toolbar, you should inherit a newBar from TBarBase class. Because this base class encapsulates the necessary structure for the two-way communication. In this chapter we will make a Cutom toolbar for inserting blocks to PowerCad.

Step 1 : First open the New Component Dialog from Delphi Component Menu. Select the TBarBase as Ancestor type. Write the class name as TPCBlockbar. And select a palette page as you want.



New Component

Ancestor type: TBarBase

Class Name: TPCBlockBar

Palette Page: CustomPower

Unit file name: d:\progs\borland\delphi5\Lib\PCBlockBar.pas

Search path: $(DELPHI)\Lib;$(DELPHI)\Bin;$(DELPHI)\Imports;$(

Step 2: Delphi will generate a new unit for us an will show it on the editor when we click on the OK button of the below dialog. Above is a complete listing of the generated unit.

Listing 1 – Generated Unit

```
unit PCBlockBar;

interface

uses
Windows, Messages, SysUtils, Classes, Graphics, Controls,
Forms, Dialogs,PCMSBar, BarBase;

type
  TPCBlockBar = class(TBarBase)
  private
    { Private declarations }
  protected
    { Protected declarations }
  public
    { Public declarations }
  published
    { Published declarations }
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('CustomPower', [TPCBlockBar]);
end;

end.
```

Step 3: Add the `PowerCad,PcTypesUtil,Buttons` units to the **uses** clause.

Listing 2 – The Uses Clause

```
uses
Windows, Messages, SysUtils, Classes, Graphics, Controls,
Forms, Dialogs,PCMSBar, BarBase,PowerCad,PcTypesUtil,Buttons;
```

Step 4: Now we will implement a new constructor. In this constructor we will create a list that will be used for the buttons created for each block in the block directory. Also for deleting this list we will implemet a destructor. Listing 2 and Listing 3-4 shows how the constructor and destructor is reimplemented.

Listing 3 – The class interface with Constructor and Destructor

```
  TPCBlockBar = class(TBarBase)
  private
    { Private declarations }
    FButtons: TList;
  protected
    { Protected declarations }
  public
    { Public declarations }
    constructor Create(AOwner: TComponent);override;
    destructor destroy;override;
  published
    { Published declarations }
  end;
```

Listing 4 – The implementation of Constructor and Destructor

```

constructor TPCBlockBar.Create(AOwner: TComponent);
begin
  inherited;
  FButtons := TList.Create;
end;

destructor TPCBlockBar.destroy;
var i: integer;
begin
  for i := 0 to FButtons.Count-1 do
    TSpeedButton(FButtons[i]).Free;
  FButtons.Free;
  inherited;
end;
```

Step 5: In this step, we will create the buttons, but since we will create one button for each block of the BlockDirectory of the PowerCad, we should know the value of the CadControl property is assigned. So in the run time of the constructor this property is nill. We will create the buttons when the user sets the CadControl property so we will override the DoSetControl method of TBarBase. Also we need an extra procedure for the OnClick Events of the created buttons. Listing 5-6 shows how DoSetControl and ButtonClick procedure is implemented.

Listing 5 – The Interface of the class with the DoSetControl and ButtonClick procedure

```
  TPCBlockBar = class(TBarBase)
  ...
  private
    ...
    Procedure DoSetControl(Control:TPowerCad);override;
    Procedure ButtonClick(Sender:TObject);
  end;
```

Listing 6 – Implementation of the DoSetControl and ButtonClick procedure

```
procedure TPCBlockBar.DoSetControl(Control: TPowerCad);
var BlockList:TStringList;
    i: integer;
    xButton:TSpeedButton;
begin
  inherited;
  for i := 0 to FButtons.Count-1 do
    TSpeedButton(FButtons[i]).Free;
  FButtons.Clear;
  if assigned(CadControl) then
  begin
     BlockList := TStringlist.Create;
     CadControl.GetBlockFileNames(BlockList);
     For i := 0 to BlockList.Count-1 do
     begin
       xButton := TSpeedButton.Create(Self);
       with xButton do
       begin
         parent := self;
         Groupindex := 1;
         Allowallup := true;
         Showhint := true;
         Hint := BlockList[i];
         Caption := 'B'+inttostr(i+1);
         OnClick := ButtonClick;
       end;
       FButtons.Add(xButton);
     end;
     BlockList.Free;
  end;
end;

procedure TPCBlockBar.ButtonClick(Sender: TObject);
var xButton:TSpeedButton;
    bName: String;
begin
  xButton := Sender as TSpeedButton;
  if assigned(CadControl) then
  begin
    bName := xButton.Hint;
    if fileexists(bName) then
    begin
      CadControl.CurrentBlock := bName;
      CadControl.ToolIdx := toInsertCurrentBlock;
    end;
  end;
end;
```

Step 6: This Block bar can do its job, now. When the user clicks on a button, the toolidx of the Powercad will be toInsertCurrentBlock and when the user clcik on PowerCad the block will be inserted to the clicked location. But we will do one more thing. Suppose that user has clicked on a button and so the clicked button is down. But after this click again suppose that user has clicked on the select button of the PCToolsBar. So in this case, we should make the our down button up in the Blockbar. We know that for each change in PowerCad, the Syncronize method of the bars are called so that if we reimplement the Syncronize method we can handle this case. Listing 7-8 shows the reimplentation of the Syncronize method.

Listing 7 – Class Interface with Syncronize Method

```
  TPCBlockBar = class(TBarBase)
  ...
  public
    ...
    Procedure Syncronize;override;
  end;
```

Listing 8 – Implementation of the Syncronize Method

```
procedure TPCBlockBar.Syncronize;
var i: integer;
begin
  inherited;
  if assigned(CadControl) then
  begin
    if CadControl.ToolIdx <> toInsertCurrentBlock then
    begin
      for i := 0 to FButtons.Count-1 do
        TSpeedButton(FButtons[i]).Down:= False;
    end;
  end;
end;
```

Step 7: Now Install this unit as a New Component from Install Component Menu. Listing 9 shows the full source code of the unit.

Listing 9 – Full Source code of the PCBlockbar.pas unit

```
unit PCBlockBar;

interface

uses
Windows, Messages, SysUtils, Classes, Graphics, Controls,Forms,
Dialogs,PCMSBar, BarBase,PowerCad,Buttons,PCTypesUtils;

type
  TPCBlockBar = class(TBarBase)
  private
    { Private declarations }
    FButtons: TList;
    Procedure ButtonClick(Sender:TObject);
    Procedure DoSetControl(Control:TPowerCad);override;
  protected
    { Protected declarations }
  public
    { Public declarations }
    constructor Create(AOwner: TComponent);override;
    destructor destroy;override;
    Procedure Syncronize;override;
  published
    { Published declarations }
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('CustomPower', [TPCBlockBar]);
end;

{ TPCBlockBar }

constructor TPCBlockBar.Create(AOwner: TComponent);
begin
  inherited;
  FButtons := TList.Create;
end;

destructor TPCBlockBar.destroy;
var i: integer;
begin
  for i := 0 to FButtons.Count-1 do
    TSpeedButton(FButtons[i]).Free;
  FButtons.Free;
  inherited;

end;

procedure TPCBlockBar.DoSetControl(Control: TPowerCad);
```

```
var BlockList:TStringList;
    i: integer;
    xButton:TSpeedButton;
begin
  inherited;

  for i := 0 to FButtons.Count-1 do
    TSpeedButton(FButtons[i]).Free;
  FButtons.Clear;
  if assigned(CadControl) then
  begin
     BlockList := TStringlist.Create;
     CadControl.GetBlockFileNames(BlockList);
     For i := 0 to BlockList.Count-1 do
     begin
       xButton := TSpeedButton.Create(Self);
       with xButton do
       begin
         parent := self;
         Groupindex := 1;
         Allowallup := true;
         Showhint := true;
         Hint := BlockList[i];
         Caption := 'B'+inttostr(i+1);
         OnClick := ButtonClick;
       end;
       FButtons.Add(xButton);
     end;
     BlockList.Free;
  end;
end;

procedure TPCBlockBar.Syncronize;
var i: integer;
begin
  inherited;
  if assigned(CadControl) then
  begin
    if CadControl.ToolIdx <> toInsertCurrentBlock then
    begin
      for i := 0 to FButtons.Count-1 do
        TSpeedButton(FButtons[i]).Down:= False;
    end;
  end;
end;

procedure TPCBlockBar.ButtonClick(Sender: TObject);
var xButton:TSpeedButton;
    bName: String;
begin
  xButton := Sender as TSpeedButton;
  if assigned(CadControl) then
  begin
    bName := xButton.Hint;
    if fileexists(bName) then
    begin
      CadControl.CurrentBlock := bName;
      CadControl.ToolIdx := toInsertCurrentBlock;
    end;
  end;
end;
end.
```

9.2 Making Custom Dialogs

PowerCad standart dialogs (PowerDialogs) are invisible components, that encapsulates a form object in it. The dialogs are registered to the PowerCad when their CadControl property is assigned a value. This regsitration provides the two-way communication between the PowerCad and the dialog. In this section we will see how we can make a standart dialog bu inheriting it from the base dialog class TDlgBase.

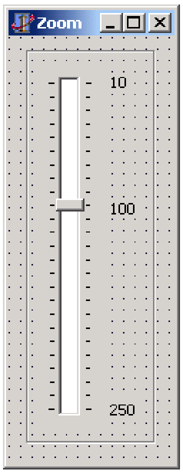
In fact you can provide a dialog fucntionality just by designing a form, and you can code specific powercad calls for specific user actions with controls on your form. And you can refresh your form values with the PowerCad values when its changed. You can be notified about this change by either a specific PowerCad event in which you are inetrested or the general change event onSycnronize. Yes, this a dialog. Why not?

So what does it provide us, to use the standart way, and develop our dialogs by inheriting the registered dialog class. This doesn't provide more functionality, the thing is that, it is more general, more comman and a standart way that can be documented.

In this section we will develop a new Dialog for Zooming the drawing with a windows TrackBar. This development will be done in two phase. First we will develop a form that includes a trackbar and fires an event procedure when the Trackbar Position changes. In the second phase we will develop our invisible dialog component that will encapsulate this form.

9.2.1 Designing the Dialog form

It is about your styles how you design a good looking form in Delphi, so we will not here force you to a standart way. The only thing you should be bound to is that you should provide an interface to the dialog component. In this example we are just interested with the position of the control. So we will provide a method for someone (the dialog class is the "someone" here) to set our trackbar's position, and we will provide a form property which is type of a callback function for transferring the trackbar changes to the dialog component. And to make our dialog form to be able to floating minimized, we will create a FormRoller in the constructor.



Step1: Create a new form. Put a TTrackBar on it. Set the Min property to 1, Max property to 25 and Position property to 10. You can put some other things to make the form look better. Name the form as ZoomForm. Make its FormStyle property fsStayOnTop.

Step2: In the OnCreate event of the form create the PCFormRoller and Make it Enabled.

Listing 1 – OnCreate Event Procedure of the form

```
procedure TZoomForm.FormCreate(Sender: TObject);
var wr : TPCRoller;
begin
  wr := TPCRoller.create(self);
  wr.Enabled := true;
end;
```

Step3: Include the unit PCFormRoll in the uses clause.

Listing 2- Uses Clause

```
uses
Windows, Messages, SysUtils, Classes, Graphics, Controls,
Forms, Dialogs,StdCtrls, ExtCtrls, ComCtrls,PCFormRoll;
```

Step4: Define a new procedural type; TChangeEvent, and a form property of this type; FChangeEvent. This event will be called when the trackbar positon changes.

Listing 3 – Change Event

```
Type
TChangeEvent = procedure(Position: integer) of object;

type
  TZoomForm = class(TForm)
  ...
  public
    FChangeEvent: TChangeEvent;
  end;
```

Step5: Call the FChangeEvent (if it is assigned) in OnChange Event of the TrackBar.

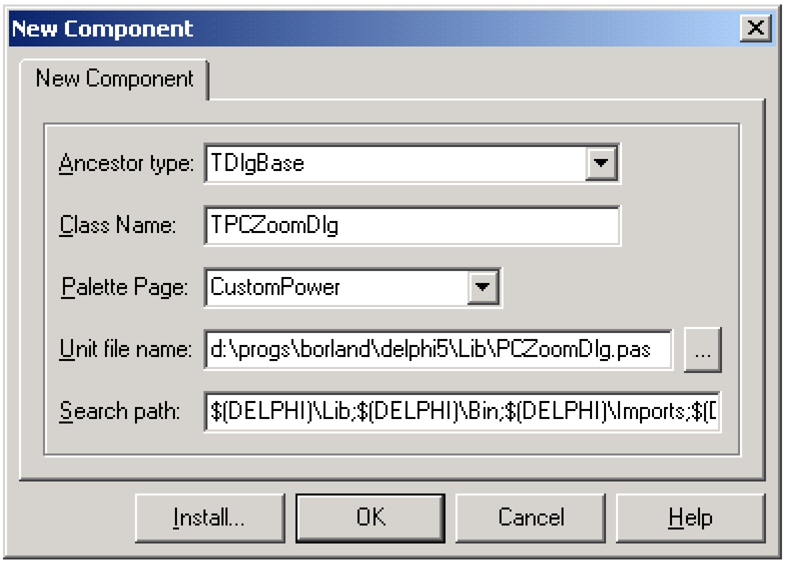
Listing 4 – Trackbar Change

```
procedure TZoomForm.TrackBar1Change(Sender: TObject);
begin
  if assigned(FChangeEvent) then
    FChangeEvent(TrackBar1.Position);
end;
```

Step6: Save this unit as frmZoom.

9.2.2 Designing the dialog component

Step 1 : First open the New Component Dialog from Delphi Component Menu. Select the TDlgBase as Ancestor type. Write the class name as TPCZoomDlg. And select a palette page as you want.



New Component

Ancestor type: TDlgBase

Class Name: TPCZoomDlg

Palette Page: CustomPower

Unit file name: d:\progs\borland\delphi5\Lib\PCZoomDlg.pas

Search path: $(DELPHI)\Lib;$(DELPHI)\Bin;$(DELPHI)\Imports;$(

Step 2: Delphi will generate a new unit for us an will show it on the editor when we click on the OK button of the below dialog. Above is a complete listing of the generated unit.

Listing1 – The generated Unit: PCZoomDlg.pas

```
unit PCZoomDlg;

interface

uses
Windows, Messages, SysUtils, Classes, Graphics, Controls,
Forms, Dialogs,DlgBase;

type
  TPCZoomDlg = class(TDlgBase)
  private
    { Private declarations }
  protected
    { Protected declarations }
  public
    { Public declarations }
  published
    { Published declarations }
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('CustomPower', [TPCZoomDlg]);
end;

end.
```

Step 2: We will use the dialog form that we have constructed in the first phase. So we will add the unit reference to the uses clause as in Listing 2.

Listing 2 – The Uses Clause

```
uses
Windows, Messages, SysUtils, Classes, Graphics, Controls,Forms,
Dialogs,DlgBase,frmZoom;
```

Step 3: First redefine the CadControl in published section, because in the base class it is in the protected section for technical reasons. Then define a private field for the dialog form named DialForm from TZoomForm type. Also define a procedure for getting the Trackbar changes in the form. And in the constructor we will create the form and we will set the event procedure of the form with our procedure.Listing 3-4 shows the class inetrface and the constructor implementation.

Listing 3 – The Class Interface

```
  TPCZoomDlg = class(TDlgBase)
  private
    { Private declarations }
    DialForm: TZoomForm;
    Procedure TrackChange(Position: integer);
  protected
    { Protected declarations }
  public
    { Public declarations }
    Constructor Create(aOwner:TComponent);override;
  published
    { Published declarations }
    Property CadControl;
  end;
```

Listing 4 – The Constructor Implementation

```
constructor TPCZoomDlg.Create(aOwner: TComponent);
begin
  inherited;
  DialForm := TZoomForm.Create(self);
  DialForm.FChangeEvent := TrackChange;
  DlgName := 'Zoom';
end;
```

Step 4: Implement the TrackChange Procedure as to pass the trackbar position to the CadControl (PowerCad) scale property. Not that this procedure will always called when the Trackbar changes in the dialog form.

Listing 5- The TrackChange Procedure

```
procedure TPCZoomDlg.TrackChange(Position: integer);
begin
  if assigned(CadControl) then
    CadControl.Scale := Position * 10;
end;
```

Step 5: Override the Show and Syncronize method. In Show proecure we will call the forms show method and in the syncronize procedure we will set the trackbar position with the cadcontrols scale value. Not that the syncronize method will be called by PowerCad in each change. Lisiting 6-7 shows the class interface for the overriden methodsa and their implementations.

Listing 6 – Class Inetrface for Show and Syncronize Method

```
  TPCZoomDlg = class(TDlgBase)
  private
  ...
  public
    Procedure Show;override;
    Procedure Syncronize;override;
  end;
```

Listing 7 – Impelementation of Show and Syncronize Method

```
procedure TPCZoomDlg.Syncronize;
begin
  inherited;
  if (DialForm.Visible) and (assigned(CadControl)) then
  begin
    DialForm.TrackBar1.Position := CadControl.Scale div 10;
  end;
end;

procedure TPCZoomDlg.TrackChange(Position: integer);
begin
  if assigned(CadControl) then
```

```
    CadControl.Scale := Position * 10;
end;
```

Step 7: Now Save and Install this unit as a New Component from Install Component Menu. Listing 8 shows the full source code of the unit.

Listing 8 – Full Unit Source Code

```
unit PCZoomDlg;

interface

uses
Windows, Messages, SysUtils, Classes, Graphics, Controls,Forms,
Dialogs,DlgBase,frmZoom;

type
  TPCZoomDlg = class(TDlgBase)
  private
    { Private declarations }
    DialForm: TZoomForm;
    Procedure TrackChange(Position: integer);
  protected
    { Protected declarations }
  public
    { Public declarations }
    Constructor Create(aOwner:TComponent);override;
    Procedure Show;override;
    Procedure Syncronize;override;
  published
    { Published declarations }
    Property CadControl;
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('CustomPower', [TPCZoomDlg]);
end;

{ TPCZoomDlg }

constructor TPCZoomDlg.Create(aOwner: TComponent);
begin
  inherited;
  DialForm := TZoomForm.Create(self);
  DialForm.FChangeEvent := TrackChange;
  DlgName := 'Zoom';
end;

procedure TPCZoomDlg.Show;
begin
  inherited;
  DialForm.Show;
  Syncronize;
end;

procedure TPCZoomDlg.Syncronize;
begin
```

```
  inherited;
  if (DialForm.Visible) and (assigned(CadControl)) then
  begin
    DialForm.TrackBar1.Position := CadControl.Scale div 10;
  end;
end;

procedure TPCZoomDlg.TrackChange(Position: integer);
begin
  if assigned(CadControl) then
    CadControl.Scale := Position * 10;
end;
end.
```

10 Using and Making Blocks

Blocks are most essential elements of a drawing environment. They are used to build up an available figure library that will be inserted to the drawings in any time. The user can store frequently used drawing elements as blocks, and can insert it to the drawings later.

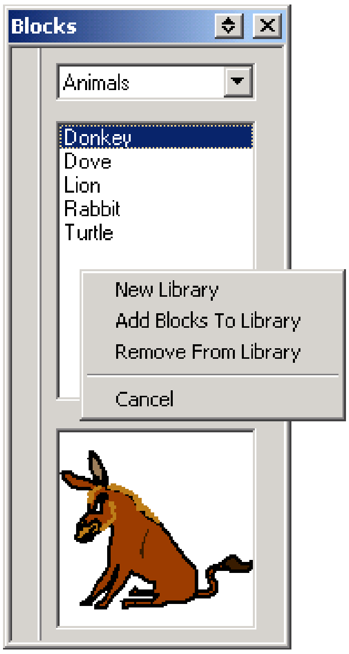
In PowerCad a Block is registered FigureGroup which includes the basig figures in itself. These blocks in PowerCad can be inserted in either through the dialog interface or by calling the appropriate methods.

10.1 The Block Directory

In fact, PowerCad can insert blocks from any location as far as its path is given. So it sound unneccassary to have a strict blcok directory for PowerCad, but for PowerCad, the block directory is a ‘must be’ information since it is used in Block Dialog . So Powercad has a property BlockDirectory that should be assigned a value ending with a back-slash.

```
Property BlockDirectory:String;
//Example
PowerCad1.BlockDirectory := 'C\MyPowerApp\Blocks\';
```

10.2 Inserting Blocks Using The Block Dialog



The standart BlockDialog, TPCBlockDlg, provides a visual way to insert a Block. The block dialog should access the BlockDirectory of the PowerCad to be able to work, so the blockdirectory property should be assigned a valid value.

To be able to use a block dialog in your application, you should put a TPCBlockDlg in your form,a nd assign the Powercad control to its CadControl property. To make it active in the runtime, call its **Show** method.

In the BlockDialog there is a concept that we call ‘library’. A library is infact a text file that includes the refrences of some blocks. The library is a logical group of blocks that serves for close purposes. For instance you can create a library named ‘KitchenStuff’ for the blocks that are used in akitchen drawing. So a library is also a quick way to find a block.

In the side picture, the top combobox includes the available librarries and the listbox includes the available blocks in the library.The bottom picture is a preview for the current block. The popup menu that is activated by right click on the blocklist is a menu to organize libraries and blocks with in them. The current library is ‘Animals’ and the current block is ‘Donkey’.

You can use the popup menu actions for creating a new blank library and inserting blcoks to the current library, or removing the current block from the library

To insert the current block to the drawing drag the preview of the block to the drawing, and drop it to the location where you want the block to be inserted.

10.3 Inserting Blocks Using ToolIdx Property

As it is declared before the *ToolIdx* property specifies which action will be executed on the PowerCad editor when the user activates mouse clicks. Mostly the *ToolIdx* property works together with a different property such as *CurentFigure*. For example; if the *ToolIdx* property is *toFigure*, the class name of the figure that will be drawn should be specified in the *CurrentFigure* property.

By setting the ToolIdx property to a ‘block insertion’ tool, we can insert a block with user’s mouse click, in two way. For these we will introduce two different ToolIdx value. The first one is *toInsertBlock* and the second one is *toInsertCurrentBlock*.

If you set the ToolIdx property to *toInsertBlock*, an opendialog will be executed after the user clicks on the editor for the location of the block. And the block will be inserted after the user selects a block from the opendialog.

```
Property ToolIdx:TPCTool;
PowerCad1.ToolIdx := toInsertBlock;
```

If you set the ToolIdx property to *toInsertCurrentBlock*, PowerCad will also use a second property value to insert the block. It is the *CurrentBlock* property. In this case, works go on a little bit different, and the difference is that the user is not asked to select a block from an open dialog, instead the block that is defined in the CurrentBlock property is inserted directly. Note that the full path of the block should be provided in the currentblock. When the toolidx property value is *toInsertCurrentBlock*, and a valid block path is given in *CurrentBlock* property then, the block will be inserted to the location that is defined by user’s click.

```
Property ToolIdx:TPCTool;
Property CurrentBlock:String;
PowerCad1.ToolIdx := toInsertCurrentBlock;
PowerCad1.CurrentBlock := ‘c:\MyPowerapp\Blocks\Valve.pwb’;
```

10.4 Inserting Block Calling PowerCad Methods

There are 2 methods in PowerCad interface designed for the developers to insert blocks. By using these calls, the standart block insertion ways are not used, instead the developer provides insertion of the block independent from the standart ways.

```
Function InsertBlockWithFileName (LayerNbr: integer;
                 FileName:string; x,y:integer):TFigHandle;
//Example
PowerCad1.InsertBlockWithFileName(0,
                ‘c:\MyPowerapp\Blocks\Valve.pwb’, 50,120);
```

```
Function InsertBlockFromStream (LayerNbr: integer;
                 Stream:TStream; x,y:integer):TFigHandle;
//Example
var xStream:TStream;
    FileName:String;
FileName := ‘c:\MyPowerapp\Blocks\Valve.pwb’;
xStream := TFileStream.Create(FileName,fmOpenRead);
PowerCad1. InsertBlockFromStream (0,xStream,50,120);
xStream.Free;
```

Both *InsertBlockWithFileName* and *InsertBlockFromStream* insert the block to the given location given in *x,y* parameters. The *LayerNbr* parameter specifies the number of the layer in which the block will reside. And in both functions the return value is the figure handle of the inserted block. The difference between the methods is the *FileName* and *Stream* parameters. In the first method the *FileName* specifies the full path of the block that will be inserted, and in the second method, the *Stream* parameter specifies the *TStream* object in which the block data is stored.

10.5 Making Blocks

Making something block in PowerCad means, saving a group of figures in block format. In PowerCad there is a method for making the selected figures block. This method is called 'MakeSelectionBlock'. This method is a function requires one string parameter for the filename of the block. The given path should referenc any location on a valid disk, but it highly suggested that the blocks should be saved to the block directory to be able to used with in Block dialog.

The *makeselectionblock* method works only there is one selected figure in the drawing. So if you want to make a block from more than one figures, first you should group them all and then call the method when the group is selected.

```
Procedure MakeSelectionBlock (FileName:string);
//Example
PowerCad1.MakeSelectionBlock('c:\MyPowerApp\Blocks\New.pwb');
```

11 Macros in PowerCad

In PowerCad, a macro is a userdefined procedure to be executed by Powercad. By use of macros, users can handle user -specific common actions that requires a logical process. For example, if user wants to draw a circle according to the result of a complex mathematical parametric formula, then there should be a need for writing a macro for this purpose. PowerCad can run macros that are written in some documented standarts.

In PowerCad, normally macros are stored as independent text files if you use the standart macro interface, the macro dialog. However, for powercad a macro is a set of text lines which forms the script to be executed, and thse lines can be stored in anywhere including the databases. When you send the macro as a stringlist to the PowerCad, it will compile and run it through the inside script engine.

So you can either use the standart macro interface, or provide your own macro interface to your users and store the macro script in anywhere you want. The only limitation here is that the script should be written in the language standarts that PowerCad expects.

11.1 PowerCad Script Language - PSCL

Normally applications use industry standart script languages like visual basic or jave, however the engines that are coded for these languages are so environment dependant that most user computers will fail to run a vb or java language. So Powercad has included its own script engine in itselg, and this script engine is designed CAD spcific and also is extendable by the user.

This scipt engine is based on Pascal language, but to make it easy to write a macro, the script language is designed as a limited pascal-like language. There are only few keywords in it, however there is almost no cad functionality that can not be done by using the macro-way.

The reserved words in PSCL (PowerCad Script Language) are the keywords of original Pascal, and serves for the same functionality. Listing 1 shows the reserved words in PSCL.

Listing 1- Reserved Words in PSCL

```
Program, Label, Goto,Var, const, type, Begin, End, procedure,
function, record,

Byte, Word, Longint, Integer, ShortInt, Cardinal, SmallInt,
Real, Single, Double, Extended, Currency, Boolean, ByteBool,
WordBool, LongBool, String, Variant, Pointer

And, Or, Xor, Not, Shl, Shr, Div, Mod

True, False, Nil

If, then, else,

While, Repeat, Until, For, To, DownTo, Do
```

In PSCL the whole structure of a script, is like the structure of a Pascal programme. Note that, in other script engines, a script has the structure of a procedure while in PSCL the script has got the structure of a programme. So You can write very complex scripts that can use procedures and functions defined in itself.

Listing 2 Shows the structure of a script in PSCL and the definitions of the script sections. Also a simple ‘HelloWorld’ example is implemented in the listing.

Listing 2 – The structure of a macro script in PSCL

```
Program ScriptName;  // The optional header.
Type // Optional Type declerations.
     // Can be used only for record definitions.
Var // Optional Variables section of the program.
Const // Optional Constants section of the program.

// Define Procedures and functions here to call from
// main block. Optional

Procedure Sample1;
Begin
End;

Function Sample2: integer;
Begin
End;

// Main Block
Begin
 /// Write the main code of the program here
End.
```

```
// The Hello World Example
begin
 PrintMessage(‘HelloWorld’);
end.
```

11.2 The Function Library in PSCL

PSCL library provides users a CAD specific library including common Delphi library procedures and PowerCad methods. This library is extending in each update because new functions are added to the library. This library can also be extended by the application developer with a special code interface as will be documented in following sections. Listing 3 shows the Delphi runtime library procedures and functions that can be used in a PSCL script. The details of these functions are explained in ‘PSCL Reference’ chapter.

Listing 3 – Delphi RTL functions and procedures in PSCL

```
SetLength, Length, Insert, Pos, Copy, Str, Delete, IOResult,
Move, Randomize, Upcase, VarClear, VarCopy, VarType, VarAsType,
VarIsEmpty, VarIsNull, VarToStr, VarFromDateTime,
VarToDateTime, VarArrayOf, VarArrayDimCount, VarArrayHighBound,
VarIsArray, UniqueString, Fpower10, TextStart, MessageDlg,
MessageDlgPos, ShowMessage, ShowMessagePos, InputBox,
InputQuery, IntToHex, IntToStr, Format, AnsiUpperCase,
AnsiLowerCase, FloatToStr, AnsiCompareStr, AnsiCompareText,
Trim, TrimLeft, TrimRight, Val, Date, Time, Now, StrToDate,
DateToStr, DecodeDate, EncodeDate, FormatDateTime, StrToInt,
StrToFloat, FormatFloat
```

There are also CAD functions that represents an action on the PowerCad drawing, mostly these CAd functions stand for a specific PowerCad class method. But the parameter structures of PSCL CAD fucntions are not always same as the PowerCad class methods. Because the macro language is as simple as possible, so that there is a special communication interface with the PowerCad. Listing 4 shows an example for comparing a method call of PowerCad and its corresponding PSCL function.

Listing 4 – PSCL Cad function Example

```
// PowerCad Method Call for Drawing Circle
Function Circle(LayerNbr,cx,cy,radius,w,s,c,brs,brc:integer;
                select: boolean):TFigHandle;
//Example
PowerCad1.Circle(0,50,50,100,1,2,clRed,1,clBlue);
```

```
// PSCL Call for Drawing Circle
Function Circle(LayerNbr,cx,cy,radius: integer):TFigHandle;
//Example
Circle(0,50,50,100);
```

Almost each method of PowerCad has a corresponding CAD function in PSCL, and also for setting the properties of tehPowerCad control PSCL serves Set and GET functions. You can find the CAD functions that can be used in a PSCL script in the 'PSCL Reference' chapter.

11.3 Running Macros Through Method Calls

The macro methods of PowerCad, provides an interface for running a script from a stringlist, from a text or from a file. All of these sources, the stringlist, the text or the file should include a valid script written in PSCL.

The *RunMacro* method is a procedure that expects one parameter, a stringlist that includes the script. Lisitng 1 shows the interface of the method and an example for it.

Listing 1 – RunMacro Method

```
Procedure RunMacro(Macro:TStringlist);
// Example
var
Script:TStringList;

Script := TStringList.Create;
Script.Add('Program Test;');
Script.Add('var x: integer;');
Script.Add('Begin;');
Script.Add('x := 100;');
Script.Add('ShowMessage(IntToStr(x));');
Script.Add('End.');

PowerCad1.RunMacro(Script);
Script.Free;
```

The *RunMacroText* method is a procedure that expects one parameter, a string that includes the script. The cript lines should be seperated by a return character in the string. Listing 2 shows the interface of the method and an example for it.

Listing 2 – RunMacroText Method

```
Procedure RunMacroText(Macro:String);
// Example
var
```

```
Script:String;

Script := 'Program Test;'+#13+'var x: integer;'+#13+
          'Begin;'+#13+'x := 100;'+#13+
          'ShowMessage(IntToStr(x));'+#13+'End.';
PowerCad1.RunMacroText(Script);
```

These two methods can run any scripts even they are written in runtime. But mostly macros are stored in files and PowerCad has also got a method for running a script from a text file. It is called RunMacroByFileName and to use this method the text file should include only the script that will be executed. Th method expects one string parameter which stands for the full path of the script file. Listing 3 shows the content of a script text file, and the way to execute it in PowerCad.

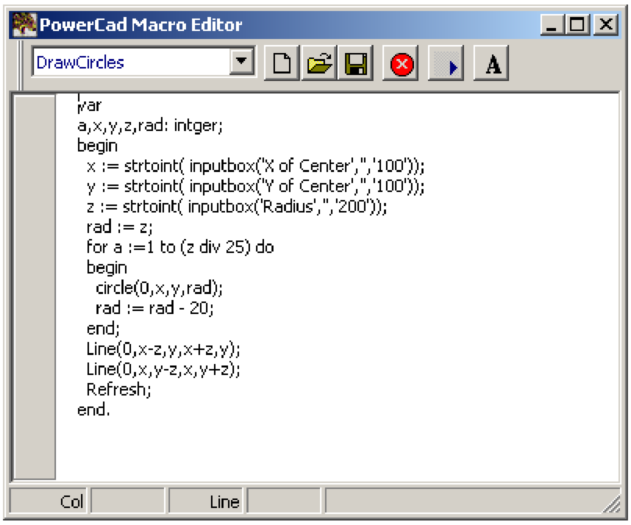
Listing 3 – The RunMacroByFileName Method

***** Content of the text file Test.cmf *****

```
Program Test;
var x: integer;
begin
  x := 100;
  ShowMessage(Inttostr(x));
end.
```

```
Procedure RunMacroByFileName(MacroName:String);
//Example
PowerCad1.RunMacroByFileName('c:\PowerApp\Macros\Test.cmf');
```

11.4 Using Macro Dialog



```
var
a,x,y,z,rad: intger;
begin
  x := strtoint( inputbox('X of Center','','100'));
  y := strtoint( inputbox('Y of Center','','100'));
  z := strtoint( inputbox('Radius','','200'));
  rad := z;
  for a :=1 to (z div 25) do
  begin
    circle(0,x,y,rad);
    rad := rad - 20;
  end;
  Line(0,x-z,y,x+z,y);
  Line(0,x,y-z,x,y+z);
  Refresh;
end.
```

Macro dialog provides a graphical user interface for writing macros and running them. This dialog stores the macros in separate text files with the extension *.cmf, and runs them by calling the appropriate methods of the PowerCad cad control. To make a macro dialog active in your application, put a TPCMacroDialog on your form and set the *CadControl* property of it to the PowerCad control on your form. The macro dialog also has a property *MacroDirectory* that a valid path should be assigned. To make the dialog visible in runtime, call the show method of the macro dialog.

The macro dialog consists of a text editor and a toolbar for organizing macros. On the toolbar there is a combobox and six buttons, the combobox includes the list of the macros that are in the specified macro directory. To edit or run a macro, you should select it from the combobox.

🗋 The new button on the toolbar creates a new macro with the name you specify through an input box.

📂 The Open button is used to load text file to the editor

💾 The Save button saves the macro

ⓧ The Delete button deletes the macro with its file

▶ The Run button runs the macro on the PowerCad

A The font button is used to set the font properties of the editor.

11.5 Extending PSCL – Add your own functions to the script language

PSCL is controlled with in a specific engine in PowerCad, so using the PowerCad class interface this script language can be extended with your own constants and functions.

To add a constant to PSCL, there is a method named *AddPsclConstant*. The constant you want to add may include any value that a variant accepts, because the constats in PSCL is behaved as variant in the background. This procedure expects two parameters, the first one is a string which will stand for the name of the constant and the second is a variant which stands for the value of the constant. Listing 1 shows the interface and an example for *AddPsclConstant*.

Listing 1 – The AddPSCLConstant Method

```
Procedure AddPsclConstant(ConstName:String;Value:Variant);
//Examples
PowerCad1. AddPsclConstant('PI',3.14);
PowerCad1. AddPsclConstant('Err','There is an error');
PowerCad1. AddPsclConstant('Days',365);
```

To add a function or a procedure to the script language, PowerCad has got two methods; *AddPsclProcedure* and *AddPsclFunction*. But there are some more things that should be done when adding procedures or functions.

The proceudures/functions that will be added to the script language should resembled by a real callback function in your code. Either it is a function or a procedure that you will add, you will use the same callback function structure to resemble it in the real code. The structure of this function should be as it is shown in listing 2.

Listing 2 – The structure of PSCL callback function

```
function MyFunc(slf:TObject;var s:array of variant):variant;
Begin
  // Implementation
End;
```

The slf parameter is used for inside purposes, so skip it. But the parameters that the macro writer has sent to your function are access with in the s variant array. The index of this array is 0 based.

After you implemet your fucntion, now you can add it to the engine bu using the *AddPsclProcedure* or *AddPsclFunction* methods. If you want this action behave as a procedure use the first method else if you want it to behave as a function use the second method. But in this case you should returna value in your function code since it expects one. Both of the methods expects the name of the function, the address of it and a Byte array for defining parameters. Listing 3 Shows the inerface for adding functions and procedures with the examples.

Listing 3 – The AddPSCLProcedure and AddPSCLFunction Method

```
Procedure AddPsclProcedure(ProcName:String;ProcAdr:TProcType;
                                 Params:array of byte);
Procedure AddPsclFunction(ProcName:String;ProcAdr:TProcType;
                                Params:array of byte);
//Examples
PowerCad1.AddPsclFunction('Func',MyFunc,[0,0]);
PowerCad1. AddPsclFunction('Proc',MyProc,[2]);
```

As you can see in Listing 3, each method has a parameter named *Params*. This parameter is an array of byte, and will be used to make the script engine informed about our PSCL functions or procedures parameter count and memory-types. Each element in the array represents a parameter of our PSCL function/procedure, and thus the count of the element is the count of the parameters. But if we want to add a PSCL function/procedure that expects no parameter then we should pass an array that has a 2 in it. If we use 0 in any array element then this will be a stack param,if we use 1 in any array element then this will be a **var** parameter, if we use 2 then this will mean that no parameters expected, and if we use 3 in the last element then the last parameter will be an unlimited-count (open array) parameter. In below sections these memory-types of parameters will be explained by examples.

Stack and Var Parameters

A stack parameter is normal parameter which the written value is copied to procedures stack. This is the most-common usage of a paramter. A var parameter is a pointer parameter wich the adress of the written variable id sent to the procedure so that the value of the sent variable can be modified in the procedure. Below, Listing 4 shows both the usage of the stack and var parameter, from writing the function to using it in the script.

Listing 4- Stack and Var Parameters in PSCL

Let us assume that we want to add a procedure to the PSCL, which will add two strings given in the first 2 string parameter, and writes the compund string to the 3rd parameter . So the first two will be a normal(stack) parameter and the last parameter will be a var parameter. The protype of the procedure is given below. Notethat this prototype is in Delphi style, and hos nothing to do with PSCL.

```
Procedure AddStrings(s1,s2:String; var NewStr: String);
```

Now First Let us see how we will add this function to the PSCL by using PowerCad interface.

```
Powercad1.AddPsclProcedure('AddStrings',MyAddStrings,
                                  [0,0,1]);
```

As you can see above, we have written 0 for the first two parameters and 1 for the 3rd parameter. Because the first twp parameters are normal stack parameters while the 3rd one is a var parameter.

Now as you can see above registration, we have given a function name for this PSCL procedure, MyAddStrings. This is the name of our callback function in our own code which does the real job. Now let us see what we will do in this callback function.

```
function MyAddStrings(slf:TObject;var s:array of
                      variant):variant;
Begin
  // s[0] : s1
  // s[1] : s2
  // s[2] : NewStr
  s[3] := s[0]+s[1];
End;
```

The parameter values that the macro writer (user) has written in the script is brought to our real fucntion in the s array. And you already know that s array is a zero based array and the PSCL procedure parameters are accessed in order. Now let us see how this added PSCL procedure can be used in a script

```
Program Strings;
var myStr: String;
begin
  AddStrings('Today is ' , 'Friday', myStr);
  ShowMessage(MyStr); // Today is Friday
end.
```

No Parameters Expected!

If you want the procedure/function that you want to add PSCL expects no parameter then you should pass one element array that has a 2 only. Below, Listing 5 shows the way adding a non-param PSCL function/procedure.

Listing 5- Non-param PSCL fucntions/procedures

First let us assume our non-param operation. It may be a function which returns the current date in string format. So it is assumed prototype will be as follows:

```
Function Today:Integer;
```

First we should add this fucntion to the PSCL engine.

```
PowerCad1.AddPsclFunction('Today',myToday,[2]);
```

As you can see above, the parameter array includes one element which has got a value 2. This means to the PSCL engine that the Year fucntion will not expect any parameter.

Now Let us write the callback function that will execute the real code.

```
function myToday(slf:TObject;var s:array of
                      variant):variant;
Begin
  result := DateToStr(Date);
End;
```

As you can see above the callback fucntion doesn't use t s array, because we know that the PSCL engine will not pass any parameter to the callback function since the Today function is a non-param function. Now let us see how we can use this fucntion in a PSCL script.

```
Program DateTime;
begin
  ShowMessage('Today is: '+Today); // Today is:12/02/2001
end.
```

Unlimited Parameters

As you know from programming, sometimes the parameter count of a procedure or a fucntion is nodt limited as it is in *WriteLn*procedure. This is also possible in PSCL, if a fucntion/procedure is added as including open-array parameter, then PSCL handles this function/procedure in a specific way. All parameters that are entered by the macro-writer is passed to the callback function in a specific array, so that the array index for the unlimited parameter holds also an array. But always we should be sure that the unlimited parameter should be the only parameter or it should be the last parameter. And if you want a parameter to behave unlimited then you shpuld pass 3 for it in the paramter array when adding the function. Listing 6 shows the usage of open array parameters.

Listing 6 – The unlimited-open array parameters in PSCL

Now first let us assume that we want to add a function which sums the numbers we have sent and returns the multiplication of this sum with the factor we have given in the first parameter. So our function will have two parameters, the first will stand for the factor and the second will stand for the numbers that will form the sum. The assumed protype of this fucntion is below. The parametrs tha are in the bracket is the optional numbers that wil be added.

```
Function FactorSum(Factor:Integer;
         Nbr1 [,Nbr2,Nbr3 …, Nbrn]:integer):integer;
```

Now let us see how we will add this function to PSCL.

```
Powercad1.AddPsclFunction('FactorSum',MyFSum,[0,3]);
```

As you can see above, we have written 0 for the first parameter and 3 for the second parameter which will be unlimited. Now let us write the callback function for this PSCL fucntion.

```
function myFSum(slf:TObject;var s:array of variant):variant;
var factor: integer;
    Count,i: integer;
    Sum: integer;
Begin
  // The first element in the array will include
  // the factor s[0]
  Factor := s[0];
  // The second element in the s array is an inner array
  // that includes the numbers to be added each other.
  // This nested array’s first (0) element is the count of
  // the nested array’s element
  Count := s[1][0];
  // Now let us add the numbers together to find the sum
  for i := 1 to Count do Sum := Sum + s[1][i];
  // The result of the fucntion is
  // the multiplication of th factor by the sum.
  Result := Sum * Factor;
End;
```

Finally, we will see how we can use this PSCL function in an example script.

```
Program Sum;
var fSum1,fSum2: integer;
    factor: integer;
begin
  factor:= 12;
  fSum1 := FactorSum(factor,1,7,5,3,4); // fSum1 = 240
  fSum2 := FactorSum(factor,6,2,2); // fSum2 = 120
  ShowMessage(InttoStr(fSum1+fSum2)); // 360
end.
```

12 Making and Using Plugins

12.1 What is a PowerCad Plugin

Mostly plugins are used to extend an application after its delivery. But this extension is not an update of the executable, it is just a different file, mostly a DLL, that provides the application new features. The most important case in the plugin technologies is the strict communication. This means that a plugin that is designed for a pecific application should have a common interface that the application expects. Powercad has encapsulated this technelogy in itself, so that any application designed by PowerCad can be extended with the plugins created independantly by the application developer or 3rd party developers.

A PowerCad plugin is a Windows Dynamic Link Library (DLL) that has got a special export interface. If this DLL is in the plugin directory of the PowerCad application and if it has got the special extension (*.pce), PowerCad Loads this Dll, and a two-way communication is built up between the DLL (plugin) and the PowerCad. We will see how this communication is executed but before this we will see the Powercad interface for using plugins.

12.2 Making use of plugins

To be able to use a plugin, the DLL should be a Valid powercad plugin, it should be located in PowerCad plugin directory, and it should have the extension pce. PowerCad has got an interface for using the plugins, there is a property and some methods that serves for the usage of the plugins.

The property **PluginDirectory** stands for the path to the location where the plugins resides. The path should end with a back slash '\' as in the example below.

```
Property PluginDirectory:String;
// Example:
PowerCad1.PluginDirectory := 'C:\MyPowerCadApp\Plugins\';
```

Powercad doesn't make any automatic loading for the plugins, the application developer should load the plugins, in the right time of the application. But normally the best time is the beginning of the application such as in the OnCreate event of the main form. Note that , before using a plugin the plugins should be loaded, and before loading the plugins, the plugin directory should be specified. PowerCad has a method for loading plugins, LoadPlugins. This method doesn't expect any parameters.

```
Procedure LoadPlugins;
// Example:
PowerCad1.LoadPlugins;
```

After loading the plugins, you should know the names of the plugins and their *verbs*. A verb is an action of a plugin that is to be called from the PowerCad application. We can get the names of the plugins with the method GetPlugins, and learn the verb names with the method GetPluginVerbs. The GetPlugin is a function that returns a stringlist and excepts no parameter. Note that you should release the returned stringlist after you use it.

```
Function GetPlugins:TStringList;
// Example:
```

```
var ThePlugins:TStringList;
ThePlugins := PowerCad1.GetPlugins;
```

And the GetPluginVerbs method is a function that returns a string which includes the names of the plugins seperated by CR. So you can directly assign this return to the Text property of a created TStringList. This method expects an integer parameter which stands for the index of the plugin. The index of the plugin is the index of its name in the returned StringList of GetPlugins method.

```
Function GetPluginVerbs(PluginIdx:Integer):String;
// Example:
var TheVerbsList:TStringList;
    TheVerbs:String;
TheVerbs := PowerCad1.GetPluginVerbs(0);
TheVerbList := TStringList.Create;
TheVerbList.Text := TheVerbs;
```

As the final step, we will explain how we can see a plugin in action. For action, we need verbs to be activated, so we should call them by using the method DoPluginVerb. This method is procedure which expects two integer parameters, the first one stands for the index of the plugin while the second stands for the Verb's.

```
Procedure DoPluginVerb (PluginIdx,VerbIdx: integer);
// Example: Calling the first verb of the first plugin
PowerCad1.DoPluginVerb(0,0);
```

Below Listing is an example which shows the full mechanism of using the plugins. In the listing there are two sections. In the first you see, in the OnCreate event of the form the plugins are loaded and assigned to application menus. And the second section is a procedure that is called by the menus to activate the plugin verbs.

Listing – Using Plugins

```
procedure TForm1.FormCreate(Sender: TObject);
var Plugs: TstringList;
    VerbList: TStringList;
    Verbs: String;
    MenuItem,MenuSubitem: TMenuItem;
    a,b: integer;
begin

  PowerCad1.PluginDirectory :=
    extractfilepath(application.ExeName)+ 'plugins\';
  PowerCad1.LoadPlugins;
  Plugs := nil;
  Plugs := PowerCad1.GetPlugins;
  VerbList := TStringList.Create;
  if plugs <> nil then begin
    For a := 0 to Plugs.Count -1 do
    begin
       MenuItem := TmenuItem.Create(self);
       MenuItem.caption := Plugs[a];
       MenuItem.Tag := a;
       PluginMenuItem.Add(MenuItem);
       Verbs := PowerCad1.GetPluginVerbs(a);
       VerbList.Text := Verbs;
       For b:= 0 to VerbList.Count -1 do
```

```
        begin
          MenuSubItem := TMenuItem.Create(self);
          MenuSubItem.tag := b;
          MenuSubItem.OnClick := PluginClick;
          MenuSubItem.Caption := VerbList[b];
          MenuItem.Add(MenuSubItem);
        end;
     end;
     Plugs.Free;
  end;
  VerbList.Free;
end;

Procedure TForm1.PluginClick(sender: TObject);
var sub,par: TMenuItem;
Begin
  sub := sender as TMenuItem;
  par := sub.Parent;
  PowerCad1.DoPluginVerb(par.tag,sub.tag);
End;
```

12.3 Designing a PowerCad Plugin

To make a plugin, we will start a DLL project in Delphi. In this project, we will implement a special dll which provides standart plugin entrypoints to the PowerCad application. The exported functions and procedures of the plugin should all have the **`stdcall`** calling convention. This Dll can include form(s) or it can be formless plugin which does the jobs with taking any information from the user.

A PowerCad application , because of the technology coded in PowerCad classes, exports plenty of functions and procedures to provide an interface to an outside application or a dll. So the plugin also uses this interface to make some actions on the Powercad application. Infact with any application, byusing the instance handle (HInstance) of the PowerCad application you can get the addresses of the exported functions and you can call them just like you are calling functions from a dll. In Powercad packages there is a unit called **`PCPlgLib`** and this unit includes all necessary things to access the powerCad application and take the exported functions' adresses and assign them to defined functions. To make all these things done, you should just call the `GetAdresses` procedure of this unit. So you should include this unit in the uses clause of the plugin DLL.

Listing 1 – The Uses Clause of the Plugin DLL

```
uses
  SysUtils,Classes,Windows,Messages,PCPlgLib;
```

But as you can guess, a plugin technology should be based on a two-way communication. This means, the PowerCad application should also call the functions of the plugin dll as the plugin can call the fucntions of the application. The application loads the plugin, and initialize this plugin by sending the application instance handle to it. By this handle the plugin retreives the adresses of the exported functions of the application. This communication continues as it is described in the previous section.

So a plugin Dll should have an exported procedure to initialize itself. This is a standart procedure that a plugin should export. The name of the procedure is Init and it has got an integer parameter for the instance handle of the PowerCad application that the plugin was loaded from.

Listing 2 – The Init Procedure of the Plugin

```
procedure Init(Owner: Integer); stdcall;
begin
  GetAdresses(Owner);
  // Plugin specific intialization code
end;
```

In the above procedure, note that the functions are retreived from the host PowerCad application, and all these are done in PCPlgLib unit by the use of GetAdresses function. The Owner parameter is the HInstance of the host PowerCad application. In this procedure, you can also write down your plugin specific initializations, such as giving the initial values to your plugin variables.

The Powercad application will need the Verbs of your plugin. We have seen "what a verb is' in the previuos section, so we know that we should specify some verbs for our plugin according to the purpose of our plugin.

For example, our plugin can be a charting plugin which creates pie charts with some given data and edits these charts. So we will have two verbs called **NewChart** and **EditChart.** We should give the names of these vers to the host application so the standart plugin interface has got a second function called GetVerbs. This function returns a pchar (it can be string in VB projects) which includes the name of the verbs seperated by a return character.

Listing 3 – The GetVerbs function of the Plugin

```
Function GetVerbs:PChar;stdcall;
Begin
  result := 'NewChart'+#13+'EditChart';
End;
```

In the below example, we have got two verbs, and these verbs will be activated by the application, using the plugins DoVerb procedure. This is also a standart and 'must be' procedure, and in this procedure according to the 0 based verb index, you excute the actions of the plugin. You can create forms, to ke some parameters from the user, or just do something directly on the PowerCad application. Note that you will use the exported functions of the application, and thse fucntion ara available for the plugin by use of the PCPlgLib unit. If you look at inside this unit, you will see plenty of functions and procedures with their parameter structures defined. These exported PowerCad functions are almost same as the PowerCad class methods, the only difference is that there is a pc prefix at the beginning of the method name such as **`pcRotateSelection`**.

Listing 4 – The DoVerb procedure of the Plugin

```
Procedure DoVerb(VerbIndex: integer);stdcall;
Begin
  Case VerbIndex of
    0: // NewChart
    begin
      //Write your plugin specific action code here
      //according to the purpose of the verb.
    end;
    1: // Edit Chart
    begin
      //Write your plugin specific action code here
      //according to the purpose of the verb.
    end;
  end;
end;
```

So by now, we have introduced three standart ‘must do’ fucntions for exporting in the plgin. And in this case our expor clause of the plugin dll will be as it is in Listing 5.

Listing 5- The export clause of the plugin

```
  exports
    Init name 'Init',
    GetVerbs name 'GetVerbs',
    DoVerb name 'DoVerb';
```

Listing 6 shows the complete code of a plugin based on PieChart example.

Listing 6 – The complete code of a plugin based on an example

```
library PieChart;

uses
  SysUtils,Classes,Windows,Messages,Graphics,Controls,Forms,
  Dialogs,StdCtrls,Buttons,Grids,PCPlgLib;

procedure Init(Owner: Integer); stdcall;
begin
  GetAdresses(Owner);
  // Plugin specific intialization code
end;

Function GetVerbs:PChar;stdcall;
Begin
  result := 'NewChart'+#13+'EditChart';
End;

Procedure DoVerb(VerbIndex: integer);stdcall;
Begin
  Case VerbIndex of
    0: CreateNewChart;
    1: EditSelectedChart;
  end;
end;

Procedure CreateNewChart;
Begin
  // Plugin-verb specific code
End;
Procedure EditSelectedChart;
Begin
  // Plugin-verb specific code
End;
  exports
    Init name 'Init',
    GetVerbs name 'GetVerbs',
    DoVerb name 'DoVerb';

begin
end.
```

Listing 7 shows a list of functions that are exported by PowerCad application. These functions will be exported from any application or Dll that encapsulates Powercad objects in itself. You can use these functions with in your plugin dll to execute any actions on the Powercad application.

Listing 7 – List of the exported functions of a PowerCad application

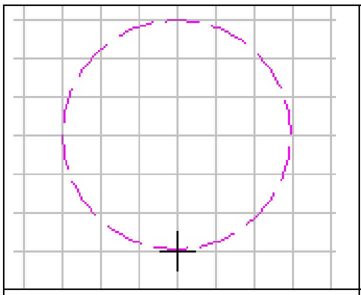
```
pcHideGrids,pcShowGrids,pcIsGrids,pcHideRulers,pcShowRulers,
pcIsRulers,pcHidePanel,pcShowPanel,pcIsPanel,pcHideGuides,
pcShowGuides,pcIsGuides,pcSetScale,pcGetScale,pcSetBgColor,
pcGetBgColor,pcSetGridColor,pcGetGridColor,pcGetGridStep,
pcSetGridStep,pcGetWWidth,pcSetWWidth,pcGetWHeight,
pcSetWHeight,pcGetActiveLayer,pcSetActiveLayer,pcSetPageLayout,
pcGetPageLayOut,pcSetPageOrient,pcGetPageOrient,
pcSetGuideTrace,pcGetGuideTrace,pcSelectAll,pcDeSelectAll,
pcNewLayer,pcDeleteLayer,pcDeleteLayerWithNbr,
pcDeleteAllUserLayers,pcShowLayer,pcHideLayer,
pcShowAllLayers,pcHideAllLayers,pcExHideLayer,pcFlueLayer,
pcExFlueLayer,pcMergeAllLayers,pcMergeVisibleLayers,pcRefresh,
pcRefreshSelection,pcDraw,pcDrawToDC,pcUndo,pcRedo,
pcGroupSelection,pcUnGroupSelection,pcDrawFigures,
pcDrawSelectionPoints,pcDrawFigureGuides,pcOrderSelection,
pcRemoveSelection,pcRotateSelection,pcMirrorSelection,
pcInvertArcsOfSelection,pcArrangeArcStyleOfSelection,
pcConvertToBezier,pcConvertToPolyline,
pcArrangePolyLineSelPoint,pcFlipImagesOfSelection,
pcSetTransparentOfSelection,pcScaleSelection,pcModifySelection,
pcGetSelectionBounds,pcAlignSelection,pcReselect,
pcGetSelectionCount,pcCollectSelectedFigurespcCheckByPoint,
pcSelectByPoint,pcSelectWithInArea,pcMoveSelection,
pcDuplicateSelection,pcArrayRectSelection,
pcArrayPolarSelection,pcMakeSelectionBlock,
pcBoundLineToFigures,pcBoundLinePoint,pcUnBoundLine,
pcMakeSelectedLinesPolyline,pcClipSelBitmapToSelFigure,
pcSaveToFile,pcLoadFromFile,pcIsTextFile,pcLoadFromSource,
pcLoadFromStream,pcSaveToStream,pcGetSourceText,
pcInsertBlockWithFileName,pcInsertBlockFromStream,
pcInsertBlockFromSource,pcExportAsWmf,pcSaveAsBitmap,
pcBmpPrint,pcPrintDrawing,pcPrintByTiling,pcImportDXF,pcClear,
pcCopyToClipBoard,pcCutToClipBoard,pcPasteFromClipBoard,
pcGetLayerNbr,pcFindFigureByName,pcLine,pcVertex,pcPolyLine,
pcvbPolyLine,pcEllipse,pcCircle,pcArc,pcRectangle,
pcInsertBitmap,pcInsertWMF,pcTextOut,pcRunMacro,pcPrintPreview,
pcCountBlock,pcGetSelectionHandles,pcGetSelectionHandle,
pcShowPropertyWindow,pcExecuteCommand,pcExecuteTBCommand
```

13 More Customization: Making Custom Figures

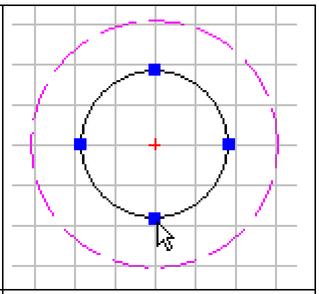
You already know what a figure is in PowerCad. It is the basic element of a drawing such as line, rectangle, circle, etc. Blocks are a solution if you want to provide your user something more than a basic figure, but what about providing something more than a "block". Yes, in this chapter we will see how to implemet a custom figure which has its own custom behaviours defined by you according to the needs of your application.

A custom figure will behave just like native basic figures as line, circle, etc. You should only write its specific class code inheriting from TFigure of other figure classes. There are some methods that you should override and reimplement.

13.1 Basic Concepts of Custom Figure Developing



Creation Shadow



Modification Shadow

Base Class: The **TFigure** class which all figures are inherited from.

Based Class: The class from which a figure is inherited. It may be TFigure or a figure class inherited from this.

Custom Figure: A Figure Class which is inherited from TFigure or a figure class inherited from this.
Modification Points: The points that are drawen when the figure is selected. The figure is reshaped (modified) from these points.
Figure Guides: The guides that are drawn with a figure such as the center mark of a circle. They are drawen to the screen but not printed to the printer.
Figure Bounds: The bounding rectangle of a figure.
Figure Shadow: The temprorary figure that is drawn on the editor when the user determines the figure locations.
Modification Shadow: The temproray figure that is drawn on the editor when the user modifies the figure.

13.2 The Virtual Methods of TFigure

When you make a new figure class inherited from TFigure or anyother Figure classes, you should reimplement some methods according to the needs of the new figure. Below is a list of these methods.

```
Constructor create(LHandle:LongInt; aDrawStyle: TDrawStyle;
               aOwner: TComponent);
```

TFigure class and each figureclasses inherited from this, has its own constructor. The base class constructor TFigure.Create creates the figure and initialize the LayerHandle, DrawStyle,Owner properties using the constructor parameters. If your figure needs some more parameters in the Create procedure, such as location or a length, you should write your own constructor.

```
Procedure Initialize;override;
```

This method is called in the constructor, and it is used to assign the default values of some parameters like PointCount. If your figure needs specific initializing, you should override this method. Note that, this method is called just after creating the figure. So if your figure doesnt need specific parameters in the constructor, but needs some specific initializing like creating a bitmap or a list in the figure creation, then don't implement a constructor,just override this method and write your initialization code.

```
Procedure draw(DEngine:TPCDrawEngine;isFlue:Boolean);override;
```

This method draws the figure by using the rawEngine provided in the parameters. If your figure will be drawn in its own style, this means if you inherit it From TFigure, then you should reimplement this method. There are two parameters of the draw procedure. *DEngine* is used to send the drawing commands to the active canvas, and if the *isFlue* parameter is true, the drawing color will be clGray.

```
Function IsPointIn(x,y:integer): boolean;override;
```

This method should return *true* if the given point is in the drawing area of the figure, so that PowerCad will decide to select it when the user clicks on the editor. If your figure has a different drawing area then the based figure, you should reimplement this method.

```
Function Duplicate:TFigure; override;
```

This method should duplicate the figure and return the new figure, so that Powercad will be able to multiply the figure in commands such as copying or arraying. In any case you make an inheritende you should override and reimplement this method.

```
Procedure DrawFigureGuides(DEngine: TPCDrawEngine);override
```

This method should draw the figure guides that are drawn on the screen but not printed. If your figure needs to show a specific location on itself for guiding purposes (such as the center of the circle), you should reimplement this method. Draw the guides in a different color then the color of the figure.

```
Procedure GetModPoints(ModList: TList);override;
```

This method registers the modification points of the figure to the owner PowerCad. These modification points are type of TModPoint, and by using *RegisterModPoint* method of PowerCad you register your modification (selection) points with their types,coordinates,shapes and colors.

```
Function RegisterModPoint(Figure: TFigure;
PType:TModPointType;
         DType: TPointType; Color: Tcolor;
         aDim,X,Y,seqNbr: integer):TModPoint;
```

The *RegisterModPoint* method creates a modification point and returns it so that you can add these points to the *ModList* given in the procedure parameters.

```
procedure GetBounds(var figMaxX,figMaxY,figMinX,figMinY:
                         integer);override;
```

This method should calculate its bounding rectangle and set the coordinates to the given parameters. If the bounding rectangle of your figure can be different then it was in the base class you should override and reimplement this method.

```
Procedure Move(deltax, deltay: integer); override;
Procedure Rotate(aAngle: integer; cPoint: TPoint); override;
Procedure Mirror(Point1,Point2: TPoint); override;
Procedure Scale(percentx,percenty: integer; rPoint: Tpoint);

override;
```

These methods do the transformations based on the points of the figure. So if your figuire can be transformed by its poinst , (normally most figures can be transformed by its figures) you shouldn't override and reimplement these method. But if your figure should calculate something in these methods then inherit thse methods and call the inherited method at the beginning of the implementation.

```
Procedure WriteToStream(Stream:TStream);override;
```

This method should write figure specific class data (fields or properties) to the stream given in the parameters. Each field you should be written to the stream with a field code prefix. The field code ranges change according to the type of the field. Use the WriteField, WriteStrField, WriteBinField, WriteStreamField, procedures defined in PCTypesUtils to write a field to the stream. Below is the prototypes of these Stream Writing procedures thath you can use in the implementation of this method.

```
Procedure WriteField(Code:Byte; Stream:TStream;
                     Const Value;Size: integer);
Procedure WriteStrField(Code:Byte;Stream:TStream;
                        Const Value:String);
Procedure WriteBinField(Code:Byte; Stream:TStream;
                        Const Value:pByte; Size: integer);
Procedure WriteStreamField(Code:Byte; Stream:TStream;
                           Const Value:TStream);
```

For writing numbers to the stream use *WriteField*, for string fields use *WriteStrField*, for binary data use *WriteBinField*, for stream fields use *WriteStreamField*.

When these proceudres writes your field values to the stream it uses an XML like format, first it writes a label (field code) for the field then the data. This label is determined by your code, however it should be in the range of the field-type as given below. Because the size information is gathered from the field code in the stream. Below is a list of field code ranges.

20-89 : Integer numbers
90-119 : Byte Numbers
120-149: Word Numbers
150-179: Binary Data
180-219: String Data
220-239: Double Numbers

```
Procedure SetPropertyFromStream(xCode:Byte;data:pointer;
                                size:integer); override;
```

When a drawing is opend from a file, if PowerCad locates your figure in the file and if your figure class have written specific data to the stream, then for each field data this procedure of your class is called. You should look at the field code given in the *xCode* parameter, and assign the value to its field in you class data.

```
Class Function ShadowType:TShadowType;override;
Class Function CreateShadow(x,y:integer): TFigure;override;
```

For standart shadows override *ShadowType*, for custom shadows override *CreateShadow* fucntion. If you inherit form TFigure you should either override *ShadowType* or *CreateShadow*. If you inherit from an existing figure class then you should decide if you will use the existing shadow or not. If you use the existing shadow you shouldn’t override these functions. But if you want to use a different shadow then the existing one, for standart shadows (Rectangle, Circle, etc.) override *ShadowType* function, for custom shadows override *CreateShadow* fucntion.

```
class function CreateFromShadow(aOwner: TComponent;
            LHandle:LongInt;Shadow:TFigure): TFigure;override;
```

After the shadow ends, powercad sends the shadow to the figure class to make it create the real figure. In anyway you **must** override and reimplement this calss function. According the sizes of the passed *Shadow* figure , create a figure from type of you figure class and return it.

```
Function ShadowClick(ClickIndex,x,y: integer):Boolean;
override;
Function ShadowTrace(ClickIndex,x,y: integer):Boolean;
override;
```

In case of custom shadows, these fucntions should be reimplemented as *CreateShadow* method. The *ShadowClick* method is called in each click of the user when the shadow is active, refresh you shadow values according to the locations given in x,y. Return *true* if this click should be last click so that the shadow will end and *CreateFromShadow* will be called. The *ShadowTrace* is called in case of mouse moves when the shadow is active.

```
Function CreateModification: TFigure;override;
Function TraceModification(CadControl:Pointer; mp:TModPoint;
  TraceFigure:TFigure; x,y:integer;
Shift:TShiftState):boolean;
  override;
Function EndModification(CadControl: Pointer; mp:TModPoint;
  TraceFigure:TFigure; x,y:integer;
Shift:TShiftState):boolean;
  override;
```

When user modifies the figure by moving the modification points, a modification shadow is created. If your figure creates the shadow itself then it should also create a modification shadow in *CreateModification*. The *TraceModification* is called when the mouse moves and modification shadow is active. The *EndModification* is called when the user drops the mousemoving. The *mp* parameter is these fucntions is the dragged modification point, so that according to this point you should make the modification on the figure.

```
Procedure RegisterToPropertyPage;override;
Procedure RefreshPropertyPage;override;
Procedure FigurePropertyChanged(PropertyName: string;
                                Data:PPropData);override;
```

You should override these methods if you want your figure in the ObjectInspector.

13.3 Inheriting Custom figures from Existing Figures

If you want to extend the ability of a current figure, inherit a new figure class from its class. In this section we will inherit a new figure from TRectangle.

Assume that we want to have frames in our drawings, so we want to provide our users a tool for drawing frames. Now, we will see how to reimplement a figure from an exisitng figure step by step.

Step 1: Create a new unit for your custom figures, name it as MyFigures.Pas, and write its uses clause as to include PCTypesUtils, DrawEngine, DrawObjects, ObjectsandProps, ObjectIns, PCDrawing units of PowerCad package. Listing1 shows the unit with its uses clause.

Listing 1 – MyFigures unit with its uses clause.

```
unit MyFigures;

interface

uses
SysUtils, WinTypes, WinProcs, Messages,Classes, Controls,
Forms, Dialogs, ExtCtrls,comctrls,buttons,stdctrls,windows,
math,Graphics,PCTypesUtils, DrawEngine, DrawObjects,
ObjectsandProps, ObjectIns, PCDrawing;

implementation

end.
```

Step 2: To have a Frame tool, we will inherit a new class from TRectangle. And our figure will behave just like Rectangle instead it will draw an inner rectangle in the draw method of the figure. So we know that we should override the draw method. Listing 2 shows the interface of the TFrame class.

Listing 2 – The interface of the TFrame class

```
TFrame = class(TRectangle)
  procedure draw(DEngine:TPCDrawEngine; isFlue:Boolean);
                                                override;
end;
```

Step 3: Now we will reimplement the *draw* method. We know that the orginal draw method which is in TRectangle class, draws a rectangle by combining the points of the figure. If the angle of the rectangle is 0, then calculating the coordinates of the inner rectangle to form a frame is easy, but we know that a rectangle can be in any angle so we must use a genaral way to calculate the coordinates of the inner rectangle. For this puprpose we will use the GetRelativePointBySclae fucntion which is in PCTypesUtils unit. Listing 3 shows the implementation of the *draw* procedure.

Listing 3 – Reimplementation of Draw method

```
procedure TFrame.draw(DEngine: TPCDrawEngine; isFlue: Boolean);
var
    acolor,bcolor : Tcolor;
    points : array [0..3] of TPoint;
    ap1,ap2,ap3,ap4,cp: TPoint;
begin
  inherited;
  acolor := color;
  bColor := brc;
  if (isFlue) then
  begin
    acolor := flueColor;
    bcolor := flueColor;
  end;

  if DrawStyle = dsTrace then
    DEngine.Canvas.Pen.Mode := pmXor
  else
    DEngine.Canvas.Pen.Mode := pmCopy;

  ap1 := actualpoints[1];
  ap2 := actualpoints[2];
  ap3 := actualpoints[3];
  ap4 := actualpoints[4];

  // Find the center of the rectangle
  cp := Point( (ap1.x+ap3.x) div 2,(ap1.y+ap3.y) div 2);

  //Scale the points according to the center
  points[0] := GetRelativePointByScale(90,90,cp,ap1);
  points[1] := GetRelativePointByScale(90,90,cp,ap2);
  points[2] := GetRelativePointByScale(90,90,cp,ap3);
  points[3] := GetRelativePointByScale(90,90,cp,ap4);

  //Draw the inner Rect
  DEngine.drawpolygon(points,4,acolor,width,style,
                      bcolor,brs,RegHandle);

end;
```

Step 4: Now our TFrame figure is ready. If you create it through code by using its constructor, and if you add it to the figures list it will work. But we want more that we will create this figure by clicking on the PowerCad editor after we have selected the assigned button for this figure. Yes in this case, powercad should be aware of this class and in a way the CurrentFigure property should be set to "TFrame" when we want to draw this figure. But before registering this figure to PowerCad there is one more case that we should implement.

Each figure drawen on the Powercad editor provides a shadow which traces by the mouse clicks or mouse movements, to form the figure or to modify the figure. For example; when you want to draw a Rectangle, you click on two different locations on the editor after you select the Rectangle tool. When you first click a shadow figure for the drawing is created and up to our second click the shadow is redrawn according to the new location of the mouse cursor. In fact, this shadow is provided by the figure class itself. Since we have inherited this class from TRectangle, we know that the TRectangle class will provide a shadow, but we still should implement a method for the end of the shadow. This means when user fnishes defining the locations, the shadow figure is sent to the figure class' CreateFromShadow fucntion to create the figure. So if we don't override this method, even the CurrentFigure is assigned 'TFrame', a Rectangle (not a Frame) will be created after the shadow ends. Listing 4 shows the interface and implementation of the CreateFromShadow class fucntion.

Listing 4- CreateFromShadow Method

```
TFrame = class(TRectangle)
  procedure draw(DEngine: TPCDrawEngine;isFlue:Boolean);
```

```
                                               override;
  class function CreateFromShadow(aOwner: TComponent;
  LHandle: Integer; Shadow: TFigure): TFigure;override;
end;
```

```
class function TFrame.CreateFromShadow(aOwner: TComponent;
  LHandle: Integer; Shadow: TFigure): TFigure;
var cad: TPCDrawing;
begin
  cad := TPCDrawing(aOwner);
  Result := TFrame.create(Shadow.actualPoints[1].x,
                          Shadow.actualPoints[1].y,
                          Shadow.actualPoints[3].x,
                          Shadow.actualPoints[3].y,
                          cad.DefaultPenWidth,
                          ord(cad.DefaultPenStyle),
                          cad.DefaultPenColor,
                          ord(cad.DefaultBrushStyle),
                          cad.DefaultBrushColor,
                          LHandle,
                          dsNormal,aOwner);
end;
```

Step 5: Each figure class registered to PowerCad should have a duplicate method which is used to copy figures. If we dont override this method, PowerCad will create Rectangle when the user copies or multiplies the Frame figure. Listing 5 shows the interface and implementation of the Duplicate fucntion.

Listing 5- Duplicate Method

```
TFrame = class(TRectangle)
  ...
  Function Duplicate:TFigure; override;
end;
```

```
Function TFrame.Duplicate: TFigure;
begin
  Result := TFrame.create( 0,0,0,0,
                            width,style,color,
                            brs,brc,
                            LayerHandle,
                            DrawStyle,Owner);
  Result.angle := angle;
  Result.actualpoints[1] := actualpoints[1];
  Result.actualpoints[2] := actualpoints[2];
  Result.actualpoints[3] := actualpoints[3];
  Result.actualpoints[4] := actualpoints[4];
  Result.originalpoints[1] := originalpoints[1];
  Result.originalpoints[2] := originalpoints[2];
  Result.originalpoints[3] := originalpoints[3];
  Result.originalpoints[4] := originalpoints[4];
  Result.rotatePoint.x := rotatePoint.x ;
  Result.rotatePoint.y := rotatePoint.y ;
end;
```

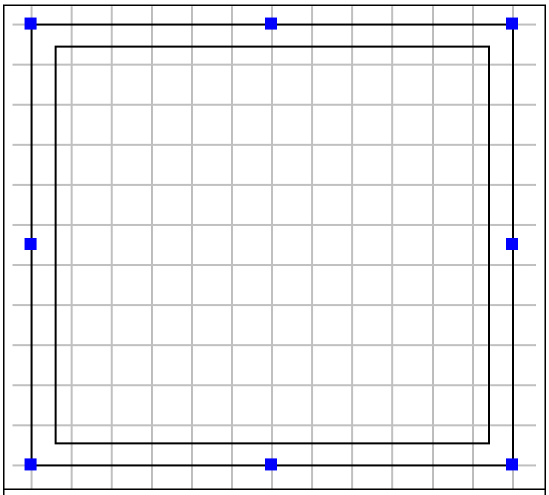
Step 6: Now our figure is ready to work, so we should register it to the PowerCad control in which we want it. And we should provide a control (a button or a menu) for setting the CurrentFigure property to 'TFrame'.

In the OnCreate Event of your form register TFrame to the powercad as folows.

```
PowerCad1.RegisterFigureClass(TFrame);
```

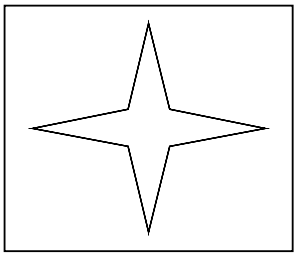
Then in the onclick event of the menu or button you specified for TFrame, write down this code.

```
PowerCad1.CurrentFigure := 'TFrame';
PowerCad1.ToolIdx := toFigure;
```



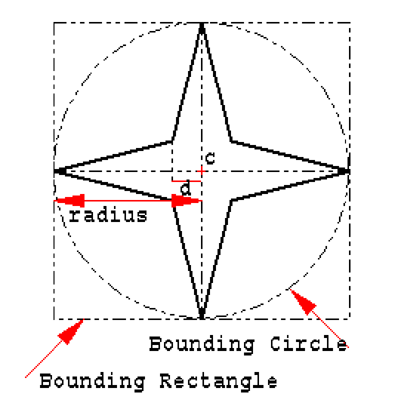
TFrame in action

13.4 Making a Brand New Figure



In this section, we will see how we can implement figure classes by directly inheriting from the base figure class, TFigure. So you can make your own custom figures with more code and more fucntionality. In this case you are not limited with the behaviours of the exisitng figures, instead your figure will behave in the way you code. So now, in this section we will try to develope a 4-point star which you can see its picture on the left. It will have two modifications point, one will be used to arrange the circle radius of the star, the other will be used to arrange to convexity of the star.

The Geometry of Our "Star"



The location marked with **c** is the center of the star, **radius** is the radius of the **bounding circle**, the **boundig rectangle** of the star will always be a square, the convexcity value will be (**d**/**radius**)*100. When the convexity is **0** the star will be a cross, when the convexity is **100** the star will be a square. By default convexity will be **25**. So **d** value will be ¼ of the **radius**.

The Figure Data

Actually the Tfigure has standart and basic fields of a figure. Our star will be some more of these. We know that we should use the ActualPoints, Width, Color, Style, Brs, Brc and Radius properties of TFigure, in addition to these we will add a new field named cValue for the Convexsity value.

```
cValue: Integer;
```

The Constructor

We will override the base constructor, because we need some parameters to form the star at the beginning. These are the center of the star, the pen width, pen style,pen color, brush style and brush color of our star.

```
Constructor Create(cX,cY,rad,w,s,c,abrs,abrc:integer;
                   LHandle:LongInt; aDrawStyle:
                   TDrawStyle;aOwner: TComponent);
```

The Initializer

We will override the Initialize method because we will initialize some fields here. The *pointcount* field should be initialized here because PowerCad need the *pointcount* of the figure for memory operations. Our pointcount will be 1, because we will only store the center coordinates in the actualpoints array.

```
Procedure Initialize;override;
```

Listing 1 shows the interface and implementation of our figure up to this point.

Listing 1 – First Step of TStar Class Implementation

```
TStar = class(TFigure)
  cValue: Integer;
  constructor create( cX,cY,rad,w,s,c,abrs,abrc:integer;
                      LHandle:LongInt; aDrawStyle:
                      TDrawStyle;aOwner: TComponent);
  procedure Initialize;override;
end;
```

```
constructor TStar.create(cX, cY, rad, w, s, c, abrs, abrc,
  LHandle: Integer;aDrawStyle:TDrawStyle;aOwner: TComponent);
begin
  inherited create(LHandle,aDrawStyle,aOwner);
  Initialize;
  originalpoints[1] := Point(cx,cy);
  actualpoints[1]   := Point(cx,cy);
  radius := rad;
  width := w;
  color := c;
  style := s;
  brs := abrs;
  brc := abrc;
end;

procedure TStar.Initialize;
begin
  pointcount := 1;
  cValue := 25;
end;
```

The Shadow Decision

We should decide what type of a shadow we will use. We know that we can use the standart shadows or we can create oir own shadow. We will first use the standart circle shadow and look how figure behaves , later in this section we will create our own shadow. So in this step, we will just implement the ShadowType and CreateFromShadow methods. Since we use a circular shadow, the user will draw a circle on the editor, then we will create our star figure from this shadow circle.

Listing 2 – The Shadow Behaviour of TStar

```
TStar = class(TFigure)
  ...
  class function ShadowType:TShadowType;override;
  class function CreateFromShadow(aOwner: TComponent;
                 LHandle:LongInt;
                 Shadow:TFigure): TFigure;override;
end;
```

```
class function TStar.ShadowType: TShadowType;
begin
  result := stCircle;
end;

class function TStar.CreateFromShadow(aOwner: TComponent;
LHandle: Integer;
  Shadow: TFigure): TFigure;
var cad: TPCDrawing;
begin
  cad := TPCDrawing(aOwner);
  Result :=
TStar.create(Shadow.actualPoints[1].x,Shadow.actualPoints[1].y,
                          TCircle(shadow).radius,
                          cad.DefaultPenWidth,
                          ord(cad.DefaultPenStyle),
                          cad.DefaultPenColor,
                          ord(cad.DefaultBrushStyle),
                          cad.DefaultBrushColor,
                          LHandle,
                          dsNormal,aOwner);
end;
```

Drawing The Star

By using the star geometric principles we will draw our start to the given draw engine. So at the end , we will draw a polygon which all points are calculated as to form a four point star. In this step we will override the Draw procedure, but there are some basic things that we should be aware of when drawing our figure. if the DrawStyle parameter is dsTrace then our pen mode will be pmXor, and our pen color will be clLime. If the isFlue parameter is true, the our pen color will be flueColor. See Listing 3 for mathematical details of the drawing.

Listing 3 – Draw Method

```
TStar = class(TFigure)
  ...
  procedure draw(DEngine: TPCDrawEngine;
                 isFlue:Boolean);override;
end;
```

```
procedure TStar.draw(DEngine: TPCDrawEngine; isFlue: Boolean);
var
    acolor,bcolor : Tcolor;
    cp: TPoint;
    points: array[0..7] of TPoint;
    d: integer;
begin
  acolor := color;
  bColor := brc;

  if (isFlue) then
  begin
    acolor := flueColor;
    bcolor := flueColor;
  end;

  cp := actualpoints[1];
  d := round((cValue/100)*radius);

  if DrawStyle = dsTrace then
    DEngine.canvas.pen.mode := pmXor
  else
    DEngine.canvas.pen.mode := pmCopy;

  Points[0] := Point(cp.x-radius,cp.y);
  Points[1] := Point(cp.x-d,cp.y+d);
  Points[2] := Point(cp.x,cp.y+radius);
  Points[3] := Point(cp.x+d,cp.y+d);
  Points[4] := Point(cp.x+radius,cp.y);
  Points[5] := Point(cp.x+d,cp.y-d);
  Points[6] := Point(cp.x,cp.y-radius);
  Points[7] := Point(cp.x-d,cp.y-d);
  DEngine.drawpolygon(points,8,acolor,width,style,
                      bcolor,brs,RegHandle);
end;
```

Selecting The Star

When the user clicks on the editor with select tool, PowerCad checks if the clicked location is in any figure. So IspointIn method of each figure is called. And incase user draws rectangle to make a selection, Powercad will need the bounding rectangle of our figure. Also about the selection case , there is one more method GetModPoints. By this method,the modification points are registered to Powercad. These points will be drawn when the figure is selected. Normally since we use an stCircle shadow type, by default, even we don’t override these methods our figure will behave as circle.

Actually, the bounding rectangle of our figure is equal to the bounding rectangle of our shadow, so it seems that there is no need to override the GetBounds method. But if the convexity of the star is biggr then 100, the star will be concave star and the bounding rectangle will exceed the bounding circle. So we should overrode the GetBoundMethod.

Since we use a standart shadow we shouldn’t override the GetModPoints method also. Because the base class will register 4 circle modification points which the user can arrange the radius of our figure. But in the following sections when we make our own shadow we will also override this method.

Since the drawing area of our star is different then the circle shadow we should override the IsPointIn function. Because we should make our own calculation if the point is on our figure or not.

Listing 4 shows the implementation of IsPointIn and GetBounds methods with their interface.

Listing 4 – IsPointIn Method

```
TStar = class(TFigure)
  ...
  function isPointIn(x,y: integer): boolean;override;
  Procedure Getbounds(var figMaxX,figMaxY,figMinX,figMinY:
                                            integer);override;
end;
```

```
function TStar.isPointIn(x, y: integer): boolean;
var cp: TPoint;
    points: array[1..8] of TPoint;
    d,i,idx1,idx2: integer;
begin
  if (TBrushStyle(brs) <> bsClear) then
  begin
    result := IsPointInRegion(x,y);
  end else
  begin
    cp := actualpoints[1];
    d := round((cValue/100)*radius);

    Points[1] := Point(cp.x-radius,cp.y);
    Points[2] := Point(cp.x-d,cp.y+d);
    Points[3] := Point(cp.x,cp.y+radius);
    Points[4] := Point(cp.x+d,cp.y+d);
    Points[5] := Point(cp.x+radius,cp.y);
    Points[6] := Point(cp.x+d,cp.y-d);
    Points[7] := Point(cp.x,cp.y-radius);
    Points[8] := Point(cp.x-d,cp.y-d);

    for i := 1 to 8 do
    begin
      idx1 := i;
      if i = 8 then idx2 := 1 else idx2 := i+1;
      if ispointinLine(points[idx1],points[idx2],Point(x,y))
        then result := true;
    end;
  end;
end;

procedure TStar.GetBounds(var figMaxX, figMaxY, figMinX,
                                            figMinY: integer);
var cp : TPoint;
    d,dist:integer;
begin
  cp := actualpoints[1];
  d := round((cValue/100)*radius);
  if cValue <= 100 then dist := radius else dist := d;
  figMaxX := cp.x+dist;
  figMinX := cp.x-dist;
  figMaxY := cp.y+dist;
  figMinY := cp.y-dist;
end;
```

Duplicating the Star

We should override the Duplicate method in any case, if you don't reimplement this method you figure will not be multiplied in copy or array operations. In the implementation of this method create a figure from you class with the current field values. Listing 5 shows the implementation of Duplicate method with its interface.

Listing 5 – Duplicate Method

```
TStar = class(TFigure)
  ...
  function duplicate:TFigure; override;
end;
```

```
function TStar.duplicate: TFigure;
begin
  result := TStar.create(actualpoints[1].x,
                         actualpoints[1].y,
                         radius,
                         width,
                         style,
                         color,
                         brs,
                         brc,
                         LayerHandle,
                         drawstyle,Owner);
  TStar(result).cValue := cValue;
end;
```

Streaming Methods

If our figure has some specific data which is not streamed by base class, the we should handle two methods for streaming. Otherwise our figure will saved to the file with some missing fields. For streaming, WriteToStreeam and SetPropertyFromStream methods will be reimplemented. See Listing 6 for the implementation of the streaming methods with their interfaces. Note that you should **always** call the inherited in WriteToStream, and **never** call inherited in SetPropertyFromStream.

Listing 6 – Streaming Methods

```
TStar = class(TFigure)
  ...
  Procedure WriteToStream(Stream:TStream);override;
  Procedure SetPropertyFromStream(xCode:Byte;data:pointer;
                                  size:integer); override;
end;
```

```
procedure TStar.WriteToStream(Stream: TStream);
var xByte:Byte;
    xInt:Integer;
begin
  inherited;
  xByte := brs; WriteField(90,Stream,xByte,1);
  xInt := brc; WriteField(20,Stream,xInt,4);
  xInt := Radius; WriteField(21,Stream,xInt,4);
  xInt := cValue; WriteField(22,Stream,xInt,4);
```

```
end;


procedure TStar.SetPropertyFromStream(xCode: Byte; data:
pointer;
  size: integer);
begin
   Case xcode of
      20: brc := pInt(data)^;
      21: Radius := pInt(data)^;
      90: brs := pByte(data)^;
      22: cValue := pInt(data)^;
   end;
end;
```

Registering to Object Inspector

If you want your figure to be show in the Object Inspector when it is selected, you should override 3 methods. RegisterToPropertyPage, RefreshPropertypage and FigurePropertyChanged methods will provide you figure field data to be edited in the ObjectInspector. In the RegisterToPropertyPgae method you will register your figure to the ObjectInspector and create properties for each field that you want to see in the InsPector. In the RefreshPropertyPage you will reassign the property values of your fields. And finally FigurePropertyChanged method will be called whenany field is edited in the ObjectInspector so in this method you should assign the property value to your field data. Listing 7 will show the implementations of these methods with their interfaces.

Listing 7 – Registering To ObjectInspector

```
TStar = class(TFigure)
  ...
  Procedure RegisterToPropertyPage;override;
  Procedure RefreshPropertyPage;override;
  Procedure FigurePropertyChanged(PropertyName: string; Data:
PPropData);override;
end;
```

```
procedure TStar.RegisterToPropertyPage;
var fPPage: TObjectInspector;
    a: integer;
begin
  if owner <> nil then begin
    fPPage := TPCDrawing(owner).fPPage;
    fPpObject := fPPage.registerObject(name,'Star',self);
    fPpObject.ChangeEvent := FigurePropertyChanged;
    props[1] := TStringProperty.create('Name',false,name);
    props[2] := TIntegerProperty.create('Handle',true,Handle);
    props[3] := TLineStyleProperty.create('PenStyle',false,
                                                      style);
    props[4] := TLineWidthProperty.create('PenWidth',false,
                                                      width);
    props[5] := TColorProperty.create('PenColor',false,Color);
    props[6] := TBrushStyleProperty.create('BrushStyle',false,
                                                        brs);
    props[7] := TColorProperty.create('BrushColor',false,brc);
    props[8] := TIntegerProperty.create('Radius',false,radius);
    props[9] := TIntegerProperty.create('Convexity',false,
                                                     cValue);
    for a := 1 to 9 do fPpObject.AddProperty(props[a]);
  end;
```

```
end;

procedure TStar.RefreshPropertyPage;
begin
  if fPpObject <> nil then begin
    props[1].Data.ValString := Name;
    props[2].Data.ValInteger := Handle;
    props[3].Data.ValByte := Style;
    props[4].Data.ValByte := Width;
    props[5].Data.ValInteger := Color;
    props[6].Data.ValByte := brs;
    props[7].Data.ValInteger := brc;
    props[8].Data.ValInteger := Radius;
    props[9].Data.ValInteger := cValue;
  end;
end;

procedure TStar.FigurePropertyChanged(PropertyName: string;
  Data: PPropData);
begin
  if PropertyName = 'Name' then begin
    Name := Data.ValString;
    fPpObject.Name := Name;
  end
  else if PropertyName = 'PenStyle' then
    style := Data.ValByte
  else if PropertyName = 'PenWidth' then
    width := Data.ValByte
  else if PropertyName = 'PenColor' then
    color := Data.ValInteger
  else if PropertyName = 'BrushStyle' then
    brs := Data.ValByte
  else if PropertyName = 'BrushColor' then
    brc := Data.ValInteger
  else if PropertyName = 'Radius' then
    radius := Data.ValInteger
  else if PropertyName = 'Convexity' then
    cvalue := Data.ValInteger;
  TPCDrawing(owner).fPPage.ObjectChangedEvent(name);
  if assigned(TPCDrawing(Owner).OnPropertyChanged) then
   TPCDrawing(Owner).OnPropertyChanged(Self,PropertyName,Data);
end;
```

Star in Action

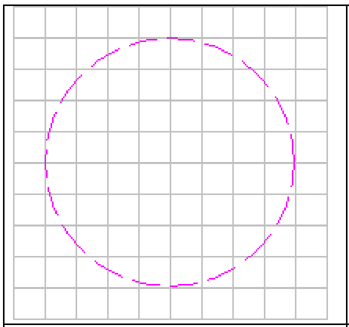
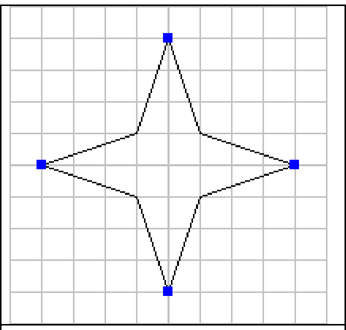
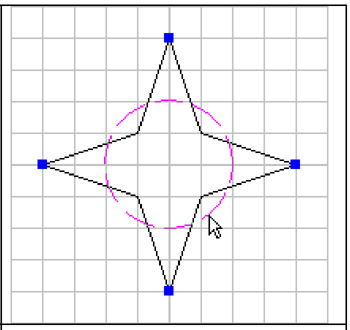
Our Figure is ready to work. We have overridden few methods and we have produced a stand alone functionalty. To make this figure work in Powercad in the OnCreate event code of your main application form write this code to register the Figure Class.

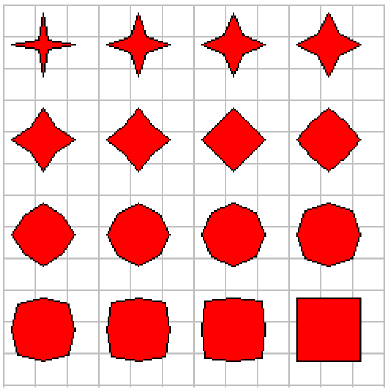
```
PowerCad1.RegisterFigureClass(TStar);
```

Then in the onclick event of the menu or button you specified for TStar, write down this code.

```
PowerCad1.CurrentFigure := 'TStar';
PowerCad1.ToolIdx := toFigure;
```

Now you can feel TStar in action.

		
Drawing Star	*Selected Star*	*Modifying Star*



Same radiuses, different convexities of stars

Using Custom Shadow

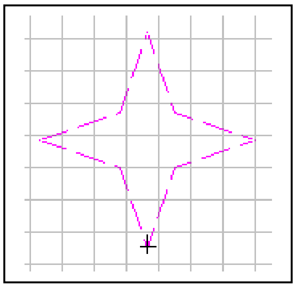
As it is done above, we have used a standart circular shadow for TStar. So the shadow functionality of our figure has behaved like circle. Now the we will create our own shadow and our shadow will behave like the figure. Now we will see creating a custom shadow step by step.

Step 1: Implementing The Figure Shadow

We should override the Shadow methods, because we want our own figure (star) to be drawn when the shadow is active, not a circle. So we should override 3 methods for having a custom shadow. The first is the createShadow method which will be used to retun a shadow to Powercad, the other will be ShadowClick method which will be called in each click of the user when the shadow is active. And the last is ShadowTrace which will be called when mousemoves, so that we will reshape the shadow before creating the figure.

In the CreateShadow method, we will create our own figure. Note that only the fist location is given in this method, we will assume it as center of the star, and the starting radius of the star will 0.

Secondly, we will implemet the ShadowClick, method. Since we need two clicks to form a star, we should return a true value, if the clickindex is 2. The first click will be handled in the CreateShadow method and it will be the center, the second click will be handled in



ShadowClick method. So we will calculate the radius of the star by this click and end the shadow by returning a true result.

Also we should reshape the shadaow between the clicks, when the user moves the mouse, so that he will see the shape of the star before selecting the final location for his click.For this purpose we will reimplement the ShadowTrace method. Since we need only two clicks, this method will be called between the first click and the second click. On the right picture, you can see our custom shadow in action.

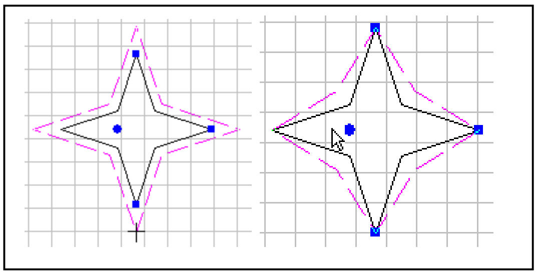
See Listing 8 for the implementation of Shadow Methods.

Listing 8 – The Shadow Methods

```
TStar = class(TFigure)
  ...
  Class Function CreateShadow(x, y: integer): TFigure;override;
  function ShadowClick(ClickIndex, x, y: integer): Boolean;
                                                   override;
  function ShadowTrace(ClickIndex, x, y: integer): Boolean;
                                                   override;
end;
```

```
class function TStar.CreateShadow(x, y: integer): TFigure;
begin
  result := TStar.Create(x,y,0,1,1,clLime,1,clWhite,0,dsTrace,
                                                           nil);
end;

function TStar.ShadowClick(ClickIndex, x, y: integer): Boolean;
begin
  if clickindex = 2 then result := true else result := false;
  radius := round(sqrt( sqr(x - actualPoints[1].x)+
                  sqr (y - actualPoints[1].y) ));
end;

function TStar.ShadowTrace(ClickIndex, x, y: integer): Boolean;
begin
  radius := round(sqrt( sqr(x - actualPoints[1].x)+
                  sqr (y - actualPoints[1].y) ));
end;
```

Step 2 : Implementing the Modification Shadow



In this step, we will also provide our custom shadow in the modification of the star. But first we should decide, what kind of modifications we will provide to the user, so we should create modifivcation points for these purposes. In this Star example, we can provide to modifications, one is the radius of the star, the other is the convexity value of the star. So let us create 3 points for the modification of the radius, on the top,bottom, and right tip of the star. and one point for the modification of the cValue to the location which is d (look at the "Geometry Of Our Star" title in this section for d value) to the center. As we have seen before , the Modification points will be registred in GetModPoints method. The sequence number for the radius points will be 0, and for the convexity point will be 1. So that we will distinguish which point the user is moving. On the upper figure, the first picture shows modificationof the radius, the second one shows modification of the convexity.

See Listing 9 for the implementation of GetModPoints.

Listing 9- GetModPoints Method

```
TStar = class(TFigure)
  ...
  procedure GetModPoints(ModList: TList);override;
end;
```

```
procedure TStar.getModPoints(ModList: TList);
var CControl : TPCDrawing;
    cp: TPoint;
    d: integer;
begin
  CControl := TPCDrawing(Owner);
  cp := actualpoints[1];
  d := round((cValue/100)*radius);
  // There modification point for radius arrangment
  ModList.Add(CControl.RegisterModPoint(self,ptCirclePoint,
              ptRect,clBlue,pointdim,cp.x,cp.y + radius,0));
  ModList.Add(CControl.RegisterModPoint(self,ptCirclePoint,
              ptRect,clBlue,pointdim,cp.x,cp.y - radius,0));
  ModList.Add(CControl.RegisterModPoint(self,ptCirclePoint,
              ptRect,clBlue,pointdim,cp.x+radius,cp.y,0));
  // One modification point for Convexity arrangement
  ModList.Add(CControl.RegisterModPoint(self,ptCirclePoint,
              ptCircle,clBlue,pointdim,cp.x-d,cp.y,1));
end;
```

And for the Shadow Creation of the modification, we will override 3 methods. *CreateModification* will be used for creating the shadow, *TraceModification* will be used to reshape the shadow when the modification is active and *EndModification* will be used for applying the changes to our real figure.

See Listing 10 for the implementation of these methods with their interfaces.

Listing 10 - Modification Methods

```
TStar = class(TFigure)
  ...
  function CreateModification: TFigure;override;
  function TraceModification(CadControl: Pointer;mp:TModPoint;
                    TraceFigure:TFigure;x,y:integer;Shift:
                    TShiftState):boolean;override;
  function EndModification(CadControl: Pointer;mp:TModPoint;
                    TraceFigure:TFigure;x,y:integer;Shift:
                    TShiftState):boolean;override;
end;
```

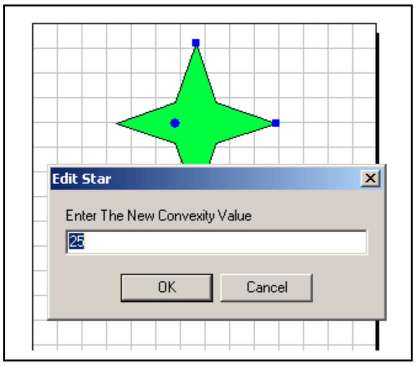
```
function TStar.CreateModification: TFigure;
```

```
begin
   result := TStar.create(actualpoints[1].x,actualpoints[1].y,
                   Radius,1,1,clLime,1,clWhite,0,dsTrace,nil);
   TStar(result).cvalue := cValue;
end;

function TStar.TraceModification(CadControl: Pointer; mp:
         TModPoint;TraceFigure: TFigure; x, y: integer; Shift:
         TShiftState): boolean;
var cp: Tpoint;
begin
   if mp.SeqNbr = 0 then
   begin
     TStar(TraceFigure).Radius :=
          round(sqrt( sqr(x - TraceFigure.actualPoints[1].x)+
                sqr(y - TraceFigure.actualPoints[1].y) ));
   end else begin
     cp := TraceFigure.ActualPoints[1];
     if x > cp.x then x := cp.x;
     TStar(TraceFigure).cValue := round(((cp.x-x)/radius)*100)
   end;
end;

function TStar.EndModification(CadControl: Pointer; mp:
         TModPoint;TraceFigure: TFigure; x, y: integer; Shift:
         TShiftState): boolean;
begin
   Radius := TStar(TraceFigure).Radius;
   cValue := TStar(TraceFigure).cValue;
   ResetRegion;
   Modified := True;
end;
```

Let Us Do Something More With Our Star

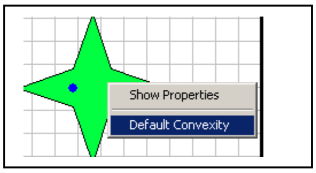
In this section, we will apply some more functionality to our figure. PowerCad custom figure interface provides developers some more features like double-click editing and pop-up menu.

When the user double clicks the figure , PowerCad calls the Edit method of the figure. If you have overridden this method, then your code in this method will work. In this method you can provide prompting forms for the user to enter some values for editing your figure. So we will reimplement the Edit method, and in this method we will prompt an Input Form to take the Convexity value from the user. If the user changes the current value then we should returna true value to Invoke powercad to refresh the drawing.



Enter The New Convexity Value
25

When the user right clicks on the editor, a pop-up menu is shown. In this menu, PowerCad provides some shortcuts to do general ections. If you override the UpdateMenu method , PowerCad can provide your figure specific menu items to do specific actions.



We will reimplemet this method with the MenuClicked procedure, to provide user a shortcut for setting the convexity to its default value. If you override the UpdateMenu method, then you should also override the MenuClicked method which is called when the user clicks on the pop-up menu.

Listing 11 Shows the implementation of the Edit and UpdateMenu methods with their interfaces.

Listing 11 – Edit and UpdateMenu Methods

```
TStar = class(TFigure)
  ...
  Function Edit:Boolean;override;
  Procedure UpdateMenu(var PopMenu: TPopUpMenu;var sIndex:
                                             integer);override;
  Procedure MenuClicked(CommandId:integer);override;
end;
```

```
Function TStar.Edit: Boolean;
var res: string;
    val: integer;
begin
  res := InputBox('Edit Star',
                  'Enter The New Convexity Value',
                   inttostr(cvalue));
  if res <> inttostr(cvalue) then
  begin
    val := strtointdef(res,cValue);
    if (val <> cvalue) then
    begin
      cValue := Val;
      result := true;
      ResetRegion;
      Modified := true;
    end;
  end;
end;

procedure TStar.UpdateMenu(var PopMenu: TPopUpMenu; var sIndex:
                                                      integer);
var mnItem: TMenuItem;
begin
   menuIndex:= sIndex;
   mnItem := TMenuItem.Create(PopMenu);
   mnItem.Caption := 'Default Convexity';
   mnItem.Tag := sIndex;
   PopMenu.Items.Add(mnItem);
   sIndex := sIndex+1;
end;

procedure TStar.MenuClicked(CommandId: integer);
var idx: integer;
begin
  idx := commandID-menuIndex;
  case idx of
    0: begin
         cValue := 25;
         ResetRegion;
         Modified := True;
       end;
  end;
end;
```

14 Component Reference

The Component Reference is published as a different document. Please download it from Tekhnelogos official website www.tekhnelogos.com .