Study Guide

embarcadero

GETTING STARTED WITH WINDOWS AND MAC DEVELOPMENT

Designing a High Definition User Interface

E-Learning Series Course Book

Lesson 5

Embarcadero Technologies

Americas Headquarters
100 California Street, 12th Floor
San Francisco, California 94111

EMEA Headquarters
York House
18 York Road
Maidenhead, Berkshire
SL6 1SF, United Kingdom

Asia-Pacific Headquarters
L7. 313 La Trobe Street
Melbourne VIC 3000
Australia

Lesson 5 – Designing a High Definition User Interface

Version: 0.9
Presented: May 31, 2012
Last Updated: June 5, 2012
Prepared by: David Intersimone "David I", Embarcadero Technologies

davidi@embarcadero.com
http://blogs.embarcadero.com/davidi/

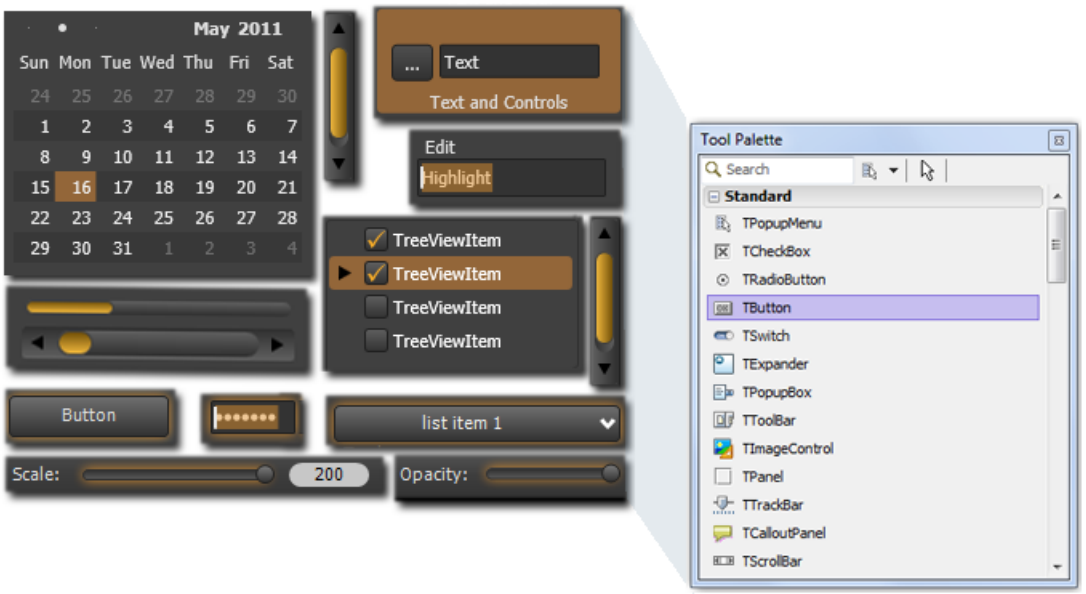
Contents

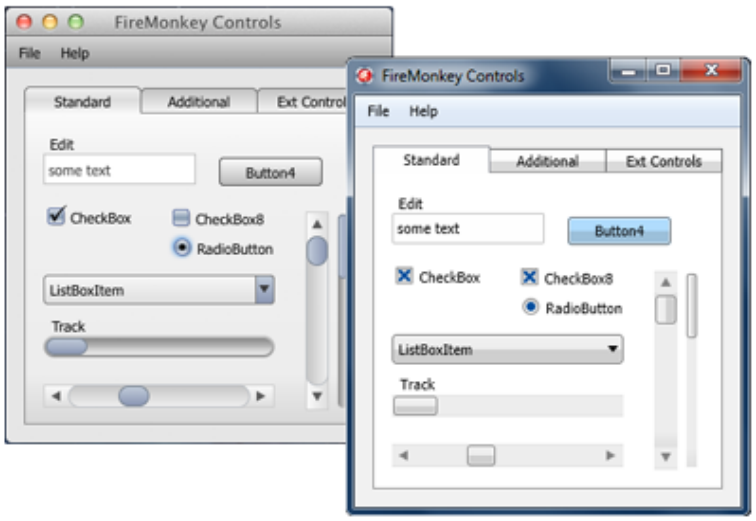
Introduction

FireMonkey includes a full suite of drag-and-drop user interface controls. Select from buttons, menus, HUDs, text, combo boxes, tables, tabs, panels and more to design your user interface. Controls are fully customizable and can be styled to your liking using Fire Monkey HD styles.

FireMonkey HD styles enable you to fully control the look and feel of HD user interfaces without programming or becoming an expert in esoteric markup languages. Styles can be created or modified by developers or designers. Choose from a library of existing styles, create custom user interfaces for your applications or if you're after a more traditional native OS look, FireMonkey gives you the option to stick with Windows 7, Mac OS X, and iOS UIs.

In lesson 5 we will focus on designing a Windows and Macintosh HD UI and follow the steps to select styles for your application. You'll also learn how you can customize the look and feel of a particular component by customizing the style.



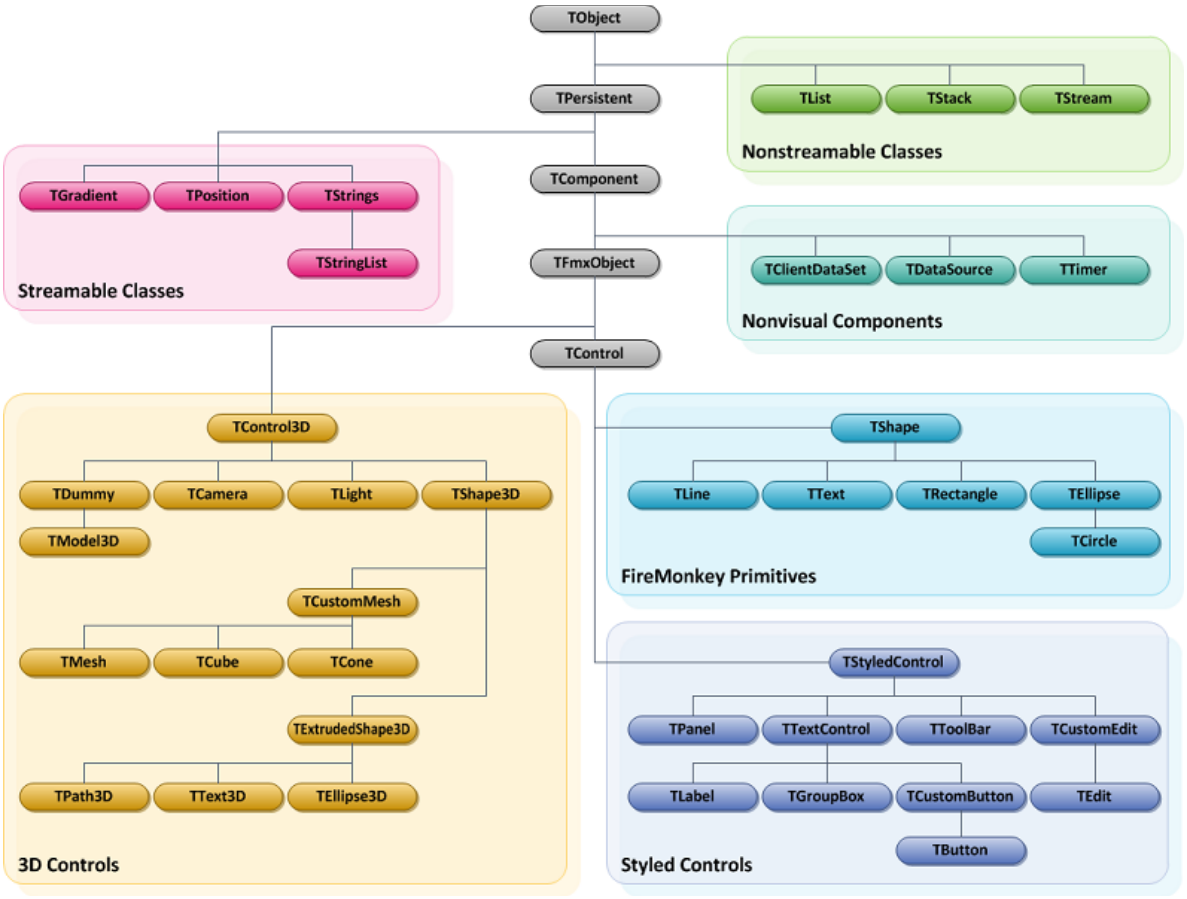


FireMonkey Business Application Platform (FMX)

FMX is the unit scope that contains the units and unit scopes of the Fire Monkey application platform. FireMonkey leverages the graphics processing unit (GPU) in modern desktop and mobile devices to create visually engaging applications on multiple platforms, targeting the entire range from the personal to the enterprise. Major features of Fire Monkey include:

- Cross-platform abstraction layer for OS features like windows, menus, timers, and dialogs
- 2D and 3D graphics
- Powerful vector engine (like Adobe Flash or Microsoft WPF)
- Fast real-time anti-aliased vector graphics; resolution independent, with alpha blending and gradients
- WYSIWYG designer and property editors
- Advanced GUI engine - window, button, textbox, numberbox, memo, anglebox, list box, and more
- Advanced skinning engine based on vector graphics styles with sample style themes
- Shape primitives for 2D graphics along with a built-in set of brushes, pens, geometries, and transforms
- Advanced animations calculated in background thread; easy to use and accurate, with minimal CPU usage and automatic frame rate correction
- Bitmap effects rendered in software, including drop shadows and blurring
- Flexible layouts and compositing of shapes and other controls
- Layered forms, Unicode-enabled
- JPEG, PNG, TIFF, and GIF format read/write support
- Multi-language engine, editor and examples

The following figure shows the relationship of some key classes that make up the FireMonkey hierarchy. You can also download a FireMonkey architecture schematic poster (PDF file) at http://www.embarcadero-info.com/firemonkey/firemonkey_chart_poster.pdf.



HD Application and Component Rendering

To understand how an HD application and its components are rendered, start by exploring the FMX class hierarchy. TFmxObject branches from TComponent to form the FMX root, providing the object lifecycle. From that comes TControl, which encapsulates a canvas and adds painting.

The two relevant branches of TControl are the primitive classes in the FMX.Objects unit, and the user-interaction styled controls in FMX.Controls, FMX. ExtCtrls, and other units. The primitives include such objects as shapes and images:

- TShape
- TImage
- TPaintBox

and their descendants:

- TLine
- TRectangle
- TRoundRect
- TText

Styled controls descend from TStyledControl and include:

- TPanel
- TLabel
- TCheckBox
- TImageControl
- TCalendar

Painting versus Compositing

Primitives override the TControl.Paint procedure and draw directly on the canvas. Each style that embodies a TStyledControl is an arrangement of a tree of subcontrols and primitives. A style eventually resolves to a layered set of primitives, and those primitives draw in turn on the canvas to render the control. Components can be drawn with one technique or the other, or a combination of the two with a control that contains custom primitives.

Subclasses of TStyledControl will attempt to find their style-resource among those assigned to the form's StyleBook property, using a simple search routine based on class names in TStyledControl.GetStyleObject.

Application developers can always customize controls by creating a style with the proper name. That style will automatically apply to all instances of the control. They can also assign styles to individual controls by setting the StyleLookup property.

Style theme artists can include styles for custom controls by creating a style with the proper name in their theme.

To define the appearance of the control when there is no matching style, a component writer can override GetStyleObject. For example, platform-specific styles can be bundled as RCDATA:

- Save each platform's finished style as a .style file.
- Create .rc files for the project, referencing the appropriate .style file as RCDATA. The .rc files will be compiled to .res files.
- Use conditional compilation directives to include each platform's .res file.
- In GetStyleObject, find that RCDATA as a stream and use CreateObjectFromStream to reconstitute the style as the result.

FireMonkey HD Application

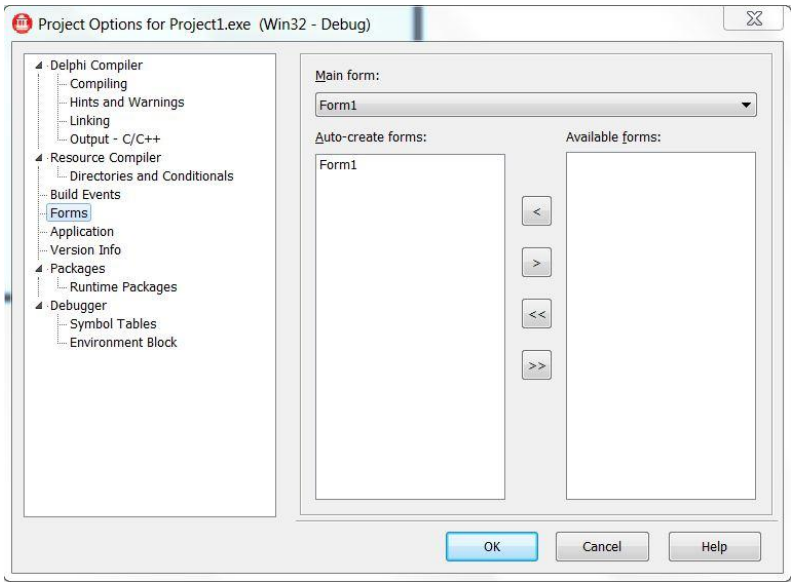
Use the menu item **File > New > FireMonkey HD Application – Delphi** or **File > New > FireMonkey HD Application – C++Builder** to create a starting FireMonkey HD Application. The project wizard creates the framework for a FireMonkey application and opens the Form Designer, displaying the base form

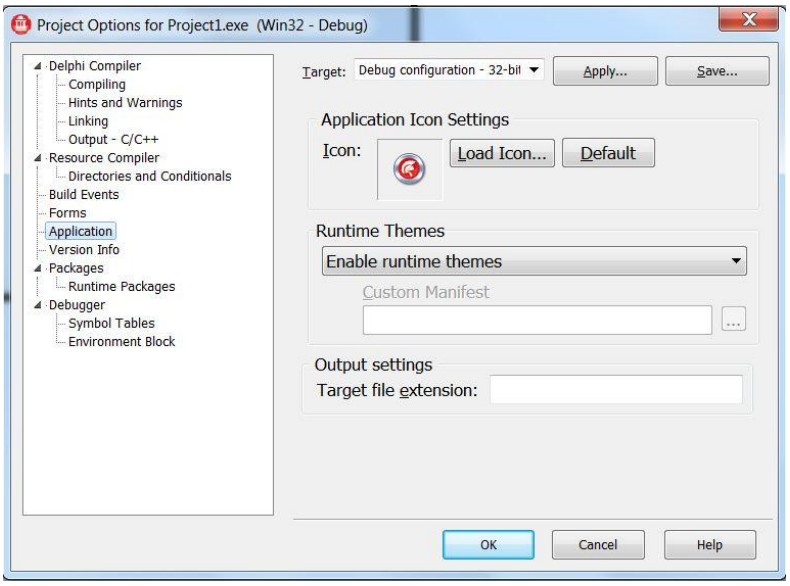
(FMX.Forms.TForm). For every FireMonkey application, the form file has the extension .fmx (instead of .dfm, the extension used for a Windows-only VCL form file).

FMX.Forms.TApplication

TApplication encapsulates a windowed application. The methods and properties introduced in TApplication reflect the fundamental actions related to creating, running, sustaining, and destroying a FireMonkey application on Windows and Mac OS X operating systems.

Each GUI application automatically declares an Application variable as the instance of the application. TApplication does not appear on the Component palette, nor is it available in the form designer to visually manipulate; so it has no published properties. Nevertheless, some of its public properties can be set or modified at design time in the Forms and Application pages of the **Project > Options** dialog box.





For application-wide properties and methods that affect the display, see TScreen.

FMX.Forms.TScreen

The TScreen (defined in the FMX.Forms unit) represents the state of the screen in which an application runs. TScreen introduces properties that specify:

- The number of forms and data modules in the application
- Lists of forms and data modules that have been instantiated by the application.

```
Delphi:

Label1.Text :=
  '# of Forms in the Application: '
  + IntToStr(Screen.FormCount);
Label2.Text :=
  '# of Data Modules in the Application: '
  + IntToStr(Screen.DataModuleCount);

C++:

Label1->Text = "# of Forms in the Application: "
  + IntToStr(Screen->FormCount);
Label2->Text = "# of DataModules in the Application: "
  + IntToStr(Screen->DataModuleCount);
```

FireMonkey Application Design

FireMonkey uses lightweight GUI controls on top of a cross-platform abstraction, which is implemented for Windows, Mac OS X, and iOS. Lightweight controls mean that every pixel is drawn by FireMonkey; no

native (heavyweight) controls are used. This approach favors fidelity across platforms over fidelity to the host platform, side-steps the "least common denominator" problem of heavyweight cross-platform frameworks, and allows FireMonkey to create its own control and application design rules.

Using the FireMonkey Coordinate System

In the FireMonkey Form Designer, the origin of the coordinate system is the top-left, extending to the bottom-right. Coordinates are expressed as single-precision floating-point numbers. All supported platforms use square pixels. One coordinate unit usually maps to one pixel, with some distinct exceptions:

- The Position property of a 2D control is a TPosition with X and Y properties. The separate Width and Height properties represent its size.

- 3D objects use a TPosition3D with an additional Z property, with positive values pointing into the screen (X goes to the left and Y points down, so this follows the "right-hand rule"); and a Depth property. Together, the position and size define one kind of bounding box that describes a control: its content box. We'll cover FireMonkey 3D applications in Lesson 7.

FireMonkey Controls Have Owners, Parents, and Children

FireMonkey allows any control to be the parent of another. The form usually owns all the controls in it, and controls laid out in the Form Designer follow this convention.

When creating components through code, if the control is intended to persist through the remaining lifetime of the form, you specify the form as the owner. The form should be readily available either as **Self (Delphi), this(C++)** or as the Owner of an existing control. The owner is responsible for disposing of the control when it is disposed itself.

For components that are transient, pass nil as the owner. The code is then responsible for disposing of the component when it is finished with it. Best practices dictate that a try/finally block is used to ensure the component is disposed of, even if there is an exception.

In order for the control to appear in the form, ownership is not enough. It must also be placed in the component tree, either as the direct child of the form, or somewhere further down the tree. Controls laid out in the Form Designer do this automatically, and the component tree is shown in the Structure View. When creating controls through code, you set the **Parent** property to the form or the appropriate parent control.

The Position of a child is relative to its Parent. If the coordinates are zero, the child starts at the same top-left as the parent.

Parentage is not restricted to container-like controls. Also, the **ClipChildren** property defaults to False (if True, it would not allow drawing of children outside control's content box). This enables ad-hoc

collections of related controls without requiring a formal container. For example, a TLabel can be a child of the **TEdit** it describes. The label can have a negative position, placing it above or before the control. Moving the **TEdit** moves both together. **TLayout** can be used as an otherwise featureless container to arrange other controls.

In addition to a shared coordinate space, child objects share other attributes like visibility, opacity, rotation, and scale. Changing these attributes of the parent affects all the children in that sub-tree.

Aligning with Margins and Padding

A control's Align property determines its participation in automatic positioning and/or sizing along its parent's four sides or center, both initially and as the parent is resized. It defaults to alNone, so that no such automatic calculations are performed, the control stays where it is. The property is an enum of type **TAlignLayout**, with over a dozen other possible values.

TAlignLayout can have one of the following values:

- alNone - The control remains where it was placed. This is the default value. No automatic positioning and sizing are performed.
- alTop - The control moves and pins to the top of its parent and resizes to fill the width of its parent. The height of the control is not affected. If another most side-pinned control already occupies part of the parent area, the control resizes to fill the remaining width of its parent.
- alBottom - The control moves and pins to the bottom of its parent and resizes to fill the width of its parent. The height of the control is not affected. If another most side-pinned control already occupies part of the parent area, the control resizes to fill the remaining width of its parent.
- alLeft - The control moves and pins to the left side of its parent and resizes to fill the height of its parent. The width of the control is not affected. If another side-pinned control already occupies part of the parent area, the control resizes to fill the remaining height of its parent.
- alRight - The control moves and pins to the right side of its parent and resizes to fill the height of its parent. The width of the control is not affected. If another side-pinned control already occupies part of the parent area, the control resizes to fill the remaining height of its parent.
- alMostTop - The control moves and pins to the top of its parent, set to be the topmost, and resizes to fill the width of its parent. The height of the control is not affected.
- alMostBottom - The control moves and pins to the bottom of its parent, set to be the bottommost and resizes to fill the width of its parent. The height of the control is not affected.
- alMostLeft - The control moves and pins to the left side of its parent, set to be the leftmost and resizes to fill the height of its parent. The width of the control is not affected. If another most side-pinned control already occupies part of the parent area, the control resizes to fill the remaining height of its parent.
- alMostRight - The control moves and pins to the right side of its parent, set to be the rightmost and resizes to fill the height of its parent. The width of the control is not affected. If another most side-pinned control already occupies part of the parent area, the control resizes to fill the remaining height of its parent.
- alClient - The control resizes to fill the client area of its parent. If another side-pinned control already occupies part of the parent area, the control resizes to fit within the remaining parent area.

- alContent - The control resizes to fill the entire bounds of its parent, overlapping it.
- alCenter - The control moves to the center of the parent area. The control's size is not affected. If another side-pinned control already occupies part of the parent area, the control moves to the center of the remaining parent area.
- alVertCenter - The control is centered vertically within the client area of the parent and resizes to fill the width of its parent. The height of the control is not affected. If another side-pinned control already occupies part of the parent area, the control resizes to fill the remaining width of its parent.
- alHorzCenter - The control is centered horizontally within the client area of the parent and resizes to fill the height of its parent. The width of the control is not affected. If another side-pinned control already occupies part of the parent area, the control resizes to fill the remaining height of its parent.
- alHorizontal - The control resizes to fill the height of its parent. The width of the control is not affected. If another side-pinned control already occupies part of the parent area, the control resizes to fill the remaining height of its parent.
- alVertical - The control resizes to fill the width of its parent. The height of the control is not affected. If another side-pinned control already occupies part of the parent area, the control resizes to fill the remaining width of its parent.

Most of the alignments cause the calculation to include two values for automatic alignment: the parent's Margins and the control's Padding.

Margins set aside space on the interior of the parent's content box; much like margins on a printed page, from the perspective of the paper. For example, if the parent's Top and Left margin are both 10, then a component that is automatically positioned in the top-left will have its position set to 10,10.

More accurately, what is automatically positioned is not the control's content box, but rather its layout box. The difference between the two is the control's Padding, if any. Padding sets aside space on the exterior of the control's content box. As it increases, the size of the layout box stays the same, and the content box shrinks if it is constrained. Going back to the 10,10 example, if the Top and Left padding are both 5, then the position of the control will be 15,15.

Padding ensures separation between controls automatically positioned by a parent, and margins ensure space between those controls and the parent's edge. That's for positive values in margins and padding; negative values are also allowed. Negative margins place children outside the parent's content box, which are still rendered if its ClipChildren property is False. Negative padding places a control's content box outside its computed layout box.

Scaling and Rotating

Two other commonly available attributes affect a control's final rendered location: scale and rotation. Scale and rotation do not alter a control's position or size properties. This is reflected in the Form Designer: a selected object's eight grip dots (four corners and four sides) mark the actual content box, set manually or computed through layout, before applying scale and rotation.

A control's Scale property is represented by an instance of the same type as its Position: TPosition for 2D objects and TPosition3D for 3D objects. Its X, Y, and Z values default to 1, meaning that the object is unscaled in all dimensions. The scale value is a simple multiplier on each axis. Values larger than one will stretch along that axis. Values less than one but greater than zero will shrink or squish along that axis. Scaling any axis by zero will cause the control to disappear. Uniform scaling requires the same value in all axes.

2D scaling is always anchored from the control's origin, the top-left of its content box. Negative scaling pivots on that origin point. For example, a negative X scale will cause the control to render down and to the left, flipping it on the content box's left edge. 3D scaling is from the object's center (Lesson 7 covers creating 3D FireMonkey applications).

In 2D rotation, the pivot is adjustable. The **RotationCenter** property is also a **TPosition**, but the value is in unit coordinates: 0,0 is the top-left of the control and 1,1 is the bottom-right. It defaults to the center of the control: 0.5,0.5. The aspect ratio of the content box does not matter. On that pivot point, the **RotationAngle** is in degrees, clockwise.

In 3D, rotation is always from the center, with the RotationAngle a TPosition3D, specifying degrees on the X, Y, and Z axes. Rotation also follows the right-hand rule; for example, with X and Y rotation zero, the Z axis points into the screen, and positive rotation on the Z axis rotates clockwise.

In 2D, scaling occurs before rotation, which matters because scaling is from the origin and the rotation is adjustable. In 3D, both occur from the center, so the order does not matter.

Layouts and Scaled Layouts

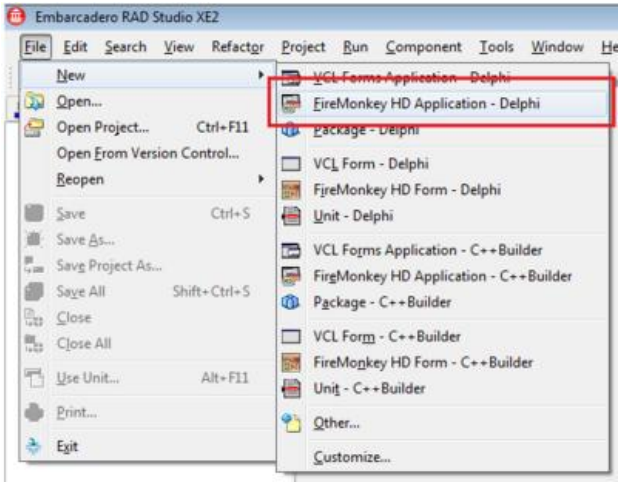
A layout is a container for other graphical objects. Use the layouts when you need to organize multiple graphical controls under the same parent.

For instance, you can use these layouts when you need to create rich FireMonkey applications with many graphical controls that are grouped on the same layer. You can set the visibility of all the controls on a layout at once by affecting the visibility of the layout.

Using Layouts to Arrange Components

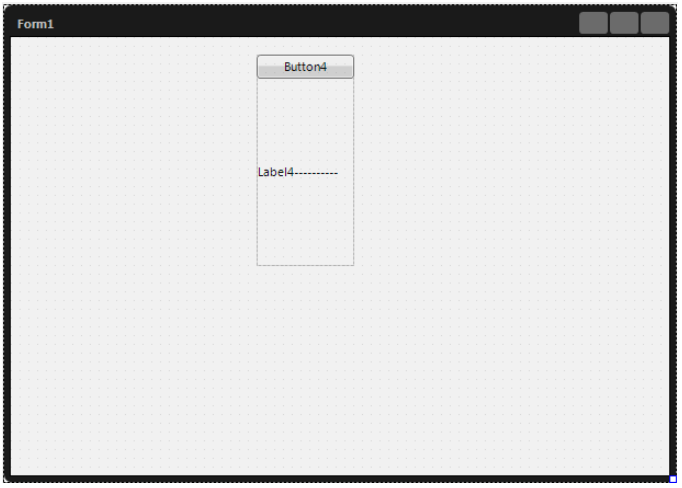
This tutorial demonstrates how to use FireMonkey layouts to arrange 2D components in a round pattern.

1. Select **File > New > FireMonkey HD Application – Delphi** or **File > New > FireMonkey HD Application – C++Builder**.



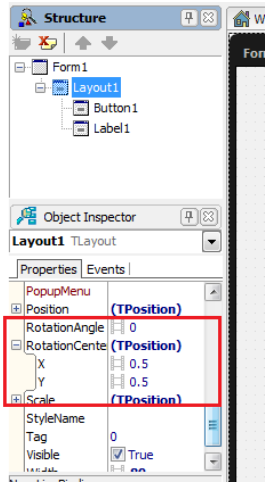
2. Add a TLayout to the form.

3. Add a TButton and TLabel to the form and parent them to the TLayout in the Structure View.

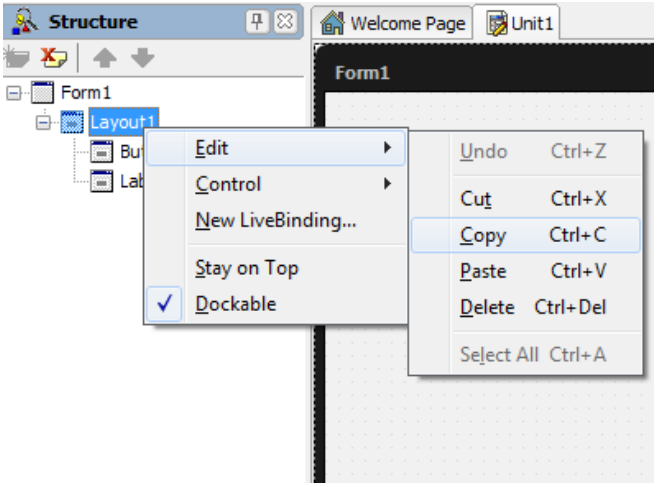


4. In the Object Inspector, make the following changes:

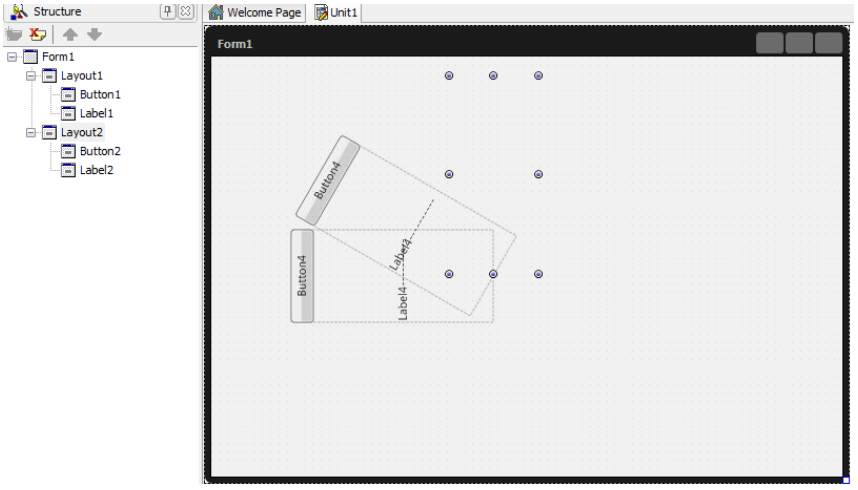
- For the button, set Align to alMostTop.
- For the label, set Align to alVertCenter.
- For the layout, move the rotation center point in the middle of the bottom edge by setting the RotationCenter.X to 0.5 and RotationCenter.Y to 1.
- Rotate the layout by setting the RotationAngle to -90



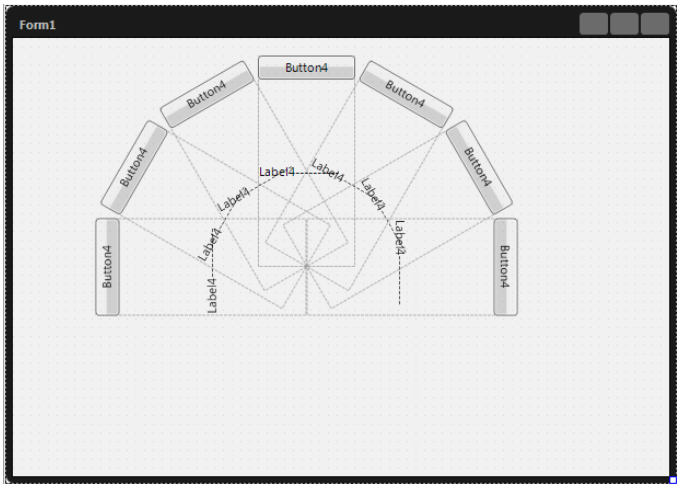
5. In the Structure View, right-click the layout and follow the steps in the images below to copy and paste the layout on the form.



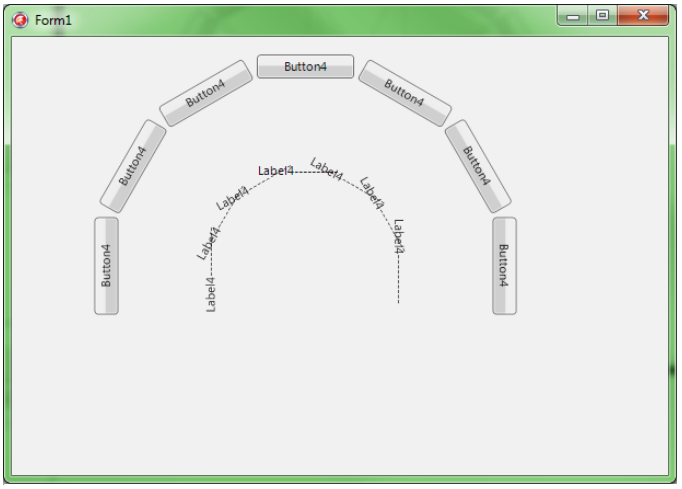
6. Rotate the second layout by setting the RotationAngle to -60



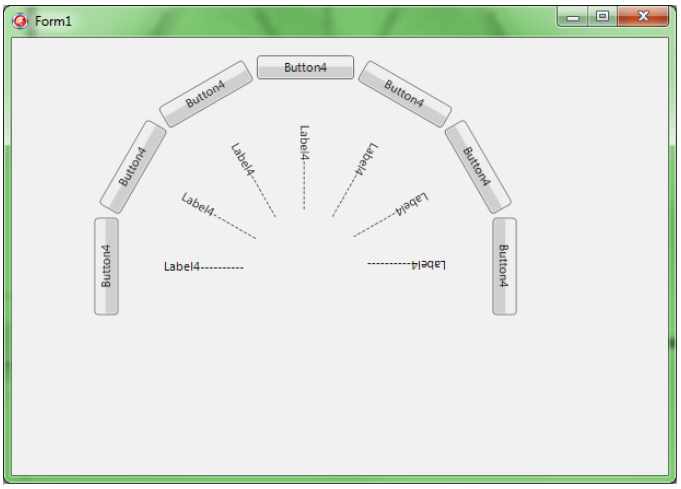
7. Continue copying and pasting the last modified layout, and change its RotationAngle with a value with 30 grades higher than the previous layout, until you reach a RotationAngle of 90.
This is the final pattern:



8. Run the project by pressing F9. The result should look like this:



9. To obtain a different visual effect, go to the Object Inspector, and change the RotationAngle of each panel on the form, to 90:



Using Layouts to Create a Scaled Effect

This tutorial demonstrates how to use FireMonkey layouts to scale a set of buttons at the same.

1. On the form created in the previous tutorial, add a **TScaledLayout** and a **TTrackBar**.

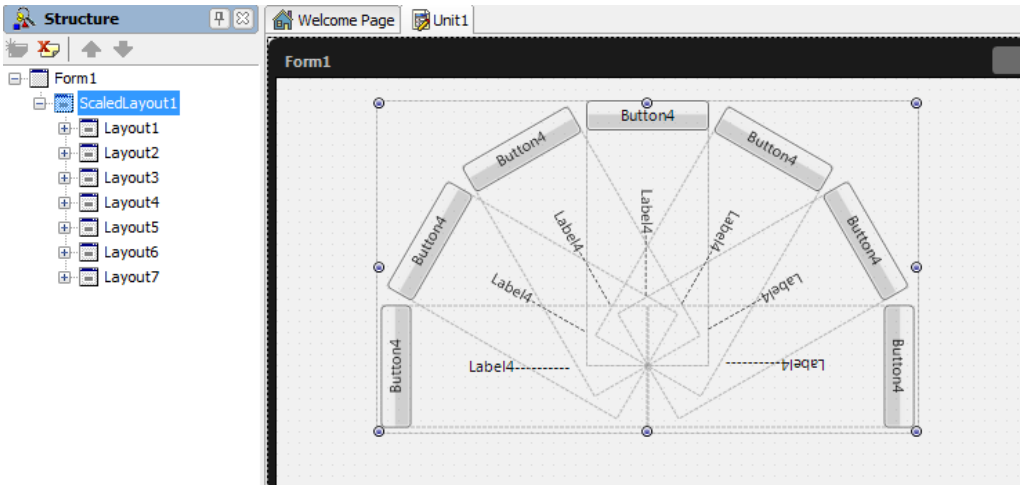
2. In the Object Inspector, make the following changes:

- For the track bar:
  - Set Align to alMostBottom.
  - Set Max property to 2.
  - Set Frequency property to 0.01.
- For the scale layout, set Align to alCenter.

3. In the Structure View, select all the layouts (Ctrl+Click each layout).

4. Drag and drop the layouts under the TScaledLayout.

5. In the Form Designer, resize the TScaledLayout so that the entire pattern fits within the TScaledLayout. With all the layouts selected, center the entire pattern within the TScaledLayout.



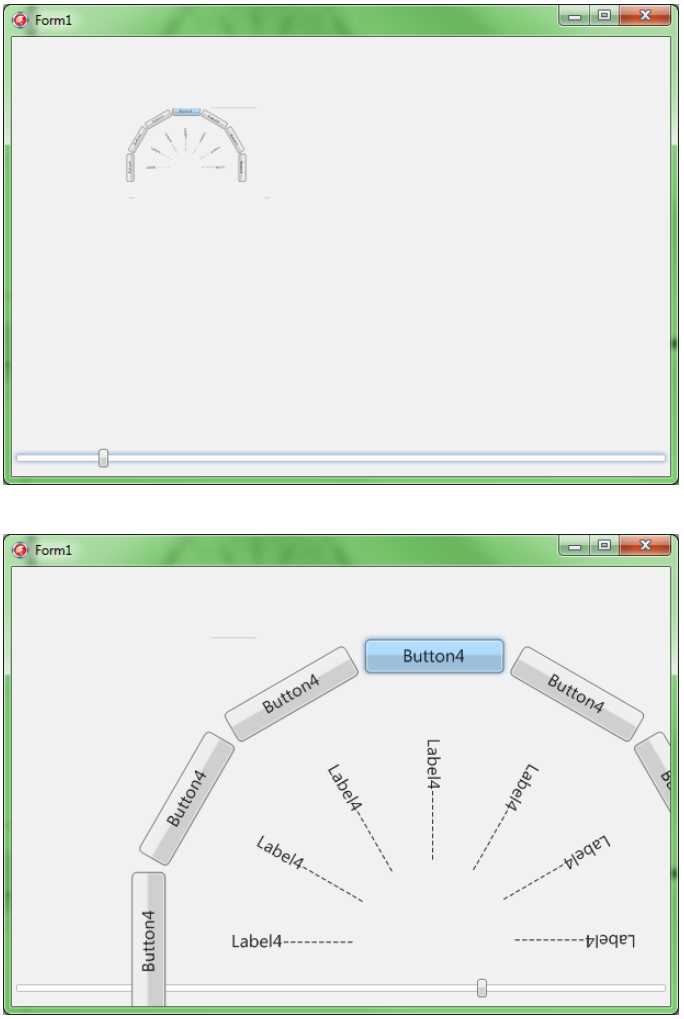
6. In the Object Inspector, scale the TScaledLayout as follows:

- Set the coordinates of the Scale property to 0.5.
- Set the Value property of the track bar to 0.5.

7. Double-click the TTrackBar to attach OnChange event handlers to it.

```
// Delphi version of the implementation
procedure TForm1.TrackBar1Change(Sender: TObject);
begin
  ScaledLayout.Scale.X := TrackBar.Value;
  ScaledLayout.Scale.Y := TrackBar.Value;
end;

// C++ version of the implementation
void __fastcall TForm3D1::TrackBar1Change(TObject *Sender)
{
  ScaledLayout->Scale->X = TrackBar->Value;
  ScaledLayout->Scale->Y = TrackBar->Value;
}
```

8. Run the project by pressing F9. The results should look like this:



When the buttons are scaled, they are not disabled. They remain active, regardless of the scale being used.

Opacity

The Opacity property of a FireMonkey visual control allows you to specify the control's transparency. Opacity also applies to the control's children. Opacity takes values between 0 and 1. If Opacity is 1, the control is completely opaque; if it is 0, the control is completely transparent. The values over 1 are treated as 1, and the ones under 0 are treated as 0.

TCanvas

FireMonkey's TCanvas class provides access to the drawing area of the form. Use **Canvas** to draw directly on the client area of the form.

TCanvas provides properties, events, and methods that assist in creating an image by:

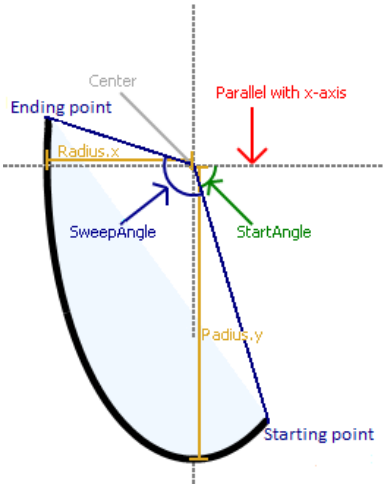
- Specifying the type of brush, stroke, and font to use.
- Drawing and filling a variety of shapes and lines.
- Writing text.
- Rendering graphic images.
- Enabling a response to changes in the current image.

The TCanvas drawing functions are:

- DrawArc
- DrawBitmap
- DrawEllipse
- DrawLine
- DrawPath
- DrawPolygon
- DrawRect
- DrawRectSides
- DrawThumbnail

FMX Canvas DrawArc Example (Delphi and C++)

This example shows how to use the DrawArc and FillArc functions and their results.

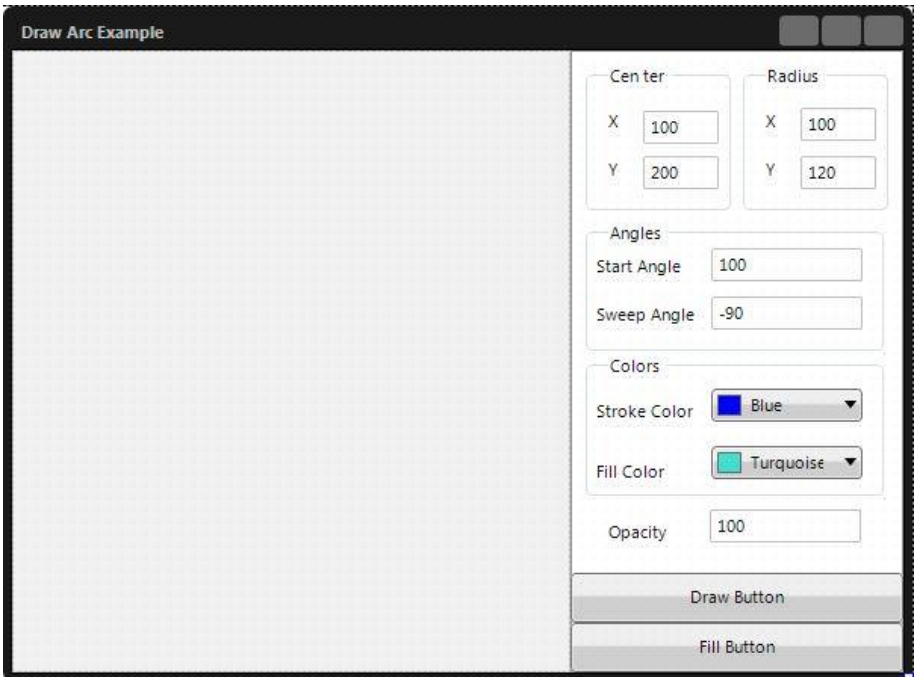


To build and test this example, create an application using **File > New > FireMonkey HD Application - Delphi** or **File > New > FireMonkey HD Application - C++Builder.** Add the following controls on the form:

- A TImage. Set the Align property to alMostLeft and resize the width of the TImage to occupy about 70% of the form width.

- A TPanel. Set the Align property to alClient. The TPanel will take up the remaining space on the form. Inside the TPanel drop
    - A TGroupBox. Set the group box Text property to "Center". Place this group box in the upper left of the TPanel and resize it to occupy about half of the width of the TPanel. Inside the group box drop:
        - Two TEdit objects to set the coordinates for the center of the arc. Name them CenterXEdit and CenterYEdit. Set the Text properties for CenterXEdit and CenterYEdit to 100 and 200 respectively.
        - Two TLabels. Set their Text property to X and Y.
    - A TGroupBox. Set the group box Text property to "Radius". Place this group box to the right of the "Center" group box and resize it to occupy the remaining half of the TPanel. Inside the group box drop:
        - Two TEdit objects to set the rays of the arc. Name them RadiusXEdit and RadiusYEdit. Set the RadiusXEdit and RadiusYEdit Text properties to 200 and 220 respectively.
        - Two TLabels. Set their Text property to X and Y.
    - A TGroup Box. Set the Group Box Text property to "Angles". Resize it to occupy the width of the TPanel. Place this group box just below the "Center" and "Radius" group boxes. Inside the group box drop:
        - Two TEdit objects to set the start angle and sweep angle. Name them StartAngleEdit and SweepAngleEdit. Set the StartAngleEdit and SweepAngleEdit Text properties to 100 and -90 respectively.
        - Two TLabels. Set their Text property to "Start Angle" and "Sweep Angle".
    - Below the group boxes drop
        - A TEdit object to set the opacity. Name it OpacityEdit. Set the OpacityEdit Text property to 100 (Note: 100 = 100% Opaque. 0 = invisible)
        - To the left of the OpacityEdit drop a TLabel. Set the Text property to "Opacity".
    - A TGroupBox. Set the Group Box Text property to "Colors". Inside the group box drop:
        - Two TColorComboBoxes to set the colors for drawing the stroke and filling. Name them StrokeColorComboBox and FillColorComboBox. Using the Object Inspector, set the Color properties for each to Blue and Turquoise respectively (or choose your favorite colors).
        - Two TLabel components. Position them to the left of each TColorComboBox and set their text properties to "Stroke Color" and "Fill Color".
    - Below the "Colors" group box drop two TButtons for drawing and filling the arc. Name them DrawButton and FillButton. Set their Text properties to "Draw Button" and "Fill Button". Set the FillButton Align property to alMostBottom. Then set the DrawButton Align property to alMostBottom.

After adding the controls, renaming them and laying them out, your form should look something like the following:



The DrawArc example needs three Event Handlers: FormCreate, FillButtonClick and FillArcClick. Using the Object Inspector, select the Form and in the Event tab, double click on the OnCreate event to create an event handler. Add the code listed below for the FormCreate event handler. Using the Object Inspector, select the DrawButton and in the Event tab, double click on the OnClick event to create an event handler. Add the code listed below for the DrawButtonClick event handler. Using the Object Inspector, select the FillButton and in the Event tab, double click on the OnClick event to create an event handler. Add the code listed below for the FillButtonClick event handler.

The example draws and fills an arc on the canvas of the bitmap. The bitmap is displayed on the TImage.

```
// Delphi
// Add the following code to the OnCreate event handler
//  of the form.
procedure TForm1.FormCreate(Sender: TObject);
begin
  //initializes the bitmap
  Image1.Bitmap.Create(Round(Image1.Width),
   Round(Image1.Height));
end;
// Add the following code to the OnClick event handler
//  of the DrawButton and FillButton:
procedure TForm1.DrawButtonClick(Sender: TObject);
var
  Center, Radius: TPointF;
  Opacity, StartAngle, SweepAngle: Single;
begin
  // takes the information from the edits
  // checks whether all the values are valid
  if (TryStrToFloat(CenterXEdit.Text, Center.X) and
    TryStrToFloat(CenterYEdit.Text, Center.Y) and
    TryStrToFloat(RadiusXEdit.Text, Radius.X) and
    TryStrToFloat(RadiusYEdit.Text, Radius.Y) and
```

```
    TryStrToFloat(StartAngleEdit.Text, StartAngle) and
    TryStrToFloat(SweepAngleEdit.Text, SweepAngle) and
    TryStrToFloat(OpacityEdit.Text, Opacity)) then
  begin
    Image1.Bitmap.Canvas.BeginScene;
    Image1.Bitmap.Canvas.Stroke.Color :=
      StrokeColorComboBox.Color;
    // draws the arc
    Image1.Bitmap.Canvas.DrawArc(Center, Radius, StartAngle,
      SweepAngle, Opacity);
    Image1.Bitmap.BitmapChanged;
    Image1.Bitmap.Canvas.EndScene;
  end
  else
    // displays message if not all edits have numerical values
    ShowMessage('All Edits text should be numbers')
end;

procedure TForm1.FillButtonClick(Sender: TObject);
var
  Center, Radius: TPointF;
  Opacity, StartAngle, SweepAngle: Single;
begin
  // takes the information from the edits
  // checks whether all the values are valid
  if (TryStrToFloat(CenterXEdit.Text, Center.X) and
    TryStrToFloat(CenterYEdit.Text, Center.Y) and
    TryStrToFloat(RadiusXEdit.Text, Radius.X) and
    TryStrToFloat(RadiusYEdit.Text, Radius.Y) and
    TryStrToFloat(StartAngleEdit.Text, StartAngle) and
    TryStrToFloat(SweepAngleEdit.Text, SweepAngle) and
    TryStrToFloat(OpacityEdit.Text, Opacity)) then
  begin
    Image1.Bitmap.Canvas.BeginScene;
    Image1.Bitmap.Canvas.Fill.Color := FillColorComboBox.Color;
    Image1.Bitmap.Canvas.FillArc(Center, Radius, StartAngle,
      SweepAngle, Opacity);
    Image1.Bitmap.BitmapChanged;
    Image1.Bitmap.Canvas.EndScene;
  end
  else
    // displays a message if not all edits have numerical values
    ShowMessage('All Edits text should be numbers')
end;
```

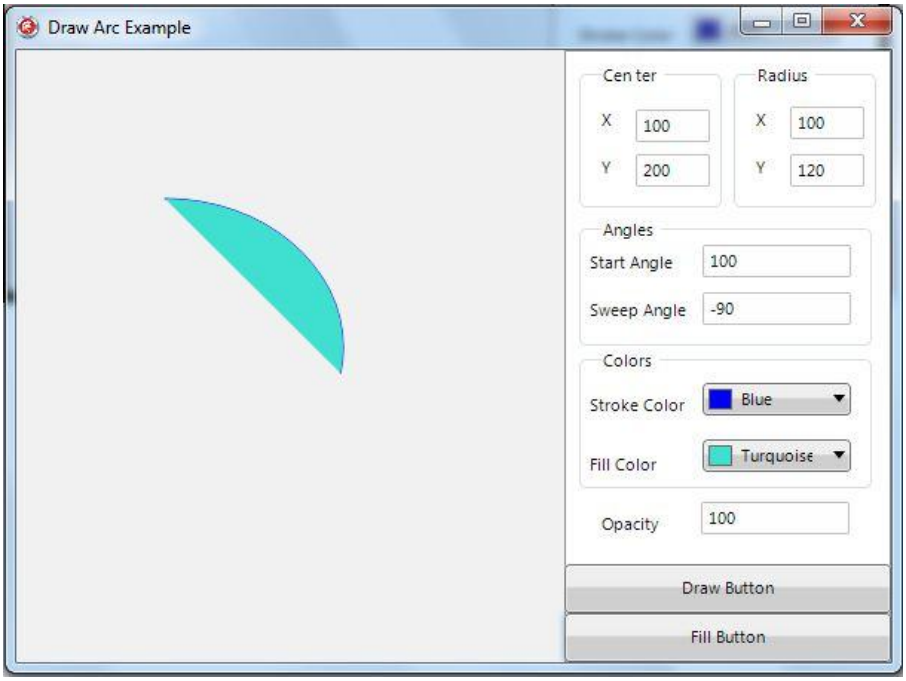
C++ Code

```
// Add the following code to the OnCreate event
// handler of the form.

void __fastcall TForm2::FormCreate(TObject *Sender) {
  // initializes the bitmap
  Image1->Bitmap = new TBitmap(int(Image1->Width),
    int(Image1->Height));
}

// Add the following code to the OnClick event
```

```
// handler of the DrawButton and FillButton:

void __fastcall TForm2::DrawButtonClick(TObject *Sender) {
  TPointF Center, Radius;
  TClipRects *AClipRect;
  Single Opacity, StartAngle, SweepAngle;
  // takes the information from the edits
  // checks whether all the values are valid
  if (TryStrToFloat(CenterXEdit->Text, Center.X)
      && TryStrToFloat(CenterYEdit->Text, Center.Y)
      && TryStrToFloat(RadiusXEdit->Text,Radius.X)
      && TryStrToFloat(RadiusYEdit->Text, Radius.Y)
      && TryStrToFloat(StartAngleEdit->Text, StartAngle)
      && TryStrToFloat(SweepAngleEdit->Text, SweepAngle)
      && TryStrToFloat(OpacityEdit->Text,Opacity)) {
    Image1->Bitmap->Canvas->BeginScene();
    Image1->Bitmap->Canvas->Stroke->Color =
      StrokeColorComboBox->Color;
    // draws the arc
    Image1->Bitmap->Canvas->DrawArc(Center, Radius,
      StartAngle, SweepAngle,Opacity);
    Image1->Bitmap->BitmapChanged();
    Image1->Bitmap->Canvas->EndScene();
  }
  else {
    // displays message if not all edits have numerical values
    ShowMessage("All Edits text should be numbers");
  }
}
// ---------------------------------------------------------------

void __fastcall TForm2::FillButtonClick(TObject *Sender) {
  TPointF Center, Radius;
  Single Opacity, StartAngle, SweepAngle;
  // takes the information from the edits
  // checks whether all the values are valid
  if (TryStrToFloat(CenterXEdit->Text, Center.x)
      && TryStrToFloat(CenterYEdit->Text, Center.y)
      && TryStrToFloat(RadiusXEdit->Text,Radius.x)
      && TryStrToFloat(RadiusYEdit->Text, Radius.y)
      && TryStrToFloat(StartAngleEdit->Text, StartAngle)
      && TryStrToFloat(SweepAngleEdit->Text, SweepAngle)
     && TryStrToFloat(OpacityEdit->Text,Opacity)) {
    Image1->Bitmap->Canvas->BeginScene();
    Image1->Bitmap->Canvas->Fill->Color =
      FillColorComboBox->Color;
    // fills the arc
    Image1->Bitmap->Canvas->FillArc(Center, Radius,
      StartAngle, SweepAngle,Opacity);
    Image1->Bitmap->BitmapChanged();
    Image1->Bitmap->Canvas->EndScene();
  }
  else {
    // displays a message if not all edits have numerical values
   ShowMessage("All Edits text should be numbers");
 }

}
```

```
// ---------------------------------------------------------------------
```

This is an example of how the example form should look after you compile and run it on Windows:

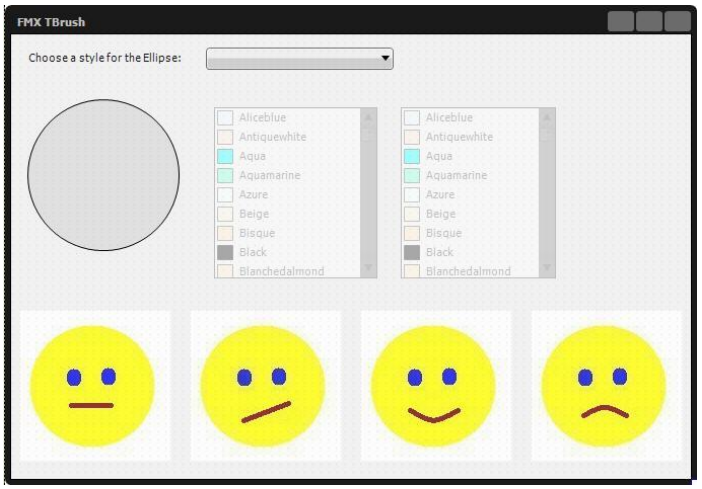


FMX TBrush Example (Delphi and C++)

This example is a FireMonkey HD Application that demonstrates how to use different properties of TBrush - http://docwiki.embarcadero.com/CodeSamples/en/FMXTBrush_(Delphi) and http://docwiki.embarcadero.com/CodeSamples/en/FMXTBrush_(C%2B%2B). This example requires the following components:

- A ComboBox object with three ListBoxItem objects.
- An Ellipse object.
- Two ColorListBox objects.
- Four Label objects.
- Four Image objects.

The form should look like in the following image.



Disable all the components, except the Ellipse, the ComboBox, and the Labels. Set the text to the labels as null (''), except for the one above the ComboBox. Set its text to 'Choose a style for the Ellipse:'. Load different Bitmap files to the Bitmap property of the Image objects.

```
//Delphi

procedure TForm1.ColorListBox1Change(Sender: TObject);
begin
  // Verify the Style of the TBrush and use the
  // selected color accordingly (as the color
  // of the brush or as the first gradient color)
  if (Ellipse1.Fill.Kind = TbrushKind.bkSolid) then
    Ellipse1.Fill.Color := ColorListBox1.Color
  else
    Ellipse1.Fill.Gradient.Color := ColorListBox1.Color;
end;

procedure TForm1.ColorListBox2Change(Sender: TObject);
begin
  // Use the selected color as the second
  // color of the gradient
  Ellipse1.Fill.Gradient.Color1 := ColorListBox2.Color;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  Ellipse1.Fill.Kind := TbrushKind.bkNone;
end;

procedure TForm1.Image1Click(Sender: TObject);
begin
  // Set the Brush's pattern to be the first image
  Ellipse1.Fill.Bitmap.Bitmap := Image1.Bitmap;
  Ellipse1.Repaint;
end;

procedure TForm1.Image2Click(Sender: TObject);
begin
  // Set the Brush's pattern to be the second image
  Ellipse1.Fill.Bitmap.Bitmap := Image2.Bitmap;
```

```
  Ellipse1.Repaint;
end;

procedure TForm1.Image3Click(Sender: TObject);
begin
  // Set the Brush's pattern to be the third image
  Ellipse1.Fill.Bitmap.Bitmap := Image3.Bitmap;
  Ellipse1.Repaint;
end;

procedure TForm1.Image4Click(Sender: TObject);
begin
  // Set the Brush's pattern to be the fourth image
  Ellipse1.Fill.Bitmap.Bitmap := Image4.Bitmap;
  Ellipse1.Repaint;
end;

// Set the style of the TBrush according to the
// selected option and
// enable the components needed to set the
// other TBrush properties

procedure TForm1.ListBoxItem1Click(Sender: TObject);
begin
  Label3.Text := 'Choose a color:';
  Label4.Text := '';
  Label2.Text := '';
  Ellipse1.Fill.Kind := TbrushKind.bkSolid;
  ColorListBox1.Enabled := True;
  ColorListBox2.Enabled := False;
  Image1.Enabled := False;
  Image2.Enabled := False;
  Image3.Enabled := False;
  Image4.Enabled := False;
end;

procedure TForm1.ListBoxItem2Click(Sender: TObject);
begin
  Label3.Text := 'Choose the top color:';
  Label4.Text := 'Choose the bottom color:';
  Label2.Text := '';
  Ellipse1.Fill.Kind := TbrushKind.bkGradient;
  ColorListBox1.Enabled := True;
  ColorListBox2.Enabled := True;
  Image1.Enabled := False;
  Image2.Enabled := False;
  Image3.Enabled := False;
  Image4.Enabled := False;
end;

procedure TForm1.ListBoxItem3Click(Sender: TObject);
begin
  Label2.Text := 'Choose an image:';
  Label3.Text := '';
  Label4.Text := '';
  Ellipse1.Fill.Kind := TbrushKind.bkBitmap;
  ColorListBox1.Enabled := False;
  ColorListBox2.Enabled := False;
```

```
  Image1.Enabled := True;
  Image2.Enabled := True;
  Image3.Enabled := True;
  Image4.Enabled := True;
  Ellipse1.Fill.Bitmap.WrapMode := TWrapMode.wmTileStretch;
end;


// C++
void __fastcall TForm1::ColorListBox1Change(
                          TObject *Sender) {
     // Verify the Style of the TBrush and use
     // the selected color accordingly (as the
     // color of the brush or as the first gradient color)
     if (Ellipse1->Fill->Kind == TBrushKind::bkSolid)
          Ellipse1->Fill->Color = ColorListBox1->Color;
     else
          Ellipse1->Fill->Gradient->Color =
                    ColorListBox1->Color;
}
// ---------------------------------------------------

void __fastcall TForm1::ColorListBox2Change(
                          TObject *Sender) {
     // Use the selected color as the second
     // color of the gradient
     Ellipse1->Fill->Gradient->Color1 =
                    ColorListBox2->Color;
}
// ---------------------------------------------------

void __fastcall TForm1::FormCreate(TObject *Sender) {
     Ellipse1->Fill->Kind = TBrushKind::bkNone;
}
// ---------------------------------------------------

void __fastcall TForm1::Image1Click(TObject *Sender) {
     // Set the Brush's pattern to be the first image
     Ellipse1->Fill->Bitmap->Bitmap = Image1->Bitmap;
     Ellipse1->Repaint();
}
// ---------------------------------------------------

void __fastcall TForm1::Image2Click(TObject *Sender) {
     // Set the Brush's pattern to be the second image
     Ellipse1->Fill->Bitmap->Bitmap = Image2->Bitmap;
     Ellipse1->Repaint();
}
// ---------------------------------------------------

void __fastcall TForm1::Image3Click(TObject *Sender) {
     // Set the Brush's pattern to be the third image
     Ellipse1->Fill->Bitmap->Bitmap = Image3->Bitmap;
     Ellipse1->Repaint();
}
// ---------------------------------------------------

void __fastcall TForm1::Image4Click(TObject *Sender) {
```

```
      // Set the Brush's pattern to be the fourth image
      Ellipse1->Fill->Bitmap->Bitmap = Image4->Bitmap;
      Ellipse1->Repaint();
}
// Set the style of the TBrush according
// to the selected option and
// enable the components needed to set
// the other TBrush properties
// ---------------------------------------------------------

void __fastcall TForm1::ListBoxItem1Click(TObject *Sender) {
      Label3->Text = "Choose a color:";
      Label4->Text = "";
      Label2->Text = "";
      Ellipse1->Fill->Kind = TBrushKind::bkSolid;
      ColorListBox1->Enabled = True;
      ColorListBox2->Enabled = False;
      Image1->Enabled = False;
      Image2->Enabled = False;
      Image3->Enabled = False;
      Image4->Enabled = False;
}
// ---------------------------------------------------------

void __fastcall TForm1::ListBoxItem2Click(TObject *Sender) {
      Label3->Text = "Choose the top color:";
      Label4->Text = "Choose the bottom color:";
      Label2->Text = "";
      Ellipse1->Fill->Kind = TBrushKind::bkGradient;
      ColorListBox1->Enabled = True;
      ColorListBox2->Enabled = True;
      Image1->Enabled = False;
      Image2->Enabled = False;
      Image3->Enabled = False;
      Image4->Enabled = False;

}
// ---------------------------------------------------------

void __fastcall TForm1::ListBoxItem3Click(TObject *Sender) {
      Label2->Text = "Choose an image:";
      Label3->Text = "";
      Label4->Text = "";
      Ellipse1->Fill->Kind = TBrushKind::bkBitmap;
      ColorListBox1->Enabled = False;
      ColorListBox2->Enabled = False;
      Image1->Enabled = True;
      Image2->Enabled = True;
      Image3->Enabled = True;
      Image4->Enabled = True;
      Ellipse1->Fill->Bitmap->WrapMode =
TWrapMode::wmTileStretch;
}
// ---------------------------------------------------------
```

Additional Canvas, Brush and Bitmap Examples

There are several additional FireMonkey Canvas, Brush and Bitmap example programs on the Embarcadero DocWiki.

Delphi FireMonkey Code Examples

- http://docwiki.embarcadero.com/CodeSamples/en/FMXTBitmapCanvas_(Delphi)
- http://docwiki.embarcadero.com/CodeSamples/en/FMXTBitmapClear_(Delphi)
- http://docwiki.embarcadero.com/CodeSamples/en/FMXTBitmapManipulationFunctions_(Delphi)
- http://docwiki.embarcadero.com/CodeSamples/en/FMXTBitmapPixels_(Delphi)
- http://docwiki.embarcadero.com/CodeSamples/en/FMXTBitmapScanLine_(Delphi)
- http://docwiki.embarcadero.com/CodeSamples/en/FMXTBitmapStartLine_(Delphi)
- http://docwiki.embarcadero.com/CodeSamples/en/FMXTBrush_(Delphi)
- http://docwiki.embarcadero.com/CodeSamples/en/FMXTCanvasSaveCanvas_(Delphi)
- http://docwiki.embarcadero.com/CodeExamples/en/FMXTCanvasDrawArc_(Delphi)
- http://docwiki.embarcadero.com/CodeExamples/en/FMXTCanvasClippingFunctions_(Delphi)
- http://docwiki.embarcadero.com/CodeExamples/en/FMXTCanvasDrawFunctions_(Delphi)
- http://docwiki.embarcadero.com/CodeExamples/en/FMXTCanvasFillFunctions_(Delphi)

C++ FireMonkey Code Examples

- http://docwiki.embarcadero.com/CodeSamples/en/FMXTBitmapCanvas_(C%2B%2B)
- http://docwiki.embarcadero.com/CodeSamples/en/FMXTBitmapClear_(C%2B%2B)
- http://docwiki.embarcadero.com/CodeSamples/en/FMXTBitmapManipulationFunctions_(C%2B%2B)
- http://docwiki.embarcadero.com/CodeSamples/en/FMXTBitmapPixels_(C%2B%2B)
- http://docwiki.embarcadero.com/CodeSamples/en/FMXTBitmapScanLine_(C%2B%2B)
- http://docwiki.embarcadero.com/CodeSamples/en/FMXTBitmapStartLine_(C%2B%2B)
- http://docwiki.embarcadero.com/CodeSamples/en/FMXTBrush_(C%2B%2B)
- http://docwiki.embarcadero.com/CodeSamples/en/FMXTCanvasSaveCanvas_(C%2B%2B)
- http://docwiki.embarcadero.com/CodeExamples/en/FMXTCanvasClippingFunctions_(C%2B%2B)
- http://docwiki.embarcadero.com/CodeExamples/en/FMXTCanvasDrawArc_(C%2B%2B)
- http://docwiki.embarcadero.com/CodeExamples/en/FMXTCanvasDrawFunctions_(C%2B%2B)
- http://docwiki.embarcadero.com/CodeExamples/en/FMXTCanvasFillFunctions_(C%2B%2B)

Using Menus in a FireMonkey Application

FireMonkey supports both lightweight styled menus displayed on the form, and native menus.

Drop-Down Menus

Traditional desktop-OS drop-down menus are hosted in a **TMenuBar**, a fully styled control. It acts as the root of a tree of TMenuItem objects. The first generation of children are visible in the menu bar. Second generation items are displayed under their parent when the parent item is clicked. Later generations are displayed as sub-menus to the right.

Menu items have label Text and an optional Bitmap image. Items can be checked through the IsChecked trigger property. Setting AutoCheck to True will automatically toggle IsChecked every time the item is clicked. A group of menu items can be designated so that only one item at most is checked, by setting RadioItem to True on each item, and GroupIndex to the same arbitrary value.

Implement menu actions in the OnClick event handler. Assign keycodes (virtual key constants added to TShortCut constants for modifier keys) to the ShortCut property for shortcut keys.

To create a menu separator item, set the Text to a single hyphen-minus character (Unicode code point U+002D, ASCII 45).

Native Menus

Setting the **TMenuBar.UseOSMenu** property to True causes FireMonkey to create the menu tree with OS calls, resulting in a native menu. On Windows, this menu is at the top of the parent form, and displayed using the current Appearance theme. On Mac OS X, the menu is displayed in the global menu bar on top of the main screen whenever the application has focus.

Popup Menus

Use TPopupMenu to describe a menu that appears only when the Popup procedure is called. TPopupMenu appears at the coordinates indicated by the parameters of the Popup procedure.

TPopupMenu is composed of TMenuItems. You can add menu items in several ways:

- To add a menu item at design time, do any of the following:
  - Right-click the component and select Add Item from the context menu.
  - Double-click the component and click the Add Item button on the Items Designer.
  - Right-click the component, select Items Editor from the context menu, and then click the Add Item button.
- To add a menu item at run time, use the AddObject procedure of TPopupMenu.

Transparent Forms

You can create FireMonkey applications with transparent forms. Create a new FireMonkey HD application. Place a button on the form. Double Click on the button to create an OnClick event handler. Add a line of code:

```
Application.Terminate();  // Delphi
Application->Terminate();  // C++
```

Using the Object Inspector, set the form's Transparent property to True.

[screen shot goes here]

Compile and Run the application. You should see the following display.

[screen shot goes here]

Click on the button to terminate the application.

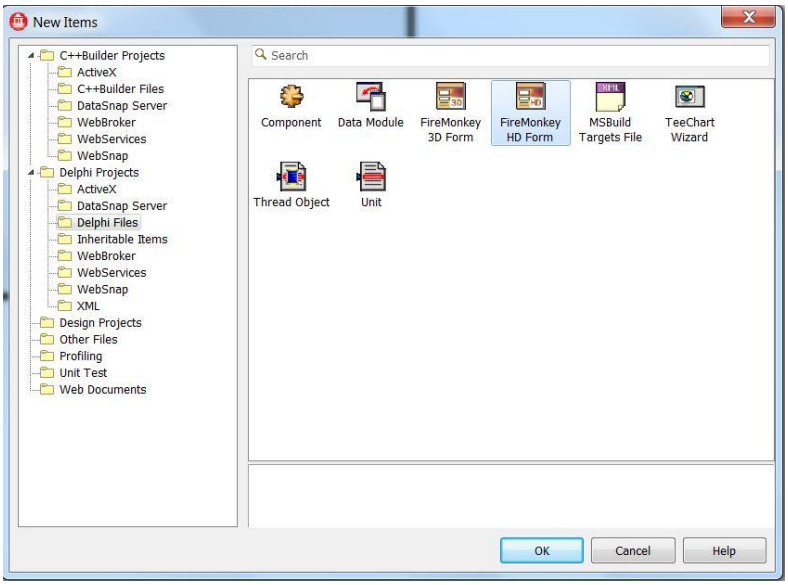
Embedding a Form inside another Form

This example shows how to embed one form inside another form by changing the Parent property of the components. http://docwiki.embarcadero.com/CodeExamples/en/FMXEmbeddedForm_(Delphi)

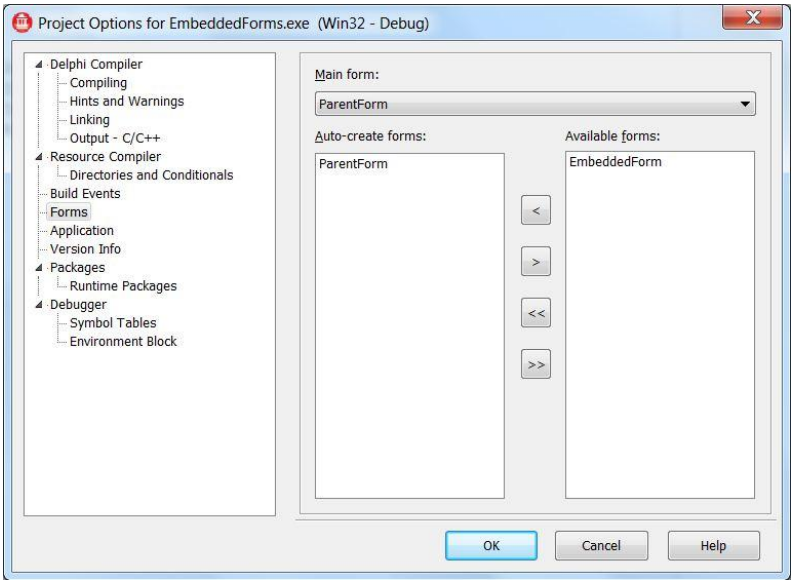
To build and test this example:

1. Create a FireMonkey HD Application, File > New > FireMonkey HD Application – Delphi or FireMonkey HD Application – C++Builder.

2. Add a second form to the project by choosing **File > New > Other.. > Delphi Projects > Delphi Files > FireMonkey HD Form** or **File > New > Other.. > C++Builder Projects > C++Builder Files > FireMonkey HD Form**



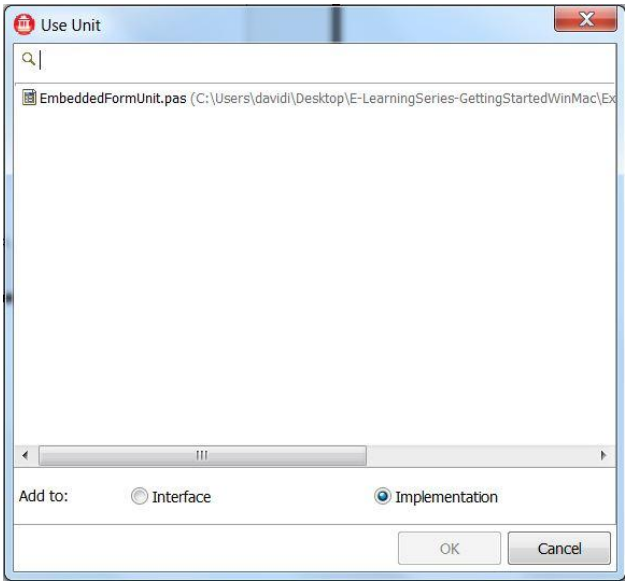
3. Use the Object Inspector to set the Name properties for the two forms to ParentForm and EmbeddedForm (also set their form Caption properties and name the unit filenames). Make sure that the ParentForm is the last form created by the application. Use the **Project > Options** menu item, click on the Forms category and move the EmbeddedForm from "Auto-create forms" to "Available forms".



4. Add a TPanel on ParentForm.

5. Add different controls to the EmbeddedForm (I added a Calendar and a Button).

6. You need to tell the ParentForm unit about the Embedded Form. You can use the **File > Use Unit** menu item to add the unit to the implementation section



or add the following code to implementation section in the source code for the ParentForm:

```
Uses EmbeddedFormUnit;

[todo C++ code]
```

7. Add the following method to the TParentForm class private section:

```
procedure EmbedForm(AParent:TControl;
                    AForm:TCustomForm);
```

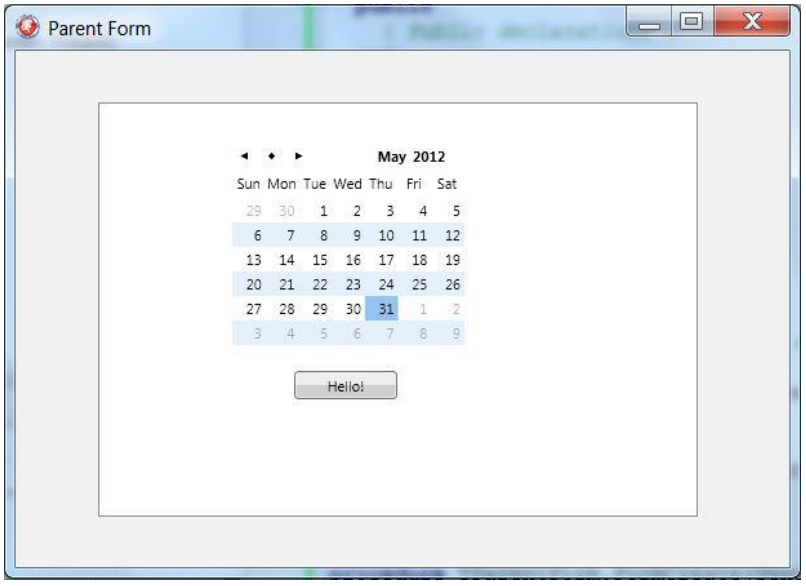
8. Add the following code to the ParentForm unit implementation section:

```
// AParent can be any control, such as a panel or a
// tabsheet item of a TabControl.
procedure TParentForm.EmbedForm(AParent:TControl;
                                AForm:TCustomForm);
begin
  while AForm.ChildrenCount>0 do
    AForm.Children[0].Parent:=AParent;
end;
```

9. Select the Parent Form and in the Object Inspector, switch to the Events tab and double click the OnCreate event to create the FormCreate event handler. Add the following code event handler:

```
EmbedForm(Panel1, TEmbeddedForm.Create(Self));
```

10. Run the application. Depending on what controls you put on your embedded form you should see your application look something like:



Customizing FireMonkey Applications with Styles

FireMonkey controls are arrangements of a tree of subcontrols and primitive shapes and brushes, decorated with effects. These compositions are defined as styles, stored in a style book. The individual elements of a style are internally called resources; because that term has several other meanings, the term style-resource is used for clarity. Styles provide a great deal of customization without subclassing. The FireMonkey styles that are provided with the product are located in C:\Program Files (x86)\Embarcadero\RAD Studio\9.0\Redist\styles\Fmx (or C:\Program Files\Embarcadero\RAD Studio\9.0\Redist\styles\Fmx for pre-Windows Vista/7 versions).

Default Styles

In FireMonkey, each control class has a default style, hard-coded per platform. To see the style definitions in the FireMonkey Style Designer:

- Drop a control on a form in the Form Designer.
- Right-click the control and choose Edit Default Style.

This creates a copy of the internal hard-coded style. A **TStyleBook** component is added to your form. For example, the default style of FMX.Controls.TPanel is defined simply as:

```
Panelstyle: TRectangle
```

The name of the style-resource that defines the style is "Panelstyle". It refers to a TRectangle. The appearance of this rectangle can be changed in the Style Designer, and then every TPanel on the form will have that appearance by default.

There is no rule that a TPanel must be represented by a TRectangle. A TRoundRect or TEllipse would work. (It makes no sense, but it could even be a TCalendar. The resulting panel would appear and superficially function as a calendar, but would not have access to all of TCalendar's properties and events.)

Even simple controls can be a complex composition. Consider FMX.Controls.TCheckBox, which looks something like:

- checkboxstyle: TLayout (The entire control)
- TLayout (The layout for the box.)
    - background: TRectangle (The box itself, which is actually a composition of:)
    - TGlowEffect (Glow when the control has focus.)
    - TRectangle (The outside rectangle that forms the box.)
    - TRectangle (The inside rectangle.)
    - TColorAnimation (Color animation when the mouse moves over.)
    - TColorAnimation (and back out.)
    - checkmark: TPath (The check inside the box, drawn as a path, which has:)
        - TColorAnimation (Its own color animation when the check is toggled on or off.)
- text: TText (And back under the top level, the text label.)

The style is named, so that it can be found and used. In addition, certain sub-elements are named, so that they can be referenced. When the IsChecked property is toggled, the "checkmark" has its visibility changed (by animating the opacity of its color from solid to transparent). Setting the Text of the TCheckBox sets the Text property of the underlying TText named "text".

Style Resource Naming and Referencing

Two properties with similar names form the links between a control, its style, and its subcomponents:

- The StyleName is the name by which a style or style subcomponent is known to others and can be found.
- A control's StyleLookup property is set to the desired style's name to adopt that style. When it is empty, the default style is used.
- Subcomponents can be found by calling FindStyleResource with the desired name.

A control has both properties because it can be styled, and it can be a style (or part of one). Simpler components like shapes cannot be styled, and can only be a style element.

Style Resource Storage

A collection of styles for a form is represented by a TStyleBook object. One is automatically created if necessary when the Style Designer is opened (by right-clicking a control and selecting either the Edit Default Style or Edit Custom Style context menu command). The newly created object is set as the form's StyleBook property, so that it takes effect for the form.

A form may have more than one TStyleBook object. The form's StyleBook property may reference any of them, one at a time; or it can be set to nil, which causes the form to revert to hard-coded default styles only.

The Style Designer edits the styles for a single TStyleBook at a time. Double-clicking a TStyleBook on a form will open the Style Designer with those styles. The Style Designer can save the TStyleBook in a text format to a .style file, and can load such a file. The entire set of hard-coded default styles can also be loaded into the Style Designer.

Custom Styles

New styles can be created by modifying default styles, or starting from scratch with a single component.

To modify a default style, right-click a control on the Form Designer and select Edit Custom Style. The generated style name is derived from the control's name, so you can save a step by choosing a good name for the control first. The generated name is assigned as the control's StyleLookup property, so that it takes effect for that control. The new style is a copy of the control's current style.

Completely new styles can be created by modifying a .style file and loading it, even using components that are not available in the Tool Palette. For example, after saving the current set of styles, edit the file to add before the final end:

```
object TBrushObject
  StyleName = 'somebrush'
end
```

Nested Styles

Styles may refer to other styled components. As always, styles are found by their top-level names in the TStyleBook. For example, to use the same gradient:

1. In the Style Designer, save the existing styles in a .style file.
2. Edit the file with a text editor to create a TBrushObject. Use an appropriate StyleName.
3. Load the .style file.
4. Select the newly defined style so that it appears in the Object Inspector.
5. Open the Brush property:
    a. Edit the Gradient property with the Brush Designer (choose Edit from the property value's drop-down menu)
    b. Set the Kind property to bkGradient
6. For each component using the gradient, for example, with a TRectangle's Fill property:
    a. Set the Kind property to bkResource
    b. Open the Resource property (a TBrushResource):
        i. Set the StyleLookup to the name of the gradient in Step 2.

Style-Resource Search Sequence

To find its style, a control goes through the following approximate sequence, stopping at the first match:

1. If the form's StyleBook property is set, that TStyleBook is searched using two names:
    a. The control's StyleLookup property, if set.
    b. A default name constructed from the control's class name:
        i. Drop the first letter (presumed to be the 'T' prefix of the standard class naming scheme).
        ii. Add 'style'.
2. The hard-coded default styles are searched using three names:
    a. The control's StyleLookup property, if set.
    b. The default name constructed from the control's class name.
    c. A default name constructed from the control's parent class name, using the same steps.

For example, the default names for TPanel are "Panelstyle" and "Controlstyle". For TCalloutPanel, the default names are "CalloutPanelstyle" and "Panelstyle".

Name matching is not case-sensitive. If no match is found, the control has no content and is effectively invisible. Code that depends on finding subcomponents will fail. (That should only happen for incomplete or improperly bundled custom controls, since all built-in controls have corresponding hard-coded styles. Direct descendants of built-in classes would have their base class content; second-generation descendants would be empty.)

Form Style

Although TForm is not a control or subclass of TStyledControl, it is also styled. Its StyleLookup property defaults to "backgroundstyle". The default style-resource with that StyleName is a grey TRectangle.

When loaded, the Align property of the resource is set to alContents to fill the form as the background. It is the first object painted, before the form's other children.

Customizing the Design of FireMonkey application

In the Visual Component Library (VCL), you can modify the color and other look and feel-related properties of each component from the Object Inspector. You do not see, however, such properties in FireMonkey. In FireMonkey, the look and feel of each component is defined as style, and you assign a style to a component. Because of the idea of style, you can now easily change the look and feel of the entire application by just applying different styles to the application. The following are predefined FireMonkey styles that you can easily use within your application (C:\Users\Public\Documents\RAD Studio\9.0\Styles):

- Air.Style
- Amakrits.Style
- AquaGraphite.Style
- Blend.Style
- Dark.style
- FMX.Platform.iOS.style
- FMX.Platform.Mac.style
- FMX.Platform.Win.style
- GoldenGraphite.Style
- iOS.Style
- iOSNative.Style
- Light.Style
- Mac.Style
- MacBlue.Style
- MacGraphite.Style
- RubyGraphite.style
- Windows7.Style

To select a style within your application, there are two typical ways to implement it. Step 1 explains how you can switch the style at run time using code. Step 2 explains how you can set the style while you design your application, and how you include the specified style in your application.

Step 1: Apply the existing style to your application at run time

Note: You can find a sample application at C:\Users\Public\Documents\RAD Studio\<9.0>\Samples\FireMonkey\ControlsDemo

The ControlsDemo sample application has many FireMonkey components already placed, and the implementation for switching the style at run time is already implemented.

To use this functionality in this demo application, click File > Load Style…, and select a style file.

Here is the code used to switch to a new style:

```
procedure TfrmCtrlsDemo.MenuItem7Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
  begin
    Application.StyleFileName := OpenDialog1.FileName;
  end;
end;
```

First, this demo code shows an 'Open Dialog' used to select a file. The name of the dialog is set to OpenDialog1 in the code. When the user selects a file, the Execute method returns True. This code checks whether the user selected a file first, and then assigns the file name of the style file from OpenDialog1.Filename (which holds the file name selected by the user) to the Application.StyleFileName property. That's it.

Step 2: Apply an existing style to your application at design time

You can also apply existing styles through the TStyleBook component at design time. To apply a style at design time:

1. Drop a TStyleBook component to your form. By default, the name of the new component is StyleBook1.
2. Select a FireMonkey form, and set the StyleBook property to StyleBook1.
3. Double-click the StyleBook1 component.
4. Click the Load… button and select the style. Styles are available at C:\Program Files (x86)\Embarcadero\RAD Studio\9.0\Redist\styles\Fmx.
5. Select Apply and Close. Now controls on the Form Designer are rendered using the specified style.

Step 3: Modify the style for a particular component

You can also customize the style for a specific component. To customize the style of a specific component:

1. Select a component on the Form Designer.
2. Right-click the component and select the Edit Custom Style… option.
3. Change the property of this particular style through the Object Inspector. You can change any property defined in the Object Inspector.
4. Select Apply and Close, and return to to Form Designer.
5. Once you select a component (whose style was just customized), you will find a new property defined (Panel1Style1).
6. Now you can apply the same style to different components. Select another component and then set the StyleLookup property to PanelStyle1.

FireMonkey Primitive Controls and Styled Controls

Primitive Controls

FireMonkey primitive controls are inherited from FMX.Objects.TShape, and therefore they know how to draw themselves. Drawing is done by the Paint method, which is introduced at FMX.Types.TControl and then overwritten by each primitive control.

The look and feel of a primitive control is defined by its own properties, such as Fill, Stroke, StrokeCap, StrokeDash, StrokeJoin, and StrokeThickness.

Here are the FireMonkey primitive controls:

- FMX.Objects.TLine
- FMX.Objects.TRectangle
- FMX.Objects.TRoundRect
- FMX.Objects.TEllipse
- FMX.Objects.TCircle
- FMX.Objects.TArc
- FMX.Objects.TCustomPath
- FMX.Objects.TText

Creating a FireMonkey Primitive Control presents step-by-step instructions for creating your own FireMonkey primitive control.
http://docwiki.embarcadero.com/RADStudio/en/Creating_a_FireMonkey_Primitive_Control

Styled Controls

FireMonkey styled controls do not define a look and feel. Instead, they read style files (or styles defined by a stylebook component) to dynamically select a list of primitive controls with their properties. Thus, styled controls can change look and feel when the application switches to a new style (load new style definitions).

A styled control can be built of any combination of primitive controls, styled controls, and effects. For example, FMX.Controls.TCheckBox is built of the following primitive controls and effects:

- TLayout (CheckBoxstyle)
- TLayout
- TRectangle (background)
- TGlowEffect
- TRectangle
- TRectangle
- TColorAnimation
- TColorAnimation
- TPath (checkmark)
- TColorAnimation
- TText (text)

When a styled control is loaded, an object is loaded with StyleName ('<classname>style). Then the control is constructed. For example:

```
objectTLayout
StyleName = 'checkboxstyle'
Position.Point = '(352,391)'
    Width = 121.000000000000000000
    Height = 21.000000000000000000
DesignVisible = False
objectTLayout
      Align = alLeft
      Width = 20.000000000000000000
      Height = 21.000000000000000000
objectTRectangle
StyleName = 'background'
        Align = alCenter
Position.Point = '(3,4)'
        Locked = True
        Width = 13.000000000000000000
        Height = 13.000000000000000000
HitTest = False
Fill.Color = xFFEFEFEF
Stroke.Kind = bkNone
objectTGlowEffect
          Trigger = 'IsFocused=true'
          Enabled = False
          Softness = 0.200000002980232200
GlowColor = x82005ACC
          Opacity = 0.899999976158142100
end
```

```
objectTRectangle
          Align = alContents
          Locked = True
          Width = 13.000000000000000000
          Height = 13.000000000000000000
HitTest = False
Fill.Kind = bkGradient
Fill.Gradient.Points = <
item
              Color = x5FA7A7A7
end
item
              Color = x5FFFFFFF
              Offset = 1.000000000000000000
end>
Stroke.Color = x90404040
end
objectTRectangle
          Align = alContents
Position.Point = '(2,2)'
          Locked = True
          Width = 9.000000000000000000
          Height = 9.000000000000000000
Padding.Rect = '(2,2,2,2)'
HitTest = False
Fill.Kind = bkNone
Stroke.Color = x30505050
end
objectTColorAnimation
          Duration = 0.100000001490116100
          Trigger = 'IsMouseOver=true'
StartValue = xFFEFEFEF
StopValue = xFFA5D9FF
PropertyName = 'Fill.Color'
end
objectTColorAnimation
          Duration = 0.100000001490116100
          Inverse = True
          Trigger = 'IsMouseOver=false'
StartValue = xFFEFEFEF
StopValue = xFFA5D9FF
PropertyName = 'Fill.Color'
end
objectTPath
StyleName = 'checkmark'
          Align = alCenter
Position.Point = '(2,2)'
          Locked = True
          Width = 9.000000000000000000
          Height = 9.000000000000000000
HitTest = False
Fill.Color = claNull
Stroke.Kind = bkNone
Data.Path = {
120000000000000000000000000000001000000CDCC4C3E0000000001000000
0000003F9A99993E01000000CDCC4C3F000000000100000000000803F00000000
010000000000803FCDCC4C3E010000003333333F0000003F010000000000803F
CDCC4C3F010000000000803F0000803F01000000CDCC4C3F0000803F01000000
```

```
0000003F3333333F01000000CDCC4C3E0000803F0100000000000000000000803F
0100000000000000CDCC4C3F010000009A99993E0000003F0100000000000000
CDCC4C3E0100000000000000000000000030000000000000000000000}
objectTColorAnimation
            Duration = 0.100000001490116100
            Trigger = 'IsChecked=true'
TriggerInverse = 'IsChecked=false'
StartValue = x00034E9E
StopValue = xFF034E9E
PropertyName = 'Fill.Color'
end
end
end
end
objectTText
StyleName = 'text'
      Align = alClient
Position.Point = '(21,2)'
      Locked = True
      Width = 99.000000000000000000
      Height = 19.000000000000000000
Padding.Rect = '(1,2,1,0)'
HitTest = False
      Text = 'CheckBox'
end
end
```

Grid and StringGrid

Grids are one of the most popular categories of third party components that developers love to use. FireMonkey also includes two grid components: TGrid and TStringGrid.

TGrid represents a grid control designed to simplify the handling of various objects in a tabular format. Add a TGrid object to a form to present various object data in a tabular format. A TGrid can have one or more of the following columns that can contain specific object data:

- TCheckColumn--holds cells with check-boxes.
- TImageColumn--holds cells with graphical images.
- TPopupColumn--holds cells with pop-up menus.
- TProgressColumn--holds cells with progress bar objects.
- TStringColumn--holds cells with strings.

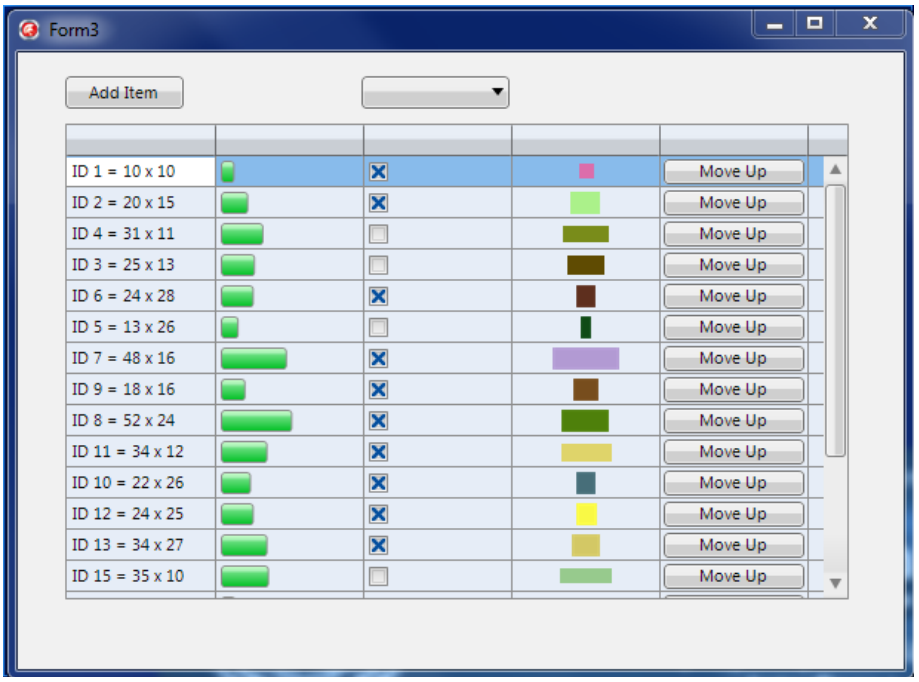
If you need a grid that can hold only strings, use a TStringGrid instead.

TStringGrid represents a grid control designed to simplify the handling of strings.

Add a TStringGrid object to a form to present textual data in a tabular format. TStringGrid provides many properties to control the appearance of the grid, as well as events and methods that take advantage of the tabular organization of the grid in responding to user actions.

TStringGrid introduces the ability to associate an object with each string in the grid. These objects can encapsulate any information or behavior represented by the strings that are presented to the

Stephen Ball, Embarcadero Software Consultant in the UK, recently wrote a superb blog post about FireMonkey Grids, "Getting to grips with using FireMonkey Grids", http://blogs.embarcadero.com/stephenball/2012/05/29/getting-to-grips-with-using-firemonkey-grids/



In his blog post, Stephen says: With the grids, there are two key events to code:

- OnGetValue is used to read in for a column and row the value
- OnSetValue is used for updating your object with the value from the grid.

The other useful call is that to TGrid.Realign - You can call this once you have managed the objects externally to ensure the grid redraws. This allows multiple changes to run before you update the grid to ensure that the process runs as quickly as possible without the grid constantly trying to update changes.

Stephen's demo source code for the Grid example is available using the following link from code central http://cc.embarcadero.com/item/28894.

Printing from a FireMonkey Application

You can print from a FireMonkey application on either Windows or Mac OS X. FMX.Printer.TPrinter encapsulates the Windows and Mac OS X standard printer interfaces.

Use TPrinter to manage any printing performed by an application. Obtain an instance of TPrinter by calling the global Printer function.

A print job is started by a call to BeginDoc. The application sends commands by rendering through a Text variable or the printer's canvas. You can move to a new page by calling the NewPage method. The job stays open until the application calls EndDoc. If a problem occurs and you need to terminate a print job that was not sent to the printer successfully, call the Abort method.

Use the TPrinter properties to configure the print job. For example, the title displayed in the Print Manager (and on network header pages) is determined by the Title property.Copies determines the number of copies to print, and Orientation lets you specify whether to print in portrait or landscape mode.

TPrinter includes several read-only properties that let you determine which page is currently being printed, the fonts available on the printer, the paper size, and so on.

When creating a TPrinter descendant, you must call the SetPrinter routine in order for the TPrinter descendant object to work correctly.

Enabling Printing in Your FireMonkey Application

The following steps show you how to print successfully from a FireMonkey application.
Add FMX.Printer to the uses clause (Delphi) or include FMX.Printer.hpp through an #include directive (C++Builder) of your form that issues the printing job.

```
uses FMX.Printer;
#include "FMX.Printer.hpp"
```

Always use the printer canvas or a graphical component's canvas (for instance, the canvas of an image) instead of the form's canvas. When not using the printer canvas, always start the printing operation with BeginScene and end the printing job with EndScene.

For full control over the printing quality, the Canvas resolution and portability, set ActiveDPIIndex, in the first place, either by assigning it a value or by calling the SelectDPImethod (the recommended way).

```
Printer.ActivePrinter.SelectDPI(1200, 1200); { one way }
Printer.ActivePrinter.ActiveDPIIndex := 1; { another way }

Printer->ActivePrinter->SelectDPI(1200, 1200); // one way
Printer->ActivePrinter->ActiveDPIIndex = 1; // another way
```

Start the actual printing job with BeginDoc.

```
Printer.BeginDoc;
Printer->BeginDoc();
```

End the actual printing job with EndDoc.

```
Printer.EndDoc;
Printer->EndDoc();
```

Never change ActiveDPIIndex during a printing job.

About DPI and Driver Support

The driver support for the same printer varies greatly on different platforms (Windows, Mac OS X). As a result, you should not depend on the fact that the number or order of supported resolutions in the DPI array is the same between Windows and Mac OS X. For instance, there are only 3 supported resolutions for the HP Laser Printer on Mac OS X, whereas there are 7 resolutions for Windows. If you have applications with printing functionality on both Windows and Mac OS X, you need to add additional code (mainly specifying the DPI) to ensure that the printing output is reasonably similar on both platforms.

The best practice is always to specify the DPI either by setting the ActiveDPIIndex property or calling the SelectDPI method, before printing, as mentioned in the steps above.

Note that ActiveDPIIndex is not set to the default DPI for a given printer. A TPrinterDevice object does not support a default DPI, especially because some Mac OS X printer drivers do not report the default DPI. Because of this, ActiveDPIIndex is set to -1 when the application starts, for all TPrinterDevice objects. The value -1 means that it will use either the last printing DPI or the default DPI. The default DPI is available only if after the application started, you did not change ActiveDPIIndex to some other index value and you did not call BeginDoc.

On the other hand, you can set ActiveDPIIndex to -1 and the application will use the last DPI of the printer that was previously set. So always remember to set a value toActiveDPIIndex before calling BeginDoc, because you want the same Canvas size on both Windows and Mac OS X.

The SelectDPI method is very easy to use to let a TPrinter object select the closest resolution to that of your printer, based on the parameters passed to SelectDPI. It is very easy to set, for instance, Printer.ActivePrinter.Select(600, 600); instead of directly setting the ActiveDPIIndex property. Given the fact that, on Windows, the number of printer resolutions reported by the driver differs from the one reported by the Mac OS X printer driver, the specified index might not be available.

In conclusion, using SelectDPI is a more convenient way of setting the printing DPI.

Printer Canvas

There are a few differences in the usage of a printer canvas compared to a screen or a window (form) canvas:

- When using the printer canvas, you must set the color and fill preferences using Canvas.Fill (`Canvas.Fill.Color := claXXX;` and `Canvas.Fill.Kind := TBrushKind.bkSolid;`).

- When using the screen or window canvas, you must set the color and fill preferences using Canvas.Stroke (`Canvas.Stroke.Color := claXXX;` and `Canvas.Stroke.Kind := TBrushKind.bkSolid;`).

Keep in mind that:

- When you are printing text and you want to change its color, use Canvas.Fill.
- When you are drawing anything else, excepting text, use Canvas.Stroke.

The `Opacity` parameter, for all the routines that support it (such as FMX.Types3D.TContext3D.FillRect), takes floating-point values in the range `0..1` (where 0 is transparent and 1 is opaque). Do not set values outside this range.

Example of Programmatic Printing

The following example uses an image and a button. Whenever you click the button, the image is printed to the printer.

```
// Delphi
procedure TMyForm.PrintButtonClick(Sender: TObject);
var
  SrcRect, DestRect: TRectF;

begin
  { Set the default DPI for the printer. The SelectDPI
  routine defaults to the closest available resolution
  as reported by the driver. }
  Printer.ActivePrinter.SelectDPI(1200, 1200);
  { Set canvas filling style. }
  Printer.Canvas.Fill.Color := claBlack;
  Printer.Canvas.Fill.Kind := TBrushKind.bkSolid;
  { Start printing. }
  Printer.BeginDoc;
  { Set the Source and Destination TRects. }
  SrcRect := Image1.LocalRect;
  DestRect := TRectF.Create(0, 0,
    Printer.PageWidth, Printer.PageHeight);
  { Print the picture on all the surface of the
    page and all opaque. }
  Printer.Canvas.DrawBitmap(Image1.Bitmap, SrcRect, DestRect, 1);
  { Finish printing job. }
  Printer.EndDoc;
end;

// C++
void __fastcall TMyForm::PrintButtonClick(TObject *Sender)
{
  TRectF SrcRect, DestRect;
  TPrinter *Printer = Printer;

  /* Set the default DPI for the printer. The SelectDPI
      routine defaults to the closest available resolution
```

```
      as reported by the driver. */
    Printer->ActivePrinter->SelectDPI(1200, 1200);
    /* Set canvas filling style. */
    Printer->Canvas->Fill->Color = claBlack;
    Printer->Canvas->Fill->Kind = TBrushKind(1);
    /* Start printing. */
    Printer->BeginDoc();
    /* Set the Source and Destination TRects. */
    SrcRect = Image1->LocalRect;
    DestRect = TRectF(0, 0,
      Printer->PageWidth, Printer->PageHeight);
    /* Print the picture on all the surface of the
       page and all opaque. */
    Printer->Canvas->DrawBitmap(Image1->Bitmap,
      SrcRect, DestRect, 1);
    /* Finish the printing job. */
    Printer->EndDoc();
  }
```

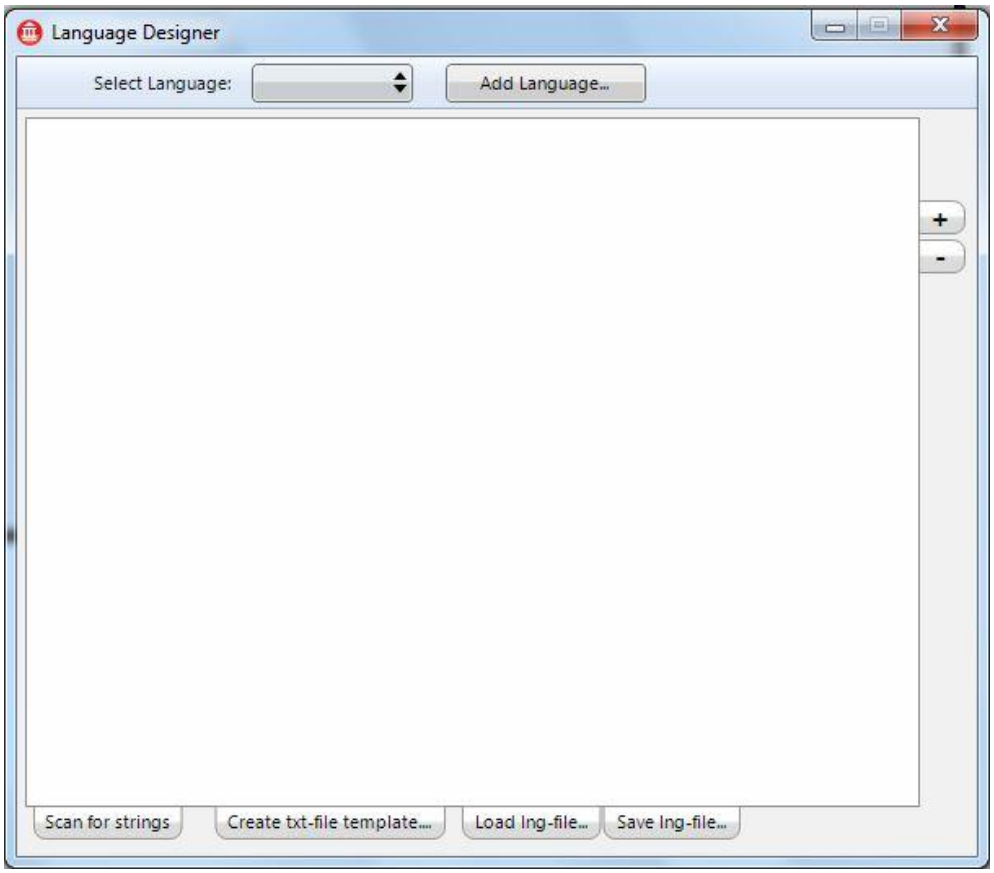
For more information regarding the printing API, please refer to the API documentation in the FMX.Printer unit.

The Embarcadero DocWiki has another Delphi printing example that prints flags at http://docwiki.embarcadero.com/CodeExamples/en/FMX.FlagsPrinting_Sample. The Delphi sample application is available as part of your install at C:\Users\Public\Documents\RAD Studio\9.0\Samples\FireMonkey\FlagsPrinting.

FireMonkey Multi-Language UI Support using TLang

You can localize the strings in your FireMonkey HD application by using the TLang component.

Use TLang for defining lists of strings that can be translated in order to localize an application. Add a TLang component from the Tool Palette on your form and double-click it to open the Language Designer.



The Language Designer finds all the strings in the application and allows you to add a list of languages in which to translate them. The original strings are placed in the first column and you can insert the translations in the second column.

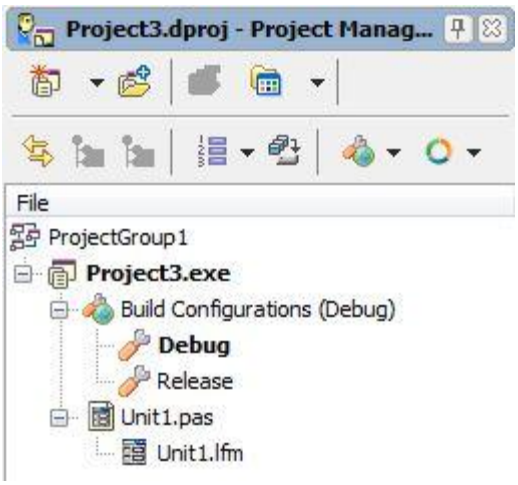
By clicking "Save lng-file..." in the Language Designer, you can save all the TLang strings in a file. Also, the designer allows you to use an existing language file (Lang file), by clicking "Load lng-file...."

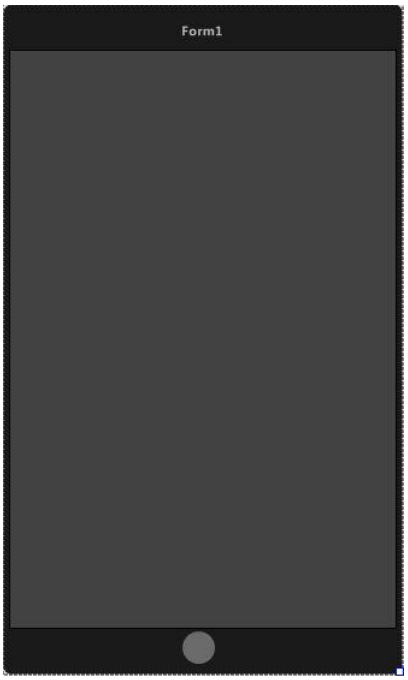
Take a look at the example program that uses TLang on the Embarcadero DocWiki at

- http://docwiki.embarcadero.com/CodeExamples/XE2/en/FMXTLang_(Delphi)
- http://docwiki.embarcadero.com/CodeExamples/XE2/en/FMXTLang_(C%2B%2B)

Creating a FireMonkey HD iOS App (Delphi)

Creating a FireMonkey HD iOS application (Delphi only) is basically the same as creating an HD application for Windows and Mac. You start with the **File>New>FireMonkey HD iOS Application – Delphi.** This project wizard will create a starting project and a form that looks like an iPhone device form.





Note: The FireMonkey iOS form filename extension is .lfm instead of .fmx. The form will have a caption in the form designer, but iOS applications do not have a caption area when they run on the device. The caption is there so you know which form you are designing.

From that starting point you can create your application for iPhone, iPad and iPod Touch. Try using one of the Delphi HD example applications listed above in an iOS application.

With XE2 you can use most of the components included in the Tool Pallet with the exception of the following categories of components (using any of these components will cause a "Cannot find unit …" error when you try to compile the application with Xcode to create your iOS application): DataAccess, dbExpress, Internet, DataSnap, InterBase, InterBaseAdmin, dbGo, Indy, WebSnap, WebServices, Cloud, IntraWeb, and Fast Reports.

Anders Ohlsson has a series of blog posts with information, examples and tips for building iOS applications at http://blogs.embarcadero.com/ao/category/ios.

The Embarcadero DocWiki has additional information about creating iOS applications at http://docwiki.embarcadero.com/RADStudio/XE2/en/Creating_a_FireMonkey_iOS_App

Summary, Looking Forward, To Do Items, Resources, Q&A and the Quiz

In Lesson 5 you learned how to create HD applications for Windows, Mac and iOS (Delphi only). You learned about layouts, using styles, and the various components to create great looking HD applications.

In Lesson 6, you'll learn how to connect the UI controls to databases.

In the meantime, here are some things to do, articles to read and videos to watch to enhance what you learned in Lesson 5 and to prepare you for lesson 6.

To Do Items

Explore the HD example applications that are included with RAD Studio. Look through the ControlsDemo example and see how all of the various UI components look and work.

Links to Additional Resources

- Getting Started Course landing page - http://www.embarcadero.com/firemonkey/firemonkey-e-learning-series
- FireMonkey Application Platform - http://docwiki.embarcadero.com/RADStudio/en/FireMonkey_Application_Platform
- FireMonkey Tutorial Series on YouTube - http://www.youtube.com/playlist?list=PL19268CFB728C1EFF
- SQLite iOS Application Development with Delphi and FireMonkey - http://www.youtube.com/watch?v=77HYR3m1cig

Delphi:

- RAD Studio Delphi sample programs on SourceForge - http://radstudiodemos.svn.sourceforge.net/viewvc/radstudiodemos/branches/RadStudio_XE2/FireMonkey/

C++:

- RAD Studio C++ sample programs on SourceForge - http://radstudiodemos.svn.sourceforge.net/viewvc/radstudiodemos/branches/RadStudio_XE2/CPP/FireMonkey/

Q&A:

Here are some of the answers for the questions I've received (so far) for this lesson. I will continue to update this Course Book during and after course.

Q:
A:

If you have any additional questions – send me an email - davidi@embarcadero.com

Self Check Quiz

1. Which of the following components is not an HD component?

a) TEdit
b) TPanel
c) TLabel
d) TMesh
e) TButton

2. Which HD component allows you to scale your layouts automatically?

a) TLayout
b) TPanel
c) TScaledLayout
d) TForm

3. What property lets you set the transparency of a component?

a) Transparent
b) Opaqueness
c) Clear
d) Opacity
e) Visible

4. In an HD application, the coordinate system includes an X,Y and Z position, true or false?

a) True
b) False

Answers to the Self Check Quiz:

1d, 2c, 3d, 4b