Delphi/400

Windowsタブレット向けカスタムグリッドの作成方法

株式会社ミガロ.
システム事業部 2課
前坂　誠二

略　歴

生年月日:1989年3月21日
最終学歴:2011年　関西大学　文学部卒業
入社年月:2011年04月　株式会社ミガロ. 入社
社内経歴:2011年04月　システム事業部配属

現在の仕事内容:

Delphi/400を利用したシステム開発や保守作業を担当。Delphi、Delphi/400の開発経験を積みながら、日々スキルを磨いている。

1.はじめに

一覧形式でデータを照会する場合、グリッド形式のレイアウトがよく使用される。Delphi/400のVCLアプリケーションでは、TDBGridやTStringGridのコンポーネントを使用して容易にグリッド形式の照会画面が作成可能である。Delphi/400で作成したVCLアプリケーションは、PCだけでなくWindowsタブレットでも実行可能である。PCとタブレットで同一のアプリケーションが利用可能であることはDelphi/400アプリケーションの魅力のひとつである。しかし利用端末が異なるということは、利用するシーンや目的が異なることが多い。本稿のテーマであるタブレットの場合は、屋外で使用することが多く、別作業をしながら片手で操作するなどの場面が想定される。特に一覧形式の画面では、文字が細かくなりやすいため、より視認性や使い勝手のよさについて考慮する必要がある。

そこで本稿では、一覧形式の画面に注目し、Windowsタブレットでの利用に特化したアプリケーションの作成方法について紹介する。

2.作成する照会画面イメージ

まずは、本稿で作成するサンプルについて解説する。本稿では、商品を点検して状態を保存するという場面をイメージし、**【図1】**のような一覧画面を利用していると想定する。一覧画面の明細には、チェックボックスやリストボックス、カメラ起動の為のボタンを配置している。**【図1】**に対して、本稿の内容を実装した完成イメージは**【図2】**とする。

尚、本稿で紹介する内容はDelphi/400 11Alexandriaを使用し、フレームワークはFireMonkeyで作成する。

図 1　点検商品照会画面（変更前）

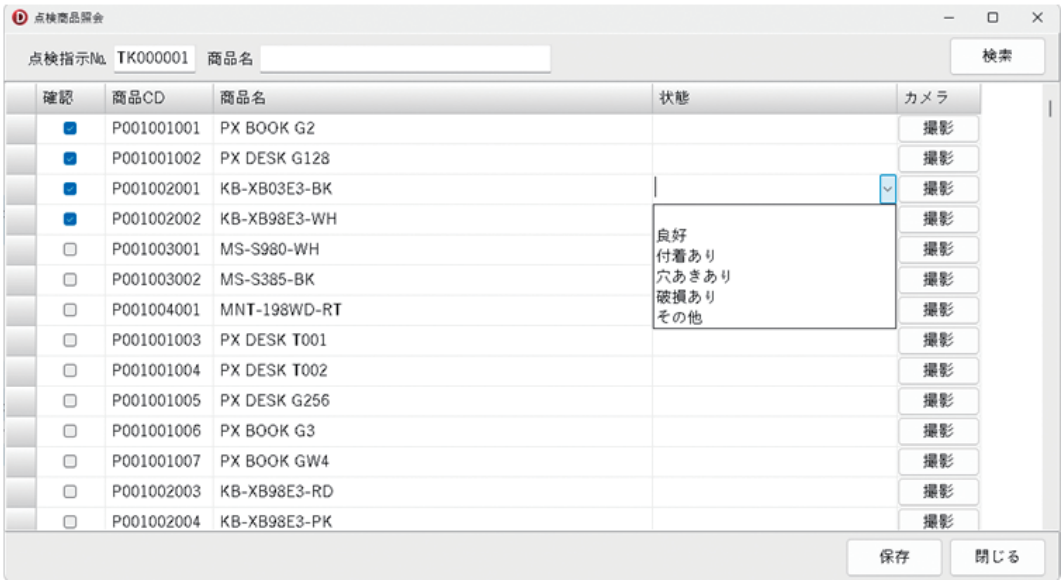


図 2　点検商品照会画面（変更後）

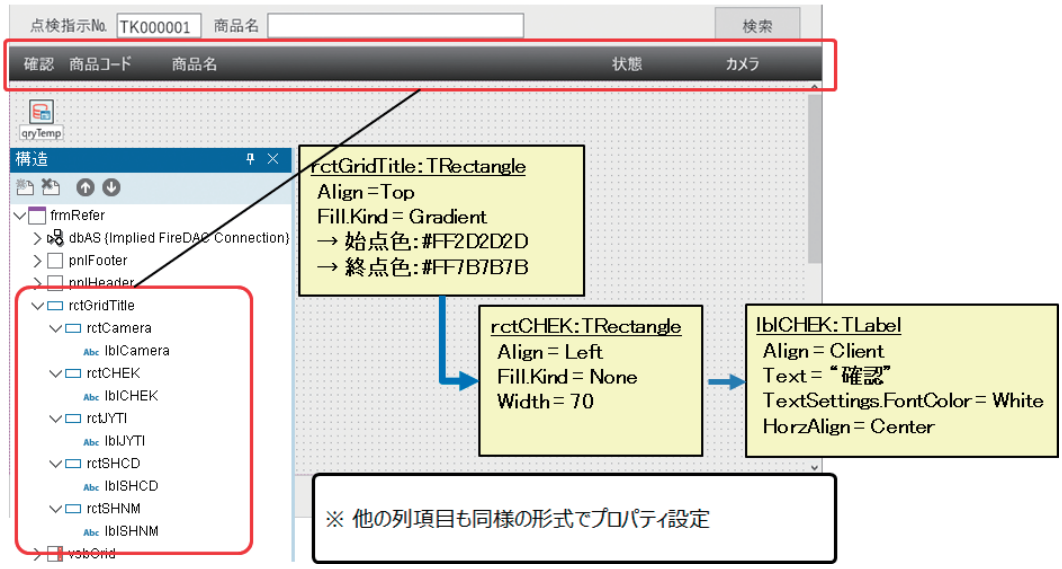
3.コンポーネントの配置

まずはコンポーネントの配置について説明する。本稿ではグリッドをメインテーマとして扱うためグリッド部分以外については説明を割愛する。

FireMonkeyは特徴として、コンポーネントに親子関係を持たせることができる。この特徴を活かし、コンポーネントの配置を行う。

明細タイトルは、列全体の色をコントロールするため、まずはTRectangleを配置し、子として各列用のTRectangle+TLabelを配置する【図3】。

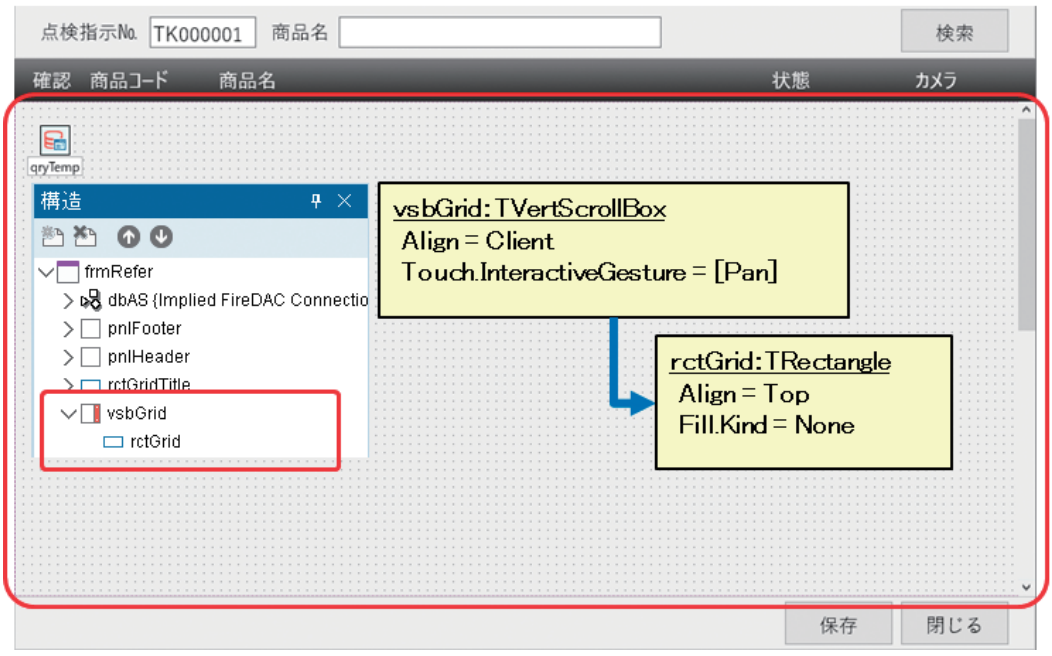
図 3　コンポーネント配置（明細タイトル）



データ内容を表示するためのコンポーネントは、TVertScrollBox→TRectangleコンポーネントを親子関係で配置する。TVertScrollBoxは、明細のスクロール可能にさせるために配置し（vsbGrid）、TRectangleは描画によるデータ表示を行うために配置する（rctGrid）。

この際、rctGridは［Fill］プロパティのKindをNoneとし、デフォルトの塗りつぶしを無しにする。［Align］はスクロール表示を行うため、Topの値に設定する。また、vsbGridのTouchプロパティにて［InteractiveGestures］の［Pan］をTrueにする。本プロパティをTrueにすることにより、タブレットで操作した際に指のスライドでのスクロールが可能となる**【図4】**。

図 4　コンポーネント配置（明細グリッド）



4.明細表示処理の実装

4-1.データ取得処理

まずは、データ取得処理の準備として今回実装に必要な変数・定数を定義する【ソース１】。
次に、btnSearchのOnClickイベントにて、TFDQueryでデータ取得→配列に保持する【ソース２】。その後、選択行を先頭にし、rctGridのHeightを行数×行の高さで設定している。
尚、本稿のサンプルでは【図５】のテーブルよりデータを取得する。

ソース１

変数定義

```
type
  // 明細データ
  TListData = record
    CHEK: Boolean;   // チェック
    SHCD: String;    // 商品CD
    SHNM: String;    // 商品名
    JYTI: String;    // 状態
  end;

  TfrmRefer = class(TForm)
    .
    .
    .

  private
    { private 宣言 }
    FListData: array of TListData; // 明細データの配列
    FSelectedRow: Integer; // 選択行保持
  public
    { public 宣言 }
  end;

    .
    .
    .
const
   cRowHeight = 55; // １明細の高さ
```

明細データ

内部保持用変数

定数

ソース２

データ取得処理

```
{*******************************************************************************
 目的: 検索ボタン押下時処理
 引数:
 戻値:
*******************************************************************************}
procedure TfrmRefer.btnSearchClick(Sender: TObject);
var
  iCnt: Integer;
begin
  // データ取得
  qryTemp.Close;
  qryTemp.SQL.Text :=
    ' SELECT * FROM TRTKSJ ';
  qryTemp.Open;
  try
```

```
    qryTemp.First;
    while not qryTemp.Eof do
    begin
      // 配列へセット
      SetLength(FListData, Length(FListData) + 1);
      iCnt := Length(FListData) - 1;

      FListData[iCnt].SHCD  := qryTemp.FieldByName('TSSHCD').AsString; // 商品コード
      FListData[iCnt].SHNM  := qryTemp.FieldByName('TSSHNM').AsString; // 商品名

      qryTemp.Next;
    end;
  finally
    qryTemp.Close;
  end;

  // 選択行をセット ※0始まり
  FSelectedRow := 0;

  // スクロールを先頭位置にする
  vsbGrid.ScrollBy(0, Length(FListData) * cRowHeight);

  // rctGridの大きさを設定
  rctGrid.Height := Length(FListData) * cRowHeight;

  // 再描画
  rctGrid.Repaint;
end;
```

図 5 データ取得用テーブル

```
A*********************************************************
A*       FILE-ID    :    TRTKSJ
A*       FUNCTION   :    点検指示ファイル
A*********************************************************
A                                    UNIQUE
A          R TRTKSR                  TEXT(' 点検指示ファイル ')
A            TSSJNO        8A        COLHDG(' 点検指示No. ')
A            TSSHCD       10A        COLHDG(' 商品コード ')
A            TSSHNM       520        COLHDG(' 商品名 ')
A            TSTKZM        1A        COLHDG(' 点検済フラグ ')
A            TSJTCD        4A        COLHDG(' 点検状態コード ')
A          K TSSJNO
A          K TSSHCD
```

4-2.描画処理

4-1で配列に保持しているデータをループ処理にて順次描画する。処理はrctGridのOnPainting処理にて記述する**【ソース3】**。

ソース 3

描画処理（行の枠、テキスト描画）

```
{*******************************************************************************
 目的: 描画処理
 引数:
 戻値:
*******************************************************************************}
procedure TfrmRefer.rctGridPainting(Sender: TObject; Canvas: TCanvas;
  const ARect: TRectF);
var
  brsTemp: TBrush;
  rRowArea, rText: TRectF;
  iData, iStartRow, iEndRow: Integer;
  bBtnColor: TBrush;
begin
  // 現在のスクロール位置より表示開始行を計算
  iStartRow := Trunc(vsbGrid.ViewportPosition.Y / cRowHeight);
  if iStartRow < 0 then
       iStartRow := 0;

  // 現在のスクロール位置より表示最終行を計算
  iEndRow := Trunc(vsbGrid.ViewportPosition.Y / cRowHeight)
           + Trunc(vsbGrid.Height / cRowHeight);

  for iData := iStartRow to iEndRow do
  begin
    if Length(FListData) - 1 < iData then
          Exit;

    // 行の枠を描画 Str ---------------------------------------------
    rRowArea        := ARect;
    rRowArea.Left   := 1;
    rRowArea.Right  := rctGrid.Width;
    rRowArea.Top    := iData * cRowHeight + 1;
    rRowArea.Bottom := (iData + 1) * cRowHeight;

    brsTemp := TBrush.Create(TBrushKind.Solid, TAlphaColorRec.White);

    if FSelectedRow = iData then
      brsTemp.Color := TAlphaColor($FFFFDC20) // 選択行
    else if ((iData mod 2) = 0) then
      brsTemp.Color := TAlphaColor($FFEAF2FC) // 偶数行
    else
      brsTemp.Color := TAlphaColorRec.White;  // 奇数行

    Canvas.FillRect(
      rRowArea,      // 描画対象範囲
      0, 0,          // XRadius、YRadius：角の曲がり具合  0…四角
      [],            // Radiusを適用する角
      1,             // Opacity：透明度
      brsTemp);      // 色
    // 行の枠を描画 End ---------------------------------------------

    // テキスト描画 Str ---------------------------------------------
    Canvas.Fill.Color := TAlphaColorRec.Black;
    Canvas.Font.Size := 19;

    // 商品コード
    rText := rRowArea;
    rText.Top := rText.Top + 3;
    rText.Left  := rctSHCD.Position.X;
    rText.Right := rctSHCD.Position.X + rctSHCD.Size.Width;
    Canvas.FillText(
      rText,                  // 描画対象範囲
      FListData[iData].SHCD,  // 値
      True,                   // WordWrap：True - 改行あり
      1,                      // Opacity：透明度
      [],                     // テキストを読む方向 ※ヘルプにて値なしを推奨
      TTextAlign.Leading,     // 水平方向の配置  Leading：左揃え
      TTextAlign.Center);     // 垂直方向の配置  Center ：中央
```

①

②

③

```
    商品名も商品コードと同手順で描画
    // テキスト描画 End ------------------------------------------
  end;
end;
```

処理の流れとしては以下の通りである。
①現在表示しているスクロール位置より
画面に表示する開始行と終了行を計算する
②各明細データ単位に行の区切り枠を描画する
③明細の値を描画する
本稿で実装している描画処理では、まずTRectF型の変数を定義し、Left、Right、Top、Bottomの値を指定して、描画の対象範囲を定める。その後、各々の描画処理を実装している。
行の区切り枠では、rRowArea変数に描画範囲をセットした後、選択行、偶数行、奇数行による色の指定を行い、Canvas.FillRect処理で描画を実施している。
明細の値では、rText変数に明細タイトルの位置と幅に合わせて描画範囲をセットした後、FillText処理で値を表示している。列の幅を大きくしたいときや位置を変更したい場合は、明細タイトルの幅や位置を変更するだけで、明細の描画も付随して変更されるという仕組みである。

4-3.明細タップ処理

本章のタップ処理では、タップ時に選択行の色の変更を行う。色の指定については4-2の処理で実装済みであるためrctGridのOnMouseDownイベントにてFSelectedRowの変数値を変更し、再描画を行うだけで実装可能である**【ソース4】**。

ソース 4

明細タップ処理（選択行の変更）

```
{*******************************************************************************
 目的: 明細タップ時処理
 引数:
 戻値:
*******************************************************************************}
procedure TfrmRefer.rctGridMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Single);
begin
  // タップ位置(Y座標)から選択行を計算
  FSelectedRow := Trunc(Y) div cRowHeight;

  // 再描画
  rctGrid.Repaint;
end;
```

4-4.スクロールバーの自動表示切替え

スクロールバーは明細をスライドしたタイミングだけ表示させるように実装する。ロジックとしては画面生成時処理に1行追加するだけで実装可能である【ソース5】。

ソース 5

スクロールの自動表示設定

```
{*******************************************************************************
 目的: 画面生成時処理
 引数:
 戻値:
*******************************************************************************}
procedure TfrmRefer.FormCreate(Sender: TObject);
begin
  // スクロールの自動表示
  vsbGrid.AniCalculations.AutoShowing := True;
end;
```

5.チェックボックスの実装

5-1.描画処理

本稿でのチェックボックスは、チェックONの場合は黒丸で塗りつぶし、チェックマークが表示される見た目で作成する。rctGridのOnPaintingに処理を追加する【ソース6】。

ソース 6

描画処理（チェックボックス）

```
{*******************************************************************************
 目的: 描画処理
 引数:
 戻値:
*******************************************************************************}
procedure TfrmRefer.rctGridPainting(Sender: TObject; Canvas: TCanvas;
  const ARect: TRectF);
var
  brsTemp: TBrush;
  rRowArea, rText: TRectF;
  rCheck: TRectF;
  iData, iStartRow, iEndRow: Integer;
  pDrawPoint1, pDrawPoint2: TPointF;
begin
  表示開始行、終了行計算【ソース3】

  for iData := iStartRow to iEndRow do
  begin
    if Length(FListData) - 1 < iData then
      Exit;
    行の枠、テキスト描画【ソース3】
```

```
    // チェックボックス描画 Str ------------------------------------
    // チェックの外側描画
    rCheck := rRowArea;
    rCheck.Left   := rctCHEK.Position.X + rctCHEK.Width / 4; // Left   ：開始位置
    rCheck.Right  := rCheck.Left + 32;                       // Right  ：幅
    rCheck.Top    := rRowArea.Top + 11;                      // Top    ：開始位置
    rCheck.Bottom := rRowArea.Bottom - 11;                   // Bottom：高さ

    // 描画する線の設定値
    Canvas.Stroke.Kind  := TBrushKind.Solid;
    Canvas.Stroke.Color := TAlphaColorRec.Black;
    Canvas.Stroke.Thickness  := 0.5;
    Canvas.DrawRect(
      rCheck,               // 描画対象範囲
      16, 16,               // Radius：横半径、縦半径
      AllCorners,           // Radiusを適用する角
      1,                    // Opacity：透明度
      TCornerType.Round);   // 角の種類

    // チェックONの場合
    if FListData[iData].CHEK then
    begin
      // チェックの中身描画
      Canvas.Fill.Color := TAlphaColorRec.Black;
      Canvas.FillArc(
        PointF(
          rCheck.Left + 16,                        // 円の中心位置：X
          Trunc(cRowHeight / 2) + rRowArea.Top),   // 円の中心位置：Y
        PointF(16, 16),                            // Radius：横半径、縦半径
        0, 360,                                    // 塗りつぶし角度
        1);                                        // Opacity：透明度

      // チェック線左
      pDrawPoint1.X := rCheck.Left + 6;            // 始点のX
      pDrawPoint1.Y := rCheck.Top  + 9;            // 始点のY
      pDrawPoint2.X := pDrawPoint1.X + 7;          // 終点のX
      pDrawPoint2.Y := pDrawPoint1.Y + 16;         // 終点のY
      Canvas.Stroke.Thickness  := 3.5;             // 線の太さ
      Canvas.Stroke.Color := TAlphaColorRec.White; // 線の色
      Canvas.DrawLine(pDrawPoint1, pDrawPoint2, 1);

      // チェック線右
      pDrawPoint1.X := pDrawPoint2.X;              // 始点のX
      pDrawPoint1.Y := pDrawPoint2.Y;              // 始点のY
      pDrawPoint2.X := rCheck.Left + 32;           // 終点のX
      pDrawPoint2.Y := rCheck.Top + 2;             // 終点のY
      Canvas.Stroke.Thickness  := 3.5;             // 線の太さ
      Canvas.Stroke.Color := TAlphaColorRec.White; // 線の色
      Canvas.DrawLine(pDrawPoint1, pDrawPoint2, 1);

      // 値を元に戻す
      Canvas.Stroke.Color := TAlphaColorRec.Black;
      Canvas.Stroke.Thickness  := 1;
    end;
    // チェックボックス描画 End --------------------------------------
  end;
end;
```

①

②

③

処理の流れとしては以下の通りである。
①外側の円を描画する
②チェックONの場合に、黒丸で塗りつぶしを行う
③チェックONの場合に、チェックマークを描画する
①の処理では、前章での描画処理と同様に、まずは描画対象範囲を指定する。その後DrawRect処理により描画処理を実施する。円の描画は、XRadius、YRadiusの引数値でどれくらいの大きさの円を描くかを決定する。XRadius、YRadiusの値が円の縦と横の半径にあたるため、設定した値に対して、Rectの値についても調整する必要がある。例えば、本サンプルの場合は各Radiusを16と設定しているため、RectのLeft-Right間、Top-Bottom間は32とならなければ、いびつな円として描画される。

②、③の処理はチェックONとした場合に実施する。②の黒丸での塗りつぶしはFillArc処理によって行う。こちらはDrawRect処理とは異なり、Rectの指定は不要であるが描画したい円の中心を起点に、塗りつぶし範囲を指定する必要がある。本サンプルでは、外側の円と同一の大きさで塗りつぶしを行っている。③のチェックマークの描画は、2本の線を組み合わせて実装している。それぞれ線の始点と終点、線の太さと色を設定した後、DrawLine処理により、線を描画している。

5-2.明細タップ処理

明細タップ時に、チェックボックスのONとOFFを切り替える処理を実装する。
タップしたポイントのX座標が”確認”列の範囲内であれば、配列の値を切り替える処理を追加するだけで実装完了である【ソース7】。

ソース 7

明細タップ処理（チェックボックス処理）

```
{*******************************************************************************
 目的: 明細タップ時処理
 引数:
 戻値:
*******************************************************************************}
procedure TfrmRefer.rctGridMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Single);
begin
  // タップ位置(Y座標)から選択行を計算
  FSelectedRow := Trunc(Y) div cRowHeight;

  // チェックボックスタップ処理 Str ---------------------------------------
  if (Trunc(X) >= rctCHEK.Position.X)                    and
     (Trunc(X) <= (rctCHEK.Position.X + rctCHEK.Width)) then
  begin
    FListData[FSelectedRow].CHEK := not (FListData[FSelectedRow].CHEK);
  end;
  // チェックボックスタップ処理 End ---------------------------------------

  // 再描画
  rctGrid.Repaint;
end;
```

処理追加

6.ボタンの実装

6-1.描画処理

“カメラ”列のボタンの描画を行う。ボタン内のテキストには“撮影”の文言を設定する。rctGridのOnPaintingに処理を追加する**【ソース8】**。

ソース 8

描画処理(ボタン)

```
{*******************************************************************************
 目的: 描画処理
 引数:
 戻値:
*******************************************************************************}
procedure TfrmRefer.rctGridPainting(Sender: TObject; Canvas: TCanvas;
  const ARect: TRectF);
var
  brsTemp: TBrush;
  rRowArea, rText: TRectF;
  rCheck: TRectF;
  rBtn: TRectF;
  iData, iStartRow, iEndRow: Integer;
  pDrawPoint1, pDrawPoint2: TPointF;
  bBtnColor: TBrush;
begin
  表示開始行、終了行計算【ソース3】
  for iData := iStartRow to iEndRow do
  begin
    if Length(FListData) - 1 < iData then
      Exit;
    行の枠、テキスト描画【ソース3】
    チェックボックス描画【ソース6】
    // ボタン描画 Str ----------------------------------------------
    rBtn := rRowArea;
    rBtn.Left   := rctCamera.Position.X;
    rBtn.Right  := rctCamera.Position.X + rctCamera.Size.Width;
    rBtn.Top    := rBtn.Top + 6;
    rBtn.Bottom := rBtn.Bottom - 6;

    // ボタン色
    bBtnColor := TBrush.Create(TBrushKind.Solid, TAlphaColor($FFE0E0E0));

    // ボタン描画
    Canvas.FillRect(
      rBtn,                 // 描画対象範囲
      5, 5,                 // XRadius、YRadius：角の曲がり具合  0…四角
      AllCorners,           // Radiusを適用する角
      1,                    // Opacity：透明度
      bBtnColor);           // 色

    // ボタン枠描画
    Canvas.Stroke.Kind  := TBrushKind.Solid;
    Canvas.Stroke.Color := TAlphaColorRec.Gray;
    Canvas.DrawRect(
      rBtn,                 // 描画対象範囲
      5, 5,                 // XRadius、YRadius：角の曲がり具合  0…四角
      AllCorners,           // Radiusを適用する角
      1,                    // Opacity：透明度
      TCornerType.Round); // 種類
```

①

②

```
    // 文字描画
    Canvas.Fill.Color := TAlphaColorRec.Black;
    Canvas.FillText(rBtn, '撮 影', True, 1, [], TTextAlign.Center, TTextAlign.Center);
    // ボタン描画 End ------------------------------------------------
  end;
end;
```

③

処理の流れとしては以下の通りである。
①指定範囲でボタンの形で塗りつぶしを行う
②ボタンの枠を描画する
③ボタンに表示する文言を描画する
①の処理では、前章での描画処理と同様に、まずは描画対象範囲を指定する。その後、FillRect処理にてボタンの形に描画する。本サンプルでは、少し丸みを帯びた形で描画するため、XRadiusとYRadiusの値を5で指定している。例えば、各Radiusの値を0で指定した場合は四角の形のボタンとなる。
②の処理では、FillRect処理で①で作成したボタンの枠を描画している。
③の処理では、FillText処理で描画したボタン上に文字をセットしている。文字の描画については、第４章と同様の手順である。

6-2.明細タップ処理

明細タップ時に、”撮影”ボタン押下時の処理を実装する。タップしたポイントのX座標が”カメラ”列の範囲内であれば、カメラ起動を行う【ソース９】。本サンプルでは、カメラ起動の処理を実装しているが、TButtonのOnClickイベントをイメージいただき、各々の処理を実装すればよい。

ソース９

明細タップ処理(ボタン処理)

```
{*******************************************************************************
 目的: 明細タップ時処理
 引数:
 戻値:
*******************************************************************************}
procedure TfrmRefer.rctGridMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Single);
begin
  // タップ位置(Y座標)から選択行を計算
  FSelectedRow := Trunc(Y) div cRowHeight;

  チェックボックスタップ処理【ソース7】

  // カメラボタンタップ処理 Str ------------------------------------------
  if (Trunc(X) >= rctCamera.Position.X)                    and
     (Trunc(X) <= (rctCamera.Position.X + rctCamera.Width)) then
  begin
    // *** カメラ起動 *** //
    ShellExecute(0, 'OPEN', PChar('microsoft.windows.camera:'), nil, nil, SW_SHOWMAXIMIZED);
  end;
  // カメラボタンタップ処理 End ------------------------------------------

  // 再描画
  rctGrid.Repaint;
end;
```

7.リストボックスの実装

7-1.描画処理

“状態”列のリストボックスの描画を行う。リストボックスはボタンの描画実施後に下向きの三角マークを描画して表示している【ソース１０】。

ソース 10

描画処理(リストボックス)

```
{*******************************************************************************
 目的: 描画処理
 引数:
 戻値:
*******************************************************************************}
procedure TfrmRefer.rctGridPainting(Sender: TObject; Canvas: TCanvas;
  const ARect: TRectF);
var
  brsTemp: TBrush;
  rRowArea, rText: TRectF;
  rCheck: TRectF;
  rBtn: TRectF;
  iData, iStartRow, iEndRow: Integer;
  pDrawPoint1, pDrawPoint2: TPointF;
  bBtnColor: TBrush;
  TempPoints: TPolygon;
begin
  表示開始行、終了行計算【ソース3】
  for iData := iStartRow to iEndRow do
  begin
    if Length(FListData) - 1 < iData then
      Exit;
    行の枠、テキスト描画【ソース3】
    チェックボックス描画【ソース6】
    ボタン描画【ソース8】
    // リストボックス描画 Str ---------------------------------------
    rBtn := rRowArea;
    rBtn.Left   := rctJYTI.Position.X;
    rBtn.Right  := rctJYTI.Position.X + rctJYTI.Size.Width;
    rBtn.Top    := rBtn.Top + 6;
    rBtn.Bottom := rBtn.Bottom - 6;

    // ボタン色
    bBtnColor := TBrush.Create(TBrushKind.Solid, TAlphaColor($FFE0E0E0));

    // ボタン描画
    Canvas.FillRect(
      rBtn,                 // 描画対象範囲
      5, 5,                 // XRadius、YRadius：角の曲がり具合　0…四角
      AllCorners,           // Radiusを適用する角
      1,                    // Opacity：透明度
      bBtnColor);           // 色

    // ボタン枠描画
    Canvas.Stroke.Kind  := TBrushKind.Solid;
    Canvas.Stroke.Color := TAlphaColorRec.Gray;
    Canvas.DrawRect(
      rBtn,                 // 描画対象範囲
      5, 5,                 // XRadius、YRadius：角の曲がり具合　0…四角
      AllCorners,           // Radiusを適用する角
      1,                    // Opacity：透明度
      TCornerType.Round); // 種類
```

①

```
    // 下三角配置設定
    SetLength(TempPoints, 4);
    TempPoints[0] := PointF(rctJYTI.Position.X + rctJYTI.Size.Width - 22, rBtn.Top    + 14); // 左支点
    TempPoints[1] := PointF(rctJYTI.Position.X + rctJYTI.Size.Width -  4, rBtn.Top    + 14); // 右支点
    TempPoints[2] := PointF(rctJYTI.Position.X + rctJYTI.Size.Width - 13, rBtn.Bottom - 12); // 下支点
    TempPoints[3] := PointF(rctJYTI.Position.X + rctJYTI.Size.Width - 22, rBtn.Top    + 14); // 左支点

    // 下三角描画
    Canvas.Fill.Color := TAlphaColorRec.Black;
    Canvas.FillPolygon(TempPoints, 1); // [0]→[1]→[2]→[3]で支点を結んで▼描画

    // 文字描画
    Canvas.Fill.Color := TAlphaColorRec.Black;
    Canvas.FillText(rBtn, FListData[iData].JYTI, True, 1, [], TTextAlign.Leading, TTextAlign.Center);
    // リストボックス描画 End ------------------------------------
  end;
end;
```

②

③

処理の流れとしては以下の通りである。

①第６章と同様の手順でボタンを描画する

②下三角(▼)を描画する

③リストボックスに表示する文言を描画する

①と③の処理については、第６章と同様のイメージでボタンと文言の描画処理を行う。

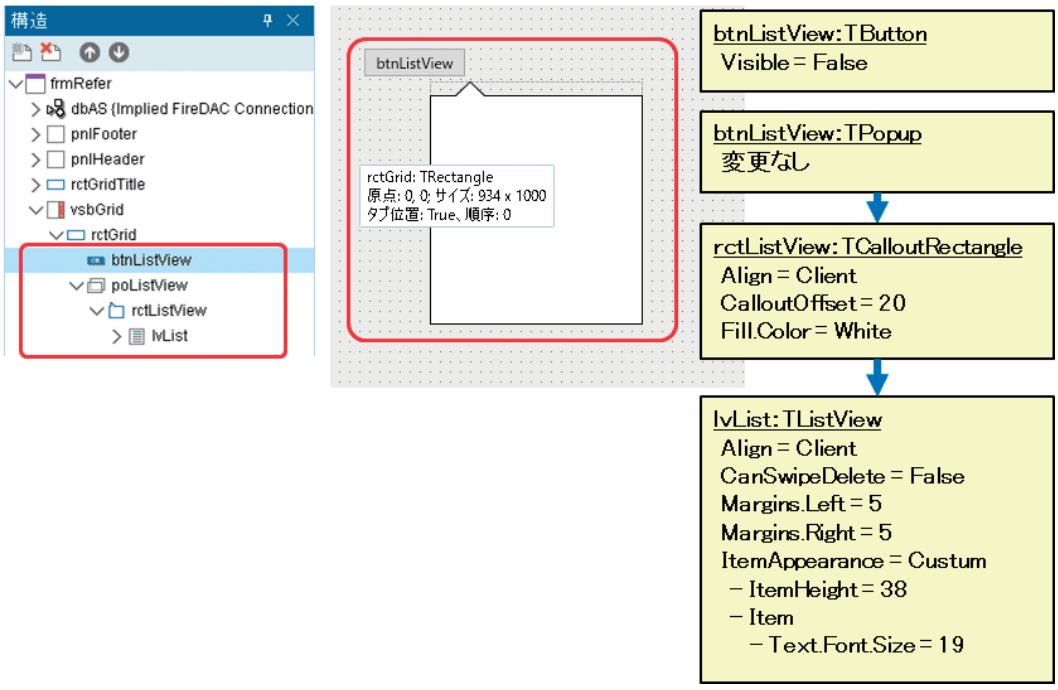
②の処理では、下三角を描画するためにFillPolygon処理を実施する。TempPoints変数では三角形を形成するために各支点を設定している。

下三角は記号文字でも表現可能であるが、実行端末の使用フォントに依存する可能性がある。しかし、②の処理のように図形で描画処理を行うと実行端末による影響は受けない。

7-2.コンポーネントの配置

本サンプルでは、明細タップ時に選択リストを表示する。リストの内容表示は別途コンポーネントを配置して実装する【図６】

図 6　コンポーネント配置(リスト)



7-3.リスト内容の取得

画面生成時に【図7】のテーブルよりデータを取得する。取得した内容をlvListのItemsに取得内容を追加する【ソース11】。

図 7　データ取得用テーブル(リスト取得)

```
A*****************************************************************
A*       FILE-ID    :    MSSHTJ
A*       FUNCTION   :    商品点検状態マスタ
A*****************************************************************
A                                      UNIQUE
A         R MSSHTR                     TEXT(' 商品点検状態マスタ ')
A           STJTCD         4A          COLHDG(' 状態コード ')
A           STJTNM        320          COLHDG(' 状態名 ')
A         K STJTCD
```

ソース 11

リストデータ取得処理

```
{*******************************************************************************
 目的: 画面生成時処理
 引数:
 戻値:
*******************************************************************************}
procedure TfrmRefer.FormCreate(Sender: TObject);
begin
  // スクロールの自動表示
  vsbGrid.AniCalculations.AutoShowing := True;

  // リスト内容取得
  lvList.Items.Clear;
  lvList.Items.Add.Text := '';
  qryTemp.Close;
  qryTemp.SQL.Text :=
    ' SELECT * FROM MSSHTJ ';
  qryTemp.Open;
  try
    while not qryTemp.Eof do
    begin
      lvList.Items.Add.Text :=
        qryTemp.FieldByName('STJTNM').AsString;
      qryTemp.Next;
    end;
  finally
    qryTemp.Close;
  end;

  // ポップアップ非表示
  poListView.Visible := False;
end;
```

処理追加

7-4.リスト表示処理

btnListViewのOnClickイベントにてリスト内容の表示処理を記述する【ソース１２】。poListViewの表示場所のターゲットをbtnListViewとし、ポップアップ表示させる。また、表示させる向きをbtnListViewの表示位置で決定する処理を記述している。

ソース 12

リスト内容表示処理

```
{*******************************************************************************
 目的: リスト表示処理
 引数:
 戻値:
*******************************************************************************}
procedure TfrmRefer.btnListViewClick(Sender: TObject);
var
  sY: Single;
begin
  poListView.PlacementTarget := btnListView;
  poListView.Popup;

  // ボタンのY座標を保持
  sY := btnListView.Position.Y;

  // CalloutPositionの設定
  if TCustomPopupForm(TMyPopupForm(poListView).PopupForm).Top < sY then
  begin
    // ポップアップを上向きに表示
    rctListView.CalloutPosition := TCalloutPosition.Bottom;
    lvList.Margins.Top    := 5;
    lvList.Margins.Bottom := 15;
  end
  else
  begin
    // ポップアップを下向きに表示
    rctListView.CalloutPosition := TCalloutPosition.Top;
    lvList.Margins.Top    := 15;
    lvList.Margins.Bottom := 5;
  end;
end;
```

7-5.明細タップ処理

明細タップ時に、“状態”列のリストボックス押下時の処理を実装する。タップしたポイントのX座標が“状態”列の範囲内であれば、リスト表示を行う【ソース１３】。
7-4でリスト表示処理を実装したbtnListViewをタップしたポイントへ移動させ、btnListViewClickイベントを動作させている。しかしbtnListViewボタンはVisible:=Falseで設定しており、画面上は表示されない。そのため、ポップアップのみがリストボックス上で表示されているように見える仕組みである。

ソース 13

明細タップ処理(リストボックス)

```
{*******************************************************************************
 目的: 明細タップ時処理
 引数:
 戻値:
*******************************************************************************}
procedure TfrmRefer.rctGridMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Single);
begin
  // タップ位置(Y座標)から選択行を計算
  FSelectedRow := Trunc(Y) div cRowHeight;
    チェックボックスタップ処理【ソース7】
    ボタンタップ処理【ソース9】
  // 状態リストボックスタップ処理 Str ------------------------------------
  if (Trunc(X) >= rctJYTI.Position.X)                  and
     (Trunc(X) <= (rctJYTI.Position.X + rctJYTI.Width)) then
  begin
    btnListView.Position.X := rctJYTI.Position.X + Trunc(rctJYTI.Width / 3);
    btnListView.Position.Y := Y - vsbGrid.ViewportPosition.Y - Trunc(cRowHeight / 2);
    btnListViewClick(nil);
  end;
  // 状態リストボックスタップ処理 End ------------------------------------

  // 再描画
  rctGrid.Repaint;
end;
```

8.おわりに

本稿ではトピックとなる処理を1つずつ分けてご紹介したため、難易度が高く感じられたかもしれない。しかし、実際の総ステップ数としてはわずか500ステップ程度で作成している。さらにフレーム化や共通関数化などを行うと、複数画面作成したい場合も容易に対応可能である。
また、本稿を活用すると、データ取得ロジックは共通化し、画面の見た目のみをPCとタブレットで切り替えるといったことも可能である。例えば、データ上でログイン情報と使用端末を紐づけておけば、ログインによって自動で画面を切り替えるといったことも実現できる。ぜひ、本稿が画面設計における課題解決への一助となれば幸いである。