sgcDiagram

Getting Started

Put flowcharts and diagrams your users can edit on a Delphi form, one small demo at a time

About this guide

This guide is for Delphi developers who know how to build a VCL form and want to show a diagram on it with sgcDiagram: a flowchart, a process, an architecture sketch, anything made of boxes and the lines between them. You do not need any experience with diagram components. If you can call a method and write an event handler, you can follow every step.

The guide follows the five QuickStart demos. Each demo adds one idea to the previous one, from three boxes joined by two lines to a small flowchart editor with a shape palette, rulers, a minimap, files and export. Chapter 4 walks through them one at a time: what you will build, the components on the form, the code, how it works and a few changes to try.

Read it with Delphi open next to it. Open a demo, run it, drag a few shapes around, then read its section and try the suggested changes. Chapters 1 to 3 explain what sgcDiagram is, how to install it and the ideas behind it, chapter 5 lists the main components, chapter 6 collects common problems and their fixes, and chapter 7 shows where to go next.

The demos are in the `Demos\Diagram\Delphi\00.QuickStart` folder of your sgcUI installation. Each demo has its own subfolder, from `01.HelloDiagram` to `05.DiagramFlowApp`, with a Delphi project (`.dpr`) that you can open and run directly. All of them are VCL applications for Windows, and every one builds its diagram in code, so they need no data file and no network.

The screenshots were taken on a Windows machine with Spanish regional settings. Numbers such as the zoom percentage follow the Windows regional settings of the machine that runs the demo, so they may look slightly different on yours.

Contents

1. What sgcDiagram is

The idea in one minute
Your first diagram in five lines
What you need
Part of sgcUI
Coming from drawing on a canvas

2. Install and the palette

Installing
The packages
The palette page
Editions
Where the demos are
Running your first demo

3. Core concepts

The view owns a document
Nodes, connectors and their Ids
Coordinates and colours
The view is already an editor
Change sets, undo and transactions
Events
Styles and themes
Files, export and printing
Threading
Summary: who does what

4. The QuickStart demos

4.1 Demo 01: Hello Diagram
4.2 Demo 02: Diagram Shapes
4.3 Demo 03: Diagram Events
4.4 Demo 04: Diagram Themes
4.5 Demo 05: Diagram Flow App

5. Main components

Reference table
The view
The palette and the minimap
The document and its items
Beyond the basics

6. Common problems

Building the diagram in code
Editing, undo and the view
Styles, files and editions

7. Where next

The other Delphi demos
Help topics
On the website
Getting help

1. What sgcDiagram is

sgcDiagram adds diagrams to Delphi applications, for the VCL and for FireMonkey. A diagram is a set of shapes, called nodes, joined by lines, called connectors: a flowchart, an approval process, the architecture of a system, an organisation chart. sgcDiagram draws them, routes the connectors between the nodes, and lets the user edit the result with the mouse and the keyboard: move and resize shapes, draw new connectors, edit text in place, undo and redo.

You drop a `TsgcVCLDiagramView` on a form and add nodes and connectors to it, either from your own code or by letting the user drag shapes from a `TsgcVCLDiagramPalette`. The view keeps the connectors attached when a node moves, repaints only what changed, and saves the whole diagram as a JSON file or exports it as a PNG or SVG image.

This guide is for Delphi developers who are new to sgcDiagram. By the end you will know how the pieces fit together, which properties and methods to reach for first, and how to avoid the mistakes that most people make in their first hour.

The idea in one minute

Think of a whiteboard with sticky notes and arrows. Every sticky note is a *node*: it has a shape (a rectangle, a rounded box, a diamond for a decision), a position, a size and a text. Every arrow is a *connector*: it starts at one node and ends at another, and it moves with them.

In sgcDiagram the whiteboard is the *document*, an object of the class `TsgcDiagramDocument`. The view on your form shows the document and lets the user work on it, and your code reaches the document through the `Diagram` property of the view:

- **`View.Diagram.AddNode`** adds a sticky note: a shape, a position, a size and a text.
- **`View.Diagram.AddConnector`** adds an arrow from one node to another. You never give it coordinates: the router computes the path.
- **`View.Diagram.Nodes`** and **`View.Diagram.Connectors`** hold everything on the whiteboard, and **`View.Diagram.Selection`** holds what the user has selected.

That is the entire idea. The rest of the guide is about how to style the items, how to react to what the user does, and how to make code changes undoable.

Your first diagram in five lines

The first QuickStart demo is the smallest diagram there is. The form holds one `TsgcVCLDiagramView` named `View`, with `Align = alClient` and nothing else set. All the work happens in `OnCreate`:

```
procedure TfrmHelloDiagram.FormCreate(Sender: TObject);
var
  oIdea, oBuild, oShip: TsgcDiagramNode;
begin
  // three rounded boxes on one row, 200 units apart
  oIdea := View.Diagram.AddNode(dgsRoundRect, 60, 140, 140, 60, 'Idea');
  oBuild := View.Diagram.AddNode(dgsRoundRect, 260, 140, 140, 60, 'Build');
  oShip := View.Diagram.AddNode(dgsRoundRect, 460, 140, 140, 60, 'Ship');

  // and two connectors: Idea to Build, Build to Ship
  View.Diagram.AddConnector(oIdea, oBuild);
  View.Diagram.AddConnector(oBuild, oShip);
end;
```

Run it and you get three boxes on a grid, joined by two arrows. You can already drag the boxes around and the arrows follow, double click a box to change its text, press Delete and then Ctrl+Z. No other code is needed for that: the view is an editor from the moment it is on the form.

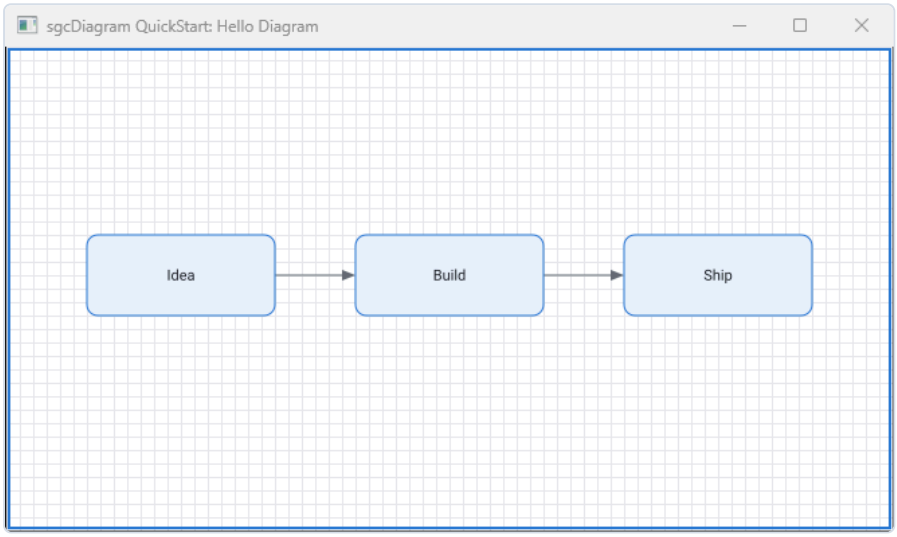


Figure 1.1. The result of demo 01: three nodes and two connectors on the default grid.

What you need

- **Delphi 7 to Delphi 13** for the VCL components, on Win32 and Win64. There is a package for every Delphi version in between, from Delphi 2007 and Delphi 2009 through the XE versions to Delphi 10.x, 11, 12 and 13.
- **Delphi XE6 to Delphi 13** for the FireMonkey components (`TsgcFMXDiagramView` and its palette and minimap), on Windows, macOS, iOS and Android.
- sgcUICore, the shared base of the sgcUI products. It is installed with sgcDiagram, so there is nothing extra to download.

This guide and its demos use the VCL. The FireMonkey components have the same properties, methods and events, so almost everything you read here applies to them as well. The document, the router and the renderer use neither the VCL nor FireMonkey, so from Delphi 10.2 a Linux64 program can also build a diagram and write PNG and SVG files without a window.

Part of sgcUI

sgcDiagram is one of the products of sgcUI, the eSeGeCe suite of visual components. All of them are built on sgcUICore, which gives them the same drawing layer, the same themes and the same accessibility support. In practice this means:

- The `Theme` property takes the same values in every sgcUI control (`utDefault`, `utLight`, `utDark`, `utHighContrast` and `utCustom`), so a diagram matches a grid or a chart from the same suite.
- Colours are `TsgcUIColor` values from the unit `sgcUICore_Canvas`, the same type every sgcUI product uses.
- Printing, the print preview and the page settings work the same way as in the other sgcUI controls.

Coming from drawing on a canvas

Many Delphi developers draw their first diagram by hand on a `TPaintBox` or a `TImage`: some rectangles, some lines, a hit test in `OnMouseDown`. That works for a static picture, and becomes a lot of code as soon as the user should move a box, redraw the lines around it, edit a text or undo a change.

sgcDiagram starts where that approach gets hard. You describe *what* is on the diagram (nodes and connectors with their styles), and the view takes care of *how* it is drawn and edited: routing, hit testing, selection handles, snapping to the grid, zoom and scroll, the in place text editor and the undo stack. Your code stays at the level of "add a decision node and connect it to the order step".

2. Install and the palette

This chapter gets sgcDiagram onto your Tool Palette and the first demo running. If the components are already installed, skim the palette section and jump to Running your first demo.

Installing

sgcDiagram is installed by the sgcUI installer, the same installer for every sgcUI product:

1. Run the sgcUI installer and sign in with the user name and password of your eSeGeCe account.
2. Choose the Delphi versions to install into.
3. On the components page, select **sgcDiagram**. The installer shows one checkbox per product and enables the products your licence includes. sgcUICore has no checkbox: it is installed automatically with sgcDiagram.
4. Optionally select **sgcDiagram for sgcWebSockets**, the bridge that feeds live status values to a diagram from a WebSocket, MQTT or HTTP client. It installs only when sgcWebSockets is present, and you do not need it for this guide.
5. The installer copies the files, compiles the packages on your machine for each selected Delphi version and installs the design time packages.

Everything goes into one folder, by default `Documents\esegece\sgcUI`. The installer adds its `Source` folder and the folder of the compiled units to the library path of each selected Delphi version, so a new project finds the sgcDiagram units without any setting on your side.

The packages

You only need this section if you build the packages yourself. sgcDiagram is split into a framework neutral package and one package per framework, so a FireMonkey application never links the VCL and the other way round. Every package name ends with the suffix of the Delphi version, for example `D13` for Delphi 13 or `D7` for Delphi 7.

Package	Kind	Contents
`sgcDiagramD13`	Runtime, framework neutral	The document, the shapes, undo, JSON, the router, the renderer, the palette and minimap models and the controller. Requires `sgcUICoreD13`.
`sgcDiagramVCLD13`	Runtime, VCL	The unit `sgcDiagram_VCL` with the view, the palette and the minimap. Requires `sgcUICoreVCLD13` and `sgcDiagramD13`.
`sgcDiagramFMXD13`	Runtime, FireMonkey	The unit `sgcDiagram_FMX`. From Delphi XE6.
`sgcDiagramWSD13`	Runtime, bridge	The sgcWebSockets status source. Optional.
`dclsgcDiagramD13`	Design time, VCL	Registers the VCL components and the component editor of the view.
`dclsgcDiagramFMXD13`, `dclsgcDiagramWSD13`	Design time	The FireMonkey components and the bridge.

Delphi 7 uses the same names with the suffix `D7`, and there is no FireMonkey package before XE6.

The palette page

After the installation, open a VCL form and look for the **SGC Diagram** page of the Tool Palette. It holds three visual components:

Component	What it does
`TsgcVCLDiagramView`	The diagram itself: shows a document and lets the user edit it. This is the one component every sgcDiagram form needs.
`TsgcVCLDiagramPalette`	A searchable grid of shapes grouped by library. A click picks the shape tool of the linked view, a drag drops the shape onto it.
`TsgcVCLDiagramMinimap`	A small overview of the whole diagram with the visible area marked, for navigating large diagrams.

In the Registered edition the same page also shows two non visual components, `TsgcDiagramDataBinding` (builds a diagram from a dataset) and `TsgcDiagramStatusBinding` (colours the items from live values), plus `TsgcDiagramWSStatusSource` when the sgcWebSockets bridge is installed. On a FireMonkey form the page shows the FireMonkey twins of the three visual components.

Right click a `TsgcVCLDiagramView` on a form and you will find three extra commands: **Load Diagram...**, **Save Diagram...** and **Clear Diagram**. A diagram loaded at design time is stored in the form file, so it is there when the form opens at run time. The QuickStart demos do not use this: they build their diagrams in code, which is easier to read in a guide.

> **Tip.** To see every property of a form at once, right click the form in the designer and choose **View as Text**. The DFM excerpts in this guide come from that view. Choose **View as Form** to go back to the designer.

Editions

sgcDiagram is sold on its own, and it is also part of sgcUI Enterprise and of the All-Access bundle. The Registered edition and the Trial edition have every feature; the trial is meant for evaluation. A Community build leaves out the advanced features: the automatic layouts, the orthogonal routing that steers around the nodes, data binding, live status, more than one layer, pools and lanes, mind maps, the import of draw.io, Visio and GraphML files, and the BPMN, UML, P&ID and mind map shape libraries. Everything else is in every edition, including all the basic shapes, the three connector kinds, undo, JSON files, PNG and SVG export, themes and printing.

> **Note.** The QuickStart demos deliberately use only features that are part of every edition. The one difference you can see is the routing: in a Community build an orthogonal connector is a simple elbow that may cross a node, while the Registered edition routes it around the nodes. Chapter 7 lists the larger demos that show the advanced features.

Where the demos are

The demos are installed under `Demos\Diagram\Delphi` in the sgcUI folder:

- `00.QuickStart` holds the five beginner demos of this guide, a `readme.html` with an overview, and this guide as `sgcDiagram_GettingStarted.pdf` and `Manual\index.html`.
- The other folders, such as `FlowchartEditor`, `OrgChart` or `Headless`, are larger applications. Chapter 7 describes each of them.

Running your first demo

1. In Delphi, choose **File, Open Project** and open `Demos\Diagram\Delphi\00.QuickStart\01.HelloDiagram\HelloDiagram.dpr`.
2. Press F9. The window from figure 1.1 appears.
3. Drag the Build box down: both connectors follow. Double click it and type a new text, then press Enter. Select it and press Delete, then Ctrl+Z to bring it back.
4. Roll the mouse wheel to scroll, and hold Ctrl while you roll it to zoom around the pointer.

If the project does not compile because a unit such as `sgcDiagram_VCL` is not found, sgcDiagram is not installed for this Delphi version, or the `Source` folder is missing from its library path. Chapter 6 has the details.

3. Core concepts

This chapter explains the ideas behind every sgcDiagram application. Read it once before chapter 4, and come back to it when a demo does something you do not expect. Each idea is short, and each one is used in at least one demo.

The view owns a document

sgcDiagram separates the diagram from the control that shows it:

- **The document**, `TsgcDiagramDocument` in the unit `sgcDiagram_Model`, holds everything that is *on* the diagram: the nodes, the connectors, the selection, the default styles, the grid size, the theme and the undo stack. It uses neither the VCL nor FireMonkey.
- **The view**, `TsgcVCLDiagramView` in the unit `sgcDiagram_VCL`, is the control on the form. It owns one document, published as the property `Diagram`, and turns the mouse and the keyboard into edits of that document.

So when you write `View.Diagram.AddNode(...)`, you are talking to the document, and the view repaints by itself because the document tells it what changed. Some commands exist on both: `View.Undo` and `View.Diagram.Undo` do the same. The view adds the things that need a screen, such as `ZoomToFit`, `SaveToPNG` or `Print`.

Three units cover almost every beginner program: `sgcDiagram_Types` for the enumerations (shapes, connector kinds, arrows, dashes, tools, change kinds), `sgcDiagram_Model` for the document and its items, and `sgcDiagram_VCL` for the controls. Add `sgcUICore_Canvas` when you use colour constants and `sgcUICore_Theme` when you set a theme in code.

Nodes, connectors and their Ids

A **node**, `TsgcDiagramNode`, is a box with a `Shape`, a position (`Left`, `Top`), a size (`Width`, `Height`), a `Text` and a `Style`. The shape is one of the values of `TsgcDiagramShape`:

Shapes	Typical use
`dgsRectangle`, `dgsRoundRect`, `dgsEllipse`	General boxes and circles.
`dgsProcess`, `dgsDecision`, `dgsTerminator`, `dgsData`, `dgsDocument`, `dgsPredefinedProcess`, `dgsConnectorCircle`	The classic flowchart symbols: a step, a yes or no question, start and end, input or output, a document, a sub process, an off page connector.
`dgsText`	A text with no border and no fill, for titles and notes.
`dgsContainer`	A box that holds other nodes.
`dgsImage`, `dgsPath`, `dgsStencil`	A picture, a custom outline, and a shape from a stencil library (the palette creates these).

A **connector**, `TsgcDiagramConnector`, joins two nodes. It has a `Kind` (`dgcStraight`, `dgcOrthogonal` or `dgcCurved`), a `StartArrow` and an `EndArrow`, `Labels` and a `Style`. You never set its points: the *router* computes the path from the two nodes, and computes it again whenever one of them moves.

Every item has an `Id`, a string that is unique in the document and is created for you by `AddNode` and `AddConnector`. A connector refers to its nodes by Id, through `SourceNodeId` and `TargetNodeId`, and `View.Diagram.FindNode(Id)` turns an Id back into the node. Each item also has a free `Tag` (an Integer) and a free `Data` (a string) for your own use, which demo 04 uses to remember the tier of each box.

Both item classes descend from `TsgcDiagramItem`, so an event that reports "an item" gives you either one, and you test it with `is`. This small routine lists every node with its Id, its text and its position:

```
procedure TfrmSnip.ListNodes(aLines: TStrings);
var
  i: Integer;
  oNode: TsgcDiagramNode;
begin
  for i := 0 to View.Diagram.Nodes.Count - 1 do
  begin
    oNode := View.Diagram.Nodes[i];
    aLines.Add(Format('%s: %s at %.0f, %.0f', [oNode.Id, oNode.Text,
      oNode.Left, oNode.Top]));
  end;
end;
```

A connector label, `TsgcDiagramLabel`, has a `Text`, a `Position` along the path (0 is the start, 1 the end) and an `Offset`, the distance from the path. The demos use the short form `AddConnector(A, B).Labels.Add.Text := 'Yes'`; the long form sets all three:

```
    oLink := oDoc.AddConnector(oCheck, oDone);
    oLabel := oLink.Labels.Add;
    oLabel.Text := 'Yes';
    oLabel.Position := 0.2;
    oLabel.Offset := 10;
```

Coordinates and colours

Positions and sizes are `Double` values in *diagram units*. One unit is one pixel at zoom 100%, with `X` growing to the right and `Y` growing downwards, so `AddNode(dgsRoundRect, 60, 140, 140, 60, 'Idea')` puts a 140 by 60 box with its top left corner at 60, 140. Zoom and scroll change only what the view shows, never the coordinates of the items. When you need to convert a mouse position, the view has `ScreenToDiagram` and `DiagramToScreen`.

Colours are `TsgcUIColor` values, a 32 bit number written as `$AARRGGBB`: alpha, red, green, blue. `$FF3B82F6` is an opaque blue. The value **0** has a special meaning: "take the colour from the theme". That is why a new node follows the theme until you give it a colour of its own, and why demo 04 sets the colours back to 0 to return to the theme.

The view is already an editor

A view on a form is a working editor without a line of code. The defaults give you a grid (`ShowGrid`), snapping to it (`SnapToGrid`), ports that appear when the pointer is over a node (`ShowPorts = dgvHover`) and the selection tool. The user can:

- click to select, Shift and click to add to the selection, drag to move, drag the handles to resize;
- drag from a port to draw a new connector;

- double click a node, press F2 or simply start typing to edit its text in place, then Enter to confirm or Esc to cancel;
- press Delete, Ctrl+Z, Ctrl+Y, Ctrl+C, Ctrl+V and Ctrl+A;
- scroll with the wheel and zoom with Ctrl and the wheel, Ctrl+0 for zoom 1 and Ctrl+1 to fit the diagram in the view.

The `Tool` property decides what a drag on the view does: `dgtSelect` (the default), `dgtConnector` (a drag from any node draws a connector), `dgtPan` (a drag scrolls) and `dgtShape` (a click creates a node of the shape in `ShapeTool`; a click on a palette item sets this tool). Set `ReadOnly` to `True` when the user should only look: then only zooming, selecting all, copying and Esc work.

Change sets, undo and transactions

Every change to the document, made by the user or by your code, is collected and reported to the view, which repaints the area that changed. Three pairs of methods control how changes are grouped, and all of them are used in the demos:

Methods	What they do	When to use them
`BeginUpdate`, `EndUpdate`	Collect the changes and report them once, at the outermost `EndUpdate`. The connectors are routed at that moment.	When you add many items at once, so the view paints once.
`BeginTransaction`, `EndTransaction`	Record everything in between as **one undo step** with the caption you pass. `Rollback` cancels it.	For every change made by code that the user should be able to undo.
`BeginUndoSuspend`, `EndUndoSuspend`	Do not record the changes at all.	When you build the initial diagram, so Ctrl+Z cannot remove it.

Every action of the user is one undo step on its own. A change made by code *outside* a transaction is not recorded, and it also clears the undo and redo stacks, because the recorded steps would no longer match the document. So the rule is simple: wrap code edits in `BeginTransaction` and `EndTransaction`, and the initial build in `BeginUndoSuspend` and `EndUndoSuspend`. `CanUndo`, `CanRedo`, `UndoCaption` and `RedoCaption` tell you what the next undo or redo would do, which is handy for enabling buttons.

Events

The view publishes events for what the user does. The ones you will use first:

Event	When it fires
`OnChange(Sender; aChanges)`	After every change set. `aChanges` is a set of `dgkGeometry`, `dgkStyle`, `dgkStructure`, `dgkSelection` and `dgkView`, so you can tell an edit from a selection change or a scroll.
`OnSelectionChange`	When the selection changes, by a click or by code.
`OnItemClick`, `OnItemDblClick`	When the user clicks or double clicks a node or a connector.
`OnTextEditing`, `OnTextEdited`	Before the in place editor opens (set `aAllow` to `False` to prevent it) and after a new text is confirmed.
`OnConnecting`, `OnConnected`	While the user draws a connector (set `aAccept` to `False` to refuse it) and after it is created.
`OnZoomChange`, `OnHover`, `OnDrawNode`	When the zoom changes, when the pointer moves over an item, and when a node is painted, for custom drawing.

Two examples of the events that let you say no. The first keeps the text of the nodes whose `Tag` is 1, the second refuses loose connector ends and connectors from a node to itself:

```
procedure TfrmSnip.ViewTextEditing(Sender: TObject; aNode: TsgcDiagramNode;
  var aAllow: Boolean);
begin
  // the title node keeps its text
  aAllow := aNode.Tag <> 1;
end;

procedure TfrmSnip.ViewConnecting(Sender: TObject; aSource: TsgcDiagramNode;
  const aSourcePort: string; aTarget: TsgcDiagramNode;
  const aTargetPort: string; var aAccept: Boolean);
begin
  // no loose ends and no connector from a node to itself
  aAccept := (aTarget <> nil) and (aTarget <> aSource);
end;
```

Styles and themes

Every node and every connector has its own `Style`, a `TsgcDiagramStyle` with `Fill`, `Stroke`, `StrokeWidth`, `Dash` (`dgdSolid`, `dgdDash`, `dgdDot`, `dgdDashDot`), `CornerRadius` and `Text` (font size, bold, colour and so on). Colours left at 0 come from the theme.

The document has two default styles, `NodeStyle` and `ConnectorStyle`, and a `DefaultConnectorKind`. They are *templates*: `AddNode` copies `NodeStyle` into the new node, and `AddConnector` copies `ConnectorStyle` and `DefaultConnectorKind` into the new connector. Changing a default later affects only the items created afterwards, never the existing ones. Demo 04 sets the defaults in the Object Inspector, so every item created in code gets them.

The theme decides the colours of everything left at 0, and of the background, the grid and the selection. At design time you set `Diagram.Theme`, which is stored in the form file; in code you can set `View.Theme`, which changes the same value:

```
  View.Theme := utDark;
```

Files, export and printing

The document saves itself as JSON with `SaveToFile`, `SaveToStream` and `SaveToJSON`, and reads it back with `LoadFromFile`, `LoadFromStream` and `LoadFromJSON`. The file contains the nodes, the connectors, their styles and the document settings, and it is the same in every edition. The view exports images with `SaveToPNG` (with an optional scale, 2 doubles the pixels) and `SaveToSVG`. An export always contains the whole diagram, not only the part scrolled into view, and never the selection handles or the other editing marks of the view.

The view also prints: `Print` sends the diagram to a printer, `ShowPrintPreview` opens the sgcUI print preview, and `PrintSettings` holds the paper, margins and scaling. `ExportToPDF` exists as well, but it needs the sgcPDF bridge, see chapter 6.

Threading

A document is changed only in the thread that owns it, and that is the main thread when the document is shown in a view. If a background thread computes a diagram, let it collect the data, then add the nodes and connectors in the main thread, for example with `TThread.Synchronize` or `TThread.Queue`.

Summary: who does what

You want to	Use
Add a box or a line in code	`View.Diagram.AddNode`, `View.Diagram.AddConnector`
Walk through the diagram	`View.Diagram.Nodes`, `Connectors`, `Selection`, `FindNode`
Make a code edit undoable	`BeginTransaction` and `EndTransaction`
Build the start diagram	`BeginUndoSuspend`, `BeginUpdate` and their partners
Change the look	`Theme`, the item `Style`, the defaults `NodeStyle` and `ConnectorStyle`
React to the user	The events of the view
Save, open, export, print	`Diagram.SaveToFile`, `Diagram.LoadFromFile`, `View.SaveToPNG`, `View.SaveToSVG`, `View.Print`
Fit the diagram in the window	`View.ZoomToFit`, in `OnShow` or later

4. The QuickStart demos

The five QuickStart demos build on each other. Run them in order: each one adds one or two ideas to the previous one, and the sections below assume you have seen the earlier ones. In every demo the view is named `View`, so the code always reads `View.Diagram.AddNode(...)`.

Demo	Project	New ideas
01.HelloDiagram	`HelloDiagram.dpr`	The view, the document, `AddNode` and `AddConnector`, the built in editor
02.DiagramShapes	`DiagramShapes.dpr`	Flowchart shapes, connector labels, connector kinds, building without undo, a transaction
03.DiagramEvents	`DiagramEvents.dpr`	The events of the view, the selection, undo and redo buttons, a code edit as one undo step
04.DiagramThemes	`DiagramThemes.dpr`	Themes, default styles, per item colours, dashes and arrows
05.DiagramFlowApp	`DiagramFlowApp.dpr`	The palette and the minimap, tools, rulers, JSON files, PNG and SVG export

> **Note.** The demos need sgcDiagram installed, so the components appear on the SGC Diagram palette page and the `sgcUI\Source` folder is on the library path. They use only features that are part of every edition, and they compile with every Delphi version from Delphi 7 to Delphi 13.

4.1 Demo 01: Hello Diagram

Folder `01.HelloDiagram`, project `HelloDiagram.dpr`.

What you will build

The smallest sgcDiagram application. Three rounded boxes, Idea, Build and Ship, stand on one row, joined by two arrows. Everything else, the grid, the dragging, the text editor and undo, comes from the view itself.

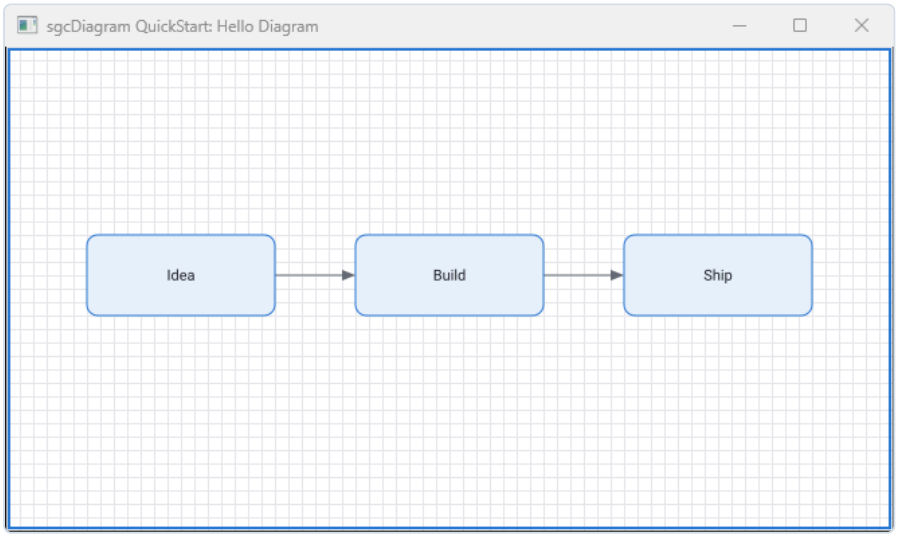


Figure 4.1. Demo 01. The grid and the blue boxes are the defaults of the view and of the default theme.

Components on the form

Component	Class	Properties you set
`View`	`TsgcVCLDiagramView`	`Align = alClient`. Nothing else: the defaults give a grid, snapping and orthogonal connectors.

The form file shows how little there is:

```
  object View: TsgcVCLDiagramView
    Left = 0
    Top = 0
    Width = 660
    Height = 360
    Align = alClient
    TabOrder = 0
  end
```

The code

The `uses` clause names the three units of sgcDiagram that a beginner needs: the types, the model and the VCL controls.

```
uses
  SysUtils, Classes, Controls, Forms,
  // sgc
  sgcDiagram_Types, sgcDiagram_Model, sgcDiagram_VCL;
```

And this is the only event handler of the demo:

```
procedure TfrmHelloDiagram.FormCreate(Sender: TObject);
var
  oIdea, oBuild, oShip: TsgcDiagramNode;
begin
  // three rounded boxes on one row, 200 units apart
  oIdea := View.Diagram.AddNode(dgsRoundRect, 60, 140, 140, 60, 'Idea');
  oBuild := View.Diagram.AddNode(dgsRoundRect, 260, 140, 140, 60, 'Build');
  oShip := View.Diagram.AddNode(dgsRoundRect, 460, 140, 140, 60, 'Ship');

  // and two connectors: Idea to Build, Build to Ship
  View.Diagram.AddConnector(oIdea, oBuild);
  View.Diagram.AddConnector(oBuild, oShip);
end;
```

How it works

1. The form loads from its DFM. The view creates its document, empty, with the default theme, a grid of 10 units and orthogonal connectors.
2. `AddNode(Shape, X, Y, Width, Height, Text)` creates a node, gives it a new Id and the style of `Diagram.NodeStyle`, adds it to `Diagram.Nodes` and returns it. X and Y are the top left corner in diagram units.
3. `AddConnector(Source, Target)` creates a connector of `Diagram.DefaultConnectorKind` with an arrow at the target. The router computes its path.
4. Each call is reported to the view at once, which repaints. These five changes are made outside a transaction, so they are not undo steps: Ctrl+Z cannot remove them.

5. From now on the user edits the document. Every drag, delete or text edit is one undo step.

Try this

- Change the shape of the boxes to `dgsEllipse` or `dgsDecision`.
- Set `ShowGrid` to `False` in the Object Inspector.
- Add `View.Diagram.AddConnector(oShip, oIdea);` and watch the router bring the line back from Ship to Idea. In the Registered edition it goes around the Build box; in a Community build the simple elbow may cross it.

4.2 Demo 02: Diagram Shapes

Folder `02.DiagramShapes`, project `DiagramShapes.dpr`.

What you will build

An order process drawn as a real flowchart: Start, Order received, the decision In stock?, Pick and pack, Order from supplier, Invoice and End, with a title in the corner and Yes and No labels on the two answers of the decision. A combo box at the top switches every connector between straight, orthogonal and curved lines.

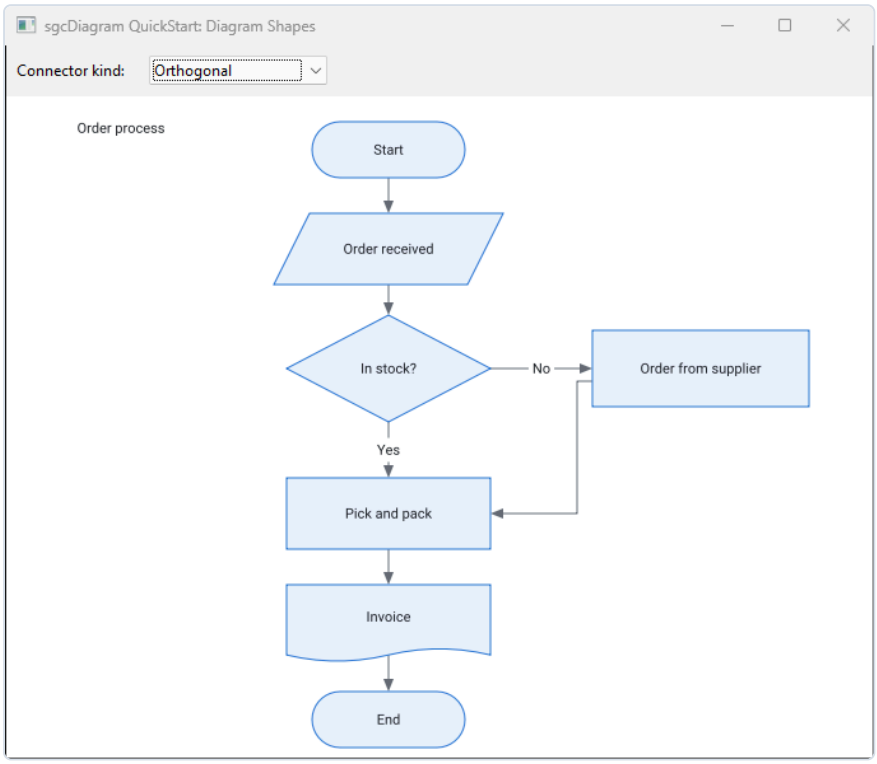


Figure 4.2. Demo 02. Each step uses the flowchart symbol for its meaning; the decision has a Yes and a No branch.

Components on the form

Component	Class	Properties you set
`pnlTop`	`TPanel`	`Align = alTop`, holds the label and the combo box.
`cboKind`	`TComboBox`	`Style = csDropDownList`, items Straight, Orthogonal and Curved, in the order of `TsgcDiagramConnectorKind`, `ItemIndex = 1`, `OnChange = cboKindChange`.
`View`	`TsgcVCLDiagramView`	`Align = alClient`, `ShowGrid = False`, `Diagram.GridSize = 20`.

`Diagram.GridSize` is a property of the document, set through the view in the Object Inspector. Expand `Diagram` to find it. The grid is hidden, but the nodes the user moves still snap to it:

```
  object View: TsgcVCLDiagramView
    Left = 0
    Top = 40
    Width = 680
    Height = 520
    Align = alClient
    TabOrder = 1
    Diagram.GridSize = 20
    ShowGrid = False
  end
```

The code

The flowchart is built in one method. The two brackets around it make the build fast and keep it out of the undo history:

```
procedure TfrmDiagramShapes.DoBuildFlowchart;
var
  oDoc: TsgcDiagramDocument;
  oStart, oOrder, oStock, oSupplier, oPick, oInvoice,
    oEnd: TsgcDiagramNode;
begin
  oDoc := View.Diagram;
  { BeginUndoSuspend: building the sample is not an undo step, so Ctrl+Z
    cannot remove it. BeginUpdate: the view paints once, at the end. }
  oDoc.BeginUndoSuspend;
  try
    oDoc.BeginUpdate;
    try
      // a title: dgsText has no border and no fill
      oDoc.AddNode(dgsText, 10, 10, 160, 30, 'Order process');

      // the main flow, top to bottom, centred on x = 300
      oStart := oDoc.AddNode(dgsTerminator, 240, 20, 120, 44, 'Start');
      oOrder := oDoc.AddNode(dgsData, 210, 92, 180, 56, 'Order received');
      oStock := oDoc.AddNode(dgsDecision, 220, 172, 160, 84, 'In stock?');
      oPick := oDoc.AddNode(dgsProcess, 220, 300, 160, 56, 'Pick and pack');
      oInvoice := oDoc.AddNode(dgsDocument, 220, 384, 160, 60, 'Invoice');
      oEnd := oDoc.AddNode(dgsTerminator, 240, 468, 120, 44, 'End');
      // the side branch, to the right of the decision
      oSupplier := oDoc.AddNode(dgsProcess, 460, 184, 170, 60,
        'Order from supplier');

      oDoc.AddConnector(oStart, oOrder);
      oDoc.AddConnector(oOrder, oStock);
      // the two answers of the decision, each with a label
      oDoc.AddConnector(oStock, oPick).Labels.Add.Text := 'Yes';
      oDoc.AddConnector(oStock, oSupplier).Labels.Add.Text := 'No';
      oDoc.AddConnector(oSupplier, oPick);
      oDoc.AddConnector(oPick, oInvoice);
      oDoc.AddConnector(oInvoice, oEnd);
    finally
      oDoc.EndUpdate;
    end;
  finally
    oDoc.EndUndoSuspend;
  end;
end;
```

The combo box changes the kind of every connector. The loop runs inside a transaction, so the user can undo the whole change with one Ctrl+Z:

```
procedure TfrmDiagramShapes.cboKindChange(Sender: TObject);
var
  oDoc: TsgcDiagramDocument;
  vKind: TsgcDiagramConnectorKind;
  i: Integer;
begin
  oDoc := View.Diagram;
  vKind := TsgcDiagramConnectorKind(cboKind.ItemIndex);
  // the connectors drawn from now on
  oDoc.DefaultConnectorKind := vKind;
  // and the existing ones, in one undo step
  oDoc.BeginTransaction('Connector kind');
  try
    for i := 0 to oDoc.Connectors.Count - 1 do
      oDoc.Connectors[i].Kind := vKind;
  finally
    oDoc.EndTransaction;
  end;
end;
```

How it works

1. `BeginUndoSuspend` switches off the recording of undo steps, and `BeginUpdate` holds back the change notifications.
2. The `AddNode` calls create the title (`dgsText`) and the seven flowchart nodes, each with the shape that matches its meaning.
3. `AddConnector` returns the new connector, so `.Labels.Add.Text := 'Yes'` adds a label to it in the same line.
4. `EndUpdate` routes all the connectors and repaints the view once. `EndUndoSuspend` switches recording back on.
5. When the user picks a kind, `DefaultConnectorKind` changes the template for new connectors, and the transaction changes the existing ones as one undo step named “Connector kind”.

> **Tip.** Casting `ItemIndex` to `TsgcDiagramConnectorKind` works only because the items of the combo box are in the same order as the enumeration: `dgcStraight` , `dgcOrthogonal` , `dgcCurved` .

Try this

- Set `Diagram.DefaultConnectorKind` to `dgcCurved` in the Object Inspector and run again.
- Add a `dgsPredefinedProcess` node for a sub process and connect it.
- Give the Yes label a `Position` of 0.2, so it sits near the decision. `Position` goes from 0 to 1 along the path, and `Offset` is the distance from it (see chapter 3).

4.3 Demo 03: Diagram Events

Folder `03.DiagramEvents`, project `DiagramEvents.dpr`.

What you will build

A purchase approval process with a log on the right that records the events of the view as you click, select and edit. Buttons add a step below the selected node, delete the selection, undo and redo, and a status bar shows the counts. At start the Manager review node is selected in code, so the first line of the log is already there.

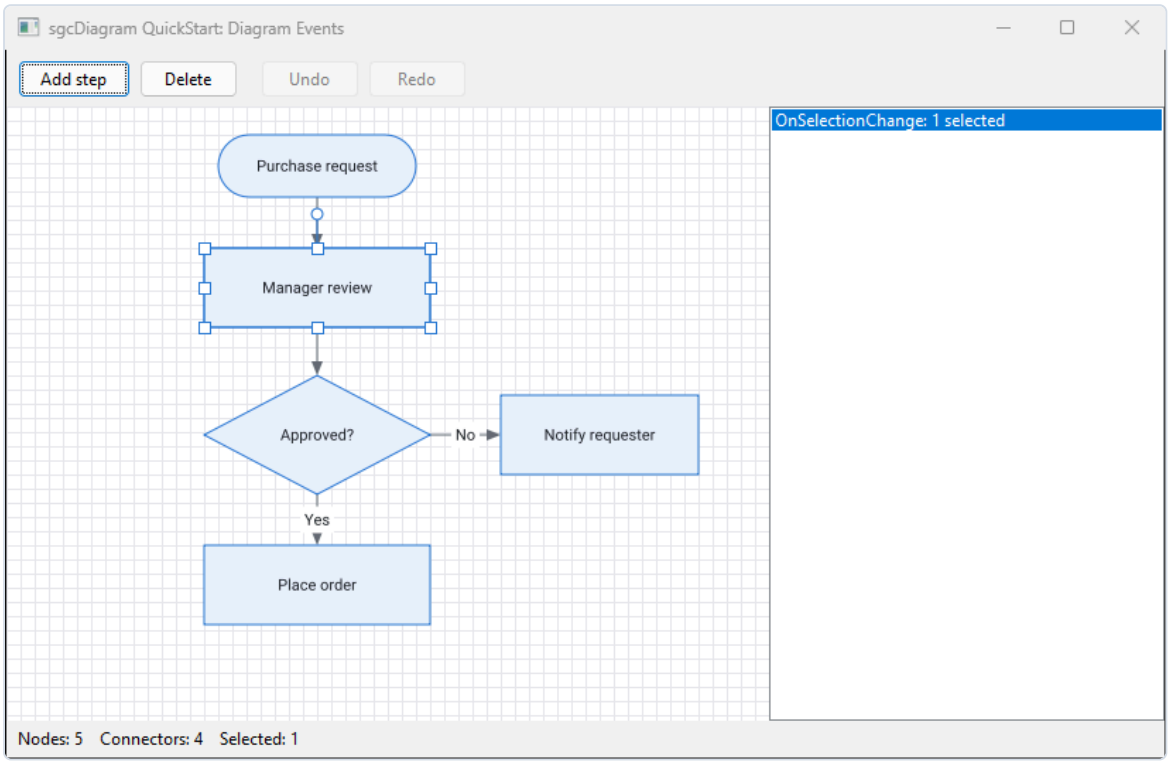


Figure 4.3. Demo 03. The log shows the events in the order they fire; the status bar is refreshed from `OnChange`.

Components on the form

Component	Class	Properties you set
`pnlTop`	`TPanel`	`Align = alTop`, with `btnAdd`, `btnDelete`, `btnUndo` and `btnRedo`.
`lstLog`	`TListBox`	`Align = alRight`.
`View`	`TsgcVCLDiagramView`	`Align = alClient`, `OnChange`, `OnSelectionChange`, `OnItemClick`, `OnTextEdited`.
`pnlStatus`, `lblStatus`	`TPanel`, `TLabel`	`Align = alBottom`, the status text.

The code

The process is built like the flowchart of demo 02. The last line selects a node in code, which fires `OnSelectionChange`:

```
  oDoc := View.Diagram;
  oDoc.BeginUndoSuspend;
  try
    // ...
    oDoc.AddConnector(oApproved, oPurchase).Labels.Add.Text := 'Yes';
    oDoc.AddConnector(oApproved, oReject).Labels.Add.Text := 'No';
  finally
    oDoc.EndUndoSuspend;
  end;
  // select a node in code: OnSelectionChange fires and writes the first
  // line of the log
  oDoc.Selection.Add(oReview);
```

`OnItemClick` receives a `TsgcDiagramItem`, which is either a node or a connector. A connector knows the Ids of its two nodes, and `FindNode` returns them:

```
procedure TfrmDiagramEvents.ViewItemClick(Sender: TObject;
  aItem: TsgcDiagramItem);
var
  oConnector: TsgcDiagramConnector;
  oSource, oTarget: TsgcDiagramNode;
begin
  if aItem is TsgcDiagramNode then
    DoLog('OnItemClick: node "' + TsgcDiagramNode(aItem).Text + '"')
  else if aItem is TsgcDiagramConnector then
  begin
    // a connector knows the Id of its nodes, FindNode returns them
    oConnector := TsgcDiagramConnector(aItem);
    oSource := View.Diagram.FindNode(oConnector.SourceNodeId);
    oTarget := View.Diagram.FindNode(oConnector.TargetNodeId);
    if (oSource <> nil) and (oTarget <> nil) then
      DoLog('OnItemClick: connector "' + oSource.Text + '" to "' +
        oTarget.Text + '"');
  end;
end;
```

The other three event handlers are short. `OnChange` fires after every change set, including every step of a drag, so it only refreshes the status:

```
procedure TfrmDiagramEvents.ViewSelectionChange(Sender: TObject);
begin
  DoLog(Format('OnSelectionChange: %d selected',
    [View.Diagram.Selection.Count]));
end;

procedure TfrmDiagramEvents.ViewTextEdited(Sender: TObject;
  aNode: TsgcDiagramNode);
begin
  DoLog('OnTextEdited: "' + aNode.Text + '"');
end;

procedure TfrmDiagramEvents.ViewChange(Sender: TObject;
  aChanges: TsgcDiagramChanges);
begin
  // fires often (every step of a drag): keep it light
  DoUpdateStatus;
end;
```

The status shows the counts and enables the buttons from the state of the document:

```
procedure TfrmDiagramEvents.DoUpdateStatus;
var
  oDoc: TsgcDiagramDocument;
begin
  oDoc := View.Diagram;
  lblStatus.Caption := Format('Nodes: %d    Connectors: %d    Selected: %d',
    [oDoc.Nodes.Count, oDoc.Connectors.Count, oDoc.Selection.Count]);
  btnDelete.Enabled := oDoc.Selection.Count > 0;
  btnUndo.Enabled := oDoc.CanUndo;
  btnRedo.Enabled := oDoc.CanRedo;
end;
```

The Add step button is the most instructive part: a node and a connector added by code, as one undo step. `DoSelectedNode` returns the first selected node, or nil:

```
procedure TfrmDiagramEvents.btnAddClick(Sender: TObject);
var
  oDoc: TsgcDiagramDocument;
  oParent, oNode: TsgcDiagramNode;
  vX, vY: Double;
begin
  oDoc := View.Diagram;
  oParent := DoSelectedNode;
  Inc(FSteps);
  // below the selected node, or in the top left corner without one
  if oParent <> nil then
  begin
    vX := oParent.Left;
    vY := oParent.Top + oParent.Height + 40;
  end
  else
  begin
    vX := 20;
    vY := 20;
  end;
  oDoc.BeginTransaction('Add step');
  try
    oNode := oDoc.AddNode(dgsProcess, vX, vY, 160, 56,
      'Step ' + IntToStr(FSteps));
    if oParent <> nil then
      oDoc.AddConnector(oParent, oNode);
  finally
    oDoc.EndTransaction;
  end;
  // select the new node, so the next click adds below it
  oDoc.Selection.Clear;
  oDoc.Selection.Add(oNode);
  View.SetFocus;
end;
```

The other three buttons call one method of the view each: `View.DeleteSelection`, `View.Undo` and `View.Redo`.

How it works

1. The build runs with undo suspended, then `Selection.Add` selects Manager review. The selection is part of the document, so the change fires `OnSelectionChange` and the log gets its first line.
2. A click on an item fires `OnItemClick`, and if the selection changes, `OnSelectionChange` as well. Watch the order in the log.

3. A double click opens the in place editor. When you confirm with Enter, `OnTextEdited` reports the new text.
4. Every change, including a selection change, fires `OnChange`, which updates the counts and the buttons. After the first edit Undo becomes enabled.
5. Add step wraps `AddNode` and `AddConnector` in a transaction named “Add step”. One Undo removes both. Without the transaction the change would not be undoable, and it would also clear the undo history.

Try this

- Add an `OnItemDblClick` handler that shows the text of the node. It also fires with `ReadOnly = True`, when the in place editor does not open.
- Add an `OnTextEditing` handler that sets `aAllow := False` for one node, so its text cannot be edited (chapter 3 has an example).
- Show `View.Diagram.UndoCaption` in the status bar, so the user sees “Add step” before pressing Undo.

4.4 Demo 04: Diagram Themes

Folder `04.DiagramThemes`, project `DiagramThemes.dpr`.

What you will build

The architecture of an online shop: a web browser and a mobile app at the top, an API gateway, three services and, at the bottom, a database and a cache. The boxes are coloured by tier, the connectors are curved, and the link between the orders service and the cache is dashed with an arrow at both ends. A combo box switches between the light and the dark theme, and a check box switches the tier colours on and off.

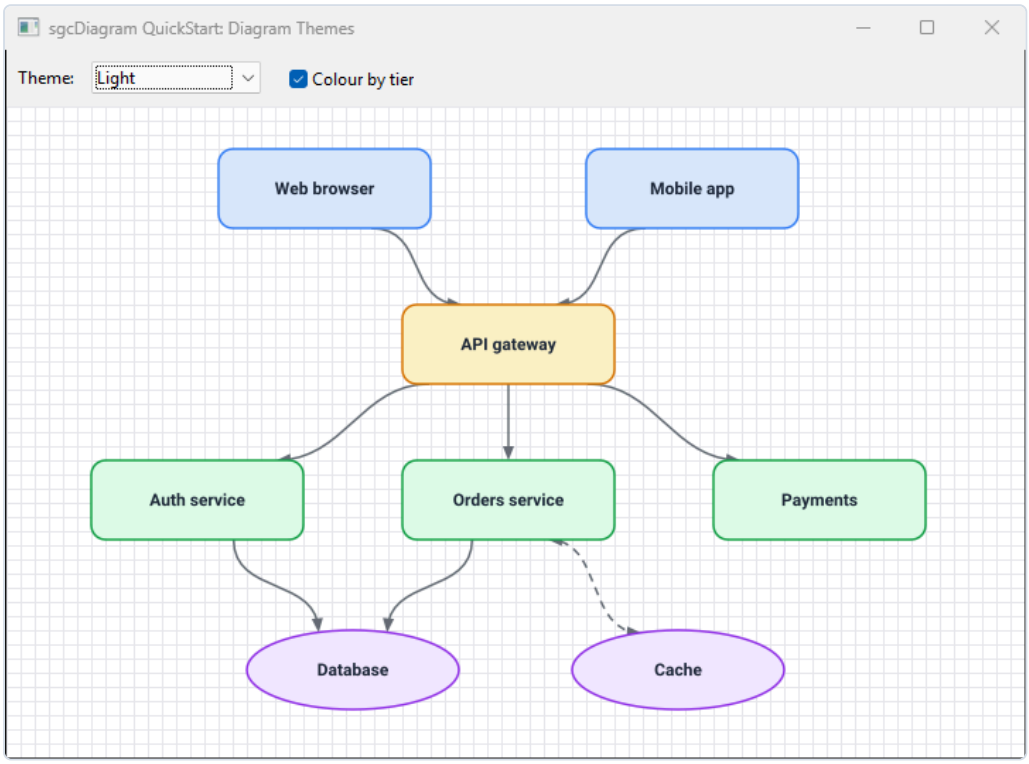


Figure 4.4. Demo 04. Blue for the clients, amber for the gateway, green for the services and purple for the data tier.

Components on the form

Component	Class	Properties you set
`cboTheme`	`TComboBox`	Items Light and Dark, `ItemIndex = 0`, `OnChange = cboThemeChange`.
`chkColours`	`TCheckBox`	`Caption = 'Colour by tier'`, `Checked = True`, `OnClick = chkColoursClick`.
`View`	`TsgcVCLDiagramView`	The theme and the default styles of the document, see below.

This time the document is configured in the Object Inspector: expand `Diagram`, then `NodeStyle` and `ConnectorStyle`. Every node and connector created in code starts with these values:

```
  object View: TsgcVCLDiagramView
    Left = 0
    Top = 40
    Width = 720
    Height = 460
    Align = alClient
    TabOrder = 1
    Diagram.Theme = utLight
    Diagram.NodeStyle.StrokeWidth = 1.500000000000000000
    Diagram.NodeStyle.CornerRadius = 10.000000000000000000
    Diagram.NodeStyle.Text.Size = 12.000000000000000000
    Diagram.NodeStyle.Text.Bold = True
    Diagram.ConnectorStyle.StrokeWidth = 1.500000000000000000
    Diagram.DefaultConnectorKind = dgcCurved
  end
```

The code

The colours are constants of type `TsgcUIColor`, written as `$AARRGGBB`. This is why the `uses` clause adds `sgcUICore_Canvas`, and `sgcUICore_Theme` for `utLight` and `utDark`:

```
uses
  SysUtils, Classes, Controls, Forms, StdCtrls, ExtCtrls,
  // sgc
  sgcUICore_Canvas, sgcUICore_Theme, sgcDiagram_Types, sgcDiagram_Model,
  sgcDiagram_VCL;

// ...

const
  { the tier of a node is kept in its Tag: 0 clients, 1 gateway,
    2 services, 3 data }
  CS_FILLS: array [0 .. 3] of TsgcUIColor = ($FFDBEAFE, $FFFEF3C7, $FFDCFCE7,
    $FFF3E8FF);
  CS_STROKES: array [0 .. 3] of TsgcUIColor = ($FF3B82F6, $FFD97706,
    $FF16A34A, $FF9333EA);
  { dark text, readable on every fill in the light and the dark theme }
  CS_TEXT = $FF1E293B;
```

The build uses a nested helper that stores the tier of each node in its `Tag`, and gives the cache link its own dash and arrows:

```
  function DoAdd(aShape: TsgcDiagramShape; const aX, aY: Double;
    const aText: string; aTier: Integer): TsgcDiagramNode;
  begin
    Result := oDoc.AddNode(aShape, aX, aY, 150, 56, aText);
    // Tag is free for the application
    Result.Tag := aTier;
  end;

begin
  oDoc := View.Diagram;
  oDoc.BeginUndoSuspend;
  try
    oBrowser := DoAdd(dgsRoundRect, 150, 30, 'Web browser', 0);
    oMobile := DoAdd(dgsRoundRect, 410, 30, 'Mobile app', 0);
    oGateway := DoAdd(dgsRoundRect, 280, 140, 'API gateway', 1);
    // ...
    oDatabase := DoAdd(dgsEllipse, 170, 370, 'Database', 3);
    oCache := DoAdd(dgsEllipse, 400, 370, 'Cache', 3);

    oDoc.AddConnector(oBrowser, oGateway);
    // ...
    oDoc.AddConnector(oOrders, oDatabase);
    // the cache link: dashed, with an arrow at both ends
    oLink := oDoc.AddConnector(oOrders, oCache);
    oLink.Style.Dash := dgdDash;
    oLink.StartArrow := dgaTriangle;
    oLink.EndArrow := dgaTriangle;
  finally
    oDoc.EndUndoSuspend;
  end;
end;
```

The colours are applied, or removed, by one loop over `Diagram.Nodes`. Setting a colour to 0 hands it back to the theme:

```
      for i := 0 to oDoc.Nodes.Count - 1 do
      begin
        oNode := oDoc.Nodes[i];
        if aColoured then
        begin
          oNode.Style.Fill := CS_FILLS[oNode.Tag];
          oNode.Style.Stroke := CS_STROKES[oNode.Tag];
          oNode.Style.Text.Color := CS_TEXT;
        end
        else
        begin
          // 0: the colours of the theme
          oNode.Style.Fill := 0;
          oNode.Style.Stroke := 0;
          oNode.Style.Text.Color := 0;
        end;
      end;
```

And the theme is one property:

```
procedure TfrmDiagramThemes.cboThemeChange(Sender: TObject);
begin
  if cboTheme.ItemIndex = 1 then
    View.Theme := utDark
  else
    View.Theme := utLight;
end;
```

How it works

1. When the form loads, the document gets the light theme, the default node style (thicker outline, rounded corners, bold 12 point text), the default connector width and curved connectors.
2. Each `AddNode` copies `NodeStyle` into the new node, and each `AddConnector` copies `ConnectorStyle` and the curved kind into the new connector.
3. The cache link then changes its own `Style.Dash` and its arrows. Only that connector is affected.
4. `DoApplyColours` writes the tier colours into the `Style` of each node, in one `BeginUpdate` so the view paints once. Without the colours, every value is 0 and the theme decides.
5. Switching to `utDark` changes the background, the grid and every colour left at 0. The tier colours are explicit, so they stay; the constant `CS_TEXT` was chosen to be readable on them in both themes.

> **Watch out.** `NodeStyle` and `ConnectorStyle` are copied when an item is created. Changing `Diagram.NodeStyle` after the nodes exist does not restyle them; change the `Style` of each node instead, as `DoApplyColours` does.

Try this

- Set `Diagram.NodeStyle.Dash` to `dgdDot` in the Object Inspector and run again: every node has a dotted outline.
- Make the API gateway stand out: after it is created, set `oGateway.Style.StrokeWidth := 3` and `oGateway.Style.Text.Size := 14`.
- Colour the form to match the dark theme with `sgcUIColorToVCL(sgcUIThemeColors(utDark).Background)`, adding `sgcUICore_Theme_VCL` to the `uses` clause.

4.5 Demo 05: Diagram Flow App

Folder `05.DiagramFlowApp`, project `DiagramFlowApp.dpr`.

What you will build

A small flowchart editor. A palette on the left offers the flowchart and basic shapes with a search box; the view has rulers; a minimap in the bottom right corner shows the whole chart. The toolbar creates, opens and saves diagrams as JSON, exports PNG and SVG images, undoes and redoes, deletes, fits the chart in the window and switches the tool between Select, Connect and Pan. The status bar shows the counts and the zoom, and the caption shows the file name with a `*` when there are unsaved changes. A support process is built at start, so the editor does not open empty.

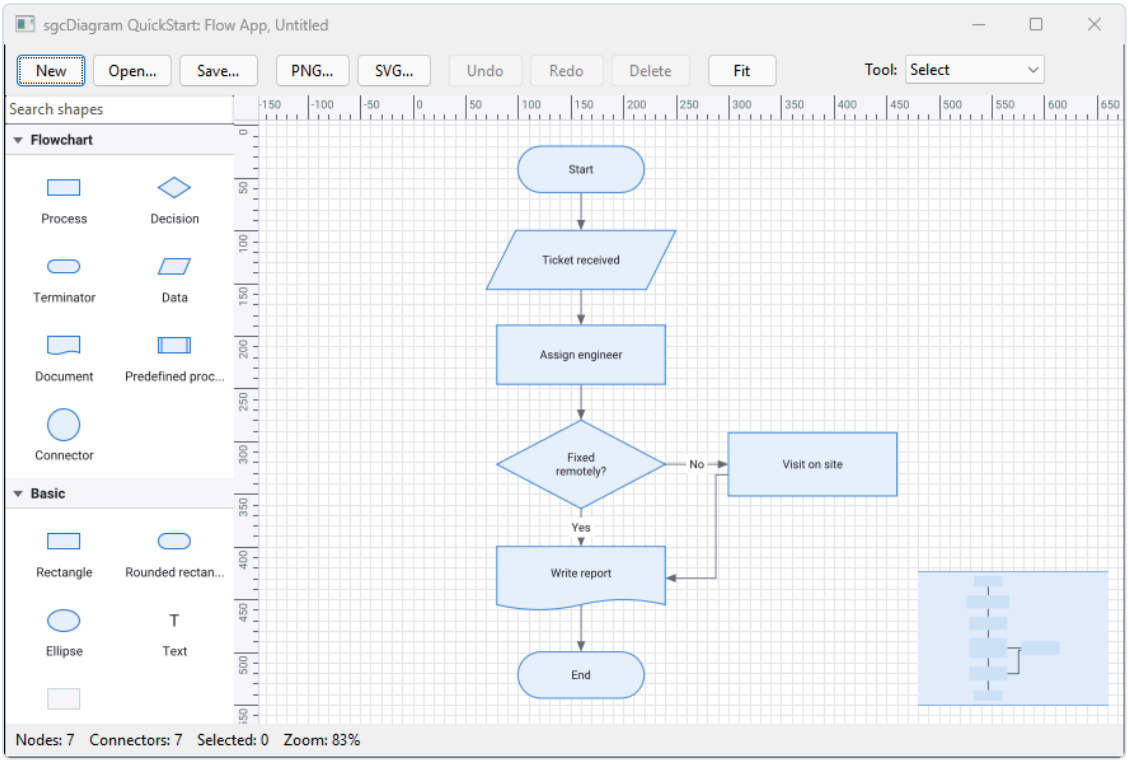


Figure 4.5. Demo 05. The palette, the view and the minimap are three components linked through the `View` property.

Components on the form

Component	Class	Properties you set
`Palette`	`TsgcVCLDiagramPalette`	`Align = alLeft`, `View = View`, `Libraries` = flowchart and basic, `ShowSearch = True`.
`View`	`TsgcVCLDiagramView`	`Align = alClient`, `ShowRulers = True`, `OnChange`, `OnZoomChange`.
`Minimap`	`TsgcVCLDiagramMinimap`	`View = View`, 150 by 106, `Anchors = [akRight, akBottom]`, placed over the bottom right corner of the view.
`dlgOpen`, `dlgSave`	`TOpenDialog`, `TSaveDialog`	`DefaultExt = 'json'`, `Filter = 'Diagram (*.json)|*.json'`.
`dlgExport`	`TSaveDialog`	Filter set in code for PNG or SVG.
Toolbar and status	`TPanel`, `TButton`, `TComboBox`, `TLabel`	`pnlTop` with the buttons and `cboTool`, `pnlStatus` with `lblStatus`.

The three diagram components in the form file. The minimap comes after the view, so it stays on top of it:

```
  object Palette: TsgcVCLDiagramPalette
    Left = 0
    Top = 40
    Width = 180
    Height = 494
    Align = alLeft
    TabOrder = 1
    View = View
    Libraries.Strings = (
      'flowchart'
      'basic')
    ShowSearch = True
  end
  object View: TsgcVCLDiagramView
    // ...
    ShowRulers = True
    OnChange = ViewChange
    OnZoomChange = ViewZoomChange
  end
  object Minimap: TsgcVCLDiagramMinimap
    Left = 718
    Top = 414
    Width = 150
    Height = 106
    Anchors = [akRight, akBottom]
    TabOrder = 3
    View = View
  end
```

The code

The sample is built in `OnCreate` exactly like the flowchart of demo 02. Fitting it in the window has to wait for `OnShow`, when the view has its final size:

```
procedure TfrmDiagramFlowApp.FormCreate(Sender: TObject);
begin
  DoBuildSample;
  FModified := False;
  DoUpdateStatus;
end;


procedure TfrmDiagramFlowApp.FormShow(Sender: TObject);
begin
  // the view has its final size now: show the whole sample
  View.ZoomToFit;
end;
```

`OnChange` decides whether the diagram was modified. Selecting, zooming and scrolling are changes too, but they do not touch the diagram itself:

```
procedure TfrmDiagramFlowApp.ViewChange(Sender: TObject;
  aChanges: TsgcDiagramChanges);
begin
  // selection and view (zoom, scroll) changes do not modify the diagram
  if aChanges * [dgkGeometry, dgkStyle, dgkStructure] <> [] then
    FModified := True;
  DoUpdateStatus;
end;
```

The file buttons are one call each on the document. `DoCanDiscard` asks before unsaved changes are lost:

```
procedure TfrmDiagramFlowApp.btnNewClick(Sender: TObject);
begin
  if not DoCanDiscard then
    Exit;
  View.Diagram.Clear;
  FFileName := '';
  FModified := False;
  DoUpdateStatus;
end;


procedure TfrmDiagramFlowApp.btnOpenClick(Sender: TObject);
begin
  if not DoCanDiscard or not dlgOpen.Execute then
    Exit;
  View.Diagram.LoadFromFile(dlgOpen.FileName);
  View.ZoomToFit;
  FFileName := dlgOpen.FileName;
  FModified := False;
  DoUpdateStatus;
end;


procedure TfrmDiagramFlowApp.btnSaveClick(Sender: TObject);
begin
  dlgSave.FileName := FFileName;
  if not dlgSave.Execute then
    Exit;
  View.Diagram.SaveToFile(dlgSave.FileName);
  FFileName := dlgSave.FileName;
  FModified := False;
  DoUpdateStatus;
end;
```

The export buttons are methods of the view, because an image needs the painter:

```
procedure TfrmDiagramFlowApp.btnPNGClick(Sender: TObject);
begin
  dlgExport.DefaultExt := 'png';
  dlgExport.Filter := 'PNG image (*.png)|*.png';
  if dlgExport.Execute then
    // the second parameter is the scale: 2 doubles the pixels
    View.SaveToPNG(dlgExport.FileName, 2);
end;


procedure TfrmDiagramFlowApp.btnSVGClick(Sender: TObject);
begin
  dlgExport.DefaultExt := 'svg';
  dlgExport.Filter := 'SVG image (*.svg)|*.svg';
  if dlgExport.Execute then
    View.SaveToSVG(dlgExport.FileName);
end;
```

The tool combo box maps its items to `TsgcDiagramTool` values through a constant array, and `DoUpdateStatus` reads the tool back, because the view returns to Select after some actions:

```
const
  CS_TOOLS: array [0 .. 2] of TsgcDiagramTool = (dgtSelect, dgtConnector,
    dgtPan);

// ...

procedure TfrmDiagramFlowApp.cboToolChange(Sender: TObject);
begin
  View.Tool := CS_TOOLS[cboTool.ItemIndex];
  View.SetFocus;
end;

// ... in DoUpdateStatus:
  // the Tool goes back to Select after some actions: show the real one
  case View.Tool of
    dgtConnector:
      cboTool.ItemIndex := 1;
    dgtPan:
      cboTool.ItemIndex := 2;
  else
    cboTool.ItemIndex := 0;
  end;
  lblStatus.Caption := Format('Nodes: %d    Connectors: %d    Selected: %d' +
    '    Zoom: %d%%', [oDoc.Nodes.Count, oDoc.Connectors.Count,
    oDoc.Selection.Count, Round(View.Zoom * 100)]);
```

How it works

1. The palette and the minimap find the view through their `View` property. The palette loads the shapes of the libraries named in `Libraries`; the minimap draws the whole document and the visible area.
2. A click on a palette shape sets the shape tool of the view, so the next click on the view creates that shape. A drag from the palette drops the shape where you release it, as one undo step.
3. Every edit fires `OnChange`. Geometry, style and structure changes set `FModified`; selection and view changes only refresh the status.
4. Save writes the document as JSON with `SaveToFile`, Open reads it back with `LoadFromFile` and fits it. New calls `Clear`, which also empties the undo history.

5. PNG and SVG export the whole diagram, whatever the zoom and the scroll position of the view.

> **Tip.** Save a diagram, open the `.json` file in a text editor and look at it: every node with its shape, position, text and style, every connector with the Ids of its nodes. It is a convenient format to generate or inspect from other tools.

Try this

- Add `network` to `Palette.Libraries`. The libraries of every edition are flowchart, basic, orgchart and network; bpmn, uml, pid and mindmap are part of the Registered edition.
- Add a Print button that calls `View.Print`, or `View.ShowPrintPreview` for a preview first.
- Add a theme switch that sets `View.Theme` and `Palette.Theme` together, so the palette matches the view.

5. Main components

The SGC Diagram palette page is short, and its three visual components cover a typical first application: the view, the palette and the minimap. This chapter introduces each of them, then the classes you meet in code, and finally the components you will want later.

Reference table

Class	Palette page	What it is for	Key members
`TsgcVCLDiagramView`	SGC Diagram	The VCL diagram view and editor	`Diagram`, `Tool`, `ReadOnly`, `Theme`, `ShowGrid`, `ShowRulers`, `Zoom`, `ZoomToFit`, `SaveToPNG`, `OnChange`
`TsgcVCLDiagramPalette`	SGC Diagram	Shapes to click or drag onto a view	`View`, `Libraries`, `ShowSearch`, `IconSize`, `ShowCaptions`, `Theme`
`TsgcVCLDiagramMinimap`	SGC Diagram	An overview of the whole diagram	`View`
`TsgcFMXDiagramView`, `TsgcFMXDiagramPalette`, `TsgcFMXDiagramMinimap`	SGC Diagram (FireMonkey)	The FireMonkey twins	The same as the VCL components
`TsgcDiagramDataBinding`	SGC Diagram	Builds a diagram from a dataset and follows its changes (Registered edition)	See its help topic
`TsgcDiagramStatusBinding`	SGC Diagram	Colours, badges and animates items from live values (Registered edition)	See its help topic
`TsgcDiagramDocument`	none, it is `View.Diagram`	Items, selection, styles, undo, files	`AddNode`, `AddConnector`, `Nodes`, `Connectors`, `Selection`, `BeginTransaction`, `SaveToFile`
`TsgcDiagramNode`	none	A box	`Shape`, `Left`, `Top`, `Width`, `Height`, `Text`, `Style`, `Id`, `Tag`
`TsgcDiagramConnector`	none	A line between two nodes	`Kind`, `SourceNodeId`, `TargetNodeId`, `StartArrow`, `EndArrow`, `Labels`, `Style`

The view

`TsgcVCLDiagramView` is the component every sgcDiagram form needs. It owns the document and the interaction: hit testing, selection, dragging, resizing, connecting, snapping, the in place editor, zoom and pan. The properties you will set first:

- **Look:** `ShowGrid`, `ShowRulers`, `ShowPorts` (`dgvNever`, `dgvHover`, `dgvAlways`), `Theme`.
- **Behaviour:** `Tool`, `ShapeTool`, `ReadOnly`, `SnapToGrid`, `SnapToGuides`.
- **Zoom:** `Zoom` (1 is 100%), `MinZoom`, `MaxZoom`, and the methods `ZoomToFit`, `ZoomToSelection` and `CenterOn`.

Its methods mirror the commands of an editor: `Undo`, `Redo`, `DeleteSelection`, `CopyToClipboard`, `CutToClipboard`, `PasteFromClipboard`, `EditText`, plus `SaveToPNG`, `SaveToSVG`, `Print`, `ShowPrintPreview` and `ExportToPDF`. Its events are listed in chapter 3.

The palette and the minimap

`TsgcVCLDiagramPalette` shows shapes grouped by library, with an optional search box. Link it to a view with its `View` property and list the libraries you want in `Libraries`, one name per line. A click on a shape sets the shape tool of the view, a drag drops the shape onto it, and the keyboard works too: the arrow keys move between the shapes and Enter or Space adds the focused one to the view.

`TsgcVCLDiagramMinimap` draws the whole diagram in a small control and marks the visible part of the view. Link it with `View` and place it wherever you like: beside the view, or over one of its corners as demo 05 does.

The document and its items

`TsgcDiagramDocument` is not on the palette: you reach it through `View.Diagram`, and you can expand it in the Object Inspector to set `Theme`, `GridSize`, `NodeStyle`, `ConnectorStyle` and `DefaultConnectorKind` at design time. In code it is where you add items, walk through them, group changes into undo steps and load and save files. The z order of a node is its index in `Nodes`.

Beyond the basics

Once the QuickStart demos feel familiar, these are the features to look at next. The help has a topic for each of them, and one topic per class with every property, method and event.

- **Ports and gluing:** named connection points on a node, so a connector starts or ends at a specific side.
- **Containers and groups:** nodes that hold other nodes, and groups that move together (`Group`, `Ungroup`).
- **Align and distribute, rulers and guides:** the commands of a drawing program, for neat diagrams.
- **Shape libraries:** the stencil libraries behind the palette, and your own.
- **Automatic layouts** (Registered edition): `TsgcDiagramLayeredLayout` for flowcharts and other directed graphs, `TsgcDiagramTreeLayout` for hierarchies such as org charts.
- **Data binding and live status** (Registered edition): a diagram built from a dataset, and items coloured from live process values.
- **Headless rendering:** `TsgcDiagramRenderer` writes PNG and SVG without a window, on Windows and Linux64.

6. Common problems

These are the problems that come up most often in a first diagram. Each entry says what you see, why it happens and how to fix it.

Building the diagram in code

The project does not compile: unit not found

What you see. `File not found: 'sgcDiagram_VCL.dcu'` or a similar message for another sgcDiagram or sgcUICore unit.

Why. sgcDiagram is not installed for this Delphi version, or the folder with its units is not on the library path.

Fix. Run the sgcUI installer again and select this Delphi version and sgcDiagram. If you build the packages yourself, add the sgcUI `Source` folder to **Tools, Options, Language, Delphi, Library** for each platform you compile for.

Undeclared identifier dgsRoundRect, TsgcDiagramNode or utDark

What you see. The compiler does not know a shape, a class or a theme value.

Why. The enumerations live in `sgcDiagram_Types`, the document and its items in `sgcDiagram_Model`, the themes in `sgcUICore_Theme` and the colour type in `sgcUICore_Canvas`. Dropping a view on a form adds only `sgcDiagram_VCL` to the `uses` clause.

Fix. Add the units you need, as in the `uses` clauses of demos 01 and 04.

Connectors have no route, or do not follow a moved node

What you see. After a bulk change in code, some connectors are missing or point to the old position.

Why. Routes are computed at the end of the outermost `BeginUpdate` and `EndUpdate` pair, and only while `Routing.AutoRoute` is `True`.

Fix. Make sure every `BeginUpdate` has its `EndUpdate`, in a `try` and `finally` block as in demo 02. If you switched `AutoRoute` off for a bulk change, call `View.Diagram.RouteAll` afterwards.

Orthogonal connectors cross the nodes

What you see. An elbow connector runs straight through a node instead of going around it.

Why. In a Community build, `dgcOrthogonal` connectors are simple elbows without obstacle avoidance. The router that steers around the nodes is part of the Registered edition.

Fix. Place the nodes so the elbows have a free path, use `dgcStraight` or `dgcCurved`, or use the Registered edition.

Editing, undo and the view

Undo does nothing after a change made by code

What you see. Your code added or changed items, and Ctrl+Z neither removes them nor undoes what the user did before.

Why. A change outside `BeginTransaction` and `EndTransaction` is not recorded, and it clears the undo and redo stacks.

Fix. Wrap code edits in a transaction, as the Add step button of demo 03 does. For the initial diagram, use `BeginUndoSuspend` and `EndUndoSuspend` on purpose, so the start diagram cannot be undone.

ZoomToFit shows the diagram too small or too large

What you see. The diagram does not fill the window at start, although you called `ZoomToFit`.

Why. In `OnCreate` the view still has its design size; the form has not been resized or maximised yet, so the fit is computed for the wrong size.

Fix. Call `View.ZoomToFit` in `OnShow`, as demo 05 does, or later.

The text editor does not open

What you see. A double click or F2 on a node does nothing.

Why. The view is `ReadOnly`, the node is `Locked`, or an `OnTextEditing` handler set `aAllow` to `False`.

Fix. Check those three. `OnItemDblClick` still fires in a read only view, so you can show the text another way.

A drag from the side of a node resizes it instead of drawing a connector

What you see. You wanted to draw a connector from a selected node, and the node changed size.

Why. The resize handles of a selected node cover its side ports.

Fix. Click empty space first, so the node is not selected, or set `Tool` to `dgtConnector`, as the Connect tool of demo 05 does.

The minimap disappears behind the view

What you see. A minimap placed over the view is hidden.

Why. The minimap is a sibling control on top of the view, and the control created later is drawn on top.

Fix. The minimap must come after the view in the form file. Right click it and choose **Control, Bring to Front**.

Paste does nothing after copying in another application

Why. The clipboard of sgcDiagram is internal to the application; it does not exchange data with other programs. Copy and paste work between diagrams of the same application.

Styles, files and editions

Changing NodeStyle does not change the existing nodes

What you see. You set `View.Diagram.NodeStyle.Fill` in code, and the nodes on the screen keep their colour.

Why. `NodeStyle` and `ConnectorStyle` are templates copied into each item when it is created. Each item draws from its own `Style`.

Fix. Set the defaults before you add the items, ideally in the Object Inspector as in demo 04, or loop over `Diagram.Nodes` and change each `Style`.

A colour set in the Object Inspector looks odd in the form file

Why. `TsgcUIColor` is an unsigned 32 bit value, `$AARRGGBB`. An opaque colour is larger than the largest Integer, so the form file writes it as a negative number. It still works, but it is hard to read.

Fix. Keep sizes, booleans and enumerations in the Object Inspector and set colours in code with readable hex constants, as demo 04 does.

ExportToPDF raises an exception

What you see. `View.ExportToPDF` raises `EsgcUIPDFExportUnavailable`.

Why. The PDF export goes through the sgcUIPDF bridge and needs a licensed sgcPDF. Printing and the print preview do not need it.

Fix. With sgcPDF installed, add the unit `sgcUIPDF_Export` to a `uses` clause of your project. Otherwise export PNG or SVG.

Layers, layouts or data binding are missing

What you see. `TsgcDiagramLayeredLayout` or `TsgcDiagramDataBinding` is undeclared, the palette has no BPMN or UML library, or `Layers.Add` raises `EsgcDiagramError`.

Why. These are advanced features, part of the Registered edition. In a Community build their units compile empty and the document allows one layer.

Fix. Use the Registered or Trial edition to work with them. A diagram file is the same in every edition, so a file saved with the advanced features still opens in a Community build.

7. Where next

Once the five QuickStart demos feel familiar, you know the core of sgcDiagram. This chapter points you to the larger demos, to the documentation and to support.

The other Delphi demos

The other folders in `Demos\Diagram\Delphi` are complete applications. Most of them have a `readme.html` that explains what they show and how to run them.

Demo	What it shows
`FlowchartEditor`	A flowchart editor for the VCL: palette and view, connector kinds, the layered layout, align and distribute, JSON, PNG and SVG, themes and a read only switch. The natural next step after demo 05.
`FlowchartEditorFMX`	The same flowchart editor for FireMonkey, on Windows and Android.
`DiagramStudioFMX`	A diagram studio for Windows and macOS with five stencil libraries, rulers, a minimap, live status colours, import and export.
`OrgChart`	An organisation chart bound to a dataset of employees, arranged with the tree layout and updated when a grid edit is posted.
`NodeEditor`	A node graph editor with typed ports: a type check in `OnConnecting` and blocks painted in `OnDrawNode`.
`BPMNEditor`	A BPMN 2.0 editor with pools and lanes, message flows, import from draw.io, Visio and GraphML, and PDF export.
`Importer`	Opens draw.io, Visio and GraphML files with `ImportFromFile` and reports what was imported.
`MindMap`	A mind map editor: Tab adds a child topic, Enter a sibling, and the map lays itself out.
`NetworkStatus`	A network diagram coloured live by a status binding, with values pushed from a timer and from a background thread.
`PIDViewer`	A piping and instrumentation diagram whose equipment is coloured, badged and blinks from simulated process values.
`Headless`	A console program with no VCL or FireMonkey unit that builds a flowchart, lays it out and writes PNG, SVG and JSON, on Windows and Linux64.

Several of these demos use features of the Registered edition, such as the layouts, data binding, live status, import or the extra shape libraries, and the PDF export needs sgcPDF. Their readme files and source comments say what they need.

Help topics

The sgcDiagram help is part of the sgcUI help at https://www.esegece.com/help/sgcUI, in the Diagram section. These topics are the best continuation of this guide:

Topic	What it covers
Introduction, What is sgcDiagram	What sgcDiagram does and where to start.
Quick Start VCL, Quick Start FMX, Headless	A first diagram on each framework, and without a window.
Editions, Platforms	What each edition includes, the Delphi versions, the platforms and the packages.
Shapes, Connectors and routing, Ports and gluing	The items in depth.
Undo and clipboard, Themes, Viewer mode	Transactions, the theme colours, and read only diagrams.
Serialisation, Export, Printing	The JSON format, PNG and SVG, and what a printed page contains.
Layered layout, Data binding	The advanced features of the Registered edition.
One topic per class	`TsgcVCLDiagramView`, `TsgcDiagramDocument`, `TsgcDiagramNode`, `TsgcDiagramConnector` and the others, with every property, its type and its default.

On the website

- https://www.esegece.com/products/sgcdiagram/: the product page, with the features, the editions and the downloads.

Getting help

If you are stuck, write to info@esegece.com or open a ticket in the helpdesk of your account at https://www.esegece.com/. It helps to include your Delphi version, your sgcDiagram version and edition, and the demo or a small project that shows the problem.

Enjoy drawing with code.

---

sgcDiagram and sgcUI are products of eSeGeCe. https://www.esegece.com