

LuciadLightspeed v2018.0.01

Developer's Guide



<http://www.luciad.com/>

Publication date: July 3, 2018

Copyright © 1999-2018 Luciad. All rights reserved.

Contents

I	Introduction	I
1.1	About LuciadLightspeed	1
1.1.1	LuciadLightspeed benefits	1
1.1.2	LuciadLightspeed modular components	3
1.1.3	LuciadLightspeed documentation.	3
1.2	About this guide.	4
1.2.1	Main sections	4
1.2.2	Conventions	5
1.3	Contact information	6
Part I	LuciadLightspeed Fundamentals	7
2	LuciadLightspeed architecture	9
2.1	The Model-View-Controller architecture.	9
2.2	LuciadLightspeed and the MVC architecture	10
2.2.1	LuciadLightspeed models.	10
2.2.2	LuciadLightspeed views	11
2.2.3	LuciadLightspeed controllers	13
2.3	Deciding between GXY view and Lightspeed view	13
2.3.1	Benefits of Lightspeed view.	14
2.3.2	Benefits of GXY view	14
2.3.3	Other benefits of hardware acceleration	15
2.4	Building applications with LuciadLightspeed	15
2.4.1	Defining the models	16
2.4.2	Defining the view	17
2.4.3	Defining the controllers	17
2.5	LuciadLightspeed coordinate reference systems	18
2.5.1	Model, world, and view coordinates	19
3	Building a basic application with a Lightspeed view	21
3.1	Creating a Lightspeed view	22
3.1.1	Setting up the Lightspeed view	22
3.1.2	Displaying the application in a standalone window	22
3.2	Adding data to the view	23
3.2.1	Creating a model from a file	23
3.2.2	Using layer factories to create layers	24
3.3	Using a controller to make the view interactive	26
3.3.1	What functionality is offered by the default controller?.	26
3.3.2	Adding a navigation controller.	27
3.4	Switching between 2D and 3D views	27
3.5	Changing the visual appearance of objects	29

4	Building a basic application with a GXY view	33
4.1	Defining the model	34
4.1.1	Loading and modeling the background data	34
4.2	Defining the view	35
4.2.1	Creating a view	35
4.2.2	Setting the world reference.	35
4.2.3	Creating a layer for the background data	36
4.2.4	Adding the background layer to the view.	37
4.2.5	Creating a simple layer control	37
4.3	Creating a controller	38
4.4	Displaying the map in a standalone window	38
5	Loading and visualizing business data	39
5.1	Modeling the business data	40
5.1.1	Modeling the domain objects	40
5.1.2	Determining the model reference	42
5.1.3	Determining the model descriptor	42
5.1.4	Choosing the model class	43
5.1.5	Creating a model decoder	43
5.2	Visualizing the business data in a Lightspeed view	45
5.2.1	Creating a layer for the business data	46
5.2.2	Fitting the view to the business data layer	47
5.2.3	Updating the basic application	48
5.3	Visualizing the business data in a GXY view	48
5.3.1	Creating a layer for the business data	49
5.3.2	Adding the business data layer to the view	50
5.3.3	Fitting the view to the business data layer	50
5.3.4	Updating the basic application	50
6	Working with layers	53
6.1	Accessing layers in a view	54
6.2	Grouping layers	54
6.2.1	Using <code>ILcdLayerTreeNode</code> to group layers	54
6.3	Working with layers in a tree structure	56
6.3.1	Approaching the view as a flat list of layers	56
6.3.2	Approaching the view as a layer tree	57
6.4	Layer ordering	58
6.4.1	How are layers ordered?	58
6.4.2	Moving layers up and down.	59
6.5	Selecting the objects in a layer.	60
7	Making objects selectable and editable	61
7.1	Making objects selectable and editable in a Lightspeed view	61
7.1.1	Adjusting a layer to make model data selectable and editable	61
7.1.2	Adding undo and redo support	61
7.1.3	Adding selection and editing support to the controller	62
7.1.4	Selecting and editing shapes in the application	62
7.2	Making objects selectable and editable in a GXY view	63
7.2.1	Adjusting a layer to make model data selectable and editable	63
7.2.2	Adding selection and editing support to the controller	63
7.2.3	Selecting and editing shapes in the application	63

8 Handling changes	65
8.1 Modifying application objects	65
8.1.1 Notifying objects of changes	65
8.2 Notifying objects of changes with listeners	66
8.2.1 Listening to changes in a selection	66
8.2.2 Listening to changes in a model	67
8.2.3 Listening to property changes	69
8.3 Notifying objects of changes programmatically	69
8.3.1 Changing domain objects	70
8.3.2 Changing indirect layer properties	70
9 Threading and locking for model and view access	73
9.1 Reading data in a LuciadLightspeed model	73
9.2 Changing data in a LuciadLightspeed model	74
9.3 Changing view and layer properties	74
9.4 Development pointers for threading and locking	74
Part II Advanced Modeling Topics	77
10 Unified access to domain objects	79
10.1 Introducing the framework	79
10.2 Getting started	80
10.2.1 Handling models	80
10.2.2 Handling domain objects	80
10.2.3 Browsing type information	82
10.2.4 Defining a data model	82
10.3 Explaining the meta model	83
10.3.1 Describing the static model	83
10.3.2 Creating data models	85
10.3.3 Comparing with ILcdFeatured	85
10.4 Advanced topics	86
10.4.1 Designing a data model	86
10.4.2 Traversing an object graph	87
10.4.3 Implementing ILcdDataObject	88
10.4.4 Adding custom metadata	89
10.4.5 Representing geometries	90
11 Modeling vector data	91
11.1 What is an ILcdShape?	91
11.1.1 Editable shapes	92
11.1.2 Commonly used shapes	94
11.2 What is an ILcdPoint?	96
11.2.1 Editable points	97
11.3 What is an ILcdBounds?	97
11.3.1 Editable bounding boxes	97
11.4 What is an ILcdShapeList?	98
11.5 What is an ILcdCurve?	98
11.6 Creating an ILcdPolyline or ILcdPolygon	100
11.7 Notifying objects of changes to shapes	101
12 Working with images	103
12.1 Using a model decoder	104

12.2	Image domain model	104
12.2.1	ALcdImage	104
12.2.2	ALcdBasicImage	105
12.2.3	Multi-level images and mosaics	106
12.3	Creating and applying image operators	108
12.3.1	Creating operators	108
12.3.2	Chaining operators	109
12.3.3	Available operators	109
12.3.4	Image processing operator examples	114
12.4	Image processing during visualization	115
12.4.1	Applying image operators in a Lightspeed view	115
12.4.2	Applying image operators in a GXY view	116
12.5	Offline processing	116
12.5.1	Accessing pixel values through ALcdImagingEngine	116
12.5.2	Encoding processed images to disk	117
13	Modeling raster data as ILcdRaster objects	119
13.1	What is an ILcdRaster?	119
13.2	What is an ILcdMultilevelRaster?	120
13.3	What is an ILcdTile?	120
13.4	ILcdRaster and ILcdTile sizes	120
14	Working with Earth tilesets	123
14.1	Earth tilesets	123
14.1.1	Earth tileset coverages	124
14.1.2	Retrieving tiles from a tileset	124
14.2	Using Earth tilesets to work with 3D terrain	125
14.2.1	Pre-processing to generate a 3D terrain	126
14.2.2	Visualizing the tile repository	130
15	Modeling multi-dimensional data	133
15.1	Getting the dimensions and values of multi-dimensional models	133
15.2	Filtering multi-dimensional data	134
15.2.1	Filtering vector models	134
15.2.2	Filtering raster models	134
15.2.3	Value filtering by interval matching	135
15.2.4	Locking your model for filtering	135
15.2.5	Snapping to filter intervals supported by the model	135
16	Decoding model references	137
16.1	How does a model decoder determine the model reference?	137
16.2	Providing model reference information	138
16.2.1	ILcdModelReferenceDecoder implementations	138
16.2.2	Providing reference information for raster data	138
16.3	Using an ILcdModelReferenceDecoder	141
16.4	Parsing references from text	141
16.5	Storing a model reference	141
17	Encoding model data	143
17.1	What is an ILcdModelEncoder?	143
17.2	Saving an ILcdModel using an ILcdModelEncoder	143
17.3	Making your own implementation of ILcdModelEncoder	144

18 Clustering	147
18.1 Using a clustered model	147
18.2 Configuring a TLcdClusteringTransformer	147
18.3 Clustering by class	148
18.4 Setting up scale-dependent clustering	149
18.5 Working with TLcdCluster objects	150
19 Grouping models hierarchically	151
19.1 Creating a model list	151
19.2 Creating a model node	152
19.3 Updating models.	153
Part III Advanced Lightspeed View and Controller Topics	155
20 Working with Lightspeed views	157
20.1 What is a view?	157
20.2 Fitting a view	158
20.3 Constraining navigation in a view.	158
20.4 Creating an off-screen view.	159
20.5 Adding balloons to the view	159
20.5.1 Creating balloons	160
20.5.2 Customizing balloons	161
20.6 Displaying a grid in your view	161
20.7 Retrieving world position from screen coordinates	162
20.8 Positioning a view programmatically.	164
20.8.1 Positioning a 2D view	164
20.8.2 Positioning a 3D view	165
20.8.3 Navigating on a Lightspeed view	166
20.9 Using animations	167
21 Working with layers in a Lightspeed view	169
21.1 Working with Lightspeed layers?.	169
21.1.1 Which layer to choose for your model	169
21.1.2 Building a layer	170
21.2 Managing selection between views	170
21.3 Filtering objects from a layer	171
21.4 Retrieving information about painted objects from a layer	171
21.5 Changing the painting orders of layers.	173
22 Visualizing domain objects in a Lightspeed view	175
22.1 Object painting in a Lightspeed view	175
22.1.1 Layers take care of object visualization	175
22.1.2 What if you want to change the look of your object?	176
22.2 Setting up your layers.	177
22.2.1 Quickly and easily creating vector layers: TLspShapeLayerBuilder . .	177
22.2.2 Customizing vector layers	177
22.2.3 Creating raster layers	177
22.2.4 Creating density layers	179
22.3 Using styles	180
22.3.1 Properties shared by style implementations	180
22.3.2 Setting visual style properties	181
22.3.3 Choosing an elevation mode.	181

22.4	Creating styles with builders	182
22.4.1	Using a builder to create new styles	182
22.4.2	Deriving a new style from an existing one	182
22.5	Defining vector styles	182
22.5.1	Visualizing simple lines with <code>TLspLineStyle</code>	182
22.5.2	Filling areas with colors and patterns with <code>TLspFillStyle</code>	183
22.5.3	Resolving overlaps for shapes with fill and line styles	183
22.5.4	Visualizing a position at ground level with <code>TLspVerticalLineStyle</code> . .	184
22.5.5	Painting point symbology with <code>TLspIconStyle</code>	185
22.5.6	Painting symbology along lines with <code>TLspComplexStrokedLineStyle</code> .	185
22.5.7	Painting 3D point symbology: <code>TLsp3DIconStyle</code>	188
22.5.8	Painting streets with their actual size with <code>TLspWorldSizedLineStyle</code>	189
22.5.9	Painting shapes in view coordinates	189
22.6	Styling objects with hardware-accelerated parameterized styles	190
22.6.1	When to use parameterized styles?	190
22.6.2	Creating parameterized styles	191
22.6.3	Using expressions for styling and filtering your vector data	191
22.7	Defining raster styles	193
22.7.1	Styling raster data with <code>TLspRasterStyle</code>	193
22.7.2	Styling the raster data bounds: <code>TLspLineStyle</code> and <code>TLspFillStyle</code> .	194
22.7.3	Color manipulation using <code>TLspColorLookupTableFilterStyle</code> . .	194
22.8	Defining density styles.	195
22.9	Linking objects to styles with <code>ILspStyler</code>	195
22.9.1	Creating a basic <code>ALspStyler</code>	196
22.9.2	Choosing <code>ILspStyler</code> implementations	197
22.9.3	Dynamically applying styles	198
22.9.4	Deriving geometry from objects	199
22.9.5	Improving the performance of your stylers	201
22.9.6	Layer styles	203
22.10	Visualizing 3D data in a Lightspeed view	203
22.10.1	Visualizing terrain elevation	203
22.10.2	Visualizing objects in 3D	204
22.11	Hardware-accelerated plot painting.	204
22.11.1	Configuring a plot layer	205
22.11.2	Styling and filtering your plot domain objects	206
22.11.3	Optimizing a plot layer's performance	209
22.12	Advanced visualization of domain objects.	210
22.12.1	Domain objects visualization in an <code>ILspPaintableLayer</code>	211
22.12.2	How does LuciadLightspeed display shapes in Lightspeed layers?	212
23	Labeling domain objects in a Lightspeed view	215
23.1	What is a label?	215
23.2	Activating labeling	216
23.3	Customizing the label content and look	216
23.3.1	Using text as labels	217
23.3.2	Using images as labels	218
23.3.3	Using Swing components as labels	219
23.3.4	Options for additional styling	219
23.3.5	Defining multiple labels for a domain object.	220
23.4	Positioning labels	220
23.4.1	Configuring relative label positions	221
23.4.2	Defining a different anchor point for labels	222

23.4.3 Defining labels with a dependency on other labels	223
23.4.4 Configuring labels along a polyline's path	224
23.5 Configuring label decluttering	225
23.5.1 Setting label priorities across layers	226
23.6 Interacting with labels	227
23.6.1 Moving labels	227
23.6.2 Creating interactive labels that users can edit	228
23.7 Using obstacle providers	229
23.8 Painting 3D labels	229
23.9 Advanced label placement	230
23.9.1 Using label placers to locate labels	231
23.9.2 Accessing label locations programmatically	232
24 Visualizing large vector data sets on a Lightspeed view	233
24.1 The role of the <code>TLcdModelQueryConfiguration</code>	233
24.2 Limiting data loading through the API	234
24.3 Limiting data loading using SLD	235
24.3.1 Introduction to SLD files	235
24.3.2 Limit data loading using SLD	235
24.4 Example: visualizing roads data	236
24.4.1 Using the <code>TLcdModelQueryConfiguration</code> API	236
24.4.2 Using SLD	238
24.5 Example: limiting visualization to a specific scale range	239
24.5.1 Using the <code>TLcdModelQueryConfiguration</code> API	239
24.5.2 Using SLD	240
24.6 Model query performance notes	241
24.7 Other ways to limit data	241
24.7.1 Minimum object size	241
24.7.2 Layer filter	242
24.8 Scales and filters in a 3D view	242
24.9 Visualizing the data on a GXY view	245
25 Creating and editing domain objects in a Lightspeed view	247
25.1 Working with editors and controllers	247
25.1.1 What is editing?	247
25.1.2 Making the objects in a layer editable	248
25.2 The editing process in a Lightspeed view	249
25.2.1 The editor	251
25.2.2 The edit controller	251
25.2.3 The handles	252
25.2.4 Working with handles	253
25.2.5 Snapping	254
25.3 The creation process in a Lightspeed view	255
25.4 Customizing the editing behavior	255
25.4.1 Visualization and styling of edit handles	256
25.4.2 Omitting edit handles or adding new ones	256
25.4.3 Editing custom domain objects	256
25.4.4 Implementing new editors	257
26 Managing your GUI and controllers in a Lightspeed view	265
26.1 Working with GUI events	265
26.1.1 What is an <code>ILcdAction</code> ?	265
26.1.2 Available implementations of <code>ILcdAction</code>	265

26.2	Using and customizing controllers	266
26.2.1	Chaining controllers	266
26.2.2	Customizing object selection with <code>TLspSelectController</code>	268
26.2.3	Customizing the selection logic with <code>TLspSelectControllerModel</code>	269
26.2.4	Custom selection examples.	270
26.2.5	Customizing object editing and object creation	270
26.2.6	Customizing ruler measurements with <code>TLspRulerController</code>	271
26.2.7	Visually comparing layers	272
26.2.8	Using touch controllers	273
26.3	Creating a custom controller	274
26.3.1	Implementing a chainable controller	274
26.3.2	Creating a custom touch controller.	274
26.3.3	Creating a tooltip controller	275
26.4	Handling non-standard input	276
26.5	Adding undo/redo support	276
26.5.1	Adding undo/redo capabilities to your application.	276
26.5.2	Making graphical edits undoable	276
27	Retrieving terrain data from a Lightspeed view	277
27.1	Querying terrain data	277
27.2	Querying tile-based terrain data	278
27.3	Determining a point on the terrain using intersection calculation	278
27.4	Draping 2D objects over the terrain	278
28	Adding graphical effects to a Lightspeed view	279
28.1	Using <code>ALspGraphicsEffect</code> and <code>TLspGraphicsEffects</code>	279
28.2	Adding lighting	279
28.3	Adding sky and atmosphere effects	280
28.4	Adding fog	280
29	Projecting images on a terrain in a Lightspeed view	281
29.1	Using <code>TLsplImageProjectionLayerBuilder</code> and <code>ILsplImageProjector</code>	282
Part IV	Advanced GXY View and Controller Topics	285
30	Working with GXY views	287
30.1	Adding a grid layer to the view	287
30.1.1	Choosing a predefined grid layer.	287
30.1.2	Using multilevel grids	288
30.2	Adding balloons to the view	289
30.3	Transformations between view and model coordinates	289
30.3.1	From view coordinates to world coordinates	289
30.3.2	From world coordinates to model coordinates	290
30.3.3	From model coordinates to view coordinates	290
30.4	Displaying the mouse cursor position	291
30.5	Positioning a view programmatically.	292
31	Working with layers in a GXY view	295
31.1	Copying layer nodes between layer trees.	295
31.2	Managing selection between views	295
31.3	Filtering objects from a layer	296

32 Painting and editing domain objects in a GXY view	297
32.1 Painting domain objects	297
32.1.1 Using an ILcdGXYPainter	298
32.1.2 Main implementations of ILcdGXYPainter	298
32.1.3 Using a TLcdGXYShapePainter	299
32.1.4 Using a TLcdGXYPointListPainter	299
32.1.5 Customizing an ILcdGXYPainter	299
32.1.6 Using an ILcdGXYPainterProvider	300
32.1.7 Main implementations of ILcdGXYPainterProvider	300
32.1.8 Using an ILcdGXYPainterProvider on a TLcdGXYLayer	300
32.1.9 Visualizing shapes based on domain object properties	300
32.1.10 Using an ILcdGXYPainterProvider on composite shapes	301
32.1.11 Using an array of ILcdGXYPainterProvider objects	301
32.1.12 Customizing an ILcdGXYPainterProvider.	302
32.1.13 Painting rasters	303
32.1.14 Main implementations of ILcdRasterPainter	304
32.1.15 Painting a 3D terrain in 2D from a tile repository.	306
32.2 Graphically editing domain objects	306
32.2.1 Using an ILcdGXYEditor.	306
32.2.2 Main implementations of ILcdGXYEditor.	306
32.2.3 Customizing an ILcdGXYEditor	307
32.2.4 Editing with multiple input points.	307
32.2.5 Using an ILcdGXYEditorProvider	307
32.3 Painting and editing support	307
32.3.1 Using an ILcdGXYPen.	307
32.3.2 Main implementations of ILcdGXYPen.	308
32.3.3 Setting an ILcdGXYPen to a layer	308
32.4 Painting symbology	309
32.4.1 Painting symbologies for points	309
32.5 Drawing halos around objects	310
32.6 Line and fill styles	311
32.7 Drawing complex strokes	312
32.7.1 Using complex strokes	312
32.7.2 Known issues and limitations	314
33 Asynchronous painting in a GXY view	315
33.1 Creating an asynchronous painting wrapper.	315
33.2 Accessing asynchronously painted layers	316
33.3 Editing the model of a wrapped layer	318
33.4 Memory and performance considerations	318
33.5 Customizing paint queue assignments	319
33.5.1 Observing paint queue assignments	319
33.5.2 Using paint hints.	319
33.5.3 Performing your own assignments	320
33.6 Providing transparent access to a custom layer interface	320
33.7 Troubleshooting	321
34 Visualizing large vector data sets on a GXY view	323
35 Implementing a painter/editor in a GXY view	325
35.1 Introducing a new shape: the hippodrome	325
35.1.1 IHippodrome, definition of a new shape	325

35.1.2	LonLatHippodrome and XYHippodrome	326
35.1.3	Representations of shapes	327
35.1.4	Creating an IHippodrome	327
35.1.5	Editing an IHippodrome	328
35.1.6	Snapping to/of an IHippodrome	328
35.2	The ILcdGXYPainter interface explained	328
35.2.1	ILcdGXYPainter modes	328
35.2.2	ILcdGXYPainter methods	329
35.3	The ILcdGXYEditor interface explained	331
35.3.1	ILcdGXYEditor modes	332
35.3.2	ILcdGXYEditor methods.	332
35.4	Common methods in ILcdGXYPainter and ILcdGXYEditor	332
35.5	Snapping explained	333
35.5.1	Snapping methods in ILcdGXYPainter	333
35.5.2	Snapping methods in ILcdGXYEditor	333
35.5.3	Use of snapping methods	333
35.5.4	What are the steps in the object snapping process?	333
35.5.5	Supporting multiple model references	335
35.6	Painting an IHippodrome.	335
35.6.1	Constructing an ILcdAWTPath with an ILcdGXYPen	335
35.6.2	An ILcdAWTPath for an IHippodrome	336
35.6.3	Improving performance: ILcdGeneralPath and ILcdCache	337
35.6.4	Rendering the ILcdAWTPath	339
35.6.5	Making sure the IHippodrome is visible in the view	339
35.7	Creating an IHippodrome	340
35.7.1	Creating a shape with TLcdGXYNewController2	340
35.7.2	Combining ILcdGXYPainter and ILcdGXYEditor	342
35.7.3	Interpreting the input position.	343
35.7.4	Adapting the view bounds	344
35.8	Editing an IHippodrome	345
35.8.1	Deciding on the editing modes	345
35.8.2	Editing a shape with TLcdGXYEditController2.	346
35.8.3	Checking if an object is touched	347
35.8.4	Implementing paint and edit for editing modes	350
35.8.5	Taking care of view bounds.	352
35.9	Snapping implemented	352
35.9.1	Specifications for snapping	352
35.9.2	snapTarget, snapping to an IHippodrome.	352
35.9.3	acceptSnapTarget, snapping of an IHippodrome	353
35.9.4	paint, highlighting the snap target.	355
35.9.5	edit, snapping to a snap target.	355
36	Labeling domain objects in a GXY view	359
36.1	A typical labeling scenario	359
36.2	Painting labels.	360
36.2.1	Choosing a label painter	360
36.2.2	Customizing labels using label stamps	361
36.2.3	Adding a halo to labels	362
36.3	Working with label locations	362
36.3.1	Using label placers	362
36.3.2	Using label algorithms.	363
36.3.3	Customizing label locations and label dependencies	363

36.4	Interacting with labels	365
36.4.1	Graphically editing labels	365
36.4.2	Working with interactive labels	365
36.5	Advanced labeling topics	368
36.5.1	The labeling process	368
36.5.2	Using label painters and label editors programmatically	368
36.5.3	Implementing your own label painter	369
36.5.4	Implementing your own label editor	370
36.5.5	Implementing your own placement algorithm	370
37	Managing your GUI and controllers in a GXY view	371
37.1	Defining GUI actions	371
37.1.1	What is an ILcdAction?	371
37.1.2	Main implementations of ILcdAction	371
37.1.3	Implementing an ILcdAction	372
37.2	Using and customizing controllers	373
37.2.1	Using and customizing a TLcdGXYSelectController2	373
37.2.2	Using and customizing a TLcdGXYEditController2	375
37.2.3	Using and customizing a TLcdGXYNewController2	376
37.2.4	Using touch controllers	376
37.2.5	Creating a custom touch controller	377
37.3	Handling non-standard input	377
37.3.1	Dispatching custom input events	377
37.3.2	Receiving LuciadLightspeed touch events	378
37.3.3	Creating and understanding touch events	378
37.4	Adding undo/redo support	379
37.4.1	Adding undo/redo capabilities to your application	379
37.4.2	Making graphical edits undoable	380
37.4.3	Describing the undo sample	380
38	Retrieving height data for 2D points	385
38.1	Using an ILcdHeightProvider	385
38.1.1	Standard implementations of ILcdHeightProvider	385
38.1.2	Typical usage of an ILcdHeightProvider	385
38.2	Creating an ILcdHeightProvider	386
38.2.1	Using properties	386
38.3	Using a TLcdViewHeightProvider	388
Part V	Common Topics for GXY and Lightspeed Views	389
39	Creating a vertical view	391
39.1	Modeling the profile	391
39.1.1	ILcdVVModel implementations	392
39.2	Adding the profile to the view	392
39.2.1	Configuring a TLcdVVJPanel	392
39.2.2	Creating a TLcdVVJPanel with controllers	393
39.2.3	Listening to cursor changes	393
39.3	Rendering the profile	393
39.3.1	Configuring a TLcdDefaultVVRenderer	394
39.3.2	Decorating the X-axis	394
39.3.3	Painting the grid	394
39.3.4	Using an ILcdVVGGridLineOrdinateProvider	395

39.4 A use case of the vertical view	395
39.4.1 Creating the flight model	396
40 Printing a view	399
40.1 Creating and configuring a <code>Printable</code>	399
40.2 Supporting printing for custom layers and painters in an <code>ILspView</code>	400
40.2.1 Tiled rendering	400
40.2.2 Feature scale	401
Part VI Combining GXY and Lightspeed Technology	403
41 Using Lightspeed technology in your GXY view application	405
41.1 Using Lightspeed technology within an <code>ILcdGXYView</code>	405
41.1.1 When would you use Lightspeed view technology?	405
41.1.2 Including a Lightspeed layer in a <code>ILcdGXYView</code>	406
41.1.3 Safely accessing asynchronously painted Lightspeed layers.	407
41.1.4 Deploying on a mix of hardware configurations	407
41.1.5 Performance tips for wrapped Lightspeed layers	407
41.2 Moving from a GXY view to an <code>ILspView</code>	408
41.2.1 Converting a GXY view to a Lightspeed view	408
41.2.2 Converting a GXY layer to a Lightspeed layer	408
41.2.3 Adapting GXY layers	409
Part VII Referencing Geographic Data	413
42 Geodesy	415
42.1 What is a geodetic datum?	415
42.2 Using an <code>ILcdGeodeticDatum</code>	416
42.3 Using an <code>ILcdEllipsoid</code>	419
42.4 Literature on geodesy.	419
43 Projections	421
43.1 Using an <code>ILcdProjection</code>	421
43.2 Extensions of <code>ILcdProjection</code>	422
43.3 Main implementations of <code>ILcdProjection</code>	424
43.4 Literature on map projections.	425
44 Coordinate systems and transformations	427
44.1 Model, world, and view coordinate systems.	427
44.2 LuciadLightspeed reference systems	429
44.3 Defining your own reference system	432
44.4 Transformations between coordinate systems	433
45 Geometric calculations	435
45.1 Ellipsoidal calculations.	435
45.2 Spherical calculations	435
45.3 Cartesian calculations	436
45.4 Sampling line segments	436
46 Rectification	437
46.1 Non-parametric rectification	437
46.1.1 Transformations in a non-rectified grid reference.	437
46.1.2 Transformations in a rectified grid reference	438
46.1.3 Interfaces and classes used for rectification	439

46.1.4	Limitations	442
46.2	Parametric rectification (orthorectification)	442
46.2.1	Parameters related to the imaging sensor	442
46.2.2	Including terrain information	443
46.2.3	Limitations	444
46.3	Combining parametric and non-parametric rectification	444
Part VIII	Data Formats and Standards	447
47	Adding instant data format support to your application	449
47.1	Selecting data format services	449
47.2	Looking up data format services	450
47.3	Configuring an action to load model data.	451
47.4	Using other format services	452
47.5	How to plug in your own data format	452
48	Working with ISO metadata	455
48.1	What is metadata?	455
48.2	ISO 19115: metadata concepts	456
48.2.1	Standard specifications	456
48.2.2	Domain model	457
48.2.3	Validation	458
48.2.4	Visualization	459
48.3	ISO 19139: an XML implementation	459
48.3.1	Introduction	459
48.3.2	Decoding metadata.	460
48.3.3	Custom metadata extensions	460
49	GML format	463
49.1	Introduction to GML	463
49.2	Integrating GML data into your application	464
49.3	The GML data model	465
49.4	The LuciadLightspeed GML domain model	466
49.4.1	Domain model classes	466
49.4.2	Feature collection integration	467
49.4.3	Feature integration	467
49.4.4	Geometry integration	468
49.4.5	Other integrations	468
49.5	GML model decoders.	468
49.5.1	XML integration	469
49.6	GML model encoders.	469
49.7	Limitations	470
50	Using OGC Filters	471
50.1	Introduction to OGC filters	471
50.2	The OGC Filter API in LuciadLightspeed	471
50.3	Quick start.	472
50.3.1	Creating or decoding a filter	472
50.3.2	Evaluating a filter	473
50.3.3	Supporting custom OGC functions	473
50.3.4	Encoding a filter	475
50.4	The OGC Filter model	475
50.4.1	Overview	475

50.4.2 Filter expressions	476
50.4.3 Filter conditions	477
50.5 Limitations	481
51 Styling data with OGC SLD	483
51.1 Using SLD styling and symbology encoding in LuciadLightspeed.	483
51.2 Quick start: styling your layers with SLD.	484
51.2.1 Decoding an SLD file	484
51.2.2 Creating a SLD-styled Lightspeed layer	484
51.2.3 Updating the SLD style on a Lightspeed layer	485
51.2.4 Creating an SLD-styled GXY layer	485
51.2.5 Updating the SLD style on a GXY layer	485
51.3 The SLD styling model	486
51.3.1 Why provide a model implementation?	486
51.3.2 Model structure	486
51.3.3 Using the content	487
51.3.4 Model implementation	488
51.4 Decoding and encoding SLD styling models	489
51.5 SLD styling limitations.	489
51.5.1 Unsupported SLD elements	489
51.5.2 Limitations on external graphics	490
51.5.3 Other SLD styling limitations	490
52 Working with XML data	491
52.1 Introduction to XML	491
52.1.1 Representing schema information	492
52.2 Decoding XML documents	492
52.3 Creating a data model based on an XML schema	492
52.3.1 Mapping XML types on Java classes	493
52.3.2 Custom domain classes	494
52.4 Custom XML decoding	495
52.4.1 Custom simple types	495
52.4.2 Custom complex types	496
52.4.3 Advanced type unmarshalling	498
52.5 Advanced features	501
52.5.1 Exporting classes using the TLcdXMLJavaClassResolver	501
52.5.2 Runtime schema extension	502
Part IX Advanced Development Guidelines	503
53 General performance guidelines	505
53.1 Models	505
53.1.1 TLcd2DRegularTiledBoundsIndexedModel	505
53.1.2 TLcdRegularTiled2DBoundsIndexedModel	505
53.2 Querying a model	506
53.2.1 In-memory models	506
53.2.2 Databases	506
53.2.3 WFS servers	506
53.2.4 SHP models	506
53.3 Raster Painting	507
53.3.1 Using tiled rasters	507
53.3.2 Using multilevel rasters	507

53.3.3 Tuning DMED/DTED fall-back.	507
53.3.4 Combining rasters	508
54 Performance guidelines for Lightspeed views	509
54.1 Tuning view performance optimizations	509
54.2 Tuning shape and raster painter settings	509
54.3 LuciadLightspeed configuration options	510
54.3.1 OpenGL resource cache.	510
54.3.2 Draping on terrain	511
54.3.3 Transparency.	511
54.4 External configuration options.	512
54.4.1 Memory guidelines	512
54.4.2 Garbage collection	513
54.4.3 Video driver	513
54.5 Monitoring the allocation of video memory	513
55 Performance guidelines for GXY views	517
55.1 Background layers	517
55.2 Smart panning.	517
55.3 Preventing raster warping	517
55.4 Tuning raster painter settings	517
56 Advanced threading and locking	519
56.1 Offloading the EDT.	519
56.1.1 Updating a model off the EDT.	519
56.1.2 Preparing model updates off the EDT	520
56.1.3 Asynchronous layer painting in GXY views	520
56.2 Supporting off-screen views	521
57 Logging and performance monitoring	523
57.1 Logging	523
57.1.1 Producing log messages	523
57.1.2 Handling log messages with the standard Java logging framework	524
57.1.3 Handling log messages with other logging frameworks	525
57.2 Performance monitoring using JConsole	525
57.3 Performance monitoring using Java VisualVM	527
58 Configuring classes with properties files	529
58.1 Creating and initializing objects	529
58.2 Serialization and the role of aPrefix	530
58.3 Creating Properties objects.	531
58.4 Using a Properties file.	531
58.4.1 Sample properties file.	531
59 Running in OSGi™	533
59.1 Creating bundles with the <code>make.osgi_bundles</code> script	533
59.2 Running the OSGi sample	533
59.2.1 Running the sample from the command prompt	534
59.2.2 Running the sample from Eclipse	534
59.3 Created bundles are fragments, use Require-Bundle and rely on boot delegation	534
60 Integrating a Lightspeed view in C/C++/C# applications	535

Part X Appendices	537
A Upgrading LuciadLightspeed	539
A.I Compatibility of LuciadLightspeed versions	539
A.I.1 What is LuciadLightspeed binary compatibility?	539
A.I.2 Benefits of binary compatibility	540
A.2 Project upgrading guidelines	540
A.2.1 Read the release notes	540
A.2.2 Document your sample code usage	540
A.2.3 Upgrading dependencies	541
B Troubleshooting for Lightspeed views	543
B.I Graphics drivers	543
B.2 Known issues	543
C OpenGL	545
C.I What is OpenGL?	545
C.2 JOGL2 OpenGL binding supported by LuciadLightspeed	545
C.3 Additional information on OpenGL	546
C.4 Generating a report of the hardware and OpenGL specifications of your development platform	546
C.5 Checking OpenGL component compatibility with your target platform	547
C.6 Debugging OpenGL errors	547
D Using LuciadLightspeed in a Maven repository	549
D.I Deploying to a repository	549
D.2 Using the libraries	549
D.2.1 Adding a license	550
E Supported data formats	551
E.I Model decoders and encoders	563
F Cookbook	565
F.I Lightspeed view programs	565
F.I.1 Building a basic LuciadLightspeed application with a Lightspeed view	565
F.I.2 Loading and visualizing business data in a Lightspeed view	569
F.I.3 Grouping Lightspeed layers in a layer tree node	570
F.I.4 Adding editing capabilities to a Lightspeed view application	572
F.2 GXY view programs	573
F.2.1 Building a basic LuciadLightspeed application with a GXY view	573
F.2.2 Loading and visualizing business data in a GXY view	575
F.2.3 Grouping layers hierarchically in a GXY view	576
F.2.4 Adding editing capabilities to a GXY view application	577
G Acronyms and abbreviations	579

CHAPTER I

Introduction

1.1 About LuciadLightspeed

LuciadLightspeed is a set of software components designed to build applications for the visualization and manipulation of geospatial data. LuciadLightspeed provides a comprehensive set of interfaces and classes that support various geographic data formats, coordinate transformations, map projections, [geodesy](#) operations, geodetic shapes, rasters, and other functionality to handle geospatial data. LuciadLightspeed provides application developers with a set of components that they can use out of the box or customize to their own needs. LuciadLightspeed is designed to leverage the power of the computer hardware to obtain exceptional visualization performance.

1.1.1 LuciadLightspeed benefits

LuciadLightspeed enables developers to build a system that can handle geospatial data and that matches their needs and requirements without any constraints. Working with LuciadLightspeed provides many benefits for you as a developer, because it is:

Object oriented

LuciadLightspeed was built using the following object-oriented rules and patterns:

- **The Model-View-Controller architecture.** LuciadLightspeed separates the model, view, and controller components of the application. This results in a higher flexibility and reusability of code as described in [Section 2.1](#).
- **The use of interfaces for basic concepts.** LuciadLightspeed provides many implementations that are readily available. In addition, the basic LuciadLightspeed interfaces enable the creation and usage of customized implementations.
- **The event mechanism for dynamic behavior.** LuciadLightspeed propagates changes in model data with events. This facilitates working with dynamic data, enhances the separation of the components, and increases code flexibility.

Open

The LuciadLightspeed components are fully customizable. LuciadLightspeed provides the components to build a system for visualizing and handling geographical data without any restrictions on:

- **The data source and format.** The LuciadLightspeed models do not depend on any data source or format. You can extend the LuciadLightspeed interfaces to model any specific

data format from any source.

- **The user interface.** Although LuciadLightspeed provides several implementations for Swing-based GUIs, you can also use all LuciadLightspeed functionality for any GUI that is not based on Swing.
- **The platform or architecture.** The MVC architecture allows you to build an application and deploy it on any platform ranging from embedded systems over ruggedized portable devices to workstation/server configurations and integrate it with application frameworks such as Eclipse RCP and the NetBeans platform.

Modular and scalable

The modular setup of LuciadLightspeed allows you to write a basic application, extend it to your needs, and add specialized functionality to it by using one or more of the LuciadLightspeed components.

Complete

LuciadLightspeed provides a complete set of implementations for:

- Developing **situational awareness** applications and **high-performance visualization** software in **aviation and defense**. LuciadLightspeed supports all commonly used military and aeronautical standards to build these types of applications.
- Visualizing data in **2D and 3D**. All LuciadLightspeed interfaces and classes for modeling data are suited for both two-dimensional and three-dimensional visualization. As a result of the MVC approach, both types of visualization can be achieved without any changes to the model.

Fast

LuciadLightspeed-based applications offer a high performance. To achieve such a high performance, LuciadLightspeed uses:

- **GPU leveraging.** LuciadLightspeed uses the [OpenGL API](#) for 2D and 3D rendering in a hardware-accelerated view. This allows LuciadLightspeed to leverage the massively parallel processing power of Graphics Processing Units (GPU). To this end, LuciadLightspeed is designed to operate with the smallest possible CPU overhead.
- **Asynchronous processing.** Expensive computations that cannot be offloaded to the GPU are performed in background threads. This ensures that LuciadLightspeed-based applications can remain responsive at all times and allows them to exploit multi-core CPUs.
- **Bulk operations.** To reduce low-level overhead from recurring actions, such as repeated method calls, LuciadLightspeed operates on batches of data rather than individual data items wherever possible. For instance, when rendering a large number of placemark icons on the screen, LuciadLightspeed collects the coordinates of these placemarks, and sends them to the GPU in a single batch, rather than drawing them one by one with separate rendering calls. This improves concurrency between the CPU and GPU.
- **Lazy loading of data.** LuciadLightspeed uses [lazy loading](#) techniques for a fast rendering of data regardless of its size and format. You can use LuciadLightspeed to write applications that retrieve large data sets without preprocessing, and visualize the data on the fly.
- **Caching.** LuciadLightspeed uses caching for many operations, particularly to optimize the speed for visualizing data. Caching is, for example, used for performing repeated geographical transformations.

- **Fast spatial queries.** LuciadLightspeed uses spatial indexes to efficiently perform spatial queries.

Sustainable

The regular LuciadLightspeed releases provide **up-to-date functionality** with a strong focus on **backwards compatibility**.

Compatible with LuciadMap

LuciadLightspeed is compatible with LuciadMap. As such, any LuciadMap-based application remains functional in LuciadLightspeed. The Lightspeed view components are designed to allow LuciadMap-based applications to very quickly benefit from the performance of the Lightspeed technology.

I.I.2 LuciadLightspeed modular components

LuciadLightspeed consists of a number of included and optional LuciadLightspeed components that each add functionality. This guide only describes the essential functionality provided by the main LuciadLightspeed components. Supplementary developer and user documentation is available for additional LuciadLightspeed component functionality.

An overview of LuciadLightspeed components is made available with the LuciadLightspeed distribution. For an outline of all other available LuciadLightspeed documentation, refer to [Section 1.1.3](#).



You need the appropriate licenses for the installation and use of LuciadLightspeed components.

I.I.3 LuciadLightspeed documentation

The following LuciadLightspeed documentation pages and code samples are available through the LuciadLightspeed launcher (`start.jar`) of the LuciadLightspeed distribution:

- **Getting Started.** The Getting Started page provides guidelines for the installation of your license, and for integrating LuciadLightspeed samples into your Integrated Development Environment (IDE).
- **Components.** The Components page provides a description of each component, and direct links to developer's guides, sample pages, release notes, and prerequisites. The developer's guides provide a high-level overview of component functionality and describe how to build applications using their API. Most of the descriptions are illustrated with graphics and code snippets.
- **Samples.** For LuciadLightspeed and for most of the LuciadLightspeed components. The code samples are small applications that illustrate the functionality of (parts of) LuciadLightspeed or a component. The Samples page provides a small screenshot and description of each code sample. You can run the samples from the Samples page.
- **API reference.** For LuciadLightspeed and all components. The API reference provides detailed descriptions of each package and its interfaces and classes. Some descriptions are illustrated with graphics and code snippets.
- **Release notes.** For LuciadLightspeed and all components. The Release Notes list the enhancements, bug fixes, and upgrade considerations for the current and previous release versions of LuciadLightspeed and the LuciadLightspeed components.

- **Technical notes.** For LuciadLightspeed and all LuciadLightspeed components. The Technical Notes provide information about supported platforms, compatibility, details on obfuscation of source code, and instructions for using hardware keys.
- **Prerequisites** For LuciadLightspeed and all components. The Prerequisites page lists license information for third-party libraries and components, additional requirements, included LuciadLightspeed JAR files, dependencies, platform requirements, and used libraries.



Note that the developer's guides, samples, and release notes for a component are only available after installing the component.

In addition to the documentation provided through the LuciadLightspeed start page, you can find a wide variety of sample data in the directory `LuciadLightspeed_<x.x>\samples\resources\data`.

I.2 About this guide

The purpose of this guide is to introduce LuciadLightspeed, and provide guidelines to developers for building an application with these components. This guide is for developers who are familiar with programming in Java and who preferably have some knowledge of design patterns. References to general Java concepts and design patterns are used throughout this guide without further explanation. Although no specific knowledge of working with geographical data is required, it is useful that the developers have a good understanding of the requirements of the application that they want to build with LuciadLightspeed. In addition, basic knowledge of OpenGL programming principles and practice will help developers who wish to expand the LuciadLightspeed API.

This guide describes the tasks that are required for building an application with LuciadLightspeed. This guide does not list all interfaces and classes of LuciadLightspeed and it does not contain detailed package and class descriptions. For a complete overview of the LuciadLightspeed API and additional information on the usage of interfaces and classes, refer to the API reference. For an overview of all LuciadLightspeed documentation, refer to [Section 1.1.3](#).

The functionality discussed in this guide is available in all LuciadLightspeed product tiers.

I.2.1 Main sections

The main sections in this guide are:

- **Part I: LuciadLightspeed Fundamentals** introduces LuciadLightspeed, and describes how to build a basic application with LuciadLightspeed. Read this part if you are new to LuciadLightspeed. It explains all the basics and gets you started immediately. Once you are familiar with the LuciadLightspeed fundamentals, you can easily use other parts of the API and the LuciadLightspeed components. The other parts of this guide cover topics that are not considered as fundamental but that allow you to extend a basic application to meet specific needs and requirements.
- **Part II: Advanced Modeling Topics** describes the options that LuciadLightspeed provides for modeling data in addition to the basic functionality as described in [Part I](#).
- **Part III: Advanced Lightspeed View and Controller Topics** describes the options that LuciadLightspeed provides for Lightspeed view implementations and controllers in addition to the basic functionality as described in [Part I](#).

- **Part IV: Advanced GXY View and Controller Topics** describes the options that LuciadLightspeed provides for GXY view implementations and controllers in addition to the basic functionality as described in [Part I](#).
- **part V: Combining GXY and Lightspeed Technology** describes how you can use Lightspeed technology in an existing application with a GXY view.
- **part VI: Creating a vertical view** describes how you can create a vertical view to visualize the third dimension of a list of points.
- **Part VII: Referencing Geographic Data** describes the options that LuciadLightspeed provides for working with geospatial data. It covers the usage of geodetic datums, projections, coordinate systems, transformations and more specialized topics such as geometric calculations and rectification.
- **Part VIII: Data Formats and Standards** describes the options that LuciadLightspeed provides for working with specific data formats and standards such as ISO metadata, OGC filters, and XML.
- **Part IX: Advanced Development Guidelines** provides guidelines for performance, logging, and the usage of properties files.

1.2.2 Conventions

This section describes the typographical conventions that are used in this guide and the naming conventions that are used in LuciadLightspeed and in this guide.

Typographical conventions

This guide uses the following typographical conventions:

- **Bold** text is used to highlight a term. Note that when a term is used multiple times in a paragraph, only the first occurrence of the term is bold.
- *Italic* text is used to quote text or to refer to a document or sample.
- Reference text is used for internal references to sections, figures, tables, or programs. For example in: [Program 1](#) shows how to instantiate a view.
- `Typewriter` text is used for API elements such as names of interfaces, classes, and methods, and for file names and extensions.



Information paragraphs contain important information or useful tips.

For reasons of text alignment and readability, the names of Java packages, Java classes, URLs and directory paths may be split over multiple lines. In these cases, the names, URLs and paths contain hyphens that are not part of the name or path. You can still open hyphenated hyperlinks successfully by clicking the link, but if you copy and paste URLs and directory paths to the browser's address bar, the hyphens may come along. If you are copy-pasting, remove the superfluous hyphens in the browser to make the link work.

Naming conventions

LuciadLightspeed uses the following naming conventions:

- Class names starting with:
 - I are interfaces

- **A** are abstract classes
 - **T** are concrete classes
 - **E** are enumerations
- The class type identifier in a class name is followed by three letters: **Lcd** or **Lsp**, marking Luciad and specific **Lightspeed view** classes respectively.
 - Names outside LuciadLightspeed (for example names of sample classes) never start with the abovementioned prefixes.
 - **SFCT** is the Luciad abbreviation for side effect. It is added as a postfix to method names and parameter names, and indicates that you are dealing with a side-effect method or parameter. The side-effect parameter of a side-effect method changes as a result of the method call.
 - Names of LuciadLightspeed Java packages have the following format: `com.luciad.*`, for example `com.luciad.view.lightspeed`.
 - The terms *LuciadLightspeed* and *LuciadLightspeed components* refer to the distribution files. The term *LuciadLightspeed Java package* refers to the Java packages that are part of the LuciadLightspeed API.

This guide refers to interfaces and classes with their complete name. To enhance readability however, the following shorthand references are used:

- The word *type* refers to interfaces or their extensions. For example, *the type ILcdModel* refers to the interface `ILcdModel` or extensions of it.
- The name of the class refers to an instance of that class. For example, *a TLcdVectorModel* means an instance of the class `TLcdVectorModel`. In plural this guide uses: *TLcdVector-Model objects*.
- The name of the interface refers to an instance of a class as an implementation of that interface. For example, *an ILspView* means an instance of a class that implements the interface `ILspView`. In plural this guide uses: *ILspView objects*.
- To refer to a class or interface itself, the class or interface is explicitly mentioned. For example, *the class TLspLayer implements the interface ILspLayer*.

I.3 Contact information

If you have any questions after reading this guide you can contact the Luciad Support Desk services by mailing to support.luciad.gsp@hexagon.com or dialing +32 (0)16 26 28 30.

PART I LuciadLightspeed Fundamentals

CHAPTER 2

LuciadLightspeed architecture

2.1 The Model-View-Controller architecture

The LuciadLightspeed API is based on the Model-View-Controller (MVC) architecture. The underlying concept of the MVC architecture is to separate the data (model), representation of the data (view), and the user interaction (controller) from each other. This separation results in a simpler design of the application and a higher flexibility and reusability of code.

The MVC parts of the LuciadLightspeed API are defined as follows:

- A LuciadLightspeed `model` stores and describes geographical data regardless of how to visualize and interact with the data. For example: a model contains the location of a number of hospitals together with additional information such as capacity.
- A LuciadLightspeed `view` contains all information for the representation of data contained in LuciadLightspeed models. A view does not contain data itself. For example: in case of the hospitals, a view represents the location of a hospital with a red cross.
- A LuciadLightspeed `controller` interprets user interaction and performs the required action on LuciadLightspeed models and views regardless of the type of model and view. For example: in case of the hospitals, right-clicking on a red cross pops up information such as location and capacity.

Separating the different parts of the application allows you to reuse objects for different purposes and redefine objects without changing other objects. You can, for example, change a view without making changes to the models represented in the view. Or you can redefine user interaction on a view without changing the view. Object reuse shortens the time for writing an application. In addition, it enhances a consistent functionality and design of all your applications.



Using the MVC architecture enables you to focus on the definition of the objects instead of solving design issues.

Figure 1 depicts the MVC architecture of the LuciadLightspeed API. Section 2.2 describes the MVC parts of the LuciadLightspeed API in more detail.

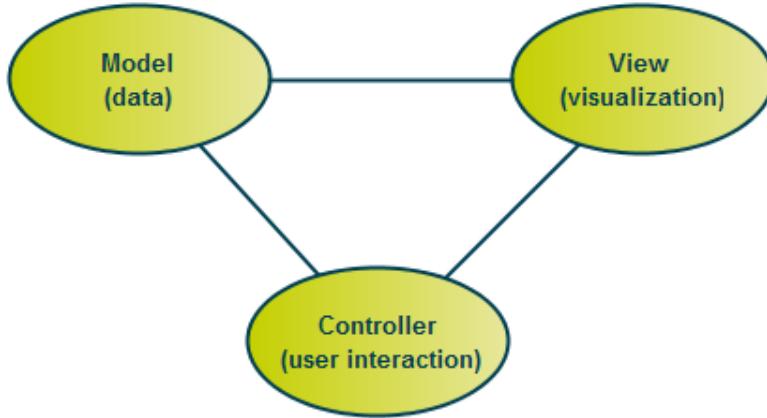


Figure 1 - The MVC parts of the LuciadLightspeed API

2.2 LuciadLightspeed and the MVC architecture

As described in [Section 2.1](#), the LuciadLightspeed API is based on the MVC architecture and separates the functionality for the model, view, and controller parts as shown in Figure 1. The LuciadLightspeed API strictly separates the functionality for:

- modeling data (model)
- visualizing data (view)
- interacting with the view or with the data (controller)

The sections below provide detailed descriptions of LuciadLightspeed models, views, and controllers, as well as instructions on how to build these components.

2.2.1 LuciadLightspeed models

A LuciadLightspeed model (`ILcdModel`) is a container for a set of geographical data of the same type, for example cities, roads, elevation data, flight plans, and so on. There are two types of geographical data:

- **Vector data** that consists of **geometries** such as points, lines, circles, and polygons. Examples of vector data are streets, buildings, airports, radars, and so on.
- **Raster data** that consists of a **grid of cells**. Examples of raster data are aerial photographs, satellite images, weather data, elevation data, and so on.

To define the model data, or domain objects, in a uniform way, each `ILcdModel` is associated with:

- A **model reference** (`ILcdModelReference`) that specifies the coordinate reference system that is used for locating the model data on earth. All LuciadLightspeed models need to be associated with a reference system. Otherwise, it is not possible to determine the location of the data. [Section 2.5](#) provides more information on the type of reference systems that are used as model reference.
- A **model descriptor** (`ILcdModelDescriptor`) that provides additional information, or metadata, about the model data. The metadata usually specifies the source of the data, the data type, and a name to refer to the data in the view (the display name). Depending on

the type of data, the model descriptor may include information about the content and/or the structure of the data or make metadata accessible that is stored with the data.

- A model metadata (`TLcdModelMetadata`) object that provides advanced metadata information about the model data. This object supports ISO metadata, and can be obtained without decoding the model itself. For more information, see [Section 2.4.1](#) and [Chapter 48](#).

Figure 2 illustrates a LuciadLightspeed model and its associated components.

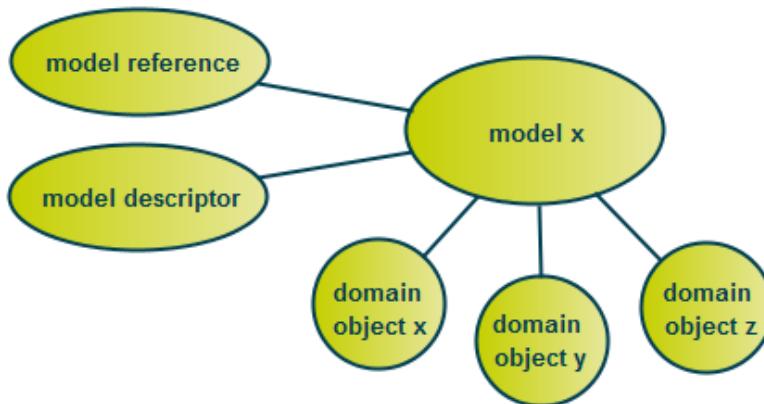


Figure 2 - A LuciadLightspeed model

The model data can be used in the application as:

- **background data:** the data is used as **reference** for the data that you want to interact with (business data). Background data, such as terrain or elevation data, is typically loaded from an external source.
- **business data:** the data that you want **to interact with**. Business data is usually visualized together with background data and can be loaded from an external source or created by the application itself.

2.2.2 LuciadLightspeed views

A LuciadLightspeed view allows you to visualize the data of one or more LuciadLightspeed models in a graphical form. LuciadLightspeed offers two types of views:

GXY views are views that display graphics in 2D, also known as [classical 2D view](#). The abbreviation GXY refers to the graphical 2D screen coordinates. A GXY view is represented by `ILcdGXYView`.

Lightspeed views are views that can display in 2D as well as in 3D, and use the OpenGL API to perform their rendering. A Lightspeed view is represented by `ILspView`.

Both view types are subtypes of the interface `ILcdView`.

Associated with a view are:

- **A world reference:** To represent geographical data in a view, all data is transformed on the fly into a common coordinate system. This reference system is called a world reference. For 2D views, it is typically defined by a map projection to map the curved surface of the earth to the flat surface of the view. For 3D views, it is typically a geocentric coordinate system. [Section 2.5](#) provides more information about the types of reference systems that are used as world references.

- One or more **layers**: To properly visualize the model data in a LuciadLightspeed view, each model is placed in a LuciadLightspeed **layer**. These layers are associated with painters and editors:
 - A **painter** takes care of displaying the model data in a view. The type of painter depends on the type of model data. LuciadLightspeed provides painters and layers for the most commonly used vector and raster data.
 - An **editor** that takes care of editing the data in a view.
 - A set of **properties** that indicate, for example, whether the layer data is visible, selectable, and editable in a view.

Figure 3 illustrates a LuciadLightspeed view and its associated components.

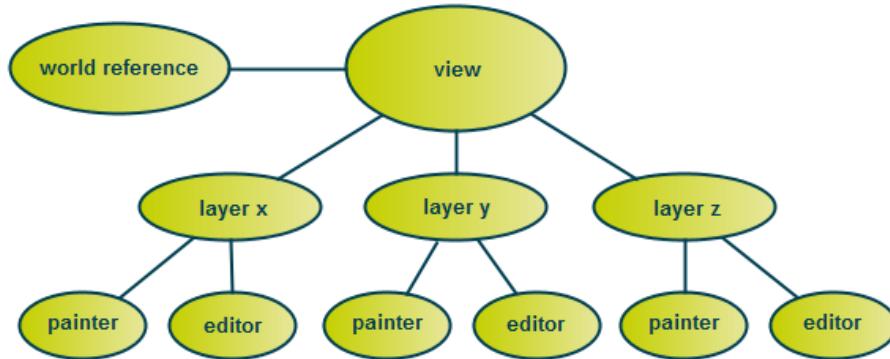


Figure 3 - A LuciadLightspeed view



Your choice of view type determines the layer type you need to use. To create layers in a GXY view, you need to use GXY layers. To create layers in a Lightspeed view, you need to use Lightspeed layers. However, adapters are available that enable you to combine GXY and Lightspeed technology.

For more information about the differences between GXY and Lightspeed views, refer to Section 2.3.

LuciadLightspeed provides implementations based on the Java Abstract Windows Toolkit (AWT) and the Swing toolkit, as well as implementations to perform off-screen rendering. Section 2.4.2 describes the main GUI implementations of a view. For more information, refer to the API reference.

At the API level, the `ILcdGXYView` and `ILspView` are both subtypes of the interface `ILcdView`. `ILspView` resides in the package `com.luciad.view.lightspeed`, while `ILcdGXYView` resides in the package `com.luciad.view.gxy`. These are subpackages of package `com.luciad.view`.

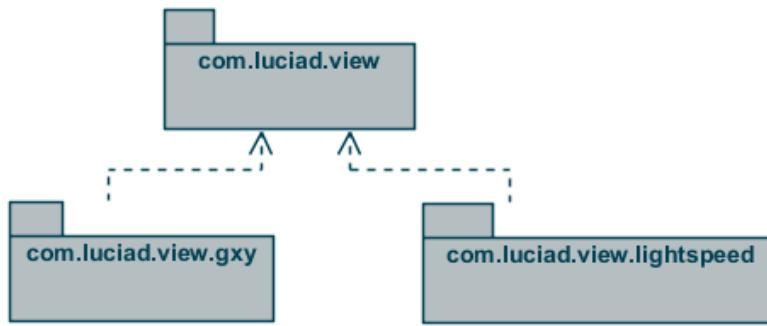


Figure 4 - com.luciad.view package structure

2.2.3 LuciadLightspeed controllers

A LuciadLightspeed controller defines the action that a user can perform on a view or on data visualized in the view. Multiple controllers can be defined in a LuciadLightspeed application but only one controller can be active in a view. To perform a controller action, a user typically uses the mouse, clicks on a button, or selects a menu item. LuciadLightspeed provides a number of commonly used controllers, for example, for selecting and editing objects visualized in a view and zooming in or out on a view. Section 2.4.3 lists the predefined LuciadLightspeed controllers that can be set on a view.

2.3 Deciding between GXY view and Lightspeed view

As discussed in Section 2.2.2, LuciadLightspeed comes with two types of view implementation: GXY view and a Lightspeed view. You can choose the most appropriate view implementation based on the nature of your application and the environment it will run in.

Your choice will be influenced by the following considerations:

- The hardware specifications of the target computer platforms on which your application is deployed
- The level of OpenGL support on the target computer platforms
- Specific requirements of your application

Section 2.3.1 and Section 2.3.2 outline the characteristics of each view type, so that you can use the appropriate implementation.



LuciadLightspeed offers adapters that allow you to integrate layers from a GXY view implementation into a Lightspeed view implementation, and the other way around. For instance, you can use a GXY view to build a core application that runs on all required platforms. For platforms that offer OpenGL support, you can use components of the Lightspeed view to tune specific functionality to make optimal use of graphics hardware acceleration. In the same way, you can gradually move an existing GXY view application to LuciadLightspeed by reusing your GXY code, and updating the performance-critical parts first. These adapter components are described in more detail in Section 41.1 and Section 41.2.

2.3.1 Benefits of Lightspeed view

The Lightspeed view implementation uses OpenGL to visualize data in 2D and in 3D. OpenGL is an industry standard for visualization, defined by the Khronos Group, which is available on many different platforms including Linux, Mac OS and Microsoft Windows. Typically, the vendor of the graphics hardware, the Graphics Processor Unit (GPU), in your system provides an OpenGL driver to enable OpenGL-accelerated applications. The Lightspeed view implementation requires such an OpenGL driver to be present on the system. For exact system requirements, refer to the LuciadLightspeed technical notes.

A GPU is basically a massively parallel processor that can process many points, triangles, and pixels simultaneously. Careful use of the GPU allows many operations involving the visualization of geospatial data to be performed extremely fast. This is exactly what the Lightspeed view does to provide you with a high-performance 2D and 3D view.

You should rely on a Lightspeed view by default when your system complies with the OpenGL and system requirements.

Examples of situations where a Lightspeed view is especially beneficial to the application's performance and functionality, are:

High-resolution displays. Because of the parallel handling of pixels, Lightspeed views scale well to high-resolution displays.

Anti-aliasing. The smoothing of jagged edges of curves and diagonal lines increases display quality. A Lightspeed view can achieve this with little to no performance loss.

Fluent animations. Lightspeed views are designed for high frame rates. A view that can be updated 60 times per second, allows for fluent animation effects, such as smooth fly-to animations or animated style changes.

Density painting. Specific use of the GPU in a Lightspeed view allows for interactive analysis capabilities such as density plots of air traffic.

Hypsometry. Analysis of terrain including slope calculation, ridge and valley detection, and shading by sunlight is instantaneous through judicious use of the GPU in the Terrain Analysis Engine component.

2D and 3D visualization. Lightspeed views have the capability to visualize in 2D and 3D within the same view. This means that you can easily switch from 2D to 3D in an application without additional development. Given the increasing importance of 3D visualization, this can save you significant development time.

2.3.2 Benefits of GXY view

The GXY view implementation uses Java AWT, in particular the `Graphics2D` components, to visualize data in 2D. `Graphics2D` is available on each of the many platforms that run Java 2 Standard Edition (J2SE). A GXY view should be used for applications that run on platforms with insufficient OpenGL support. In that sense, a GXY view may be a good choice if you are not entirely certain about the capabilities of the platforms that your application will be running on. Typical examples of such a situation are applications running on very old hardware, or web services running on server hardware without any graphics acceleration.

In addition, the GXY view implementation is highly optimized, and offers features such as asynchronous painting of domain objects to warrant a responsive view. As adapter components to integrate GXY into Lightspeed are available, there is no risk in choosing either a GXY view or

a Lightspeed view as a view implementation. Both implementations will be fully supported in future releases of LuciadLightspeed.

For information about the file formats supported in GXY views and Lightspeed views, see [Appendix E](#).

2.3.3 Other benefits of hardware acceleration

OpenCL analysis. The Open Computing language ([OpenCL](#)) allows performing general computations on the GPU, which is used in LuciadLightspeed for line-of-sight computations in the Terrain Analysis Engine component, for instance. OpenCL computations are performed at the model level. As such, OpenCL can be used with Lightspeed views as well as with GXY views.

2.4 Building applications with LuciadLightspeed

Figure 5 shows the separate components of a LuciadLightspeed application. The sections below provide more details on how to build each of these components.

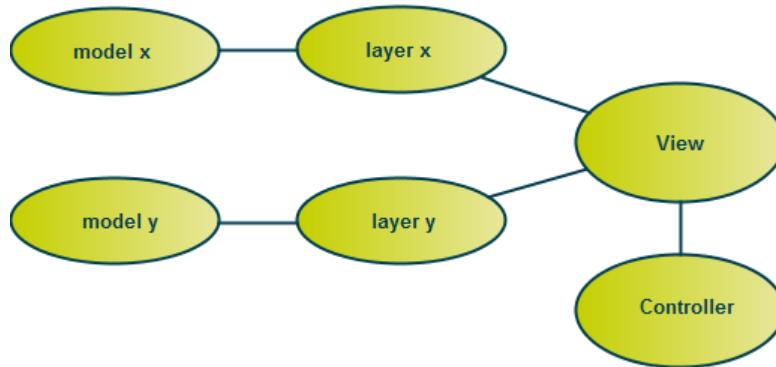


Figure 5 - The separate components of a LuciadLightspeed application

Referring to Figure 5, each LuciadLightspeed application needs to cover the following functionality:

- **Defining the models** Modeling the data, including the model reference and model descriptor
- **Defining the view.** Creating layers for the models, creating a view, setting the world reference, adding layers to the view
- **Defining the controllers.** Defining and setting one or more controllers on the view



Once you have defined a view it is very easy to add new data to a view by simply modeling the data, creating a layer for the model, and adding the layer to the view.

Because of the strict separation of model from view you can easily:

- Add new layers to an existing view
- Display multiple layers in one view as shown in [Figure 6](#)
- Display the same model in multiple views as shown in [Figure 7](#)

- Update all views that display the same model when the model data changes

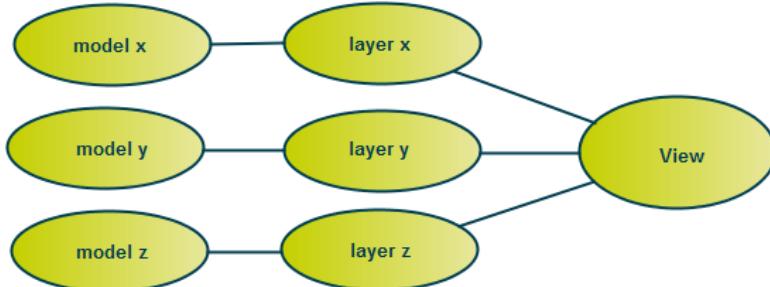


Figure 6 - Multiple layers visualized in one view

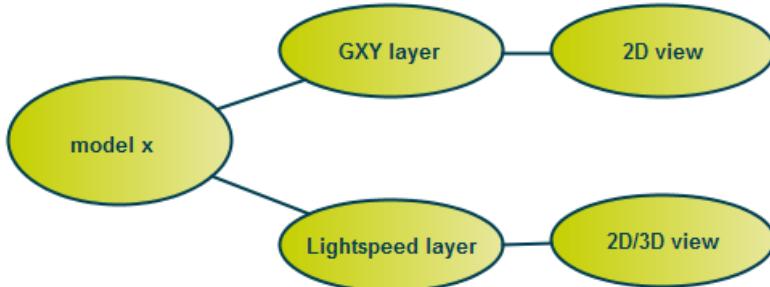


Figure 7 - One model visualized in multiple views

2.4.1 Defining the models

Depending on the origin of the model data there are two ways to create a LuciadLightspeed model:

- **Use a model decoder.** A model decoder is typically used to model data that comes from an external source. The external source can be of any type: online databases, web servers, data streams from radars or satellites, files on disk, and more. A model decoder loads data from the external source and creates a model for the loaded data including the model reference and model descriptor. LuciadLightspeed provides model decoders for a wide variety of commonly used vector and raster formats. [Table 14](#) gives an overview of all file formats that LuciadLightspeed supports. In case LuciadLightspeed does not provide a model decoder for the data that you want to load, you can create a custom model decoder, as described in [Section 5.1](#).

Each LuciadLightspeed model decoder is an implementation of the `ILcdModelDecoder` interface. Typically there is one implementation per data format. The two main methods of an `ILcdModelDecoder` implementation are:

- `canDecodeSource`: checks whether the implementation can decode the specified data source
- `decode`: creates a new `ILcdModel` from the specified data source



If you are only interested in the metadata of a dataset, you can use the `decodeModelMetadata` method to obtain it directly, without decoding the model itself.

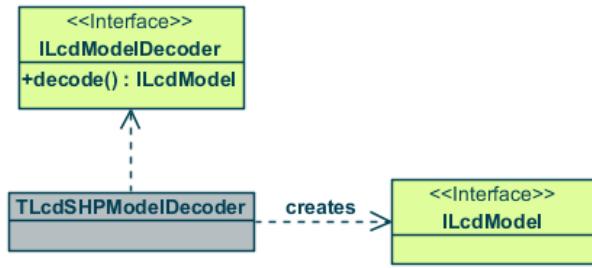


Figure 8 - Creating an `ILcdModel` using an `ILcdModelDecoder`

- **Create a model.** When the data is not available from an external source but is created by the application itself, the application needs to take care of modeling the data. You can use one of the existing LuciadLightspeed model implementations and define the model reference and descriptor as described in section [Section 5.1](#). Once the model is created you can additionally create a model encoder to store the data to an external source as described in [Section 17.3](#).

2.4.2 Defining the view

The following steps are required for defining a LuciadLightspeed view and visualizing model data in the view:

1. **Create a view.** To create a Lightspeed view, you need an implementation of `ILspView`. To create a GXY view, you need an implementation of `ILcdGXYView`. Sections [4.2.1](#) and [3.1](#) describe how to configure such views. LuciadLightspeed offers several view implementations, including Swing and AWT extensions.
2. **Define a world reference.** You need to define which coordinate system is used by the view. [Section 4.2.1](#) describes how a world reference is set and how the view can be configured.
3. **Create a layer for each model.** The recommended way of creating layers is using a **layer factory**. A layer factory creates a layer for a model and provides a painter to properly visualize the model data in a view. Using layer factories simplifies the design of your application and promotes the reusability of code. The LuciadLightspeed samples contain predefined layer factories that you can use in your application to create layers for models decoded by the predefined model decoders as listed in [Table 14](#). Check the `samples` directory in the LuciadLightspeed distribution for samples starting with `decoder`, for example `decoder.shp.*.*`, to find a specific code sample. You can also create custom layer factories for models that you have defined yourself or customize a predefined layer factory.
4. **Add layers to the view.** Once you have created layers and a view, you simply need to add the layers to the view as shown in [Section 4.2.4](#) and [Section 3.2](#).

2.4.3 Defining the controllers

To allow user interaction with a view and with the objects visualized in the view, you need to define one or more controllers. LuciadLightspeed offers a number of predefined controllers. Refer to the API reference for a detailed description of each of those controllers. [Chapter 26](#) provides you with more information about controller usage. You can customize the predefined controllers as described in [Section 26.2](#) and [Section 37.2](#).

Each controller is associated with one or more input actions, such as mouse clicks, or touches. The active controller translates the input action that a user performs to a change in the view or on the objects in the view. You can also associate a controller action to a button or a menu item in the GUI. [Section 4.3](#) and [Section 3.3](#) describe, for example, how to add controller actions to toggle buttons on a toolbar.



You can define multiple controllers in your application but only one controller can be active in a view. You can however combine different controllers into one controller by chaining them together.

To allow user interaction with a GXY view and with the objects visualized in the view, you need to define one or more controllers of the type `ILcdGXYController`. To allow for user interaction with a Lightspeed view, you need to define one or more controllers of the type `ILspController`.

2.5 LuciadLightspeed coordinate reference systems

A coordinate reference system (CRS) specifies the coordinate system that is used to define locations on the Earth or on a flat surface representing the Earth. Based on this definition, there are two types of CRSs:

- **Geodetic reference systems** (`ILcdGeodeticReference`) that represent geographic locations on an ellipsoidal (or a spherical) surface using **longitude and latitude coordinates**. Geodetic reference systems are based on an [ellipsoid](#) that approximates the shape of the earth. A commonly used geodetic reference system is the World Geodetic System 1984 ([WGS 1984](#)).
- **Cartesian or grid reference systems** (`ILcdGridReference`) that represent geographic locations on a flat surface using **x and y coordinates**. A [Cartesian](#) reference system is usually based on a geodetic reference system and requires a [map projection](#) to represent the curved surface of the earth on a flat surface as shown in [Figure 9](#).

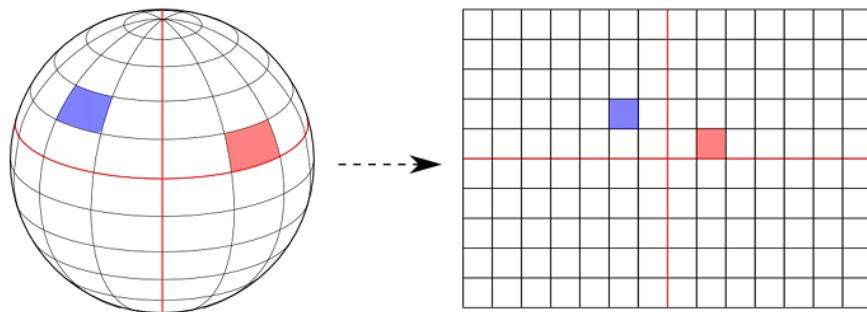


Figure 9 - Projecting the Earth's surface to a flat surface

Each CRS is associated with a **geodetic datum** (`ILcdGeodeticDatum`). A geodetic datum typically defines an ellipsoid and its position in relation to the Earth. The ellipsoid's position is specified by linking the ellipsoid to a number of physical points on the surface of the Earth. Alternatively, the ellipsoid's position can be specified by means of a translation and a rotation relative to a known geodetic datum, for example the one used by the WGS 1984.

Refer to the API reference for more details on `ILcdGeodeticReference` and `ILcdGridReference` and their implementations. [Section 2.5.1](#) introduces the model, view, and world coor-

dinate systems and the relationships between them. [Part VII](#) provides more information on how to use reference systems, map projections, and transformations in LuciadLightspeed.

2.5.1 Model, world, and view coordinates

The position of the model data is defined at three different levels in LuciadLightspeed using:

- **Model coordinates** defined within a coordinate system determined by the model reference. The model reference (`ILcdModelReference`), or CRS associated with an `ILcdModel`, usually is an `ILcdGeodeticReference` or `ILcdGridReference`.
- **World coordinates** defined within a coordinate system determined by the world reference. The world reference (`ILcdXYZWorldReference`), or CRS associated with an `ILspView`, usually is an `ILcdGridReference` for 2D or an `ILcdGeocentricReference` for 3D. For an `ILcdGXYView`, the associated world reference (`ILcdXYWorldReference`) usually is an `ILcdGridReference`.
- **View coordinates** defined by the 2D screen coordinates (or pixel coordinates).

Both the world and the view coordinate systems are associated with an `ILspView` or an `ILcdGXYView`. The world level is an intermediate level between the model and the view, which is used internally in the `ILspView` and `ILcdGXYView`, mainly for performance reasons. One of the benefits of LuciadLightspeed is that the model references of your data can differ from each other and from the world reference. LuciadLightspeed automatically converts the model data to the world reference. All model data is displayed in the same reference system in the view and you do not need to transform any data yourself. For more information on the different coordinate systems and transformations between them, refer to [Chapter 44](#).

CHAPTER 3

Building a basic application with a Light-speed view

This chapter provides a step-by-step guide to building a basic LuciadLightspeed application with a Lightspeed view. The application built in this chapter allows users to view and select country shapes on a raster image of the world. The chapter discusses the setup of the basic View, Model and Controller components of the application, before expanding this basic configuration with additional functions.

More specifically, the application contains the following functionality:

- A view displayed inside a `JFrame`
- A raster image of the world used as background data
- The countries of the world, read from a `.SHP` file
- The ability to navigate through the view
- The ability to switch between 2D and 3D visualization
- Check boxes to toggle the visibility of layers in the view
- Customized styling

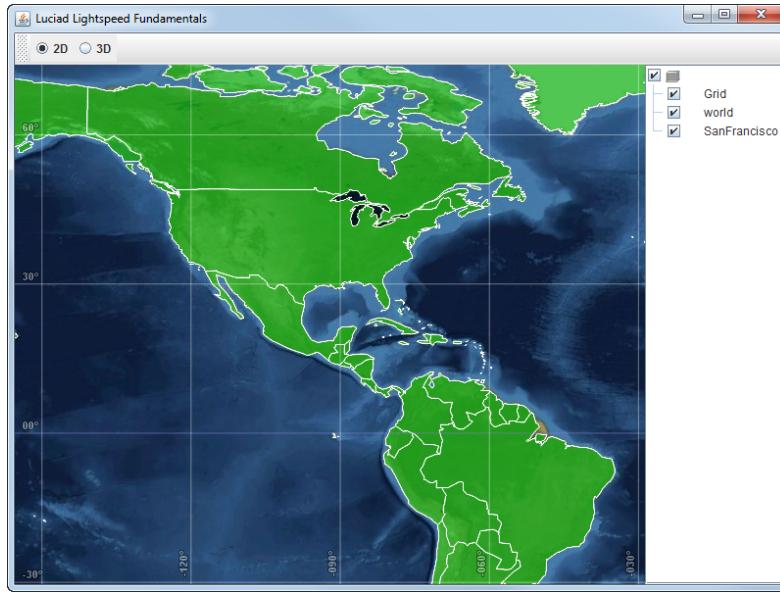


Figure 10 - The window with the components of a basic LuciadLightspeed application with a Lightspeed view

The concepts introduced in this chapter are explained in more detail in later chapters.

3.1 Creating a Lightspeed view

As a first step, we set up the window that we will use to display our model data in.

3.1.1 Setting up the Lightspeed view

To display a LuciadLightspeed application on screen, we need to create a Lightspeed view. Lightspeed views are defined by the interface `ILspView`. In [Program 1](#) we use the builder `TLspViewBuilder` to instantiate a `TLspAWTView`, which is the default implementation of `ILspView`. `TLspAWTView` defines a view which renders into an AWT component, Java's user interface toolkit.

```

1  private ILspAWTView createView() {
2      ILspAWTView view = TLspViewBuilder.newBuilder().buildAWTView();
3
4      // Set layer factory of the view. When adding models to the view, this factory
5      // is used to create layers for those models.
6      view.setLayerFactory(createLayerFactory());
7
8      return view;
9  }

```

Program 1 - Creating a view
 (from samples/lightspeed/fundamentals/step1/Main)

The layer factory used to create data layers later, is already added to the view here.

3.1.2 Displaying the application in a standalone window

The `TLspAWTView`, once it has been created, can be added to a frame and displayed on the screen:

```

1  fFrame = new JFrame("Luciad Lightspeed Fundamentals");
2  fFrame.getContentPane().setLayout(new BorderLayout());
3  fFrame.getContentPane().add(aView.getHostComponent(), BorderLayout.CENTER);
4  fFrame.add(createToolBar(aView), BorderLayout.NORTH);
5  fFrame.add(new JScrollPane(createLayerControl(aView)), BorderLayout.EAST);
6  fFrame.setSize(800, 600);
7  fFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
8  fFrame.setVisible(true);

```

Program 2 - Displaying the view
 (from samples/lightspeed/fundamentals/step1/Main)

Running the program at this point results in an empty window. Because the view does not contain any layers yet, it only displays a window background.

3.2 Adding data to the view

Now that the view has been set up, our next step is to add some data to the view. You can do this by creating an `ILcdModel` that contains the data, and then creating a layer for that model in the view.

3.2.1 Creating a model from a file

Program 3 shows how a model is created from a file, with an `ILcdModelDecoder`. Calling `ILspView.addLayersFor()` creates a layer for the resulting model.

```

1 // TLcdSHPModelDecoder can read ESRI SHP files
2 ILcdModelDecoder decoder = new TLcdSHPModelDecoder();
3
4 // Decode world.shp to create an ILcdModel
5 ILcdModel shpModel = decoder.decode("Data/Shp/World/world.shp");
6
7 // Calling addLayers() will cause the view to invoke its layer factory with
8 // the given model and then add the resulting layers to itself
9 Collection<ILspLayer> shpLayer = aView.addLayersFor(shpModel);

```

Program 3 - Creating a SHP model and adding it to the view
 (from samples/lightspeed/fundamentals/step1/Main)

In a very similar way, a model containing raster data is added in Program 4.

```

1 // Create a TLcdEarthRepositoryModelDecoder to decode Luciad Earth repositories
2 ILcdModelDecoder earthDecoder = new TLcdEarthRepositoryModelDecoder();
3
4 // Decode a sample data set (imagery data)
5 ILcdModel earthModel = earthDecoder.decode("Data/Earth/SanFrancisco/tilerepository.cfg");
6
7 // Calling addLayersFor() will cause the view to invoke its layer factory with
8 // the given model and then add the resulting layers to itself
9 aView.addLayersFor(earthModel);

```

Program 4 - Creating a raster model and adding it to the view
 (from samples/lightspeed/fundamentals/step1/Main)

At this point, an important part of the application is still missing. The data from the input models needs to be added to the view by means of one or more layers. However, the view does not know by itself how to create a corresponding layer for any given input model, as the configuration of the layer is partially dependent on the type of data present in the model.



This sample uses the Luciad Earth API, found in the package `com.luciad.earth`. This package is built around the concept of tilesets, which allows applications to extract a limited working set from a larger amount of data by using a hierarchical tiling mechanism. For more information about the Earth package, see [Chapter 14](#).

3.2.2 Using layer factories to create layers

We will use layer factories to create layers for the model data in the basic application. An `ILspLayerFactory` can create layers that have been tailored to display data from specific input models. For this reason, the application view uses an `ILspLayerFactory` to create layers for the SHP and raster models we just created. When the `addLayersFor()` method is called with a model as an argument, the view asks the layer factory to create an `ILspLayer` for the specified model. [Program 5](#) shows an example implementation of a layer factory that can handle the SHP and Luciad Earth models.

```

1 public class BasicLayerFactory extends ALspSingleLayerFactory {
2
3     @Override
4     public boolean canCreateLayers(ILcdModel aModel) {
5         // Check the model descriptor to see if this is a SHP model or an Earth
6         // repository model
7         return (aModel.getModelDescriptor() instanceof TLcdSHPModelDescriptor) ||
8             (aModel.getModelDescriptor() instanceof TLcdEarthModelDescriptor);
9     }
10
11    @Override
12    public ILspLayer createLayer(ILcdModel aModel) {
13        // Create a layer depending on the type of model.
14        if (aModel.getModelDescriptor() instanceof TLcdSHPModelDescriptor) {
15            // Create a layer with the given model.
16            TLspShapeLayerBuilder layerBuilder = TLspShapeLayerBuilder.newBuilder();
17            layerBuilder.model(aModel);
18            layerBuilder.layerType(ILspLayer.LayerType.BACKGROUND);
19            return layerBuilder.build();
20        } else if (aModel.getModelDescriptor() instanceof TLcdEarthModelDescriptor) {
21            // Create a raster layer using its builder, using all default settings.
22            return TLspRasterLayerBuilder.newBuilder().model(aModel).build();
23        } else {
24            return null;
25        }
26    }
27}

```

Program 5 - A layer factory

(from `samples/lightspeed/fundamentals/step1/BasicLayerFactory`)

The layer factory uses the model's `ILcdModelDescriptor` to determine the type of the model. If the descriptor is recognized, an appropriate layer is built in the `createLayer` method. Layer builders are used to facilitate the actual layer creation.



The builder design pattern is used extensively in LuciadLightspeed. It allows you to abstract away many details in the process of object creation, and to create the objects in small steps. For most of those steps, good default values are provided. As such, only the deviations from the default values need to be specified. The sequence of creation steps is finalized with the `build` method, which returns the created object.

There are three general-purpose builders that allow you to obtain layers in LuciadLightspeed. The type of the model data you are working with determines your choice of builder:

- `TLspShapeLayerBuilder`: used for vector data types such as points, polylines, polygons, arcs, and so on
- `TLspRasterLayerBuilder`: used to create layers for imagery or elevation data in the form of rasters, aerial photos or satellite imagery for example. See [Section 22.2.3](#) for more details.
- `TLspLonLatGridLayerBuilder`: used to create a longitude-latitude grid layer

The layer factory in the sample application uses the `TLspRasterLayerBuilder`, for which it only specifies the mandatory model argument and otherwise relies on the default settings.

For the `TLspShapeLayerBuilder`, the layer type is also changed to `BACKGROUND`. This instructs the view that this data set rarely changes, thereby allowing the view to optimize its caching strategy. As a consequence, the objects of this layer are not selectable by default, but you can make them selectable if required. Other layer types are `EDITABLE` by default. This layer type is intended for use with data that can be graphically edited by the user. Another important layer type is `REALTIME`, used for constantly changing data, moving tracks for example.

The layer factory must be assigned to the view with the `ILspView.setLayerFactory()` method, before the first model is added to the view. This was done in [Program 1](#).

It is also possible to add layers to the view directly. This is shown in [Program 6](#) where a grid layer is created and added to the view.

```
1 // Create and add the grid layer
2 aView.addLayer(TLspLonLatGridLayerBuilder.newBuilder().build());
```

Program 6 - Creating and adding a longitude/latitude grid layer
(from samples/lightspeed/fundamentals/step1/Main)



In this sample, the abstract class `ALspSingleLayerFactory` was extended. Implementing the interface `ILspLayerFactory` directly is a good option if you need to create multiple layers for a single input model. However, in this case, only one layer is required per input model. `ALspSingleLayerFactory` is more convenient, as its methods are tailored to a situation where there is a one-to-one mapping of model to layer.



The order of the layers in the view does not necessarily correspond with the order in which they were added. See [Section 6.4.1](#) for more information.

When we run the application now, we see a map of the world as in [Figure 11](#).

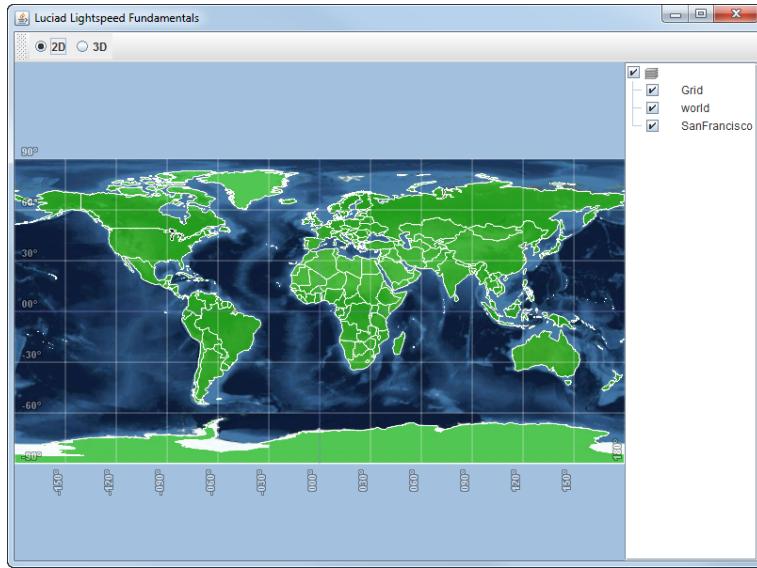


Figure 11 - Basic application displaying raster imagery, a SHP model and a longitude/latitude grid

The San Francisco layer contains elevation data. You can easily see this when you hide the world layer, and use the right mouse button to tilt the camera in 3D. The topographic relief is clearly visible.

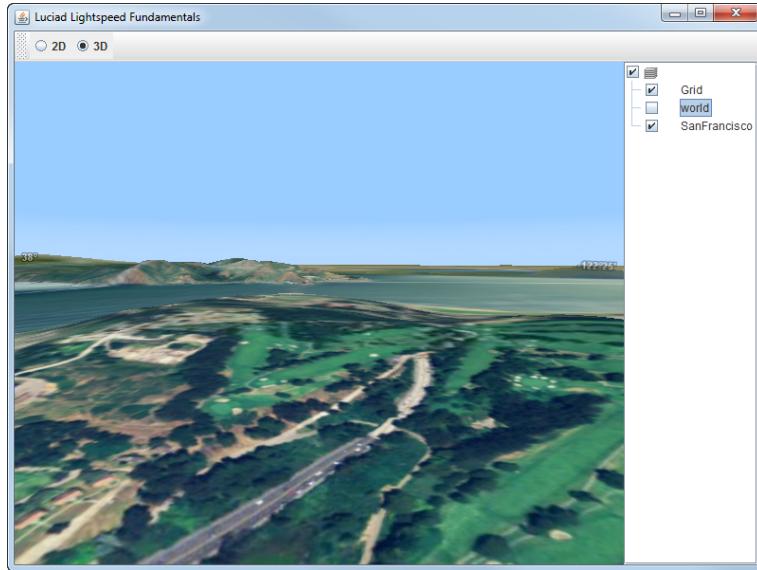


Figure 12 - Elevated terrain in a 3D Lightspeed view

To read more about the visualization of elevation data, see [Section 22.10](#).

3.3 Using a controller to make the view interactive

3.3.1 What functionality is offered by the default controller?

At this point, our program displays some data on a map, and allows us to use the mouse to navigate in the view. This function is provided by the view's default controller, which is repre-

sented by the `ILspController` interface. Essentially, a controller interprets input events and translates them into actions that are performed on the view, or on the models it contains.

The default controller combines view *navigation*, *selection* and *editing* with the following navigation and selection configuration:

- Dragging the left mouse button pans the view, unless a selected editable object is under the cursor.
- Dragging the middle mouse button pans the view.
- Dragging the right mouse button rotates the view.
- Scrolling the mouse wheel zooms in and out on the cursor location.
- Double-clicking the left mouse button starts a smooth, animated fly-to on the cursor location.
- Clicking the left mouse button selects an object.
- Dragging the left mouse button over one or more selectable objects while holding the **Shift** key selects the objects that are fully covered by the resulting selection rectangle.
- Clicking the left mouse button over one or more selectable objects while holding the **Alt** key displays a pop-up menu. The menu allows you to indicate which object you want to select.
- Clicking the left mouse button over a selectable object while holding the **Shift** key inverts the selection state of the object. An unselected object is selected, a selected object is deselected.
- Dragging the left mouse button while a selected object is under the cursor, moves the selected object.
- Dragging the left mouse button on a handle of a selected object edits the object.

In 3D, the controller allows you to navigate around tall or airborne objects like air tracks easily, by moving around and toward the object that is under the mouse.

3.3.2 Adding a navigation controller

There are a number of `ILspController` implementations available in the LuciadLightspeed API, but the `ControllerFactory` class that comes with the samples offers a very convenient way to instantiate one or more frequently used controllers. This class makes it easy to create controllers that can perform various tasks such as navigation, selection or editing. You can use it to add a navigation controller that provides navigation with all mouse buttons, as outlined in the default controller feature list in [Section 3.3.1](#), except that it does not select or edit objects. After creating a controller, we assign it to the view using its `setController()` method. Note that at this point, the default controller fits our needs. Later on, we replace the default controller in [Section 7.1.3](#).

For more information about controllers, see also [Chapter 26](#).

3.4 Switching between 2D and 3D views

LuciadLightspeed supports two-dimensional (2D) as well as three-dimensional (3D) visualization of the same data. Switching between the two visualization types is made easy by the availability of the class `TLspViewTransformationUtil`, which has methods to configure a view

as either 2D or 3D. These methods perform a few basic steps, including the switch between an orthographic top-down projection and a perspective projection.

Program 7 shows the creation of a JToolBar with radio buttons for 2D and 3D.

```

1  // Create and add toolbar to frame
2  JToolBar toolBar = new JToolBar();
3
4  // Create a button group for the radio buttons
5  ButtonGroup group = new ButtonGroup();
6
7  // Create a button to switch to 2D
8  JRadioButton b2d = new JRadioButton("2D", true);
9  b2d.setAction(new AbstractAction("2D") {
10    @Override
11    public void actionPerformed(ActionEvent e) {
12      TLspViewTransformationUtil.setup2DView(
13        aView,
14        new TLcdGridReference(new TLcdGeodeticDatum(),
15                               new TLcdEquidistantCylindrical()),
16        true
17      );
18    }
19  });
20  b2d.setToolTipText("Switch the view to 2D");
21  group.add(b2d);
22
23 // Create a button to switch to 3D
24 JRadioButton b3d = new JRadioButton("3D", false);
25 b3d.setAction(new AbstractAction("3D") {
26    @Override
27    public void actionPerformed(ActionEvent e) {
28      TLspViewTransformationUtil.setup3DView(aView, true);
29    }
30  });
31  b3d.setToolTipText("Switch the view to 3D");
32  group.add(b3d);
33
34 // Add the two buttons to the toolbar
35  toolBar.add(b2d);
36  toolBar.add(b3d);

```

Program 7 - Creating toolbar buttons to switch between 2D and 3D
 (from samples/lightspeed/fundamentals/step1/Main)

If we add this toolbar to our frame, we can switch between a 2D and 3D view type at runtime. Figure 13 shows an oblate spheroid in a 3D view.

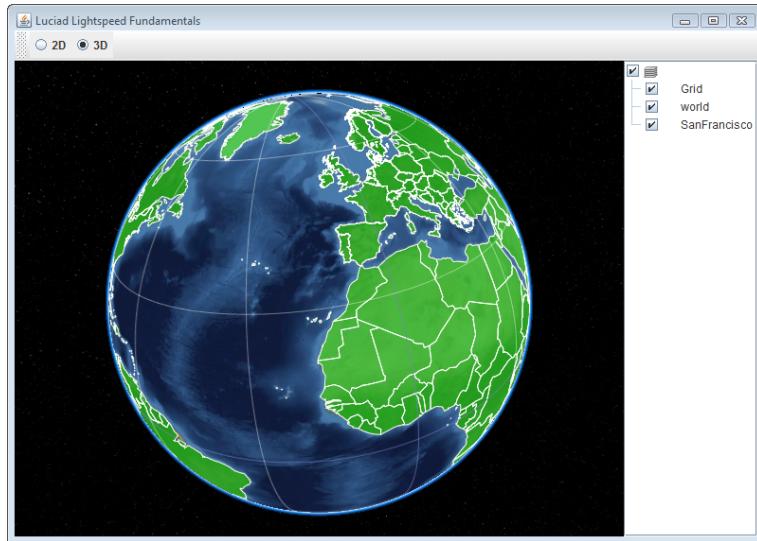


Figure 13 - The example application after switching to 3D

3.5 Changing the visual appearance of objects

We put the finishing touches on our basic application by changing the visual appearance of our countries layer. The management of the visual appearance of objects is also called styling. The appearance of a shape is determined by the styler assigned to the layer on which the shape object resides. The package `com.luciad.view.lightspeed.style` defines a mechanism used by most layers to configure the visual appearance of the objects they draw. Layers use an `ILspStyler` to obtain style information for each domain object they paint. `ILspStyler` objects provide a list of `ALspStyle` objects for every domain object. The layer's painter applies these to the object it is rendering. It can, for example, contain `TLspLineStyle` or `TLspFillStyle` objects, which in turn define colors, opacity, and so forth.

The `TLspShapeLayerBuilder` directly accepts a series of `ALspStyle` objects. This allows to conveniently assign a style to an entire layer, for example both a line and fill style when painting polygons. Additionally, every `ALspStyle` also implements `ILspStyler`, providing itself for every given domain object. Consequently, if you want to paint an entire layer with the same style, you can create that style and use it directly as a styler. Implementing your own styler is recommended when your styling requirements are more advanced, for example, when your styling is based on object attributes. A dedicated API supporting this use case is available. See [Chapter 22](#) for more information.

Builders are used to create the `ALspStyle` objects, as shown in [Program 8](#). All styles have a static `newBuilder` method. The builders themselves have self-explanatory methods, such as `color()` or `width()`. Of particular interest in this example is the use of `elevationMode()`: this style builder method indicates whether objects should be draped on the 3D terrain or not when painted in a 3D view. Here we use the mode `ON_TERRAIN`, which indicates that we want the layer to be draped. For more information about draping, see [Section 22.10](#).

```

1 // Create the fill and line styles using builders. The elevation mode for both
2 // styles is set to ON_TERRAIN, so that the data will be draped over the 3D terrain
3 // when the view is set to 3D.
4 TLspFillStyle fill = TLspFillStyle.newBuilder()
5     .color(Color.GREEN)
6     .elevationMode(ILspWorldElevationStyle.ElevationMode.
7         ON_TERRAIN)
8     .stipplePattern(TLspFillStyle.StipplePattern.
9         HALF_TONE_2x2)
10    .build();
11 TLspLineStyle line = TLspLineStyle.newBuilder()
12     .elevationMode(ILspWorldElevationStyle.ElevationMode.
13         ON_TERRAIN)
14     .build();

```

Program 8 - Building a styler

(from samples/lightspeed/fundamentals/step1/BasicLayerFactory)

Program 9 shows how the styles are provided to the layer builder. Different styles are provided, depending on the objects' state: selected or regular (not selected). Note how the `asBuilder` method is conveniently used to derive a builder from an existing style, thereby inheriting its properties such as the stipple pattern. Consequently, the selection style only needs to specify the style properties that differ from those of the regular style.

```

1 TLspShapeLayerBuilder layerBuilder = TLspShapeLayerBuilder.newBuilder();
2 layerBuilder.model(aModel);
3 layerBuilder.layerType(ILspLayer.LayerType.BACKGROUND);
4 // Assign the line and fill styles to the bodies in their regular object state
5 layerBuilder.bodyStyles(TLspPaintState.REGULAR, fill, line);
6
7 // Derive the selection styling from the regular styling, but change the color
8 layerBuilder.selectable(true);
9 layerBuilder.bodyStyles(TLspPaintState.SELECTED,
10     fill.asBuilder().color(Color.RED).build(),
11     line.asBuilder().color(Color.RED).width(3).build());

```

Program 9 - Assigning the styles to the layer builder

(from samples/lightspeed/fundamentals/step1/BasicLayerFactory)

Note that next to the body styles, the label styles can be modified as well. This is explained later, in Section 5.2.1.

When you run the application, the countries layer uses the settings you chose for your stylers, as shown in Figure 14.

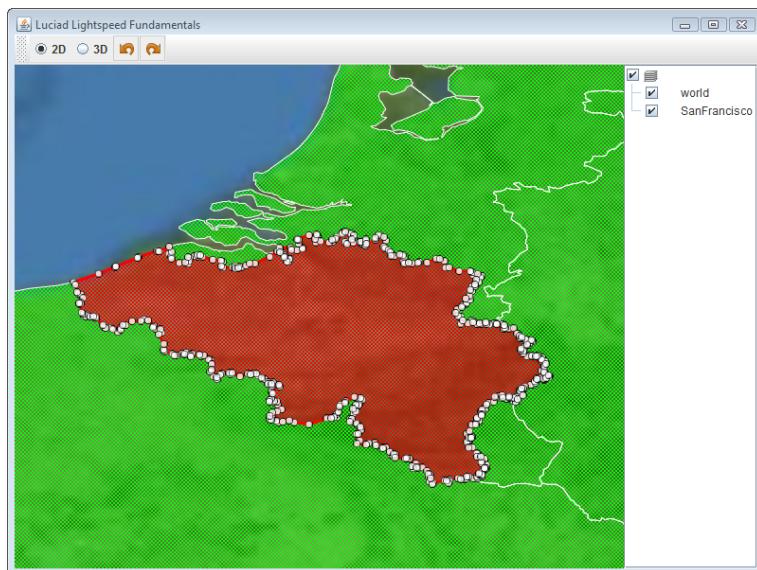


Figure 14 - Countries layer with a customized style

CHAPTER 4

Building a basic application with a GXY view

This chapter provides a step-by-step guide to building a basic LuciadLightspeed application with a GXY view. The application built in this chapter allows users to view a raster image of the world. The chapter discusses the setup of the basic View, Model and Controller components of the application, before expanding this basic configuration with additional functions.

More specifically, the application contains the following functionality:

- A view displayed inside a `JFrame`
- A raster image of the world used as background data
- The ability to navigate through the view
- Check boxes to toggle the visibility of layers in the view
- The ability to arrange and configure the layers in the view

To build a basic LuciadLightspeed application, following the outline given in [Section 2.4](#), these steps are required:

- **Defining the model:** loading and modeling the background data
- **Defining the view:** creating a view, setting the world reference, creating a layer for the background data, adding the background layer to the view, creating a simple layer control
- **Defining a controller:** creating and setting up a controller that pans and zooms

The sections below describe each of these steps in detail and provide code snippets per step. [Program 304](#) is the collection of the code snippets in this chapter. [Figure 15](#) shows the window and the components of the basic application built in this chapter.

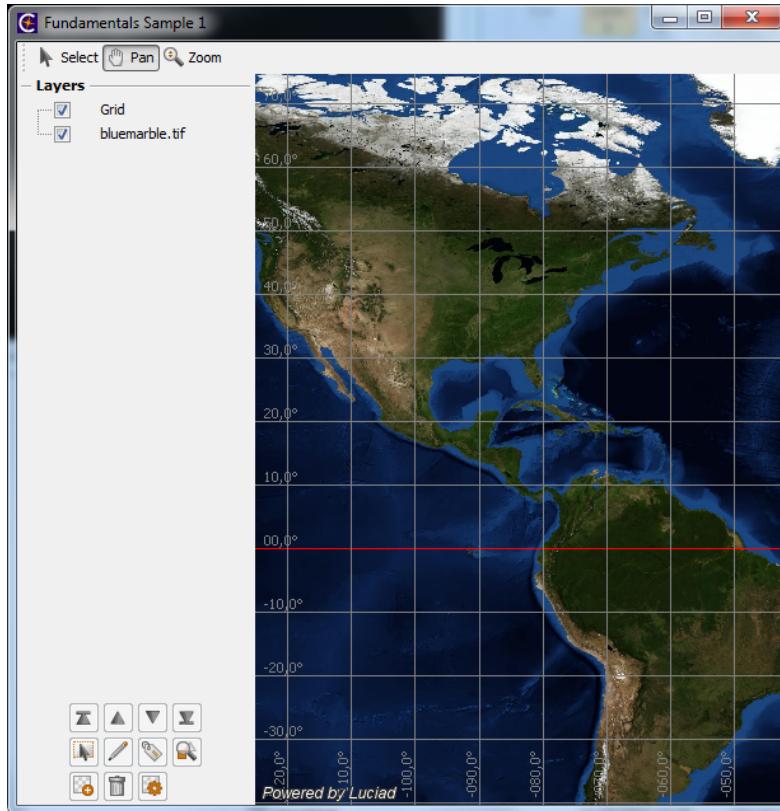


Figure 15 - The window with the components of a basic LuciadLightspeed application

4.1 Defining the model

As described in [Section 2.4.1](#), background data is usually loaded from an external source and decoded by a model decoder. [Section 4.1.1](#) describes how to load and model a GeoTIFF file using a LuciadLightspeed model decoder.

4.1.1 Loading and modeling the background data

The file `bluemarble.tif` contains an image of the world that serves well as background in a basic application. You can find the file `bluemarble.tif` in the directory `LuciadLightspeed_<x.x>\samples\resources\Data\GeoTIFF\BlueMarble`.

To load and model a `.tif` file, you can use the `TLcdGeoTIFFModelDecoder`. [Program 10](#) shows how to set up the model decoder and decode `bluemarble.tif`.

```

1 // Creates the model.
2 TLcdGeoTIFFModelDecoder modelDecoder = new TLcdGeoTIFFModelDecoder();
3 ILcdModel model = modelDecoder.decode("Data/GeoTIFF/BlueMarble/bluemarble.tif");

```

Program 10 - Loading and decoding a GeoTIFF file
(from `samples/gxy/fundamentals/step1/Main`)

For `bluemarble.tif`, the `TLcdGeoTIFFModelDecoder` returns a model of the type `ILcd2DBoundsIndexedModel` including:

- A model reference of the type `ILcdGeodeticReference`
- A model descriptor of the type `TLcdMultilevelGeoTIFFModelDescriptor`

As the model descriptor indicates, the decoded model is a multilevel raster model. This means that the model contains a set of rasters, each with another level of detail as explained in [Section 13.2](#). For more information on `TLcdGeoTIFFModelDecoder` and the returned model, references, and descriptors, refer to the API reference. [Section 4.2.3](#) describes how to create a layer for the decoded model.

4.2 Defining the view

The following sections describe the steps required to define a GXY view and add the layer with the background data to the view. Note that once you have defined the view, you can reuse it in any other LuciadLightspeed application.

4.2.1 Creating a view

As described in [Section 2.4.2](#) you need an implementation of `ILcdGXYView` to create a GXY view. For a basic application the `TLcdMap JPanel` is recommended as it provides a grid and a default world reference. Associated with a `TLcdMap JPanel` are:

- a default `ILcdGXYLayer` that displays a longitude/latitude grid. This grid can be set on and off using the `withGridLayer` property.
- a `TLcdGridReference` as `ILcdXYWorldReference` that has an instance of `TLcdEquidistantCylindrical` as `ILcdProjection`. The `TLcdGeodeticDatum` of the grid reference is the geodetic datum WGS84.

[Program 11](#) shows how to create and set up the map. Creating a paint queue manager allows you to wrap a layer in an `ILcdGXYAsynchronousLayerWrapper`, so it can be rendered in a background thread, without blocking the GUI thread. We'll come back to this when we talk about adding a layer to the view.

```

1 // Creates the 2D view.
2 TLcdMap JPanel map = new TLcdMap JPanel();
3 TLcdGXYAsynchronousPaintQueueManager manager = new TLcdGXYAsynchronousPaintQueueManager();
4 manager.setGXYView(map);

```

Program 11 - Setting up the map
(from samples/gxy/fundamentals/step1/Main)

4.2.2 Setting the world reference

Since a `TLcdMap JPanel` already has a default world reference, you do not need to define one yourself. If you require another geodetic datum and/or map projection you can easily change the default world reference to your requirements. [Program 12](#) shows how to change the default map projection to the Mercator projection (`TLcdMercator`). [Chapter 43](#) provides more information on the map projections that LuciadLightspeed offers and how to use them.

```

1 // Defines a custom world reference for the map.
2
3 map.setXYWorldReference( new TLcdGridReference( new TLcdGeodeticDatum( ), new TLcdMercator() )
    );

```

Program 12 - Changing the map projection

4.2.3 Creating a layer for the background data

As explained in [Section 2.4.2](#), the recommended way to create a layer is using a layer factory. The decoded model for `bluemarble.tif` is a multilevel raster model. [Program 13](#) shows how you can create and set up an `ImageLayerFactory` to create layers for multilevel raster models.

```

1 // Creates the MultilevelRasterLayerFactory.
2
3 public class ImageLayerFactory implements ILcdGXYLayerFactory {
4     @Override
5     public ILcdGXYLayer createGXYLayer(ILcdModel aModel) {
6         if (!(aModel.getModelDescriptor() instanceof ILcdImageModelDescriptor)) {
7             return null;
8         }
9         TLcdGXYLayer layer = new TLcdGXYLayer(aModel);
10
11        // Sets a pen on the layer.
12
13        layer.setGXYPen(ALcdGXYPen.create(aModel.getModelReference()));
14
15        // Creates an ILcdGXYPainter to paint an ALcdBasicImage.
16
17        TLcdGXYImagePainter painterProvider = new TLcdGXYImagePainter();
18        painterProvider.setFillOutlineArea(true);
19        layer.setGXYPainterProvider(painterProvider);
20
21        layer.setSelectable(false);
22
23        return layer;
24    }
25}
```

Program 13 - Creating and setting up a ImageLayerFactory
 (from `samples/gxy/fundamentals/step1/ImageLayerFactory`)

As [Program 13](#) shows, the `ImageLayerFactory` returns a layer (`TLcdGXYLayer`) for a model (`ILcdModel`) that has a multilevel raster model descriptor (`TLcdMultilevelGeoTIFFModelDescriptor`).

A `TLcdGXYLayer` is an implementation of `ILcdGXYLayer` which is suited for an `ILcdGXYView` and sets:

- A **model** to the layer
- A **label** to the layer by getting the display name from the model descriptor of the model
- A **pen** to the layer. A pen is a utility class that supports the painting of lines and other basic `Shapes` based on the required model, world, and view transformations. The type of pen (geodetic or grid) depends on the model reference. Since `bluemarble.tif` has a geodetic reference, the corresponding pen is a `TLcdGeodeticPen`. For more information about the usage of an `ILcdGXYPen`, refer to [Section 32.3](#). [Section 2.5.1](#) provides more information on the model, world, and view transformations.
- A **painter provider** to the layer. The painter provider provides a painter. A **painter** defines the visual representation of the model data and paints it properly in the view. The pre-defined LuciadLightspeed painters, as listed in [Section 32.1.2](#), implement `ILcdGXYPainterProvider` for convenience; they return themselves. This is useful in case the model contains only one type of model data. The type of painter depends on the type of model data. For raster data, LuciadLightspeed provides a `TLcdGXYImagePainter`, or a `TLcdRasterPainter` and a `TLcdMultilevelRasterPainter`, depending on how you are modeling your raster data. For more information about your options for raster

data modeling, see [Chapter 13](#) and [Chapter 12](#).

For more information on painters and painter providers, refer to [Section 32.1](#).

Program 14 shows how to create the background layer using the `ImageLayerFactory`.

```
1 // Creates the background layer.
2 ILcdGXYLayer layer = new TLcdGXYAsynchronousLayerWrapper(new ImageLayerFactory().  
createGXYLayer(model));
```

Program 14 - Creating an asynchronously painted background layer using the `ImageLayerFactory` and `TLcdGXYAsynchronousLayerWrapper`
(from `samples/gxy/fundamentals/step1/Main`)

Raster data might take a considerable amount of time to load and render. That's why we wrap the layer in a `TLcdGXYAsynchronousLayerWrapper`. This offloads all rendering to a background thread. For more information on this, refer to [Chapter 33](#).

4.2.4 Adding the background layer to the view

Program 15 shows how to add the background layer to the view. The grid layer provided by `TLcdMapView` is moved to the top so that it is visualized above the background layer. For more information on working with layers, refer to [Chapter 6](#).

```
1 // Adds the background layer to the view and moves the grid layer to the top.  
2 aView.addGXYLayer(layer);  
3 aView.moveLayerAt(aView.layerCount() - 1, aView.getGridLayer());
```

Program 15 - Adding the background layer to the view and moving the grid layer
(from `samples/gxy/fundamentals/step1/Main`)

Another way to add a model to a view is to set an `ILcdGXYLayerFactory` to the view and call `addModel`. The view then automatically creates and adds the layer.

4.2.5 Creating a simple layer control

A layer control is a frequently used component in LuciadLightspeed applications and allows the user to view, rearrange, and configure the individual layers that are added to the view. LuciadLightspeed provides a Swing layer tree for this: `TLcdLayerTree`. You can add the layer tree to a GUI or a standalone window as described in [Section 4.4](#).

```
1 return new TLcdLayerTree(aView);
```

Program 16 - Creating a layer tree
(from `samples/gxy/fundamentals/step1/Main`)

The sample class `LayerControlPanelSW` further builds on this layer tree. You can find it in `samples.common.layerControls.swing`. `LayerControlPanelSW` adds buttons to the layer tree that allow the user to change the layer ordering, set layer properties, and fit the view to the data in a layer.

4.3 Creating a controller

LuciadLightspeed offer a variety of controllers that you can set on a 2D view. Each of these controllers is associated with certain user input, such as mouse drags or clicks, multi-touch, and so on. Program 17 shows how to create a pan and zoom controller.

Chapter 37 provides more information on LuciadLightspeed controllers and controller actions.

```

1 TLcdGXYCompositeController compositeController = new TLcdGXYCompositeController();
2 TLcdGXYPanController controller = new TLcdGXYPanController();
3 controller.setAWTFilter(TLcdAWTEventFilterBuilder.newBuilder().leftMouseButton().build());
4 controller.setDragViewOnPan(true);
5 compositeController.addGXYController(controller);
6 compositeController.addGXYController(new TLcdGXYZoomWheelController());
7 aView.setGXYController(compositeController);

```

Program 17 - Creating a controller for a 2D view
 (from samples/gxy/fundamentals/step1/Main)

4.4 Displaying the map in a standalone window

Once the map (TLcdMapJPanel) is set up as described in Section 4.2.1, you can insert it into a GUI. Since the basic application built in this chapter is not part of a GUI, the default code is adapted so that the code can run standalone as shown in the header of Program 304.

The Swing class JPanel is used to lay out the GUI of the basic application. Program 18 illustrates how to add the map in the middle, and the layer control at the left. The main method puts this panel in a top-level window (JFrame). For more information on using Swing and AWT panels and frames, refer to the Java documentation.

```

1 fFrame = new JFrame("LuciadLightspeed GXY Fundamentals");
2 fFrame.getContentPane().setLayout(new BorderLayout());
3 fFrame.getContentPane().add(aView, BorderLayout.CENTER);
4 fFrame.add(new JScrollPane(createLayerControl(aView)), BorderLayout.EAST);
5 fFrame.setSize(800, 600);
6 fFrame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
7 fFrame.setVisible(true);

```

Program 18 - Adding the map to a Swing panel
 (from samples/gxy/fundamentals/step1/Main)



Just as with Swing, LuciadLightspeed imposes some threading rules on its view implementations. As long as you execute your code in Java's Event Dispatch Thread (EDT) you do not have to worry about this. If your code runs on another thread (for example, the one that calls public static void main), you can execute a Runnable on the EDT using java.awt.EventQueue.invokeLater as shown in Program 304. See Chapter 9 for an overview of LuciadLightspeed's threading rules.

CHAPTER 5

Loading and visualizing business data

Once you have created a basic LuciadLightspeed application with a GXY view, as described in [Chapter 4](#), or with a Lightspeed view, as described in [Chapter 3](#), you can model data that you want to interact with (business data). Next, you can visualize it in the view together with the background data. The business data can either be loaded from an external source or created by the application itself. This chapter describes how to model and visualize business data that is loaded from an external source in a custom file format that is not supported out of the box.

The main steps of loading and visualizing business data from an external source are:

- **Modeling the business data:** creating a model and model decoder for the business data
- **Visualizing the business data:** creating a layer for the business data and adding the layer to the view
- **Interacting with the business data:** the ability to select and edit objects

This chapter describes each of these steps in detail and provides code snippets per step. The business data samples are a combination of the application built in the previous two Chapters and the code snippets described in this chapter. They are shipped with the release as part of the fundamentals sample collection.

In this chapter, we use screenshots from the basic application with a Lightspeed view in 2D. The basic application with a GXY view looks nearly identical.

The sample uses flight plans as business data. A set of flight plans is loaded from a custom text file. The file contains records (flight plans) separated by an empty line. Each record has one line with the name of the flight plan, followed by several lines, each containing one coordinate pair. The longitude and latitude coordinates are expressed as double values and are separated by a space. For example:

```
Flight Plan A001
```

```
10 10
```

```
10 12
```

```
10 14
```

```
Flight Plan A002
```

```
20 20
```

```
21 20
```

```
...
```

```
30 20
```

<EOF>

5.1 Modeling the business data

Most of the time, it suffices to use one of the many model decoders that are provided by the LuciadLightspeed API to read business data from an external source. For example, in the basic application of [Chapter 3](#), the `TLcdSHPModelDecoder` is used to decode .SHP files.

However, since the sample flight plan data is defined in a custom format for which LuciadLightspeed does not provide a default model decoder, you need to create a model decoder yourself (*note that it is also possible to create the model without a model decoder, for example when reading data from a live stream*).

To create a model decoder you need to choose or define LuciadLightspeed classes for:

- the domain objects (the model data)
- the model reference
- the model descriptor
- the model

The sections below describe each of these steps in detail illustrated by the flight plan example. At the end, [Section 5.1.5](#) describes how to create the model decoder for the flight plan model.

5.1.1 Modeling the domain objects

As described in [Section 2.2.1](#), a LuciadLightspeed model is a container for a set of domain objects. The domain objects are typically defined by:

- a **geometry**, or **shape**, which is typically but not necessarily the same for all domain objects of one model. Examples are points, lines, circles, and any combination of shapes.
- a set of one or more **properties**, of which the values for each domain object are different



A LuciadLightspeed model can contain vector or raster data. This section describes how to model vector data. To model raster data, you can use a default LuciadLightspeed model decoder as described in [Chapter 13](#).

To define a class for the domain objects, you first need to:

- **Determine the geometry** of the domain objects and map this to one of the LuciadLightspeed shapes (`ILcdShape`). [Section 11.1.2](#) lists the subinterfaces of `ILcdShape` that all define a shape which, in most cases, maps to the geometry of the business data. If the geometry of the business data does not map to a LuciadLightspeed shape, you can define a shape yourself as shown in [Chapter 35](#). The sample business data, the flight plans, are defined by a list of coordinate pairs. Each coordinate maps to a point (`ILcdPoint`). The flight plan itself maps to an `ILcdPolyline` which is a list of `ILcdPoint` objects. Which implementation of the `ILcdPolyline` to use, depends on the model reference as described in [Section 5.1.2](#). For the flight plans, `TLcdLonLatPolyline` objects are used that each consist of a list of `TLcdLonLatPoint` objects.
- **Determine the properties** of the domain objects. The source file with the flight plans contains one property for each flight plan: the name of the flight plan.

To represent and access domain objects and their properties, LuciadLightspeed provides the interface `ILcdDataObject`. The properties of a domain object are defined by a data type (`TLcdDataType`). To logically group a number of data types, a `TLcdDataModel` is used. In case of the flight plans, only one data type is needed since all domain objects have the same structure. The flight plan data type defines one property: the flight plan name.

Program 19 shows how the `createDataModel` method is used to create a `TLcdDataModel` for the single flight plan type with a `TLcdDataModelBuilder`. The data model is given a unique name space but you can also use a simple name such as `FlightPlanModel`.



For illustration purposes this section describes sample data of a simple data type with only one property. In practice domain objects can have different geometries with large sets of (hierarchically structured) properties. Chapter 10 provides detailed information on modeling domain data and using `ILcdDataObject` as the general interface for representing and accessing domain objects and their properties.

```

1 /**
2 * This class builds the structural description of the flight plan model, and provides
3 * static access to it. The method getDataModel() provides the full flight plan data model.
4 * The public constant FLIGHT_PLAN_DATA_TYPE refers to the only defined type of this model:
5 * flight plans.
6 */
7
8 public final class FlightPlanDataTypes {
9
10    // The data model for the flight plans, fully describing the structure of the data.
11    private static final TLcdDataModel FLIGHT_PLAN_DATA_MODEL;
12
13    // The data model contains a single data type - the flight plan data type.
14    public static final TLcdDataType FLIGHT_PLAN_DATA_TYPE;
15
16    public static final String NAME = "name"; //Starts with lower case, same as Java property
17    public static final String POLYLINE = "polyline";
18
19    static final String FLIGHT_PLAN_TYPE = "FlightPlanType"; //Starts with capital, same as Java
20        class
21
22    static {
23        // Assign the constants
24        FLIGHT_PLAN_DATA_MODEL = createDataModel();
25        FLIGHT_PLAN_DATA_TYPE = FLIGHT_PLAN_DATA_MODEL.getDeclaredType(FLIGHT_PLAN_TYPE);
26    }
27
28    private static TLcdDataModel createDataModel() {
29        // Create the builder for the data model.
30        // Use some unique name space, to prevent name clashes. This isn't really needed
31        // for the sample but might be useful when exposing it externally.
32        TLcdDataModelBuilder builder = new TLcdDataModelBuilder(
33            "http://www.mydomain.com/datamodel/FlightPlanModel");
34
35        builder.typeBuilder("PolylineType").superType(TLcdShapeDataTypes.SHAPETYPE).primitive(
36            true).instanceClass(ILcd2DEditablePolyline.class);
37
38        // Define the types and their properties (only one type and one property here)
39        TLcdDataTypeBuilder flightPlanBuilder = builder.typeBuilder(FLIGHT_PLAN_TYPE);
40        flightPlanBuilder.addProperty(NAME, TLcdCoreDataTypes.STRING_TYPE);
41        flightPlanBuilder.addProperty(POLYLINE, "PolylineType");
42
43        // Finalize the creation
44        TLcdDataModel dataModel = builder.createDataModel();
45
46        TLcdDataType type = dataModel.getDeclaredType(FLIGHT_PLAN_TYPE);
47        // make sure LuciadLightspeed finds the geometry
48        type.addAnnotation(new TLcdHasGeometryAnnotation(type.getProperty(POLYLINE)));
49
50        return dataModel;

```

```

49 }
50
51 public static TLcdDataModel getDataModel() {
52     return FLIGHT_PLAN_DATA_MODEL;
53 }
54 }
```

Program 19 - Creating the data model for the flight plan data type
 (from samples/gxy/fundamentals/step2/FlightPlanDataTypes)

There is no need to code a specific domain object class for the flight plan. `TLcdDataObject` is used to store all properties and implement the `ILcdDataObject` behavior. `ILcdDataObject` provides access to both the geometry and the flight plan name, as defined by the flight plan data type. Note how you can define constants to conveniently access the name and geometry properties of the flight plans. The geometry of the flight plan is a `TLcdLonLatPolyline`. For more information on `TLcdLonLatPolyline` and `ILcd2DEditablePointList`, refer to [Chapter 11](#).

If you want to implement the `ILcdDataObject` interface in your own domain object class, make sure to configure that class in the data type builder. This ensures that when using `newInstance` to create new instances of the flight plan data type, the correct class is used.

5.1.2 Determining the model reference

In order to determine the location of the source data, a reference system needs to be associated with the source data (the model reference). The reference system is usually provided together with or as part of the source data. The model decoders provided by LuciadLightspeed automatically retrieve the model reference from the source data. In case you do not use a predefined model decoder, you can decode the reference information from the source data as described in [Chapter 16](#).

In the sample case of the flight plans, the model reference is the World Geodetic System 1984 (WGS 1984), a commonly used `TLcdGeodeticReference`. Since the model reference is geodetic, the implementation of the LuciadLightspeed class for defining the geometry contained in the model needs to be geodetic as well. In case of the flight plan geometry, the corresponding class is `TLcdLonLatPolyline`. For more information on the different types of reference systems, refer to [Section 2.5](#).

5.1.3 Determining the model descriptor

To provide a general description of the contents of a LuciadLightspeed model, LuciadLightspeed provides the interface `ILcdModelDescriptor`. An `ILcdModelDescriptor` typically contains the name, the type, and the display name of the model data. In case the domain objects implement `ILcdDataObject`, the model should have an `ILcdModelDescriptor` that implements `ILcdDataModelDescriptor`. The `ILcdDataModelDescriptor` gives access to the data model that contains the data types of the domain objects as described in [Chapter 10](#). The model descriptor is, for example, used by the layer factory to check the type of model as described in [Section 5.3.1](#) and [Section 5.2.1](#).

[Program 20](#) in [Section 5.1.5](#) shows how to create the model descriptor for models of the flight plan data type using the `createModelDescriptor` method.

5.1.4 Choosing the model class

Once you have modeled the domain objects, you need to choose a LuciadLightspeed class for the model that serves as a container for the domain objects. Defining the appropriate class depends on the geometry and properties of the domain objects and their usage. The two general LuciadLightspeed classes for modeling vector data are:

- `TLcdVectorModel` is an implementation of `ILcdIntegerIndexedModel` which stores the domain objects in a **vector** allowing **fast integer-based queries**. A `TLcdVectorModel` is suited for models that have a limited set of domain objects that are not retrieved often or for models with **dynamic data** - data that is subject to many changes. Since the domain objects are not spatially indexed, this model is not suited for spatial queries. It is more suited for sequential access of the domain objects and to quickly add or update objects.
- `TLcd2DBoundsIndexedModel` is an implementation of both `ILcdIntegerIndexedModel` and `ILcd2DBoundsIndexedModel` which stores the domain objects in a **spatial index**. The domain objects are indexed based on their coordinates, or **bounds**, which enables **fast spatial queries**. `TLcd2DBoundsIndexedModel` is less suited for dynamic data since it is costly to update the spatial index. It is more suited for models with many domain objects that are retrieved often by a spatial query or for models with **static data** - data that is not likely to change. Layers in a 2D view, for example, perform a spatial query on a `TLcd2DBoundsIndexedModel` to quickly find the objects that are visible in the view.

`TLcd2DBoundsIndexedModel` is used to model the sample flight plans as it is not very likely that the data is subject to many changes.

5.1.5 Creating a model decoder

Once you have chosen a model class for your business data, including the model reference and the model descriptor, you can create a model decoder to model your data. As described in the previous sections, the flight plans are defined by the following classes:

- Domain objects: an `ILcdDataObject` containing a `TLcdLonLatPolyline` and a name
- Model: `TLcd2DBoundsIndexedModel`
- Model reference: `TLcdGeodeticReference`
- Model descriptor: `TLcdDataModelDescriptor`

With this information you can create a model decoder for the flight plans as shown in Program 20.

```

1 /**
2 * A model decoder that reads vector data stored in a custom text format.
3 * The file contains records (flight plans) separated by an empty line.
4 * Each record has one line with the name of the flight plan, followed by
5 * several lines containing one coordinate pair each.
6 * The longitude and latitude coordinates are expressed as double values and
7 * are separated by a space.
8 *
9 * For example:
10 *
11 * Flight Plan A001
12 * 10 10
13 * 10 12
14 * 10 14
15 */

```

```

16  *
17  * Flight Plan A002
18  * 20 20
19  * 21 20
20  * ...
21  * 30 20
22  * <EOF>
23  */
24 @LcdService
25 public class FlightPlanModelDecoder implements ILcdModelDecoder {
26     private static final String DISPLAY_NAME = "Flight Plans";
27     private static final String TYPE_NAME = "Custom";
28
29     @Override
30     public String getDisplayName() {
31         return DISPLAY_NAME;
32     }
33
34     @Override
35     public boolean canDecodeSource(String aSourceName) {
36         return aSourceName.endsWith(".cfp");
37     }
38
39     @Override
40     public ILcdModel decode(String aSourceName) throws IOException {
41         if (!canDecodeSource(aSourceName)) {
42             throw new IOException("Cannot decode " + aSourceName);
43         }
44
45         // Create a spatially indexed model, with a geodetic (lon/lat) reference system and
46         // a descriptor for the meta-data
47         TLcd2DBoundsIndexedModel model = new TLcd2DBoundsIndexedModel();
48         model.setModelReference(new TLcdGeodeticReference());
49         model.setModelDescriptor(createModelDescriptor(aSourceName));
50
51         // Create an input stream from the file name.
52         TLcdInputStreamFactory inputStreamFactory = new TLcdInputStreamFactory();
53         InputStream inputStream = inputStreamFactory.createInputStream(aSourceName);
54         TLcdDataInputStream dataInputStream = new TLcdDataInputStream(inputStream);
55         try {
56             // Read the records from the file one by one and add them to the model.
57             ILcdDataObject flightPlan;
58             while ((flightPlan = readRecord(dataInputStream)) != null) {
59                 model.addElement(flightPlan, ILcdFireEventMode.NO_EVENT);
60             }
61
62         } finally {
63             dataInputStream.close();
64         }
65         return model;
66     }
67
68     private ILcdModelDescriptor createModelDescriptor(String aSourceName) {
69         return new TLcdDataModelDescriptor(
70             aSourceName,
71             TYPE_NAME,
72             DISPLAY_NAME,
73             FlightPlanDataTypes.getDataModel(),
74
75             // ILcdModel.elements only returns objects of type FLIGHT_PLAN_DATA_TYPE
76             Collections.singleton(FlightPlanDataTypes.FLIGHT_PLAN_DATA_TYPE),
77
78             FlightPlanDataTypes.getDataModel().getTypes());
79     }
80
81     /**
82      * Reads one single record (flight plan) from the input file.
83      * @param aInputStream: a reader for the input file
84      * @return a single flight plan record
85      * @throws IOException In case of I/O failure.
86      */
87
88     private ILcdDataObject readRecord(TLcdDataInputStream aInputStream) throws IOException {

```

```

89     String line;
90
91     // Skip any empty lines.
92     do {
93         line = aInputStream.readLine();
94         if (line == null) {
95             return null;
96         }
97     } while (line.trim().length() == 0);
98
99     // The first non-empty line contains the flight plan name.
100
101    String flightplanName = line;
102
103    // The next non-empty lines contain (lon,lat,height) pairs, separated by a space.
104
105    ArrayList<TLcdLonLatHeightPoint> points = new ArrayList<TLcdLonLatHeightPoint>();
106    while ((line = aInputStream.readLine()) != null &&
107           line.trim().length() > 0) {
108        String[] splitString = line.trim().split(" ");
109        if (splitString.length != 3) {
110            throw new IOException("Expected <lon lat height>, but found <" + line + ">");
111        }
112        try {
113            double lon = Double.parseDouble(splitString[0]);
114            double lat = Double.parseDouble(splitString[1]);
115            double height = Double.parseDouble(splitString[2]);
116            if (lon < -180 || lon > 180 || lat < -90 || lat > 90) {
117                throw new NumberFormatException("The longitude and latitude must be in the interval
118                                         " +
119                                         "[ -180, 180 ] and [ -90, 90 ], respectively");
120            }
121            points.add(new TLcdLonLatHeightPoint(lon, lat, height));
122        } catch (NumberFormatException ex) {
123            IOException io = new IOException();
124            io.initCause(ex);
125            throw io;
126        }
127    }
128    ILcd3DEditablePoint[] pointArray = new ILcd3DEditablePoint[points.size()];
129    ILcd3DEditablePointList pointList = new TLcd3DEditablePointList(points.toArray(pointArray)
130
131
132    ILcdDataObject plan = new TLcdDataObject(FlightPlanDataTypes.FLIGHT_PLAN_DATA_TYPE);
133    plan.setValue(FlightPlanDataTypes.NAME, flightplanName);
134    plan.setValue(FlightPlanDataTypes.POLYLINE, new TLcdLonLatPolyline(pointList));
135    return plan;
136}
137

```

Program 20 - Creating a FlightPlanModelDecoder
 (from samples/gxy/fundamentals/step2/FlightPlanModelDecoder)

The FlightPlanModelDecoder reads the flight plan records (name and coordinates) one by one from the custom file and adds them to the defined model. Since no events are needed at this stage, ILcdFireEventMode.NO_EVENT is used to make sure that no events are fired. For more information on events, refer to Section 8.3.1.

5.2 Visualizing the business data in a Lightspeed view

Once you have defined a view as described in Section 3.1, you need to create a layer for your business data and add this to the view.

5.2.1 Creating a layer for the business data

To visualize the data in the flight plan model created by the `FlightPlanModelDecoder`, the view needs to create a layer for that model. As discussed in [Chapter 3](#), this is done by setting a layer factory on the view. [Program 21](#) shows you how this is done. A `TLspCompositeLayerFactory` is used to combine the behavior of both the `FlightPlanLayerFactory` and the `BasicLayerFactory` we had created earlier. As such, both background and flight plan models can be added to the view.

```

1  // Create a layer factory that composes both the flight plan layer factory and the
2  // basic implementation (that supports shp and rasters)
3  return new TLspCompositeLayerFactory(
4      new FlightPlanLayerFactory(), new BasicLayerFactory());

```

Program 21 - Creating the flight plan layer factory
 (from samples/lightspeed/fundamentals/step2/Main)

The used `FlightPlanLayerFactory` is shown in [Program 22](#). Note that the layer factory only creates a flight plan layer if the data model of the domain objects is of the flight plan type. The `ILcdDataModelDescriptor` is used to retrieve the data model, as described in [Section 5.1.3](#).

```

1 /**
2  * Factory to create layers for models that contain flight plan data.
3  */
4 public class FlightPlanLayerFactory extends ALspSingleLayerFactory {
5
6     @Override
7     public boolean canCreateLayers(ILcdModel aModel) {
8         ILcdModelDescriptor md = aModel.getModelDescriptor();
9         return md instanceof ILcdDataModelDescriptor &&
10            ((ILcdDataModelDescriptor) md).getDataModel().equals(
11                FlightPlanDataTypes.getDataModel());
12    }
13
14    @Override
15    public ILspLayer createLayer(ILcdModel aModel) {
16        if (!canCreateLayers(aModel)) {
17            return null;
18        }
19
20        // Create a TLspLayer with the given model.
21        TLspShapeLayerBuilder layerBuilder = TLspShapeLayerBuilder.newBuilder();
22        layerBuilder.model(aModel);
23        return layerBuilder.build();
24    }
25
26 }

```

Program 22 - The layer factory used to create layers for models with flight plan objects
 (from samples/lightspeed/fundamentals/step2/FlightPlanLayerFactory)

The flight plans contain regular polylines. To display them, you need to pass the flight plan model to a shape layer builder, and build the layer.

To tune the flight plan styling, a two-pixel yellow line is configured for the regular state. In 3D, the flight plans are configured to be painted `ABOVE_ELLIPSOID`. This makes sense as our flight plans contain altitude information. The selection state is not defined, and default settings are used. This is shown in [Program 23](#).

```

1   layerBuilder.bodyStyler(TLspPaintState.REGULAR,
2     // Use yellow lines, drape them onto the terrain
3     TLspLineStyle.newBuilder()
4       .color(Color.YELLOW)
5       .width(2)
6       .elevationMode(ElevationMode.ABOVE_ELLIPSOID)
7       .build()
8 );

```

Program 23 - Defining the style for the flight plans
 (from samples/lightspeed/fundamentals/step2/FlightPlanLayerFactory)

To show the flight plan names on the map as textual labels, the label style is configured in [Program 24](#). A TLspTextStyle is used, so that font and colors can be changed. Secondly, a TLspDataObjectLabelTextProviderStyle is created, which retrieves the NAME property of the flight plans as label content. Please refer to the reference documentation of the labelStyles method for other useful styles.

```

1   layerBuilder.labelStyles(TLspPaintState.REGULAR,
2     // Use yellow text with a black outline
3     TLspTextStyle.newBuilder()
4       .textColor(Color.YELLOW)
5       .haloColor(Color.BLACK)
6       .build(),
7     // Use the flight plan name as the label content
8     TLspDataObjectLabelTextProviderStyle.newBuilder()
9       .expressions(
10         FlightPlanDataTypes.NAME)
11       .build()

```

Program 24 - Configuring the flight plan labels
 (from samples/lightspeed/fundamentals/step2/FlightPlanLayerFactory)

Many more configuration settings are available for labeling objects, please refer to [Chapter 23](#) for more information.

[Program 25](#) shows how to decode the flight plan model and add it to the view.

```

1 // FlightPlanModelDecoder can read the custom file format
2 ILcdModelDecoder decoder = new FlightPlanModelDecoder();
3
4 // Decode custom file to create an ILcdModel for flight plans
5 ILcdModel flightPlanModel = decoder.decode("Data/Custom1/custom.cfp");
6
7 // Calling addLayer() will cause the view to invoke its layer factory with
8 // the given model and then add the resulting layer to itself
9 Collection<ILspLayer> flightPlanLayer = aView.addLayersFor(flightPlanModel);

```

Program 25 - Decoding the flight plan model and adding it to the view
 (from samples/lightspeed/fundamentals/step2/Main)

5.2.2 Fitting the view to the business data layer

The background layer covers an area that is larger than that of the flight plans. To fit the view to the area of the flight plans, use `TLspViewNavigationUtil.fit` as shown in [Program 26](#). The method `fit` rescales and pans the view so that all objects of the flight plan layer are visible in the view. Basic exception handling is provided, using message dialogs.

```

1  protected void fitViewExtents(ILspView aView, Collection<ILspLayer> aLayers) {
2      try {
3          // Fit the view to the relevant layers.
4          new TLspViewNavigationUtil(aView).fit(aLayers);
5      } catch (TLcdOutOfBoundsException e) {
6          JOptionPane.showMessageDialog(fFrame,
7              "Could not fit on layer, layer is outside the valid bounds
8              ");
9      } catch (TLcdNoBoundsException e) {
10         JOptionPane.showMessageDialog(fFrame,
11             "Could not fit on destination, no valid bounds found");
12     }
13 }
```

Program 26 - Fitting the view on the given layers
 (from samples/lightspeed/fundamentals/step1/Main)

5.2.3 Updating the basic application

The business data sample shows the basic application after it has been updated with the functionality for loading and visualizing the flight plans. The main change to the basic application is the definition of a model for the background data and a model for the flight plans, and the modification of the existing code for this change.

Figure 16 shows the loaded flight plans on top of the background layer in the window of the basic application as built in Chapter 3.

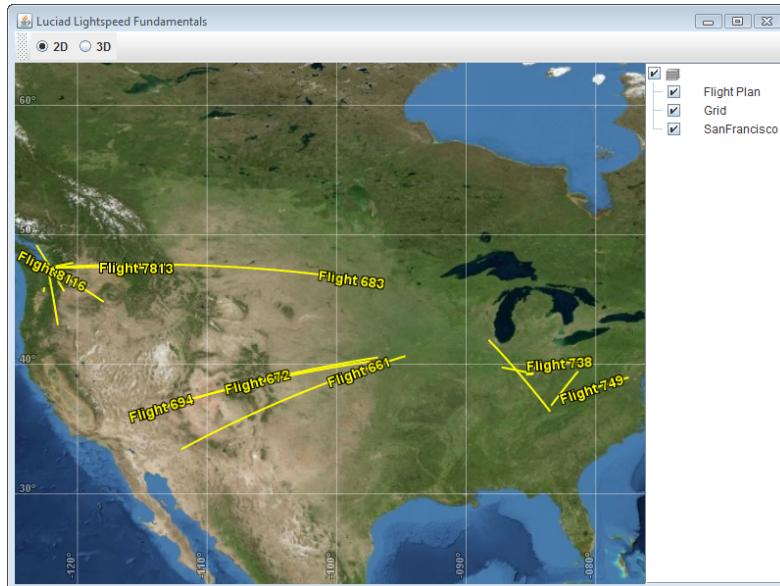


Figure 16 - The flight plans visualized in the window of the basic application

5.3 Visualizing the business data in a GXY view

Once you have defined a view as described in Section 4.2.1, you only need to create a layer for your business data and add this to the view as described in the sections below.

5.3.1 Creating a layer for the business data

To visualize the data in the flight plan model that is created by the `FlightPlanModelDecoder` that was discussed in the previous sections, the view needs to create a layer for that model. Similar to the creation of a layer for the background data as described in [Section 4.2.3](#), the layer of the business data is created using a layer factory. [Program 27](#) shows how you can create and set up a `FlightPlanLayerFactory` to create layers for the flight plan model. Note that the layer factory only creates a layer if the data model of the domain objects is of the flight plan type. The `ILcdDataModelDescriptor` is used to retrieve the data model as described in [Section 5.1.3](#).

```

1 /**
2  * Factory to create layers for models that contain flight plan data.
3 */
4
5 public class FlightPlanLayerFactory implements ILcdGXYLayerFactory {
6
7     private boolean isFlightPlanModel(ILcdModel aModel) {
8         ILcdModelDescriptor md = aModel.getModelDescriptor();
9         return md instanceof ILcdDataModelDescriptor &&
10            ((ILcdDataModelDescriptor) md).getDataModel().equals(
11                FlightPlanDataTypes.getDataModel());
12    }
13
14    @Override
15    public ILcdGXYLayer createGXYLayer(ILcdModel aModel) {
16        if (!isFlightPlanModel(aModel)) {
17            return null;
18        }
19
20        TLcdGXYLayer layer = new TLcdGXYLayer(aModel);
21
22        // Sets a pen on the layer.
23
24        layer.setGXYPen(ALcdGXYPen.create(aModel.getModelReference()));
25
26        // Creates a painter to display the flight plans.
27
28        TLcdGXYShapePainter painter = new TLcdGXYShapePainter();
29        painter.setLineStyle(TLcdStrokeLineStyle.newBuilder().
30            color(Color.ORANGE).selectionColor(Color.YELLOW).build());
31        layer.setGXYPainterProvider(painter);
32
33        // Creates a label painter that displays the first feature of each model element.
34
35        TLcdGXYDataObjectPolylineLabelPainter labelPainter = new
36            TLcdGXYDataObjectPolylineLabelPainter();
37        labelPainter.setExpressions(FlightPlanDataTypes.NAME);
38        labelPainter.setForeground(Color.WHITE);
39        labelPainter.setHaloEnabled(true);
40        labelPainter.setHaloColor(Color.BLACK);
41
42        layer.setGXYLabelPainterProvider(labelPainter);
43        layer.setLabeled(true);
44
45        return layer;
46    }
}

```

Program 27 - Creating and setting up a `FlightPlanLayerFactory`
 (from `samples/gxy/fundamentals/step2/FlightPlanLayerFactory`)

To display the flight plans you need two types of painters:

- a `TLcdGXYPointListPainter` for the flight plan geometry (which is a set of `TLcdLonLatPoint` objects). [Section 32.1.4](#) describes in detail how to use a `TLcdGXY-`

`PointListPainter` and how to specify the line color and other rendering options.

- a `TLcdGXYDataObjectPolylineLabelPainter` for the labels (in this case the names) of the flight plans. This is shown in [Program 24](#). `TLcdGXYDataObjectPolylineLabelPainter` is an implementation of `ILcdGXYLabelPainter` and an extension of `TLcdGXYPolylineLabelPainter`. The painter retrieves the label content (the name property) as defined by the `setExpressions` method of the `ILcdDataObject` interface. [Chapter 36](#) provides more information on labeling domain objects.

For more details on the interfaces and classes used in [Program 27](#), refer to the API reference.

[Program 28](#) shows how to create the flight plan layer using the `FlightPlanLayerFactory`.

```
1 ILcdGXYLayer flightplanLayer = new FlightPlanLayerFactory().createGXYLayer(flightplanModel
);
```

**Program 28 - Creating a layer for the flight plan model using the
FlightPlanLayerFactory**
(from samples/gxy/fundamentals/step2/Main)

As shown in [Program 305](#), you have now defined two layer factories. One for the background layer and one for the flight plan layer. You can also create one layer factory for the two types of layers.

5.3.2 Adding the business data layer to the view

You can simply add the layer for the flight plans to the view as shown in [Program 29](#).

```
1 aView.addGXYLayer(flightplanLayer);
```

Program 29 - Adding the flight plan layer to the view
(from samples/gxy/fundamentals/step2/Main)

5.3.3 Fitting the view to the business data layer

The background layer covers an area that is larger than that of the flight plans. To fit the view to the area of the flight plans, use `TLcdGXYViewFitAction`. The method `fitGXYLayer` rescales and pans the view so that all objects of the flight plan layer are visible in the view. [Program 30](#) shows how to fit the view to the flight plan layer.

```
1 TLcdGXYViewFitAction fitAction = new TLcdGXYViewFitAction();
2 fitAction.fitGXYLayer(flightplanLayer, aView);
```

Program 30 - Fitting the view to the flight plan layer
(from samples/gxy/fundamentals/step2/Main)

5.3.4 Updating the basic application

[Program 305](#) shows the basic application updated with the functionality for loading and visualizing the flight plans. The main change to the basic application is the definition of a model for the background data and a model for the flight plans and adapting the existing code to this change.

Figure 17 shows the loaded flight plans on top of the background layer in the window of the basic application as built in Chapter 4.

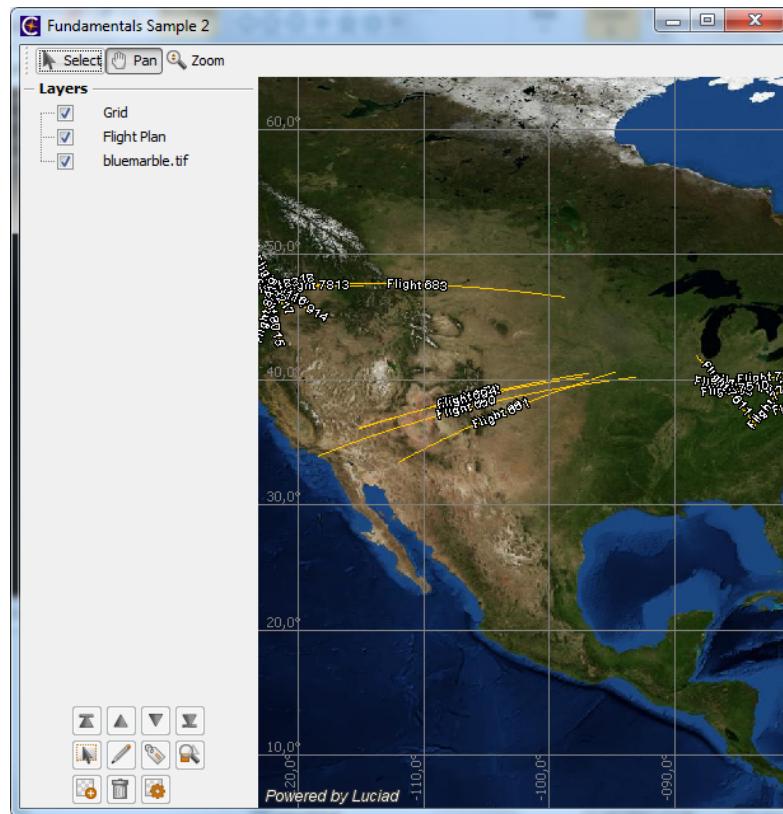


Figure 17 - The flight plans visualized in the window of the basic application

CHAPTER 6

Working with layers

The previous chapters describe how to create multiple layers in a basic application, and add these to a view.

This chapter provides more details about working with layers in LuciadLightspeed, and explains how to:

- Access layers added to the view
- Group layers into nodes
- Manage layers in layer hierarchies

To illustrate how to access and organize layers, a layer with way points is created in the fundamental LuciadLightspeed samples, and combined with the flight plan layer created in [Chapter 5](#).

[Program 302](#) is a combination of the applications built in [Chapter 3](#) and [Chapter 5](#), and the code to add and visualize the way points. [Figure 18](#) shows the window with the loaded background data, flight plans, and the way points.

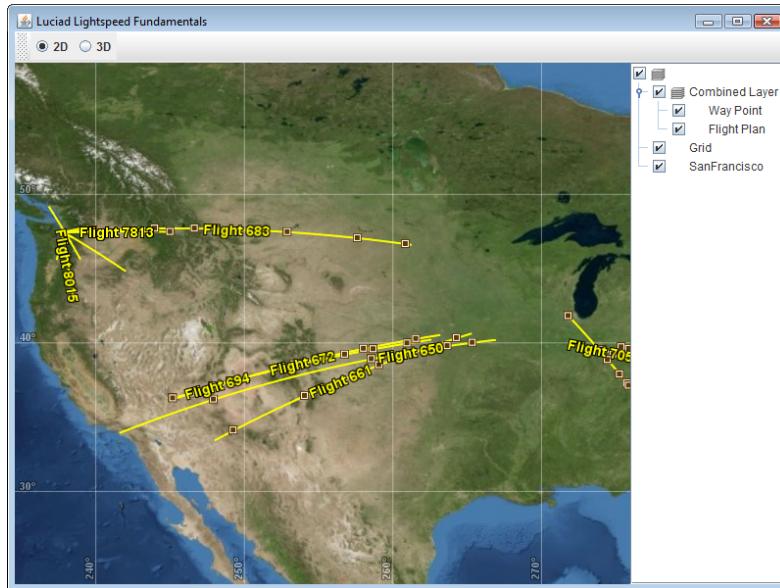


Figure 18 - The way points visualized together with the flight plans in a Lightspeed view

6.1 Accessing layers in a view

Both `ILspView` and `ILcdGXYView` implement the `ILcdLayered` interface to support the addition of layer sequences. A layer list of the type `ILcdLayered` is suitable for organizing layers with data sets that are not related to each other and that you want to access sequentially, like the layers created in [Chapter 4](#) and [Chapter 5](#). These are shown in [Figure 19](#). The numbers refer to the index numbers of the layers in the layer list.

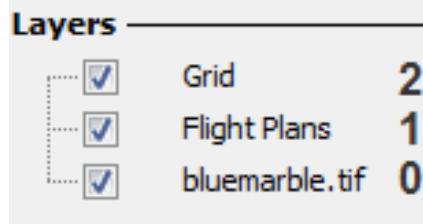


Figure 19 - A sequential list of layers

Each layer in a view has an index number that ranges from 0 to the total number of layers in the list minus one (`layerCount () - 1`). The layers are ordered by their index number, and painted in the view in the order of addition. The top layer, identified with `layerCount () - 1`, is the layer that was added to the view last, and is painted last in the view.



To organize your layers thematically, you can add related layers to a layer node and create a hierarchical layer structure, as explained in [Section 6.2](#).

6.2 Grouping layers

If you have layers containing data sets that logically belong together, you can group those layers, as illustrated by step 3 of the GXY and Lightspeed fundamentals samples. These samples extend the applications built in [Chapter 5](#) with the code to add and visualize way points. A new layer with way points is created, and combined with the flight plan layer created in [Chapter 5](#).

6.2.1 Using `ILcdLayerTreeNode` to group layers

For the purpose of grouping layers, LuciadLightspeed provides the interface `ILcdLayerTreeNode`. It extends both the `ILcdLayer` and `ILcdLayered` interfaces, as shown in [Figure 20](#).

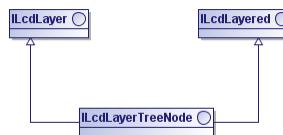


Figure 20 - `ILcdLayerTreeNode` allows you to group layers

`ILcdLayerTreeNode` allows you to create a layer node. A layer node acts as a layer by itself, but also lets you add other layers and layer nodes as child layers.

To illustrate the usage of `ILcdLayerTreeNode`, the flight plan layer created in [Chapter 5](#) is combined with a new layer for a set of way points that are loaded from an external file, as shown in the layers sample of the fundamental samples.

Program 31 shows how to create an empty layer node, the `combinedLayer`, for a Lightspeed view. Next, the existing flight plan layer and the new way point layer are added as child layers. Once you have created a layer node and added the child layers, you can add the layer node to the view. The child layers are automatically added to the view as well, because they are part of the layer node. Similarly, if you remove a layer node from a layer list or a view, the child layers will be removed as well.



An `ILcdLayerTreeNode` is empty by default. To associate an `ILcdLayerTreeNode` with a model, use the method `setModel`. In practice, most layer nodes with child layers are themselves empty, though.

```

1 // Create a combined layer that holds both the waypoint- and the flight plan layer
2 TLspLayerTreeNode combinedLayer = new TLspLayerTreeNode("Combined Layer");
3 combinedLayer.addLayers(flightPlanLayer);
4 combinedLayer.addLayers(waypointLayer);
5
6 // Add the combined layer to the view
7 aView.addLayer(combinedLayer);

```

Program 31 - Create a layer node with the flight plan layer and the waypoint layer
(from samples/lightspeed/fundamentals/step3/Main)

Figure 21 shows the layer group with the flight plans and way points grouped in a layer node with the Combined layer name. It also shows the layer index numbers returned when the layers in the view are enumerated.

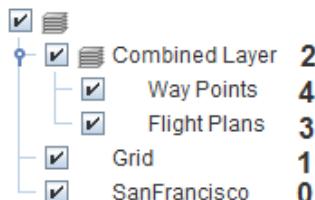


Figure 21 - The Combined layer with the layers for the flight plans and way points

Grouping layers in GXY views is very similar to layer grouping in Lightspeed views. Program 32 shows how to create a layer node in a GXY view, and add the layers for the flight plans and the way points as child layers, resulting in the Combined layer.

```

1 TLcdGXYLayerTreeNode combinedLayer = new TLcdGXYLayerTreeNode("Combined layer");
2 combinedLayer.addLayer(flightplanLayer);
3 combinedLayer.addLayer(waypointLayer);

```

Program 32 - Create a layer node with the flight plan layer and the waypoint layer in a GXY view
(from samples/gxy/fundamentals/step3/Main)



Each `ILcdLayer` and `ILcdLayerTreeNode` may appear only once in one hierarchical layer structure.

Keep in mind that all methods inherited from `ILcdLayer` only apply to the layer node itself, as a layer. They do not affect the child layers of the layer node. For example, calling `ILcdLayer.setVisible(boolean)` for a layer node only changes the visibility of the layer node and not of its child layers. On the other hand, all methods inherited from `ILcdLayered` just apply to the child layers, and not to the node. For example, the `ILcdLayered` method `layerCount()` returns the number of child layers of a given layer node.

6.3 Working with layers in a tree structure

Section 6.1 explained how to access layers sequentially. If you have grouped your layers into nodes, however, you may need to access the layers in your view as a hierarchy. To this end, all implementations of `ILspView` and `ILcdGXYView` extend both `ILcdLayered` and `ILcdTreeLayered`.

These allow you to approach the layers in a view in two manners. `ILcdLayered` allows you to access all layers in the view regardless of their position in the layer hierarchy. `ILcdTreeLayered` provides access to the layers through an additional root node: a layer node that is inserted at the top of the layer hierarchy, and provides a gateway to the layer tree structure through its child layers.

The following sections use the view layer structure in Figure 22 as an example to provide more information about each approach.

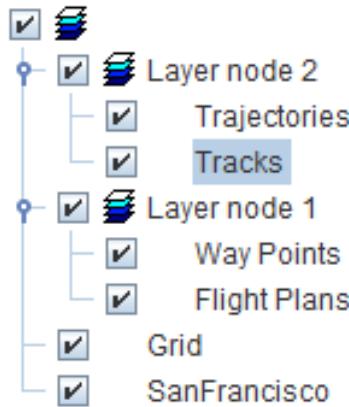


Figure 22 - An example of a view with grouped layers

6.3.1 Approaching the view as a flat list of layers

A view allows you to perform all the operations defined in the `ILcdLayered` interface on the hierarchical structure as if it were a flat list, with the layers listed in their painting order. If you call the `layers` method on the view, it returns such a flat layer list. For example, Figure 23 displays the layers of Figure 22 as a flat list. Calling the `layers` method results in the layers Grid, San Francisco, Layer node 1, Way Points, Flight Plans, Layer node 2, Trajectories and Tracks.

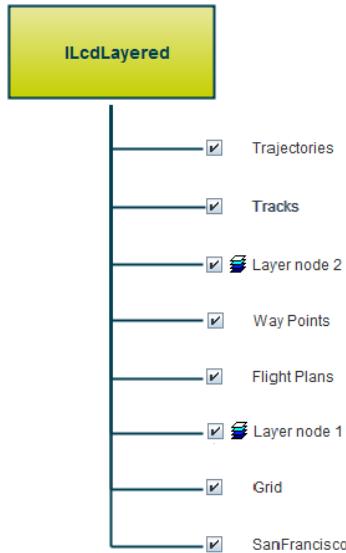


Figure 23 - The view approached as a flat list

You can perform the same operations on the layers in the layer tree as on the layer list, such as enumeration and reordering. For more information about layer reordering, see [Section 6.4](#).

If you perform an operation on the layers in the flat list, it will have an effect on the layer tree. If a layer is removed from the flat list presentation, for example, it will be removed from the hierarchical layer structure as well. This works both ways: if a layer is removed from the hierarchical structure, for example, it will also be removed from the flat list representation.



Be careful when copying all layers from one view to another. For more information, refer to [Section 31.1](#).

6.3.2 Approaching the view as a layer tree

You can approach a view as a layer tree by exposing a root node for the tree. To access the root node, call the method `get rootNode`, and access all tree levels, including layer nodes and layer descendants, from there. If you call the `layers` method on the root node, it returns its child layers.

Figure 24 illustrates the approach of the layers in [Figure 22](#) through a root node. Calling the `layers` method on the root node results in the layers Grid, San Francisco, Layer node 1, and Layer node 2.

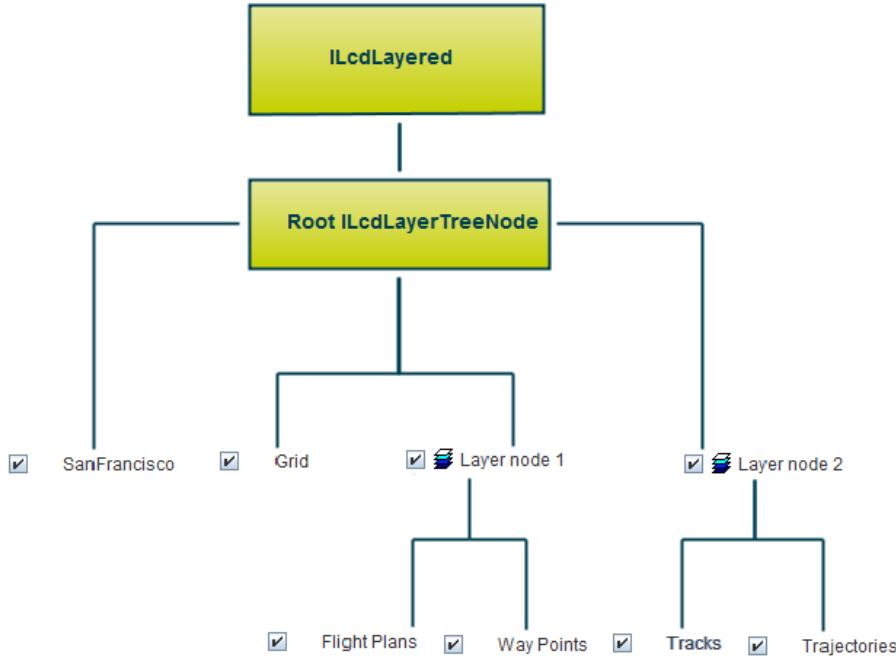


Figure 24 - An example of view layers approached from the root node

The root node of a view is not considered a regular layer node. This means that you cannot retrieve it using `containsLayer`. You cannot move or remove it either.

Approaching all descendants of a layer node

By default, calling `layers` on a layer node, such as the root node, results in its direct descendants only. The returned list only includes the child layers of the node. To work with all descendants of a layer node, you can make use of the utility class `TLcdLayerTreeNodeUtil`. It provides methods to access all layers in a layer node, and retrieve the hierarchical relations between the grouped layers.

For more information, see the API reference documentation.

6.4 Layer ordering

6.4.1 How are layers ordered?

In general, the order of addition determines the layer order in a view. The first layer is assigned layer index zero in an `ILcdLayered` list, is painted first and ends up at the bottom of the layer stack. Each extra layer is added to the layer stack, and painted on top of the previous layer.

In Lightspeed views, you can use `ILspPaintingOrder` to further set up layer ordering. You can set its default implementation `TLspPaintingOrder` on the `ILspView` to ensure that layers are painted according to their indexing order, but that selected objects and labels are painted on top.

Within layer nodes, child layers are added in the same order as in the view. You can influence the layer order in nodes using `ILcdInitialLayerIndexProvider`. Each `ILcdLayerTreeNode`, including the root node of a layer tree, can have such an `ILcdInitialLayerIndexProvider`. It determines the appropriate index for a new layer, if it is added without a specific index. This allows you to centralize the layer ordering logic in the node, and base it on the type

of data that is loaded. For example, you could set up an `ILcdInitialLayerIndexProvider` that adds vector data on top of typical background data like raster data.

Lightspeed views work with a specific implementation of this interface, `TLspInitialLayerIndexProvider`. It offers automatic correct layer placement based on the type of the layer: For instance, BACKGROUND layers are moved as far as possible towards the bottom, while INTERACTIVE layers are moved as far as possible to the top.

6.4.2 Moving layers up and down

Moving tree layers in a LuciadLightspeed application can mean two things:

- Move layers up and down in the layer painting order. To move a layer up in the painting order, you move it to a higher index, closer to the top of the list order. As a result, the moved layer may end up on top of other layers, and could cover up those layers.
- Move layers up and down in GUI widgets displaying the layer tree.

LuciadLightspeed allows you to move layers up and down in the GUI, and change the painting ordering at the same time. This also works the other way around: if you change the painting order, the layers will also move up and down in the GUI.

In a GXY view, you can also choose to disconnect the painting order and GUI layer order, so that one does not influence the other. See [Independently ordering layers in a GXY view](#) for more information.

Moving a layer to a different index

To change the painting order of one layer in an `ILcdLayered` list, you can move it to another index using the method `moveLayerAt`. [Program 33](#) shows how to move a layer of the view to the top of the layer list so that it is displayed on top of the other layers in the view.

```
1 ILspLayer layer = view.addLayersFor( model );
2 view.moveLayerAt( view.layerCount() - 1, layer );
```

Program 33 - Moving a layer to the top of the layer list in a Lightspeed view

[Program 15](#) shows how to move the grid layer of `TLcdMapJPanel` to the top of the layer list in a GXY view, so that it is displayed above the other layers in the view.

When a layer node is moved, the child layers also move by default.



All actions that rely on the painting order of the layers, such as selecting objects in a layer, should use the `layers` method of the view, and not the layer tree structure.

Independently ordering layers in a GXY view

In the event that you want to move layers around in the layer tree of a GXY view without affecting the painting order, you can pass a `TLcdIndependentOrderTreeLayeredSupport` object to the view constructor when you create the view. You can also use `TLcdIndependentOrderTreeLayeredSupport` to move layers around in the painting order, without affecting the layer order in the hierarchical tree. Moving a layer node in the layer list in independent order only moves the node and not its child layers.

6.5 Selecting the objects in a layer

To select individual domain objects from the model that is wrapped in a layer, LuciadLightspeed provides the interface `ILcdSelection`. An `ILcdSelection` basically is a container for one or more selected objects. `ILspLayer` and `ILcdGXYLayer` extend `ILcdSelection` and provide methods to:

- Select objects in a layer: `selectObject`
- List the selected objects in a layer: `selectedObjects`, this method returns a `java.util.Enumeration`



To list the selected objects in a view, use the method `selectedObjects` for all `ILcdLayer` objects in the `ILcdLayered`.

In addition, `ILcdSelection` provides methods to add and remove selection listeners that notify registered objects of a change in the selection. For more information refer to [Section 8.2.1](#).

CHAPTER 7

Making objects selectable and editable

So far, we have built a simple but usable application for viewing business data in 2D and 3D. Most substantial LuciadLightspeed applications, however, require interaction with the data. In this chapter, we make objects on the map selectable and editable. These are the two fundamental features that allow users to inspect and manipulate the data in an `ILcdModel`. We need to perform two simple steps: slightly adjust the configuration of the layer, and expand our controller.

7.1 Making objects selectable and editable in a Lightspeed view

7.1.1 Adjusting a layer to make model data selectable and editable

Program 34 shows the required changes to the layer configuration to make data selectable and editable. To allow for the selection and editing of layer objects, we need to indicate to the layer builder that it needs to construct a selectable and editable layer. As layer type we use the `EDITABLE` layer type.

```
1  TLspShapeLayerBuilder layerBuilder = TLspShapeLayerBuilder.newBuilder()
2      .model(aModel)
3      .layerType(ILspLayer.LayerType.EDITABLE)
4      .selectable(true)
5      .bodyEditable(true);
```

Program 34 - Making the objects on a layer selectable and editable

(from

`samples/lightspeed/fundamentals/step4/EditableWayPointLayerFactory`)

7.1.2 Adding undo and redo support

The next step towards enabling selection and editing is to adapt our controller. However, because we want to add undo and redo support, we need to create a `TLcdUndoManager` first, as shown in Program 35. The undo manager keeps track of operations that have been performed by the user, and provides the bulk of the undo and redo functionality. The undo manager needs to be associated with the controller, so that the controller can register the user's actions with it.

```

1 // Used by the edit controller to track undoable operations.
2 private TLcdUndoManager fUndoManager = new TLcdUndoManager();

```

Program 35 - Creating a manager for undo/redo support
 (from samples/lightspeed/fundamentals/step4/Main)

At this point, we create the toolbar buttons that allow users to perform the actual undo and redo operations. [Program 36](#) shows how the buttons are added to the toolbar. `TLcdUndoAction` and `TLcdRedoAction` are actions that invoke the corresponding operations on the `TLcdUndoManager`. `TLcdSWAction` is an adapter class that turns the undo and redo actions into a `javax.swing.Action` so that they can be added to the toolbar.

```

1 // Add buttons for undo and redo
2 TLcdUndoAction undo = new TLcdUndoAction(fUndoManager);
3 toolBar.add(new TLcdSWAction(undo));
4 TLcdRedoAction redo = new TLcdRedoAction(fUndoManager);
5 toolBar.add(new TLcdSWAction(redo));

```

Program 36 - Adding undo/redo buttons to the toolbar
 (from samples/lightspeed/fundamentals/step4/Main)

7.1.3 Adding selection and editing support to the controller

The final step consists of adding selection and editing support. We can do this as before by using the `ControllerFactory` to add a general controller that chains navigation, selection and edit support together. [Program 37](#) shows how to add a general controller.

```

1 // We use a utility method of the samples to assign the undo manager to the edit
   controller.
2 // The controller also allows to select on the map, and to navigate around.
3 // Buttons for undo/redo are added to the toolbar in the createToolBar() method.
4 ILspController c = ControllerFactory.createGeneralController(fUndoManager, aView);
5
6 // Assign the controller to the view
7 aView.setController(c);

```

Program 37 - Adding controllers for selection and editing
 (from samples/lightspeed/fundamentals/step4/Main)

For more information about controller chaining, see [Section 26.2.1](#).

Note that the behavior defined here is identical to the default behavior, when the controller is not changed at all, except that the undo manager is now used.

7.1.4 Selecting and editing shapes in the application

When you run the application now, you can click a way point to select it. Simultaneously, the editor for the object is activated, meaning that you can now modify its geometry. To move a way point, click and drag it around. At any point, you can use the undo and redo buttons to move back and forth in our history of edit operations.



The undo manager has a default undo limit of 10 steps, but higher values can be passed to the constructor of `TLcdUndoManager`.

7.2 Making objects selectable and editable in a GXY view

7.2.1 Adjusting a layer to make model data selectable and editable

Program 38 shows the required changes to the layer configuration to make data selectable and editable. To allow for the selection and editing of layer objects, we need to provide a suitable editor and indicate to the layer that it needs to be selectable and editable.

```

1 // Creates a painter to display the way points.
2 TLcdGXYShapePainter painter = new TLcdGXYShapePainter();
3 layer.setGXYPainterProvider(painter);
4
5 // Also configure the painter as editor so we can move way points around.
6 layer.setGXYEditorProvider(painter);
7
8 layer.setSelectable(true);
9 layer.setEditable(true);

```

Program 38 - Making the objects on a layer selectable and editable
 (from samples/gxy/fundamentals/step4/EditableWayPointLayerFactory)

7.2.2 Adding selection and editing support to the controller

The final step consists of adding selection and editing support. We can do this by using the `TLcdGXYEditController2`. Program 39 shows how to add pan and zoom behavior.

```

1 TLcdGXYEditController2 editController = new TLcdGXYEditController2();
2 editController.setAWTFilter(TLcdAWTEventFilterBuilder.newBuilder().leftMouseButton().build
3 ());
4
5 TLcdGXYPanController panController = new TLcdGXYPanController();
6 panController.setAWTFilter(TLcdAWTEventFilterBuilder.newBuilder().middleMouseButton().
7 build());
8 panController.setCursor(editController.getCursor());
9 panController.setDragViewOnPan(true);
10
11 TLcdGXYZoomWheelController zoomWheelController = new TLcdGXYZoomWheelController();
12
13 TLcdGXYCompositeController compositeController = new TLcdGXYCompositeController();
14 compositeController.addGXYController(editController);
15 compositeController.addGXYController(panController);
16 compositeController.addGXYController(zoomWheelController);
aView.setGXYController(compositeController);

```

Program 39 - Adding a controller for selection and editing
 (from samples/gxy/fundamentals/step4/Main)

7.2.3 Selecting and editing shapes in the application

When you run the application now, you can click a way point to select it. Simultaneously, the editor for the object is activated, meaning that you can now modify its geometry. To move a way point, click and drag it around.



Chapter [Section 37.4](#) describes how to add undo/redo behavior to your GXY application.

CHAPTER 8

Handling changes

LuciadLightspeed application objects can change during their lifetime. Not only does the object itself change, in most cases the change also affects other application objects. This chapter describes how application objects can change, how LuciadLightspeed handles these changes automatically, and in which case the program needs to take care of these changes. Finally, the LuciadLightspeed multi-threading fundamentals to handle those changes are discussed.

8.1 Modifying application objects

A LuciadLightspeed application object can change in many ways. Model data, for example, can change directly when the model receives its data from a real-time data stream through a radar feed. In order to visualize the updated model data in the view, the associated layer objects, painter objects, and view objects need to be updated as well. Model data can also change indirectly, for example when a user interacts with the view and moves objects or changes their shape. To visualize the changed objects and save the changes to the model, the model data and its associated layer, painter, and view objects need to change accordingly.

Changing application objects is typically a three-step process:

1. **Change:** An action object calls a method on an application object and changes that object.
2. **Notification:** The application objects that are interested in the changed object are notified of the change. There are two notification mechanisms as described in [Section 8.1.1](#).
3. **Action:** Each notified object optionally performs an action as a result of the change. This action itself can change another application object, making this the first step of a new change.

8.1.1 Notifying objects of changes

There are two ways to notify application objects of changes in another application object. The applicable notification mechanism depends on the ability of the changed object to register listeners to it. A listener is an application object that registers to another application object to get notified of a change in the object it has been registered to. The two notification mechanisms are:

1. **Notifying objects of changes with listeners.** In this case, the changed object automatically fires events to notify its listeners of the change. This mechanism is based on the so called **Observer pattern** and is applied to most interfaces and classes of LuciadLightspeed . [Section 8.2](#) describes the usage of listeners in detail.

2. **Notifying objects of changes programmatically.** In this case, the object that executed the change (the action object) needs to notify other application objects of the change. The use cases for this scenario are limited and are listed in [Section 8.3](#).

8.2 Notifying objects of changes with listeners

The generally applied scenario for notifying application objects of changes in another application object, is based on the **Observer** (or Listener) pattern. According to this pattern, one or more application objects (listeners or observers) can register to a single application object (subject). When the subject changes, it automatically fires an event to its listeners with a notification of the change. The listeners itself have the ability to receive events and react to it. This mechanism is also referred to as the publish-subscribe interaction. Almost any of the most commonly used LuciadLightspeed interfaces and classes provide methods to add and remove listeners. The limited use cases in which an application object does not have the ability to register listeners are listed in [Section 8.3](#).

LuciadLightspeed provides a set of listener interfaces for the handling of changes for specific types of application objects. For example, to listen to changes in an `ILcdSelection` object, the interface `ILcdSelectionListener` is used as described in [Section 8.2.1](#). Similar interfaces are available for `ILcdModel` and `ILcdLayered` objects, and other. In many use cases LuciadLightspeed uses a Property Change Listener that listens to changes of specific properties of application objects. The following sections provide more information on selection listeners, model listeners, and Property Change Listeners.

8.2.1 Listening to changes in a selection

This section describes how to listen to changes in a selection. As described in [Section 6.5](#), an `ILcdSelection` is a container for objects that are selected from a model that is wrapped in a layer. `ILcdSelection` provides methods to add and remove `ILcdSelectionListener` objects. Whenever the selection state of one of the objects in an `ILcdSelection` changes, the selection automatically notifies its registered listeners of the change by calling the method `selectionChanged`. The parameter of `selectionChanged` is `TLcdSelectionChangedEventArgs`, which contains both the selection that fired the event as the changes in the selection. It also provides the following methods to retrieve the objects in the selection that have changed:

- `selectedElements`: returns all objects for which the selection state has changed from deselected to selected
- `deselectedElements`: returns all objects for which the selection state has changed from selected to deselected
- `elements`: returns all objects for which the selection state has changed

[Figure 25](#) shows the usage of `ILcdSelection` and `ILcdSelectionListener` in a diagram.

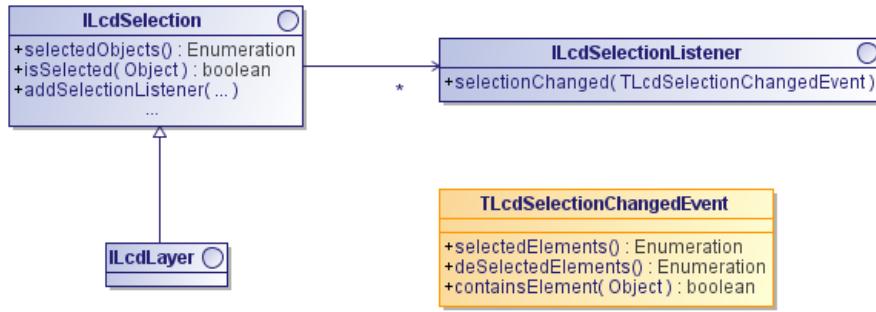


Figure 25 - Listening to changes in a selection

Additionally you can retrieve more information about the change in selection of a given object by using the method `retrieveChange`. Refer to the API reference for a detailed description of `ILcdSelection`, `ILcdSelectionListener`, and `TLcdSelectionChangedEvent`.

Program 40 shows how to create a selection listener that listens to changes in a layer (`MyLayerSelectionListener`). The layer selection listener is added to the view and listens to selection changes in each of the layers that are added to the view.

```

1 //Add a selection listener to the view that will create a balloon when a selection occurs.
2 BalloonViewSelectionListener listener = new BalloonViewSelectionListener(getView(),
3     fBalloonManager);
4 getView().addLayerSelectionListener(listener);

```

Program 40 - Listening to changes in selection of the `ILcdLayer` objects of a view
 (from samples/gxy/balloon/MainPanel)

8.2.2 Listening to changes in a model

The interface `ILcdModelListener` allows application objects to listen to changes in a model. One or more interested application objects can register to a model by using the method `ILcdModel.addModelListener(ILcdModelListener)`. Whenever an object of the model is removed, added, or changed, the registered listeners are notified of the change. Particular in this case is the fact that when an object of the model is changed, the model itself first needs to be notified of the change before it can fire an event to its listeners as described in Section 8.3.1. When an object is removed or added from a model, the model fires an event directly to its listeners since these changes are performed on the model itself.

To notify its listeners of changes, the model uses the method `modelChanged` with as parameter a `TLcdModelChangedEvent`. The `TLcdModelChangedEvent` contains details on the model change. Program 41 shows an implementation of `ILcdModelListener` (`ModelUpdateListener`) that listens to all possible changes in the flightplan model as created in Chapter 5. All events are displayed in a text box (`JTextArea`).

```

1 public class ModelUpdateListener extends JTextArea implements ILcdModelListener {
2
3     public ModelUpdateListener() {
4         setEditable(false);
5         setRows(8);
6     }
7
8     @Override
9     public void modelChanged(TLcdModelChangedEvent aEvent) {
10        if ((aEvent.getCode() & TLcdModelChangedEvent.ALL_OBJECTS_CHANGED) != 0) {
11            this.append("ALL_OBJECTS_CHANGED (" + aEvent.elementCount() + " elements)\n");
12        }
13        if ((aEvent.getCode() & TLcdModelChangedEvent.ALL_OBJECTS_REMOVED) != 0) {
14            this.append("ALL_OBJECTS_REMOVED (" + aEvent.elementCount() + " elements)\n");
15        }
16        if ((aEvent.getCode() & TLcdModelChangedEvent.OBJECTS_ADDED) != 0) {
17            this.append("OBJECTS_ADDED (" + aEvent.elementCount() + " elements)\n");
18        }
19        if ((aEvent.getCode() & TLcdModelChangedEvent.OBJECTS_CHANGED) != 0) {
20            this.append("OBJECTS_CHANGED (" + aEvent.elementCount() + " elements)\n");
21        }
22        if ((aEvent.getCode() & TLcdModelChangedEvent.SOME_OBJECTS_REMOVED) != 0) {
23            this.append("OBJECTS_REMOVED (" + aEvent.elementCount() + " elements)\n");
24        }
25
26        // Display the list of objects that have been added/removed or changed.
27        Enumeration en = aEvent.elements();
28        while (en.hasMoreElements()) {
29            this.append("\t" + en.nextElement() + "\n");
30        }
31    }
32}

```

Program 41 - An implementation of an `ILcdModelListener`
(from `samples/gxy/modelChanges/ModelUpdateListener`)

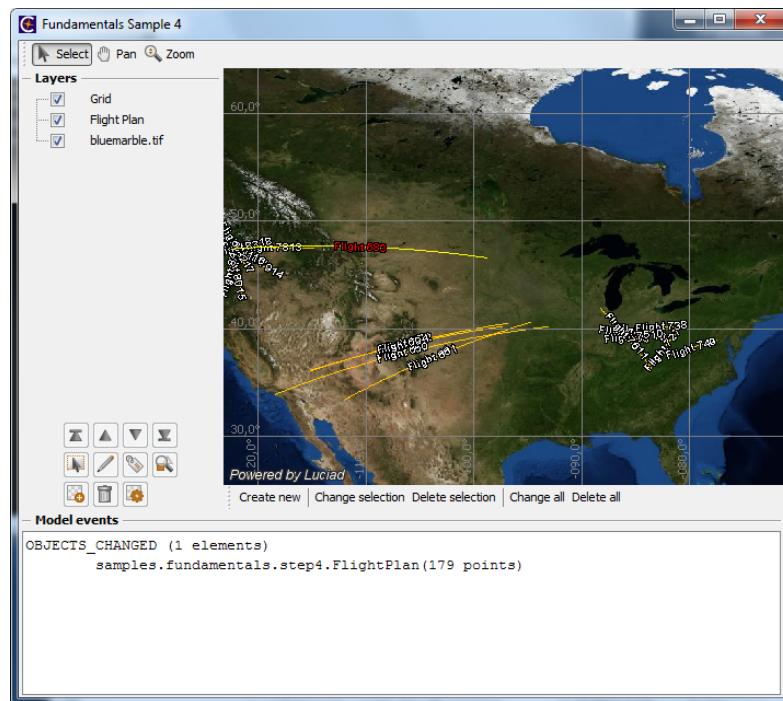


Figure 26 - The model events are shown in a text box under the view

8.2.3 Listening to property changes

A specific use case of the Observer pattern is the use of a `java.beans.PropertyChangeListener` which listens to changes in a Java bean property. This kind of property is a named attribute of a Java bean that you can read using a `get` method and that you can change using a similarly named `set` method. The name of the property is derived from the `get/set` method pair. It is the name that would be used for the matching private field, according to the Java naming conventions: drop the `get` or `set`, and start the name with lower case. For example, for the methods `getScale` and `setScale`, the returned property name is `scale` (in lower case). In the case of the methods `setGXYController` and `getGXYController`, the returned property name is `GXYController`. As there are multiple consecutive capitals in the returned name, all capitals are kept.

Whenever a property of an application object changes, the changed object notifies its `Property Change Listeners` using the `propertyChanged` method with a `PropertyChangeEvent` as parameter. Program 42 shows how to create a `Property Change Listener` (`ScalePropertyChangeListener`) that listens to changes in the return value of `ILcdGXYView.getXYWorldReference`. It thus listens to the property that is named `XYWorldReference`. The method `getNewValue` is used to retrieve the new value of the changed property which is then identical to `ILcdGXYView.getXYWorldReference`.

```

1  private static class ScalePropertyChangeListener implements PropertyChangeListener {
2
3      @Override
4      public void propertyChange(PropertyChangeEvent evt) {
5          if ("XYWorldReference".equals(evt.getPropertyName())) {
6              TLcdMapJPanel map = (TLcdMapJPanel) evt.getSource();
7              ILcdXYWorldReference oldWorldReference = (ILcdXYWorldReference) evt.getOldValue();
8              ILcdXYWorldReference newWorldReference = (ILcdXYWorldReference) evt.getNewValue();
9              if (oldWorldReference != null && newWorldReference != null) {
10                  map.setMinScale(convertScale(map.getMinScale(), oldWorldReference, newWorldReference
11                      ));
12                  map.setMaxScale(convertScale(map.getMaxScale(), oldWorldReference, newWorldReference
13                      ));
14              }
15          }
}

```

Program 42 - Using a `PropertyChangeListener`
 (from samples/gxy/common/SampleMap JPanelFactory)

Finally, the (`ScalePropertyChangeListener`) is added to the view as shown in Program 43.

```

1  map.addPropertyChangeListener(new ScalePropertyChangeListener());

```

Program 43 - Adding a `PropertyChangeListener` to the view
 (from samples/gxy/common/SampleMap JPanelFactory)

8.3 Notifying objects of changes programmatically

Because of efficiency reasons it is in some cases not feasible to register listeners and fire events for every possible change that can occur in an application object. In these cases, the application object that executes the change (the action object), needs to notify other application objects of the change. In the following common use cases a method is required to notify other application objects that a change has occurred:

- **Changing a domain object.** When changing a domain object from a model, the action object needs to notify the model of the change using the method `elementChanged`. Section 8.3.1 describes this common use case in more detail.
- **Changing indirect layer properties.** This applies to changing properties that affect a layer indirectly. In this case, the action object needs to use the method `TLcdLayer.invalidate` to notify the layer directly of the change or notify the view of the change in the layer. Section 8.3.2 describes this use case in more detail.

There are a few other, advanced use cases which are mentioned in the advanced parts of this guide. In all other cases, listeners are used as described in Section 8.2. The sections below provide more information on changing domain objects and changing indirect layer properties.

8.3.1 Changing domain objects

When a domain object changes, the model that contains the object is not aware of this change until it gets notified. To notify a model of changes in one of its domain objects, use the method `elementChanged`. The model can then, in its turn, notify registered `ILcdModelListeners` of the change by firing events.

Whenever objects are added to or removed from a model, or when a model is notified of a change in one of its objects, the firing event mode `ILcdFireEventMode` has to be given to the model as well. There are three modes for firing events:

- `ILcdFireEventMode.FIRE_NOW`: to send out the event immediately.
- `ILcdFireEventMode.FIRE_LATER`: to collect the event and send out an event later. After all changes have been made, the method `fireCollectedModelChanges` should be called on an `ILcdModel` so that the model can fire one event for the collected changes. This mode is useful in case of bulk changes to a model.
- `ILcdFireEventMode.NO_EVENT`: to not send an event at all. This mode is mostly used when adding objects to a model that has just been created.

Program 44 shows how the model gets notified that an object has changed and how the model is instructed to wait with firing an event to its listeners.

```
1     model.elementChanged(object, ILcdFireEventMode.FIRE_LATER);
2 }
```

Program 44 - Notifying a model that an object has changed and that it has to wait with firing the event

(from samples/gxy/modelChanges/ChangeSelectionAction)

8.3.2 Changing indirect layer properties



This section applies to GXY Views only.

When changing one of the layer properties on an `ILcdGXYLayer`, for example using `setVisible` or `setLabeled`, the view automatically gets notified of the change. But when you change properties that indirectly affect a layer, for example the properties of an associated painter, you need to use the `invalidate` method to notify the layer of the change. You can also notify the view of the fact that one of its layers has changed using the `invalidateGXYLayer` method.

Program 45 shows how a new `TLcdIndexColorModel` is set to the raster painter to reflect a change in color style made by the `setColorModel` method. The raster painter is responsible for painting the model that is wrapped in an `fDMEDGXYLayer`. The last line of the code snippet shows how the view gets notified that one of its layers needs a repaint because the underlying painter has changed.

```
1 ALcdImageOperatorChain operatorChain = createIndexLookupOperatorChain(colorMap);
2 raster_painter.setOperatorChain(operatorChain);
3
4 //notify the view that the fDMEDGXYLayer needs a repaint
5 getView().invalidateGXYLayer(fDMEDGXYLayer, true, this, "Changed color model");
```

Program 45 - The color model of a layer has changed and the view is notified of the change so it can repaint the layer

(from samples/gxy/colormap/MainPanel)

CHAPTER 9

Threading and locking for model and view access

LuciadLightspeed uses various threads to prepare, process and display data. This way, LuciadLightspeed can take advantage of your computer's multiple processor cores, and it can keep the UI responsive.

This chapter describes some basic threading and locking rules for LuciadLightspeed applications that you must comply with to prevent concurrency errors and ensure optimal efficiency. They apply to all operations that access LuciadLightspeed models and change LuciadLightspeed views.

Next, the chapter lists some development pointers for the use of threads and locks in LuciadLightspeed applications.



Complying with these basic rules guarantees that you will not encounter any threading issues. Under specific circumstances you can make an exception to those rules, to deal with performance issues in your application for instance. For more information, see [Chapter 56](#). However, it is strongly recommended to read the information in this chapter first, before moving on to more complex threading solutions.

9.1 Reading data in a LuciadLightspeed model

- Always take read locks, and use `TLcdLockUtil` to do so.
- You can take read locks on any thread, at any time.

```
1 try (TLcdLockUtil.Lock lock = TLcdLockUtil.readLock(model);  
2     Stream<elements> elements = model.query(all())) {  
3     elements.forEach(...)  
4 }
```

Program 46 - Taking a read lock

Operations that require a read lock include:

- Model operations like `elements` or `applyOnInteract2DBounds`
- Element operations such as looping over points of a polyline

9.2 Changing data in a LuciadLightspeed model

- Explicitly lock your models for all model writing, and use `TLcdLockUtil` to do so.
- Use the Event Dispatch Thread (EDT) for write locks.
- Fire the `TLcdModelChangedEvent` for model operations outside the `TLcdLockUtil` lock.

```

1 try (TLcdLockUtil.Lock lock = TLcdLockUtil.writeLock(model)) {
2     model.addElement(element, FIRE_LATER);
3 }
4 model.fireCollectedModelChanges();

```

Program 47 - Taking a write lock

Operations that require a write lock include:

- Model operations, like `addElement` or `elementChanged`
- Element operations, like adding points to a polyline

By applying these threading and locking rules for model changes, you prevent deadlocks in model listeners and ensure that they get events in the correct order. As a result, your model listeners can remain simple, because they do not have to take precautions for such problems.



Always call the `fireCollectedModelChanges` method as soon as you release a write lock. For instance, if you take two model write locks in a row, call `fireCollectedModelChanges` right after you release the first lock, and before you take the second lock. Next, call `fireCollectedModelChanges` again right after you release the second lock.

9.3 Changing view and layer properties

- To perform view and layer operations, use the EDT thread.

View and layer operations include:

- View changes, like map navigation, the addition and removal of layers, controller and georeference changes
- Layer changes, like changes to visibility, properties or georeferences

9.4 Development pointers for threading and locking

Detecting threading rule violations automatically Violations of the threading rules for model read and write locks can trigger subtle issues. See the reference documentation of `ALcdModel` to learn how certain types of violations can be detected automatically.

Thread-safety of LuciadLightspeed objects LuciadLightspeed objects are not thread-safe, unless their thread-safety is stated explicitly in the reference documentation.

Switching to the EDT If you need to switch to the EDT from another thread to execute an operation, use `SwingUtilities.invokeLater()` or `TLcdAWTUtil.invokeLater()` to schedule work on the EDT.

Locking for users of Java versions prior to version 1.7 Use a `try/finally` block instead of the `try-with-resources` statement used in the previous examples.

```
1 TLcdLockUtil.writeLock(model);
2 try {
3     model.addElement(element, FIRE_LATER)
4 } finally {
5     TLcdLockUtil.writeUnlock(model);
6 }
7 model.fireCollectedModelChanges();
```

Program 48 - Using a try/finally block with locks

Using a non-EDT thread for model changes You can use a thread other than the EDT only if your model is not part of a view yet, or if you need to offload the EDT for performance reasons. Chapter 56 explains this in detail.

Consider using a non-EDT thread if:

- The performance of your application is hampered by a frequently blocked EDT. Therefore, you wish to offload the EDT and move some of the work load to another thread.
- You are using offscreen views.

PART II Advanced Modeling Topics

CHAPTER 10

Unified access to domain objects

LuciadLightspeed provides a framework to access domain objects in a unified way. With this framework it is possible to access all domain objects of a LuciadLightspeed model, independent of their specific domain. Additionally, the framework provides information about the static structure of the domain objects as is typically provided by a UML diagram.



This chapter uses the term **domain model** to refer to the set of domain objects and their relationship as used in a specific domain. A LuciadLightspeed **model** refers to an `ILcdModel`, the container for a set of domain objects as described in [Section 2.2.1](#).

This chapter provides information on representing and accessing domain objects in a unified way. The framework that LuciadLightspeed provides for unified domain access differs from the previously used interfaces `ILcdFeatured` and `ILcdFeaturedDescriptor`. [Section 10.3.3](#) describes the differences between the two approaches. The remainder of this chapter is dedicated to the unified domain access which is the recommended framework to represent and access domain object.

10.1 Introducing the framework

LuciadLightspeed supports a multitude of domain models, for example AIXM51, SHP, DAFIF, and VPF. All these domain models have their own types of domain objects that represent the different concepts in their domains. LuciadLightspeed provides a framework for these specific domain objects which makes it easy to interact with the model and allows for example to retrieve the name of an airport, to create a new airspace, or to change the designator of a certain procedure.

In some cases, you might need to access the different domain models in a unified way. Consider for example the case of a label painter that is responsible for drawing a label containing the name of a domain object on the map. This label painter should work both for an AIXM51 airspace and for a DAFIF airport. For these use cases, LuciadLightspeed provides a framework that defines unified access to domain objects.

The `com.luciad.datamodel` package defines a common **meta model** that describes the static structure of the different domain models. The meta model structures domain objects using types, each with a different set of properties. `TLcdDataType` and `TLcdDataProperty` explicitly represent types and properties as objects at runtime, which makes the static structure of a domain model visible. The package also defines the `ILcdDataObject` interface that enables unified

access to domain objects.

The `java.lang.reflect` package provides similar introspection capabilities for Java classes and Java objects. One advantage of the `com.luciad.datamodel` package is that you can use it in situations where new data models are created at run-time. In such cases, different types often share the same instance class (for example `TLcdDataObject`). You will not be able to extract all necessary type information using Java reflection. Also, the `com.luciad.datamodel` package provides these capabilities at a higher level of abstraction, much closer to the actual data model.



A model is an abstraction of objects in the real world; a **meta model** is yet another abstraction, highlighting properties of the model itself. A well-known meta model is the Unified Modeling Language (UML) that is used to specify, visualize, modify, construct, and document the artifacts of an object-oriented software system.

10.2 Getting started

10.2.1 Handling models

The elements contained in an `ILcdModel` are domain objects. In case these domain objects implement `ILcdDataObject`, the model should have an `ILcdModelDescriptor` that implements `ILcdDataModelDescriptor`. This interface gives access to the *data model* on which the model's elements are based.

A data model, represented using the `TLcdDataModel` class, is defined as a collection of types. A type is always declared by exactly one data model. Data models allow you to group types in logical units, much like Java classes are grouped in packages or XML schema types are grouped in XML schema documents.

[Program 49](#) shows how you can use an `ILcdDataModelDescriptor` to print out all data types that are defined for elements within a certain model. Note that this does not require iteration over the elements of the model.

```

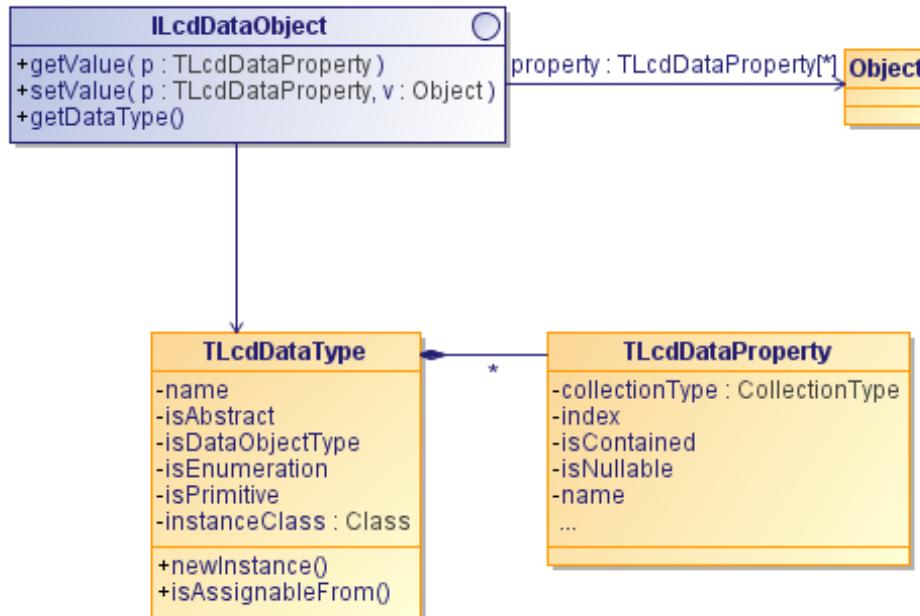
1 public void print( ILcdModel model ) {
2     if ( model.getModelDescriptor() instanceof ILcdDataModelDescriptor ) {
3         ILcdDataModelDescriptor desc = ( ILcdDataModelDescriptor ) model.getModelDescriptor();
4         for ( TLcdDataType t : desc.getModelElementTypes() ) {
5             System.out.println( t.getDataModel() + ":" + t.getName() );
6         }
7     }
8 }
```

Program 49 - Displaying all element types that are defined for a certain model

For a more elaborate code sample refer to `samples\util\dataModelDisplayTree`. The sample shows how to display the information of a data model in a tree model.

10.2.2 Handling domain objects

The interface `ILcdDataObject` is the unified representation of a domain object. A data object holds a set of named properties. Each property contains either a simple primitive-type value or a reference to another data object. The interface `ILcdDataObject` provides methods to manipulate these properties. A data object also provides access to its type. The type defines which properties an object can have. [Figure 27](#) shows `ILcdDataObject` in a diagram.

**Figure 27 - ILcdDataObject**

Using `ILcdDataObject` allows you to access the data stored inside a domain object. For example, [Program 50](#) shows how you can print the values of all properties of an object on the console.

```

1 public void print( ILcdDataObject dataObject ) {
2     for ( TLcdDataProperty p : dataObject.getDataType().getProperties() ) {
3         System.out.println( p.getName() + " => " + dataObject.getValue( p ) );
4     }
5 }
```

Program 50 - Printing all properties of an object

For a more elaborate code sample refer to `samples\util\dataObjectDisplayTree`. The sample shows how to display the properties of a data object in a tree model.

To change an object, you can use `ILcdDataObject` as shown in [Program 51](#). This code snippet shows how to translate all string typed properties for a given data object.

```

1 public void translateDataObject( ILcdDataObject dataObject ) {
2     for ( TLcdDataProperty p : dataObject.getDataType().getProperties() ) {
3         if ( p.getType() == TLcdCoreDataTypes.STRING_TYPE ) {
4             dataObject.setValue( p, translate( ( String ) dataObject.getValue( p ) ) );
5         }
6     }
7 }
8 public String translate( String aString ) { ... }
```

Program 51 - Translating the data inside an object

Finally, you can create new instances with a `TLcdDataType`. [Program 52](#) shows how to create an instance for a given type. The code also sets the value of the `name` property.

```

1 public ILcdDataObject createObject( TLcdDataType type, String name ) {
2     ILcdDataObject result = type.newInstance();
3     TLcdDataProperty p = type.getProperty( "name" );
4     if ( p != null ) {
```

```

5     result.setValue( p, name );
6 }
7     return result;
8 }
```

Program 52 - Creating instances with TLcdDataType

10.2.3 Browsing type information

Given a data model, it is possible to browse for all kinds of type information. Program 53 shows for example how to find all types that are declared in a given data model and that are a subtype of a given type.

```

1 public List<TLcdDataType> findSubTypes( TLcdDataModel dataModel, TLcdDataType type ) {
2     List<TLcdDataType> result = new ArrayList<TLcdDataType>();
3     for ( TLcdDataType t : dataModel.getDeclaredTypes() ) {
4         if ( type.isAssignableFrom( t ) ) {
5             result.add( t );
6         }
7     }
8     return result;
9 }
```

Program 53 - Finding child types

10.2.4 Defining a data model

Section 5.1 introduced a simple model that defines a flight plan. Figure 28 shows how to extend this flight plan with an association to a way point.

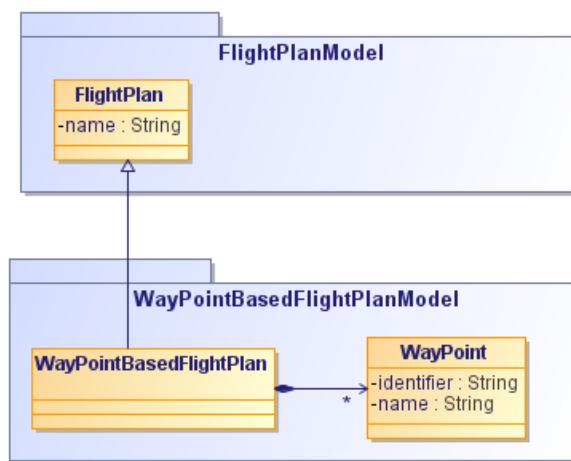


Figure 28 - Extending the Flight Plan model

You can create a data model for this extension using a `TLcdDataModelBuilder`. With such a builder, you create the `WayPointBasedFlightPlan` type and the `WayPoint` type as shown in Program 54.

```

1 public TLcdDataModel createWayPointBasedFlightPlanModel() {
2     TLcdDataModelBuilder builder = new TLcdDataModelBuilder( "http://www.mydomain.com/datamodel/
3     WayPointBasedFlightPlanModel" );
4     TLcdDataTypeBuilder typeBuilder = builder.typeBuilder( "WayPointBasedFlightPlanType" );
4     typeBuilder.superType( FlightPlanDataTypes.FLIGHT_PLAN_DATA_TYPE );
```

```

5   typeBuilder.addProperty( "wayPoints", "WayPointType" ).collectionType( CollectionType.LIST )
6   ;
7   typeBuilder = builder.typeBuilder( "WayPointType" );
8   typeBuilder.addProperty( "name", TLcdCoreDataTypes.STRING_TYPE );
9   typeBuilder.addProperty( "identifier", TLcdCoreDataTypes.STRING_TYPE );
10  return builder.createDataModel();
}

```

Program 54 - Creating the data model

Note how the `wayPoints` property on `WayPointBasedFlightPlanType` is created. Unlike the other properties, the type of this property has not yet been created. Therefore, you cannot use a type object to set the property's type. Instead, you need to refer to that type using its name (`WayPointType`). Note that it does not matter if `WayPointType` is defined before or after the `WayPointBasedFlightPlanType`. The `CollectionType.LIST` marks the property as a list, this is an ordered collection implemented at runtime by instances of `java.util.List`.

The super type of `WayPointBasedFlightPlanType` is `FlightPlanType`. As such, `WayPointBasedFlightPlanType` inherits the name and geometry properties of `FlightPlanType`. Also, because of this relation, the `WayPointBasedFlightPlanModel` data model depends on the original `FlightPlanModel` data model.

Program 54 does not assign instance classes for the two new types. As a result, new instances will be instances of `TLcdDataObject`. You can change this by setting appropriate instance classes.

10.3 Explaining the meta model

Figure 29 shows a UML diagram of the LuciadLightspeed meta model. The three main classes defined by the meta model are `TLcdDataModel`, `TLcdDataType`, and `TLcdDataProperty`. The following sections provide more information on each of the main classes.

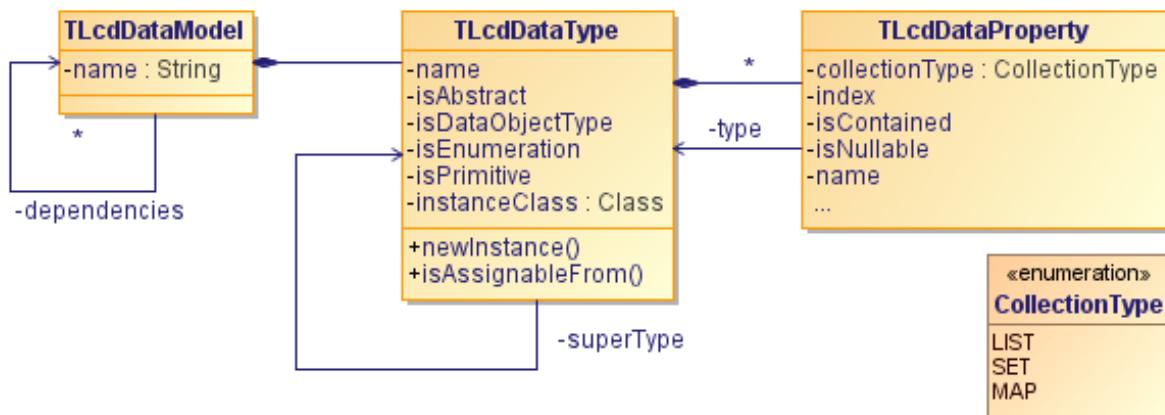


Figure 29 - The meta model

10.3.1 Describing the static model

`TLcdDataModel`

A `TLcdDataModel` is a collection of types that forms a logical entity. As such you can compare it to a Java or UML package or an XML schema. A data model is identified by a unique name and can have a number of dependencies. A data model depends on another data model when

domain model objects of the data model need to refer to domain model objects of another data model. This is for example the case when a type of the data model declares a property of a type of another data model. This is actually the most common case of dependency. As such, these dependencies are automatically defined by the framework. Dependencies can be cyclic.

A special kind of data model is an **anonymous** data model. By definition, this is a data model that has no name and declares no types. It only has dependencies. These data models are typically used as a simple way to represent a group of data models as a single data model.

TLcdDataType

A `TLcdDataType` represents the type of a data object. A data type describes the structure of a data object as a list of properties. Each of these properties themselves are of a certain type. A type always has a super type. The only exception to this rule is the `Object` type, which has no super type. A type (recursively) inherits all properties from its super type. Note that a type cannot redefine these inherited properties, they are always inherited as is.

A type is either a **primitive** type or a **data object** type. Primitive types are types which have no internal structure (no properties) and typically represent simple objects such as strings, numbers, dates, and so on. All primitive types either extend from another primitive type or directly from the `Object` type. A data object type is structured as a list of properties. All data object types either extend from another data object type or from the `DataObject` type. Instances of data object types implement `ILcdDataObject`.

Types are always defined in the context of a `TLcdDataModel`. Within a certain data model, types are uniquely defined by their name. As such, two types are equal if their data models are equal and they have the same name.

Types are mapped on Java classes. This is a many-to-one mapping; each type maps on one Java class, but different types can map on the same Java class. This class is called the type's instance class. Instances of a type are always instances of the type's instance class.

A type can be defined as an enumeration. This means that there are only a fixed set of possible instances. This set is directly available from the `TLcdDataType`'s interface. A prime example of such a type is a type of which the instance class is a Java enumeration.

You can create an instance of a data object type using the `newInstance` method. Note that primitive types do not support this method.

TLcdDataProperty

A `TLcdDataProperty` represents a property of a type. A property always belongs to a certain type (called its declaring type) and also has a certain type (called its type).

A property either has a single value or multiple values. In case of a single value, the value of the property for a certain data object is an instance of the type of the property. In case of multiple values, the property has a `CollectionType` that defines if property values are represented as a set, a list, or a map. Values of the property will be instances of which the class implements one of the Java `Set`, `List` or `Map` interfaces. The elements of these `Set` and `List` objects and the values of these `Map` objects are instances of the property's type. The keys in the `Map` objects are instances of the property's map key type.

By default, a property is defined as **contained**. This means that, by default, the values for properties are not shared among data objects. In other words, the graph of data objects of which the types only have contained properties is a tree. Cycles are only possible if a property is not contained.

Properties which are not nullable should always have a value that is different from `null`.

10.3.2 Creating data models

You create a data model using a `TLcdDataModelBuilder`. With such a builder, you create a `TLcdDataTypeBuilder` for each type you need to create. With the `TLcdDataTypeBuilder` you can build a data type. When you add a property to a data type builder, a `TLcdDataPropertyBuilder` is returned. That allows you to configure the properties.

Once all types and properties are built, you can ask the `TLcdDataModelBuilder` to create a data model. At the same time, this creates all types and properties. From that point on, the data model builder and all type and property builders depending on it will no longer accept any interaction and throw exceptions to indicate that the data model has been built. This ensures that you cannot modify a data model once it has been created.

The `TLcdDataModelBuilder` allows the creation of other `TLcdDataModelBuilder` instances. This is required to create data models that have a cyclic dependency.

10.3.3 Comparing with `ILcdFeatured`

Before the framework for unified domain access was introduced, the interface `ILcdFeatured` was used to represent and access domain objects. This section describes the main differences between the unified domain access and `ILcdFeatured` and is only relevant for developers who have already used `ILcdFeatured` before. The framework for unified domain access is the recommended framework to represent and access domain object.

`ILcdFeatured` and `ILcdFeaturedDescriptor` provide similar functionality as `ILcdDataObject` and `TLcdDataType` and also allow access to domain data in a unified way. But these interfaces are more suited for domain models with a simple structure. They are less suited for domain models, such as GML3, AIXM5 and KML, with a rich hierarchical structure.

The `com.luciad.datamodel` package provides all of the capabilities of `ILcdFeatured` and adds some enhancements. `ILcdDataObject` is the counterpart for `ILcdFeatured`. Both interfaces are to be implemented by domain classes and allow unified data access. A notable difference is that `ILcdFeatured` refers to a property using an index where `ILcdDataObject` uses a full-fledged property object. `TLcdDataType` is the counterpart for `ILcdFeaturedDescriptor`. Both provide additional information about the static structure of a domain object. However, `TLcdDataType` is much more expressive and easily allows descriptions of complex or many-valued properties and type hierarchies. As a result, the static type information of an entire domain model can be represented and accessed independent of domain object instances. `ILcdFeaturedDescriptor` does not readily support this.

Another notable difference is that `ILcdDataObject` provides immediate access to its type. Contrary to retrieve a descriptor given an `ILcdFeatured` object, for which you need an additional `ILcdFeaturedDescriptorProvider`. The need for this provider often results in additional code. While traversing an object hierarchy, you not only need to keep track of the object but also of the provider.

Finally, the `com.luciad.datamodel` package supports attachment of application-specific metadata to types and properties. This makes the meta model extensible. The `ILcdFeaturedDescriptor` does not support this.



The `com.luciad.datamodel` package is the preferred way to model domain objects from LuciadLightspeed V10 onwards. All formats supported by LuciadLightspeed implement `ILcdDataObject`. New formats no longer support `ILcdFeatured`. For reasons of backwards compatibility, `ILcdFeatured` will remain supported as is. No new features will be added to it.

10.4 Advanced topics

Section 10.2 gives a quick introduction to the meta model and describes a simple use case. This section covers some more advanced topics.

10.4.1 Designing a data model

The following subsections describe the considerations to take when modeling domain objects.

Single or multiple data models

You have to decide whether you want to model your domain with one or more data models. Using a single model is the simplest. Using multiple data models allows partitioning of types into different groups, which can benefit larger models. For example, suppose that the sample described in Section 10.2 is extended so that it includes multiple different types of way points to cater for nation-specific differences. In that case, you might decide to use two data models: one for the `WayPointsBasedFlightPlanType`, and one for the different way point types.

Choosing the right super type

The meta model supports single inheritance. Each type has exactly one super type. The only exception is the predefined `TLcdCoreDataTypes.OBJECT_TYPE` that serves as the root of all types. This type has no super type. Another essential core type is `TLcdCoreDataTypes.DATA_OBJECT_TYPE`. This type is the root of all *data object types*. By definition, only data object types can have properties defined on them. Types that do not extend from `DATA_OBJECT_TYPE` are called **primitive types**. These types are typically used for data of which the internal structure is a black box. Typical examples are simple types like `String`, `Number`, and `Date` of which the values are immutable. But that is not a hard constraint. In the end, it is up to you to decide which types are data object types and which are primitives. For example, it is a perfectly good practice to define a primitive type with instance class `ILcd2DEditablePoint` (which obviously is not immutable) if you do not want to expose the internal structure of the point (its x and y coordinate) using the data model API.

Many-valued properties

The meta model also supports many-valued properties. These properties are typically represented in the domain class as some form of collection. For example, in the Program 54 the `wayPoints` property is represented by a `List`. Note that if you provide an instance class for the `WayPointsBasedFlightPlanType`, then that class does not have to provide an accessor to set the value of the `wayPoints` property. Instead, modifications are done by changing the list object returned by the getter. Note that `ILcdDataObject` requires that implementations provide an initial (empty) list object.

```

1 ...
2 TLcdDataTypeBuilder typeBuilder = builder.typeBuilder( "WayPointsBasedFlightPlanType" );
3 ...
4 typeBuilder = builder.typeBuilder( "WayPointType" );
5 ...

```

```

6 ILcdDataObject flightPlan = ...;
7 ILcdDataObject wayPoint = ...;
8 flightPlan.getValue(WAYPOINTS_PROPERTY).add(wayPoint);

```

Program 55 - A list property

Data model dependencies

Data models can depend on other data models. You can define dependencies either implicitly or explicitly. An implicit dependency is created when one of the types of a data model has a property of a type from another data model. These dependencies are automatically detected and added by the framework. This is the typical use case. However, the `TLcdDataModelBuilder` also allows to explicitly add dependencies. This is, for example, used to create an anonymous data model that merely serves as a container of other data models. Note that graphs with cyclic dependencies are supported.

10.4.2 Traversing an object graph

An often occurring pattern when dealing with data objects is traversal through the data contained in a graph of data objects. The `ILcdDataObject` interface allows you to traverse the object graph. **Program 56** traverses an `ILcdDataObject` by looking at the values of all its properties. If the value is not null, then depending on whether the property is a map, a collection, or just a single value, traversal continues on the value.

```

1 public static void traverseDataObject(ILcdDataObject aObject) {
2     for (TLcdDataProperty property : aObject.getDataType().getProperties()) {
3         Object value = aObject.getValue(property);
4         if (value != null) {
5             if (property.getCollectionType() == null) {
6                 // Single-valued property
7                 traverseChild(property.getType(), value);
8             } else {
9                 switch (property.getCollectionType()) {
10                     case MAP:
11                         Map<Object, Object> map = (Map<Object, Object>) value;
12                         for (Map.Entry<Object, Object> entry : map.entrySet()) {
13                             traverseChild(property.getMapKeyType(), entry.getKey());
14                             traverseChild(property.getType(), entry.getValue());
15                         }
16                         break;
17                     case LIST:
18                     case SET:
19                         for (Object element : (Collection<?>) value) {
20                             traverseChild(property.getType(), element);
21                         }
22                         break;
23                 }
24             }
25         }
26     }
27 }

```

Program 56 - Traversing a data object

(from `samples/common/dataObjectTraversal/DataObjectTraversalUtil`)

Program 57 traverses a single object. The type of the object is passed as argument. Note that this is merely the *declared* type. The object itself may be of a specialized type. This is in accordance with the standard rules on polymorphism.

- If the type is primitive, then no further (generic) introspection is possible.

- If the type is a data object type, then the object has to be an `ILcdDataObject` and traversal can continue as shown in [Program 56](#).
- Otherwise the type is `TLcdCoreDataTypes.OBJECT_TYPE`. This is an exceptional case where you do not have enough type information available. You have to examine the object that is passed as parameter. The method `traverseObject` uses the Java `instanceof` operator to determine how to continue the traversal.

```

1  public static void traverseChild(TLcdDataType aType, Object aObject) {
2      if (aType.isPrimitive()) {
3          traversePrimitive(aType, aObject);
4      } else if (aType.isDataObjectType()) {
5          traverseDataObject((ILcdDataObject) aObject);
6      } else {
7          traverseObject(aObject);
8      }
9  }
10
11 public static void traverseObject(Object aObject) {
12     if (aObject instanceof Map<?, ?>) {
13         for (Map.Entry entry : ((Map<?, ?>) aObject).entrySet()) {
14             // ...
15         }
16     } else if (aObject instanceof Collection<?>) {
17         for (Object element : (Collection<?>) aObject) {
18             traverseObject(element);
19         }
20     } else if (aObject instanceof ILcdDataObject) {
21         traverseDataObject((ILcdDataObject) aObject);
22     } else {
23         traversePrimitive(null, aObject);
24     }
25 }
26
27 public static void traversePrimitive(TLcdDataType aType, Object aValue) {
28     // ...
29 }
```

Program 57 - Traversing an object

(from samples/common/dataObjectTraversal/DataObjectTraversalUtil)

A practical example with properties of type `TLcdCoreDataTypes.OBJECT_TYPE` is the modeling of a choice from XML schema or a union from C. Both a choice and a union property can have values of different types. You can model this by representing the choice or union by a single property. The type of that property is the common super type of the choice or union types. If the choice or union types contain both data object types and primitive types, this common super type is `TLcdCoreDataTypes.OBJECT_TYPE`.

Traversal as shown in the programs above works fine for graphs of objects in which there are no cycles. This is the case for most of the supported domain models in LuciadMap. In case of cycles, you need to add a check when a data object is traversed to see if the object has not already been traversed before.

10.4.3 Implementing `ILcdDataObject`

In some cases, you might not want your domain classes to extend from `TLcdDataObject`. In that case you have to implement `ILcdDataObject` yourself. Probably the easiest way to do this is to use delegation as shown in [Section 5.1.1](#). There are cases where you may not want to keep the state in a delegate `TLcdDataObject`. [Program 58](#) shows an alternative implementation for a simple `Person` type with two properties. The `getValue` and `setValue` operations can be implemented easily based on a switch on the index of the property. This index is the position

of this property in its declaring type's properties. Because this index is fixed by the data model, you can use it as an efficient way of resolving properties. The `getDataType` operation can easily be implemented using a static reference to the type. In cases where more than one type uses the same domain class, you may need to add the data type as an instance variable.

```

1 public class Person implements ILcdDataObject {
2     private static final TLcdDataType PERSON_TYPE = ...;
3
4     private String lastName;
5     private String firstName;
6
7     ...
8     public String getFirstName() { return firstName; }
9     ...
10    public Object getValue( TLcdDataProperty p ) {
11        if ( p.getDeclaringType() != PERSON_TYPE )
12            throw new IllegalArgumentException( "Invalid property " + p );
13        switch( p.getIndex() ) {
14            case 0:
15                return getFirstName();
16            case 1:
17                return getLastName();
18            }
19            return null;
20        }
21
22    public TLcdDataType getDataType() {
23        return PERSON_TYPE;
24    }
25
26    public void setValue( TLcdDataProperty p, Object value ) {
27        if ( p.getDeclaringType() != PERSON_TYPE )
28            throw new IllegalArgumentException( "Invalid property " + p );
29        switch( p.getIndex() ) {
30            case 0:
31                setFirstName( ( String ) value );
32                break;
33            case 1:
34                setLastName( ( String ) value );
35                break;
36            }
37        }

```

Program 58 - Implementing `ILcdDataObject`

10.4.4 Adding custom metadata

In some cases, you might want to attach additional information to a `TLcdDataModel`. The meta model supports this through the use of `ILcdAnnotation`. You can attach implementations of this tag to a data model. As data models are immutable, all annotations that are or will be attached to them should be immutable as well.

Program 59 shows how to do this for a simple example. Suppose you have a data model that is derived from an XML schema. You want to add the location of this schema to the data model. First you define the `SchemaLocation` class that implements `ILcdAnnotation` and that keeps track of the location. Then, when your data model is being built, you can annotate the data model with an appropriate `SchemaLocation` instance.

```

1 public class SchemaLocation implements ILcdAnnotation {
2     private final String location;
3     public String getLocation() { return location; }
4     public SchemaLocation( String location ) {
5         this.location = location;
6     }
7 }

```

```

8 ...
9 TLcdDataModelBuilder dataModelBuilder = ...;
10 ...
11 dataModelBuilder.annotate( new SchemaLocation( "http://www.luciad.com/samples/model/1.1" ) );

```

Program 59 - Annotating a data model

TLcdDataModel provides methods to retrieve all annotations that are attached to it. Program 60 shows how to retrieve the schema location annotation from the data model.

```

1 SchemaLocation location = dataModel.getAnnotation( SchemaLocation.class );
2 if ( location != null ) {
3     System.out.println( "Data model " + dataModel.getName() + " is derived from " + location.
4         getLocation() );
}

```

Program 60 - Accessing annotations

Note that exactly the same annotation functionality is also defined on TLcdDataType and TLcdDataProperty.

10.4.5 Representing geometries

Some data types are directly linked to a geometry. For example, the FlightPlanType defined in Section 5.1 is represented by a polyline. In most cases, you should **not** add the information about the geometry to the data type. Instead, the data type should be given an appropriate instance class that implements an appropriate ILcdShape interface. This keeps the data type focused on the data while the implementation of the interface ensures integration with LuciadLightspeed.

There are two main exceptions to this general rule:

- Sometimes you may want to use a certain instance class for different geometries. The common practice in LuciadLightspeed for such cases is to have the instance class implement ILcdShapeList and put the geometry that needs to be supported inside the shape list. Most parts of LuciadLightspeed can handle such composite shapes. However, some information is lost. Given a data type with such an instance class, it is no longer possible to determine which type of shape it represents. You should use the TLcdShapeListAnnotation class to annotate the data type and prevent this loss of information. Another, less common, option is to use an ILcdShape with TLcdShapeAnnotation.
- In some cases, it does make sense to represent geometry information also in the data type through properties. An example of such a data model is GML. This data model closely resembles the GML XML schema. As such, all data (whether it corresponds to a geometry or not) is modeled using properties. In such cases, you should annotate these data types with the TLcdHasAShapeAnnotation. This asserts that the state exposed by the ILcdShape interface is also exposed by one or more of these types' properties.

You can consult the API reference for a more detailed explanation on these annotation classes.

CHAPTER II

Modeling vector data

Vector data consists of geometries such as lines, circles, or polygons. Examples of vector data are buildings, roads, railways, stations, and so on. To create a model containing vector data you can either use one of the model decoders for a specific format as listed in [Appendix E](#), or create a model decoder as described in [Section 5.1](#). When creating a model decoder for vector data yourself, or when modeling vector data without a model decoder, you can use a LuciadLightspeed shape to model (the geometrical part of) the domain objects.



When you are decoding a model, you must make sure that the coordinate reference system of the data is decoded too, as the model's reference. For more information, see [Chapter 16](#).

This chapter describes the shapes that LuciadLightspeed provides to model vector data. Depending on the geometry of the domain object, you can use a simple geometry such as a point, polyline, or [bounding box](#) to model the domain object. Or you can compose a more complex geometry using different simple geometries. It is also possible to model domain objects as a list of multiple (unconnected) shapes.

Each of the LuciadLightspeed shapes has a corresponding painter to render the shape in a 2D view as listed in [Table 4](#).

II.1 What is an `ILcdShape`?

A LuciadLightspeed shape (`ILcdShape`) is a geometrical object with a bounding box (`ILcdBounds`) and a focus (or center) point (`ILcdPoint`):

- The **bounding box** is defined by a width, a height, and a depth, respectively the x, y, and z coordinates of the box. All shapes are essentially 3D. For 2D shapes, the z coordinate of `ILcdBounds` is either ignored or set to 0. To retrieve the bounds of a shape, use the `getBounds` method.
- The **focus point** (or center point) of the shape is typically used as the handle or the labeling point of the shape in a GUI. You can retrieve the focus point using the `getFocusPoint` method.



Do not use `ILcdBounds.getHeight` instead of `ILcdPoint.getZ` to retrieve the height of a point. The height of the bounding box of a point is always 0 whereas the z coordinate of the point can have a value.

All LuciadLightspeed shapes implement `ILcdShape` and are available for both geodetic (LonLat) and Cartesian (XY) reference systems. For example, the implementation for 2D geodetic circles is `TLcdLonLatCircle` and the implementation for 2D Cartesian circles is `TLcdXYCircle`. Note that operations between shapes, such as containment tests, are only possible if the shapes are defined in the same reference system. An `ILcdShape` also has an editable variant, both for 2D and 3D shapes as described in [Section 11.1.1](#).

Figure 30 shows the `ILcdShape` hierarchy with some extensions and implementations. [Section 11.1.2](#) briefly describes the main extensions. Some of the extensions are described in more detail in a separate section. For a complete overview of all extensions and a more detailed description of all their properties and methods, refer to the API reference.



To find memory saving implementations of LuciadLightspeed shapes in the API, search for classes that contain `Float`.

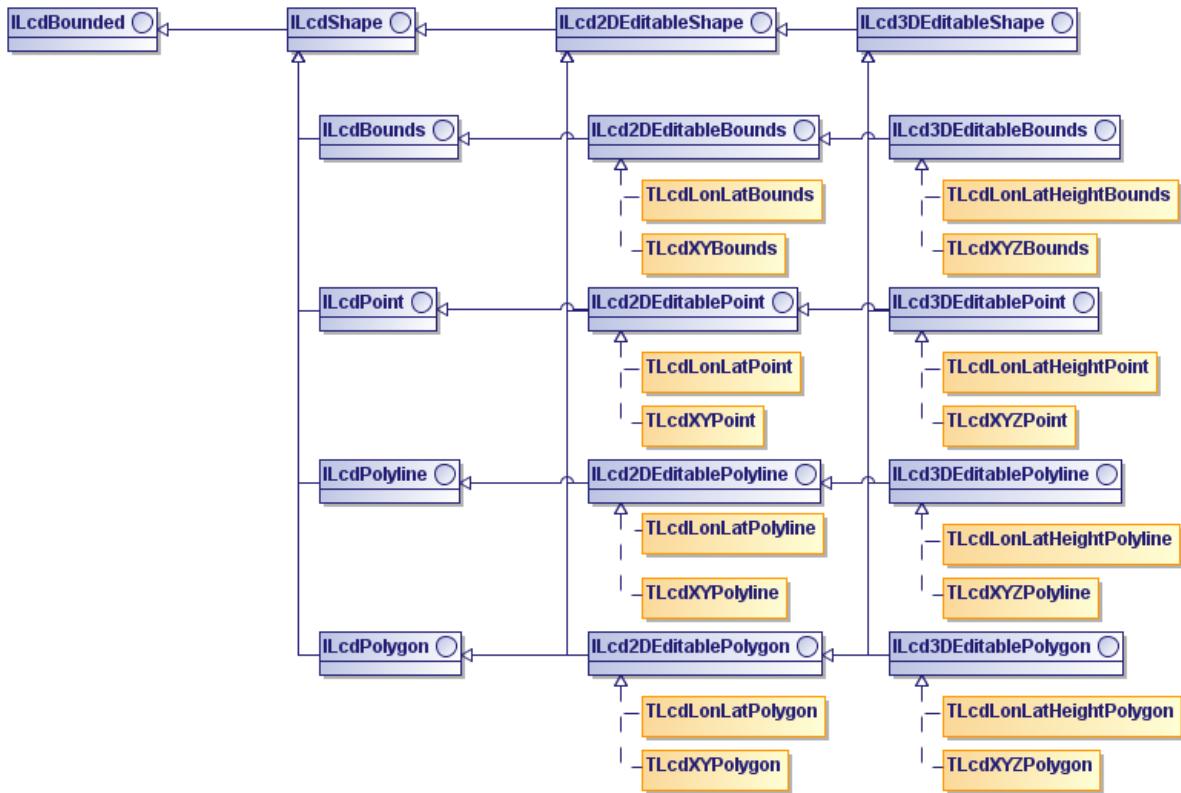


Figure 30 - The `ILcdShape` hierarchy with some extensions and implementations

11.1.1 Editable shapes

An `ILcd2DEditableShape` is an extension of `ILcdShape` that allows users to edit the X/Y- or Lon/Lat-coordinates of a 2D shape using the following methods:

- `move2D`: moves the shape to a given location. Usually, this method moves the focus point of the shape to the given location. But it is also possible to implement another move action.
- `translate2D`: translates the shape over a given delta in both dimensions.

`ILcd3DEditableShape` is an extension of `ILcd2DEditableShape` and represents a shape that can be moved in three dimensions. The reason for providing separate 2D objects is because

it saves memory. The implementing class of, for example a 2D point, only stores the first two coordinates in memory. For the third coordinate, it always returns 0.

11.1.2 Commonly used shapes

LuciadLightspeed provides the following commonly used shapes that all extend `ILcdShape`:

- `ILcdArc`: an elliptical arc. All points of an arc are located on an ellipse. The arc starts at a start angle and extends over an arc angle. You can rotate its semi-major axis.

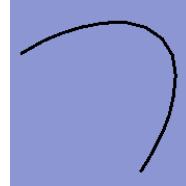


Figure 31 - An `ILcdArc`

- `ILcdArcBand`: a circular arc with a width. All of its points are located on the band with a minimum radius and a maximum radius from the focus point. The arc band starts at a start angle and extends over an arc angle.

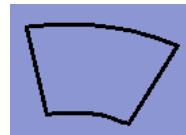


Figure 32 - An `ILcdArcBand`

- `ILcdBounds`: a box with a location (`ILcdPoint`), a width, a height, and a depth. `ILcdBounds` is often used as a bounding box for more complex shapes. For more information refer to [Section 11.3](#).



Figure 33 - An `ILcdBounds`

- `ILcdCircle`: a circle with a focus point (`ILcdPoint`) and a radius.

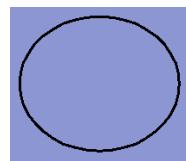


Figure 34 - An `ILcdCircle`

- `ILcdCircularArc`: a circular arc. All of its points are located on a circle with a focus point (`ILcdPoint`) and a radius. The arc starts at a start angle and extends over an arc angle. Refer to [Section 11.5](#) for the main subinterfaces of `ILcdCircularArc`.

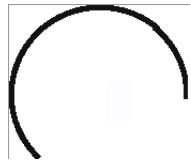


Figure 35 - An `ILcdCircularArc`

- `ILcdComplexPolygon`: a list of `ILcdPolygon` objects that results in a polygon with one or more holes. Each hole is a polygon as shown in [Figure 36](#).

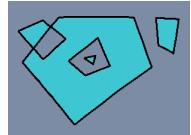


Figure 36 - An `ILcdComplexPolygon`

- `ILcdEllipse`: an ellipse with a focus point (`ILcdPoint`), a semi-major axis with a rotation angle, and a semi-minor axis.



Figure 37 - An `ILcdEllipse`

- `ILcdGeoBuffer`: a shape (`ILcdShape`) with a surrounding buffer that is defined by a width.

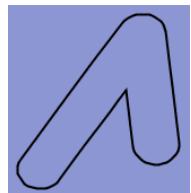


Figure 38 - An `ILcdGeoBuffer`

- `ILcdPolygon`: a closed polyline defined by multiple points and an orientation.

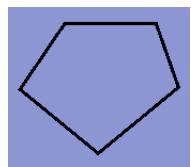


Figure 39 - An `ILcdPolygon`

- `ILcdPolyline`: a line defined by multiple points.

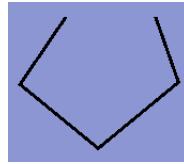


Figure 40 - An `ILcdPolyline`

- `ILcdPolyline`: a set of points.

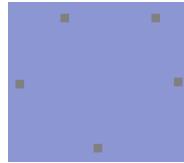


Figure 41 - An `ILcdPolypoint`

- `ILcdPoint`: a single point. For more information refer to [Section 11.2](#).
- `ILcdCurve`: a continuous `ILcdShape`. For more information refer to [Section 11.5](#).
- `ILcdRing`: a closed `ILcdCurve`, which means that the start and end point of the ring are the same. Since rings can be filled, they form the basis for surfaces.
- `ILcdShapeList`: a set of `ILcdShape` objects. For more information refer to [Section 11.4](#).
- `ILcdSurface`: a continuous 2D shape of which the exterior boundary is defined by a single `ILcdRing`. A surface contains zero or more holes. Each hole is also defined by an `ILcdRing`.
- `ILcdText`: a text string defined by a location (`ILcdPoint`), a character width and a character height, a horizontal and a vertical alignment, and a rotation angle.



Figure 42 - An `ILcdText`



The interfaces `ILcdComplexPolygon`, `ILcdPolygon`, `ILcdPolyline`, and `ILcdPolypoint` are extensions of `ILcdPointList`. An `ILcdPointList` is an indexed list of `ILcdPoint` objects. [Section 11.6](#) describes how to create an `ILcdPolyline` and `ILcdPolygon`.

11.2 What is an `ILcdPoint`?

An `ILcdPoint` is an `ILcdShape` representing a point in a 3D coordinate system. It is not possible to change an `ILcdPoint`, the interface only provides methods to retrieve properties. Its main methods are:

- `getX`, `getY`, and `getZ`: to retrieve the first, second, and third coordinate of the point respectively.

- `getCosX`, `getCosY`, `getSinX`, `getSinY`, `getTanX`, and `getTanY`: these methods are provided for efficiency reasons. Implementing classes can use these methods to retrieve and cache these values instead of recalculating them whenever they are needed.
- `cloneAs2DEditablePoint` and `cloneAs3DEditablePoint`: to provide a copy that can be moved in two and three dimensions respectively.

11.2.1 Editable points

An `ILcd2DEditablePoint` is an `ILcdPoint` that you can move in two dimensions. It inherits the methods of the interfaces `ILcdPoint` and `ILcd2DEditableShape`. Similarly, an `ILcd3DEditablePoint` is an `ILcdPoint` that you can move in three dimensions. It inherits the methods of the interfaces `ILcd2DEditablePoint` and `ILcd3DEditableShape`.



`ILcd2DEditablePoint` and `ILcd3DEditablePoint` objects are often used as output arguments. These are arguments that are changed as a side effect of the method called. The use of output arguments avoids the creation of a new object for each call of the method. The name of a method or parameter with a side effect always ends in SFCT.

11.3 What is an `ILcdBounds`?

An `ILcdBounds` is a 3D box of which the sides are aligned with the coordinate axes of the associated coordinate system. It is specified by a location and a width, height, and depth that give the extents along the first, second, and third coordinate direction respectively. The location is specified by an `ILcdPoint` that represents the bottom left corner of the box. An `ILcdBounds` is typically used as bounding box for a more complex geometry. Because of its simple geometry, you can use it to speed up certain types of calculations. Its main methods are:

- `getLocation`, `getWidth`, `getHeight`, and `getDepth`
- `interacts2D` and `interacts3D`: to check if there is any kind of interaction, for example overlap or touching, with another `ILcdBounds` object.
- `contains2D` and `contains3D`: to check if the `ILcdBounds` contains another `ILcdBounds`.
- `cloneAs2DEditableBounds` and `cloneAs3DEditableBounds`: to provide a copy that can be moved in two and three dimensions respectively.

11.3.1 Editable bounding boxes

The interfaces `ILcd2DEditableBounds` and `ILcd3DEditableBounds` extend the interface `ILcdBounds` with methods to change the bounding box in two and three dimensions respectively.

To change an `ILcd2DEditableBounds` you can use one of the following methods:

- `move2D`, `translate2D`, `setWidth`, and `setHeight`.
- `setToIncludePoint2D`: to minimally extend the `ILcd2DEditableBounds` so that it includes a given `ILcdPoint`.
- `setTo2DIntersection`: to calculate the intersection with a given `ILcdBounds`.

- `setTo2DUnion`: to calculate the union with a given `ILcdBounds`. This method returns the smallest `ILcdBounds` that contains both `ILcdBounds`.

You can move an `ILcd3DEditableBounds` in three dimensions using methods that are equivalent to the methods provided by `ILcd2DEditableBounds`.

11.4 What is an `ILcdShapeList`?

An `ILcdShapeList` is an `ILcdShape` that is composed of multiple `ILcdShape` objects according to the **Composite design pattern**. The `ILcdShape` objects in an `ILcdShapeList` are ordered as shown in Figure 43. A shape list is useful for defining shapes that are topologically not connected. For example, countries with islands or with overseas areas. Its main methods are:

- `getShape`: to retrieve the `ILcdShape` at a given index in the `ILcdShapeList`.
- `getShapeCount`: to retrieve the number of `ILcdShape` objects in the `ILcdShapeList`.

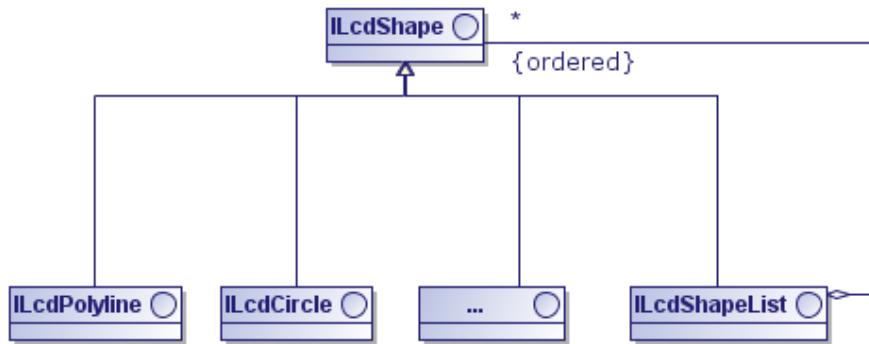


Figure 43 - An `ILcdShapeList` is a composite `ILcdShape`

11.5 What is an `ILcdCurve`?

An `ILcdCurve` is a continuous `ILcdShape` such as a line, an arc, or any connected combination of such shapes. The principal method of a curve is: `void computePointSFCT(double aParam, ILcd3DEditablePoint aPointsFCT)` whereby `aParam` can be between 0 and 1. Use this method to retrieve any point from the start point (`aParam=0`) until the end point (`aParam=1`) of the curve. You can also retrieve the start and end point of the curve using the `getStartPoint` and `getEndPoint` methods.

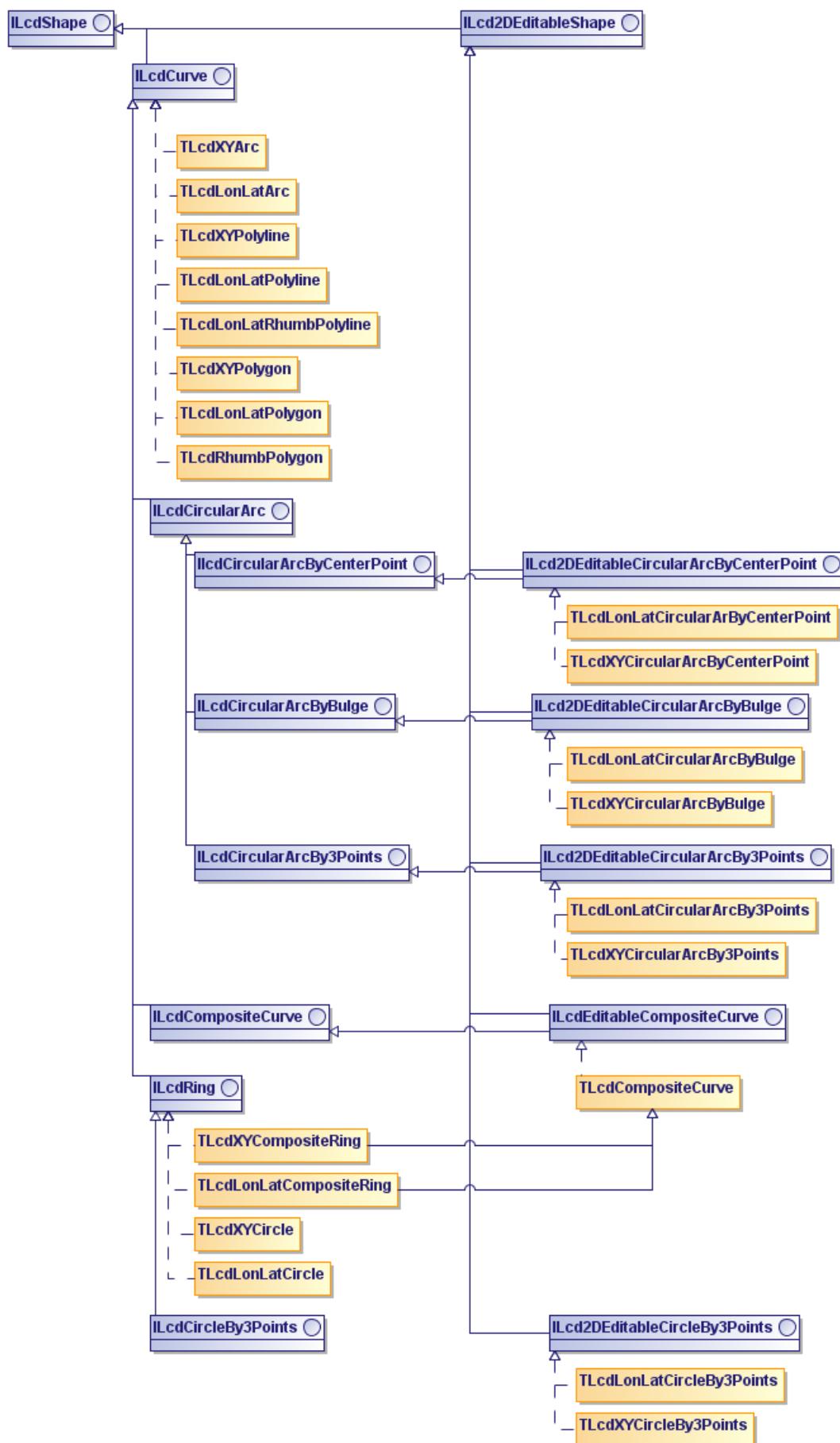
Figure 44 - The main `ILcdCurve` implementations

Figure 44 shows the main implementations of `ILcdCurve`. The following extensions are particularly noteworthy:

- An `ILcdCompositeCurve` is composed of curves similarly as an `ILcdShapeList` is composed of shapes, but it is an `ILcdCurve` itself. Hence, all sub-curves are connected to each other. Together they form a new shape of which you can retrieve the points using a single parameter. The `ILcdCompositeCurve` starts at the start point of the first sub-curve and ends at the end point of the last sub-curve.
- `ILcdCircleBy3Points`: a circle defined by a start control point, intermediate control point, and end control point.
- The main subinterfaces of `ILcdCircularArc`:
 - `ILcdCircularArcByCenterPoint`: an arc defined on a circle, using a center point, radius, start angle, and arc angle.
 - `ILcdCircularArcByBulge`: an arc defined on a circle, using the start point, end point, and the bulge factor. The bulge factor is the ratio between the distance from the center of the chord to the midpoint of the arc and the length of this chord.
 - `ILcdCircularArcBy3Points`: an arc defined on a circle, using the start point, end point, and a third point on the arc.

11.6 Creating an `ILcdPolyline` or `ILcdPolygon`

To create a `TLcdLonLatPolyline` or `TLcdLonLatPolygon` (and their Cartesian counterparts), you first need to create an `ILcd2DEditablePointList` as shown in Program 61.

```

1  private Object createLonLatPolygon(ILcdEllipsoid aEllipsoid) {
2      ILcd2DEditablePoint[] points = {
3          new TLcdLonLatPoint(3.4, 51.5),
4          new TLcdLonLatPoint(4.3, 51.8),
5          new TLcdLonLatPoint(5.2, 51.6),
6          new TLcdLonLatPoint(5.2, 50.5),
7          new TLcdLonLatPoint(3.7, 50.4),
8      };
9      ILcd2DEditablePointList pointList =
10         new TLcd2DEditablePointList(points, false);
11     return new TLcdLonLatPolygon(pointList, aEllipsoid);
12 }
13 }
```

Program 61 - Creation of a `TLcdLonLatPolygon` object
(from samples/gxy/shapes/MainPanel)

Program 61 shows the steps involved with creating a `TLcdLonLatPolygon`:

1. Create an array of `ILcd2DEditablePoint` objects (with instances of the class `TLcdLonLatPoint`).
2. From this array create an `ILcd2DEditablePointList` (which is an instance of the class `TLcd2DEditablePointList`).
3. From the `ILcd2DEditablePointList` create a `TLcdLonLatPolygon`.

These steps are similar for the creation of a `TLcdXYPolygon`, `TLcdLonLatPolyline`, and `TLcdXYPolyline`.

11.7 Notifying objects of changes to shapes

Whenever a shape changes, LuciadLightspeed automatically notifies registered objects of the change using the Listener pattern as described in [Chapter 8](#). The following specific cases, are an exception to this rule:

- **Changing the values of an object that is contained in another (composite) object.** In this case, the action object needs to use the method `ILcdInvalidateable.invalidateObject` to notify the composite object of the change. The composite object can then for example recompute its cached values. An example of this case is an `ILcdShapeList`.
- **Editing a shape of `ILcdEditableShapeList`.** In this case, the action object needs to use the method `shapeChanged` to notify the `ILcdEditableShapeList` of the change.
- **Changing the base shape of an `ILcd2DEditableGeoBuffer`.** In this case, the action object needs to use the method `invalidateShape` to notify the `ILcd2DEditableGeoBuffer` of the change.

CHAPTER 12

Working with images

LuciadLightspeed's image processing framework allows you to access images built from raster data, and manipulate the data at the pixel level. This is typically useful for enhancing an image, or to highlight certain aspects of the raster data when you display the image on screen. The image processing API offers various image manipulation operations that you can apply to raster data. The supported operations include the typical image operations required to work with:

- Multi-band images, acquired from multi-spectral remote sensing equipment such as Earth-observing satellite sensors.
- High Dynamic Range (HDR) images with a large bit depth
- Low quality images that require enhancing

Using the available image processing operations, you can:

- Determine the type of data in an image
- Select which bands to use for visualizing a multi-spectral image
- Apply tone mapping to images with a large dynamic range
- Apply a convolution filter to sharpen an image

To process images in these ways, you need to use the `ALcdImage` objects decoded by your `ILcdModelDecoder`.



All LuciadLightspeed raster model decoders provide domain objects that either are an image, and extend from `ALcdImage`, or have-an-image, which means that they have a property that is an `ALcdImage`. In either case, you can retrieve the `ALcdImage` using `ALcdImage.fromDomainObject`. In addition, the domain objects implement the `ILcdRaster` or `ILcdMultilevelRaster` interfaces.

To be able to apply image operations to the data, use `ALcdImage`. If you are upgrading from an older version of LuciadLightspeed, you can still use the `ILcdRaster` API, which remains available for backward compatibility reasons. For more information about `ILcdRaster`, see [Chapter 13](#).

To apply image operations, you use any one of the LuciadLightspeed image operators. By chaining these operators together, you can build sophisticated image processing graphs. You can apply the processing offline on disk, or interactively to the image data visualized in a Lightspeed view.

Once you have finished manipulating your image, you can encode and store it in the GeoTIFF format. This chapter discusses the main LuciadLightspeed image processing API modeling and visualization concepts and operations.

12.1 Using a model decoder

To model image data, you typically use a model decoder. LuciadLightspeed offers model decoders for loading and modeling data from an external source in the most commonly used image formats. To properly represent an image in a view, the following information is required:

- The **model reference** that defines the coordinate system used to locate the image data on the Earth as described in [Section 2.2.1](#). When you are decoding a model, you must make sure that the coordinate reference system of the data is decoded too, as the model's reference. For more information, see [Chapter 16](#).
- The **image bounds** that define the coordinates of the four corners of the image in the associated reference system.

Based on these requirements you can distinguish two types of raster formats:

- **Georaster formats** that specify both the reference system and the raster bounds. The model decoders for georaster formats can create the raster model based on the source data. Examples of georaster formats are DMED, DTED, DEM, and GeoTIFF.
- **Non-georaster formats** that define the raster bounds but do not specify the reference system. Examples of these formats are GIF, JPEG, TIFF, and BMP. For these raster formats you need to define a model reference yourself as described in [Section 16.2.2](#).

[Table 14](#) gives an overview of all file formats that LuciadLightspeed supports and indicates if a model decoder is available for the format or not. For more information on a specific model decoder, refer to the API reference.

Each decoded image model contains a single element that is an extension of `ALcdImage`.

12.2 Image domain model

12.2.1 `ALcdImage`

`ALcdImage` is the main class for you to use when you are working with images, and processing those images. It represents a geographically bounded, pixel-oriented data source. You can create instances of `ALcdImage` using the `TLcdImageBuilder` class.

Each image provides metadata about the internal storage and the semantics of its pixel data via its `getConfiguration()` method. An image can contain measurement values such as elevations, plain RGB image data such as aerial photos, or multi-band data such as LandSat satellite imagery. The semantics of each band are represented by the class `ALcdBandSemantics`, and can be queried through the `getSemantics()` method. The band semantics provide the following information:

- The **data type** defines the type of data stored in the band. This indicates whether the data is stored as unsigned bytes, signed integers, floating point values, and so on.
- The **number of significant bits** indicates how many bits are used to represent the pixel value for non-floating point data types, for example 12 of the 16 bits of a short represent the pixel value.

- The **no data value** is a number that is used to indicate that the pixel contains no data.
- Whether or not the data has been **normalized** and what the **normalization range** is.

Two implementations of `ALcdBandSemantics` are available:

- `ALcdBandColorSemantics`: represents a color band in an image. This class allows querying the type, color space, and component index. A color band can be a gray band, a Red-Green-Blue band, or a Red-Green-Blue-Alpha channel band.
- `ALcdBandMeasurementSemantics`: represents a band with measurement values, for example elevation data. This class allows querying the unit of measure, an `ILcdISO19103UnitOfMeasure`, of the band.

`ALcdImage` itself only defines the image metadata. In practice, a LuciadLightspeed image processing application will work with one of four base classes derived from `ALcdImage`. These classes are discussed in the following sections.

12.2.2 `ALcdBasicImage`

`ALcdBasicImage` is the most basic image representation in LuciadLightspeed. It represents a uniform two-dimensional grid of pixels.

`ALcdBasicImage` is an opaque handle, which by itself does not expose the underlying pixel data directly. You need an `ALcdImagingEngine` to access the pixel values. Images can be created using `TLcdImageBuilder`, or they can be the result of a chain of image operators, applied to one or more other images, as explained in [Section 12.3](#).

Information on the size and layout of the image, such as the number of pixels in each dimension, can be obtained from the image's Configuration.

Determining whether an image is area-sampled or point-sampled

The image's configuration object also indicates whether the image data is of an area-sampled or point-sampled nature, through the `ELcdImageSamplingMode` enumeration.

Color imagery obtained with a digital camera, be it RGB or multi-band imagery, is typically area-sampled. In an area-sampled image, each pixel value represents some form of average value for the area covered by the pixel, the average intensity of red, green or blue light within the area for example. Images containing measurement values, such as elevation data or weather data, are more commonly point-sampled. In a point-sampled image, each pixel represents the exact measurement value at a particular coordinate.

[Figure 45](#) illustrates the distinct composition of an area-sampled image and a point-sampled image. The numbers indicate pixel coordinates, and the green rectangles indicates the bounds of the images. In the area-sampled image, each pixel value is associated with a small, rectangular sub-region of the image. In the point-sampled image, each value is associated with a sample point that lies on an intersection of a horizontal and vertical "grid line".

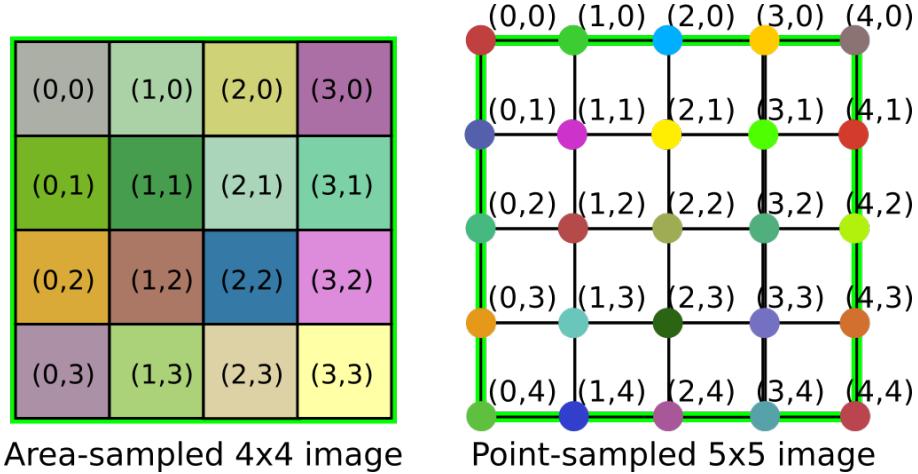


Figure 45 - Comparison of area-sampled and point-sampled images

Given two images with the same pixel dimensions and equal geographical pixel spacing, the area-sampled image will therefore have slightly larger geographical bounds than the point-sampled one. The bounds of the area-sampled image will be the union of the areas covered by all individual pixels, whereas the bounds of the point-sampled image will be the minimal bounds containing all the sample points corresponding to each pixel.

12.2.3 Multi-level images and mosaics

`ALcdBasicImage` is the main building block for more advanced data structures containing pixel data, such as multi-level images and (multi-level) image mosaics.

ALcdMultilevelImage

A multi-level image is a collection of `ALcdBasicImages` covering the same geographic area. Typically these images have a different resolution, for example the lower levels provide a lower-resolution overview of the higher levels. For visualization, the spatial resolution of the images will be matched to the current scale of the view. The image with the best match will be selected.

The relation between the image levels is defined by the `ELcdLevelRelationship`, which can be:

- **OVERVIEW**: at each level, the image offers a different level of detail for the same data.
- **STACK**: the images represent data in the same geographic region, but are otherwise unrelated.

ALcdImageMosaic and ALcdMultilevelImageMosaic

An image mosaic is a grid of multiple `ALcdBasicImages`. Each image in the grid is called a tile. This grid can be sparse: some tiles are not present in the grid. The tiles may also have different pixel resolutions.

Like `ALcdBasicImage`, `ALcdImageMosaic`'s configuration object contains an `ELcdImageSamplingMode` property. All tiles in a mosaic are expected to have the same sampling mode.

A multi-level image mosaic is a collection of `ALcdImageMosaics`, with properties similar to `ALcdMultilevelImage`.

Why use multi-leveled mosaics?

Multi-leveled mosaics allow for multiple imagery resolutions It is virtually impossible to visualize useful amounts of imagery or elevation data using brute force rendering methods. Because of the increasing availability of satellite imagery and aerial photography with resolutions of less than 1 meter per pixel, the size of the datasets that are typically involved has grown orders of magnitude beyond the amount of memory that is available in typical workstations. The surface area of Earth's land mass is approximately 150,000,000 km². To cover this area with 1m resolution imagery would require an image of 150 terapixels. When subdivided into 256x256 pixel tiles, this would require over 2 billion tiles in the image.

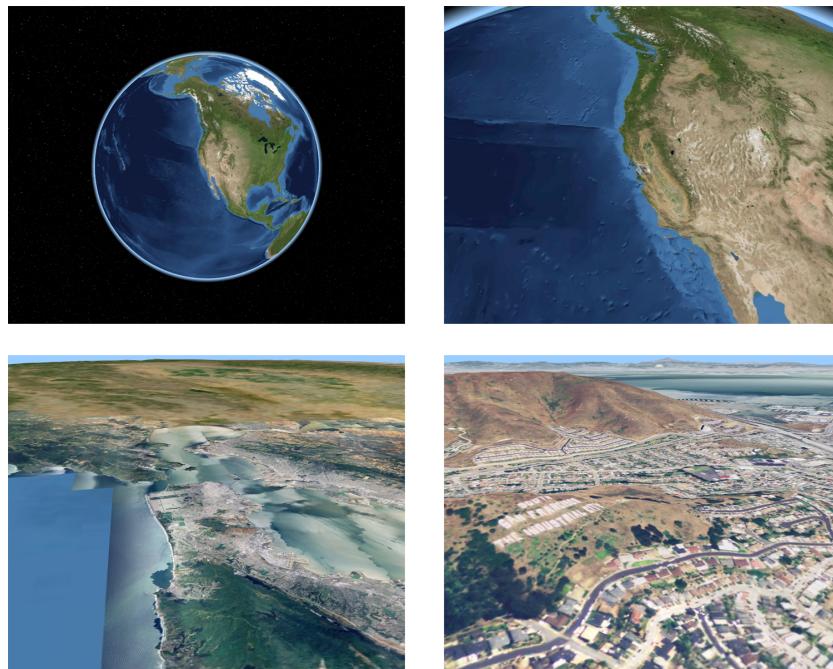


Figure 46 - Sequence of images showing the large range of detail levels required for high resolution 3D terrain at a global scale

Hence, applications need to employ [multi-resolution](#) techniques and on-demand loading for the data. This is where multi-level image mosaics are particularly useful.

Sparse tile grids allow for optimal data storage Another important aspect in this context is that image data providers typically do not offer data with a uniform resolution across the globe. Aerial photography, for instance, may be acquired with sub-meter resolution in populated areas, but not over the oceans.

Although it can be tiled internally, an `ALcdBasicImage` is always completely populated with pixel data of a uniform resolution. An `ALcdImageMosaic`, on the other hand, can consist of tiles with varying resolution *and* does not require all tiles to be present. This allows for optimized data storage, in which tiles are omitted in areas where no data is available.

In addition, `ALcdMultilevelImageMosaic` combines support for sparse data with support for multi-resolution representations. Thus, a multilevel mosaic can describe, for instance, high-resolution insets in low-resolution base data. Popular data sources such as Google Maps, Bing Maps, or OpenStreetMap use hierarchical tile pyramids that follow this principle. They can therefore be modeled as an `ALcdMultilevelImageMosaic`.

Determining the right level of detail in a multi-level image

Multi-level images are essential for working efficiently with large sets of image data. You can select the image with the right level of detail by checking the image's pixel density. The right level of detail depends on the context: when you are visualizing an image, for instance, the most appropriate level of detail is the one that most closely matches the current scale of the view. Users of an image typically perform a discrete sampling of pixel values at a certain sampling rate. If the sampling rate is too high with respect to the resolution of the image, the sampling will exhibit a staircase or pixelation effect, as neighboring samples get the same values. If the sampling rate is too low, it will miss details.

As an example, consider a multi-level image of 1000m by 1000m in a UTM grid system, as described in [Section 30.1](#). Suppose it contains three simple images: the first one with 100x100 pixels, the second one with 500x500 pixels, and the third one with 1000x1000 pixels. Consequently, those images have pixel densities of 0.01, 0.25, and 1.0 pixels/m², respectively.

Now, suppose that an algorithm wants to sample pixel values at a rate of 1 sample every 10m (0.1 samples/m) along a line in the given coordinate system. First, the algorithm must select the best image for the job, based on the required pixel density: ideal pixel density = sampling rate². In this case, the ideal pixel density would be $0.1^2 = 0.01$ pixels/m². This means that the first image is the perfect choice for the algorithm.

In practice, the image with the ideal pixel density is typically not available. Instead, an available image that approaches the ideal density is selected. This selection is often based on a few criteria such as required quality and the acceptable amount of data to be loaded. Both `TLcdGXYImagePainter` and `TLspRasterStyle` provide settings to adjust these criteria. See the reference documentation of these classes for more detail.



The sampling is often performed in a reference system different from the coordinate system of the raster. If this is the case, you must transform the sampling reference system to the coordinate system of the raster. Otherwise you will not obtain comparable values. Since the transformation is not necessarily linear, the transformed sampling rate is generally just a numerical approximation that is only valid around a given location.

12.3 Creating and applying image operators

`LuciadLightspeed` provides various `ALcdImageOperator` operators that can be applied to images, such as cropping, convolution, compositing of two images, and so on. Operators take a number of inputs, typically one or more images and a number of other parameters, and produce a new `ALcdImage` as output.

Operators can be chained together to form an `ALcdImageOperatorChain`. Such operator chains allow you to apply complex sequences of operations to images in one go.

12.3.1 Creating operators

You can apply image operators in two ways:

- Use the static methods that are available in all concrete `ALcdImageOperator` classes in the `com.luciad.imaging.operator` package. [Program 62](#) illustrates this for an operator that creates an image consisting of the first band of a multi-band image. This way of creating operators is the most convenient, and type-safe, when creating operators programmatically.

```

1 ALcdImage input = ...;
2 ALcdImage output = TLcdBandSelectOp.bandSelect(input, new int[]{0});

```

Program 62 - Applying a band select operator to an image

- Use the operators' generic `apply()` methods. The `apply()` method accepts an `ILcdDataObject` as input. Through this `ILcdDataObject`, all input parameters of the operator are supplied. The operator's `getParameterDataType()` method returns the `TLcdDataType` that the `apply()` method expects.

[Program 63](#) illustrates how to use the data type to apply an operator to an image. This approach to creating operators allows for introspection of the operator's parameters, which is useful when operators need to be serialized/de-serialized or created interactively via a GUI.

```

1 ALcdImage input = ...;
2 ALcdImageOperator operator = new TLcdBandSelectOp();
3 ILcdDataObject parameters = operator.getParameterDataType().newInstance();
4 parameters.setValue(TLcdBandSelectOp.INPUT_BANDS, new int[]{0});
5 parameters.setValue(TLcdBandSelectOp.INPUT_IMAGE, input);
6 ALcdImage output = operator.apply(parameters);

```

Program 63 - Creating an operator from its data type

12.3.2 Chaining operators

As indicated in [Section 12.3](#), operators can be chained together by providing the output image of one operator as input to the next. The most convenient way to create linear chains of operators is to use the builder provided by `ALcdImageOperatorChain`. This builder allows you to quickly append a series of image operators to one another. You can then apply such a series of operators to any number of images by passing them to `ALcdImageOperatorChain.apply()`.

To create more complex operator graphs, which work on multiple images for instance, you can create custom implementations of `ALcdImageOperatorChain`.

12.3.3 Available operators

This section provides more background information about each of the available image operators. For more detailed development information, code examples, and illustrations, see the API reference documentation of the `com.luciad.imaging.operator` package

TLcdBandMergeOp: merging the bands of multiple images

You can use `TLcdBandMergeOp` to take the bands of multiple images, and merge them into a single output image. This operation is typically used to create one multi-band image from several single-band images. For instance, you can take three single-band grayscale images, and merge them into one image with each grayscale band mapped to an RGB color channel.

The operation will sequentially add the image bands in the order in which they occur in the images: the first band of input image 2 is added after the last band of input image 1, and so on. You can merge up to four image bands.

TLcdBandSelectOp: creating an image from image bands

You can use `TLcdBandSelectOp` to select one or more color bands from an image, and display just the selected bands in an output image. You can specify the bands you want as an array of

integers, and re-order the selected bands in the array. If you select all available color bands, the bands will be mapped to the Red-Green-Blue (color channels).

TLcdBinaryOp: applying binary operations to two images

Using `TLcdBinaryOp`, you can perform binary operations on two images: by specifying two perfectly overlapping input images, you can add, subtract, multiply, and divide the values of pixels with an identical position in the two images. Using the MIN and MAX operations, you can also select the pixel with the highest or lowest value, and use that in the output image.

These binary operations are often part of image processing sequences intended to enhance images in some way. If you have an unclear image, for instance, you could try adding a second image, and perform binary operations on the two images to enhance the clarity of the first image.

TLcdColorConvertOp: changing the color model of an image

The colors of an image are specified in a certain way, determined by the color model linked to the image. Such a color model may be a simple indexed color model, a grayscale color model, or an RGB model, for example. To be able to display an image in different environments, you may need to convert the color model linked to the image to another color model.

To transform the color model of an input image to a different color model, use `TLcdColorConvertOp`. The operation takes care of all the required transformations to convert the indexed color model of an image to the standard RGB color space (sRGB), for example.

For a full list of color space conversion options, see the API reference documentation

TLcdColorLookupOp: transforming image colors

You can change the colors in your images to other colors. For this purpose, LuciadLightspeed allows you to set up color lookup tables consisting of color mappings for a color transformation. Next, you can transform the colors in your images to their corresponding colors in the lookup table. This transformation operation is called a color lookup.

To perform a color lookup operation on an input image, use `TLcdColorLookupOp`. It transforms the colors in the input image to the corresponding colors in a three-dimensional `ALcdColorLookupTable`. The output image displays the transformed colors.

TLcdCompositeOp: combining two images into one

`TLcdCompositeOp` combines two images into one image, so that the resulting image covers the union of the bounds of the input image. The second image overlaps the first image, and may be re-sampled during the image compositing operation.

You can use this operation to overlay two partially overlapping images, and create an image that covers a larger area.



`TLcdCompositeOp` alters the `ILcdBounds` of an image, and should therefore not be used when you are working with `ALcdImageMosaics`. This is because the tiles in an image mosaic must have perfectly aligned spatial bounds.

TLcdConvolveOp: changing an image through convolution

Convolution is a mathematical operation that is frequently applied in image processing to change an image comprehensively. It takes two sets of input values and puts them into linear combinations to produce output values. In LuciadLightspeed image processing, you can combine the pixel values of an input image linearly with a two-dimensional array of numbers, also known as a convolution kernel, to produce an output image.

A `TLcdConvolveOp` operation can be of use for image sharpening, blurring, edge detection, and so on.

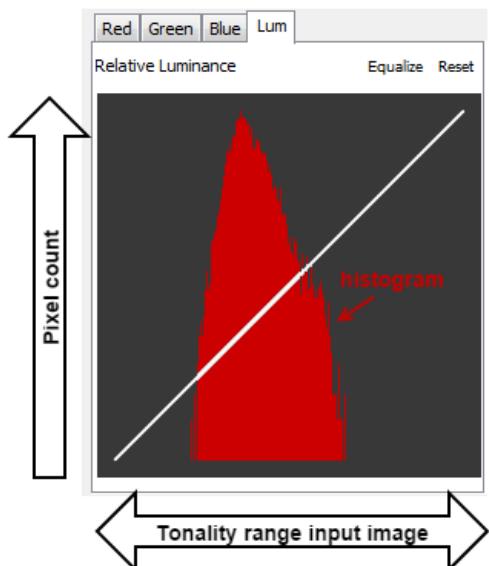
TLcdCropOp: cropping an image

The cropping operation reduces the size of your image by retaining only the selected rectangular portion of an image, and discarding the rest. In a `TLcdCropOp`, you can define the position and size of the rectangle that needs to be kept.

Keep in mind that after a cropping operation, the spatial bounds of the new image may be different from the original image. If you want to bring the cropped image back to the size of the original image, use the `TLcdExpandOp` operation. This can be useful when you are working with a `ALcdImageMosaic` object. In an `ALcdImageMosaic` grid, you are not allowed to change the bounds of the `ALcdBasicImage` objects constituting an image mosaic.

TLcdHistogramOp: creating a histogram of an image

An image histogram is a graph that displays the tonal range of an image on its horizontal axis, and the distribution of tones across that range. Each histogram bin in the graph displays the concentration of pixels for that pixel value, and as such the intensity of that tone in the image.



**Figure 47 - An image luminance histogram generated in the
lightspeed.imaging.multispectral sample**

An image's tonality varies from dark to bright for an entire image, or from high color intensity to low color intensity for a color channel.

To generate such a histogram for an image, use the image as input for a `TLcdHistogramOp` operation. The result is a one-dimensional `ALcdImage` of the histogram.

TLcdCurvesOp: using curves to change image color tones or brightness

Changing the distribution of tones across an image or image band can improve its contrast and luminance. Curves provide a mapping for such a re-distribution of tones in an image or image band. They map the tonal distribution of an input image to the tonal distribution of the image resulting from the curves operation.

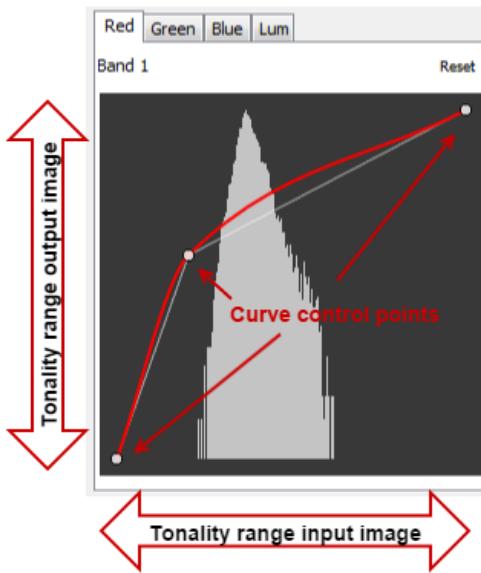


Figure 48 - Using curves for the band mapped to the red channel in the `lightspeed.imaging.multi-spectral` sample

Curves are the graphical representation of the mathematical function that describes the relationship between the tonal range of an input image and the tonal range of an output image. They are drawn on a graph displaying the input tonal range on the horizontal axis and the output tonal range on the vertical axis. To start with, the curve will always be a straight line: the tonal distributions of the input and output image are identical.

The curves are defined by their control points. To re-distribute the tonal range of an image, you can change the control points of the curve so that the curve is adjusted, and the tonal distribution shifts to a different part of the tonal range in the output image. If you adjust the luminance curve for an image, the luminance of the output image changes, and the image will become darker or brighter. If you mapped an image band to a color channel, you can adjust the curve on the color channel, and emphasize or de-emphasize certain colors.

The `TLcdCurvesOp` operation allows you to specify the position of the control points for the curves, and the curve type, for one or more image bands. You can choose between two curve types, determined by the mathematical function used to draw the curves: piecewise linear or Catmull-Rom.

TLcdExpandOp: expanding an image

In an expand operation, you make an image larger so that it covers a specified pixel area. To specify the new size of the image, you provide the X and Y coordinates for the new pixel area, as well as its width and height.



`TLcdExpandOp` alters the `ILcdBounds` of an image and should be used with care when you are working with `ALcdImageMosaic` objects. This is because the tiles in an image mosaic should have perfectly aligned spatial bounds.

TLcdIndexLookupOp: transforming the colors of a single-band image

The `TLcdIndexLookupOp` operation is similar to `TLcdColorLookupOp`, but instead of using a three-dimensional color lookup table to transform image colors, you use a one-dimensional

lookup table. This is useful to map and transform the colors of images with just one band.

For a demonstration, see [Section 12.3.4](#).

TLcdMedianOp: applying a median to pixel values

The `TLcdMedianOp` median operator computes the median for an area around each pixel in an image, and then replaces the center pixel value with the calculated median value. You need to specify the area around the pixel as a square with odd dimensions, for example a window with both a width and height of 3. Median filters are useful to reduce noise in an image.

TLcdPixelRescaleOp: applying a scale and offset to pixel values

You can perform a `TLcdPixelRescaleOp` operation on an image to apply a scale and offset to all its pixel values. For instance, if you set a scale of 0.5, a pixel value of 100 will be scaled to a pixel value of 50. As a result of the rescaling operation, the colors or brightness of the entire image shift linearly.

TLcdPixelTransformOp: applying an affine transformation to pixels

The `TLcdPixelTransformOp` operation is similar to the `TLcdPixelRescaleOp` operation: you can apply a scale and offsets to all pixel values in an image, but the result will be an affine transformation instead of a linear transformation. Instead of providing a single value to rescale pixel values with, you can provide a matrix of values. The pixel values in the image will be multiplied with the entire matrix, before any additional offsets are applied. This increases the number of transformations you can apply to a pixel.

TLcdResizeOp: changing the resolution of an image

To change the resolution of an image, use the `TLcdResizeOp`. Changing the resolution of an image allows you to adapt the image resolution to the zoom level, for instance. You can display lower image resolutions when a user has zoomed out of the map, and higher image resolutions when a user zooms in.

This image operator allows you to specify double values as scales for adjusting the resolution along both the X axis and the Y axis of the image.

Note that the bounds of the image do not change during this operation, and that the output image covers the same area on the map as the input image.

TLcdSemanticsOp: Adjusting the semantics of an image

Use `TLcdSemanticsOp` to re-interpret the semantics of an image band, like its data type for example. You can also use the semantics operation to convert an image's measurement semantics to color semantics, as demonstrated in [Section 12.3.4](#).

While you are changing the semantics, you can also scale and translate the semantic values.

For more information about band semantics, see [Section 12.2.1](#).

TLcdSwipeOp: swiping between two images

The `TLcdSwipeOp` operation overlays two `ALcdBasicImage` objects, and adds a line across the resulting image. On the left side of the line, the first image is visible. On the right side of the line, the second image is visible.

You determine the position of the swipe line by specifying a distance in pixels from the left hand side of the image.

You can use this operation to add an image swiping function in your application. It allows a user to find differences between two images captured at the same location, but at a different point in time, for example.



Figure 49 - Swiping between an image taken in 2000 and an image taken in 2003



If you are working in a Lightspeed view, and want to set up a swipe operation between two images that reside in different layers or have distinct references, it is recommended to use a `TLspSwipeController` instead. For more information, see [Section 26.2.7](#).

12.3.4 Image processing operator examples

This section demonstrates how image operators can be used as building blocks for practical image processing.

>Selecting 3 bands from a 7-band LandSat image as RGB

When you are using 7-band Landsat imagery, you typically want to visualize it as Near infrared, or the standard false color composite. This requires the use of bands 4, 3, and 2, where band 4 corresponds to the near infrared band. In this combination, vegetation appears in shades of red.

This use case is illustrated in [Program 64](#). First you use a band select operator to select the desired bands. Because the LandSat7 image contains measurement data, the result is an image with 3 measurement bands. To visualize the measurement data, we need to convert it to an image with color data. So in the second step you use a semantics operator to convert the image to an RGB image by simply using one band for each of the color channels.

```

1 ALcdImageOperatorChain chain = ALcdImageOperatorChain
2     .newBuilder()
3     .bandSelect(new int[]{3, 2, 1})
4     .semantics(
5         TLcdBandColorSemanticsBuilder.newBuilder()
6             .colorModel(ColorModel.getRGBdefault())
7             .buildSemantics()
8             .subList(0, 3)
9     )
10    .build();

```

Program 64 - Image operator for visualizing a 7-band LandSat image as Near infrared.

Attaching a color table to an image with elevation data

When you are using elevation data, you typically visualize it by mapping the elevation values to colors using a color table.

This is illustrated in [Program 65](#). First you create the color map covering the entire range of the image's data type. Then you use this in a color lookup operator. The operator uses the elevation values to choose a color from the table. The result is an image with color data that can be visualized by the painter.

```

1 // Create a color map
2 Color sea = new Color(0x6AB2C7);
3 Color grass = new Color(0x5EBC5A);
4 Color hills = new Color(0x9B6E40);
5 Color snow = new Color(0xB5A4BA);
6 TLcdColorMap colorMap = new TLcdColorMap(
7     new TLcdInterval(Short.MIN_VALUE, Short.MAX_VALUE),
8     new double[] {Short.MIN_VALUE, 0.0, 100.0, 500.0, 2000.0, Short.MAX_VALUE},
9     new Color[] {sea , sea, grass, hills, snow , snow}
10 );
11
12 // Create the operator chain
13 ALcdImageOperatorChain chain = ALcdImageOperatorChain
14     .newBuilder()
15     .indexLookup(
16         TLcdLookupTable.newBuilder().fromColorMap(colorMap).build()
17     )
18     .build();

```

Program 65 - Image operator for visualizing a elevation data using a color table.

12.4 Image processing during visualization

You can apply and update the discussed image operators interactively when you are visualizing image data.

12.4.1 Applying image operators in a Lightspeed view

To apply image operator chains to images in a layer, you can use a `TLspImageProcessingStyle` in conjunction with `TLspRasterLayerBuilder`. This is illustrated in [Program 66](#).

```

1 // Decode the model
2 ILcdModel model = decoder.decode("input.tiff");
3
4 // Create the operators and style
5 ALcdImageOperatorChain ops = ALcdImageOperatorChain.newBuilder()
6     .convolve(new double[]{-1,-1,-1,-1,9,-1,-1,-1,-1}, 3, 3)
7     .build();
8
9 TLspImageProcessingStyle style = TLspImageProcessingStyle.newBuilder()
10    .operatorChain(ops)
11    .build();
12
13 // Create a raster layer for the given model and apply the filter style
14 TLspRasterStyle rasterStyle = TLspRasterStyle.newBuilder().build();
15 TLspStyler styler = new TLspStyler(rasterStyle, style);
16 ILspLayer layer = TLspRasterLayerBuilder.newBuilder()
17     .model(model)
18     .styler(REGULAR_BODY, styler)
19     .build();

```

Program 66 - Applying a filter as a style on a raster layer

Note that you must use `TLspImageProcessingStyle` in conjunction with `TLspRasterStyle`. You can use `TLspRasterStyle` to adjust the brightness, contrast, and opacity of the result of the processed image, for example.

12.4.2 Applying image operators in a GXY view

To visualize processed images in a GXY view, use the `TLcdGXYImagePainter`. You can use the `setImageOperatorChain()` method to apply any `ALcdImageOperatorChain` to the images visualized in the layer.

12.5 Offline processing

In some cases, access to the pixel data of an image is required for reasons other than visualization. You may want to query pixel values for data inspection, for example, or you want to decode, process, and encode image data on disk.

12.5.1 Accessing pixel values through `ALcdImagingEngine`

The main building block of all `ALcdImage` implementations is `ALcdBasicImage`. As discussed in Section 12.2.2, `ALcdBasicImage` is an opaque handle to image data. To access the data, you need an imaging engine. To create an engine, you can use the `createEngine()` factory method in `ALcdImagingEngine`. Imaging engines implement `ILcdDisposable`. Hence, they need to be disposed after use. This is illustrated in Program 67.

```

1 ALcdImagingEngine engine = ALcdImagingEngine.createEngine();
2 try {
3     ...
4 }
5 finally {
6     engine.dispose();
7 }

```

Program 67 - Creation and disposal of an imaging engine

You can make multiple implementations of imaging engines available, for example, if multiple compute devices (CPU, GPU) are found. To list all available imaging engine implementations,

use `ALcdImagingEngine.getEngineDescriptors`. A Descriptor reports whether an engine is hardware-accelerated, and evaluates operators on the GPU. It also reports whether an engine can be used for tasks other than visualization. You can create an engine for a specific descriptor using the `ALcdImagingEngine.createEngine(Descriptor)` method.

Once an engine has been created, you can query the content of images using the `getImageDataReadOnly()` method. This is illustrated in [Program 68](#).

```

1 ALcdImagingEngine engine = ALcdImagingEngine.createEngine();
2 try {
3     ALcdBasicImage input1 = ...;
4     ALcdBasicImage input2 = ...;
5     ALcdBasicImage output = (ALcdBasicImage) TLcdBinaryOp.binaryOp(input1, input2, ADD);
6     int width = output.getConfiguration().getWidth();
7     int height = output.getConfiguration().getHeight();
8     Raster outputData = engine.getImageDataReadOnly(output, new Rectangle(0,0,width,height));
9     float[] pixel = new float[1]; // Assuming a single band image with float pixel data
10    for (int j=0; j<height; j++) {
11        for (int i=0; i<width; i++) {
12            outputData.getPixel(i,j,pixel);
13            System.out.println("Pixel at " + i + "," + j + ":" + pixel[0]);
14        }
15    }
16 }
17 finally {
18     engine.dispose();
19 }
```

Program 68 - Accessing the pixel values of an `ALcdBasicImage`

Imaging engines are safe for use on multiple threads.

12.5.2 Encoding processed images to disk

Processed `ALcdBasicImages` and `ALcdMultilevelImages` can be encoded back to disk with `TLcdGeoTIFFModelEncoder`. [Program 69](#) illustrates this for an operator that applies a convolution kernel to a decoded GeoTIFF image.

```
1 // Decode the GeoTIFF file
2 TLcdGeoTIFFModelDecoder decoder = new TLcdGeoTIFFModelDecoder();
3 ILcdModel model = decoder.decode("input.tiff");
4
5 // Convolve the first element
6 ILcdModel processedModel = new TLcdVectorModel(model.getModelReference(), new
7 TLcdImageModelDescriptor());
8 Enumeration e = model.elements();
9 if (e.hasMoreElements()) {
10     Object o = e.nextElement();
11     if (o instanceof ALcdImage) {
12         // Convolve the image using a 3x3 kernel
13         ALcdImage convolved = TLcdConvolveOp.convolve(
14             (ALcdImage) o,
15             new double[]{-1,-1,-1,-1,9,-1,-1,-1,-1},
16             3,
17             3
18         );
19         processedModel.addElement(o, ILcdModel.NO_EVENT);
20     }
21 }
22 // Create an imaging engine and encode the filtered result
23 ALcdImagingEngine engine = ALcdImagingEngine.createEngine();
24 try {
25     TLcdGeoTIFFModelEncoder encoder = new TLcdGeoTIFFModelEncoder(engine);
26     encoder.export(processedModel, "output.tiff");
27 } finally {
28     engine.dispose();
29 }
```

Program 69 - Decoding, processing, and encoding the result

CHAPTER 13

Modeling raster data as `ILcdRaster` objects

This chapter discusses the `ILcdRaster` and `ILcdMultilevelRaster` interfaces. Just like the `ALcdImage` class and its variants, they represent grids of pixels with geographical bounds. The `ILcdRaster` interfaces and their implementations pre-date `ALcdImage`, however. The `ALcdImage` classes offer more functionality, as they allow working with multi-band and HDR imagery, and let you apply the image operators discussed in [Section 12.3](#).



If you want to process or access decoded data, it is recommended to work with `ALcdBasicImage` and its related classes, as opposed to working with the `ILcdRaster` or `ILcdMultilevelRaster` interface. You can use `TLcdImageBuilder` to create `ALcdImage` objects.

Note that `ALcdImage` and `ILcdRaster` and the related classes are compatible in the sense that a single Java object can extend from `ALcdImage` and implement `ILcdRaster` at the same time. All model decoders that produce raster data provide domain objects that do so.

13.1 What is an `ILcdRaster`?

An `ILcdRaster` is a basic extension of `ILcdBounded` and represents a 2D rectangular area consisting of rows and columns with equally sized **tiles**. Each tile (`ILcdTile`) is a matrix of integer values. The tiling structure allows a fast retrieval of data for a specific area in a specific resolution. [Section 13.3](#) provides more information on the usage of `ILcdTiles`.

You can retrieve the value of a tile in two ways:

- By using the `retrieveValue` method and specifying the raster coordinates. Note that for some data types, such as elevation data, the returned value can be the result of interpolation or filtering.
- By using the `retrieveTile` method and specifying a tile row index and tile column index. The returned tile can then be queried for the actual value without interpolation or filtering. For more information on working with tiles, refer to [Section 13.3](#).

The level of detail of a raster is determined by the pixel density of the raster. The pixel density provides an estimate for the number of pixels per unit surface area in model coordinates. The higher the pixel density, the more detailed the raster. You can retrieve the pixel density by using the method `getPixelDensity`. This allows you to determine whether a raster is too

detailed or not detailed enough for visualization or for other purposes. Especially since rasters are typically loaded lazily, a simple check of the pixel density can prevent the loading of massive amounts of data with the wrong level of detail.

For visualization purposes, you can set a `java.awt.image.ColorModel` on the raster by using the `setColorModel` method. Because `ILcdRaster` implements `ILcdCache`, it is also possible to attach cached data to an `ILcdRaster`. The main implementation of the `ILcdRaster` interface is the `TLcdRaster` class. For more information on `ILcdRaster`, `TLcdRaster`, their properties and methods, refer to the API reference.

An `ILcdRaster` can also be part of an `ILcdMultilevelRaster` or an `ILcdMultivalueRaster` as described in the following sections.

13.2 What is an `ILcdMultilevelRaster`?

An `ILcdMultilevelRaster` is essentially a list of `ILcdRaster` instances covering the same area at different levels of detail. The different levels are ordered by an increasing pixel density, from the least detailed to the most detailed level.

An `ILcdMultilevelRaster` has bounds of its own, which are the union of the bounds of the rasters that it contains. The bounds of the rasters are typically identical but can be different, when rasters from different sources cover approximately the same area, for example. The main implementation of the `ILcdMultilevelRaster` interface is the `TLcdMultilevelRaster` class. For more information on their properties and methods, refer to the API reference.

13.3 What is an `ILcdTile`?

An `ILcdTile` raster tile is a matrix of integer values. A tile has a resolution expressed in pixels and a pixel size expressed in bits. There are two ways to retrieve the contents of a tile:

- As individual values, by using the method `retrieveValue` with coordinates expressed in pixels.
- As images, by using the method `createImage`. The tile contents are retrieved in bulk.

Similar to an `ILcdRaster`, an `ILcdTile` can have a `ColorModel` for visualization purposes. The major difference between an `ILcdRaster` and an `ILcdTile` is that an `ILcdRaster` is defined in model coordinates (doubles) and an `ILcdTile` in pixel coordinates (integer).

It is not required that all tiles in a raster have the same tile resolution. For example, a raster representing a DTED data set covers a rectangular area in geodetic coordinates. Each tile covers a cell of 1 degree by 1 degree, but tiles near the poles have a lower resolution than tiles near the equator.

13.4 `ILcdRaster` and `ILcdTile` sizes

The tiles in a raster usually all have the same size defined by the tile width and the tile height, in the coordinates of the associated reference system. In some cases, the tiles in the last grid column and the last grid row may be truncated to smaller sizes. This may happen if the number of columns times the nominal tile width is larger than the raster width. Or if the number of rows times the nominal tile height is larger than the raster height. For example, a raster with a size of 5000m by 2500m may have 5 by 3 tiles with nominal sizes of 1000m by 1000m. The tiles in the last row are then truncated to 1000m by 500m. [Figure 50](#) provides a schematic overview of a raster and its tiles.

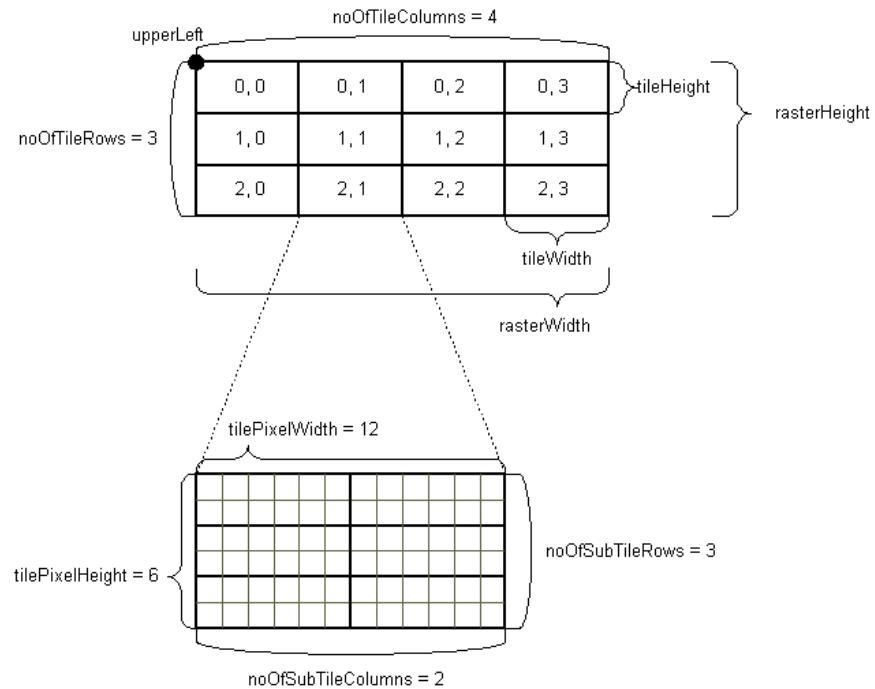


Figure 50 - Properties of an `ILcdRaster` and its tiles

CHAPTER 14

Working with Earth tilesets

The class `ALcdMultilevelImageMosaic` in LuciadLightspeed's image processing framework `com.luciad.imaging` is the recommended way of working with multi-leveled images. However, you can continue using the `com.luciad.earth` package and `ILcdEarthTileSet` objects for reasons of continuity and backward compatibility. If you need to convert vector data into tilesets, and you are not using the LuciadFusion product for this, you may also still need to use `ILcdEarthTileSet`.

For more information about working with `ALcdMultilevelImageMosaic`, see Chapter 12.

14.1 Earth tilesets

A tileset is a dataset which has been prepared into a hierarchical tiling structure. In the LuciadLightspeed Earth package, it is defined by the interface `ILcdEarthTileSet`.

A tileset provides multiple levels of detail, each of which contains a regular grid of tiles. Level 0 is the least detailed, and each subsequent level doubles the numbers of rows and columns in the tile grid. The combined levels form a multi-resolution tile pyramid in which one tile on level N corresponds to a block of 2x2 tiles on level N+1.

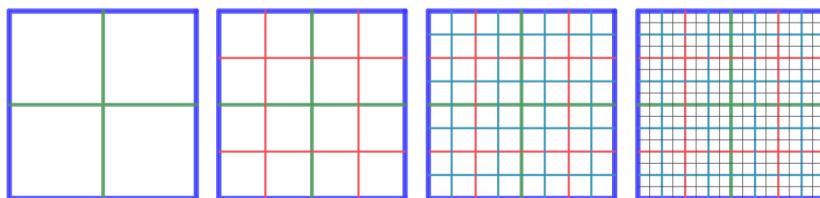


Figure 51 - Detail levels in an `ILcdEarthTileSet`

The tile grid is defined in a 2D coordinate system, a geodetic reference or a grid reference for example, and its bounds. Each level covers the exact same geographic area, but does so using an increasingly large number of tiles. All tiles contain approximately the same amount of data, but the higher levels contain more detailed data than the lower levels.

A tileset may load tiles from a disk or some other storage medium, or it may generate tiles on the fly based on some other data source. The tileset may process requests on a separate thread

of execution, so that the application does not pause or stutter while tiles are being retrieved. This is important to obtain a fluid and responsive visualization.



Although the tileset defines tiles as being laid out in a regular grid, it does not require that every cell in the grid is effectively populated. This allows tilesets to create sparse data structures, which is useful to create high-resolution insets in low-resolution base data, for instance.

The contents of the tiles are not specified by the tileset. The tileset interface does not enforce that tiles contain data of a regularly gridded nature, for example. Tiles may also contain 3D representations of geographic features, such as buildings, roads and vegetation, 2D or 3D raster data, such as weather data, or even non-visual data, for example textual descriptions of the area underlying the tile.

For a 3D terrain, each tile may contain a 3D geometric representation of the terrain elevations, derived from regularly spaced sample locations in a digital elevation model, and a 2D image to be used as the texture map for that terrain tile.

14.1.1 Earth tileset coverages

An `ILcdEarthTileSet` contains one or more *data coverages*. Each data coverage is represented by a `ILcdEarthTileSetCoverage`, and all data coverages of a single tileset share the same tiling structure. The coverages describe different sets of data that are available from the containing tileset. Therefore, when a tile is requested from an `ILcdEarthTileSet` it is always for one particular coverage, which is passed to the tileset as an argument. Each coverage has a name which must be unique within its containing tileset.

Each `ILcdEarthTileSetCoverage` has a coverage type, which is one of the following:

- **ELEVATION**: the coverage contains gridded elevation data.
- **MULTIVALUED**: the coverage contains multidimensional scalar values.
- **IMAGE**: the coverage contains image data.
- **GEOMETRY**: the coverage contains geometric primitives: points, lines, polygons, 3D meshes, and so on.
- **OTHER**: the coverage contains non-visual data.

14.1.2 Retrieving tiles from a tileset

A tile is obtained from a tileset by invoking the `produceTile()` method. This method takes the following arguments:

- The level from which a tile is desired
- The column and row indices of the desired tile, starting from the origin of the tile grid in the lower left corner
- The `ILcdEarthTileSetCoverage` from which a tile is desired
- The geographic reference in which the resulting tile should be defined - this is explained in more detail in the next section.

- The format in which the tile is to be returned - the format is specified as a `TLcdEarthTileFormat` and determines the type (java class) of the data object in the requested tile.
- A mode that steers the asynchronous behavior of the tiles
- A callback that is to be invoked when the tile is ready
- A `context` parameter that can be used to pass application-specific data into the tiles

The callback mechanism is what allows tiles to operate asynchronously. If a tiles supports multithreaded operation, `produceTile()` only schedules a request to be executed at a later time. When the tile becomes available, the callback is invoked in order to return it to its requestor. The mode parameter of `produceTile()` allows you to specify whether you want asynchronous behavior (`TLcdEarthTileOperationMode.PREFER_ASYNCROUS`) or not (`TLcdEarthTileOperationMode.FORCE_SYNCROUS`).

The `ILcdEarthTileSetCallback` interface contains two methods. The first, `tileAvailable()`, is used to notify the requestor that a tile has become available for use. The argument of this method is an `ALcdEarthTile` object that contains the requested tile data. The second method in the callback is `tileNotAvailable()`. This method is used to inform the requestor that a tile could not be produced. This mechanism allows for the creation of sparse tile grids, in which not all areas are necessarily subdivided to the same level of detail.

If a requested tile is available, it is returned in the form of an `ALcdEarthTile` object.

14.2 Using Earth tilesets to work with 3D terrain

A 3D terrain uses elevation data and image data as input. Both the elevation data and the image data consist of regularly spaced sample points, so that the terrain can be trivially split into tiles of equal sizes, 64x64 values per tile for the elevation and 256x256 pixels per tile for the images, for example.

The basic steps to visualize 3D terrain with the `com.luciad.earth.*` packages are:

- Pre-processing
 - Collect source data: elevation models from which to derive the terrain elevations, and raster data to use as terrain images
 - Preprocess the source data to create a terrain tile repository on disk.
- Visualization
 - Open the terrain tile repository in the application
 - Add a layer with the repository to view



A tile repository is an on-disk backing store for tiles. Once the source data has been processed into a set of tiles, the tiles are written to a tile repository, from which other tilesets can retrieve them.

These steps are explained in more detail in the following sections.

14.2.1 Pre-processing to generate a 3D terrain

Pre-processing work flow



The tools described below for preprocessing of 3D terrain data are available with LuciadLightspeed out of the box. However, the LuciadFusion product provides data management tools that are both more powerful and easier to use. The use of LuciadFusion to prepare 3D terrain data is particularly recommended when very large amounts of data need to be processed and/or when the data needs to be updated after the initial preprocessing step.

The general work flow when working with `ILcdEarthTileSet` is as follows:



Figure 52 - Work flow when using `ILcdEarthTileSet`

When pre-processing 3D terrain, the work flow is extended as follows:

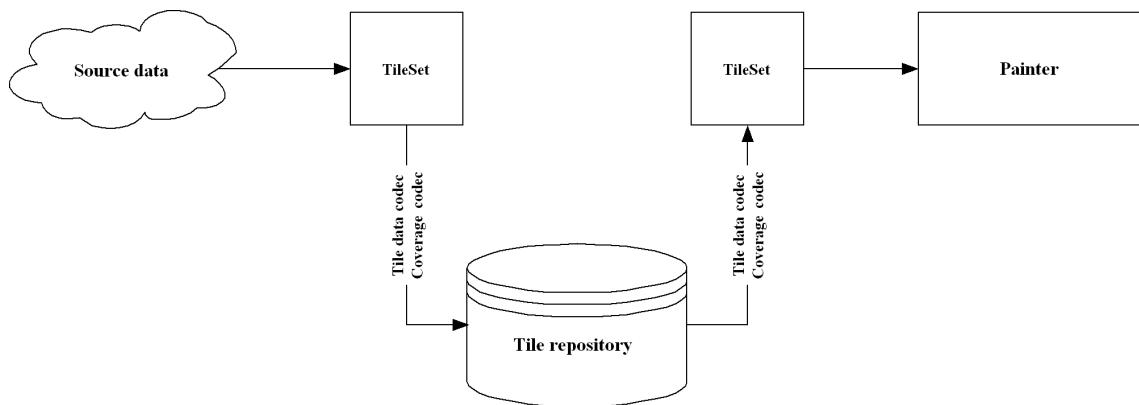


Figure 53 - Work flow when pre-processing 3D terrain

Figure 53 shows the following steps:

1. A tileset is created which produces terrain imagery and elevation tiles based on the provided source data. This tileset may perform resampling, transformation or other operations on the data which are too expensive for the application to perform on the fly.
2. Tiles from this tileset are written into a tile repository using a codec mechanism.
3. Another tileset can read the tiles back from the tile repository using those same codecs.
4. From there, the application can use the tiles for visualization. The tiles are delivered in a ready-to-use format, so the application does not need to repeat the expensive processing steps performed by the original tileset.

Working with tile repositories

The central component in the terrain pre-processing work flow is a `TLcdEarthTileRepository`. As illustrated in the previous section, a tile repository acts as a backing store for a

tileset. `TLcdEarthTileRepository` implements the `ILcdEarthEditableTileSet` interface. These methods transparently read or write data from or to the repository, thus providing a simple persistence framework for the output of a tileset. `TLcdEarthTileRepository` also implements the `ILcdEarthIterableTileSet` interface, allowing iteration over its tiles.

`TLcdEarthTileRepository` stores tiles in a database on disk using a codec mechanism. The high-level data structure and indexing mechanisms of this database are defined by `TLcdEarthTileRepository` itself, but the encoding format of the individual tiles is defined by the codecs.

You can use the `TLcdEarthRepositoryModelDecoder` to open an existing repository and the `TLcdEarthRepositoryModelFactory` to create a new repository. These classes do not only open or create the database but also read and write the repository model reference and the asset model that describes its contents.

Before a repository can be used, for each `ILcdEarthTileSetCoverage`, an appropriate `ILcdEarthTileDataCodec` needs to be registered. The tile data codec is responsible for encoding and decoding the contents of tiles in a coverage. The output of the tile data codec is what is eventually written into the repository. To register a codec, call the repository's `addTileDataCodec()` method, with the name of the `ILcdEarthTileSetCoverage` and the codec itself as arguments. The coverage name must be unique.

In the case of a 3D terrain repository, for instance, the terrain elevation is in a coverage with `TLcdEarthElevationData` as its data format. In this case, a `TLcdEarthElevationDataTileDataCodec` can be registered for that coverage. Similarly, a `TLcdEarthImageTileDataCodec` or `TLcdEarthSeparateAlphaImageTileDataCodec` can be registered to handle the terrain image tiles, which use `BufferedImage` as their data exchange format. The first codec can only encode opaque terrain texture tiles while the latter also supports (partially) transparent terrain texture tiles. Other `ILcdEarthTileDataCodec` implementations can be found in the `com.luciad.earth.repository.codec` package. Custom-written codecs can also be plugged in, so that `TLcdEarthTileRepository` can be used with any tileset regardless of the data formats it uses.

The repository must also contain all the `ILcdEarthTileSetCov`erages from which you wish to store tiles. Data coverages can be added to the repository using the `addTileSetCoverage()` method. This step is only necessary the first time you want to add tiles from that coverage. When you later reopen an existing repository, the data coverages will already be there.

Once the repository has been opened and codecs have been registered, it can be used to read or write tiles. This is as easy as calling `produceTile()` or `addTile()`, respectively. The `addTile()` method takes an `ALcdEarthTile` as its argument and writes it to the repository, using the appropriate `ILcdEarthTileDataCodec` to serialize the tile's data. The `produceTile()` method takes the `ILcdEarthTileSetCoverage`, the level and the row and column indices of the tile as arguments, and returns the corresponding `ALcdEarthTile` if it exists.

Note that in order to minimize disk access, `TLcdEarthTileRepository` may buffer up multiple `addTile()` calls. To ensure that all submitted tiles are effectively written to the disk, the application must call the `commit()` method. When the application finishes working with the repository, it should call the `dispose()` method to ensure that all resources held by the repository, such as file locks, are released.

Configuring the repository for tileset compatibility When pre-processing data to a repository the configuration of this repository must be chosen carefully to avoid a performance penalty

when combining this data later on-the-fly with other tilesets. Such a situation occurs for example when compositing raster data on-the-fly on top of a 3D terrain repository.

To ensure optimal performance in these situations all repositories must have the same general structure. It is advised to use the same structure as other Luciad products that use Earth, Lucy for example. The `TLcdEarthRepositoryModelFactory` can be used to create a repository that has this advised structure:

- model reference: `TLcdGeodeticReference`
- bounds: from $-90^{\circ}, -180^{\circ}$ to $90^{\circ}, 180^{\circ}$. The entire earth surface, for example.
- #levels: 24
- #rows at level 0: 2
- #columns at level 0: 4

Defining Earth asset metadata

When building a terrain repository, the terrain pre-processor requires some metadata about the source data used to construct the 3D terrain. Each item of source data is represented by an `ILcdEarthAsset`, and the combined collection of assets is stored in an `ILcdModel` which in turn provides the pre-processor with all the metadata it needs.

An `ILcdEarthAsset` has the following properties:

- A model decoder that can be used to decode the data represented by the asset
- A source name, which points to the data that should be decoded by the aforementioned model decoder
- The bounds of the asset
- The asset's coverage type: elevation, image or geometry, as explained in [Section 14.1.1](#)
- The asset's last modification data, which is used by the pre-processor to check if incremental updates to the 3D terrain can or should be performed.

The following implementations of `ILcdEarthAsset` are available:

- `TLcdEarthAsset` is the default implementation, which contains only the properties mentioned above.
- `TLcdEarthRasterAsset` can be used specifically for raster data and introduces a pixel density property that indicates the detail level of the asset.
- `TLcdEarthClippedRasterAsset` extends `TLcdEarthRasterAsset` with a clipping shape, which can be used to remove unwanted parts from the raster.

An asset model can be encoded and decoded to and from disk using `TLcdEarthAssetModelCodec`. This class makes use of `ILcdEarthAssetCodec` objects to convert `ILcdEarthAssets` into a set of key/value pairs, represented by a `java.util.Properties` object. These pairs are then saved in an XML-based file format, and can be used to reconstruct the asset when decoding the asset model. By saving the asset model to disk, it can be used to perform incremental updates to a 3D terrain by adding assets and synchronizing an existing 3D terrain with the new version of the asset model.

Here is an overview of the steps required to supply the pre-processor with an `ILcdEarthAsset` model:

- To create a new 3D terrain:

1. Create `ILcdEarthAsset` objects and add them to an `ILcdModel`.
 2. Encode the asset model using `TLcdEarthAssetModelCodec` for later use.
 3. Use the asset model with the terrain pre-processor.
- To update an existing 3D terrain:
 1. Decode an asset model using `TLcdEarthAssetModelCodec`.
 2. Create additional `ILcdEarthAssets` and add them to the asset model.
 3. Use the asset model with the terrain pre-processor.

Producing Earth Tilesets

This section briefly describes the `ILcdEarthTileSet` implementations that can be used to produce 3D terrain tiles:

- Create the terrain elevation tiles using a `TLcdEarthTerrainElevationTileSet`. Use the convenience class `TLcdEarthTerrainElevationProvider` to retrieve elevations from multiple raster models.
- Create the terrain textures using a `TLcdEarthGXYViewTileSet`. This class uses an offscreen `ILcdGXYView` to generate images which are then used as texture tiles. `TLcdEarthGXYViewTileSet` is an `ILcdLayered`, so standard `ILcdGXYLayers` can be added to it for inclusion in the texture images.
- Use `TLcdEarthMetadataTerrainTileSet` as a front-end for the two tilesets discussed above. It takes an `ILcdModel` containing `ILcdEarthAssets` as input, creates a `TLcdEarthTerrainElevationTileSet` and a `TLcdEarthGXYViewTileSet`, and populates both of them with the appropriate data.

The image tiles produced by these tilesets are opaque by default. This can result in visible borders if the source data does not cover the entire area of the tileset. To prevent these borders the produced images should be transparent when there is no data. This can be achieved by setting a transparent background color and image type on the `TLcdEarthGXYViewTileSet`.

Starting the pre-processor

The final step in generating a 3D terrain is to call the pre-processor itself. To recapitulate, the following steps should be performed before the pre-processing can start:

- Open or create a `TLcdEarthTileRepository`, and register the appropriate `ILcdEarthTileDataCodecs` with it.
- Use `TLcdEarthAssetModelCodec` to decode the asset model that will drive the pre-processor.
- Create a `TLcdEarthMetadataTerrainTileSet` to generate input for the pre-processor based on the asset model.

Pre-processing can now be initiated using `TLcdEarthTileRepositoryPreprocessor`. By calling the `synchronizeRepository()` method, this class iterates over all assets in the asset model, uses the tileset to generate the tiles corresponding to those assets, and then adds those tiles to the repository.

Furthermore, it keeps track of changes in the asset model: if an asset is added after a repository has already been created, the pre-processor will not rebuild the entire repository from scratch.

Instead, it only produces those tiles that are affected by the new asset. Similarly, if the pre-processing is aborted for whatever reason and restarted at a later date, it will pick up where it left off rather than starting over.

The pre-processing progress can be monitored using an `ILcdEarthPreprocessorProgressListener`, which provides detailed information about what the pre-processor is doing. The `progress()` method in the listener is invoked after every tile that is generated. This method returns a boolean value which acts as a stop condition for the pre-processor. By having it return false, the pre-processor stops working and exits cleanly.

`TLcdEarthTileRepositoryPreprocessor` has the option of registering one or more `ILcdEarthTileCombiners`. An `ILcdEarthTileCombiner` is a class that takes four tiles, consisting of a 2x2 block of adjacent tiles, and combines them into a single new one of a lower detail level. Available tile combiners are `TLcdEarthImageCombiner` for terrain images and `TLcdEarthElevationDataCombiner` for terrain elevations. If the terrain texture tiles contain 256x256 pixel images, for instance, `TLcdEarthImageCombiner` takes four such images, scales them down and then combines them into a single new 256x256 pixel images.



Figure 54 - Operation of a tile combiner

The tile combiner mechanism exists to reduce disk access during pre-processing. Pre-processing is performed top-down, starting with the highest detail level. Because the geographic area of the tiles grows larger and larger as the detail level decreases, the tiles also cover an increasingly large number of metadata assets. Eventually, the bottom detail level will require *all* assets to be processed just to generate a handful of tiles. It is therefore more efficient to let the `TLcdEarthMetadataTerrainTileSet` handle only the topmost detail level for each asset, and to use the combiner mechanism for everything below that. Thus, once the pre-processor switches to the combiner mechanism, the source assets are no longer accessed. From this point on, each tile can be generated by combining exactly four other tiles, and because the number of tiles decreases with the level of detail, the required disk access decreases as well.

14.2.2 Visualizing the tile repository

To view a previously created repository, the first step is to invoke `TLcdEarthRepositoryModelDecoder`. This decoder creates an `ILcdModel`. It has a single element, which is an `ILcdEarthTileSet` representing the tile repository. This model can then be added to the

view to visualize the repository, e.g. using `TLspRasterLayerBuilder`.

CHAPTER 15

Modeling multi-dimensional data

Typically, geospatial data has a location component: data is captured at a specific geographic location. In many cases though, geospatial data has more components than just a location component. The most common data dimension is the time dimension, which is often used in NVG, ASTERIX, and NetCDF data sets. Another common dimension is vertical position, which is often used in NetCDF data to model values recorded at distinct levels in the atmosphere.

Data that is captured along more than one dimension is called multi-dimensional data. Its data dimensions are represented by the axes of a chart, and the range of values captured in the various dimensions are represented as intervals along the chart axes.

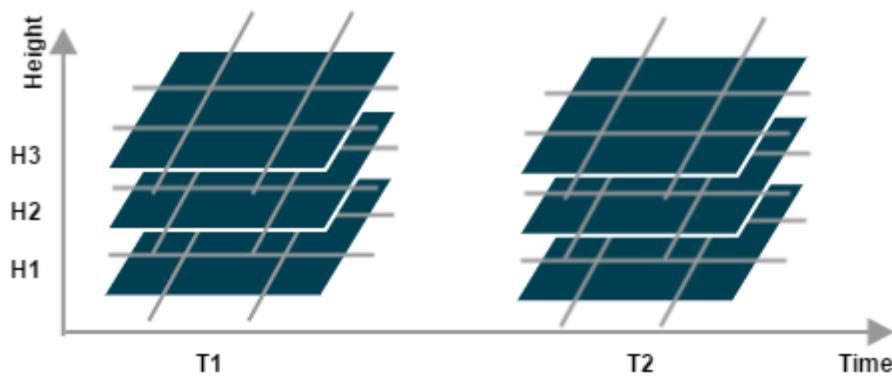


Figure 55 - Image data captured along a time dimension and a vertical position dimension

This chapter describes the API that LuciadLightspeed provides to model multi-dimensional data, and how filtering should behave for such models.

15.1 Getting the dimensions and values of multi-dimensional models

Multi-dimensional LuciadLightspeed models implement the `ILcdMultiDimensionalModel` interface. Such models advertise the dimension axes and the supported values through `ILcdMultiDimensional.getDimensions`.

15.2 Filtering multi-dimensional data

Multi-dimensional models allow you to apply a `TLcdDimensionFilter`, using `ILcdMultiDimensionalModel.applyDimensionFilter`. In such a filter, you define filtering axes and intervals and match the multi-dimensional data against those. As a result of the filtering, only the data that passes the filter will be available in the model. If you apply the empty filter `TLcdDimensionFilter.EMPTY_FILTER`, the model resets to its default dimensional filter. To learn more about the default filters for vector models and raster models, see [Section 15.2.1](#) and [Section 15.2.2](#).

You can also get the current filter by calling `ILcdMultiDimensionalModel.getDimensionFilter`. The returned filter is the one that was most recently applied to the model. If no filter was set, a default filter is returned. The returned filter may be the empty filter `TLcdDimensionFilter.EMPTY_FILTER`, but it must never be `null`.

If the returned filter defines intervals for more or fewer axes than the model supports, the model treats those filters gracefully:

- When a filter contains *more* axes than the model supports, the model ignores those axes.
- When a filter contains *fewer* axes than the model supports, the model resets its defaults for those axes.

If the multi-dimensional model is a vector model, the multi-dimensional data and its filtering is modeled slightly different than in the case of a raster model.

15.2.1 Filtering vector models

For vector models, such as NVG or ASTERIX models, the model elements themselves *are* the multi-dimensional data: the elements define the intervals for which they are valid.

When a filter is applied, the elements themselves are evaluated against the filter. Elements are added to or removed from the filtered model, and appropriate events are fired for all affected elements. If none of the elements pass the filter, the filtered model will be empty.

The default filter for vector models is typically the empty filter. As a result, all elements pass the default filter.

15.2.2 Filtering raster models

In raster models, such as NetCDF models, the model element *has* multi-dimensional data: it is an `ILcdDataObject` backed by several possible `ALcdImages`. In this case, the multi-dimensional data in this case is not the element itself, but it is the collection of backing images. The `ILcdDataObject` element itself is always present, but the image it points to may change as a result of filtering. The element can only point to one image at most, so even if multiple images pass the filter, only the first of the passing images will be made available.

If none of the elements pass the filter, the model will *not* be empty: the element itself remains, but the image it points to will be `null`.

The default filter for raster models is typically the *first* filter. As a result, only the first image passes the default filter.

15.2.3 Value filtering by interval matching

The value range of a multi-dimensional data set is defined by its intervals, for example an interval between two times on a time axis. When you apply a filter, the filter also defines a specific interval. Models match their multi-dimensional data against a given filter based on the *overlap* between the data interval and the filter interval, as defined by `TLcdDimensionInterval.overlaps`. The model matches all data valid in the intervals defined by the filter. This means that if there is the slightest amount of overlap between the two intervals, the evaluated data is considered valid, and passes the filter.

15.2.4 Locking your model for filtering

You must apply a filter from within a write lock, using `TLcdLockUtil.writeLock(model)` and event mode `ILcdModel.FIRE_LATER`. As an exception, you do not need to lock your model when you are initially creating it. In that case, there are no references to it yet, and it is safe not to lock at all and use event mode `ILcdModel.NO_EVENT`.

15.2.5 Snapping to filter intervals supported by the model

The matching of multi-dimensional data against a filter happens in a strict manner, as defined by `TLcdDimensionInterval.overlaps`. There needs to be an exact match for filtering to occur. In practice, however, the validity of multi-dimensional data is often defined as a singular interval. For example, NetCDF data often offers multiple images, each of which is valid at one specific time. That may pose some difficulties when you are implementing a filter based on a time slider, for example: a time slider slides along the continuous spectrum of the time axis. More often than not, the filter will not match anything in the model.

In such a case, you may want the filter to snap to the nearest valid multi-dimensional data. Such functionality is offered by `TLcdDimensionFilter.createSnappingFilter`. It offers a more lenient form of filter matching, and ensures that multi-dimensional data matches are *always* available for any filter. You can choose between the NEAREST, NEXT, or PREVIOUS snapping modes, or even the no-snapping mode. You can create a filter and convert it to a snapping one with `TLcdDimensionFilter.createSnappingFilter` before invoking `ILcdMultiDimensionalModel.applyDimensionFilter`. Snapping only happens when there are no strict matches. If there is at least one strict match, snapping behaves identically to the no-snapping mode.

`createSnappingFilter` does more than just snapping. These are the differences of using and not using `TLcdDimensionFilter.createSnappingFilter`:

	<code>applyDimensionFilter</code>	<code>createSnappingFilter -> applyDimensionFilter</code>
Snapping behavior	Never, possibly resulting in no matches	Snaps to intervals defined by the model if needed, such that there is always at least one match
Supported snap modes	None	Nearest, previous, next, none
Filter has fewer axes than supported by model	Reset to default for those axes	Keep the current filter value for those axes, so that there are minimal model changes
Empty filter	Reset to default filter	Keep current filter, so that there are no model changes

	applyDimensionFilter	createSnappingFilter -> applyDimensionFilter
ILcdModel which is not ILcdMultiDimensionalModel	Not supported. Throws ClassCastException.	Checks instance of ILcdMultiDimensionalModel, leaves others alone

Table 1 - Differences between applyDimensionFilter and applyAndSnapToDimensionFilter

CHAPTER 16

Decoding model references

For the application to display data correctly on a map, the data should have a coordinate reference system associated with it. This allows the application to interpret the coordinates of the data correctly.

Not all data has its coordinate reference system (CRS) stored digitally. Sometimes, the CRS is known for a whole data repository and is implied when the data is used. In other cases, the data is in a format that does not support storing the CRS, for example JPEG or GIF, or that does not impose the storage of the CRS, such as TIFF. If the CRS is not available for the source data, you have to define a CRS for the data yourself. In that case, you need to contact the supplier of the data to find out the reference system and the format in which to store it.

This chapter explains how to use an `ILcdModelDecoder` to handle the CRS associated with a model, also known as the model reference. First, it describes the logic used to determine a model's coordinate reference. Next, it discusses how to provide reference information. Finally, this chapter explains how to encode and store a model reference.

16.1 How does a model decoder determine the model reference?

To decode a model, the model decoder tries to determine the model reference by:

1. Retrieving it from the data. In this case, the CRS is included in the data files.
2. If the reference is not available in the data itself, checking for an additional file that defines the model reference. The CRS can be included with the data in a separate file. An `ILcdModelReferenceDecoder` is used to decode such files. LuciadLightspeed offers `ILcdModelReferenceDecoder` implementations that allow you to read Well-Known Text (WKT) or EPSG codes, for example. You can set the `ILcdModelReferenceDecoder` as a property of the model decoder, so that the reference is constructed and set directly on the decoded model.
3. Using a fixed default reference, set on the decoder. Some model decoders offer the possibility to set a default model reference, which is used as a fall-back mechanism if no other model reference is available. Use the method `setDefaultModelReference` on the `ILcdModelDecoder` implementation.

16.2 Providing model reference information

This section discusses some available implementations of the `ILcdModelReferenceDecoder` interface, and goes into more detail about using particular `ILcdModelDecoder` implementations to decode raster models and their model references.

16.2.1 `ILcdModelReferenceDecoder` implementations

These are some of the available implementations of `ILcdModelReferenceDecoder`:

- `TLcdModelReferenceDecoder` to decode `.ref` files, a format created by Luciad defining the properties in a Java properties file. The CRS is encoded using the mechanism explained in [Chapter 58](#).
- `TLcdWKTModelReferenceDecoder` to decode `.prj` files which store the CRS in the Well-Known Text (WKT) format. Refer to the OpenGIS Coordinate Transformation Service Implementation Specification at <http://www.opengeospatial.org/standards/ct>.
- `TLcdEPSGModelReferenceDecoder` to decode `.epsg` files. An `.epsg` file is a text file containing a single decimal EPSG code, formatted as `n` or `EPSG:n`. For details about EPSG codes, refer to <http://www.epsg.org/>.
- `TLcdRPCModelReferenceDecoder` to decode RPC files, which typically come with GeoTIFF files and have the extension `_rpc.txt`.

See the API reference documentation for more information about the available `ILcdModelReferenceDecoder` implementations.



There is also a composite model reference decoder implementation available (`TLcdCompositeModelReferenceDecoder`) which can handle all those files. See the class javadoc of the composite decoder for example usages.

16.2.2 Providing reference information for raster data

To decode raster data, additional information about the raster bounds is required. The raster bounds are the real-world extents of the raster data, expressed in coordinates. LuciadLightspeed supports a number of file formats and decoders, which allow you to define and decode referencing information for raster data.

Decoding the JAI file format

The JAI file format, defined by Luciad, points to a single image that can be decoded by the Java Advanced Imaging library. It contains all the necessary referencing information, in terms of LuciadLightspeed classes and their properties. The `TLcdJAIRasterModelDecoder` decodes JAI files, including the model reference.

The generic `TLcdJAIRasterModelDecoder` decodes properties files with extension `.jai`. A JAI file defines a raster with a single tile image, and the reference information of the raster. The JAI file itself does not contain any image data. It points to an external file that can be decoded by the Java Advanced Imaging library and that lacks georeferencing information, for example, PNM, GIF, JPEG, and TIFF. [Program 70](#) shows an example of a JAI file. It specifies a GIF image, positioned in a grid reference with a transverse mercator projection and the WGS-84 geodetic datum. The properties supported in JAI files

are described in detail in the reference documentation of `TLcdJAIRasterModelDecoder`.

```

1 # Some general information.
2 TLcdJAIRasterModelDecoder.displayName = DC_SPOT
3 TLcdJAIRasterModelDecoder.typeName = geoTIFF
4
5 # The model reference.
6 TLcdJAIRasterModelDecoder.ILcdModelReference.class = com.luciad.reference.TLcdGridReference
7 TLcdJAIRasterModelDecoder.TLcdGridReference.geodeticDatum.ILcdGeodeticDatumFactory.class = \
8         com.luciad.geodesy.
9             TLcdJPLGeodeticDatumFactory
9 TLcdJAIRasterModelDecoder.TLcdGridReference.geodeticDatum.TLcdJPLGeodeticDatumFactory.
10        geodeticDatumName = \
10            WGS\ 84
11 TLcdJAIRasterModelDecoder.TLcdGridReference.projection.ILcdProjection.class = \
12             com.luciad.projection.
12                 TLcdTransverseMercator
13 TLcdJAIRasterModelDecoder.TLcdGridReference.projection.TLcdTransverseMercator.centralMeridian
13     =-75.0
14 TLcdJAIRasterModelDecoder.TLcdGridReference.falseEasting = 500000
15 TLcdJAIRasterModelDecoder.TLcdGridReference.falseNorthing = 0.0
16 TLcdJAIRasterModelDecoder.TLcdGridReference.scale = 0.9996
17 TLcdJAIRasterModelDecoder.TLcdGridReference.unitOfMeasure = 1
18
19 # The raster bounds in model coordinates.
20 TLcdJAIRasterModelDecoder.ulx = 316393.8
21 TLcdJAIRasterModelDecoder.uly = 4319175.0
22 TLcdJAIRasterModelDecoder.rasterWidth = 10430.0
23 TLcdJAIRasterModelDecoder.rasterHeight = 21180.0
24
25 # The image filename.
26 TLcdJAIRasterModelDecoder.fileName = wash_spot_small.gif

```

Program 70 - Example of a JAI file

Decoding the TAB file format

The TAB file format points to a single image, and specifies its coordinate system and its position in this coordinate system. The `TLcdTABRasterModelDecoder` decodes TAB files, including the model reference. The image file can be in any format that is supported by the Java Advanced Imaging library.

Decoding the RST file format

The RST file format, defined by Luciad , is similar to the TAB and JAI file format, but it allows defining multiple levels and tiles. The tiles point to an entire dataset of images that can be decoded by the `TLcdRasterModelDecoder`, which adds all the necessary referencing information.

The generic `TLcdRasterModelDecoder` decodes properties files with extension `.rst`. An RST file defines a raster with any number of levels and tiles, and the reference information of the raster. Like a JAI file, an RST file itself does not contain any image data. It points to the external files that can be decoded by the `ILcdTileDecoder` that is specified in the RST file as well. [Program 71](#) shows an example of a simple RST file. It contains only a single level and a single tile in GIF format, positioned in a grid reference with a transverse mercator projection and the WGS-84 geodetic datum. The properties supported in RST files are described in detail in the reference documentation of `TLcdRasterModelDecoder`.

```

1 # Some general information.
2 TLcdRasterModelDecoder.displayName = DC_SPOT
3 TLcdRasterModelDecoder.sourceName = DC_SPOT2
4 TLcdRasterModelDecoder.typeName = gif
5
6 # The model reference.
7 TLcdRasterModelDecoder.ILcdModelReference.class = com.luciad.reference.TLcdGridReference
8 TLcdRasterModelDecoder.TLcdGridReference.geodeticDatum.ILcdGeodeticDatumFactory.class = \
9           com.luciad.geodesy.
10          TLcdJPLGeodeticDatumFactory
11 TLcdRasterModelDecoder.TLcdGridReference.geodeticDatum.TLcdJPLGeodeticDatumFactory.
12   geodeticDatumName = \
13           WGS\ 84
14 TLcdRasterModelDecoder.TLcdGridReference.projection.ILcdProjection.class = \
15           com.luciad.projection.
16           TLcdTransverseMercator
17 TLcdRasterModelDecoder.TLcdGridReference.projection.TLcdTransverseMercator.centralMeridian =
18   -75.0
19 TLcdRasterModelDecoder.TLcdGridReference.falseEasting = 500000
20 TLcdRasterModelDecoder.TLcdGridReference.falseNorthing = 0.0
21 TLcdRasterModelDecoder.TLcdGridReference.scale = 0.9996
22 TLcdRasterModelDecoder.TLcdGridReference.unitOfMeasure = 1
23
24 # The decoder that can decode the image.
25 TLcdRasterModelDecoder.ILcdTileDecoder.class = com.luciad.format.raster.TLcdGIFImageDecoder
26 TLcdRasterModelDecoder.pixelByteSize = 1
27 TLcdRasterModelDecoder.defaultPixel = 255
28 TLcdRasterModelDecoder.forcedTransparentColorIndex = 255
29
30 # The raster bounds in model coordinates.
31 TLcdRasterModelDecoder.ulx = 316393.8
32 TLcdRasterModelDecoder.uly = 4319175.0
33 TLcdRasterModelDecoder.rasterWidth = 10430.0
34 TLcdRasterModelDecoder.rasterHeight = 21180.0
35
36 # The raster size in pixels.
37 TLcdRasterModelDecoder.tilePixelWidth = 348
38 TLcdRasterModelDecoder.tilePixelHeight = 707
39
40 # The image filename.
41 TLcdRasterModelDecoder.fileName = wash_spot_small.gif

```

Program 71 - Example of an RST file

Decoding the EPSG, PROJ and REF file formats

The EPSG, PROJ, and REF file formats all define part of the necessary referencing information, namely the coordinate system in which the raster is positioned. They are decoded by model reference decoders as described in [Section 16.2.1](#). These model reference decoders can be set on raster model decoders for which they are useful, for instance, `TLcdBILModelDecoder` and `TLcdGeoTIFFModelDecoder`.

Decoding the TFW file format

The TFW file format defines the necessary referencing information for a corresponding TIFF file, namely its position in a given coordinate system. Similarly, JGW files provide positioning information for JPEG images, BLW files provide positioning information for BIL images, and BPW files provide positioning information for BMP images. This format is decoded by the raster model decoders for which this information is useful, for example: `TLcdBILModelDecoder`, `TLcdTFWRasterModelDecoder`, and `TLcdGeoTIFFModelDecoder`.

16.3 Using an ILcdModelReferenceDecoder

As illustrated in [Program 72](#) and [Program 73](#), the reference decoder can be set as a property of the model decoder, or can be used directly to obtain a reference that is manually set onto the model itself.

```

1 TLcdModelReferenceDecoder reference_decoder = new TLcdModelReferenceDecoder;
2 // Note that not all model decoders allow the user to set a model reference decoder.
3 model_decoder.setModelReferenceDecoder(reference_decoder);
4 ILcdModel model = model_decoder.decode("aSourceFile");

```

Program 72 - Using a reference decoder with a model decoder

```

1 TLcdModelReferenceDecoder reference_decoder = new TLcdModelReferenceDecoder;
2 ILcdModelReference reference = reference_decoder.decodeModelReference("reference_file.ref");
3 TLcdVectorModel model = new TLcdVectorModel(reference);

```

Program 73 - Direct decoding of the reference from an external file

16.4 Parsing references from text

In some cases, there is no additional file that stores the coordinate reference system. Instead, the model reference is expressed as a code in the data file, in text format. The code used to identify the CRS is defined by widely accepted standards.

You can use an implementation of `ILcdModelReferenceParser` to handle these cases, and convert the reference's string representation into an `ILcdModelReference` instance. LuciadLightspeed provides the following implementations of `ILcdModelReferenceParser`:

- `TLcdEPSGReferenceParser`: the European Petroleum Survey Group (EPSG, <http://www.epsg.org>) provides an extensive database of coordinate reference systems, uniquely identified by 4 or 5 digit numbers. `TLcdEPSGReferenceParser` parses most commonly used coordinate reference systems in the EPSG database.
- `TLcdAUTOREferenceParser` and `TLcdAUTO2ReferenceParser` parse a CRS formatted according to the OpenGIS Web Map Service (WMS) 1.1.1 and 1.3.0 specifications.
- `TLcdWKTReferenceParser` parses a CRS in the Well-Known Text format of the OpenGIS Specification.
- `TLcdMIFModelReferenceParser` parses a CRS code formatted in the MapInfo Interchange Format (MIF).
- `TLcdSpatialReferenceParser` parses CRS codes formatted in one of the Spatial Reference formats listed on <http://spatialreference.org>.

16.5 Storing a model reference

A model reference can be read from either a `String` or a file. Therefore, it can also be formatted as a `String`, using `ILcdModelReferenceFormatter`, or stored in a file, using `ILcdModelReferenceEncoder`. LuciadLightspeed provides encoding counterparts for most of the implementations in [Section 16.2.1](#) and [Section 16.4](#). Refer to the API reference for an overview of the available implementations.

CHAPTER 17

Encoding model data

For saving model data to an external data source, LuciadLightspeed provides the interface `ILcdModelEncoder` and a number of implementations that support specific file formats. The following sections describe `ILcdModelEncoder`, its main implementations, how to use it, and how to make your own implementation of it.

17.1 What is an `ILcdModelEncoder`?

An `ILcdModelEncoder` writes the data of an `ILcdModel` to a data source. An `ILcdModelEncoder` has two ways to encode an `ILcdModel`:

- `save`: this method encodes the `ILcdModel` to the data source described by its `ILcdModelDescriptor`. To check whether an `ILcdModelEncoder` can save a given `ILcdModel` before attempting to do so, you can use the method `canSave`.
- `export`: this method encodes the `ILcdModel` to the data source specified by a `String`. To check whether an `ILcdModelEncoder` can export a given `ILcdModel` before attempting to do so, you can use the method `canExport`.

LuciadLightspeed provides an `ILcdModelEncoder` for a number of formats, including the ESRI Shape format (`TLcdSHPModelEncoder`) and the MapInfo MIF format (`TLcdMIFModelEncoder`) for vector data, and the GeoTIFF format (`TLcdGeoTIFFModelEncoder`) for raster data. Refer to the API reference for more information on their usage. [Table 14](#) gives an overview of all file formats that LuciadLightspeed supports and indicates if a model encoder is available for the format or not.

17.2 Saving an `ILcdModel` using an `ILcdModelEncoder`

The method `getModelEncoder` of `ILcdModel` retrieves either a valid `ILcdModelEncoder` for the given `ILcdModel`, or `null`. Given a particular `ILcdModelEncoder`, the method `canSave` checks whether or not a particular `ILcdModelEncoder` is valid for a given `ILcdModel`. The method `save` saves a given `ILcdModel` according to the data source of the `ILcdModel`.

```
1     model_to_save.getModelEncoder().save(model_to_save);
```

Program 74 - Saving an `ILcdModel` using an `ILcdModelEncoder`
(from samples/common/action/SaveAction)

Program 74 shows how to save an `ILcdModel` (called `model_to_save`) using its `ILcdModelEncoder` (provided that `getModelEncoder` does not return `null`).

17.3 Making your own implementation of `ILcdModelEncoder`

This section describes the main methods for a new implementation of the interface `ILcdModelEncoder`. You can find the complete source code of the described example in `samples.decoder.custom1`.

The sample implementation writes an `ILcdModel` to a file in `custom1` format which is a simple file format that consists of:

- a line indicating the format
- a sequence of shape data. Each shape is described as:
 - one line indicating the type of shape
 - multiple lines containing the data needed to create the shape. Depending on the type of shape, these lines are formatted differently.
- a line containing `EOF`

Note that this section discusses the methods `save` and `canSave`. The methods `export` and `canExport` are similar.

```

1  @Override
2  public void save(ILcdModel aModel)
3      throws IllegalArgumentException, IOException {
4
5      if (!canSave(aModel)) {
6          throw new IllegalArgumentException(
7              "Cannot save [" + aModel + "]. Need type name[" + Custom1ModelDecoder.TYPE_NAME + "
8                  ], " +
9                  "but is [" + aModel.getModelDescriptor().get TypeName() + "]");
10     }
11
12     OutputStream source_output_stream = fOutputStreamFactory.createOutputStream(
13         aModel.getModelDescriptor().getSourceName());
14
15     BufferedWriter buffered_writer =
16         new BufferedWriter(new OutputStreamWriter(source_output_stream));
17
18     try {
19         buffered_writer.write(H HEADER_CHECK);
20         buffered_writer.newLine();
21         Object element;
22         for (Enumeration e = aModel.elements(); e.hasMoreElements(); ) {
23             element = e.nextElement();
24             if (element instanceof ILcd3DEditablePoint) {
25                 buffered_writer.write("point3d");
26                 buffered_writer.newLine();
27                 write3DEditablePoint((ILcdPoint) element, buffered_writer);
28             } else if (element instanceof ILcd2DEditablePoint) {
29                 buffered_writer.write("point2d");
30                 buffered_writer.newLine();
31                 write2DEditablePoint((ILcdPoint) element, buffered_writer);
32             }
33         // ...
34     }
35     buffered_writer.write("EOF");
36     buffered_writer.flush();
37     finally {
38         source_output_stream.close();
39     }
40     }
41 }
```

Program 75 - Saving an ILcdModel using an ILcdModelEncoder
 (from samples/gxy/decoder/custom1/Custom1ModelEncoder)

Program 75 implements the method `save`, by first examining the `ILcdModelDescriptor` of the given `ILcdModel`, called `aModel`. In particular, it checks the property `TypeName` of the `ILcdModelDescriptor`. If that property equals the constant `TYPE_NAME` of this `ILcdModelEncoder`, it sets up an `OutputStream` called `source_output_stream` and a `BufferedWriter` for `source_output_stream`.

First, the header line is written to `source_output_stream`. Then all elements of `aModel` are enumerated, using the method `elements` of the interface `ILcdModel`. For each type of `ILcdShape` in `aModel`, a different method for writing that type of `ILcdShape` to `source_output_stream` is called. When all elements of `aModel` are written to `source_output_stream`, the closing line is written, and `source_output_stream` is closed. This finishes the method `save`.

```
1  
2     @Override  
3     public boolean canSave(ILcdModel aModel) {  
4         // Can only save the type of model this encoder was created for.  
5         return aModel.getModelDescriptor().get TypeName() .equals(Custom1ModelDecoder.TYP  
6     }
```

Program 76 - Saving an ILcdModel using an ILcdModelEncoder
(from samples/gxy/decoder/custom1/Custom1ModelEncoder)

Program 76 shows an implementation of the method `canSave`. It verifies that the type name matches the type name of the matching model decoder.

CHAPTER 18

Clustering

When there are many objects in a view, it can become difficult to keep an overview. Therefore you can introduce clustering to help reduce the clutter. Figure 56 contrasts the visualization of an original model with the visualization of a derivative transformed through clustering.

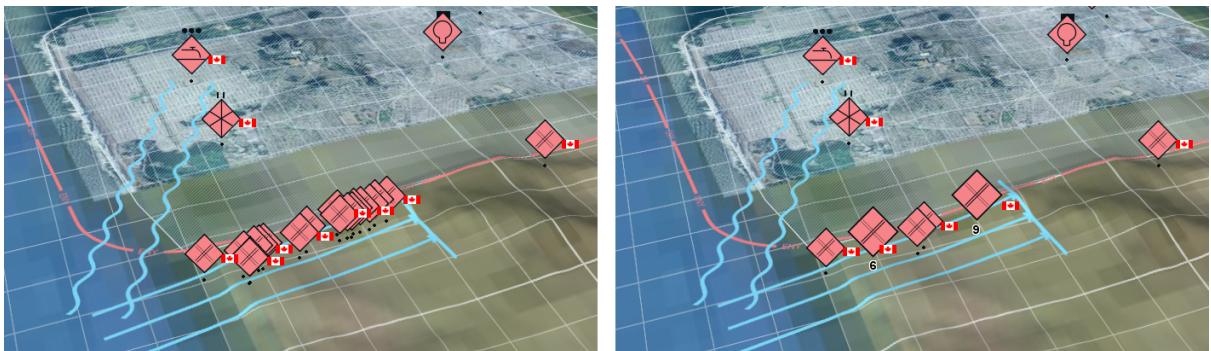


Figure 56 - Original versus clustered

18.1 Using a clustered model

To cluster objects in your model, you make use of an `ALcdTransformingModel`. A transforming model wraps around an original model, and applies a clustering transformation to it. To create a transforming model, you need at least an original model and a clustering transformer, as Program 77 demonstrates.

```
1 ILcdModel transformingModel = TLcdTransformingModelFactory.createTransformingModel(  
    originalModel, clusteringTransformer);
```

Program 77 - Creating a transforming model
(from samples/lightspeed/clustering/MainPanel)

The clustering transformer is responsible for the actual transformation from objects in the original model to clusters.

18.2 Configuring a `TLcdClusteringTransformer`

A `TLcdClusteringTransformer` aggregates model elements that are close together into a cluster object. The cluster object is added to the transforming model instead of the individual

elements. Model elements that are not part of a cluster are added as-is.

The clustering algorithm operates on the basis of a few parameters. You can specify those parameters when you create the `TLcdClusteringTransformer`. You can define:

- **clusterSize**: a measure to express the approximate size of a cluster. The smaller you set the cluster size, the fewer points are clustered together.
- **minimumPoints**: the minimum number of points required to form a cluster. For example, when you set this parameter to 3, there will never be a cluster that only represents 2 points. Each cluster will at least contain 3 points.
- **cluster shape provider**: an `ILcdClusterShapeProvider` determines the shape of the cluster based on its contained elements. For example, if a cluster contains cities, you can choose to place the cluster at the point location of the city with the largest population, as demonstrated by [Program 78](#).

```

1  public final class BiggestCityClusterShapeProvider implements ILcdClusterShapeProvider {
2
3      @Override
4      public ILcdShape getShape(Set<Object> aComposingElements, ILcdModel aOriginalModel) {
5          ILcdDataObject biggestCity = null;
6          for (Object element : aComposingElements) {
7              if (element instanceof ILcdDataObject) {
8                  ILcdDataObject city = (ILcdDataObject) element;
9                  if (biggestCity == null || ((int) city.getValue("Population")) > ((int) biggestCity.
10                     getValue("Population"))) {
11                      biggestCity = city;
12                  }
13              }
14          }
15      }
16  }
```

Program 78 - A custom shape provider

[Program 79](#) shows how you can specify custom parameters for the clustering algorithm.

```

1  TLcdClusteringTransformer transformer =
2      TLcdClusteringTransformer.newBuilder()
3          .defaultParameters()
4              .clusterSize(100)
5              .minimumPoints(3)
6              .shapeProvider(new BiggestCityClusterShapeProvider())
7              .build()
8          .build();
```

Program 79 - Configuring a custom shape provider

If no parameters are specified, sensible defaults are used: the location of the element closest to the center of mass of the cluster elements is chosen as the cluster location. That default prevents the positioning of clusters in illogical places, a cluster of cities placed in a nearby body of water, for example.

18.3 Clustering by class

One of the key aspects of the clustering algorithm is its use of an `ILcdClassifier`.

The algorithm organizes the elements to be transformed according to a classifier. An `ILcdClassifier` returns a classification for a model element. This is a `String` employed by the algorithm to separate the elements. Two elements with the same classification may or may not be clustered together. Two elements with distinct classifications are never clustered together.

For example, if your model contains cities and you do not want cities to be clustered across country borders, the classifier can simply return the name of the country to which a city belongs. Program 80 shows how to classify cities according to their countries.

```

1  private static class CityClassifier implements ILcdClassifier {
2
3      public static final String NO_CLASSIFICATION = "";
4
5      @Override
6      public String getClassification(Object aObject) {
7          if (aObject instanceof ILcdDataObject) {
8              ILcdDataObject city = (ILcdDataObject) aObject;
9              return city.getValue("Country").toString();
10         }
11         return NO_CLASSIFICATION;
12     }
13 }
14 }
```

Program 80 - A classifier that classifies cities according to country

Classifications also allow more fine-grained control over the clustering algorithm. You can specify parameters per classification or classification group. Program 81 sets a custom shape provider that will be used for all classifications. However, elements classified as "Belgium" will not be clustered. Note that for this to work, the `ILcdClassifier` and the configuration of the `TLcdClusteringTransformer` must work in harmony. In the example, the `ILcdClassifier` should return "Belgium" for Belgian cities.

```

1  TLcdClusteringTransformer transformer =
2      TLcdClusteringTransformer.newBuilder()
3          .classifier(new CityClassifier())
4          .defaultParameters()
5          .shapeProvider(new BiggestCityClusterShapeProvider())
6          .build()
7          .forClass("Belgium")
8          .noClustering()
9          .build()
10         .build();
```

Program 81 - Classification specific parameters

The `forClass` method is overloaded. You can specify an exact classification or an `ILcdFilter<String>`. The latter allows you to specify parameters for multiple classes with one `forClass` call. The order of the `forClass` calls is important. The parameters of the first classification match are used.

18.4 Setting up scale-dependent clustering

For certain data sets, it makes sense to apply another clustering setting when the scale of the view changes. To set up distinct clustering settings for various view scales, create a `TLcdClusteringTransformer` for each of the scales. Next, use the `TLcdClusteringTransformer#createScaleDependent` method to combine them into a single, scale-dependent `TLcdClusteringTransformer`.

```

1  //When zoomed in, cluster the events per country and avoid grouping events
2  //happening in different countries in different clusters.
3  TLcdClusteringTransformer zoomedInClusteringTransformer =
4      TLcdClusteringTransformer.newBuilder()
5          .classifier(new CountryClassifier())
6          .defaultParameters()
7              .clusterSize(CLUSTER_SIZE)
8                  .minimumPoints(MIN_CLUSTER_COUNT)
9                      .build()
10                         .build();
11 //When zoomed out, all the events can be clustered together.
12 //Otherwise, we would end up with overlapping clusters as the countries become rather
13     small
14 TLcdClusteringTransformer zoomedOutClusteringTransformer =
15     TLcdClusteringTransformer.newBuilder()
16         .defaultParameters()
17             .clusterSize(CLUSTER_SIZE)
18                 .minimumPoints(MIN_CLUSTER_COUNT)
19                     .build()
20                         .build();
21
22 //Switching between the two clustering approaches should happen at a scale 1 : 25 000 000
23 double mapScaleDenominator = 25e6;
24 double internalScaleDenominator =
25     1d / AScaleSupport.mapScale2InternalScale(1d / mapScaleDenominator, -1, aMapReference)
26 ;
27
28 TLcdClusteringTransformer scaleDependentClusteringTransformer =
29     TLcdClusteringTransformer.createScaleDependent(
30         new double[]{internalScaleDenominator},
31         new TLcdClusteringTransformer[]{zoomedInClusteringTransformer,
32             zoomedOutClusteringTransformer}
33     );

```

Program 82 - Creating a scale-dependent clustering transformer
 (from samples/lightspeed/clustering/MainPanel)

18.5 Working with TLcdCluster objects

The clusters in the transforming model are instances of `TLcdCluster`. This class implements `ILcdDataObject`. `TLcdClusterDataTypes` defines the objects that use the `TLcdCluster` data model API. Of note are the classification, composing elements and shape properties. You can use those properties to retrieve the classification, the set of contained elements and the cluster shape respectively.

CHAPTER 19

Grouping models hierarchically

In a similar way as you can group layers, which is discussed in [Section 6.2](#), you can also group models. You may find this useful to group models containing related data sets. LuciadLightspeed provides the following options to group a set of models in a hierarchical model structure:

- Create a **model list with submodels** using the class `TLcdModelList`. A model list is a composite model. As such applications can handle a model list as a single model regardless of its submodels. [Section 19.1](#) provides more details on creating and using a model list.
- Create a **model node with child models** using the interface `ILcdModelTreeNode`. A model node is a model itself. Applications consider the model node and its child models as separate models. A `TLcdZipModelListDecoder` for example, creates a model node which contains models for each data set included in a given zip file. [Section 19.2](#) provides more details on creating and using a model node.

19.1 Creating a model list

The class `TLcdModelList` allows you to create an `ILcdModel` that is composed of an ordered list of other `ILcdModel` objects, based on the Composite design pattern. Each added model, or submodel, can be a `TLcdModelList` itself. Using a `TLcdModelList` instead of adding all elements (domain objects) of the submodels to a flat model, allows fast reordering of the submodels once the model list has been created. Additionally, each submodel can have its own model descriptor, next to the model descriptor of the model list. Note however that when you use a `TLcdEditableModelListDescriptor` for the model list, the model descriptor is automatically updated whenever a submodel is added or removed from the model list. Note also that since a `TLcdModelList` is an `ILcdModel`, all of the submodels should have the same model reference.

To enumerate the elements of the models in a model list you can use the method `elements()`. This method iterates the submodels in the list one by one. Note that a `TLcdModelList` itself is not an editable model; you cannot add or remove domain objects to it.



If all submodels implement `ILcd2DBoundsIndexedModel`, it is best to use the subclass `TLcd2DBoundsIndexedModelList` so you can perform spatial queries on the model list.

Another option for grouping models is creating an `ILcdModelTreeNode` as described in [Section 19.2](#). The main difference between a model list and a model node is that a model list is a

composite model and a model node is a model itself that can have child models.

19.2 Creating a model node

Similarly to creating a layer node, as described in [Section 6.2](#), you can group models in a model (tree) node by using the interface `ILcdModelTreeNode`. As [Figure 57](#) shows, an `ILcdModelTreeNode` is an `ILcdModel` that also implements `ILcdModelContainer` which allows you to add child models to the model node.

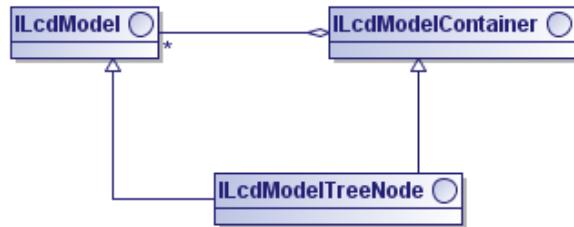


Figure 57 - Class diagram of `ILcdModelTreeNode`

Since the model node is a model itself, all methods inherited from `ILcdModel` only apply to the model node and not to its child models. For example, calling `addElement(Object, int)` results in the addition of the given element (domain object) to the model node, and not to its child models. On the other hand, the methods inherited from `ILcdModelContainer` only apply on the child models. Calling `modelCount()` on the model node, for example, returns the number of child models of the model node, not including the node itself.

The difference with a `TLcdModelList` is that a model list is a composite model and a model node is a model itself that can have child models. As a consequence, the methods inherited from `ILcdModel` behave differently on the model node than on the model list. For example, calling `ILcdModel.elements()` on a model node only returns the elements of the model node and not of its child models. Calling the method on a model list returns the elements of all the submodels of the model list. Also, each of the child models of a model node can have a different model reference, contrary to the submodels of a model list.



A model node can contain domain objects itself but you can also use it as a mere container for other models. A model node without domain objects is empty and should thus return true for `isEmpty()`.

In addition to the methods inherited from `ILcdModelContainer`, to add/remove child models and retrieve a specific model, the `ILcdModelTreeNode` has methods to add a hierarchical listener of the type `ILcdModelListener` and `ILcdModelContainerListener`. These listeners receive events for all changes to the models and model nodes contained in the hierarchical model structure. For more information on listeners, refer to [Section 8.2](#).

As shown in [Figure 58](#), `ILcdModelTreeNode` has two main implementations: `TLcdModelTreeNode` and `TLcd2DBoundsIndexedModelTreeNode`. As long as the model node itself contains no domain objects but only child models, it is best to use the `TLcdModelTreeNode`. For model nodes that contain domain objects itself, it is more efficient to use an `ILcd2DBoundsIndexedModel` such as the `TLcd2DBoundsIndexedModelTreeNode`.

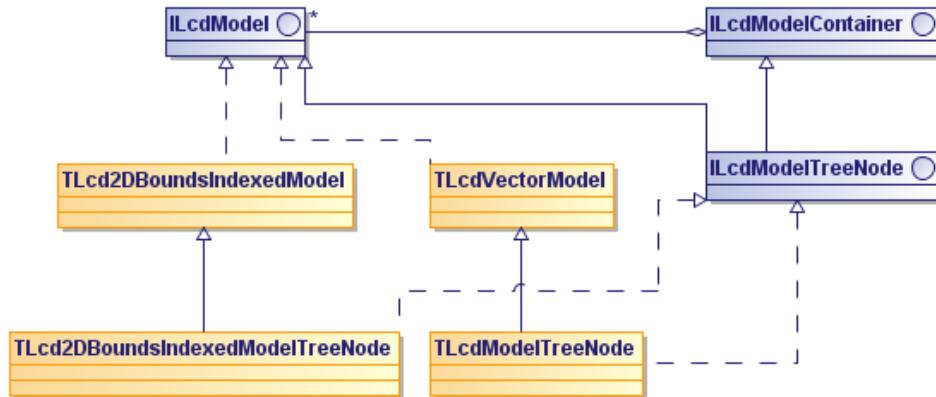


Figure 58 - Main implementations of `ILcdModelTreeNode`

19.3 Updating models

Changes in your model need to be reflected in the view. To achieve this efficiently, you need to take threading and concurrency issues into consideration.

When your model is displayed in a GXY view, perform the model change in your view painting thread. This is typically the Java Event Dispatch Thread. If you wish to perform the model change in another thread, you need to take extra measures to ensure thread-safety.

When your model is displayed in a Lightspeed view, you can perform the model change in any thread, but you need to set the object write locks appropriately yourself.

For more information on threading, refer to [Chapter 9](#).

PART III Advanced Lightspeed View and Controller Topics

CHAPTER 20

Working with Lightspeed views

The main entry point for visualization in LuciadLightspeed is the view. You use a view to determine which layers to show, and to perform functions such as zooming in on particular areas, limiting the area that the user can see or interact with, determining world position of screen coordinates, and so forth. This chapter discusses various topics related to the customization of your Lightspeed view. It describes a number of ways to tailor your Lightspeed view to the application you are building:

- [Section 20.1](#) describes the purpose of Lightspeed views.
- [Section 20.2](#) describes how to fit a view to a certain area.
- [Section 20.3](#) explains how to constrain the navigation in a view.
- [Section 20.4](#) describes how to render a view offscreen.
- [Section 20.5](#) explains how you can display small pop-up windows, commonly known as balloons, in your view.
- [Section 20.6](#) demonstrates how you can add a grid layer to your view.
- [Section 20.7](#) describes how you can derive the world position from the screen position of an object, such as a mouse cursor.
- [Section 20.9](#) describes how to add animations to your view.

20.1 What is a view?

The view is the Lightspeed component that handles the positioning and visualization of your data, although it does not contain the data itself. The data is kept in the models, which in turn are added to the layers. The view takes care of:

- Translation of world to screen: in the view you set the world reference, to which all the layers in the view are mapped. The view's camera component takes care of the transformation of world coordinates to view coordinates, and the other way around, so that objects can be correctly positioned and moved on the screen. The camera represents the user's view point on the application. You can use the camera to change the orientation of the view point.
- Layer management: the view manages layer painting order, allows you to add layers from layer factories and track activity in a layer. It also lets you set up listeners for changes in the layer models. To read more about layers, see [Chapter 21](#).

- Resource management: memory, threading, and so on.
- Controller management. To learn more about controllers, see [Chapter 26](#).
- Background color: you can define which color the view uses to clear the view's background.

This chapter only addresses specific view-related topics. To read more about any of the major components the view interacts with, such as layers and controllers, see the appropriate chapter in [Part III](#).

20.2 Fitting a view

To let the user focus on specific data displayed in the view, you often have to shift the view to the world area containing the data. You can use the various `ALspViewXYZWorldTransformation` implementations or the navigation functionality to modify a view's position, but this can be quite cumbersome.

Therefore, LuciadLightspeed offers functionality to fit the view onto the data in a layer or some pre-defined bounds. For example, [Program 83](#) shows how to fit the view on specific bounds.

```
1 TLspViewNavigationUtil navigationUtil = new TLspViewNavigationUtil(view);
2 navigationUtil.fit(new TLcdLonLatBounds( -180, -90, 360, 180 ), new TLcdGeodeticReference());
```

Program 83 - Using `TLspViewNavigationUtil.fit()` to navigate to specific bounds

The method `TLspViewNavigationUtil.animatedFit()` offers similar functionality, but instead of initiating a jump to the new state, it starts an animation to move fluidly from the current state to the fitted state. [Program 84](#) demonstrates this.

```
1 TLspViewNavigationUtil navigationUtil = new TLspViewNavigationUtil(view);
2 navigationUtil.animatedFit(Collections.singletonList(layer));
```

Program 84 - Using animated fit to navigate to a specific layer

By default, animations are automatically started.

To center the view to the middle of the data in a layer or some bounds without zooming and rescaling the view, you can call the `TLspViewNavigationUtil.center()` methods.

20.3 Constraining navigation in a view

It is possible to constrain the navigation in a view, so that limitations are imposed on the boundaries of the view as displayed on screen. You can achieve this by installing a camera constraint on the view-to-world transformation. The main constraint class is `ALspCameraConstraint`. The main camera constraint implementations are:

- `TLspAboveTerrainCameraConstraint3D`: to make sure that the camera always stays above the terrain.
- `TLspLookAtTrackingCameraConstraint2D`: to look at a specific tracking point in 2D. This constraint also offers the possibility of following the orientation of the tracked point.
- `TLspLookAtTrackingCameraConstraint3D`: to look at a specific tracking point in 3D.

- `TLspLookFromTrackingCameraConstraint3D`: to look from a specific tracking point in 3D. This constraint also offers the possibility of following the orientation of the tracked point.

Program 85 demonstrates how to install a camera constraint on a view to world transformation.

```

1 TLspViewXYZWorldTransformation3D transformation = new TLspViewXYZWorldTransformation3D(
2   aView);
transformation.addConstraint(new TLspAboveTerrainCameraConstraint3D());
```

Program 85 - Adding a camera constraint to the view to world transformation
 (from samples/lightspeed/trackingcamera/MainPanel)

The `lightspeed/limitnavigation` sample shows how you can create a custom camera constraint to limit navigation in the view. It contains a 2D and 3D constraint that:

- Limits the scale
- Makes sure that the camera always looks inside an area of interest
- Makes sure that the camera is oriented in such a way that North is always up

20.4 Creating an off-screen view

In some situations, you may prefer to create a Lightspeed view that is kept off screen, and rendered in an internal memory buffer, rather than place the view inside an on-screen graphical user interface.

You may be developing a server that builds requested images and renders them off-screen before sending them to its clients, for instance. You can also use an off-screen view if you want to prepare an image in an internal buffer, and then display the image on-screen from the buffer at a later stage. Off-screen views can also be useful when you are setting up automated UI tests.

For such use cases, LuciadLightspeed provides the `TLspOffscreenView` class.

`TLspOffscreenView` behaves exactly like any other `ILspView` implementation, except that unlike an `ILspAWTView`, it does not have a host component through which it can be displayed on the screen. As an alternative, `TLspOffscreenView` has a method `getImage()`, which returns a screenshot of the view's current state as a `BufferedImage`.

20.5 Adding balloons to the view

Graphically, a balloon is a small popup window displaying information about an element in a user interface. Conceptually, a balloon is a container for a `JComponent` which you can add to any Swing component, including the Lightspeed overlay panel. The `JComponent` inside a balloon is referred to as a **balloon content panel**. Balloons additionally can have an arrow, pointing to a location in the view. Figure 106 shows an example of a balloon in a Lightspeed view.

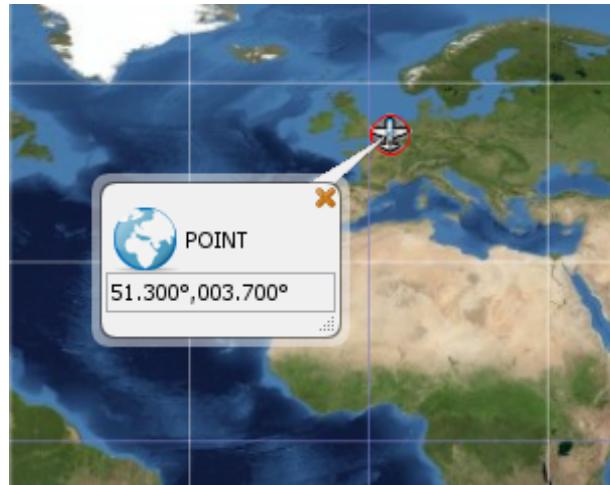


Figure 59 - An example of a balloon in a Lightspeed view.

Figure 60 shows the typical workflow of creating and displaying balloons.

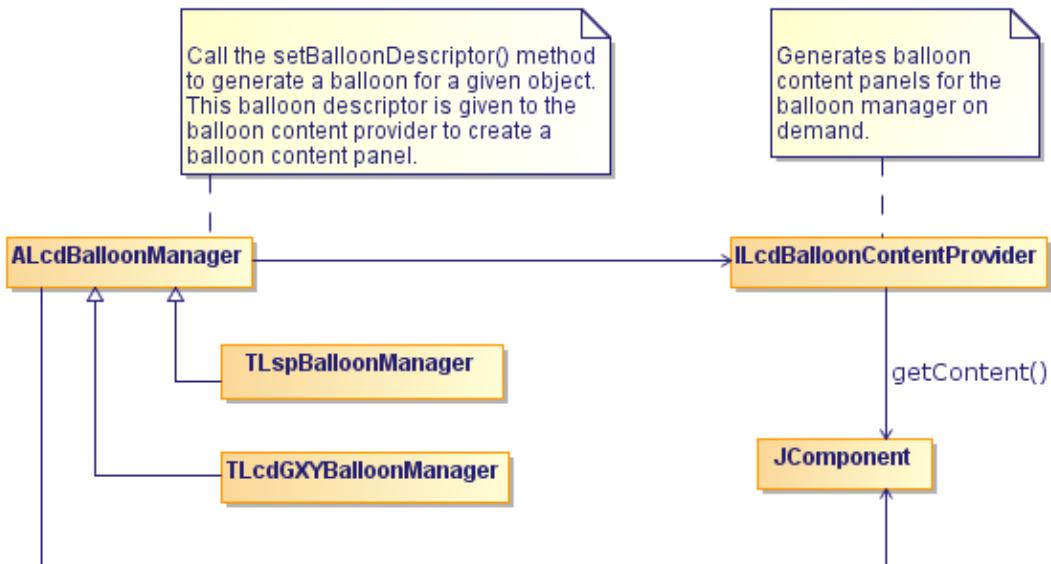


Figure 60 - High-level overview of creating and displaying balloons

20.5.1 Creating balloons

You can use the interfaces `ALcdBalloonManager` and `ILcdBalloonContentProvider` to create and handle balloons. The `ALcdBalloonManager` is responsible for controlling the visibility, size, and location of a balloon. The `ILcdBalloonContentProvider` provides the `ALcdBalloonManager` with balloon content panels on demand.

The main implementation of `ALcdBalloonManager` for an `ILspView` is `TLspBalloonManager`. The balloon manager sets up a number of listeners to update the state of the balloon when needed. It also implements the `getFocusPoint()` method which is responsible for determining the position of the arrow of the balloon.

To set the object for which you want to create a balloon, use the `setBalloonDescriptor()` method of the `ALcdBalloonManager` interface. An `ALcdBalloonDescriptor` contains all

the necessary information to describe a balloon. It provides the following implementations:

- `TLcdGeoAnchoredBalloonDescriptor`: to create a balloon that has an arrow that points to world coordinates. The balloon is updated when the view moves around.
- `TLcdUnanchoredBalloonDescriptor`: to create a balloon without an arrow.
- `TLcdViewAnchoredBalloonDescriptor`: to create a balloon that has an arrow that points to a view coordinate. The balloon is not updated when the map is panned or zoomed.
- `TLcdModelElementBalloonDescriptor`: to create a balloon that has an arrow that points to a model element. The balloon is updated when the view moves around. When the model element changes, the balloon is updated.



To remove the balloon of a balloon manager, pass `null` as an argument to the `setBalloonDescriptor()` method.

The given balloon descriptor in the `setBalloonDescriptor()` method is passed to an `ILcdBalloonContentProvider`. The balloon content provider is responsible for generating the balloon content. The balloon content can be any custom `JComponent`. If no content is generated, the balloon manager will not display the balloon. It is important that the `canGetContent()` method of `ILcdBalloonContentProvider` only returns `true` for balloon descriptors that it can handle to allow for composite balloon content providers.

You can use the `setBalloonsEnabled()` method of `ALcdBalloonManager` to globally enable or disable the rendering of balloons.

20.5.2 Customizing balloons

You can customize the default positioning algorithm by creating an extension of the balloon manager and override the `computeBalloonBounds()` method. You can supply a parameter that is used for calculating the bounds of a balloon, such as:

- The balloon descriptor
- The bounds of the view
- A position that should be enforced by the algorithm
- A size that should be enforced by the algorithm
- Two booleans to see whether the balloon has been manually resized or relocated by the user

20.6 Displaying a grid in your view

When you are working with a georeferenced view and data, it is often convenient to display a grid for quick orientation. You can add this grid to the view as a layer. LuciadLightspeed offers the following grid layer builders:

- `TLspLonLatGridLayerBuilder` creates a latitude-longitude grid formed by evenly spaced meridians and parallels. It allows locating geographically referenced data immediately. This grid is used in all of the Lightspeed view samples that are shipped with LuciadLightspeed. If necessary, you can change the style of the grid lines and labels using the `ILspStyler` implementation for lon-lat grids, `TLspLonLatGridStyler`. You

can also configure the spacing between the grid lines. Consult the javadoc `TLspLonLatGridLayerBuilder` for more information about how to do this.

- `TLspMGRSGridLayerBuilder` creates an **MGRS** grid. It divides the earth into grid zones that are refined depending on the view scale. As opposed to a longitude-latitude grid block, a block in an MGRS grid always covers the same size area, independent of its location on earth. If necessary, you can change what grid lines and labels are displayed and how they are visualized, by configuring a `TLspMGRSGridStyle` object on the layer builder. The MGRS grid is demonstrated in the grid sample.
- `TLspGeorefGridLayerBuilder` creates a Georef grid. It is similar to the latitude-longitude grid, but uses a simpler and more concise notation. If necessary, you can change what grid lines and labels are displayed and how they are visualized, by configuring a `TLspGeorefGridStyle` object on the layer builder. The Georef grid is demonstrated in the grid sample.
- `TLspXYGridLayerBuilder` creates a Cartesian grid. It is typically defined in the world reference of the view, which results in an axis-aligned grid.

Program 86 shows how to add a grid layer to a view.

```
1 view.addLayer(TLspLonLatGridLayerBuilder.newBuilder().build());
```

Program 86 - Creating a grid layer

20.7 Retrieving world position from screen coordinates

In some cases, you may want to find out what the lon-lat coordinates of a map position are, the coordinates for the map position of a mouse cursor for instance. In LuciadLightspeed, you can calculate this lon-lat position by using `ALspViewXYZWorldTransformation`, which can transform view coordinates to world coordinates.

`ALspViewXYZWorldTransformation` provides the method `viewAWTPoint2worldSFCT` which calculates the location in world coordinates of the given view point. The world point then needs to be converted to lon-lat coordinates. It is possible to tweak the calculation by setting the `LocationMode` parameter to one of two location modes:

- `ELLIPSOID`. The coordinates of the result are calculated on the ellipsoid of the geodetic datum of the view's world reference.
- `CLOSEST_SURFACE`. The resulting point will be on the surface that is closest to the viewer. In other words, the returned coordinates may be on the surface of a domain object, if one is under the specified view coordinates.

Program 87 shows an example of a component that displays the lon-lat coordinate of the mouse cursor. It does this by adding a mouse listener to the view and transforming all the incoming mouse locations to world coordinates using a `ALspViewXYZWorldTransformation` object. The world coordinates are then converted to lon-lat coordinates.

```
1 /**
2  * Displays the coordinates and measurements of the location under the mouse pointer on a map.
3  */
4 public class MouseLocationComponent extends AMouseLocationComponent {
5
6     private ILspView fView;
7     private final Component fOverlayComponent;
```

```

9  private TLspViewMeasureProvider fViewMeasureProvider;
10
11 public MouseLocationComponent(ALspAWTView aView,
12                             Iterable<ILcdModelMeasureProviderFactory>
13                                         aMeasureProviderFactories,
14                                         Iterable<ILcdLayerMeasureProviderFactory>
15                                         aLayerMeasureProviderFactories) {
16     this(aView, aView.getHostComponent(), aView.getOverlayComponent(),
17           aMeasureProviderFactories, aLayerMeasureProviderFactories);
18 }
19
20 public MouseLocationComponent(ILspView aView,
21                               Component aHostComponent,
22                               Component aOverlayComponent,
23                               Iterable<ILcdModelMeasureProviderFactory>
24                                         aMeasureProviderFactories,
25                                         Iterable<ILcdLayerMeasureProviderFactory>
26                                         aLayerMeasureProviderFactories) {
27     super(aHostComponent);
28     fView = aView;
29     fOverlayComponent = aOverlayComponent;
30     fViewMeasureProvider = new TLspViewMeasureProvider(aView, aMeasureProviderFactories,
31                                                       aLayerMeasureProviderFactories);
32 }
33
34 @Override
35 protected TLcdISO19103Measure[] getValues(ILcdPoint aPoint, ILcdModelReference
36                                         aPointReference) {
37     ALcdMeasureProvider.Parameters parameters = ALcdMeasureProvider.Parameters.newBuilder()
38                                         .build();
39     return fViewMeasureProvider.retrieveMeasuresAt(aPoint, aPointReference, parameters);
40 }
41
42 @Override
43 protected double getHeight(ILcdPoint aPoint, ILcdModelReference aPointReference) {
44     if (aPointReference instanceof ILcdGeoReference) {
45         return fView.getServices().getTerrainSupport().getViewDependentHeightProvider(
46             (ILcdGeoReference) aPointReference, true).retrieveHeightAt(aPoint);
47     }
48     return super.getHeight(aPoint, aPointReference);
49 }
50
51 @Override
52 protected ILcdPoint getCoordinates(Point aAWTPoint, ILcdModelReference aReference) throws
53     TLcdOutOfBoundsException {
54     double scaleX = (double) fView.getWidth() / fOverlayComponent.getWidth();
55     double scaleY = (double) fView.getHeight() / fOverlayComponent.getHeight();
56     ILcdPoint worldPoint = fView.getServices().getTerrainSupport().getPointOnTerrain(
57         new TLcdXYZPoint(aAWTPoint.x * scaleX, aAWTPoint.y * scaleY), new TLspContext(null,
58         fView));
59     if (worldPoint == null) {
60         return null;
61     }
62     TLcdXYZPoint modelPoint = new TLcdXYZPoint();
63     TLcdDefaultModelXYZWorldTransformation transformation = new
64         TLcdDefaultModelXYZWorldTransformation();
65     transformation.setXYZWorldReference(fView.getXYZWorldReference());
66     transformation.setModelReference(aReference);
67     transformation.worldPoint2modelSFC(worldPoint, modelPoint);
68     return modelPoint;
69 }
70 }
```

Program 87 - Implementation of a component that displays the lon-lat coordinates of the mouse cursor

(from samples/lightspeed/common/MouseLocationComponent)



To transform the obtained world coordinates further into model coordinates, you can make use of `ILcdModelXYZWorldTransformation`. from the `TLspContext` and continue with the transformation to model coordinates. Use the `worldPoint2modelSFCT()` method. The SFCT model point in this method should be an `ILcdPoint` that can be used in the reference of the model to which you are transforming, a `TLcdLonLatHeightPoint` for geodetic references or a `TLcdXYZPoint` for other references.

20.8 Positioning a view programmatically

The position of a view is often manipulated through a controller. In many cases, however, you need to position the view programmatically, if you want to center the view on a certain object, for instance.

In general, a view's positioning is determined by its view-to-world transformation. In LuciadLightspeed, the `ALspViewXYZWorldTransformation` class represents a view-to-world transformation. Among other things, this transformation determines where the “eye” of the view is placed and how the viewing volume is defined. The class also allows you to link an identifier to transformations. An identifier is useful to prevent cloning: you can compare transformations, and check whether the transformation stored in the cache is still valid by looking at the identifier. The LuciadLightspeed API offers different implementations for 2D and 3D views. Each implementation offers different parameters to position the view.

[Section 20.8.1](#) and [Section 20.8.2](#) describe the low-level details of positioning in a 2D and 3D view. You can go straight to [Section 20.8.3](#) for higher-level abstractions.

For more information about coordinate systems and transformations, see [Chapter 44](#).

20.8.1 Positioning a 2D view

For 2D views, the API provides the `TLspViewXYZWorldTransformation2D` class. The most important parameters that this class offers to adjust the position of the view are:

- The world origin
- The view origin
- The scale parameter

The first two parameters are important for the view's position. Under the hood, the view's configuration is calculated in such a way that the view origin, a point in view space, is mapped to the world origin, a point in the world.

To position your view:

1. Choose a **view origin**, a point in view space, defined in pixel coordinates.
2. Choose a **world origin**, a point in world space defined in Cartesian coordinates. This point must map to the view origin.
3. Choose a **scale** to determine the range of map content shown around the origin, and define it in pixels per world unit.
4. Specify the **rotation** to apply around the origin, in degrees

The scale parameter is defined in pixels per world reference unit, typically expressed in meters. It determines the distance at which the view is positioned from the world, which gives an indication of how much the view is zoomed in or out. This means that low-scale parameter values specify a zoomed-out view and high values specify a zoomed-in view.

Suppose that you want to use the `TLspViewXYZWorldTransformation2D` class to fit and center a map of the entire world in a view of 1000 pixels wide. Then, you need to supply:

- The center of the view as the view origin: a point with pixel coordinates (500, 500).
- The center of the world projection on the map as the world origin: a point with the XY coordinates (0,0).
- The scale parameter is 1.0/40.000, or 1000 pixels of view width for 40000 kilometers of world circumference.
- The rotation is 0 degrees.

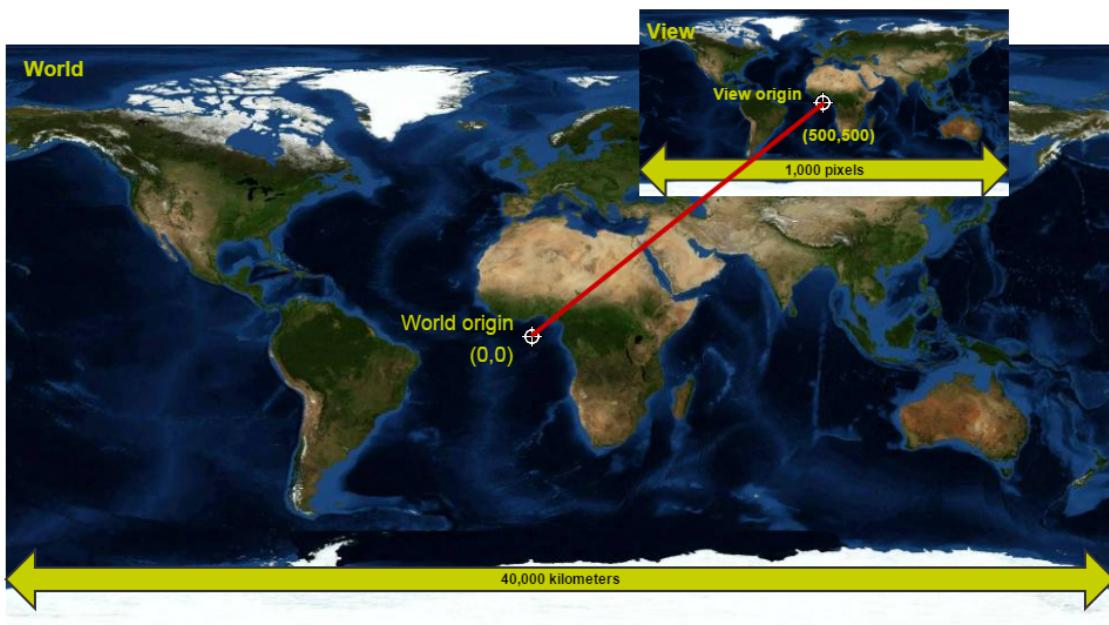


Figure 61 - Positioning a world map in your view



To position a view, it is often easier to use `TLspViewNavigationUtil` instead of writing the code to manipulate the view yourself. `TLspViewNavigationUtil` allows you to set the view so that the data in a specific layer fits into it. You can also fit or center a view to specific bounds. For more information, see [Section 20.2](#).

20.8.2 Positioning a 3D view

For 3D views, the API provides the `TLspViewXYZWorldTransformation3D` class, which makes abstraction of the [viewing frustum](#) that defines the view in 3D. This class allows you to position the view point using two similar, but subtly different methods: `lookAt()` and `lookFrom()`. These methods take the same arguments:

- The world reference of the view in which the `TLspViewXYZWorldTransformation3D` is being applied.

- A location in world coordinates.
- A distance to be maintained between the view point and the reference point.
- Yaw, pitch and roll angles to determine the orientation of the viewer.

From this parameter input, `TLspViewXYZWorldTransformation3D` derives an orthogonal coordinate frame. The frame is defined by three properties:

- The eye point: the position of the viewer in world coordinates.
- The reference point: a point towards which the viewer is oriented. When projected, this is a point in the middle of the screen.
- The up vector: a vector that indicates, in world coordinates, the direction of the vertical axis of the screen.

The difference between the `lookAt()` and `lookFrom()` methods resides in the way they derive the eye point, reference point and up vector from their arguments:

- `lookAt()` places the reference point at the specified location, and then uses the given distance, yaw, pitch and roll to compute the eye point. Use this method to orbit the view around a particular point, by keeping the location constant and varying the yaw and pitch angles.
- `lookFrom()` places the eye point at the specified location, and then uses the given distance, yaw, pitch and roll to compute the reference point. Use this method to let the viewer look around freely from a fixed position.

20.8.3 Navigating on a Lightspeed view

Navigation on a Lightspeed view boils down to configuring the `ALspViewXYZWorldTransformation` it uses. This is not trivial in most cases, so the LuciadLightspeed API offers several components to help you accomplish this:

- Navigation controllers: controllers that interpret input events and adapt the view to world transformation accordingly. [Section 3.3](#) describes the default navigation controllers and how to attach them to a view. [Section 26.2](#) describes how to create or customize your own controller.
- On-map navigation controls: a set of GUI components that you can control with the mouse to navigate the view. These are `TLspCompassNavigationControl`, `TLspPanNavigationControl`, and `TLspZoomNavigationControl`. `TLspNavigationControlsBuilder` allows you to construct such navigation control components.
- The class `TLspViewNavigationUtil`: a utility class to navigate programmatically.

`TLspViewNavigationUtil` offers various high-level methods for navigation:

- Panning in the view
- Zooming the view
- Rotating the view
- Fitting the view to an area. See also [Section 20.2](#).
- Panning the view to an area

These operations can either be performed immediately, so that the view snaps to the requested state without transition, or as an animation. This allows for a smooth transition from the current state to the new state.

20.9 Using animations

LuciadLightspeed provides a simple animation framework which can be used to improve an application's user experience. Animations can be used to create smooth transitions between viewpoints, fade-in and fade-out effects, and so on.

The animation framework consists of two main components:

- The **animation manager** provides global timing and bookkeeping.
- **Animations** are implemented by the user and submitted for playback to the animation manager.

The **animation manager** is implemented by `ALcdAnimationManager`. This class is a singleton: only one instance can be started per application.

The animation manager uses an internal clock to keep track of time. At every clock tick, the manager tells all currently running animations what the current time is, allowing the animations to take appropriate action. The clock ticks happen at a user-specified frequency: all animations are synchronized to the same frame rate, 60 frames per second for example. If you want, you can change the frame rate. If the animations cannot be updated at the desired rate because they take up too much CPU time, frames will be dropped. Consequently, animations are updated with larger time steps.

When an animation is submitted to the animation manager with the `putAnimation()` method, it is associated with a particular object. For instance, if an animation is used to slide a toolbar into the view, it could be associated with the Swing component representing that toolbar. This is a convenience mechanism that you can use to prevent simultaneously running animations from interfering with each other. If `putAnimation()` is called with the same object parameter twice, the animation manager terminates the first animation associated with the object before starting the second one.



Animations that change the view's camera in a Lightspeed view should use the view's `ALspViewXYZWorldTransformation`.

Animations are defined by the interface `ILcdAnimation`. An animation is essentially a class that gets notified by the manager when the clock has ticked. To notify the animation, the animation's `setTime()` method is called. The current time is passed to this method as an argument to determine the elapsed time relative to the time at which the animation was started. What happens in the `setTime()` method is entirely up to the developer of the animation. Typically, some properties of an object, such as position, color and transparency, are `interpolated` from an initial state to a target state over the duration of the animation.

An animation can specify its own duration, and can indicate whether it wants to be played back in a repeating loop or not. These properties are taken into account by the animation manager: if an animation has reached the end of its duration, it is detached from its corresponding object and is no longer notified of clock ticks.

The animation manager may contain animations associated with objects relevant to multiple views. Some animations may change the property of a view, causing it to repaint itself, while other animations do not affect the view. Therefore, the animation manager does nothing to

invalidate the appropriate views. This is entirely the responsibility of the individual animations. LuciadLightspeed provides the abstract `ALcdAnimation` which invalidates any views provided to it at construction.

The package `samples.lightspeed.customization.style.animated` contains an example of how the animation framework can be used to smoothly vary the fill color of a domain object. See [Section 22.9.3](#) for more information.

CHAPTER 21

Working with layers in a Lightspeed view

Beside the fundamental layer management activities described in [Chapter 6](#), such as the creation of layer lists and hierarchical layer structures, LuciadLightspeed allows you to control the layers in your view in more advanced ways. This chapter allows you to learn more about managing layers in a Lightspeed view, such as how to copy layer nodes between layer trees, how to filter objects from layers or how to retrieve information about painted objects.

21.1 Working with Lightspeed layers?

Layers are the LuciadLightspeed view components that process and group the objects of a model to take care of their visualization. A view can display one or more layers, which allows you to have objects from multiple data models on screen.

As you typically have multiple layers in one view, this chapter discusses various topics related to the management of your layers, as well as the management of the objects in your layer. For more information about the visualization of domain objects, refer to [Chapter 22](#).

21.1.1 Which layer to choose for your model

LuciadLightspeed offers several different kinds of Lightspeed layers. Which type of layer you choose depends on what you want to do with the model data in the layer.

The root interface for all layers is `ILspLayer`. Most likely, you will use one of two default implementations:

- `TLspLayer`. Use this layer type to create layers that can paint domain objects, and that allow users to edit those objects. This layer type is the default implementation of `ILspInteractivePaintableLayer`. It also implements `ILspEditableStyledLayer`, which lets you use a styler to link styling to the objects in the layer.
- `TLspRasterLayer`. Lets you display domain objects such as rasters, multilevel rasters and Earth tilesets. You can define and control the styling of the raster objects. This layer is an implementation of `ILspEditableStyledLayer`.

The view displays the layers in a certain order. To optimize layer ordering, the view also considers the type of the layer.

There are four predefined layer types, represented by `ILspLayer.LayerType`. The performance of each layer type has been optimized for a specific data type and user interaction level:

- **BACKGROUND**: intended for data that rarely changes, and resides in the background of the view. Background layers have been optimized to achieve the best possible performance for static data. To obtain optimal performance for data that changes more frequently, use one of the other layer types in this list.
- **INTERACTIVE**: intended for static data that does not change often, but may be manipulated by the user. This layer type is a good choice for layers that are frequently switched on and off, or for layers in which the styling changes often. In the event of user interaction with the layer, the interaction will be smooth.
- **EDITABLE**: intended for data that changes more often, because a user can manipulate it extensively, for example vector data that is graphically edited by the application user.
- **REALTIME**: intended for data that changes constantly over time, like moving aircraft.

For more information about the visualization of the data in your models, see [Chapter 22](#).

21.1.2 Building a layer

There are many configuration options available to you during the setup of a layer. LuciadLightspeed offers layer builders to construct your layers. These layer builders allow you to specify just those layer aspects that you want to configure. Defaults are chosen for the other aspects, so that most of the configuration can be left to the builder. Layer builders let you select:

- The styling for your objects
- Whether selection is activated
- Whether editing is activated
- Whether labels are visible
- Whether objects are filtered or not

To construct layers that are well-suited to display shapes, you can use `TLspShapeLayerBuilder`. By default, the construction of a layer with `TLspShapeLayerBuilder` results in a `ILspInteractivePaintableLayer` of the `EDITABLE` type.

To construct layers that can display raster layers, use `TLspRasterLayerBuilder`. This builder creates `TLspRasterLayer` objects. You can choose to set the layer type to either `INTERACTIVE` or `BACKGROUND`.

21.2 Managing selection between views

Section 6.5 describes how you can select domain objects from a model that is contained in a layer. Consider the case in which multiple views share the same `ILcdModel` but do not share the layer that contains the model. When a change in selection in one of the layers occurs, all layers that contain the same model should be notified of the change and the method `selectObject` has to be called on these layers to synchronize the selection.



Layers cannot be shared between Lightspeed views.

The utility class `TLcdSelectionMediator` provides a convenient way to handle selection amongst different `ILcdLayered` (or view) objects. You need to initialize the class with two arrays of `ILcdLayered` objects. One array acts as a **master** or **source** and contains all the `ILcdLayered` objects that listen to selection changes in the `ILcdLayer` objects it contains. The other array acts as a **slave** and contains all the `ILcdLayered` objects that should be synchronized upon a selection change in one of the master objects. The selection of a layer that belongs to a slave `ILcdLayered` is updated if that layer refers to the same `ILcdModel`.

To keep views synchronized constantly, you can select the same array to act as master and slave.

21.3 Filtering objects from a layer

In some cases, you may want to filter certain objects from a layer and exclude them from visualization or other actions such as selection or displaying tool tips. Cases where such a filter is useful are:

- You want to focus on the most relevant data during analysis
- You want to enhance rendering performance
- You want to keep the map view clear

There are four ways to filter objects in a layer:

- Define a **scale range** on the layer to only display objects in a limited range of scales (in 2D) or within a certain distance range from the view point (in 3D). For more information, see the scale range getter and setter methods of `TLspShapeLayerBuilder`, `TLspRasterLayerBuilder`, `TLspLayer` and `TLspRasterLayer` in the API reference.
- Write a **generic filter**: every object of the `ILcdModel` is given to this filter. You can use any criterion to either include or exclude the object. For example, use a generic filter to only display cities with more than one million citizens. An advanced implementation is available as OGC filtering, which is described in [Chapter 50](#). Refer to the filter getter and setter methods of the layer builders in the API reference for more information.
- Set a **minimum object size**: when zoomed out, small objects might no longer be relevant. For example, displaying runways on a scale of 1:100.000.000 does not make much sense. `TLspLayer.setMinimumObjectSizeForPainting` allows you to set a size threshold (in pixels) for objects. Objects that are smaller than the given threshold, are automatically excluded. You can disable this behavior by setting the threshold to 0. Refer to the API reference for more information.
- Use the painter's **styler** to exclude individual objects from visualization. While you can define complex filtering rules by writing your own filter, you can only use `ALspStyleTargetProvider` to separate individual objects from a group of objects, and make those invisible. For more information, see [Section 22.9.4](#).

For more information about layer builders and stylers, see [Chapter 22](#).

21.4 Retrieving information about painted objects from a layer

Often, you need to know visual details about the objects that are currently displayed. For example, you need to know the view or world bounds of an object, or you need to know what objects are displayed at a certain screen location.

You can use the `ILspInteractivePaintableLayer.query` method to get this information. It accepts a query object, and returns the relevant information. The return value is dependent on the query. This allows you to write concise queries. This general construct is shown in Program 88.

```
1 queryResult = layer.query(queryObject, context);
```

Program 88 - Retrieving information through `ILspInteractivePaintableLayer.query`

There are various pre-defined queries:

- `TLspPaintedObjectsQuery`: returns all domain objects overlapping with a certain region on the screen. Only currently painted objects are taken into account.
- `TLspPaintedObjectsTouchQuery`: returns all domain objects overlapping with a certain screen region or screen point. The query also returns the coordinates at which the object touches the region or point. Only currently painted objects are taken into account.
- `TLspPaintedObjectsBoundsQuery`: returns all domain objects overlapping with a certain screen region or screen point. The query also returns the view or world bounds of those objects. Only currently painted objects are taken into account.
- `TLspIsTouchedQuery`: indicates whether the specified domain object or label is painted at a certain screen point. The query also returns the coordinates at which the object touches this point. The object does not have to be painted yet.
- `TLspBoundsQuery`: returns the view or world bounds of the given domain object or label. The object does not have to be painted yet.

Note that these queries are delegated to the painter implementations. The painters take the actual representation on the screen into account as much as possible. If the size of the object is changed on the screen through a specific styling or an icon, for example, the painters will use the size of the object on the screen to perform the query.

Program 89 demonstrates how the high light controller uses query's to determine the object under the mouse cursor. It first determines the different paint steps that the view takes when painting a view (not demonstrated in the program). Then it performs a `TLspPaintedObjectsTouchQuery` for each of the paint steps and keeps track of the object that is closest to the viewer. Note that we override the `touched` methods so we can handle the results immediately as they become available instead of copying them into a list first.

```

1 // Perform a query for each of the paint steps and update the highlighting candidate
2 final CandidateChooser candidateChooser = new CandidateChooser(aViewPoint);
3 for (ILspPaintingOrder.PaintStep paintStep : paintSteps) {
4     // Query for the current paint step
5     final ILspInteractivePaintableLayer layer = (ILspInteractivePaintableLayer) paintStep.
6         getLayer();
7     layer.query(
8         new TLspPaintedObjectsTouchQuery(
9             paintStep.getPaintRepresentationState(),
10            aViewPoint,
11            SENSITIVITY
12        ) {
13            @Override
14            public boolean touched(ALspWorldTouchInfo aTouchInfo) {
15                // We have touched an object at a point in world coordinates, update the
16                // candidate
17                candidateChooser.touched(layer, getPaintRepresentationState(), aTouchInfo);
18                return true;
19            }
20            @Override
21            public boolean touched(ALspViewTouchInfo aTouchInfo) {
22                // We have touched an object at a point in view coordinates, update the
23                // candidate
24                candidateChooser.touched(layer, getPaintRepresentationState(), aTouchInfo);
25                return true;
26            }
27        },
28        new TLspContext(layer, view)
29    );
30 }
31 return candidateChooser.getCandidate();

```

Program 89 - Using query's to determine the object under the cursor.
 (from samples/lightspeed/customization/style/highlighting/HighlightController)

21.5 Changing the painting orders of layers

You can determine the order in which layers are painted in a view through an implementation of `ILspPaintingOrder`, which can be retrieved from or set on an `ILspView`.

For most users, the default implementation, `TLspPaintingOrder`, should be sufficient. It ensures that layers are painted in the order in which they are present in the view's layer tree, but also makes sure that selected objects are painted on top, and that labels appear above the domain object to which they belong. For more information about implementing custom ordering behavior, refer to the API documentation of `ILspPaintingOrder` and `TLspPaintingOrder`.

CHAPTER 22

Visualizing domain objects in a Lightspeed view

This chapter covers the most important aspects of visualizing domain objects by adding them to a view, and determining the way they should look on screen. Before we go into the setup of your layers, styles and stylers, we introduce the main concepts of visualization in a Lightspeed view.

22.1 Object painting in a Lightspeed view

This section briefly explains the role of the main players in LuciadLightspeed visualization.

22.1.1 Layers take care of object visualization

Object painting in a Lightspeed view starts from the layer, the container that visualizes the objects of a model in a view.

Different layer types support different data types. LuciadLightspeed offers several layer types, each of which offers its specific benefits to the visualization of a certain model object type. While some layer types are tailored to the visualization of rasters, other layers are particularly suitable for shape objects, positioned at particular coordinates in the world. Regardless of the data types they display, LuciadLightspeed layers support multiple ways of visually representing an object. These modes of object visualization are called paint representations.

Paint representations. Raster layers and shape layers can display the same object in different ways. A shape layer, for instance, typically uses the geometrical information about the object to visualize the object.

However, the shape layer may also paint the objects as labels, positioned at screen coordinates. In this case, the layer uses additional information available about the object, such as text, to construct the label.

The layer can even paint both the shapes and the labels, and also display shape handles that allow users to manipulate, or edit, the shape. In the latter case, the layer uses three paint representations of one and the same object:

- The object's BODY paint representation for the shape

- The object's LABEL paint representation for the label
- The object's HANDLE paint representation for the edit handles

This list is not final. You can expand this list with other paint representations.

Each of the paint representations has a specific place in the layer's painting cycle. The layer uses a specific order to paint these representations into the view. By default, it paints shapes first, then it paints the labels, and finally the handles. This painting order can be customized.

Paint states. An additional factor taken into account for object painting is paint state. An object's paint state indicates whether an object is selected, or in the process of being edited. If an object is selected, for instance, there usually has to be some visual indication of this state on screen. The paint state allows the layer to distinguish between all those visualization requirements, so that it can paint selected shapes (BODY paint representation, SELECTED paint state) very differently from deselected labels (LABEL paint representation, REGULAR paint state). To specify how these paint representations look, you can make use of styles. See [Section 22.1.2](#) for more information.

Painters. To display the paint representation, the layer makes use of painter objects. Painters take care of the mechanics to render a visual representation of an object on screen. For instance, to paint shapes, shape handles and labels, the layer uses both a shape painter and a label painter.



LuciadLightspeed offers layer builders to construct your layers. If you use such a layer builder, painters are automatically created as soon as a layer is set up. You only very rarely need to work with painters directly, let alone implement your own painters. If you want to change the way in which an object is visualized, it is far more convenient to make use of the myriad possibilities of the LuciadLightspeed styling API.

For a discussion of these concepts at the level of the API, see [Section 22.12](#).

22.1.2 What if you want to change the look of your object?

Your object's styling determines if it is visible on the screen. If you do not define any specific styling for the objects of a new model, LuciadLightspeed displays them in a default styling. Styling can be anything from a line width to a shape's transparency. To change the way objects look, you can make use of the LuciadLightspeed styles and stylers.

Styles. Styles offer you an abundance of options to manipulate the look of your domain objects. There is no need for any painter customization. You can set up your styling so that an object's paint representation and paint state influence the styling characteristics of an object, or a group of objects. You can go so far as to add a nice animation to highlight a certain object, or to apply distinct styles to different parts of a shape's geometry.

The first thing you need to do is build your styles, like a line style object or a fill style object, using a style builder. Style builders allow you to focus on the construction of the look of your style, rather than on LuciadLightspeed style representation details. Next, you need to apply the styles to one or more objects. To link styles to objects, you use stylers.

Stylers. Stylers reside on the layer, and provide one or more styles to the objects that are being visualized. They determine which styles can be applied to which objects. Each available style is also a styler. Stylers allow you to evaluate whether objects fulfill a number of conditions required to receive a certain style. You can set up stylers so that they single out one or more objects in a group of objects to apply a distinct styling to, while they apply generic styling to all the other objects in the group.

Style collectors. To actually apply the appropriate styles to the objects, you must pass these styles to a style collector, supplied to the styler by the painter. Style collectors are passed to the styler as arguments, and provide the methods necessary to apply styles to objects.



Concepts such as paint representations, paint states and style collectors allow for the organization of individual objects into groups. The approach of grouping objects for styling contributes greatly to an efficient use of OpenGL visualization techniques.

22.2 Setting up your layers

The first step in the visualization of your domain objects is the setup of your layers. The LuciadLightspeed layer builders are especially suited to accomplish this goal. You can make use of layer builders for the setup of various layer types. In this section, the setup of vector layers, raster layers ad density plot layers is discussed.

22.2.1 Quickly and easily creating vector layers: `TLspShapeLayerBuilder`

When you are writing a layer factory for vector data, always start from `TLspShapeLayerBuilder`. It offers you a convenient way to create your layer, without having to consider too many details. In fact, it hides the actual implementation details of your layer setup. This way, LuciadLightspeed tries to choose the optimal setup for you.

If you use `TLspShapeLayerBuilder`, you get layers of the type `ILspInteractivePaintableLayer`. These layers paint and style model objects using stylers.



Layers built with `TLspShapeLayerBuilder` support all predefined `ILcdShape` implementations in LuciadLightspeed.

22.2.2 Customizing vector layers

You can configure many different styles if you use `TLspShapeLayerBuilder`. If the possibilities of `TLspShapeLayerBuilder` do not cover your requirements after all, you can always set up a custom layer and painter combination. This provides you with more options, but may take a little more programming.

In that case, you will most likely use `TLspLayer` in combination with `TLspShapePainter` for vector painting.

22.2.3 Creating raster layers

Beside vector data, you probably also want to visualize raster images and terrain using raster layers. We recommend the use of `TLspRasterLayerBuilder` for the creation of raster layers

wherever possible. If you use such a raster layer builder, LuciadLightspeed automatically tries to choose the optimal setup for you. Alternatively, you can directly create a `TLspRasterLayer`.

A raster layer supports the following domain objects:

- **Rasters:** Regularly and irregularly structured rasters in general, modeled by `ALcdBasicImage`, `ILcdRaster` and `ILcdMultilevelRaster`. Rasters are used by many data formats in LuciadLightspeed, such as GeoTIFF and JPEG2000 for example. See [Chapter 13](#) for more details.
- **Tilesets:** regularly structured raster data, modeled by `ILcdEarthTileSet` with either `CoverageType.IMAGE` or `CoverageType.ELEVATION`, depending on the data type in the tile set. Tilesets are used in the Luciad Earth repositories, for example. See [Chapter 14](#) for more details.

You can influence the behavior of a raster layer by setting the `LayerType`:

- **BACKGROUND:** indicates that the raster is used as background data in the view and rarely changes.
- **INTERACTIVE:** indicates that the layer should support smooth interaction, such as changes of the visibility or style of the raster data.



It is preferable to keep background raster layers at the bottom of the layer stack. This allows the Lightspeed view to optimize performance and memory consumption, by combining and caching the data of the different raster layers.

Visualizing rasters at an appropriate scale Aerial or satellite image files can be very large, and may be over a gigabyte in size. Such files take a long time to load into the application completely, although a computer screen cannot display these images in full detail. To prevent excessive image loading times when you have zoomed out too much to view any image detail anyway, LuciadLightspeed displays the contours of the raster data, on the area of the map that the full image file would cover. Actual image content is omitted, and the contours are filled with a hatch pattern. To make the raster data appear, just zoom in. The use of this technique is key to prevent the loading of excessive amounts of data at a scale at which it was never intended to be viewed.



Figure 62 - Raster contours filled with a hatch pattern

22.2.4 Creating density layers

Visualizing density plots of vector data is done with the `TLspShapeLayerBuilder` in combination with `ALspDensityStyle` instances. Doing so sets up a layer that visualizes the amount of overlap between vector data. You can modify the color used for the density plot by letting the styler also provide a `TLspIndexColorModelStyle` instance.

The `lightspeed.density` sample demonstrates density plots in LuciadLightspeed. The sample uses flight trajectory lines to provide an overview of trajectory density per area.

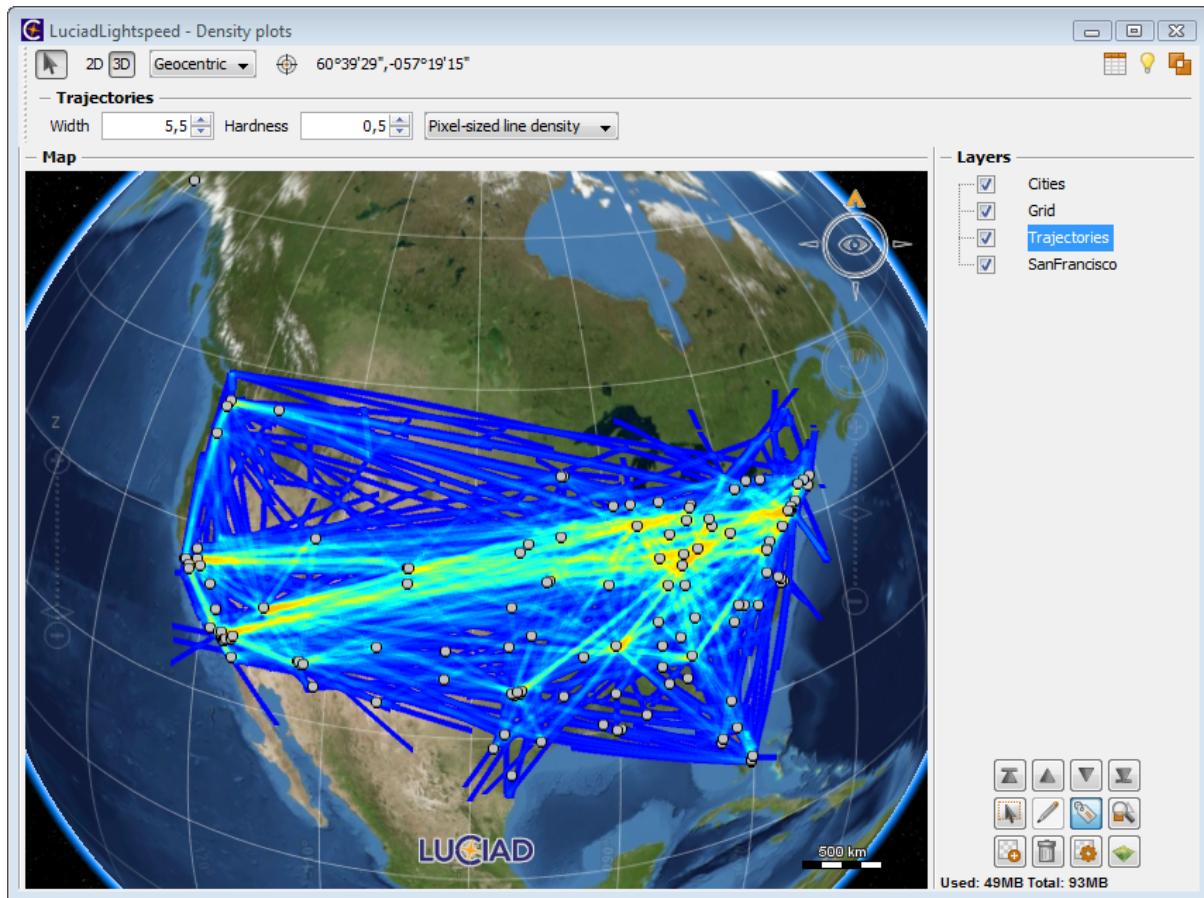


Figure 63 - The density plot in the lightspeed.density sample.

For more information about density styling, see [Section 22.8](#).

22.3 Using styles

`ALspStyle` is the abstract base class from which all styles are extended. As a user of the API, you do not have to create your own extensions. It is recommended to use the concrete styles that are available in the API instead. These are discussed in [Section 22.5](#) and [Section 22.7](#).

22.3.1 Properties shared by style implementations

All styles that extend from `ALspStyle` have a number of properties in common:

Immutable The properties of a given style can never change, but the API does allow you to easily derive a new style based on an existing one, as described in [Section 22.4.2](#).

Simple properties Styles only contain simple properties that can be queried. This makes it easier to write user interfaces that show the properties of a given style, and allows persisting styles.

Combinable All styles are kept as simple as possible, but you can combine individual styles with each other to achieve a certain effect. This is why there is no halo property in a basic line style. You can achieve a halo effect by combining two line styles, however.

22.3.2 Setting visual style properties

Two styling properties are available at the level of `ALspStyle`, and are shared by all styles:

- `isTransparent()`: determines whether objects are transparent or opaque. Transparent objects receive special treatment from the view to ensure that their colors are correctly blended with other objects. It is important that this flag is set correctly: if an object is marked as opaque but a non-opaque fill color is specified, the object and other objects which overlap with it on the screen may be rendered incorrectly. Conversely, flagging an object as transparent but using an opaque fill color adversely affects performance.
- `getZOrder()`: Determines which place a style takes in an overlapping range of styles. If you define multiple styles of the same type for a given object, you can also define the Z-order of each style. This property reflects that. Styles are applied from lowest to highest Z-order.



Assigning a different Z-order to each object may significantly affect performance. Therefore, try to keep the assignment of Z-orders to individual objects to a minimum.

22.3.3 Choosing an elevation mode.

Many styles offer the property `getElevationMode()`. The `elevationMode` property is a styling property that determines how objects should be positioned relative to the terrain. This property can have one of four values: `ON_TERRAIN`, `ABOVE_TERRAIN`, `ABOVE_ELLIPSOID`, or `OBJECT_DEPENDENT`.

`ON_TERRAIN` means that the object is draped over a terrain. Draping allows you to paint directly onto a terrain. It is used in 3D to ensure that the objects follow the surface of the terrain. Anything can be painted in this mode, but it makes the most sense for objects that do not have an elevation, and which are not drawn in alignment with the screen.

`ABOVE_TERRAIN` is only available for `TLspIconStyle`, `TLspLineStyle` and `TLspFillStyle`. It means that the styled objects are painted above the terrain. The elevation of the object is determined by adding the object's elevation to the terrain's elevation.

For fill and line styles, only one terrain elevation is computed for a given geometry or object. This way, fluctuations in the underlying terrain are not visible in the geometry of the object.

Icons are normally centered around this elevated point, so even with this elevation mode, they can be partially obscured by the terrain, depending on the view angle. To prevent this, you can either set a fixed offset on the `TLspIconStyle`, or add a `TLspViewDisplacementStyle`, so that an icon can be moved above the terrain dynamically, depending on the view angle.

`ABOVE_ELLIPSOID` means that the absolute elevation specified for the domain object is used for the painting of the object. As a result, the terrain may obscure those objects.

`OBJECT_DEPENDENT` means that the object type and the object geometry determine which of the other three elevation modes is most suitable. This is currently the default elevation mode applied to stylers and most styles. It is only available for `TLspIconStyle`, `TLspLineStyle` and `TLspFillStyle`.

Use this elevation mode as a style parameter if you are writing a styler for an input format with potentially diverse types of objects: streets, rivers, and icons, for example. If you are using a `TLspFillStyle` or a `TLspLineStyle`, an `OBJECT_DEPENDENT` elevation mode results in objects draped on the terrain if the objects do not have elevation themselves. If the objects have

elevation, they are painted ABOVE_ELLIPSOID.

If you are using `TLspIconStyle`, the icon objects are painted ABOVE_TERRAIN if they do not have any elevation themselves, meaning that their Z value is zero. The point is painted ABOVE_ELLIPSOID if it does have elevation.



If you know up front what type of objects must be visualized, choose one of the other, more specific elevation modes. Base your choice on the type of input data. If you want to display street data for example, choose the ON_TERRAIN elevation mode.

22.4 Creating styles with builders

All styles expose a `Builder` that allows you to create new styles from scratch, or copy the properties of an existing style.

22.4.1 Using a builder to create new styles

Program 90 illustrates the creation of a `TLspLineStyle`.

```
1 TLspLineStyle s = TLspLineStyle.newBuilder()
2   .color( Color.black )
3   .width( 3f )
4   .elevationMode( ON_TERRAIN )
5   .build();
```

Program 90 - Using a builder to create a line style

This code fragment creates a line style which is three pixels wide, solid black and draped on the terrain in 3D.

22.4.2 Deriving a new style from an existing one

Styles are immutable. If you want to modify a style, you cannot simply change one of its properties. Instead, the LuciadLightspeed API provides a convenient way to create a new style from an existing style: base your style builder on the existing style, and add the required property modification. This is shown in Program 91.

```
1 TLspLineStyle existingStyle = ...;
2 TLspLineStyle s = existingStyle.asBuilder()
3   .width( 4f )
4   .build();
```

Program 91 - Deriving a new style

22.5 Defining vector styles

Vector styling can be defined with a number of elementary building blocks. Once we know the properties of each one of them, we can combine the building blocks to define richer styling.

22.5.1 Visualizing simple lines with `TLspLineStyle`

Both lines and surface outlines can be styled with a `TLspLineStyle`. For a basic line, you can simply specify a color and a width in pixels, but you can also use textures and dash patterns to draw more complex lines. Halo effects can be created by applying two line styles with a

different width to the same object. If you wish to specify line widths in a world unit, in meters for example, you should have a look at [Section 22.5.8](#) instead.

Using `TLspLineStyle` is the most efficient way to visualize lines, and suited for visualizing a large number of objects simultaneously. One common way to paint lines efficiently is to use simple lines at large map scales, and use more complex line styles when the user zooms in on the map. This is shown in the `lightspeed.streets` sample.

22.5.2 Filling areas with colors and patterns with `TLspFillStyle`

Fill styles are mostly used to specify a fill color for an object. More advanced properties include a stipple pattern or texturing. Keep in mind that only closed surfaces can be styled with a `TLspFillStyle`.

Only one fill style should be specified for any given object. To specify more than one fill style, use an `ALspStyleTargetProvider` to split the object into separate parts. To learn more about style target providers, see [Section 22.9.4](#).

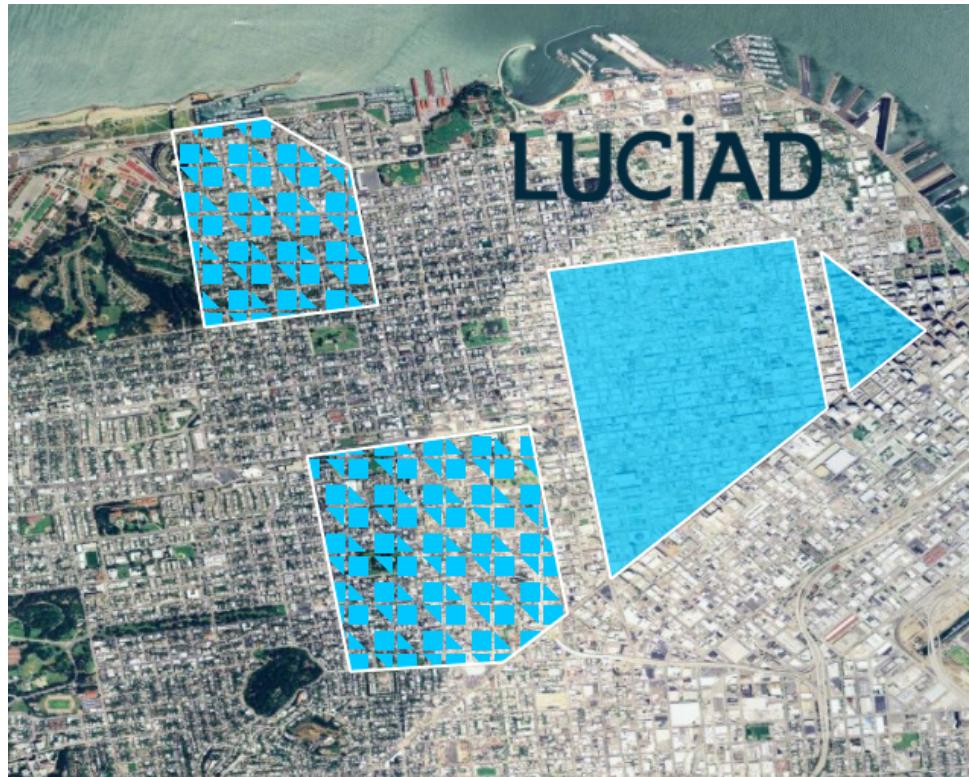


Figure 64 - Styling areas with `TLspFillStyle`

22.5.3 Resolving overlaps for shapes with fill and line styles

`TLspShapePainter` uses the most optimal approach to visualize shapes by default, by painting shape fills and shape outlines in separate batches. This approach produces the required results for layers with objects that do not overlap, such as countries, buildings, and so on. However, when different shapes within a layer cover the same region, this approach might result in unexpected behavior: outlines appear on top, even for a shape that lies below other shapes. This is illustrated by the figure on the left in [Figure 65](#).

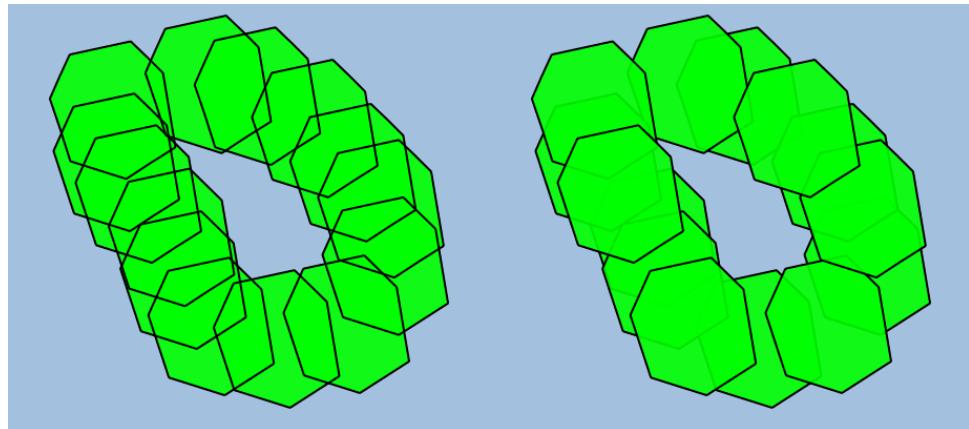


Figure 65 - Left - Lines are painted on top of fills by default. Right - Fills and lines interleaved.

The shape painter can be configured to resolve such overlaps, and to produce the results on the right of [Figure 65](#). In the figure on the right, the fill style and the line style of a single shape remain together. This painter mode has a performance and memory impact. Activate it only if you really need it.

To configure the overlap resolution mode, `TLspShapePaintingHints` is used, as shown here with `TLspShapeLayerBuilder`:

```

1 TLspShapeLayerBuilder.newBuilder()
2     .model( model )
3     .paintingHints(
4         TLspShapePaintingHints.newBuilder()
5             .overlapResolution( ELspQualityHint.NICEST )
6             .build()
7     )
8     .build();

```

Program 92 - Setting shape painting hints

The `QualityHint` allows you to indicate that you want to use the painter's default settings, or that you give preference to either painting performance or visual quality. You can select either `AUTOMATIC`, `FASTEST`, or `NICEST` respectively.

22.5.4 Visualizing a position at ground level with `TLspVerticalLineStyle`

In a 3D view, it may be useful to visualize the projection of a point on the ground. One way to do this is by simply drawing a perpendicular line from the earth surface to the point. You can achieve this most efficiently with `TLspVerticalLineStyle`, which allows you to specify the color and the width of the line. It is applied to objects in exactly the same way as `TLspIconStyle`, so that they can easily be used together.

A `TLspVerticalLineStyle` does not have any impact on 2D views. To see an effect, switch to 3D.

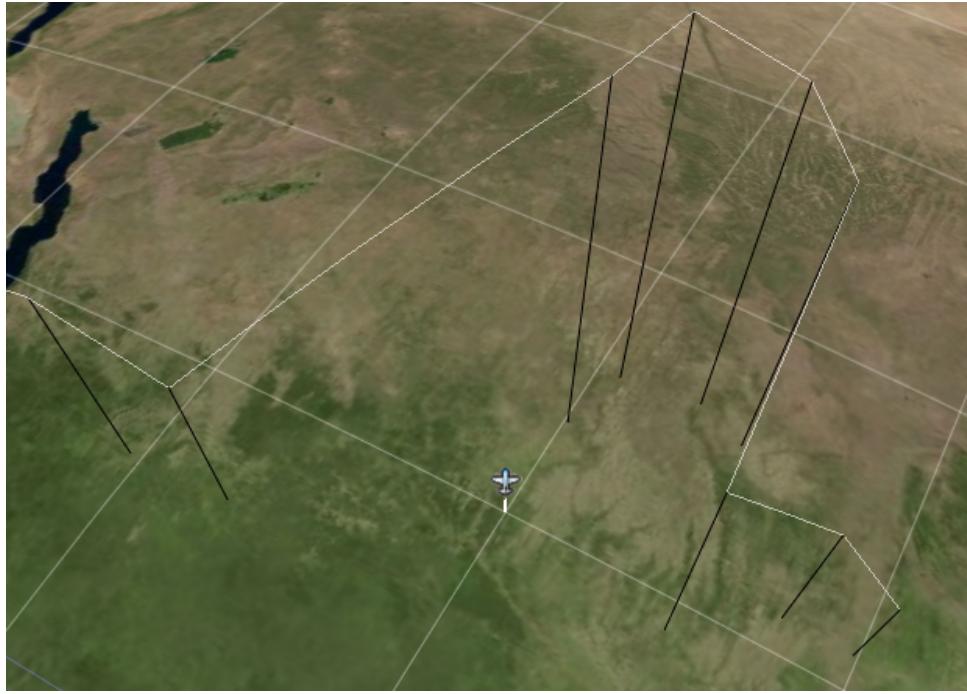


Figure 66 - Projecting points on the ground with `TLspVerticalLineStyle`

22.5.5 Painting point symbology with `TLspIconStyle`

A `TLspIconStyle` defines an icon that needs to be painted parallel to the screen. It allows you to change important properties of the icon, like the size, rotation, and the positioning. Multiple icon styles can be used for the same object if necessary. Usually, this style is applied to `ILcdPoint` objects, but it can also be applied to a `ILcdPointList`. In that case, the icon is painted for every point in the list.

Rotating an icon based on world orientation

A unique property of `TLspIconStyle` is that it allows you to efficiently paint an object with world coordinates in view space. This is especially useful in 3D projections. Assume that you want to paint an arrow on the view that shows the direction of a moving point: you can do this by computing the corresponding world coordinates and paint them with a line style, but it may involve recomputing the arrows each time the view changes.

A better solution consists of the following steps

1. Make sure your points implement `ILcdOriented`.
2. Paint one arrow in an icon.
3. Enable the `useOrientation` property of `TLspIconStyle` to rotate the arrow appropriately.

22.5.6 Painting symbology along lines with `TLspComplexStrokedLineStyle`

`TLspComplexStrokedLineStyle` supports complex patterns that can be applied to a line. This allows you to implement just about any complex line symbology. The `lightspeed.style.strokedline` sample shows how to use this style.

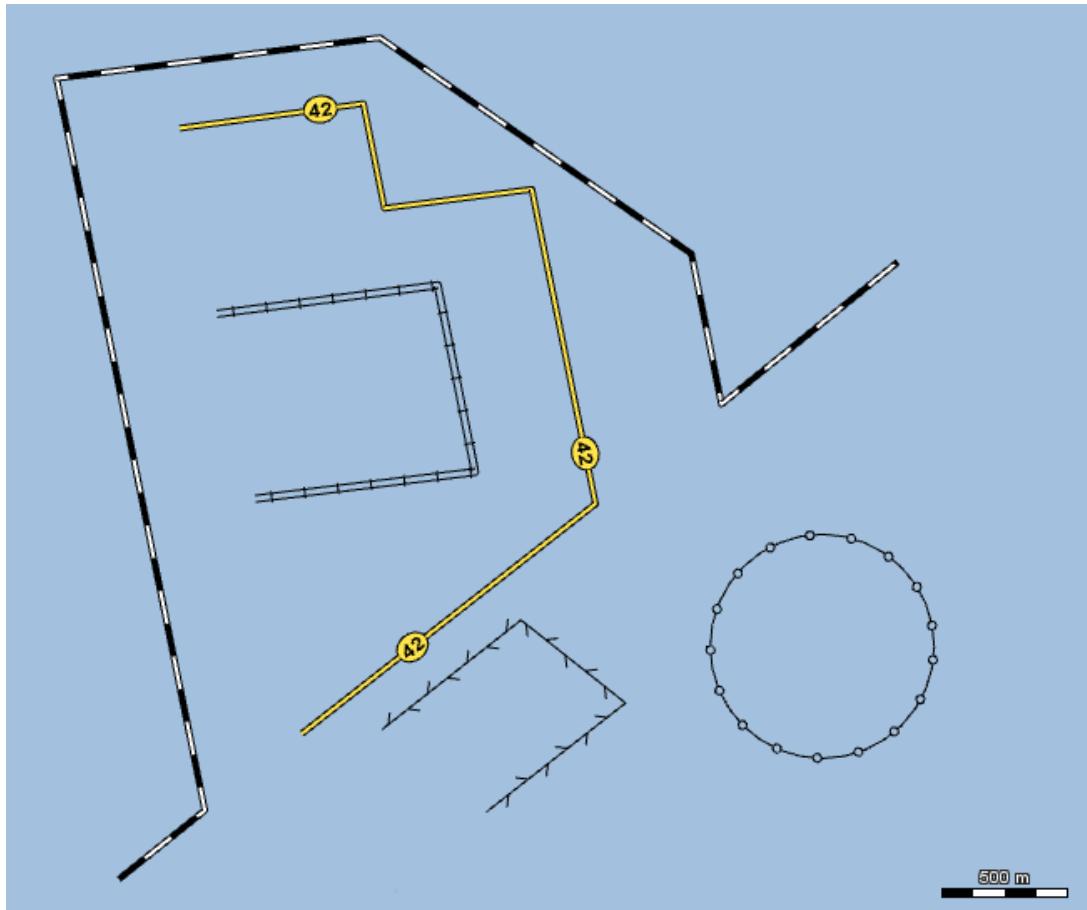


Figure 67 - Lines styled with `TLspComplexStrokedLineStyle` in sample `lightspeed.style.strokedline`

A `TLspComplexStrokedLineStyle` consists of `ALspComplexStrokes`, along with a few properties that affect the visualization, such as the elevation mode and halo settings. The following section shows how to create a `TLspComplexStrokedLineStyle`.

Creating a `TLspComplexStrokedLineStyle`

A `TLspComplexStrokedLineStyle` defines how a line is painted. It combines a number of stroke patterns, that are for example repeated along the line, or added as a decoration at the start or end. A `TLspComplexStrokedLineStyle` can be created using a `Builder`. This builder makes it possible to define decorations and regular repeated strokes patterns. For example, the complex stroked line in the following figure contains an arrow decoration, a repeated block pattern and a dash as fallback. It is generated using the following code:

```

1 // Create an arrow stroke pattern
2 ALspComplexStroke arrow = arrow().type( ArrowType.OUTLINED ).size( 10 ).build();
3
4 // Create a block stroke pattern
5 ALspComplexStroke block = append( line().length( 0 ).y0( 0 ).y1( 8 ).build(),
6                                     line().length( 16 ).y0( 8 ).y1( 8 ).build(),
7                                     line().length( 0 ).y0( 8 ).y1( -8 ).build(),
8                                     line().length( 16 ).y0( -8 ).y1( -8 ).build(),
9                                     line().length( 0 ).y0( -8 ).y1( 0 ).build() );
10
11 // Create a dash stroke pattern
12 ALspComplexStroke dash = append( parallelLine().length( 4 ).build(), gap( 4 ) );
13
14 // Create a stroked line using these stroke patterns
15 TLspComplexStrokedLineStyle strokedLine = TLspComplexStrokedLineStyle.newBuilder()
16     .decoration( 1, arrow )           // Add the arrow as decoration
17     .regular( atomic( block ) )      // Repeat the block stroke pattern along the line
18     .fallback( dash )               // Use the dash pattern as fallback
19     .build();

```

Program 93 - Creating a complex stroked line



Figure 68 - A complex stroked line with an arrow decoration, a repeated block pattern and a dash as fallback

When a decoration or regular stroke pattern is omitted, for example when there is not enough space for it, or when it crosses a sharp corner, the fallback stroke is used.

Creating complex stroke patterns

Decorations, regular repeated patterns and fallback patterns can be created using complex stroke patterns. These patterns can be composed to form more complex strokes patterns. For example, the block pattern from previous example was created by using 5 appended line primitives.

The following figure shows many of the possible complex stroke pattern primitives.

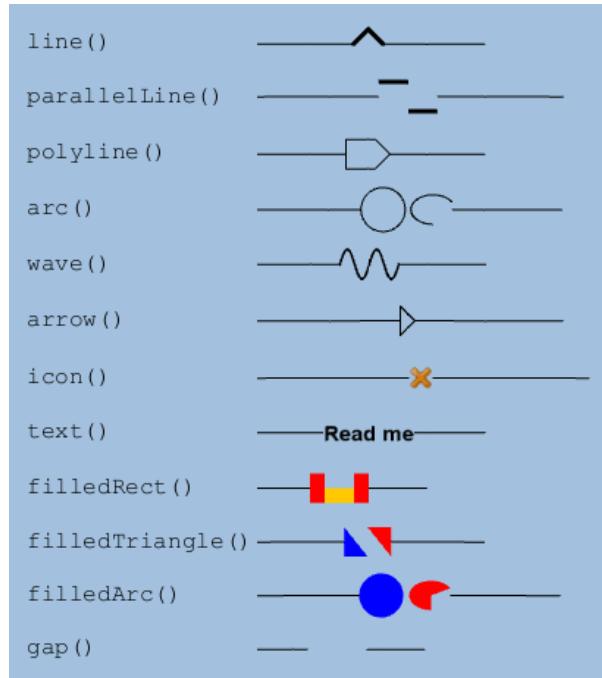


Figure 69 - Complex stroke patterns primitives

The following figure shows how primitives can be composed:

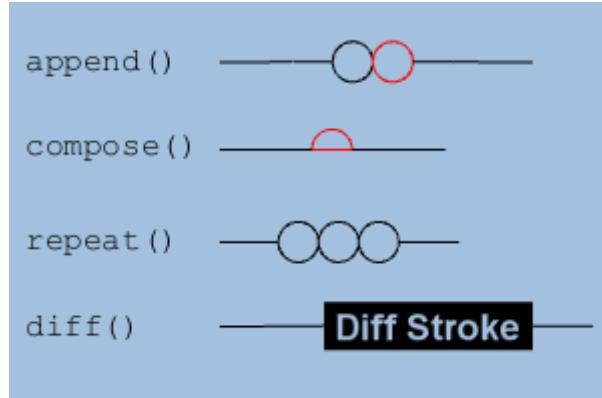


Figure 70 - Complex stroke patterns composition

22.5.7 Painting 3D point symbology: `TLsp3DIconStyle`

A `TLsp3DIconStyle` allows you to define a 3D symbol that is painted at a point location. Most of its properties are similar to the ones available in `TLspIconStyle`. LuciadLightspeed also allows you to paint these 3D icons in a 2D projection, which gives you a top-down view.

A common way to create 3D icons is to use one of the 3D file formats supported by LuciadLightspeed, such as Wavefront OBJ *.obj files, or OpenFlight *.flt files. The resulting mesh can be used to construct an `ILsp3DIcon`, which you can pass directly into the `TLsp3DIconStyle`. An alternative is to provide a 3D file path to the `TLsp3DIconStyle`. Using this method, you can display Collada *.dae files in addition to Wavefront and OpenFlight files. The `lightspeed.icons3d` sample shows you how to do this.

Painting 3D icons is less efficient than painting regular icons. Therefore, limit the number of 3D icons painted at the same time.

22.5.8 Painting streets with their actual size with `TLspWorldSizedLineStyle`

You can use `TLspWorldSizedLineStyle` if you want to show a road of 20 meters wide on the map, using its real size. Next to the color and width, you can also apply a texture.

A typical street style is achieved by combining two `TLspWorldSizedLineStyle` objects with a different width. The thinnest one represents the actual road, while the wider one represents the pavement next to the road. This is shown in [Figure 71](#). Note how the streets that are further away appear less wide, because perspective has been applied. This would not be the case with a regular line style that uses a fixed width in pixels.

The 'Streets' layer in the `lightspeed.streets` sample uses this style.

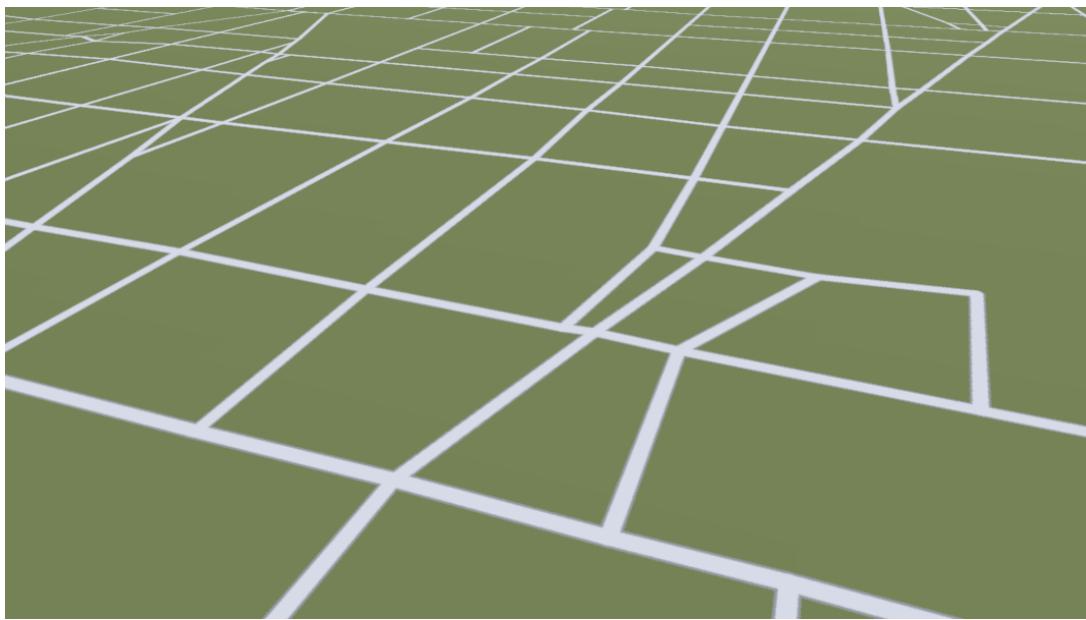


Figure 71 - Painting streets with a world sized width.

22.5.9 Painting shapes in view coordinates

LuciadLightspeed's painting capabilities are honed towards the visualization of shapes at an earth position. Regardless, you can also use vector styling to draw shapes in view coordinates, or in other words, in screen pixels. This can be useful when you want to draw elements that have a fixed position on the screen, rather than a position defined on the earth.

You can achieve this in two ways, depending on your choice of model:

- **Your model does not have a reference.** You enter your shapes in a model with a `null` model reference.
- **Your model has a reference.** Your shapes are in a model with a valid reference. In that case, use a custom `ILspStyler` that defines new geometry through an `ALspStyleTargetProvider`. If you let the style target provider return `null` as target reference in `getStyleTargetReference`, the geometry is interpreted in screen coordinates. This approach allows you to mix geometry on the earth and geometry on the screen in one styler. For more information about style target providers, see [Section 22.9.4](#).

Regarding the interpretation of the screen coordinates, keep in mind that the coordinate origin (0,0) is located in the upper left corner of the map.

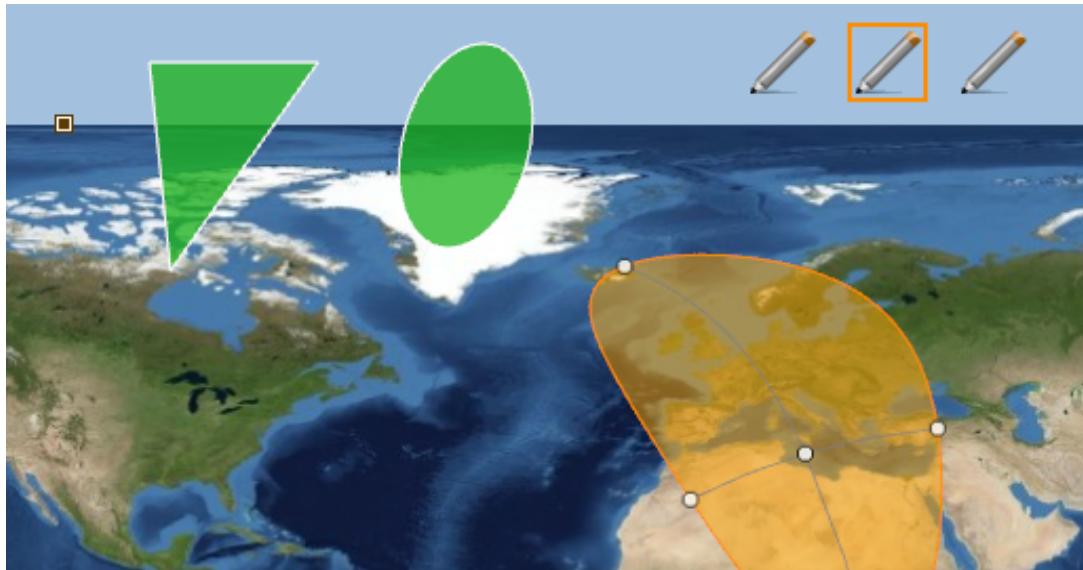


Figure 72 - Painting shapes in view coordinates in the lightspeed.paintinview sample.



If you want to draw interactive Swing components, you do this by adding them to the view's overlay component: see `TLspAWTView.getOverlayComponent()`.

22.6 Styling objects with hardware-accelerated parameterized styles

The LuciadLightspeed API offers parameterized styles. Parameterized styles allow you to base the styling of an object on the values of its properties and style it independently of the other objects in the layer. Parameterized styling is driven by the graphics hardware in your computer. If you compare it with the application of multiple regular styles such as `TLspIconStyle`, parameterized styling results in a significant performance increase. It also allows you to filter the data and change the style applied to the filtered data instantly, without re-processing the data.

The following parameterized styles are available:

- `TLspParameterizedIconStyle`, the parameterized equivalent of `TLspIconStyle`
- `TLspParameterizedLineStyle`, the parameterized equivalent of `TLspLineStyle`
- `TLspParameterizedFillStyle`, the parameterized equivalent of `TLspFillStyle`

See how you can use each of these styles in the `lightspeed.style.expressions` sample.

22.6.1 When to use parameterized styles?

Use parameterized styles if you want to style the objects in your layer based on their properties. Because you use expressions to determine the styling in parameterized styles, you will only need to provide a single style in most use cases. As a result, performance increases significantly, in comparison with the use of multiple regular styles.



For large static point data sets, it is recommended to use a plot layer with a `TLspPlotStyle`, since this layer has additional optimizations for static data. The usage of the plot layer is explained in [Section 22.11](#).

22.6.2 Creating parameterized styles

You can use a dedicated Builder to create each of the parameterized style types. Its use is similar to that of the style builders for other styles such as `TLspIconStyle`. See [Section 22.4](#) for more information.

The style itself is immutable. To change the used style, fire a style change event through `ALspStyler.fireStyleChangeEvent`. The new style will be applied instantly. The creation of expressions is explained in more detail in [Section 22.6.3](#).

22.6.3 Using expressions for styling and filtering your vector data

Expressions are rules that are evaluated by the layer, or more specifically, by the graphics hardware.

To build expressions, you can use various static methods on `TLcdExpressionFactory`. First, you add a static import to your Java class, as shown in [Program 94](#).

```
1 import static com.luciad.util.expression.TLcdExpressionFactory.*;
```

Program 94 - Using a static import for easy access to static factory methods on `TLcdExpressionFactory`

Expressions are opaque `ILcdExpression<T>` objects that can only be evaluated internally by the layer. They do have a generic parameter that indicates the return value of the expression. It helps you with the construction of expressions of the correct type, and with the creation of nested expressions.

[Program 95](#) shows a simple expression that you can use to determine whether an object should be shown. The expression evaluates to `true` if the value of an attribute equals 43.

```
1 ILcdExpression<Boolean> visibility = eq(typeAttribute, constant(43));
```

Program 95 - An expression that evaluates to `true` or `false`

You can nest expressions as long as they evaluate to the same type.



See the class javadoc of `TLcdExpressionFactory` for a list of all possible expressions and more examples.

Using attributes to extract properties from your domain objects

For most business rules, you need access to the properties of your domain object.

[Program 96](#) shows how you can use an attribute based on a `TLcdDataProperty`.

```
1 ILcdExpression<Integer> type = attribute(Integer.class, fDataType.getProperty("type"));
```

Program 96 - Using an attribute to retrieve properties from domain objects

When you are styling lines with an `TLspParameterizedLineStyle`, you may need to provide different attribute values for each point on the line. To do so, implement a `TLcdExpressionFactory.AttributeValueProvider` as shown in [Program 97](#). The index parameter corresponds to the point index on the line. In this program, points at an even index get a green color, the uneven ones a blue color. Between the points, the color of the line will interpolate between green and blue.

```

1 ILcdExpression<Color> color = attribute("ColorAttribute", Color.class, new
2     TLcdExpressionFactory.AttributeValueProvider<Color>() {
3         @Override
4         public Color getValue(Object aDomainObject, Object aGeometry) {
5             throw new RuntimeException("Not used for lines");
6         }
7         @Override
8         public Color getValue(Object aDomainObject, Object aGeometry, int aIndex) {
9             return aIndex % 2 == 0 ? Color.green : Color.blue;
10        }
11    });

```

Program 97 - Attribute value provider for points of a line

Due to GPU hardware restrictions, you cannot store all Java types as attributes. You are limited to:

- `Float`
- `Integer`: stored as floating point numbers
- `Short`, `Byte`: stored as floating point numbers, but take up less video memory
- `Boolean`
- `ILcdPoint`
- `Color`

Strings in particular cannot be used, although you could keep a mapping to numbers, for example, to resolve such an issue.

Using constantly changing values in your expressions

Often, you want to define an expression based on a value that changes many times. For example, you want to visualize only the objects within a time range that is constantly moving, or you want to color objects based on a height threshold backed by a slider UI component.

To deal with changing values, you could use those values as constants, re-build your expressions whenever they change and fire style change events to apply them. Using these values as parameters is often a better solution, though. It allows you to build your expressions only once, and vary the parameters later.

[Program 98](#) shows how to create and use such a parameter. In this example, objects below a varying height threshold are green, and the other objects are red.

```

1 ILcdExpression<Float> heightAttribute = ...
2 ILcdParameter<Float> heightThreshold = parameter("height", 1000.0f);
3 ILcdExpression<Color> colorExpression = ifThenElse(lt(heightAttribute, heightThreshold),
4     constant(Color.green), constant(Color.red));
5 TLspParameterizedLineStyle style = TLspParameterizedLineStyle.newBuilder().color(
6     colorExpression).build();
...
6 heightThreshold.setValue(2000.0f);

```

Program 98 - Using a parameter for constantly changing values in an expression

Your layer automatically picks up any changes. It is not necessary to fire a style change event in your `ILspStyler`, or to invalidate the layer or view.

22.7 Defining raster styles

A raster styling consists of the style of the raster data itself and the style of the data bounds. The data bounds are displayed if the raster data is not suited for the current view, when the data is too detailed for a zoomed-out view for example.



This section discusses the use of `TLspRasterStyle` to style raster data. If you are using the LuciadLightspeed image processing framework to process your raster data as `ALcdImages`, you can use `TLspRasterStyle` in conjunction with `TLspImageProcessingStyle`. `TLspImageProcessingStyle` allows you to apply the image operators of the image processing framework during visualization. `TLspRasterStyle` can serve to adjust brightness, contrast, and opacity of the image processing result. For more information about `TLspImageProcessingStyle`, see [Section 12.4.1](#) in the [Working with images](#) chapter.

22.7.1 Styling raster data with `TLspRasterStyle`

You can use `TLspRasterStyle` to define the style of raster data. The elevation mode of this style is `ON_TERRAIN` by default.

You can change the point at which the raster data or its bounds are visualized by setting the `startResolutionFactor`.

Note that you need to set a `ColorModel` on the style if you want to visualize elevation data as a colored texture on the terrain. Otherwise, the elevation data will only be used to elevate the terrain. For more information about customizing the terrain visualization itself, see [Paragraph 22.10.1](#).

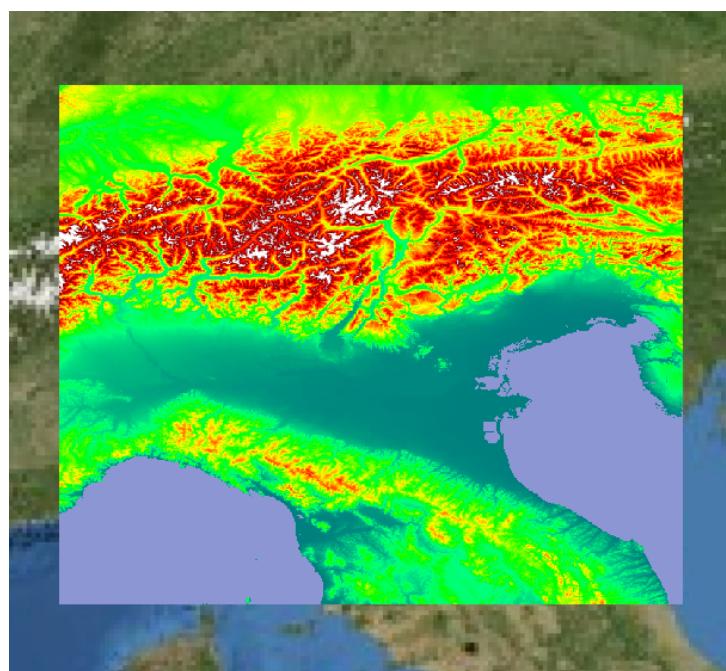


Figure 73 - Elevation raster data styled with `TLspRasterStyle`

22.7.2 Styling the raster data bounds: `TLspLineStyle` and `TLspFillStyle`

You can use `TLspLineStyle` and `TLspFillStyle` to define the appearance of the raster data bounds. Note that you typically need to enable draping on the style explicitly.

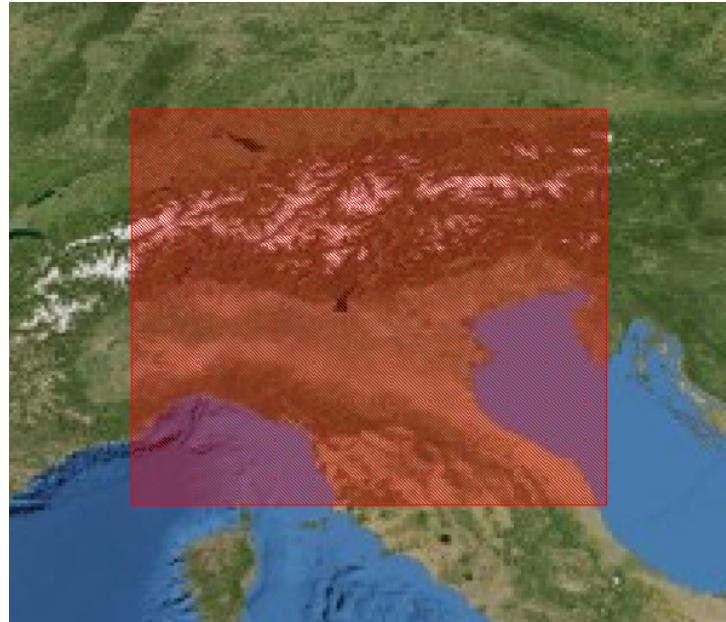


Figure 74 - Raster data bounds styled with `TLspLineStyle` and `TLspFillStyle`

22.7.3 Color manipulation using `TLspColorLookupTableFilterStyle`

You can manipulate the colors of a raster layer by adding a `TLspColorLookupTableFilterStyle` to the layer. This style defines one or more color lookup tables, which are indexed using the red, green and blue components of the pixels in the raster, and which produce a new color as output. The color lookup is performed on the GPU, and therefore has very little performance cost.

Color lookup tables allow for a wide variety of image processing operations, such as:

- Brightness and contrast adjustments
- Replacing one color with another, making all black pixels in a raster transparent for example
- Swapping color channels, converting BGR to RGB for example

For more details, see the reference documentation of the `com.luciad.view.lightspeed.style.filter` package and `TLspRasterLayerBuilder` class.



As mentioned above, you can perform more elaborate image processing, including color lookup operations, by using `TLspImageProcessingStyle`. For more information, see [Chapter 12](#).

22.8 Defining density styles

Visualizing density plots is done by using `ALspDensityStyle` extensions in your styler. Predefined implementations are available for points (`TLspDensityPointStyle`), lines (`TLspDensityLineStyle`) and polygons (`TLspDensityFillStyle`). Controlling the color mapping is done by specifying a `TLspIndexColorModelStyle`.



Mixing density and non-density styles in the same styler is not supported. Your styler should submit only density styles, or only non-density styles.

You use the density style builder to create density styles. Such a builder allows you to customize both the width and the hardness of the density plot:

- Width of points and lines. The density plot can be displayed with pixel-sized density or world-sized density. With pixel-sized density, the width is specified in pixels. A density plot with pixel-sized width adapts when the zoom level changes. With world-sized density, the width is specified in world size, with respect to the world reference. A density plot with world-sized width remains fixed when the zoom level changes.
- Hardness. This parameter controls to what extent the influence of a shape on the density plot decreases with distance. You can specify any value within the range of 0 and 1. With a hardness value of 0, the influence of a shape decreases fast. With a hardness value of 1, the influence of the shape remains constant over the entire width of the line. This results in very blurry plot lines when the hardness is set to 0, while the plot lines are sharp when the hardness is set to 1.

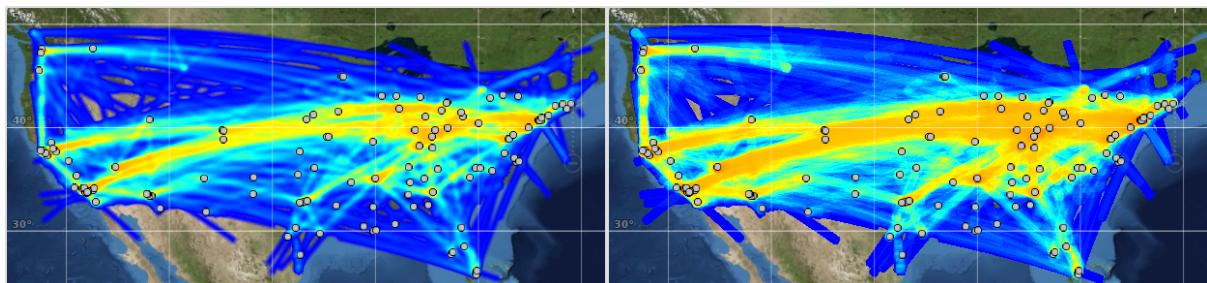


Figure 75 - Density styles with hardness 0.2 (left) and 1.0 (right)

For more information about setting up a density layer, see [Section 22.2.4](#).

22.9 Linking objects to styles with `ILspStyler`

Once the style objects have been defined, you need to specify to which objects they apply. In the simplest case, you use the same style, independent of the object. In any other case, you will want to base style selection on intrinsic properties of the object. The base interface that allows you to do all of this is `ILspStyler`.

Objects are styled through a **callback** mechanism: a painter object, for instance, needs information about the styles associated with a set of objects it needs to paint, and calls `ILspStyler.style(...)`. The set of objects are passed to the styler as arguments, along with a `ALspStyleCollector`. The `ILspStyler` implementation determines which styles go with the supplied object set, or a subset of that object set. The styler then uses the methods of the `ALspStyleCollector` to apply the styles to the appropriate objects.

The styling API allows you to specify to which geometry styling needs to be applied. This can be a geometry that is entirely different from the original object's geometry. The `ALspStyleCollector.geometry(...)` method is the most direct way to specify a different geometry in the form of an `ILcdShape` object, and does not require you to write a custom class. This can be quite convenient in a number of cases. It may not be the best choice in terms of optimization, however, mainly because it forces the implementation to keep track of the shape, for each object.

A better way is to use `ALspStyleTargetProvider` implementations that can derive the shape from the object, and pass such an implementation to the `ALspStyleCollector.geometry(...)` method. This ensures that the shape is only created when it is needed, and avoids the bookkeeping overhead of associating one or more shapes with each object.

Section 22.9.4 discusses the use of `ALspStyleTargetProvider` in detail.

22.9.1 Creating a basic `ALspStyler`

Whenever your data does not allow you to apply the same style to all objects in a layer, you probably need to write a custom `ALspStyler` implementation. To get you started, it is recommended that you have a look at the available samples and documentation.

Implementing a basic `ALspStyler` is quite straightforward, but there are a couple of points that require more attention. Here is an example that shows what such an implementation can look like:

```

1 public class MyStyler extends ALspStyler {
2
3     private ALspStyle style1;
4     private ALspStyle defaultStyle;
5
6     @Override
7     public void style( Collection<?> aObjects, ALspStyleCollector aStyleCollector, TLspContext
8         aContext ) {
9         for ( Object object : aObjects ) {
10             if ( hasStyle1( object ) ) {
11                 aStyleCollector.object( object ).style( style1 ).submit();
12             } else {
13                 aStyleCollector.object( object ).style( defaultStyle ).submit();
14             }
15         }
16     }
17     private boolean hasStyle1( Object aObject ) {
18         ...
19     }
20 }
```

Program 99 - A simple styler

Quite often, this type of implementation works just fine, but consider the following questions:

Is my implementation thread-safe? The styler can be used from any thread, so the `ILspStyler` implementation should implement the necessary safeguards against any potential threading issue.

Is my implementation efficient enough? Depending on the nature of the data, the number of times your styler is called may vary. For instance, if you have a model containing 100000 elements that are updated every 100 milliseconds, you may want to think about avoiding an

iteration over all objects, to maximize throughput. For more tips to improve the performance of your stylers, see [Section 22.9.5](#).

22.9.2 Choosing `ILspStyler` implementations

Beside the abstract base class, LuciadLightspeed also offers a number of concrete implementations that cover common use cases. These help you get started, and provide optimal implementations wherever possible. This section discusses most of the available implementations.

Your choice of `ILspStyler` type depends on the following criteria:

- Do I want to use styling to differentiate between objects, or do all objects require the same style?
- Does the object styling remain the same, or can the application users change the styling properties?
- Does the geometry I want to style my object with match the original geometry of the object? For more information, see [Section 22.9.4](#).



Even though almost any styler can be written as a hierarchy of these basic implementations, such an approach may not be desirable. To increase performance, you should consider limiting the total number of calls to `ALspStyleCollector`. This is hard to achieve if you are delegating the styling of one object to a large number of separate stylers. The right implementation depends a lot on the type and the amount of your data, so the general rule of implementing functionality first, and optimizing for performance if necessary also applies here.

Using one style for all objects: `ALspStyle`

If only one style needs to be applied to all objects, simply use the style itself. It implements `ILspStyler` directly. Its implementation of `ILspStyler` just returns one style for all objects.

Defining multiple styles for all objects: `TLspStyler`

`TLspStyler` is a convenience implementation that allows you to specify a set of styles. You can define multiple styles for each object, a line and a fill style for polygons for instance. A `TLspStyler` object passes all of these styles directly, regardless of the object it is called with. This is very efficient, because it does not require any iterations over objects.

It also supports the use of `ALspStyleTargetProvider`, which allows basic filtering on objects. As a result, you can already achieve quite complex styles using only this class, a few basic styles, and an `ALspStyleTargetProvider`.

Using mutable stylers: `ILspCustomizableStyler`

The basic `ILspStyler` interface provides no control over the styles that are being used. The styles are not exposed, so you cannot simply change the styles. This contrasts with the requirements of many applications to allow users to change styling properties. To meet these requirements, LuciadLightspeed offers the `ILspCustomizableStyler` interface, which can be implemented by objects that allow style changes. It ensures that these components change styles in the same way.

Implementations of `ILspCustomizableStyler` expose their styles as a modifiable list of `TLspCustomizableStyle` objects. You can modify the style objects in the list to adapt, enable or disable certain styles. To remove or add styles, you can modify the list itself.

A `TLspCustomizableStyle` is a wrapper around a plain `ALspStyle`. It adds some additional properties that allow for easy identification and integration in a user interface. Contrary to the regular `ALspStyle` object it is wrapping, a `TLspCustomizableStyle` object is a mutable object. Whenever the wrapped style object is replaced by a different style object, the `TLspCustomizableStyle` object fires the necessary events. When a style collector requests the styles it needs to apply, the wrapped immutable `ALspStyle` object is sent to the collector.

TLspCustomizableStyler A default implementation of `ILspCustomizableStyler` is provided by `TLspCustomizableStyler`. Its behavior is very similar to that of `TLspStyler`, but it adds the advantages provided by `ILspCustomizableStyler`. The most important additional feature is the ability to activate and deactivate styles. You can use this to toggle a fill or line style for a given layer, for example. For more information about `TLspStyler`, see [Section 22.9.2](#).

Changing styles per object: `TLspEditableStyler`

If you want the style of individual objects on the map to be defined by the end user of your application, you can use `TLspEditableStyler`. It contains a default set of styles for all objects, and also keeps a mapping from objects to more specific styles.



Do not use this class if your style can be derived from the properties of the object. It is mostly suited if the end user can randomly assign styles to individual objects.

Note that this class does not support the changing of a style for a given category of objects. For that, you should have a look at `ILspCustomizableStyler` which is discussed in [Section 22.9.2](#).

Toggling stylers

LuciadLightspeed provides concrete and abstract `ILspStyler` classes that support the delegation of objects to one of two stylers.

`ALspToggleStyler` splits objects in two sets and applies separate stylers to each set.

`TLspDrapingToggleStyler` is a simple styler toggler that distinguishes between drapable objects and non-drapable objects.

22.9.3 Dynamically applying styles

LuciadLightspeed fully supports, and is optimized for styles that change arbitrarily. This opens up a wide range of possibilities, such as animations, scale ranges and highlighting. To get an idea of what can be achieved with these dynamic effects, run the corresponding samples.

The most important thing to remember when implementing a styler that changes its styles, is to fire a style-changed event whenever your style changes. This is the only way to ensure that your style change is taken into account. For optimal performance, the event should also include the objects that are affected by the style change, to prevent another query to the style to retrieve the affected objects.

The only exception to this rule are style changes due to object changes: if a model-changed event is fired, the styles of the affected objects are queried again automatically. The styler does not need to fire events in this case.

Animated styles

The performance offered by LuciadLightspeed allows you to implement animations by gradually changing the properties of a style. You can for instance change the opacity of the styles for a given object from 0.0 to 1.0 to achieve a fade-in effect.

Have a look at the `lightspeed.customization.style.animated` sample for a possible implementation.



To achieve a smooth animation, your styler implementation needs to be called quite often. Hence, the performance and thread safety of your implementation is very important in the case of animations. A typical styler that is suited for animation is shown in [Program 104](#).

22.9.4 Deriving geometry from objects

Sometimes you do not want to apply styles to an object as it occurs in your `ILcdModel`. Situations where you may want to apply a different styling are:

- The object consists of separate parts, and you want to apply a distinct styling to each part.
- The object does not directly implement an `ILcdShape`, but contains one. This case is known as a “has-a-shape” situation. You want to style the object so that it uses that shape.
- You want to paint a geometry that does not correspond to the geometry of the domain object. You could create and display pie charts for country objects, for instance.

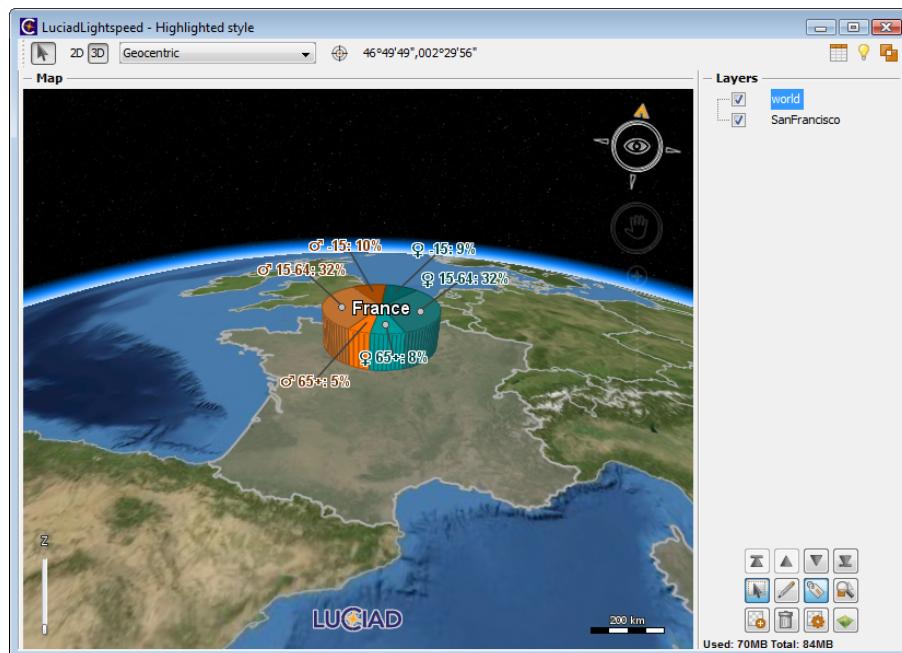


Figure 76 - Displaying pie charts for country objects with the `lightspeed.customization.style.highlighting` sample.

For these types of cases, you can implement `ALspStyleTargetProvider`. Use the side-effect method `getStyleTargetsSFCT()` to derive geometries for a given object. These geometries are usually `ILcdShape` instances that can be used together with the associated styles. Program 100 shows a simple implementation that replaces a specific type of object with its contained

shape.

```

1 ALspStyleTargetProvider customTargetProvider = new ALspStyleTargetProvider() {
2     @Override
3     public void getStyleTargetsSFCT( Object o, TLspContext aContext, List<Object> aObjects ) {
4         if ( o instanceof CustomShape ) {
5             aObjects.add( ( ( CustomShape ) o ).fShape );
6         }
7         else {
8             aObjects.add( o );
9         }
10    }
11 };
12 TLspStyler styler = new TLspStyler();
13 styler.addStyles( customTargetProvider, Collections.singletonList( lineStyle ) );

```

Program 100 - A simple ALspStyleTargetProvider for “has-a-shape” domain objects.

This class can also be used as a very basic filter: if you do not return any geometry for the given object, the object will not be visible. You cannot implement complex rules with this class, however.

Implementers of `ALspStyleTargetProvider` must make sure that the `equals()` and `hashCode()` methods are correctly implemented. If the geometry that is to be used changes, a different `ALspStyleTargetProvider` instance that is not equal to the previous one should be provided. Of course, if the object itself changes, and a proper model-changed event is thrown, the `ALspStyleTargetProvider` will be queried again to obtain an updated geometry.

Ensuring correct selection and culling results

By default, the shape layer only retrieves those model objects that are visible within the view, and leaves out the others. This culling activity occurs during object painting and during user interactions such as selection. It is key to maintaining efficiency when you are working with large or slow models like databases. During the culling process, the model bounds of objects implementing `ILcdBounded` are used to determine whether an object should be considered or not.

However, it is possible that you are using a styler to produce geometry that does not necessarily correspond to these bounds. You paint a circle with a radius of 1km around a point object, for example. The point object has size 0, while the circle covers a region with a 2km size. By default, the circle will not be painted if the point lies outside of the view. Users cannot select it either.

To prevent this type of situation, set a culling margin in pixels or meters when you are building your shape layer. You can use the `objectViewMargin` or `objectWorldMargin` for this purpose. The shape layer extends the considered view region with these margins when it is retrieving domain objects. In the example of the circle, a world margin of 1km results in the desired behavior: the circle is still painted.

The view margin is particularly useful when you are painting icons for point objects. Whereas the point object has a size of 0, the icon can have a size of 64 by 64 pixels for example. A view margin of 32 pixels ensures that an icon is painted even if the center point falls just outside of the view.

Using an `ALspStyleTargetProvider` in your `ILspStyler`

The `ALspStyleCollector` allows you to specify the `ALspStyleTargetProvider` that is to be used in combination with a number of styles.

```

1 aStyleCollector.object( myObject )
2     .geometry( myFillStyleTargetProvider )

```

```

3 .style(myFillStyle)
4 .submit();
5 aStyleCollector.object(myObject)
6 .geometry(myLineStyleTargetProvider)
7 .style(myLineStyle)
8 .submit();

```

Program 101 - How to specify a `ALspStyleTargetProvider`

As you can see in Program 101, it is possible to specify multiple style target providers for the same object. Note that `myFillStyleTargetProvider` and `myLineStyleTargetProvider` may be compared to each other, and considered the same if they are equal.

Example: Styling the points in a list with different icons

This example shows how you can implement a style target provider to apply different icon styles to points in a list, depending on the position of the points in the point list. As you can see, this `ALspStyleTargetProvider` implementation has an `fIndex` field that indicates which point needs to be retrieved from the incoming object. If such a point can be found, it is added to the result.

```

1  @Override
2  public void getStyleTargetsSFCT(Object aObject, TLspContext aContext, List<Object>
3  	aResultSFCT) {
4  	final EnrouteAirwayTrack track = (EnrouteAirwayTrack) aObject;
5  	if (track.isAirborne()) {
6  		final ILcdPoint point = track.getTrackHistory().safeGetPoint(fIndex);
7  		if (point != null) {
8  			aResultSFCT.add(point);
9 		}
10  }

```

Program 102 - Specifying a specific geometry.

(from samples/lightspeed/common/tracks/TrackHistoryPointProvider)

Example: Using styling to convert a point to a circle

Suppose that you want to draw a circle with a radius of 10km around your current position on the map. You can easily achieve this with a simple line style and a style target provider.

```

1 public void getStyleTargetsSFCT( Object aObject, TLspContext aContext, List<Object>
2  	aResultSFCT ) {
3  	ILcdEllipsoid ellipsoid = ((ILcdGeoReference)aContext.getModelReference()).getGeodeticDatum
4  	().getEllipsoid();
5  	aResultSFCT.add( new TLcdLonLatCircle( (ILcdPoint) aObject, 10000, ellipsoid ) );
6 }

```

Program 103 - Converting geometry

22.9.5 Improving the performance of your stylers

This section provides recommendations for the improvement of your stylers' performance.

Maximizing object throughput

In the example in Section 22.9.1, an iteration over all objects was used to specify the styling. While this is a perfectly usable implementation that may be sufficient in a lot of cases, there is still room for improvement.

Suppose that all of your objects share the same styles, but one specific object requires a special style. The following sample shows how you can apply that style efficiently. In the sample, a collection of objects possibly contains animated objects, which require a styling that is different from the styling of the regular, unanimated objects.

```

1  @Override
2  public void style(Collection<?> aObjects, ALspStyleCollector aStyleCollector, TLspContext
3      aContext) {
4      //use contains to determine if the animated object is part of the collection in the most
5      //efficient way
6      if (aObjects.contains(fAnimatedObject)) {
7          if (aObjects.size() == 1) {
8              aStyleCollector.object(fAnimatedObject).styles(fAnimatedStyle).submit();
9          } else {
10             final Collection<Object> objectsWithoutAnimated;
11             if (aObjects instanceof ILcdCloneable) {
12                 //some LuciadLightspeed collection implementations such as TLcdIdentityHashSet
13                 //can be cloned to allow efficient copying
14                 objectsWithoutAnimated = (Collection<Object>) ((ILcdCloneable) aObjects).clone();
15             } else {
16                 objectsWithoutAnimated = new TLcdIdentityHashSet<Object>((Collection<Object>)
17                     aObjects);
18             }
19             // remove the animated object
20             objectsWithoutAnimated.remove(fAnimatedObject);
21             aStyleCollector.object(fAnimatedObject).styles(fAnimatedStyle).submit();
22             aStyleCollector.objects(objectsWithoutAnimated).styles(fDefaultStyle).submit();
23         }
24     }

```

Program 104 - More efficient styler implementation.

(from

`samples/lightspeed/customization/style/animated/AnimatedAreaStyler`)

This program takes only a few more lines to write, but is much more efficient: we only submit special styles if the list of objects to be styled contains the `fAnimatedObject`.

In this sample, it is assumed that the collection of objects that is passed into the styler supports efficient lookups. This is a safe assumption if the styler is used in the default LuciadLightspeed layers and painters. Note as well the use of `ILcdCloneable`, which allows for efficient cloning. This is also something that is safe to assume in LuciadLightspeed and can result in performance improvements.

Minimizing the number of `submit()` calls

Another common styling performance issue can occur if you use multiple `submit()` calls per object to specify styles, although you can specify the same styling with less calls. The basic idea behind styling objects in batch is that only the person who defines the styling, knows best how to match objects to styles. If the styler implementation does not take advantage of this knowledge, the painter will get to the same result eventually, but might need more time to achieve the result.

A first opportunity for optimization presents itself when your objects can be divided in categories: you have four types of objects and want to associate a style with each one of them, for instance. You can loop over all objects, and submit the appropriate styles per object. This works just fine, but it may be a better idea to split the objects into four groups first, and then submit the styles per group. Again, the performance impact of this mostly depends on how many times the styler needs to be used.

A second example of optimization consists of specifying two separate geometries for a given object. The geometries need to be styled with the same style. A basic implementation could use `ALspStyleCollector.geometry(myShape)` in two separate `submit()` calls with the same style. In a better implementation, you could either write an `ALspStyleTargetProvider` implementation that returns both geometries at once, or you could use a `TLcdShapeList` that contains both geometries as an argument for the `geometry(...)` call, so that you only need to call `submit()` once.

22.9.6 Layer styles

Lightspeed layers support a number of graphical effects which are applied as a whole contrary to `ILspStyler` or `ALspStyle` that perform styling per object. An example of such effects is the ability to modify the opacity of the complete layer. These effects are represented by the class `TLspLayerStyle`.

See the javadoc on `TLspLayerStyle` for more information.

22.10 Visualizing 3D data in a Lightspeed view

This section discusses how the elevation of layers and layer objects is visualized in a Lightspeed view.

LuciadLightspeed handles two types of elevation:

- Terrain elevation: the visualization of rasters and tilesets with elevation information as background data.
- 3D object altitude: the visualization of 3D objects, which contain altitude information, on the terrain.

22.10.1 Visualizing terrain elevation

In each view, a global terrain layer is constructed automatically. A **global terrain** displays the elevation of the Earth surface in 3D. Raster layers and Earth tileset layers typically serve to construct a global terrain. Both rasters and tilesets may contain elevation data or not. The presence of elevation data determines how a layer is used to build a global terrain. In Lightspeed views, `ILspTerrainSupport` takes care of terrain construction. You can obtain it from the view by calling `ILspView.getServices().getTerrainSupport()`.

`ILspTerrainSupport` automatically picks up and uses the elevation data from raster layers and Earth tileset layers. To determine if a raster layer contains elevation data, it checks whether `TLcd(Multilevel)RasterModelDescriptor.isElevation()` returns true. To determine if an `ILcdEarthTileSet` consists of elevation data, it checks whether `CoverageType.ELEVATION` is used to identify the tileset's data type.

If multiple layers with elevation data are available, and if they overlap in certain areas, only the elevation of the topmost layer is used. Which layer is the top layer is determined by the ordering of the layers. For more information about layer ordering, see [Chapter 6](#). This means that you can take control of the layer order to display coarse elevation data on a worldwide scale, and move to a layer with more detailed data when a user zooms in on a smaller area.



To obtain the best performance, use a single Earth tileset with elevation data.

Data without 3D coordinates is draped over the constructed terrain in the order of the layers containing the data. **Draping** means that a layer is spread across the terrain in such a manner that the draped layer is positioned directly on top of the terrain's surface relief. This applies to vector as well as raster data. Both background raster layers and a vector layer displaying rivers can be painted on top of the available terrain, for example. See [Section 22.10.2](#) for more information about displaying objects on top of a terrain.



In both 2D and 3D, you can also use the global terrain as input for analysis operations such as line-of-sight calculations.

Customizing the terrain visualization You can change terrain styling such as the terrain's brightness, contrast or opacity with the help of `ILspTerrainSupport`. The method `setBackgroundStyler()` lets you set a styler for the terrain.

For more information about working with `ILspTerrainSupport`, see [Chapter 27](#).

22.10.2 Visualizing objects in 3D

Objects may or may not contain altitude or elevation information. In 3D mode, the global terrain is used for the visualization of objects: if an object does not contain information about its altitude, it is draped over 3D terrain as a 2D object. If it does have altitude information, it is displayed as a 3D object.

How objects are painted in the view depends on the `ALspStyle` you use to paint the object. All Lightspeed styles have an elevation mode. It can be set to the following options:

ON_TERRAIN The data is draped on the terrain. Any altitude information stored with the objects is ignored. This is a useful elevation mode for painting and styling street data, for example.

ABOVE_TERRAIN The Z-coordinates in the data are used to set the altitude of an object above the terrain. This mode can be used for painting and styling icons.

ABOVE_ELLIPSOID The terrain elevation is not taken into account. Only the altitude stored with the objects is used to paint the objects: the Z-coordinates in the data are used to set an object's altitude above the reference point. You could select this style to paint flight plans, for example, as these objects typically contain such altitude information.

OBJECT_DEPENDENT The most appropriate mode is chosen based on the object, and the style that is applied to it. This elevation mode is especially convenient when you are writing generic stylers for undetermined data input formats.

In 2D mode, Z-coordinates are always ignored when layers are visualized.

22.11 Hardware-accelerated plot painting

LuciadLightspeed includes a plots package that allows you to display a large number of points as plots. You can use it to plot hundreds of thousands of aircraft positions on a map, for instance.

The `com.luciad.view.lightspeed.layer.plots` package applies many of the visualization concepts explained in this chapter. In combination with the styling options of `TLspPlotStyle` in the `com.luciad.view.lightspeed.style` package, it offers:

- A Lightspeed layer that is targeted specifically at large static datasets containing points, such as plot data.
- Extensive configuration options for the styling of plots
- Plot styling through expressions, which are rapidly evaluated by your computer's graphics hardware

The Lightspeed plot layer makes use of graphics hardware to expand its visualization capabilities up to millions of points. It also allows you to filter the data and change the style applied to the filtered data instantly, without re-processing the data.

This layer can handle any `ILcdModel` that contains point data. However, the data should not change once it is loaded. In the case of model change events in particular, all the data is re-processed for visualization.

22.11.1 Configuring a plot layer

To create a Lightspeed plot layer, use `TLspPlotLayerBuilder`, as shown in [Program 105](#).

```
1 ILspLayer layer = TLspPlotLayerBuilder.newBuilder().model(myModel).build();
```

Program 105 - Creating a layer for plot objects using `TLspPlotLayerBuilder`

The `TLspPlotLayerBuilder` has a few optional configuration methods, similar to the options of other layer builders such as `TLspShapeLayerBuilder`:

- `label` and `icon`: to set the label and icon to use in a layer tree GUI.
- `bodyStyles` and `bodyStyler`: to configure styling options. For more details on styling, see [Section 22.11.2](#).
- `bodyScaleRange`: to configure at what scale range the data should be displayed. For more details on performance, see [Section 22.11.3](#).
- `labelStyles`, `labelStyler` and `labelScaleRange`: to configure labels for plot layers. For more details on labels, see [Section 22.11.2](#).

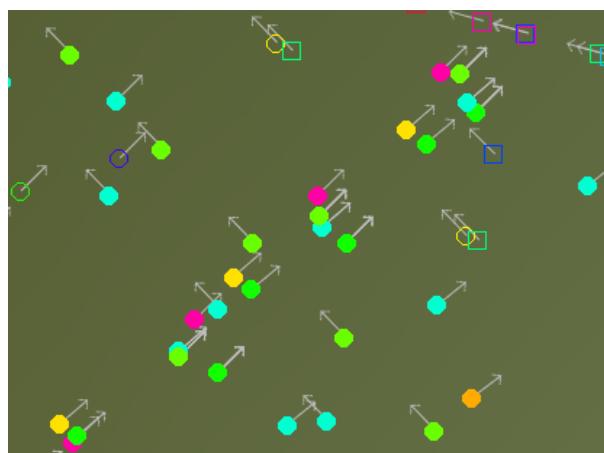


Figure 77 - The sample `lightspeed.plots` shows plots using a `TLspPlotLayerBuilder`.

22.11.2 Styling and filtering your plot domain objects

By default, the plots are visualized as white dots, but you can fully customize the styling of the plots.

Applying style to a plot layer is similar to styling in other layers: you can use an `ILspStyler` or set a fixed style. For more information about using a styler to apply styles, see [Section 22.9](#).

Instead of calculating the style for each object individually in your styler, though, you must use one shared style for all objects, `TLspPlotStyle`, as shown in [Program 106](#).

```

1 public class MyStyler extends ALspStyler {
2     public void style(Collection<?> aObjects, ALspStyleCollector aStyleCollector, TLspContext
3                         aContext) {
4         TLspPlotStyle style = TLspPlotStyle.newBuilder().build();
5         aStyleCollector.objects(aObjects).style(style).submit();
6     }

```

Program 106 - Using `ILspStyler` for plot layers that uses a single shared `TLspPlotStyle`

You can configure the custom plot style with expressions that calculate the styling options. The layer will use graphics hardware to evaluate these expressions efficiently. See the following sections for examples. See [Section 22.6.3](#) for more details about using expressions.

Creating a plot style

You can create a `TLspPlotStyle` using its `Builder`, which works in a similar way as the style builder for `TLspIconStyle`. You can assign an expression to each style aspect of the icons in your plot:

- icon
- opacity
- modulationColor
- scale
- visibility

The style itself is immutable. To change the used style, fire a style change event through `ALspStyler.fireStyleChangeEvent`. The new style will be applied instantly. For performance details, see [Section 22.11.3](#).

The creation of expressions is explained in more detail in [Section 22.6.3](#). The rest of this section focuses on the various styling options, and already shows a few examples of expressions.

Choosing an icon for your plot objects

The `icon` aspect of your `TLspPlotStyle` lets you control what icon is displayed. There is no performance penalty for the use of many different icons, even if there are several thousands.

For example, [Program 107](#) shows a style that assigns a different icon depending on an attribute of the domain objects.

```

1 TLspPlotStyle.newBuilder().icon(ifThenElse(eq(inFlightAttribute, constant(43)), constant(
    inFlightIcon), constant(onGroundIcon)).build());

```

Program 107 - Using an expression to choose icons



Figure 78 - Using an expression to choose different icons for domain objects.

To learn more about the use of icons with domain objects other than plots, see [Section 22.5.5](#).

Applying transparency to your plot icons

The `opacity` aspect of your `TLspPlotStyle` allows you to make the plot icons more transparent. You can use this to fade out less important objects, for instance.

[Program 108](#) shows a style that fades out objects based on a time interval.

```
1 TLspPlotStyle.newBuilder().opacity(fraction(timeAttribute, minTimeParameter, maxTimeParameter))
    .build();
```

Program 108 - Using an expression to apply transparency to icons



Figure 79 - Using an expression to apply transparency.

Applying a modulation color to your plot icons

With the `modulationColor` aspect, you can adapt the color of the icons. You can use this to vary icon color based on an attribute, for example.

The color is mixed with the icon. This usually works best if the icons themselves are in grayscale.

For example, [Program 109](#) shows a style that changes the color from red to green based on a time interval.

```
1 TLspPlotStyle.newBuilder().modulationColor(mix(Color.red, Color.green, fraction(timeAttribute,
    minTimeParameter, maxTimeParameter))).build();
```

Program 109 - Using an expression to apply a modulation color to icons



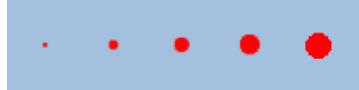
Figure 80 - Using an expression to apply modulation color.

Scaling your plot icons

With the `scale` aspect, you can adapt the size of the icons. Use this to make less important objects smaller, for example.

[Program 110](#) shows a style that shrinks objects based on a time interval.

```
1 TLspPlotStyle.newBuilder().scale(fraction(timeAttribute, minTimeParameter, maxTimeParameter)).
    build();
```

Program 110 - Using an expression to scale icons**Figure 81** - Using an expression to scale icons.

`TLspPlotStyle` also offers an automatic scaling mode. In this mode, the objects are scaled based on their distance to the camera. This means that when you have zoomed out the map, the icons will be smaller. You will also have more objects on the screen. This has two benefits:

- An uncluttered view: if the icons remained large, the view would be cluttered with many large icons.
- Performance: the visualization performance improves. See Section 22.11.3 for more details on performance.

See the javadoc on `automaticScaling` for more information.

Filtering your plot objects

The `visibility` aspect lets you filter out specific objects. You can use this to show only relevant objects, for instance, based on the value of an object attribute.

Program 111 shows a style that hides objects of a certain type, namely those with attribute value 43.

```
1 TLspPlotStyle.newBuilder().visibility(neq(typeAttribute, constant(43)).build();
```

Program 111 - Using an expression to filter icons

Note that object filtering has a positive impact on visualization performance. See Section 22.11.3 for more details on performance.

Using oriented plot icons

You can activate icon orientation with the `useOrientation` flag. As a result, the icons will be rotated towards their orientations.

For icon orientation to work, your domain objects must implement `ILcdOriented`. You can also use an `ALspStyleTargetProvider`, about which you can find more information in Section 22.11.2.

**Figure 82** - Using oriented icons.**Adding labels to your plot domain objects**

You can enable labels in the same way as you would enable them with a regular `TLspShapeLayerBuilder`. Use `TLspPlotLayerBuilder.labelStyles` to configure labels.

You can use all the label styling and decluttering options available in LuciadLightspeed. See Chapter 23 for more information about labeling in Lightspeed views, or the API reference documentation for `TLspLabelStyler`.



Painting plot labels is more CPU-intensive than painting the plots themselves. Therefore, we recommend using a scale range so that labels are only displayed for a subset of the data. See [Section 22.11.3](#) for more performance tips.

Visualizing plot densities

You can visualize plot densities with the `density` flag. If you want to enable density, you must make sure that `icon` is not set. Activating density will also override any modifications to `modulationColor`. [Figure 83](#) shows an example of what a density plot looks like.

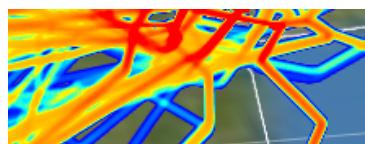


Figure 83 - An example of plots visualized as densities. Blue symbolizes low plot density, red symbolizes high plot density.

You can modify the parameters of the density calculation with the `density` method:

- `IndexColorMap`: A map that translates densities to colors. By default, low densities are blue and high densities are red.
- `Hardness`: A double between the values of 0 and 1 that indicates how hard the density plots should be. A value of 0 indicates a soft, smooth plot density sampling, while a value of 1 indicates a sharp, hard density sampling. The default for this value is 0.5.

Extracting geometry from your plot objects

By default, the plot layer works with domain objects that are `ILcdPoint` objects, and optionally `ILcdOriented` objects.

If you want to work with domain objects that are not `ILcdPoint` objects themselves, you can use an `ALspStyleTargetProvider` to extract the geometry from your domain objects. For oriented icons, the returned geometry must also implement `ILcdOriented`. [Program 112](#) shows you how to use an `ALspStyleTargetProvider`.

```

1  public void style(Collection<?> aObjects, ALspStyleCollector aStyleCollector, TLspContext
2      aContext) {
3      aStyleCollector.objects(aObjects).geometry(fStyleTargetProvider).style(fStyle).submit();
4  }

```

Program 112 - Using an `ALspStyleTargetProvider` to extract geometry from domain objects

See [Section 22.9.4](#) for more information about `ALspStyleTargetProvider`.

22.11.3 Optimizing a plot layer's performance

In the majority of cases, the plot layer will perform well with the default configuration and most of the styling techniques.

This section provides some insight into ways to get the best performance for your particular case.

Choosing an appropriate model implementation

`TLspPlotLayerBuilder` works with any `ILcdModel` implementation, including implementations with lazy-loaded data.

However, there is an important distinction between models, depending on the model type:

- If the model implements `ILcd2DBoundsIndexedModel`, the data is processed incrementally, based on what is visible. It is still cached and processed in bulk for optimal performance, though.
- If the model does not implement `ILcd2DBoundsIndexedModel`, all data is processed once, up front, and never discarded.

Pre-loading attributes

Attributes are processed when they are used by any of the expressions in your `TLspPlotStyle`. If you switch to a different style that uses different attributes, these new attributes have to be processed at the time of the switch. This will cause a noticeable performance hit.

To prevent such a pause in performance, you can set all necessary attributes up front with `TLspPlotLayerBuilder.mandatoryAttributes`.

Similarly, if you switch between styles that have orientation on or off, you can force the load of orientation information with `TLspPlotLayerBuilder.mandatoryOrientation`.

Optimizing visualization by filtering

By setting a visibility rule that filters out elements on `TLspPlotStyle`, you can significantly improve performance. You could also create such a filter based on the view scale, to provide a level-of-detail.

Despite these filtering measures, it may still be necessary to set a hard scale range on the layer. Outside this scale range, the data will not be displayed at all, and does not even have to be loaded.

Optimizing visualization by reducing icon size

Icon size is an important factor in visualization performance, particularly when many objects are visible, and are therefore more likely to overlap on the screen.

Hence, it is recommended to reduce the size of the icons based on scale or distance to camera, as explained in [Section 22.11.2](#).

Optimizing visualization by setting a scale range for plots or labels

Some very large datasets simply cannot be visualized because they do not fit in memory. In such a case, use a scale range so that only a subset that fits in memory is visualized.

The display of labels is more demanding than the display of plots. Use a scale range so that only a few thousands of objects are labeled.

22.12 Advanced visualization of domain objects

This section provides an introduction to the more complex visualization of domain objects. It provides you with some more background information about how LuciadLightspeed uses painters and discretizers to visually represent domain objects in a view.

22.12.1 Domain objects visualization in an `ILspPaintableLayer`

`ILspPaintableLayer` uses a set of one or more `ILspPainter` instances to render visual representations of domain objects into the view. In the remainder of this chapter, it is assumed that the default implementation `TLspLayer` is used. Painters can be added to a `TLspLayer` using the `setPainter(TLspPaintRepresentation, ILspPainter)` method.

`TLspPaintRepresentation` is an enumeration class that defines various visual representations that a domain object can have. By default, `LuciadLightspeed` defines the following representations:

- `BODY` identifies the main geometric representation of the object. Body painters typically render a discretized version of an `ILcdShape`, icons at the location of an `ILcdPoint` or images obtained from an `ILcdRaster`.
- `LABEL` defines a textual representation of the object. Labels are typically drawn on or near the body of the corresponding object. As an example, street data may have bodies consisting of lines, and labels indicating the street names.
- `HANDLE` identifies the representation of the edit handles of the object that is being edited.

A `TLspPaintRepresentationState` is a combination of a `TLspPaintRepresentation` and an `TLspPaintState`. The latter has three possible values: `REGULAR`, `SELECTED`, and `EDITED` which results in the default `TLspPaintRepresentationState` instances:

- `REGULAR_BODY` corresponds to `BODY`, but the corresponding painter is only invoked for geometry objects that are not selected, nor being edited.
- `REGULAR_LABEL` corresponds to `LABEL`, but the corresponding painter is only invoked for label objects that are not selected, nor being edited.
- `REGULAR_HANDLE` corresponds to `HANDLE`, but the corresponding painter is only invoked for handle objects that are not selected, nor being edited.
- `SELECTED_BODY` corresponds to `BODY`, but the corresponding painter is only invoked for objects which are currently selected.
- `SELECTED_LABEL` corresponds to `LABEL`, but the corresponding painter is only invoked for objects which are currently selected.
- `EDITED_BODY` corresponds to `BODY`, but the corresponding painter is only invoked for objects which are currently being edited.
- `EDITED_LABEL` corresponds to `LABEL`, but the corresponding painter is only invoked for objects which are currently being edited.



A painter for a given `TLspPaintRepresentation` paints all the corresponding `TLspPaintRepresentationState` instances. It will therefore be used for regular, selected and edited objects.

When an `ILspView` is repainting itself, it asks its layers to list all their available paint representations (that is the ones for which the layers have a painter). The view then determines the order in which all these layers and paint representations are to be drawn. Labels, for instance, are typically drawn on top of bodies. For more information about painting order, see [Section 21.5](#).

Next, the view asks the layers to render their paint representations one by one. The layers do this by determining which objects in the model are visible in the current view, and then invoking

the appropriate `ILspPainter` to paint those objects.

22.12.2 How does LuciadLightspeed display shapes in Lightspeed layers?

Layers created through the `TLspShapeLayerBuilder` use a `TLspShapePainter` to display shapes. In the majority of cases, you do not have to interact with this shape painter directly to display the shape's geometry. Instead, you can use `ALspStyleTargetProvider` to style the shape with a geometry. For more information, see [Section 22.9.4](#).

`TLspShapePainter` converts the vector-based objects in your data model into a format that Lightspeed layers can display. The shape painter uses an `ILspShapeDiscretizer` for this, which discretizes the geometry of domain objects into a representation consisting of meshes. Meshes describe shapes with geometric primitives, such as lines and triangles. The result of the LuciadLightspeed discretization is a 3D mesh object.

If the shape painter cannot determine the object's geometry from an associated `ALspStyleTargetProvider`, it passes the object to the shape discretizer directly. The discretizer determines the geometry of the shape.

The following shapes are supported by `TLspShapeDiscretizer`, the default implementation of `ILspShapeDiscretizer`:

- `ILcdArc`
- `ILcdArcBand`
- `ILcdCircle`
- `ILcdEllipse`
- `ILcdPolyline`
- `ILcdPolygon`
- `ILcdComplexPolygon`
- `ILcdVariableGeoBuffer`
- `ILcdGeoBuffer`
- `ILcdCurve`
- `ILcdRing`
- `ILcdSurface`
- `ILcdText`
- `ILcdBounds`
- `ILcdExtrudedShape` if the base shape is one of the previous shapes in the list
- `TLcdLonLatBuffer`
- `TLcdLonLatHeightBuffer`
- `TLcdLonLatHeightDome`
- `TLcdLonLatHeightSphere`
- `ILcd3DArcBand`
- `ILcdShapeList`

If you want to paint a custom shape other than the supported shapes, you must extend `TLspShapeDiscretizer`. The class `TLspShapeDiscretizationParameters` allows you to modify the discretization process and results.



For `ILcdText` objects, the style in the discretization parameters is used to determine the font in which the text is rendered, provided that the style contains a `TLspTextStyle`. If not, a default font is used. Other shape types currently do not use the style.

CHAPTER 23

Labeling domain objects in a Lightspeed view

This chapter explains the support that LuciadLightspeed offers for labeling the representations of your domain objects in a Lightspeed view.

- Section 23.2 describes how to enable labels for your domain objects.
- Section 23.3 describes different styling options available for labels and how to define label content.
- Section 23.4 describes how to control the position of the labels.
- Section 23.5 describes how to configure label decluttering.
- Section 23.6 describes how to interactively move labels or edit them.
- Section 23.7 describes how to set up label-free areas on the screen.
- Section 23.8 describes how to handle painting order for 3D labels.
- Section 23.9 describes the details of the label placement algorithms.

23.1 What is a label?

A label is any description or classifier you attach to a domain object representation: a single word, a multi-line text string, an icon, or even an interactive panel.

LuciadLightspeed makes no distinction between regular painting and label painting. Just as regular painters, label painters implement the `ILspPainter` interface. Label painters can also be added to a layer, and they can be customized with an `ILspStyler`.

There are a few notable differences between label objects and regular painted objects though:

- Labels are usually painted on top of all regular object representations. See also Section 21.5.
- Labels are usually painted in screen coordinates, not as objects that are part of the displayed world.
- The position and visibility of individual labels can be coordinated across different layers.

23.2 Activating labeling

The recommended way to activate label painting for a model is to attach `ALspLabelStyler` objects to a layer's label painter using a `TLspShapeLayerBuilder`. The labels will appear if the styler returns at least one style for the domain objects. [Program 113](#) demonstrates how you can do this.

```
1 TLspShapeLayerBuilder layerBuilder = TLspShapeLayerBuilder.newBuilder();
2 layerBuilder.labelStyler(TLspPaintState.REGULAR, styler);
3 // other layer builder calls ...
4 ILspLayer layer = layerBuilder.build();
```

Program 113 - Enabling label painting by setting stylers on a `TLspShapeLayerBuilder`

You can also just set one or more fixed styles directly on the layer builder, without using a styler. [Program 114](#) demonstrates how you can do this.

```
1 layerBuilder.labelStyles(TLspPaintState.REGULAR, TLspTextStyle.newBuilder().build());
```

Program 114 - Enabling label painting by setting styles on a `TLspShapeLayerBuilder`

Alternatively, if you are not using a layer builder to create a `TLspLayer`, you can set a `TLspLabelPainter` on the layer for paint representation `LABEL`, and attach the stylers to that painter. [Program 115](#) shows how you can do this.

```
1 TLspLayer layer = new TLspLayer();
2 TLspLabelPainter labelPainter = new TLspLabelPainter();
3 labelPainter.setStyler(TLspPaintState.REGULAR, styler);
4 layer.setPainter(TLspPaintRepresentation.LABEL, labelPainter);
```

Program 115 - Enabling label painting by attaching a `TLspLabelPainter` to a layer

23.3 Customizing the label content and look

You can handle all of the customization of the labels through `ALspLabelStyler` objects. Styling allows you to specify the content, the look, the positioning and the number of displayed labels. `ALspLabelStyler` objects offer advanced options like defining multiple labels or specifying positioning or priorities. In some cases a regular `ILspStyler` is sufficient. It doesn't offer advanced labeling options though.

The criteria for choosing an `ALspLabelStyler` for labels are very much the same as those for choosing stylers for regular shape painting. You can pick a simple `TLspLabelStyler` with just one style, or you can decide to use advanced stylers that use the view scale to manage level-of-detail, or that handle animations. As with regular painting, you can set a styler for each `TLspPaintState`: `REGULAR`, `SELECTED` and `EDITED`. If you do not specify a styler for `SELECTED`, the painter will fall back to a modified `REGULAR` styler. If you do not specify a styler for `EDITED`, the painter will fall back to the `SELECTED` styler if available and otherwise to a modified `REGULAR` styler.

To see your labels, define your label styler so that it applies at least one style for the domain objects, for example a `TLspTextStyle` or a `TLspIconStyle`. Otherwise, nothing is displayed. For more information about applying text or icon styles, see [Section 23.3.1](#) and [Section 23.3.2](#). [Program 116](#) shows the most basic label stylers possible.

```

1 ILspStyler regularStyler = TLspTextStyle.newBuilder().textColor(Color.white).build();
2 ILspStyler selectedStyler = TLspTextStyle.newBuilder().textColor(Color.red).build();

```

Program 116 - Very basic label stylers

Program 117 shows a custom label styler that applies different styles to different domain objects.

```

1 public class MyStyler extends ALspLabelStyler {
2     @Override
3     public void style(Collection<?> aObjects, ALspLabelStyleCollector aStyleCollector,
4             TLspContext aContext) {
5         for (Object domainObject : aObjects) {
6             if (...) {
7                 aStyleCollector.object(domainObject)
8                     .styles(someStyles)
9                     .algorithm(someAlgorithm)
10                    .submit();
11             } else {
12                 aStyleCollector.object(domainObject)
13                     .styles(otherStyles)
14                     .algorithm(otherAlgorithm)
15                     .submit();
16             }
17         }
18     }

```

Program 117 - A styler that applies different styles to different domain objects

23.3.1 Using text as labels

The most common purpose of labels is to display textual information next to the domain object. You can display a textual label by using a `TLspTextStyle` in your label styler, as already shown in Program 116.

`TLspTextStyle` allows you to specify not only the presence of a label, but also the look of the label text. `TLspTextStyle` offers basic settings such as font, color and halo. It also allows you to set the alignment and line spacing for multi-line labels. Table 2 demonstrates these settings.

Style	Result
The default text style: <code>TLspTextStyle.newBuilder().build()</code>	
Text style with a different font: <code>TLspTextStyle.newBuilder().font("Serif-BOLD-18").build()</code>	
Text style in red without halo: <code>TLspTextStyle.newBuilder().textColor(Color.red).haloThickness(0).build()</code>	
Text style for multi-line labels: <code>TLspTextStyle.newBuilder().alignment(TLspTextStyle.Alignment.CENTER).build()</code>	

Style	Result
-------	--------

Table 2 - Text label styling options

If you would like to add properties of your domain objects as label content, use `TLspDataObjectLabelTextProviderStyle`. This implementation of `ALspLabelTextProviderStyle` is highly suitable for the usage of object properties as label text. See the section below for more information.

Customizing text label content

By default, the text content of a label is retrieved using the `toString()` method of the domain object. You can override the default content by adding an `ALspLabelTextProviderStyle` to your label styler. You can implement a text provider style yourself, or use a `TLspDataObjectLabelTextProviderStyle` to add properties of `ILcdDataObject` domain objects to your label. For more information about `ILcdDataObject` objects, see [Chapter 10](#).

Formulate the label content as expressions. If you specify more than one expression, each expression appears on a separate line in the label, as shown in [Program 118](#).

```
1 TLspDataObjectLabelTextProviderStyle.newBuilder()
2   expressions("STATE_NAME", "POP1996").build()
```

Program 118 - Using two `ILcdDataObject` properties as one multi-line text label

Changing text label content

Changes to the text content of labels are not necessarily picked up immediately.

This means you have to notify the painters when a domain object's text has changed. You can use two kinds of events to notify painters:

- Model change events for the relevant domain objects, for example through the `ILcdModel.elementChanged` method. These events should for example be used when the content of a data object is changed.
- Style change events for the relevant domain objects, for example through the `ALspStyler.fireStyleChangeEvent` method. These events should for example be used when using a new `TLspDataObjectLabelTextProviderStyle` with different expressions.

23.3.2 Using images as labels

Beside text content, you can also display arbitrary image content as a label. To display an image as a label, use a `TLspIconStyle` in your stylers. The `TLspIconStyle` contains an `ILcdIcon`. There are various ways to use this feature:

- A simple `TLcdImageIcon` based on a file. You can define the styler so that a property of the domain object determines which file is selected.
- A custom `ILcdIcon` implementation that uses AWT Graphics for icon drawing.

Changing icon-based label content

Similar to text content, labels painted by icons are not updated automatically when the icon changes. Updates are triggered by model change events or style change events only.

To detect a possible change in the icon, the painter uses the `equals` method. Therefore, two different icons should not compare as equal. For performance reasons though, make sure that identical icons always compare as equal. Preferably use the same instance.

23.3.3 Using Swing components as labels

Beside text and images, you can also display Swing components as a label. To do this, use `ALspSwingLabelStyler`. This styler has a method that returns a Swing component for a specific label. Using this component, the styler generates an image that is used by the label painter. Because of this it is not possible to interact with the used component. To see how it is possible to interact with labels that are composed of Swing components, see [Section 23.6.2](#).

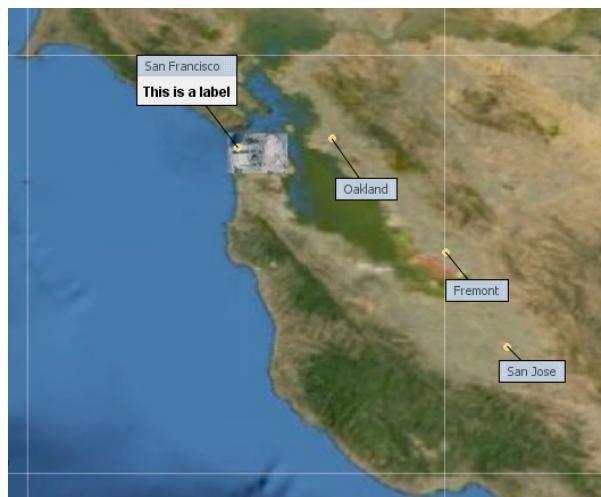


Figure 84 - Labels painted using Swing components. Screenshot taken from the `lightspeed.labels.interactive` sample.

23.3.4 Options for additional styling

In addition to the text and image styling options described above, you have a few other styles at your disposal to change the look of your labels. These additional styles can be used either with text labels or image labels:

- `TLspPinLineStyle`: indicate that a pin line should be drawn from the label to its domain object anchor point.
- `TLspLabelBoxStyle`: specifies stylings such as the background color, the presence of a label frame or a halo.
- `TLspLabelOpacityStyle`: specifies transparency of the labels, or a color to blend the label with. This style can be used efficiently with animations, highlighting or other changes that happen frequently, as this style is applied on the label using the graphics hardware without any texture redrawing.

[Table 3](#) shows how to use these styles and the results.

Style	Result
Pins: <code>TLspPinLineStyle.newBuilder().color(Color.red).width(2).build()</code>	

Style	Result
<p>Frame and fill:</p> <pre>TLspLabelBoxStyle.newBuilder().frameThickness(2). frameColor(Color.red).filled(true). fillColor(Color.lightGray).build()</pre>	
<p>Opacity and blend color:</p> <pre>TLspLabelOpacityStyle.newBuilder(). color(Color.red).opacity(0.5f).build()</pre>	

Table 3 - Additional labeling styling options

23.3.5 Defining multiple labels for a domain object

To display more than one label for your domain objects you need a custom `ILspStyler` that uses the `ALspLabelStyleCollector.label(aSublabelID)` call. As shown in [Program 119](#), you can specify different sets of styles for different sublabels for one domain object using `ALspLabelStyleCollector`.

The argument `aSublabelID` is an object that identifies the sublabel for a domain object. You would typically use a fixed set of different sublabels, such as a Java enum or a set of incrementing numbers.

You can safely use your own objects by following these guidelines:

- Make sure the sublabel ID is not equal for different labels on the same domain object. Otherwise only one of the two labels is displayed.
- The sublabel ID must be equal for the same label of a domain object on subsequent styling calls. Although the label will be displayed correctly, an unequal sublabel ID for the same label creates unnecessary overhead: the painter will assume one label was removed and a new one was added.
- The sublabel ID can be shared between different domain objects, so you can use constants.
- This `label()` call is optional if you need just one label: a default sublabel ID of "single" will be used.

[Program 119](#) shows you how you can define multiple labels.

```

1  public void style(Collection<?> aObjects, ALspLabelStyleCollector aStyleCollector,
2      TLspContext aContext) {
3      aStyleCollector.objects(aObjects).label(1).styles(someStyles).submit();
4      aStyleCollector.objects(aObjects).label(2).styles(otherStyles).submit();
5      aStyleCollector.objects(aObjects).label(3).styles(moreStyles).submit();
}
```

Program 119 - Defining multiple labels for domain objects in a `ALspLabelStyler`

23.4 Positioning labels

This section describes the way LuciadLightspeed positions labels with respect to the shape representing the domain object.

By default, LuciadLightspeed uses the type of shape to determine where to place labels, as shown in Figure 85:

- For **point** shapes, the labels are placed at an offset in one of the compass directions relative to the point. Figure 86 shows these possible positions.
- For **line** shapes, the labels are placed along the line.
- For **area** shapes, the labels are placed inside the area, near the center of mass.



Figure 85 - Default label positions for points, lines and polygons

You can control the positioning of your labels in various ways using the styling API.

The styling API allows you to:

- Define one or more relative locations. See [Section 23.4.1](#) for more information.
- Override the anchor point relative to which the position is calculated. See [Section 23.4.2](#) for more information.
- Specify another label relative to which the position is calculated. See [Section 23.4.3](#) for more information.

Alternatively, you can use a custom label placement algorithm, although this is more involved. For more information, refer to [Section 23.9.1](#).

23.4.1 Configuring relative label positions

You can fully customize the positioning of your label by specifying an algorithm in your `ILspStyler` using `ALspLabelStyleCollector`. [Section 23.9.1](#) describes the algorithms delivered with LuciadLightspeed and how to use them.

For convenience, labels on point-based domain objects are automatically positioned relative to the point using the `TLspLabelingAlgorithm`. By default, one of the eight compass direction locations shown in Figure 86 is used, or the label is placed on top of the point itself.

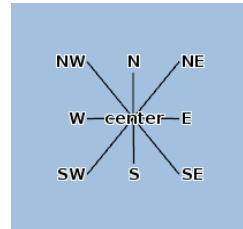


Figure 86 - All possible locations when using `TLspLabelLocationProvider`

You can override these positions by specifying a different set of locations in your styler:

- Using the `locations` method on `ALspLabelStyleCollector`. Note that this is a convenience method that internally calls the `algorithm` method with a `TLspLabelIn-gAlgorithm` configured with a `TLspLabelLocationProvider`.
- Using the `algorithm` method on `ALspLabelStyleCollector` with a `TLspLabelIn-gAlgorithm` configured with an `ALspLabelLocationProvider`. This can for example be `TLspLabelLocationProvider`, which is a convenience implementation that allows you to use the eight compass directions or place the label on top of the point, as shown in [Figure 86](#). For example, [Program 120](#) limits the label positioning possibilities to two locations, namely above and below the point. The offset between point and label is set to 50 pixels. Alternatively a custom implementation of `ALspLabelLocationProvider` can be provided. Consult the API documentation for more details.
- Using the `algorithm` method on `ALspLabelStyleCollector` with an other `ILspLa-belingAlgorithm` implementations.

```

1 private final ALspLabelLocationProvider fLocationProvider = new TLspLabelLocationProvider(50,
2   Location.NORTH, Location.SOUTH);
3
4 public void style(Collection<?> aObjects, ALspLabelStyleCollector aStyleCollector, TLspContext
5   aContext) {
6   aStyleCollector.objects(aObjects).algorithm(fAlgorithm).submit();
}

```

Program 120 - Defining two possible locations, above and below the object, offset by 50 pixels

23.4.2 Defining a different anchor point for labels

By default, a label is positioned relative to the domain object's focus point. Defining a different anchor point for the label may help you achieve the following goals:

- Add labels to different shapes in an `ILcdShapeList`.
- Add labels to different parts of a shape.
- Label a domain object which itself is no shape, by specifying a shape that should be labeled. For example, by using a property of the domain object that defines a location.

To specify a different anchor point for a label, you can use `ALspStyleCollector.geometry()` calls.

Just as for regular styling, use either a specific `ILcdShape` for each domain object, or pass an `ALspStyleTargetProvider` that calculates the anchor point when requested.

It is important to note that the anchor points defined this way need to be specified in the reference of the `ILcdModel` that contains the domain objects.

[Figure 87](#) shows such an example: each river polyline not only has a label for its name, but also a label indicating the river's spring and mouth. [Program 121](#) shows the `ALspStyler` and `ALspStyleTargetProvider` used to accomplish this.

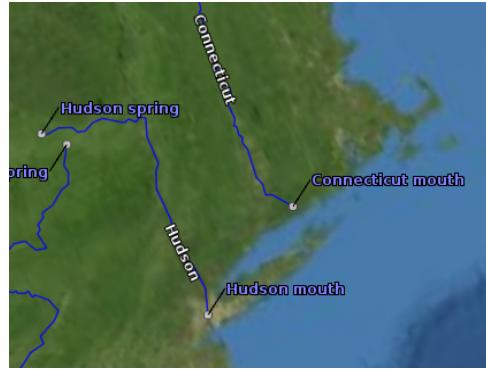


Figure 87 - Labels anchored to specific points

```

1 public void style(Collection<?> aObjects, ALspLabelStyleCollector aStyleCollector, TLspContext
      aContext) {
2     ALspLabelTextProviderStyle textProvider = new ALspLabelTextProviderStyle() {
3         public String[] getText(Object aDomainObject, Object aSublabelID, TLspContext aContext) {
4             return new String[] {aDomainObject.toString() + " " + aSublabelID.toString()};
5         }
6     };
7
8     ALspStyleTargetProvider firstPointProvider = new ALspStyleTargetProvider() {
9         public void getStyleTargetsSFCT(Object aObject, TLspContext aContext, List<Object>
      aResultSFCT) {
10        ILcdPointList points = (ILcdPointList) aObject;
11        aResultSFCT.add(points.getPoint(0));
12    }
13 };
14
15     ALspStyleTargetProvider lastPointProvider = new ALspStyleTargetProvider() {
16         public void getStyleTargetsSFCT(Object aObject, TLspContext aContext, List<Object>
      aResultSFCT) {
17            ILcdPointList points = (ILcdPointList) aObject;
18            aResultSFCT.add(points.getPoint(points.getPointCount() - 1));
19        }
20    };
21
22     aStyleCollector.objects(aObjects).label("river").styles(textStyle).submit();
23     aStyleCollector.objects(aObjects).label("spring").geometry(firstPointProvider).styles(
      textStyle, pinStyle, textProvider).submit();
24     aStyleCollector.objects(aObjects).label("mouth").geometry(lastPointProvider).styles(
      textStyle, pinStyle, textProvider).submit();
25 }
```

Program 121 - Defining different anchor points for labels

23.4.3 Defining labels with a dependency on other labels

If you are using multiple labels per domain object, you can position one label relative to another label, instead of to the labeled domain object. To achieve this, use the `ALspLabelStyleCollector.anchorLabel(aSublabelID)` call, and specify a label previously defined for the domain object as sublabel ID.

For example, [Figure 88](#) show a city with a couple of labels: the top icon represents the "info" label; with the city and state next to it as text labels. The bottom icon represents a "statistics" label with the city's population and housing units numbers next to it. If the icon labels move, the text labels stay with them.

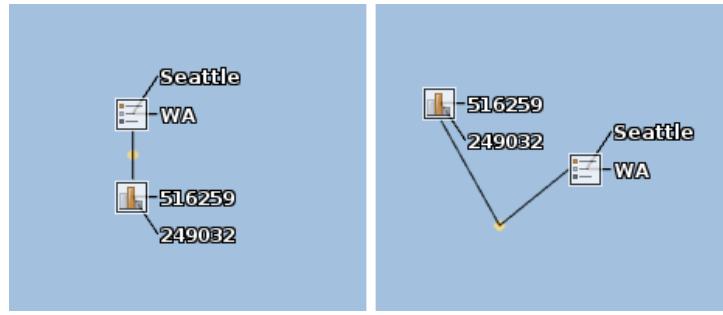


Figure 88 - Labels positioned relative to other labels

Program 122 shows the ALspLabelStyler used to accomplish this.

```

1 public void style( Collection<?> aObjects, ALspLabelStyleCollector aStyleCollector,
2   TLspContext aContext ) {
3   aStyleCollector.objects(aObjects).label("info").styles(infoStyles).submit();
4   aStyleCollector.objects(aObjects).label("name").anchorLabel("info").styles(nameStyles).
5     submit();
6   aStyleCollector.objects(aObjects).label("state").anchorLabel("info").styles(stateStyles).
7     submit();
8   aStyleCollector.objects(aObjects).label("statistics").styles(fStatsStyles).submit();
9   aStyleCollector.objects(aObjects).label("population").anchorLabel("statistics").styles(
10     populationStyles).submit();
11  aStyleCollector.objects(aObjects).label("housing_units").anchorLabel("statistics").styles(
12    housingUnitsStyles).submit();
13 }
```

Program 122 - Defining labels positioned relative to other labels

23.4.4 Configuring labels along a polyline's path

By default, for polyline objects, LuciadLightspeed uses straight labels that are aligned with the polyline at a certain point. Sometimes however, you want the labels to closely follow the polyline, so that the labels themselves are bent according to the polyline. You can do this by using the TLspCurvedPathLabelingAlgorithm.

Program 123 shows how to use this algorithm in your styler. Figure 89 shows the default behavior, regular labels placed aligned with a path, next to curved labels following a path closely using TLspCurvedPathLabelingAlgorithm.

```

1 private final ILspLabelingAlgorithm fCurvedPathAlgorithm = new TLspCurvedPathLabelingAlgorithm
2   ();
3
4 public void style(Collection<?> aObjects, ALspLabelStyleCollector aStyleCollector, TLspContext
5   aContext) {
6   aStyleCollector.objects(aObjects).styles(...).algorithm(fCurvedPathAlgorithm).submit();
7 }
```

Program 123 - Defining labels closely following a polyline

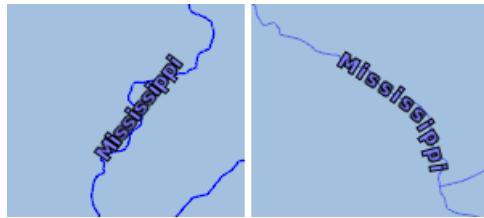


Figure 89 - Labels positioning along a path (left) versus following a path (right)

23.5 Configuring label decluttering

Each Lightspeed view has decluttering algorithms to make sure labels never overlap with each other. By default, all layers in the view and their labels are automatically decluttered together, so that no label in the view overlaps with another label.

You can customize this behavior for each layer or even for each individual label. To do so, you must define and use a placement group.

A declutter group specifies:

- The decluttering algorithm to use:
 - If you want labels to be decluttered, use a `TLspLabelConflictChecker`.
 - If you do not want any label decluttering, use a `TLspNoDeclutterLabelConflictChecker`.
 - You can also provide your own implementation of `ILspLabelConflictChecker`.
- Areas in the view that cannot be used to place labels. You can optionally attach `ILspLabelObstacleProvider` objects to a group using `TLspLabelPlacer.addLabelObstacleProvider`. Section 23.7 explains this in detail.

The label placement process treats each placement group separately, so labels from different placement groups can overlap.

You can add your own placement group using `TLspLabelPlacer.addPlacementGroup`. Each view already contains four pre-defined placement groups that you can use:

- `TLspLabelPlacer.DEFAULT_DECLUTTER_GROUP`: this group is used by default for all layers and all their labels. No labels in this group can overlap.
- `TLspLabelPlacer.DEFAULT_NO_DECLUTTER_GROUP`: this group can be used for labels that should always be shown, even if they overlap.
- `TLspLabelPlacer.DEFAULT_GRID_GROUP`: this group is used for grid labels by default. Because this is a separate group, grid labels are decluttered amongst themselves, but they can still overlap with other labels.
- `TLspLabelPlacer.DEFAULT_REALTIME_GROUP`: this convenience group can be used when labeling dynamic data, for example tracks. This group allows them to be decluttered independently from static labels.

The easiest way to use a declutter group is through the `ALspLabelStyleCollector`, e.g. by using `ALspLabelStyler`. You can specify the group to use for all labels, or for differently for individual labels if necessary. Program 124 shows an example.

```

1 public void style(Collection<?> aObjects, ALspLabelStyleCollector aStyleCollector, TLspContext
      aContext) {
2     aStyleCollector.objects(aObjects).label(1).group(TLspLabelPlacer.DEFAULT_DECLUTTER_GROUP).
      styles(...).submit();
3     aStyleCollector.objects(aObjects).label(2).group(TLspLabelPlacer.DEFAULT_NO_DECLUTTER_GROUP)
      .styles(...).submit();
4 }

```

Program 124 - Defining the declutter group for labels

23.5.1 Setting label priorities across layers

During decluttering, the label placement process tries to find a location for each label where no other label has been placed yet. Because certain labels may be more important than others, you can specify a priority for each label type. All labels in the declutter group are sorted by this priority and placed in this order. High priority labels are placed first, and lower priority labels may be skipped if there is no spare view room left. Since a declutter group can contain labels from different layers, the priorities are compared across layers.



The priorities are compared across different layers, so you have to align them. Also note that a lower number means a higher priority.

Priorities are defined through the `ALspLabelStyleCollector`:

- Use a fixed priority. This can be used in simple cases where all labels have the same priority. [Program 125](#) shows this: domain objects have two labels with different (static) priorities.
- Use a `ILspLabelPriorityProvider`. This is often easier when each label has its own calculated priority. [Program 126](#) shows this: labels of capital cities have precedence over others.

```

1 public void style(Collection<?> aObjects, ALspLabelStyleCollector aStyleCollector, TLspContext
      aContext) {
2     aStyleCollector.objects(aObjects).label(1).priority(100).styles(...).submit();
3     aStyleCollector.objects(aObjects).label(2).priority(200).styles(...).submit();
4 }

```

Program 125 - Defining label priorities

```

1 ILspLabelPriorityProvider citiesPriorityProvider = new ILspLabelPriorityProvider() {
2     public int getPriority(TLspLabelID aLabelID, TLspPaintState aPaintState, TLspContext
      aContext) {
3         if (isCapital( aDomainObject )) {
4             return 10;
5         } else {
6             return 20;
7         }
8     }
9 }
10
11 public void style( Collection<?> aObjects, ALspLabelStyleCollector aStyleCollector,
      TLspContext aContext ) {
12     aStyleCollector.objects( aObjects ).priority(citiesPriorityProvider).styles( ... ).submit();
13 }

```

Program 126 - Defining label priorities using a `ILspLabelPriorityProvider`

23.6 Interacting with labels

Users can interact with labels just like they can with representations of domain objects: they can select labels and perform certain operations on them. Section 23.6.1 explains how to move labels. If you want to offer more complex interaction, you can use interactive labels, which allow you to represent your label with any Swing component and interact with it. This is explained in Section 23.6.2.

23.6.1 Moving labels

Just as domain object modifications can be triggered by input events, an input event can also trigger the editing of one or more labels painted in the view. You can make use of the LuciadLightspeed label editor classes to modify the label positions.

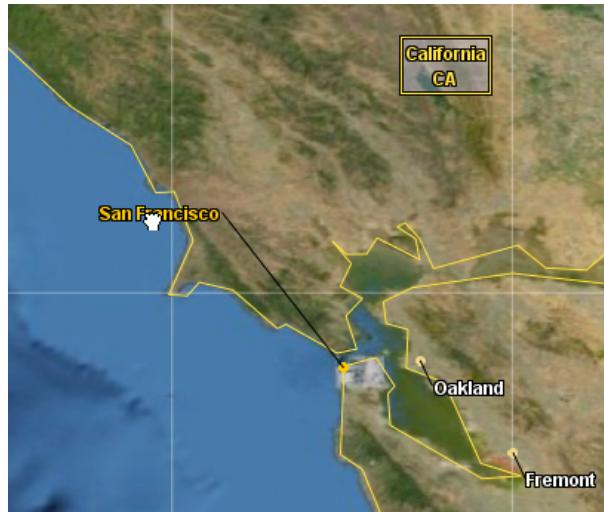


Figure 90 - Moving a label

You can easily enable label editing on a `TLspShapeLayerBuilder`, as shown in Program 127. The layer will use the default label editor implementation, `TLspLabelEditor`.

```
1 layerBuilder.setEditable(true);
```

Program 127 - Enabling editable labels using a `TLspShapeLayerBuilder`

If you create your own layer without using a layer builder, the following steps are required:

1. **Choose an editor:** either use the default `TLspLabelEditor`, or create a custom implementation of `ILspEditor`.
2. **Create an editable layer:** to configure a layer as editable, set the `editable` flag to `true`: `layer.setEditable(true)` and register the chosen label editor with the correct paint representation: `layer.setEditor(paintRepresentation, chosenEditor)`.
3. **Set the edit controller:** set the `TLspEditController` on the view with `view.setController(new TLspEditController())`, so that the editor can receive input events from the UI handles it provides. Note that the default controller can edit as well.

The default label editor implementation provided by LuciadLightspeed is `TLspLabelEditor`. This label editor converts label locations to `TLspStampLabelLocation` objects. This makes

it possible to move a label anywhere on the screen.

23.6.2 Creating interactive labels that users can edit

An interactive label is a label that users can interact with to modify properties of domain objects. You can configure interactive labels using an `ALspInteractiveLabelProvider`. To display and edit interactive labels, you can use a `TLspInteractiveLabelsController`. For more information about the usage of controllers, see [Chapter 26](#).

For performance reasons, interactive labels are only shown when the user moves the mouse cursor over a regular label. If the `ALspInteractiveLabelProvider` indicates that it can provide an interactive label, the interactive label is presented to the user instead of the regular label.

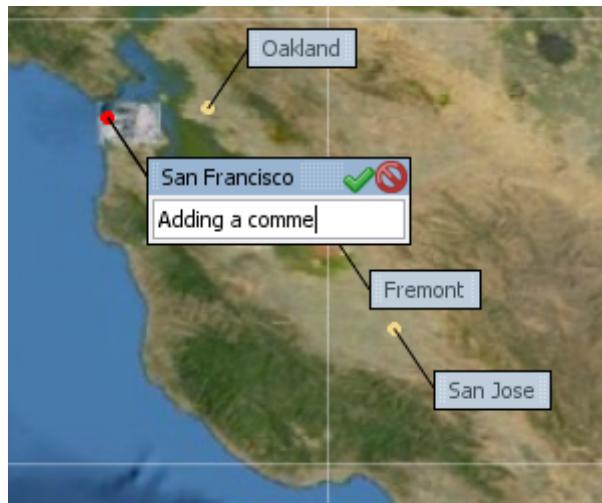


Figure 91 - Interactive label during editing. Screenshot taken from the `lightspeed.labels.interactive` sample.

Implementing an `ALspInteractiveLabelProvider`

When you are implementing an `ALspInteractiveLabelProvider`, the most important method is the `startInteraction` method. This method returns the Swing component with which the user can interact. This component can be any Swing component: it can be a single text field or a `JPanel` containing check boxes, combo boxes, and more.

There can be only one interactive label from the same provider at any time. Therefore, the implementation of this method can always return the same instance of the Swing component. The methods of `ALspInteractiveLabelProvider` are always called in a certain order. There are never two calls to `startInteraction` before the previous interaction is stopped or canceled. This allows you to attach listeners to the domain object and the Swing components in the `startInteraction` method, for example, and to remove these listeners in the implementation of the `stopInteraction` and `cancelInteraction` methods.

Activating interactive labels

`TLspInteractiveLabelsController` shows interactive labels based on mouse-moved events. When the mouse hovers over a label, it provides an interactive label for that label. For touch input, where hover actions do not exist, the interactive label is put in place at the first touch.

It is also possible to bypass the automatic behavior. When the `setProvideInteractiveLabelOnMouseOver` method of `TLspInteractiveLabelsController` is set to `false`, no interactive labels are shown automatically. You can then ask the `ALspInteractiveLabelProvider` to provide an interactive label with the `startInteraction` method. To stop or cancel the label interaction, you can use the `stopInteraction` and `cancelInteraction` methods respectively as this no longer happens automatically either. Bypassing the automatic behavior allows you, for example, to implement a scenario in which a user cycles through the labels by pressing a certain key on the keyboard, making each label interactive one after the other.

Using mouse and touch events

In order to move the interactive labels, the `ALspInteractiveLabelProvider` can dispatch mouse or touch events that happen on the interactive label to the view. The active `ILspController` can then receive the events and use those, for example, to move the label with the label editor. Which events are dispatched to the view and which events are dispatched to the interactive label is decided by the `dispatchAWTEvent` method of the `ALspInteractiveLabelProvider`. By default this method dispatches the event to the view when it happened on a `JLabel` or `JPanel`. It dispatches all other events to the originating component itself. As a result, when the user clicks and drags on a `JLabel`, the controller of the view can handle these events and can, for example, move the label. Clicking on, for example, a `JSlider` would as expected drag the knob. You can override this method to customize which events are dispatched to the interactive label and which are dispatched to the view.

23.7 Using obstacle providers

The label decluttering algorithms used by the `ILspLabelPlacer` can ensure that no two labels overlap on the screen.

However, it is often useful to mark areas in the screen where no labels should be displayed. For example:

- The body representations of domain objects. You could avoid displaying labels over icons in the view.
- GUI components overlaid on the view. You could avoid displaying labels behind those components since they would be hard to read.

To accomplish this, you can attach `ILspLabelObstacleProvider` objects to a declutter group. An obstacle provider returns a set of areas on the screen where no labels should be placed.

To attach an obstacle provider to a declutter group, use `TLspLabelPlacer.addLabelObstacleProvider()`. Note that you are responsible for removing the obstacle provider from the placer at the appropriate time.



Since an obstacle provider is attached to a declutter group, all labels and algorithms used in that group take the areas returned by the providers into account.

23.8 Painting 3D labels

All the labeling functionality described above applies both to 2D and 3D views. Positioning and decluttering work in the same way, although terrain can influence the visibility of the labels.

You need to take into account a subtlety with respect to painting order, however. In 2D, labels are always painted on top of other content. In 3D, labels are painted in the `OPAQUE` paint phase. This means other content painted in the `TRANSPARENT` phase can be over the labels if it is closer to the camera. [Figure 92](#) shows this.



Figure 92 - Labels can be obscured by content in 3D.

If you do not want this behavior, you can activate full overlay using `TLspLabelPainter.setOverlayLabels`. [Figure 93](#) shows the result.



Figure 93 - Labels overlaid in 3D.

23.9 Advanced label placement

The previous sections explain the various ways to configure labeling in your application and should be enough for most usages. This section provides more background information on label placement.

It is possible for the view to position and declutter labels. For this purpose `ILspLabelPlacer` is used. Whereas label painters determine the content and appearance of the labels, `ILspLabelPlacer` computes if and where the label is positioned. On top of that, it stores the label location information so that the painter, for example, can access it.



Note: the default label placer works asynchronously. This means the label visibility is not updated on every paint, and can scale better with many labels.

The label placement and decluttering functions, of which the goal is to avoid overlap between labels, can take multiple labels of a group of domain objects into account. The group can be so large as to contain all the objects visible in the view, including objects of different layers. This allows for the more advanced placement of labels, to improve readability and avoid overlap.

[Section 23.9.1](#) explains the label placer implementations and how they use an algorithm to compute the label position. It has a subsection that details the labeling algorithm implementations and one that shows how custom labeling algorithms can be created or custom priorities can be set. [Section 23.9.2](#) explains how label locations can be accessed programmatically.

23.9.1 Using label placers to locate labels

Because a label placer operates on the labels of all layers, it is located in the `ILspView`, and can be retrieved using the `getLabelPlacer` method. In the Lightspeed view implementations, a default `ILspLabelPlacer` is already set. It can be configured in the layers. If this does not suffice, you can configure the label placer manually, and set it on the layer using the `setLabelPlacer` method of `ILspView`.

Lightspeed views offers one label placer implementation:

- `TLspLabelPlacer` is an implementation of `ILspLabelPlacer`. It can be configured with a small set of methods.

Using an algorithm for label decluttering

The LuciadLightspeed label placer implementations use labeling algorithms to place and declutter labels. It is possible to customize the placement of labels by choosing a different labeling algorithm.

LuciadLightspeed offers the following labeling algorithms:

- `TLspLabelingAlgorithm` is a decluttering algorithm that iterates over a number of customizable locations.
- `TLspOnPathLabelingAlgorithm` is a decluttering algorithm that places straight labels along a path.
- `TLspInPathLabelingAlgorithm` is similar to `TLspOnPathLabelingAlgorithm` but positions labels inside a shape instead of on its perimeter.
- `TLspCurvedPathLabelingAlgorithm` is similar to `TLspOnPathLabelingAlgorithm` but places curved labels on a path.
- `TLspCompositeLabelingAlgorithm` allows you to specify a label placement algorithm per layer, per object, or even per label. The composite algorithm partitions the labels into groups that are passed on to the relevant delegate algorithms.
- `TLspCompositeDiscreteLabelingAlgorithm` is similar to `TLspCompositeLabelingAlgorithm` but allows interleaving the algorithms during the placement step. This means that you can, for example, first place a line label, then a point label, then another line label, and so on.

See the package `com.luciad.view.lightspeed.label.algorithm` for more information.

Enabling a certain algorithm can be done through the `ALspLabelStyler`, as shown in Program 128.

```

1  public void style(Collection<?> aObjects, ALspLabelStyleCollector aStyleCollector,
2                     TLspContext aContext) {
3     aStyleCollector.objects(aObjects).label(1).algorithm(algorithm1).submit();
4     aStyleCollector.objects(aObjects).label(2).algorithm(algorithm2).submit();
}

```

Program 128 - Specifying labeling algorithms using `ALspLabelStyleCollector`

Defining a custom algorithm for the placement of labels

One way to customize labeling algorithms is to wrap them, to further remove unwanted labels for example. If you want even more control of label placement, you can consider implementing

your own labeling algorithm.

23.9.2 Accessing label locations programmatically

Apart from placing labels, `ILspLabelPlacer` also keeps track of the storage location of labels. For this, it uses `ALspLabelLocations`, accessed with the `getLabelLocations()` method of `ILspLabelPlacer`. This object makes it possible to check if labels are visible and where they are located, or to listen to visibility or location changes. It also makes it possible to manipulate label locations or visibility manually.

CHAPTER 24

Visualizing large vector data sets on a Lightspeed view

The visualization of a large data set on a map poses 2 big challenges:

- *Preventing visual clutter* so that the user can distinguish between the features. Typically, this is achieved by painting less data when the map is zoomed out.
- *Keeping the map and application responsive* by preventing that all data is loaded in memory.

Those two requirements go hand in hand: if the number of features that is visible on the map is limited at all times, the amount of data that must be loaded into memory is kept under control as well.

This chapter describes how you can use a `TLcdModelQueryConfiguration` to control the amount of data that is loaded by the layer during a paint operation. You can create such a `TLcdModelQueryConfiguration` through the API or through the use of an SLD styling file. See [Section 24.2](#) and [Section 24.3](#)) for more information about both approaches. The subsequent sections provide illustration with some examples.

For a walkthrough of a concrete example, see the article “How to visualize OpenStreetMap data with LuciadLightspeed” on the Luciad Developer Platform (<https://dev.luciad.com>).

24.1 The role of the `TLcdModelQueryConfiguration`

When a layer is visualized (= painted) onto a map, the layer needs to retrieve the relevant data from the `ILcdModel`. It is important that the layer queries the model for as little data as possible.

Data in the model does not necessarily consume JVM memory. For example, when the data from a database is being visualized, the data is stored in a database outside of the JVM. However, once the model sends data to the layer to be painted, the data gets loaded into the JVM memory.

Therefore, the layer should only request data that:

- *Lies within the current bounds of the view.* For example, when the map is centered on America, it makes no sense to query the model for data of Europe because that data will not be visible to the user.
- *Will effectively be painted.* When a layer is painted, the `ILspStyler` (for Lightspeed layers) or the `ILcdGXYPainter` (for GXY layers) can decide to skip a feature instead of

visualizing it. If the feature is not going to be painted, there is no need to retrieve it from the model.

While the layer can automatically derive the currently visible bounds from the view, it has no information about the features that need to be visualized. You can provide that information to the layer by configuring a `TLcdModelQueryConfiguration` on the layer.

The `TLcdModelQueryConfiguration` defines a filter for each map scale. That filter is used to query the model through the `ILcdModel.query` method. The filter takes the form of an `ILcdOGCCcondition`, which allows the model to interpret the filter.



For more information about how a filter can influence the performance of a query, see [Section 24.6](#).

Once a `TLcdModelQueryConfiguration` has been created, it can be configured on the layer using:

- `TLspLayer.setModelQueryConfiguration` or `TLspShapeLayerBuilder.setModelQueryConfiguration` for Lightspeed layers.
- `TLcdGXYLayer.setModelQueryConfiguration` for GXY layers.

Examples of the creation and configuration of such a `TLcdModelQueryConfiguration` are available in the following sections.

24.2 Limiting data loading through the API

You can limit the amount of data that is loaded by the layer during a paint operation by configuring a `TLcdModelQueryConfiguration` on the layer. As a result, you prevent that the layer needs to request all data from the model.

For example, when visualizing roads data, you would:

- Only show the highways when the map has been zoomed out
- Start including the minor roads when the user zooms in on the map

You can express this in a `TLcdModelQueryConfiguration`, and configure it on the layer:

```

1 import static com.luciad.ogc.filter.model.TLcdOGCFilterFactory.*;
2 import static com.luciad.view.TLcdModelQueryConfiguration.*;
3
4 //Define filters for the different types of data you want to load
5 //This code uses the utility methods from TLcdOGCFilterFactory
6 ILcdOGCCcondition highwaysFilter =
7     eq(propertyName("roadType"), literal("highway")); //roadType == highway
8 ILcdOGCCcondition minorRoadsFilter =
9     eq(propertyName("roadType"), literal("minor")); //roadType == minor
10
11 //Create the model query configuration using the Builder class
12 //A model query configuration is used to define which data should be loaded at which scale
13 TLcdModelQueryConfiguration config =
14     TLcdModelQueryConfiguration.newBuilder()
15         //always include the highways
16         .addContent(FULLY_ZOOMED_OUT, FULLY_ZOOMED_IN, highwaysFilter)
17         //include the minor roads only from a certain scale
18         .addContent(1/50000, FULLY_ZOOMED_IN, minorRoadsFilter)
19         .build();
20
21 //Set it on your Lightspeed layer
22 TLspLayer layer = TLspShapeLayerBuilder.newBuilder()
23             .model(model)

```

```

24             .modelQueryConfiguration(config)
25             .build;
26
27 //Or, set it on a GXY layer
28 TLcdGXYLayer gxyLayer = ...;
29 gxyLayer.setModelQueryConfiguration(config);

```

Program 129 - Creating and configuring a `TLcdModelQueryConfiguration`

See [Section 24.4](#) for more details of this example.

24.3 Limiting data loading using SLD

24.3.1 Introduction to SLD files

SLD or Styled Layer Descriptor is an OGC standard for describing the appearance of a layer. It allows you to define:

- *Which features to load* at a certain scale
- *How to style* those features

You can do so by defining SLD rules. For example, the following rule ensures that bicycle roads are visualized only when the user zooms in beyond a scale of 1/2000. The bicycle roads are visualized with a blue dashed line.

```

1 <Rule>
2   <!-- The filter defines which to which features the rule applies -->
3   <ogc:Filter>
4     <ogc:PropertyIsEqualTo>
5       <ogc:PropertyName>roadType</ogc:PropertyName>
6       <ogc:Literal>bicycle</ogc:Literal>
7     </ogc:PropertyIsEqualTo>
8   </ogc:Filter>
9
10  <!-- Minimum and maximum scale denominators define the valid scale range of the rule -->
11  <MaxScaleDenominator>2000</MaxScaleDenominator>
12
13  <!-- The symbolizer(s) define the styling to use -->
14  <LineSymbolizer>
15    <Stroke>
16      <CssParameter name="stroke">#0000FF</CssParameter>
17      <CssParameter name="stroke-width">3</CssParameter>
18      <CssParameter name="stroke-dasharray">5 2</CssParameter>
19    </Stroke>
20  </LineSymbolizer>
21 </Rule>

```

Program 130 - Example of an SLD rule

See [Chapter 51](#) for more information about SLD, and how to use SLD in LuciadLightspeed.

24.3.2 Limit data loading using SLD

To limit the amount of data loaded during a paint operation, you can manually define filters and store them in a `TLcdModelQueryConfiguration`, as explained in [Section 24.2](#). However, when you are using SLD to style your data, the SLD rules already define which filter to use for each scale. LuciadLightspeed offers utility methods and classes to re-use that information for the automatic construction of the `TLcdModelQueryConfiguration`.

The following program illustrates how you can do that:

```

1 //First parse the SLD file
2 TLcdSLDFeatureTypeStyleDecoder decoder = new TLcdSLDFeatureTypeStyleDecoder();
3 String source = "/path/to/SLD/file.sld";
4 TLcdSLDFeatureTypeStyle sldStyle = decoder.decodeFeatureTypeStyle(source);
5
6 //Configure the SLD styling on your Lightspeed layer
7 TLspLayer layer = TLspShapeLayerBuilder.newBuilder()
8                         .model(roadsModel)
9                         .sldStyle(sldStyle)
10                        .build();
11
12 //Or, create a GXY layer that uses this SLD
13 TLcdGXYSLDLayerFactory layerFactory = new TLcdGXYSLDLayerFactory();
14 ILcdGXYLayer gxyLayer = layerFactory.createGXYLayer(roadsModel, Collections.singletonList(
    sldStyle));

```

Program 131 - Configuring a `TLcdModelQueryConfiguration` through SLD

The utility method `TLspShapeLayerBuilder.sldStyle`, or the `TLcdGXYSLDLayerFactory` for GXY layers, ensure that the layer under construction is configured to use the SLD styling for the visualization. It will use the filters defined in the SLD rules to construct the `TLcdModelQueryConfiguration` and limit the amount of data loading.



When you open data on a Lightspeed view in Lucy, Lucy automatically picks up any SLD files stored next to the source data. This allows you to limit the data loading in Lucy. Consult the Lucy developer's guide for more information.

24.4 Example: visualizing roads data

In this example, we assume that we have a data set containing roads of different types: highways, major roads, minor roads, and so on.

When the map user zooms out and looks at the whole world, we want to show the highways only. At that zoom level, it does not make sense to try and visualize any other road types, because the whole map would be cluttered with data.

When the user zooms in to a scale at which the map shows a single country, we want to show the highways and major roads. As the user zooms in further, we gradually want to include more of the smaller road types.

You can approach this directly through the `TLcdModelQueryConfiguration` API, or you can use an SLD file.

24.4.1 Using the `TLcdModelQueryConfiguration` API

You create a `TLcdModelQueryConfiguration` through a Builder class:

```
1 TLcdModelQueryConfiguration.Builder builder = TLcdModelQueryConfiguration.newBuilder();
```

Program 132 - Creating a builder for a `TLcdModelQueryConfiguration`

On that builder, we specify when each of the road types should be queried from the model. We start with the highways, which are visible at all times. As such, they must be queried from the model at each scale.

```
1 //Create a condition which only accepts features where the "roadType" property has "highway"
   as value
```

```

1 ILcdOGCCondition highwaysCondition =
2   TLcdOGCFactorFactory.eq(TLcdOGCFactorFactory.property("roadType"), TLcdOGCFactorFactory.
3     literal("highway"));
4
5 //Put the condition on the filter for the whole scale range
6 builder.addContent(TLcdModelQueryConfiguration.FULLY_ZOOMED_OUT, TLcdModelQueryConfiguration.
7   FULLY_ZOOMED_IN, highwaysCondition);

```

Program 133 - Configuring the filter for the highways

The major roads should only be visible when the user starts zooming in.

```

1 ILcdOGCCondition majorRoadsCondition =
2   TLcdOGCFactorFactory.eq(TLcdOGCFactorFactory.property("roadType"), TLcdOGCFactorFactory.
3     literal("major_road"));
4
5 //Major roads should be visible from a certain scale until completely zoomed in
6 builder.addContent(1.0 / 700000.0, TLcdModelQueryConfiguration.FULLY_ZOOMED_IN,
7   majorRoadsCondition);

```

Program 134 - Configuring the filter for the major roads

Once we have done the same for all other road types, we build the configuration and assign it to the layer:

```

1 TLcdModelQueryConfiguration config = builder.build();
2
3 //Lightspeed layer
4 TLspLayer roadsLayer =
5   TLspShapeLayerBuilder.newBuilder().model(roadsModel)
6     .modelQueryConfiguration(config)
7     .bodyStyler(...)
8     .labelStyler(...)
9     .build();
10
11 //GXY layer
12 TLcdGXYLayer roadsGXYLayer = ...;
13 layer.setModelQueryConfiguration(config);

```

Program 135 - Setting the TLcdModelQueryConfiguration on the layer

Code readability can be improved by using some static imports, and by using the fluent API of the builder class:

```

1 import static com.luciad.ogc.filter.model.TLcdOGCFactorFactory.*;
2 import static com.luciad.view.TLcdModelQueryConfiguration.*;
3
4 ILcdOGCCondition highwaysCondition = eq(property("roadType"), literal("highway"));
5 ILcdOGCCondition majorRoadsCondition = eq(property("roadType"), literal("major_road"));
6
7 TLcdModelQueryConfiguration config =
8   TLcdModelQueryConfiguration.newBuilder()
9     .addContent(FULLY_ZOOMED_OUT, FULLY_ZOOMED_IN, highwaysCondition)
10    .addContent(1.0 / 700000.0, FULLY_ZOOMED_IN, majorRoadsCondition)
11    .build();
12
13 //Lightspeed layer
14 TLspLayer roadsLayer =
15   TLspShapeLayerBuilder.newBuilder().model(roadsModel)
16     .modelQueryConfiguration(config)
17     .bodyStyler(...)
18     .labelStyler(...)
19     .build();
20
21 //GXY layer
22 TLcdGXYLayer roadsGXYLayer = ...;

```

```
23 | layer.setModelQueryConfiguration(config);
```

Program 136 - Full code

24.4.2 Using SLD

To express the required behavior with SLD, we need to define rules for each road type. For example, highways should always be painted, so we can use a rule that does not include any scale denominators.

```
1 <Rule>
2   <ogc:Filter>
3     <ogc:PropertyIsEqualTo>
4       <ogc:PropertyName>roadType</ogc:PropertyName>
5       <ogc:Literal>highway</ogc:Literal>
6     </ogc:PropertyIsEqualTo>
7   </ogc:Filter>
8   <LineSymbolizer>
9     <Stroke>
10    <SvgParameter name="stroke-width">2</SvgParameter>
11    <SvgParameter name="stroke">#809bc0</SvgParameter>
12  </Stroke>
13 </LineSymbolizer>
14 </Rule>
```

Program 137 - Rule for the highways

The major roads should only become visible when the user starts zooming in. Therefore, we specify a minimum scale denominator in the major roads rule.

```
1 <Rule>
2   <ogc:Filter>
3     <ogc:PropertyIsEqualTo>
4       <ogc:PropertyName>roadType</ogc:PropertyName>
5       <ogc:Literal>major_road</ogc:Literal>
6     </ogc:PropertyIsEqualTo>
7   </ogc:Filter>
8   <MinScaleDenominator>2500000</MinScaleDenominator>
9   <LineSymbolizer>
10    <Stroke>
11      <SvgParameter name="stroke-width">1</SvgParameter>
12      <SvgParameter name="stroke">#97d397</SvgParameter>
13    </Stroke>
14 </LineSymbolizer>
15 </Rule>
```

Program 138 - Rule for the major roads

Once we have rules for each type of road we want to include, we combine them into an SLD file:

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <FeatureTypeStyle
3   xsi:schemaLocation="http://www.opengis.net/sld StyledLayerDescriptor.xsd"
4   xmlns="http://www.opengis.net/sld"
5   xmlns:ogc="http://www.opengis.net/ogc"
6   xmlns:xlink="http://www.w3.org/1999/xlink"
7   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
8
9   <!-- Put all the rules created above here -->
10  <Rule></Rule>
11 </FeatureTypeStyle>
```

Program 139 - Combining the rules in a single SLD file

You apply the SLD file by decoding it and configuring it on the layer:

```

1 TLcdSLDFeatureTypeStyleDecoder decoder = new TLcdSLDFeatureTypeStyleDecoder();
2 String source = "/path/to/SLD/file.sld";
3 TLcdSLDFeatureTypeStyle sldStyle = decoder.decodeFeatureTypeStyle(source);
4
5 //For Lightspeed layers
6 TLspLayer layer =
7     TLspShapeLayerBuilder.newBuilder().model(roadsModel)
8         .sldStyle(sldStyle)
9         .build();
10
11 //For GXY layers
12 TLcdGXYSLDLayerFactory layerFactory = new TLcdGXYSLDLayerFactory();
13 ILcdGXYLayer gxyLayer =
14     layerFactory.createGXYLayer(roadsModel, Collections.singletonList(sldStyle));

```

Program 140 - Applying the SLD on a layer

24.5 Example: limiting visualization to a specific scale range

In this example, we assume that we have a data set containing very detailed features. The data should be visualized on the map only when the user has zoomed in to a sufficiently detailed level. When the data is visualized, all features for the current region must be painted. No extra filtering is required.

We can approach this by using the `TLcdModelQueryConfiguration` API directly, or by using an SLD file.

This example is the equivalent of calling `TLspLayer.setScaleRange` or `TLcdGXYLayer.setScaleRange`.



The major benefit of the approach in this example is that it uses unit-less scales like the ones used on a paper map (for example 1/50000). The scales used in the `setScaleRange` methods are internal scales, expressed in pixels / world unit. Those internal scales are harder to reason about, and are dependent on the view.

24.5.1 Using the `TLcdModelQueryConfiguration` API

You create a `TLcdModelQueryConfiguration` through a Builder class:

```
1 TLcdModelQueryConfiguration.Builder builder = TLcdModelQueryConfiguration.newBuilder();
```

Program 141 - Creating a builder for a `TLcdModelQueryConfiguration`

On this builder, we define that all data must be loaded in the required scale range:

```

1 //Use null as condition to indicate that no filtering is needed
2 ILcdOGCCondition loadAllCondition = null;
3
4 //Put the condition on the filter for the desired scale range
5 builder.addContent(1.0 / 75000.0, TLcdModelQueryConfiguration.FULLY_ZOOMED_IN,
    loadAllCondition);

```

Program 142 - Load everything in the desired scale range

Next, we have to indicate that no data should be loaded for any scale outside this range:

```
1 builder.loadNothingForUndefinedScales();
```

Program 143 - Load nothing outside the desired scale range

The configuration can now be put on the layer:

```
1 TLcdModelQueryConfiguration config = builder.build();
2
3 //Lightspeed layer
4 TLspLayer layer =
5   TLspShapeLayerBuilder.newBuilder().model(model)
6     .modelQueryConfiguration(config)
7     .build();
8
9 //GXY layer
10 TLcdGXYLayer gxyLayer = ...;
11 layer.setModelQueryConfiguration(config);
```

Program 144 - Setting the `TLcdModelQueryConfiguration` on the layer

Code readability can be improved by using some static imports, and by using the fluent API of the builder class:

```
1 import static com.luciad.ogc.filter.model.TLcdOGCFilterFactory.*;
2 import static com.luciad.view.TLcdModelQueryConfiguration.*;
3
4 ILcdOGCCondition loadAllCondition = null;
5 TLcdModelQueryConfiguration config =
6   TLcdModelQueryConfiguration.newBuilder()
7     .addContent(1.0 / 75000.0, FULLY_ZOOMED_IN, loadAllCondition)
8     .loadNothingForUndefinedScales()
9     .build();
10
11 //Lightspeed layer
12 TLspLayer layer =
13   TLspShapeLayerBuilder.newBuilder().model(model)
14     .modelQueryConfiguration(config)
15     .build();
16
17 //GXY layer
18 TLcdGXYLayer gxyLayer = ...;
19 layer.setModelQueryConfiguration(config);
```

Program 145 - Full code

24.5.2 Using SLD

To express the required behavior with SLD, we need an SLD file with a single rule. The rule should not contain a filter and apply a maximum scale denominator to ensure that the data becomes visible only when the user zooms in.

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <FeatureTypeStyle
3   xsi:schemaLocation="http://www.opengis.net/sld StyledLayerDescriptor.xsd"
4   xmlns="http://www.opengis.net/sld"
5   xmlns:ogc="http://www.opengis.net/ogc"
6   xmlns:xlink="http://www.w3.org/1999/xlink"
7   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
8   <Rule>
9     <MaxScaleDenominator>750000</MaxScaleDenominator>
10    <LineSymbolizer>
11      <Stroke>
12        <SvgParameter name="stroke-width">1</SvgParameter>
13        <SvgParameter name="stroke">#97d397</SvgParameter>
14      </Stroke>
15    </LineSymbolizer>
```

```

16  </Rule>
17 </FeatureTypeStyle>
```

Program 146 - The SLD containing a single rule

You apply the SLD file by decoding it and configuring it on the layer:

```

1 TLcdSLDFeatureTypeStyleDecoder decoder = new TLcdSLDFeatureTypeStyleDecoder();
2 String source = "/path/to/SLD/file.sld";
3 TLcdSLDFeatureTypeStyle sldStyle = decoder.decodeFeatureTypeStyle(source);
4
5 //For Lightspeed layers
6 TLspLayer layer =
7   TLspShapeLayerBuilder.newBuilder().model(roadsModel)
8     .sldStyle(sldStyle)
9     .build();
10
11 //For GXY layers
12 TLcdGXYSLDLayerFactory layerFactory = new TLcdGXYSLDLayerFactory();
13 ILcdGXYLayer gxyLayer =
14   layerFactory.createGXYLayer(roadsModel, Collections.singletonList(sldStyle));
```

Program 147 - Applying the SLD on a layer

24.6 Model query performance notes

The retrieval of the model data during a paint operation goes through the `ILcdModel.query` method. Therefore, the painting performance depends heavily on the performance of that method.

Consult [Section 53.2](#) for guidelines for influencing this performance.

24.7 Other ways to limit data

Apart from specifying a model query configuration on your layer, there are other ways to limit the amount of data loaded in a layer.

24.7.1 Minimum object size

You can specify a minimum object size, in pixels on the screen, for the appearance of objects. For example, if you specify a minimum size of 5, an object is displayed only if its geometry would be about 5x5 pixels large on the screen.

You can achieve an automatic level-of-detail effect in this way: small objects automatically disappear when the user zooms out, but re-appear when the user zooms in.

This is particularly useful for polygon datasets with building or land usage features:



Figure 94 - Left: with minimum pixel size - Right: without minimum pixel size

For more information, see `TLspShapeLayerBuilder.minimumObjectSizeForPainting` or `TLcdGXYLayer.setMinimumObjectSizeForPainting`.



The minimum object size condition is evaluated by the model. As with an OGC condition, the performance depends largely on the model implementation. Database models will evaluate the condition using SQL, WFS clients will send it to the WFS server if the server supports it. SHP models will do a fairly cheap pre-check to avoid loading the entire geometry.

24.7.2 Layer filter

You can specify a layer filter (`ILcdFilter`) that is evaluated on all loaded elements. For more information, see the `TLspLayer.setFilter` and `TLcdGXYLayer.setFilter` reference documentation.

We recommend using `TLcdModelQueryConfiguration` whenever possible, because that is evaluated inside the model, and the model can optimize it greatly. A layer filter can only be applied inside the layer after all elements have been loaded into memory.

24.8 Scales and filters in a 3D view

Unlike a 2D view, a 3D view does not have a single map scale. Each point in the visible area in a 3D view has a distinct scale.

This picture shows the local scale computed at different spots in the view. You can see that the scale near the horizon is vastly different from the scale near the bottom of the screen:



Figure 95 - Local scale at different spots in a 3D view.

If you have a model query configuration on a layer in 3D, the layer will use different OGC conditions for different regions in your view at the same time. Each scale range in your model query configuration is a contiguous geographic area shaped like an arc band:

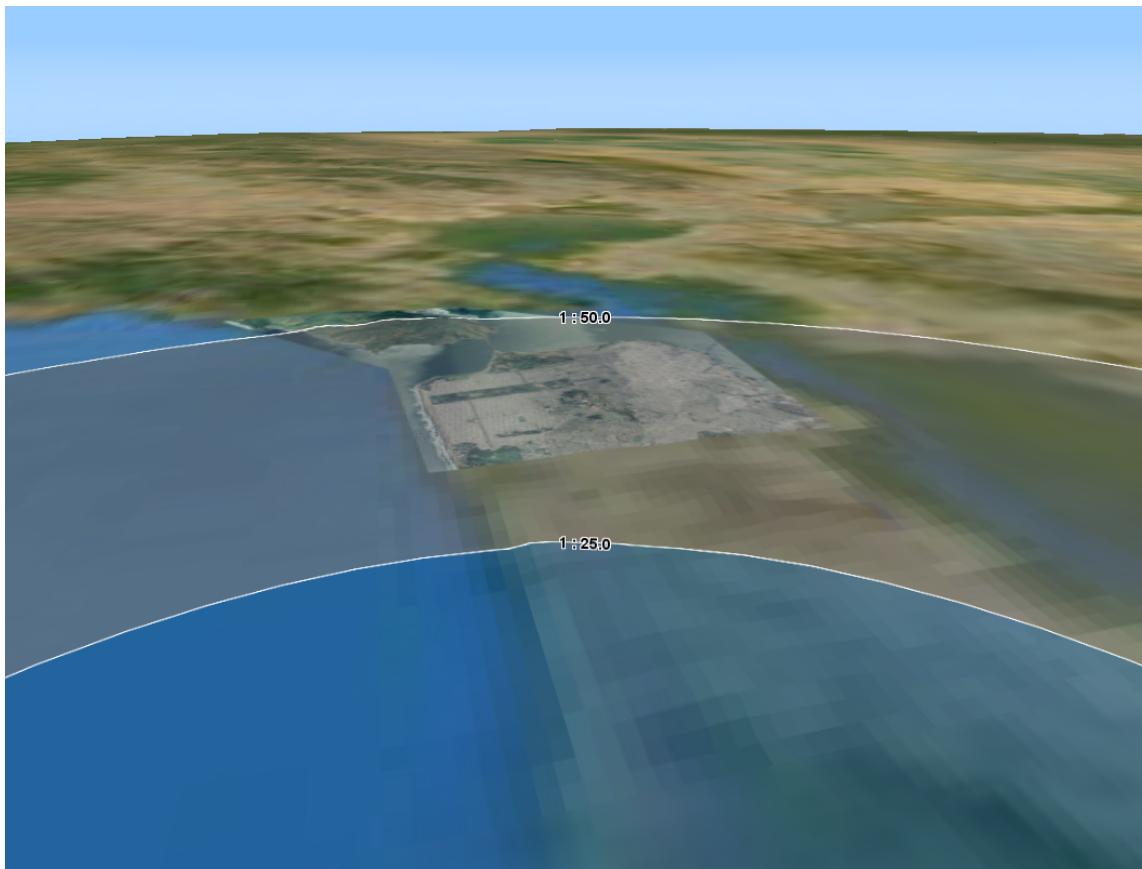


Figure 96 - Arc-shaped area with constant scale in a 3D view.

The layer approximates these areas with a set of tiles, and loads data using the corresponding OGC condition:

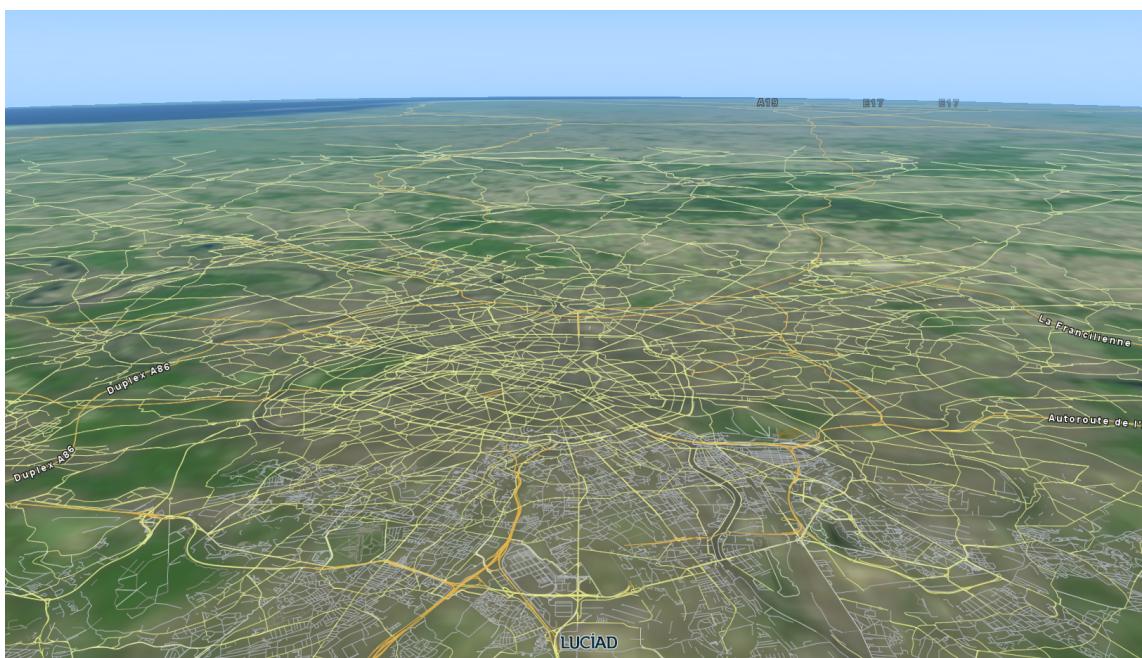


Figure 97 - OpenStreetMap roads data example, illustrating the different conditions used for each tile.

24.9 Visualizing the data on a GXY view

The concept of limiting the data queried by the layer by configuring a `TLcdModelQueryConfiguration` on the layer applies to both GXY layers and Lightspeed layers. This is illustrated in all the examples in this chapter: each example contains code to create a Lightspeed layer and a GXY layer.

CHAPTER 25

Creating and editing domain objects in a Lightspeed view

Domain object editing deals with the modification of the properties of domain objects triggered by user interaction. If a user drags one or more corners of a shape to make the shape bigger, for example, the result of the size modification needs to be stored with the domain object represented by the shape. Dragging a particular object handle to another location may move the object itself to that location.

In addition, the functional domain of LuciadLightspeed editing covers the creation of new domain objects as a result of user interaction. A user could click several points on the screen to create the vertices of a new polygon, for instance.

If you want to let input events trigger the creation or modification of one or more objects painted in the view, you can make use of the LuciadLightspeed editor and controller classes to initiate or modify the object's graphical representation.

This chapter discusses in detail how editing works in a Lightspeed view, how to configure your object layers for editing, what the roles of the various participants in the editing and creation processes are, and how to customize the editing process.

25.1 Working with editors and controllers

25.1.1 What is editing?

In LuciadLightspeed, an editor instance is able to create and edit a certain type of object. To allow users to use those capabilities, it provides sets of object handles. This means that when users start creating a new object, or select an editable object, the associated object editor provides the users with the user interface (UI) elements that allow them to create or edit that object. For example, when a user selects a bounds object (`ILcdBounds`), the corresponding bounds editor (`TLspBoundsEditor`) provides five UI handles: four point handles that allow the user to edit the corner points, and one translation handle that allows the user to translate the bounds object as a whole.

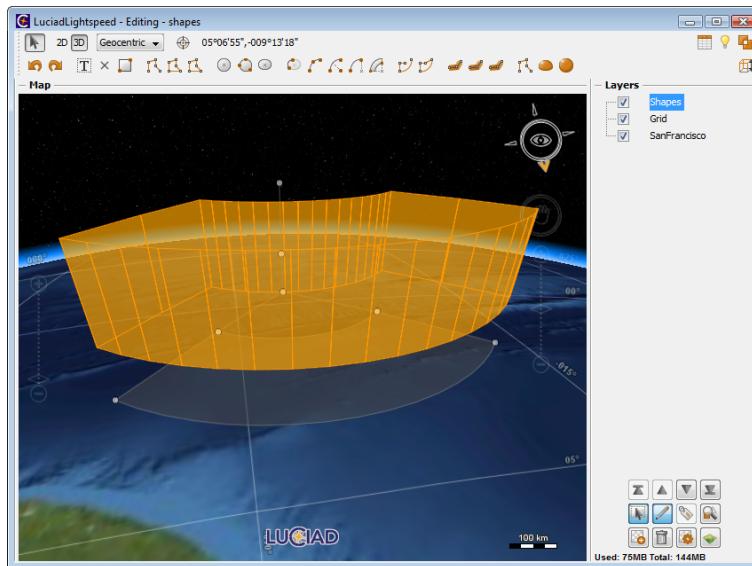


Figure 98 - Shapes can be edited by interacting with edit handles

The controller coordinates the interaction between the object handles, the editor and the domain model. The editor needs to interpret the information stored about the domain object, as well as the information generated from the input event, to initiate or modify the object appropriately.

25.1.2 Making the objects in a layer editable

To make a Lightspeed layer editable, the following steps are required:

- Choose an editor
- Configure your layer for editing
- Register the chosen editor with the correct paint representation
- Set the create or edit controller on the view

Choosing an editor

LuciadLightspeed provides a number of pre-defined editors for all the shapes in the API. These editors implement the basic creation and editing behavior and are sufficient in most cases. To work with shapes such as circles, arcs, polygons, and so on, you can use tailored editors such as `TLspCircleEditor`, `TLspArcEditor`, `TLsp2DPointListEditor` respectively.

Some of the other `ILspEditor` implementations allow you to:

- Define and edit the base shape as well as the minimum width and maximum height of an extruded shape. `TLspExtrudedShapeEditor` takes the editor of the base shape as an argument.
- Use multiple editors in one editor instance: `TLspCompositeEditor`

For a full list and detailed descriptions, see the API reference documentation of the editor classes. Note that most of the editors that edit shape objects also allow the user to translate the shapes.



For convenience, `TLspShapeEditor` aggregates all the editors in the API, making it the preferred editor for a large number of use cases.

In cases where specialized editing behavior is required, you can create your own `ILspEditor` implementations or customize existing implementations. For more information about custom implementations, see [Section 25.4.4](#).

Configuring a layer for editing

To configure a layer as editable, use the method call `layer.setEditable(true)`. This sets the layer's editable flag to `true`. If you are using a layer builder, the layer is automatically configured as editable when you call the methods for registering body and label editors with the layer. For more information about layer builders, see [Section 22.2](#).

Registering an editor with the layer

In the Lightspeed view and layer framework, editors are linked to the paint representations of objects. If you want to add creation and editing capabilities for an object, you need to register an editor with the layer, and link it to the appropriate paint representation.

To register an editor for a `TLspPaintRepresentation` with the layer, either use the `setEditor(TLspPaintRepresentation, ILspEditor)` method on the layer, or use the available layer builder methods for setting body and label editors.

Note that labels can be edited as well, by calling `labelEditor()`.

```

1  return TLspShapeLayerBuilder
2      .newBuilder()
3      .model(aModel)
4      .bodyEditable(true)
5      .build();

```

Program 148 - Making the shape layer builder body editable is enough to add the `TLspShapeEditor` to the layer.

(from `samples/lightspeed/editing/EditableLayerFactory`)

Setting the controller

The editor needs to receive input events from its UI handles. For this, you need to set a create or edit controller on the view. This controller passes the input events on to the editor. To set a `TLspEditController` on the view, for example, call `view.setController(new TLspEditController())`.

25.2 The editing process in a Lightspeed view

The editing process involves four main actors:

- `ILspEditor`
- `ALspEditHandle`
- `TLspEditController`
- `ILspSnapper`

The following diagram shows the flow of interactions between these classes:

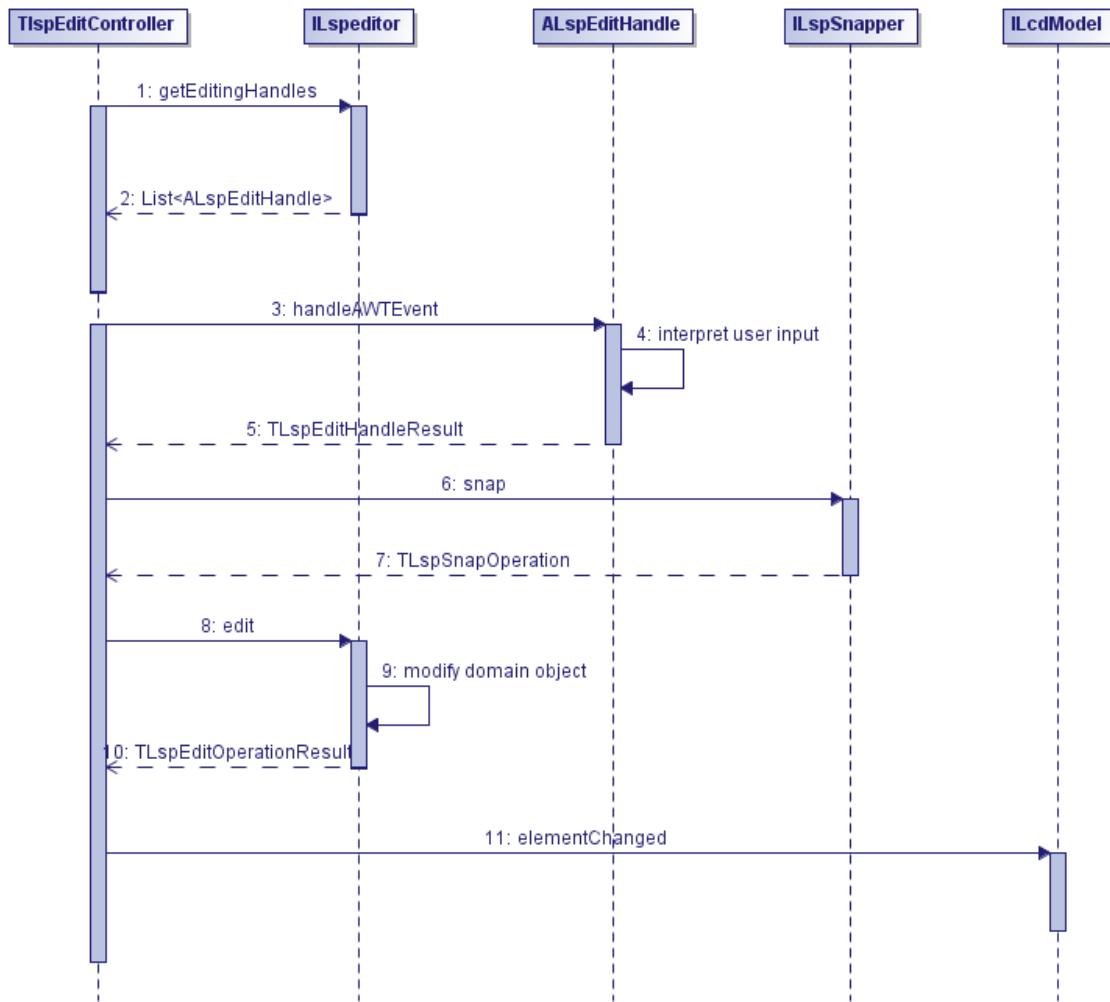


Figure 99 - Sequence diagram illustrating the editing process

Suppose that a user clicks a polygon object with the intention of moving one of the polygon points.

1. When the polygon object is selected, the edit controller requests edit handles for the selected object from the editor set up on the layer for this object type.
2. The editor returns a list of polygon handles to the controller. The handles are displayed on-screen.
3. When the user clicks and drags one of the handles, the edit controller issues a user input event to the touched handle.
4. The touched handle interprets the user input event, and moves to the new position on screen.
5. The handle confirms that it has been moved by returning an edit operation to the edit controller.
6. The edit controller contacts the snapper to check if there is a suitable object in the vicinity that the moved handle can be snapped to. If there is a suitable object, the snapping location is marked by a visual indicator. If the handle continues to move closer to the snapping location, it is snapped.

7. A snap operation containing a list of edit operations is returned to the edit controller.
8. The edit controller confirms that an edit has been performed on the polygon object by sending the edit operation to the editor. The editor modifies the polygon domain object itself, and returns the result of this operation to the edit controller.
9. The edit controller notifies the model that one of its objects has changed.



To deactivate the snapping function while they are dragging a handle around, users must press the CTRL (CMD on Mac OS) key.

The following sections discuss the responsibilities of each participant in the editing process in more detail. Afterwards, the possibilities for customization of this process are explained using the `samples.lightspeed.customization.hippodrome` sample as a guide.

25.2.1 The editor

An `ILspEditor` is the key component of the editing functionality in LuciadLightspeed. It has two main responsibilities:

- **Creating edit handles.** The method `getEditHandles()` creates a list of `ALsEditHandle`s for a given editable domain object. Each edit handle is a small GUI element that users can interact with. It interprets user input and converts it into a higher-level description of the user input, called an edit operation. In the case of a polygon, for instance, an editor typically creates a handle for each vertex. When these handles are dragged around, they report the geographic coordinates to which they have been moved. See [Section 25.2.3](#) for more details.
- **Editing domain objects.** The `edit()` method applies edit operations, represented by the class `TLspEditOperation`, to domain objects. If we resume the example of the polygon, the `edit()` method is responsible for actually setting the point of the polygon to its new coordinates. The edit handle itself only reports that it has been moved; it never modifies the underlying domain object.

25.2.2 The edit controller

The coordination between editors and edit handles is performed by `TLspEditController`. When you activate an edit controller, it retrieves editing candidates, objects that are currently editable, from the view. By default, these correspond to the currently selected objects. For these objects, edit handles are obtained from each corresponding editor. From then on, the controller forwards the input events it receives to the edit handles. The edit operations received from the handles are then routed back to the editor.

The controller is also responsible for visualizing the edit handles. To this effect, it calls the `getHandleGeometry()` method of the handles, and draws the resulting geometry using an internally created layer.

The only aspect of the edit controller you can customize, is the `getEditingCandidates()` method. If you want to customize the edit controllers any further, you may want to create a new `ILspController` implementation. Keep in mind, though, that the interaction with editors relies on a correct setup, so be careful when you decide to take this approach.

25.2.3 The handles

Handles take input events as input and produce edit operations as output. To this effect, handles have a `handleAWTEvent()` method, which takes an `AWTEvent` and returns a `TLspEditHandleResult`. The latter is a container for one or more `TLspEditOperation` objects.

Using the handle API

There are two main categories of handles:

- To edit single domain objects, choose the regular edit handle type `ALspEditHandle`.
- To edit collections of domain objects, start from multi object handles, `ALspMultiObjectHandle`. To learn more about multi object handles, see [Section 25.2.4](#).

Figure 100 shows the main handle implementations available in LuciadLightspeed.

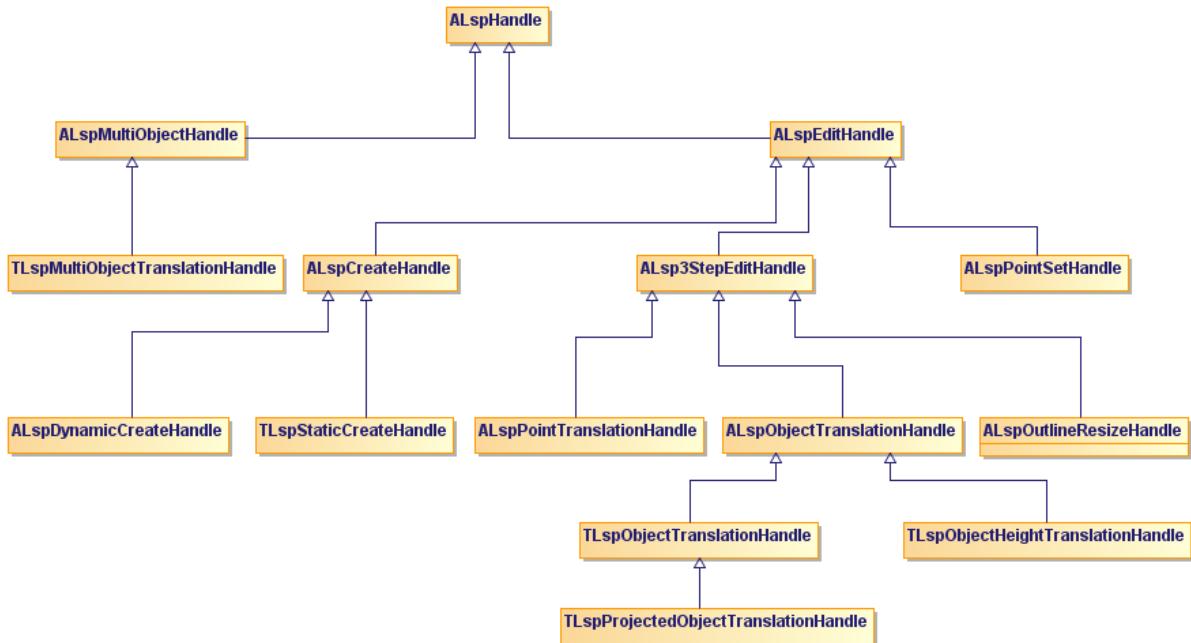


Figure 100 - Main edit handle implementations

The APIs of both handle types are mostly identical. In the next few sections, we focus on the general properties of handles and on the regular edit handles in particular.

The most important handle classes are:

- `TLspPointTranslationHandle` allows the user to drag a point to a new location. The point can be an `ILcdPoint` object, but also a derived property of another object, such as the center point of a circle, or the start and end points of an arc.
- `TLspPointSetHandle` is the counterpart of `TLspPointTranslationHandle`, used for the creation of points. It allows the user to position a point by clicking on the map, instead of by dragging an existing point to a new location.
- `ALspOutlineResizeHandle` can be used for shapes which can be resized by dragging their contours in and out. Examples include the radius of a circle or the width of a buffer.

- `TLspObjectTranslationHandle` allows the user to translate an object. This means moving an object to a new location as a whole by clicking anywhere on the object and then dragging it to the new location.
- `ALspCreateHandle` is a list of edit handles. It is used only when new objects are created. Create handles are discussed in more detail in [Section 25.3](#).

25.2.4 Working with handles

Defining properties

The limited set of edit handles listed in [Section 25.2.3](#) can enable a very wide range of editing functionality. Many edit operations can be expressed as dragging some form of object control point to a new location, for instance. However, the editor obviously needs some way to associate each of its edit handles with such a control point. To this effect, each handle has a set of properties. These properties are key/value pairs that can be set by the editor inside its `getEditHandles()` method.

For instance, an editor may set a value for the key `HANDLE_IDENTIFIER`. This key indicates that it identifies the role of the handle in a set of handles. If the handle concerned is the one that allows users to expand the radius of a circular object, the value that is paired with the handle identifier key could be `RADIUS`.

Properties allow the editor to freely attach any kind of semantic information to a handle. When the handle creates a `TLspEditOperation` object, the latter contains a copy of these properties. This allows the editor to recover this semantic information in its `edit()` method.

Visualizing handles

Most edit handles require some form of visual representation in the view to be useful. A point translation handle, for instance, is typically visualized as a small icon, so that the application user can actually see the control points that are available to manipulate an object. `ALspHandle` uses the `ILspStyler` API to create the visual representation of the handle. Each handle has a `getStyleTargetProviders()` method, which creates the `ALspStyleTargetProvider` objects used to visualize the handles.

`TLspEditController` and `TLspCreateController` internally add edit handles to a layer that is painted in the view. This layer uses a `TLspEditHandleStyler` (which can be accessed via the controller). This styler, in turn, uses the style target providers obtained from the handles. The edit controller makes it possible to access the handle layer in the editors and handles using `TLspEditContext.getHandleContext()`. This makes it for example possible to do `isTouched` queries on the projected base shape of a 3D object.

The `getStyleTargetProviders()` method takes a `TLspHandleGeometryType` as input. This is an enumeration type which splits handle representations into a few broad categories, such as points and visual aid lines. Each of these types has its own style target provider, and `TLspEditHandleStyler` allows you to register a list of styles for each type.

For more information about style target providers, see [Chapter 22](#).

How does a handle become active?

To decide which handle to activate, LuciadLightspeed relies on the concepts of handle activation, priority and focus.

Handle activation Depending on the current view, edit handles may overlap with each other on the screen. In such cases, it is undesirable for user input to be handled by more than one handle simultaneously. To this effect, handles have an activation mechanism, which is implemented by the simple method `ALspHandle.isActive()`.

A handle can choose whether or not it wants to become active when an input event comes in. If the handle does become active, the edit controller only forwards input events to that handle from then on, until the handle advertises that it deactivated.

Handle priorities To make the handle activation mechanism reliable and predictable for the user, each handle has a priority. When no handle is active, the controller forwards input events to all its handles in descending priority order. This means that the highest-priority handle is always the first one that gets a chance to activate itself. `ALspHandle` predefines a number of priority levels you can use as a reference. In addition, most edit handles have an appropriate priority by default. Editors can always choose to modify the priorities when they create handles.

Handle focus Edit handles support a concept of focus, which is closely tied to the activation and priority mechanisms. A handle can request focus if the cursor hovers over it, for example. If it is the highest-priority handle at that location on the screen, it becomes focused. If the mouse is clicked at that same location, the focused handle is also the one that becomes active from then on.

The focus mechanism exists primarily to give visual feedback to the user: the focused handle can be styled differently than the others. To this effect, `TLspEditHandleStyler` has two sets of styles: one for regular handles and one for focused handles. The default behavior is that point handles turn red when they are in focus.

Working with multi object handles

Edit and create handles are created by an `ILspEditor`. Multi object handles, on the other hand, are created by controllers. The edit and create controllers may create handles of their own, which perform operations on multiple objects simultaneously. The most typical example of such an operation is selecting and moving a whole group of objects at once. Multi object handles are represented by the class `ALspMultiObjectHandle` and are created by the `getMultiObjectHandle()` method of the controller. The main available implementation is `TLspMultiObjectTranslationHandle`, which performs the previously mentioned translation of a group of objects.

The multi object handle API differs slightly from the regular handle API because multi object handles work on a collection of objects. Other than that, their functionality is identical to that of other handles. Editors do need to be aware, however, that they may receive edit operations that did not originate from one of their own handles. The discussion of the hippodrome editor below shows how an editor can support the multi-object translation feature.

25.2.5 Snapping

When the `TLspEditController` receives a `TLspEditOperation` from an edit handle, it does not forward it to the `ILspEditor` directly. Instead, the controller has an `ILspSnapper` which receives the edit operation first. The snapper is given the opportunity to modify the edit operation before it is passed on. To this effect, `ILspSnapper` defines a `snap()` method which takes a single `TLspEditOperation` as input and produces zero, one or more new operations as output.

The `ILspSnapper` implementation `TLspPointToPointSnapper` specifically looks for move operations, like a vertex of a polygon being dragged around. When it sees such an operation, it tries to find other domain objects in the vicinity of the dragged point. If the dragged point comes within a certain pixel distance of a nearby vertex of another object, the snapper overrides the incoming edit operation so that its coordinates correspond exactly to this nearby point.

By convention, if the `snap()` method returns `null`, the editor forwards the unmodified incoming edit operation to the editor. Otherwise, it uses the operations returned by the snapper.

25.3 The creation process in a Lightspeed view

Until now, we have only discussed the editing of existing shapes. It is also possible to draw new shapes on the map. The actors in the shape creation process are similar to those required in the shape editing process: editor, controllers and handles. To create new shapes, you need to work with a create controller instead of an edit controller, however, and with create handles instead of editing handles.

The create controller

To draw new shapes on the map, use `TLspCreateController` instead of `TLspEditController`. The inner workings of the create and edit controllers are largely identical, although `ILspEditor` defines a separate `getCreateHandle()` method that is used instead of `getEditHandles()`. Many editors require subtly different handles for editing and creation.

Create handles

`TLspCreateController` invokes the editor's `getCreateHandle()` method instead of the `getEditHandles()` method. The `getCreateHandle()` method returns an `ALspCreateHandle`, which is an extension of the regular `ALspEditHandle`. A create handle is actually a list of `ALspEditHandles`. Since this is a creation process rather than an editing process, the handles are activated in a fixed sequential order to guide a user through the creation of an object. The handles are responsible for initializing the various properties of the object one by one, until the object is ready to be committed to the model.

LuciadLightspeed offers the choice between two types of create handles, depending on whether you know up front how many delegate handles are required to create a shape:

- If you know the number of delegate handles beforehand, use `TLspStaticCreateHandle`. For example, it is possible to draw a circle on a map with two mouse clicks: one that sets the center and another that sets the radius of the circle. In such cases, the editor would use a static create handle containing handles for these two properties.
- If you do not know the number of delegate handles up front, use `ALspDynamicCreateHandle`. A dynamic create handle instantiates new delegate handles on the fly, and continues to do so until some stop condition is met. A polygon, for instance, is typically drawn with a dynamic create handle. The dynamic handle continues to create new point handles, each of which add an extra vertex to the polygon. The stop condition is a double-click, for example, through which users indicates that they have finished drawing the polygon.

25.4 Customizing the editing behavior

This section lists a number of areas in which the editing functionality of Lightspeed views is customizable. This ranges from tweaks to the edit handle visualization to tailor-made editors for custom domain objects.

25.4.1 Visualization and styling of edit handles

Simple tweaks to the handle visualization, such as changes to colors, line widths or icons, can be made by setting different styles on the `TLspEditHandleStyler`. More advanced customization, such as adding additional visual aid lines, can be performed by overriding the `getStyleTargetProviders()` method in the handles and/or extending the handle styler.

25.4.2 Omitting edit handles or adding new ones

It is possible to customize the set of edit handles created by an editor by removing certain handles or by adding new ones. For instance, to constrain a circle editor so that it does not allow users to change the radius of the circle, you could leave out the edit handle responsible for defining the radius. Analogously, to allow users to rotate a polygon around its center, for instance, you could add an extra edit handle to the polygon editor.

The obvious way to approach this is by overriding either `getEditHandles()` or `getCreateHandle()`. The editor implementations supplied with LuciadLightspeed, however, also follow the convention that each edit handle is created by a dedicated protected method, `TLspCircleEditor.createRadiusHandle()` for example. This enables you to override these creation methods individually. You can make these methods return `null` if you want to remove the handle in question, or you can decorate or replace the handle to customize its behavior, to add modifier keys to certain functions for instance.

When customizing the edit handles, it is important to keep in mind that the editor may expect that certain properties are set, to guarantee that a particular edit operation will work. A polygon editor, for instance, expects the handles of the polygon's vertices to have a property linking each handle to its corresponding vertex. The properties set by the default editor implementations are described in the API reference documentation. Each editor has an internal enumeration class named `PropertyKeys`, the values of which are used as the keys for properties. The documentation of each key lists the expected type of the property associated with that key. For non-intrusive changes to edit handles, you should ensure that the editor finds all the properties it expects. If you deviate from these expectations, it is likely that you need to override `ILspEditor.edit()` as well to handle the deviation.

25.4.3 Editing custom domain objects

LuciadLightspeed's `ILcdModel` interface and implementations do not impose any restrictions on the type of elements that are added to them. On the visualization side, the `ILspStyler` API allows developers to convert model elements of any type into `ILcdShape` objects that can be painted in a view. Similarly, the edit and create controllers call on the same styler that is used during painting to convert model elements into editable geometry. Since the editor picks up the same geometry that is drawn in the view, in many cases editing for custom domain objects comes "for free" if you have already implemented visualization support for these objects.

It is only in cases where your domain object cannot be decomposed into one or more existing `ILcdShape` objects, that you will need to develop a fully customized editor. The following section provides more information on how to approach this task.



It is important to note that in order for an object to be editable in a Lightspeed view, it must implement the `ILcdCloneable` interface.

25.4.4 Implementing new editors

This section discusses the implementation of a custom editor using the `samples.lightspeed.customization.hippodrome` sample as a guide. This sample contains a class `HippodromeEditor`, which will be our main focus. The sample builds on the GXY view-based hippodrome sample which is discussed at length in [Chapter 35](#). Please refer to this chapter first for a definition of the hippodrome shape with which we are working.

The new hippodrome editor needs to perform the following tasks:

- Create and provide edit handles for the hippodrome shape
- Use the information provided by the edit controllers to apply edit information to the domain objects
- Support the creation of new hippodrome shapes by providing create handles to the create controller

Creating edit handles

The first task of an editor is to create handles for the domain object being edited. It is generally recommended to reuse the existing `ALspEditHandle` implementations in LuciadLightspeed where possible. `HippodromeEditor` follows this recommendation: it uses `TLspPointTranslationHandle`, `TLspObjectTranslationHandle` and `ALspOutlineResizeHandle`.

Semantic information is attached to each handle by setting properties on it. Like the editors in the LuciadLightspeed API, `HippodromeEditor` defines an enumeration with possible property keys, as shown in [Program 149](#).

```
1 /**
2  * Keys used for properties on edit handles.
3  */
4 public static enum PropertyKeys {
5     /**
6      * Maps to a {@link HandleIdentifier}, which indicates the purpose of the edit handle.
7      */
8     HANDLE_IDENTIFIER,
9 }
10
11 /**
12  * Describes the type of an edit handle created by the enclosing editor implementation.
13  *
14  * @since 2012.0
15  */
16 public static enum HandleIdentifier {
17     /**
18      * Identifies the handle at the hippodrome's start point.
19      */
20     START_POINT,
21     /**
22      * Identifies the handle at the hippodrome's end point.
23      */
24     END_POINT,
25     /**
26      * Identifies the whole-object translation handle.
27      */
28     TRANSLATE,
29     /**
30      * Identifies the hippodrome radius (or width) handle.
31      */
32     RADIUS
33 }
34 }
```

Program 149 - Defining handle property keys and values
(from samples/lightspeed/customization/hippodrome/HippodromeEditor)

The `getEditHandles()` method is implemented as shown in Program 150. It applies the convention of creating each handle through a separate method.

```

1  @Override
2  public List<ALspEditHandle> getEditHandles(TLspEditContext aContext) {
3      // Don't edit if the object is not a hippodrome
4      Object object = aContext.getGeometry();
5      if (!(object instanceof IHippodrome)) {
6          return Collections.emptyList();
7      }
8      final IHippodrome hippodrome = (IHippodrome) object;
9
10     // Create handles and add them to a list
11     ArrayList<ALspEditHandle> handles = new ArrayList<ALspEditHandle>(4);
12
13     ALspEditHandle start = createStartPointHandle(hippodrome, aContext, true);
14     handles.add(start);
15
16     ALspEditHandle end = createEndPointHandle(hippodrome, aContext, true);
17     handles.add(end);
18
19     ALspEditHandle outline = createWidthHandle(hippodrome, true);
20     handles.add(outline);
21
22     ALspEditHandle translate = createTranslationHandle(hippodrome);
23     handles.add(translate);
24
25     return handles;
26 }
```

Program 150 - Creating edit handles for a hippodrome

(from samples/lightspeed/customization/hippodrome/HippodromeEditor)

Program 151 is an example of the creation of a handle through a separate method. It shows the code that creates a point translation handle for the end point of the hippodrome. Other handles are created in a similar fashion.

```

1  private ALspEditHandle createEndPointHandle(final IHippodrome aHippodrome, TLspEditContext
2      aEditContext, boolean aEditing) {
3      final ILcdModelReference modelReference = aEditContext.getObjectContext() .
4          getModelReference();
5      TLspPointTranslationHandle end = new TLspPointTranslationHandle(
6          aHippodrome, aHippodrome.getEndPoint(), modelReference
7      );
8      end.getProperties().put(PropertyKeys.HANDLE_IDENTIFIER, HandleIdentifier.END_POINT);
9      end.setTranslateOnDrag(aEditing);
10     return end;
11 }
```

Program 151 - Creating a point translation handle

(from samples/lightspeed/customization/hippodrome/HippodromeEditor)

Applying edit operations

The second main task of an editor is to apply edit operations to the underlying domain object. Edit operations are represented by the class `TLspEditOperation`. You can use its properties to specify:

- A general indication of what the user is doing with the object: `TLspEditOperationType` describes in general terms what kind of operation the user performed. Possible values include `MOVE`, `INSERT_POINT`, `PROPERTY_CHANGE`. The type may also be `NO_EDIT`, indicating that the domain object does not need to be modified at this point.
- Additional information as a set of properties, in the form of a key/value map. These properties always include all the properties that were set on the handle that triggered the edit operation.

- Details of the edit operation, in the form of an operation descriptor. The descriptor is stored in the operation's properties, using a key which is given by `TLspEditOperationType.getPropertyKey()`. The type of the descriptor depends on the edit operation type. For MOVE operations, for instance, the descriptor is a `TLspMoveDescriptor`. The API reference documentation for each predefined operation type also lists the corresponding operation descriptor class.
- The model reference in which the operation is described. This normally is the reference of the model that contains the domain object.
- An indication of whether the operation is considered complete or not. The `InteractionStatus` indicates whether the edit operation leaves the edited object in a commitable state or not. One of the uses for this property is undo/redo functionality: the edit or create controller only registers an undoable step for operations that are marked as `FINISHED`, not for ones that are marked as `IN_PROGRESS`.

The combined information in `TLspEditOperation` should give the editor sufficient information to make changes to the edited domain object. [Program 152](#) shows the implementation of the `edit()` method for hippodromes.

```

1  private static final String RADIUS_PROPERTY_NAME = "width";
2
3  @Override
4  protected TLspEditOperationResult editImpl(TLspEditOperation aOperation,
5      ELspInteractionStatus aInteractionStatus, TLspEditContext aContext) {
6      Object object = aContext.getGeometry();
7      if (!(object instanceof IHippodrome)) {
8          return TLspEditOperationResult.FAILED;
9      }
10     IHippodrome hippodrome = (IHippodrome) object;
11
12     TLspEditOperationType type = aOperation.getType();
13     if (type == TLspEditOperationType.MOVE) {
14         TLspMoveDescriptor descriptor =
15             (TLspMoveDescriptor) aOperation.getProperties().get(type.getPropertyKey());
16         applyMove(hippodrome, aOperation, descriptor);
17         return TLspEditOperationResult.SUCCESS;
18     } else if (type == TLspEditOperationType.PROPERTY_CHANGE) {
19         TLspPropertyChangeDescriptor descriptor =
20             (TLspPropertyChangeDescriptor) aOperation.getProperties().get(type.getPropertyKey());
21         if (RADIUS_PROPERTY_NAME.equals(descriptor.getPropertyName()) &&
22             descriptor.getNewValue() != null) {
23             hippodrome.setWidth(Math.abs((Double) descriptor.getNewValue()));
24             return TLspEditOperationResult.SUCCESS;
25         }
26     }
27     return TLspEditOperationResult.FAILED;
}

```

Program 152 - Implementing the `edit()` method for hippodromes
 (from `samples/lightspeed/customization/hippodrome/HippodromeEditor`)

The method first performs a quick sanity check by testing if the edited object is effectively a hippodrome. Next, it looks at the edit operation type and extracts the corresponding operation descriptor from the operation's properties. The hippodrome supports two operation types:

- `MOVE` is used for changes to the start and end point of the hippodrome, or for a translation of the object as a whole. These changes are described by a `TLspMoveDescriptor`.
- `PROPERTY_CHANGE` is used for changes to the radius of the hippodrome. Since the radius of the arcs is equal to the width of the hippodrome, it is referred to as the width. This

operation is described by a `TLspPropertyChangeDescriptor`.

The `PROPERTY_CHANGE` case is the simplest: if the property change descriptor reports "width" as the name of the property being changed, the editor knows that the value stored in the descriptor is a double that it can pass in a call to `IHippodrome.setWidth()`.

MOVE operations are dealt with in a separate method, which is shown in Program 153.

```

1  private void applyMove(
2      IHippodrome aHippodrome,
3      TLspEditOperation aOperation,
4      TLspMoveDescriptor aDescriptor
5  ) {
6      ILcdPoint startPoint = aDescriptor.getStartPoint();
7      ILcdPoint targetPoint = aDescriptor.getTargetPoint();
8      HandleIdentifier handleIdentifier = (HandleIdentifier) aOperation.getProperties().get(
9          PropertyKeys.HANDLE_IDENTIFIER
10     );
11
12     if (handleIdentifier == null) {
13         // This can happen for multi-object translation handles, which are created
14         // by the controller rather than the editor.
15         if (startPoint != null) {
16             aHippodrome.translate2D(
17                 targetPoint.getX() - startPoint.getX(),
18                 targetPoint.getY() - startPoint.getY()
19             );
20         }
21     } else {
22         switch (handleIdentifier) {
23             case START_POINT:
24                 aHippodrome.moveReferencePoint(targetPoint, IHippodrome.START_POINT);
25                 break;
26             case END_POINT:
27                 aHippodrome.moveReferencePoint(targetPoint, IHippodrome.END_POINT);
28                 break;
29             case TRANSLATE:
30                 if (startPoint != null) {
31                     aHippodrome.translate2D(
32                         targetPoint.getX() - startPoint.getX(),
33                         targetPoint.getY() - startPoint.getY()
34                     );
35                 }
36                 break;
37             }
38         }
39     }
}

```

Program 153 - Applying a move operation to a hippodrome

(from samples/lightspeed/customization/hippodrome/HippodromeEditor)

To perform the move operation on a hippodrome, the editor first retrieves the properties from the edit operations that allow it to identify which handle triggered the edit operation. Based on these properties, the editor moves either the start point, the end point or the entire hippodrome.



The editor also specifically checks for the case in which properties are missing from the operation. This can happen when the operation was fired by a multi-object handle instead of a regular edit handle. In practice, this means that the user selected and dragged multiple shapes simultaneously, so the editor applies the whole-object translation.

Creating create handles

The hippodrome editor also supports the drawing of new hippodrome objects in the view. Program 154 shows the implementation of the `getCreateHandle()` method.

```

1  @Override
2  public ALspEditHandle getCreateHandle(TLspEditContext aContext) {
3      // Don't edit if the object is not a hippodrome
4      Object object = aContext.getGeometry();
5      if (!(object instanceof IHippodrome)) {
6          return null;
7      }
8      final IHippodrome hippodrome = (IHippodrome) object;
9
10     // We use a static create handle, since we know beforehand
11     // how many handles are needed to initialize the hippodrome
12     Collection<ALspEditHandle> handles = new ArrayList<ALspEditHandle>();
13
14     ALspEditHandle start = createStartPointHandle(hippodrome, aContext, false);
15     handles.add(start);
16
17     ALspEditHandle end = createEndPointHandle(hippodrome, aContext, false);
18     handles.add(end);
19
20     ALspEditHandle width = createWidthHandle(hippodrome, false);
21     handles.add(width);
22
23     return new TLspStaticCreateHandle(hippodrome, handles);
24 }
```

Program 154 - Create handle for a hippodrome
 (from samples/lightspeed/customization/hippodrome/HippodromeEditor)

This method uses a static create handle, because a hippodrome can be drawn with a fixed number of mouse clicks:

- The first click to set the start point
- The second click to set the end point
- The last click to set the width of the hippodrome

Each of these corresponds to an edit handle that is added to the `TLspStaticCreateHandle`. These handles are the same as the ones returned by `getEditHandles()`, with one exception. The first handle, which sets the hippodrome's start point, is created differently in creation mode. Program 155 shows how.

```

1  private ALspEditHandle createStartPointHandle(
2      final IHippodrome aHippodrome,
3      TLspEditContext aContext,
4      boolean aEditing
5  ) {
6      ALspEditHandle start;
7      // When editing, use a TLspPointTranslationHandle. This handle allows a point to
8      // be dragged using the mouse.
9      final ILcdModelReference modelReference = aContext.getObjectContext().getModelReference();
10     if (aEditing) {
11         start = new TLspPointTranslationHandle(
12             aHippodrome, aHippodrome.getStartPoint(), modelReference
13         );
14     }
15     // When creating, use an TLspPointSetHandle instead. This handle allows the
16     // point to be positioned using a mouse click.
17     else {
18         start = new TLspPointSetHandle(aHippodrome, aHippodrome.getStartPoint(), modelReference)
19             {
20                 @Override
21                 protected TLspEditHandleResult createEditHandleResult(ILcdPoint aViewPoint,
```

```

21                                     AWTEvent aOriginalEvent,
22                                     AWTEvent aProcessedEvent,
23                                     ELspInteractionStatus
24                                     aInteractionStatus,
25                                     TLspEditContext aEditContext) {
26
27     // Add operations to move the end point, and initialize the width
28     // This results in a better visualization during creation
29     TLspEditHandleResult editHandleResult = super.createEditHandleResult(aViewPoint,
30                           aOriginalEvent,
31                           aProcessedEvent
32                           ,
33                           aInteractionStatus
34                           ,
35                           aEditContext);
36
37     List<TLspEditOperation> operations = new ArrayList<TLspEditOperation>();
38     for (TLspEditOperation operation : editHandleResult.getEditOperations()) {
39         // Add the original operation
40         operations.add(operation);
41
42         if (operation.getType() == TLspEditOperationType.MOVE) {
43             String movePropertyKey = TLspEditOperationType.MOVE.getPropertyKey();
44             TLspMoveDescriptor moveDescriptor =
45                 (TLspMoveDescriptor) operation.getProperties().get(movePropertyKey);
46
47             // Add an operation to move the end point to the same location
48             Map<Object, Object> properties1 = new HashMap<Object, Object>();
49             properties1.put(PropertyKeys.HANDLE_IDENTIFIER, HandleIdentifier.END_POINT);
50             properties1.put(movePropertyKey, moveDescriptor);
51             operations.add(new TLspEditOperation(TLspEditOperationType.MOVE, properties1));
52
53             // Add an operation to set the width to an initial value
54             Map<Object, Object> properties2 = new HashMap<Object, Object>();
55             properties2.put(PropertyKeys.HANDLE_IDENTIFIER, HandleIdentifier.RADIUS);
56             properties2.put(TLspEditOperationType.PROPERTY_CHANGE.getPropertyKey(),
57                             new TLspPropertyChangeDescriptor<Double>(RADIUS_PROPERTY_NAME,
58                                             aHippodrome.getWidth(),
59                                             0.001)
60             );
61             operations.add(new TLspEditOperation(TLspEditOperationType.PROPERTY_CHANGE,
62                     properties2));
63         }
64     }
65
66     // Return a new handle result with the extra operations
67     return new TLspEditHandleResult(operations,
68                                     editHandleResult.getProcessedEvent(),
69                                     editHandleResult.getInteractionStatus());
70 }
71
72 }
73 start.getProperties().put(PropertyKeys.HANDLE_IDENTIFIER, HandleIdentifier.START_POINT);
74
75
76
77
78
79
79 }
```

Program 155 - Differentiating between editing and creation modes
 (from samples/lightspeed/customization/hippodrome/HippodromeEditor)

To create a start handle for hippodrome editing purposes, a standard `TLspPointTranslationHandle` is used. This handle allows the user to drag an existing point to a new location. However, when a new hippodrome is being drawn in creation mode, the start point has not been defined yet. Therefore, an `TLspPointSetHandle` is used instead of an `TLspPointTranslationHandle` for creation purposes. An `TLspPointSetHandle` allows the user to position the point by clicking on the map. When the user places the start point, the editor puts the end point at the same coordinates. This means that the subsequent end point edit handle does not need to be an `TLspPointSetHandle`.

CHAPTER 26

Managing your GUI and controllers in a Lightspeed view

This chapter discusses several topics that allow you to expand your application with specific options for user navigation and interaction with the GUI.

26.1 Working with GUI events

In most applications, you need to handle GUI input events. GUI input events are typically end user actions such as a click on one of the buttons in a toolbar. To define a GUI event, you typically use an `ILcdAction`. By using an `ILcdAction`, the functionality of a GUI component is separated from the effect it has on the application.

Take, for example, a button that has an `ILcdAction` associated to it. When the button is pressed by the user, the button calls its `actionPerformed()` method. The method `actionPerformed` has one argument that specifies the event (user interaction) invoked on the button. It is then the `ILcdAction` that defines the effect of this user interaction on the application. As you can see, the button has no direct effect on the application. It merely serves as a messenger that forwards user interaction to the application. The same behavior applies to a menu item in a menu bar.

26.1.1 What is an `ILcdAction`?

An `ILcdAction` is an extension of `java.awt.event.ActionListener` that applies the Command design pattern (objects are modeled as executable commands). It is very similar to the Swing `javax.swing.Action` interface. To create a Swing Action from a given `ILcdAction` you simply create an instance of `TLcdSWAction`, which is a Swing wrapper around an `ILcdAction`.

The interface `ILcdAction` applies the Listener pattern in collaboration with `java.beans.PropertyChangeListener`. A `PropertyChangeListener` accepts `java.beans.PropertyChangeEvent` objects from the `ILcdAction` on which it is registered. For more information on listening to changes, refer to [Section 8.2](#).

26.1.2 Available implementations of `ILcdAction`

Currently, LuciadLightspeed offers the following main `ILcdAction` implementations out of the box for Lightspeed views:

- `TLspSetControllerAction`: sets the associated controller on the view.
- `TLcdOpenAction`: loads an `ILcdModel` from a local or distant source and sends events to its listeners, notifying them that a model has been created.
- `TLcdSaveAction`: saves an `ILcdModel` to disk.

See the API reference information for more details. For more information about implementing an `ILcdAction`, see [Section 37.1](#).

26.2 Using and customizing controllers

A controller allows interaction with a view. It translates the low-level user interactions into higher-level operations on `ILspPainter` objects, `ILspEditor` objects, `ILspInteractivePaintableLayer` objects, and so on. The following sections describe how to use and customize the main implementations of `ILspController`.

26.2.1 Chaining controllers

All implementations of `ILspController` are chainable. The concept of chaining controllers is also used in GXY views for touch-based controllers. Events come in at the top of the chain, and each controller decides whether or not it handles the event. If the controller does not handle the event, or only handles it partially, the event is passed to the next controller in the chain.

Controllers decide whether or not to handle an event by checking their event filters. Each instance of `ILspController` can be configured with a filter for `AWTEvents`. Events that do not pass the filter are delegated directly to the next controller in the chain. Events that pass the filter are handled by the controller implementation, but may still be delegated if the controller has no use for them (based on the current state for instance). The class `TLcdAWTEventFilterBuilder` can be used to easily create complex filters.

The combination of chaining and filtering possibilities makes it easy to create complex controllers. For example, in [Program 156](#), a number of navigation controllers are created, along with some event filters. Those navigation controllers are then chained together. The result is a single navigation controller that offers a wide range of navigation functions.

```

1  /*
2   * The default navigation controller: fly-to, pan, zoom and rotate.
3   */
4  public static ALspController createNavigationController() {
5      // First we create the controllers we want to chain.
6      ALspController zoomToController = createZoomToController();
7      ALspController panController = createPanController();
8      ALspController zoomController = createZoomController();
9      ALspController rotateController = createRotateController();
10
11     //Chain the controllers together, events will be offered to the first and trickle down.
12     zoomToController.appendController(panController);
13     zoomToController.appendController(zoomController);
14     zoomToController.appendController(rotateController);
15
16     //Set general properties on the top of the chain.
17     zoomToController.setIcon(TLcdIconFactory.create(TLcdIconFactory.HAND_ICON));
18     zoomToController.setShortDescription(
19         "<html><p>Navigate:</p><p><b>Left mouse</b>: <ul><li>Drag: pan</li>" +
20         "<li>Double click: fly to</li></ul></p><p><b>Mouse wheel</b>: zoom</p>" +
21         "<p><b>Right mouse</b>: rotate</p></html>"
22     );
23
24     return zoomToController;
25 }
```

```

26
27  /*
28   * Fly-to controller with left mouse button filter. This controller will only use double
29   * click events, so in combination with the applied filter, only left mouse double
30   * clicks or right mouse double clicks will trigger a fly-to.
31   * Left mouse zooms in and right mouse zooms out.
32   */
33 private static ALspController createZoomToController() {
34     TLspZoomToController zoomToController = new TLspZoomToController();
35     zoomToController.setAWTFilter(TLcdAWTEventFilterBuilder.newBuilder().
36       leftMouseButton().or().rightMouseButton().build());
37     return zoomToController;
38   }
39
40  /*
41   * Panning is the backup left mouse button behaviour (if editing is not possible), as well
42   * as the default action mapped to the middle mouse button.
43   */
44 public static ALspController createPanController() {
45   // Use a pan controller that consumes events during panning, e.g. mouse wheel events.
46   TLspPanController panController = new GreedyPanController();
47   panController.setEnableInertia(true);
48   panController.setAWTFilter(TLcdAWTEventFilterBuilder.newBuilder().
49     leftMouseButton().or().
50                               middleMouseButton().or().
51                               mouseWheelFilter().build());
52   return panController;
53 }
54
55  /*
56   * Zooming is the default action mapped to the mouse-wheel.
57   */
58 public static ALspController createZoomController() {
59   TLspZoomController zoomController = new TLspZoomController();
60   zoomController.setAWTFilter(TLcdAWTEventFilterBuilder.newBuilder().
61     mouseWheelFilter().build());
62   return zoomController;
63 }
64
65  /*
66   * Rotating is the default action mapped to the right mouse button.
67   */
68 public static ALspController createRotateController() {
69   TLspRotateController rotateController = new TLspRotateController();
70   rotateController.setAWTFilter(TLcdAWTEventFilterBuilder.newBuilder().
71     rightMouseButton().build());
72   return rotateController;
73 }
74
75 public static TLspRecenterProjectionController createRecenterProjectionController() {
76   final TLspRecenterProjectionController recenterController = new
77   TLspRecenterProjectionController();
78   recenterController.setAWTFilter(TLcdAWTEventFilterBuilder.newBuilder().leftMouseButton().
79     build());
80   ALspController navigationController = createNavigationController();
81   recenterController.appendController(navigationController);
82   return recenterController;
83 }
84
85 public static TLspRulerController createRulerController(ILcdUndoableListener aListener) {
86   TLspRulerController ruler = new RulerControllerWithPanel();
87   ruler.addUndoableListener(aListener);
88   ruler.setAWTFilter(TLcdAWTEventFilterBuilder.newBuilder().leftMouseButton().or().
89     rightMouseButton().or().keyEvents().build());
90   ruler.appendController(createNavigationController());
91   return ruler;
92 }
```

Program 156 - Creating a composite navigation controller.
 (from samples/lightspeed/common/controller/ControllerFactory)

26.2.2 Customizing object selection with `TLspSelectController`

`TLspSelectController` allows a user to select objects in an `ILspView` using the mouse. Changes to the selection can be made by clicking on an object in the view. Both the objects themselves and their labels can be selected. Using modifier keys such as **Shift**, **Ctrl** and **Alt** allow you to further customize the behavior.

When the user clicks the mouse, the controller looks for objects that the mouse pointer *touches*. Every visible and selectable instance of `ILspInteractivePaintableLayer` in the view is queried for object representations touched by the mouse pointer. This is indicated by the method `query` in the layer implementations. The object corresponding to the topmost candidate, the one actually visible where the mouse is clicked, is then selected.

When a view uses a `TLspViewXYZWorldTransformation2D`, it is also possible to drag a rectangle around the objects to select them. When a mouse drag is detected, the select controller normally delegates the incoming events to the next controller, typically a pan controller. When the user presses the **Shift** key however, the mouse drag results in a selection rectangle. Typically, all objects whose representation is entirely enclosed by the rectangle are selected.

Based on the manner of input (mouse click or rectangle dragged) an instance of `ALspSelectInput` is created (`TLspSelectPointInput` and `TLspSelectRectangleInput` respectively). This object represents all information in view coordinates, necessary to process selection.

The select controller allows a lot of customization. The next sections elaborate on some methods that can be overridden, and finally it is explained how to customize the select controller model used by the controller.

Using the `selectWhat` method

The method `selectWhat` of a `TLspSelectController` controls which representations of objects are considered selection candidates when a user performs a selection action on screen. It returns a set of `TLspPaintRepresentation` objects.

For example, if the set contains only `TLspPaintRepresentation.BODY`, this means that for an object to be considered a selection candidate, its body representation must be touched by the cursor.

Using the `selectChoice` method

When the select controller interacts with the view, there typically are several candidates for selection. The method `selectChoice` controls how the candidates are selected. It returns an instance of `TLspSelectChoice`, an extensible enumeration that by default offers the following options

- **DEFAULT:** when the select input is an instance of `TLspSelectPointInput`, only the topmost object is considered a valid selection candidate. Otherwise all selection candidates are valid.
- **CHOOSE:** the user is presented with the list of selection candidates and can then pick one or more of them for actual selection. By default the candidates are shown in a popup menu. This allows users to select an object covered by another object.

Using the `selectMode` method

The method `selectMode` of a `TLspSelectController` controls what to do with the remaining selection candidates. It returns an instance of `TLspSelectMode` an extensible enumeration with the following default options:

- REPLACE the current selection is replaced by the new selection. Before selection is applied, the existing selections are cleared in all layers in the relevant view. Next, the candidates for selection are selected.
- ADD the candidates for the selection are added to the existing selection.
- REMOVE The candidates for the selection are removed from the current selection.
- INVERT The current selection state of the selection candidates is toggled: those that were selected are deselected, those that were deselected are selected.
- NO_CHANGE No changes are made to the selection.

See [Section 26.2.4](#) for an example of a customized select mode.

Other configuration options for `TLspSelectController`

LuciadLightspeed provides a number of setters that allow you to further customize the controller:

- `setSelectControllerModel` allows for customization of the selection-related logic by setting an (extension of) `TLspSelectControllerModel`. See [26.2.3](#) for more information.
- `setDoubleClickAction` allows you to set an action that will be triggered when a mouse double-click is detected. In addition to selection behavior, the `TLspSelectController` class also offers some functionality to trigger actions that are tightly integrated with the selection behavior. You can use this method to let a double-click trigger an action. Since a double-click event also triggers selection, this is useful for actions closely related to the selected state of an object, like the display of information about the selected object.
- `setContextAction` allows you to set an action that will be triggered when the requirements for a context action are met. It is typically used for right mouse button clicks, to display a context menu for the selected object, for instance. This method is similar to the `setDoubleClickAction` method.

26.2.3 Customizing the selection logic with `TLspSelectControllerModel`

A `TLspSelectControllerModel` contains the logic for selecting objects in an `ILspView`.

Default object selection logic

The main entry point in this class is the method `select`. This method invokes, in the following order, the two methods below:

1. `selectionCandidates`: retrieves a list of objects based on the `select` input and the set of relevant `TLspPaintRepresentation` objects. Whether only one or all possible selection candidates are returned is controlled by a flag. For instance, when a point is passed and the `select` choice is `CHOOSE`, all objects under the point are valid selection candidates. When the `select` choice is `DEFAULT`, only the topmost object is returned.
2. `applySelection`: updates the state of the selection, respecting the selection modes and the candidates involved. This method delegates to the `ILspSelectionCandidateHandler` associated with the passed `select` choice.

Subclasses can override these methods to alter their behavior. It is often useful to invoke the method of the super class with modified parameters.

Configuring the default selection controller model

To customize the behavior of the selection controller model, you can change its properties:

- `setSelectionCandidateHandlerFor`: sets the handler responsible for applying selection for the given `TLspSelectChoice`. By default, a `TLspPopupMenuSelectionCandidateHandler` is used for `CHOOSE` and a `TLspDefaultSelectionCandidateHandler` is used for `DEFAULT`.
- `setSensitivity`: sets the selection sensitivity. This is the maximum distance at which the representation of an object can be removed from a selection point, such as the position of the mouse cursor, and still be considered a valid selection candidate. The distance is expressed in pixels. Tweaking the selection sensitivity is especially useful when you are using less exact input methods like a touch screen.

26.2.4 Custom selection examples

In [Program 157](#), the method `selectMode` is overridden to customize the behavior of the selection controller. A left-click adds objects to the selection, while a right-click removes the objects from the selection.

```

1  @Override
2  protected TLspSelectMode selectMode(MouseEvent aMouseEvent,
3      ALspSelectInput aInput,
4      Set<TLspPaintRepresentation> aRepresentations,
5      TLspSelectChoice aChoice) {
6      TLspSelectMode mode;
7      // Add for left mouse button, remove for right mouse button.
8      if (SwingUtilities.isLeftMouseButton(aMouseEvent)) {
9          mode = TLspSelectMode.ADD;
10     } else if (SwingUtilities.isRightMouseButton(aMouseEvent)) {
11         mode = TLspSelectMode.REMOVE;
12     } else {
13         mode = TLspSelectMode.NO_CHANGE;
14     }
15     return mode;
16 }
```

Program 157 - Overriding `selectHowMode`
(from

`samples/lightspeed/customization/selection/CustomSelectController`)

[Program 158](#) shows how to set a context action on a select controller. The action consists of showing a pop-up menu containing an action to fly to the current selection in the view.

```

1  ILcdAction[] actions = new ILcdAction[1];
2  actions[0] = new FlyToSelectionAction(aView);
3  setContextAction(new ShowPopupAction(actions, ((ILspAWTView) aView).getHostComponent()))
;
```

Program 158 - Set a context action
(from

`samples/lightspeed/customization/selection/CustomSelectController`)

26.2.5 Customizing object editing and object creation

After activation, the `TLspEditController` is responsible for setting up a state in which selected objects can be edited. The editing process and the role of `TLspEditController` in this process is explained in detail in [Chapter 25](#). Refer to this chapter to find out more about

customizing the editing of objects.

The `TLspCreateController` is used to create new objects in a specific layer. Just like the edit controller, the create controller creates a setup in which an object can be initialized.

After activation, the create controller performs the following steps:

1. Call the `create` method on the `ALspCreateControllerModel` that is associated with the controller. This method creates the actual object that will be initialized later on by the editor.
2. Determine the layer the object should be added to, by calling the `getLayer` method on the controller model.
3. Query the editor associated to the layer for the create handles that serve as UI elements to initialize the object.
4. Pass all further input events to the create handles, and leave interpretation up to the editor, until the editor decides that the creation process is finished.

To customize the `TLspCreateController`, you can:

- Set an action that is performed when the creation process is finished, via the `setActionToTriggerAfterCommit` property setting: this property setting sets the action that is performed when the creation process is finished.
- Let the create controller use a different `ALspCreateControllerModel`. You can customize the create controller model by creating a new implementation of the interface, and providing the implementation in the create controller's constructor. See the API reference documentation for more information. A useful example is available in the LuciadLightspeed samples: `light-speed.customization.hippodrome.CreateControllerModelHippodrome`.

26.2.6 Customizing ruler measurements with `TLspRulerController`

You can use `TLspRulerController` to draw polylines in a Lightspeed view and retrieve measurement information about them. The measurement information is displayed on a label with the resulting polyline. The polylines can be edited after creation. As such, `TLspRulerController` is very similar to the `TLspCreateController` and `TLspEditController`.

The created polylines are instances of `ALspRulerMeasurement`. This class exposes the distances (in meters) and azimuths (in degrees) calculated by the ruler controller. You can access the last measurement in the controller itself, using `getCurrentMeasurement`. You can also register a change listener to be notified when the current polyline changes.

Configuring and customizing `TLspRulerController`

`TLspRulerController` offers a number of configuration options. The following methods can be used to configure the behavior of the `TLspRulerController`:

- `setMeasureMode`: sets the measurement mode, which controls how the distances and azimuths are calculated. It also changes the shape of the polylines. By default, the following measurement modes are available:
 - `MEASURE_GEOGRAPHIC`: the distance is calculated along a geodetic line, constituting the shortest path between two points. The azimuth is calculated as the forward azimuth between the two points.

- `MEASURE_RHUMB`: the distance is calculated along a rhumbline, the line between two points with constant compass heading. The azimuth is set to that heading.
- `MEASURE_CARTESIAN`: the distance is evaluated as the Cartesian distance between points, in the world reference used by the view. This mode is especially useful in a 3D geocentric reference, where it can be used to evaluate straight line distances between points on the terrain. In 2D, this mode is of limited use, because most projections severely distort those distances.
- `setKeepLayer`: the measurements are displayed in a dedicated layer, and are normally no longer visible once the controller is deactivated. To ensure that the layer is kept, and that measurements remain visible even if another controller has become active, set this property to `true`.
- `setKeepMeasurements`: normally, when a new measurement is started, the existing measurement is removed. Setting this property to `true` allows you to keep multiple measurements.
- `setLineStyler`: sets the styler that controls the appearance of the polylines themselves.
- `setDisplayEqualDistanceCircles`: if this property is `true`, equal distance circles are shown when you create or modify a measurement. This is only available in geodetic mode.
- `setCircleStyler`: sets the styler that controls the appearance of the equal distance circles.
- `setLabelStyler`: sets the styler that controls the appearance and content of the labels. Note that support is available to label each segment, as well as the entire polyline. A useful class that offers a lot of functionality is the `TLspRulerLabelStyler`.

LuciadLightspeed sample `lightspeed.ruler` demonstrates the configuration and use of a ruler controller.

It is possible to customize the ruler controller at a deeper level still. You can customize the distance and azimuth calculations used by the ruler controller by overriding the methods `calculateAzimuth` or `calculateDistance`.

26.2.7 Visually comparing layers

LuciadLightspeed offers the following controllers to visually compare layers:

- `TLspSwipeController` compares two sets of layers as shown in Figure 101. The swipe controller splits the view with a vertical or horizontal swipe line. The first set of layers is shown on one side of the swipe line, the second set of layers on the other side. To reveal a larger portion of one of the two layer sets, drag the swipe line away from that layer set. When the controller is initialized, a vertical swipe line divides the screen in two equal halves. You can change the position and orientation of the line interactively by dragging it vertically or horizontally.



Figure 101 - Using the swipe controller to compare two satellite images in the lightspeed.imaging.multispectral sample

- `TLspFlickerController` compares two or more sets of layers by toggling layer visibility. When the controller is activated, the first set of layers is visible, while all other sets are made invisible. By clicking the mouse, you reveal the next set of layers and hide the others. The visibility toggling is optimized for speed to allow visual assessment of differences between layers.



Flickering is supported for more than two sets of layers.

- `TLspPortholeController` compares two sets of layers by cutting a hole in one layer set to expose a part of the other layer set. When the controller is activated, you will see that the first set of selected layers covers the view, but that it also has a hole that exposes the second selection of layers in a small square region around your mouse position. By moving the mouse, you move the hole around. By pressing the Shift key and scrolling the mouse wheel, you change the size of the porthole.

If you just want to peel off layers in the hole around your mouse position, select only those layers that you want to cut through in the first set of layers. You can leave the second set of layers empty. As a result, the first set of layers will be peeled off in the porthole but is still visible around the porthole. All other layers are still visible in the porthole.

All controllers require you to specify which layers must be compared. To do so, call the `setLayers` method. You can also use these controllers programmatically, or implement your own controller that works with key strokes or touch events, for example.

The Lightspeed decoder samples and the `lightspeed.imaging.multispectral` sample demonstrate all visual inspection controllers.

26.2.8 Using touch controllers

If you are using a touch-based input device, you can make use of LuciadLightspeed's built-in touch controllers. These controllers are designed for touch input specifically. They provide visual and touch-based alternatives for the traditional mouse and keyboard input, and provide multi-touch input.

The main touch implementations are:

- `TLspTouchSelectEditController`: selects and edits objects in an `ILspView`. It uses the same select controller model as used by `TLspSelectController`.
- `TLspTouchNavigateController`: pans, zooms and rotates the `ILspView`.
- `TLspTouchCreateController`: creates objects on the view.

Refer to Section [Section 37.3](#) for more information on the touch input device support in LuciadLightspeed.

26.3 Creating a custom controller

This section provides a number of pointers that help you create a custom controller from scratch. For an illustration of how to create a custom controller, see the sample `lightspeed.customization.controller`.

26.3.1 Implementing a chainable controller

Keep the following pointers in mind when you implement a chainable controller:

- Take care when you implement `handleAWTEventImpl`. Events that are handled by one controller should not be passed on to other controllers. Therefore, let the method return null if the controller handles the event. In the other case, when a controller does not use an event, you should let the method return the event so that the next controller gets a chance to handle it.
- Some events are tightly coupled, and should be handled by the same controller. A typical example is a mouse-pressed event, followed by a couple of mouse-dragged events, and finally, a released event. It would be very strange if one controller handled the drag events, while another controller handled the pressed and released events. To let one controller deal with this sequence of events, you can let a controller hold on to a pressed event, and only delegate to the next controller if it also receives a dragged event. The select controller described in [Section 26.2.2](#) demonstrates this behavior.

26.3.2 Creating a custom touch controller

To let you create touch controllers easily, LuciadLightspeed offers `ALspTouchController`, a base class that simplifies the process of writing your own touch controllers. This base class provides three abstract methods that allow you to indicate what points will be handled by the controller, and how to react to changes at those points:

- `touchPointAvailable`: called each time a new touch point becomes available. It determines which touch points are handled by the controller.
- `touchPointMoved`: called whenever one of the tracked touch points has moved. It allows you to stop tracking certain points.
- `touchPointWithDrawn`: called each time a touch point is no longer available. Similar to the `touchPointMoved` method, it allows you to stop the tracking of certain points by this controller.

If your custom controller is tracking touch points, you have a number of utility methods at your disposal to retrieve the original, previous, and current locations of the tracked points. See the API reference documentation for more information about `ALspTouchController`.

26.3.3 Creating a tooltip controller

The creation of a custom tooltip controller is demonstrated in the `lightspeed.customization.controller` sample. This controller shows an information panel when hovering over an object.

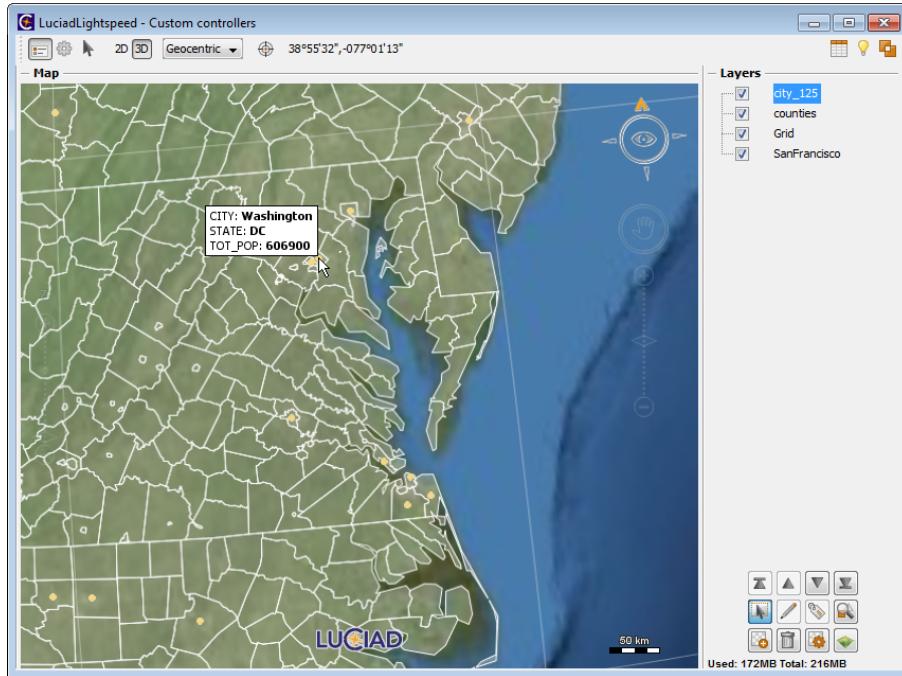


Figure 102 - The `lightspeed.customization.controller` sample

Program 159 shows how the tooltip controller is implemented in general, and the `handleAWTEventImpl` method in particular. On a mouse-moved or mouse-dragged event, the tooltip panel is either moved, or when the mouse cursor does not touch any object, removed. A mouse-exited event removes the tooltip. No events are consumed, because they may still be of use to other controllers. If a pan controller has been included in the chain, for example, it will still receive mouse-dragged events.

```

1  @Override
2  public AWTEvent handleAWTEventImpl(AWTEvent aAWTEvent) {
3      // Handle mouse events only
4      if (aAWTEvent instanceof MouseEvent) {
5          MouseEvent event = (MouseEvent) aAWTEvent;
6
7          if (event.getID() == MouseEvent.MOUSE_MOVED || event.getID() == MouseEvent.MOUSE_DRAGGED)
8              {
9                  // move the information panel to the mouse cursor
10                 movePanel(event.getX(), event.getY());
11             } else if (event.getID() == MouseEvent.MOUSE_EXITED) {
12                 // remove the information panel if the mouse is outside the view
13                 removePanel();
14             }
15         }
16
17         // Do not consume events, in order to make it possible for other controllers to use them.
18         return aAWTEvent;
19     }

```

Program 159 - Implementation of the custom tooltip controller

(from samples/lightspeed/customization/controller/InformationPanelController)

26.4 Handling non-standard input

Non-standard input is handled in the same way in a GXY view and a Lightspeed view. For more information, see [Section 37.3](#). That section explains how you can dispatch custom `TLcdTouchEvent` instances and other input events into your LuciadLightspeed application. It also contains the necessary information for making your custom touch device work with the existing LuciadLightspeed touch controllers.

The creation of `TLcdTouchEvents` is demonstrated in the samples.lightspeed.touch.touchEvents sample. In this sample, touch hardware is simulated by an object that generates a stream of hardware events. These hardware events are then converted to `TLcdTouchEvents` and dispatched.

26.5 Adding undo/redo support

The `ILcdUndoable` interface provides support to reverse the effect of an action. A class that wants its effects to be reversible should provide methods to register an `ILcdUndoableListener`. Whenever an `ILcdUndoable` object is created by the class, the listener gets notified of this object and can react to it in an appropriate way. One implementation of `ILcdUndoableListener` is `TLcdUndoManager`. This class collects all `ILcdUndoable` objects of which it is notified and provides methods to undo and redo these undoable objects in the correct order. The following sections describe the steps for adding undo support to your application in more detail. For more information on listening to changes, refer to [Section 8.2](#).

26.5.1 Adding undo/redo capabilities to your application

To add undo/redo support to your application, you should first create a `TLcdUndoManager`. You can then add this `TLcdUndoManager` as a listener to the appropriate class(es). The `TLcdUndoAction` and `TLcdRedoAction` can then be used to interact with the `TLcdUndoManager`.

26.5.2 Making graphical edits undoable

`TLspEditController` and `TLspCreateController` provide undo/redo support for the editing of domain objects. However, because the actual modification of the domain objects is delegated to `ILspEditor` instances, the creation of the `ILcdUndoable` objects must be delegated to them as well. In order to let the controllers capture these `ILcdUndoable` objects, it needs to attach itself as a listener to the `ILspEditor`. That is why the `ILspEditor` implementation needs to implement `ILcdUndoableSource` as well if the changes made by the `ILspEditor` need to be reversible. This interface allows `ILcdUndoableListener` objects to be attached.

For convenience, LuciadLightspeed provides a full implementation of `ILcdUndoableSource` in the abstract class `ALspEditor`. Hence, all editors derived from this abstract class automatically provide full undo/redo support.

[Section 7.1](#) shows how the edit controller can be linked to undo/redo actions on a toolbar.

CHAPTER 27

Retrieving terrain data from a Light-speed view

As discussed in [Chapter 22](#), any Lightspeed view has built-in support for 3D terrain data. The `TLspRasterLayer` class takes care of many of the visualization aspects of 3D terrain, but many applications need to perform various queries or analysis tasks on the terrain data. This chapter describes the tools that are available for this in the LuciadLightspeed API. These tools are bundled in the `ILspTerrainSupport`, an instance of which can be obtained from the view's `TLspViewServices`.

27.1 Querying terrain data

To obtain the terrain elevation for a given point in the world, `ILspTerrainSupport` provides two different methods: `getViewDependentHeightProvider()` and `getModelHeightProvider()`. Both of these methods return an `ILcdHeightProvider`, but the request method determines what characteristics it has.

The view-dependent provider returns height values that correspond to the terrain data as it is currently being rendered in the view. This means that the view-dependent height provider never causes additional height data to be loaded from the underlying elevation model(s), from disk or from the network. On the other hand, this also implies that the resolution or accuracy of the returned values is dependent on the level of detail at which the terrain is being visualized.

The terrain model height provider, on the other hand, can return values at any resolution, up to and including the highest level of detail available in the underlying data. To do this, the provider may have to read additional data from disk or from the network, if the required level of detail is not currently in use by the view. As a result, querying the terrain model height provider is potentially a lot more costly than using the view-dependent one.

It is therefore important to choose the appropriate provider for a given use case. The view-dependent provider is strongly recommended for all interactive usages, such as visualization or graphical editing of objects. For instance, to determine a vertical offset to reposition 2D points to ground level in a painter or styler, choose the view-dependent provider.

For computation or analysis tasks where accuracy is key, line-of-sight computations for example, choose the terrain model provider.

27.2 Querying tile-based terrain data

The height providers discussed in the previous section provide support for point-sampled access to the terrain elevation. In some cases, however, it might be desirable to sample whole regions of the terrain at a uniform resolution. Sampling a large number of points using this approach does not provide the optimal performance.

For these cases, `ILspTerrainSupport` provides access to the terrain elevations in the form of an `ILcdEarthTileSet`, which you can obtain through the `getElevationTileSet()` method. The returned tileset contains a single coverage which provides tiles containing `TLcdEarthElevationData` objects. Like the terrain model elevation provider, the tileset can provide data at any requested resolution.

For more information about tilesets, see [Chapter 14](#).

27.3 Determining a point on the terrain using intersection calculation

The methods described in [Section 27.1](#) and [Section 27.2](#) have in common that they provide terrain elevation values for a given 2D point or area. Another useful feature offered by `ILspTerrainSupport` is the ability to compute the intersection between a line, or ray, in the 3D world and the 3D terrain surface. These methods are:

- `getPointOnTerrain()`: takes a point in view coordinates as input and returns the front-most point on the terrain which projects to the same view coordinates. This is useful, for example, to query the terrain under the mouse cursor.
- `intersectTerrain()`: takes two points in world coordinates as input. A ray is cast from the first point in the direction of the second, and the first location where this ray intersects the terrain is returned. This is more general than the previous method, and could be used, for instance, to determine where a moving point could collide with the terrain.

27.4 Draping 2D objects over the terrain

As discussed in [Chapter 22](#), 2D domain objects in a model can be draped over the 3D terrain for visualization. `ILspTerrainSupport` provides some utilities for correct draping. These are:

- `getDrapingContext()`: converts a `TLspContext` received by a painter into one that is suitable for use with draped objects. Specifically, it transforms the cursor coordinates in the context to compensate for the presence of the 3D terrain. This is particularly important when you are implementing the `isTouchedSFCT()` method, for instance.
- `convertFromDrapingContextSFCT()` takes a world point as input and computes the corresponding location where that point would end up if it were draped on the terrain.

CHAPTER 28

Adding graphical effects to a Lightspeed view

Lightspeed views support a number of graphical effects which are applied globally, across all layers. These effects serve to improve the realism, sense of depth or scale, or simply the aesthetics of the images produced by the view. An example of such an effect is lighting, which greatly improves the user's perception of the shape of 3D objects. This chapter discusses the graphics effects API and explains the effects that are available.

28.1 Using ALspGraphicsEffect and TLspGraphicsEffects

Graphics effects are represented by the abstract class `ALspGraphicsEffect`. The design of graphic effects classes is conceptually similar to that of `ALspStyle`: they describe the intended result, but do not contain any implementation logic. It is up to the layers and painters to interpret the properties of the effects, and to apply the effects in terms of OpenGL rendering as they see fit. Because of this approach, `ALspGraphicsEffect` itself is a fairly minimal class: it only contains an enable/disable flag and support for property change listeners.

The class `TLspGraphicsEffects` serves as a central repository for all the effects that are currently applied to a given `ILspView`. Each view creates its own instance of `TLspGraphicsEffects`, and exposes the instance via its `TLspViewServices`. `TLspGraphicsEffects` is essentially a collection of `ALspGraphicsEffects`. When you add an effect to this collection, it is automatically applied to the view.

```
1 ILcdCollection<ALspGraphicsEffect> fx = getView().getServices().getGraphicsEffects();  
2 fx.add( new TLspHeadLight( getView() ) );
```

Program 160 - Adding graphics effects to a view

The following sections describe the concrete implementations of `ALspGraphicsEffect`.

28.2 Adding lighting

Lighting can add important visual cues about the size and shape of 3D objects in a view. Without it, objects are always drawn at full brightness. When the lighting effect is enabled, the object sides that face the light are brighter, whereas the sides that face away from the light are darkened. There are three concrete types of `ALspGraphicsEffect` that enable lighting:

- `TLspDirectionalLight` describes a light source coming from a specified direction. the light source is represented by a direction vector in world coordinates. A directional light can be used to simulate sunlight and its various directions in the course of a day.
- `TLspHeadLight` is also a directional light, but its direction is linked to the orientation of the camera in the Lightspeed view. This can be a useful feature in many applications where lighting can improve a user's perception of shape. In this case, lighting is not used to represent the time of day. The user does not want the whole Earth to have a bright side and a dark side.
- `TLspAmbientLight` is a light of constant intensity that affects all objects equally. It is used to prevent that the unlit sides of objects turn totally black.

The package `samples.lightspeed.demo.application.data.lighting` demonstrates the various ways in which lighting can be used. In particular, the demo contains code that orients a directional light in function of the time of day, so as to simulate sunlight.



The Lightspeed view implementation currently only supports a single directional light, either `TLspDirectionalLight` or `TLspHeadLight`, and a single `TLspAmbientLight`. If more than one of each is added to `TLspGraphicsEffects`, only the first addition of each type will be used. The lighting effects that are added afterwards will be ignored. Also note that lighting is mostly ignored if you are using a 2D view.

28.3 Adding sky and atmosphere effects

To enable the display of a sky or atmosphere in the background of the view, add the class `TLspAtmosphere` to `TLspGraphicsEffects`. Note that when you use `TLspViewBuilder`, the atmosphere is added by default.

28.4 Adding fog

The class `TLspFog` enables a basic fog effect when it is used in a 3D view. Fog can be used to simulate weather conditions to some extent, and can also improve the sense of scale in the view. `TLspFog` computes the amount of fog in the view based on the altitude of the viewer above the Earth. Above a certain altitude, fog is no longer applied.

The following properties are available to tweak the fog effect:

- The fog color: use `setColor()`.
- The visibility distance: use `setVisibilityAtMinAltitude()`. Objects located beyond this distance from the viewer are completely hidden in the fog.
- A minimum and maximum altitude: uses `setMinAltitude()` and `setMaxAltitude()`. These define the altitude range in which the visibility is varied. At or below the minimum altitude, the visibility is equal to the visibility distance, to be set with `setVisibilityAtMinAltitude()`. As the camera approaches maximum altitude, visibility gradually increases. Beyond the maximum altitude, fog is no longer applied.

CHAPTER 29

Projecting images on a terrain in a Lightspeed view

Lightspeed views supports the projection of images on a 3D terrain. A typical use case consists of representing the field of view recorded from a moving Unmanned Aerial Vehicle (UAV).



Figure 103 - Image projected from a UAV

This chapter explains how to get started with the LuciadLightspeed image projection API. The package `com.luciad.view.lightspeed.layer.imageprojection` defines the tools to accomplish such image projections. The sample `lightspeed.imageprojection` provides a demonstration of a moving plane projecting a single image.

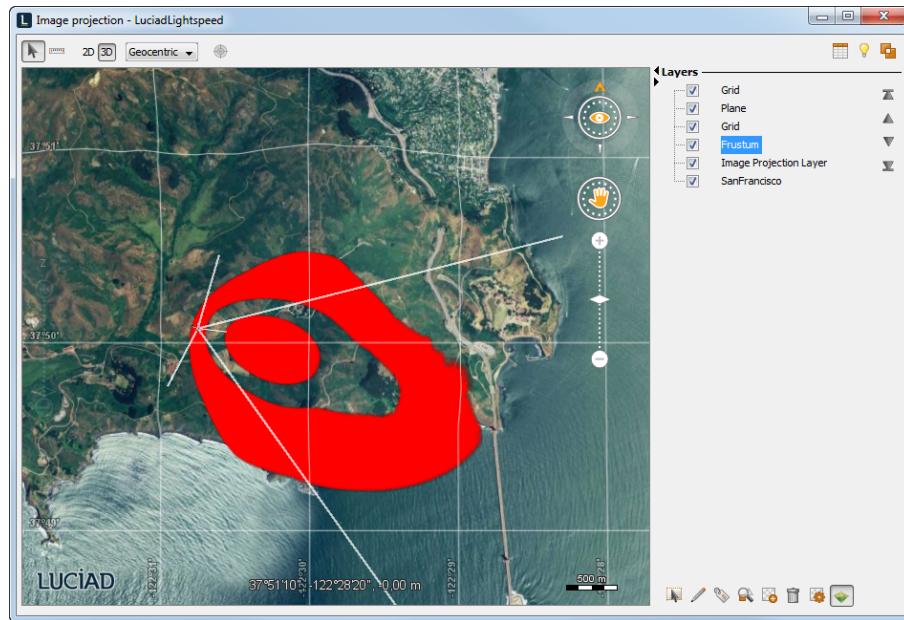


Figure 104 - Sample of image projection in a Lightspeed view

29.1 Using **TLspImageProjectionLayerBuilder** and **ILspImageProjector**

You can visualize image projections in **ILspImageProjectionLayer** layers only. The **ILspImageProjectionLayer** interface is not intended for implementation. Create the layer with the layer builder **TLspImageProjectionLayerBuilder**.

As usual when you are building a Lightspeed layer, you need to assign a styler to the layer. The styler must map each domain object of the model to a **TLspImageProjectionStyle**. You need instances of **TLspImageProjector** and **ALspTextureObject** to build a **TLspImageProjectionStyle**. They represent the projector and the projected image respectively.

```

1  TLspImageProjectionStyle style = TLspImageProjectionStyle
2      .newBuilder()
3      .image(fTexture)
4      .projector(domainObject.getProjector())
5      .build();

```

Program 161 - Create a **TLspImageProjectionStyle** for a single element model
 (from samples/lightspeed/imageprojection/ProjectionStyler)

The most common **ALspTextureObject** implementation is **TLsp2DImageTextureObject**, which is used here to project a buffered image. The **TLspImageProjector** interface defines the location and the direction of the projector. There is a default implementation, called **TLspImageProjector**, which contains setters for the projector parameters.

```
1  public void setLocation(LonLatHeightOrientedPoint aPoint, ILcdModelReference aReference) {
2      TLcdDefaultModelXYZWorldTransformation transformation = new
3          TLcdDefaultModelXYZWorldTransformation();
4      transformation.setModelReference(aReference);
5      transformation.setXYZWorldReference(fReference);
6      TLcdXYZPoint worldPoint = new TLcdXYZPoint();
7      try {
8          transformation.modelPoint2worldSFCT(aPoint, worldPoint);
9      } catch (TLcdOutOfBoundsException e) {
10         // not visible
11         worldPoint.move3D(Double.NaN, Double.NaN, Double.NaN);
12     }
13     fProjector.lookFrom(
14         worldPoint,
15         1000,
16         aPoint.getOrientation(),
17         Math.max(-90, aPoint.getPitch() - 75),
18         aPoint.getRoll(),
19         fReference
20     );
}
```

Program 162 - Change the point of view of a Projector
(from samples/lightspeed/imageprojection/ImageProjector)

PART IV Advanced GXY View and Controller Topics

CHAPTER 30

Working with GXY views

This chapter describes various ways to tailor an `ILcdGXYView` to the application you are building. For many view customizations, such as the addition of grid layers or the creation of balloons, the implementation approach is identical in Lightspeed views and GXY views. For these customizations, we refer you to the appropriate sections in [Chapter 20](#).

30.1 Adding a grid layer to the view

Grid layers do not contain actual data. Their main purpose is to provide a visual, usually uniform, reference that makes it easier to locate data. LuciadLightspeed allows you to add such grid layers to your GXY view. For more information about the grid layer implementations offered by LuciadLightspeed, see [Section 30.1.1](#).

30.1.1 Choosing a predefined grid layer

LuciadLightspeed contains a number of grid layer implementations:

- `TLcdMapLonLatGridLayer`: this is the most commonly used grid. The longitude-latitude grid is formed by evenly spaced meridians and parallels. It allows locating geographically referenced data immediately.
- `TLcdMapLonLatGridLayer` with `TLcdLonLatBorderGridPainter`: this is a maritime-style grid that shows a border around the view with major and minor ticks.
- `TLcdMGRSGridLayer`: based on the MGRS coordinate system, this grid divides the world in 60 longitudinal zones, six degrees wide, along meridians and 20 latitudinal bands, 8 degrees tall, along parallels. It also divides each circular polar zone in half by the Prime Meridian (0 degrees Longitude) and 180 degrees Longitude (East or West). The grid is further divided into blocks of 100km by 100km, blocks of 10km by 10km, up to blocks of 1m by 1m. Contrary to the longitude-latitude grid, a block in this grid always covers the same size area, independent of its location on earth. The representation of a location with regard to this grid is however more complex.
- `TLcdMapGeorefGridLayer`: the World Geographic Reference System (GEOREF) divides the earth into 12 bands of latitude and 24 zones of longitude, each 15 degrees in extent. These areas are further divided into one degree units identified by 15 characters.
- `TLcdXYGridLayer`: A cartesian (XY) grid. It is typically defined in the world reference of the view, which results in an axis-aligned grid.

30.1.2 Using multilevel grids

The package `com.luciad.view.map.multilevelgrid` contains classes to define, render, and perform computations on a uniform, axis-aligned, multi-level grid. The grid depends on the underlying reference system and enables to locate areas using multilevel coordinates.

An `ILcdMultilevelGrid` is defined by:

- the location of its extreme points
- the number of levels it consists of
- the number of subdivisions per level and per axis

An area of an `ILcdMultilevelGrid` is designated by an `ILcdMultilevelGridCoordinate`.

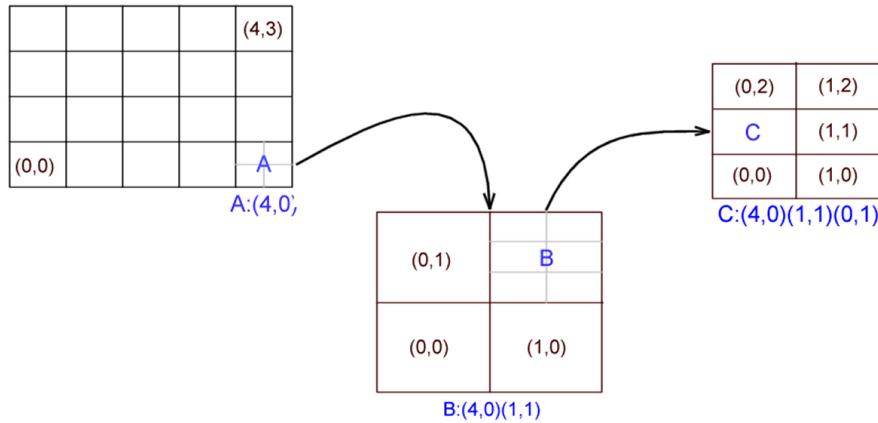


Figure 105 - Sample setup of a multilevel grid

Figure 105 demonstrates the setup of a multilevel grid which is defined by the following parameters:

- the number of levels is 3
- the subdivisions for level 0: 5 along the first axis, 4 along the second axis
- the subdivisions for level 1: 2 along each axis
- the subdivisions for level 2: 2 along the first axis, 3 along the second axis

You can derive the multilevel grid coordinates for the areas A, B, and C from the figure. Note that B denotes a smaller area than A, as you can see from its more specific coordinates. Also from the coordinates, you can see that for example $(4,0)(x,y)$ lies inside $(4,0)$, for any value of (x,y) . `TLcdMultilevelGridCoordinateModel` is an `ILcdModel` implementation for `ILcdMultilevelGridCoordinate` objects that use this type of setup.

The samples in the packages `samples.gxy.grid.multilevel.cgrs` and `samples.gxy.grid.multilevel.gars` contain two examples of a multilevel grid: the Common Grid Reference System (CGRS) grid and the Global Area Reference System (GARS) grid. Both of these grids are based on geodetic references and differ only by the area they cover and the order of subdivisions.

30.2 Adding balloons to the view

Adding balloons to a GXY view works the same as adding balloons to a Lightspeed view. For information on how to create, customize and add balloons, see [Section 20.5](#). To work with balloons in a GXY view, use `TLcdGXYBalloonManager`, the GXY view equivalent of `TLsp-BalloonManager` in a Lightspeed view.

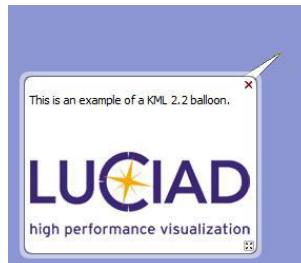


Figure 106 - An example of a balloon

30.3 Transformations between view and model coordinates

An important aspect of working with an `ILcdGXYView` is the ability to transform between view coordinates and model coordinates. [Chapter 44](#) introduces the different coordinate systems that are used internally by an `ILcdGXYView`, as well as the classes that are used to transform data from one coordinate system to another. This section describes in more detail how to transform view coordinates to model coordinates. A typical application of such a transformation is to find out which object(s) in a model are located under the mouse cursor. The cursor position is defined in view coordinates, and in order to find the corresponding domain objects, the application needs to transform the cursor's position to model coordinates.

This transformation happens in two steps.

1. The first step is a transformation from view coordinates to world coordinates. This step is performed using an `ILcdGXYViewXYWorldTransformation`.
2. The second step is to transform the world coordinates to model coordinates, using an `ILcdModelXYWorldTransformation`.

The most convenient way to obtain the appropriate transformation classes is through an `ILcdGXYContext`. An `ILcdGXYContext` is typically used by painters and editors as described in [Chapter 32](#). In other situations, the class `TLcdGXYContext` can be used to construct a context manually. `TLcdGXYContext` must be provided with an `ILcdGXYView` and an `ILcdGXYLayer`. The context then provides immediate access to the transformations needed.

30.3.1 From view coordinates to world coordinates

To go from view to world coordinates, the `ILcdGXYViewXYWorldTransformation` is obtained from the `ILcdGXYContext`. You can then use either `viewAWTPoint2worldSFCT()` or `viewXYPoint2worldSFCT()` to transform from view coordinates to world coordinates. The former method requires a `Point` as input, the latter requires an `ILcdPoint`. Both methods store the transformed result in a side effect parameter. Note that this parameter should be of type `TLcdXYPoint` rather than `TLcdLonLatPoint`, as the latter cannot represent coordinates with the large range that is usually required for world coordinates.

Note that the methods mentioned above operate on points. Equivalent methods enable you to transform bounding boxes: `ILcdGXYViewXYWorldTransformation` has methods `viewAWTBounds2worldSFCT()` and `viewXYBounds2worldSFCT()`

30.3.2 From world coordinates to model coordinates

Given the world coordinates, you can now obtain the `ILcdModelXYWorldTransformation` from the `ILcdGXYContext` and continue with the transformation to model coordinates. This is done using the `worldPoint2modelSFCT()` method. The SFCT model point in this method should be an `ILcdPoint` that can be used in the reference of the model to which you are transforming, a `TLcdLonLatHeightPoint` for geodetic references or a `TLcdXYZPoint` for other references.

Note that the methods mentioned above operate on points. An equivalent method enables you to transform bounding boxes: `ILcdModelXYWorldTransformation` has a method `worldBounds2modelSFCT()`

30.3.3 From model coordinates to view coordinates

The methods discussed in the previous sections have counterparts that work in the opposite direction, which allows for the transformation from model coordinates to view coordinates. Be aware however that an `ILcdBounds` may be enlarged as a result of the transformation. Therefore, transforming a bounding box from view to model coordinates and back may produce a result that differs significantly from the input.

Program 163, taken from `samples.gxy.transformation`, shows how to transform an `ILcdBounds` from view to model coordinates.

```

1 /**
2  * Computes a bounding box in model coordinates that corresponds to the given
3  * rectangle in view coordinates.
4  *
5  * @param aViewBounds The rectangle in view coordinates (pixels)
6  * @param aView the ILcdGXYView on which the rectangle is located
7  * @param aLayer the ILcdGXYLayer for whose model we're computing the bounds
8  * @return an ILcdBounds in model coordinates
9  * @throws TLcdOutOfBoundsException
10 *         if the world to model transformation is unsuccessful
11 */
12
13 private ILcdBounds getModelBoundsForViewBounds(
14     Rectangle aViewBounds,
15     ILcdGXYView aView,
16     ILcdGXYLayer aLayer
17             ) throws TLcdOutOfBoundsException {
18
19     // The (2D) world coordinate system works with XY coordinates.
20     TLcdXYBounds worldBounds = new TLcdXYBounds();
21
22     // The model coordinate system isn't known. The model reference helps
23     // to create compatible bounds (e.g. XY or LonLat).
24     ILcdPoint modelPoint = aLayer.getModel().getModelReference().makeModelPoint();
25     ILcd3DEditableBounds modelBounds = modelPoint.getBounds().cloneAs3DEditableBounds();
26
27     // We use TLcdGXYContext to easily obtain the appropriate transformations.
28     TLcdGXYContext context = new TLcdGXYContext(aView, aLayer);
29
30     // Transform from view to world and from world to model coordinates.
31     // Various other methods are available on the transformations, for example to transform
32     // just points.
33     context.getGXYViewXYWorldTransformation().viewAWTBounds2worldSFCT(aViewBounds, worldBounds
34     );
35     context.getModelXYWorldTransformation().worldBounds2modelSFCT(worldBounds, modelBounds);
36
37     return modelBounds;
38 }
```

Program 163 - Transforming a bounding box from view to model coordinates
 (from samples/gxy/transformation/mouseToGeodetic/TransformCoordinatesController)

30.4 Displaying the mouse cursor position

In some cases, you may want to find out what the lon-lat coordinates of a map position are, the coordinates for the map position of a mouse cursor for instance. In LuciadLightspeed, you can calculate this lon-lat position by using `ILcdGXYViewXYWorldTransformation`, which can transform view coordinates to world coordinates. The world point then needs to be converted to lon-lat coordinates. This is done here using `TLcdGeoReference2GeoReference`.

Program 164 shows an example of a component that displays the lon-lat coordinate of the mouse cursor. It does this by adding a mouse listener to the view and transforming all the incoming mouse locations to world coordinates using a `ILcdGXYViewXYWorldTransformation` object. The world coordinates are then converted to lon-lat coordinates.

```

1 /**
2  * Displays the coordinates and measurements of the location under the mouse pointer on a map.
3  */
4
5 public class MouseLocationComponent extends AMouseLocationComponent {
6
7     private ILcdGXYView fView;
8     private TLcdGXYViewMeasureProvider fViewMeasureProvider;
```

```

9  private CachedMeasures fCachedMeasures = null;
10 public MouseLocationComponent(ILcdGXYView aView,
11                             Iterable<ILcdModelMeasureProviderFactory>
12                             aModelMeasureProviderFactories,
13                             Iterable<ILcdLayerMeasureProviderFactory>
14                             aLayerMeasureProviderFactories) {
15     super((Component) aView);
16     fView = aView;
17     fViewMeasureProvider = new TLcdGXYViewMeasureProvider(aView,
18                                                       aModelMeasureProviderFactories, aLayerMeasureProviderFactories);
19 }
20
21 @Override
22 protected ILcdPoint getCoordinates(Point aAWTPoint, ILcdModelReference aReference) throws
23     TLcdOutOfBoundsException {
24     TLcdXYZPoint worldPoint = new TLcdXYZPoint();
25     TLcdXYZPoint modelPoint = new TLcdXYZPoint();
26
27     fView.getGXYViewXYWorldTransformation().viewAWTPoint2worldSFCT(aAWTPoint, worldPoint);
28     TLcdDefaultModelXYWorldTransformation transformation = new
29         TLcdDefaultModelXYWorldTransformation(
30             aReference,
31             fView.getXYWorldReference());
32     transformation.worldPoint2modelSFCT(worldPoint, modelPoint);
33     return modelPoint;
34 }
35
36 @Override
37 protected TLcdISO19103Measure[] getValues(final ILcdPoint aPoint, final ILcdModelReference
38                                         aPointReference) {
39     // measures are retrieved asynchronously
40     if (fCachedMeasures == null || !fCachedMeasures.isValid(aPoint, aPointReference)) {
41         ALcdMeasureProvider.Parameters parameters = ALcdMeasureProvider.Parameters.newBuilder().
42             build();
43         fViewMeasureProvider.retrieveMeasuresAt(aPoint, aPointReference, parameters, new
44             TLcdGXYViewMeasureProvider.Callback());
45     }
46     return fCachedMeasures != null ? fCachedMeasures.fMeasures : new TLcdISO19103Measure[0];
47 }
48
49 }
```

Program 164 - Implementation of a component that displays the lon-lat coordinates of the mouse cursor

(from samples/gxy/common/MouseLocationComponent)

30.5 Positioning a view programmatically

In general, a view's positioning is determined by its view-to-world transformation. You can use TLcdGXYViewXYWorldTransformation to reposition your view programmatically, by defining a new view origin, mapping it to a world origin, and defining a scaling factor. For more information, see [Section 20.8.1](#), which discusses the use of the equivalent Lightspeed view class TLspViewXYZWorldTransformation2D to reposition views.



It is usually easier to manipulate a view's position by fitting a layer into a view, so that the view focuses on the data in the layer. You can use `TLcdGXYViewFitAction` to do this. For more information, see [Section 5.3.3](#).

CHAPTER 3 I

Working with layers in a GXY view

Chapter 6 describes how to create and use a layer list, a hierarchical layer structure, and a layer tree. This chapter expands those layer fundamentals with more advanced layer topics.

3I.1 Copying layer nodes between layer trees

When you are adding a new view to your application, for the purpose of creating a printout of the displayed map for example, you may want to copy child layer nodes and child layers from one layer tree to another.

Layer trees and layer nodes must be manipulated with care, because adding, removing and copying layer nodes may lead to unexpected results. When you are adding or removing an `ILcdLayerTreeNode` to or from an `ILcdTreeLayered`, for example, the layer node, as well as all of its child layers, will be added or removed.

The same applies to copying layer nodes between layer trees. You cannot simply retrieve all layers from one layer tree and add these to another layer tree as this would add certain layers multiple times. Instead you have to copy each child layer node and each layer from one root to the other, as shown in [Program 165](#).

```
1 ILcdTreeLayered original, copy;
2 //obtaining the root node of the original ILcdTreeLayered
3 ILcdLayerTreeNode rootNode = original.getRootNode();
4 //obtaining the root node of the copy ILcdTreeLayered
5 ILcdLayerTreeNode copyRootNode = copy.getRootNode();
6 //asking all the child layers of the original ILcdTreeLayered
7 Enumeration layers = rootNode.layers();
8 //adding all the layers to the copy ILcdTreeLayered
9 while ( layers.hasMoreElements() ) {
10     ILcdLayer layer = ( ILcdLayer ) layers.nextElement();
11     copyRootNode.addLayer( layer );
12 }
```

Program 165 - Copying layers from one `ILcdTreeLayered` to another

3I.2 Managing selection between views

Section 6.5 describes how you can select domain objects from a model that is contained in a layer. In case the layer is shared by multiple view objects, the selection is visible in all views. Often it is not possible to share a layer between different view objects. For example, in case

one view is graphical and another is tabular or when a model is shared by different layers with another representation of the objects within the model.

Consider the case in which multiple views share the same `ILcdModel` but do not share the layer that contains the model. When a change in selection in one of the layers occurs, all layers that contain the same model should be notified of the change and the method `selectObject` has to be called on these layers to synchronize the selection. The utility class `TLcdSelectionMediator` provides a convenient way to handle selection amongst different `ILcdLayered` (or view) objects. You need to initialize the class with two arrays of `ILcdLayered` objects. One array acts as a **master** or **source** and contains all the `ILcdLayered` objects that listen to selection changes in the `ILcdLayer` objects it contains. The other array acts as a **slave** and contains all the `ILcdLayered` objects that should be synchronized upon a selection change in one of the master objects. The selection of a layer that belongs to a slave `ILcdLayered` is updated if that layer refers to the same `ILcdModel`.

31.3 Filtering objects from a layer

Sometimes you might want to filter certain objects from a layer and exclude them from visualization or other actions such as selection or displaying tool tips. Use cases are:

- To focus on the most relevant data during analysis
- To enhance rendering performance
- To keep the map view clear

There are three ways to filter objects in a layer:

- Set a **(label) scale range**: to only display objects (or labels) in a limited range of scales. For example, set a scale range to only display runways on a scale of 1:100.000 and higher. Refer to the methods `getScaleRange` and `setScaleRange` of `TLcdGXYLayer` in the API reference for more information. For details on converting map scales to layer scales, consult `samples.gxy.common.ScaleSupport`.
- Write a **generic filter**: every object of the `ILcdModel` is given to this filter. You can use any criteria to either include or exclude the object. For example, use a generic filter to only display cities with more than 1.000.000 citizens. An advanced implementation is available as OGC filtering which is described in [Chapter 50](#).
- Set a **minimum object size**: when zoomed out, small objects might no longer be relevant. For example, displaying runways on a scale of 1:100.000.000 does not make much sense. `TLcdGXYLayer.setMinimumObjectSizeForPainting` allows you to set a size threshold (in pixels) for objects. Objects that are smaller than the given threshold, are automatically excluded. You can disable this behavior by setting the threshold to 0. Refer to the API reference for more information.

CHAPTER 32

Painting and editing domain objects in a GXY view

This chapter describes all aspects of graphically representing, or painting, domain objects in a view. After these objects have initially been displayed, you or a user of your application may wish to change the object representation by clicking the mouse or manipulating the object on a touch screen. The process of modifying a domain object after an input event is called editing.

In this context, the support provided by the `ILcdGXYPen` interface is discussed.

32.1 Painting domain objects

Figure 107 shows the typical workflow of painting domain objects in a view.

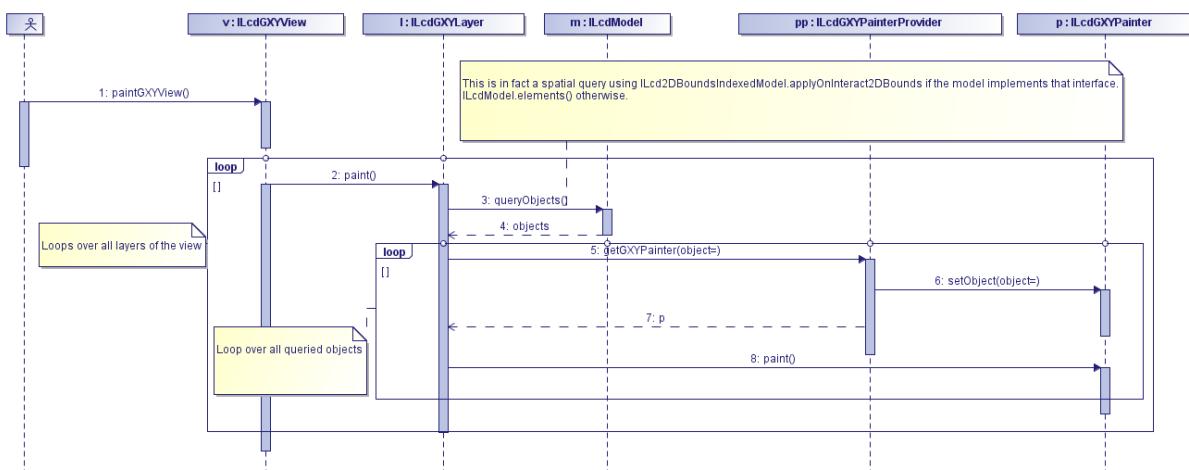


Figure 107 - Paint sequence diagram

When the view receives a paint request through the method `paintGXYView()`, it loops over all of its layers and asks each layer to paint itself. In its turn the layer queries the objects from its model. The layer distinguishes between the geometry and the textual representation (label) of a domain object. To paint the label, a label painter is required as described in Section 36.2.1. The geometry of a domain object is painted with an `ILcdGXYPainter` as described in Section 32.1.1.

32.1.1 Using an `ILcdGXYPainter`

An `ILcdGXYPainter` locates and paints a domain object in an `ILcdGXYView`. The main methods of an `ILcdGXYPainter` are:

- `setObject` to set the domain object that needs to be painted on the view
- `paint` to paint the domain object. Its arguments are:
 - The `java.awt.Graphics` where to paint the domain object on
 - The `mode`: defines which part of the domain object to paint and in which mode (default, selected, or other)
 - the `context`: defines the layer, the view, the pen, and the required transformations between model, world, and view coordinates as described in [Section 30.3](#)

LuciadLightspeed provides several implementations of `ILcdGXYPainter` to properly represent a specific type of domain object. [Section 32.1.2](#) lists the main implementations that LuciadLightspeed provides. You can also create your own implementation, for example, for a customized shape as illustrated in [Section 32.1.5](#).

32.1.2 Main implementations of `ILcdGXYPainter`

The main implementations of `ILcdGXYPainter` that LuciadLightspeed provides are painters:

- For **shapes** as listed in [Table 4](#), these are available in the package `com.luciad.view.gxy.painter`
- For **rasters** such as the `TLcdGXYImagePainter`. If you are still using `ILcdRaster` implementations, you can use `TLcdRasterPainter` and `TLcdMultilevelRasterPainter`. These are available in the package `com.luciad.format.raster`
- That **extend the functionality of other painters** such as the `TLcdGXYHaloPainter` described in [Section 32.5](#)

[Table 4](#) lists the painters for each of the shapes listed in [Section 11.1.2](#).

Shape	Painter
all supported shapes	<code>TLcdGXYShapePainter</code>
<code>ILcdArc</code>	<code>TLcdGXYArcPainter</code>
<code>ILcdArcBand</code>	<code>TLcdGXYArcBandPainter</code>
<code>ILcdBounds</code>	<code>TLcdGXYBoundsPainter</code>
<code>ILcdCircle</code>	<code>TLcdGXYCirclePainter</code>
<code>ILcdComplexPolygon</code>	<code>TLcdGXYPointListPainter</code>
<code>ILcdCircularArc</code>	<code>TLcdGXYCircularArcPainter</code>
<code>ILcdEllipse</code>	<code>TLcdGXYEllipsePainter</code>
<code>ILcdGeoBuffer</code>	<code>TLcdGXYGeoBufferPainter</code>
<code>ILcdPoint</code>	<code>TLcdGXYIconPainter</code>
<code>ILcdPointList</code>	<code>TLcdGXYPointListPainter</code>
<code>ILcdPolygon</code>	<code>TLcdGXYPointListPainter</code>
<code>ILcdPolyline</code>	<code>TLcdGXYPointListPainter</code>
<code>ILcdPolypoint</code>	<code>TLcdGXYPointListPainter</code>
<code>ILcdShapeList</code>	<code>TLcdGXYShapeListPainter</code>
<code>ILcdSurface</code>	<code>TLcdGXYSurfacePainter</code>
<code>ILcdText</code>	<code>TLcdGXYTextPainter</code>

Shape	Painter
-------	---------

Table 4 - LuciadLightspeed shape painters

32.1.3 Using a `TLcdGXYShapePainter`

Program 27 shows the setup of a `TLcdGXYShapePainter` when used for painting `ILcdPolyline` instances. This painter paints all supported shapes and is a good first choice of painter. The program shows how to set the line style of a `TLcdGXYShapePainter`. To set specific rendering effects such as line style, filling pattern, transparency, and so on, you can implement the interface `ILcdGXYPainterStyle`. In the sample a `TLcdStrokeLineStyle` is used. Similarly, `TLcdGXYShapePainter` and all `ALcdGXYAreaPainter` extensions have a `setFillStyle` method to set up a fill style for shapes that enclose an area. See [Section 32.6](#) for more information on line and fill styles.

32.1.4 Using a `TLcdGXYPointListPainter`

This is a specific `ILcdGXYPainter` for painting an `ILcdPointList`, which is a sequence of `ILcdPoint` objects. A `TLcdGXYPointListPainter` paints a point list either as:

- Individual points
- A polyline (each point, except the last one, is connected to the previous point)
- A polygon (same as a polyline, whereby the last point is connected to the first point)
- An area (a filled polygon of which the outline is not painted)
- An outline area (a filled polygon of which the outline is painted)

The `TLcdGXYPointListPainter` paints the lines between the points either as straight lines, as curves that represent geodetic lines, or as rhumb lines. The painter delegates this kind of low-level painting to the `ILcdGXYPen` that is associated to the `ILcdGXYLayer`. For more information on the usage of pens, refer to [Section 32.3](#).

An extension of `TLcdGXYPointListPainter` is `TLcdGXYRoundedPointListPainter` which paints a line with rounded corners between the points. A spline algorithm is used to transform the set of points into a smooth line that interpolates between the original points.

32.1.5 Customizing an `ILcdGXYPainter`

There are several ways to make a specific implementation of the interface `ILcdGXYPainter`. If you need to configure a painter based on the properties of your current domain object, consider creating a custom `ILcdGXYPainterProvider` instead. This is shown in [Section 32.1.6](#). If you need more flexibility, reusing one of the predefined painter implementations is often most appropriate. You can do this by extension (inheritance), by composition, or by a combination of both. Sometimes it may also be required to make your own implementation, for example as shown in [Chapter 35](#).

When a domain object is required which is not a trivial combination of existing domain objects you can create a new domain object and provide an `ILcdGXYPainter` implementation for it. [Chapter 35](#) presents a case study of an `ILcdGXYPainter` implementation for a hippodrome shape.

32.1.6 Using an `ILcdGXYPainterProvider`

An `ILcdGXYPainter` is, in general, suited to paint only one type of domain object. The purpose of an `ILcdGXYPainterProvider` is to associate a valid `ILcdGXYPainter` to each domain object that needs to be painted on a `java.awt.Graphics`. The painter is returned by the method `getGXYPainter`, that takes the particular domain object as an argument as shown in Figure 107.

An `ILcdGXYPainterProvider` is typically used on a `TLcdGXYLayer` to get a valid `ILcdGXYPainter` for representing domain objects contained in its `ILcdModel`. In this case, the `ILcdGXYPainterProvider` must be set as a property of the `TLcdGXYLayer` as described in Section 32.1.8.

32.1.7 Main implementations of `ILcdGXYPainterProvider`

Each class of `LuciadLightspeed` implementing the interface `ILcdGXYPainter` also implements the interface `ILcdGXYPainterProvider`. The method `getGXYPainter` returns the painter itself for each object. This is useful when all objects can be painted with the same painter, since it avoids the creation of a new painter provider as illustrated in Program 166.

```
1 TLcdGXYShapePainter painter = new TLcdGXYShapePainter();
2 layer.setGXYPainterProvider(painter);
```

Program 166 - Using an `ILcdGXYPainter` as `ILcdGXYPainterProvider`
 (from samples/gxy/transformation/geodeticToGrid/MainPanel)



In case you have your own implementation of `ILcdGXYPainter` that does not implement `ILcdGXYPainterProvider`, or in case not all the objects can be painted with the same painter, you should explicitly use an `ILcdGXYPainterProvider`.

A simple implementation of the interface `ILcdGXYPainterProvider` is the class `TLcdSingleGXYPainterProvider`. A `TLcdSingleGXYPainterProvider` is set up by giving it one `ILcdGXYPainter`, which it will return upon each request for an `ILcdGXYPainter`.

32.1.8 Using an `ILcdGXYPainterProvider` on a `TLcdGXYLayer`

The collaboration of a `TLcdGXYLayer` and an `ILcdGXYPainterProvider` to paint an object is shown in Program 13. First, an `ILcdGXYPainterProvider` must be set as painter provider of the `TLcdGXYLayer`. Whenever the `TLcdGXYLayer` needs to paint a domain object, it asks its `ILcdGXYPainterProvider` for a valid `ILcdGXYPainter`, in this case a `TLcdGXYImagePainter`, and delegates the painting to the painter.

32.1.9 Visualizing shapes based on domain object properties

`ILcdGXYPainterProvider` provides a good entry point to examine a domain object and configure a painter accordingly. To illustrate this, we will show a painter provider that visualizes population density. Take a look at the painter provider snippet in Program 167. It configures a `TLcdGXYShapePainter` with a fill color before returning it. The chosen color is determined by the features of the `ILcdDataObject` domain object. You can find the full example in the class `PopulationLayerFactory` of the package `samples.gxy.statisticalPainter`.

```

1  private class PopulationDensityPainterProvider implements ILcdGXYPainterProvider {
2
3      private final TLcdGXYShapePainter fPainter = new TLcdGXYShapePainter();
4      private final TLcdGXYPainterColorStyle fFillStyle = new TLcdGXYPainterColorStyle();
5
6      PopulationDensityPainterProvider() {
7          fPainter.setFillStyle(fFillStyle);
8          fPainter.setLineStyle(new TLcdGXYPainterColorStyle(Color.GRAY, Color.ORANGE));
9          // Outlines the selection so the density color remains visible underneath.
10         fPainter.setSelectionMode(ALcdGXYAreaPainter.OUTLINED);
11     }
12
13     @Override
14     public ILcdGXYPainter getGXYPainter(Object aObject) {
15         if (aObject instanceof ILcdDataObject) {
16             ILcdDataObject dataObject = (ILcdDataObject) aObject;
17             Color color = colorForFactor(getDensity(dataObject));
18             fFillStyle.setDefaultColor(color);
19             fPainter.setObject(aObject);
20             return fPainter;
21         }
22         return null;
23     }
24 }
```

Program 167 - Configuring an ILcdGXYPainter for an ILcdDataObject in an ILcdGXYPainterProvider
 (from samples/gxy/statisticalPainter/PopulationLayerFactory)

32.1.10 Using an ILcdGXYPainterProvider on composite shapes

An ILcdShapeList consists of several ILcdShape objects. The task of a TLcdGXYShapeListPainter is to paint each of these ILcdShape objects. For this, it needs to be given an ILcdGXYPainterProvider, which must provide a valid ILcdGXYPainter to paint each ILcdShape object of the ILcdShapeList.

Similarly, TLcdGXYCompositeCurvePainter, TLcdGXYCompositeRingPainter and TLcdGXYSurfacePainter need an ILcdGXYPainterProvider to paint each ILcdCurve object of the ILcdCompositeCurve, ILcdRing or ILcdSurface respectively.

TLcdGXYShapePainter automatically unpacks ILcdShapeList objects and delegates to the appropriate painter.

32.1.11 Using an array of ILcdGXYPainterProvider objects

In addition to a single ILcdGXYPainterProvider, TLcdGXYLayer supports the use of an array of ILcdGXYPainterProvider objects. Every object contained in the ILcdModel of a TLcdGXYLayer is rendered with the ILcdGXYPainter provided by the first ILcdGXYPainterProvider in the array, then with the ILcdGXYPainter provided by the second ILcdGXYPainterProvider and so on.

Figure 108 shows a use case for an array of painter providers. On the left are two lines which should be represented as highways. A typical representation for a highway is a thick red line with a smaller yellow line in the middle. Implementing an ILcdGXYPainter which paints a line like this is trivial. However, when two line cross each other, the result is as displayed in the middle of Figure 108. You can achieve the preferred representation, on the right, by providing two ILcdGXYPainter implementations that are returned by two ILcdGXYPainterProvider objects in an array.

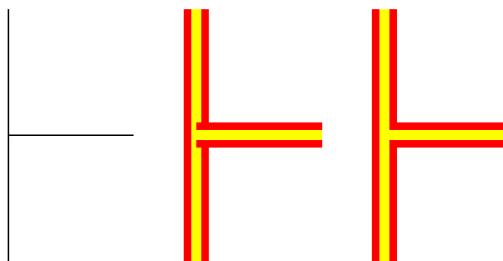


Figure 108 - Painting a highway

32.1.12 Customizing an `ILcdGXYPainterProvider`

An implementation of an `ILcdGXYPainterProvider` often contains a pool of available `ILcdGXYPainter` objects. Which painter is returned depends on the object that needs to be painted as shown in [Program 168](#).

```

1  private static class MyPainterEditorProvider implements ILcdGXYPainterProvider,
2      ILcdGXYEditorProvider {
3          @Override
4          public ILcdGXYPainter getGXYPainter(Object aObject) {
5              ILcdGXYPainter painter = null;
6              if (aObject instanceof ILcd2DEditablePolyline) {
7                  painter = fPolylinePainter;
8              } else if (aObject instanceof ILcd2DEditablePolygon) {
9                  painter = fPolygonPainter;
10             } else if (aObject instanceof ILcdPoint) {
11                 painter = fPointPainter;
12             } else if (aObject instanceof ILcd2DEditableCircle) {
13                 painter = fCirclePainter;
14             }
15             if (painter != null) {
16                 painter.setObject(aObject);
17             }
18             return painter;
}

```

Program 168 - A simple implementation of `ILcdGXYPainterProvider`
 (from samples/gxy/decoder/custom1/Custom1LayerFactory)

[Program 168](#) shows the a painter provider, which can provide an `ILcdGXYPainter` for `ILcdCircle` objects, for `ILcdPolygon` objects, for `ILcdPolyline` objects, for `ILcdArcBand` objects, for `ILcdArc` objects, and for `ILcdPoint` objects. A `GXYPainterEditorProvider` contains one particular `ILcdGXYPainter` per type of supported `ILcdShape`. The `ILcdGXYPainter` objects returned for `ILcdPolygon` and `ILcdPolyline` are instances of the class `TLcdGXYPointListPainter`, but they are configured differently.

The method `getGXYPainter` checks the type of the object it receives as an argument, and returns the corresponding `ILcdGXYPainter`. Note that the object must be set to the `ILcdGXYPainter` before `ILcdGXYPainter` is returned by the `ILcdGXYPainterProvider`.

32.1.13 Painting rasters



This section discusses the `ILcdRasterPainter` extensions of `ILcdGXYPainter`. These are part of the `com.luciad.format.raster` package, and allow you to paint `ILcdRaster` objects. LuciadLightspeed also offers image processing API in the `com.luciad.imaging` package, which allows you to process and access images at the pixel level. If you work with LuciadLightspeed image processing in a GXY view, you must use `TLcdGXYImagePainter` to paint `ALcdBasicImage` objects. For more information about the LuciadLightspeed image processing framework, see [Section 12.3](#).

The `ILcdRasterPainter` interface is an extension of the `ILcdGXYPainter` interface for painting rasters and multilevel rasters. It contains additional methods for controlling properties that affect the appearance of the rasters:

- `transparency` specifies the transparency of the raster in the view. Making a raster partially transparent allows the visualization of underlying rasters or other data in a single view, for example an aerial picture and a map.
- `brightness` specifies the brightness of the raster in the view. Reducing the brightness of a raster can be useful for reducing the contrast of a background map, for example.
- `colorModel` specifies the color model that is applied to the raster. Different color models are mostly useful when working with `IndexColorModel` instances. For example, you can change the coloring of elevation rasters by setting an `IndexColorModel` with 2^{16} colors. In this particular case, the color index should be interpreted as an unsigned short elevation.

In addition, most raster painters have an `RGBImageFilter` that allows you to alter the colors of the displayed rasters. For example, you can map the original RGB values to grayscale values, or to RGB values with enhanced contrast.

If the pixel density of a raster appears too high, that is, if too many raster pixels would project onto a single view pixel, a painter may paint the raster outlines instead of the raster itself. This approach avoids having to load too many raster data (lazily) for just painting a few pixels when zoomed out. On the other hand, if the pixel density appears too low, the raster is not painted at all. This approach avoids displaying large blocky raster pixels when zoomed in.

The exact behavior is controlled by these properties:

- `startResolutionFactor` is the highest resolution ratio (number of raster pixels per view pixel) at which a raster is painted. The default is 10.0, meaning that a raster is not painted if more than 10 raster pixels would be projected onto a single view pixel, on average. Smaller values (for example 2.0) avoid displaying the raster at an even lower resolution ratio. Smaller values therefore generally improve performance when zoomed out.
- `stopResolutionFactor` is the lowest resolution ratio (number of raster pixels per view pixel) at which a raster is painted. The default is 0.0, meaning that a raster is always painted, even if a single raster pixel is projected onto many view pixels. Larger values (for example 0.5) avoid displaying the raster at a higher resolution ratio. Larger values therefore avoid displaying blocky rasters when zoomed in.
- `forcePainting` is a flag that allows you to override the above settings. Although useful for testing, setting it to true generally causes severe performance drops when zooming out

on large rasters. The default is false.

- `paintOutline` specifies whether or not to paint the raster outlines and tile outlines when the pixel density is too high. The default is true.
- `outlineColor` specifies the color in which the raster outlines and tile outlines are painted.
- `maxNumberOfOutlineTiles` specifies how many outlines of raster tiles can be painted at most. If a raster contains more tiles, only its global outlines are painted.

32.1.14 Main implementations of `ILcdRasterPainter`

The sections below describe the main implementations of `ILcdRasterPainter`.

TLcdRasterPainter

`TLcdRasterPainter` is the main implementation of `ILcdGXYPainter` for `ILcdRaster` instances. It paints rasters onto `ILcdGXYView` instances, scaling and transforming them on the fly. In addition to the normal raster painter properties, you can tune `TLcdRasterPainter` by some additional properties. For rasters that have to be transformed to a different reference system, the following properties are relevant:

- `warpBlockSize` specifies the size of the blocks in which the raster transformation is approximated using bilinear transformation. Smaller values result in more accurate results, but also in larger computation times. The default is 64. With a value of 1, every pixel is transformed without approximation.
- `reuseInternalBuffer` specifies whether the painter can reuse an internal buffer for storing warped images. If true, this buffer is reused over successive calls. If false, buffers are allocated each time, when required. In the latter case, warped images may be cached along with the painted rasters, depending on the `paintCache` property. This allocation and caching strategy may be memory consuming, so reusing the internal buffer is generally recommended.
- `paintCache` specifies whether the painter should cache the raster images that it has drawn. The images are cached with the rasters that are painted, provided they implement the `ILcdCache` interface. For example, `TLcdRaster` and `TLcdMultilevelRaster` implement `ILcdCache`. Note that this caching strategy may be memory consuming if many raster objects are painted at the same time.
- `border` specifies the size of an additional border around the view, expressed in pixels, inside which the raster is transformed and then cached. For small pan operations in the view, the cached version can then be reused. When the pan operations exceed the border size, the raster is transformed again.

For rasters that do not have to be transformed to a different reference system, the following properties are relevant:

- `useSubTileImageCaching` specifies whether the raster painter should perform internal caching of images corresponding to raster subtiles. This helps to avoid disk access during painting (due to lazy loading of raster tiles) and may also speed up the painting of rasters with non-standard color models (for example 16-bit indexed color images).
- `useDeferredSubTileDecoding` specifies whether the `paint` method of the raster painter may defer the painting of tiles that have not been decoded yet. Parts of the displayed raster may then remain temporarily blank, but the application is not held up by

the decoding. The display is updated completely when all tiles have been decoded. This setting is only applicable if `useSubTileImageCaching` is set to true.

If you know that the rasters are defined in the same reference system as the view, you can use the `TLcdNoWarpRasterPainter` instead of the more general `TLcdRasterPainter`. This implementation always uses the method `retrieveTile` of the interface `ILcdRaster`, to retrieve entire images at once. These images can then be scaled and painted efficiently.

If you know that the rasters are defined in a different reference system than the view, you can use `TLcdWarpRasterPainter`. This implementation always uses the method `retrieveValue` of `ILcdRaster` to retrieve single raster values at a time, constructing transformed images to be painted.

The current implementation of `TLcdRasterPainter` delegates to `TLcdNoWarpRasterPainter` if possible. Otherwise, it delegates to `TLcdWarpRasterPainter`.

TLcdMultilevelRasterPainter

`TLcdMultilevelRasterPainter` is the main implementation of `ILcdGXYPainter` for `ILcdMultilevelRaster` instances. When painting a multilevel raster, it first selects the most appropriate raster level. The most appropriate level of detail depends on the scale of the view, on the transformation from the model reference to the world reference, and on the pixel densities of the available raster levels.

You can control the level selection by means of the following properties:

- `levelSwitchFactor` is the factor that affects the scale threshold at which a raster level is selected. The default value is 1.0; as soon as a single raster pixel would project to several screen pixels, a higher resolution level is used. This way, the highest practical raster quality is chosen. A value smaller than 1.0 (for example 0.2) delays this switching. A single raster pixel may then project to multiple screen pixels. The raster may become pixelated, meaning that it may be visualized with large blocky pixels. However, as lower resolution levels are accessed, less raster data has to be decoded. Setting a small value therefore generally improves the painting performance.
- `levelSwitchScales` is an optional list of scales that can achieve the same effects as the `levelSwitchFactor`. It explicitly expresses the scales at which the painter switches between raster levels, instead of relying on a single factor.

Similarly as to the `TLcdRasterPainter`, if you know that the multilevel rasters are defined in the same reference system as the view, you can use the `TLcdNoWarpMultilevelRasterPainter` instead of the more general `TLcdMultilevelRasterPainter`. If you know that the multilevel rasters are defined in a different reference system than the view, you can use `TLcdWarpMultilevelRasterPainter`.

The current implementation of `TLcdMultilevelRasterPainter` delegates to `TLcdNoWarpMultilevelRasterPainter` if possible. Otherwise, it delegates to `TLcdWarpMultilevelRasterPainter`.

TLcdAllInMemoryRasterPainter

The `TLcdAllInMemoryRasterPainter` implements `ILcdGXYPainter` for `ILcdRaster` instances. It computes the warped version of each raster once and caches it with the raster (provided it implements the `ILcdCache` interface). It is therefore only practical for painting small rasters.

32.1.15 Painting a 3D terrain in 2D from a tile repository

You can also visualize 3D terrain data in a 2D view using the `TLcdGXYImagePainter`. This class paints the texture of a 3D terrain as if it were a 2D raster.

By default the `TLcdGXYImagePainter` paints the first image coverage available in the tile set. If no such coverage exists it will paint the first elevation coverage. So if you want to visualize the terrain elevation as a 2D raster, you can simply construct a tile set containing only the elevation coverage and paint that instead. This can be achieved using `ALcdEarthCoverageFilterTileSet`.

32.2 Graphically editing domain objects

The following sections describe how to use an `ILcdGXYEditor` and `ILcdGXYEditorProvider` to graphically edit domain objects.

32.2.1 Using an `ILcdGXYEditor`

To edit/modify domain objects represented/painted in a view from input events, such as `MouseEvents` or `TLcdTouchEvent`s, LuciadLightspeed provides the interface `ILcdGXYEditor`. Depending on the current representation of the domain object and the information contained in the input event, the `ILcdGXYEditor` can modify the domain object accordingly. Since the representation of a domain object is only known to the `ILcdGXYPainter` that painted the domain object, it is almost always necessary for an `ILcdGXYEditor` to work in combination with an `ILcdGXYPainter`. Therefore, all LuciadLightspeed implementations of the interface `ILcdGXYEditor` also implement the interface `ILcdGXYPainter`. And conversely, almost all implementations of `ILcdGXYPainter` also implement `ILcdGXYEditor`.

Similar to an `ILcdGXYPainter`, an `ILcdGXYEditor` is suited to edit one type of domain object. For example, a `TLcdGXYPointListPainter` can paint any domain object that implements the interface `ILcdPointList`. But it can also edit in two dimensions any domain object that implements the interface `ILcd2DEditablePointList` or `ILcd3DEditablePointList`.

The two main methods of the interface `ILcdGXYEditor` are:

- `setObject`. This method is used to set the domain object that needs to be edited on a view.
- `edit`. Its arguments are:
 - the `java.awt.Graphics` where to paint the domain object on
 - the `mode`: defines how to edit the domain object, for example it defines whether the object is being created, translated, reshaped, and so on.
 - the `context`: defines the layer, the view, and the valid pen for editing support

32.2.2 Main implementations of `ILcdGXYEditor`

As for `ILcdGXYPainter`, LuciadLightspeed provides an implementation of `ILcdGXYEditor` for most of the `ILcdShape` implementations: `TLcdGXYArcBandPainter`, `TLcdGXYArcPainter`, `TLcdGXYBoundsPainter`, `TLcdGXYGeoBufferPainter`, `TLcdGXYCirclePainter`, `TLcdGXYEllipsePainter`, `TLcdGXYIconPainter`, `TLcdGXYPointList`

Painter `TLcdGXYTextPainter`, and `TLcdGXYShapeListPainter`. All these implementations are available in the package `com.luciad.view.gxy.painter`.

32.2.3 Customizing an `ILcdGXYEditor`

Chapter 35 provides a case study of implementing an `ILcdGXYEditor` from scratch. Take the following two notes into account when customizing an `ILcdGXYEditor`:

- Composition is less suited to create an `ILcdGXYEditor` implementation: editing one object may affect other objects in a composite object.
- To allow for reversible changes made by an `ILcdGXYEditor`, make sure that the `ILcdGXYEditor` implements `ILcdUndoableSource`. Refer to Section 37.4 for more information.

32.2.4 Editing with multiple input points

It might be that you want to define editing behavior that uses multiple input points. Examples of this are rotating an object or editing multiple handles of an object at the same time using a touch screen. All the input points are available from the given `TLcdGXYContext`. Chapter 35 shows an example of this multi-point edit behavior for a hippodrome shape.

32.2.5 Using an `ILcdGXYEditorProvider`

As for the interface `ILcdGXYPainter`, some classes may need a way to retrieve a valid `ILcdGXYEditor` to edit a given domain object. Examples of such classes are `TLcdGXYLayer` and `TLcdGXYShapeListPainter` as described in Section 32.1.6. The behavior of an `ILcdGXYEditorProvider` is similar to that of an `ILcdGXYPainterProvider`.

32.3 Painting and editing support

The following sections explain how to use an `ILcdGXYPen` to provide support for painting and editing domain objects.

32.3.1 Using an `ILcdGXYPen`

An `ILcdGXYPen` is a low-level utility that is associated to an `ILcdGXYLayer` and that is mainly used in implementations of `ILcdGXYPainter` and `ILcdGXYEditor`. An example of another class that uses `ILcdGXYPen` is the controller `TLcdMapRulerController`, which draws the shortest distance path between two points on the ellipsoid of the world reference and a circle with this distance as radius.

An `ILcdGXYPen` contains methods for the handling of basic shapes like lines, polylines, polygons, and arcs. It has methods for:

- drawing the basic shapes on a `java.awt.Graphics`
- accumulating (combining) several basic shapes with a polyline approximation represented in view coordinates, given the transformations `ILcdModelXYWorldTransformation` and `ILcdGXYViewXYWorldTransformation`. This is used, for example, to fill the polyline approximation of a shape object that consists of lines and arcs. This approximation is stored within the pen. Alternatively you can also construct the approximation in an external `ILcdAWTPath` for later usage.

- approximating basic shapes as a polyline in world coordinates. Multiple basic shapes can be added one after another in an implementation of `ILcdGeneralPath`, given the transformation from model coordinates to world coordinates (`ILcdModelXYWorldTransformation`). This approximation then only needs a transformation from world coordinates to view coordinates in order to be displayed. If the approximation is kept somewhere in memory, it is not necessary to perform the expensive calculations anymore to get a polyline approximation and the transformations from model coordinates to world coordinates.
- checking whether a basic shape is touched by a given screen coordinate with some sensitivity level, expressed in pixels.

32.3.2 Main implementations of `ILcdGXYPen`

LuciadLightspeed provides the following main implementations of the interface `ILcdGXYPen`:

- `TLcdGeodeticPen`
- `TLcdProjectionPen`
- `TLcdGridPen`
- `TLcdGXYPen`

`TLcdGeodeticPen` and `TLcdProjectionPen` both provide support for painting and editing basic shapes in geodetic model references, such as geodetic lines. `TLcdGeodeticPen` assumes that the basic shapes are defined on the `ILcdEllipsoid` of the model reference, while `TLcdProjectionPen` considers them on the `ILcdEllipsoid` of the world reference. `TLcdGridPen` is used for shapes of which the model coordinates are defined in a grid reference system. Refer to [Chapter 44](#) for more information about model references.

`TLcdGXYPen` is a more general implementation of the interface `ILcdGXYPen` as it can handle both geodetic and Cartesian grid coordinates as model coordinates. Most of its methods are implemented in an approximate way for the sake of performance. For example, a geodetic line is always drawn as a straight line in the view coordinate system, which is a good approximation if the line segments are small. `TLcdGeodeticPen`, `TLcdProjectionPen`, and `TLcdGridPen` have a method `setStraightLineMode` which allows the usage of this straight line mode approximation with these pen classes as well.

All pen implementations approximate curved lines like geodetic lines, arcs, and so on, by means of lists of points. Five parameters allow you to tune this discretization process: the minimum and maximum recursion depth of the adaptive algorithm, a threshold on the angle between successive line segments, and thresholds on the distances between consecutive points (in world or view coordinates).

32.3.3 Setting an `ILcdGXYPen` to a layer

[Program 169](#) shows the setup of a `TLcdGeodeticPen` called `geodetic_pen` which is provided to an `ILcdGXYLayer` (called `gxy_layer`) that contains a model with a geodetic model reference.

```
1    layer.setGXYPen(ALcdGXYPen.create(aModel.getModelReference()));
```

Program 169 - Configuring a `TLcdGeodeticPen` and attaching it to an `ILcdGXYLayer`
 (from samples/gxy/shapes/MainPanel)

32.4 Painting symbology

A symbology defines a standard for visualizing domain objects, such as points and lines. The domain object properties determine the used icon and/or style. LuciadLightspeed provides support for painting symbologies, explained in this chapter. Implementations of standardized symbologies are available in the LuciadLightspeed components:

- The military symbologies MIL-STD 2525b, MIL-STD 2525c, APP-6A, APP-6B, APP-6C and TTA-106 V4 in the Defense Symbology component
- The naval symbology S-52, part of the Maritime Standards component
- ICAO symbology, part of the Aviation Standards component
- Geosym, part of the Defense Standards component

32.4.1 Painting symbologies for points

The generic mechanisms that LuciadLightspeed provides for painting symbology for points are:

- Icons: `ILcdIcon` objects to represent the graphical symbols
- An icon painter `TLcdGXYIconPainter` to display the symbol
- An object icon provider `ILcdObjectIconProvider` to link the domain object (point or shape) to its correct icon

The following sections describe each of the involved classes and interfaces in more detail.

Using an `ILcdIcon`

The interface `ILcdIcon` is the core interface for representing a graphical point symbol in LuciadLightspeed. It represents a figure or picture with a fixed width and height specified in pixels. Its method `paint` allows you to paint the icon on any `Graphics` instance. Next to the visualization of point or shape data in an `ILcdGXYView`, it is also frequently used in a GUI to represent an action, a controller, a layer, and so on.

LuciadLightspeed offers various implementations, of which the most important are:

- `TLcdSymbol`: offers a number of predefined symbols, like a circle, a triangle, and more. The implementation allows you to configure the size, the border color, and the fill color of each icon.
- `TLcdIconFactory`: this class contains a number of images that are frequently used in LuciadLightspeed as a GUI icon associated with a certain action, for example an image of a magnifying glass for a zoom action or a polygon image for a polygon controller.
- `TLcdMessageIcon`: This class displays a message in a box. Message icons are useful to display messages on an `ILcdGXYView`. You can use the method `putCornerIcon`, for example, to give a user an indication that the view is loading data.
- `TLcdImageIcon`: this class is used to create an icon from an image file.
- `TLcdAnchoredIcon`: creates a wrapper icon which is used to anchor an icon with a point other than its center point.
- `TLcdHaloIcon`: creates a wrapper icon which is used to add a halo around another icon.

Because the Java Foundation Classes (JFC/Swing) contain a similar interface `javax.swing.Icon`, LuciadLightspeed provides two conversion classes `TLcdIconSW`

and `TLcdSWIcon` that are used to wrap `ILcdIcon` implementations into an `Icon` and the other way around. These classes are useful when you want to use an `ILcdIcon` in a Swing class or conversely, an `Icon` in a `LuciadLightspeed` class.

Using a `TLcdGXYIconPainter`

To visualize `ILcdPoint` and `ILcdShape` objects on a map using icons, `LuciadLightspeed` offers the class `TLcdGXYIconPainter`. This class is both an implementation of `ILcdGXYPainter` and `ILcdGXYEditor`, so it can be used for painting and editing on a view. It paints an `ILcdIcon` at the location of the `ILcdPoint` or the focus point of the `ILcdShape`. The painter uses three icons, which you can configure independently: one to paint as default, one to paint when the object is selected, and one to paint Snapping target points of the object. Program 170 illustrates its use by creating a layer that is initialized with this painter.

```
1 painter.setIcon(new TLcdImageIcon("images/mif/mif20_airplane.gif"));
2 painter.setSelectedIcon(new TLcdSymbol(TLcdSymbol.CIRCLE, 22, Color.red));
```

Program 170 - Using a `TLcdGXYIconPainter` to visualize point data
(from samples/gxy/shapes/MainPanel)

Using an `ILcdObjectIconProvider`

Most applications require the use of a set of icons, each to be used with a different group of points or shapes. To support this, `LuciadLightspeed` provides the interface `ILcdObjectIconProvider`, which is used to retrieve an icon for a given `Object` instance. A typical implementation of this interface has an internal icon repository from which icons are returned for different objects, based on their type, properties, or features. Program 171 contains a simple implementation that always returns the same icon.

```
1 /**
2  * This implementation of <code>ILcdObjectIconProvider</code> always returns
3  * the same icon, regardless of the supplied object.
4 */
5 public class MyIconProvider implements ILcdObjectIconProvider {
6
7     private ILcdIcon fIcon = new TLcdSymbol(TLcdSymbol.FILLED_RECT, 5, Color.black, Color.black
8         );
9
10    public ILcdIcon getIcon(Object aObject) throws IllegalArgumentException {
11        return fIcon;
12    }
13
14    public boolean canGetIcon(Object aObject) {
15        return true;
16    }
}
```

Program 171 - Sample implementation of `ILcdObjectIconProvider`

To use icon providers with a `TLcdGXYIconPainter`, you can use the methods `setSelectionIconProvider` and `setIconProvider`.

32.5 Drawing halos around objects

A halo is an outline of constant width that is drawn around a shape or label. It is typically drawn in a color that contrasts with that of the haloed object itself, to make the object clearly visible on all backgrounds.



Figure 109 - Halos improve label visibility

The following painters have built-in support for halos:

- `TLcdGXYIconPainter`
- `TLcdLonLatGridPainter`

You can enable halos on other painters using the class `TLcdGXYHaloPainter`. This class wraps around an existing `ILcdGXYPainter` and adds a halo to anything drawn by the wrapped painter.

All halo-enabled painters share a common set of methods to configure the halo effect:

- `setHaloEnabled` specifies whether or not to draw halos.
- `setHaloThickness` sets the thickness (in pixels) of the halo. Note that higher values result in reduced performance. A value of 1 or 2 pixels is recommended.
- `setHaloColor` sets the color of the halo. Choose this such that it contrasts well with the color of the shape or label to which the halo is added.
- `setUseImageCache` specifies whether or not the painter should cache image representations of the haloed objects. Enabling this can compensate for the performance overhead associated with enabling halos. When caching is off, the painter must redraw the halo for every repaint. When caching is enabled, the painter can paint the object and its halo once and draw the cached image on the view for subsequent repaints.
- `clearImageCache` removes objects from the image cache. There may be situations in which an object or its label changes, without the painter knowing about it. In that case, any image that the painter may have already cached for this object becomes invalid and should thus be recreated. By calling `clearImageCache`, applications can force this behavior whenever necessary. Two versions of `clearImageCache` exist: one with an `Object` parameter, which removes only the image cached for that particular object, and one without parameters, which removes all currently cached images.

32.6 Line and fill styles

This section discusses the common `ILcdGXYPainterStyle` implementations in LuciadLightspeed. These styles can be set on just about every painter in LuciadLightspeed. Note that all styles accept transparent or translucent colors.

`TLcdGXYPainterColorStyle`: simplest `ILcdGXYPainterStyle` implementation. It allows you to set a different color for selected and unselected objects and can be both used as line style and fill style.

TLcdStrokeLineStyle: draws lines with a given stroke and color. **TLcdStrokeLineStyleBuilder** contains convenience methods to build common strokes.

TLcdGXYComplexStroke: draws lines with one or more provided custom patterns. See Section 32.7 for more information on setting up such this style.

TLcdGXYHatchedFillStyle: fills areas by hatching them with a given line width and color.

32.7 Drawing complex strokes

A complex stroke is a **Stroke** that consists of one or more patterns that are painted repetitively along a path.

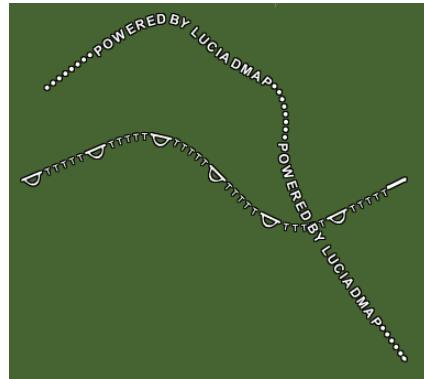


Figure 110 - Examples of complex strokes

The LuciadLightspeed class **TLcdGXYComplexStroke** class provides complex stroke functionality.

32.7.1 Using complex strokes

You can define the following properties for a complex stroke:

- A list of one or more patterns that are painted repetitively along the path to be stroked
- The width of each of the patterns. The width of the patterns determine the spacing between the patterns on the path.
- The option to split the pattern sequence or not. If a split is allowed, a path can end with only half of a pattern sequence. In some cases this might not be desirable, for example when the pattern sequence forms a word or a number.
- A fallback stroke that is used whenever the path is too detailed to use the patterns
- A tolerance that controls when to switch between patterns and the fallback stroke

Understanding the stroke algorithm

The stroke algorithm moves a pivot point over the segments of the path. Each time the pivot is moved, the algorithm draws the next pattern on the current pivot position, such that:

- the pivot point of the shape pattern (its origin) coincides with the current pivot position
- the base line of the shape pattern (the X axis) is aligned with the current path segment (unless the path segment is too short as described below).

The pivot is moved each time over a distance that corresponds to the width of the pattern that is being drawn.

Whenever the pattern width is too small to fit in (the last part of) the current path segment, the algorithm tries to fit the patterns as well as possible by cutting off the corners of the segments. You can specify a tolerance to control how large the cut-off error can be. If the error is too large, the fall-back stroke is used to paint the remaining part of the path segment. The pivot is then moved to the next segment and the algorithm tries to continue from the new start point. If the fall-back stroke is null, these parts of the path are not stroked. If the last part of the shape's path is too small to fit the next pattern, the fall-back stroke is also used to draw this part.

Figure 111 illustrates briefly how the algorithm works. Two patterns and a basic fallback stroke are defined. The first row with shapes shows how the algorithm works when the tolerance level is not exceeded. Note that one of the corners has been cut-off and that the last part of the path is drawn using the fallback stroke (shown here in a different color for clarity). The first image of the second row shows the errors that are introduced by cutting off corners. The last two images illustrate how these errors are avoided when the tolerance is decreased, by painting parts of the path using the fallback stroke.

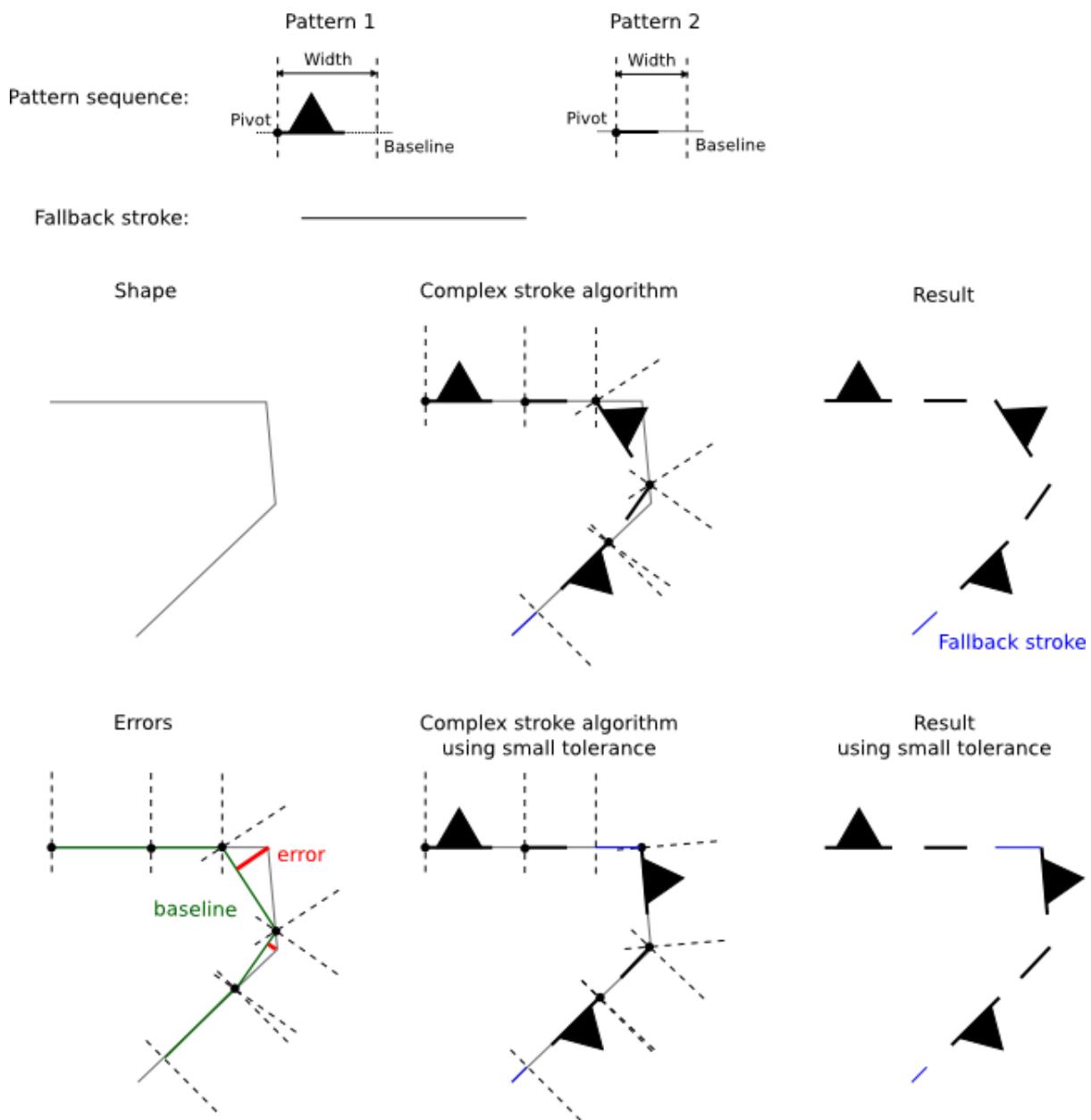


Figure 111 - Illustration of how the complex stroke algorithm works

32.7.2 Known issues and limitations

When drawing complex strokes, keep these known issues in mind:

- Strokes can only be drawn in the color that is configured on the `Graphics`. A multi-colored stroke can be achieved by defining one stroke for each color. Then paint the shape multiple times, each time with a different stroke and its corresponding color.
- The stroke algorithm does not know anything about the `Graphics` clip. Drawing very long lines exceeding the `Graphics` clip can result in very slow rendering performance, as the algorithm scans the full line, not just the visible part. In this case, consider using an additional constructor with a clip to improve performance.

CHAPTER 33

Asynchronous painting in a GXY view

Asynchronous painting is an important high-level application of concurrency. Painting data in a view can be slow for many reasons. Sometimes, simply retrieving the data takes a considerable amount of time because I/O is slow, when the data is retrieved across a network or from large databases, for example. In some cases, the data still has to be decoded on the fly, imposing additional overhead. And for some data types, the painting process itself is complex and relatively time-consuming. For example, painting large rasters that are warped on the fly is a non-trivial operation.

In simple applications, repainting components of the GUI is done synchronously, in a single thread. Since the view is also a GUI component, the entire user interface freezes until the view has been painted when it is updated. The experience of the end-user can be greatly enhanced if the user interface remains responsive during each repaint. This can be achieved by painting (parts of) the view in one or more background image buffers, in one or more separate threads. When the threads have finished painting, they can copy the buffers to the actual view. Even though the view may lag behind, the user can continue to interact with the rest of the user interface.

33.1 Creating an asynchronous painting wrapper

LuciadLightspeed provides support for asynchronous painting at the layer level. Selected layers or groups of layers can be painted in buffer images in different threads. If some buffer images are not up to date yet, a best effort is made to provide previews of their contents. The view is updated automatically whenever the final buffer images are ready.

The support for asynchronous painting is based on a wrapper for a standard `ILcdGXYLayer`. Whenever a layer is created for which the painting could be slow, the code can create an asynchronous wrapper, as shown in [Program 172](#). The wrapper can then be added to the view like any other layer.

```
1 TLcdGXYAsynchronousLayerWrapper asynchronousGeoTIFFLayer =  
2 new TLcdGXYAsynchronousLayerWrapper(geoTIFFLayer, null);
```

Program 172 - Creating an asynchronous layer for a slow layer
(from `samples/gxy/concurrent/painting/MainPanel`)

A wrapper always needs a paint queue, which manages one or more image buffers and a painting thread. The paint queue maintains an image buffer for each painting mode (bodies, labels,

selection, and so on). It keeps the images updated in its painting thread. In the event dispatch thread (AWT thread, Swing thread), it can then composite the created images, interleaving them with other layers, respecting the proper painting order. Paint queues can be automatically provided by a paint queue manager.

Program 173 shows how to create a default paint queue manager. The current paint queue and paint queue manager implementations support asynchronous painting on the following view implementations:

- `TLcdGXYViewJPanel`
- `TLcdGXYViewJPanelLightWeight`
- `TLcdGXYViewBufferedImage`, provided that it is painted in the event dispatch thread

```

1 TLcdGXYAsynchronousPaintQueueManager manager = new TLcdGXYAsynchronousPaintQueueManager
2   ());
manager.setView( getView() );

```

Program 173 - Creating a paint queue manager for a view
 (from samples/gxy/concurrent/painting/MainPanel)

Section 33.5 describes how to customize paint queue assignments.

Once you wrap a layer for asynchronous painting, the asynchronous painting thread can access the original layer anytime. It is thus unsafe to access the original layer (including its painters and label painters) from any other thread, unless you take specific precautions. The layer wrapper itself can be accessed from the event dispatch thread. Section 33.2 provides more information on how to access your original layer after it has been wrapped. See Chapter 9 for a more general overview of the concurrency best practices in LuciadLightspeed.

LuciadLightspeed offers the following asynchronous layer wrapper implementations:

- `TLcdGXYAsynchronousEditableLabelsLayerWrapper` is the best implementation to use with layers implementing `ILcdGXYEditableLabelsLayer`, such as `TLcdGXYLayer`. The wrapper ensures that all LuciadLightspeed functionality can transparently access the advanced labeling functionality offered by the wrapped layer.
- `TLcdGXYAsynchronousLayerTreeNodeWrapper` is best used with layers implementing `ILcdLayerTreeNode`. The wrapper ensures that all LuciadLightspeed functionality can transparently access the tree functionality offered by the wrapped layer. Use `TLcdGXYAsynchronousEditableLabelsLayerTreeNodeWrapper` if the layer tree node also implements `ILcdGXYEditableLabelsLayer`, such as `TLcdGXYLayerTreeNode`. Note that the wrapper only makes the layer itself paint asynchronously, not its children. If you want the children to be painted asynchronously, you can wrap them individually.
- `TLcdGXYAsynchronousLayerWrapper` can be used for layers that neither implement `ILcdGXYEditableLabelsLayer` nor `ILcdLayerTreeNode`.

33.2 Accessing asynchronously painted layers

The interface `ILcdGXYAsynchronousLayerWrapper` provides access to the standard properties of `ILcdGXYLayer`, which can be safely read and written from the Event Dispatch Thread (EDT). It also provides standard methods like `getBounds` and `applyOnInteract`. In the default implementation, these methods delegate to the wrapped layer. To ensure thread-safety,

they will block if the layer is being painted in the painting thread.

The layer interface `ILcdGXYLayer` also provides access to the painter and label painter for any given object. The default implementation of the asynchronous layer wrapper returns a clone of the painter returned by the inner layer. It blocks if the layer is being painted in the painting thread, in order to safely create the clone. The clone can then be used safely, assuming the painter class provides a sufficiently deep clone. If you use your own painters, you can use `ILcdDeepCloneable` to handle recursive cloning loops.

If an application wants to access the wrapped layer directly, the simplest way is to retrieve it from the layer wrapper, with `getGXYLayer`. It can then, for example, read the layer's properties. It is, however, not safe to modify the layer properties this way. Or more generally, it is not safe to invoke methods that are not thread-safe, since the layer may be painted in the painting thread of a paint queue.

A safe way to retrieve and to modify non-standard layer properties is to ask the layer wrapper to execute the modification code at a safe point in time, on an appropriate thread. The interface `ILcdGXYAsynchronousLayerWrapper` provides the methods `invokeAndWaitOnGXYLayer`, `invokeLaterOnGXYLayer`, `invokeLaterOnGXYLayerInEDT` and `invokeNowOnGXYLayer` for this purpose:

- The method `invokeAndWaitOnGXYLayer` executes a given runnable on the caller's thread when the paint queue is not painting. The method receives the wrapped layer as an argument and it can safely work on it (see Figure 112).
- The method `invokeLaterOnGXYLayer` executes a given runnable on the asynchronous painting thread. The method receives the wrapped layer as an argument. It can safely work on this layer, since the painting thread is obviously not painting at this point (see Figure 113). The method must not invoke any methods on the event dispatch thread with `SwingUtilities.invokeAndWait`, since this might result in dead-locks. Instead, use the method `invokeLaterOnGXYLayerInEDT` to safely execute a given runnable on the event dispatch thread.
- The method `invokeNowOnGXYLayer` executes a given runnable on the caller's thread, much like `invokeAndWaitOnGXYLayer`. However, instead of waiting for any paint operations to finish, it can invoke the method while painting operations are in progress. The runnable can therefore only safely invoke methods that do not interfere with the `paint` method (see Figure 114).

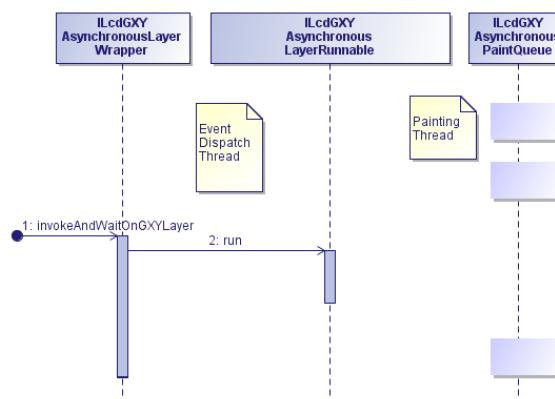


Figure 112 - The method `invokeAndWaitOnGXYLayer` executes a runnable on the caller's thread, in between painting operations of the paint queue on the painting thread

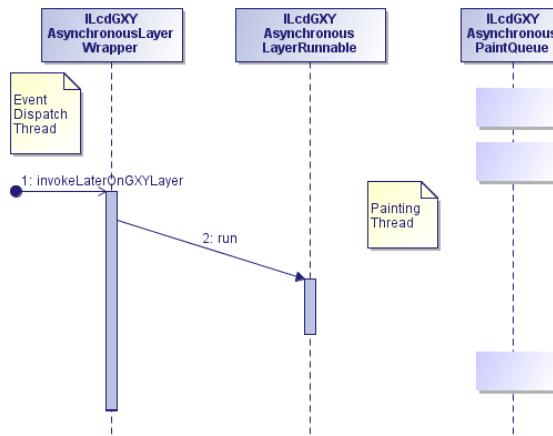


Figure 113 - The method `invokeLaterOnGXYLayer` executes a runnable on the painting thread, in between painting operations of the paint queue

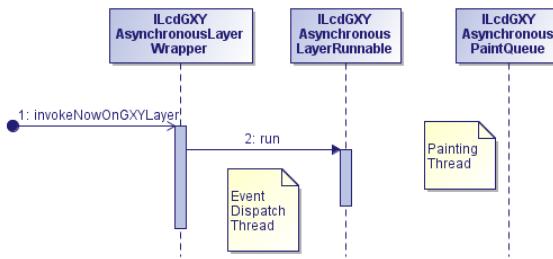


Figure 114 - The method `invokeNowOnGXYLayer` executes a runnable on the caller's thread, possibly overlapping with painting operations of the paint queue

Section 33.6 explains another, more advanced alternative to accessing the wrapped layer's functionality: extending the layer wrapper.

33.3 Editing the model of a wrapped layer

When editing the model of a wrapped layer you need to do this in a thread-safe way since the model is being painted in a separate painting thread. The layer wrapper and the paint queue guard any read access by means of read locks on the model, as described in Section 33.2. Any code that modifies the model or its elements therefore has to make sure that it obtains a write lock on the model, as shown in Chapter 9.

33.4 Memory and performance considerations

Since each image buffer has the same size as the view, the required amount of memory can be considerable. It can therefore be useful to share paint queues between asynchronous layer wrappers. The contents of the actual layers are then painted in the same threads and in the same image buffers. Shared paint queues also reduce view repaints for the compositing created images, which can be important when the view contains slow, synchronously painted layers.

On the downside, sharing a paint queue implies that view repaints are as slow as the combined repaints of the layers. Layers that share a paint queue must also be consecutive. Otherwise their image buffers cannot be composited properly in the view. Section 33.5 explains how to simplify

the non-trivial task of assigning paint threads and buffers to asynchronously painted layers by using a paint queue manager.

A paint queue also has a flag to determine whether it should create image buffers for all painting modes or not. It is possible to reduce the number of buffers in order to reduce the memory usage. Depending on the type of layer, this comes at the cost of some performance or quality. By default, the value `EVERYTHING` is specified. The paint queue then creates image buffers for all painting modes. The value `BODIES_AND_SKIP` instructs the paint queue to only create an image buffer for painting the bodies of the model elements. The labels and selection are painted synchronously, but only when the image buffer is ready. To avoid the continuous appearing and disappearing of the labels and selection in this mode, a minimum waiting delay can be specified using `setSkipDelay`.

33.5 Customizing paint queue assignments

A paint queue manager can greatly simplify the non-trivial task of assigning paint threads and buffers to asynchronously painted layers. LuciadLightspeed offers two ways to customize a paint queue manager. The easiest way is to use the default paint queue manager, `TLcdGXYAsynchronousPaintQueueManager`, with a custom paint hint provider. Paint hints are a powerful way to specify how to paint a layer and whether or not the layer can share a paint queue with its neighboring layers. If you want more flexibility, you can choose to directly subclass `ALcdGXYAsynchronousPaintQueueManager`, and manage the assignments yourself.

33.5.1 Observing paint queue assignments

You can visually examine how layers are mapped to paint queues by enabling a debug feature in `TLcdGXYBusyLayerTreeCellRenderer`. The class javadoc explains how to make it use separate colors for asynchronous layers with separate paint queues.

33.5.2 Using paint hints

The default paint queue manager, `TLcdGXYAsynchronousPaintQueueManager`, takes its decisions based on paint hints. With a paint hint, you can specify asynchronous painting settings for a specific layer, such as the thread priority and whether or not labels should be painted asynchronously. A paint hint also allows you to enforce that certain layers are never painted in the same thread.

The paint queue manager takes the paint hint settings into account to decide whether or not layers can share the same paint queue. For this purpose, a paint hint contains properties that can or cannot be combined. As an example, consider the following layers:

layer A a fast vector layer, with important labels

layer B another fast vector layer, with unimportant labels

layer C a slow raster layer, without labels

Consider the following painting requirements:

- fast layers should be rendered in the same paint thread, to minimize overhead
- fast and slow layers may not be rendered in the same paint thread, to ensure timely view updates
- important labels should always be visible

- it does not matter if unimportant labels are always visible or not

This translates into the following paint hints:

layer A “painting group” == “fast”, “painting mode”== EVERYTHING. In other words: a layer can use the same paint queue as this layer if the latter also has the property value “fast” for the “painting group” and if the “painting mode” property is EVERYTHING.

layer B “painting group” == “fast”, “painting mode”== EVERYTHING or BODIES_AND_SKIP.

In other words, a layer can use the same paint queue as this layer if the latter also has the property value “fast” for the “painting group” and if the “painting mode” property is EVERYTHING or BODIES_AND_SKIP.

layer C “painting group” == “slow”, “painting mode”== EVERYTHING or BODIES_AND_SKIP.

In other words, a layer can use the same paint queue as this layer if the latter also has the property value “slow” for the “painting group” and if the “painting mode” property is BODIES_AND_SKIP or EVERYTHING.

To further illustrate the use of paint hints, the class `PaintTimeBasedAsynchronousPaintQueueManager` shows how to assign paint queues with paint hints, based on how long it takes to paint the layers.

33.5.3 Performing your own assignments

By extending `ALcdGXYAsynchronousPaintQueueManager` you have the opportunity to focus on paint queue assignments without having to worry about whether layers are contiguous, or how moving layers around affects the assignments.

This paint queue manager divides the view into a number of *paint blocks*. If a paint block is asynchronous, its layers will always share the same paint queue. The paint queue manager offers methods to split up and merge paint blocks. To implement your own manager, you only need to implement the `evaluateModifiedPaintBlocks` methods and split up and/or merge the relevant paint blocks. As an example, the class `FixedCountPaintQueueManager` shows how to implement a manager that uses a fixed number of paint threads.

33.6 Providing transparent access to a custom layer interface

If your application contains an extension of `ILcdGXYLayer`, and you do not want to use a blocking method to change its non-standard properties, you can extend the layer wrapper to implement your `ILcdGXYLayer` extension. The layer wrapper can then offer the same additional properties in its interface. To do this, the layer wrapper extension must provide an implementation of `ILcdGXYLayerChangeTracker` which ensures that the additional properties remain consistent between the layer wrapper and the wrapped layer.

The asynchronous layer wrapper isolates the wrapped layer from the outside world. This way, the program can change properties on the layer wrapper at any time in one thread, for example the Event Dispatch Thread (EDT). At the same time, the paint queue paints the wrapped layer in its painting thread. Neither thread has to wait for the other one. The paint queue synchronizes the properties of the layer wrapper and the properties of the wrapped layer at the appropriate and safe points in time.

The default implementation of `ILcdGXYAsynchronousLayerWrapper`, `TLcdGXYAsynchronousLayerWrapper`, wraps basic implementations of `ILcdGXYLayer`, like `TLcdGXYLayer`. The synchronization of properties between the layer wrapper and the wrapped layer is the responsibility of implementations of `ILcdGXYLayerChangeTracker`.

Figure 115 shows a sequence diagram of this mechanism. The `paint` method of the paint queue first propagates any changes from the layer wrapper to the wrapped layer. It then queues the actual painting operation on the painting thread, so it can return right away. Note that the paint queue can be asked to wait a while for the asynchronous painting to complete, so that the results are painted immediately. This behavior is set using the method `setSynchronousDelay`. After the painting operation is queued, the wrapped layer is painted in an image buffer in the painting thread. When the painting is done, any changes (for example, label positions) are propagated from the wrapped layer back to the layer wrapper. Finally, the paint queue invalidates the view, so the newly created image can be blitted to the view on the next repaint.

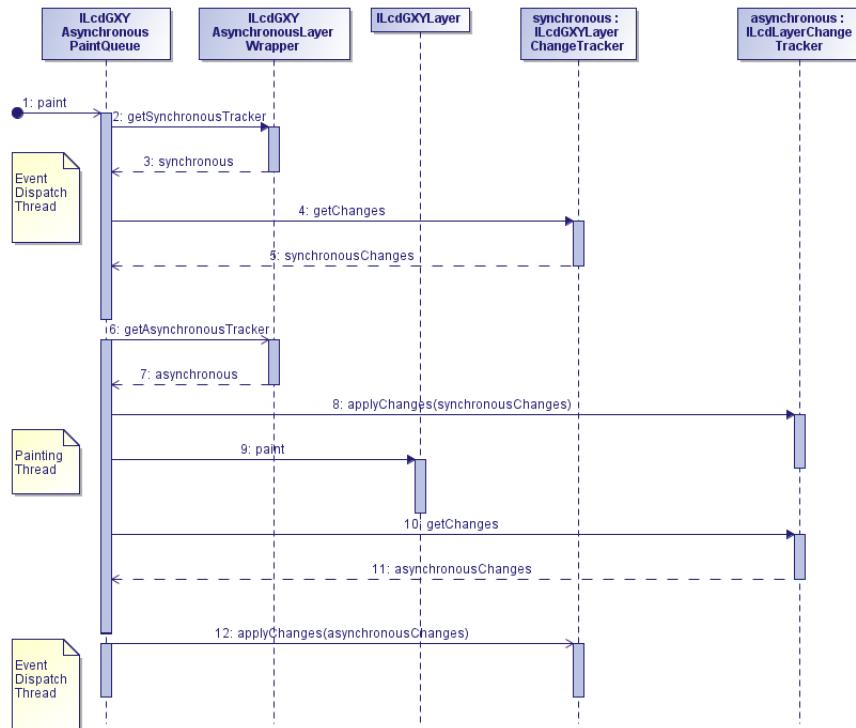


Figure 115 - Using an `ILcdGXYLayerChangeTracker`

The class `TLcdGXYAsynchronousLayerWrapper` provides an implementation that keeps the basic layer properties consistent.

An example of an asynchronous layer wrapper providing transparent access to another layer interface is `TLcdGXYAsynchronousEditableLabelsLayerWrapper`, which can wrap implementations of `ILcdGXYEditableLabelsLayer`. In addition to the basic layer properties, it also manages the synchronization of the label positions, from the wrapper layer to the layer with editable layers, but also the other way round, whenever the label painter has updated the label positions.

33.7 Troubleshooting

Developing and debugging multi-threaded applications can be difficult and error-prone. When using and extending the asynchronous painting API, the following typical problems may occur:

- Refer to `TLcdGXYBusyLayerTreeNodeCellRenderer` to visually examine how layers are mapped to paint queues.

- If an asynchronous layer wrapper is not painted asynchronously, you should check if the layer wrapper has a paint queue. Without a paint queue, the layer wrapper just paints synchronously. This only applies to situations in which no paint queue manager is used, which is normally not the case.
- If your code ends up in a dead-lock, you should check if the paint methods of your layers and painters do not wait on the event dispatch thread, for example for popping up a menu. The event dispatch thread itself may already be waiting on the painting thread, so the additional wait would result in a dead-lock. Similarly, runnables that are scheduled to be invoked on the painting thread must not wait on the event dispatch thread.
- If the asynchronous painting mechanism gets stuck in an infinite painting loop, you should check if you have written any state trackers that propagate properties incorrectly. Notably, they should only propagate actual changes to properties. Furthermore, they must not track and then propagate the changes that they are propagating themselves.
- If you see drawing anomalies, synchronous and asynchronous drawing code are probably sharing objects which they should not share. Make sure that your wrapped layer is not referenced by, for example, a controller. Instead, pass the layer wrapper. Another possible cause could be incorrect clone implementations of layer painters and/or the layer pen. Their clones should be sufficiently deep so that they can be safely accessed in another thread.
- If you edit your model, make sure that you lock it as described in [Section 33.3](#).

CHAPTER 34

Visualizing large vector data sets on a GXY view

The challenges and solutions for visualizing large data sets on a GXY view are identical to the ones for the visualization on a Lightspeed view. See [Chapter 24](#) for more information.

CHAPTER 35

Implementing a painter/editor in a GXY view

This chapter provides an in-depth explanation of the interfaces `ILcdGXYPainter` and `ILcdGXYEditor`, and introduces some patterns to implement these interfaces. The first part (Section 35.1) introduces a new shape, the hippodrome, and a set of requirements on how the shape should be handled. The second part (Section 35.2 through Section 35.5) focuses on explaining the interfaces `ILcdGXYPainter` and `ILcdGXYEditor`. The third part (remaining sections) demonstrates how the requirements are met by implementing the interfaces `ILcdGXYPainter` and `ILcdGXYEditor` for the hippodrome shape.



Note that the hippodrome can be modeled using predefined shapes, such as an `ILcdGeoBuffer` or `ILcdCompositeCurve`. The purpose of this chapter is however to show how to write a painter for a new shape.

35.1 Introducing a new shape: the hippodrome

35.1.1 IHippodrome, definition of a new shape

Suppose a new shape should be displayed on the map with the following requirement: it consists of two half circles connected by two lines. Or, more graphically, it would have to look like Figure 116.



Figure 116 - Hippodrome graphical specification

Three basic elements of the shape can be derived: the two points around which the path turns and the radius of the turn around those points. Since the radius of the arcs is equal to the width of the hippodrome, it is referred to as the width¹.

The `IHippodrome` thus contains three methods which define the shape:

¹Section 35.1.2 demonstrates that this is not necessarily true.

- `getWidth` returns the width of the hippodrome, W in Figure 117
- `getStartPoint` returns one of the turning points, $P1$ in Figure 117
- `getEndPoint` returns the other turning point, $P2$ in Figure 117

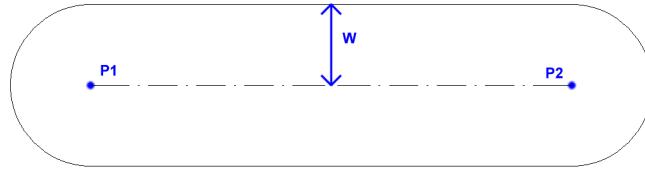


Figure 117 - Defining elements of an hippodrome

`IHippodrome` extends `ILcdShape`. While this is not required, it allows to take full advantage of the API, for example, `IHippodrome` can then be in a `TLcd2DBoundsIndexedModel` instead of in a `TLcdVectorModel`. `ILcdShape` contains methods to return the bounds of an object and to check if a point or a location is contained in the object. Since the hippodrome has quite a simple geometry, implementing these methods is straightforward.

35.1.2 LonLatHippodrome and XYHippodrome

`IHippodrome` does not define how the points and the width should be interpreted. This is defined by the implementations: `LonLatHippodrome` for geodetic reference systems and `XYHippodrome` for grid reference systems.

The major difference is the path between the start point and the end point. For an `XYHippodrome` the path between the start and end point is a straight line on a Cartesian plane. For a `LonLatHippodrome` the path between the start point and the end point is situated on an ellipsoid and is therefore a geodesic line. The consequences are:

- The angle from start to end point and the angle from end point to start point differs for a `LonLatHippodrome`
- Since the angles are different, calculating the location of the points at the ends of the arcs requires different computations for `LonLatHippodrome` and `XYHippodrome`
- A `LonLatHippodrome` is situated on the ellipsoid. When the arcs are connected with geodesic lines on the ellipsoid, the width of the `IHippodrome` will not be a constant along these lines, and the connection of the arc with the line is not C1 continuous



The classes described here are only provided as a sample. If you need a hippodrome implementation that does not have the limitations described above, see `ILcdGeoBuffer` and `TLcdLonLatGeoBuffer`.

Including the methods to retrieve the angles and the contour points in the interface hides these differences and reduces the complexity of the `ILcdGXYPainter` and `ILcdGXYEditor`. The interface `IHippodrome` therefore also has the following methods:

- `getContourPoint` returns the point at the start or the end of an arc
- `getStartEndAzimuth` returns the `azimuth`, or angle of the line from the start point to the end point, at the start point
- `getEndStartAzimuth` returns the angle of the line from the end point to the start point, at the end point

These methods return elements which can be derived from the three defining elements. Making them available in the interface hides the complexity of computing their values. Adding these methods also facilitates caching these values, which need only be recomputed when one of three defining elements changes.

35.1.3 Representations of shapes

Although the requirements above lead to the three elements constituting the `IHippodrome` shape, these elements could also be rendered differently, for example, see [Figure 118](#) and [119](#). The implementation of the `ILcdGXYPainter` for the `IHippodrome` defines how the object is rendered.

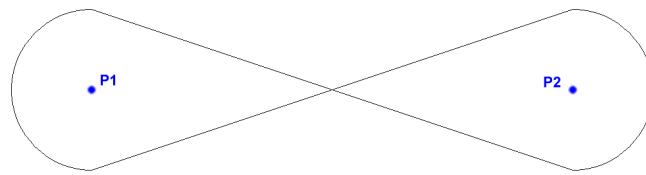


Figure 118 - Alternative rendering for an `IHippodrome`



Figure 119 - Alternative rendering for an `IHippodrome`

35.1.4 Creating an `IHippodrome`

Next to visualizing an `IHippodrome`, it should also be possible to create an `IHippodrome` graphically. The `IHippodrome` should be created in three different steps, which determine:

1. the location of the first point
2. the location of the second point
3. the width

After the location of the first point has been specified by a mouse click, or by touch input for example, a line should be drawn between the first point and the location of the input pointer, the mouse pointer for example. The line indicates where the axis of the `IHippodrome` is located. Once the second point has been specified, an `IHippodrome` should be displayed. The hippodrome width is such that the hippodrome touches the input pointer.

To allow setting the location of the points and setting the width of the `IHippodrome`, the interface has two methods:

- `moveReferencePoint` moves one of the turning points
- `setWidth` changes the width of the hippodrome

35.1.5 Editing an IHippodrome

It should be possible to edit an `IHippodrome`, that is to change the values of one (or more) of its defining elements. Fulfilling the creation requirements already provides methods to do this. On top of that, one should be able to move the `IHippodrome` as a whole. This requirement is covered by making `IHippodrome` extend `ILcd2DEditableShape`.

35.1.6 Snapping to/of an IHippodrome

Creating or editing an `IHippodrome` graphically is not very accurate since it is limited to the length one pixel on the screen corresponds to. Though that length can be calculated very accurately, changing for example the location of the first point of the `IHippodrome` is only possible in steps of that length. Therefore two requirements are posed:

- It should be possible to create or edit an `IHippodrome` so that its start point or end point coincides with a point of another shape on the map.
- Conversely, it should be possible for another shape, when it is being edited or created to have one or more of its points coincide with the start point or the end point of the `IHippodrome`.

The process of making the points coincide is called snapping. No methods should be added to the `IHippodrome` to comply with this requirement, since moving the points of the `IHippodrome` is already possible. It is up to the actor who controls the move to ensure that the move results in coincident points.

35.2 The ILcdGXYPainter interface explained

Chapter 32 explains how a layer is rendered by selecting the objects that need to be painted, retrieving the painter for those objects one by one, and instructing the painter to render the object in the view. The painter may be instructed to paint the object differently depending on the circumstances: an object may have been selected, or it is ready to be edited, These circumstances are expressed in the painter modes. These modes pertain to all `ILcdGXYPainter` implementations and should not be confused with style modes which pertain to single `ILcdGXYPainter` implementations, like `TLcdGXYPointListPainter.POLYGON`.

35.2.1 ILcdGXYPainter modes

The interface `ILcdGXYPainter` defines eight painter modes which can be divided into three groups depending on their semantics:

Status of the object in the view

- BODY: the basic mode, indicates that the body of the object should be painted.
- HANDLES: indicates that the handles of the object should be painted. The handles are marker points on the representation of an object. They can be used for editing.
- SNAPS: indicates that targets for snapping should be highlighted. Snapping is a mechanism to ensure that multiple shapes have points at the exact same location. This mechanism is further explained in [Section 35.5](#).

Selection status of the object in a layer

Depending on the selection status of an object in a layer it is rendered differently.

- **DEFAULT:** the default, not selected state.
- **SELECTED:** indicates that the object should be rendered as selected. This usually implies some kind of highlighting.

User interaction

Through the `TLcdGXYNewController2` and the `TLcdGXYEditController2` the user can interact with the objects in a model. The former creates a new object, the latter edits it. A `TLcdGXYEditController2` calls an `ILcdGXYPainter` with the following modes while in the process of editing a shape, so that the user gets visual feedback about the execution of the editing operation:

- **TRANSLATING:** the first editing mode. This mode is typically used for repositioning a shape as a whole.
- **RESHAPING:** the second editing mode. This mode is typically used when a part of the shape is being adapted.

While the names of the editing modes, TRANSLATING and RESHAPING, indicate what action could be undertaken on the object being edited, they should not be interpreted as exact definitions but rather as guidelines on how the object should be edited. An example can clarify this: when translating one `ILcdPoint` in an `ILcdPointList`, the `TLcdGXYEditController2` calls the `TLcdGXYPointListPainter` with the TRANSLATING mode. However, one could also argue that the shape as a whole is being *reshaped*.

The `TLcdGXYNewController2` calls an `ILcdGXYPainter` with the CREATING mode when a shape is in the process of being created:

- **CREATING:** creation mode. While creating a new shape, the user should see the current status of the shape.

35.2.2 `ILcdGXYPainter` methods

The `ILcdGXYContext` parameter

`ILcdGXYContext` is a utility interface to hold all information necessary to render an object. It is passed in methods which require this information, such as `paint`, `boundsSFCT` and `isTouched`. It is also passed to some `ILcdGXYEditor` methods. It contains:

- The `ILcdGXYView` where the drawing or editing occurs. This can be helpful for, for example, retrieving the scale. Some objects might be painted differently depending on the scale.
- the `ILcdGXYLayer` and therefore the `ILcdModel` the object to be painted belongs to. From the `ILcdModel`, you can retrieve the model's reference, which might be useful when not all operations can be abstracted into the shapes interface. Calculating the distance (in meters) of a point to the axis of an `IHippodrome` is an example of such an operation: it depends on the model reference the `IHippodrome` is defined in, though it can hardly be regarded as a functionality that should be part of `IHippodrome` as a shape.
- The `Graphics` on which to perform the rendering.
- The `ILcdModelXYWorldTransformation` and the `ILcdGXYViewXYWorldTransformation`. Both transformations are required to determine the exact location of the mouse in model coordinates or to define the location in the `ILcdGXYView` component where to draw an object. For a detailed explanation, refer to [Section 44.4](#).

- The current location of the mouse and its translation (for example, mouse drag) in view coordinates when this information is required. This is useful to interpret the user's last interaction.
- Snapping information: the snap target and the layer the snap target belongs to. This is explained in [Section 35.5](#).
- An `ILcdGXYPen` to support drawing and editing on a `Graphics`. The pen provides an abstraction of low level rendering operations, which hides some of the complexities depending on model references.

`TLcdGXYContext`, the main implementation of `ILcdGXYContext`, provides support for multiple input points and multiple snapping targets. This is for example used when handling multi-touch input.

paint

This method is responsible for rendering the object in the view in all possible modes. This includes painting the object while the user is interacting with the object using controllers; in that case controllers typically render a preview of what the result of the interaction would be using the `ILcdGXYPainter`. Only at the end of the user interaction (usually when the mouse is released, or when a touch point is removed) the `ILcdGXYEditor` is called to perform the actual editing on the object. This usually implies a tight coupling with the `ILcdGXYEditor`, which is the main reason why most `ILcdGXYPainter` implementations in LuciadLightspeed also implement `ILcdGXYEditor`.

boundsSFCT

`boundsSFCT` adapts a given bounds object so that it reflects the bounds of the shape when it would be rendered given the `ILcdGXYContext` passed. This is called the view bounds of the object. The view bounds' coordinates are view coordinates, expressed in pixels.

`boundsSFCT` can be called when checking if an object should be painted. For performance reasons an `ILcdGXYView` wants to paint as little objects as possible. To select the objects that should be painted, two mechanisms exist, based on the model contained in a layer.

- When an `ILcdGXYLayer` has an `ILcd2DBoundsIndexedModel`, its objects are required to be `ILcdBounded`. By transforming the `ILcdGXYView` boundary, the view bounds, to model coordinates, it is straightforward to check if an object will be visible in the `ILcdGXYView` or not.
- When the `ILcdGXYLayer` contains another type of `ILcdModel` this mechanism cannot be applied. The `ILcdGXYView` then relies on the `ILcdGXYPainter` for an object to provide reliable view bounds of the object. These bounds are then compared to the view's view bounds, which amounts to comparing it to the view's size.

The latter mechanism implies that this method could be called frequently, for every paint for every object in the `ILcdModel`, and that its implementation should be balanced between accuracy and speed. An accurate implementation ensures that the object will not be painted if it is outside the `ILcdGXYView` boundary. If the shape is complex, however, it might be costly to compute the object's view bounds accurately and it may be more economic to have a rough estimate of the object's view bounds. [Figure 120](#) illustrates a view bounds and the model bounds of an `IHippodrome`.

The object's view bounds also depend on the `ILcdGXYPainter` mode passed: `boundsSFCT` should for example, take into account user interaction with the shape. [Figure 121](#) illustrates

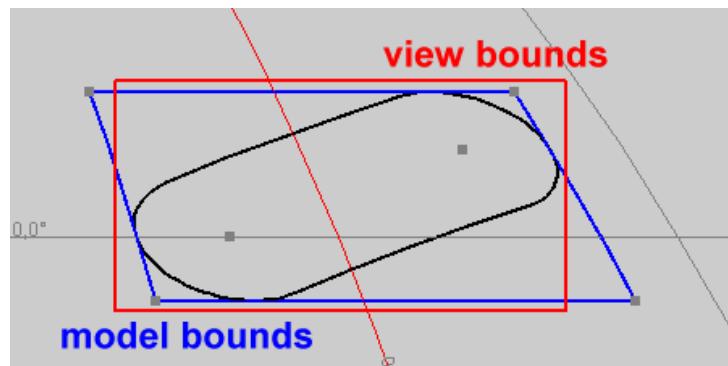


Figure 120 - Model bounds and view bounds of an IHippodrome

how the view bounds changes when the shape is being moved. Remember that the `ILcdGXYPainter` only depicts a preview of the result of the user interaction. The view bounds returned by the `ILcdGXYPainter` return the bounds of the object rendered *as if the user interaction had already taken place*. The object's model bounds do not change as the object itself is not changed.

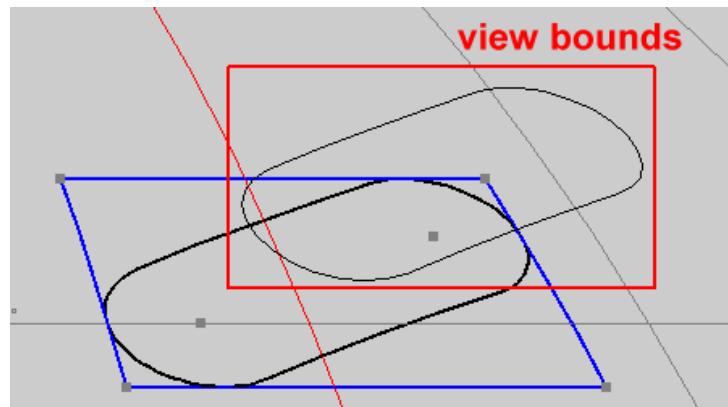


Figure 121 - Model bounds and view bounds of a translated IHippodrome

isTouched

This method checks if the object's representation is touched by the input. The input location and displacement typically need to be taken into account. They can be retrieved from the `ILcdGXYContext` passed. User interaction often needs to be interpreted locally, that is, on the object that is touched by the mouse. This method is generally implemented with greater accuracy than the `boundsSFCT` method. It may include a call to the `boundsSFCT` method for performance reasons: if the input point is not within the bounds of the object's representation, it is surely not touching the object. Though this method does not specify which part of the shape is being touched, that information may be made public in implementation specific methods.

getCursor

The cursor provides a visual indication of the user interaction with the shape. Since there are two editing modes and a creation mode, displaying a different cursor per mode gives an indication of what to expect when interacting with the shape.

35.3 The `ILcdGXYEditor` interface explained

The `ILcdGXYEditor` executes in the model space what the `ILcdGXYPainter` displays in the view space. As the `ILcdGXYEditor` is called by `TLcdGXYEditController2` and `TL-`

`cdGXYNewController2` to execute the user interaction on the shape, it is less complex than the `ILcdGXYPainter` interface.

35.3.1 `ILcdGXYEditor` modes

Analogously to the `ILcdGXYPainter`, the `ILcdGXYEditor` defines a number of modes. Depending on the mode, the `ILcdGXYEditor` should react differently to user interaction. Since the `ILcdGXYPainter` provides a preview of what the `ILcdGXYEditor` will do, the modes are closely linked to the `ILcdGXYPainter` modes.

Creation modes

Since the creation process can be complex and additional interaction may be required at the beginning or at the end of the creation process, three creation modes are provided:

- `START_CREATION`
- `CREATING`
- `END_CREATION`

Editing modes

Two editing modes are provided, the counterparts of the `ILcdGXYPainter` modes `RESHAPING` and `TRANSLATING`:

- `RESHAPED`
- `TRANSLATED`

The names of the `ILcdGXYEditor` modes indicate that the `ILcdGXYEditor` is responsible for processing the result of the user interaction, whereas the `ILcdGXYPainter` is responsible while the user interaction is taking place.

35.3.2 `ILcdGXYEditor` methods

`edit`

This method carries out the change on the shape requested by the user. It retrieves the information of the user interaction (being the input position and displacement) from the `ILcdGXYContext`, and the intention of the change by interpreting the mode. To ensure an intuitive user experience, the result of the change should be a shape that resembles the one rendered by the corresponding `ILcdGXYPainter` during the user interaction.

`getCreationClickCount`

The `ILcdGXYEditor` creation mode passed to the `ILcdGXYEditor` depends on the value returned by the `getCreationClickCount` method. Controllers may keep a log of the number of clicks and compare that to the creation click count.

35.4 Common methods in `ILcdGXYPainter` and `ILcdGXYEditor`

Both interfaces share a number of methods, which are mainly utility methods.

- `setObject` and `getObject` set and return the object the `ILcdGXYPainter` (resp. `ILcdGXYEditor`) should be painting (resp. editing).

- `addPropertyChangeListener` and `removePropertyChangeListener` enable other classes to be notified of changes in a number of properties of the `ILcdGXYPainter` or `ILcdGXYEditor`. A typical example is a map legend implementation which is notified of changes in the color a painter uses to paint an object.
- `getDisplayName` provides a textual representation of the `ILcdGXYPainter` or the `ILcdGXYEditor`.

35.5 Snapping explained

Snapping is an interaction process with three actors: an `ILcdGXYPainter`, an `ILcdGXYEditor` and either the `TLcdGXYEditController2` or the `TLcdGXYNewController2`. The objective is to visually edit or create a shape so that part of it is at the exact same location as part of another shape. The shapes do not share that part to ensure that consequent editing of either of the shapes does not influence the other shape.

35.5.1 Snapping methods in `ILcdGXYPainter`

supportSnap

`supportSnap` returns if it is possible to snap to (a part of) the shape that is rendered by this `ILcdGXYPainter`. If snapping is supported, the `paint` method should provide the user a visual indication whenever a snap target is available.

snapTarget

The snap target is the part of the shape that is made available for snapping. This may change depending on the context passed. Most implementations return only that part of the shape that is being touched given the `ILcdGXYContext` passed in this method, for example the `TLcdGXYPointListPainter` returns the `ILcdPoint` of the `ILcdPointList` which is touched by the mouse pointer.

35.5.2 Snapping methods in `ILcdGXYEditor`

acceptSnapTarget

The user may not want the shape being edited or created to snap to any target provided by a painter. `acceptSnapTarget` enables filtering on the objects returned as snap target. Care should be taken that the snap target returned by an `ILcdGXYPainter` is not necessarily an object contained in a model for example, it might be just one `ILcdPoint` of an `ILcdPointList`. As such it might not be possible to check for example, on the features of the object that contains the snap target.

35.5.3 Use of snapping methods

The snapping methods are called by the `TLcdGXYNewController2` and the `TLcdGXYEditController2`. Both controllers contain an `ILcdGXYLayerSubsetList`, a list of objects sorted per `ILcdGXYLayer`, called the snappables. Only objects in the snappables are passed to the `ILcdGXYEditor`.

35.5.4 What are the steps in the object snapping process?

Figure 122 shows the sequence of method calls in an `ILcdGXYView` where an object B is in the process of being edited or created and the mouse is moving near an object A. The parties involved are:

- the `TLcdGXYNewController2` or `TLcdGXYEditController2` active on the view, referred to as `Controller`. It contains the object A in its snappables.
- the `ILcdGXYPainter` for the object A, an object beneath the current input point
- the `ILcdGXYEditor` for the object B, an object which is in the process of being created or edited

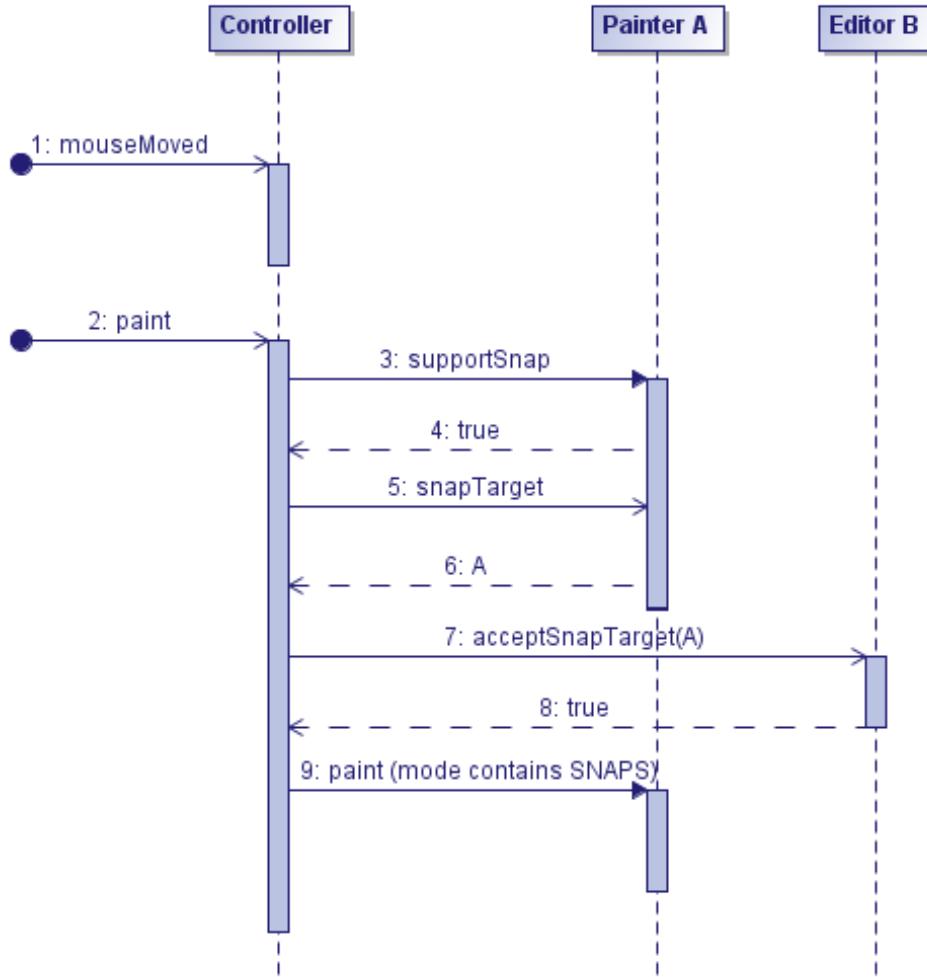


Figure 122 - Snapping: sequence of calls when moving the mouse

When the input point(s) move(s) (as illustrated by the `mouseMoved` call in Figure 122), the `Controller` interprets the move(s) and then requests the `ILcdGXYView` it is active on to repaint itself. At the end of the paint cycle, after all `ILcdGXYLayer` objects have been rendered, the method `paint` of `Controller` is called. The `Controller` traverses its snappables, and checks:

1. If the `ILcdGXYPainter` supports snapping for that object in that layer
2. If the `ILcdGXYPainter` returns a snap target in the given context

until a snap target, in this case A, is returned. If the snap target is accepted by the object B editor, the object A is painted with the `SNAPS` mode, indicating that it is available for snapping.

Figure 123 shows the sequence of method calls when object B is edited by snapping to object A. The process is initiated by a `mouseReleased` call on the `Controller`. The `Controller`

performs the same checks as above, but when the snap target is accepted by the object B `ILcdGXYEditor` it sets the snap target (and its layer, not shown here) to a Context, which is then passed to the `ILcdGXYEditor` for editing.

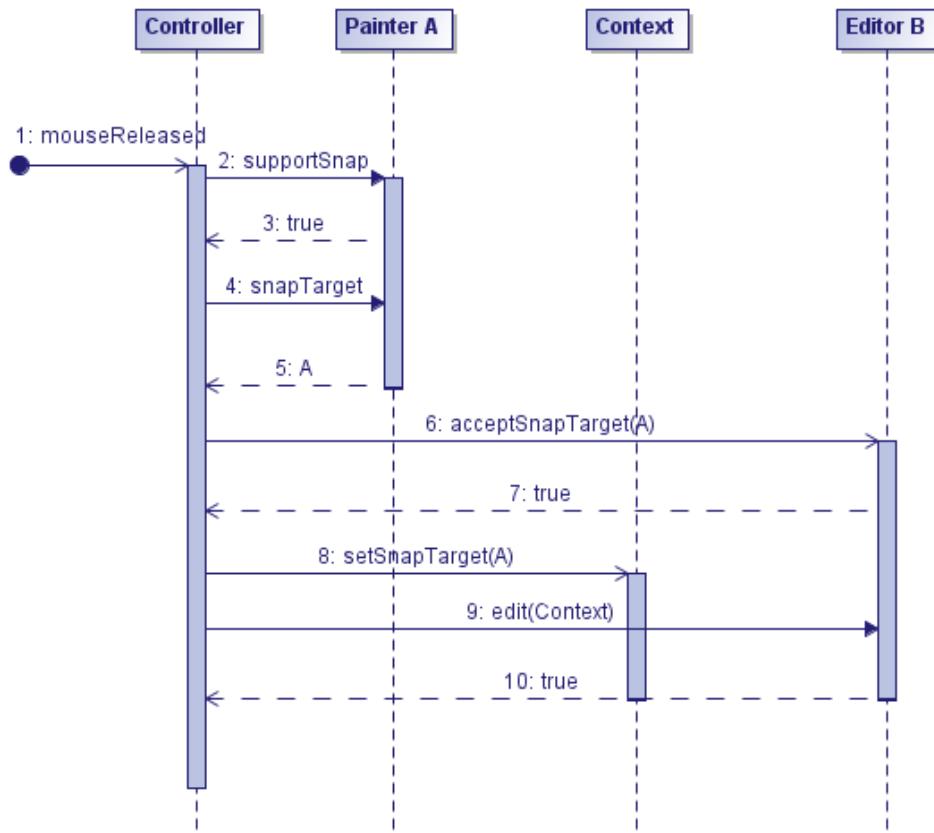


Figure 123 - Snapping: sequence of calls when releasing the mouse

35.5.5 Supporting multiple model references

Two shapes that are at the exact same location do not necessarily have their coordinates expressed in the same model reference. This should be taken into account when snapping to a shape: appropriate transformations should be applied to compute the new coordinates of the point in the snapping shape's reference. The information required to set up the transformations is contained in the `ILcdGXYContext` passed to the `edit` method.

35.6 Painting an IHippodrome

This section covers rendering the body of an `IHippodrome` in a basic state, that is, not being edited or created.

35.6.1 Constructing an `ILcdAWTPPath` with an `ILcdGXYPen`

Objects to be rendered are expressed in model coordinates, while rendering on the `Graphics` requires view coordinates. The interface `ILcdGXYPen` provides basic rendering operations (for example, drawing lines, arcs) using model coordinates while hiding the complexity of coordinate transformations.

An `ILcdAWTPath` defines a path in AWT coordinates. It has useful methods for implementing an `ILcdGXYPainter`. An `ILcdAWTPath` is capable of:

- rendering itself as a polyline or a polygon on a `Graphics`
- calculating the enclosing rectangle, in view coordinates
- detecting if it touches a given point, given in view coordinates

Many `ILcdGXYPainter` implementations construct an `ILcdAWTPath` using an `ILcdGXYPen` and then instruct the AWT path to render itself on the `Graphics`.

35.6.2 An `ILcdAWTPath` for an `IHippodrome`

Program 174 shows how the `ILcdAWTPath` of an `IHippodrome` is constructed. The `ILcdAWTPath` is cleared using the `reset` method, and is then constructed in four steps:

1. the line between the start-upper and end-upper point is added to the `ILcdAWTPath`
2. the arc around the end point is added to the `ILcdAWTPath`
3. the line between the end-lower and start-lower point is added to the `ILcdAWTPath`
4. the arc around the start point is added to the `ILcdAWTPath`

Every step checks if the previous step caused a `TLcdOutOfBoundsException`. If so, the `ILcdGXYPen` moves to the next point that would not cause a `TLcdOutOfBoundsException` and resumes constructing the `ILcdAWTPath`. Note that by using the `ILcdGXYPen`, adding a line or an arc, expressed in model coordinates, to the `ILcdAWTPath` is very simple.

```

1  private void calculateContourAWTPathSFCT(IHippodrome aHippodrome,
2  											 ILcdGXYContext aGXYContext,
3  											 ILcdAWTPath aAWTPathSFCT) {
4
5  	ILcdGXYPen pen = aGXYContext.getGXYPen();
6  	ILcdModelXYWorldTransformation mwt = aGXYContext.getModelXYWorldTransformation();
7  	ILcdGXYViewXYWorldTransformation vwt = aGXYContext.getGXYViewXYWorldTransformation();
8
9  	// Resets the current position of the pen.
10  pen.resetPosition();
11
12  // reset the AWT-path
13  aAWTPathSFCT.reset();
14
15  // a boolean to check if a part of the hippodrome is not visible in the current world
16  // reference.
17  // we try to paint as much as possible.
18  boolean out_of_bounds = false;
19  try {
20
21  	// moves the pen to the startUpper-point of the hippodrome
22  	pen.moveTo(aHippodrome.getContourPoint(IHippodrome.START_UPPER_POINT), mwt, vwt);
23
24  	// append a line to the endUpper-point
25  	pen.appendLineTo(aHippodrome.getContourPoint(IHippodrome.END_UPPER_POINT),
26  					mwt, vwt, aAWTPathSFCT);
27
28  } catch (TLcdOutOfBoundsException ex) {
29  	out_of_bounds = true;
30  }
31
32  try {
33  	if (out_of_bounds) {
34  	pen.moveTo(aHippodrome.getContourPoint(IHippodrome.END_UPPER_POINT), mwt, vwt);
35  	out_of_bounds = false;
36  }
37  // append an arc using the endpoint as center and taking into account the backward

```

```

37     // azimuth of the hippodrome
38     pen.appendArc(aHippodrome.getEndPoint(), aHippodrome.getWidth(),
39                     aHippodrome.getWidth(), 270 - aHippodrome.getEndStartAzimuth(),
40                     -aHippodrome.getEndStartAzimuth(), -180.0, mwt, vwt, aAWTPathSFCT);
41 } catch (TLcdOutOfBoundsException e) {
42     out_of_bounds = true;
43 }
44
45 try {
46     if (out_of_bounds) {
47         pen.moveTo(aHippodrome.getContourPoint(IHippodrome.END_LOWER_POINT), mwt, vwt);
48         out_of_bounds = false;
49     }
50     // append a line to the startLower-point
51     pen.appendLineTo(aHippodrome.getContourPoint(IHippodrome.START_LOWER_POINT),
52                      mwt, vwt, aAWTPathSFCT);
53 } catch (TLcdOutOfBoundsException e) {
54     out_of_bounds = true;
55 }
56
57 try {
58     if (out_of_bounds) {
59         pen.moveTo(aHippodrome.getContourPoint(IHippodrome.START_LOWER_POINT), mwt, vwt);
60     }
61
62     // append an arc using the startPoint as center and taking into account the forward
63     // azimuth of the hippodrome.
64     pen.appendArc(aHippodrome.getStartPoint(), aHippodrome.getWidth(),
65                   aHippodrome.getWidth(), 90.0 - aHippodrome.getStartEndAzimuth(),
66                   -aHippodrome.getStartEndAzimuth(), -180.0, mwt, vwt, aAWTPathSFCT);
67 } catch (TLcdOutOfBoundsException e) {
68     // we don't draw the last arc.
69 }
70 }

```

Program 174 - Constructing the `ILcdAWTPath` for an `IHippodrome`
 (from samples/gxy/hippodromePainter/GXYHippodromePainter)

35.6.3 Improving performance: `ILcdGeneralPath` and `ILcdCache`

Section 2.5 shows that transforming model coordinates to view coordinates requires two transformations:

1. an `ILcdModelXYWorldTransformation` to convert model coordinates to world coordinates
2. an `ILcdGXYViewXYWorldTransformation` to convert world coordinates to view coordinates

While the first step can be quite a complex calculation, taking into account an ellipsoid and a projection, the second step consists of a scaling, a rotation and a translation. All these operations can be executed in a small number of additions and multiplications. Since the elements that make up the complexity of the first transformation (the model reference of the object and the world reference of the `ILcdGXYView`) do not change often, it is advantagous to cache the world coordinates, thus eliminating the costly first step. To this end, two interfaces are available: `ILcdGeneralPath` to collect the world coordinates and `ILcdCache` to cache an `ILcdGeneralPath`.

`ILcdGeneralPath` is the world equivalent of `ILcdAWTPath` in the sense that it stores world coordinates. An `ILcdGeneralPath` is constructed using an `ILcdGXYPen`, analogous to constructing an `ILcdAWTPath`. An `ILcdAWTPath` can be derived from an `ILcdGeneralPath` using the method `appendGeneralPath` of `ILcdGXYPen`.

`ILcdCache` enables storing information key-value pairs for caching purposes. The combination of the key and the information stored should allow to determine if the cached value is valid.

```

1  private void retrieveAWTPathSFCT(IHippodrome aHippodrome, ILcdGXYContext aGXYContext,
2                                  ILcdAWTPath aAWTPathSFCT) {
3
4      // Do we have a cache and may it be used (defined by fPaintCache)
5      if (fPaintCache && aHippodrome instanceof ILcdCache) {
6
7          // Get the general path through the cache.
8          ILcdGeneralPath general_path = retrieveGeneralPathThroughCache(aGXYContext, aHippodrome)
9              ;
10
11         // Convert it into an AWT path.
12         fTempAWTPath.reset();
13         fTempAWTPath.ensureCapacity(general_path.subPathLength(0));
14
15         // Appends the given path general_path (in world coordinates) to the given path
16         // aAWTPathSFCT
17         // (in AWT view coordinates), applying the given ILcdGXYViewXYWorldTransformation
18         aGXYContext.getGXYPen().appendGeneralPath(general_path,
19                                         aGXYContext.getGXYViewXYWorldTransformation(),
20                                         aAWTPathSFCT);
21
22     } else {
23         // We don't have a cache.
24         // Compute the AWT path directly.
25         calculateContourAWTPathSFCT(aHippodrome, aGXYContext, aAWTPathSFCT);
26     }
27 }
```

Program 175 - Retrieving an ILcdAWTPath using an ILcdCache
 (from samples/gxy/hippodromePainter/GXYHippodromePainter)

```

1  private ILcdGeneralPath retrieveGeneralPathThroughCache(ILcdGXYContext aGXYContext,
2                                                        IHippodrome aHippodrome) {
3
4      // Get the cache.
5      ILcdCache cacher = (ILcdCache) aHippodrome;
6      HippodromeCache cache = (HippodromeCache) cacher.getCachedObject(this);
7
8      // Make sure we have a cached path.
9      TLcdGeneralPath general_path = cache == null ? new TLcdGeneralPath() : cache.fGeneralPath;
10
11     ILcdXYWorldReference current_world_reference = aGXYContext.getGXYView().
12         getXYWorldReference();
13
14     // Is the path valid?
15     if (cache == null || !cache.isValid(current_world_reference)) {
16
17         // Cached path not available or valid ==> update the path and cache it.
18         calculateContourGeneralPath(aHippodrome, aGXYContext, general_path);
19
20         // Trim it to take up as little space as possible.
21         general_path.trimToSize();
22
23         // Put it in the cache.
24         cache = new HippodromeCache(current_world_reference, general_path);
25         cacher.insertIntoCache(this, cache);
26     }
27
28     return general_path;
29 }
```

Program 176 - caching!an ILcdGeneralPath
 (from samples/gxy/hippodromePainter/GXYHippodromePainter)

Program 175 and Program 176 show how an ILcdGeneralPath is often retrieved from the ILcdCache. Program 175 checks if a valid ILcdGeneralPath is available from the ILcdCache and transforms it into an ILcdAWTPath. Program 176 details how the ILcdCache

is kept up to date. The `ILcdCache` contains the world reference of the view and the `ILcdGeneralPath`. This is sufficient to check if the `ILcdGeneralPath` is valid, since the model reference of the `IHippodrome` will not change. Note that the layer is used as a key. In another layer the `IHippodrome` may be rendered using another `ILcdGXYPainter` for which the `ILcdGeneralPath` may be different (see for example Section 35.1.3).

35.6.4 Rendering the `ILcdAWTPath`

Since `ILcdAWTPath` supports rendering itself as a polyline or a polygon on a `Graphics`, it is straightforward to enhance the hippodrome painter to fill the hippodrome. Program 177 demonstrates that once the `ILcdAWTPath` is constructed, rendering it in the view is quite simple. The method `drawAWTPath` provides two extra functionalities:

- A parameter indicates if the axis or the outline of the `IHippodrome` should be rendered.
- Depending on whether the `IHippodrome` should be painted filled or not, a line style and a fill style are applied to the `Graphics`.

```

1  private void drawAWTPath(ILcdAWTPath aAWTPath,
2                           Graphics aGraphics,
3                           int aRenderingMode,
4                           ILcdGXYContext aGXYContext,
5                           boolean aDrawAxisPath) {
6
7     if (aDrawAxisPath) {
8         setupGraphicsForLine(aGraphics, aRenderingMode, aGXYContext);
9         aAWTPath.drawPolyline(aGraphics);
10    } else {
11        int paint_mode = retrievePaintMode(aRenderingMode);
12        switch (paint_mode) {
13            case FILLED: {
14                setupGraphicsForFill(aGraphics, aRenderingMode, aGXYContext);
15                aAWTPath.fillPolygon(aGraphics);
16                break;
17            }
18            case OUTLINED_FILLED: {
19                setupGraphicsForFill(aGraphics, aRenderingMode, aGXYContext);
20                aAWTPath.fillPolygon(aGraphics);
21                setupGraphicsForLine(aGraphics, aRenderingMode, aGXYContext);
22                aAWTPath.drawPolyline(aGraphics);
23                break;
24            }
25            case OUTLINED:
26            default: {
27                setupGraphicsForLine(aGraphics, aRenderingMode, aGXYContext);
28                aAWTPath.drawPolyline(aGraphics);
29            }
30        }
31    }
}

```

Program 177 - Rendering an `ILcdAWTPath` on a `Graphics`
(from `samples/gxy/hippodromePainter/GXYHippodromePainter`)

35.6.5 Making sure the `IHippodrome` is visible in the view

As Section 35.2.2 explained, `boundsSFCT` should be implemented so that the `ILcdGXYView` can determine if the object should be rendered or not. To do so, it should return the view bounds of the object. `ILcdAWTPath` provides a method `calculateAWTBoundsSFCT` which does exactly this. Implementation of `boundsSFCT` is now straightforward: construct the `ILcdAWTPath`, be it from a cached `ILcdGeneralPath` or not, and call `calculateAWTBoundsSFCT`.

```

1  @Override
2  public void boundsSFCT(Graphics aGraphics,
3      int aPainterMode,
4      ILcdGXYContext aGXYContext,
5      ILcd2DEditableBounds aBoundsSFCT) throws TLcdNoBoundsException {
6
7      // ...
8      calculateContourAWTPathSFCT(fHippodrome, aGXYContext, fTempAWTPath);
9      // use the calculated AWT-path to set the AWT-bounds
10     fTempAWTPath.calculateAWTBoundsSFCT(fTempAWTBounds);
11
12     aBoundsSFCT.move2D(fTempAWTBounds.x, fTempAWTBounds.y);
13     aBoundssFCT.setWidth(fTempAWTBounds.width);
14     aBoundssFCT.setHeight(fTempAWTBounds.height);
15 }
```

Program 178 - Calculating view bounds, based on an `ILcdAWTPath`
 (from samples/gxy/hippodromePainter/GXYHippodromePainter)

35.7 Creating an IHippodrome

35.7.1 Creating a shape with `TLcdGXYNewController2`

Recalling from Section 35.1.4, the `IHippodrome` should be created in three steps. Figure 124 shows the sequence of calls made by the `TLcdGXYNewController2` when a `IHippodrome` is created with three mouse clicks. The new `IHippodrome` is constructed in the `ALcdGXYNewControllerModel2` set on the `TLcdGXYNewController2`. Once the object is constructed, its `ILcdGXYPainter` and `ILcdGXYEditor` are retrieved from the `ILcdGXYLayer` to which the `IHippodrome` will be added. It is important to note that neither the `ILcdGXYPainter`, nor the `ILcdGXYEditor` are responsible to construct the `IHippodrome`. The `ILcdGXYPainter` and `ILcdGXYEditor` only act on an `IHippodrome` constructed by the `ALcdGXYNewControllerModel2`.

All calls to the `ILcdGXYEditor` are the result of mouse clicks, whereas the calls to the `ILcdGXYPainter` are the result of the mouse moving over the `ILcdGXYView`. The `ILcdGXYEditor` returns true on the `edit` method to indicate that the object was indeed modified. This is the signal for the `TLcdGXYNewController2` to proceed to the next step. When creation is terminated, the `ALcdGXYNewControllerModel2` selects the object in the `ILcdGXYLayer` where it was created, thus triggering a paint which includes the `SELECTED` mode.

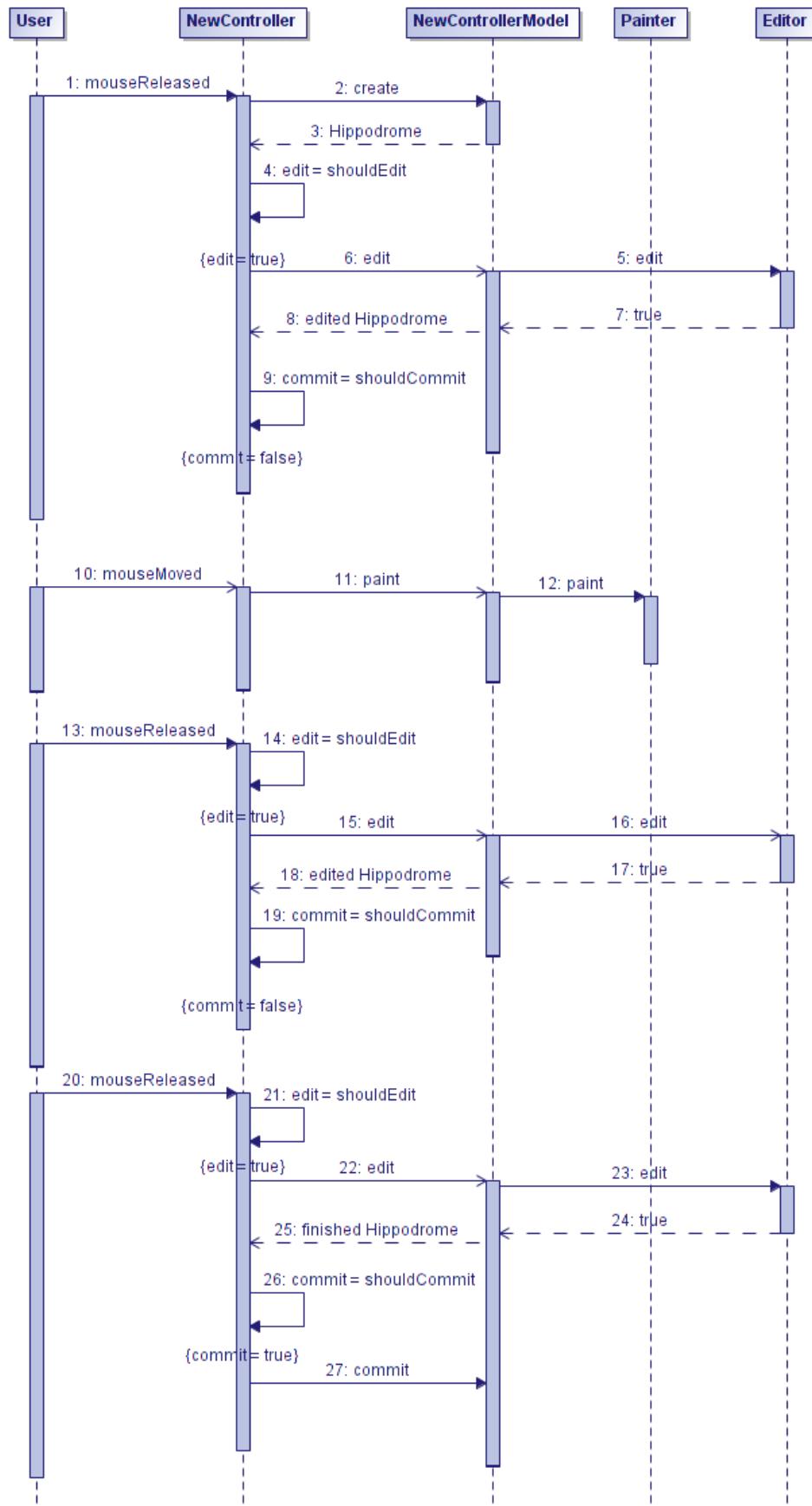


Figure 124 - Creating an object with three clicks

35.7.2 Combining `ILcdGXYPainter` and `ILcdGXYEditor`

Implementing the creation of an `IHippodrome` requires implementing both the `ILcdGXYPainter` and `ILcdGXYEditor` interface. While the `ILcdGXYEditor` is responsible for the object creation, the `ILcdGXYPainter` presents a preview of what the `ILcdGXYEditor` would do when the input is finished. Combining the implementations of both interfaces in one class allows using the following pattern to facilitate the preview:

- create a class that implements both `ILcdGXYPainter` and `ILcdGXYEditor`
- implement a private method that contains the logic of creating and editing an `IHippodrome` object
- create a copy of the object set to the painter/editor
- whenever a preview is needed, edit the copy of the object using the private edit method
- calculate the `ILcdAWTPath` of the copy
- render the preview using the `ILcdAWTPath` of the copy
- implement the `edit` method by delegating to the private edit method

Since the editing logic is contained in one method, the preview presented by the `ILcdGXYPainter` is guaranteed to be identical to the result of the `edit` operation.

The structure of the `paint` method would be as in [Program 179](#), resulting in an `ILcdAWTPath` for the partially created shape. Once the `ILcdAWTPath` is constructed, it can be rendered on the view as explained in [Section 35.6.4](#). Note that only the axis of the `IHippodrome` is included in the `ILcdAWTPath` if the mouse has not been clicked twice. After the second click the contour of the `IHippodrome` is rendered.

The structure of the `edit` method is illustrated in [Program 180](#).

```

1  @Override
2  public void paint(Graphics aGraphics, int aPaintMode, ILcdGXYContext aGXYContext) {
3      // ...
4      if (
5          ((aPaintMode & ILcdGXYPainter.CREATING) != 0))
6      {
7          copyHippodromeToTemp();
8
9          // edit this copy to apply the edited changes
10         // make sure to pass in ILcdGXYEditor-modes based on the received (as argument) paint-
11         // modes
12         privateEdit(
13             aGraphics,
14             fTempHippodrome,
15             aGXYContext,
16             (aPaintMode & ILcdGXYPainter.TRANSLATING) != 0 ? ILcdGXYEditor.TRANSLATED :
17             (aPaintMode & ILcdGXYPainter.CREATING) != 0 ? ILcdGXYEditor.CREATING :
18             ILcdGXYEditor.RESHAPED
19         );
20
21         IHippodrome hippodrome = fTempHippodrome;
22         if (
23             (((aPaintMode & ILcdGXYPainter.CREATING) != 0) &&
24             (!Double.isNaN(hippodrome.getWidth())))
25         )
26         // Get the AWT path from the edited shape.
27         {
28             calculateContourAWTPathSFCT(hippodrome, aGXYContext, fTempAWTPath);
29         } else {
30             // only the axis needs to be drawn, so only create the bounds of the axis
31             calculateAxisAWTPathSFCT(hippodrome, aGXYContext, fTempAWTPath);
32             paint_axis = true;
33         }
34         drawAWTPath(fTempAWTPath, aGraphics, aPaintMode, aGXYContext, paint_axis);
35     }
36 }
```

Program 179 - Implementation of paint using the private edit method
 (from samples/gxy/hippodromePainter/GXYHippodromePainter)

```

1  @Override
2  public boolean edit(Graphics aGraphics,
3                      int aEditMode,
4                      ILcdGXYContext aGXYContext) {
5
6      return privateEdit(aGraphics, fHippodrome, aGXYContext, aEditMode);
7
8  }
```

Program 180 - Implementation of edit using the private edit method
 (from samples/gxy/hippodromePainter/GXYHippodromePainter)

35.7.3 Interpreting the input position

Program 181 shows how the position of the input point is used to determine the location of the first turning point. The input point position and the transformations needed to transform it to model coordinates are contained in the ILcdGXYContext passed. The interpretation of setting the width of the IHippodrome, is illustrated in Program 186 in Section 35.8.4.

```

1  private boolean privateEdit(Graphics aGraphics,
2      IHippodrome aHippodrome,
3      ILcdGXYContext aGXYContext,
4      int aEditMode) {
5
6      boolean shape_modified = false;
7      // ...
8      // retrieve the necessary transformations from the context.
9      ILcdModelXYWorldTransformation mwt = aGXYContext.getModelXYWorldTransformation();
10     ILcdGXYViewXYWorldTransformation vwt = aGXYContext.getGXYViewXYWorldTransformation();
11     if (((aEditMode & ILcdGXYEditor.START_CREATION) != 0)
12         || ((aEditMode & ILcdGXYEditor.CREATING) != 0)
13         && (!Double.isNaN(aHippodrome.getStartPoint().getX()))
14         && (Double.isNaN(aHippodrome.getEndPoint().getX())))
15     ) {
16
17         // move the temp. point to the input point coordinates.
18         fTempAWTPoint.move(aGXYContext.getX(), aGXYContext.getY());
19
20         // transform input point-position to model-coordinates
21         vwt.viewAWTPoint2worldSFCT(fTempAWTPoint, fTempXYWorldPoint);
22         mwt.worldPoint2modelSFCT(fTempXYWorldPoint, model_point);
23
24         if (((aEditMode & ILcdGXYEditor.START_CREATION) != 0)) {
25             // we just started creating aHippodrome,
26             // so place the start point of the aHippodrome on the input location
27             aHippodrome.moveReferencePoint(model_point, IHippodrome.START_POINT);
28         } else {
29             // we're busy creating the hippodrome and the start point has already been placed,
30             // so place the end point of the aHippodrome on the input location
31             aHippodrome.moveReferencePoint(model_point, IHippodrome.END_POINT);
32         }
33
34         aShape_modified = true;
35     }

```

Program 181 - Interpreting input movement when creating a IHippodrome
(from samples/gxy/hippodromePainter/GXYHippodromePainter)

35.7.4 Adapting the view bounds

The view bounds need to be adapted while creating the shape. When the IHippodrome is not fully defined the view bounds should contain at least the axis connecting both points. When the location of the two turning points is defined, the mouse position should be interpreted as an indication of the width. [Program 182](#) illustrates how this is taken into account when computing the bounds while creating a IHippodrome:

1. an ILcdAWTPath is created based on the axis or the contour, depending on the number of input points
2. the bounds of the ILcdAWTPath are computed
3. the computed bounds are copied to the bounds passed to the method

```

1  @Override
2  public void boundsSFCT(Graphics aGraphics,
3      int aPainterMode,
4      ILcdGXYContext aGXYContext,
5      ILcd2DEditableBounds aBoundsSFCT) throws TLcdNoBoundsException {
6
7      // ...
8      if ((aPainterMode & ILcdGXYPainter.CREATING) != 0) {
9
10         // when creating a hippodrome, the bounds are different depending on how far
11         // you are in the creation-process.
12         if (Double.isNaN(fHippodrome.getEndPoint().getX())) {
13             // start point is set, now the endpoint needs to be set; in the meantime a line is
14             // drawn between the start point and the current input point-location, therefore
15             // calculating
16             // the bounds of this drawn line.
17             calculateAxisAWTPathSFCT(fHippodrome, aGXYContext, fTempAWTPath);
18         } else if (Double.isNaN(fHippodrome.getWidth())) {
19             // start- and endpoint are set, now the width of the new hippodrome is being set;
20             // make a clone of the set hippodrome, as the set hippodrome isn't yet the complete
21             // (resulting) hippodrome
22             copyHippodromeToTemp();
23
24             boolean hippodrome_width_changed = privateEdit(
25                 aGraphics,
26                 fTempHippodrome,
27                 aGXYContext,
28                 ILcdGXYEditor.CREATING);
29
30             if (hippodrome_width_changed) {
31                 calculateContourAWTPathSFCT(fTempHippodrome, aGXYContext, fTempAWTPath);
32             } else {
33                 // the hippodrome could not be adapted, probably because
34                 // the current location of the input point could not be transformed to model-
35                 // coordinates.
36                 // We just calculate the bounds of the axis
37                 calculateAxisAWTPathSFCT(fTempHippodrome, aGXYContext, fTempAWTPath);
38             }
39         }
40         // use the calculated AWT-path to set the AWT-bounds
41         fTempAWTPath.calculateAWTBoundsSFCT(fTempAWTBounds);
42
43         aBoundsSFCT.move2D(fTempAWTBounds.x, fTempAWTBounds.y);
44         aBoundsSFCT.setWidth(fTempAWTBounds.width);
45         aBoundsSFCT.setHeight(fTempAWTBounds.height);
46     }
47 }
```

Program 182 - View bounds when creating a IHippodrome
 (from samples/gxy/hippodromePainter/GXYHippodromePainter)

35.8 Editing an IHippodrome

35.8.1 Deciding on the editing modes

Recalling from Section 35.1.5, the specifications for editing a hippodrome are the following: it should be possible to

1. move the hippodrome as a whole
2. move one of the two turning points
3. change the width of the hippodrome

The interface `ILcdGXYEditor` provides two different editing modes: `TRANSLATED` and `RESHAPED`, while three are needed according to the requirements. To fit the requirements, the

`IHippodrome ILcdGXYEditor` is designed so that the TRANSLATED mode covers the first two, while the last one maps to the RESHAPED mode. To distinguish between the first two modes, the `ILcdGXYEditor` checks which part of the `IHippodrome` is touched by the input points: when one of the turning points is touched, that point will be moved. Otherwise, the shape as a whole will be moved. This implies implementing a method more specific than `isTouched`, since the method should be able to indicate what part of the shape is being touched.

Moving the shape as a whole can only be implemented in function of the defining elements of the shape. Thus, moving the shape as a whole over a distance means moving the turning points over that distance. This does not imply that the point at which the input point touched the `IHippodrome` moves over the same distance. As such, it is possible that the `IHippodrome` moves relatively to the mouse pointer.

35.8.2 Editing a shape with `TLcdGXYEditController2`

Figure 125 shows the typical sequence of calls to the `ILcdGXYPainter` and the `ILcdGXYEditor` when an `IHippodrome` is being translateds using the mouse. As long as the mouse is being dragged, the `IHippodrome` is painted in TRANSLATING mode by the `ILcdGXYPainter`. As soon as the mouse is released, the `IHippodrome` is adapted by the `ILcdGXYEditor`. The `edit` method returns `true` to confirm that the `IHippodrome` has been adapted. Every call to the `ILcdGXYPainter` and to the `ILcdGXYEditor` is preceded by a check on whether the `IHippodrome` is being touched by the mouse pointer. Note that the controller may still decide to call `paint` or `edit` operations, regardless of the outcome of the check. An example edit operation without touching the object would be when the user is trying to move the currently selected objects. Here the controller only requires one of the selection's objects to be touched.

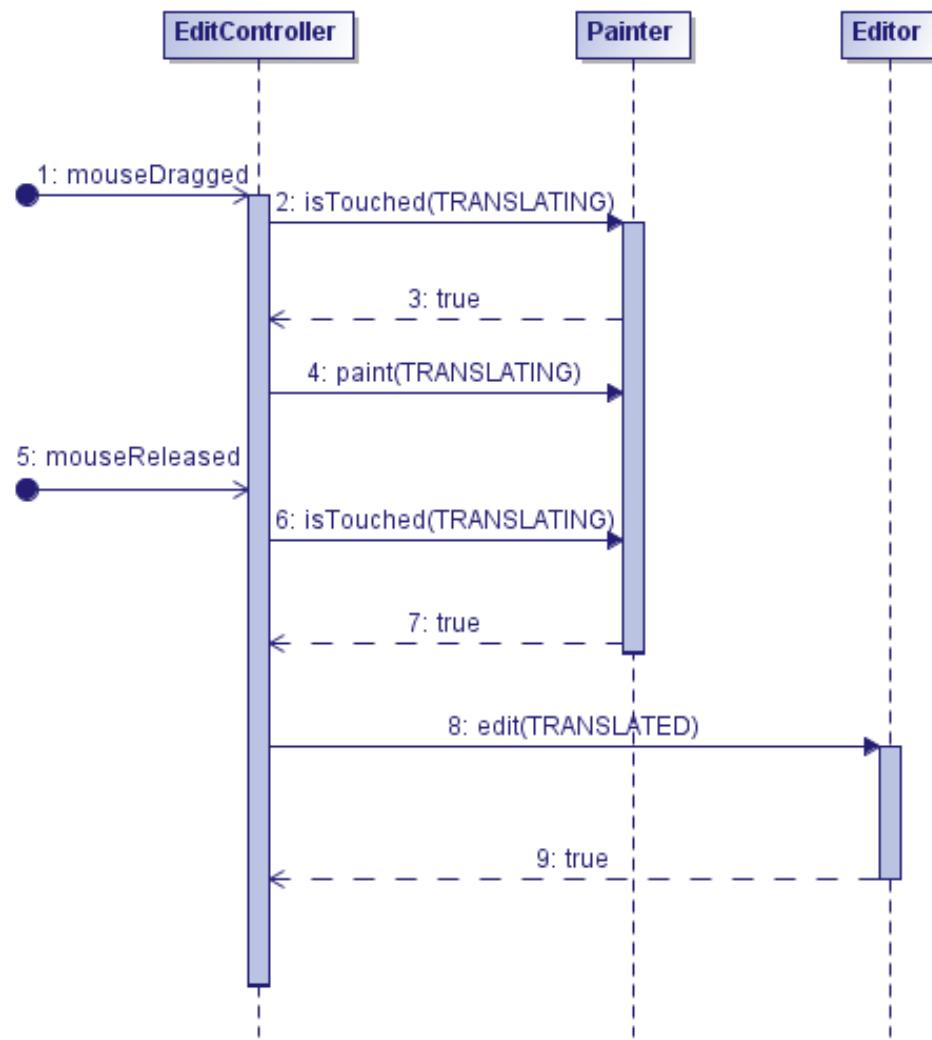


Figure 125 - Editing an object with `TLcdGXYEditController2`

35.8.3 Checking if an object is touched

The implementation of `isTouched` is crucial for editing since every interaction with the `ILcdGXYPainter` and the `ILcdGXYEditor` starts with one or more calls to this method.

Checking if a shape is touched while it is being edited is not possible, since the `ILcdGXYPainter` renders a preview of what the result of the editing operation would be, while the object itself is not being edited. The implementation of `isTouched` therefore checks if the object was touched when the move of the input point(s) that initiated the editing started. This requires the original location of the input points. The `ILcdGXYContext` passed to `isTouched` contains the current location of an input point (`getX()` and `getY()`) and its displacement (`getDeltaX()` and `getDeltaY()`), from which the starting location can be derived. Program 185 demonstrates how to check if one of the turning points of the hippodrome is touched by the input point while editing.

Checking the input point location at the start of the movement also imposes a requirement on the preview presented by the `ILcdGXYPainter`. To obtain an intuitive user interaction the `ILcdGXYPainter` `paint` method should be implemented such that the input pointer continues touching the shape during the move of the input point(s). If not, `isTouched` could return `true`, without visual confirmation of that status.

Program 183 illustrates how the `isTouched` method is implemented in the `IHippodrome` `ILcdGXYPainter`. It consists of calling a private method that returns which part of the `IHippodrome`, if any, was touched by the input point. That method, shown in Program 184, checks for all parts of the representation of the `IHippodrome` if they are touched by the input point. Note that this may depend on the way the `IHippodrome` is rendered. If the `IHippodrome` is rendered filled, then the interior of the `IHippodrome` is also taken into account.

Finally, Program 185 demonstrates how to check if one of the turning points of the `IHippodrome` is touched by the input point. Note that the coordinates of the position which is checked are `aGXYContext.getX() - aGXYContext.getDeltaX()` and `aGXYContext.getY() - aGXYContext.getDeltaY()`, which amounts to the location where the movement drag started.

```
1  @Override
2  public boolean isTouched(Graphics aGraphics, int aPainterMode, ILcdGXYContext aGXYContext) {
3
4      // retrieve which part of the hippodrome is touched
5      int touchedStatus = retrieveTouchedStatus(aGXYContext, aPainterMode, fHippodrome);
6
7      // return true if a part is touched
8      return (touchedStatus != NOT_TOUCHED);
9  }
```

Program 183 - Checking if the `IHippodrome` is touched
(from samples/gxy/hippodromePainter/GXYHippodromePainter)

```

1  private int retrieveTouchedStatus(ILcdGXYContext aGXYContext,
2                               int aRenderingMode,
3                               IHippodrome aHippodrome) {
4
5     boolean checkHandles = (aRenderingMode & ILcdGXYPainter.HANDLES) != 0;
6     boolean checkBody = (aRenderingMode & ILcdGXYPainter.BODY) != 0;
7
8     if (checkHandles) {
9         if (isPointTouched(aGXYContext, aHippodrome, START_POINT)) {
10            return START_POINT;
11        } else if (isPointTouched(aGXYContext, aHippodrome, END_POINT)) {
12            return END_POINT;
13        }
14    }
15    if (checkBody) {
16        if (isArcTouched(aGXYContext, aHippodrome, START_ARC)) {
17            return START_ARC;
18        } else if (isArcTouched(aGXYContext, aHippodrome, END_ARC)) {
19            return END_ARC;
20        } else if (isLineTouched(aGXYContext, aHippodrome, UPPER_LINE)) {
21            return UPPER_LINE;
22        } else if (isLineTouched(aGXYContext, aHippodrome, LOWER_LINE)) {
23            return LOWER_LINE;
24        } else if ((getMode() == FILLED || getMode() == OUTLINED_FILLED) ||
25                   ((aRenderingMode & ILcdGXYPainter.SELECTED) != 0 &&
26                   (getSelectionMode() == FILLED || getSelectionMode() == OUTLINED_FILLED))) {
27            // retrieve (or calculate) an AWTPath for aHippodrome to be able to determine if the
28            // original
29            // input point-location lies within the aHippodrome
30            retrieveAWTPathSFCT(aHippodrome, aGXYContext, fTempAWTPath);
31
32            // take into account the original input point-location instead of the current
33            // ==> aGXYContext.getX() - aGXYContext.getDeltaX() as argument for polygonContains()
34            if (fTempAWTPath.polygonContains(aGXYContext.getX() - aGXYContext.getDeltaX(),
35                                              aGXYContext.getY() - aGXYContext.getDeltaY())) {
36                return INNER;
37            }
38        }
39    }
40    return NOT_TOUCHED;
}

```

Program 184 - Checking what part of the IHippodrome is touched by the input point
 (from samples/gxy/hippodromePainter/GXYHippodromePainter)

```

1  private boolean isPointTouched(ILcdGXYContext aGXYContext,
2                               IHippodrome aHippodrome,
3                               int aHippodromePoint) {
4     ILcdGXYPen pen = aGXYContext.getGXYPen();
5     ILcdModelXYWorldTransformation mwt = aGXYContext.getModelXYWorldTransformation();
6     ILcdGXYViewXYWorldTransformation vwt = aGXYContext.getGXYViewXYWorldTransformation();
7
8     ILcdPoint point_to_check;
9
10    if (aHippodromePoint == START_POINT) {
11        point_to_check = aHippodrome.getStartPoint();
12    } else if (aHippodromePoint == END_POINT) {
13        point_to_check = aHippodrome.getEndPoint();
14    } else {
15        return false;
16    }
17
18    return pen.isTouched(
19        point_to_check,
20        aGXYContext.getX() - aGXYContext.getDeltaX(),
21        aGXYContext.getY() - aGXYContext.getDeltaY(),
22        getSensitivity(aGXYContext, pen.getHotPointSize()),
23        mwt,
24        vwt);
25}
26
27 private int getSensitivity(ILcdGXYContext aGXYContext, int aDefaultValue) {
28     int sensitivity = aGXYContext.getSensitivity();
29     if (sensitivity >= 0) {
30         return sensitivity;
31     }
32     return aDefaultValue;
33 }

```

Program 185 - Checking if a turning point is touched by the input point
 (from samples/gxy/hippodromePainter/GXYHippodromePainter)

35.8.4 Implementing paint and edit for editing modes

The paint and edit methods for the editing modes are exactly the same as their equivalents for the creation modes. The code in [Program 179](#) and [Program 180](#) can be reused. [Program 186](#) shows how privateEdit is extended to enable changing the width of the IHippodrome, based on the modes passed. Note that this code is also called at the end of the creation process. The implementation for changing the width of the IHippodrome is shown in [Program 187](#):

1. the current position of the input point is retrieved
2. the distance to the axis is calculated
3. the result of the calculation is set as the new width of the IHippodrome

Since calculation of the distance to the axis is different on a Cartesian plane than on an ellipsoid, the model reference of the IHippodrome has to be retrieved from the ILcdGXYContext.

```

1  private boolean privateEdit(Graphics aGraphics,
2      IHippodrome aHippodrome,
3      ILcdGXYContext aGXYContext,
4      int aEditMode) {
5
6      boolean shape_modified = false;
7      // ...
8      return shape_modified;
9  }
10
11     if ((
12         ((aEditMode & ILcdGXYEditor.RESHAPED) != 0)
13         && (isContourTouched(aGXYContext, aHippodrome)))
14     || (((aEditMode & ILcdGXYEditor.CREATING) != 0)
15         && (!Double.isNaN(aHippodrome.getEndPoint().getX()))
16         && (Double.isNaN(aHippodrome.getWidth())))
17     ) {
18         // when these modes are received, the user is busy changing or
19         // has just changed the width of aHippodrome
20         aShape_modified = changeWidth(aGXYContext, aHippodrome);
21     }

```

Program 186 - Enabling changing the width in private edit
 (from samples/gxy/hippodromePainter/GXYHippodromePainter)

```

1  private boolean changeWidth(ILcdGXYContext aGXYContext,
2      IHippodrome aHippodrome) throws TLcdOutOfBoundsException {
3      boolean shape_modified;
4      ILcd3DEditablePoint model_point = aHippodrome.getStartPoint().cloneAs3DEditablePoint();
5
6      // move a temp. viewpoint to the current location of the input point
7      fTempAWTPoint.move(aGXYContext.getX(), aGXYContext.getY());
8
9      ILcdModelXYWorldTransformation mwt = aGXYContext.getModelXYWorldTransformation();
10     ILcdGXYViewXYWorldTransformation vwt = aGXYContext.getGXYViewXYWorldTransformation();
11
12     // transform input point-position to model-coordinates
13     vwt.viewAWTPoint2worldSFCT(fTempAWTPoint, fTempXYWorldPoint);
14     mwt.worldPoint2modelSFCT(fTempXYWorldPoint, model_point);
15
16     // calculate distance (in the model) from the input point-location to the axis of
17     // aHippodrome
18     double distance_to_axis;
19     ILcdModelReference model_reference = aGXYContext.getGXYLayer().getModel().
20         getModelReference();
21     if (model_reference instanceof ILcdGeodeticReference) {
22         distance_to_axis = TLcdEllipsoidUtil.closestPointOnGeodesic(aHippodrome.getStartPoint(),
23             aHippodrome.getEndPoint(), model_point, ((ILcdGeodeticReference) model_reference).
24             getGeodeticDatum().getEllipsoid(), 1e-10,
25                                         1.0, new TLcdLonLatPoint());
26     } else {
27         // we assume we are computing in a plane
28         distance_to_axis = TLcdCartesian.closestPointOnLineSegment(aHippodrome.getStartPoint(),
29             aHippodrome.getEndPoint(), model_point, new TLcdXYPoint());
30     }
31
32     // set this distance as new width of aHippodrome
33     aHippodrome.setWidth(distance_to_axis);
34
35     shape_modified = true;
36     return shape_modified;
37 }

```

Program 187 - Changing the width of an IHippodrome
 (from samples/gxy/hippodromePainter/GXYHippodromePainter)

35.8.5 Taking care of view bounds

As explained in [Section 35.2.2](#) the `ILcdGXYPainter` should take into account changes to the object in `boundsSFCT` to ensure that it is rendered in the `ILcdGXYView`. [Program 188](#) illustrates how `privateEdit` facilitates implementing `boundsSFCT`.

```

1  @Override
2  public void boundsSFCT(Graphics aGraphics,
3      int aPainterMode,
4      ILcdGXYContext aGXYContext,
5      ILcd2DEditableBounds aBoundsSFCT) throws TLcdNoBoundsException {
6
7      // ...
8      if (((aPainterMode & ILcdGXYPainter.TRANSLATING) != 0) ||
9          ((aPainterMode & ILcdGXYPainter.RESHAPING) != 0)) {
10
11          // when changing the size or location of the original hippodrome, make a clone of it,
12          // so the original hippodrome is unchanged until the complete edit has changed
13          copyHippodromeToTemp();
14
15          // apply the editing changes, hereby transforming the painter-modes to editor-modes
16          privateEdit(
17              aGraphics,
18              fTempHippodrome,
19              aGXYContext,
20              (aPainterMode & ILcdGXYPainter.TRANSLATING) != 0 ? ILcdGXYEditor.TRANSLATED
21                                         : ILcdGXYEditor.RESHAPED);
22
23          // calculate the AWT-path
24          calculateContourAWTPathSFCT(fTempHippodrome, aGXYContext, fTempAWTPath);
25      }
26      // use the calculated AWT-path to set the AWT-bounds
27      fTempAWTPath.calculateAWTBoundsSFCT(fTempAWTBounds);
28
29      aBoundsSFCT.move2D(fTempAWTBounds.x, fTempAWTBounds.y);
30      aBoundsSFCT.setWidth(fTempAWTBounds.width);
31      aBoundsSFCT.setHeight(fTempAWTBounds.height);
32  }

```

Program 188 - View bounds of an `IHippodrome` while editing
(from samples/gxy/hippodromePainter/GXYHippodromePainter)

35.9 Snapping implemented

35.9.1 Specifications for snapping

The `ILcdGXYPainter` and `ILcdGXYEditor` for `IHippodrome` should support snapping in two ways:

- snap the turning points of the `IHippodrome` onto any `ILcdPoint`, and
- any other shape should be able to snap to either of the turning points.

35.9.2 snapTarget, snapping to an `IHippodrome`

Since snapping to either turning point should be supported, `snapTarget` should return the turning point that is being touched by an input point. If neither of the points is touched, it should return `null`. This indicates that in the given `ILcdGXYContext` snapping to this `IHippodrome` is not possible. [Program 189](#) illustrates how reuse of methods that check which part of the `IHippodrome` is touched simplifies retrieving the snap target.

```
1  @Override
2  public Object snapTarget(Graphics aGraphics,
3      ILcdGXYContext aGXYContext) {
4
5      Object snap_target = null;
6
7      // determine if the start point is touched,
8      if (isPointTouched(aGXYContext, fHippodrome, START_POINT)) {
9          snap_target = fHippodrome.getStartPoint();
10     }
11     // or the endpoint
12     else if (isPointTouched(aGXYContext, fHippodrome, END_POINT)) {
13         snap_target = fHippodrome.getEndPoint();
14     }
15
16     return snap_target;
17 }
```

Program 189 - Presenting the turning points as snap target
(from samples/gxy/hippodromePainter/GXYHippodromePainter)

35.9.3 acceptSnapTarget, snapping of an IHippodrome

The `ILcdGXYEditor` may decide to reject snap targets. In this case study, see Program 190, any object that is not an `ILcdPoint` is rejected. `acceptSnapTarget` also checks if snapping to the snap target is *possible* and *required*. If the snap target is an `ILcdPoint` defined in another reference and it is on a location which cannot be expressed in coordinates of the model reference in which the `IHippodrome` is defined, snapping is impossible.

If the outline of the `IHippodrome` was touched, and not one of the turning points, snapping is not required. This means that if a user moves the `IHippodrome` as a whole, by dragging its outline, neither of the turning points will snap to another point even if they are touching.

```

1  @Override
2  public boolean acceptSnapTarget(Graphics aGraphics,
3  								ILcdGXYContext aGXYContext) {
4
5  	Object snap_target = aGXYContext.getSnapTarget();
6  	ILcdGXYLayer hippodrome_layer = aGXYContext.getGXYLayer();
7  	ILcdGXYLayer snap_target_layer = aGXYContext.getSnapTargetLayer();
8
9  	// we only snap to points
10  if (!(snap_target instanceof ILcdPoint)) {
11  	return false;
12  }
13
14  // we do not snap to ourselves.
15  if (snap_target == fHippodrome.getStartPoint() || snap_target == fHippodrome.getEndPoint())
16  	{
17  	return false;
18  }
19
20  ILcdPoint snap_point = (ILcdPoint) snap_target;
21
22  // are the snap target and the hippodrome in the same layer?
23  boolean same_layer = (snap_target_layer == hippodrome_layer);
24
25  // or do they have the same model reference?
26  ILcdModel snap_target_model = snap_target_layer.getModel();
27  ILcdModel hippodrome_model = hippodrome_layer.getModel();
28  boolean same_model_ref = same_layer ||
29  	((hippodrome_model.getModelReference().equals(snap_target_model.
30  								 getModelReference())));
31
32  boolean transformation_supported = false;
33  if (!same_model_ref && (fModelReference2ModelReference != null)) {
34  	fModelReference2ModelReference.setSourceReference(snap_target_model.getModelReference())
35  	;
36  	fModelReference2ModelReference.setDestinationReference(hippodrome_model.
37  								 getModelReference());
38
39  try {
40  	// retrieve the location of the snap point in the hippodromes reference.
41  	fModelReference2ModelReference.sourcePoint2destinationSFCT(snap_point, fTempModelPoint
42  	);
43
44  	// no exception occurred, so compatible references
45  	transformation_supported = true;
46  } catch (TLcdOutOfBoundsException ex) {
47  	// different model reference and the transformation does not support transforming
48  	// from one to the other, so we reject the snap target.
49  	return false;
50  }
51
52  // was one of the handles of the hippodrome touched by the input point?
53  int touched_status = retrieveTouchedStatus(aGXYContext, ILcdGXPainter.HANDLES,
54  										 fHippodrome);
55
56  // are we creating the start or end point?
57  boolean creating = Double.isNaN(fHippodrome.getEndPoint().getX());
58  // or are we editing the start or end point of an existing hippodrome?
59  boolean editing = !Double.isNaN(fHippodrome.getWidth()) && touched_status != NOT_TOUCHED;
60
61  // only accept the target if the transformation allows transforming to it,
62  // and if we are dragging or creating one of the points,
63  return (same_layer || same_model_ref || transformation_supported) &&
64  	(creating || editing);
65
66 }

```

Program 190 - Accepting a point as snap target
 (from samples/gxy/hippodromePainter/GXYHippodromePainter)

35.9.4 paint, highlighting the snap target

The method `paint` provides a visual indication that a snap target is available using the `SNAPS` mode. Program 191 illustrates that highlighting the snap target with an icon suffices, since the snap target returned for an `IHippodrome` is always an `ILcdPoint`.

```

1  @Override
2  public void paint(Graphics aGraphics, int aPaintMode, ILcdGXYContext aGXYContext) {
3      // ...
4      if ((aPaintMode & ILcdGXYPainter.SNAPS) != 0) {
5          // when requested to draw a snap target of the hippodrome, we first need to know which
6          // point was returned as snap target, so we can highlight it with the snap icon.
7
8          ILcdPoint snap_target = (ILcdPoint) snapTarget(aGraphics, aGXYContext);
9
10         if (snap_target != null) {
11             try {
12                 pen.moveTo(snap_target, mwt, vwt);
13                 int x = pen.getX();
14                 int y = pen.getY();
15                 fSnapIcon.paintIcon(null,
16                                     aGraphics,
17                                     x - (fSnapIcon.getIconWidth() / 2),
18                                     y - (fSnapIcon.getIconHeight() / 2));
19             } catch (TLcdOutOfBoundsException e) {
20                 // we don't draw the snap target since it is not visible in this world reference.
21                 // this will actually never happen.
22             }
23         }
24     }
25 }
26 }
```

Program 191 - Highlighting a snap target
 (from samples/gxy/hippodromePainter/GXYHippodromePainter)

35.9.5 edit, snapping to a snap target

The method `edit` should be implemented so that if a snap target is present in the `ILcdGXYContext` and if it is accepted, the snap operation is executed: one of the turning points should be moved to the snap target. Program 192 shows how the private `edit` method, which contains the editing logic, is extended to support snapping of the `IHippodrome` to an `ILcdPoint`. Program 193 demonstrates how the snapping is implemented, taking into account that the snap target may be defined in another reference.

```

1  private boolean privateEdit(Graphics aGraphics,
2                               IHippodrome aHippodrome,
3                               ILcdGXYContext aGXYContext,
4                               int aEditMode) {
5
6      boolean shape_modified = false;
7      // ...
8      return shape_modified;
9  }
10
11     if ((aEditMode & ILcdGXYEditor.TRANSLATED) != 0 &&
12         (aGXYContext.getDeltaX() != 0 || aGXYContext.getDeltaY() != 0)) {
13         // We were/are translating the shape (depending on whether it was called from paint() or
14             edit()).
15         // apply different changes to aHippodrome based on input positions and transformations.
16
17         // determine which part has been changed
18         switch (retrieveTouchedStatus(aGXYContext, convertToRenderingMode(aEditMode),
19                                         aHippodrome)) {
20             case START_POINT:
21
22                 // the location of the start point is (being) changed,
23                 // check if it can be snapped to a snap-target
24                 if (!(aShape_modified = linkToSnapTarget(aHippodrome, aGraphics,
25                                                 aEditMode, aGXYContext))) {
26                     // ...
27                     break;
28             case END_POINT:
29
30                 // the location of the endpoint is (being) changed,
31                 // check if it can be snapped to a snap-target
32                 if (!(aShape_modified = linkToSnapTarget(aHippodrome, aGraphics,
33                                                 aEditMode, aGXYContext))) {
34                     // ...
35                     break;
36             }
37         }
38     }

```

Program 192 - Snapping in the private edit method
 (from samples/gxy/hippodromePainter/GXYHippodromePainter)

```

1  private boolean linkToSnapTarget(IHippodrome aHippodrome,
2                                 Graphics aGraphics,
3                                 int aEditMode,
4                                 ILcdGXYContext aGXYContext) {
5
6      boolean snapped_to_target = false;
7
8      // test if the snap target returned by aGXYContext can be accepted by aHippodrome; in
9      // other words,
10     if (acceptSnapTarget(aGraphics, aGXYContext)) {
11         // if it is possible, retrieve which reference-point is touched, so we know which
12         // point needs to be moved to that snap-target
13
14         ILcdPoint snap_point = (ILcdPoint) aGXYContext.getSnapTarget();
15         ILcdGXYLayer hippodrome_layer = aGXYContext.getGXYLayer();
16         ILcdGXYLayer snap_layer = aGXYContext.getSnapTargetLayer();
17
18         // which point is touched ? acceptSnapTarget already ensures that either
19         // the start or the end point is touched.
20         // Since checking whether an object is touched is a functionality of the painter,
21         // the edit mode has to be converted to a rendering mode.
22         int renderingMode = convertToRenderingMode(aEditMode);
23         int touched_status = retrieveTouchedStatus(aGXYContext, renderingMode, aHippodrome);
24
25         int point_to_move = IHippodrome.START_POINT;
26     }

```

```

27  if (touched_status == END_POINT) {
28      point_to_move = IHippodrome.END_POINT;
29  }
30
31  // if the snap target is in the same layer or has the same model-references, just move
32  // the point
33  // of the hippodrome
34  if ((snap_layer == hippodrome_layer) ||
35      (snap_layer.getModel().getModelReference().equals(
36          hippodrome_layer.getModel().getModelReference()))) {
37
38      aHippodrome.moveReferencePoint(snap_point, point_to_move);
39      snapped_to_target = true;
40  } else {
41      try {
42          // acceptSnapTarget is called, so the transformation is setup correctly.
43          // fTempModelPoint is moved to the location of the snap_point
44          fModelReference2ModelReference.setDestinationReference(
45              snap_layer.getModel().getModelReference());
46          fModelReference2ModelReference.setSourceReference(
47              hippodrome_layer.getModel().getModelReference());
48          fModelReference2ModelReference.destinationPoint2sourceSFCT(snap_point,
49              fTempModelPoint);
50
51          // move the touched reference-point to the snap-target
52          aHippodrome.moveReferencePoint(fTempModelPoint, point_to_move);
53
54          snapped_to_target = true;
55      } catch (TLcdOutOfBoundsException e) {
56          snapped_to_target = false;
57      }
58  }
59
60  return snapped_to_target;
}

```

Program 193 - Snapping to a snap target
 (from samples/gxy/hippodromePainter/GXYHippodromePainter)

CHAPTER 36

Labeling domain objects in a GXY view

This chapter explains the support that LuciadLightspeed offers for labeling the representations of your domain objects. A label can be anything: a single word, a multi-line text string, an icon, or even an interactive panel. Painting labels is similar to painting domain objects, with the following differences:

- Labels are usually painted on top of all regular object representations.
- The labeling API explicitly models multiple labels for an object's representation.
- Labels have more advanced positioning options. This allows decluttering of labels to avoid overlap.

The following sections describe a typical labeling scenario, how to paint labels, how to work with label locations, how to interact with labels, and some more advanced labeling topics.

36.1 A typical labeling scenario

The following requirements are exemplary for a typical labeling scenario:

- You want to place a single label next to your domain object's representation.
- The label is text-based, its text comes from a certain property of the object.
- To avoid cluttering, overlapping labels should be dropped.

To determine the number of labels, the label content, and the rendering of the label, you need to configure an `ILcdGXYLabelPainterProvider` on the layer so that it returns an `ILcdGXYLabelPainter2` for all domain objects. All the main implementations of `ILcdGXYLabelPainter2` that LuciadLightspeed provides also implement the `ILcdGXYLabelPainterProvider` interface, and return themselves as label painter for every object. This is convenient when all objects of a layer need the same label painter.

Section 36.2.1 details the LuciadLightspeed implementations of the label painters and label painter providers. In the sample scenario, a `TLcdGXYDataObjectLabelPainter` is used: a text-based label painter which you can configure to paint one of the properties of the domain objects. Sample Program 194 shows how to configure this label painter on the layer.

```

1 TLcdGXYDataObjectLabelPainter labelPainter = new TLcdGXYDataObjectLabelPainter();
2 labelPainter.setExpressions(WayPointDataTypes.NAME);
3 labelPainter.setForeground(Color.WHITE);
4 labelPainter.setHaloEnabled(true);
5 labelPainter.setHaloColor(Color.BLACK);
6
7 layer.setGXYLabelPainterProvider(labelPainter);
8 layer.setLabeled(true);

```

Program 194 - Configuring an `ILcdGXYLabelPainter2` for a `TLcdGXYLayer`
 (from samples/gxy/fundamentals/step3/WayPointLayerFactory)

The layer by default paints all labels for all objects on the location suggested by the label painter. To add decluttering behavior to this, you need to set up and configure an `ILcdGXYViewLabelPlacer`. [Section 36.3.1](#) details the LuciadLightspeed implementations of the label placers. Because a label placer operates on the labels of all layers, you need to configure it on the view by calling the `setGXYViewLabelPlacer` method. In the sample scenario, a `TLcdGXYLabelPlacer` is used: an algorithm-based label placer implementation which is configured with a `TLcdGXYLocationListLabelingAlgorithm`. This algorithm places as many labels as possible without introducing overlapping labels. Sample [Program 195](#) shows how to do this.

```

1 TLcdGXYLabelPlacer labelPlacer = new TLcdGXYLabelPlacer(
2     new TLcdGXYLocationListLabelingAlgorithm() );
3 map.setGXYViewLabelPlacer( labelPlacer );

```

Program 195 - Configuring the label placement

36.2 Painting labels

LuciadLightspeed provides the interface `ILcdGXYLabelPainter2` for painting labels. A label painter determines the content and visualization of your labels. It typically paints the label at a position near the domain object. For more information on how to influence where and when labels are placed, refer to [Section 36.3](#).

[Section 36.2.1](#) describes LuciadLightspeed's default implementations. One particular implementation, `TLcdGXYStampLabelPainter`, allows you to customize the painting of labels by providing a label stamp as explained in [Section 36.2.2](#). If you want complete control of the painting of labels, you can provide your own implementation of `ILcdGXYLabelPainter2` as explained in [Section 36.5.3](#). Finally, [Section 36.2.3](#) explains how to add a halo effect to your labels.

36.2.1 Choosing a label painter

As described in [Section 36.1](#), you need to install an `ILcdGXYLabelPainter2` on your `ILcdGXYLayer` to add labels for your domain objects. LuciadLightspeed provides several `ILcdGXYLabelPainter2` implementations in the packages `com.luciad.view.gxy` and `com.luciad.view.gxy.painter`:

- `TLcdGXYLabelPainter` is the default label painter for text-based labels and allows full configuration of the appearance of the label (the font, the color, whether or not the text should be in a frame, and more). Refer to the API reference to see the complete list of properties which you can modify to customize the appearance.

The text for the label is retrieved by using the `toString` method of the domain object that is labeled, but this can be overridden by overwriting the `retrieveLabels` method

in a subclass. There is an extension of this class: `TLcdGXYDataObjectLabelPainter` which takes the text from the properties of a domain object that implement `ILcdDataObject`.

- `TLcdGXYStampLabelPainter` delegates the actual painting of the label to a *label stamp*. This allows you to easily customize the graphical representation of a label without having to worry about its location. This stamp could, for example, paint the label as simple text, or could use a complex AWT/Swing component to paint the label. Refer to [Section 36.2.2](#) for more information on label stamps. This painter also support interactive labels, which are described in [Section 36.4.2](#).
- `TLcdGXYLabelPainterAdapter` wraps around a regular painter so it can be used as a label painter. Painting many domain objects usually results in a cluttered view, making selection more difficult and making it difficult to distinguish one object from another. Painting object representations as a label has two advantages. Firstly, it provides the possibility to paint labels at a certain offset which reduces cluttering as shown in [Figure 126](#).

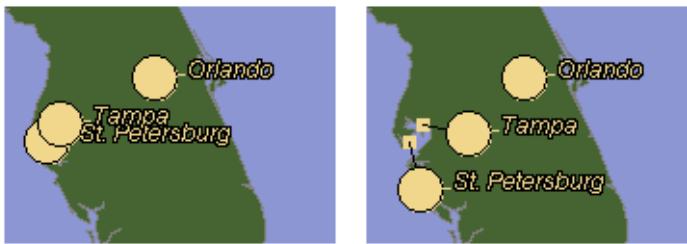


Figure 126 - Offset icons can reduce clutter and improve readability

Secondly, it provides the possibility to position the labels using the label placers as described in [Section 36.3.1](#). Refer to the `labels.offset.*` sample on how to use the `TLcdGXYLabelPainterAdapter`, and [Section 36.3.3](#) and [Section 36.3.3](#) for more information on using labels to represent the domain object.

- `TLcdGXYCompositeLabelPainter` allows you to combine different label painters for the different labels of an object. For example, it can hold a `TLcdGXYLabelPainterAdapter` to render an icon representation of the domain object and a `TLcdGXYLabelPainter` to render a text label attached to the icon.

36.2.2 Customizing labels using label stamps

The easiest way to implement your own labels is to setup a `TLcdGXYStampLabelPainter` with a custom `ALcdGXYLabelStamp` implementation.

Splitting up the label painter in the `TLcdGXYStampLabelPainter` and an `ALcdGXYLabelStamp` separates the information on where to draw the label from how to draw the label. The `TLcdGXYStampLabelPainter` takes care of determining the label location and then passes this position to the `ALcdGXYLabelStamp`.

In order to calculate the label location, the dimensions of the label must be known in advance. As this is related to how the labels are drawn, the `ALcdGXYLabelStamp` needs to provide this information. Therefore the `ALcdGXYLabelStamp` class contains the following three methods:

- `dimensionSFCT` determines the size of the label. This needs to be independent of the location, because the size is needed to determine the location.

- `paint` paints the label on the specified `java.awt.Graphics` instance. The `Graphics` is already translated and rotated, so the stamp can start painting at (0,0). Nevertheless, location and rotation information is provided, in case the stamp needs it. The `ALcdGXYLabelStamp` should make sure not to paint outside of the bounds that it has returned, as the labeling algorithms typically rely on the bounds to declutter the labels.
- `isTouched` determines if the label is present at the specified location. For example, when the label is circular, the bounds of that label, which are rectangular by definition, cover a larger area than is actually used by the label. This method then returns `false` for points that fall outside of the circle and `true` for those within the circle.

If `TLcdGXYStampLabelPainter` does not suit your needs, it is possible to make your own implementation of the interface `ILcdGXYLabelPainter2` from scratch as explained in Section 36.5.3.

36.2.3 Adding a halo to labels

Similarly as in Section 32.5, you can add a halo effect around a label. `TLcdGXYLabelPainter` and `TLcdGXYStampLabelPainter` have built-in support for halos. You can enable halos on other label painters using the class `TLcdGXYHaloLabelPainter`. This class wraps around an existing `ILcdGXYLabelPainter2`, adding a halo to anything drawn by the wrapped painter. The methods to configure the label halo functionality are the same as those mentioned in Section 32.5. Make sure to disable the image cache or to call the method `clearImageCache` on `TLcdGXYHaloLabelPainter` when the label is modified.

36.3 Working with label locations

It is possible to complement a label painter with an `ILcdGXYViewLabelPlacer`. Whereas `ILcdGXYLabelPainter2` determines the content and appearance of the labels, `ILcdGXYViewLabelPlacer` computes if and where the label is positioned.

A label placer takes multiple labels of a group of domain objects into account. The group can be as large as all the objects visible in the view, including objects of different layers. This allows more advanced placing of labels, to improve readability and avoid overlap.

Section 36.3.1 explains the label placer implementations and how they use an algorithm to compute the label positions. Section 36.3.2 details the label algorithm implementations. Section 36.3.3 explains where the label locations are stored and how you can customize them.

36.3.1 Using label placers

Because a label placer operates on the labels of all layers, it is configured on the view by calling the `setGXYViewLabelPlacer` method of the `ILcdGXYView` implementation. LuciadLightspeed offers two label placer implementations (see the `com.luciad.view.gxy.labeling` package), which you can configure both with a placement algorithm that encapsulates the actual positioning logic:

- `TLcdGXYLabelPlacer` is the default label placer implementation. It allows setting an obstacle provider to prevent the placing of labels at certain positions.
- `TLcdGXYAsynchronousLabelPlacer` offers the same functionality, but positions the labels in its own thread. Use this implementation in combination with asynchronously painted layers as described in Chapter 33.

36.3.2 Using label algorithms

You need to configure `TLcdGXYLabelPlacer` and `TLcdGXYAsynchronousLabelPlacer` with an `ILcdGXYLabelingAlgorithm` to determine which labels are placed and on which locations.

LuciadLightspeed offers the following labeling algorithms (see the package `com.luciad.view.gxy.labeling.algorithm`):

- `TLcdGXYLocationListLabelingAlgorithm` is a default decluttering algorithm, iterating over a number of customizable positions relative to the domain object's anchor point.
- `TLcdGXYOnPathLabelingAlgorithm` is a decluttering algorithm that places labels along the border of the domain object.
- `TLcdGXYInPathLabelingAlgorithm` is similar to `TLcdGXYOnPathLabelingAlgorithm` but positions labels inside the shape instead of on its perimeter.
- `TLcdGXYCurvedPathLabelingAlgorithm` is similar to `TLcdGXYOnPathLabelingAlgorithm` but drapes curved labels on the border of the domain object.
- `TLcdGXYCompositeLabelingAlgorithm` allows you to specify a label placement algorithm per layer, per object, or even per label. The composite algorithm partitions the labels into groups that are passed on to the relevant delegate algorithms. For more information on how to use composite algorithms, refer to the `label.placement.*` sample.
- `TLcdGXYCompositeDiscretePlacementsLabelingAlgorithm` is similar to `TLcdGXYCompositeLabelingAlgorithm` but allows interleaving the algorithms during the placement step. This means that you can, for example, first place a line label, then a point label, then another line label, and so on. For more information on how to use composite algorithms, refer to the `label.placement.*` sample.

One way to customize these algorithms is to wrap them, for example to further remove unwanted labels. To illustrate this, the `labels.createalgorithm.*` sample shows some possible algorithm wrappers. If you want even more control of label placement, you can consider implementing your own labeling algorithm as described in Section 36.5.5.

36.3.3 Customizing label locations and label dependencies

Storing and retrieving label locations

Objects that compute label positions, such as label placers or edit controllers (see Section 36.4.1) store these locations in the `ALcdLabelLocations` of an `ILcdGXYEditableLabelsLayer`.

With the `getLabelLocation` method of an `ALcdLabelLocations` instance you can retrieve the location information for a given label. This information needs to be interpreted by the label painter. The `ALcdLabelLocations` instance not only stores the label locations, but also which labels are painted. For example, in the case that a certain labeling algorithm (see Section 36.3.2) decides that there is not enough place on the screen to paint all labels. In this case you can use the method `applyOnPaintedLabelLocations` of the `ALcdLabelLocations` instance to go over all labels that are painted, for example to determine which labels are visible at a certain location on the screen.

Using label locations

All the information regarding the location of a label is contained in a `TLcdLabelLocation` instance. This instance can specify the location of the label in several ways: as a location index,

as a position relative to the labeled domain object, or as an absolute position on the view. The actual placing of the label depends on how the location is specified:

- If the `getLocationIndex` method returns a value smaller than zero, the label is freely placed and its position is specified by the `getLocationX` and `getLocationY` methods. This type of placement is used by most labeling algorithms. How these methods are interpreted depends on the label edit mode:
 - If the label edit mode indicates that the `TLcdLabelLocation` contains absolute information, the result of the methods is interpreted as absolute coordinates on the view.
 - If the label edit mode indicates that the `TLcdLabelLocation` contains a relative position, the result of the methods is interpreted as the horizontal and vertical offset with respect to a certain point of the domain object. What that point exactly is depends on the implementation of `ILcdGXYLabelPainter2`. It can, for example, be the anchor point of the domain object as specified by the `anchorPointSFCT` method of the `ILcdGXYPainter` of the domain object.
- If the `getLocationIndex` method of the `TLcdLabelLocation` instance returns a value larger than or equal to zero, the label location is placed on a discrete location. What the actual position is, depends on how the `ILcdGXYLabelPainter2` interprets this location index.

It is possible to subclass `TLcdLabelLocation` so it can contain more information. Refer to the API reference for more information.

Labels as domain object representations

`TLcdLabelLocation` allows declaring a label as being a domain object representation instead of a label of the domain object representation with the method `setBodyLabel`. This information is used by the `TLcdGXYEditControllerModel2` to influence editing behavior: moving the label moves the entire domain object. It can also be used to influence the paint order of labels. The main advantage of painting domain objects as labels is that they can reduce clutter because they can take part in the label placement process.

The `labels.offset.*` sample has an icon representation of its domain objects. These icon labels are modeled as body labels. The sample shows how to make a layer paint these body labels along with the regular domain object representations, and the other labels along with the other labels of the view.

Labeling labels

`TLcdLabelLocation` allows declaring a dependency on another label with the method `setParentLabel`. `ILcdGXYLabelPainter2` implementations can use this information to paint the label relative to this parent label, instead of the domain object. `ILcdGXYViewLabelPlacer` implementations can use this information to make sure that the parent labels are positioned first. Hence, you can use this to *label a label* which is especially useful when you are painting the object representation as a label. `TLcdLabelLocations` allows setting a dependency provider so that the label locations automatically have their parent labels configured.

The `labels.offset.*` sample shows how to use dependencies to paint labels relative to an icon representation of the object, which is in turn modeled as a label.

36.4 Interacting with labels

Just like with representations of domain objects, it is possible to interact with labels: users can select them and perform certain operations on them. Section 36.4.1 explains how to graphically edit labels so users can, for example, move them around. If you want more complex interaction, you can use interactive labels, which allow you to represent your label with any `java.awt.Component`. This is explained in Section 36.4.2.

36.4.1 Graphically editing labels

Similar to graphically editing domain objects with an `ILcdGXYEditor`, you can edit the labels of a domain with an `ILcdGXYLabelEditor`. The following sections provide more details on the main implementations of `ILcdGXYLabelEditor` and how to install and obtain an `ILcdGXYLabelEditor`.

Main implementations of `ILcdGXYLabelEditor`

`TLcdGXYLabelPainter` and `TLcdGXYStampLabelPainter` implement `ILcdGXYLabelEditor`, allowing users to move labels around. When asked to edit the location of a label, it modifies the `TLcdLabelLocation` by setting the location index to `-1` which indicates free placement. It also adjusts the location information so that the offset, specified in the given `ILcdGXYContext`, is added to the label position.

Installing and obtaining an `ILcdGXYLabelEditor`

`ILcdGXYLabelEditor` instances are obtained with the `getGXYLabelEditor` method of the `ILcdGXYEditableLabelsLayer` interface. This interface is an extension of `ILcdGXYLayer` which provides methods to support editable labels. `TLcdGXYEditController2` retrieves an `ILcdGXYLabelEditor` using this interface to offer mouse-based editing.

`TLcdGXYLayer` implements `ILcdGXYEditableLabelsLayer` and delegates the retrieval of a label editor to an `ILcdGXYLabelEditorProvider`. All the main implementations of `ILcdGXYLabelEditor` provided in LuciadLightspeed also implement this `ILcdGXYLabelEditorProvider` interface and return themselves as label editor for every object.

Program 196 shows how to install a label editor on a layer.

```
1 aCitiesLayer.setGXYLabelEditorProvider(composite);
```

Program 196 - Installing an `ILcdGXYLabelEditor` on an `ILcdGXYEditableLabelsLayer`
 (from samples/gxy/labels/interactive/MainPanel)

36.4.2 Working with interactive labels

An interactive label is a label with which the user can interact to modify properties related to the domain object. You can configure interactive labels using an `ALcdGXYInteractiveLabelProvider`. To display and edit interactive labels you can use a `TLcdGXYInteractiveLabelsController`.

Interactive labels are only shown when the user moves the mouse cursor over a regular label. This is done for performance reasons. In case the `ALcdGXYInteractiveLabelProvider` indicates that it can provide an interactive label for a certain label, it is asked to provide the interactive label. The interactive label is then presented to the user.

Implementing an ALcdGXYInteractiveLabelProvider

The most important method when implementing an `ALcdGXYInteractiveLabelProvider` is the `provideInteractiveLabel` method. This method returns the `java.awt.Component` with which the user can interact. This component can be any AWT/Swing component; it can be a single text field or a `JPanel` containing check boxes, combo boxes, and more. As there can only be one interactive label from the same provider at the same time, the implementation of this method can always return the same instance of the AWT/Swing component.

Additionally an `ALcdGXYInteractiveLabelProvider` provides the following methods:

- The `canProvideInteractiveLabel` method is used to determine if the `ALcdGXYInteractiveLabelProvider` can provide an interactive label for the given object and indices. This allows fine-grained control over which labels can be interactive and which not. This method is always called before the `ALcdGXYInteractiveLabelProvider` is asked to actually provide an interactive label.
- The method `canStopInteraction` is called to ask if the interactive label can be stopped. This is useful for validation purposes and similar cases. Consider, for example, a text field in the interactive label that contains invalid text. In this case the interactive label should not stop. Otherwise, the user cannot correct the text. If the method returns `false`, the `stopInteraction` method will not be called.
- The method `stopInteraction` tells the `ALcdGXYInteractiveLabelProvider` that the last provided interactive label should apply all outstanding changes, and prepare the label to be removed from the user interface. It returns a `boolean` to indicate if the provider was successful in stopping the interactive label. If the method returns `false`, the label will not be removed from the user interface and the user can keep interacting with the label. Consider, for example, an interactive label with a text field that contains invalid text. When the label is asked to stop the interaction, it can return `false` and change the text field.
- The `cancelInteraction` method tells the `ALcdGXYInteractiveLabelProvider` to prepare the interactive label to be removed from the user interface without applying any changes. This request cannot be denied, consequently this method does not return a `boolean`.

As mentioned before, there can only be one interactive label from the same provider at the same time. As a result, the methods of `ALcdGXYInteractiveLabelProvider` are always called in a certain order. There are never two calls to `provideInteractiveLabel` before the previous interaction is stopped or canceled. This allows you to attach all the necessary listeners to the domain object and the AWT/Swing components in the `provideInteractiveLabel` method, and to remove these listeners in the implementation of the `stopInteraction` and `cancelInteraction` methods.

Activating interactive labels

`TLcdGXYInteractiveLabelsController` uses a mouse listener to show interactive labels. When the mouse hovers over a label, it provides an interactive label for that label. For touch input, where no hover exists, the interactive label is put in place at the first touch.

It is also possible to bypass the automatic behavior. When the `setProvideInteractiveLabelOnMouseOver` method of `TLcdGXYInteractiveLabelsController` is set to `false`, no interactive labels are shown automatically. You can then ask the `ALcdGXYInteractiveLabelProvider` to provide an interactive label with the `provideInteractiveLa-`

bel method. To stop or cancel the label interaction, you can use the `stopInteraction` and `cancelInteraction` methods respectively as this no longer happens automatically either. Bypassing the automatic behavior allows you, for example, to implement a scenario in which a user cycles through the labels by pressing a certain key on the keyboard, making each label interactive one after the other.

Using mouse and touch events

In order to move the interactive labels, the `ALcdGXYInteractiveLabelProvider` can dispatch mouse or touch events that happen on the interactive label to the `ILcdGXYView`. The active `ILcdGXYController` can then receive the events and use those, for example, to move the label with the `ILcdGXYLabelEditor`. Which events are dispatched to the view and which events are dispatched to the interactive label is decided by the `dispatchAWTEvent` method of the `ALcdGXYInteractiveLabelProvider`. By default this method dispatches the event to the view when it happened on a `JLabel` or `JPanel`. It dispatches all other events to the originating component itself. As a result, when the user clicks and drags on a `JLabel`, the controller of the view can handle these events and can, for example, move the label. Clicking on, for example, a `JSlider` would as expected drag the knob. You can override this method to customize which events are dispatched to the interactive label and which are dispatched to the `ILcdGXYView`.

Whereas the `dispatchMouseEvent` is used strictly for mouse events, a more general method is available to dispatch AWT events: `dispatchAWTEvent`. Currently this method is called with touch events and mouse events. Mouse events are by default passed to the `dispatchMouseEvent` method by the `dispatchAWTEvent` method.

Placing the interactive label in a different java.awt.Container

By default the `TLcdGXYInteractiveLabelsController` uses the view itself as the `java.awt.Container` that is used to add the interactive label to the user interface. You can customize this by overriding `addComponentToGXYView` and related methods. Override this method, for example, if the `ILcdGXYView` implementation used in your application does not extend from `java.awt.Container`.

36.5 Advanced labeling topics

This section describes more advanced labeling topics that are not covered in the previous sections. Section 36.5.1 starts with explaining the labeling process.

36.5.1 The labeling process

Figure 127 shows the typical workflow of labeling domain objects in a view.

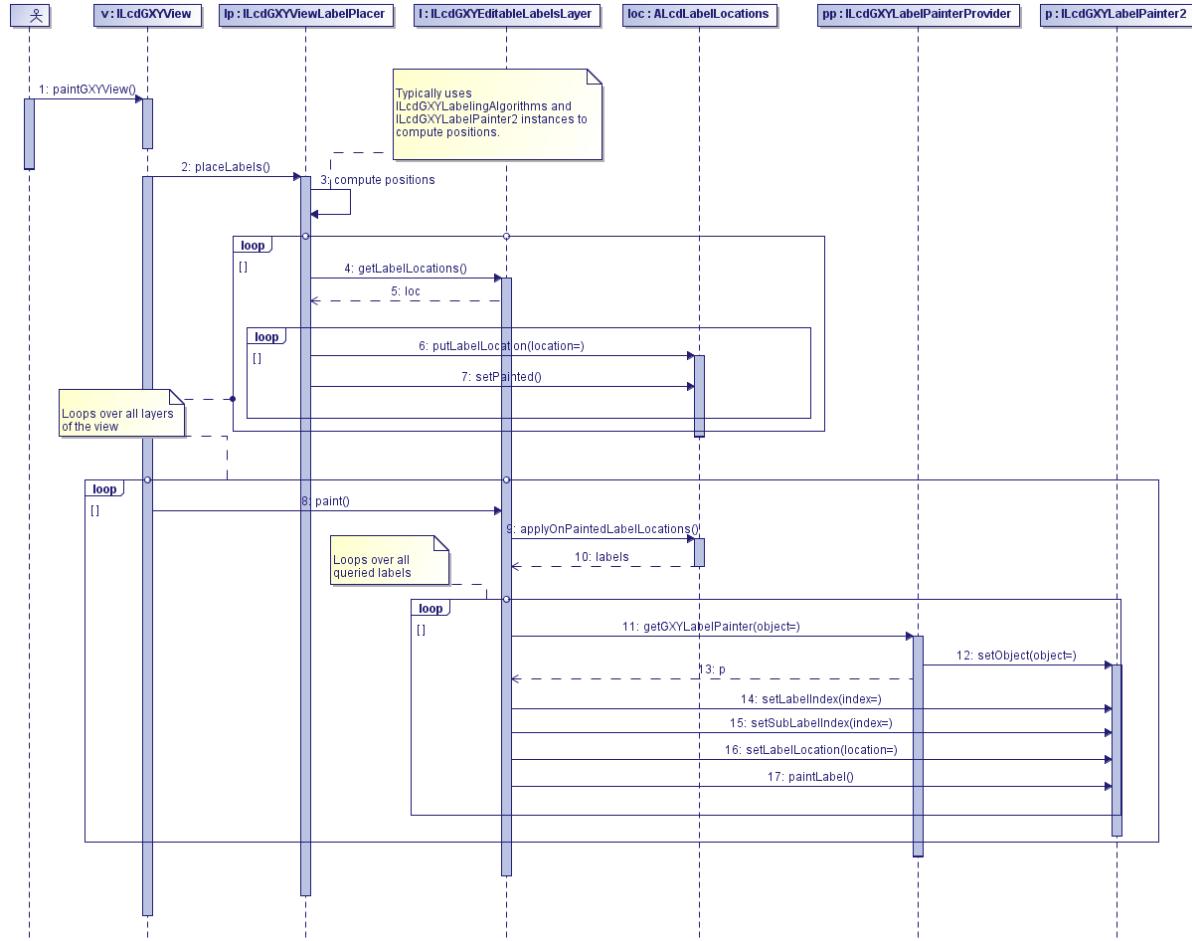


Figure 127 - Labeling sequence diagram

When the view receives a paint request through the method `paintGXYView()`, it asks the view label placer to compute positions for all layers. The view label placer collects relevant label information (for example, by querying the layer) and decides which labels get painted and where they are painted. This information is stored in the layer's label locations. After this, the view loops over all of its layers and asks each layer to paint their labels. In its turn the layer queries its stored label locations. For each label, a label painter is retrieved and asked to paint the label.

36.5.2 Using label painters and label editors programmatically

For some reasons you might want to access a label painter or label editor programmatically, for example to retrieve the bounds of a label, or to manually paint or edit a label. After setting

the domain object on the painter and/or editor using `setObject`, the following information is required:

- the label index of the label to work with, in case your domain object requires multiple labels (see `setLabelIndex`). It is up to the label painter implementation to determine how many labels it allows to paint (see `getLabelCount`).
- the sub label index of the label to work with, in case your domain object requires multiple labels (see `setSubLabelIndex`). It is up to the label painter implementation to determine how many sublabels it allows to paint (see `getSubLabelCount`).
- the location of the label to work with (see `setLabelLocation`). The location instance is used to position the label by the label painter. It is adjusted by the label editor in the `editLabel` method.

If you want to reuse the label location stored in the `ILcdGXYEditableLabelsLayer`, as determined, for example, by the view label placer, you can retrieve it from the layer's label location using `ALcdLabelLocations.getLabelLocationSFCT`, as described in Section 36.3.3. To store a label location that has been changed by a label editor use `ALcdLabelLocations.putLabelLocation`.

Program 197 shows how to access an `ILcdGXYLabelPainter2` to determine the bounds of a label.

```

1 //obtain the location information from the ALcdLabelLocations instance
2 label_locations.getLabelLocationSFCT( domain_object, label_index, sub_label_index, view,
3   label_location );
4
5 //let the label painter interpret this information
6 ILcdGXYLabelPainter label_painter = layer.getGXYLabelPainter( domain_object );
7 if ( label_painter instanceof ILcdGXYLabelPainter2 ) {
8   ILcdGXYLabelPainter2 label_painter_2 = ( ILcdGXYLabelPainter2 ) label_painter;
9   label_painter_2.setLabelIndex( label_index );
10  label_painter_2.setSubLabelIndex( sub_label_index );
11  label_painter_2.setLabelLocation( label_location );
12  label_painter_2.labelBoundsSFCT( graphics, mode, context, rectangleSFCT );
13  //rectangleSFCT now contains the location (in view coordinates) of the label

```

Program 197 - Determining the location of the labels

36.5.3 Implementing your own label painter

To implement an `ILcdGXYLabelPainter2`, you need to support the following major operations:

- `labelBoundsSFCT`: calculates the bounds and rotation of the label. The bounds of the label is stored in the given Rectangle and the rotation is returned as a double.
- `paintLabel`: actually paints the label.
- `isLabelTouched`: allows interaction with the label.
- `labelAnchorPointSFCT`: allows to retrieve an anchor point of the label for other application objects to use, for example as a base position for depending labels (see Section 36.3.3).

Optionally, you can provide snap targets for other editors to snap to with the `supportLabelSnap` and `labelSnapTarget` methods.

36.5.4 Implementing your own label editor

The most important method when implementing an `ILcdGXYLabelEditor` is `editLabel`. This method is responsible for changing the label location in response to the user interaction. The information about the input is available in the `ILcdGXYContext` instance that is passed as a parameter. How this input is interpreted depends on the mode that is passed as parameter.

It is the task of the editor to apply the information that is given in the context and the mode to the `TLcdLabelLocation` instance that was previously set on it. The class that asked the editor to apply the changes can then retrieve the `TLcdLabelLocation` instance. Thus, it is not the responsibility of the `ILcdGXYLabelEditor` to persist the changes, for example in an `ALcdLabelLocations` instance, but it is up to the client of the label editor to do this.

Because the label painter provides the visual indication while the user is editing the label, the implementation of the label editor is usually tightly coupled with the implementation of the label painter. This is similar to the situation with the painters and editors used to paint/edit the domain objects as described in [Section 32.2.1](#).

36.5.5 Implementing your own placement algorithm

To completely customize the label placement in your application, you can write your own `ILcdGXYLabelingAlgorithm`. A labeling algorithm has two major steps:

1. collect relevant label information (see the `collectLabelInfo` method)
2. determine label positions (see the `computeLabelPlacements` method)

Because the labeling itself may take place asynchronously, it is important to collect all relevant information that cannot be accessed asynchronously in the collect step.

Because labeling is often about iterating over a number of possible positions and checking if they overlap, a special class `ALcdGXYDiscretePlacementsLabelingAlgorithm` is provided to support this. To implement this algorithm, you need to implement the following:

- an iterator for the possible labels
- an iterator for the possible label positions of each label
- an evaluator that checks if a label position is qualified for placement. Usually an evaluator checks if a label overlaps with the already placed labels using an `ILcdLabelConflictChecker`.

The `labels.createalgorithm.*` sample shows how to create a simple `ALcdGXYDiscretePlacementsLabelingAlgorithm` that places all labels on the object's anchor point, avoiding overlapping labels.

CHAPTER 37

Managing your GUI and controllers in a GXY view

This chapter handles several topics that allow you to expand your application with specific options for user interaction and working with a GUI.

37.1 Defining GUI actions

To define a GUI action you typically use an `ILcdAction`. By using an `ILcdAction`, the functionality of a GUI component is separated from the effect it has on the application. Take, for example, a button that has an `ILcdAction` associated to it. When the button is pressed by the user, the button calls the method `actionPerformed` of the `ILcdAction`. The method `actionPerformed` has one argument which specifies the event (user interaction) that was invoked on the button. It is then the `ILcdAction` that defines the effect of this user interaction on the application. As you can see, the button has no direct effect on the application. It merely serves as a messenger that forwards user interaction to the application. The same behavior applies to a menu item in a menu bar.

37.1.1 What is an `ILcdAction`?

An `ILcdAction` is an extension of `java.awt.event.ActionListener` which applies the Command pattern. It is very similar to the Swing `javax.swing.Action` interface. To create a Swing Action from a given `ILcdAction` you simply create an instance of `TLcdSWAction`, which is a Swing wrapper around an `ILcdAction`.

The interface `ILcdAction` applies the Listener pattern in collaboration with `java.beans.PropertyChangeListener`. A `PropertyChangeListener` accepts `java.beans.PropertyChangeEvent` objects from the `ILcdAction` on which it is registered. For more information on listening to changes, refer to [Section 8.2](#).

37.1.2 Main implementations of `ILcdAction`

The main implementations of the interface `ILcdAction` for GXY views are:

- `TLcdOpenAction` to load an `ILcdModel` from a file. For more information on configuring an `ILcdAction` to load model data, refer to [Chapter 47](#).
- `TLcdSaveAction` to save an `ILcdModel` of an `ILcdView`.

- `TLcdMoveLayerAction` to move an `ILcdLayer` in an `ILcdLayered`.
- `TLcdGXYSetControllerAction` to make a given `ILcdGXYController` the active controller of the `ILcdGXYView` as illustrated in [Section 4.3](#).
- `TLcdGXYViewFitAction` to fit a particular `ILcdGXYLayer` in the bounds of the containing `ILcdGXYView` as shown in [Section 5.3.3](#).
- `TLcdPrintComponentAction` which prints the contents of a `java.awt.Component` on a printer as described in [Chapter 40](#).

Additionally, the sample `SetProjectionAction` shows how to set the `ILcdProjection` on an `ILcdGXYView`.

37.1.3 Implementing an `ILcdAction`

[Program 198](#) shows the class `RemoveLayerAction`, which is an implementation of the interface `ILcdAction`. `RemoveLayerAction` extends the abstract class `ALcdAction`, to have a default implementation of most of the methods.

```

1 class RemoveLayerAction extends ALcdAction implements ILcdAction {
2
3     ILcdLayered fLayered;
4     ILcdLayer fLayer;
5
6     public RemoveLayerAction() {
7         this( null, null );
8     }
9
10    public RemoveLayerAction( ILcdLayered aLayered, ILcdLayer aLayer ) {
11        fLayer = aLayer;
12        fLayered = aLayered;
13        setName( "RemoveLayer" );
14        setShortDescription( "Remove layer" );
15        setLongDescription( "Remove layer" );
16    }
17
18    public void actionPerformed( ActionEvent ae ) {
19        fLayered.removeLayer( fLayer );
20    }
21
22    public void setLayer( ILcdLayer aLayer ) {
23        fLayer = aLayer;
24    }
25
26    public void setLayered( ILcdLayered aLayered ) {
27        fLayered = aLayered;
28    }
29}
```

Program 198 - A sample implementation of the interface `ILcdAction`

The class `RemoveLayerAction` is used to remove an `ILcdLayer` from an `ILcdLayered`. The `ILcdLayer` and `ILcdLayered` objects are stored in the member variables `fLayer` and `fLayered`, respectively. They can be set using the constructor, and can be changed afterwards by the methods `setLayer` and `setLayered`.

The principal method of the class `RemoveLayerAction` is the method `actionPerformed`, which is used to perform the `ILcdAction`. Calling `actionPerformed` removes `fLayer` from `fLayered`.

37.2 Using and customizing controllers

A controller allows interaction with a view. It translates the low-level user interactions into higher-level operations on `ILcdGXYPainter` objects, `ILcdGXYEditor` objects, `ILcdGXYLayer` objects, and so on. The following sections describe how to use and customize the main implementations of `ILcdGXYController`.

37.2.1 Using and customizing a `TLcdGXYSelectController2`

A `TLcdGXYSelectController` allows a user to select objects on an `ILcdGXYView` using the mouse. Changes to the selection can be made by clicking on the view or by dragging a rectangle. Both the objects themselves and their labels are sensitive. Using modifier keys such as Shift and Alt allow you to further customize the behavior.

When a mouse click or drag is performed, the controller looks for the objects that are touched under the mouse pointer. For this, it loops over all layers and queries the objects using the method `applyOnInteract`. For each object, the painter is retrieved from the layer it belongs to and the method `isTouched` of the painter is called.

LuciadLightspeed provides a number of methods to customize this controller:

- `setDoubleClickAction` sets the action to be triggered when the mouse is double clicked. By default no action is triggered.
- `setRightClickAction` sets the action to be triggered when the right mouse button is pressed. By default no action is triggered.
- `setMouseDraggedSensitivity` sets the sensitivity of a mouse drag. If a dragged rectangle is greater than `getMouseDraggedSensitivity`, selection is done within the rectangle, otherwise selection is performed with a point location.
- `setSelectControllerModel` allows to customize all selection related logic by setting an (extension of) `TLcdGXYSelectControllerModel2` as described in [Section 37.2.1](#).

Customizing the selection logic

A `TLcdGXYSelectControllerModel2` contains the logic for selecting objects on an `ILcdGXYView`. For example, it can define which modifier keys (Shift, Alt) have what effects, whether the selection should change if a label is clicked, and so on.

The most important methods of this class, typically invoked in the following order, are:

- `selectByWhatMode`: Given a mouse event, returns if only objects themselves should be selectable, or if the labels are sensitive as well, if clicking or dragging should have an effect, ...
- `selectHowMode`: Given a mouse event, returns whether the chosen objects should be added to the selection, removed from the selection, if their selection state should be inverted, ...
- `selectionCandidates`: Given the selection point/rectangle and the two modes, retrieves those objects that are involved in the selection. For example all objects within the dragged rectangle.
- `applySelection`: Given the modes and the selection candidates, actually changes the selection state.

Defining the selectByWhatMode

The method `selectByWhatMode` defines by what objects can be selected: by clicking on their body (for example, the polyline, see `ILcdGXYPainter.BODY`), or by clicking on their label, or anything else. The possibilities for the select-by-what mode are:

- `SELECT_BY_WHAT_BODIES_ON_CLICK`: Mode defining that clicking on the body of an object can change the selection.
- `SELECT_BY_WHAT_BODIES_ON_DRAG_CONTAINS`: Mode defining that dragging over the body of an object can change the selection, if the body lies within the selection rectangle.
- `SELECT_BY_WHAT_BODIES_ON_DRAG_OVERLAPS`: Mode defining that dragging over the body of an object can change the selection, if the body overlaps with the selection rectangle. Note that this relies on `ILcdGXYPainter.boundsSFCt` and therefore this mode might not behave intuitively if the bounds of the object are different from the actual representation of the object, for example for an L-shaped polyline.
- `SELECT_BY_WHAT_LABELS_ON_CLICK`: Mode defining that clicking on the label of an object can change the selection.
- `SELECT_BY_WHAT_LABELS_ON_DRAG_CONTAINS`: Mode defining that dragging over the label of an object can change the selection, if the label lies within the selection rectangle.
- `SELECT_BY_WHAT_LABELS_ON_DRAG_OVERLAPS`: Mode defining that dragging over the label of an object can change the selection, if the label overlaps with the selection rectangle. The same remark as for `SELECT_BY_WHAT_BODIES_ON_DRAG_OVERLAPS` applies here as well.

Typically a bitwise or of these constants is returned, for example the default: `SELECT_BY_WHAT_BODIES_ON_CLICK | SELECT_BY_WHAT_BODIES_ON_DRAG_CONTAINS | SELECT_BY_WHAT_LABELS_ON_CLICK | SELECT_BY_WHAT_LABELS_ON_DRAG_CONTAINS`.

Defining the selectHowMode

The result of the method `selectHowMode` defines what should happen with the newly selected objects: should they be added to the current selection, removed, or anything else. The possibilities for the select-how-mode are:

- `SELECT_HOW_FIRST_TOUCHED`: Mode defining that the first (topmost) object should be selected, and that all other objects that were previously selected should be deselected. This mode is used if no modifier keys are pressed.
- `SELECT_HOW_CHOOSE`: Mode defining that, in case multiple objects are candidate for selection (for example, several objects overlap), the user should choose which object(s) to select. The default implementation is to pop up a menu with a String representation of each touched object; a separator is drawn between objects of different layers. The user can select the desired object in this menu. This mode is typically combined (bitwise or) with another mode, such as `SELECT_HOW_ADD`. `TLcdGXYSelectController2` returns this mode when the Alt-key is pressed while selecting.
- `SELECT_HOW_INVERT`: Mode defining that the current selection state of the candidates for the selection should be toggled: those that were selected are deselected, those that were deselected are selected. This mode is used when the Shift-key is pressed while selecting.

- `SELECT_HOW_ADD`: Mode defining that the candidates for the selection should be added to the current selection. This mode is not returned by `TLcdGXYSelectController2` (except in combination with `SELECT_HOW_CHOOSE`). Subclasses can however return this mode for certain modifier key combinations.
- `SELECT_HOW_REMOVE`: Mode defining that the candidates for the selection should be removed from the current selection. This mode is not returned by `TLcdGXYSelectController2`, but subclasses can return this mode for certain modifier key combinations.
- `SELECT_HOW_NO_CHANGE`: Mode defining that no changes must be made to the selection. This mode is returned for modifier key combinations that are not mentioned for the other modes as mentioned above.

Defining selectionCandidates and applySelection

The method `selectionCandidates` retrieves the objects that are involved in the selection. If, for example, a click is performed at a certain location on the map, this method returns the objects under the mouse position. Its behavior depends on the selected mode. For example, it returns all objects under the mouse position (see `SELECT_HOW_CHOOSE`), or only the first one that is encountered (see `SELECT_FIRST_TOUCHED`).

The method `applySelection` updates the state of the selection, respecting the modes and the candidates that were involved. It can, for example, add the candidates to the selection (see `SELECT_HOW_ADD`), for every selection candidate toggle its selection state (see `SELECT_HOW_INVERT`), or display a pop-up menu allowing the user to decide which object should be picked from the available candidates (see `SELECT_HOW_CHOOSE`).

Subclasses can override these methods to alter their behavior. It can often be useful to invoke the method of the super class with modified parameters.

37.2.2 Using and customizing a `TLcdGXYEditController2`

`TLcdGXYEditController2` is an extension of `TLcdGXYSelectController2` with additional editing capabilities. It allows a user to graphically edit objects on an `ILcdGXYView` using the mouse.

The selected objects are translated by moving the mouse pointer over them and dragging them to a new location. Selecting and dragging can be performed as one operation. If the Ctrl-key is pressed during the drag, the single object that is touched by the mouse pointer is reshaped. Note that the meaning of translating and reshaping is defined by the implementations of `ILcdGXYPainters` and `ILcdGXYEditors`.

Both the objects themselves (bodies) and their labels can be edited. Editing an object requires that an `ILcdGXYEditor` is available from the layer, editing the labels requires that an `ILcdGXYLabelEditor` and an `ILcdGXYLabelPainter2` (not just `ILcdGXYLabelPainter`) are available from the layer.

You can customize all editing related logic by setting a custom `TLcdGXYEditControllerModel2` using the method `setEditControllerModel`. For example, to change the behavior of modifier keys, disable editing of labels, influence which objects should be edited, and so on. The controller model for the edit controller has a similar structure as the `TLcdGXYSelectControllerModel2` mentioned in [Section 37.2.1](#). Refer to the API reference of `TLcdGXYEditControllerModel2` for more information.

`TLcdGXYEditController2` supports undo and snapping capabilities. Refer to [Section 37.4](#) and [Section 35.5](#) respectively for more information.

When passing an `TLcdGXYContext` to the `TLcdGXYEditController2` instead of an `ILcdGXYContext`, it is possible to provide multiple input points at once. This allows to edit multiple points at the same time as illustrated in the `touch.multiEdit` sample.

37.2.3 Using and customizing a `TLcdGXYNewController2`

`TLcdGXYNewController2` allows a user to graphically create objects on an `ILcdGXYView` using the mouse. The controller needs a controller model extending from `ALcdGXYNewControllerModel2` to translate mouse events to the steps necessary for creating and initializing a new object. The default controller model uses the object's target layer's `ILcdGXYPainter` and `ILcdGXYEditor`, which already offer most of the functionality for creating new objects and visualizing the creation. To instantiate the `ALcdGXYNewControllerModel2`, you only need to implement two more methods:

- `getGXYLayer` to retrieve a relevant `ILcdGXYLayer` to insert the object into. By default, the controller model delegates painting and editing steps to the layer's `ILcdGXYPainter` and `ILcdGXYEditor`, and commits the object to the layer when the object is successfully initialized.
- `create` to create a new object ready to be initialized by the mouse.

You can further customize the creation process in the following two ways:

- customize the key actions involved in the creation process:
 - `edit`: to perform a mouse-based initialization step
 - `cancel`: to cancel the creation process
 - `revert`: to go back a step in the creation process
 - `paint`: to paint the creation in progress
 - `commit`: to insert the object into the target layer
- customize the translation of mouse events into the above actions

Refer to the API reference of `ALcdGXYNewControllerModel2` for more information on customizing the creation process. [Section 35.7.1](#) explains implementing a controller model to create a hippodrome shape. It includes a diagram showing the interaction between the controller and the controller model.

`TLcdGXYNewController2` supports undo and snapping capabilities. Refer to [Section 37.4](#) and [Section 35.5](#) respectively for more information.

37.2.4 Using touch controllers

If you are using a touch-based input device, you can make use of the built-in touch controllers. These controllers are specifically designed for touch input, making use of multi-touch input and providing visual and touch-based alternatives for the traditional mouse and keyboard input.

The main touch implementations are:

- `TLcdGXYTouchSelectEditController`: to select and edit objects in an `ILcdGXYView`. It has the same select controller model as used by `TLcdGXYSelectController2`, and an edit controller model that is very similar to the one used by `TLcdGXYEditController2`.

- `TLcdGXYTouchNewController`: to create new objects in an `ILcdGXYView`. It has the same new controller model as used by `TLcdGXYNewController2`.
- `TLcdGXYTouchNavigateController`: to pan, zoom, and/or rotate the `ILcdGXYView`.
- `TLcdMapTouchRulerController`: to measure distances in the `ILcdGXYView`.

The standard controllers all extend from `ALcdGXYChainableController`, which allows controllers to be chained. If one controller cannot handle the input, or only partially, the next controller in the chain can react as well on this input. A possible use case is to chain the `TLcdGXYTouchSelectEditController` with a `TLcdGXYTouchNavigateController`. Such a controller would allow to select, edit, and navigate without switching controllers.

Note that in order to use these controllers, you need a compatible touch device. Refer to Section 37.3 for more information on the touch device support in LuciadLightspeed.

37.2.5 Creating a custom touch controller

LuciadLightspeed offers `ALcdGXYTouchChainableController`, a base class which simplifies the process of writing your own implementation of `ILcdGXYChainableController`. This base class provides three abstract methods that allow you to indicate which points will be handled by the controller, and how to react to changes to those points:

- `touchPointAvailable`: this method is called every time a new touch point becomes available and determines which points are handled by the controller.
- `touchPointMoved`: this method is called whenever one of the tracked touch points has moved. It allows to stop tracking certain points.
- `touchPointWithDrawn`: this method is called every time a touch point is no longer available. Similar to the `touchPointMoved` method, it allows to stop the tracking of certain points by this controller.

When touch points are tracked by the custom controller, utility methods are available to retrieve the original, previous, and current locations of the tracked point(s). Refer to the API reference for more information on `ALcdGXYTouchChainableController`.

37.3 Handling non-standard input

This section explains how you can dispatch custom `TLcdTouchEvent` instances and other input events into your LuciadLightspeed application. It also contains the necessary information for making your custom touch device work with the existing LuciadLightspeed touch controllers.

37.3.1 Dispatching custom input events

To dispatch your custom `TLcdAWTEvent` instances to the GXY views, you can make use of the `dispatchEvent` method of `Component`. The GXY views by default forward all received `TLcdAWTEvent` instances to its controller if the latter implements `ILcdAWTEventListener`.

Take care when determining the target of your custom `TLcdAWTEvent`. You might end up dispatching your event to a `Component` that lies on top of the view.

37.3.2 Receiving LuciadLightspeed touch events

LuciadLightspeed dispatches its supported touch events by mapping the first touch point to a target component. It then uses the `dispatchEvent` method of the target component to get the event to the component. To handle touch events in your component, you can override `processEvent`. Take care if your component has siblings: the event might wind up in a different component than the one you expect. Note that the view implementations contain methods to add and/or remove `ILcdAWTEventListener` instances. Just like with `MouseListener`s and `KeyListener`s, as soon as one such a listener is registered, the corresponding incoming events are handled by this component. The events are handled by passing the event to the registered listeners. For these classes, there is no need to override the `processEvent` method.

To receive `TLcdAWTEvent` instances for the entire application, you can make use of `TLcdAWTEventDispatcher`. This is the central access point to dispatch `TLcdAWTEvents` into the application. It allows you to register `ILcdAWTEventListeners` to be notified of any dispatched `TLcdAWTEvent`.

37.3.3 Creating and understanding touch events

LuciadLightspeed provides support for all Windows 7 Touch compatible devices with Oracle's 1.8 JDK implementation. You can check if your device is supported with the method `TLcdTouchDevice.getTouchDeviceStatus()`. The `TLcdTouchDevice` class also encapsulates some touch specific settings, for example, to configure double tap behavior.

If you have a custom touch device or are working on another operating system and want to get its output to the LuciadLightspeed touch controllers, you can create your own `TLcdTouchEvent` instances.

The most important properties of touch events are:

1. each touch event contains all fingers that are touching the screen
2. all touch events generated from the moment a finger touches the screen have the same ID until all fingers are lifted again
3. a touch event only includes one single change, such as one moved finger, or one lifted finger
4. the event's target component is that of the first finger touching the screen
5. a touch event includes double-tap information (as Java's `MouseEvent`). A double-tap originates in two separate events: one for the first tap, and one for the second.

The main classes are `TLcdTouchEvent`, an extension of `AWTEvent` representing (multi-)touch input and `TLcdTouchPoint`, a class describing the state of one touch input point.

Associated with every touch input point is a `TLcdTouchPoint`, describing the (change in) state of a single touch input point. These `TLcdTouchPoint` instances are then used in every `TLcdTouchEvent` instance to describe the whole touch input.

A touch point can have four different states which are grouped in the `TLcdTouchPoint.State` enumeration. When touch points change state, the application is warned by means of a `TLcdTouchEvent`. This event contains `TLcdTouchPoint` instances for every touch point, both the one that has changed state as well as the ones which remained unchanged. Hence there exists a one-to-one mapping between changes in the touch input points and the created `TLcdTouchEvent` instances. Every `TLcdTouchEvent` instance contains exactly one `TLcdTouchPoint` with a non-stationary touch point state.

Every touch point has a unique identifier which remains constant for a touch point as long as it exists. This identifier is assigned when the touch point is created, and all other `TLcdTouchPoint` instances describing (a change in) the state of that particular touch point will have the same identifier. As long as the touch point exists, no other existing touch point from the same touch device and the same user may have the same identifier.

This allows to track movements and/or state of multiple touch points simultaneously. As soon as a touch point is removed and the tap count no longer increases, this identifier becomes available and may be reused to describe the state of a new touch point.

Similar to the `TLcdTouchPoints`, every `TLcdTouchEvent` instance has a unique ID. This ID is assigned when the first touch point is created, and remains the same until all touch points are removed.

Once all touch points are removed, the next touch point that is created:

- Generates a new `TLcdTouchEvent` with the same ID as the current event if, and only if, the new touch point increases the tap count of one of the previously removed touch points.
- Generates a new `TLcdTouchEvent` with an ID different from the one of the current event. The ID of the current event may be reused later on.

Note that both the ID of the `TLcdTouchPoint` and `TLcdTouchEvent` instances are only unique for a certain user and device combination. You can retrieve the identifiers for the user and the device from the `TLcdTouchEvent` instances.

The creation of `TLcdTouchEvents` is demonstrated in the `touch.touchEvents` sample. In this sample, touch hardware is simulated by an object that generates a stream of hardware events. These hardware events are then converted to `TLcdTouchEvents` and dispatched.

37.4 Adding undo/redo support

The `ILcdUndoable` interface provides support to reverse the effect of an action. A class that wants its effects to be reversible should provide methods to register an `ILcdUndoableListener`. Whenever an `ILcdUndoable` object is created by the class, the listener gets notified of this object and can react to it in an appropriate way. One implementation of `ILcdUndoableListener` is `TLcdUndoManager`. This class collects all `ILcdUndoable` objects of which it is notified and provides methods to undo and redo these undoable objects in the correct order. The following sections describe the steps for adding undo support to your application in more detail. For more information on listening to changes, refer to [Section 8.2](#).

37.4.1 Adding undo/redo capabilities to your application

To add undo/redo support to your application, you should first create a `TLcdUndoManager`. You can then add this `TLcdUndoManager` as a listener to the appropriate class(es). The `TLcdUndoAction` and `TLcdRedoAction` can then be used to interact with the `TLcdUndoManager`. [Program 199](#) demonstrates how to create a `TLcdUndoManager` with the undo and redo actions.

```

1 //create the undo manager
2 TLcdUndoManager undoManager = new TLcdUndoManager(10);
3
4 // Set up the actions that interact with the undo manager
5 TLcdUndoAction undoAction = new TLcdUndoAction(undoManager);
6 TLcdRedoAction redoAction = new TLcdRedoAction(undoManager);
7
8 //insert these actions in the toolbar
9 getToolBars()[0].addAction(undoAction);
10 getToolBars()[0].addAction(redoAction);

```

Program 199 - Creating a TLcdUndoManager and its corresponding actions
 (from samples/gxy/undo/MainPanel)

Program 200 shows how the TLcdUndoManager is added as an ILcdUndoableListener to the TLcdGXYEditController2. The edit controller creates ILcdUndoable objects when it edits the domain objects, and notifies the TLcdUndoManager of these objects. The TLcdUndoManager then places this ILcdUndoable in its queue so that it can be undone and redone when the TLcdUndoAction and TLcdRedoAction are performed.

```

1 //add the TLcdUndoManager as an ILcdUndoableListener to the edit controller.
2 aEditController.addUndoableListener(aUndoManager);

```

Program 200 - Making the changes made by the edit controller undoable
 (from samples/gxy/undo/MainPanel)

37.4.2 Making graphical edits undoable

As demonstrated above, the TLcdGXYEditControllerModel2 provides undo/redo support for editing the domain objects. However, because the TLcdGXYEditController2 delegates the actual modification of the domain objects to ILcdGXYEditor instances, the creation of the ILcdUndoable objects must be delegated to them as well. In order to let the TLcdGXYEditController2 capture these ILcdUndoable objects, it needs to attach itself as a listener to the ILcdGXYEditor. That is why the ILcdGXYEditor implementation needs to implement ILcdUndoableSource as well if the changes made by the ILcdGXYEditor need to be reversible. This interface allows ILcdUndoableListener objects to be attached.

The undo sample in the LuciadLightspeed distribution has an implementation of an ILcdGXYEditor that demonstrates these concepts. Section 37.4.3 analyzes this sample in more detail.

37.4.3 Describing the undo sample

The LuciadLightspeed distribution contains the undo sample which demonstrates the undo/redo support described in this chapter. It initializes the undo/redo support as seen in Program 199. The major part of the sample is however dedicated to the ILcdGXYEditor and ILcdUndoable implementation and the domain objects which it edits.

Program 201 shows the relevant portion of the editor. The editor saves the state of the object it is about to change right before the actual change and right after that. These states are saved in the undoable and the listeners are notified of this undoable.

```

1  @Override
2  public boolean edit(Graphics aGraphics, int aMode, ILcdGXYContext aContext) {
3      StateAware edited_object = (StateAware) getObject();
4
5      // Create a ModelElementEditedUndoable and store the original state of the object
6      // that is about to be edited.
7      // Note that TLcdGXYNewController2 uses the creation undoables internally (i.e. backspace
8      // will
9      // undo a single creation step) and generates a single undoable for the addition of the
10     // object
11     // when creation is finished.
12     ModelElementEditedUndoable undoable = null;
13     try {
14         undoable = new ModelElementEditedUndoable(
15             generateDisplayName(edited_object),
16             edited_object,
17             aContext.getGXYLayer(),
18             false
19         );
20     } catch (StateException ignored) {
21     }
22
23     boolean object_changed = super.edit(aGraphics, aMode, aContext);
24
25     finishAndFireUndoable(undoable, object_changed);
26
27     return object_changed;
28 }
```

Program 201 - Saving the state of the objects before and after the change
 (from samples/gxy/undo/UndoablePointListPainter)

```

1  private void finishAndFireUndoable(ModelElementEditedUndoable aUndoable, boolean
2      aObjectChanged) {
3      if (aUndoable != null) {
4          if (aObjectChanged) {
5              aUndoable.finishedEditingAndFire(fUndoSupport::fireUndoableHappened);
6          } else {
7              aUndoable.die();
8          }
9      }
10 }
```

Program 202 - Wrapping up the undoable and notifying the listeners of it
 (from samples/gxy/undo/UndoablePointListPainter)

The ILcdUndoable implementation is based on a variation of the memento pattern. The object that is about to be edited is asked to store its state right before and right after the change. Actually reverting the change and redoing it then only consist of telling the domain object to restore itself to the appropriate state, as demonstrated in [Program 203](#).

```

1  @Override
2  protected final void undoImpl() throws TLcdCannotUndoRedoException {
3      restoreState(fBeforeMap);
4  }
5
6  @Override
7  protected final void redoImpl() throws TLcdCannotUndoRedoException {
8      restoreState(fAfterMap);
9  }
10
11 private void restoreState(Map aState) {
12     StateAware editedObject = getEditedObject();
13     Object domainObject = getModelObject();
14     ILcdModel model = getModel();
15     try {
16         if (!fFireEvents) {
17             editedObject.restoreState(aState, model);
18         } else {
19             ILcdLayer layer = getLayer();
20             try (Lock autoUnlock = writeLock(model)) {
21                 editedObject.restoreState(aState, model);
22                 model.elementChanged(domainObject, ILcdFireEventMode.FIRE_LATER);
23             } finally {
24                 model.fireCollectedModelChanges();
25             }
26         if (layer != null) {
27             if (!layer.isVisible()) {
28                 layer.setVisible(true);
29             }
30             layer.selectObject(domainObject, true, ILcdFireEventMode.FIRE_NOW);
31         }
32     } catch (StateException e) {
33         LOGGER.error(e.getMessage(), e);
34         throw new TLcdCannotUndoRedoException(e.getMessage(), e);
35     }
36 }
37 }
```

Program 203 - Undoing and redoing actions consist of simply restoring the correct state
 (from samples/common/undo/ModelElementEditedUndoable)

How the state of the domain object is stored and restored depends on the object itself. In the case of this sample, the object is a Polyline which stores its state by storing the location of each of its points. It restores its state by moving each of its points to the stored location as shown in [Program 204](#).

```
1  @Override
2  public void storeState(Map aMap, ILcdModel aSourceModel) {
3      //store the location of the points.
4      int count = getPointCount();
5      ILcdPoint[] points = new ILcdPoint[count];
6      for (int i = 0; i < count; i++) {
7          // create separate clones of the point, to make this stored state
8          // independent of the actual state of the polyline.
9          points[i] = (ILcdPoint) getPoint(i).clone();
10     }
11     aMap.put(POINTS_KEY, points);
12 }
13
14 @Override
15 public void restoreState(Map aMap, ILcdModel aTargetModel) {
16     ILcdPoint[] points = (ILcdPoint[]) aMap.get(POINTS_KEY);
17     if (points != null) {
18         equalizeNumberOfPoints(points);
19         for (int i = 0; i < points.length; i++) {
20             move2DPoint(i, points[i].getX(), points[i].getY());
21         }
22     }
23 }
```

**Program 204 - Storing and restoring the state of a polyline comes down to storing and
restoring the location of its points**
(from samples/gxy/undo/Polyline)

CHAPTER 38

Retrieving height data for 2D points

The `com.luciad.util.height` package contains general interfaces and implementations to retrieve heights from elevation rasters or other data. The two interfaces are:

- `ILcdHeightProvider`
- `ILcdModelHeightProviderFactory`

38.1 Using an `ILcdHeightProvider`

The interface `ILcdHeightProvider` provides height values for 2D points inside known bounds. The spatial reference of the points, of the bounds, and of the resulting elevation is determined by the implementation of this interface. When no data is found at a certain point, the returned height value is NaN.

38.1.1 Standard implementations of `ILcdHeightProvider`

The `height` package contains six standard `ILcdHeightProvider` implementations:

- `TLcdFixedHeightProvider`: returns a constant height for any given point. The bounds are not taken into account.
- `TLcdImageHeightProvider`: retrieves height values from a given image.
- `TLcdRasterHeightProvider`: retrieves height values from a given raster. A similar implementation is `TLcdInterpolatingRasterHeightProvider`. The latter also retrieves height values from a given raster, but uses interpolation between height values to obtain a smoother result.
- `TLcdTransformedHeightProvider`: retrieves height values from another height provider with a different reference.
- `TLcdCompositeHeightProvider`: combines a list of height providers. All height providers must have the same reference as the composite height provider. When height providers have overlapping bounds, the first height provider that returns a valid height prevails.

38.1.2 Typical usage of an `ILcdHeightProvider`

In most cases two or more height providers are combined into one height provider. Figure 128 shows an example of such a height provider.

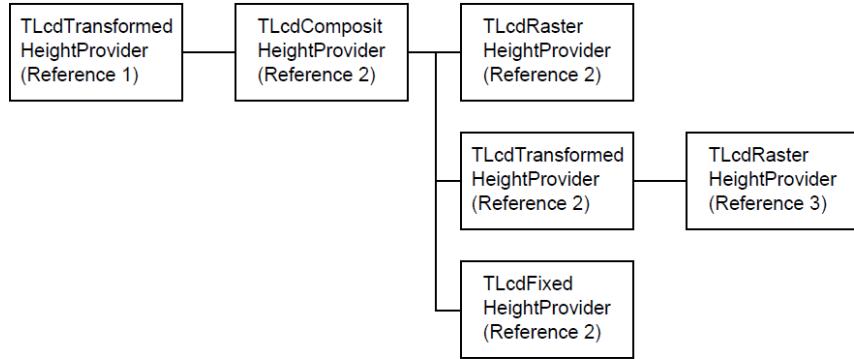


Figure 128 - A height provider composed of multiple height providers

When querying a height, the 2D point should be defined in reference 1. The point is then transformed to reference 2 and passed to the composite height provider. This height provider first tries to retrieve a height from the first raster height provider. When NaN is returned, it tries to retrieve a height from the next transformed height provider. When this height provider returns NaN, a fixed value is returned from the fixed height provider.

38.2 Creating an `ILcdHeightProvider`

Use the `ILcdModelHeightProviderFactory` interface to create an `ILcdHeightProvider` instance from a model. The `ILcdModelHeightProviderFactory` interface has four standard implementations:

- `ALcdModelHeightProviderFactory`: returns an `ILcdHeightProvider` instance from a model using an abstract method that creates an `ILcdHeightProvider` from a model element.
- `TLcdImageModelHeightProviderFactory`: returns an `ILcdHeightProvider` instance from an image model. It is an extension of `ALcdModelHeightProviderFactory` and implements the abstract method for image model elements.
- `TLcdRasterModelHeightProviderFactory`: returns an `ILcdHeightProvider` instance from a raster model. It is an extension of `ALcdModelHeightProviderFactory` and implements the abstract method for raster model elements. These raster model elements can be an `ILcdRaster` or `ILcdMultilevelRaster`.
- `TLcdCompositeModelHeightProviderFactory`: returns an `ILcdHeightProvider` instance from a model using a list of other `ILcdModelHeightProviderFactory` instances. The first model height provider factory that creates a height provider prevails.

38.2.1 Using properties

When using an `ILcdModelHeightProviderFactory` to create an `ILcdHeightProvider`, you can use properties to provide additional arguments. You can classify properties as required or optional:

- **Required:** when you want to make sure that this property is actually used. When this property is not used during the creation of an `ILcdHeightProvider`, a `TLcdUnsupportedPropertyException` must be thrown. When implementing an `ILcdModel-`

`HeightProviderFactory`, make sure that required properties are removed when they are used, and that the list of required properties is verified when a height provider is created.

- **Optional:** when it is not necessary to use this property.

When implementing an `ILcdModelHeightProviderFactory`, you can also classify properties as follows:

- **Necessary:** the implementation needs this property to work correctly. When a necessary property is not provided, a `TLcdMissingPropertyException` must be thrown.
- **Not necessary:** the implementation does not need this property to work correctly.

Properties that are often used are:

- `ALcdModelHeightProviderFactory.KEY_GEO_REFERENCE`: `ALcdModelHeightProviderFactory` uses this key as a necessary property. The given reference is used as the reference of the height provider. This means that the given 2D points, the bounds, and the returned height are expressed in function of this reference.
- `ALcdRasterModelHeightProviderFactory.KEY_PIXEL_DENSITY`: This property represents a desired pixel density. The implementations use this property to select an image or raster with a similar pixel density from multilevel data. The pixel density should be expressed in function of the given `KEY_GEO_REFERENCE` property. If this property is not provided, or set to `Double.NaN`, the most detailed level is chosen.
- `ALcdRasterModelHeightProviderFactory.KEY_INTERPOLATE_DATA`: This property is used to indicate whether the data should be interpolated.

Program 205 shows the typical usage of an `ILcdModelHeightProviderFactory`.

```

1 // Create a height provider factory.
2 ILcdModelHeightProviderFactory factory = new TLcdRasterModelHeightProviderFactory();
3
4 try {
5     // Specify the desired properties for the height provider.
6     Map<String, Object> requiredProperties = new HashMap<String, Object>();
7     ILcdGeoReference geoReference = new TLcdGeodeticReference();
8     requiredProperties.put(ALcdModelHeightProviderFactory.KEY_GEO_REFERENCE, geoReference);
9
10    Map<String, Object> optionalProperties = new HashMap<String, Object>();
11    optionalProperties.put(TLcdRasterModelHeightProviderFactory.KEY_INTERPOLATE_DATA, true);
12
13    // Create a height provider from a raster model.
14    ILcdHeightProvider heightProvider =
15        factory.createHeightProvider(aModel, requiredProperties, optionalProperties);
16
17    // Retrieve height values.
18    double height = heightProvider.retrieveHeightAt(new TLcdLonLatPoint(10.0, 20.0));
19
20} catch (TLcdMissingPropertyException ex) {
21    // A necessary property is missing : see ex.getMissingPropertyName()
22    // ...
23} catch (TLcdUnsupportedPropertyException ex) {
24    // A required property was not used during the creation of the height provider :
25    // see ex.getUnsupportedProperties()
26    // ...
27}

```

Program 205 - Typical creation of an `ILcdHeightProvider` using an `ILcdModelHeightProviderFactory`

38.3 Using a TLcdViewHeightProvider

A special instance of `ILcdHeightProvider` is `TLcdViewHeightProvider`. This height provider uses a view and an `ILcdModelHeightProviderFactory`. It uses the view to retrieve a list of models. After that it uses the height provider factory internally to create a height provider. When layers are added or removed from the view, or made visible or invisible, the height provider updates itself. [Program 206](#) shows how to create a `TLcdViewHeightProvider`.

```
1  /*
2   * Create a height provider using the entire view as height source.
3   */
4  private ILcdHeightProvider createHeightProvider() {
5      ILcdModelHeightProviderFactory factory = new TLcdImageModelHeightProviderFactory();
6
7      Map<String, Object> requiredProperties = new HashMap<String, Object>();
8      Map<String, Object> optionalProperties = new HashMap<String, Object>();
9      requiredProperties.put(ILcdModelHeightProviderFactory.KEY_GEO_REFERENCE,
10                          fGeoReference);
11
12     return new TLcdViewHeightProvider<ILcdGXYView>(getView(),
13                                                 factory,
14                                                 requiredProperties,
15                                                 optionalProperties);
16 }
```

Program 206 - Typical creation of a `TLcdViewHeightProvider`
(from `samples/gxy/height/MainPanel`)

PART V Common Topics for GXY and Lightspeed Views

CHAPTER 39

Creating a vertical view

A vertical view is used to visualize the third dimension of a list of points. Figure 129 shows the vertical view of a flight.

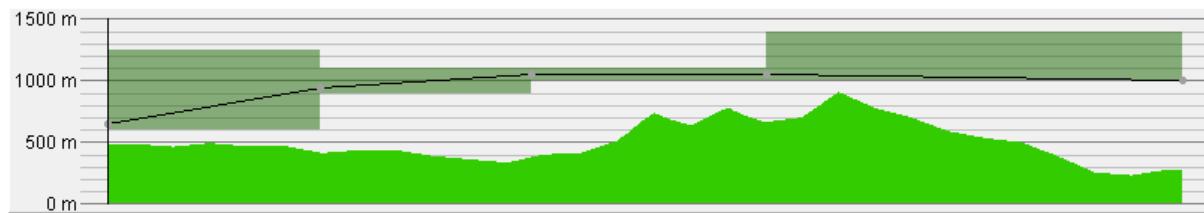


Figure 129 - An example of a vertical view

A vertical view typically shows a main profile and its associated sub-profiles. The flight profile in Figure 129 is the main profile. As sub-profiles, Figure 129 shows the altitude range between which an aircraft has to fly and the terrain profile. The line section drawn between two successive points of the main profile is called a segment.

A sub-profile can contain different points per main profile segment. For each of these sub-profile points, it is possible to retrieve the minimum and maximum altitude. A sub-profile is divided into sub-profile steps for each segment of the main profile. The number of steps of the sub-profile is equal to the number of sub-profile points in a segment minus one. In Figure 129, the altitude range sub-profile has one step for each segment. The terrain sub-profile step has multiple steps for each segment. A sub-profile step has a ratio, that is the percentage on the X-axis it represents, relative to the segment of the main profile.

The sections in this chapter describe how to create a model and add the model to the view.

39.1 Modeling the profile

To model the profile of a vertical view, LuciadLightspeed provides the interface `ILcdVVModel`. An `ILcdVVModel` holds the main profile points of the vertical view and the information about every sub-profile associated with the main profile.

You can add a `javax.swing.event.ChangeListener` to an `ILcdVVModel` to receive a notification when the internal state of the model has changed and recalculation is required. For more information on listening to changes, refer to [Section 8.2](#).

Implementations of `ILcdVVModel` provide methods to:

- Get the number of points of the main profile and get the point at a specified index
- Get the distance between two points on the main profile
- Indicate if editing of the main profile is allowed and change the Z dimension of a point on the main-profile by dragging it
- Get the number of sub-profiles
- Get the number of points of a sub-profile in a specified segment of the main profile
- Get the minimum or the maximum altitude for a specific sub-profile-point
- Get the step length ratio of a given sub-profile step
- Manage and notify change listeners

39.1.1 `ILcdVVModel` implementations

LuciadLightspeed offers an `ILcdVVModel` implementation, `TLcdVVTerrainModel`, that shows the profile of the polyline as the main profile, and the terrain profile as a sub-profile. The `lightspeed.vertical` sample shows how to use this `ILcdVVModel`.

39.2 Adding the profile to the view

To add a profile to the view, LuciadLightspeed provides the interface `TLcdVVJPanel`. A `TLcdVVJPanel` is a `javax.swing.JPanel` that allows to display one main profile and a number of sub-profiles that are associated with the main profile.

Associated with a `TLcdVVJPanel` are:

- An `ILcdVVModel`: the model that contains the main profile points and the information about the associated sub-profiles, described in [Section 39.1](#)
- An `ILcdVVRenderer`: a renderer for the vertical view that is responsible for the way the vertical view is drawn, described in [Section 39.3](#)
- An `ILcdVVGridRenderer`: a grid renderer to which the painting of the grid is delegated, described in [Section 39.3.3](#)
- An `ILcdVVGridLineOrdinateProvider`: a grid-line ordinate provider that determines the range of the vertical view after snapping and the visible grid-line and sub grid-line ordinates, described in [Section 39.3.4](#)
- An `ILcdVVXAxisRenderer`: a renderer for the X-axis that is responsible for decorating the X-axis, described in [Section 39.3.2](#)

39.2.1 Configuring a `TLcdVVJPanel`

The `TLcdVVJPanel` provides some methods to set the `ILcdVVModel` and the renderers. If the `ILcdVVModel` is set, the `TLcdVVJPanel` is automatically registered as `javax.swing.event.ChangeListener` to display changes immediately. Methods are also provided to indicate if the X-axis should be labeled, if the main points should be labeled, and if the vertical cursor should be visible. The different renderers describe the way in which the labels and vertical cursor are painted if they are visible.

Additionally, some methods are provided to change the part of the model that has to be visible in the vertical view:

- indicate a start point and an end point of the main profile to take into account
- define a ratio (between 0 and 1) of the main profile to hide at the left and the right side
- set the lowest and the highest altitude that should be painted when displaying values

39.2.2 Creating a `TLcdVVJPanel` with controllers

A `TLcdVVWithControllersJPanel` contains a `TLcdVVJPanel` and some controllers that allow users to change the values that define the visible part of the model in the vertical view. You can define in the constructor of the `TLcdVVWithControllersJPanel` which controllers to include.

39.2.3 Listening to cursor changes

The user can position a vertical cursor anywhere along the main profile. When the position of the cursor changes, all the registered `ILcdVVCursorChangeEvent` objects are notified. The `TLcdVVCursorChangeEvent` contains the point of the main profile on the left of the cursor position, on the right of the cursor position, and the percentage of the segment on the left side of the cursor.

39.3 Rendering the profile

An `ILcdVVRenderer` is responsible for rendering, or painting, the main profile and the sub-profiles. You can render a profile in (a combination of) the following modes:

- `ILcdVVRenderer.NO_PAINTING`: the profile is not painted
- `ILcdVVRenderer.TOP_LINE`: draws a line that is constituted by invoking `maxZ` of the points of a sub-profile
- `ILcdVVRenderer.BOTTOM_LINE`: draws a line that is constituted by invoking `minZ` of the points of a sub-profile
- `ILcdVVRenderer.LEFT_LINE`: draws a line that connects the leftmost point of the topline and the leftmost point of the bottomline
- `ILcdVVRenderer.RIGHT_LINE`: draws a line that connects the rightmost point of the topline and the rightmost point of the bottomline
- `ILcdVVRenderer.FILLED`: draws a topline and bottomline and fills the space between them
- `ILcdVVRenderer.POLYGON`: draws a topline, bottomline, leftline and rightline

For the main profile, only `NO_PAINTING` and `TOP_LINE` are valid options.

An `ILcdVVRenderer` provides some basic methods to paint the different parts of the main profile and the sub-profiles. The basic methods for painting the main profile are: `paintPointIcon`, `drawProfileLine`, `paintPointLabel`. The `paintYAxisParallelLine` method paints a vertical line from a given point on the main profile to the X-axis. The `paintVerticalCursor` method paints a vertical cursor. The basic methods for painting the sub-profiles are `drawSubProfileLine` and `fillSubProfileStepPolygon`. The sub-profile in [Figure 129](#) is rendered in `FILLED` mode. [Figure 130](#) shows the same example, but with the sub-profile rendered in `BOTTOM_LINE` mode.

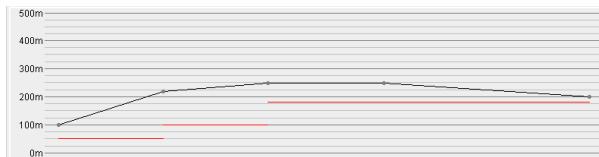


Figure 130 - An example of rendering mode BOTTOM_LINE

The default implementation of `ILcdVVRenderer` is `TLcdDefaultVVRenderer`. Another implementation is `TLcdDefaultVVRendererJ2D` that you can only use with Java 2D.

39.3.1 Configuring a `TLcdDefaultVVRenderer`

`TLcdDefaultVVRenderer` has several properties, which you can change by means of the corresponding set methods. [Program 207](#) shows the creation of a `TLcdDefaultVVRenderer` and its configuration using several set methods:

- `setMainProfileRenderingMode` to set the rendering mode for the main profile
- `setProfileColor` to set the color for the main profile
- `setSubProfileRenderingMode` to set the rendering mode for the sub-profiles
- `setSubProfileColor` to set the color for the sub-profiles

```

1 AltitudeVVRenderer viewRenderer = new AltitudeVVRenderer(verticalView);
2 // The rendering mode for the main profile is set to top_line: a line is
3 // drawn connecting the points.
4 viewRenderer.setMainProfileRenderingMode(ILcdVVRenderer.TOP_LINE);
5 // The rendering mode for the sub-profiles is set to filled: filled polygons
6 // are drawn at each subProfileStepIndex.
7 viewRenderer.setSubProfileRenderingMode(ILcdVVRenderer.FILLED);
8 // Color configuration.
9 viewRenderer.setProfilePaint(Color.black);
10 viewRenderer.setSubProfilePaintArray(aColors);
11 viewRenderer.setMainProfileLabelPaint(Color.darkGray);

```

Program 207 - Creation and configuration of an `TLcdDefaultVVRenderer`
(from samples/gxy/vertical/MainPanel)

39.3.2 Decorating the X-axis

An `ILcdVVXAxisRenderer` is responsible for decorating the X-axis. This interface only contains two methods: `getHeight` to get the number of pixels that the renderer needs for decorating the X-axis and `paintOnXAxis` to decorate the X-axis.

`ALcdVVXAxisRendererJ2D` is an abstract implementation of `ILcdVVXAxisRenderer`. It paints icons on the X-axis and paints labels underneath the icons. The labels can be rotated over a rotation angle. This class can only be used with Java 2D.

39.3.3 Painting the grid

All the painting that needs to be done for the grid is delegated to an `ILcdVVGridRenderer`. This interface provides methods for painting the basic grid lines, for painting sub grid lines, for rendering ordinate labels for the grid, and for retrieving the font size of the labels.

`TLcdDefaultVVGridRenderer` is a default implementation of `ILcdVVGridRenderer`. This class provides some set methods to customize the default behavior.

39.3.4 Using an `ILcdVVGridLineOrdinateProvider`

An `ILcdVVGridLineOrdinateProvider` is responsible for the snapped range and the grid-line ordinates of the vertical view. The interface provides methods for retrieving the range after snapping and for retrieving the grid-line and sub grid-line ordinates.

`TLcdDefaultVVGridLineOrdinateProvider` is a default implementation of `ILcdVVGridLineOrdinateProvider`.

39.4 A use case of the vertical view

This section describes a use case that shows how to implement a vertical view that displays the altitude of some flights. Each flight is a list of 3D points. A normal view only shows the first two coordinates(X,Y) of each of the points. The vertical view is used to show the vertical profile of the flight. In this case, one sub-profile is associated with the view that expresses the altitude-range between which the aircraft has to fly. For each segment of the flight, the minimum and maximum altitudes are shown. Thus, for each segment of the main profile, the sub-profile has only one step. Because there is just one sub-profile step for each segment, the ratio of each of the sub-profile steps is 1.

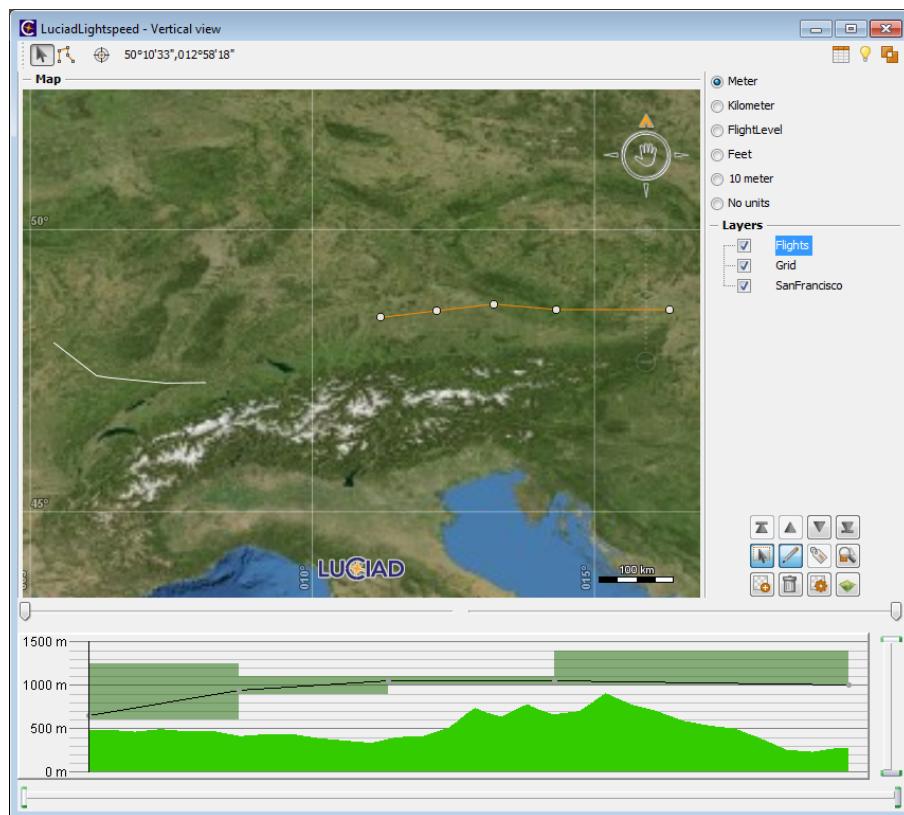


Figure 131 - The vertical view of the use case

Figure 131 shows three panels: the upper panel contains a 2D view with two flights. If the user selects one of the flights, the vertical view of the selected flight is shown in the bottom panel. The panel on the right enables a user to change the altitude unit on the vertical view.

A `TLcdVVWithControllers JPanel` is used to display the vertical view. This `javax.swing.JPanel` contains a `TLcdVV JPanel`. To configure a `TLcdVV JPanel`, you can

set an `ILcdVVModel`, an `ILcdVVRenderer`, an `ILcdVVGridRenderer` and an `ILcdVVX-AxisRenderer`.

`FlightVVModel` is an `ILcdVVModel` that contains all the information about flights and the altitude-ranges. `TLcdDefaultVVRenderer` is used as `ILcdVVRenderer` and some of its properties are configured using its set methods. Since no `ILcdVVGridRenderer` is set in the sample, the default implementation `TLcdDefaultVVGridRenderer` is used. This implementation can take the altitude unit of the vertical view into account to display labels.

Notice that changing the unit might have an influence on the upper and lower limits of the vertical view as it attempts to set round numbers as the limits (see Figure 132). The X-axis is not decorated, so no `ILcdVVXAxisRenderer` is set.

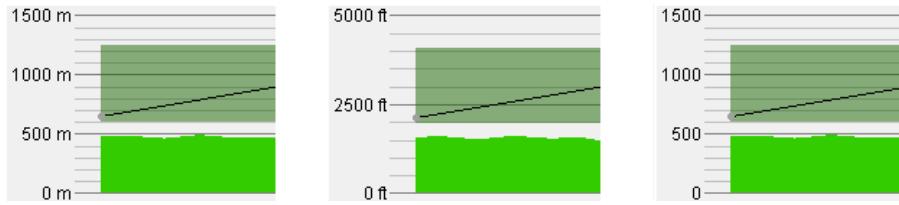


Figure 132 - Setting units on the vertical view (meter, feet, none)

39.4.1 Creating the flight model

`FlightVVModel` is an implementation of `ILcdVVModel`. It holds the main profile points of a flight and the minimum and maximum altitudes associated to the segments of the flight.

```

1 public class FlightVVModel extends ALcdVVModel {
2
3     private Flight fFlight;
4
5     public void setFlight(Flight aFlight) {
6         fFlight = aFlight;
7         super.fireChangeModel();
8     }
9
10    // points of the main profile
11    @Override
12    public ILcdPoint getPoint(int aIndex) throws IndexOutOfBoundsException {
13        if (fFlight != null) {
14            return fFlight.getPoint(aIndex);
15        } else {
16            return null;
17        }
18    }
19
20    // there is only one sub-profile: the air route associated to each segment
21    @Override
22    public int getSubProfileCount() {
23        if (fFlight != null) {
24            return 1;
25        } else {
26            return 0;
27        }
28    }
29
30    // for each segment, the sub-profile has just 2 points: the begin point and
31    // the end point of the segment
32    @Override
33    public int subProfilePointCount(int aSegmentIndex, int aSubProfileIndex)
34        throws IndexOutOfBoundsException {
35        if (fFlight != null) {
36            return 2;
37        } else {

```

```

38     return 0;
39 }
40 }
41
42 // the sub-profile contains one step for each segment so the ratio of the
43 // step length is always 1
44 @Override
45 public float stepLengthRatio(int aSubProfilePointIndex, int aSegmentIndex, int
46     aSubProfileIndex)
47     throws IndexOutOfBoundsException {
48     return 1;
49 }
50
51 @Override
52 public double minZ(int aSubProfilePointIndex, int aSegmentIndex, int aSubProfileIndex)
53     throws IndexOutOfBoundsException {
54     if (fFlight != null) {
55         return fFlight.getRouteMinAltitude(aSegmentIndex);
56     } else {
57         return 0;
58     }
59 }
60
61 @Override
62 public double maxZ(int aSubProfilePointIndex, int aSegmentIndex, int aSubProfileIndex)
63     throws IndexOutOfBoundsException {
64     if (fFlight != null) {
65         return fFlight.getRouteMaxAltitude(aSegmentIndex);
66     } else {
67         return 0;
68     }
69 // ...

```

Program 208 - An implementation of `ILcdVVModel` used for flights
 (from `samples/gxy/vertical/FlightVVModel`)

Program 208 shows a part of the implementation of `FlightVVModel`. The implementation of the methods of `ILcdVVModel` is straightforward.

Program 209 shows the creation and configuration of the `TLcdVVWithControllersJPanel`, the `javax.swing.JPanel` that contains the vertical view.

```
1  final TLcdVVJPanel verticalView = new TLcdVVJPanel();
2  verticalView.setVVModel(aVModel);
3  // Indicates that labels should be painted on the X-axis.
4  verticalView.setXAxisLabeled(true);
5  verticalView.setVVXAxisRenderer(new VVXAxisDistanceRenderer());
6  // Indicates whether or not all the point-icons should be labeled
7  verticalView.setPaintAllLabels(false);
8  // Determines how the grid is subdivided
9  verticalView.setVVGridLineOrdinatesProvider(new VVGridLineOrdinatesProvider());
10 AltitudeVVRenderer viewRenderer = new AltitudeVVRenderer(verticalView);
11 // The rendering mode for the main profile is set to top_line: a line is
12 // drawn connecting the points.
13 viewRenderer.setMainProfileRenderingMode(ILcdVVRenderer.TOP_LINE);
14 // The rendering mode for the sub-profiles is set to filled: filled polygons
15 // are drawn at each subProfileStepIndex.
16 viewRenderer.setSubProfileRenderingMode(ILcdVVRenderer.FILLED);
17 // Color configuration.
18 viewRenderer.setProfilePaint(Color.black);
19 viewRenderer.setSubProfilePaintArray(aColors);
20 viewRenderer.setMainProfileLabelPaint(Color.darkGray);
21 // Constructs a TLcdVVWithControllersJPanel that contains the
22 // TLcdVVJPanel and some controllers used to change the left and right
23 // offset, the altitude range and the start and end index.
24 TLcdVVWithControllersJPanel vvWithControllers =
25     new TLcdVVWithControllersJPanel(aVerticalView);
26 vvWithControllers.setMinimumSize(new Dimension(0, 200));
27 vvWithControllers.setPreferredSize(new Dimension(0, 200));
```

Program 209 - Construction and configuration of the TLcdVVWithControllersJPanel
(from samples/gxy/vertical/MainPanel)

CHAPTER 40

Printing a view

LuciadLightspeed allows you to print out the maps displayed in a LuciadLightspeed view, by creating printable objects. In addition, you can display a preview of the layout of your print before you start the printout.

40.1 Creating and configuring a Printable

LuciadLightspeed views can be printed using the `java.awt.print` API. The class `ALcdViewComponentPrintable` is an implementation of `java.awt.print.Printable` which prints the contents of a given view.

`ALcdViewComponentPrintable` defines the following printing properties:

- DPI: the desired resolution of the print in dots per inch.
- Feature scale: the scaling of features in the print relative to their size on the screen. A value of 1 means that features such as icons, labels or line widths are printed at the same proportional size as they have on the screen. Values smaller than 1 mean that features become proportionally smaller. For instance, lines appear thinner, and labels become smaller than they are on screen. If label decluttering is used, this also implies that the print potentially displays more labels than the screen. You can also proportionally increase the size of the features by choosing a value between 1 and 2. Feature scale may also affect level-of-detail decisions, for example for raster layers. Lower feature scales may cause higher detail levels to be shown, and the other way round.
- Horizontal and vertical page count: the number of pages across which the print should be spread out, for multi-page prints.
- Map scale: the actual map scale that the print will have. Take, for example, a map scale of 1/10000, or 1:10000. This means that 1 meter in world coordinates corresponds to 1/10000m = 0.1mm on paper. The map scale can be derived from the number of pages and the print DPI, or you can set it explicitly. If you do, the page count is derived from the scale, instead of the other way round.

Two implementations of `ALcdViewComponentPrintable` are available:

- `TLcdGXYViewComponentPrintable` works with an `ILcdGXYView`.
- `TLspViewComponentPrintable` works with an `ILspView`.

These classes allow you to set all the printing properties, as well as some additional settings such as the inclusion of crop marks on multi-page prints. Please see the reference documentation for

more details.

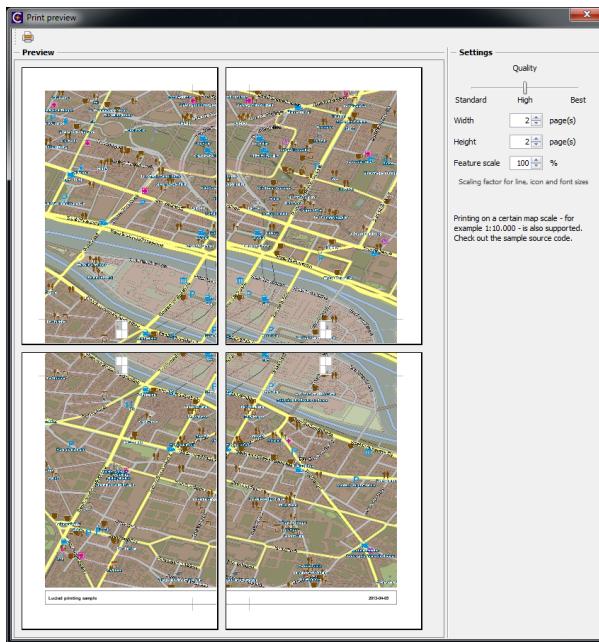


Figure 133 - An example of a printed component containing a footer and a map view

Examples of how these classes are used can be found in `samples.gxy.printing` and `samples.lightspeed.printing`, respectively. These samples also demonstrate how the printables can be used in conjunction with `TLcdPrintPreview` to display a multi-page print preview dialog. Finally, the samples also illustrate how to add decorations, such as headers and footers or a legend, to the page surrounding the map.

40.2 Supporting printing for custom layers and painters in an `ILspView`

All the standard layer types available in Lightspeed views support printing out of the box. When you are implementing your own `ILspPaintableLayer` or `ILspPainter`, however, it may be necessary to take some additional steps to make your implementation printing-capable. This section explains these extra steps.

40.2.1 Tiled rendering

Lightspeed views use a tiled rendering approach to generate the high-resolution images needed for a print. This is normally transparent to the layers and painters in the view, as the tiling is performed through the manipulation of the OpenGL projection matrix. If your custom painting code modifies the projection matrix, however, it may nullify the transformations needed for the tiled rendering.

In such cases, the painter can obtain the bounds of the current tile via `ALspViewXYZWorldTransformation.getClip()`. This method returns a rectangle in view coordinates which bounds the area of the view that is currently being printed. The painter must apply the necessary scaling and translation to make this portion of the view fill the entire viewport.

A typical use case is drawing all manner of overlays on top of the view in screen coordinates. The following code fragment shows how you could set up an orthographic projection matrix

for your layer to achieve this. If you do so, however, the layer will no longer print correctly. Therefore, you need to perform additional transformations if the printing clip is not null.

```

1 int width = aView.getWidth();
2 int height = aView.getHeight();
3 double rasterizationScale = 1.0;
4
5 // Switch to an orthographic projection. This breaks printing.
6 gl.glMatrixMode( ILcdGL.GL_PROJECTION );
7 gl.glPushMatrix();
8 gl.glLoadIdentity();
9 gl.gOrtho( 0, width, height, 0, -1, 1 );
10
11 // Apply the print clip if it is set to make the layer print correctly.
12 ALspViewXYZWorldTransformation w2v = aView.getViewXYZWorldTransformation();
13 Rectangle2D clip = w2v.getClip();
14 if ( clip != null ) {
15     // Scale so the clip fills the view
16     rasterizationScale = aView.getWidth() / clip.getWidth();
17     gl.gScaled( rasterizationScale, rasterizationScale, 1.0 );
18     // Translate so the clip is aligned to the view
19     gl.gTranslated( -clip.getX(), -clip.getY(), 0 );
20 }
21
22 gl.glMatrixMode( ILcdGL.GL_MODELVIEW );
23 gl.glPushMatrix();
24 gl.glLoadIdentity();
25
26 // From this point on, draw things in view coordinates.

```

Program 210 - Applying the print clip to custom OpenGL rendering code

40.2.2 Feature scale

Custom painters may also want to take the feature scale property, as discussed in Section 40.1, into account. The current feature scale is available via `ALspViewXYZWorldTransformation.getFeatureScale()`. The feature scale can be applied to line widths, font sizes, texture resolutions, level-of-detail decisions, and so on, as you see fit. It is important to note, however, that the feature scale is relative to the print resolution. If the print resolution is 10 times that of the view, and the feature scale is 0.5, a painter should apply a combined scale factor of 5 to get the expected result.

The print resolution can be derived from the clip mentioned in the previous section: divide the width or height of the clip by that of the view to obtain the print rasterization scale factor.

The following code, which is continued from the example in Section 40.2.1, shows how a line thickness is scaled for printing:

```

1 int lineWidth = 2;
2 gl.gLineWidth( ( float ) (lineWidth * rasterizationScale * w2v.getFeatureScale()) );

```

Program 211 - Scaling line width using print resolution and feature scale

PART VI Combining GXY and Lightspeed Technology

CHAPTER 4I

Using Lightspeed technology in your GXY view application

If you are already running an application with a GXY view, LuciadLightspeed offers the possibility to selectively adapt an application based on your current view implementation. Lightspeed views are able to show their contents in both 2D and 3D.

Depending on your needs, you can choose to:

- Move specific GXY layers to Lightspeed technology but keep your GXY view implementation. See [Section 4I.1](#) for more information.
- Gradually make the switch from a GXY view to a Lightspeed view implementation, by moving the view component and performance-critical layers first, and keeping other layers until later. See [Section 4I.2](#) for more information.

Regardless of the choice you make, LuciadLightspeed's Model-View-Controller paradigm ensures that you do not have to change anything on the model side.

4I.1 Using Lightspeed technology within an `ILcdGXYView`

LuciadLightspeed provides an easy transition path to convert selected parts of your application to use Lightspeed technology, while keeping your existing `ILcdGXYView` and all code that interacts with it. This allows you to speed up rendering in situations where you are painting many objects, or displaying complex visualizations such as hypsometry data and density plots.

GXY views already offered a facility to paint heavy data sets asynchronously, on a separate thread. This mechanism has been extended with a special layer wrapper implementation that delegates rendering to a Lightspeed layer. This allows for perfect GXY view compatibility, combined with Lightspeed view rendering.

This chapter requires prior knowledge of asynchronous layer wrappers. Please refer to [Chapter 33](#) for more information about asynchronous painting.

4I.1.1 When would you use Lightspeed view technology?

Even though the Lightspeed view technology accelerates many rendering operations, there are some typical use cases where Lightspeed technology can noticeably speed up rendering in an `ILcdGXYView`. Lightspeed view technology can accelerate the display of:

- Complex icons

- Anti-aliased vector shapes
- Density plots
- Hypsometry visualizations



For large resolutions (beyond Full HD), it is recommended to use a native Lightspeed view to reduce the 2D compositing overhead that is typically required for `ILcdGXYView` implementations.

41.1.2 Including a Lightspeed layer in a `ILcdGXYView`

To include a Lightspeed layer in a `ILcdGXYView`, wrap the Lightspeed layer in an asynchronous layer wrapper. [Program 212](#) shows how to include a Lightspeed layer in an `ILcdGXYView`.

```

1  ILcdGXYLayer dmedGXYLayer = GXYDataUtil.instance().model(SampleData.ALPS_ELEVATION).layer(
2      rasterLayerFactory).getLayer();
3
4  ILspLayer dmedLspLayer = LspDataUtil.instance().model(dmedGXYLayer.getModel()).layer().
5      getLayer();
6
7  TLcdGXYLspAsynchronousLayerWrapper asynchronousDmedLayer =
8      new TLcdGXYLspAsynchronousLayerWrapper(dmedGXYLayer, dmedLspLayer);
9
10 GXYLayerUtil.addView( getView(), asynchronousDmedLayer );

```

Program 212 - Using an asynchronous layer wrapper to include a Lightspeed layer in an `ILcdGXYView`.

(from samples/gxy/concurrent/painting/lightspeed/MainPanel)

The `TLcdGXYLspAsynchronousLayerWrapper`'s constructor takes both an `ILspLayer`, as well as an `ILcdGXYLayer`. Just as any other `ILcdGXYAsynchronousLayerWrapper`, the Lightspeed layer wrapper uses a paint queue to delegate its paint requests to. In order to make use of the Lightspeed layer, a special Lightspeed paint queue must be used: `TLcdGXYLspAsynchronousPaintQueue`.

This paint queue delegates rendering as follows:

- Regular paint representations (the `ILcdGXYLayer.ALL` + `ILcdGXYLayer.BODIES` paint mode) use the Lightspeed layer.
- Edited and selected objects (the `ILcdGXYLayer.SELECTION` paint modes) use the `ILcdGXYLayer`.
- Labels (the `ILcdGXYLayer.LABELS` and `ILcdGXYEditableLabelLayer.PLACED_LABELS` paint modes) use the `ILcdGXYLayer`.

Hence, the Lightspeed layer should at least have a painter configured for painting the regular representations (`TLspPaintRepresentationState.REGULAR_BODY`). The `ILcdGXYLayer` should at least have a painter provider configured. If needed, you can prevent selection by overriding the `ILcdGXYLayer`'s `isSelectableSupported` method. It is good practice to let the layer wrapper create the paint queue as is done in the snippet.

Just as with regular asynchronous layer wrappers, you can use a paint queue manager with the Lightspeed layer wrapper. The paint queue manager automatically creates and assigns the correct paint queue to each layer wrapper. [Program 213](#) shows how to set up a paint queue manager capable of dealing with both `TLcdGXYLspAsynchronousLayerWrapper` instances, as well as other `ILcdGXYAsynchronousLayerWrapper` implementations.

```
1 TLcdGXYLspAsynchronousPaintQueueManager manager = new  
2   TLcdGXYLspAsynchronousPaintQueueManager();  
manager.setGXYView(aMap);
```

Program 213 - Creating a paint queue manager for a view
(from samples/gxy/concurrent/painting/lightspeed/MainPanel)

`TLcdGXYLspAsynchronousPaintQueueManager` extends from `TLcdGXYAsynchronousPaintQueueManager` and hence also takes its decisions based on paint hints. By default the manager uses a preconfigured paint hint provider. Please refer to [Section 33.5](#) for more information on customizing paint queue assignments.

41.1.3 Safely accessing asynchronously painted Lightspeed layers

The Lightspeed layer is rendered in a separate background thread. Hence, unless it is explicitly mentioned that this layer is thread-safe, precautions must be taken to safely access it. In general, the rules for accessing the wrapped `ILspLayer` are the same as for accessing a wrapped `ILcdGXYLayer`. In particular, it is recommended to access the Lightspeed layer only through one of the invocation methods in the `TLcdGXYLspAsynchronousLayerWrapper`: `invokeAndWaitOnGXYAndLspLayer`, `invokeLaterOnGXYAndLspLayer`, `invokeLaterOnGXYAndLspLayerInEDT` and `invokeNowOnGXYAndLspLayer`. See [Chapter 33](#) on asynchronous painting for more information on safely accessing the wrapped layer.

41.1.4 Deploying on a mix of hardware configurations

The hardware requirements for Lightspeed view technology are higher than those for GXY views, because Lightspeed technology takes advantage of modern GPU hardware. Nevertheless, even when your application is deployed on a mixed range of hardware configurations, which do not all support Lightspeed technology, you can still take advantage of Lightspeed layers. `TLcdGXYLspAsynchronousPaintQueue` checks the OpenGL requirements of its Lightspeed layers. If the OpenGL requirements for a Lightspeed layer do not match those of the target platform, LuciadLightspeed will use the wrapped `ILcdGXYLayer` for all visualization, providing a convenient fallback. A warning message will be logged if this is the case.

For more information on hardware and software requirements, see the [LuciadLightspeed technical notes](#).

41.1.5 Performance tips for wrapped Lightspeed layers

Just as with regular asynchronous layer wrappers, you can reduce the compositing overhead for multiple layers if you make them share the same paint queue. `TLcdGXYLspAsynchronousPaintQueueManager`, is configured to do exactly this, provided that the Lightspeed layer wrappers are adjacent. They have to be directly on top of one another without any non-adapter layer inserted between the two.

If you suspect that the Lightspeed layer is not being used at all, make sure that you are using the appropriate paint queue or paint queue manager, either `TLcdGXYLspAsynchronousPaintQueue` or `TLcdGXYLspAsynchronousPaintQueueManager`). Other implementations may ignore the Lightspeed layer.

41.2 Moving from a GXY view to an ILspView

LuciadLightspeed provides an easy transition path to convert your application to use a Lightspeed view. It allows for a quick upgrade of standard GXY layer implementations, which speeds up rendering when painting many objects or displaying complex visualizations such as hypsometry data and density plots. At the same time, you can keep certain existing GXY layer implementations and the code interacting with it. Refer to [Section 2.3.1](#) for more information on the benefits of using Lightspeed rendering.

[Section 41.2.1](#) and [Section 41.2.2](#) highlight the most important similarities and differences you must keep in mind when moving your view and your layers. [Section 41.2.3](#) discusses how to adapt your custom GXY layer implementations to a Lightspeed view.

41.2.1 Converting a GXY view to a Lightspeed view

In many aspects, a Lightspeed view is similar to a GXY view: it also has a specific world reference and separates its content into layers that are added, removed, moved and retrieved, just as in an GXY view. `TLspViewBuilder` provides a convenient way to set up a Lightspeed view, as demonstrated in [Section 3.1.1](#).

The default Lightspeed view (`TLspAWTView`), unlike its GXY counterpart (for example, `TLcdMapJPanel`), is in itself not a Java Component. This means that existing `instanceof` checks and cast operations in your code may fail. Instead, you can retrieve the view's host component by calling the `getHostComponent` method. As an added convenience, the view also exposes an overlay panel to insert Java Components on top of the view (see the `getOverlayComponent` method).

A Lightspeed view comes by default with a controller for panning, zooming, selecting and editing, but you can create your own controller or chain of controllers and assign it using the `ILspView`'s `setController` method. This is demonstrated in the fundamentals part in [Program 37](#). In addition, `TLspViewNavigationUtil` offers functionality to fit and center on layers.

Setting the number of cached background layers is no longer necessary in a Lightspeed view: a Lightspeed layer can indicate whether it is a background layer or not. See the `getLayerType` method in `ILspLayer` for more information.

An `ILspView` picks up elevation data automatically if the data is visualized in a `TLspRasterLayer` and one of the following conditions is fulfilled:

- The data is part of an `ILcdEarthTileSet` with a `CoverageType.ELEVATION` type.
- The data is part of an `ILcd(Multilevel)Raster`. The model descriptor is a `TLcd(Multilevel)RasterModelDescriptor`, and its `isElevation()` method returns true.

The combined elevation data of a view can be accessed from `ILspView.getServices().getTerrainSupport()`. If layers with elevation data overlap, the elevation of the top-most layer will be selected.

41.2.2 Converting a GXY layer to a Lightspeed layer

The Lightspeed equivalent of a `ILcdGXYLayer` is an `ILspInteractivePaintableLayer`. `ILspInteractivePaintableLayer` paints and styles the objects of its model using stylers,

and optionally uses one or more `ILspEditor` instances to edit these objects. The default `ILspInteractivePaintableLayer` implementation is `TLspLayer`.

Raster and vector layers

A raster layer is particularly easy to convert. A special layer builder, `TLspRasterLayerBuilder`, allows you to visualize any model with an `ILcdRaster`, `ILcdMultilevelRaster`, or `ILcdEarthTileSet`, without any configuration. Simply create an `ILspLayer` using the builder, add it to your view, and you are ready to go.

Similarly, you can use `TLspShapeLayerBuilder` for vector layers. This builder creates layers that can paint all LuciadLightspeed shapes. Hence, there is no need for a specific circle, point list, or icon painter.

Both approaches are demonstrated in the fundamentals samples, in [Program 5](#).

Note the use of a `TLspPaintRepresentation` object. This is discussed in more detail in the [Paint representations](#) section.

Paint representations

LuciadLightspeed allows you to visualize many different representations of the same object. All these visualizations use the same painter interface: `ILspPainter`. As opposed to the requirements of a GXY layer, there is no need for a special label painter interface in a Lightspeed layer. Instead, you can configure different painters for different paint representations, including labels.

[Program 24](#) shows how to configure a label painter on a Lightspeed layer.

Styling

LuciadLightspeed has an elaborate API for styling, from line types and fill colors to icons and label content. The styling API can be used to replace most, if not all, of your custom GXY layer implementations.

A major difference with GXY view styling is that Lightspeed view styling can be configured per object. To achieve this, Lightspeed painters map styles onto objects using an `ILspStyler`. A `TLspStyler` returns the same style(s) for every object, and is particularly useful when you are replacing GXY painters. This is illustrated in [Program 5](#), in the fundamentals Chapter.

While styling is usually performed by the painter, most of the Lightspeed layer implementations implement `ILspStyledLayer`, allowing you to directly configure styling by calling `setStyler`.

To see other examples of layer and painter styles, have a look at the many Lightspeed samples on labels, icons, and styling. Note that no explicit layer invalidation is necessary when you change a styler.

41.2.3 Adapting GXY layers

For some custom layer implementations; such as layers with a custom painter or custom styling, you might prefer to keep the entire layer implementation. In that case, you can add the GXY layer to a Lightspeed view by wrapping it into an adapter. For performance reasons, the adapter will paint the original layer implementation in another thread.

This process is similar to using an `ILcdGXYAsynchronousLayerWrapper` (see [Chapter 33](#)). Converting such layers is often only a matter of replacing your `ILcdGXYAsynchronousLayerWrapper` with a Lightspeed `TLspGXYLayerAdapter` and adjusting some calls.

[Program 214](#) shows how to adapt an `ILcdGXYLayer` to visualize it in an `ILspView`.

```
1 TLspGXYLayerAdapter adapter = new TLspGXYLayerAdapter((ILcdGXYLayer) aLayer);
```

**Program 214 - Using an adapter to include an ILcdGXYLayer in an ILspView
(from samples/lightspeed/integration/gxy/AdaptedLayerFactory)**

As an added bonus, the Lightspeed layer adapter will work in both 2D and 3D Lightspeed views.

Using a layer adapter in 2D

The 2D rendering of layer adapters is similar to the rendering of asynchronously painted layers in an ILcdGXYView. The active Lightspeed controller deals with layer interaction, such as selection and editing.

The following restrictions apply in 2D views:

- No support for interactive labels. These are labels that respond to mouse and keyboard input to let users modify object properties.
- No support for view-wide label placement: label placers only take the objects within a layer into account, and can no longer handle all the objects visible in the view.

Using a layer adapter in 3D

As opposed to a GXY view, a Lightspeed view can display 3D content. A TLspGXYLayer-Adapter is capable of displaying the GXY layers in 3D, with the following restrictions:

- All 2D restrictions listed in [Using a layer adapter in 2D](#)
- No interaction
- The two-dimensional content is rendered onto small image tiles which are combined together and draped on the world. Because of this, view-dependent content, such as labels, may be cut off at the tile boundaries.

Safely accessing adapted layers

Like all Swing and AWT components, the Lightspeed view implementations and its layers are accessed and manipulated on the Event Dispatch Thread (EDT), with the exception of TLspOffscreenView.

To ensure performance, a TLspGXYLayerAdapter paints its GXY layer in a separate thread. However, a GXY layer in itself is not thread-safe. Therefore, you must take special care before accessing or modifying an adapter's wrapped layer. A safe way to retrieve and modify the wrapped layer is to ask the layer adapter to execute the modification code at a safe point in time, on an appropriate thread. The adapter provides the methods `invokeAndWaitOnGXYLayer` and `invokeLaterOnGXYLayer` for this purpose:

- The method `invokeAndWaitOnGXYLayer` executes a given runnable on the caller's thread (usually the EDT) when the adapter is not painting. The method receives the wrapped layer as an argument and it can safely work on it. Use this method when you need the operation to complete before carrying on with the rest of your code.
- The method `invokeLaterOnGXYLayer` executes a given runnable on the asynchronous painting thread. The method receives the wrapped layer as an argument. The method must not invoke any methods on the event dispatch thread with `SwingUtilities.invokeAndWait`, because this might result in dead-locks. Use this method when you do not want to wait for the operation to complete, for example to prevent the blockage of the Event Dispatch Thread.

For more information on thread safety for asynchronously painted layers, refer to [Chapter 33](#) and [Chapter 56](#).

Editing the model of an adapted wrapped layer

You need to edit the model of a wrapped layer in a thread-safe way, because the model is being painted in a separate painting thread. The layer adapter guards asynchronous read access by means of read locks on the model, as described in [Section 33.2](#). Any code that modifies the model or its elements therefore has to make sure that it obtains a write lock on the model, as shown in [Chapter 9](#).

Labeling adapted layers

GXY labeling is distinct from Lightspeed labeling. However, the layer adapter is capable of applying a GXY labeling algorithm to its wrapped layer. To configure this algorithm, call `setGXYLayerLabelingAlgorithm`.

As an alternative if you are using a `TLcdGXYLayer`, you can set up a layer label painter on this layer by calling `setGXYLayerLabelPainter`.

In both cases, label decluttering will only happen for the GXY layer itself. The decluttering process will not take into account other GXY or Lightspeed layers.

Adapting layer tree nodes

`TLspGXYLayerTreeNodeAdapter` is an `ILcdGXYLayer` adapter that can have Lightspeed child layers. Therefore, to adapt an entire `ILcdLayerTreeNode`, all its children need to be adapted as well. This is illustrated in [Program 215](#).

```

1  private static TLspGXYLayerAdapter adapt(ILcdLayer aLayer) {
2      if (aLayer instanceof ILcdLayerTreeNode) {
3          return adaptNode((ILcdLayerTreeNode) aLayer);
4      } else if (aLayer instanceof ILcdGXYLayer) {
5          @SuppressWarnings("UnnecessaryLocalVariable")
6          TLspGXYLayerAdapter adapter = new TLspGXYLayerAdapter((ILcdGXYLayer) aLayer);
7          return adapter;
8      } else {
9          throw new IllegalArgumentException("Cannot handle : " + aLayer.getClass());
10     }
11 }
12
13 private static TLspGXYLayerTreeNodeAdapter adaptNode(ILcdLayerTreeNode aLayerTree) {
14     // adapt the node
15     TLspGXYLayerTreeNodeAdapter adapter = new TLspGXYLayerTreeNodeAdapter((ILcdGXYLayer)
16         aLayerTree);
17     adapter.setPaintOnTopOfChildrenHint(aLayerTree.isPaintOnTopOfChildrenHint());
18     adapter.setInitialLayerIndexProvider(aLayerTree.getInitialLayerIndexProvider());
19     for (int i = 0, n = aLayerTree.layerCount(); i != n; i++) {
20         ILcdLayer layer = aLayerTree.getLayer(i);
21         // adapt the child layers as well
22         adapter.addLayer(adapt(layer));
23     }
24 }
```

Program 215 - Using an adapter to include an `ILcdLayerTreeNode` in an `ILspView`
(from `samples/lightspeed/integration/gxy/AdaptedLayerFactory`)

Layer adapter performance tips

Just as with asynchronous layer wrappers in `ILcdGXYView` instances, you can reduce the compositing overhead for multiple adapted layers if you make them adjacent. This means that, for the best performance, all adapters should be stacked on top of one another, and no regular (non-adapter) layer should be inserted in between the adapters.

PART VII Referencing Geographic Data

CHAPTER 42

Geodesy

Geodesy is the scientific discipline that deals with the measurement and representation of the earth. Many mathematical models have been proposed to approximate the shape of the earth, ranging from flat-earth models to spherical approximations, ellipsoids, and [geoids](#). Spherical and ellipsoidal approximations are used because of their mathematical simplicity. A more accurate approximation is the earth's geoid, a hypothetical surface that coincides with the earth's mean sea level. Although the earth's geoid is considerably smoother than the actual surface of the earth, it is still highly irregular and therefore more difficult to represent and to use in computations. Note that there are many ellipsoid models (for example WGS84 and NAD83) and geoid models (for example EGM2008 and NAVD88) in use around the world.

The following sections describe what geodetic datums and geoids are and how they are supported in LuciadLightspeed.

42.1 What is a geodetic datum?

A geodetic datum is a reference from which position measurements are made. A **horizontal datum** is a known and constant surface on which the positions of points can be precisely expressed. Because of their relative simplicity, ellipsoids are often used as the basis for horizontal datums. A **vertical datum** is an additional vertical reference for expressing the elevation of points. It is typically based on geoid or ellipsoid models.

Geodetic datums provide a basis for coordinate reference systems as described in [Section 44.2](#). For example, a geodetic datum based on an ellipsoid model of the earth's surface allows to define geodetic lon-lat-height coordinates. The height coordinate may for example be computed with respect to the ellipsoid on which the lon-lat coordinates are defined (ellipsoidal height), or with respect to a geoid model (orthometric height), as shown in [Figure 134](#). Geodetic datums and the coordinate system based on them are widely used in surveying, mapping, and navigation. Refer to [Chapter 43](#) for more information on map projections.

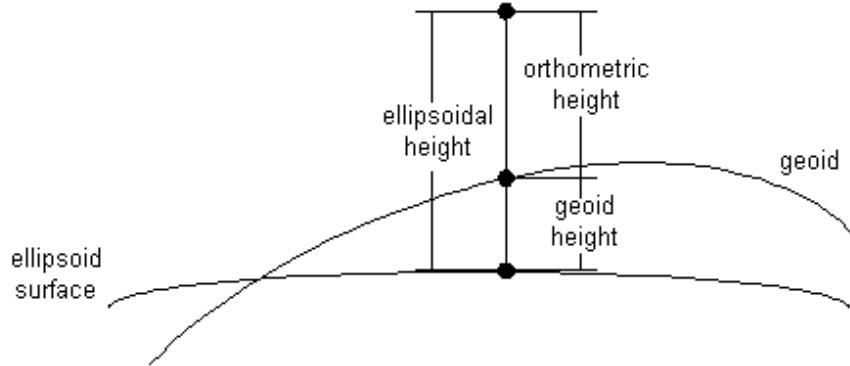


Figure 134 - Ellipsoidal versus orthometric heights

There are many datums in use today, as the basis for even more coordinate reference systems. Because referencing geodetic coordinates to the wrong datum can result in position errors of hundreds of meters, you need to be careful when conversing between coordinates defined with respect to different datums. You can find more information on coordinate transformations in [Section 44.4](#).

[Section 42.2](#) describes how geodetic datums are supported in LuciadLightspeed. The API reference provides all the necessary details in the package `com.luciad.geodesy`. There is a vast literature on geodesy for the interested reader. [Section 42.4](#) provides a few references.

42.2 Using an `ILcdGeodeticDatum`

LuciadLightspeed provides the interface `ILcdGeodeticDatum` to represent geodetic datums and support geodetic datum conversions. In this representation, a geodetic datum encompasses both a horizontal datum and a vertical datum. The horizontal datum is based on an ellipsoid of the type `ILcdEllipsoid` as described in [Section 42.3](#). A vertical datum is represented as a height function above this reference ellipsoid (see method `getHeight`). Most instances, like WGS84, reference height values to their ellipsoid (their method `getHeight` always returns zero).

`TLcdGeodeticDatum` is an implementation of `ILcdGeodeticDatum` which is based on an eight-parameter transformation to a global reference datum. The transformation defines a rotation, a translation, a scaling and a shift of the prime meridian. This transformation is applied to convert (lon, lat, height) coordinates from one geodetic datum to the reference datum.

```

1   fTransformation.setSourceReference(fReference);
2   fTransformation.setDestinationReference(fChosenGeodeticReference);
3   fTransformation.destinationPoint2sourceSFCT(fLonLatCoordinate,
4                                             fReferenceLonLatHeightPoint);
```

Program 216 - Transforming coordinates to reference geodetic datum
(from samples/gxy/transformation/geodeticToGrid/GridCalculation)

Program 216 shows how to transform lon-lat-height coordinates from one `ILcdGeodeticDatum` (called `fChosenGeodeticDatum`) to the reference `ILcdGeodeticDatum`. The second argument of the method `datum2refLLHSFCT` is of the type `ILcd3DEditablePoint`, of

which the coordinates are set to the result of the method¹.

```

1  fTransformation.setDestinationReference(fChosenGeodeticReference);
2  fTransformation.sourcePoint2destinationSFCT(fReferenceLonLatHeightPoint,
3                                              fLonLatCoordinate);
```

Program 217 - Transforming coordinates from reference geodetic datum
 (from samples/gxy/transformation/geodeticToGrid/GridCalculation)

Program 217 shows the inverse transformation: from the reference ILcdGeodeticDatum back to the coordinates of the fChosenGeodeticDatum.

By means of these two transformations, it is possible to transform coordinates from any ILcdGeodeticDatum to any other ILcdGeodeticDatum using the reference datum. The reference datum is an internally chosen ILcdGeodeticDatum.

LuciadLightspeed has implementations of most geodetic datums used in the world. These implementations are grouped together in an ILcdGeodeticDatumFactory. LuciadLightspeed has five implementations of ILcdGeodeticDatumFactory:

- TLcdNIMA8350GeodeticDatumFactory which creates geodetic datums that were defined by NIMA
- TLcdJPLGeodeticDatumFactory which creates geodetic datums that were defined by JPL
- TLcdEPSGGeodeticDatumFactory which creates geodetic datums based on the EPSG database
- TLcdGenericGeodeticDatumFactory a generic geodetic datum factory
- TLcdGeoidGeodeticDatumFactory which creates some common geodetic datums with geoid models as vertical datums. This is currently the only factory creating geodetic datums that do not have ellipsoids as vertical datums. See the API reference documentation for TLcdGeoidGeodeticDatumFactory to see which geodetic datums with a geoid model as vertical datum are supported.

All factories use WGS-84 as the reference datum.

```

1  int id = fGeodeticDatumFactory.getAliasNames().findID(datum_name);
2  ILcdGeodeticDatum chosen_datum = fGeodeticDatumFactory.createGeodeticDatum(id);
3  fChosenGeodeticReference = new TLcdGeodeticReference(chosen_datum);
```

Program 218 - Create a geodetic datum from a name
 (from samples/gxy/transformation/geodeticToGrid/GridCalculation)

Program 218 shows how the particular instance fGeodeticDatumFactory of the class TLcdNIMA8350GeodeticDatumFactory returns a list of alias names of the class TLcdAliasNames. By means of the method findID on this TLcdAliasNames, the id of the ILcdGeodeticDatum with name datum_name is retrieved. Using this id, an ILcdGeodeticDatum called fChosenGeodeticDatum is created by fGeodeticDatumFactory.

In general, the creation of an ILcdGeodeticDatum using an ILcdGeodeticDatumFactory is done by giving the ILcdGeodeticDatumFactory a java.util.Properties object. The general mechanism of properties files in LuciadLightspeed is described in Chapter 58. For the

¹In LuciadLightspeed this is called a side-effect method. Side-effect methods are used for performance reasons. In LuciadLightspeed, the names of side-effect methods end in SFCT.

specific `ILcdGeodeticDatumFactory` properties, see `ILcdGeodeticDatumFactory` and its implementations in the API reference.

42.3 Using an ILcdEllipsoid

An `ILcdGeodeticDatum` is based on an `ILcdEllipsoid`. The interface `ILcdEllipsoid` provides geometrical calculations on an ellipsoid. LuciadLightspeed has a single implementation of the interface `ILcdEllipsoid`: `TLcdEllipsoid`.

LuciadLightspeed also provides a utility class `TLcdSphereUtil`. This class implements the methods of the interface `ILcdEllipsoid` as static methods. In this implementation the geometrical calculations are done on a sphere. Spherical calculations approximate the calculations on the ellipsoid, but are more efficient.

The main methods of the interface `ILcdEllipsoid` are:

- `geodesicDistance` to return the shortest distance between two `ILcdPoint` objects
- `forwardAzimuth2D` to return the azimuth direction of the shortest distance path between two `ILcdPoint` objects
- `geodesicPointSFCT` to return the geodetic coordinate of an `ILcdPoint` at a given distance in a given azimuth direction

LuciadLightspeed groups together instances of `ILcdEllipsoid` in an interface `ILcdEllipsoidFactory`. It has two factories that define most ellipsoids used in the world:

- `TLcdDMA1987bEllipsoidFactory`, which creates ellipsoids defined by DMA
- `TLcdJPLEllipsoidFactory`, which creates ellipsoids defined by JPL

42.4 Literature on geodesy

- Burkehard, *Geodesy for the layman*, 1985 (http://www.ngs.noaa.gov/PUBS_LIB/Geodesy4Layman/toc.htm)
- Ewing, Mitchell, *Introduction to Geodesy*, 1970
- Bomford, *Geodesy*, 1952

CHAPTER 43

Projections

This chapter describes the LuciadLightspeed support for map projections. It does not describe the theory of map projections. For a theoretical background refer to the literature as given in Section 43.4.

Section 43.1 describes the interface `ILcdProjection`, which forms the basis of this chapter. Section 43.2 describes the main extensions of this interface, and Section 43.3 describes the main implementations of these extensions.

43.1 Using an `ILcdProjection`

An `ILcdProjection` is a transformation of geographically referenced data into a planar (flat) representation of the earth's surface and conversely. In LuciadLightspeed a planar coordinate representation is denoted as **world coordinates**. The geographically referenced data are expressed as geodetic lon-lat-height coordinates with respect to an `ILcdGeodeticDatum`. The planar representation is expressed in a Cartesian two-dimensional coordinate system using X and Y coordinates. The planar representation is usually used for visualization or for the definition of a grid reference system.

The main methods of the interface `ILcdProjection` concern the transformation of geographically referenced data to the planar representation and the other way around. The approximation of the earth can either be with a sphere or an ellipsoid.

Program 219 shows how to transform geodetic lon-lat-height coordinates based on an `ILcdEllipsoid` (called `ellipsoid`) to planar XY coordinates which are represented by `ILcd2DEditablePoint` objects. For the inverse transformation of projections that take the ellipsoidal height into account, the assumption is that this height is already available within the side effect parameter `LLHP2` (of type `ILcd3DEditablePoint`), such that only the longitude and latitude have to be determined.

```
1 projection.lonlatheight2worldOnEllipsoidSFCT( LLHP, ellipsoid, XY );
2 projection.world2lonlatOnEllipsoidSFCT( XY, ellipsoid, LLHP2 );
3
4 projection.lonlatheight2worldOnSphereSFCT( LLHP, radius, XY );
5 projection.world2lonlatOnSphereSFCT( XY, radius, LLHP2 );
```

Program 219 - Transformations between geodetic and world coordinates

In general, an `ILcdProjection` projects only a part of the world to a plane. If a coordinate is passed to the methods shown in Program 219, and that coordinate cannot be projected to a

plane by the given `ILcdProjection`, a `TLcdOutOfBoundsException` is thrown.

The interface `ILcdProjection` has several methods that define the valid area of a map projection. The method `inLonLatBounds` checks whether a given lon-lat-height coordinate is inside the valid area of the `ILcdProjection`. The methods `boundaryLons(aLatitude)` and `boundaryLats(aLongitude)` determine the valid longitude and latitude intervals, respectively, for a given latitude or a given longitude. The inverse transformation of an `ILcdProjection` also has a valid domain. The methods `inWorldBoundsOnSphere` and `inWorldBoundsOnEllipsoid` check whether a given Cartesian coordinate is inside the domain of the inverse transformation for a sphere and an ellipsoid, respectively.

Classes implementing the interface `ILcdProjection` provide transformations for both spherical and ellipsoidal models of the earth. If the ellipsoidal formulas are not useful or not available, you can approximate the ellipsoidal methods by the spherical methods using the semi-major axis as the earth radius.

Each `ILcdProjection` has a **point of origin**. The point of origin is the geodetic coordinate that corresponds to the origin of the planar Cartesian coordinate system. How this point of origin is specified or how it can be set, depends on the characteristics of the `ILcdProjection` and is specified by the classes implementing the interface `ILcdProjection`. In addition to the projection origin, an `ILcdProjection` has other properties depending on the type of `ILcdProjection` as described in [Section 43.2](#).

The interface `ILcdProjection` applies the Listener pattern in collaboration with `java.beans.PropertyChangeListener`. A `PropertyChangeListener` accepts `java.beans.PropertyChangeEvent` objects from the `ILcdProjection` objects to which it is registered. A `PropertyChangeEvent` is sent from an `ILcdProjection` to its listeners whenever one of its properties has changed. For more information on listening to changes, refer to [Section 8.2](#).

43.2 Extensions of `ILcdProjection`

The interface `ILcdProjection` is further extended in several interfaces. This is done through a classification of the projections according to their properties.

- `ILcdCylindrical` specifies a cylindrical `ILcdProjection`. It is defined by wrapping a cylinder around the earth globe such that it touches the equator. The point of origin of the `ILcdProjection` is always located on the equator and is only determined by the longitude.

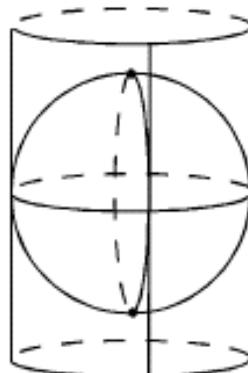


Figure 135 - Cylindrical

- `ILcdTransverseCylindrical` specifies a transverse cylindrical `ILcdProjection`. It is defined by a cylinder that touches the earth globe at the poles. The point of origin of the `ILcdProjection` is defined by the central meridian and is always located on the equator.

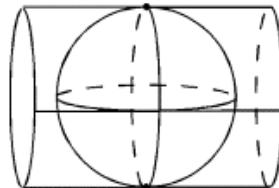


Figure 136 - Transverse Cylindrical

- `ILcdObliqueCylindrical` specifies a cylindrical `ILcdProjection` with the cylinder tilted over an azimuth angle.

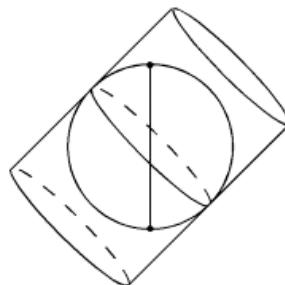


Figure 137 - Oblique Cylindrical

- `ILcdAzimuthal` specifies an azimuthal `ILcdProjection`. It is defined by a tangent plane at the point of origin defined by a geodetic latitude and longitude.

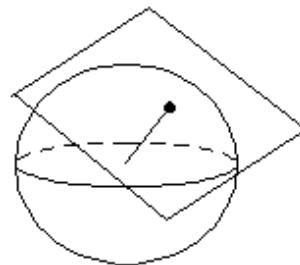


Figure 138 - Azimuthal

- `ILcdConic` specifies a conical `ILcdProjection` that cuts the earth globe at two standard parallels and for which the point of origin of the `ILcdProjection` is defined by a geodetic latitude and longitude.

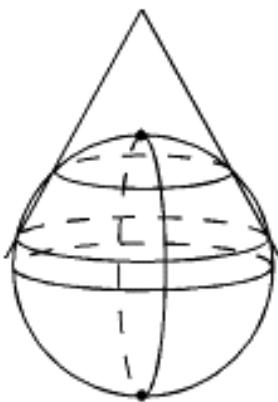


Figure 139 - Conic

- `ILcdPerspective` is an extension of `ILcdAzimuthal`, where the point of origin is not necessarily located on the surface of the earth.

For a complete list of available sub-interfaces, refer to the package `com.luciad.projection` in the API reference.

43.3 Main implementations of `ILcdProjection`

LuciadLightspeed provides implementations for commonly used projections. All implementations extend abstract adapter classes for the interfaces defined in [Section 43.2](#). These abstract classes implement the properties specific for each type of `ILcdProjection`. They also have a method `loadProperties` to initialize instances of each class using the properties mechanism explained in [Chapter 58](#). `TLcdProjectionFactory` creates `ILcdProjection` objects using this properties mechanism.

- `TLcdEquidistantCylindrical` implements the interface `ILcdCylindrical`. It transforms the lon-lat coordinates to a planar representation by considering longitude and latitude as simple rectangular coordinates. It is often used as a grid system for basic un-projected geodetic coordinates.
- `TLcdMercator` implements the interface `ILcdCylindrical`. It is used for world maps and maritime navigational charts. It is also used for grid systems for maritime navigational raster data such as the British Admiralty charts.
- `TLcdTransverseMercator` implements the interface `ILcdTransverseCylindrical`. It is the basis for the UTM grid system (`TLcdUTMGrid` and `TLcdUTMGridSystem`).
- `TLcdLambertConformal` implements the interface `ILcdConic`. It is used for aeronautical navigational charts. It is also the basis for a number of national grid systems, for example, `TLcdLambert1972BelgiumGridReference` and `TLcdLambertFrenchGridReference`.
- `TLcdStereographic` implements the interface `ILcdAzimuthal`. It is used for aeronautical operational display systems such as an Air Track Display. It is also the basis for a number of national grid systems such as `TLcdRD1918DutchGridReference`.
- `TLcdGnomonic` implements the interface `ILcdAzimuthal`. It is used for maritime operational display systems.
- `TLcdOrthographic` implements the interface `ILcdAzimuthal`.

- `TLcdVerticalPerspective` implements the interface `ILcdPerspective`. It is typically used for interpreting satellite images.
- `TLcdObliqueMercator` implements the interface `ILcdObliqueCylindrical`.
- `TLcdCassini` implements the interface `ILcdTransverseCylindrical`.
- `TLcdPolarStereographic` implements the interface `ILcdAzimuthal`, used for mapping polar areas. It is also used to define the UTM grid system for polar areas.

For a detailed description of the available implementations of the interface `ILcdProjection`, refer to the package `com.luciad.projection` in the API reference.

43.4 Literature on map projections

- Snyder, *Map Projections: A Working Manual*, 1987.
- Bugayevskiy, Snyder, *Map Projections : A Reference Manual*, 1995.
- Snyder, *Flattening the Earth : Two Thousand Years of Map Projections*, 1997.
- Pearson, *Map Projections : Theory and Applications*, 1990.
- Maling, *Coordinate systems and map projections*, 1992.

CHAPTER 44

Coordinate systems and transformations

This chapter describes the three different coordinate systems that are used within LuciadLightspeed, and the transformations that exist between them. [Section 44.1](#) explains the three different coordinate systems and their interrelation. [Section 44.2](#) gives an overview of the different reference systems that are implemented within LuciadLightspeed to define a coordinate system, and [Section 44.3](#) explains how you can define your own reference system. Finally, the transformations between the different coordinate systems are treated in [Section 44.4](#).

44.1 Model, world, and view coordinate systems

As described in [Section 2.5.1](#), LuciadLightspeed uses three different coordinate systems to define the location of domain objects on the level of model, world, and view. [Figure 140](#) shows these systems and the relations between them.

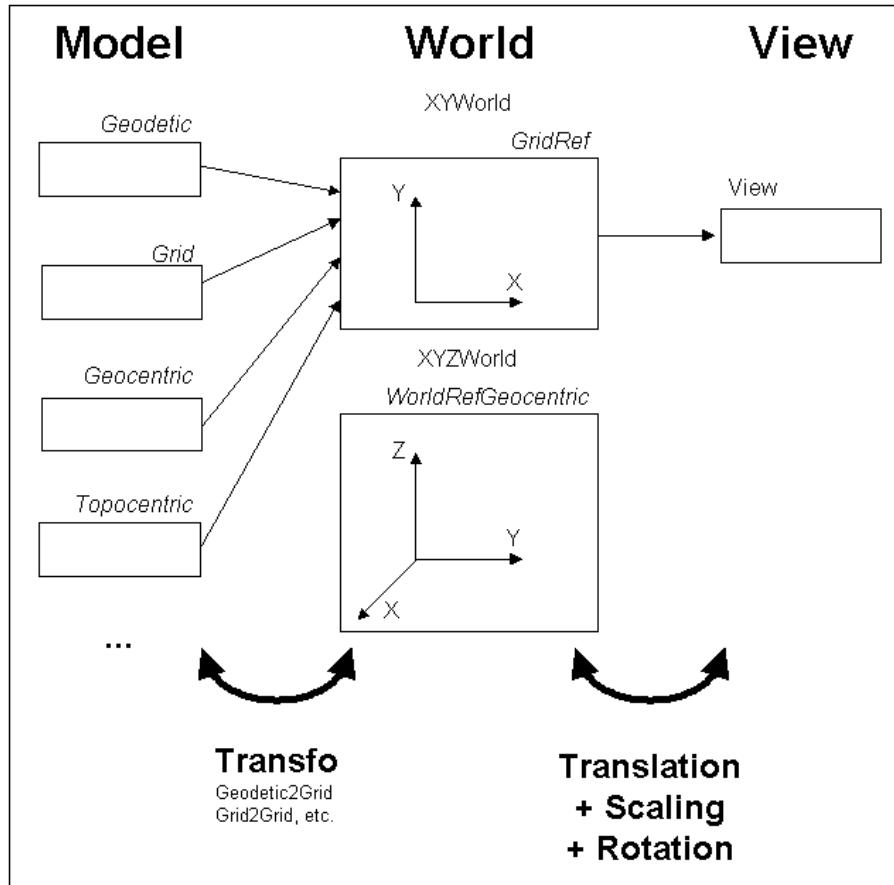


Figure 140 - Coordinate systems and the relation between them

Model coordinates are defined within a coordinate system determined by the **model reference** of the type `ILcdModelReference`. All the domain objects of an `ILcdModel` share this same coordinate system. In the case of geographical data, the `ILcdModelReference` is often an implementation of `ILcdGeoReference`. If so, the unit of measure for the linear coordinates of a model geometry is determined by the `ILcdGeodeticDatum` of the `ILcdGeoReference`.

In particular, the linear unit of measure is given by the unit of measure that is used to express the values of the semi-major and semi-minor axis of the `ILcdEllipsoid` property of the `ILcdGeodeticDatum` (see Chapter 42). Angular units, giving longitude and latitude values, are normally expressed in decimal degrees. The left side of Figure 140 shows several types of `ILcdModel` (drawn as a rectangle), each having its own model reference (labeled in italics). In one `ILcdModel` the domain objects are expressed in a geodetic coordinate system, in another `ILcdModel` they are expressed in a grid system, and so on.

World coordinates are defined within a coordinate system determined by the **world reference** associated with an `ILcdGXYView` or a `ILspView`. This coordinate system is by definition Cartesian. It can either be a 2D coordinate system, if the world reference is of type `ILcdXYWorldReference`, or a 3D coordinate system, if it is of type `ILcdXYZWorldReference`. The most commonly used `ILcdXYWorldReference` for mapping is the `TLcdGridReference`. It is defined by the combination of an `ILcdGeodeticDatum` and an `ILcdProjection` that transforms positions on the `ILcdEllipsoid` of the `ILcdGeodeticDatum` to planar Cartesian coordinates, a scale, a rotation, and a false origin. The `ILcdGeodeticDatum` determines the unit of measure for the Cartesian world coordinates. For more information on projections, see Chapter 43.

View coordinates are the 2D screen coordinates (or pixel coordinates). The origin is the upper-left corner of the view, with y -values increasing downwards, and the coordinate values are integer pixel values.

Both the world and the view coordinate system are associated with an `ILcdGXYView`. The world level is an intermediate level between the model and the view, which is used internally in the `ILcdGXYView`, mainly for performance reasons.

When a domain object is displayed in a view, model coordinates have to be transformed into view coordinates. Conversely, when objects are being edited in a view, the view coordinates have to be transformed into the correct model coordinates. LuciadLightspeed performs such transformations always in two steps:

- Transformation between the model and the world coordinates. These transformations are typically non-linear in case of geo-referenced model data. This means that they are expensive with respect to computing time. Dedicated transformations are used for each particular combination of a model reference and a world reference.
- Transformation between the world and the view coordinates. This transformation is linear, and therefore inexpensive with respect to computing time. It corresponds to an affine transformation in the most general case. Only translation, rotation, and scaling are needed.

Figure 140 shows one 2D world reference (`TLcdGridReference`) and one 3D world reference (`TLcdGeocentricReference`). `TLcdGeodetic2Grid` and `TLcdGrid2Grid` are examples of dedicated model-world transformations.

44.2 LuciadLightspeed reference systems

LuciadLightspeed provides a number of model and world reference systems. They are defined in the package `com.luciad.reference`. Implementations in that package implement one or more of the model and/or world references.

LuciadLightspeed is intended to work with geo-referenced data, that is data tied to an earth-based reference system. All earth-based reference systems within LuciadLightspeed extend the interface `ILcdGeoReference`. As a common property they have an `ILcdGeodeticDatum` (see Section 42.1).

Specific reference interfaces extending `ILcdGeoReference` are:

- `ILcdGeodeticReference`: a reference system for longitude-latitude-height values based on an `ILcdGeodeticDatum`.

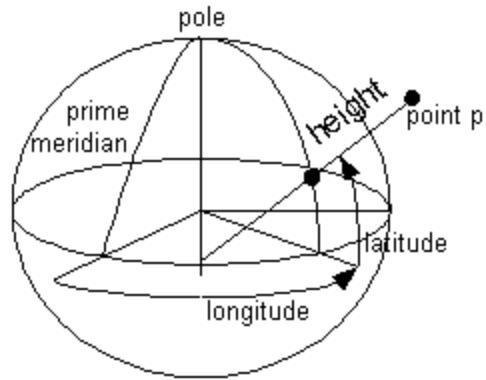


Figure 141 - ILcdGeodeticReference

- **ILcdGeocentricReference:** a 3D Cartesian coordinate system attached to the center of the earth.

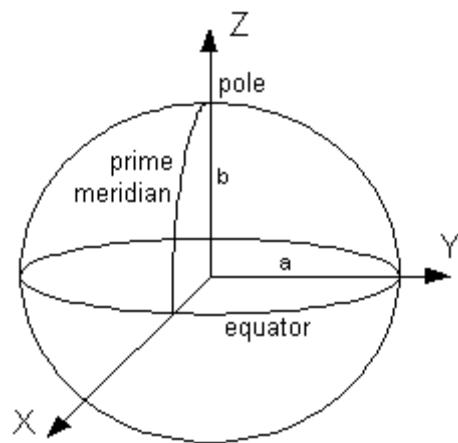


Figure 142 - ILcdGeocentricReference

- **ILcdTopocentricReference:** a 3D Cartesian coordinate system located at some lon-lat-height point P_0 : the Y -axis intersects the polar axis, the XY -plane is parallel to the tangential plane through P , with the X -axis pointing to the east, and the Z -axis is perpendicular to the XY -plane (thus it also intersects the polar axis).

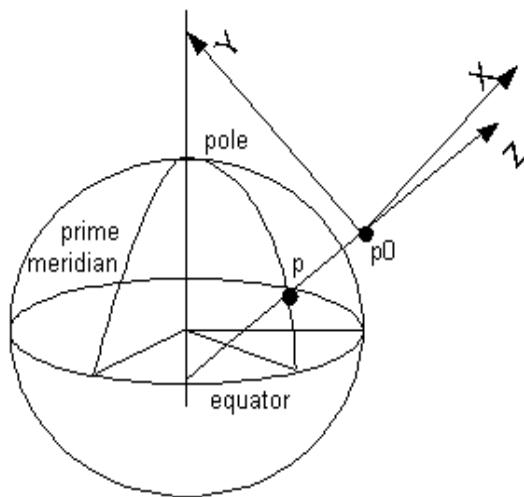


Figure 143 - ILcdTopocentricReference

- `ILcdGridReference`: specifies a grid coordinate `ILcdGeoReference`, which is defined by an `ILcdGeodeticDatum`, an `ILcdProjection`, a scale, a rotation, a unit of measure and a false origin (given by a false Easting and a false Northing).

Table 5 cross-references the specific reference interfaces in the LuciadLightspeed API with the interface they extend. An interface in the left column extends an interface in the top row if and only if the corresponding cell is marked.

extends	<code>ILcdGeoReference</code>	<code>ILcdModelReference</code>	<code>ILcdXYZWorldReference</code>	<code>ILcdXYZWorldReference</code>
<code>ILcdGeodeticReference</code>	Yes	Yes		
<code>ILcdGeocentricReference</code>	Yes	Yes	Yes	
<code>ILcdTopocentricReference</code>	Yes	Yes	Yes	
<code>ILcdGridReference</code>	Yes	Yes	Yes	Yes

Table 5 - Specializations of `ILcdGeoReference`

Each of the specific reference interfaces has a default implementation (which has the same name as the interface, except that it starts with `TLcd` instead of `ILcd`).

In addition to the default implementations, LuciadLightspeed offers implementations useful for particular data. These are implementations in which the parameters of the `TLcdGridReference` are set:

- `TLcdRD1918DutchGridReference`
- `TLcdLambert1972BelgiumGridReference`
- `TLcdLambertFrenchGridReference`
- `TLcdDHDNGermanGridReference`
- `TLcdSwissGridReference`

44.3 Defining your own reference system

Program 220 shows the implementation of `TLcdRD1918DutchGridReference` as an extension of `TLcdGridReference`.

```

1 double scale = 0.9999079;
2 double false_easting = 155000.0;
3 double false_northing = 463000.0;
4
5 double originLat = 52.0 + 9.0/60.0 + 22.178/3600.0;
6 double originLon = 5.0 + 23.0/60.0 + 15.5/3600.0;
7 TLcdDutchStereographic projection =
8     new TLcdDutchStereographic(originLon,originLat);
9 double uom = 1.0;
10
11 // Geodetic datum not defined in the factory
12 ILcdEllipsoid wgs84_ellipsoid =
13     ( new TLcdEllipsoidFactoryDMA1987b() ).createEllipsoid(
14         TLcdEllipsoidFactoryDMA1987b.WGS_1984 );
15
16 ILcdGeodeticDatum ref_datum =
17     new TLcdGeodeticDatum( wgs84_ellipsoid, "WGS-1984" );
18
19 ILcdEllipsoid ellipsoid =
20     ( new TLcdEllipsoidFactoryDMA1987b() ).createEllipsoid(
21         TLcdEllipsoidFactoryDMA1987b.BESSEL_1841 );
22
23 ILcdGeodeticDatum geodetic_datum =
24     new TLcdGeodeticDatum( 565.040,
25         49.91,
26         465.84,
27         1.9848E-06,
28         -1.7439E-06,
29         9.0587E-06,
30         1.0 - 4.0772E-06,
31         ellipsoid,
32         ref_datum,
33         "BESSEL_AMERSFOORT" );
34 this.setGeodeticDatum( geodetic_datum );
35 this.setProjection( projection );
36 this.setFalseEasting( false_easting );
37 this.setFalseNorthing( false_northing );
38 this.setScale( scale );
39 this.setUOMLengthInMeter( uom );
40 this.setRotation( 0.0 );

```

Program 220 - Implementing a specific `ILcdGeoReference`

LuciadLightspeed also provides a factory, `TLcdModelReferenceFactory`, to create instances of implementations of the interface `ILcdGeoReference`. This class lets you create an instance of `ILcdGeoReference` based on properties as described in Chapter 58.

44.4 Transformations between coordinate systems

Domain objects are represented in different coordinate systems. As explained in [Section 44.1](#), transformations between model, world, and view coordinates are required to display data coherently and to edit data correctly.

LuciadLightspeed defines a number of interfaces and abstract classes for transformations between coordinate systems. You can find them in `com.luciad.transformation`, `com.luciad.view.gxy` and `com.luciad.view.lightspeed.camera`. All interfaces provide both forward and backward transformations:

- `ILcdModelXYWorldTransformation`: specifies the transformations between model coordinates and 2D world coordinates. Implementations are given by `TLcdGeocentric2Grid`, `TLcdGeodetic2Grid`, `TLcdGrid2Grid`, and `TLcdTopocentric2Grid`. These transformations are all dedicated to a particular combination of a model reference and a world reference. The first part of the class name refers to the model reference, the second part to the world reference.
- `ILcdModelXYZWorldTransformation`: specifies the transformations between model coordinates and 3D world coordinates. The interface is implemented by `TLcdGeodetic2Geocentric` and `TLcdGrid2Geocentric`.
- `ILcdXYWorldXYWorldTransformation`: specifies the transformation between one world reference and another world reference. The interface is implemented by `TLcdMapWorldTransformation`.
- `ILcdGXYViewXYWorldTransformation`: specifies the transformations between world coordinates and view coordinates in a GXY view. The interface is implemented by `TLcdGXYViewXYWorldTransformation`.
- `ILcdGXYViewModelTransformation`: combines an `ILcdModelXYWorldTransformation` and an `ILcdGXYViewXYWorldTransformation` into a single transformation for a GXY view. `TLcdGXYViewModelTransformation` implements this interface.
- `ILcdModelModelTransformation`: specifies the transformation between one model reference and another model reference. The interface is implemented by `TLcdGeoReference2GeoReference`.
- `ALspViewXYZWorldTransformation`: defines a transformation between world coordinates and view coordinates for a Lightspeed view. This class is extended to subclasses `TLspViewXYZWorldTransformation2D` and `TLspViewXYZWorldTransformation3D`.

There are two groups of transformation methods of these interfaces:

- methods that transform points: `ILcdPoint` (see [Section 11.2](#)) or `java.awt.Point` objects.
- methods that transform bounding boxes: `ILcdBounds` (see [Section 11.3](#)) or `java.awt.Rectangle` objects.

For example, the interface `ILcdModelXYWorldTransformation` has the methods:

- `modelPoint2worldSFCT`
- `worldPoint2modelSFCT`
- `modelBounds2worldSFCT`

- `worldBounds2modelsFCT`

Additionally LuciadLightspeed provides the following interfaces and support classes to perform transformations:

- The interface `ILcdTopocentricCoordSysTransformation` performs transformations of `ILcdPoint` objects between an `ILcdTopocentricReference` and an `ILcdGeodeticReference` or an `ILcdGeocentricReference`. The class `TLcdTopocentricCoordSysTransformation` implements this interface.
- The class `TLcdGridReferenceUtil` is a utility class that performs transformations between an `ILcdGridReference` and an `ILcdGeodeticReference`.

CHAPTER 45

Geometric calculations

Many applications that deal with geospatial data need to be able to perform various geometric calculations on the data. A few examples of such calculations are area computation for surfaces, containment and intersection tests, and distance calculations. LuciadLightspeed offers support for this kind of operations in the packages `com.luciad.geodesy` and `com.luciad.geometry`. Most of this functionality is specific for the coordinate system or in which the operations are performed. The first sections of this chapter cover ellipsoidal, spherical, and Cartesian calculations. The chapter concludes with a discussion of `ILcdSegmentScanner`.

45.1 Ellipsoidal calculations

The interface `ILcdEllipsoid` defines several geometric calculations that you can perform on the surface of an ellipsoid (see [Chapter 42](#)). You can perform various additional calculations using `TLcdEllipsoidUtil`. The methods of `TLcdEllipsoidUtil` all accept an `ILcdEllipsoid` as one of their parameters.

`TLcdEllipsoidUtil` supports the following operations:

- closest point determination
- containment tests for line segments
- intersection tests for line segments
- conformal latitude computation
- buffer contour construction

Refer to the API reference for more information on `TLcdEllipsoidUtil` and its methods.

45.2 Spherical calculations

The class `TLcdSphereUtil` provides various spherical trigonometry operations. Whereas `TLcdEllipsoidUtil` works on an ellipsoid and returns results measured in meters, `TLcdSphereUtil` works entirely in geodetic coordinates and returns results measured in arc degrees. Several of `TLcdSphereUtil`'s methods correspond to those found in `ILcdEllipsoid`. Using a spherical instead of an ellipsoidal calculation produces an approximate result, but may be desirable for performance reasons.

`TLcdSphereUtil` supports the following operations:

- various containment and intersection tests
- distance, area, and orientation calculations
- closest point determination
- conversion between Cartesian and geodetic coordinates

Refer to the API reference for more information on `TLcdSphereUtil` and its methods.

45.3 Cartesian calculations

For operations in Cartesian space, LuciadLightspeed provides the class `TLcdCartesian`. Its functionality is, to a large extent, equivalent to `TLcdSphereUtil`, although obviously the inputs and outputs of `TLcdCartesian`'s methods are expressed in Cartesian coordinates rather than geodetic coordinates. Also note that while the majority of methods operate in 2D space (XY coordinates), some operations are also implemented in 3D (XYZ coordinates). Refer to the API reference for an overview of the specific methods and their usage.

45.4 Sampling line segments

The interface `ILcdSegmentScanner` allows you to compute sample points along a line segment. The sample points are at a regular interval. Sampling of line segments is for example used to extract the underlying terrain heights for every sample point (refer to [Chapter 38](#)). With the sample data, it is then for example possible to construct a profile view of the terrain covered by the line segment.

LuciadLightspeed provides three implementations of `ILcdSegmentScanner`:

- `TLcdGeodeticSegmentScanner`: generates points along a geodetic line
- `TLcdRhumblineSegmentScanner`: generates points along a rhumb line, that is, a line of constant azimuth
- `TLcdGridSegmentScanner`: generates points along a Cartesian line

The main method of `ILcdSegmentScanner` is `scanSegment()`, which takes the following arguments:

- a distance delta that indicates the desired spacing of the generated points
- the start and end point of the line segment from which to generate the points
- an `ILcdOnPointFunction` to be invoked for every generated point

All implementations of `ILcdSegmentScanner` compute points along the input line segment, and pass each of them in turn to the supplied `ILcdOnPointFunction`. Refer to the API reference for further details.

CHAPTER 46

Rectification

Rectification is the process that corrects the distortion introduced in aerial and satellite imagery, so that the images can be more accurately georeferenced. In general, rectification methods can be classified into parametric and non-parametric methods. Parametric rectification uses prior knowledge about the properties of the imaging sensor (plane or satellite trajectory, altitude, orientation) and the terrain elevation. Non-parametric rectification relies on manually defined tie points, relating image coordinates to known geographic locations using a mathematical function. It does not explicitly model the variations in terrain altitude.

The first part of this chapter presents non-parametric rectification. It is mathematically simpler, and therefore potentially less accurate. Section 46.1.1 starts by illustrating the chain of transformations used for mapping the pixels from an image to their corresponding geographical locations in more common geographical references. Section 46.1.2 continues with an explanation of the place of the rectification within this framework, and section 46.1.3 briefly describes the LuciadLightspeed classes involved. The second part of this chapter presents parametric rectification, also known as [orthorectification](#). Section 46.2.1 introduces the camera parameters and section 46.2.2 explains the usage of terrain elevation. Finally, the last part of the chapter shows how parametric and non-parametric rectification can be combined.

46.1 Non-parametric rectification

46.1.1 Transformations in a non-rectified grid reference

A grid reference system is composed of two main parts: a geodetic datum and a projection as shown in Figure 144.

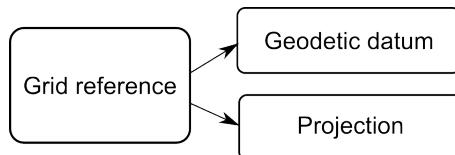


Figure 144 - A grid reference requires a datum and a projection

In general, to display an image in a geographical system, you need to define three distinct coordinate systems:

- the image coordinate system, expressed in pixels

- the grid (raster) coordinate system, usually expressed in meters
- the geodetic coordinate system, expressed as longitude/latitude in a given geodetic datum

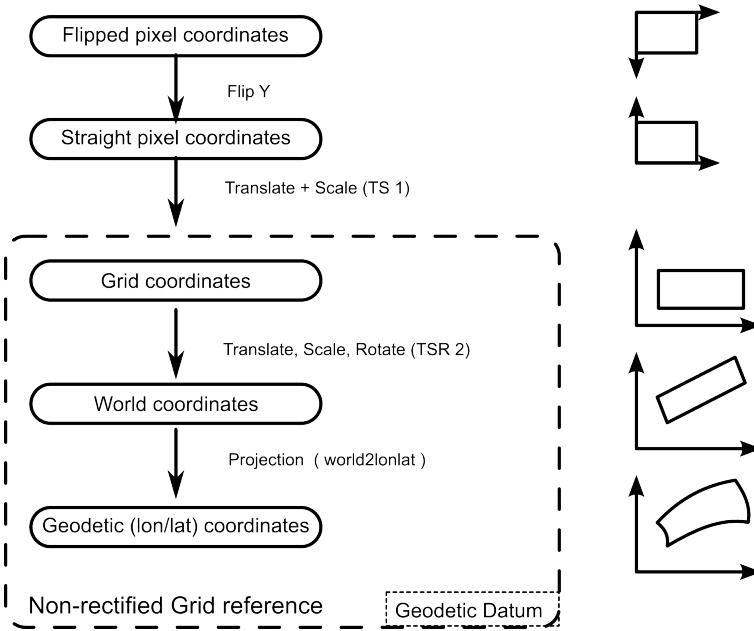


Figure 145 - Transformations from pixels to geodetic coordinates in a non-rectified grid

Apart from these three, other auxiliary systems may be used; see Figure 145. For example, when an image is initially read from disk, it is in a **flipped** system – the origin of the image is in the upper-left corner with the Y axis pointing down, while in most Cartesian coordinate systems, the origin is considered to be the lower-left corner of the image with the Y axis pointing up (straight pixel coordinates).

In the grid reference system, the boundaries of the raster are assumed to be aligned with the axis of coordinates, so in order to map a point from image coordinates to grid coordinates you only need to perform a translation and a scaling transformation (TS 1). This transformation is performed transparently by the `TLcdRaster` class or by the equivalent implementation of the `ILcdRaster` interface.

From grid coordinates, a translation+scale+rotation operation (TSR 2) is applied to get to world coordinates, followed by a projection that produces geodetic coordinates (`world2lonlat`). Geodetic coordinates can then be transformed to any other reference system (for example, to the `ILcdXYWorldReference` of the map, during displaying).

46.1.2 Transformations in a rectified grid reference

Conceptually, the principle behind a rectified grid reference is identical to the non-rectified case. There are only a few practical differences, as shown in Figure 146. The grid coordinates and the pixel coordinates are usually the same. After the initial translation+scale+rotation (TSR2) operation, non-rectified world coordinates are obtained which are passed through a rectifying function to obtain world coordinates. A normal projection is then applied to finally get geodetic coordinates.

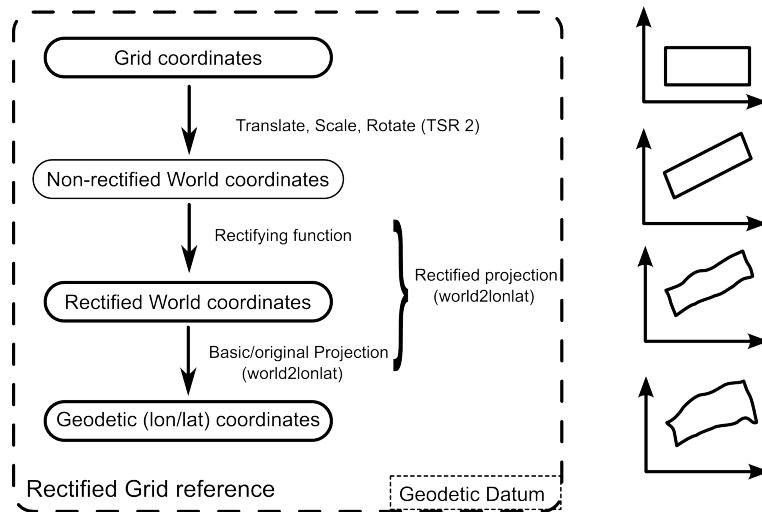


Figure 146 - Transformations from pixels to geodetic coordinates in a non-rectified grid

To keep the API consistent, the rectifying function and the projection step are grouped in a special type of projection – an `ILcdRectifiedProjection` – which wraps the traditional projection. This allows for a rectified reference to be used just like any other grid reference. The data structure is illustrated in [Figure 147](#).

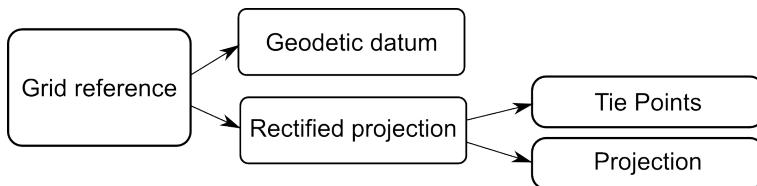


Figure 147 - The components of a rectified reference

46.1.3 Interfaces and classes used for rectification

Projections

As already mentioned, the central role is played by the `ILcdRectifiedProjection`. The interface offers access to two sets of points, called source points and target points. The source points are expressed in the non-rectified world coordinates and the target points are expressed in the rectified world coordinates (see [Figure 146](#)). The two sets of points are used internally to compute the parameters of the rectifying function. In addition, the interface offers access to the internal projection.

There are a number implementations of this interface in LuciadLightspeed: `TLcdRectifiedPolynomialProjection`, `TLcdRectifiedRationalProjection`, and `TLcdRectifiedProjectiveProjection`. The polynomial projection applies a 2D polynomial of the form:

$$\begin{cases} x' = \sum_{i,j=0}^N a_{i,j} x^i y^j; \\ y' = \sum_{i,j=0}^N b_{i,j} x^i y^j; \end{cases}$$

The rational projection applies a rational function of the form:

$$\begin{cases} x' = \frac{\sum_{i,j=0}^N a_{i,j} x^i y^j}{\sum_{i,j=1}^D b_{i,j} x^i y^j} & \text{where } b_{0,0} = 1; \\ y' = \frac{\sum_{i,j=0}^N c_{i,j} x^i y^j}{\sum_{i,j=1}^D d_{i,j} x^i y^j} & \text{where } d_{0,0} = 1; \end{cases}$$

The projective projection applies a projective transformation of the form:

$$\begin{cases} x' = \frac{a_1 x + a_2 y + a_3}{c_1 x + c_2 y + c_3}; \\ y' = \frac{b_1 x + b_2 y + b_3}{c_1 x + c_2 y + c_3}; \end{cases}$$

All classes take pairs of tie points as input, defined in (rectified/non-rectified) world coordinates. They internally compute the coefficients of the polynomials, minimizing the least-squares error of the mapping.

Raster referencers

The rectifying projections discussed in the previous section only work in world coordinates. For convenience, LuciadLightspeed also provides a number of higher-level classes that allow you to create raster references starting from image points, defined in pixel coordinates, and model points, defined in any georeference. The classes implement the interface `ILcdRasterReferencer`. The basic implementations only perform simple operations like rotations and translations. The more advanced implementations, corresponding to the projections above, are `TLcdPolynomialRasterReferencer`, `TLcdRationalRasterReferencer`, and `TLcdProjectiveRasterReferencer`.

[Program 221](#) illustrates a typical usage scenario. Given image tie points expressed in pixel coordinates, and model tie points expressed in a given model reference, the raster referencer derives an appropriate model reference and bounds for the raster.

```

1 int image_width = 1000;
2 int image_height = 1000;
3
4 ILcdPoint[] image_points = new ILcdPoint[] {
5     new TLcdXYPoint( 0,      0),
6     new TLcdXYPoint(1000,    0),
7     new TLcdXYPoint( 0, 1000),
8     new TLcdXYPoint(1000, 1000),
9     new TLcdXYPoint( 126,   274),
10    new TLcdXYPoint( 983,   829),
11 };
12
13 ILcdModelReference model_points_reference = new TLcdGeodeticReference();
14
15 ILcdPoint[] model_points = new ILcdPoint[] {
16     new TLcdLonLatPoint(20.4, 30.8),
17     new TLcdLonLatPoint(25.1, 30.6),
18     new TLcdLonLatPoint(20.9, 35.7),
19     new TLcdLonLatPoint(25.2, 35.3),
20     new TLcdLonLatPoint(21.0, 31.9),
21     new TLcdLonLatPoint(24.9, 35.0),
22 };
23
24 int degree = 2;
25
26 ILcdRasterReferencer raster_referencer =
27     new TLcdPolynomialRasterReferencer(degree);
28
29 ILcdRasterReference raster_reference =
30     raster_referencer.createRasterReference(image_width,
31                                             image_height,
32                                             image_points,
33                                             model_points_reference,
34                                             model_points,
35                                             null);
36
37 ILcdModelReference model_reference = raster_reference.getModelReference();
38 ILcdBounds bounds = raster_reference.getBounds();

```

Program 221 - Rectifying an image with the help of a raster referencer

Once the raster reference and its bounds are known, the raster can be added to a model as shown in [Program 222](#).

```

1 ILcdTile[][] tiles = ...;
2 double pixel_density = (image_width / bounds.getWidth()) *
3     (image_height / bounds.getHeight());
4 int default_value = 0;
5 ColorModel color_model = ColorModel.getRGBdefault();
6
7 ILcdRaster raster = new TLcdRaster(bounds,
8                                     tiles,
9                                     pixel_density,
10                                    default_value,
11                                    color_model);
12
13 ILcdModel model = new TLcd2DBoundsIndexedModel(model_reference,
14                                                 model_descriptor);
15
16 model.addElement(raster, ILcdFireEventMode.NO_EVENT);

```

Program 222 - Adding a raster to a model

46.1.4 Limitations

Polynomial functions are notoriously unstable when their degree increases. For this reason it is recommended that you use only low-degree polynomials (2, maximum 3). Rational functions have the added problem of the denominator possibly becoming 0, which may make certain points unmappable.

46.2 Parametric rectification (orthorectification)

When both the imaging sensor parameters and a digital terrain elevation model are available, the distortion introduced by the terrain can be taken into account, as illustrated in Figure 148. If only the camera parameters would be available, a geodetic point located on the surface of the ellipsoid would be projected directly to a corresponding point in image coordinates (denoted as source point). However, due to the terrain elevation the geodetic point should actually mapped to a different position in the image: the target point. The process of canceling the distortions introduced by the camera projection and the terrain is sometimes called orthorectification.

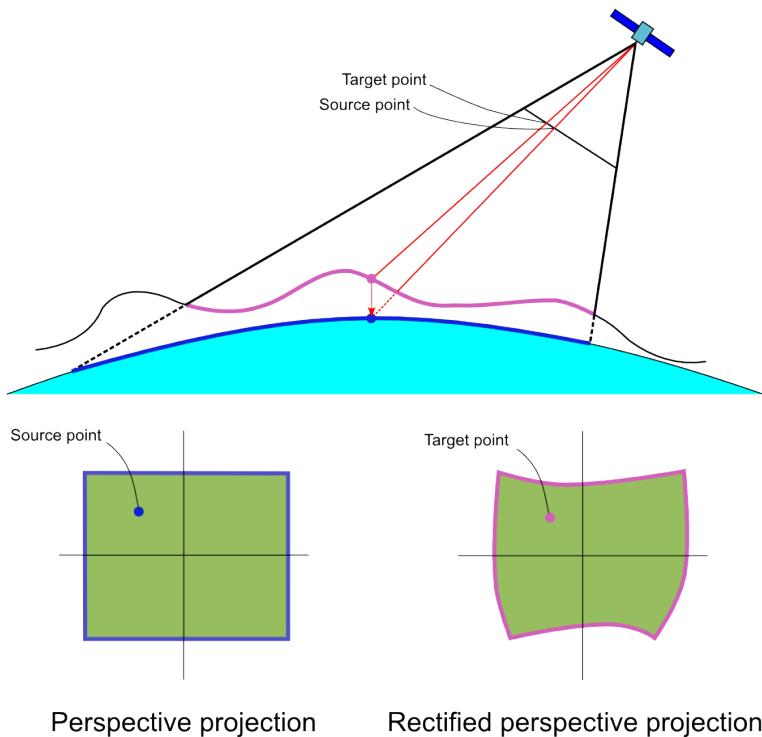


Figure 148 - Image rectification when the sensor parameters and the terrain elevation are available

46.2.1 Parameters related to the imaging sensor

The mapping from the 3D world to the camera's 2D sensor can be expressed as a perspective projection. This projection is implemented by the class `TLcdPerspectiveProjection`, which requires the following information:

- the center of perspective
- the principal point
- the camera rotation around the principal axis

The center of perspective can be thought of as the sensor location, while the principal point is the foot of the perpendicular to the projection plane through the perspective center. Together, these two points define the principal axis (also known as optical axis) and the position of the projection plane. The principal point is also the origin of the projection plane.

The constructors of the `TLcdPerspectiveProjection` expect the center of perspective and the principal point to be expressed in geocentric coordinates. The camera rotation around the principal axis can be expressed either as a roll angle or as an up vector. The roll angle is measured in degrees, clockwise in the projection plane, relative to the plane determined by the camera vector and the optical axis - see [Figure 149](#). An up vector is a 3D vector orthogonal to the principal axis and parallel to the projection plane's Y axis.

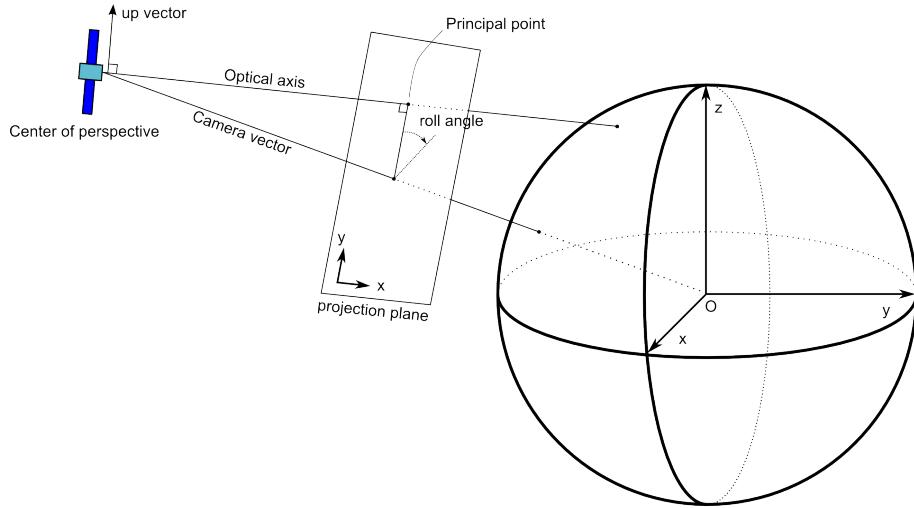


Figure 149 - The parameters of the imaging sensor

46.2.2 Including terrain information

The distortion caused by the topographic relief is modeled in LuciadLightspeed as a projection called `TLcdOrthorectifiedProjection`. As with non-parametric rectification, this allows for a rectified reference to be used like any other grid reference. The orthorectified projection combines the `TLcdPerspectiveProjection` presented in the previous section with elevation data coming from an `ILcdHeightProvider` - see [Figure 150](#).

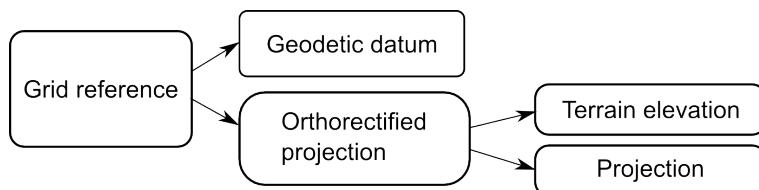


Figure 150 - The components of an orthorectified and corrected grid reference

In its simplest form, the orthorectified projection behaves as a slightly modified version of the perspective projection itself ([Program 223](#)).

```

1 TLcdPerspectiveProjection camera_projection = new TLcdPerspectiveProjection(
2   0, 0, 10000,      // lon, lat, height
3   20, -70, 0,       // azimuth, pitch, roll
4   1000,             // focal length
5 );
6
7 ILcdHeightProvider height_provider = new TLcdFixedHeightProvider(1000);
8 TLcdOrthorectifiedProjection orthorectified =
9   new TLcdOrthorectifiedProjection(camera_projection, height_provider);

```

Program 223 - Creating an orthorectifying projection from camera parameters and terrain elevation

If you need to orthorectify an image that is already reprojected in a different coordinate system you can pass the existing projection as the second argument of the `TLcdOrthorectifiedProjection` constructor:

```

1 ILcdProjection original_projection = new TLcdEquidistantCylindrical();
2 TLcdOrthorectifiedProjection orthorectified =
3   new TLcdOrthorectifiedProjection(camera_projection,
4                                     original_projection,
5                                     height_provider);

```

Program 224 - Creating an orthorectifying projection from camera parameters, terrain elevation and an existing projection

46.2.3 Limitations

The current implementation of `TLcdOrthorectifiedProjection` cannot handle shadowed areas - low elevation terrain of which the visibility is obstructed by nearby high elevation terrain. Shadowed areas result in local mapping discontinuities. Also, the projection cannot be successfully stored and restored from a `Properties` object because the actual mapping is determined by the terrain elevation map. Attempting to store the projection properties will save a single elevation sample, located at the principal point. When the projection is loaded back, that unique sample is used for the entire projection.

46.3 Combining parametric and non-parametric rectification

Non-parametric rectification works well for large areas but cannot cope with localized distortions caused by the terrain. On the other hand, the parametric rectification's explicit modeling of the terrain produces very accurate results, but parametric rectification is very sensitive to the quality of the camera parameters. For example, an error of a fraction of a degree in the camera orientation can easily introduce errors of hundreds of meters at the ground level. Fortunately, the two methods are complementary. By combining them, the overall results can significantly be improved. Any error in the camera parameters can be corrected with just a few tie points, while the local terrain elevation is still taken into account. In terms of code implementation it is sufficient to chain together the original projection, the `TLcdOrthorectifiedProjection` and the `ILcdRectifiedProjection`:

```

1 TLcdFixedHeightAltitudeProvider altitude_provider =
2     new TLcdFixedHeightAltitudeProvider( 1000,
3                                         TLcdCoverageAltitudeMode.ABOVE_ELLIPSOID );
4 ILcdProjection original = new TLcdEquidistantCylindrical( );
5 TLcdPerspectiveProjection perspective = new TLcdPerspectiveProjection(
6     TLcdEllipsoid.DEFAULT,
7     0, 0, 10000,           // camera lon, lat, height
8     20, -70, 0,            // sensor yaw, pitch, roll
9     1000                  // focal length
10 );
11 ILcdPoint[] source_points = new ILcdPoint[]{
12     new TLcdXYPoint( 0, 0 ),
13     new TLcdXYPoint( 1000, 0 ),
14     new TLcdXYPoint( 0, 1000 ),
15     new TLcdXYPoint( 1000, 1000 ),
16 };
17
18 ILcdPoint[] target_points = new ILcdPoint[]{
19     new TLcdXYPoint( -2.5, 0 ),
20     new TLcdXYPoint( 985, 1.2 ),
21     new TLcdXYPoint( 1.3, 1002 ),
22     new TLcdXYPoint( 988.3, 1002 ),
23 };
24 TLcdXYBounds world_bounds = new TLcdXYBounds( -3, 0, 1000, 1002 );
25
26 TLcdOrthorectifiedProjection orthorectified =
27     new TLcdOrthorectifiedProjection( perspective,
28                                     original,
29                                     altitude_provider );
30 TLcdRectifiedPolynomialProjection corrected =
31     new TLcdRectifiedPolynomialProjection( 2,
32                                         source_points,
33                                         target_points,
34                                         orthorectified,
35                                         world_bounds );
36 TLcdGridReference grid_reference = new TLcdGridReference( new TLcdGeodeticDatum(), corrected );

```

Program 225 - Combining the parametric and non-parametric rectification

A visual illustration of the resulting coordinate transformations is presented in Figure 151.

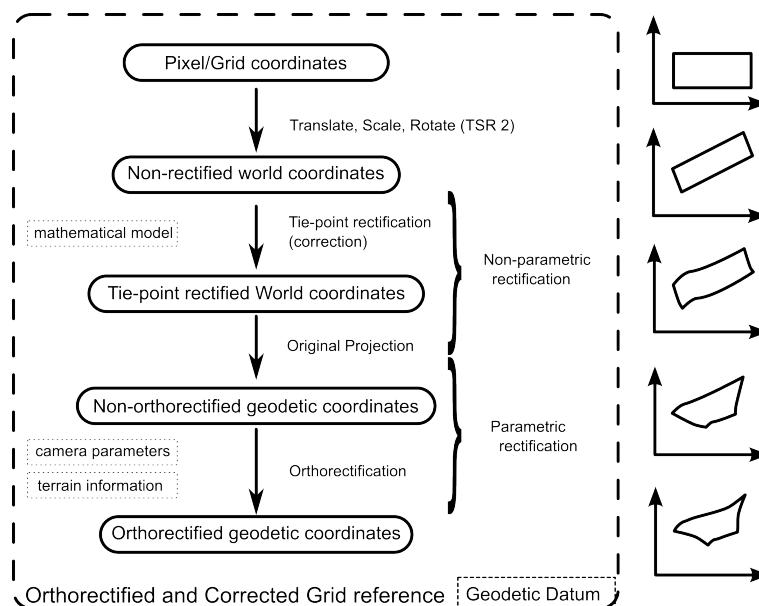


Figure 151 - The result of combining non-parametric and parametric rectification

The final data structure is displayed in Figure 152.

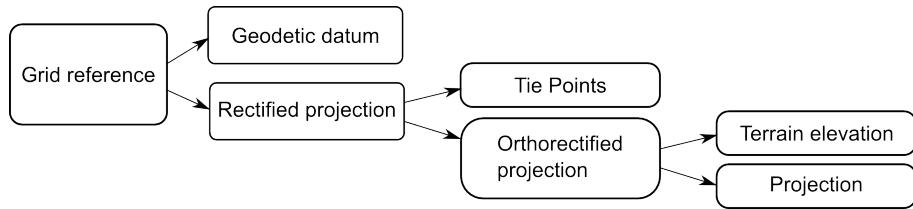


Figure 152 - The components of an orthorectified and corrected grid reference

PART VIII Data Formats and Standards

CHAPTER 47

Adding instant data format support to your application

This chapter explains how you can quickly add support for each data format used in your application. As a result, your application will be able to load each data type as soon as a user opens a new file via a dialog box, or drags and drops a file into the application's view, for example.

LuciadLightspeed comes with pre-configured code that allows you to add all available formats with just a few lines of code, as demonstrated in [Program 226](#).

```
1  fOpenSupport = new GXYOpenSupport(getView());
2  fOpenSupport.addStatusListener(getStatusBar());
3  toolBar.addAction(new OpenAction(fOpenSupport));
4  toolBar.addAction(new OpenURLAction(fOpenSupport));
```

Program 226 - Adding GUI actions to open data from a file or a URL
(from samples/gxy/decoder/MainPanel)

To set up the service look-up mechanism for each data format used in your application:

1. Get a hold of the relevant format-specific services such as model decoders and layer factories. For more information, see [Section 47.1](#).
2. Integrate these data decoding services into a GUI for opening data. See [Section 47.3](#) for more information.
3. Integrate other format-specific services. See [Section 47.4](#) for more information.
4. Define and integrate your own format-specific services, if you have any. See [Section 47.5](#) for more information.

This chapter goes into more detail about each step of the setup of the service look-up mechanism. To read more about such a look-up mechanism as well as other solutions for data format support, see [Section 47.1](#).

47.1 Selecting data format services

Most LuciadLightspeed formats simply consist of a model decoder, an implementation of `ILcdModelDecoder`, and a layer factory, an implementation of `ILcdGXYLayerFactory` or `ILspLayerFactory`. Some formats come with a few extra services, as explained in [Section 47.4](#).

To add support for LuciadLightspeed's formats, you need to instantiate and add these to your application.

There are several ways to add instant data format support in an application:

Direct instantiation: the most straightforward way to add format support. Create your own composite model decoder and layer factory and add the model decoders and layer factories you require in your application. Direct instantiation of a model decoder and layer factory is demonstrated in the LuciadLightspeed fundamentals samples.

A dependency injection framework: this approach wires the required model decoders and layer factories using a dependency injection framework such as Guice or Spring. For more information about dependency injection, refer to the documentation of the respective frameworks.

A service look-up: using a look-up mechanism to query all supported model decoders and layer factories at run-time. In this approach, the support provided for each data format, such as its model decoder and layer factory, is regarded as a service that can be looked up, retrieved and activated. LuciadLightspeed comes with a look-up mechanism that supports both the built-in formats, as well as your own, custom-defined or custom-configured data formats.

While all three options have their advantages and disadvantages, this chapter primarily focuses on the service look-up option, because it is used throughout the LuciadLightspeed samples, and because it does not depend on third-party libraries.

47.2 Looking up data format services

The service look-up mechanism in LuciadLightspeed is compatible with Java's `java.util.service.ServiceLoader`, but offers extra functionality. The most important extra functionality of the service registry in LuciadLightspeed is the ability to sort services based on their priority.

The LuciadLightspeed model decoders and layer factories are automatically registered with the service registry through the annotation `@LcdService`. For more information about these annotations, see [Section 47.5](#). In the sample code, the look-up mechanism is accessed by calling the method `samples.common.serviceregistry.ServiceRegistry.getInstance`. The API consists of a single method `query` that returns an `Iterable` for the given Java class.

[Program 227](#) shows how to retrieve the `ILcdModelDecoder` instances that are available in LuciadLightspeed and the samples.

```
1 ServiceRegistry.getInstance().query(ILcdModelDecoder.class)
```

**Program 227 - Retrieving all available `ILcdModelDecoder` objects
(from `samples/common/formatsupport/OpenSupport`)**

In the same way, you can retrieve all available `ILcdGXYLayerFactory` or `ILspLayerFactory` instances. Next to some format-specific layer factories, LuciadLightspeed also offers layer factories like `GXYUnstyledLayerFactory/UnstyledLayerFactory` and `GXYSLDFileLayerFactory/SLDFileLayerFactory`. These sample layer factories handle formats that do not contain or impose specific styling, for example. They are also registered as a service with the service registry.

Composites are available for many services. Program 228 shows how they can be instantiated. These composites, when instantiated using the service loader, combine all model decoders, layer factories or other services of all available optional components. They can therefore provide sensible default behavior for almost all data formats. For example the optional Ecdis maritime charts are decoded and styled correctly by default if this option is available. Do keep in mind that the exact behavior may change in future versions.

```

1 new TLcdCompositeModelDecoder(ServiceRegistry.getInstance().query(ILcdModelDecoder.class))
2 new TLcdCompositeGXYLayerFactory(ServiceRegistry.getInstance().query(ILcdGXYLayerFactory.class
   ))
3 new TLspCompositeLayerFactory(ServiceRegistry.getInstance().query(ILspLayerFactory.class))
```

Program 228 - Creating composites for services

If you need full control, or if customization is desired, you're recommended to implement your own service. Many examples are available as sample code. If you annotate them with `@LcdService` and generate the services files using the annotation processor, your services take precedence when using the default, or higher, priority. See Section 47.5 for more information on the annotation processor.

47.3 Configuring an action to load model data

This section shows how to set up and configure an `ILcdAction` to load `ILcdModel` data from an external source. The `OpenSupport` class offers the most important functionality for this. It wires a composite model decoder with a customizable GUI support class.

The `openSource` method of the support class finds the most appropriate model decoder and decodes the given source into an `ILcdModel`.

The `GXYOpenSupport` and `LspOpenSupport` classes further build upon this, creating a layer for the decoded model and adding it to a view. This is shown in Program 229.

When adding layers to a GXY view for client side use, you will likely want to wrap the layers with an asynchronous layer. That is not taken care of by the layer factories discovered using `TLcdServiceLoader` as it is undesired for server-side views. `GXYOpenSupport` decorates layers with asynchronous behavior using `AsynchronousLayerFactory`.

```

1 fLayerFactory = new TLcdCompositeGXYLayerFactory(aLayerFactories);
2 ILcdGXYLayer layer = new AsynchronousLayerFactory(fLayerFactory).createGXYLayer(aEvent
   .getModel());
3 if (layer == null) {
4     noLayerFactory(getParentComponent(), aEvent.getModel());
5 } else {
6     GXYLayerUtil.addGXYLayer(fView, layer);
7     if (layer.isVisible()) {
8         GXYLayerUtil.fitGXYLayer(fView, layer);
9     }
10 }
```

Program 229 - Adding a layer for a decoded model
(from samples/common/formatsupport/GXYOpenSupport)

Three clients of this support class link the data loading functionality with the functionality for data selection by the application user:

OpenAction presents the user with a file chooser dialog

OpenURLAction presents the user with a dialog box that asks for a URL

OpenTransferHandler allows dragging files or URLs onto the view

This is illustrated in the decoder sample.

47.4 Using other format services

The format functionality in LuciadLightspeed is not restricted to model decoders and layer factories: other format services can be registered with the service registry as well. One example is the `ILcdModelMeasureProviderFactory`, which is used in `samples.util.MouseLocationComponent` for reading out measurements under the current mouse location. See the samples for an illustration of the readout functionality.

47.5 How to plug in your own data format

To add your own model decoder or layer factory to the service look-up mechanism, simply annotate it with `@LcdService`. If your class implements multiple interfaces, make sure that you also specify the interface you want to include as a service, for example, `ILcdModelDecoder.class`. Optionally, you can specify a priority for the service. The default priority ensures that your own service takes precedence over the LuciadLightspeed services. See the API reference documentation of `LcdService` for a list of available priorities.

Program 230 shows how to annotate a layer factory with low priority.

```
1 @LcdService(service = ILcdGXYLayerFactory.class, priority = LcdService.LOW_PRIORITY)
2 public class Custom1LayerFactory implements ILcdGXYLayerFactory {
```

Program 230 - Annotating a layer factory as a service
 (from `samples/gxy/decoder/custom1/Custom1LayerFactory`)

During compilation, the `TLcdServiceAnnotationProcessor` makes sure that the annotation is translated into a service entry in the META-INF directory. The `ServiceRegistry` look-up mechanism queries this directory at run-time to provide the requested format services. This is illustrated in Figure 153.

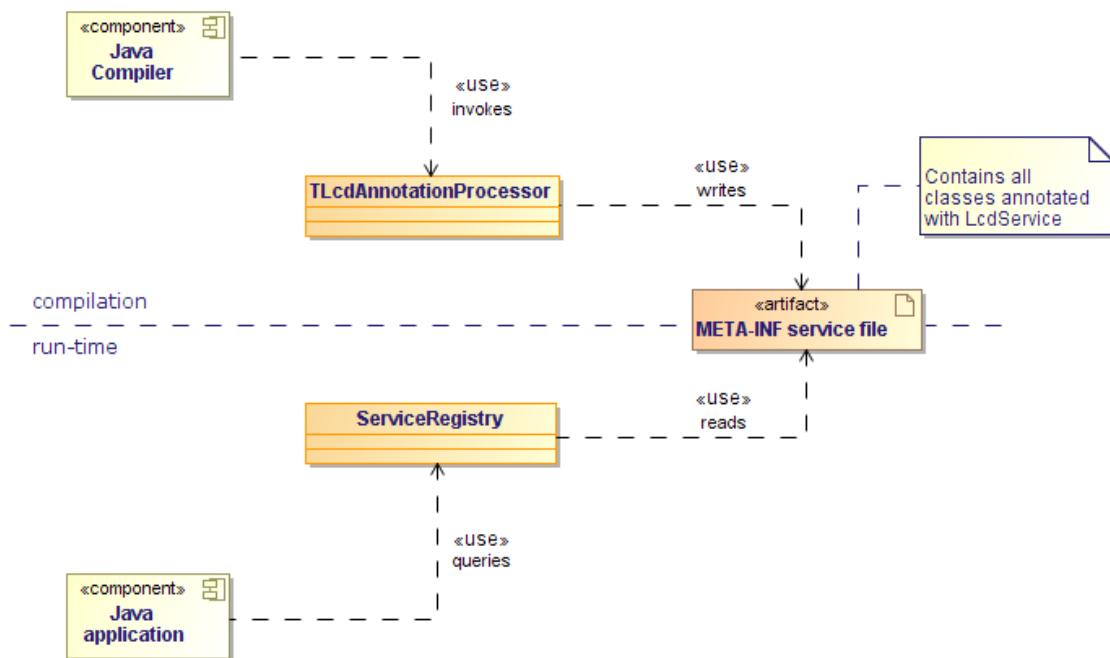


Figure 153 - Service look-up mechanism



Note that annotation processing should be enabled when compiling and that the jar files containing the annotation processor (`lcd_annotations*`) should be in the compiler classpath. All major IDE implementations support annotation processing. Also note that the NetBeans IDE contains a known bug causing annotation processing not to work well in combination with the 'Compile on Save' option. A workaround is to disable this option.

CHAPTER 48

Working with ISO metadata

This chapter covers the implementation of two standards on metadata published by the International Organization for Standardization¹, ISO 19115 and ISO 19139. The contents of the standards are discussed very briefly, the reader is assumed to have knowledge of the standard or at least have a copy at hand. [Section 48.2](#) explains how the ISO 19115 UML diagrams were converted into LuciadLightspeed interfaces and classes. [Section 48.3](#) explains how an ISO 19115 meta data can be encoded and decoded based on ISO 19139.

See `com.luciad.format.metadata.model` and `com.luciad.format.metadata.xml` in the LuciadLightspeed reference manual.

48.1 What is metadata?

Metadata is extra information on data, which is not part of the data itself. An ancient form of such metadata is the legend on a paper map. While it is not part of the data (the map itself), it can tell you more about the data, for example the scale of the map, or who was responsible for the creation of the map. Nowadays geographical information is stored in digital form and metadata can be stored next to the actual data, avoiding access to the data sets themselves.

As both the capabilities of producers and consumers of geographic information expand, there is an increasing need for geographic data exchange. But consumers are not interested in the bulk of available data, they require data pertaining to a certain domain, covering their area of interest, in a limited set of formats they can handle. Metadata can help them:

- Locate data they are interested in
- Manage large data sets by archiving the data not solely based on its geographic extent.
- Evaluate available data
- Publish data to interested parties.

As the number of producers grows, the need for a standard on metadata becomes apparent, since both producers and consumers need to be able to communicate on the data they wish to exchange. To this end the ISO 19115 and ISO 19139 standards are conceived. While the 19115 standard provides a conceptual schema on metadata presented as UML diagrams, the ISO 19139 standard describes an XML implementation of that schema.

¹ISO, International Organization for Standardization, www.iso.org

48.2 ISO 19115: metadata concepts

48.2.1 Standard specifications

Geographic data can be archived using different characteristics, of which the most obvious one is the location and extent of the data. The ISO 19115 standard covers a comprehensive set of more than 300 characteristics, bundled from a large number of disciplines. These characteristics are divided into following categories:

- Extent: Where is the data located? This could be a geographical location or a temporal location as some data may only be valid for a certain period in time.
- Constraint: What restrictions apply to the data: when can it be used, what licenses are required? Is the data sensitive and can it only be used by persons who have a security clearance?
- Data quality: What is the quality of the data? Does it comply with a given standard? What process was used to create the data?
- maintenance: Where and how to get updates of the data? When will the next update be available?
- Spatial representation and reference system: what locations on earth does the data represent? What is the geographic reference of the data?
- Content: What data is linked to the dataset? Does it contain features, for example a DBF file for a SHP file.
- Distribution: Where can I get the data or information on the data? Who should I contact and how? In what formats is it available?
- Portrayal catalog: What portrayal catalog is used to display the data?
- Application schema information: what application schema was used to build the data set?
- Identification: What topic is the data about? Where does it come from? Where can you get it? Who is responsible for it? What should it be used for? Is it part of a larger data set? What language is it expressed in? This information should enable you to uniquely identify the data.

It is clear that not all of this information is available for every dataset. Moreover, it might not be necessary for every dataset to keep or maintain all of this information. The standard does not require to provide all of these categories for a dataset. The standard marks every characteristic as either mandatory, optional or conditional. Mandatory characteristics are required, optional are not and conditional characteristics are only required when certain conditions hold. Of the above categories only the identification is mandatory when providing metadata.

While the standard does not require you to provide all characteristics of the data, it may well be possible that you may want to provide some which are not available in the standard. To that end the standard specifies rules to extend the set of characteristics. The result of such an extension is called a profile. The standard contains one category, extension, which enables the description of the profile inside the metadata. Theoretically this enables reading metadata from every profile. Note that every profile should contain a number of core elements defined in the standard, so that at least this core can be exchanged by everyone supporting the standard.

48.2.2 Domain model

The domain model for ISO 19115 can be found in `com.luciad.format.metadata.model` and its sub packages. It is generated based on the ISO 19139 schema documents using a small set of rules.

- Domain classes are put into a package based on the UML diagram in which they are defined in the specification. Each of the diagrams corresponds to one Java package, a subpackage of `com.luciad.format.metadata.model`, see table 6.

Category	subpackage
Identification information	identification
Constraint information	constraint
Data quality information	quality
Lineage information	lineage
Maintenance information	maintenance
Spatial representation information	spatial
Reference system information	reference
Content information	content
Portrayal catalog information	portrayal
Distribution information	distribution
Metadata extension information	extension
Application schema information	applicationschema
Metadata types, extent information	extent
Metadata types, citation information	citation

Table 6 - LuciadLightspeed subpackages for metadata categories

- Each schema type is converted to a Java domain class. For example, the `CI_Citation` types becomes `TLcdISO19115Citation`. These Java classes have properties which correspond to the attributes and elements listed in the schema type.
- For each property, a public getter and setter method is defined.
- Some properties can have multiple values defined. For these properties, a getter method is defined that returns a `List`. A setter method is not defined; modifications are done by modifying the list returned by calling the getter method.

The starting class from which all attributes can be reached is the class `TLcdISO19115Metadata` located in `com.luciad.format.metadata`.

Codelists

Codelists are used throughout the standard to define a specified set of values which can be extended during runtime. Where the enumeration pattern does not allow you to create new instances the code list will. Code lists should be used when reasonable values, but not all values of an attribute are known. Instead of having to list all possible values, including the exotic, once-in-a-lifetime values, it suffices to list a set of reasonable values which cover almost every occurrence.

Each code list is represented by a domain class that extends from `TLcdISO19115Code`. The domain class defines public constants for all the code values listed in the standard.

Generic access

All domain classes implement `ILcdDataObject`². This enables generic access to the content of every class regardless of the location in the metadata hierarchy. The `TLcdISO19115DataTypes` class provides access to the ISO 19115 data model. Note that this data model is an anonymous model that groups the data models provided by `TLcdGCODataTypes`, `TLcdGMDDataTypes`, `TLcdGSRDataTypes`, `TLcdGSSDataTypes` and `TLcdGTSDataTypes`. These latter data models represent the types defined in the different xml schema components (`gco.xsd`, `gmd.xsd`, and so on) that make up ISO 19115.

Dealing with property types

The ISO 19139 standard specifies that properties of a class have to be encoded using XML class property types. In simple terms this means that when a class A refers to a class B, a new association class called `B_PropertyType` is introduced to implement the link between A and B. As in most cases these association classes carry no useful information, they have been suppressed from the domain object API. This makes the model far more easier to use. Take for example the `TLcdISO19115Metadata` class. This class has an association to the `TLcdISO19115Distribution` class. Program 231 shows how this association is immediately accessible on the domain object class.

```

1 public class TLcdISO19115Metadata extends ... {
2 ...
3     public TLcdISO19115Distribution getDistributionInfo() {
4         return ...;
5     }
6
7     public void setDistributionInfo(TLcdISO19115Distribution aValue) {
8         ...;
9     }
10    ...
11 }
```

Program 231 - Representation of associations

Because association classes can carry useful information, such as for example a nil reason, they cannot be removed completely. They are merely suppressed from the public accessors provided by the domain classes. However, they are accessible using the `ILcdDataObject` API. Program 232 shows how for example the nil reason of a distribution info can be set. First, the value of the distribution info property is retrieved. If this value is null, a new property is created and assigned to the `TLcdISO19115Metadata` object. Finally, the nil reason of the property is set.

```

1 public void setDistributionNilReason(TLcdISO19115Metadata object, String nilReason) {
2     TLcdISO19118Property<?> property = (TLcdISO19118Property<?>) object.getValue(
3         DISTRIBUTION_INFO_PROPERTY);
4     if (property == null) {
5         property = (TLcdISO19118Property<?>) DISTRIBUTION_INFO_PROPERTY.getType().newInstance();
6         object.setValue(DISTRIBUTION_INFO_PROPERTY, property);
7     }
8     property.setNilReason(nilReason);
}
```

Program 232 - Accessing nil reason

48.2.3 Validation

The implementing classes do not contain validation code. The standard defines which attributes are mandatory, optional and conditional. For some attributes specific restrictions apply, for ex-

²See Chapter 10 for a detailed discussion on generic access and data models.

ample a latitude should be a value between -90.0 and 90.0 . Mandatory elements can be recognized as they are passed in the constructor. However, on construction it is allowed to pass null values. When an element is multivalued, an array of objects of the correct type needs to be passed. There is no check whether the array contains null values. Since the implementation is Vector based, null values are allowed, though it is not advisable to pass them. In some cases where the conditions are simple and exclusive, additional constructors are provided with conditional parameters.

As there is no validation the standard is not imposed when creating metadata objects. This facilitates creation of new metadata objects.

48.2.4 Visualization

The class `MetadataTree` in the samples package `samples.metadata.util` can be used to visualize the contents of a `TLcdISO19115Metadata`. Figure 154 displays what the contents of a metadata object might look like.

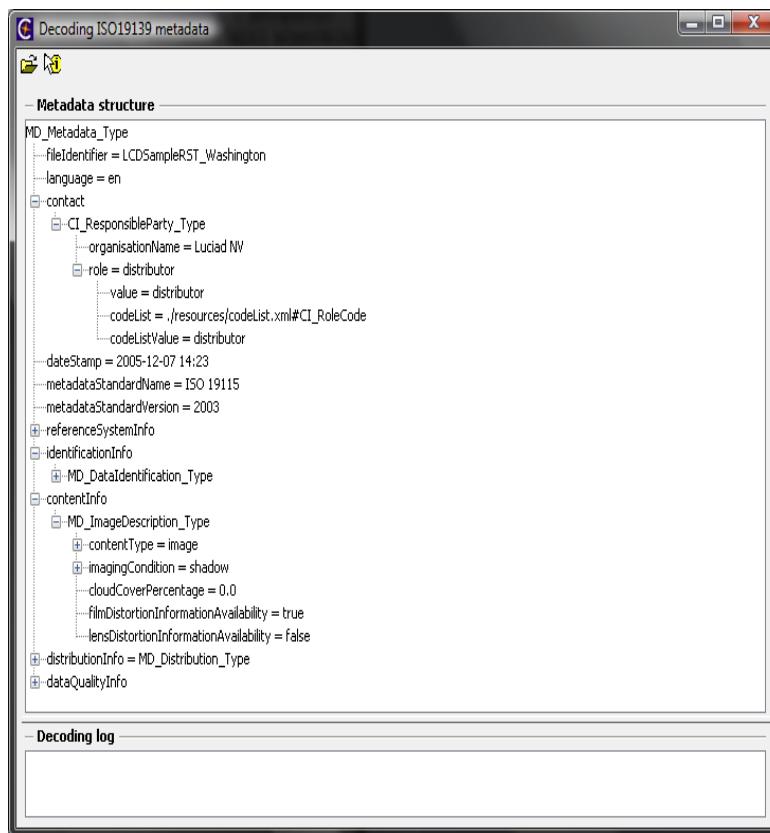


Figure 154 - Metadata visualized using a `MetadataTree`

48.3 ISO 19139: an XML implementation

48.3.1 Introduction

The ISO 19139 standard provides a universal implementation of ISO 19115 through an XML Schema encoding. This encoding conforms to the ISO 19118 standard, which defines a set of encoding rules for transforming a UML conceptual schema from the ISO 19100 series of documents into an XML schema.

48.3.2 Decoding metadata

LuciadLightspeed provides a decoder and encoder to decode and encode metadata in the ISO 19139 format in the package `com.luciad.format.metadata.xml`. Both the decoder and the encoder are implemented using the LuciadLightspeed XML framework (see [Chapter 52](#) for a more detailed explanation).

The primary class to be used for decoding ISO 19139 data is `TLcdISO19139MetadataDecoder`. The following code sample illustrates how to decode a ISO 19139 data source.

```
1 TLcdISO19139MetadataDecoder decoder = new TLcdISO19139MetadataDecoder();
2 TLcdISO19115Metadata metadata = decoder.decode("Data/metadata/iso19139/wash_spot_small.xml");
```

Program 233 - Decoding metadata

Note that this decoder does not implement `ILcdModelDecoder`, as the decoded objects do not implement `ILcdModel`. You can use `decodeMetadata(String)` to decode a data source containing a `<MD_Metadata>` element as root element; it will return a `TLcdISO19115Metadata` object. For any other data source, use `decodeObject(String)`, which returns an instance of the metadata domain model class that corresponds to the XML root element of the data source.

Because the decoded metadata does not implement `ILcdModel`, it cannot be displayed directly on a view. It is however possible to extract information that can be viewed from the metadata. The sample `samples.metadata.gazetteer` decodes metadata files containing the bounds and source name of geographic data files. The bounds are displayed on a map, enabling the user to see for which locations data is available. For a particular bounds, the user can request the metadata and/or load the actual data.

48.3.3 Custom metadata extensions

The ISO193139 specification allows you to include custom metadata in ISO metadata files by extending ISO19139 XML types to custom XML types. The `TLcdISO19139MetadataDecoder` and `TLcdISO19139MetadataEncoder` also support these extensions.

XML documents that properly refer to extension schemas in their `xsi:schemaLocation` attribute are automatically decoded by the metadata decoder: the decoder automatically detects the extension schemas and configures itself accordingly.

If no schema references are present in the XML data, or if the schema references are incorrect, you can explicitly preconfigure the decoder and encoder via an extension data model, as shown in snippet [Program 234](#).

```
1 TLcdDataModelBuilder dataModelBuilder = new TLcdDataModelBuilder( "http://my-extension-
2 namespace.com" );
3 dataModelBuilder.addDependency( TLcdISO19115DataTypes.getDataModel() );
4 // Add any other relevant dependency here
5
6 TLcdXMLDataModelBuilder extensionDataModelBuilder = new TLcdXMLDataModelBuilder(
7     TLcdISO19115DataTypes.getDataModel() );
8 extensionDataModelBuilder.setEntityResolver( new TLcdOGCEntityResolver() );
9 extensionDataModelBuilder.buildDataModel( dataModelBuilder, "http://my-extension-namespace.com
10      ", "path-to-my-extension-xsd" );
11
12 TLcdDataModel extensionDataModel = dataModelBuilder.createDataModel();
13
14 TLcdISO19139MetadataDecoder metadataDecoder = new TLcdISO19139MetadataDecoder(
15     extensionDataModel );
```

Program 234 - Adding support for a custom XML schema to the metadata decoder

CHAPTER 49

GML format

49.1 Introduction to GML

Open Geospatial Consortium's (OGC) Geography Markup Language (GML) is an XML-based data format for describing, storing and exchanging geographical data. LuciadLightspeed provides support for several versions of the GML format. This chapter gives a short introduction to the GML format, and how to incorporate it in your LuciadLightspeed applications using the GML packages.

The GML packages allow you to:

- Decode GML data files and import them into a LuciadLightspeed-based domain model
- Encode LuciadLightspeed models to a GML data file
- Customize the GML decoders, as well as the encoders, to generate or use custom-defined domain objects

LuciadLightspeed provides support for GML 2.x and GML 3.x data. There are three implementations provided, each based on one specific GML version, as shown in table 7. Other 2.x and 3.x versions are very similar to one of these versions, and most data files based on other 2.x/3.x versions can be handled by one of these three implementations.

<i>GML version</i>	<i>Covered versions</i>	<i>LuciadLightspeed package</i>
GML 2.1.2.1	GML 2.x	com.luciad.format.gml2.*
GML 3.1.1	GML 3.0.x, 3.1.x	com.luciad.format.gml31.*
GML 3.2.1	GML 3.2.x	com.luciad.format.gml32.*

Table 7 - Supported GML versions with their corresponding LuciadLightspeed package

All specifications can be found on the OGC website: <http://www.opengeospatial.org/standards/>. The XML schemas can be found on the following location: <http://schemas.opengis.net/gml/>.

Section 49.2 briefly summarizes the essential steps to integrate GML data in your application. Section 49.3 gives a short overview of the GML data model, section 49.4 shows how the model is mapped on the LuciadLightspeed API. Section 49.5 continues with a description of the GML model decoders, followed by section 49.6, discussing the GML model encoders. Finally, 49.7 presents an overview of the limitations of the current implementation.

This chapter is not a course about GML or XML; it assumes you are already familiar with the basic concepts of these formats.

The GML package is built upon the LuciadLightspeed XML binding framework. If you want to customize the GML decoder or encoder and you are not familiar with the XML binding framework, it is advised to read appendix 52 first, before starting with this chapter.

The following packages also provide limited support for GML but are deprecated; it is advised not to use them anymore.

- com.luciad.format.gml
- com.luciad.format.gml3

49.2 Integrating GML data into your application

Usage of the GML packages is very similar to using other LuciadLightspeed packages that provide data formats. The basic steps to show GML data in your Java application, using LuciadLightspeed components, are:

- Setting up a *map*, for example, a `TLcdMapJPanel`, and integrating it in your application.
- Setting up a GML *model decoder*.
- *Decoding* a model from a GML file.
- Creating an appropriate *layer factory* that can handle GML models, for example, the class `GMLLayerFactory` provided with the sample code.
- *Creating a layer* for the decoded model.
- *Adding the layer* to the map.
- *Fitting* the map to the model.

Sample Program 235 gives an overview of these steps for GML 3.1 data; the sample can be easily adjusted for GML 2 or GML 3.2. by just replacing the model decoder that is created.

```

1 // Create a map view and add it to the application's container
2 TLcdMapJPanel fMapJPanel = new TLcdMapJPanel();
3 getContentPane().add(fMapJPanel);
4
5 // Set up a GML model decoder
6 TLcdGML31ModelDecoder modelDecoder = new TLcdGML31ModelDecoder();
7
8 try {
9     // Decode the GML data
10    ILcdModel model = modelDecoder.decode(my_gml_file);
11
12    // Create a layer
13    GMLLayerFactory layerFactory = new GMLLayerFactory();
14    ILcdGXYLayer layer = layerFactory.createGXYLayer( model );
15
16    // Add the layer to the map view
17    fMapJPanel.addGXYLayer( layer );
18
19    // Fit on the GML layer
20    TLcdGXYViewFitAction fitAction = new TLcdGXYViewFitAction(fMapJPanel);
21    fitAction.fitGXYLayer((ILcdGXYLayer) fMapJPanel.getLayer(fMapJPanel.layerCount() - 1),
22                           fMapJPanel);
22 }
23 catch (IOException e) {
24     System.out.println("Could not load model: " + e.getMessage());
25 }
```

Program 235 - Loading and displaying a GML file

49.3 The GML data model

The GML specifications contain an elaborate set of data types, grouped in a number of XML schemas. The three most important schema groups are:

- The *base schemas*, containing some general XML types, used frequently by other schemas;
- The *feature schema*, providing a framework for the creation of GML features and feature collections (base and feature schema are the same in case of GML 2);
- The *geometry schemas*, providing a set of geometries to describe features.

Besides these schemas, a number of other schemas are available in GML 3.x for describing temporal features, topology, coordinate reference systems, and so on.

Features and Feature Collections

The basic element of a GML dataset is the *feature*, which is the GML representation of a real world entity (for example, a road, a lake, a building).¹

Each feature is defined by:

- zero, one or more *geometric property(ies)*, describing the spatial extent of the feature, and
- zero, one or more *non-geometric properties*, describing all other characteristics of the feature.

GML offers a large set of geometric primitives, ranging from basic shapes such as points, polylines and polygons to complex structures such as [Bézier curves](#) and triangulated surfaces. GML also allows to define and use your own, custom-defined shapes.

Non-geometric properties are not defined by the GML standard, as they are specific to each application domain. These properties should be defined in a custom application schema that extends the GML schema.

Features can be grouped into *feature collections* comparable to the way the `ILcdModel` interface is used to group objects in LuciadLightspeed. Collections can also be nested themselves.

A GML dataset typically exists of a feature collection containing some features. Sample Program 236 summarizes the general structure of a GML document.

¹Do not confuse GML features with LuciadLightspeed features, provided by the `ILcdFeatured` interface. LuciadLightspeed features correspond to feature properties in GML. In this chapter, the word feature always refers to GML features, not to LuciadLightspeed features.

```

1  <gml:FeatureCollection>
2
3      <!-- the total bounds of the collection -->
4      <gml:boundedBy> .. </gml:boundedBy>
5
6      <gml:featureMember>
7          <myns:MyFeature>
8              <!-- the feature's bounds -->
9              <gml:boundedBy> ... <gml:boundedBy>
10
11             <!-- geometric properties -->
12             <myns:geomProperty>
13                 <gml:Polygon> ... </gml:Polygon>
14             </myns:geomProperty>
15             ...
16
17             <!-- non-geometric properties -->
18             <myns:customProperty>customValue</myns:customProperty>
19             ...
20         </myns:MyFeature>
21     </gml:featureMember>
22
23     ...
24
25 </gml:FeatureCollection>

```

Program 236 - Typical outline of a GML document

The Simple Features Profile (GML 3.1.1 only)

The Simple Features Profile was introduced as a subset of GML, to reduce the effort and time required for implementing and supporting GML in software applications. There are three different levels defined, the first one (Level 0) being the most restrictive, the third one (Level 2) the most flexible. These profiles mainly restrict:

- which geometric primitives are allowed to appear in data², and
- which non-geometric properties are allowed (in Level 0, for example, the properties may only be simple datatypes such as boolean, integer, double and a few others).

49.4 The LuciadLightspeed GML domain model

49.4.1 Domain model classes

LuciadLightspeed provides a dedicated domain model for each supported GML version (2, 3.1 and 3.2) in the respective packages `com.luciad.format.gml2.model`, `com.luciad.format.gml31.model` and `com.luciad.format.gml32.model`³. All these domain classes provide accessors (getters and setters) to easily access the data contained in the objects. In addition, they also implement the `ILcdDataObject` interface; thereby providing generic access⁴. Finally, they also implement `ILcdDeepCloneable`. For reasons of backwards compatibility, the domain classes for GML 2 and GML 3.1 also implement `ILcdSelfDescribedFeatured` and `ILcdFeatured`.

²The complete list of primitives supported by this profile, can be found in the *GML Simple Features Profile* specification, section 7.2, which can be downloaded at the Open Geospatial Consortium's website (<http://www.opengeospatial.org/standards/>).

³For GML 3.x, only the most important schema groups (see Section 49.3) are explicitly represented by domain classes; the other concepts (modeling temporal features, topologies, coordinate reference systems etc) are represented using a generic domain model based on `ILcdDataObject`.

⁴You can find a detailed description of generic access in Chapter 10

LuciadLightspeed provides direct access to the GML data model and types in the `TLcdGML2DataTypes`, `TLcdGML31DataTypes` and `TLcdGML32DataTypes` classes.

The domain models are generated from the GML XML schema documents. Care has been taken to make the domain classes as easy to use as possible. XML implementation details such as element names and schema types are hidden as much as possible. The exact mapping rules for mapping XML Schema types on `ILcdDataObject` are described in detail in the package documentation of the `com.luciad.format.xml.bind.schema.datamodel` package, which provides support for binding XML Schema-based XML documents to `ILcdDataObject` domain models.

Some GML types, such as feature collections and geometries, are mapped on domain objects implementing additional interfaces to make them easier to integrate with other components in LuciadLightspeed. The following paragraphs describe these additional mappings in more detail. More information can be found in the class documentation of the GML model decoders.

49.4.2 Feature collection integration

Feature collections in GML correspond to models in LuciadLightspeed. The following GML types are considered feature collections and are thus mapped on domain objects implementing LuciadLightspeed model interfaces:

- `AbstractFeatureCollectionType` and extensions
- extensions of `AbstractFeatureType`: only if they are allowed to have child elements extending from `AbstractFeatureMemberType` (GML 3.x) or if they have child elements with name 'member' (GML 3.1).

Each of the domain objects for these types implements the following model interfaces:

- `ILcdModel`: contains as elements all GML domain objects which are directly contained (through a feature member) in the parent collection, of which the type extends from `AbstractFeatureType`, and which are not feature collections themselves.
- `ILcd2DBoundsIndexedModel`: the `applyOnInteract` methods only visit those elements of the model which have non-null bounds, that is, the GML features that have geometric content.
- `ILcdModelTreeNode`: contains as submodels all feature collections which are directly contained (through a feature member) in the parent collection.

The model interfaces provided by a feature collection are all non-editable, that is, elements cannot be added to or removed from the models.

All GML models have as model descriptor a `TLcdGML2ModelDescriptor`, `TLcdGML31ModelDescriptor`, or `TLcdGML32ModelDescriptor`.

49.4.3 Feature integration

Features in GML correspond to model elements in LuciadLightspeed. All GML types extending from `AbstractFeatureType` are considered as features, except feature collections (see Section 49.4.2). All relevant contents of a GML feature can be retrieved through the following interfaces:

- `ILcdDataObject`: all properties of a feature, both geometric and non-geometric, are accessible through the `ILcdDataObject` interface.

- `ILcdBounded`: returns as bounds the union of all geometries which are part of the feature.
- `ILcdShapeList`: contains as shapes the set of geometries which are part of the feature. The default implementation for `AbstractFeatureType` recursively searches in all the properties of the feature, and collects all properties implementing `ILcdShape`.

The `ILcdDataObject` interface is editable, as with all domain objects.

49.4.4 Geometry integration

Geometries in GML correspond to shapes in LuciadLightspeed. All types extending from one of the following GML types are considered as geometries, and are mapped on domain objects implementing an `ILcdShape` interface:

<i>GML type</i>	<i>LuciadLightspeed interface</i>
<code>AbstractGeometryType</code>	<code>ILcdShape</code>
<code>AbstractCurveSegmentType</code>	<code>LcdCurve</code>
<code>AbstractRingType</code>	<code>LcdRing</code>
<code>AbstractSurfacePatchType</code>	<code>ILcdShape</code>
<code>CoordinatesType</code>	<code>ILcdPointList</code>
<code>DirectPositionType</code>	<code>ILcdPoint</code>
<code>DirectPositionListType</code>	<code>ILcdPointList</code>
<code>EnvelopeType</code>	<code>ILcdBounds</code>

Table 8 - Mapping of GML geometry types on LuciadLightspeed interfaces

Refer to the documentation's mapping table of the GML mapping library classes for a detailed overview of how other geometries extending from these types are mapped on `ILcdShape` interfaces.

49.4.5 Other integrations

Table 9 gives an overview of other GML types which are mapped on additional LuciadLightspeed interfaces.

<i>GML type</i>	<i>LuciadLightspeed interface</i>
<code>MeasureType</code>	<code>ILcdISO19103Measure</code>

Table 9 - Mapping of GML miscellaneous types on LuciadLightspeed interfaces

49.5 GML model decoders

Table 10 gives an overview of the GML model decoders provided by the LuciadLightspeed API and the GML versions they support.

<i>Model decoder</i>	<i>Supported GML versions</i>
TLcdGML2ModelDecoder	2.x
TLcdGML31ModelDecoder	3.0.x, 3.1.x
TLcdGML32ModelDecoder	3.2.x
TLcdGMLModelDecoder	2.x, 3.0.x, 3.1.x, 3.2.x

Table 10 - Overview of GML model decoders

The `TLcdGMLModelDecoder` is provided as a convenience model decoder, which is capable of handling all GML formats. Its implementation analyzes the header of the documents to be decoded to detect which GML version they are, and then delegates the actual decoding to one of the other GML model decoders.



The `TLcdGMLModelDecoder` and `TLcdGML3ModelDecoder` are deprecated and should no longer be used.

49.5.1 XML integration

All non-deprecated GML decoders share a similar package structure, with similar classes (described here for GML 3.2):

- `TLcdGML32ModelDecoder`: model decoder, can decode GML 3.2 data
- `TLcdGML32ModelEncoder`: model encoder, can encode GML 3.2 data and export other models to GML 3.2

The `TLcdGML32ModelDecoder` and `TLcdGML32ModelEncoder` classes can be used as a standalone, out-of-the-box model decoder and encoder for decoding and encoding GML 3.2 data.

The GML32 data model, accessible on the `TLcdGML32DataTypes` class, can be used to configure to a `TLcdXMLSchemaBasedDecoder` or `TLcdXMLSchemaBasedEncoder` for use with GML data. This is typically done by other XML-based formats that embed GML in them. The chapter on the XML binding framework provides more details on how to do this.

49.6 GML model encoders

Table 11 gives an overview of the GML model encoders provided by the LuciadLightspeed API and which GML versions they support.

<i>Model encoder</i>	<i>Supported GML versions</i>
<code>com.luciad.format.gml2.TLcdGML2ModelEncoder</code>	2.x
<code>com.luciad.format.gml31.TLcdGML31ModelEncoder</code>	3.0.x, 3.1.x
<code>com.luciad.format.gml32.TLcdGML32ModelEncoder</code>	3.2.x

Table 11 - Overview of GML model encoders

The `TLcdGMLModelEncoder`, `TLcdGML2ModelEncoder` and `TLcdGML3ModelDecoder` are deprecated and should no longer be used.

GML model encoders can operate in two modes: save or export. Saving can only be done for GML data having the same version as the encoder; for example, the `TLcdGML32ModelEncoder`

can only save data which was decoded by the `TLcdGML32ModelDecoder`. For all other data, export mode is used.

In saving mode, the data will be written in exactly the same way as it was read. The main application schema describing the data is copied next to the data, by default.

In export mode, the encoder automatically generates an application schema for the data being exported, based on its `ILcdDataObject` and `ILcdShape` structure and contents. This generated application schema is written next to the data file.

49.7 Limitations

GML data are currently subject to the following requirements and limitations:

Decoding

- GML documents must be compliant with GML 2.0 or higher (up to 3.2.1).
- Only the GML types listed in the mapping tables are mapped on specific LuciadLightspeed interfaces. Although other types could also be mapped on more specific interfaces than `ILcdDataObject` (for example, the GML `EllipsoidType` could be mapped on `ILcdEllipsoid`), this is currently not supported.
- A valid XML application schema must be available for each GML document.
- GML allows individual geometries to have their own, local coordinate reference system, different from their parent's reference system. LuciadLightspeed `ILcdModel` objects can only have one model reference (all elements in an `ILcdModel` should be defined using the same reference system). The first reference system that is encountered in a GML document (typically the reference system of the feature collection's envelope) is used as the model reference for the `ILcdModel` that is created. All geometries with local reference system different from the `ILcdModel`, will be transformed to the `ILcdModel`'s reference system.
- XLink references are currently not resolved yet.

Encoding

- Data cannot be encoded yet with XLINK references.

CHAPTER 50

Using OGC Filters

50.1 Introduction to OGC filters

For many applications, you need to identify a particular subset of a data set by specifying a number of conditions, so that you can take further action on that subset only. LuciadLightspeed allows you to use OGC Filters to identify that data subset.

An *OGC Filter* is a construct used to constrain the property values of an object type for the purpose of identifying a subset of object instances. It is based on the *OpenGIS Filter Encoding Implementation specification* document (Geospatial Consortium documents 02-059, 04-095 and 09-026). The document describes a system-neutral representation of a query predicate, encoded in XML.

In addition, the specification document defines a set of *OGC expressions* that can also be used for expressing a computed property of a given object, which is a parametric value that depends on the given object.

A large class of OpenGIS web-based services require the ability to express filters in XML. The Web Feature Service (WFS) uses the OGC filters in `GetFeature` and `Transaction` operations to define query constraints. The Styled Layer Descriptor (SLD) specification uses OGC filters to condition the application of styling rules. It also uses OGC expressions to define variable parameters of styling rules. You can use an OGC expression to define how the stroke color depends on the object that is being painted, for example.

50.2 The OGC Filter API in LuciadLightspeed

The LuciadLightspeed OGC Filter API provides a Java object representation of OGC filters and expressions independently from the XML encoding. This representation is called the OGC Filter model and is described in [Section 50.4](#). The OGC Filter model offers the following advantages:

- It is independent of any version of the OpenGIS Filter Encoding Implementation specification. It can be obtained by decoding documents conforming to several versions of the specification. [Section 50.4](#) also details how filters can be encoded according several versions of the specification. Moreover, this abstraction layer can help to minimize the impact of the evolution of the specification.
- It is easier to build and manipulate programmatically than an XML document. It can be used for the creation and the edition of conforming OGC Filter XML documents.

- It is decoupled from any of its possible usages. It can then be evaluated in different ways under different circumstances. But it could also be translated into any other query language like SQL or XQuery in order to be used for querying an external datastore.

For the evaluation of an OGC Filter model, the LuciadLightspeed OGC Filter API offers a configurable and extensible evaluator that supports most of the constructs of the OGC Filter elements.

The OGC Filter evaluator allows you to:

- Build an `ILcdFilter` that can be used for accepting or rejecting a given object. This `ILcdFilter` can then be applied to a `TLcdGXYLayer` for example.
- Build an `ILcdPropertyRetriever` than can be used for computing a property of a given object. You can use this `ILcdPropertyRetriever` for building SLD symbolizer painters, for example. See [Chapter 51](#) for more information.

For a better understanding of the following sections, it is recommended to read the OpenGIS Filter Encoding Implementation specification document first.

50.3 Quick start

This quick start section offers examples of the most common use cases for working with filters. For more detail about the concepts and API, see [Section 50.4](#).

50.3.1 Creating or decoding a filter

Creating a filter You can create filters using the classes in the `com.luciad.ogc.filter.model` package. For example:

```

1 //use a static import to have easy access to the static methods from that class
2 import static com.luciad.ogc.filter.model.TLcdOGCFilterFactory.*;
3
4 ILcdOGCCondition condition = eq(property("roadType"), literal("highWay"));
5 TLcdOGCFilter filter = new TLcdOGCFilter(condition);

```

Program 237 - Creating a filter through the API

This results in a `TLcdOGCFilter`, which is the Java representation of an OGC filter.

Decoding a filter Often filters are encoded in the XML format. For decoding purposes, you can convert the XML to an `TLcdOGCFilter` using the `TLcdOGCFilterDecoder`.

```

1 TLcdOGCFilterDecoder decoder = new TLcdOGCFilterDecoder();
2 String filterSourceName = ...;//path to an xml file containing the filter
3 TLcdOGCFilter filter = (TLcdOGCFilter)decoder.decode(filterSourceName);

```

Program 238 - Decoding a filter from an XML file

When the filter XML is in a `String` or an `InputStream` rather than a file, you can use the static factory methods on `TLcdOGCFilterDecoder`:

```

1 String filterAsXML = ...;
2 TLcdOGCFilter filter = TLcdOGCFilterDecoder.decodeFromString(filterAsXML);

```

Program 239 - Decoding a filter directly from a `String`

If the filter XML is not stored in a file, you can either decode directly from an `InputStream` or use the static utility method to decode directly from a `String`. For example

```
1 try(InputStream is = createInputStream()) {
2     TLcdOGCFilter filter = (TLcdOGCFilter) new TLcdOGCFilterDecoder().decode(is);
3 }
```

Program 240 - Decoding a filter directly from a `InputStream`

50.3.2 Evaluating a filter

The `TLcdOGCFilter` instance contains the definition of the filter only. It cannot be evaluated without extra information. During the evaluation of an OGC filter, the following functionality is needed:

- An `ILcdPropertyRetriever`: typical OGC filters compare the value of a property of a domain object with a certain value (for example `property(roadType) == "high-Way"`). The retrieval of the property value from the domain object is delegated to this interface.
- An `ILcdOGCFeatureIDRetriever`: if the OGC filter is based on the ID of the objects, the filter delegates the retrieval of the ID to this interface.
- A default `ILcdGeoReference`: this reference is used when the filter contains geometries which are defined without a georeference. The filter assumes that such geometries are defined in that default georeference.

By providing this information it is possible to convert the `TLcdOGCFilter` to an `ILcdFilter` instance that can be evaluated:

```
1 ILcdOGCCondition condition = ...;
2 TLcdOGCFilter ogcFilter = new TLcdOGCFilter(condition);
3
4 TLcdOGCFilterEvaluator evaluator = new TLcdOGCFilterEvaluator();
5 ILcdFilter filter = evaluator.buildFilter(ogcFilter, new TLcdOGCFilterContext());
6
7 Object toEvaluate = ...;
8 boolean accepted = filter.accept(toEvaluate);
```

Program 241 - Evaluating a filter

In the code above, default retrievers are used and no custom functions are supported. This works in most cases. If you don't need to configure the filter evaluator, you can convert an OGC filter directly into an `ILcdFilter` as follows:

```
1 ILcdOGCCondition condition = ...;
2 TLcdOGCFilter ogcFilter = new TLcdOGCFilter(condition);
3 ILcdFilter filter = ogcFilter.asPredicate(new TLcdOGCFilterContext());
4
5 Object toEvaluate = ...;
6 boolean accepted = filter.accept(toEvaluate);
```

Program 242 - Evaluating a filter without `TLcdOGCFilterEvaluator`

50.3.3 Supporting custom OGC functions

OGC filter expressions can contain a number of pre-defined operations or filter functions: comparison operations spatial operations, and so on. If you have a need for custom operations, you

can define your own filter functions in addition to the pre-defined OGC filter operations.

Supporting custom functions through the whole application

If you have custom functions, and you want to support them throughout the application, you can register them globally so that they become available to all `TLcdOGCFilterEvaluator` instances. This is done by calling the static `TLcdOGCFilterEvaluator.registerDefaultFunction` method. An example use case for this is a WFS server, where you want to support the custom function for all incoming requests.

```

1 //At the start of the application
2 TLcdXMLName customFunctionName = ...;
3 ILcdEvaluatorFunction customFunction = ...;
4 TLcdOGCFilterEvaluator.registerDefaultFunction(customFunctionName, customFunction);
5
6 /**
7 //Later on in the application
8 ILcdOGCCondition condition = ...;
9 TLcdOGCFilter ogcFilter = new TLcdOGCFilter(condition);
10
11 //This evaluator will support the custom function registered at the start of the application
12 TLcdOGCFilterEvaluator evaluator = new TLcdOGCFilterEvaluator();
13 ILcdFilter filter = evaluator.buildFilter(ogcFilter, new TLcdOGCFilterContext());
14
15 Object toEvaluate = ...;
16 boolean accepted = filter.accept(toEvaluate);

```

Program 243 - Adding support for a custom function for the whole application

Program 245 shows an example implementation of a custom `ILcdEvaluatorFunction`.

Supporting custom functions in a single evaluator

You can add support for custom functions by registering them on the evaluator instance using the `TLcdOGCFilterEvaluator.registerFunction` method. Note that the custom function will only be supported in that specific evaluator.

```

1 ILcdOGCCondition condition = ...;
2 TLcdOGCFilter ogcFilter = new TLcdOGCFilter(condition);
3
4 TLcdOGCFilterEvaluator evaluator = new TLcdOGCFilterEvaluator();
5 //register the custom function on the evaluator
6 TLcdXMLName customFunctionName = ...;
7 ILcdEvaluatorFunction customFunction = ...;
8 evaluator.registerFunction(customFunctionName, customFunction);
9
10 //The evaluation of the filter is identical as without custom functions
11 ILcdFilter filter = evaluator.buildFilter(ogcFilter, new TLcdOGCFilterContext());
12
13 Object toEvaluate = ...;
14 boolean accepted = filter.accept(toEvaluate);

```

Program 244 - Adding support for a custom function in a single evaluator

Program 245 shows an example implementation of a custom `ILcdEvaluatorFunction`.

Example of a custom function

Program 245 shows an example of a custom function implementation, in which a string is converted to upper case.

```

1  ILcdEvaluatorFunction toUpperCaseFunction = new ILcdEvaluatorFunction() {
2      @Override
3      public Object apply(Object[] aArguments,
4                          Object aCurrentObject,
5                          TLcdOGCFilterContext aOGCFilterContext) {
6          return aArguments.length == 1 && aArguments[0] != null ?
7              aArguments[0].toString().toUpperCase() : null;
8      }
9
10     @Override
11     public int getArgumentCount() {
12         return 1;
13     }
14 };

```

Program 245 - Registering a custom function
 (from samples/ogc/filter/xml/MainPanel)

50.3.4 Encoding a filter

TLcdOGCFilter instances can be converted back to their XML representation using the TLcdOGCFilterEncoder. For example:

```

1 TLcdOGCFilter filter = ...;
2 TLcdOGCFilterEncoder encoder = new TLcdOGCFilterEncoder();
3 String outputFile = "path/to/file.xml";
4 encoder.encode(filter, outputFile);

```

Program 246 - Encoding an OGC filter to a file

The TLcdOGCFilterEncoder class also allows to store the XML representation of the TLcdOGCFilter into a String or an OutputStream:

```

1 TLcdOGCFilter filter = ...;
2 try(OutputStream os = ...){
3     new TLcdOGCFilterEncoder().encode(filter, os);
4 }

```

Program 247 - Storing the XML representation of a TLcdOGCFilter into an OutputStream

```

1 TLcdOGCFilter filter = ...;
2 String xmlEncodedFilter = TLcdOGCFilterEncoder.encodeToString(filter);

```

Program 248 - Storing the XML representation of a TLcdOGCFilter into a String

50.4 The OGC Filter model

50.4.1 Overview

An OGC Filter is represented by a TLcdOGCFilter object, which contains a filter condition. The filter is intended to only accept the objects that meet this condition.

The filter conditions are based on filter expressions and they are built using conditional operators. The OGC Filter model has several kinds of conditional operators: comparison operators, logical operators, spatial operators, null check operator, and so on. Each of those conditions is represented by a class listed in Section 50.4.3.

The filter expressions are the basic constructs of a filter model. They can be either literals, object properties referenced by their name or an XPath expression, functions or the result of arithmetic operations. For those four kinds of filter expression, the OGC Filter model contains classes that are described in [Section 50.4.2](#).

The most commonly used conditions and expressions of the OGC filters can be constructed by using the static methods of the `TLcdOGCFilterFactory`.

50.4.2 Filter expressions

The filter expressions are the basic constructs of the filter conditions. They also allow to express computed properties, or parametric values, that depend on a given object.

Each XML element of the OpenGIS Filter Encoding Implementation specification that represents an expression is modeled by a class in the LuciadLightspeed OGC Filter model. This mapping is detailed in [Table 12](#).

The `ILcdOGCExpression` interface is the common marker for OGC Filter model classes that represent a filter expression.

XML element	Corresponding LuciadLightspeed class
<code><ogc:expression></code>	<code>ILcdOGCExpression</code>
<code><ogc:PropertyName></code>	<code>TLcdOGCPropertyName</code>
<code><ogc:Literal></code>	<code>TLcdOGCLiteral</code>
<code><ogc:Function></code>	<code>TLcdOGCFunction</code>
<code><ogc:Add></code>	<code>TLcdOGCBinaryOperator</code> with operation type equal to <code>TLcdOGCBinaryOperator.ADD</code>
<code><ogc:Mul></code>	<code>TLcdOGCBinaryOperator</code> with operation type equal to <code>TLcdOGCBinaryOperator.MULTIPLY</code>
<code><ogc:Sub></code>	<code>TLcdOGCBinaryOperator</code> with operation type equal to <code>TLcdOGCBinaryOperator.SUBSTRACT</code>
<code><ogc:Div></code>	<code>TLcdOGCBinaryOperator</code> with operation type equal to <code>TLcdOGCBinaryOperator.DIVIDE</code>

Table 12 - LuciadLightspeed classes for OGC Filter expression elements

Examples of filter expressions are shown in [Program 249](#).

```

1  /*
2   <ogc:PropertyName xmlns="http://someserver/myns">lastName</ogc:PropertyName>
3  */
4  TLcdOGCPropertyName propertyName = property("lastName", "http://someserver/myns");
5
6  /*
7   <ogc:PropertyName xmlns="http://someserver/myns">Person/lastName</ogc:PropertyName>
8  */
9  TLcdOGCXPath xpath;
10 xpath = new TLcdOGCXPath("Person/lastName", "http://someserver/myns");
11 propertyName = new TLcdOGCPropertyName(xpath);
12
13 /*
14  <ogc:Literal>John Smith</ogc:Literal>
15 */
16 TLcdOGCLiteral literal = literal("John Smith");
17
18 /*
19  <ogc:Literal>6000000</ogc:Literal>
20 */
21 literal = literal(6000000);
22
23 /*
24  <ogc:Function name="SIN" >
25   <ogc:Literal>3.14159265359</ogc:Literal>
26  </ogc:Function>
27 */
28 TLcdOGCFunction function = function("SIN", literal(Math.PI));
29
30 /*
31  <ogc:Add>
32   <ogc:PropertyName xmlns="http://someserver/myns">lane</ogc:PropertyName>
33   <ogc:Literal>-1</ogc:Literal>
34  </ogc:Add>
35 */
36 TLcdOGCBinaryOperator binaryOperator = add(property("lane", "http://someserver/myns"),
37   literal(-1));

```

Program 249 - Representation of OGC Filter expressions
 (from samples/ogc/filter/model/OGCFilterModelSample)

50.4.3 Filter conditions

The filter conditions are the filter constructs that return either true or false for a given object. They are used in `TLcdOGCFilter` for defining which object must be accepted or rejected.

Each XML element of the OpenGIS Filter Encoding Implementation specification that represents a condition is modeled by a class in the LuciadLightspeed OGC Filter model. This mapping is detailed in [Table 13](#).

The `ILcdOGCCondition` interface is the common marker for OGC Filter model classes that represent a filter condition.

XML element	Corresponding LuciadLightspeed class
<ogc:PropertyIsEqualTo>	TLcdOGCBinaryComparisonOperator with comparison type equal to TLcdOGCBinaryComparisonOperator.EQUAL
<ogc:PropertyIsNotEqualTo>	TLcdOGCBinaryComparisonOperator with comparison type equal to TLcdOGCBinaryComparisonOperator.NOT_EQUAL
<ogc:PropertyIsGreater Than>	TLcdOGCBinaryComparisonOperator with comparison type equal to TLcdOGCBinaryComparisonOperator.GREATER
<ogc:PropertyIsGreaterOrEqualThan>	TLcdOGCBinaryComparisonOperator with comparison type equal to TLcdOGCBinaryComparisonOperator.GREATER_OR_EQUAL
<ogc:PropertyIsLessThan>	TLcdOGCBinaryComparisonOperator with comparison type equal to TLcdOGCBinaryComparisonOperator.LESS
<ogc:PropertyIsLessOrEqualThan>	TLcdOGCBinaryComparisonOperator with comparison type equal to TLcdOGCBinaryComparisonOperator.LESS_OR_EQUAL
<ogc:PropertyIsBetween>	TLcdOGCIsBetweenOperator
<ogc:PropertyIsLike>	TLcdOGCIsLikeOperator
<ogc:PropertyIsNull>	TLcdOGCIsNullOperator
<ogc:BBOX>	TLcdOGCBoxOperator
<ogc:Equals>	TLcdOGCBinarySpatialOperator with spatial interaction type equal to TLcdOGCBinarySpatialOperator.EQUALS
<ogc:Disjoint>	TLcdOGCBinarySpatialOperator with spatial interaction type equal to TLcdOGCBinarySpatialOperator.DISJOINT
<ogc:Touches>	TLcdOGCBinarySpatialOperator with spatial interaction type equal to TLcdOGCBinarySpatialOperator.TOUCHES
<ogc:Within>	TLcdOGCBinarySpatialOperator with spatial interaction type equal to TLcdOGCBinarySpatialOperator.WITHIN
<ogc:Overlaps>	TLcdOGCBinarySpatialOperator with spatial interaction type equal to TLcdOGCBinarySpatialOperator.OVERLAPS
<ogc:Crosses>	TLcdOGCBinarySpatialOperator with spatial interaction type equal to TLcdOGCBinarySpatialOperator.CROSSES
<ogc:Intersects>	TLcdOGCBinarySpatialOperator with spatial interaction type equal to TLcdOGCBinarySpatialOperator.INTERSECTS
<ogc:Contains>	TLcdOGCBinarySpatialOperator with spatial interaction type equal to TLcdOGCBinarySpatialOperator.CONTAINS

<ogc:DWithin>	TLcdOGCDistanceBuffer with spatial test type equal to TLcdOGCDistanceBuffer.DWITHIN
<ogc:Beyond>	TLcdOGCDistanceBuffer with spatial test type equal to TLcdOGCDistanceBuffer.BEYOND
<ogc:And>	TLcdOGCBinaryLogicOperator with logic operation type equal to TLcdOGCBinaryLogicOperator.AND
<ogc:Or>	TLcdOGCBinaryLogicOperator with logic operation type equal to TLcdOGCBinaryLogicOperator.OR
<ogc:Not>	TLcdOGCNotOperator
<ogc:After>	TLcdOGCBinaryTemporalOperator with temporal interaction type equal to TLcdOGCBinaryTemporalOperator.AFTER
<ogc:Before>	TLcdOGCBinaryTemporalOperator with temporal interaction type equal to TLcdOGCBinaryTemporalOperator.BEFORE
<ogc:Begins>	TLcdOGCBinaryTemporalOperator with temporal interaction type equal to TLcdOGCBinaryTemporalOperator.BEGINS
<ogc:BeginedBy>	TLcdOGCBinaryTemporalOperator with temporal interaction type equal to TLcdOGCBinaryTemporalOperator.BEGUN_BY
<ogc:TContains>	TLcdOGCBinaryTemporalOperator with temporal interaction type equal to TLcdOGCBinaryTemporalOperator.T_CONTAINS
<ogc:During>	TLcdOGCBinaryTemporalOperator with temporal interaction type equal to TLcdOGCBinaryTemporalOperator.DURING
<ogc:EndedBy>	TLcdOGCBinaryTemporalOperator with temporal interaction type equal to TLcdOGCBinaryTemporalOperator.ENDED_BY
<ogc:Ends>	TLcdOGCBinaryTemporalOperator with temporal interaction type equal to TLcdOGCBinaryTemporalOperator.ENDS
<ogc:TEquals>	TLcdOGCBinaryTemporalOperator with temporal interaction type equal to TLcdOGCBinaryTemporalOperator.T_EQUALS
<ogc:Meets>	TLcdOGCBinaryTemporalOperator with temporal interaction type equal to TLcdOGCBinaryTemporalOperator.MEETS
<ogc:MetBy>	TLcdOGCBinaryTemporalOperator with temporal interaction type equal to TLcdOGCBinaryTemporalOperator.MET_BY
<ogc:TOverlaps>	TLcdOGCBinaryTemporalOperator with temporal interaction type equal to TLcdOGCBinaryTemporalOperator.T_OVERLAPS

<ogc:OverlappedBy>	TLcdOGCBinaryTemporalOperator with temporal interaction type equal to TLcdOGCBinaryTemporalOperator.OVERLAPPED_BY
<ogc:AnyInteracts>	TLcdOGCBinaryTemporalOperator with temporal interaction type equal to TLcdOGCBinaryTemporalOperator.ANY_INTERACTS
<ogc:rid	TLcdOGCResourceId with comparison type equal to TLcdOGCResourceId.RID_PROPERTY
<ogc:previousRid	TLcdOGCResourceId with comparison type equal to TLcdOGCResourceId.PREVIOUS_RID_PROPERTY
<ogc:version	TLcdOGCResourceId with comparison type equal to TLcdOGCResourceId.VERSION_PROPERTY
<ogc:startDate	TLcdOGCResourceId with comparison type equal to TLcdOGCResourceId.START_DATE_PROPERTY
<ogc:endDate	TLcdOGCResourceId with comparison type equal to TLcdOGCResourceId.END_DATE_PROPERTY

Table 13 - LuciadLightspeed classes for OGC Filter condition elements

Examples of filter conditions are shown in Program 250, Program 251 and Program 252.

```

1  /*
2   * <ogc:PropertyIsGreater Than>
3   *   <ogc:PropertyName xmlns="http://someserver/myns">elevation</ogc:PropertyName>
4   *     <ogc:Literal>0</ogc:Literal>
5   *   </ogc:PropertyIsGreater Than>
6   */
7   TLcdOGCBinaryComparisonOperator binaryComparisonOperator =
8     gt(property("elevation", "http://someserver/myns"), literal(0));
9
10 /*
11  * <ogc:PropertyIsLike wildCard="*" singleChar="?" escapeChar="\">
12  *   <ogc:PropertyName xmlns="http://someserver/myns">lastName</ogc:PropertyName>
13  *     <ogc:Literal>John*</ogc:Literal>
14  *   </ogc:PropertyIsLike>
15  */
16  TLcdOGCIsLikeOperator isLikeOperator =
17    like(property("lastName", "http://someserver/myns"), "John*");

```

Program 250 - Representation of OGC Filter comparison conditions
(from samples/ogc/filter/model/OGCFilterModelSample)

```

1  /*
2   * <ogc:BBOX>
3   *   <ogc:PropertyName>Geometry</ogc:PropertyName>
4   *   <gml:Envelope srsName="EPSG:4326">
5   *     <gml:lowerCorner>13.0983 31.5899</gml:lowerCorner>
6   *     <gml:upperCorner>35.5472 42.8143</gml:upperCorner>
7   *   </gml:Envelope>
8   * </ogc:BBOX>
9   */
10 TLcdOGCBBBoxOperator bboxOperator = new TLcdOGCBBBoxOperator(
11     property("Geometry", "http://www.opengis.net/ogc"),
12     new TLcdLonLatBounds(13.0983, 31.5899, (35.5472 - 13.0983), (42.8143 - 31.5899)),
13     new TLcdGeodeticReference(new TLcdGeodeticDatum()));
14
15 /*
16  * <ogc:Intersects>
17  *   <ogc:PropertyName>Geometry</ogc:PropertyName>
18  *   <gml:Envelope srsName="EPSG:4326">
19  *     <gml:lowerCorner>13.0983 31.5899</gml:lowerCorner>
20  *     <gml:upperCorner>35.5472 42.8143</gml:upperCorner>
21  *   </gml:Envelope>
22  * </ogc:Intersects>
23  */
24 TLcdOGCBinarySpatialOperator binarySpatialOperator = new TLcdOGCBinarySpatialOperator(
25     TLcdOGCBinarySpatialOperator.INTERSECTS,
26     property("Geometry", "http://www.opengis.net/ogc"),
27     new TLcdLonLatBounds(13.0983, 31.5899, (35.5472 - 13.0983), (42.8143 - 31.5899)),
28     new TLcdGeodeticReference(new TLcdGeodeticDatum()));
29

```

Program 251 - Representation of OGC Filter spatial conditions
 (from samples/ogc/filter/model/OGCFilterModelSample)

```

1 /*
2  * <ogc:And>
3  *   <ogc:Not>
4  *     <ogc:PropertyIsNull>
5  *       <ogc:PropertyName xmlns="http://someserver/myns">elevation</ogc:PropertyName>
6  *     </ogc:PropertyIsNull>
7  *   </ogc:Not>
8  *   <ogc:PropertyIsEqualTo>
9  *     <ogc:PropertyName xmlns="http://someserver/myns">elevation</ogc:PropertyName>
10    <ogc:Literal>0</ogc:Literal>
11  </ogc:PropertyIsEqualTo>
12 </ogc:And>
13 */
14 TLcdOGCBinaryLogicOperator binaryLogicOperator =
15     and(not(isNull(property("elevation", "http://someserver/myns"))),
16         eq(property("elevation", "http://someserver/myns"), literal(0)));
17

```

Program 252 - Representation of OGC Filter logic conditions
 (from samples/ogc/filter/model/OGCFilterModelSample)

50.5 Limitations

The TLcdOGCFilterEvaluator supports spatial operators with the following limitations:

- If the Advanced GIS Engine component is not available in your LuciadLightspeed installation, only the BBOX operator is supported.
- With the exception of the BBOX operator, the spatial evaluators do not perform georeference transformations. As a consequence, they cannot compare geometries with two distinct georeferences.

- The `Crosses` and `Overlaps` operators only work for 2D curves.

CHAPTER 51

Styling data with OGC SLD

LuciadLightspeed allows you to structure styling information according to OpenGIS standards:

- OGC 05-077r4, the OpenGIS Symbology Encoding Implementation Specification version 1.1.0 (SE 1.1.0)
- OGC 05-078r4, the OpenGIS Styled Layer Descriptor Profile of the Web Map Service Implementation Specification, versions 1.0.0 and 1.1.0

This chapter discusses how you can apply such structured styling information to an `ILcdModel` to create an `ILspLayer` or an `ILcdGXYLayer`.



Originally, styling was included in the OGC Styled Layer Descriptor (OGC SLD) Implementation Specification. Later on, the styling part was split off from OGC SLD, and renamed to Symbology Encoding (SE). The most recent version of the OGC SLD basically only describes how Symbology Encoding styles can be used in the context of a Web Map Service (WMS).

In the LuciadLightspeed product and its documentation, we will continue to refer to Symbology Encoding with the acronym SLD, as this is the original prefix used for all classes that now provide the Symbology Encoding support.

51.1 Using SLD styling and symbology encoding in LuciadLightspeed

The OpenGIS standard SLD, from here on referred to as *the standard*, allows you to define uniform styling information for geospatial data through symbology encoding. The standard provides a uniform way to store a comprehensive model for styling information in an XML format.

Although the standard was originally conceived to convey styling information to a Web Map Server, the styling model can also be applied to data in standalone applications. This chapter does not cover the whole SLD implementation. Applying SLD in a Web Map Server context is discussed in the OGC Web Server Suite developers' guide that comes with the server product LuciadFusion. See that guide to learn more about assigning style to a layer in a Web Map Server.

Symbology encoding is based on the paradigm that style and content should be separated. As LuciadLightspeed adheres to the same paradigm, it is particularly well-suited to implement SLD: it provides the `ILcdModel` interface as a container for content. You can use the `ILspStyler`, `ILcdGXYPainter` and `ILcdGXYLabelPainter` interfaces to apply style. The styling information is modeled in the package `com.luciad.ogc.sld.model` discussed in [Section 51.3](#).

Although the styling model may be constructed programmatically, it is usually decoded from XML files. The package `com.luciad.ogc.sld.xml` contains classes to decode and encode styling models from and to XML files, see [Section 51.4](#).

Once the styling model has been constructed, you can apply it to an `ILcdModel`. [Section 51.2](#) explains what is required to configure the layers in `com.luciad.ogc.sld.view` so that the `ILcdModel` is rendered according to the style defined in the styling model.

[Section 51.5](#) lists the limitations of the styler and painter implementation for the styling model and how these limitations may be overcome.



It is recommended to keep a copy of the standard specification document close by while you are reading this chapter.

51.2 Quick start: styling your layers with SLD

Styling a layer with SLD is a two-step process:

- Convert a SLD file to a Java domain object
- Apply that styling to the layer

This quick start section offers examples of the most common use cases for completing those two steps. For more detail about the concepts and API, see [Section 51.3](#).

51.2.1 Decoding an SLD file

To convert an SLD file, containing the XML encoded version of the SLD, to a Java domain object, you can use the `TLcdSLDFeatureTypeStyleDecoder.decode` method.

```
1 TLcdSLDFeatureTypeStyleDecoder decoder = new TLcdSLDFeatureTypeStyleDecoder();
2 String sldFile = "path/to/sld/file.sld";
3 TLcdSLDFeatureTypeStyle sldStyle = decoder.decodeFeatureTypeStyle(sldFile);
```

Program 253 - Decoding a SLD file

If the SLD is not stored in a file, you can either decode directly from an `InputStream` or use the static utility method to decode directly from a `String`. For example

```
1 try(InputStream is = createInputStream()) {
2     TLcdSLDFeatureTypeStyle sldStyle = new TLcdSLDFeatureTypeStyleDecoder().
3         decodeFeatureTypeStyle(is);
4 }
```

Program 254 - Decoding SLD from an InputStream

```
1 String sldAsString = ...;
2 TLcdSLDFeatureTypeStyle sldStyle = TLcdSLDFeatureTypeStyleDecoder.decodeFromString(sldAsString
) ;
```

Program 255 - Decoding SLD from a String

51.2.2 Creating a SLD-styled Lightspeed layer

You can use the `TLcdSLDFeatureTypeStyle` to style your layer. This is illustrated in the examples below for the different types of layers.

```

1 TLcdSLDFeatureTypeStyle sldStyle = ...;
2
3 ILspLayer layer = TLspShapeLayerBuilder.newBuilder()
4                               .model(model)
5                               .sldStyle(sldStyle)
6                               .build();

```

Program 256 - Creating an SLD-styled Lightspeed vector layer

```

1 TLcdSLDFeatureTypeStyle sldStyle = ...;
2
3 ILspLayer layer = TLspRasterLayerBuilder.newBuilder()
4                               .model(model)
5                               .sldStyle(sldStyle)
6                               .build();

```

Program 257 - Creating an SLD-styled Lightspeed raster layer

51.2.3 Updating the SLD style on a Lightspeed layer

If you have an existing Lightspeed layer and you want to switch to SLD styling, or update the current SLD style, you can use the `configureForSLDStyling` method:

```

1 TLspLayer vectorLayer = ...;
2
3 TLcdSLDFeatureTypeStyle styleToApply = ...;
4 vectorLayer.configureForSLDStyling(styleToApply);

```

Program 258 - Applying an SLD style to an existing Lightspeed vector layer

```

1 TLspRasterLayer rasterLayer = ...;
2
3 TLcdSLDFeatureTypeStyle styleToApply = ...;
4 rasterLayer.configureForSLDStyling(styleToApply);

```

Program 259 - Applying an SLD style to an existing Lightspeed raster layer

51.2.4 Creating an SLD-styled GXY layer

You can use `TLcdSLDFeatureTypeStyle` to style your GXY layer, as illustrated below:

```

1 TLcdSLDFeatureTypeStyle sldStyle = ...;
2
3 //This works for both vector and raster data
4 ILcdGXYLayer layer = new TLcdSLDGXYLayerFactory().createGXYLayer(model, Collections.
      singletonList(sldStyle));

```

Program 260 - Creating a SLD-styled GXY layer

51.2.5 Updating the SLD style on a GXY layer

If you have an existing GXY layer, and you want to switch to SLD styling, or update the current SLD style, you can use the static method `TLcdSLDGXYLayerFactory.configureForSLDStyling` method:

```

1 TLcdGXYLayer layer = ...;
2

```

```

3 TLcdSLDFeatureTypeStyle styleToApply = ...;
4 TLcdSLDGXYLayerFactory.configureForSLDStyling(styleToApply, layer);

```

Program 261 - Applying an SLD style to an existing GXY layer

51.3 The SLD styling model

51.3.1 Why provide a model implementation?

Although the styling aspects in SLD are numerous, the standard is far from comprehensive. Different fields of application require different representations, and these cannot all be mapped to the styling elements provided by the standard. The package `com.luciad.ogc.sld.model` contains the Java equivalent of the Schema files that comprise the SLD styling model. An extension of the SLD styling model would result in an equivalent extension of the Java model. An `ILcdGXYPainter` can then be created on top of the extended Java model, taking into account the new styling elements.

Having a model implementation also provides a way to customize styles at runtime. Applications can provide a GUI to customize the model implementation, so that you can avoid tedious editing of an XML file. Applying the new style to an `ILcdModel` in a GXY view then only requires creating a GXY painter based on the new styling model.

51.3.2 Model structure

As the styling model is completely described in the standard, this section only gives an overview of the elements and explains how they are converted to classes in LuciadLightspeed.

The SLD styling model contains three levels: *feature type style*, *rule* and *symbolizer*, each of which has its own function. The first two levels enable selection of the objects to which the style should be applied, while the third contains the actual style information.

The feature type style enables initial filtering on the object. For example, you would want to apply a 'blue' feature type style only to objects that represent hydrographic data.

The feature type style contains a number of rules. The rule level enables filtering at a more finely grained level, using filters as defined in the Filter Encoding Implementation Specification ¹. For details on the filters, refer to the standard and Chapter 50, which discusses the LuciadLightspeed implementation of this standard. Use of filters enables selection of objects on:

- Properties of the object
- The result of computations on properties of objects
- Location of the object
- And so on

Rules are only applied within a certain scale range, allowing you to switch styles when zooming in on a view. An example is a set of rules for a city where:

- On a very small scale, the city is rendered as a point.
- On a small scale, the city is rendered as a point combined with a label.
- On a large scale, the city is portrayed as a polygon which depicts the built-up area of the city.

¹OGC 04-095, OpenGIS Filter Encoding Implementation Specification

- On a very large scale, satellite imagery shows the details of the buildings of the city.

The standard also introduces an *else filter*. If an object is not accepted by the filters in all other rules, the style defined in the rule containing the else filter is applied to it.

Every rule contains a number of symbolizers, which contain the actual styling information. Five types of symbolizers are defined in the standard, each containing specific styling aspects:

- A point symbolizer
- A line symbolizer
- A polygon symbolizer
- A text symbolizer
- A raster symbolizer

Symbolizers can select the geometry they should be applied to, since an object may be represented by different geometries as demonstrated in the example above. If the selected geometry does not fit the symbolizer type a conversion should be applied to it.

The first three symbolizers should be considered as symbolizers for spatial dimensions. The polygon symbolizer is not restricted to defining styles for polygons, rather it defines styles for any object that covers an area. For LuciadLightspeed this includes objects which should be rendered as filled circles, arcbands, buffers, and so on. Analogously, when an object should be rendered as a circle which should not be filled, a line symbolizer style may be applied to it.

For details on the styling aspects of the symbolizers, refer to the standard and the API documentation.

51.3.3 Using the content

The strength of the styling model is that it enables the use of the content to customize the style, through the use of parameter values. A parameter value is a mix of plain text and expressions as defined in the Filter specification. [Program 262](#) shows an example: the value of the fill color for a polygon symbolizer is composed from the RGB values, extracting the value for the green channel from the object, while the red and blue channel are set to 0. Using this parameter value assumes that the object has the property `GreenValue` and that the property is stored as a hexadecimal value.

```

1 <PolygonSymbolizer>
2   <Fill>
3     <CssParameter name="fill">#00<ogc:PropertyName>GreenValue</ogc:PropertyName>00</
4       CssParameter>
5   </Fill>
</PolygonSymbolizer>
```

Program 262 - Using parameter value for channel selection

[Program 263](#) demonstrates how the width of a stroke is computed when rendering a highway. It takes into account the number of lanes, the width of a lane and a fixed scaling factor.

```

1 <LineSymbolizer>
2   <Geometry>
3     <ogc:PropertyName>CenterLine</ogc:PropertyName>
4   </Geometry>
5   <Stroke>
6     <CssParameter name="stroke-width">
7       <ogc:Multiply>
8         <ogc:PropertyName>NUMBER_LANES</ogc:PropertyName>
9         <ogc:Div>
10          <ogc:PropertyName>LANE_WIDTH_METERS</ogc:PropertyName>
11          <ogc:Literal>3</ogc:Literal>
12        </ogc:Div>
13      </ogc:Multiply>
14    </CssParameter>
15  </Stroke>
16</LineSymbolizer>

```

Program 263 - Computing the stroke width using parameter value

51.3.4 Model implementation

The standard comprises Schema files describing the styling model and how it should be stored in XML files. `com.luciad.ogc.sld.model` provides a Java translation of the Schema. As rule of thumb, a type in the Schema corresponds to one class. Exceptions to this rule are rare and due to one of these reasons:

- An attempt to take into account future changes or extensions to the SLD specification, or
- The fact that no Java class corresponded well to the semantics intended by the type

This relation facilitates the creation of the Java equivalent model for the styling information contained in XML. [Program 264](#) demonstrates this for the XML example in [Program 263](#): it sets which property of the object should be interpreted as the geometry of the object to be rendered, builds the stroke containing the expression to compute the width and constructs a line symbolizer containing both.

```

1 public ALcdSLDSymbolizer createLineSymbolizer() {
2
3   // indicate which property of the object should be interpreted as the geometry to render
4   TLcdOGCPropertyName geometry_property =
5     new TLcdOGCPropertyName( TLcdXMLName.getInstance("", "CenterLine") );
6   TLcdSLDGeometry geometry = new TLcdSLDGeometry( geometry_property );
7
8   // build the expression that computes the stroke width, based on the object.
9   ALcdOGCExpression stroke_width_expression = buildStrokeWidthExpression();
10  // embed the expression in the parameter value.
11  TLcdSLDParameterValue stroke_width_parameter = new TLcdSLDParameterValue(
12    stroke_width_expression );
13
14  // create the stroke and set the stroke width parameter.
15  TLcdSLDStroke stroke = new TLcdSLDStroke();
16  stroke.setCssParameter( TLcdSLDStroke.STROKE_WIDTH, stroke_width_parameter );
17
18  // create the symbolizer and assign the stroke and the geometry to it.
19  return new TLcdSLDLineSymbolizer( geometry, stroke );
}

```

Program 264 - Constructing a style programmatically

[Program 265](#) shows the Java equivalent of the expression to compute the stroke width. The expression is built from the inside out: first the arguments of the division are defined and a division expression is created, then the argument for the multiplication is defined and the multiplication

expression is returned.

```

1  public ALcdOGCExpression buildStrokeWidthExpression() {
2
3      // indicate which property should be interpreted as the first argument of the division
4      TLcdOGCPropertyName lane_width_property =
5          new TLcdOGCPropertyName( TLcdXMLName.getInstance("", "LANE_WIDTH_METERS") );
6      // the second argument to the division is a constant. It should not be interpreted.
7      TLcdOGCLiteral scale_factor = new TLcdOGCLiteral( new Integer(3) );
8
9      // construct the division expression
10     ALcdOGCExpression division_expression = new TLcdOGCBinaryOperator( TLcdOGCBinaryOperator.
11         DIVIDE,
12             lane_width_property,
13             scale_factor );
14
15     // indicate which property should be interpreted as the first argument of the
16     // multiplication
17     TLcdOGCPropertyName number_of_lanes_property =
18         new TLcdOGCPropertyName( TLcdXMLName.getInstance("", "NUMBER_LANES") );
19
20     // return the multiplication expression
21     return new TLcdOGCBinaryOperator( TLcdOGCBinaryOperator.MULTIPLY,
22         division_expression,
23         number_of_lanes_property );
24 }
```

Program 265 - Computing the stroke width using parameter value programmatically

51.4 Decoding and encoding SLD styling models

While providing an implementation of the SLD model in Java is useful, SLD styles are typically created from XML files. The package `com.luciad.ogc.sld.xml` contains classes to assist with the decoding of styling models from XML files and the encoding of styling models to XML files: The package contains a decoder, `TLcdSLDFeatureTypeStyleDecoder`, which decodes XML files with `FeatureTypeStyle` as the top element to a `TLcdSLDFeatureTypeStyle` object. Analogously, the `TLcdSLDFeatureTypeStyleEncoder` encodes a `TLcdSLDFeatureTypeStyle` object to an XML file.

51.5 SLD styling limitations

You need to take into account a number of limitations when you use SLD styling in LuciadLightspeed.

51.5.1 Unsupported SLD elements

The classes in `com.luciad.ogc.sld.model` support the complete styling model. The `TLcdSLDFeatureTypeStyleDecoder` decodes all XML elements and attributes in a `FeaturedTypeStyle` element. The implementation of the conversion to painters in `com.luciad.ogc.sld.view.gxy` however does not encompass the complete styling model. The following items are not supported:

- A combination of `CssParameter` and a `GraphicFill` or `GraphicStroke` in a `Fill` or in a `Stroke`. It is not clear from the standard how this should be interpreted, but an interpretation can be implemented in an `ILcdSLDGraphicsProvider`.
- The contrast enhancement features `Normalize` and `Histogram` in `RasterSymbolizer`.

- OverlapBehavior in RasterSymbolizer. Support for OverlapBehavior would require a new implementation of `ILcdGXYLayer`, since all objects in `TLcdGXYLayer` are painted one after another, not taking into account relations between the objects.
- SE 1.1 specifies a color replacement feature for external graphics. This element will be ignored in LuciadLightspeed. In most cases, it is better to use a mark graphic with a stroke and fill. In addition, it is unclear whether this feature will still be available in the next version of the standard.
- SE 1.1 also allows rules to be retrieved through a link to an external resource. These links will not be resolved automatically, but users can support them by replacing the links with the actual resolved rules after decoding the style.

There are also some minor limitations for the `TLspSLDStyler` specifically:

- Lines painted with a perpendicular offset only support pixels as unit of measure, both for the offset and the line width.
- `TLcdSLDStroke.CSS_PARAMETER_MARKER_START` and `CSS_PARAMETER_MARKER_END` are ignored.

51.5.2 Limitations on external graphics

The default `ILcdObjectIconProvider` only supports the retrieval of an `ExternalGraphic` if its `OnlineResource` locates a file encoded in the SVG format, or a file with a format that can be decoded with `java.awt.Toolkit` (JPEG, GIF, PNG). It suffices to replace the default `ILcdObjectIconProvider` with another implementation to add support for other formats.

51.5.3 Other SLD styling limitations

- LuciadLightspeed selection and snapping do not take into account styling information. This is of interest for styles containing line displacements: in that case, the representation of the object is not at the actual location of the object.
- The implementation `TLcdSLDFeatureTypeStylePainter` does not apply feature type filtering.

CHAPTER 52

Working with XML data

This chapter explains the LuciadLightspeed XML framework.

- Section 52.1 gives a short introduction to XML and the XML framework.
- Section 52.2 describes how to decode and encode XML documents.
- Section 52.3 explains how to derive a data model from an XML schema.
- Section 52.4 shows how to customize the XML decoding.
- Section 52.5 explains some other advanced features that are not handled in the other sections.

52.1 Introduction to XML

The XML binding framework allows you to convert XML documents into Java content trees and the other way around. The framework is introduced to support other LuciadLightspeed XML-based formats, such as GML (discussed in chapter 49), AIXM and OGC Filter. The `com.luciad.format.xml.bind`, `com.luciad.format.xml.bind.schema`, and `com.luciad.format.xml.bind.schema.dataobject` are the main packages.

The new framework replaces the deprecated `com.luciad.format.xml` framework. The new framework is more efficient and easier to use and also provides additional functionality.

The current LuciadLightspeed implementation is based on the XML 1.0 specification, which you can find at W3C's website (<http://www.w3.org/TR/REC-xml/>). Reading and writing XML data is done using the streaming API for XML (StAX). This chapter is not meant to be an XML course; it assumes the reader has a thorough understanding of XML and XML schema, and at least a basic knowledge of StAX.

Creating a Java content tree from an XML document is called unmarshalling, the other way around is called marshalling. In this chapter, these terms are used interchangeably with decoding and encoding. Conceptually encoding and decoding are treated in a very similar fashion implemented with parallel classes (typically for each encoding class there is a similar decoding class). Because of that, the focus of the rest of this chapter is only on decoding.

This chapter focuses on documents based on a XML schema as all supported XML-based formats in LuciadLightspeed are based on a schema. The framework can also handle documents without a schema. The API Reference of the package `com.luciad.format.xml.bind` contains more information about this.

The XML binding framework is tightly integrated with the unified domain object approach described in [10](#). This chapter assumes you have a good understanding of that chapter.

52.1.1 Representing schema information

Because an XML schema is such an important aspect of most standardized XML formats, LuciadLightspeed introduces dedicated classes to represent information about a schema.

- The framework uses `TLcdXMLSchemaTypeIdentifier` and `TLcdXMLSchemaElementIdentifier` instances to uniquely identify XML schema types and elements.
- The framework uses `TLcdXMLSchemaType` and `TLcdXMLSchemaElement` instances to represent information about substitution groups.
- `TLcdXMLBuiltInDataTypes`, `TLcdXMLBuiltInConstants`, and `TLcdXLinkDataTypes` define public constants that give immediate access to the types, elements, and qualified names (`QName`) defined in the `XMLSchema` and `XLink` schema.

52.2 Decoding XML documents

Use a `TLcdXMLSchemaBasedDecoder` to decode schema-based XML documents. In order to create the right domain objects, you need to configure the decoder so that it knows the structure of the document and how to map it on the Java domain object classes. For data models that are based on an XML schema, this is straightforward. [Program 266](#) shows this for the ISO19115 data model¹. First a decoder is created and then it is configured for the ISO19115 data model. This makes the decoder ready to decode ISO 19115 documents.

```
1 TLcdXMLSchemaBasedDecoder decoder = new TLcdXMLSchemaBasedDecoder();
2 decoder.configure( TLcdISO19115DataTypes.getDataModel() );
3 return decoder.decode( "document.xml" );
```

Program 266 - Decoding ISO19115 XML documents

The XML framework also provides the more basic interface `ILcdXMLSchemaBasedDecoderLibrary`. A library is used when no such data model is available. You can find a more detailed explanation of how these libraries work in the API Reference.

You can control the decoding process of the `TLcdXMLSchemaBasedDecoder` as follows:

- Use the `ILcdInputStreamFactory` to control the creation of the `InputStream` instances that the decoder uses.
- Use the `XMLInputFactory` to control the creation of the `XMLStreamReader` that is used by the decoder.
- Use the `EntityResolver2` to resolve schema documents. You can find the `EntityResolver2` on the `TLcdXMLSchemaBasedMapping` associated with the decoder.

52.3 Creating a data model based on an XML schema

This section explains how to create and customize a data model for an XML schema. Refer to [Chapter 10](#) for more information on data models.

¹You can find a more detailed explanation of ISO 19115 in [Chapter 48](#)

52.3.1 Mapping XML types on Java classes

The main responsibility of the `TLcdXMLDataModelBuilder` class is to create a data model of which the data types map on the types defined in an XML schema. Given an XML schema, this class creates a data model according to a number of pre-defined rules. The most important rules are the following:

- The builder creates a data type in the data model for each type defined in the XML schema. This data type will have the same name as the XML schema type. For anonymous types, a unique name is derived from the location where the type is defined.
- Complex types map on data object types, simple types on primitive types.
- Type inheritance in XML schema maps directly on type inheritance. In other words, two types that inherit from each other in XML schema are mapped as two types that inherit from each other in the data model.
- The following list shows how complex content maps on properties:
 - Attributes: typically one property per attribute
 - Elements: typically one property per element declaration
 - Character content: one property
- Multi-valued elements (elements which have a `maxOccurrence` larger than 1) map on list properties.
- All data types inherit the `instance` class from their parent type. The instance class for the XML schema type `anyType` is `ILcdDataObject`, for `anySimpleType` it is `Object`.

You can find the complete and detailed list of rules in the API Reference of the `TLcdXMLDataModelBuilder` class.

To illustrate the use of the decoder, have a look at the schema shown in [Program 267](#). This schema defines two types: the anonymous type of the `Model` element and the global `Address` type.

```

1 <xsd:element name="Model">
2   <xsd:complexType>
3     <xsd:sequence>
4       <xsd:element ref="tns:address" minOccurs="0" maxOccurs="unbounded" />
5     </xsd:sequence>
6   </xsd:complexType>
7 </xsd:element>
8
9 <xsd:element name="address" type="tns:AddressType" />
10
11 <xsd:complexType name="AddressType">
12   <xsd:sequence>
13     <xsd:element name="street" type="xsd:string" />
14     <xsd:element name="number" type="xsd:int" />
15     <xsd:element name="city" type="xsd:string" />
16   </xsd:sequence>
17 </xsd:complexType>
```

Program 267 - A simple XML schema

`TLcdXMLDataModelBuilder` maps the anonymous type of the `Model` element on a type named `_Model` with a single list property of type `Address`. Note that the name of this data type starts with an underscore to indicate that it maps on the anonymous type of a global element. The `Address` type is mapped on a data type named `Address`. This data type has three

properties: street (a String), number (an Integer) and city (a String).

The `TLcdXMLDataModelBuilder` annotates all data models it creates with `TLcdXMLSchemaMappingAnnotation` and `TLcdXMLSchemaTypeMappingAnnotation` annotations. These annotations carry the XML mapping and XML schema information.

Program 268 shows how to create a data model based on the sample schema.

```

1 private static TLcdDataModel createDataModel() {
2     TLcdXMLDataModelBuilder builder = new TLcdXMLDataModelBuilder();
3     return builder.createDataModel("http://www.luciad.com/samples.xml", "path/to/schema.xsd");
4 }
```

Program 268 - Creating a data model for an XML schema

Once the data model has been created, you can configure a decoder with it. This ensures that the decoder uses the appropriate domain classes during decoding. Program 269 shows this for the sample data model.

```

1 public Object decode(String document) throws IOException, XMLStreamException {
2     TLcdXMLSchemaBasedDecoder decoder = new TLcdXMLSchemaBasedDecoder();
3     decoder.configure(createDataModel());
4     return decoder.decode(document);
5 }
```

Program 269 - Decoding XML documents

52.3.2 Custom domain classes

In many cases, you want to use dedicated domain classes for certain XML schema types. That way, you can implement certain interfaces and add methods to make your domain classes easier to use. If the domain class implements `ILcdDataObject` and follows the mapping rules of `TLcdXMLDataModelBuilder` then you only need to set the instance class of the corresponding type. Section 5.1 explains in detail how to do this.

Program 270 shows how custom domain classes are configured for the `xml.customdomainclasses` sample. First a `TLcdDataModelBuilder` is created. This builder is passed to a `TLcdXMLDataModelBuilder` which creates all types and properties for the given XML schema. Then the data model builder is used to set the instance classes for the `Address` and `_Model` types to the domain classes `Address` and `Model`. Both classes extend `TLcdDataObject`.

```

1 private static TLcdDataModel createDataModel() {
2     TLcdXMLDataModelBuilder builder = new TLcdXMLDataModelBuilder();
3     TLcdDataModelBuilder dataModelBuilder = new TLcdDataModelBuilder("sample");
4     builder.buildDataModel(dataModelBuilder, "http://www.luciad.com/samples.xml.
5         customdomainclasses",
6             Main.class.getResource("/samples/xml/customdomainclasses/samples.
7                 xml.customdomainclasses.xsd").toString());
8     dataModelBuilder.typeBuilder("_Model").instanceClass(Model.class);
9     dataModelBuilder.typeBuilder("AddressType").instanceClass(Address.class);
10    return dataModelBuilder.createDataModel();
11 }
```

Program 270 - Using custom domain classes

(from `samples/xml/customdomainclasses/CustomDomainClassesDataTypes`)

52.4 Custom XML decoding

You can also fully customize the XML mapping, encoding and decoding process. This allows you, for example, to have domain classes which do not implement `ILcdDataObject` or to provide custom code for decoding.

52.4.1 Custom simple types

A typical use case is to provide a custom domain class for an XML schema simple type. You can do this by performing the following four steps:

1. Set the instance class of the type to the custom domain class. This is exactly the same as shown in [Program 270](#).
2. Provide an instance of `ILcdXMLDatatypeUnmarshaller`. This interface defines the contract for unmarshalling XML simple types. Implementations of this interface basically transform text content into a Java object.
3. Define a custom `ILcdXMLSchemaBasedDecoderLibrary`. Such a library is responsible for the configuration of `TLcdXMLSchemaBasedDecoder` instances for the data model. This is done by extending the default `TLcdXMLDataObjectDecoderLibrary` class with additional code that registers the custom `ILcdXMLDatatypeUnmarshaller`.
4. Associate the custom decoder library with the data model by adding a `TLcdXMLSchemaMappingAnnotation` on the data model.

The `xml.binding.custom` sample illustrates this. It defines the XML schema type `ColorType` as a simple type that extends `xsd:int`. In the Java domain model, this `ColorType` is mapped on the `java.awt.Color` class. [Program 271](#) shows the datatype unmarshaller for the `Color` class. Note that this fragment also shows the implementation of the `ILcdXMLDatatypeMarshaller` that is required for encoding.

```

1 public class ColorDatatypeAdapter implements ILcdXMLDatatypeMarshaller<Color>,
2     ILcdXMLDatatypeUnmarshaller<Color> {
3
4     @Override
5     public String marshal(Color aValue, XMLStreamWriter aWriter,
6             ILcdXMLDocumentContext aContext) throws XMLStreamException {
7         return Integer.toString(aValue.getRGB());
8     }
9
10    @Override
11    public Color unmarshal(String aLexicalValue, XMLStreamReader aReader,
12             ILcdXMLDocumentContext aContext) throws XMLStreamException {
13        return new Color(Integer.parseInt(aLexicalValue));
14    }
15}
```

Program 271 - Custom marshalling and unmarshalling for the `ColorType`
(from `samples/xml/customdecodingencoding/ColorDatatypeAdapter`)

[Program 272](#) defines the `CustomDecoderLibrary` as an extension of the `TLcdXMLDataObjectDecoderLibrary`. The class overrides the `doConfigure` method and registers a new instance of the `ColorDatatypeAdapter` with the decoder. The adapter is registered for the XML schema type `ColorType` and the java class `Color`.

```

1 class CustomDecoderLibrary extends TLcdXMLDataObjectDecoderLibrary {
```

```

2 ...
3 ...
4 ...
5 ...
6 ...
7 ...
8 ...
9 ...
10 ...
11 ...
12 ...
13 ...
14 ...

```

Program 272 - A custom decoder library

Finally, Program 273 shows how the data model is annotated with a `TLcdXMLSchemaMappingAnnotation` with a new instance of the `CustomDecoderLibrary`.

```

1 // associates the custom decoder and encoder libraries with the data model
2 dataModelBuilder.annotateFromFactory(new DataModelAnnotationFactory<
3     TLcdXMLSchemaMappingAnnotation>() {
4     @Override
5     public TLcdXMLSchemaMappingAnnotation createAnnotation(TLcdDataModel aDataModel) {
6         return new TLcdXMLSchemaMappingAnnotation(
7             new MappingLibrary(aDataModel),
8             new DecoderLibrary(aDataModel),
9             new EncoderLibrary(aDataModel)
10            );
11        });
12    });
13  );
14  );

```

**Program 273 - A custom schema mapping
(from**

`samples/xml/customdecodingencoding/CustomDecodingEncodingDataTypes`)

Note that this snippet uses the `CustomMappingLibrary`. This library is not strictly necessary for a simple custom type. It is necessary in case of custom complex types as explained in Section 52.4.2.

52.4.2 Custom complex types

Is it also possible to fully customize the mapping of complex XML schema types. The two main differences with customizing simple types are that the `TLcdXMLSchemaBasedMapping` should be properly configured and that the decoding process of complex types is typically a hierarchical process that uses delegation to many other unmarshallers.

The main responsibility of the `TLcdXMLSchemaBasedMapping` is to keep track of information that is common to both encoding and decoding. This includes information about XML schema elements and types. It also manages the `ILcdXMLObjectFactory` instances that are responsible for creating new instances for a certain XML schema type. Both the `TLcdXMLSchemaBasedEncoder` and the `TLcdXMLSchemaBasedDecoder` aggregate a `TLcdXMLSchemaBasedMapping` instance.

To decode XML complex types, an instance of `ILcdXMLTypeUnmarshaller` is used. This instance needs to be registered just like an `ILcdXMLDatatypeUnmarshaller` on the `TLcdXMLSchemaBasedDecoder`.

The `xml.binding.custom` sample shows how this can be done in code. The sample defines the XML schema type `PointType` as a complex type with two attributes (`x` and `y`). In the Java

domain model, this `PointType` is mapped on the `TLcdLonLatPoint` class. This class does not implement `ILcdDataObject`, so you need to map it as a primitive type. Program 274 shows how you can do this using a custom extension of `TLcdXMLDataModelBuilder`. By overriding the `buildType` method you can control how the data model type is defined for an XML schema type. In this case, the type is defined as a primitive type with instance class `ILcd2DEditablePoint`. Note that you need an extension because otherwise the type would be defined as a data type with two properties.

```

1 TLcdXMLDataModelBuilder builder = new TLcdXMLDataModelBuilder() {
2
3     @Override
4     protected void buildType( TLcdDataTypeBuilder aTypeBuilder, TLcdXMLSchemaTypeIdentifier
5                               aTypeId ) {
6         if ( aTypeBuilder.getName().equals( "PointType" ) ) {
7             aTypeBuilder.instanceClass( ILcd2DEditablePoint.class );
8             aTypeBuilder.primitive( true );
9         } else {
10            super.buildType( aTypeBuilder, aTypeId );
11        }
12    }
13 });

```

Program 274 - Mapping a complex XML schema type on a primitive type

Program 275 shows how the `MappingLibrary` class defines and configures the object factory for the `PointType`. Note that this mapping library also registers the `ILcd2DEditablePoint` interface with the `TLcdXMLJavaClassResolver` of the mapping. Section 52.5.1 explains in more detail why you need to do this.

```

1 class MappingLibrary extends TLcdXMLDataObjectMappingLibrary {
2
3     public MappingLibrary(TLcdDataModel aDataModel) {
4         super(aDataModel);
5     }
6
7     @Override
8     protected void doConfigureMapping(TLcdXMLSchemaBasedMapping aMapping) {
9         super.doConfigureMapping(aMapping);
10        aMapping.getTypeObjectFactoryProvider().registerTypeObjectFactory(
11            CustomDecodingEncodingConstants.POINT_TYPE_ID,
12            ILcd2DEditablePoint.class, new ILcdXMLObjectFactory<ILcd2DEditablePoint>() {
13
14             @Override
15             public ILcd2DEditablePoint createObject(ILcdXMLDocumentContext aContext) {
16                 return new TLcdLonLatPoint();
17             }
18
19             @Override
20             public ILcd2DEditablePoint resolveObject(ILcd2DEditablePoint aObject,
21                 ILcdXMLDocumentContext aContext) {
22                 return aObject;
23             }
24         );
25
26         // register the interface
27         List<Class<?>> interfaces = new ArrayList<Class<?>>();
28         interfaces.add(ILcd2DEditablePoint.class);
29         aMapping.getJavaClassResolver().registerClassPriorityList(interfaces);
30     }
31 }

```

Program 275 - A custom object factory
(from samples/xml/customdecodingencoding/MappingLibrary)

Program 276 shows how the `ILcdXMLTypeUnmarshaller` for the `PointType` moves the point to the location determined by the `x` and `y` attributes. Note that the type marshaller does not need to create a new instance; this is already done by the object factory.

```

1  @Override
2  public ILcd2DEditablePoint unmarshalType(ILcd2DEditablePoint aObject,
3                                              XMLStreamReader aReader, ILcdXMLDocumentContext
4                                              aContext)
5      throws XMLStreamException {
6      String x = aReader.getAttributeValue(CustomDecodingEncodingConstants.NAMESPACE_URI, "x");
7      String y = aReader.getAttributeValue(CustomDecodingEncodingConstants.NAMESPACE_URI, "y");
8      aObject.move2D(Double.parseDouble(x), Double.parseDouble(y));
9      aReader.nextTag();
10     return aObject;
11 }
```

Program 276 - Custom schema type unmarshalling
 (from samples/xml/customdecodingencoding/PointTypeAdapter)

52.4.3 Advanced type unmarshalling

In general, writing an `ILcdXMLTypeUnmarshaller` can be much more complex than shown in the examples of the previous section. Fortunately, the XML framework is designed to help you as much as possible with this as described in this section.

Unmarshalling contract

The core interface of schema type unmarshalling is `ILcdXMLTypeUnmarshaller`. The contract of this interface defines how a schema type is unmarshalled into a Java model object. The interface defines a single `unmarshalType` method. The responsibility of this method is to unmarshal the attributes and content defined in a given XML type.

At the moment this method is called, the `XMLStreamReader` is positioned at the start tag of the element to be unmarshalled. The type of this element is the given XML type or an extension of that type. When this method returns, the cursor of the `XMLStreamReader` should be left at the start tag of the first child element that cannot be handled by this unmarshaller. Or, if no such element exists, at the end tag of the element to be unmarshalled. The former typically happens in case the element is of a derived type that has been extended with additional elements.

The object into which the contents of the XML element should be unmarshalled is passed as an argument to the method. Only in case this argument is null, the method should create a new Java object. This typically happens when there is no appropriate `ILcdXMLObjectFactory` registered or if the factory cannot create a new instance.

Dealing with type extension

The `ILcdXMLTypeUnmarshaller` interface supports extension by allowing to chain a set of schema type unmarshallers corresponding to the XML Schema type hierarchy. In this chain, each unmarshaller is responsible for unmarshalling only those attributes and child elements which are declared in the corresponding type. In order to set up such a chain, the `TLcdXMLSchemaBasedDecoder` provides a `TLcdXMLTypeUnmarshallerProvider` that you can use to find an appropriate schema type unmarshaller for a given schema type and Java class. This provider is also used to find an appropriate parent type unmarshaller.

When a type unmarshaller needs to unmarshal an object, it first unmarshals its own declared attributes, then optionally delegates to its parent and finally unmarshals its own declared elements and content. This approach is consistent with XSD type extension since XML does not define

an order on attribute and since elements of extended types always appear after the elements of the super types. Note that in order for this chaining to work, the type unmarshaller should not consume the start element and end element event. These events are handled automatically by the framework.



The `ILcdXMLTypeUnmarshaller` of a restricted type should not delegate to its parent type.

To be able to reuse common functionality, the `xml.bind.custom` sample introduces a common ancestor class, `AbstractSchemaTypeAdapter`. This class structures the (un)marshalling code and provides some utility methods to, for example, (un)marshal child elements, skip whitespace, and more. [Program 277](#) shows how the basic contract of the `ILcdXMLTypeUnmarshaller` interface is implemented. First, if no instance is passed as argument, a new instance that represents the current XML element in the model, is created. Then the attributes declared in this type are processed. If the type has a base type, then the base type's unmarshaller is called. Finally, the type unmarshaller unmarshals its declared content and the new instance is returned. The `createNewInstance`, `unmarshalDeclaredAttributes` and `unmarshalDeclaredContent` are placeholder methods that need to be overridden by subclasses.

```

1  @Override
2  public T unmarshalType(T aObject, XMLStreamReader aReader,
3                         ILcdXMLDocumentContext aContext) throws XMLStreamException {
4      if (aObject == null) {
5          aObject = createNewInstance();
6      }
7      unmarshalDeclaredAttributes(aObject, aReader, aContext);
8      ILcdXMLTypeUnmarshaller<? super T> parentMarshaller = getParentUnmarshaller();
9      if (parentMarshaller != null) {
10          parentMarshaller.unmarshalType(aObject, aReader, aContext);
11      } else {
12          aReader.next(); // consume start element
13      }
14      unmarshalDeclaredContent(aObject, aReader, aContext);
15      return aObject;
16  }
17
18  private ILcdXMLTypeUnmarshaller<? super T> getParentUnmarshaller() {
19      if (fParentID == null) {
20          return null;
21      }
22      return getSchemaDecoder().getTypeUnmarshallerProvider().getTypeUnmarshaller(fParentID,
23          fParentClass, true);
23 }
```

Program 277 - Type unmarshalling

(from `samples/xml/customdecodingencoding/AbstractSchemaTypeAdapter`)

Child elements

When a type unmarshaller needs to unmarshal child elements, it looks up an appropriate unmarshaller for its children using the `TLcdXMLSchemaBasedUnmarshallerProvider` of the `TLcdXMLSchemaBasedDecoder`.

At this point, you need to know how the child element is defined. In XML schema, child elements can either be locally defined or be references to global elements. An important difference between local and global elements is that XML Schema allows substitution of global elements by other elements. This means that a document can use another element, instead of the declared global element, without becoming invalid. An important constraint here is that XML schema only allows elements that are in the global element's substitution group as replacement

elements. Substitution groups are explicitly defined in the XML schema. Only elements of the same type or of an extension of the type of the substituted element can be part of a substitution group. As such, you can use substitution groups to model polymorph relations.

The important consequence for the type unmarshaller is that only for locally defined elements the XML schema type is known in advance. Because of potential substitution of global elements, you cannot determine the type of the element based on the schema alone. Note that, next to substitution groups, XML defines a second mechanism called `xsi:type` that enables the use of extension types in a document. This mechanism is currently not supported by the LuciadLightspeed XML framework.

To unmarshal a local element, the type unmarshaller should look up an appropriate `ILcdXMLUnmarshaller` based on the schema element identifier and the required Java class. [Program 278](#) shows how the `xml.customdecodingencoding` sample does this. The returned `ILcdXMLUnmarshaller` is then used to unmarshal the local element into a Java object of the given class.

```

1  protected <U> ILcdXMLUnmarshaller<? extends U> getUnmarshaller(
2      TLcdXMLSchemaElementIdentifier aSchemaElement,
3      Class<U> aClass) {
4          return getSchemaDecoder().getUnmarshallerProvider().getUnmarshaller(aSchemaElement, aClass);
    }

```

Program 278 - Finding an unmarshaller for a local element

(from `samples/xml/customdecodingencoding/AbstractSchemaTypeAdapter`)

To unmarshal a global element, the type unmarshaller should ask the framework for an appropriate unmarshaller based on the current element name and the required Java class. The `TLcdXMLUnmarshallerProvider` of the `TLcdXMLSchemaBasedDecoder` consults the registered information about substitution groups to find an appropriate result. The advantage of this approach is that the parent unmarshaller does not have to know anything about the substitutes of the child element. As a consequence, extending the substitution group with a new element can simply be done by registering the unmarshaller for the new element on the unmarshaller provider, without having to modify the parent type's unmarshaller code. [Program 279](#) shows how this is implemented in the `xml.customdecodingencoding` sample.

```

1  protected <U> U unmarshalChild(XMLStreamReader aReader, Class<U> aClass,
2      ILcdXMLDocumentContext aContext) throws XMLStreamException {
3      ILcdXMLUnmarshaller<? extends U> unmarshaller = getSchemaDecoder().getUnmarshallerProvider()
4          ().getUnmarshaller(aReader.getName(), aClass);
5      U result = unmarshaller.unmarshal(aReader, aContext);
6      return result;
    }

```

Program 279 - Unmarshalling a global child element

(from `samples/xml/customdecodingencoding/AbstractSchemaTypeAdapter`)

An example

The `xml.binding.custom` sample provides an example of a more complex custom type unmarshaller for the `ExtendedAddressType` XML schema type. [Program 280](#) shows how this type extends the `AddressType` with a local and a global element.

```

1  <xsd:complexType name="ExtendedAddressType">
2      <xsd:complexContent>

```

```

3   <xsd:extension base="base:AddressType">
4     <xsd:sequence>
5       <xsd:element ref="tns:location" />
6       <xsd:element name="color" type="tns:ColorType" />
7     </xsd:sequence>
8   </xsd:extension>
9 </xsd:complexContent>
10</xsd:complexType>
```

Program 280 - Definition of the ExtendedAddressType

Program 281 shows the type unmarshaller for the ExtendedAddressType. It is designed such that it can also correctly decode elements where the location and the color child elements are switched. When the reader's current name equals color, a color object is decoded. This is done using the color unmarshaller for the local color element. Note that this unmarshaller can be cached in the adapter because the type of the the color element is fixed from the XML schema. When the reader's current name is in the substitution group of the global location element, the location is unmarshalled based on the current name and the ILcd2DEditablePoint class. When an unknown element is encountered, an exception is thrown.

```

1 protected void unmarshalDeclaredContent(ExtendedAddress aResult, XMLStreamReader aReader,
2                                         ILcdXMLDocumentContext aContext) throws
3   XMLStreamException {
4     skipAllWhiteSpace(aReader);
5     while (aReader.isStartElement()) {
6       // expect either location or color
7       if (aReader.getName().equals(CustomDecodingEncodingConstants.COLOR_ELEMENT_ID.
8         getElementNames()[0])) {
9         aResult.setColor(getColorUnmarshaller().unmarshal(aReader, aContext));
10      } else if (getSchemaDecoder().getMapping().getSchemaSet().isSubstitutableBy(
11        CustomDecodingEncodingConstants.LOCATION_ELEMENT_ID.getElementName(), aReader.
12        getName())) {
13        aResult.setLocation(unmarshalChild(aReader, ILcd2DEditablePoint.class, aContext));
14      } else {
15        throw new XMLStreamException("Unexpected element", aReader.getLocation());
16      }
17      aReader.nextTag();
18    }
19    private ILcdXMLUnmarshaller<? extends Color> getColorUnmarshaller() {
20      if (fColorUnmarshaller == null) {
21        fColorUnmarshaller = getUnmarshaller(CustomDecodingEncodingConstants.COLOR_ELEMENT_ID,
22          Color.class);
23      }
24      return fColorUnmarshaller;
25    }
```

Program 281 - Unmarshalling child elements

(from samples/xml/customdecodingencoding/ExtendedAddressTypeAdapter)

52.5 Advanced features

52.5.1 Exporting classes using the TLcdXMLJavaClassResolver

In a typical XML-to-Java binding, there is a one-on-one mapping between XML elements and Java classes. In the XML binding framework, this means that a marshaller is normally registered for a specific XML element and specific Java class. However, there are cases in which this mechanism is not powerful enough:

- Users of a Java domain model might want to extend a class to add some functionality, but map the extension class on the same XML element as its super class. This would require to register the marshaller for all extension classes as well, which is not always possible (it might be unknown at the moment an XML library is written which extensions will be made in the future).
- Users might want to bind an XML element to a Java interface, instead of a class. This becomes complex however when a class implements multiple interfaces: to which interface should the framework bind the class to?

The `TLcdXMLSchemaBasedMapping` has a `TLcdXMLJavaClassResolver` which provides functionality for mapping a Java class to another class or an interface. Whenever no marshaller is found for the class to be marshalled, this resolver makes a list of all Java classes/interfaces that this class extends or implements, and choose one of them to bind against, based on a priority list. This allows to register marshallers on the marshaller provider for super classes or interfaces only, and let the class resolver take care of the mapping of derived classes to one of these base classes/interfaces.

52.5.2 Runtime schema extension

The XML framework also provides support for runtime schema extension. This allows decoding of XML documents of which the XML schema is not known in advance. A particular interesting case here is the situation where a subset of the schemas used in a document is known in advance, as is the case with, for example, GML. The standardized GML schema defines the core geographic information concepts and is abstract. Applications that want to express content in GML need to define an extension to GML in a specialized application schema. Such an application schema defines the concepts that are specific to that application making use of and possibly extending from existing GML types and elements. The support for runtime schema extensions then makes it possible to decode documents of application schemas using the default GML decoder.

In order to support runtime schema extension, the Java model classes for which the types can be extended by extension schemas, need to implement `ILcdDataObject`.

To enable schema extension, a decoder needs to be created with a `TLcdXMLDataObjectSchemaHandler`. This handler reads all unknown schemas encountered. For all these schemas, it creates a new data model and configures the decoder(encoder) with it. This allows the decoder to handle these schemas as well.

PART IX Advanced Development Guidelines

CHAPTER 53

General performance guidelines

This chapter provides some general guidelines for optimizing the performance of LuciadLightspeed applications. The guidelines apply to LuciadLightspeed applications in general, no matter what view type you are using. They are related to models, painting, and raster data.

The performance guidelines outlined in this chapter are generally applicable. There are other performance measures you can take, but these are specific to either Lightspeed views or GXY views. To read more about improving performance in Lightspeed views or GXY views, see [Chapter 54](#) or [Chapter 55](#) respectively.

53.1 Models

LuciadLightspeed provides several implementations of the `ILcdModel` interface. Which implementation to choose depends on the type of operation that you typically perform on the model data. [Section 5.1.4](#) describes when to use `TLcdVectorModel` or `TLcd2DBoundsIndexedModel`. Next to these main implementations of `ILcdModel` you can also use two implementations for tiled data as described below.

53.1.1 `TLcd2DRegularTiledBoundsIndexedModel`

Like the `TLcd2DBoundsIndexedModel`, the `TLcd2DRegularTiledBoundsIndexedModel` maintains an internal spatial data structure that allows you to quickly retrieve domain objects based on their bounds. In this case, the data structure is a regular grid. It provides an efficient implementation of the `ILcd2DBoundsIndexedModel` interface with its `applyOnInteract2DBounds` method. In addition, the class also implements the `ILcdIntegerIndexedModel` interface.

Because of its regular tiling, this class is more suited for elements that are fairly evenly distributed inside the model bounds. The simplicity of the data structure is also best suited for elements that change frequently of position.

53.1.2 `TLcdRegularTiled2DBoundsIndexedModel`

The `TLcdRegularTiled2DBoundsIndexedModel` is mainly intended for lazily loading large data sets. A data set must be organized as a regular grid, with each tile in the grid corresponding to a file. You still have to implement the `ILcdTileProvider` that provides the individual tiles.

53.2 Querying a model

When you query a model with the `ILcdModel#query` method, using a query containing an `ILcdOGCCondition`, the performance of this method depends heavily on the type of condition, the model implementation and the backend of the model.

That performance becomes very relevant when you are trying to visualize large vector data sets. See [Chapter 24](#) for more information.

53.2.1 In-memory models

In-memory models are usually based on `TLcd2DBoundsIndexedModel`. Many file formats, such as GML or GeoJson, are based on `TLcd2DBoundsIndexedModel`, and load the entire file into memory.

Its `query()` method will use `applyOnInteract2DBounds()` to first find the elements within the requested spatial extent, and then apply the OGC condition on each of those.

This is also the default behavior for all `ILcd2DBoundsIndexedModels`.

If a model is not spatially indexed, the `query()` method will use `elements()` to loop over all objects and apply the OGC condition on each of them.

53.2.2 Databases

`LuciadLightspeed` models for databases convert the OGC condition into a SQL statement. The SQL statement will contain a bounding box combined with other restrictions:

```

1 SELECT ST_AsEWKB(geometry), id, name
2 FROM osm_roads
3 WHERE (geometry && ST_SetSRID('BOX3D(4.6187 50.7863, 4.8134 50.9403)::box3d, 4326))
4 AND (type = 'motorway')
```

Program 282 - Example SQL select statement which combines spatial and non-spatial filtering

The performance depends heavily on the configuration of the database. Make sure to have the appropriate indices on your data. You will certainly need a spatial index. In the example above, an index on “type” would be beneficial as well.

This performance not applies to Oracle, Microsoft SQL Server, Postgresql, and GeoPackage.

53.2.3 WFS servers

The client model for a WFS server simply sends the OGC condition to the server as XML.

The WFS server then delegates that OGC condition to its back-end models.

Therefore, the performance depends mostly on the back-end model implementation.

53.2.4 SHP models

Models for Shapefiles are of the lazy-loading type. They have been optimized to accelerate `query()` calls with OGC conditions:

- The spatial extent (if any) is evaluated by means of a spatial index file next to the `.shp` file. This file is created automatically if it is absent.
- The results of other OGC conditions are cached.

53.3 Raster Painting

Painting rasters efficiently is challenging because rasters are generally large data sets. This section focuses on optimizing the performance of the raster painters in LuciadLightspeed. Most standard `ALcdImage` and `ILcdRaster` implementations use lazy loading to retrieve pixel data. Efficient raster painting is therefore also closely tied to efficient data loading.

53.3.1 Using tiled rasters

When painting raster data, the LuciadLightspeed raster decoders use the tiled structure of the raster to only load the tiles that are required. Many formats are always tiled (for example, DMED, DTED, DEM). Some formats may be tiled (such as GeoTIFF) and other formats are never tiled (notably JPEG). Non-tiled data negatively affects decoding time, memory usage, and therefore ultimately, painting performance.

The GeoTIFF format provides a flexible way to tile images inside a single file, while still allowing different color schemes (8-bits with color map, 24-bits true color,...) and encoding schemes (ZIP, LZW, JPEG,...). The `TLcdGeoTIFFModelDecoder` and `TLcdGeoTIFFModelEncoder` support such tiled GeoTIFF images. The GeoTIFF raster encoding sample `Converter` provides a simple conversion tool for reading rasters in any supported format and writing them out as tiled GeoTIFFs. Typical tile sizes are 256x256 or 512x512 pixels.

Note that the freely available `libtiff` library contains a command-line tool `tiffinfo` that allows you to inspect the structure and encoding of GeoTIFF files. This is a good starting point for checking how efficient your GeoTIFF rasters will be in your application.

53.3.2 Using multilevel rasters

The presence of large rasters can still slow down raster painting, in spite of any tiling. Especially if the view is zoomed out, a large amount of data may be visible. Creating data with multiple levels of detail then provides a major performance improvement. When the view is zoomed out, the painters select lower levels of detail, thus loading less data and painting more rapidly. The typical formats that offer multiple levels of detail are the wavelet formats JPEG2000, ECW, and MrSID.

The GeoTIFF format provides an alternative, as it can store multiple images at different resolutions in a single file. A benefit over the wavelet formats is that it does not require any native libraries. The `TLcdGeoTIFFModelDecoder` and the `TLcdGeoTIFFModelEncoder` support such multilevel (and tiled) GeoTIFF images. The GeoTIFF raster encoding sample `Converter` provides a simple conversion tool. A multilevel raster should typically contain 3, 4, or more levels, depending on the expected scale range of the application.

53.3.3 Tuning DMED/DTED fall-back

If you are working with DMED/DTED files, the `TLcdDMEDModelDecoder` and the `TLcdDTEDDirectoryModelDecoder` provide some specific settings that allow to fall back on lower or higher DTED levels in case not all DTED levels are available. Enabling fall-back on higher levels ensures that some elevation data is always visible, for example in case DTED level 1 is available but DTED level 0 is not. However, such a setup generally performs badly, since each level that is actually loaded is much larger than the preceding level that is missing. The best solution is to ensure that low levels are always available. For example, DTED level 0 data can easily be downloaded and installed in the DTED directory structure. In general, it helps to inform the

model decoders of the minimum and maximum levels that are available, using the corresponding setters.

53.3.4 Combining rasters

Rasters are objects that are typically put in models, just like any other model elements. This means that they are painted and processed sequentially, just like any other model elements. If a lot of rasters have to be warped, this causes a large overhead, since a warped image is computed for each raster in turn. Applications should therefore try to avoid creating models with many individual rasters, for example from individual DTED files. If the rasters lie in a regular grid pattern, it is much more efficient to group their tiles into a single raster, for example as is done when reading DTED files based on a DMED file.

CHAPTER 54

Performance guidelines for Lightspeed views

LuciadLightspeed comes with default performance settings that ensure a consistent, high frame rate on most hardware and most screen sizes. Most of the time, the only thing you need to specify is the available heap memory. LuciadLightspeed can work with less than 512MB, but more heap memory is preferable. This is especially the case on 64-bit systems and with higher screen resolutions, such as on screens offering Full HD and more. For more information on memory, refer to [Section 54.4.1](#).

If you need to display an exceptional amount of data, or if your hardware has limited resources, you can enable more aggressive performance optimizations using the view builder API. This is explained in [Section 54.1](#). Specific performance settings for shape and raster painting are described in [Section 54.2](#).

The remainder of this chapter discusses more advanced performance-related options.

54.1 Tuning view performance optimizations

`TLspViewBuilder` has several performance-related settings. The most obvious one is the `buildAWTView` method. It allows you to create an AWT view that is perfectly compatible with a Swing-based application and provides the highest frame rate.

If you need to display an exceptional amount of data, or if your hardware has limited resources, you can activate more aggressive performance optimizations by calling the `paintingHints` method and passing in the `MAX_PERFORMANCE` painting hint. If you do so, you will trade in some visual quality. The view painting hints may also affect the default behavior of other settings that are explained further on in this chapter. However, any manual override of the default settings will take precedence.

Finally, if your device has a high-resolution screen and a low-end GPU, you could consider rendering the view in a lower resolution by calling the `resolutionScale` method with a parameter smaller than 1. As a result of the upscaling, text and icons will look bigger .

54.2 Tuning shape and raster painter settings

The `TLspShapeLayerBuilder` and `TLspShapePainter` allow controlling the quality and performance of vector rendering. For more information, refer to the javadoc of `TLspShapePaintingHints`.

The `TLspRasterStyle` provides a few basic properties to control whether a given raster is visible at a given scale:

- `startResolutionFactor` is the highest pixel density (number of raster pixels per screen pixel) at which a raster is painted. Small values (for example 2.0) generally improve performance, by not displaying the raster or raster level if its resolution appears to be high, relative to the resolution of the view.
- `levelSwitchFactor` is the factor that affects the scale point at which a raster level is selected. A value of 1.0 gives the highest practical raster quality: as soon as a single raster pixel would project to several screen pixels, a higher resolution level is used. A lower value delays this switching: a single raster pixel may project to multiple screen pixels. This can cause pixelation or blurring effects, but it will improve the painting performance and reduce memory usage. The default value is 0.3, which provides a good trade-off between quality and performance for most applications.

The behavior of a raster layer is also influenced by its layer type:

- `INTERACTIVE`: indicates that the layer should support smooth interaction, such as changes of the visibility or style of the raster data.
- `BACKGROUND`: indicates that the raster is used as background data in the view, and rarely changes. The view combines all background layers at the bottom of the view to optimize memory usage and painting performance.

54.3 LuciadLightspeed configuration options



The guidelines discussed in this section are related to OpenGL configuration options, and are intended for developers with solid knowledge of OpenGL.

54.3.1 OpenGL resource cache

LuciadLightspeed layers that use OpenGL data such as textures, buffers, and so on, cache this data in the `ILspGLResourceCache`, which can be obtained from the view's services (`TLspViewServices`). The resource cache takes ownership of the data. It pushes out and destroys the least recently used data at its own discretion when its limit is reached.



By default, all views created with `TLspViewBuilder` share a single GL resource cache instance. This prevents applications which use multiple views from eventually using up all available GPU memory. To disable automatic sharing and give each view its own resource cache, use the `disableAutomaticContextSharing()` method in the view builder.

The default resource cache size for a Lightspeed view is suited for all view sizes and a moderate amount of content. You can override the resource cache size while you are creating the view with `TLspViewBuilder`. Alternatively, you can modify the cache size using the system property `-Dcom.luciad.view.lightspeed.opengl.cacheSize=X`, where `X` corresponds to the requested resource cache size in megabytes.

The number of entries that can be stored in the cache is by default not limited. You can change the default behavior during view construction with `TLspViewBuilder`, or you can specify a maximal entry count using the system property

`-Dcom.luciad.view.lightspeed.opengl.maxCacheEntries=X`, where X corresponds to the requested maximal number of cache entries.



You can only access the `ILspGLResourceCache` during the paint loop, for example within a `paint` method call of an `ILspPaintableLayer` object, or a `paintObjects` call of an `ILspPainter` object.

54.3.2 Draping on terrain

The draping algorithm implementation tries to use an OpenGL frame buffer object (FBO) if that is supported, and resorts to using the main framebuffer if not. If you are using the main framebuffer, it should be backed by a buffer that has an alpha channel. If you are not using an FBO for draping and if an alpha channel is not available, your draped objects will look oversaturated. The following options allow you to configure buffer usage for the draping algorithm:

- Enforce the use of the main framebuffer instead of an offscreen framebuffer:
`-Dcom.luciad.view.lightspeed.terrain.disableFBO=true`
- If the main framebuffer is requested, back it with a buffer with an alpha channel:
`-Dcom.luciad.view.lightspeed.graphicsConfiguration.requestAlpha=true`
- Specify which visual to use. On some Linux systems, the above does not work correctly. In that case, you should explicitly specify which visual to use (see the Linux command `glxinfo`):
`-Dcom.luciad.view.lightspeed.glx.visual=0x24`

By default, LuciadLightspeed uses the following configuration option:

`-Dcom.luciad.view.lightspeed.terrain.disableFBO=false`.

54.3.3 Transparency

By default, a 3D Lightspeed view is configured with an order-independent transparency algorithm. This algorithm performs offscreen rendering to an OpenGL floating point frame buffer object (FBO). If the graphics board supports line smoothing when rendering to floating point buffers, a regular floating point buffer is used. In the other case, a multi-sample buffer with two samples per pixel is used, if that is supported.

This behavior can be overridden by specifying:

- `-Dcom.luciad.view.lightspeed.transparency.maxDefaultColorSamples=value`, where value is 1, 2, or 4.

Instead of overriding the default maximal number of color samples, you can also explicitly specify how many color samples and coverage samples must be used:

- To explicitly request the number of color samples (MSAA):
`-Dcom.luciad.view.lightspeed.transparency.colorSamples=4`
- To explicitly request the number of coverage samples (CSAA):
`-Dcom.luciad.view.lightspeed.transparency.coverageSamples=8`

For example, NVIDIA hardware can typically be configured using the anti-aliasing settings 8x (4xMS, 4xCS) which corresponds to the following LuciadLightspeed settings:

`-Dcom.luciad.view.lightspeed.transparency.colorSamples=4`
`-Dcom.luciad.view.lightspeed.transparency.coverageSamples=8`

If FBOs are not supported, you can explicitly disable the default transparency algorithm with
-Dcom.luciad.view.lightspeed.transparency.simple=true

In any case, the transparency algorithm will fall back to a lower quality setting if the requested parameters are not supported by the graphics hardware.

54.4 External configuration options

54.4.1 Memory guidelines

A LuciadLightspeed application consumes Java heap memory, native heap memory and video memory:

- **Java heap memory**
 - Stores all objects created by calling Java's new operator.
 - Is recollected by the garbage collector (GC) whenever possible.
 - Allows for the setting of the initial and maximum heap size with the JVM options -Xms and -Xmx.
 - * Example: `java -Xms512m -Xmx512m MainPanel`
- **Native heap memory**
 - Allocated for example through the NIO call `java.nio.ByteBuffer.allocateDirect`.
 - * The resulting buffer that holds the data is in native memory.
 - * The Java heap object only contains a reference to this native buffer.
 - Automatically reclaimed through the GC process, but there is no incentive to do so if there is pressure on the native heap.
 - * The GC process is triggered by the Java heap, *not* the native heap.
 - * Can run out of native memory when the Java heap almost empty, and no GC events are triggered.
 - We strongly advise *against* using direct buffers yourself for any (and certainly for short-lived) objects in a LuciadLightspeed application.
- **Video memory (VRAM)**
 - Dedicated graphics memory, typically about 512MB, 1GB, 2GB or larger.
 - Not controlled by the Java GC process.
 - Indirectly allocated by OpenGL and OpenCL API calls such as `glTexImage2D`.
 - Preferably controlled by the view's `ILspGLResourceCache`.

It is important not to exhaust a memory type. For example, on a Windows 32 bit JVM, there is typically only 1.5 gigabytes available to the Java process. Hence, running your application with the flags `-Xms1500m -Xmx1500m` is a bad idea, as there will be no space left for the native heap. It would be preferable to use a 750 MB Java heap and a 250 MB OpenGL resource cache, for example. This would ensure that there is still about 500 MB native memory that remains available if the OpenGL graphics driver duplicates and caches its data in native memory. Your application is unlikely to experience out-of-memory errors due to native heap exhaustion.

Note that the memory limit for each process is typically tighter on a 32 bit Linux JVM. If memory limitations are an issue, we recommend running your application on a 64 bit JVM.

Follow [this link¹](#) for a more in-depth discussion on Java memory and the behavior when running out of (native) memory.

54.4.2 Garbage collection

Java objects created with the `new` operator are collected by the garbage collector when possible. We recommend using the concurrent low pause time garbage collector, also known as concurrent mark sweep or CMS, by using the following VM options:

- `-XX:+UseConcMarkSweepGC`
- `-XX:+CMSParallelRemarkEnabled`

This results in reduced application pause times, which is desirable in an interactive application.

Please refer to [this link²](#) for more Java performance tuning tips.

54.4.3 Video driver

Your graphics driver offers various configuration options, each of which impact the performance and visual quality of your LuciadLightspeed application. The most important options to consider, are:

- Anti-aliasing: in many cases, the best performance and the smallest memory footprint are obtained when anti-aliasing is disabled, at the cost of reduced image quality. However, anti-aliasing of point, line and triangle primitives may still occur, depending on your OpenGL implementation, even if the anti-aliasing option is disabled. Note that if you use `TLspLineStyle` objects with a width larger than one, disabling anti-aliasing will not improve performance. Increasing the anti-aliasing quality setting will also increase video memory consumption of your LuciadLightspeed application. Please consult the configuration options outlined in [Section 54.3.3](#) as well.
- Performance levels: some graphics drivers can be configured for adaptive GPU clocking. Choosing the maximum performance mode rather than the adaptive mode will increase your LuciadLightspeed application performance.

54.5 Monitoring the allocation of video memory

As mentioned earlier, video memory is not controlled by the Java garbage collector process, but through the `ILspGLResourceCache` instead. You can monitor the amount of video memory allocated or used by registering an `ILspGLResourceCacheListener`. Such a listener gets notified whenever a new resource is added, an existing resource is used, or an existing resource is removed from the cache.

The LuciadLightspeed sample applications use this mechanism to display video memory statistics in an overlay that can be enabled by pressing the statistics button in the top right corner of the sample frame. [Program 283](#) shows the listener that is registered with the resource cache.

```
1 public class StatisticsGLResourceCacheListener implements ILspGLResourceCacheListener {
```

¹ <http://www.ibm.com/developerworks/java/library/j-nativememory-linux/>

² <http://www.oracle.com/technetwork/java/tuning-139912.html>

```

3  private static final ILcdLogger sLogger = TLcdLoggerFactory.getLogger(
4      StatisticsGLResourceCacheListener.class);
5
6  private long fMaxBytesUsed = 0;
7  private int fMaxCountUsed = 0;
8  private Map<ALspGLResource, Long> fResourcesUsed
9      = new IdentityHashMap<ALspGLResource, Long>();
10
11 private static void checkThread() {
12     if (!EventQueue.isDispatchThread()) {
13         sLogger.error("Wrong thread: " + Thread.currentThread(), new RuntimeException());
14     }
15 }
16 /**
17 * Resets the count of bytes used since the last call to reset.
18 * Also updates the maximal bytes used values.
19 * This method is called between each frame.
20 */
21 public void reset() {
22     checkThread();
23     fMaxBytesUsed = Math.max(fMaxBytesUsed, getBytesUsedSinceLastReset());
24     fMaxCountUsed = Math.max(fMaxCountUsed, getResourcesCountUsedSinceLastReset());
25     fResourcesUsed.clear();
26 }
27
28 @Override
29 public void resourceCacheEvent(TLspGLResourceCacheEvent aEvent) {
30     checkThread();
31     // Store the resource and its size if it was added or used.
32     if (aEvent.getType() == TLspGLResourceCacheEvent.Type.RESOURCE_ADDED || 
33         aEvent.getType() == TLspGLResourceCacheEvent.Type.RESOURCE_USED) {
34         ALspGLResource resource = aEvent.getResource();
35         fResourcesUsed.put(resource, resource.getBytes());
36     }
37 }
38
39 /**
40 * Gets the total bytes of resources that are used
41 * since the last call to {@link #reset()}. If a resource
42 * was used multiple times, it is only counted once.
43 *
44 * @return the total number of bytes of used resources.
45 */
46 public long getBytesUsedSinceLastReset() {
47     checkThread();
48     long result = 0;
49     for (Map.Entry<ALspGLResource, Long> entry : fResourcesUsed.entrySet()) {
50         result += entry.getValue();
51     }
52     return result;
53 }
54
55 /**
56 * Gets the number of resources used since the last
57 * call to {@link #reset()}. If a resource
58 * was used multiple times, it is only counted once.
59 *
60 * @return the number of resources used
61 */
62 public int getResourcesCountUsedSinceLastReset() {
63     return fResourcesUsed.size();
64 }
65
66 /**
67 * Returns the maximal total bytes used between two
68 * consecutive calls to {@link #reset()}.
69 *
70 * @return the maximal bytes used.
71 */
72 public long getMaxBytesUsed() {
73     return fMaxBytesUsed;
74 }
```

```
75 /**
76  * Returns the maximal number of resources used
77  * between two consecutive calls to {@link #reset()}.
78  *
79  * @return the maximal number of resources used
80  */
81 public int getMaxCountUsed() {
82     return fMaxCountUsed;
83 }
84 }
```

Program 283 - Listener that counts the total bytes of VRAM used per frame
(from samples/lightspeed/debug/StatisticsGLResourceCacheListener)

CHAPTER 55

Performance guidelines for GXY views

This chapter discusses measures you can take to improve painting performance in GXY views.

55.1 Background layers

Many applications work with 2D views that contain relatively static background layers, and dynamic foreground layers. In these cases, it is useful to identify the background layers and set the property `numberOfCachedBackgroundLayers` of the interface `ILcdGXYView`. This is a hint for the view that the given number of background layers should be cached in an image. When the view has to be repainted, for example due to changes in the foreground layer, the cached layers can then be repainted quickly.

55.2 Smart panning

`TLcdGXYViewJPanel` and `TLcdGXYViewCanvas` offer a smart panning functionality. If it is enabled, panning operations will reuse the cached background layers image and only paint the newly exposed areas for those layers. This can noticeably improve performance for models with many objects or complex painting algorithms.

Notice, however, that smart panning may cause incorrect renderings in some circumstances. For example, consider an object of which the visual representation is larger than its bounds, such as a point represented by an icon. If a pan operation exposes the part of the object outside its bounds, the object will not get painted on the newly exposed area.

55.3 Preventing raster warping

On-the-fly warping between projections is a relatively expensive operation when painting rasters. To improve performance, the current view reference should therefore be equal to the model reference of the raster data. The painters automatically select a more efficient non-warping mode in such cases.

55.4 Tuning raster painter settings

The `TLcdGXYImagePainter` and all `ILcdRasterPainter` implementations provide a few basic properties to control whether a given raster is visible at a given scale:

- `startResolutionFactor` is the highest pixel density (number of raster pixels per screen pixel) at which a raster is painted. Small values (for example 2.0) generally improve performance, by not displaying the raster or raster level if its resolution appears to be high, relative to the resolution of the view.
- `stopResolutionFactor` is the lowest pixel density (number of raster pixels per screen pixel) at which a raster is painted. Small values (for example 0.5) prevent the display of the raster or raster level if its resolution appears to be low, relative to the resolution of the view.
- `forcePainting` is a flag that allows to override the above settings. Although useful for testing, setting this flag generally causes performance to peak when zooming out on large rasters.

If you are working with multilevel raster data, the `TLcdGXYImagePainter` and other multilevel raster painters will select the appropriate level to paint in the current view, based on a number of settings. Tweaking these settings, in addition to the above properties, can help finding a right balance between quality and performance.

- `levelSwitchFactor` is the factor that affects the scale point at which a raster level is selected. The default value is 1.0: as soon as a single raster pixel would project to several screen pixels, a higher resolution level is used. This way, the highest practical raster quality is chosen. A value smaller than 1.0 (for example 0.2) delays this switching: a single raster pixel may project to multiple screen pixels. Setting a small value may cause pixelating effects, but it will generally improve the painting performance.
- `levelSwitchScales` is an optional list of scales that can achieve the same effects as the `levelSwitchFactor`. It explicitly expresses the scales at which the painter switches between raster levels, instead of relying on a single factor.

These properties can be combined with the properties of the layer, such as the scale range.

CHAPTER 56

Advanced threading and locking

To perform basic operations on models and views in LuciadLightspeed, you must make use of threading and locking. LuciadLightspeed provides the `TLcdLockUtil` class in its API to assist with these operations, and defines a set of multi-threading rules.



For an overview of those fundamental LuciadLightspeed threading and locking rules, see [Chapter 9](#).

To deal with possible performance issues in your application, however, LuciadLightspeed allows you to go beyond the basic rules. This chapter describes scenarios that deviate from LuciadLightspeed threading fundamentals to improve your application's responsiveness. In this approach, you do not have to use the EDT at all times to apply model and view changes. You can offload the EDT by:

- Updating a model off the EDT to handle live data updates and similar scenarios
- Preparing model updates off the EDT, and swapping the updates into the model on the EDT.
- Painting layers asynchronously in GXY views

Section 56.1 provides more information about these three scenarios.

If you are using offscreen views, you do not need to use the EDT either. For more information about this specific case, see [Section 56.2](#).

56.1 Offloading the EDT

You can apply various strategies to reduce the workload for the EDT and divert some of the work to a different thread.

56.1.1 Updating a model off the EDT

If your application model undergoes substantial or frequent changes, performing these changes on the EDT could affect UI response time: the view is slow to respond while all these changes are processed on the EDT. This might happen if you are updating airplane tracks from a live feed over a network connection, for example.

To prevent such a negative impact on the user experience, LuciadLightspeed allows you to perform model updates off the EDT. You can use any thread on condition that:

- You are using a Lightspeed view, not a GXY view
- You do not have other GUI elements that depend on the model, like a table view
- Your application does not allow users to edit the data using an `ILspEditor`
- You do not have selection listeners that perform GUI operations
- You are not using the `TLcdSimulator` of the Real-time Engine component
- You are not building your application in Lucy, which offers all the elements of the previously listed conditions



Keep in mind that you still need to use a single thread for your operations, and that you still need to lock the model.

If any of the conditions in the list above are not met, see Section 56.1.2 for a possible alternative.

56.1.2 Preparing model updates off the EDT

If you want to perform model updates off the EDT, but cannot meet the conditions listed in Section 56.1.1, you can use a prepare-and-swap pattern to offload most of the work. In this approach, you:

1. Prepare the model updates on any thread. You are not changing the model itself yet, so model write locks are not necessary at this point. You may still need to take read locks, though. For example, you can read data from disk or from a socket, parse, convert or pre-process data.
2. Swap the model updates to the EDT, and apply them to the model in batch, with a write lock.

You can use `SwingUtilities.invokeLater` to schedule the swap on the EDT.

56.1.3 Asynchronous layer painting in GXY views

In GXY views, you can use asynchronous layer wrappers to paint the views asynchronously. This is useful if your GXY views has layers that take a long time to paint.

`TLcdGXYAsynchronousLayerWrapper` objects behave as regular layers: you can access the wrapped layers from the view's painting thread, the EDT.

The layer wrappers manage their own painting threads. The wrapped layer itself is painted in such a separate background thread, and the painting result is then added to the view.

The asynchronous painting thread can access the wrapped layer at any time. Therefore, you need to be careful when you access the wrapped layer from any other thread to make changes. To change the layer properties, you must tell the layer wrapper to execute the layer change at a safe point in time, on an appropriate thread, so that it does not interfere with the layer painting. To safely manipulate the wrapped layer, use one of the methods offered by `TLcdGXYAsynchronousLayerWrapper`:

- `invokeAndWaitOnGXYLayer`: schedules the layer change immediately in the EDT, blocking the caller's thread
- `invokeLaterOnGXYLayer`: schedules the change on the background thread
- `invokeLaterOnGXYLayerInEDT`: schedules the change on the EDT

For more details on asynchronous layer painting in GXY views, see [Chapter 33](#).

56.2 Supporting off-screen views

Many servers, such as OGC WMS servers, build images and render them to an internal memory buffer, rather than directly displaying the images on a screen. Such off-screen views are also useful when you want to prepare an image for visualization at a later stage, or in a testing scenario. LuciadLightspeed offers the classes `TLspOffscreenView` and `TLcdGXYViewBufferedImage` for this purpose. For more information about `TLspOffscreenView`, see [Section 20.4](#).

To paint offscreen views, you do not have to switch to the EDT. You can freely choose the thread from which you call `view.paint()`. It can be the EDT or any other thread. Keep in mind that you need to perform all model updates on the chosen view painting thread as well, unless you use the technique described in [Section 56.1.1](#).

CHAPTER 57

Logging and performance monitoring

The first section of this chapter describes how to enable logging and how to handle the log messages. The other sections provide information on performance monitoring using JConsole and JavaVisualVM.

57.1 Logging

57.1.1 Producing log messages

Each class that wishes to send messages to the log should do so by using a `com.luciad.util.logging.ILcdLogger` instance. To obtain an `ILcdLogger` instance you have to use the `getLogger` static method of the `TLcdLoggerFactory` class. The method takes as argument a string representing the logger's name. By convention the logger name tracking the execution of a particular class is called `com.luciad.<package>.<class_name>`. For example:

```
1 import com.luciad.util.logging.*;
2 class MyClass {
3     private static ILcdLogger sLogger = TLcdLoggerFactory.getLogger(MyClass.class.getName());
4     ...
5 }
```

Program 284 - Instantiating an `ILcdLogger`

The `ILcdLogger` acts as a wrapper around a concrete logging framework implementation, for example Java standard logging (the default), or log4j. The interface allows you to log messages at several levels: trace, debug, info, warn and error. Internally, these are mapped to corresponding levels of the concrete implementation. For the standard Java logging framework (`java.util.logging` - referred to as JUL), the mapping is as follows:

<code>ILcdLogger</code>	JUL Level
error	SEVERE
warn	WARNING
info	INFO
debug	CONFIG, FINE
trace	FINER, FINEST

The `ILcdLogger` methods allow you to specify an optional Object and/or a Throwable in addition to the log message itself. Both of them are passed unchanged to the concrete logging implementation, if supported.

57.1.2 Handling log messages with the standard Java logging framework

By default, LuciadLightspeed uses the standard Java logging framework in `java.util.logging` for recording debug and trace information. Logging can be configured by modifying the global `$JAVA_HOME/lib/logging.properties`. Alternatively, if you wish to avoid editing the global Java logging properties file you can create a custom one and specify it as a system property:

```
1 java -Djava.util.logging.config.file=mylogging.properties MyApplication
```

Program 285 - Specifying a custom logging properties file

The logging level and handlers can be configured separately for each class or package. If a class or package is not explicitly configured, the settings of its parent package are used instead. For example, the following listing shows how to enable logging at the FINEST level for the class `com.luciad.format.shp.TLcdSHPModelDecoder` and at the INFO level for all the other LuciadLightspeed classes:

```
1 # Debug/trace logging
2 com.luciad.format.shp.TLcdSHPModelDecoder.level = FINEST
3 com.luciad.level = INFO
4 com.luciad.useParentHandlers = false
5 com.luciad.handlers = java.util.logging.ConsoleHandler
6 java.util.logging.ConsoleHandler.level = FINEST
7
8 # Optionally, you can specify a TLcdJULSimpleFormatter for messages.
9 # The arguments for the format string are:
10 #   1: Message
11 #   2: Logger name
12 #   3: Level
13 #   4: Date
14 #   5: Source class name
15 #   6: Source method name
16 #   7: Thread ID
17 #   8: Throwable
18 #java.util.logging.ConsoleHandler.formatter = com.luciad.util.logging.TLcdJULSimpleFormatter
19 #com.luciad.util.logging.TLcdJULSimpleFormatter.format = %3$s: %5$s#%6$s %1$s %8$s\n
```

Program 286 - Logging properties file

The presentation of the log messages can be configured by a formatter associated with the log handler. If nothing is specified, the Java `ConsoleHandler` uses the standard class `SimpleFormatter`. For a more flexible message display, you can use the LuciadLightspeed formatter class `TLcdJULSimpleFormatter`, as illustrated in the sample above.

The display message is obtained by passing the `format` property as the first argument to the `String.format(String, Object... args)` method and the JUL log record properties as the remaining arguments in this order:

1. `LogRecord.getMessage()`
2. `LogRecord.getLoggerName()`
3. `LogRecord.getLevel()`
4. `LogRecord.getMillis()` (as a Date object)
5. `LogRecord.getSourceClassName()`
6. `LogRecord.getSourceMethodName()`
7. `LogRecord.getThreadID()`

8. LogRecord.getThrown()

The configuration from the snippet above results in log messages being displayed as <Log level>: <Class>#<Method> <Message> (<Exception>). More details on configuring and using the logging framework can be found in <http://docs.oracle.com/javase/1.5.0/docs/guide/logging/>.

57.1.3 Handling log messages with other logging frameworks

If your application already uses a different logging framework than the default Java logging framework, you may want to use that one instead, for handling any log messages coming from your application and from LuciadLightspeed. At initialization, LuciadLightspeed instantiates a global `ILcdLoggerFactory` that is then used to create all the `ILcdLogger` objects. You can replace the default `ILcdLoggerFactory`, in order to provide different `ILcdLogger` implementations. To do so, you can add a file called `META-INF/services/com.luciad.util.logging.ILcdLoggerFactory` to a directory or a jar on the classpath. This file should contain a single line with the name of your factory class. For example:

```
1 com.mycompany.MyLog4JLoggerFactory
```

Program 287 - Replacing the logging framework backend

LuciadLightspeed already provides several implementations of the `ILcdLoggerFactory` and `ILcdLogger` interfaces:

- `TLcdJULLoggerFactory` (the default). Creates `ILcdLogger` instances that delegate to the standard Java logging framework (`java.util.logging`).
- `TLcdNullLoggerFactory`. Creates `ILcdLogger` instances that discard all log messages. Use this implementation if you wish to maximize the performance of LuciadLightspeed.
- `TLcdSimpleLoggerFactory`. Creates `ILcdLogger` instances that send the error/warning/info messages to the console and discard all the others.

57.2 Performance monitoring using JConsole

JConsole is an application that comes bundled with Sun's Java Development Kit. It allows you to connect to a running Java Virtual Machine and display information about its performance. For example, you can monitor the memory usage, CPU load, number of threads, class loading, and so on. Moreover, you can remotely invoke operations on the running JVM, like triggering the garbage collection.

When you start JConsole (command `jconsole`), it displays a list of running Java processes, and invites you to choose the one you wish to monitor.

LuciadLightspeed leverages the logging and MBean frameworks to provide information about the performance of some of its components. First of all, you have to enable performance logging in your Java logging properties file. The logger tracing the performance (execution time) of a class is called `performance.com.luciad.<package>. <class_name>`. Program 288 shows a sample of typical logging properties. The handler `TLcdMBeanPerformanceLogHandler` provides the bridge between the performance measurements and JConsole.

```
1 # Performance monitoring
```

```

2 performance.com.luciad.level = FINEST
3 performance.com.luciad.useParentHandlers = false
4 performance.com.luciad.handlers = com.luciad.util.logging.TLcdMBeanPerformanceLogHandler
5 com.luciad.util.logging.TLcdMBeanPerformanceLogHandler.level = FINEST

```

Program 288 - Logging properties for performance monitoring

Once performance logging has been enabled, you can start up JConsole and connect to your application. The information explicitly exported by the application can be found under the last tab (MBeans). The left panel displays the hierarchy of available MBeans. An example of information that can be exported by a LuciadLightspeed application is presented in Figure 155.

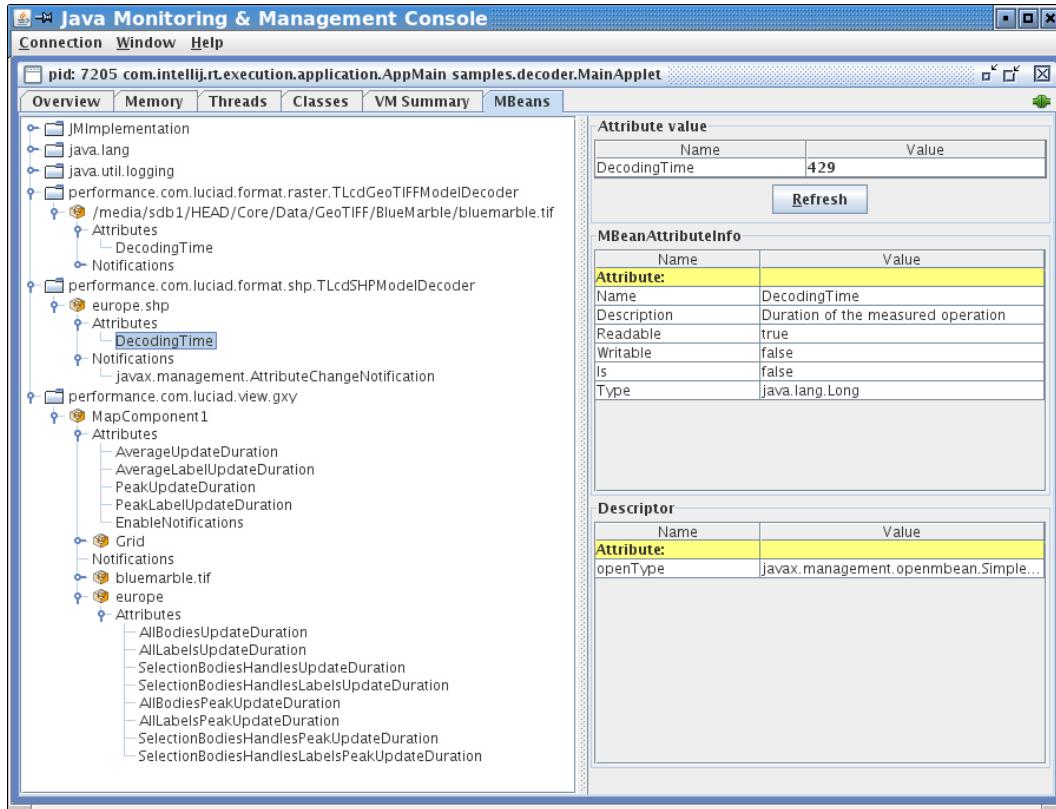


Figure 155 - Performance data displayed in JConsole

For each model that has been decoded by LuciadLightspeed, a new entry is added under the corresponding decoder's name. For example, if you have opened the file `europe.shp`, you will see an `europe.shp` entry appearing under `performance.com.luciad.format.shp.TLcdSHPModelDecoder`. The entry contains a single attribute: the decoding time of the model expressed in milliseconds.

Under the hierarchy `performance.com.luciad.view.gxy`, you will also observe one entry for each map view of the application (`MapComponent1` in the image above) and for their layers. The attributes describe the average and the peak view update times in milliseconds, computed over the last second. The underlying communication mechanism is a pull mechanism. You can click on the Refresh button at the bottom of the tab to update the displayed value. Double-clicking on the value itself opens a graph display that is automatically updated every second.

If you prefer to see the update times directly for each update, you can enable notifications, which provide an independent push mechanism. First, you have to let the view broadcast notifications, by changing the view's `EnableNotifications` attribute in the list of Attributes

to true. Then, you have to actually subscribe to these notifications, by clicking on the view's Notifications entry in the tree, and clicking on the Subscribe button at the bottom of the Notification buffer tab. Every time the view is repainted, a new entry is added to the notification buffer.

Note that currently performance monitoring is only possible when using the default TLcdJUL-LoggerFactory, since the performance monitoring MBean implements a JUL log handler.

57.3 Performance monitoring using Java VisualVM

Java VisualVM is the modernized version of JConsole in Sun's Java Development Kit, version 1.6 and higher. It provides the same functionality and more, with an improved user interface. You can start it with the command `jvisualvm`. For compatibility with JConsole, you first have to make sure the `VisualVM-MBeans` plugin is installed. You can check this in the menu Tools, Plugins, in tab Available Plugins.

As with JConsole, you first select the running application that you want to monitor, in this case by double-clicking on it in the tree in the left panel. You then get a similar MBeans tab, in turn containing Attributes and Notifications tabs.

CHAPTER 58

Configuring classes with properties files

This chapter describes what a Properties object is, in which cases it is used, and how to create a Property object and properties file.

58.1 Creating and initializing objects

In LuciadLightspeed, objects of some classes can be created and initialized using a `java.util.Properties` object. This `Properties` object contains the values to set for the properties of the object. A `Properties` object is essentially a hash table with keys referring to the names of the properties, and values containing the values for the corresponding properties.

In LuciadLightspeed, factory methods that *create* objects using this mechanism have the following signature pattern:

```
1  public static ILcd<ClassName>
2      create<ClassName>( String aPrefix, Properties aProperties )
3          throws IllegalArgumentException
```

Program 289 - create

In LuciadLightspeed, three classes follow this pattern: `TLcdGeodeticDatumFactory`, `TLcdModelReferenceFactory`, and `TLcdProjectionFactory`. Their `create` method creates an instance of the respective classes, and initializes its properties with the values defined by `aProperties`. The meaning of `aPrefix` will be explained shortly.

Interfaces and classes that *initialize* objects using the `Properties` mechanism have a method with the signature shown in Program 290.

```
1  public void loadProperties( String aPrefix, Properties aProperties )
2      throws IllegalArgumentException
```

Program 290 - loadProperties

Calling this method on an object sets the properties of that object to values specified in `aProperties`.

How a class interprets `aProperties` and how a factory uses `Properties` to create objects depends on the actual implementation of the class and the factory. So, the API reference of the `create` and method `loadProperties` must be consulted to make sure a correct prefix and

properties object are passed as arguments. If any of these arguments is erroneous, an `IllegalArgumentException` is thrown.

58.2 Serialization and the role of aPrefix

The `Properties` object must in some way serialize an object down to its primitive instance variables. Clearly, this is a recursive process as is illustrated in the following abstract example.

Suppose we have the two classes shown in Program 291.

```

1  public class T implements IT {
2      private IS fA, fB;
3      public IS getA() { □ }
4      public void setA( S aA ) { □ }
5      public IS getB() { □ }
6      public void setB( S aB ) { □ }
7      public void loadProperties( String aPrefix, Properties aProperties ) { □ }
8  }
9
10 public class S implements IS {
11     private int fNo;
12     public void setNo( int aNo ) { □ }
13     public int getNo() { □ }
14     public void loadProperties( String aPrefix, Properties aProperties ) { □ }
15 }
```

Program 291 - Example classes

A `Properties` file for an object of class `T` could have the entries shown in Program 292.

```

1  T.a.S.no= 1
2  T.b.S.no= 2
```

Program 292 - Example Entries

The corresponding implementation of the method `loadProperties` in class `T` could be as in Program 293.

```

1  public void loadProperties( String aPrefix, Properties aProperties ) {
2      fA.loadProperties( aPrefix + "T.a.", aProperties );
3      fB.loadProperties( aPrefix + "T.b.", aProperties );
4  }
```

Program 293 - Example implementation of `loadProperties` in T

The corresponding implementation of the method `loadProperties` in class `S` could be as in Program 294.

```

1  public void loadProperties( String aPrefix, Properties aProperties ) {
2      fNo = Integer.valueOf( aProperties.getProperty(
3                                     aPrefix + ".S.no" ) ).intValue();
4  }
```

Program 294 - Example implementation of `loadProperties` in S

The meaning of `aPrefix` should be clear from the example code. The class `S` cannot know what properties in other classes are instances of its class. Class `T` for example has two properties of

class S, and therefore two entries in the Properties file define its values. In order for class S to decode the correct entry, the prefix is used to indicate which entry to use for its initialization.

The pattern for Properties followed in this example is <className>.<propertyName>. This is the recommended scheme which is applied consistently in the LuciadLightspeed API. However, it does not necessarily have to be this way and it is up to the actual implementation to specify its Properties' naming scheme.

58.3 Creating Properties objects

In the previous examples, you can see that class S implements the interface IS. Suppose that you want to initialize an object of type T with new instances of type IS for its properties. A Properties file could be defined as in [Program 295](#).

```

1 T.a.IS.class= demo.S
2 T.a.S.no= 1
3 T.b.IS.class= demo.S
4 T.b.S.no= 2

```

Program 295 - Creating Objects

The entries T.a.IS.class= demo.S and T.b.IS.class= demo.S define that the class demo.S is to be used to create an instance of type IS for the properties a and b of the class T. Again, it is up to the actual implementations to specify how to code this creational behaviour in a Properties object, but the LuciadLightspeed API follows the pattern <Interface>.class to specify the class to use to create an instance of an Interface for a given property.

58.4 Using a Properties file

The previous sections have already referred to the use of a Properties file to define properties. The java.util.Properties has a method load to load properties from a given File into a Properties object. This mechanism is used extensively by LuciadLightspeed to specify properties of ILcdModelReference, ILcdProjection, and ILcdGeodeticDatum objects, which are necessary for the proper definition of ILcdModel objects from data sources.

58.4.1 Sample properties file

[Program 296](#) shows a Properties file for an ILcdModelReference.

```

1 ILcdModelReference.class = com.luciad.reference.TLcdGridReference
2 TLcdGridReference.falseNorthing = 0
3 TLcdGridReference.falseEasting = 500000
4 TLcdGridReference.scale = 0.9996
5 TLcdGridReference.unitOfMeasure = 1
6 TLcdGridReference.projection.ILcdProjection.class =
7     com.luciad.projection.TLcdMercator
8 TLcdGridReference.projection.TLcdMercator.centralMeridian =-75.0
9 TLcdGridReference.geodeticDatum.ILcdGeodeticDatumFactory.class =
10    com.luciad.geodesy.TLcdJPLGeodeticDatumFactory
11 TLcdGridReference.geodeticDatum.TLcdJPLGeodeticDatumFactory.geodeticDatumName =
12      WGS84

```

Program 296 - An example of a Properties file

CHAPTER 59

Running in OSGi™

LuciadLightspeed is compatible with the OSGi framework. The OSGi framework is a module system and service platform for the Java programming language. The *Eclipse Rich Client Platform*, for instance, uses it as its module system. This means that you can use LuciadLightspeed in applications which are based on the *Eclipse Rich Client Platform*, for instance.

For more information about OSGi, see <http://www.osgi.org>.



To use the LuciadLightspeed OSGi bundles in Eclipse, you need to add the *Eclipse Plug-in Development Environment* software to your Eclipse installation.

59.1 Creating bundles with the `make osgi bundles` script

Before you can use LuciadLightspeed within an OSGi framework, you must create the necessary OSGi bundles for LuciadLightspeed. These are JAR files with additional, OSGi-specific manifest headers. To create the bundles, run the `make osgi bundles` script in the `build/osgi` directory of your LuciadLightspeed installation.

The script creates the OSGi bundles in the `distrib/osgi` directory. You can add these bundles to your OSGi framework. In Eclipse, for instance, you can do this by editing the target platform. Click the Add button in the *Window → Preferences → Plug-in Development → Target Platform* window and add the `distrib/osgi` directory to the target platform.

The `make osgi bundles` script runs a Java application to make the OSGi bundles. The source files for the Java application are available in the `src` directory in the `build/osgi` directory. This way you can inspect and, if required, modify the way the script works.

59.2 Running the OSGi sample

The `build/osgi` directory also contains the directory `com.luciad.osgi.sample`. This directory contains a sample bundle that demonstrates how to develop applications based on OSGi and LuciadLightspeed.

The sample is similar to the `lightspeed.fundamentals.step1` sample. The difference is that it is now delivered as an OSGi bundle. It includes the required information in the `MANIFEST.MF` file:

- The LuciadLightspeed packages used by the sample are specified in the `Import-Package` header.

- The Activator of the bundle is specified in the `Bundle-Activator` header. The activator is responsible for actually starting the sample.

Luciad's Lucy product also contains a sample that integrates into the *Eclipse Rich Client Platform*. If you have Lucy, you can consult the `Lucy_EclipseIntegrationGuide` for a larger example of how to integrate LuciadLightspeed into an OSGi application.

59.2.1 Running the sample from the command prompt

To start the sample, run the `start_felix` script in the `distrib` directory. This script starts the Apache Felix OSGi runtime. The runtime is configured to load the LuciadLightspeed OSGi bundles and start the sample.

59.2.2 Running the sample from Eclipse

The sample directory is also a full Eclipse project. You can import it into Eclipse as an *Existing Project*, using Eclipse's `File → Import...` menu item.

You can run the sample by creating a run configuration of the 'OSGi Framework' category. To run the sample without Eclipse-specific errors, you need to disable and enable specific plug-ins in the 'Bundles' tab of the run configuration.

To enable the appropriate plug-ins in the run configuration:

1. In the 'Bundles' tab, click 'Deselect All'.
2. Enable the 'com.luciad.osgi.sample' bundle in the 'Workspace' node in the list of bundles.
3. Click the 'Add Required Bundles' button.
4. Enable the 'org.eclipse.equinox.console' bundle.
5. Enable all bundles starting with 'org.apache.felix.gogo'.

59.3 Created bundles are fragments, use Require-Bundle and rely on boot delegation

The `make osgi bundles` script converts all regular LuciadLightspeed jars to OSGi *Fragments* of the `com.luciad.osgi` bundle. All OSGi Fragments logically become a part of their host bundle. This means that they all share the same classloader. LuciadLightspeed uses this approach because it was not built from the ground up with OSGi in mind.

The `make osgi bundles` script also creates a `thirdparty` bundle, which contains all the third-party libraries used by LuciadLightspeed. The fragments use the `Require-Bundle` OSGi header to import the classes from these libraries. LuciadLightspeed uses this approach because not all third-party dependencies work well in the alternative setup, in which each library is a bundle on its own.

For the same reasons, the bundles created by the `make osgi bundles` script rely on the OSGi boot delegation feature to import the classes from the JRE. You need to configure your OSGi runtime to allow boot delegation for all classes. The startup script of the OSGi sample described in this chapter shows an example for the Apache Felix runtime.

CHAPTER 60

Integrating a Lightspeed view in C/C++/C# applications

The class `TLspExternalView` is intended to facilitate the integration of a Lightspeed view in applications written in a language other than Java, such as C or C++. In a nutshell, `TLspExternalView` is a Lightspeed view implementation which, rather than creating an OpenGL context of its own, renders into an externally supplied and managed context.

More information about the use of `TLspExternalView`, along with a complete sample program written in C++, can be found in the article "How to integrate LuciadLightspeed in a C++ application" on the Luciad Developer Platform (<https://dev.luciad.com>).

PART X Appendices

APPENDIX A

Upgrading LuciadLightspeed

This chapter explains how patch updates of LuciadLightspeed are binary compatible.

Secondly, it offers upgrading tips that help you integrate new versions of LuciadLightspeed in your existing development project.

A.1 Compatibility of LuciadLightspeed versions

This section discusses in what way LuciadLightspeed patch updates are binary compatible, and how such binary compatibility can benefit you.

The terms Major version, Minor version, and Patch version refer to the components of a LuciadLightspeed version number. Typically, these numbers are composed as a `Major.Minor.Patch` sequence. For instance, in LuciadLightspeed version 2014.0.02, 2014 indicates the Major version, 0 indicates the Minor version, and 02 indicates the Patch version.

A.1.1 What is LuciadLightspeed binary compatibility?

As of version 2014, LuciadLightspeed patch updates are binary compatible. This means that you can compile your development project with a version of LuciadLightspeed, install a patch update of LuciadLightspeed based on that version, and then run your project without re-compiling it first. For instance, you can integrate LuciadLightspeed patch update 2014.0.01 in your development project compiled with version 2014.0, and immediately run the project with the patch update.

Patch releases of LuciadLightspeed are binary compatible with their initial minor version only. For instance, LuciadLightspeed versions **2014.0.01** and **2014.0.02** will be binary compatible with version **2014.0**, but version **2014.0.02** will not be binary compatible with version **2014.1**, nor will version **2015.1.01** be binary compatible with version **2015.0**.

Versions of LuciadLightspeed are not necessarily binary compatible with LuciadLightspeed releases of other minor versions. For instance, LuciadLightspeed version 2014.0 can be binary incompatible with LuciadLightspeed version 2014.1 or version 2015.0.



There may be occasional exceptions to these LuciadLightspeed binary compatibility principles. These are explained in detail in the release notes of LuciadLightspeed.

The Java Language Specification describes in great technical detail what binary compatibility means for Java libraries. You can find this specification on <http://docs.oracle.com/>

<javase/specs/jls/se7/html/jls-13.html>.

A.1.2 Benefits of binary compatibility

Binary compatibility is particularly useful when you work with multiple teams on a single project, or when you have third-party contributors providing components for your application. In those cases, it is possible for your team to use the compiled JAR files provided by other contributors. You will be able to use that JAR in your own application, even if the other contributor used a different patch version of the LuciadLightspeed than the one you are currently using.

A.2 Project upgrading guidelines

Find out what steps you can take to make a LuciadLightspeed upgrade progress as smoothly as possible.

If you are developing an application based on the Lucy component, it is recommended to start preparing for future product upgrades by setting up your Lucy development project in a specific way. To find out more, see the chapter about *Upgrading your Lucy development project* in the Lucy developer's guide.

A.2.1 Read the release notes

When you install an upgrade, read all information pertaining to your LuciadLightspeed version in the release notes. The release notes help you figure out where you might need to update your own code.

For each LuciadLightspeed version, the release notes provide an overview of:

- Enhancements
- Fixes
- Upgrade considerations

for each LuciadLightspeed version.

To find the release notes, open the HTML documentation that comes with LuciadLightspeed: on the application launcher (`start.jar`), click **Documentation**, and select the **Release Notes** page from the navigation bar.

A.2.2 Document your sample code usage

A LuciadLightspeed upgrade usually comes with a range of updates and improvements of sample code. If you do not keep track of the sample code snippets you have copy-pasted for re-use, it may become a challenge to benefit from these improvements in your own application.

Therefore, keep track of all the sample code copies you take, and the changes you apply to those copies. You can do this by adding copy markers at those spots where you re-use sample code in your own application code, for instance. You could also copy the Luciad copyright statements along with the sample code, and use those as copy markers. In addition, mark any changes you made to the code copies, or rely on your versioning system to track the changes.

When you receive a new product version, use a file comparison tool to find the differences between the sample code of the previous distribution and the new LuciadLightspeed version, and verify if and how those differences affect you.

A.2.3 Upgrading dependencies

During a LuciadLightspeed upgrade, a third-party library included in the LuciadLightspeed version may have been upgraded as well. As a result, some of the LuciadLightspeed code may have changed. If you also coded directly against that third-party library API, and the library changes are incompatible, you may need to change your code as well. Note that LuciadLightspeed never requires you to do so.

To find out which third-party libraries have been upgraded, check the **Prerequisites** pages for LuciadLightspeed. The **Libraries** section lists the third-party libraries used with LuciadLightspeed and its components, along with their version number. To find the **Prerequisites** page, open the HTML documentation that comes with the product: on the application launcher (`start.jar`), click **Documentation**, and select **Prerequisites** from the navigation bar.

When you have established that a third-party library has been upgraded, go to the website of the library provider to find out which improvements and possibly incompatible changes have been introduced by the library upgrade.

It is strongly discouraged to upgrade a third-party library yourself to benefit from a certain bug fix or improvement. LuciadLightspeed may rely on third-party library features that become backward-incompatible after a library upgrade.

APPENDIX B

Troubleshooting for Lightspeed views

This chapter lists general troubleshooting tips and known issues that you may encounter while working with OpenGL or OpenCL based components of LuciadLightspeed.

B.1 Graphics drivers

Bugs in graphics drivers can have a serious impact on the performance and/or stability of your LuciadLightspeed application. If you experience problems, it is therefore always strongly advisable to install the most recent available drivers for your graphics card.

B.2 Known issues

The following are known issues that may be encountered when using LuciadLightspeed applications on specific hardware platforms:

ATI/AMD or Intel on Linux The use of ATI/AMD or Intel graphics hardware on Linux is not recommended, due to driver issues.

NVIDIA Optimus Notebooks equipped with NVIDIA Optimus technology actually contain both a dedicated Nvidia graphics card and an integrated Intel graphics chip. The system switches back and forth between these at runtime to preserve battery power. To prevent your application from running on the Intel graphics chip and achieving suboptimal performance, it may be necessary to disable the Optimus technology. You can do this in the system's BIOS settings.

Others Some graphics chips do not perform line smoothing if the lines are drawn during the view's transparency rendering phase, which results in reduced quality. Notably NVIDIA GeForce cards have this problem, whereas the Quadro line does not. To overcome this problem, you can use multi-sample buffers, although this will lead to an increase in memory consumption. For more information about this topic, see [Section 54.3.3](#).

APPENDIX C

OpenGL

If you are developing an application based on a Lightspeed view, the application will make use of OpenGL. This chapter discusses how and why LuciadLightspeed makes use of the OpenGL API. It also explains how to use the OpenGL tools offered by LuciadLightspeed to help you develop OpenGL-based applications efficiently.

The painters in Lightspeed view applications allow you to work directly with the OpenGL API by offering access to an OpenGL context. [Section C.2](#) provides more information about that OpenGL connection. [Section C.3](#) lists a number of resources that can help you familiarize with the OpenGL API. [Section C.4](#) and [Section C.5](#) explain how you can generate reports about the capabilities and compatibility of your platform and your OpenGL application components, while [Section C.6](#) assists with OpenGL debugging.

C.1 What is OpenGL?

OpenGL is a cross-platform standard API for hardware-accelerated 3D rendering. The software runtime library ships with all Windows, Mac OS, Linux and Unix systems.

Two important OpenGL terms you will come across in this chapter are:

OpenGL profile An OpenGL profile corresponds to a version of the OpenGL specification.

OpenGL extensions OpenGL extensions are extensions to the OpenGL specifications, typically proposed by graphics hardware vendors, to support new features of their hardware. These extensions may be included in the next version of the specification to become a standard feature.

C.2 JOGL2 OpenGL binding supported by LuciadLightspeed

OpenGL bindings allow Java applications to talk to the OpenGL runtime installed on the underlying operating system. As part of this connection process, the bindings provide a link between the native windowing system and Java Swing/AWT toolkits. LuciadLightspeed supports version 2 of the JOGL OpenGL binding.

Support for OpenGL bindings is achieved through the use of an abstraction layer, defined in the package `com.luciad.view.opengl.binding`.

To make an abstraction of how OpenGL support is achieved through the binding, and to remain independent of the used binding, LuciadLightspeed lets users access OpenGL functionality

through the `ILcdGLDrawable` OpenGL context. An `ILcdGLDrawable` offers an `ILcdGL` instance, which fully implements the OpenGL API.

For more information about the usage of the OpenGL API, see the reference documentation for `ILspPainter` and `ILcdGLDrawable`.

C.3 Additional information on OpenGL

There are several books on OpenGL available, but the two most revered ones are the “red book” and the “blue book”:

- OpenGL Programming Guide: The Official Guide to Learning OpenGL, Seventh Edition, Dave Shreiner et al. ISBN 978-0321552624 (also known as the red book). This book offers a tutorial, with each chapter focusing on an important part of the OpenGL functionality.
- OpenGL Reference Manual, Fourth Edition, Dave Shreiner (Editor), et al. ISBN 032117383X (also known as the blue book). This book is an API reference manual.
- OpenGL Shading Language. Second Edition, Randi J. Rost. ISBN 978-0321334893 (also known as the orange book). This book provides tutorial as well as reference information on the OpenGL Shading Language.

There are also several OpenGL websites that may be of interest. The main online sources of information are:

- <http://www.opengl.org/> is the official OpenGL website. It provides access to the latest versions of the OpenGL specification, as well as active development forums and numerous links to other OpenGL development resources.
- <http://www.opengl.org/registry/> is a registry containing the specifications for OpenGL and OpenGL extensions.

C.4 Generating a report of the hardware and OpenGL specifications of your development platform

`LuciadLightspeed` allows you to check the capabilities of your system by offering the `TLspPlatformInfo` utility class. This class lets you generate a detailed specifications report on the hardware, OpenGL and OpenCL capabilities of the platform on which a view is running. As explained in [Section 2.3](#), OpenCL can also be used in `LuciadLightspeed`, to perform computations on the GPU.

The report is returned as a `String` when you call the `getReport` method. It contains information related to the following items:

- Operating system
- OpenGL basic information: vendor, renderer, version
- OpenGL available extensions
- OpenGL state variables
- OpenCL basic information: vendor, version
- OpenCL available extensions
- OpenCL device information

The `samples.lightspeed.debug.report` sample uses this utility class to automatically generate a capability report. It gives you the option to save this report to disk.

For more information about LuciadLightspeed system requirements, see the LuciadLightspeed technical notes.

C.5 Checking OpenGL component compatibility with your target platform

Components that have access to OpenGL through the `ILcdGLDrawable` interface can test the compatibility of that `ILcdGLDrawable` against a certain OpenGL profile through the `TLspOpenGLProfile` class.

The `TLspOpenGLProfile` class allows you to define an OpenGL profile with a required version number and required extensions through its constructor. Furthermore, you can set up another `TLspOpenGLProfile` to serve as the baseline for the current profile. `TLspOpenGLProfile` reports whether a given `ILcdGLDrawable` is compatible through the `isCompatible` method.

A number of ready-made profiles are available as static constants in `TLspOpenGLProfile`. This includes the `TLspOpenGLProfile.LIGHTSPEED_MINIMUM` profile, which defines the minimum OpenGL profile needed to run LuciadLightspeed.

C.6 Debugging OpenGL errors

LuciadLightspeed provides the `TLcdGLDebugGL` class to help you localize errors related to OpenGL. This class implements the `ILcdGL` interface. It is able to wrap an underlying `ILcdGL` implementation, and checks for errors after each OpenGL method call. As soon as an OpenGL error is found, it throws a `TLcdGLEException`, which allows you to locate the source of any OpenGL-related error quickly.

APPENDIX D

Using LuciadLightspeed in a Maven repository

This chapter explains how you can make LuciadLightspeed available through Apache Maven. Maven is a development project management and build automation tool. More info is available on its website: <http://maven.apache.org/>.

D.1 Deploying to a repository

Luciad provides POM files for all libraries available in LuciadLightspeed. Those are available in the `build/maven/pom` directory. They can easily be deployed to your repository by running the Ant script `deploy.xml` in the `build/maven` directory. The script asks for a URL and credentials to the Maven repository, and proceeds to deploy all available libraries.



The deployment script requires at least Ant 1.9.1 and the Maven tasks `antlib`. Ant 1.9.4 and version 2.1.3 of the Maven tasks are provided in the `build/ant` directory of the LuciadLightspeed installation. Support for protocols other than `file` and `http` is available through Wagon (<http://maven.apache.org/wagon/>).

D.2 Using the libraries

You can use the Luciad libraries like any other Maven dependency by adding them to your project POM.

```
1 <dependency>
2   <groupId>com.luciad</groupId>
3   <artifactId>lcd_lightspeed</artifactId>
4   <version>${luciad.version}</version>
5 </dependency>
```

Program 297 - Adding a Luciad dependency

It is recommended to define a Maven property for the LuciadLightspeed version as a value placeholder, to prevent an accidental mixing of versions. As a result, you only have to update the version number once when you upgrade to a new LuciadLightspeed version.

D.2.1 Adding a license

A license for LuciadLightspeed is not automatically uploaded to the repository. You can do so manually by means of Maven's deploy-file goal, as in the example below.

```
1 mvn deploy:deploy-file \
2   -DgroupId=com.yourcompany \# do not use com.luciad
3   -DartifactId=luciad_development_license \\
4   -Dversion=2016.0-20160121 \\ # for instance the date that the license was
received
5   -Dpackaging=jar \\
6   -Dfile=development.jar \\
7   -Durl=yourCompanyRepositoryUrl
```

Program 298 - Adding a license

In your project's pom.xml, you should then specify:

```
1 <dependency>
2   <groupId>com.yourcompany</groupId>
3   <artifactId>luciad_development_license</artifactId>
4   <version>2016.0-20160121</version>
5 </dependency>
```

Program 299 - Adding a license

APPENDIX E

Supported data formats

The tables in this appendix list the data formats that are supported by LuciadLightspeed or one of its components.

Format	Decoder	Encoder	Component	GXY view	Lightspeed view*	Notes	Type	File extensions
ACE2	x		Advanced Raster Connectors (GDAL)	x	x		Raster	*.ace2
AIXM 3.3 and 4.5	x		Aviation Standards	x	x		Vector	*.xml
AIXM 5.1	x	x	Aviation Standards	x	x		Vector	*.xml, *.aixm5, *.aixm51
AML (STANAG 7170)	x	x	Maritime Standards	x	x		Vector	*.00, *.031
ArcSDE raster	x	x	ArcSDE	x	x	Requires legacy ArcSDE component.	Raster	*.agr

Format	Decoder	Encoder	Component	GXY view	Lightspeed view*	Notes	Type	File extensions
ArcSDE vector	x	x	ArcSDE	x	x	Requires legacy ArcSDE component.	Vector	*.agv
ARINC 424	x		Aviation Standards	x	x		Vector	*.txt, *.pc, FAA*, *ari, *.dat
ASDI/ETMS	x		ASDI	x	x	Requires Real-time Engine	Vector	*.asdi
ASRP/USRP/ ADRG	x		Defense Standards	x	x		Raster	*01.THF, *01.GEN, *01.QAL, *SOU, *.IMG
AutoCAD DWG and DXF	x		CAD Connectors	x		supported up to version 2013	Vector	*.dwg, *.dwg.zip, *.dwg.gz, *.dxf, *.dxf.zip, *.dxf.gz
Azavea Raster Grid	x		Advanced Raster Connectors (GDAL)	x	x		Raster	*.arg, *.json
BCI	x		Defense Standards	x	x		Raster	*.matrxmap
BigTIFF	x	x	LuciadLightspeed	x	x		Raster	*.tif, *.tiff
Bing Maps	x		LuciadLightspeed Essential	x	x	Microsoft Imagery Service	Raster	
BIL	x		LuciadLightspeed Essential	x	x		Raster	*.bil

Format	Decoder	Encoder	Component	GXY view	Lightspeed view*	Notes	Type	File extensions
BMP (and BPW)	x		LuciadLightspeed Essential**	x	x	Requires additional reference and positioning files	Raster	*.bmp, *.bpw
BSB Nautical Chart	x		Advanced Raster Connectors (GDAL)	x	x		Raster	*.kap
CADRG, CIB	x		Defense Standards	x	x		Raster	*.toc (CADRG)
CGM	x		LuciadLightspeed Essential	x			Symbology	*.cgm, *.cgm.gz, *.cgm.zip
Collada 1.4.1	x		LuciadLightspeed Essential		x	Visualization as 3D icons or embedded in KML	3D icons	*.dae
Convair Pol-GASP	x		Advanced Raster Connectors (GDAL)	x	x		Raster	*.img
CSV	x		Comma Separated Value	x	x		Vector	*.csv, *.tsv
DAFIF, DAFIFT	x		Aviation Standards	x	x		Vector	
DB2 Spatial and Geodetic Extender	x	x	Database Connectors	x	x		Vector	*.db2
DMED, DTED	x		LuciadLightspeed Essential**	x	x		Elevation	*.dmed, *.dt0, *.dt1, *.dt2

Format	Decoder	Encoder	Component	GXY view	Lightspeed view*	Notes	Type	File extensions
DNG	x	x	LuciadLightspeed Essential	Not applicable	Not applicable	Metadata format based on ISO metadata standards ISO 19115 and ISO 19139.	Metadata	
ECRG	x		Defense Standards **	x	x	Includes JPEG2000 decoder	Raster	TOC.xml
ECW	x		Advanced Raster Connectors**	x	x		Raster	*.ecw, *.ers
ELAS DIPEX	x		Advanced Raster Connectors (GDAL)	x	x		Raster	
ENVI .hdr Labelled Raster	x		Advanced Raster Connectors (GDAL)	x	x		Raster	*.hdr
EOSAT FAST Format	x		Advanced Raster Connectors (GDAL)	x	x		Raster	
ERDAS Imagine	x		Advanced Raster Connectors (GDAL)	x	x		Raster	*.img
ERDAS Imagine Raw	x		Advanced Raster Connectors (GDAL)	x	x		Raster	*.img
ESRI ArcInfo ASCII Grid	x		LuciadLightspeed Essential **	x	x		Raster	*.asc, *.grd

Format	Decoder	Encoder	Component	GXY view	Lightspeed view*	Notes	Type	File extensions
ESRI ArcInfo Bi-nary Grid	x		Advanced Raster Connectors (GDAL)	x	x		Raster	*.adf
ESRI ArcInfo Export E00 Grid	x		Advanced Raster Connectors (GDAL)	x	x		Raster	*.e00
ESRI Shape	x	x	LuciadLightspeed Essential	x	x		Vector	*.shp, *.shp.gz, *.shp.zip
ESRI TFW, JGW	x		LuciadLightspeed Essential	x	x		Georeference	*.tfw, *.jgw
ETOPO	x		LuciadLightspeed Essential	x	x		Raster	*.ETOPO2.raw.bin, *.ETOPO2v2_MSP.raw, ETOPO5.DAT
Eurocontrol As-terix	x	x	Radar Connectors	x	x	Requires Real-time Engine	Raster	*.ast, *.asterix, *.astfin, *.pcap
GeoPackage	x	x	LuciadLightspeed Essential	x	x	OGC GeoPack-age	Vector, raster	*.gpkg
GDAL Virtual	x		Advanced Raster connectors (GDAL)	x	x		Raster	*.vrt
GDF	x		Graph & Routing Engine	x			Vector	*.gdf
GeoJSON	x	x	LuciadLightspeed Essential	x	x		Vector	*.json, *.json.gz, *.json.zip, *.geojson
Geospatial PDF	x		Advanced Raster Connectors **	x	x		Raster	*.pdf

Format	Decoder	Encoder	Component	GXY view	Lightspeed view*	Notes	Type	File extensions
GeoSPOT	x		Advanced Raster Connectors	x	x		Raster	*.DSC, *.REP
GeoTIFF	x	x	LuciadLightspeed Essential**	x	x	Multi-spectral and HDR images supported	Raster	*.tif, *.tiff
GIF	x		LuciadLightspeed Essential	x	x	Requires additional reference and positioning files	Raster	*.gif
Golden Software ASCII Grid	x		Advanced Raster Connectors (GDAL)	x	x		Raster	
Golden Software Binary Grid	x		Advanced Raster Connectors (GDAL)	x	x		Raster	
Golden Software Surfer 7 Binary Grid	x		Advanced Raster Connectors (GDAL)	x	x		Raster	
Google Earth Enterprise globe	x		Beta release of Google Earth Enterprise component	x	x		Raster	*.kiasset (directory)
GRASS ASCII Grid	x		Advanced Raster Connectors (GDAL)	x	x		Raster	

Format	Decoder	Encoder	Component	GXY view	Lightspeed view*	Notes	Type	File extensions
GRIB V1/V2	x		Weather & Environment Standards **	x	x		Raster	*.grb, *.grib2, *.grb2, *.dat
GXF Grid eX-change File	x		Advanced Raster Connectors (GDAL)	x	x		Raster	*.gxf
HF2/HFZ heightfield raster	x		Advanced Raster Connectors (GDAL)	x	x		Raster	*.hf2
Informix Geodetic Datablade	x	x	Database Connectors	x	x		Vector	*.ifx
Informix Spatial Datablade	x	x	Database Connectors	x	x		Vector	*.isd
ISO 19115 and ISO 19139	x	x	LuciadLightspeed Essential	Not applicable	Not applicable	ISO metadata standards.	Metadata	
JPEG (and JGW)	x		LuciadLightspeed Essential**	x	x	Requires additional georeference and positioning files	Raster	*.jpg
JPEG2000	x		LuciadLightspeed Essential**	x	x	Multi-spectral and HDR images supported	Raster	*.jp2, *.j2k
JPEG2000		x	Advanced Raster Connectors	x	x		Raster	
JPIP	x		LuciadLightspeed Essential	x	x		Raster	

Format	Decoder	Encoder	Component	GXY view	Lightspeed view*	Notes	Type	File extensions
KML/KMZ	x	x	LuciadLightspeed Essential (KML-SUPEROVER-LAY**)	x	x		Vector, raster	*.kml, *.kmz
LASeR/LASzip	x		LuciadLightspeed Essential		x		Lidar	*.las, *.laz
LRDB	x	x	LuciadLightspeed Essential	x	x	LuciadMobile Raster Database	Raster	*.lrdb
LVDB	x	x	LuciadLightspeed Essential	x	x	LuciadMobile Vector Database	Vector	*.lvdb
LuciadLightspeed JAI	x		LuciadLightspeed Essential	x	x		Raster	*.jai
LuciadLightspeed RST	x		LuciadLightspeed Essential	x	x	Georeference and positioning file	Georeference	*.rst
Magellan BLX TOPO	x		Advanced Raster Connectors (GDAL)	x	x		Raster	*.blx, *.xlb
MapInfo MAP	x		LuciadLightspeed Essential	x	x		Vector	*.map
MapInfo MIF	x	x	LuciadLightspeed Essential	x	x		Vector	*.mif, *.map, *.mif.gz, *.map.gz, *.mif.zip, *.map.zip
MapInfo TAB	x		LuciadLightspeed Essential	x	x		Raster	*.tab

Format	Decoder	Encoder	Component	GXY view	Lightspeed view*	Notes	Type	File extensions
Microsoft SQL Server	x	x	Database Connectors	x	x	Support for version 2008-2012	Vector	*.mss
Microstation DGN	x		CAD Connectors	x		Support for DGN V7 and V8	Vector	*.dgn, *.dgn.zip, *.dgn.gz
MrSID	x		Advanced Raster Connectors	x	x		Raster	*.sid
NASA ELAS	x		Advanced Raster Connectors (GDAL)	x	x		Raster	
NetCDF	x		Weather & Environment Standards **	x	x		Raster	*.nc, *.netcdf, *.xml(NcML file)
NITF/NSIF (STANAG 4545)	x		Defense Standards **	x	x	Includes JPEG2000 decoder. Multi-spectral and HDR images supported	Raster	*.ntf
NOAA .gtx vertical datum shift	x		Advanced Raster Connectors (GDAL)	x	x		Raster	*.gtx
NVG	x	x	Defense Symbology	x	x		Vector	*.nvg, *.xml
OGC 3D Tiles	x		LuciadLightspeed Essential		x		3D meshes and point clouds	tileset.json

Format	Decoder	Encoder	Component	GXY view	Lightspeed view*	Notes	Type	File extensions
OGC GML 2 and 3.1.1	x	x	LuciadLightspeed Essential	x	x	Extensible encoder/decoder	Vector	*.gml, *.gml31, (*.xml)
OGC GML 3.2.1/ISO 19136	x	x	LuciadLightspeed Essential	x	x	Extensible encoder/decoder	Vector	*.gml, *.gml32, (*.xml)
OpenFlight	x		LuciadLightspeed Essential		x		3D icons	*.flt
Oracle Locator	x	x	Database Connectors	x	x		Vector	*.ora
Oracle Spatial (GeoRaster)	x		Database Connectors	x	x		Raster	*.ogr
OpenSceneGraph Binary	x		LuciadLightspeed Essential		x		3D meshes	*.osgb
PNG	x		LuciadLightspeed	x	x	Requires additional georeference and positioning files	Raster	*.png
POL	x	x	LuciadLightspeed Essential	x			Vector	*.pol
PostgreSQL PostGIS	x	x	Database Connectors	x	x		Vector	*.pgs
PPM	x		LuciadLightspeed Essential	x	x	Requires additional georeference and positioning files	Raster	*.ppm

Format	Decoder	Encoder	Component	GXY view	Lightspeed view*	Notes	Type	File extensions
S-52	Not applicable	Not applicable	Maritime Standards	x	x		Symbology	Not applicable
S-57	x	x	Maritime Standards	x	x		Vector	*.00*, *.031
S-63 Decryption	x		S-63	x	x	Requires Maritime Standards component	Vector	*.00*, *.031
SAP HANA	x	x	SAP HANA	x	x	Beta release	Database	
SGI Image Format	x		Advanced Raster Connectors (GDAL)	x	x		Raster	*.bw, *.rgb, *.rgba, *.sgi
Snow Data Assimilation System	x		Advanced Raster Connectors (GDAL)	x	x		Raster	*.hdr
Spot DIMAP	x		Advanced Raster Connectors **	x	x		Raster	*.dim
SQLite SpatiaLite	x	x	LuciadLightspeed Essential	x	x		Database	*.spl
SRTM HGT Format	x		Advanced Raster Connectors (GDAL)	x	x		Raster	*.hgt, *.DEM
SVG	x	x	LuciadLightspeed Essential	x			Vector	*.svg
Swiss DHM	x		Advanced Raster Connectors	x	x		Raster	*.mlt, *.mbl

Format	Decoder	Encoder	Component	GXY view	Lightspeed view*	Notes	Type	File extensions
TIFF (and TFW)	x		LuciadLightspeed Essential**	x	x	Requires additional georeference and positioning files	Raster	*.tif, *.tiff, *.tfw
USGS DEM	x		LuciadLightspeed Essential**	x	x		Elevation raster	*.dem, *.dem.gz, *.dem.zip
USGS DOQ First Generation	x		Advanced Raster Connectors (GDAL)	x	x		Raster	*.doq
USGS DOQ New Labelled	x		Advanced Raster Connectors (GDAL)	x	x		Raster	*.doq
USGS LULC Composite Theme Grid	x		Advanced Raster Connectors (GDAL)	x	x		Raster	
VPF (VMAP, VMAP2i DNC, UVMap,...)	x		Defense Standards	x	x	Includes Geosym symbology	Vector	Various extensions for various formats based on VPF.
WMS	x	Not applicable	LuciadLightspeed Essential	x	x	Web Map Service	Raster	Not applicable
WMTS	x	Not applicable	LuciadLightspeed Essential	x	x	Web Map Tile Service	Raster	Not applicable
WCS	x	Not applicable	LuciadLightspeed Essential	x	x	Web Coverage	Raster	Not applicable
WFS-T	x	x	LuciadLightspeed Essential	x	x	Web Feature Service	Vector	Not applicable

Format	Decoder	Encoder	Component	GXY view	Lightspeed view*	Notes	Type	File extensions
VTP Binary Terrain format	x		Advanced Raster Connectors (GDAL)	x	x		Raster	*.bt
Wavefront OBJ	x		LuciadLightspeed Essential		x		3D icons	*.obj

Table 14 - Supported data formats

* If a data format has a check mark in the Lightspeed view column, you can take full advantage of Lightspeed view features such as hardware acceleration while you are visualizing the format in a Lightspeed view. If a format does not have a check mark in the Lightspeed view column, you should still be able to visualize it in a Lightspeed view by means of the layer adapting tools offered by LuciadLightspeed. For more information, see [Chapter 41](#).

** Optionally supported by GDAL component as well.

E.I Model decoders and encoders

For all formats supported in LuciadLightspeed there is a model decoder available. The naming convention for the model decoders is: `TLcdFormatModelDecoder`, for example `TLcdSHPMModelDecoder`, `TLcdGeoTIFFModelDecoder`, and so on. The model encoders follow the same naming convention: `TLcdSHPMModelEncoder`, `TLcdGeoTIFFModelEncoder`, and so on. A special case is `TLcdGDALModelDecoder`, which can decode any raster format that GDAL can decode by default and for which Luciad doesn't have a proper model decoder. For more information on a specific model decoder or encoder, refer to the API reference.

In addition to the supported file formats as listed in [Table 14](#), LuciadLightspeed provides model decoders for:

- **Multilevel formats:** `TLcdMultilevelTiledModelDecoder` and `TLcdMultilevelRegularTiledModelDecoder` are available for multilevel models with irregular tiles at each level and with a regular grid of tiles respectively.
- **Magnetic field formats:** `TLcdIGRFModelDecoder` and `TLcdWMMModelDecoder` for International Geomagnetic Reference Field (IGRF) models and World Magnetic Models (WMM) respectively. These decoders provide support for magnetic variation (or magnetic declination) and isogonic lines.

APPENDIX F

Cookbook

This cookbook contains the programs that are based on the code snippets described in separate Fundamentals chapters of this guide. You can consult both the Lightspeed view programs and the GXY view programs.

F.1 Lightspeed view programs

- Program 300 shows how to build a basic LuciadLightspeed application with a Lightspeed view and a background layer.
- Program 301 shows how to load and display a set of flight plans.
- Program 302 shows how to load flight plan and way point data, and add both sets of data to a layer tree node.
- Program 303 shows how to add editing capabilities to a basic Lightspeed view.

F.1.1 Building a basic LuciadLightspeed application with a Lightspeed view

Program 300 contains the code snippets for building a basic application as described in Chapter 3. Figure 13 shows the window with the components of the basic application in a 3D view.

```
1 /**
2  * The sample demonstrates how to create and set up a 2D/3D view with some background data.
3  */
4 public class Main {
5
6     static {
7         // Set-up Swing to support heavy weight components, such as the TLspAWTView
8         JPopupMenu.setDefaultLightWeightPopupEnabled(false);
9         ToolTipManager.sharedInstance().setLightWeightPopupEnabled(false);
10    }
11
12    // The application frame
13    private JFrame fFrame;
14
15    /**
16     * Instantiates an TLspAWTView that can be added to our GUI.
17     * @return The created view.
18     */
19    private ILspAWTViewcreateView() {
20        ILspAWTView view = TLspViewBuilder.newBuilder().buildAWTView();
21
22        // Set layer factory of the view. When adding models to the view, this factory
23        // is used to create layers for those models.
24    }
```

```

25     view.setLayerFactory(createLayerFactory());
26
27     return view;
28 }
29
30 /**
31 * Creates the layer factory which is set to the view.
32 * @return The layer factory.
33 */
34 protected ILspLayerFactory createLayerFactory() {
35     return new BasicLayerFactory();
36 }
37
38 /**
39 * Creates and adds the layers that will be visible in the view.
40 * @param aView The view.
41 * @throws IOException In case of I/O failure.
42 */
43 protected void initLayers(ILspView aView) throws IOException {
44     // Create a TLcdEarthRepositoryModelDecoder to decode Luciad Earth repositories
45     ILcdModelDecoder earthDecoder = new TLcdEarthRepositoryModelDecoder();
46
47     // Decode a sample data set (imagery data)
48     ILcdModel earthModel = earthDecoder.decode("Data/Earth/SanFrancisco/tilerepository.cfg");
49
50     // Calling addLayersFor() will cause the view to invoke its layer factory with
51     // the given model and then add the resulting layers to itself
52     aView.addLayersFor(earthModel);
53
54     // TLcdSHPModelDecoder can read ESRI SHP files
55     ILcdModelDecoder decoder = new TLcdSHPModelDecoder();
56
57     // Decode world.shp to create an ILcdModel
58     ILcdModel shpModel = decoder.decode("Data/Shp/World/world.shp");
59
60     // Calling addLayers() will cause the view to invoke its layer factory with
61     // the given model and then add the resulting layers to itself
62     Collection<ILspLayer> shpLayer = aView.addLayersFor(shpModel);
63
64     fitViewExtents(aView, shpLayer);
65
66     // Create and add the grid layer
67     aView.addLayer(TLspLonLatGridLayerBuilder.newBuilder().build());
68 }
69
70 protected void fitViewExtents(ILspView aView, Collection<ILspLayer> aLayers) {
71     try {
72         // Fit the view to the relevant layers.
73         new TLspViewNavigationUtil(aView).fit(aLayers);
74     } catch (TLcdOutOfBoundsException e) {
75         JOptionPane.showMessageDialog(fFrame,
76             "Could not fit on layer, layer is outside the valid bounds"
77             );
78     } catch (TLcdNoBoundsException e) {
79         JOptionPane.showMessageDialog(fFrame,
80             "Could not fit on destination, no valid bounds found");
81     }
82 }
83
84 /**
85 * Initializes the controller of the view, responsible for the interaction with the
86 * end-user. Nothing is done here, meaning the view's default controller is used.
87 * Sub-classes can override this method.
88 * @param aView The view.
89 */
90 protected void initController(ILspView aView) {
91 }
92
93 /**
94 * Opens a JFrame containing our view, tool bar and layer control.
95 * @param aView The view.
96 */
97 private void initGUI(ILspAWTView aView) {

```

```

97     fFrame = new JFrame("Luciad Lightspeed Fundamentals")
98     {
99         @Override
100        public void dispose() {
101            aView.destroy(); // clean up the view
102            super.dispose();
103        }
104    };
105    fFrame.getContentPane().setLayout(new BorderLayout());
106    fFrame.getContentPane().add(aView.getHostComponent(), BorderLayout.CENTER);
107    fFrame.add(createToolBar(aView), BorderLayout.NORTH);
108    fFrame.add(new JScrollPane(createLayerControl(aView)), BorderLayout.EAST);
109    fFrame.setSize(800, 600);
110    fFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
111    fFrame.setVisible(true);
112 }
113
114 private Component createLayerControl(ILspView aView) {
115     return new TLcdLayerTree(aView);
116 }
117
118 protected JToolBar createToolBar(final ILspView aView) {
119     // Create and add toolbar to frame
120     JToolBar toolBar = new JToolBar();
121
122     // Create a button group for the radio buttons
123     ButtonGroup group = new ButtonGroup();
124
125     // Create a button to switch to 2D
126     JRadioButton b2d = new JRadioButton("2D", true);
127     b2d.setAction(new AbstractAction("2D") {
128         @Override
129         public void actionPerformed(ActionEvent e) {
130             TLspViewTransformationUtil.setup2DView(
131                 aView,
132                 new TLcdGridReference(new TLcdGeodeticDatum(),
133                                         new TLcdEquidistantCylindrical()),
134                 true
135             );
136         }
137     });
138     b2d.setToolTipText("Switch the view to 2D");
139     group.add(b2d);
140
141     // Create a button to switch to 3D
142     JRadioButton b3d = new JRadioButton("3D", false);
143     b3d.setAction(new AbstractAction("3D") {
144         @Override
145         public void actionPerformed(ActionEvent e) {
146             TLspViewTransformationUtil.setup3DView(aView, true);
147         }
148     });
149     b3d.setToolTipText("Switch the view to 3D");
150     group.add(b3d);
151
152     // Add the two buttons to the toolbar
153     toolBar.add(b2d);
154     toolBar.add(b3d);
155
156     return toolBar;
157 }
158
159 /**
160 * Entry point for starting the application.
161 */
162 protected void start() {
163     try {
164         ILspAWTView view =createView();
165         initGUI(view);
166         initLayers(view);
167         initController(view);
168     } catch (IOException e) {
169         e.printStackTrace();
}

```

```
170     JOptionPane.showMessageDialog(fFrame, "Starting the sample failed:\n\t" + e.getMessage())
171     ,
172 }
173 }
174
175 public static void main(String[] args) {
176     // Switch to the Event Dispatch Thread, this is required by any Swing based application.
177     EventQueue.invokeLater(new Runnable() {
178         @Override
179         public void run() {
180             new Main().start();
181         }
182     });
183 }
184 }
```

Program 300 - Building a basic Lightspeed view application
(from samples/lightspeed/fundamentals/step1/Main)

F.I.2 Loading and visualizing business data in a Lightspeed view

Program 301 contains the code snippets for loading and visualizing a set of flight plans as described in Chapter 5. Figure 16 shows the flight plans in the window of the basic Lightspeed view application.

```

1 /**
2 * The sample demonstrates how to decode and display data that is stored in a file in a custom
3 * format.
4 */
5
6 public class Main extends samples.lightspeed.fundamentals.step1.Main {
7     @Override
8     protected ILspLayerFactory createLayerFactory() {
9         // Create a layer factory that composes both the flight plan layer factory and the
10        // basic implementation (that supports shp and rasters)
11        return new TLspCompositeLayerFactory(
12            new FlightPlanLayerFactory(), new BasicLayerFactory());
13    }
14
15    protected void initLayers(ILspView aView) throws IOException {
16        ///////////////////////////////
17        // CREATE BACKGROUND AND GRID LAYER
18
19        // Load some background data
20        ILcdModelDecoder earthDecoder = new TLcdEarthRepositoryModelDecoder();
21        ILcdModel earthModel = earthDecoder.decode("Data/Earth/SanFrancisco/tilerepository.cfg");
22        aView.addLayersFor(earthModel);
23
24        // Create and add the grid layer
25        aView.addLayer(TLspLonLatGridLayerBuilder.newBuilder().build());
26
27        ///////////////////////////////
28        // CREATE FLIGHT PLAN LAYER
29
30        // FlightPlanModelDecoder can read the custom file format
31        ILcdModelDecoder decoder = new FlightPlanModelDecoder();
32
33        // Decode custom file to create an ILcdModel for flight plans
34        ILcdModel flightPlanModel = decoder.decode("Data/Custom1/custom.cfg");
35
36        // Calling addLayer() will cause the view to invoke its layer factory with
37        // the given model and then add the resulting layer to itself
38        Collection<ILspLayer> flightPlanLayer = aView.addLayersFor(flightPlanModel);
39
40        // Fit the view to the flight plan layer we just added.
41        fitViewExtents(aView, flightPlanLayer);
42    }
43
44    public static void main(String[] args) {
45        // Switch to the Event Dispatch Thread, this is required by any Swing based application.
46        EventQueue.invokeLater(new Runnable() {
47            @Override
48            public void run() {
49                new Main().start();
50            }
51        });
52    }
53}

```

Program 301 - Loading and visualizing business data in a Lightspeed view
 (from samples/lightspeed/fundamentals/step2/Main)

F.1.3 Grouping Lightspeed layers in a layer tree node

Program 302 contains the code snippets for creating a layer tree node containing both the flight plan layer created in Chapter 5 and a way point layer. Figure 18 shows the window with all the loaded data and the layer control panel with the combined layer.

```

1 /**
2  * The sample demonstrates how to combine data from two different models into an
3  * ILcdLayerTreeNode.
4 */
5 public class Main extends samples.lightspeed.fundamentals.step1.Main {
6     @Override
7     protected ILspLayerFactory createLayerFactory() {
8         // Create a layer factory that composes the flight plan and the way point layer
9         // factories, and the basic layer factory so that all those model types are supported.
10        return new TLspCompositeLayerFactory(
11            new FlightPlanLayerFactory(), new WayPointLayerFactory(), new BasicLayerFactory());
12    }
13
14    @Override
15    protected void initLayers(ILspView aView) throws IOException {
16        ///////////////////////////////
17        // CREATE BACKGROUND AND GRID LAYER
18
19        // Load some background data
20        ILcdModelDecoder earthDecoder = new TLcdEarthRepositoryModelDecoder();
21        ILcdModel earthModel = earthDecoder.decode("Data/Earth/SanFrancisco/tilerepository.cfg");
22        aView.addLayersFor(earthModel);
23
24        // Create and add the grid layer
25        aView.addLayer(TLspLonLatGridLayerBuilder.newBuilder().build());
26
27        ///////////////////////////////
28        // CREATE WAYPOINT LAYER
29
30        // Create the waypoint model
31        ILcdModelDecoder waypointModelDecoder = new WayPointModelDecoder();
32        ILcdModel waypointModel = waypointModelDecoder.decode("Data/Custom1/custom.cwp");
33
34        // Do not yet add the waypoint model to the view,
35        // we will add it as a combined layer later on
36        Collection<ILspLayer> waypointLayer = aView.getLayerFactory().createLayers(waypointModel);
37
38        ///////////////////////////////
39        // CREATE FLIGHT PLAN LAYER
40
41        // FlightPlanModelDecoder can read the custom file format
42        ILcdModelDecoder flightPlanModelDecoder = new FlightPlanModelDecoder();
43
44        // Decode custom file to create an ILcdModel for flight plans
45        ILcdModel flightPlanModel = flightPlanModelDecoder.decode("Data/Custom1/custom.cfp");
46
47        // As with the waypoint layer, we will not yet add the flight plan layer to the view
48        Collection<ILspLayer> flightPlanLayer = aView.getLayerFactory().createLayers(
49            flightPlanModel);
50
51        ///////////////////////////////
52        // CREATE COMBINED LAYER
53
54        // Create a combined layer that holds both the waypoint- and the flight plan layer
55        TLspLayerTreeNode combinedLayer = new TLspLayerTreeNode("Combined Layer");
56        combinedLayer.addLayers(flightPlanLayer);
57        combinedLayer.addLayers(waypointLayer);
58
59        // Add the combined layer to the view
60        aView.addLayer(combinedLayer);
61
62        fitViewExtents(aView, flightPlanLayer);
63    }

```

```
64  public static void main(String[] args) {
65      // Switch to the Event Dispatch Thread, this is required by any Swing based application.
66      EventQueue.invokeLater(new Runnable() {
67          @Override
68          public void run() {
69              new Main().start();
70          }
71      });
72  }
73 }
```

Program 302 - Grouping layers in a tree node
(from samples/lightspeed/fundamentals/step3/Main)

F.1.4 Adding editing capabilities to a Lightspeed view application

Program 303 contains the code snippets for adding the capabilities that allow users to select and edit data in a Lightspeed view application. This is described in [Section 7.1](#).

```

1 /**
2  * The sample demonstrates how to support on-map editing of vector data, including undo/redo.
3 */
4
5 public class Main extends samples.lightspeed.fundamentals.step1.Main {
6
7     // Used by the edit controller to track undoable operations.
8     private TLcdUndoManager fUndoManager = new TLcdUndoManager();
9
10    @Override
11    protected ILspLayerFactory createLayerFactory() {
12        // Create a layer factory that composes the way point layer factory
13        // and the basic layer factory so that all those model types are supported.
14        return new TLspCompositeLayerFactory(new EditableWayPointLayerFactory(), new
15            BasicLayerFactory());
16    }
17
18    @Override
19    protected void initLayers(ILspView aView) throws IOException {
20        super.initLayers(aView);
21
22        // Create the waypoint model and add it to the view
23        ILcdModelDecoder waypointModelDecoder = new WayPointModelDecoder();
24        ILcdModel waypointModel = waypointModelDecoder.decode("Data/Custom1/custom.cwp");
25
26        Collection<ILspLayer> wayPointLayer = aView.addLayersFor(waypointModel);
27
28        // Fit the view on it
29        fitViewExtents(aView, wayPointLayer);
30    }
31
32    @Override
33    protected JToolBar createToolBar(ILspView aView) {
34        JToolBar toolBar = super.createToolBar(aView);
35
36        // Add buttons for undo and redo
37        TLcdUndoAction undo = new TLcdUndoAction(fUndoManager);
38        toolBar.add(new TLcdSWAction(undo));
39        TLcdRedoAction redo = new TLcdRedoAction(fUndoManager);
40        toolBar.add(new TLcdSWAction(redo));
41
42        return toolBar;
43    }
44
45    @Override
46    protected void initController(ILspView aView) {
47        // We use a utility method of the samples to assign the undo manager to the edit
48        // controller.
49        // The controller also allows to select on the map, and to navigate around.
50        // Buttons for undo/redo are added to the toolbar in the createToolBar() method.
51        ILspController c = ControllerFactory.createGeneralController(fUndoManager, aView);
52
53        // Assign the controller to the view
54        aView.setController(c);
55    }
56
57    public static void main(String[] args) {
58        // Switch to the Event Dispatch Thread, this is required by any Swing based application.
59        EventQueue.invokeLater(new Runnable() {
60            @Override
61            public void run() {
62                new Main().start();
63            }
64        });
65    }
66}
```

Program 303 - Adding editing capabilities to a Lightspeed view application
 (from samples/lightspeed/fundamentals/step4/Main)

F.2 GXY view programs

- Program 304 shows how to build a basic LuciadLightspeed application with a 2D view and a background layer.
- Program 305 shows how to load and visualize a set of flight plans in the 2D view created with Program 304.
- Program 306 shows how to load and visualize a set of way points on top of the flight plans as created with Program 305. The way point and flight plan layers are grouped into a layer tree.
- Program 307 shows how to edit the way points as created with Program 306.

F.2.1 Building a basic LuciadLightspeed application with a GXY view

Program 304 contains the code snippets for building a basic application as described in Chapter 4. Figure 15 shows the window with the components of the basic application.

```

1 /**
2  * Fundamentals sample 1: a basic application.
3  * Shows how to create a 2D view and set it up with some background data.
4  */
5
6 public class Main {
7
8     // The application frame
9     private JFrame fFrame;
10
11    private TLcdMapJPanelcreateView() {
12        // Creates the 2D view.
13        TLcdMapJPanel map = new TLcdMapJPanel();
14        TLcdGXYAsynchronousPaintQueueManager manager = new TLcdGXYAsynchronousPaintQueueManager();
15        manager.setGXYView(map);
16        return map;
17    }
18
19    protected void initLayers(TLcdMapJPanel aView) throws IOException {
20        addBackgroundLayer(aView);
21    }
22
23    protected void addBackgroundLayer(TLcdMapJPanel aView) throws IOException {
24        // Creates the model.
25        TLcdGeoTIFFModelDecoder modelDecoder = new TLcdGeoTIFFModelDecoder();
26        ILcdModel model = modelDecoder.decode("Data/GeoTIFF/BlueMarble/bluemarble.tif");
27
28        // Creates the background layer.
29        ILcdGXYLayer layer = new TLcdGXYAsynchronousLayerWrapper(new ImageLayerFactory())
30            .createGXYLayer(model));
31
32        // Adds the background layer to the view and moves the grid layer to the top.
33        aView.addGXYLayer(layer);
34        aView.moveLayerAt(aView.layerCount() - 1, aView.getGridLayer());
35    }
36
37    /**
38     * Creates a controller that pans using the left mouse button, and zooms using the mouse
39     * wheel.
40     */
41    protected void initController(TLcdMapJPanel aView) {

```

```

40     TLcdGXYCompositeController compositeController = new TLcdGXYCompositeController();
41     TLcdGXYPanController controller = new TLcdGXYPanController();
42     controller.setAWTFilter(TLcdAWTEventFilterBuilder.newBuilder().leftMouseButton().build());
43     controller.setDragViewOnPan(true);
44     compositeController.addGXYController(controller);
45     compositeController.addGXYController(new TLcdGXYZoomWheelController());
46     aView.setGXYController(compositeController);
47 }
48
49 /**
50 * Opens a JFrame containing our view, tool bar and layer control.
51 * @param aView The view.
52 */
53 private void initGUI(TLcdMapJPanel aView) {
54     fFrame = new JFrame("LuciadLightspeed GXY Fundamentals");
55     fFrame.getContentPane().setLayout(new BorderLayout());
56     fFrame.getContentPane().add(aView, BorderLayout.CENTER);
57     fFrame.add(new JScrollPane(createLayerControl(aView)), BorderLayout.EAST);
58     fFrame.setSize(800, 600);
59     fFrame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
60     fFrame.setVisible(true);
61 }
62
63 private Component createLayerControl(TLcdMapJPanel aView) {
64     return new TLcdLayerTree(aView);
65 }
66
67 /**
68 * Entry point for starting the application.
69 */
70 protected void start() {
71     try {
72         TLcdMapJPanel view =createView();
73         initGUI(view);
74         initLayers(view);
75         initController(view);
76     } catch (IOException e) {
77         // In a real application, exception handling should of course be more graceful.
78         e.printStackTrace();
79         JOptionPane.showMessageDialog(fFrame, "Starting the sample failed:\n\t" + e.getMessage()
80             ,
81             "Error", JOptionPane.ERROR_MESSAGE);
82     }
83 }
84
85 public static void main(final String[] aArgs) {
86     // Switch to the Event Dispatch Thread, as required by any Swing based application.
87     EventQueue.invokeLater(() -> new Main().start());
88 }

```

Program 304 - Building a basic LuciadLightspeed application
 (from samples/gxy/fundamentals/step1/Main)

F.2.2 Loading and visualizing business data in a GXY view

Program 305 contains the code snippets for loading and visualizing a set of flight plans as described in Chapter 5. Figure 17 shows the flight plans in the window of the basic application.

```

1 /**
2  * Fundamentals Sample 2: adding business data.
3  * Displays a number of flight plans on top of the background layer created in Sample 1.
4  * The flight plans are decoded from a file in a custom format.
5  */
6
7 public class Main extends samples.gxy.fundamentals.step1.Main {
8
9     private static final String FLIGHTPLAN_MODEL = "Data/Custom1/custom.cfp";
10
11    @Override
12    protected void initLayers(TLcdMap JPanel aView) throws IOException {
13        addBackgroundLayer(aView);
14
15        FlightPlanModelDecoder flightplanDecoder = new FlightPlanModelDecoder();
16        ILcdModel flightplanModel = flightplanDecoder.decode("") + FLIGHTPLAN_MODEL;
17
18        ILcdGXYLayer flightplanLayer = new FlightPlanLayerFactory().createGXYLayer(flightplanModel
19            );
20
21        aView.addGXYLayer(flightplanLayer);
22        aView.moveLayerAt(aView.layerCount() - 1, aView.getGridLayer());
23
24        // Fits the map to the bounds of the flightplan layer.
25        TLcdGXYViewFitAction fitAction = new TLcdGXYViewFitAction();
26        fitAction.fitGXYLayer(flightplanLayer, aView);
27    }
28
29    /**
30     * Creates a controller that selects using the left mouse button, pans using the middle
31     * mouse button,
32     * and zooms using the mouse wheel.
33     */
34    @Override
35    protected void initController(TLcdMap JPanel aView) {
36        TLcdGXYSelectController2 selectController = new TLcdGXYSelectController2();
37        selectController.setAWTFilter(TLcdAWTEventFilterBuilder.newBuilder().leftMouseButton().
38            build());
39
40        TLcdGXYPanController panController = new TLcdGXYPanController();
41        panController.setAWTFilter(TLcdAWTEventFilterBuilder.newBuilder().middleMouseButton().
42            build());
43        panController.setCursor(selectController.getCursor());
44        panController.setDragViewOnPan(true);
45
46        TLcdGXYZoomWheelController zoomWheelController = new TLcdGXYZoomWheelController();
47
48        TLcdGXYCompositeController compositeController = new TLcdGXYCompositeController();
49        compositeController.addGXYController(selectController);
50        compositeController.addGXYController(panController);
51        compositeController.addGXYController(zoomWheelController);
52        aView.setGXYController(compositeController);
53    }
54
55    public static void main(final String[] aArgs) {
56        EventQueue.invokeLater(() -> new Main().start());
57    }
58}
```

**Program 305 - Loading and visualizing business data
(from samples/gxy/fundamentals/step2/Main)**

F.2.3 Grouping layers hierarchically in a GXY view

Program 306 contains the code snippets for creating a new layer with way points and combining this layer with the flight plan layer created in Chapter 5.

```

1 /**
2  * Fundamentals sample: combining business data in a layer tree node.
3  * Displays a number of way points on top of the flight plans created in Sample 2.
4  * The way points are decoded from a file in a custom format. The flight plan layer and
5  * the way point layer are combined in an ILcdLayerTreeNode.
6 */
7
8 public class Main extends samples.gxy.fundamentals.step2.Main {
9
10    private static final String FLIGHTPLAN_MODEL = "Data/Custom1/custom.cfp";
11    private static final String WAYPOINT_MODEL = "Data/Custom1/custom.cwp";
12
13    @Override
14    protected void initLayers(TLcdMapJPanel aView) throws IOException {
15        addBackgroundLayer(aView);
16
17        FlightPlanModelDecoder flightplanDecoder = new FlightPlanModelDecoder();
18        ILcdModel flightplanModel = flightplanDecoder.decode("") + FLIGHTPLAN_MODEL;
19
20        WayPointModelDecoder waypointDecoder = new WayPointModelDecoder();
21        ILcdModel waypointModel = waypointDecoder.decode("") + WAYPOINT_MODEL;
22
23        ILcdGXYLayer flightplanLayer = new FlightPlanLayerFactory().createGXYLayer(flightplanModel
24            );
24        ILcdGXYLayer waypointLayer = new WayPointLayerFactory().createGXYLayer(waypointModel);
25
26        // Creates a layer for the combined model.
27        TLcdGXYLayerTreeNode combinedLayer = new TLcdGXYLayerTreeNode("Combined layer");
28        combinedLayer.addLayer(flightplanLayer);
29        combinedLayer.addLayer(waypointLayer);
30
31        // Adds the flightplan/waypoint layers to the view.
32        // Moves the grid layer on top.
33        aView.addGXYLayer(combinedLayer);
34        aView.moveLayerAt(aView.layerCount() - 1, aView.getGridLayer());
35
36        // Fits the map to the bounds of the flightplan layer.
37        TLcdGXYViewFitAction fitAction = new TLcdGXYViewFitAction();
38        fitAction.fitGXYLayer(flightplanLayer, aView);
39    }
40
41    public static void main(final String[] aArgs) {
42        EventQueue.invokeLater(() -> new Main().start());
43    }
44}

```

Program 306 - Grouping layers
 (from samples/gxy/fundamentals/step3/Main)

F.2.4 Adding editing capabilities to a GXY view application

Program 307 contains the code snippets for adding the capabilities that allow users to select and edit data in a GXY view application.

```

1 /**
2  * Fundamentals sample: editing business data.
3  * Displays a number of way points that can be dragged around.
4  */
5 public class Main extends samples.gxy.fundamentals.step1.Main {
6
7     private static final String WAYPOINT_MODEL = "Data/Custom1/custom.cwp";
8
9     @Override
10    protected void initLayers(TLcdMap JPanel aView) throws IOException {
11        addBackgroundLayer(aView);
12
13        WayPointModelDecoder wayPointDecoder = new WayPointModelDecoder();
14        ILcdModel wayPointModel = wayPointDecoder.decode("") + WAYPOINT_MODEL);
15
16        ILcdGXYLayer waypointLayer = new EditableWayPointLayerFactory().createGXYLayer(
17            wayPointModel);
18
19        // Adds the waypoint layers to the view.
20        // Moves the grid layer on top.
21        aView.addGXYLayer(waypointLayer);
22        aView.moveLayerAt(aView.layerCount() - 1, aView.getGridLayer());
23
24        // Fits the map to the bounds of the waypoint layer.
25        TLcdGXYViewFitAction fitAction = new TLcdGXYViewFitAction();
26        fitAction.fitGXYLayer(waypointLayer, aView);
27    }
28
29 /**
30  * Creates a controller that selects and edits using the left mouse button, pans using the
31  * middle mouse button,
32  * and zooms using the mouse wheel.
33  */
34 @Override
35 protected void initController(TLcdMap JPanel aView) {
36     TLcdGXYEditController editController = new TLcdGXYEditController2();
37     editController.setAWTFilter(TLcdAWTEventFilterBuilder.newBuilder().leftMouseButton().build
38         ());
39
40     TLcdGXYPanController panController = new TLcdGXYPanController();
41     panController.setAWTFilter(TLcdAWTEventFilterBuilder.newBuilder().middleMouseButton()
42         .build());
43     panController.setCursor(editController.getCursor());
44     panController.setDragViewOnPan(true);
45
46     TLcdGXYZoomWheelController zoomWheelController = new TLcdGXYZoomWheelController();
47
48     TLcdGXYCompositeController compositeController = new TLcdGXYCompositeController();
49     compositeController.addGXYController(editController);
50     compositeController.addGXYController(panController);
51     compositeController.addGXYController(zoomWheelController);
52     aView.setGXYController(compositeController);
53
54     public static void main(final String[] aArgs) {
55         EventQueue.invokeLater(() -> new Main().start());
56     }
57 }
```

Program 307 - Adding editing capabilities to a GXY view application
 (from samples/gxy/fundamentals/step4/Main)

APPENDIX G

Acronyms and abbreviations

Acronym	Full term
ADRG	Arc Digitized Raster Graphics
AIS	Aeronautical Information Services
AIXM	Aeronautical Information Exchange Model
API	Application Programming Interface
ARINC	Aeronautical Radio Incorporated
ASDI	Aircraft Situation Display to Industry
ASRP	Arc Standard Raster Product
ASTERIX	All purpose STructured Eurocontrol suRveillance Information eXchange
ATC	Air Traffic Control
ATM	Air Traffic Management
BIL	Band Interleaved by Line
BMP	Bitmap
BPW	BMP World
CADRG	Compressed Arc Digitized Raster Graphic
CGM	Computer Graphics Metafile
CGRS	Common Grid Reference System
CIB	Controlled Image Base
CRGS	Common Grid Reference System
CRS	Coordinate Reference System
DAFIF	Digital Aeronautical Flight Information File
DEM	(USGS) Digital Elevation Model
DGN	Design (file format)
DHM	Digital Height Model
DIMAP	Digital Image Map
DMA	Defense Mapping Agency
DMED	Digital Mean Elevation Data
DTED	Digital Terrain Elevation Data
DWG	(AutoCAD) Drawing
DXF	(AutoCAD) Drawing
ECDIS	Electronic Chart Display and Information System
ECRG	Enhanced Compressed Raster Graphic
ECW	Enhanced Compression Wavelet
EPSG	European Petroleum Survey Group
ESRI	Environmental Systems Research Institute

Acronym	Full term
GARS	Global Area Reference System
GEE	Google Earth Enterprise
GDAL	Geospatial Data Abstraction Library
GDF	Geographic Data File
GEOREF	World Geographic Reference System
GeoTIFF	Geographic TIFF
GIF	Graphic Interchange Format
GML	Geography Markup Language
GRIB	Gridded Binary
GUI	Graphical User Interface
IGRF	International Geomagnetic Reference Field
JAI	Java Advanced Imaging
JGW	JPEG World
JPEG	Joint Photographic Experts Group
JPL	Jet Propulsion Laboratory
MAP	MapInfo Format
MIF	MapInfo Interchange Format
MrSID	(LizardTech) Multiresolution Seamless Image Database
MTM	Multileveled Tiled Model
MVC	Model View Controller
NetCDF	Network Common Data Form
NIMA	National Imagery and Mapping Agency
NITF	National Imagery Transmission Format
NVG	NATO Vector Graphic
NOAA	National Oceanic and Atmospheric Administration
OBJ	(Alias WaveFront) OBject
OGC	Open Geospatial Consortium
PBM	Portable BitMap
PGM	Portable GrayMap
PNM	Portable aNyMap
PPM	Portable PixMap
POL	(TAAM) Polygon
SLD	Styled Layer Descriptor
SVG	Scalable Vector Graphics
SwissDHM	SwissTopo Digital Height Model
TAB	(MapInfo) Tab
TEA	Terrain Elevation Analysis
TFW	Tiff World
TIFF	Tagged Image File Format
UPS	Universal Polar Stereographic
USGS	US Geological Survey
USRP	UPS/UTM Standard Raster Product
UTM	Universal Transverse Mercator
VPF	Vector Product Format
WCS	Web Coverage Service
WFS	Web Feature Service
WKT	Well-Known Text
WMM	World Magnetic Model
WMS	Web Map Service

Acronym	Full term
WMTS	Web Map Tile Service
XML	eXtensible Markup Language

Table 15 - Acronyms and abbreviations

Glossary

azimuth An azimuth is defined as a horizontal angle measured clockwise from a north base line. This north base line could be true north, magnetic north, or grid north. The azimuth is the most common military method to express direction. When using an azimuth, the point from which the azimuth originates is the center of an imaginary circle . This circle is divided into 360 degrees or 6400 mils. NORTH IS 0/360 AZIMUTH. [326](#)

Bézier curves Curved lines (splines) defined by mathematical formulas. Bézier curves employ at least three points to define a curve. The two endpoints of the curve are called anchor points. The other points, which define the shape of the curve, are called handles, tangent points, or nodes. Attached to each handle are two control points. By moving the handles themselves, or the control points, you can modify the shape of the curve. In vector graphics, Bézier curves are used to model smooth curves that can be scaled indefinitely. [465](#)

background data Reference data for the data that users want to interact with (business data). [11](#)

bounding box An invisible box surrounding a graphical object and determining its size. The minimum bounding box for a point set in N dimensions is the box with the smallest measure (area, volume, or hypervolume in higher dimensions) within which all the points lie. [91](#)

bounds Represents an axis-aligned bounding box. [43](#)

business data The data that users want to interact with. [11](#)

Cartesian Cartesian coordinates provide a method of indicating the positions of points on a two-dimensional (2D) surface or in three-dimensional (3D) space. The Cartesian plane consists of two perpendicular axes that cross at a central point called the origin. Positions or coordinates are determined according to the east/west and north/south displacements from the origin. The east/west axis is often called the x axis, and the north/south axis is called the y axis. For this reason, the Cartesian plane is also known as the xy -plane. Cartesian three-space, also called xyz -space, has a third axis, oriented at right angles to the xy -plane. This axis, usually called the z axis, passes through the origin of the xy -plane. [18](#)

classical 2D view 2D view, as available in the latest version of LuciadMap. It is also known as GXY view. This view uses Java 2D technology for visualization. [11](#)

controller application component that handles user interaction. [9](#)

domain object A separate data element that is part of a business domain and that is contained in a model. [10](#)

ellipsoid In geodesy, a reference ellipsoid is a mathematically-defined surface that approximates the geoid, the truer figure of the earth, or other planetary body. Because of their relative simplicity, reference ellipsoids are used as a preferred surface on which geodetic network computations are performed and point coordinates such as latitude, longitude, and elevation are defined. [18](#)

geodesy also named geodetics, a branch of earth sciences, is the scientific discipline that deals with the measurement and representation of the earth, including its gravitational field, in a three-dimensional time-varying space. Geodesists also study geodynamical phenomena such as crustal motion, tides, and polar motion. For this they design global and national control networks, using space and terrestrial techniques while relying on datums and coordinate systems. [1](#)

geodetic datum A geodetic datum is a reference from which measurements are made. In surveying and geodesy, a datum is a set of reference points on the earth's surface against which position measurements are made, and (often) an associated model of the shape of the earth (reference ellipsoid) to define a geographic coordinate system. Horizontal datums are used for describing a point on the earth's surface, in latitude and longitude or another coordinate system. Vertical datums measure elevations or depths. [5](#)

geoid Essentially the figure of the earth abstracted from its topographical features. It is an idealized equilibrium surface of sea water, the mean sea level surface in the absence of currents, air pressure variations and so forth, and continued under the continental masses. [415](#)

interpolate Insert an intermediate item into a series of items by estimating or calculating it from surrounding known values. [167](#)

layer Container used to visually represent model data in a view. [12](#)

lazy loading A design pattern commonly used in computer programming to defer initialization of an object until the point at which it is needed. It can contribute to efficiency in the program's operation if properly and appropriately used. [2](#)

Lightspeed view the view implementation that is introduced with LuciadLightspeed, and uses OpenGL for hardware-accelerated visualization. A Lightspeed view can visualize data in 2D as well as in 3D. [6](#)

MGRS The Military Grid Reference System (MGRS) is a geocoordinate standard used by NATO. The MGRS is derived from the UTM and UPS grid systems, but uses a different labeling convention. The MGRS is used for the entire earth. [162](#)

model A container for domain objects. [9](#)

model descriptor A container for the metadata that describes the model data. [10](#)

model reference The coordinate reference system that is used to locate model data on earth. [10](#)

multi-resolution a multi-resolution object can be described at different levels of resolution. [107](#)

oblate spheroid An oblate spheroid is a rotationally symmetric ellipsoid having a polar axis shorter than the diameter of the equatorial circle whose plane bisects it. The shape of the Earth is very close to that of an oblate spheroid. [28](#)

OpenCL Open Computing Language, the cross-platform standard interface for general parallel computation on GPUs. [15](#)

OpenGL Open Graphics Language, the standard interface for visualization using the GPU. This standard is available on many platforms and drivers are available from all major graphics card vendors. [2](#)

orthographic as in orthographic projection. When the human eye looks at a scene, objects in the distance appear smaller than objects close by. Orthographic projection ignores this effect to allow the creation of to-scale drawings. Orthographic projections are a small set of transforms often used to show profile, detail or precise measurements of a three dimensional object. Common names for orthographic projections include plane, cross-section, bird's-eye, and elevation. While orthographically projected images represent the three dimensional nature of the object projected, they do not represent the object as it would be recorded photographically or perceived by a viewer observing it directly. [28](#)

orthorectification Processing an image to geometrically correct it so that the scale of the photograph is uniform and it can be measured as a map. [437](#)

pan Move the map by clicking or touching it and dragging it. [38](#)

perspective as in perspective projection. When the human eye looks at a scene, objects in the distance appear smaller than objects close by - this is known as perspective. While orthographic projection ignores this effect to allow accurate measurements, perspective definition shows distant objects as smaller to provide additional realism. [28](#)

primitive A geometric primitive is of the simplest (meaning 'atomic' or irreducible) geometric objects that the system can handle. The most primitive primitives are point and straight line segments. [124](#)

Raster data Data that consists of a grid of evenly sized cells. [10](#)

Shape A geometrical object with a bounding box and a focus (or center) point. [36](#)

Snapping Snapping layer objects into position pulls the objects to one another or to ruler subdivisions, grid lines, guides, or guide points so that you can control the placement and alignment of the objects. [310](#)

Vector data Data that consists of geometries. [10](#)

view Visual representation of model data. [9](#)

viewing frustum The region of space in the modeled world that may appear on the screen. It is the field of view of the notional camera. [165](#)

WGS The World Geodetic System is a standard for use in cartography, geodesy, and navigation. It comprises a standard coordinate frame for the earth, a standard spheroidal reference surface (the datum or reference ellipsoid) for raw altitude data, and a gravitational equipotential surface (the geoid) that defines the nominal sea level. The latest revision is WGS 84 (dating from 1984 and last revised in 2004), which will be valid up to about 2010. WGS 84 is the reference coordinate system used by the Global Positioning System. [18](#)

world reference A coordinate reference system and a map projection used to represent earth data in a 2D view. [11](#)

Index

- .jai, 138
- .rst, 139
- LuciadLightspeed
 - benefits, 1
 - controller, 13
 - model, 10
 - view, 11
- LuciadLightspeed applications, 15
- LuciadLightspeed shapes
 - memory saving implementations, 92
- 2D object draping, 278
- 2D screen coordinates, 429
- 3D box, 97
- abbreviations, 579
- absolute label location, 364
- acceptSnapTarget, 333
- accessing
 - asynchronously painted layers, 316
 - domain objects, 79
- accessing adapted layers, 410
- acronyms, 579
- activating interactive labels, 366
- activating interactive lightspeed labels, 228
- activating labels, 216
- adapting
 - part of a shape, 329
 - the view bounds, 344
- adding
 - a halo around an icon, 309
 - a halo to labels, 362
 - a raster to a model, 441
 - custom metadata, 89
 - grid layer to the view, 161
 - layer to view, 37, 50
 - undo/redo support, 276, 379
 - view to window, 38
- adding data formats, 449
- adding data to a Lightspeed view, 23
- adding handles, 256
- adding layers to a Lightspeed view, 23
- aeronautical navigational charts, 424
- aeronautical operational display systems, 424
- Air Track Display, 424
- ALcdBalloonManager, 160
- ALcdBandSemantics, 105
- ALcdBasicImage, 105
- ALcdGXYChainableController, 377
- ALcdGXYDiscretePlacementsLabelingAlgorithm, 370
- ALcdGXYInteractiveLabelProvider, 365
- ALcdGXYLabelStamp, 361
- ALcdGXYNewControllerModel2, 376
- ALcdGXYTouchChainableController, 377
- ALcdImage, 103, 104, 507
- ALcdImageMosaic, 106
- ALcdImagingEngine, 116
- ALcdLabelLocations, 363
- ALcdModelHeightProviderFactory, 386
- ALcdMultilevelImage, 106
- ALcdMultilevelImageMosaic, 106
- ALcdVWXAxisRendererJ2D, 394
- ALspCreateHandle, 252
- ALspDynamicCreateHandle, 255
- ALspInteractiveLabelProvider, 228
- ALspLabelStyleCollector, 216
- ALspLabelStyler, 216
- ALspMultiObjectHandle, 254
- ALspOutlineResizeHandle, 252
- ALspSingleLayerFactory, 25
- ALspStyle, 29, 180
- ALspStyleCollector, 195, 200
- ALspStyler, 196
- ALspStyleTargetProvider, 195, 199, 253
- ALspViewXYZWorldTransformation, 433
- anchoring an icon, 309
- anchoring points for labels, 222
- animated styles, 199
- animations, 167, 199
- annotating data models, 494
- API reference, 3
- appendGeneralPath, 337
- applyOnInteract, 373
- applyOnPaintedLabelLocations, 363
- applySelection, 373

defining, 375
aPrefix, 530
arc, 94
 circular, 94
 elliptical, 94
arc band, 94
architecture, 9
array of ILcdGXYPainterProvider objects, 301
asynchronous painting, 315
asynchronously painted layers
 accessing, 316
ATI/AMD, 543
Aviation Standards component, 309
avoiding warping, 517
AWT path, 336
azimuthal projection, 423

background data, 11
background image buffers
 painting, 315
background layer
 creating, 36
background layers
 caching, 517
balloon, 159
balloon content panel, 159
balloons
 adding to the view, 159, 289
 customizing, 161
 removing, 161
band semantics, 104
BIL images, 140
binary compatibility, 539
BLW file format, 140
BMP images, 140
BODY, 328
body labels, 364
body of object
 painting, 328
border for rasters, 304
boundaryLats, 422
boundaryLons, 422
bounding box, 91
 height, 91
 transforming, 433
 transforming from view to model coordinates, 291
BPW file format, 140
brightness of rasters, 303
British Admiralty charts, 424
browsing type information, 82
buffer, 95

Builder design pattern, 24
building
 LuciadLightspeed applications, 15
building a basic Lightspeed view application
 cookbook, 565
building a basic LuciadLightspeed application
 cookbook, 573
business data, 11
 loading and visualizing, 39
 modeling, 40
business data in a GXY view, 48
business data in a Lightspeed view, 45
 cookbook, 569

C, C++, C# integration, 535
caching
 background layers, 517
 image of haloed object, 311
Caching an ILcdGeneralPath, 338
calculateAWTBoundsSFCT, 339
calculating view bounds based on an ILcdAWT-
 Path, 340
camera constraints, 158
cancelInteraction, 366
canProvideInteractiveLabel, 366
canStopInteraction, 366
Cartesian calculations, 436
Cartesian coordinate system, 421, 428
Cartesian reference system, 18
Cartesian shapes, 92
center point, 91
centering a view, 158
CGRS grid, 288
chaining controllers, 62, 266
chaining image operators, 109
changes
 handling, 65
changes to shapes
 notification, 101
changing
 a composite object, 101
 application objects, 65
 domain objects, 70
 indirect layer properties, 70
 map projection, 35
changing object look, 176
changing painting order, 173
checking
 if an object is touched, 331, 347
 if an object should be painted, 330
child layers, 54
child models, 152

choosing super type, 86
 circle, 94
 circle painting, 201
 circular arc, 94
 classes
 configuring with properties files, 529
 classification, 148
 clearing image cache, 311
 cloneAs2DEditableBounds, 97
 cloneAs2DEditablePoint, 97
 cloneAs3DEditableBounds, 97
 cloneAs3DEditablePoint, 97
 cluster shape provider, 148
 clustering, 147
 clusterSize, 148
 minimumPoints, 148
 clustering classification, 148
 clusterSize, 148
 Codelists
 ISO 19115, 457
 color model of rasters, 303
 com.luciad.datamodel, 79
 com.luciad.format.raster, 298
 com.luciad.format.xml, 491
 com.luciad.format.xml.bind.schema.datamodel, 467
 com.luciad.format.xml.bind.schema.dataobject, 491
 com.luciad.ogc.sld.xml, 489
 com.luciad.reference, 429
 com.luciad.transformation, 433
 com.luciad.util.height, 385
 com.luciad.view.gxy, 433
 com.luciad.view.gxy.painter, 298
 com.luciad.view.lightspeed.camera, 433
 com.luciad.view.map.multilevelgrid, 288
 combining
 ILcdGXYPainter and ILcdGXYEditor, 342
 parametric and non-parametric rectification, 444
 rasters, 508
 commonly used shapes, 94
 comparing layers, 272
 complex geometry, 91
 complex stroke imitations, 314
 complex strokes
 drawing, 312
 complex XML schema types, 496
 composing complex geometries, 91
 composite curve, 100
 composite model, 151
 composite object
 changes to, 101
 composite shape, 98
 computePointSFCT, 98
 configuring
 an ILcdGXYLabelPainter2 for a TLcdGXY-Layer, 360
 classes with properties files, 529
 label placement, 360
 conical projection, 423
 constraining view navigation, 158
 constructing an ILcdAWTPPath with an ILcdGXYPen, 335
 contains2D, 97
 contains3D, 97
 controller chaining, 62
 controller for selection and editing, 62, 63
 controllers, 13
 chaining, 266
 customizing, 266
 defining, 17, 38
 controllers in a Lightspeed view, 26
 conventions, 5
 convolution, 103
 cookbook
 building a basic Lightspeed view application, 565
 building a basic LuciadLightspeed application, 573
 business data in a Lightspeed view, 569
 editing capabilities in a basic GXY view application, 577
 editing capabilities in a basic Lightspeed view application, 572
 grouping layers hierarchically, 576
 grouping Lightspeed layers in a tree node, 570
 loading and visualizing business data, 575
 coordinate reference system, 18
 coordinate systems, 427
 transformations between, 433
 coordinates
 screen, 429
 view, 429
 copying layer nodes between layer trees, 295
 create controller, 255
 create handles, 255
 createImage, 120
 creating
 a background layer, 36
 a layer control, 37

- a model list, 151
- a model node, 152
- a shape with TLcdGXYNewController2, 340
- a vertical view, 391
- a view, 17, 35
- an asynchronous painting wrapper, 315
- an icon from an image file, 309
- an ILcdPolygon, 100
- an ILcdPolyline, 100
- data model, 85
- layer nodes, 54
- model decoders, 43
- objects using the mouse, 376
- creating a Lightspeed view, 22
- creating a styler, 196
- creating image operators, 108
- creating layers in a Lightspeed view, 24
- creating raster layers, 177
- creating styles, 182
- creating vector layers, 177
- creation controller, 255
- creation density layers, 179
- CRS
 - see *coordinate reference system*, 18
- CssParameter, 489
- custom controllers, 274
- custom domain classes for XML schema types, 494
- custom editor, 257
- custom label algorithms, 231
- custom layer interface
 - providing transparent access, 320
- custom logging properties file, 524
- custom marshalling and unmarshalling, 495
- custom metadata
 - adding, 89
- custom object editing, 256
- custom touch controller, 274, 377
- custom XML decoding, 495
- CustomDecoderLibrary, 495
- customized reference system, 432
- customizing
 - a label editor, 370
 - a label painter, 369
 - a TLcdGXYEditController2, 375
 - a TLcdGXYNewController2, 376
 - a TLcdGXYSelectController2, 373
 - an ILcdGXYEditor, 307
 - an ILcdGXYPainter, 299
 - an ILcdGXYPainterProvider, 302
- an ILcdModelEncoder, 144
- balloons, 161
- label dependencies, 363
- label locations, 363
- label placement algorithm, 370
- labels using label stamps, 361
- paint queue assignments, 319
- customizing label text, 218
- customizing labels, 216
- customizing selection controller, 270
- customizing stylers, 197
- Customizing terrain, 204
- customizing vector layers, 177
- cylindrical projection, 422
- data model, 80
 - annotating, 494
 - creating, 85
 - defining, 82
 - designing, 86
 - GML, 465
 - GML32, 469
 - single or multiple, 86
- data model based on an XML schema, 492
- data model dependencies, 87
- data object type, 84
- Debugging, 547
- decoding
 - GeoTIFF file, 34
 - ISO 19115 XML documents, 492
 - metadata in ISO 19139 format, 460
 - SLD styling models, 489
 - TAB files, 139
 - XML complex types, 496
 - XML documents, 492
 - XML documents with unknown schema, 502
- default controller in a Lightspeed view, 26
- Defense Standards component, 309
- Defense Symbology component, 309
- defining
 - a view, 17
 - a world reference, 17
 - controllers, 17, 38
 - data model, 82
 - GUI actions, 265, 371
 - models, 16
- defining density styles, 195
- defining fill styles, 183
- defining icon styles, 185
- defining line styles, 182
- defining raster styles, 193

- defining symbols, 185
- defining vector styles, 182
- defining vertical line styles, 184
- density layers
 - creation, 179
- density plots, 179
- density styles, 195
- designing
 - data model, 86
- determining
 - model descriptor, 42
 - model reference, 42
 - the location of the labels, 369
- determining a point on terrain, 278
- diagram
 - labeling sequence, 368
 - paint sequence, 297
- dimensional data
 - modeling, 133
- dispatchAWTEvent, 367
- dispatchEvent, 377
- displaying
 - performance in JConsole, 526
 - the mouse position, 162, 291
- distortion in images, 437
- DMA ellipsoid, 419
- DMED/DTED fall-back tuning, 507
- domain classes
 - ISO 19115, 458
- domain model, 79
 - GML, 466
 - ISO 19115, 457
- domain objects, 10
 - accessing, 79
 - changing, 70
 - editing, 306
 - geometry, 40
 - labeling, 359
 - modeling, 40
 - painting, 297
 - properties, 40
 - representing, 79
- draping, 204
- draping 2D objects, 278
- Draping in a Lightspeed view, 203
- drawing
 - complex strokes, 312
 - halos around objects, 310
- Drivers
 - Troubleshooting, 543
- dynamic data, 43
- dynamic styles, 198
- Earth API, 24
- edit, 306
- edit context, 306
- edit controller, 249
- edit handles
 - creation, 251
- edit mode, 306
- editable bounding boxes, 97
- editable objects, 61, 63
- editable points, 97
- editable shapes, 92
- editing
 - custom objects, 256
 - customization, 255
 - domain objects, 306
 - labels, 365
 - layers, 249
 - process, 249
- editing a shape with TLcdGXYEditController2, 346
- editing capabilities in a basic GXY view application
 - cookbook, 577
- editing capabilities in a basic Lightspeed view application
 - cookbook, 572
- editing controller
 - responsibilities, 251
- editing customization, 270
- editing domain objects, 247
- editing the model of a wrapped layer, 318, 411
- editing with multiple input points, 307
- editLabel, 370
- editor, 12, 247
 - custom implementation, 257
 - customization, 257
 - implementations, 248
 - registration, 249
 - responsibilities, 251
- elementChanged, 70
- elevation
 - objects, 204
 - terrain, 203
- Elevation in a Lightspeed view, 203
- elevation mode, 181, 204
- ellipse, 95
- ellipsoid, 18
 - DMA, 419
 - JPL, 419
- ellipsoidal calculations, 435

elliptical arc, 94
encoding
 model data, 143
 SLD styling models, 489
EntityResolver2, 492
enumerating elements of a model list, 151
EPSG file format, 140
EPSG geotic datum, 417
EPSG reference information, 141
evaluateModifiedPaintBlocks, 320
excluding objects from visualization, 171, 296
explaining
 meta model, 83

fall-back stroke, 313
fast reordering of models, 151
feature
 GML, 465
FeatureTypeStyle, 489
file formats supported in LuciadLightspeed,
 551
fill styles, 183
filter conditions, 477
filtering objects
 from a layer, 171, 296
firing event mode, 70
fitting a view, 158
fitting the view to a layer, 47, 50
FixedCountPaintQueueManager, 320
flipped system, 438
Float, 92
focus point, 91
forcePainting, 303, 518
format support, 449
forwardAzimuth2D, 419
freezing user interface, 315

Garbage collection, 513
GARS grid, 288
generic geotic datum, 417
geodesicDistance, 419
geodesicPointSFCT, 419
geodesy, 415
 literature, 419
geodetic datum, 18, 415
 EPSG, 417
 generic, 417
 JPL, 417
 NIMA, 417
geodetic reference system, 18
geodetic shapes, 92
Geography Markup Language see GML, 463

geoid, 415
geometric calculations, 435
geometries, 10
 modeling, 91
geometries and TLcdDataModel, 90
geometry
 GML, 468
geometry information
 representing with properties, 90
georaster formats, 104
GEOREF grid, 162, 287
Geosym symbology, 309
GeoTIFF files
 decoding, 34
GeoTIFF images
 tiled, 507
getBounds, 91
getCosX, 97
getCosY, 97
getDepth, 97
getElevationMode(), 181
getFocusPoint, 91
getHeight, 97
getLabelLocation, 363
getLabelPlacer, 231
getLocation, 97
getLocationIndex, 364
getPixelDensity, 119
getShape, 98
getShapeCount, 98
getSinX, 97
getSinY, 97
getTanX, 97
getTanY, 97
getWidth, 97
getX, 96
getY, 96
getZ, 96
getZOrder(), 181
GIF file format, 138
GML data
 integrating into LuciadLightspeed applica-
 tion, 464
GML data model, 465
GML domain model, 466
GML feature collection integration, 467
GML feature integration, 467
GML features, 465
GML file
 loading and displaying, 464
GML format, 463

- GML geometry integration, 468
- GML model decoders, 468
- GML model descriptor, 467
- GML model encoders, 469
- GML Simple Features Profile, 466
- GML versions supported in LuciadLightspeed, 463
- GML32 data model, 469
- graph, 109
- graphical effects, 279
- graphical point symbol, 309
- graphs, 103
- grayscale values, 303
- grid
 - CGRS, 288
 - GARS, 288
 - GEOREF, 162, 287
 - maritime-style, 287
 - MGRS, 287
 - UPS, 287
 - UTM, 287, 424
 - XY, 162, 287
- grid layer
 - adding to the view, 161
- grid layers
 - predefined, 287
- grid reference system, 18, 421
- grid systems
 - national, 424
- grouping layers hierarchically
 - cookbook, 576
- grouping Lightspeed layers in a tree node
 - cookbook, 570
- grouping models hierarchically, 151
- GUI actions
 - defining, 265, 371
- halo, 310
 - adding to an object, 310
 - adding to labels, 362
- halos
 - cached image, 311
 - painters, 311
- HANDLES, 328
- handles, 247
 - activation, 253
 - add and remove, 256
 - API, 252
 - controller, 254
 - creation, 255
 - focus, 254
 - priority, 254
- properties, 253
- responsibilities, 252
- visualization, 253, 256
- handles of object
 - painting, 328
- handling
 - changes, 65
 - non-standard input, 276, 377
- Hardware report, 546
- hatch pattern rasters, 178
- HDR images, 103
- height data
 - retrieving, 385
- height of a point, 91
- height of bounding box, 91
- height provider, 277
- hierarchically grouping models, 151
- High Dynamic Range, 103
- hippodrome
 - create handles, 262
 - edit operations, 259
 - handles, 257
- hippodrome editor, 257
- horizontal datum, 415
- ICAO symbology, 309
- icon
 - adding a halo, 309
 - anchor, 309
 - creating from an image file, 309
- Icon styling, 201
- icon-based label content, 218
- icons, 185
- IGRF models
 - model decoder, 563
- ILcd2DBoundsIndexedModel for model nodes, 152
- ILcd2DEditableBounds, 97
- ILcd2DEditablePoint, 97
- ILcd2DEditableShape, 92
- ILcd3DEditableBounds, 97
- ILcd3DEditablePoint, 97
- ILcd3DEditableShape, 92
- ILcdAction, 265, 371
 - implementations, 265, 371
 - to load model data, 451
- ILcdAnnotation, 89
- ILcdArc, 94
- ILcdArcBand, 94
- ILcdAWTEventListener, 377
- ILcdAWTPath, 336
 - constructing with an ILcdGXYPen, 335

- rendering, 339
- ILcdAzimuthal**, 423
- ILcdBalloonContentProvider**, 160
- ILcdBounds**, 91, 94, 97
- ILcdBounds.getHeight**, 91
- ILcdCircle**, 94
- ILcdCircleBy3Points**, 100
- ILcdCircularArc**, 94
- ILcdCircularArcBy3Points**, 100
- ILcdCircularArcByBulge**, 100
- ILcdCircularArcByCenterPoint**, 100
- ILcdClassifier**, 148
- ILcdClusterShapeProvider**, 148
- ILcdComplexPolygon**, 95, 96
- ILcdCompositeCurve**, 100
- ILcdConic**, 423
- ILcdCurve**, 96, 98
- ILcdCylindrical**, 422
- ILcdDataModelDescriptor**, 80
- ILcdDataObject**, 79, 80
 - implementing, 88
- ILcdDeepCloneable**, 317
- ILcdEllipse**, 95
- ILcdEllipsoid**, 416, 419, 435
- ILcdEllipsoidFactory**, 419
- ILcdFeatured**, 85
- ILcdFeaturedDescriptor**, 85
- ILcdFireEventMode**, 70
- ILcdGeneralPath**, 337
- ILcdGeoBuffer**, 95
- ILcdGeocentricReference**, 430
- ILcdGeodeticDatum**, 416
- ILcdGeodeticDatumFactory**, 417
- ILcdGeodeticReference**, 18, 429
- ILcdGeoReference**, 428, 429
- ILcdGeoReference extensions**, 429
- ILcdGLDrawable**, 547
- ILcdGridReference**, 18, 431
- ILcdGXYAsynchronousLayerWrapper**, 316, 320
- ILcdGXYEditor**
 - combining with **ILcdGXYPainter**, 342
 - customizing, 307
 - snapping methods, 333
 - using, 306
- ILcdGXYEditor for shapes**, 306
- ILcdGXYEditor implementations**, 306
- ILcdGXYEditor methods**, 332
- ILcdGXYEditor modes**, 332
- ILcdGXYEditor with **ILcdGXYPainter****, 306
- ILcdGXYEditorProvider**
 - using, 307
- ILcdGXYLabelEditor implementations**, 365
- ILcdGXYLabelingAlgorithm**, 363, 370
- ILcdGXYLabelPainter2**, 360
- ILcdGXYLabelPainterProvider**, 359
- ILcdGXYLayerChangeTracker**, 320
- ILcdGXYLayerSubsetList**, 333
- ILcdGXYPainter**
 - combining with **ILcdGXYEditor**, 342
 - customizing, 299
 - snapping methods, 333
 - using, 298
- ILcdGXYPainter implementations**, 298
- ILcdGXYPainter with **ILcdGXYEditor****, 306
- ILcdGXYPainterProvider**
 - customizing, 302
 - using, 300
- ILcdGXYPainterProvider implementations**, 300
- ILcdGXYPainterStyle**, 311
- ILcdGXPen**
 - setting to a layer, 308
 - using, 307
- ILcdGXPen implementations**, 308
- ILcdGXYView**, 11
- ILcdGXYViewLabelPlacer**, 360, 362
- ILcdGXYViewModelTransformation**, 433
- ILcdGXYViewXYWorldTransformation**, 289, 433
- ILcdHeightProvider**
 - creating, 386
 - using, 385
- ILcdHeightProvider implementations**, 385
- ILcdIcon**, 309
 - using, 309
- ILcdInputStreamFactory**, 492
- ILcdInvalidateable.invalidateObject**, 101
- ILcdLabelConflictChecker**, 370
- ILcdLayer**, 12, 54, 56
- ILcdLayered**, 54
 - removing a layer, 372
- ILcdLayerTreeNode**, 54
- ILcdLogger**, 523
- ILcdLoggerFactory**, 525
- ILcdModel**, 10, 23, 79
- ILcdModelContainer**, 152
- ILcdModelContainerListener**, 152
- ILcdModelDecoder**, 16
- ILcdModelDescriptor**, 10, 42
- ILcdModelEncoder**, 143
 - customizing, 144

ILcdModelHeightProviderFactory properties, 386
 ILcdModelListener, 67
 ILcdModelModelTransformation, 433
 ILcdModelReference, 10, 19, 428
 ILcdModelReferenceDecoder, 137
 using, 141
 ILcdModelReferenceDecoder implementations, 138
 ILcdModelReferenceFormatter, 141
 ILcdModelReferenceParser implementations, 141
 ILcdModelTreeNode, 152
 ILcdModelXYWorldTransformation, 289, 290, 433
 ILcdModelXYZWorldTransformation, 164, 433
 ILcdMultilevelGridCoordinate, 288
 ILcdMultilevelRaster, 120
 ILcdObjectIconProvider, 309
 using, 310
 ILcdObliqueCylindrical, 423
 ILcdPerspective, 424, 425
 ILcdPoint, 91, 96
 ILcdPoint.getZ, 91
 ILcdPointList, 96
 ILcdPolygon, 95, 96
 creating, 100
 ILcdPolyline, 95, 96
 creating, 100
 ILcdPolypoint, 96
 ILcdProjection, 421
 ILcdProjection extensions, 422
 ILcdProjection implementations, 424
 ILcdRaster, 119, 507
 ILcdRaster implementations, 304
 ILcdRasterPainter, 303
 ILcdRasterReferencer, 440
 ILcdRing, 96
 ILcdSegmentScanner, 436
 ILcdSelection, 60
 ILcdSelectionListener, 66
 ILcdShape, 40, 91
 ILcdShapeList, 90, 96, 98
 ILcdSurface, 96
 ILcdText, 96
 ILcdTile, 119, 120
 ILcdTopocentricCoordSysTransformation, 434
 ILcdTopocentricReference, 430
 ILcdTransverseCylindrical, 423
 ILcdUndoable, 276, 379
 ILcdUndoableListener, 276, 379
 ILcdVariableGeoBuffer, 212
 ILcdVVGridLineOrdinateProvider, 392
 ILcdVVGridRenderer, 392, 394
 ILcdVVMModel, 391
 ILcdVVMModel implementation, 397
 ILcdVVRenderer, 392, 393
 ILcdVVXAxisRenderer, 392, 394
 ILcdXMLDatatypeUnMarshaller, 495
 ILcdXMLSchemaBasedDecoderLibrary, 492, 495
 ILcdXMLTypeUnMarshaller, 496, 498
 ILcdXYWorldReference, 19, 35, 428
 ILcdXYWorldXYWorldTransformation, 433
 ILcdXYZWorldReference, 19
 ILspCustomizableStyler, 197
 ILspEditor, 247
 implementations, 248
 responsibilities, 251
 ILspGLResourceCache, 510
 ILspInteractivePaintableLayer, 177
 ILspSnapper, 254
 ILspStyler, 29, 195
 ILspTerrainSupport, 277
 ILspView, 11
 image
 bounds, 104
 orthorectifying, 444
 processing
 multi-level images, 106
 image cache
 remove objects, 311
 image data
 using a model decoder, 104
 image date type, 104
 image domain model, 104
 image file
 creating an icon, 309
 image metadata, 104
 image normalization, 105
 image operator chains, 108
 image operators, 108
 chaining, 109
 creation, 108
 image processing, 103
 convolution, 103
 use cases, 114
 visualizing elevation data, 115
 visualizing LandSat infrared image, 114
 data type, 104
 encoding to disk, 117
 graphs, 103, 109

HDR images, 103
imaging engine, 116
metadata, 104
mosaics, 106
multi-band, 103
multi-spectral, 103
offline, 116
operator chains, 108
operators, 108
swiping, 113
tone mapping, 103
visualization, 115
image projection, 281
image semantics, 104
images
 distortion, 437
 retrieving, 305
images in labels, 218
imaging engine
 creation, 116
imaging sensor, 443
implementing a painter/editor, 325
implementing ILcdDataObject, 88
inLonLatBounds, 422
input
 touch-based, 376
input events
 dispatching, 377
input position
 interpreting, 343
integrating GML data into a LuciadLightspeed application, 464
Intel, 543
interacting with labels, 227, 365
interactive labels, 365
 activating, 366
 mouse events, 229, 367
 touch events, 229, 367
interacts2D, 97
interacts3D, 97
internal caching of images, 304
International Geomagnetic Reference Field see IGRF, 563
International Organization for Standardization, 455
interpreting satellite images, 425
interpreting the input position, 343
intersection calculation, 278
invalidate, 70
invalidateShape, 101
invokeAndWaitOnGXYLayer, 317, 410
invokeLaterOnGXYLayer, 317, 410
invokeNowOnGXYLayer, 317
inWorldBoundsOnEllipsoid, 422
inWorldBoundsOnSphere, 422
isLabelTouched, 369
ISO 19100, 459
ISO 19115 Codelists, 457
ISO 19115 domain classes, 458
ISO 19115 domain model, 457
ISO 19115 metadata model
 visualizing contents, 459
ISO 19115 standard, 456
ISO 19115 XML documents
 decoding, 492
ISO 19118 standard, 459
ISO 19139, 459
 decoding metadata, 460
ISO 19139 properties, 458
ISO metadata, 455
isogonic lines, 563
isTouched, 347
isTransparent(), 181
iterating over label positions, 370
JAI file format, 138
Java Advanced Imaging library see JAI, 138
Java content trees
 converting into XML documents, 491
Java logging framework, 524
Java VisualVM
 performance monitoring, 527
java.awt.event.ActionListener, 265, 371
java.awt.EventQueue.invokeLater, 38
java.awt.image.ColorModel, 120
java.lang.reflect, 80
java.util.Properties, 529
javax.swing.Action, 265, 371
javax.swing.Icon, 309
JConsole
 displaying performance, 526
jconsole, 525
JOGL, 545
 JPanel, 38
JPEG file format, 138
JPEG images, 140
JPL ellipsoid, 419
JPL geocentric datum, 417
jvisualvm, 527
Known issues, 543
label, 36, 215, 359

painting objects as, 361
 label algorithms, 231, 363
 label dependencies, 364
 customizing, 363
 label editor, 227
 customizing, 370
 label location
 absolute, 364
 checking, 370
 customizing, 363
 determining, 369
 determining , 363
 relative, 364
 retrieving, 363
 specifying, 363
 storing, 363
 using, 363
 label locations, 232, 362
 label painter
 customizing, 369
 text-based, 359
 label placement
 configuring, 360
 label placement algorithm
 customizing, 370
 label placers, 231, 362
 label priorities, 226
 label size, 361
 label stamp, 361
 label stamps
 customize labels, 361
 labelAnchorPointSFCT, 369
 labelBoundsSFCT, 369
 labeling, 215
 activating, 216
 anchor points, 222
 customization, 216
 customizing text, 218
 decluttering, 225
 dependent labels, 223
 icons, 218
 images, 218
 interaction, 227
 moving, 227
 multiple labels, 220
 positioning, 220
 priorities, 226
 Swing components, 219
 text, 217
 labeling a label, 364
 labeling adapted layers, 411
 labeling domain objects, 359
 labeling sequence diagram, 368
 labels, 215
 adding a halo, 362
 determine placing of, 363
 editing, 365
 interacting with, 365
 painting, 360
 process, 368
 text-based, 360
 labels 3d, 229
 large data sets
 loading, 505
 layer, 12
 add to view, 37, 50
 editing configuration, 249
 filtering objects, 171, 296
 layer control, 37
 creating, 37
 layer factories, 24
 layer factory, 17
 layer node, 54
 layer nodes
 copying between layer trees, 295
 layer properties
 changing, 70
 layer trees
 asynchronously painting, 316
 copying layer nodes, 295
 layer types, 175
 LayerControlPanelSW, 37
 level of detail of a raster, 119
 levelSwitchFactor, 305, 510, 518
 levelSwitchScales, 305, 518
 libtiff library, 507
 Lightspeed in GXY view, 405
 Lightspeed integration, 405
 2D layer adapting, 410
 3D layer adapting, 410
 adapting GXY layers, 409
 adapting layer tree nodes, 411
 converting a GXY view, 408
 converting a Lightspeed layer, 408
 converting raster and vector layers, 409
 elevation information, 408
 layer adapter performance, 411
 layer wrapper, 406
 Lightspeed layers, 406
 mixed hardware configuration, 407
 moving to Lightspeed view, 408
 paint queue, 406

painters, 409
performance tips, 407
reasons, 405
safe layer access, 407
styling, 409
Lightspeed view performance, 509
limiting a view, 158
line styles, 182
line symbols, 185
lines exceeding the Graphics clip, 314
linking objects to styles, 195
Linux, 543
Listener Pattern, 66
listeners
 using, 66
listening
 to changes in a model, 67
 to changes in a selection, 66
 to property changes, 69
loading
 business data, 39
 large data sets, 505
 model data with ILcdAction, 451
Loading a raster model, 23
Loading a SHP model, 23
loading and displaying a GML file, 464
loading and visualizing business data
 cookbook, 575
loadProperties, 530
log messages
 producing, 523
log messages in Java logging framework, 524
log messages in other than Java framework,
 525
logging and performance monitoring, 523
logging framework backend
 replacing, 525
logging frameworks, 525
logging properties file, 524
 custom, 524
logging properties for performance monitor-
 ing, 525
LonLat shapes, 92
LonLatHippodrome, 326
LuciadLightspeed
 components, 3
 documentation, 3
 samples, 3
LuciadLightspeed supported file formats, 551
magnetic variation, 563
major version, 539
managing
 image buffers, 315
 paint thread, 315
many-valued properties, 86
map projection, 18
 changing, 35
map projections, 421
 literature, 425
mapping polar areas, 425
maritime navigational charts, 424
maritime operational display systems, 424
Maritime Standards component, 309
maritime-style grid, 287
marshalling, 491
 custom, 495
Maven, 549
 Deploying, 549
maxNumberOfOutlineTiles, 304
MBean frameworks, 525
measure provider, 452
measurement customization, 271
memory saving implementations of Luciad-
 Lightspeed shapes, 92
meta model, 79
 explaining, 83
metadata, 10, 11
metadata categories in LuciadLightspeed, 457
MetadataTree, 459
method
 side effect, 97
MGRS grid, 287
MIL-STD 2525 and APP-6, 309
military symbologies, 309
minimumPoints, 148
minor version, 539
model, 10
 listening to changes, 67
 saving, 143
model and world coordinates
 transformation, 429
model coordinates, 428
 transformation to 2D world coordinates,
 433
 transformation to 3D world coordinates,
 433
 transformations with view coordinates,
 289
model data, 11
model decoders
 creating, 43
 GML, 468

using, 16
 model decoders for multilevel models, 563
 model decoders for specific file formats, 551
 model descriptor, 10
 determining, 42
 GML, 467
 model descriptor for model list, 151
 model encoders
 GML, 469
 model encoders for specific file formats, 551
 model for a regular grid, 505
 model list, 151
 creating, 151
 model metadata, 11
 model node, 151
 creating, 152
 model reference, 10, 19, 428
 determining, 42
 providing, 138
 retrieving, 137
 storing, 137, 141
 model reference for model list, 151
 model references
 transformations between, 433
 model with child models, 152
 model, world, and view coordinate systems, 427
 modelChanged, 67
 modeling
 business data, 40
 dimensional data, 133
 domain objects, 40
 geometries, 91
 multi-dimensional data, 133
 profile of vertical view, 391
 vector data, 40, 91
 models
 defining, 16
 encoding, 143
 hierarchically grouping, 151
 performance enhancing, 505
 monitoring logging and performance, 523
 monitoring performance using Java VisualVM, 527
 mosaics, 106
 mouse
 create objects, 376
 mouse events with interactive labels, 367
 mouse events with interactive lightspeed labels, 229
 mouse position
 displaying, 162, 291
 move2D, 92
 moving a shape, 92
 multilevel models
 model decoders, 563
 multi object handles, 252, 254
 multi-band images, 103
 multi-colored stroke, 314
 multi-dimensional data
 modeling, 133
 multi-level images, 106
 level of detail, 108
 multi-level raster, 120
 multi-spectral images, 103
 multi-threading, 519
 multilevel grid coordinates, 288
 multilevel grids, 288
 multilevel raster
 painting, 305
 multilevel rasters
 performance, 507
 multiple data models, 86
 multiple input points
 editing behavior, 307
 multiple labels, 369
 multiple objects, 98
 mutable stylers, 197
 Mutable styles, 197
 MVC architecture, 9
 naming conventions, 5
 national grid systems, 424
 native application integration, 535
 naval symbology, 309
 navigation controller, 27
 navigation on a Lightspeed view, 166
 NIMA geotic datum, 417
 non-georaster formats, 104
 non-parametric and parametric rectification
 combining, 444
 non-parametric rectification, 437
 non-rectified grid reference
 transformations, 437
 non-standard input, 276, 377
 non-warping mode, 517
 normalization, 105
 notifying
 a painter of changes, 311
 objects of changes, 65
 objects of changes to shapes, 101
 objects of changes without listeners, 69
 numberOfRowsInSectionBackgroundLayers, 517

- NVIDIA Optimus, 543
- object editing, 247
- object elevation, 204
- object graph
 - traversing, 87
- objects
 - changing, 65
 - notifying of changes, 65
- oblique cylindrical projection, 423
- Observer pattern, 66
- obstacle providers, 229
- off-screen view, 159
- offset icons, 361
- OGC, 463
 - OGC Filter comparison conditions, 481
 - OGC Filter conditions and LuciadLightspeed classes, 478
 - OGC Filter expressions, 476
 - OGC Filter expressions and LuciadLightspeed classes, 476
 - OGC Filter logic conditions, 481
 - OGC Filter spatial conditions, 481
 - OGC Filters, 471
 - OGC SLD, 483
 - OGC SLD styling see SLD styling, 483
- Open Geospatial Consortium documents 02-059, 04-095 and 09-026, 471
- Open Geospatial Consortium see OGC, 463
- OpenCL report, 546
- OpenGIS Filter Encoding Implementation specification, 471
- OpenGL, 545
 - Compatibility, 547
 - Draping algorithm, 511
 - Extra information, 546
 - FBO, 511
 - JOGL, 545
 - Resource cache, 510
 - Transparency, 511
- OpenGL error debugging, 547
- OpenGL extensions, 545
- OpenGL profile, 545
- OpenGL report, 546
- optimizing performance of raster painters, 507
- origin of projection plane, 443
- orthorectification, 442
- orthorectified projection, 443
- orthorectifying an image, 444
- OSGi
 - creating bundles for OSGi, 533
- outlineColor, 304
- output arguments, 97
- OverlapBehavior, 490
- paint blocks, 320
- paint context, 298
- paint hints
 - using, 319
- paint mode, 298
- paint queue, 315
 - waiting, 321
- paint queue assignments
 - customizing, 319
 - debugging, 319
- paint queues
 - sharing, 318
- paint representations, 175
- paint sequence diagram, 297
- paint states, 176
- paintCache, 304
- painter, 12, 36
 - notification of changes, 311
- painter modes, 328
- painter provider, 36
- painters for halos, 311
- painters for rasters, 298
- painters for shapes, 298
- paintGXYView, 297, 368
- painting
 - a multilevel raster, 305
 - background image buffers, 315
 - body of object, 328
 - domain objects, 297
 - handles of object, 328
 - labels, 360, 362
 - objects as a label, 361
 - rasters, 303
 - small rasters, 305
 - symbology, 309
 - View coordinates, 189
- painting and editing support, 307
- painting order
 - changing, 173
- painting performance
 - enhancing, 507
- painting symbols along lines, 185
- paintLabel, 369
- paintOutline, 304
- panel of vertical view, 392
- panning
 - performance enhancements, 517
- parametric and non-parametric rectification
 - combining, 444

- parametric rectification, 442
- patch update, 539
- patch version, 539
- pen, 36
- Performance
 - Draping, 511
 - Garbage collection, 513
 - Lightspeed views, 509
 - Memory guidelines, 512
 - OpenGL resource cache, 510
 - Transparency, 511
 - Video driver, 513
 - Video memory allocation, 513
- performance
 - displaying in JConsole, 526
 - multilevel rasters, 507
 - panning, 517
 - raster painters, 507
 - tiled rasters, 507
- performance guidelines, 505
- performance monitoring, 523
 - logging properties, 525
- performance monitoring using Java VisualVM, 527
- performance of models
 - enhancing, 505
- performance of painting
 - enhancing, 507
- perspective projection, 442
 - rectified, 442
- pixel coordinates, 429
- pixel density of a raster, 119
- placing labels, 230
- plotpainting, 204
- PNM file format, 138
- point height, 91
- point of origin
 - projection, 422
- point on terrain, 278
- Point styling, 201
- points, 185
 - transforming, 433
- polar areas
 - mapping, 425
 - UTM grid, 425
- polygon, 95
- polyline, 95
- porthole, 272
- positioning labels, 220
- prerequisites, 4
- primitive type, 84, 86
- printing, 399
- producing log messages, 523
- PROJ file format, 140
- projection
 - azimuthal, 423
 - conical, 423
 - cylindrical, 422
 - oblique cylindrical, 423
 - orthorectified, 443
 - point of origin, 422
 - transverse cylindrical, 423
- projection plane
 - origin, 443
- projections, 421
 - literature, 425
- properties, 256
 - for representing geometries, 90
- Properties file
 - sample, 531
- properties files
 - configuring classes, 529
- properties for painting rasters, 303
- properties for transforming rasters, 304
- properties handles, 253
- Properties object, 529
 - creating, 531
- properties with many values, 86
- Property Change Listener, 69
- property changes
 - listening to, 69
- Property file
 - using, 531
- property types
 - ISO 19139, 458
- propertyChanged, 69
- provideInteractiveLabel, 366
- providing
 - a model reference, 138
 - referencing information for raster data, 138
- putCornerIcon, 309
- QName, 492
- querying tile-based terrain data, 278
- raster
 - adding to a model, 441
 - brightness, 303
 - color model, 303
 - level of detail, 119
 - painting, 303
 - painting properties, 303

pixel density, 119
properties for transforming, 304
retrieving single values, 305
transparency, 303
raster border, 304
raster contours, 178
raster data, 10
 providing referencing information, 138
raster hatch pattern, 178
raster layers
 creation, 177
raster painter settings
 tuning, 517
raster painters, 298
 optimizing performance, 507
raster referencer
 rectifying an image, 441
raster referencers, 440
raster styles, 193
raster with different levels of detail, 120
rasters
 combining, 508
rasters and tiles, 121
rectification, 437
 non-parametric, 437
 parametric, 442
rectified grid reference
 transformations, 438
rectified perspective projection, 442
rectifying an image with a raster referencer,
 441
reducing label cluttering, 361
REF file format, 140
reference data, 11
reference system
 Cartesian/grid, 18
 customized, 432
 geodetic, 18
regular grid
 model, 505
relative label location, 364
release notes, 3
remote sensing, 103
RemoveLayerAction, 372
removing
 balloon, 161
removing a layer from an ILcdLayered, 372
removing handles, 256
rendering
 the ILcdAWTPPath, 339
 the vertical view profile, 393
replacing the logging framework backend, 525
repositioning a shape, 329
representing
 domain objects, 79
 geometries with TLcdDataModel, 90
 geometry information with properties, 90
RESHAPING, 329
responsive
 user interface, 315
retrieveTile, 119
retrieveValue, 119, 120
retrieving
 a model reference, 137
 height data, 385
 images, 305
 label locations, 363
 selected objects, 375
 single raster values, 305
 tile contents, 120
 value of a tile, 119
retrieving painted objects at the mouse loca-
tion, 171
retrieving painted objects in the view, 171
retrieving terrain data, 277
retrieving view bounds of painted objects, 171
Retrieving world position, 162, 291
reuseInternalBuffer, 304
RGB values, 303
RGBImageFilter, 303
roll angle, 443
RST file format, 139
ruler customization, 271
S-52 symbology, 309
sample Properties file, 531
samples.common.layerControls.swing, 37
samples.metadata.gazetteer, 460
sampling line segments, 436
satellite images
 interpreting, 425
saving model data, 143
scanSegment, 436
screen coordinates, 429
Select controller model, 269
selectable objects, 61, 63
selectByWhatMode, 373
 defining, 374
SELECTED, 329
selected objects
 retrieving, 375
selectHowMode, 373
 defining, 374

selection
 listening to changes, 66
 selection and styling information, 490
 selection between views, 170, 295
 selection customization, 268
 selection logic, 269
 selection modes of `TLcdGXYSelectController-Model2`, 373
 selection of objects
 defining, 374
 selection state
 updating, 375
 selection status of objects, 328
 selectionCandidates, 373
 defining, 375
 selectionChanged, 66
 semantics, 104
 set of points, 96
 setBalloonDescriptor, 160
 setBodyLabel, 364
 setColorModel, 120
 setDoubleClickAction, 373
 setGXYViewLabelPlacer, 360, 362
 setHaloColor, 311
 setHaloEnabled, 311
 setHaloThickness, 311
 setMainProfileRenderingMode, 394
 setMouseDraggedSensitivity, 373
 setParentLabel, 364
 setProfileColor, 394
 setProvideInteractiveLabelOnMouseOver, 229,
 366
 setRightClickAction, 373
 setSelectControllerModel, 373
 setSubProfileColor, 394
 setSubProfileRenderingMode, 394
 setSynchronousDelay, 321
 setting
 an `ILcdGXYPen` to a layer, 308
 setTo2DIntersection, 97
 setTo2DUnion, 98
 setToIncludePoint2D, 97
 SFCT, 97
 shape
 adapting a part of, 329
 moving, 92
 repositioning, 329
 translating, 92
 shape painters, 298
 shapeChanged, 101
 shapes
 commonly used, 94
 shapes for GML geometries, 468
 sharing paint queues, 318
 side effect of method, 97
 Simple Features Profile
 GML, 466
 simple geometry, 91
 SimpleFormatter, 524
 single data model, 86
 size of label, 361
 SLD feature type style, 486
 SLD styling, 483
 SLD styling model, 486
 SLD styling models
 decoding, 489
 encoding, 489
 slow painting, 315
 slow rendering performance, 314
 snappables, 333
 snapping, 254, 333
 snapping and styling information, 490
 snapping implemented, 352
 snapping methods in `ILcdGXYEditor`, 333
 snapping methods in `ILcdGXYPainter`, 333
 snapping targets, 328
 SNAPS, 328
 snapTarget, 333
 sparse grids, 106
 specifying
 a custom logging properties file, 524
 label location, 363
 spherical calculations, 435
 spherical trigonometry operations, 435
 startInteraction, 228
 startResolutionFactor, 303, 510, 518
 static data, 43
 StAX, 491
 stopInteraction, 366
 stopResolutionFactor, 303, 518
 storing
 a model reference, 141
 a model references, 137
 label locations, 363
 streaming API for XML, 491
 stroke algorithm, 312
 style, 175
 style animating, 199
 style builders, 182
 style collectors, 177
 style target provider, 253
 Style toggling, 198

stylers, 177
creation, 196
customizing, 197

styles, 176
creation, 182
dynamic, 198

Styling
Target specification, 199
Editable stylers, 198
toggling styles, 198

Styling a point list, 201

Styling geometry, 195

styling information and selection, 490

styling information and snapping, 490

styling objects, 29

styling point as circle, 201

styling raster data bounds, 194

Styling terrain, 204

super type
choosing, 86

support for painting and editing, 307

supported file formats in LuciadLightspeed, 551

supported GML versions, 463

supportSnap, 333

Swing components in labels, 219

swiping, 113, 272

switching between 2D and 3D views, 27

symbology, 185
painting, 309

TAB file format, 139

TAB files
decoding, 139

Target styling, 199

targets for snapping, 328

technical notes, 4

terrain data, 277

terrain data querying, 277

terrain elevation, 203

text in labels, 217

text string, 96

text-based label painter, 359

text-based labels, 360

TFW file format, 140

TIFF file format, 138

tiffinfo, 507

tile, 119, 120
retrieving value of, 119

tile contents
retrieving, 120

tile resolution, 120

tile size, 120

tile-based terrain data, 278

tiled GeoTIFF images, 507

tiled rasters
performance, 507

tiles and rasters, 121

TLcd2DBoundsIndexedModel, 43

TLcd2DBoundsIndexedModelTreeNode, 152

TLcd2DRegularTiledBoundsIndexedModel, 505

TLcdAllInMemoryRasterPainter, 305

TLcdAnchoredIcon, 309

TLcdAUTO2ReferenceParser, 141

TLcdAUTOReferenceParser, 141

TLcdAWTEvent, 377

TLcdBandMergeOp, 109

TLcdBandSelectOp, 109

TLcdBILModelDecoder, 140

TLcdBinaryOp, 110

TLcdCartesian, 436

TLcdCassini, 425

TLcdCluster, 150

TLcdClusterDataTypes, 150

TLcdClusteringTransformer, 147

TLcdColorConvertOp, 110

TLcdColorLookupOp, 110

TLcdCompositeHeightProvider, 385

TLcdCompositeModelHeightProviderFactory, 386

TLcdCompositeOp, 110

TLcdConvolveOp, 110

TLcdCropOp, 111

TLcdCurvesOp, 111

TLcdDataModel, 80, 83, 89

TLcdDataModel and geometries, 90

TLcdDataModelBuilder, 82, 85

TLcdDataProperty, 79, 84

TLcdDataPropertyBuilder, 85

TLcdDataType, 79, 84

TLcdDataTypeBuilder, 85

TLcdDefaultVVGridLineOrdinateProvider, 395

TLcdDefaultVVGridRenderer, 394

TLcdDefaultVVRenderer, 394

TLcdDefaultVVRendererJ2D, 394

TLcdDHNGermanGridReference, 432

TLcdDMA1987bEllipsoidFactory, 419

TLcdEditableModelListDescriptor, 151

TLcdEllipsoid, 419

TLcdEllipsoidUtil, 435

TLcdEPSGGeodeticDatumFactory, 417

TLcdEPSGReferenceParser, 141

TLcdEquidistantCylindrical, 35, 424
TLcdExpandOp, 112
TLcdFixedHeightProvider, 385
TLcdGCODATypes, 458
TLcdGenericGeodeticDatumFactory, 417
TLcdGeoAnchoredBalloonDescriptor, 161
TLcdGeocentric2Grid, 433
TLcdGeodetic2Geocentric, 433
TLcdGeodetic2Grid, 433
TLcdGeodeticDatum, 35, 416
TLcdGeodeticPen, 36, 308
TLcdGeodeticReference, 42
TLcdGeodeticSegmentScanner, 436
TLcdGeoidGeodeticDatumFactory, 417
TLcdGeoReference2GeoReference, 433
TLcdGeoTIFFModelDecoder, 34, 140
TLcdGeoTIFFModelEncoder, 143
TLcdGLDebugGL, 547
TLcdGMDDATypes, 458
TLcdGML2DATypes, 467
TLcdGML2ModelDecoder, 469
TLcdGML2ModelDescriptor, 467
TLcdGML31DATypes, 467
TLcdGML31ModelDecoder, 469
TLcdGML31ModelDescriptor, 467
TLcdGML32DATypes, 467
TLcdGML32ModelDecoder, 469
TLcdGML32ModelDescriptor, 467
TLcdGML32ModelEncoder, 469
TLcdGnomonic, 424
TLcdGrid2Geocentric, 433
TLcdGrid2Grid, 433
TLcdGridPen, 308
TLcdGridReference, 35, 428
TLcdGridReferenceUtil, 434
TLcdGridSegmentScanner, 436
TLcdGSRDataTypes, 458
TLcdGSSDataTypes, 458
TLcdGTSDATypes, 458
TLcdGXYArcBandPainter, 298
TLcdGXYArcPainter, 298
TLcdGXYAsynchronousEditableLabelsLayer-Wrapper, 321
**TLcdGXYAsynchronousEditableLabelsLayerWrap-
per**, 316
TLcdGXYAsynchronousLabelPlacer, 362, 363
TLcdGXYAsynchronousLayerTreeNodeWrapper, 316
TLcdGXYAsynchronousLayerWrapper, 316, 320
TLcdGXYAsynchronousPaintQueueManager, 319
TLcdGXYBoundsPainter, 298
TLcdGXYCirclePainter, 298
TLcdGXYCircularArcPainter, 298
TLcdGXYComplexStroke, 312
TLcdGXYCompositeDiscretePlacementsLabelingAlgorithm, 363
TLcdGXYCompositeLabelingAlgorithm, 363
TLcdGXYCompositeLabelPainter, 361
TLcdGXYCurvedPathLabelingAlgorithm, 363
TLcdGXYDataObjectLabelPainter, 359, 361
TLcdGXYDataObjectPolylineLabelPainter, 50
TLcdGXYEditController2
 customizing, 375
 editing a shape, 346
 using, 375
TLcdGXYEllipsePainter, 298
TLcdGXYGeoBufferPainter, 298
TLcdGXYHaloPainter, 311
TLcdGXYHatchedFillStyle, 312
TLcdGXYIconPainter, 298, 309–311
 using, 310
TLcdGXYImagePainter, 36, 116
TLcdGXYInPathLabelingAlgorithm, 363
TLcdGXYInteractiveLabelsController, 365
TLcdGXYLabelPainter, 360
TLcdGXYLabelPainterAdapter, 361
TLcdGXYLabelPlacer, 362
TLcdGXYLayer, 36
TLcdGXYLocationListLabelingAlgorithm, 360, 363
TLcdGXYLspAsynchronousLayerWrapper, 406
TLcdGXYLspAsynchronousPaintQueue, 407
TLcdGXYLspAsynchronousPaintQueueManager, 407
TLcdGXYNewController2
 creating a shape, 340
 customizing, 376
 using, 376
TLcdGXYOnPathLabelingAlgorithm, 363
TLcdGXYPainterColorStyle, 311
TLcdGXYPen, 308
TLcdGXYPointListPainter, 49, 298, 299
TLcdGXYRoundedPointListPainter, 299
TLcdGXYSelectController2
 customizing, 373
 using, 373
TLcdGXYSelectControllerModel2
 selection modes, 373
TLcdGXYSetControllerAction, 372

TLcdGXYShapeListPainter, 298
TLcdGXYShapePainter, 298, 299
TLcdGXYStampLabelPainter, 360, 361
TLcdGXYSurfacePainter, 298
TLcdGXYTextPainter, 298
TLcdGXYTouchNavigateController, 377
TLcdGXYTouchNewController, 377
TLcdGXYTouchSelectEditController, 376
TLcdGXYViewFitAction, 372
TLcdGXYViewXYWorldTransformation, 433
TLcdHaloIcon, 309
TLcdHasAShapeAnnotation, 90
TLcdHistogramOp, 111
TLcdIconFactory, 309
TLcdIconSW, 309
TLcdIGRFModelDecoder, 563
TLcdImageHeightProvider, 385
TLcdImageIcon, 309
TLcdImageModelHeightProviderFactory, 386
TLcdIndexLookupOp, 112
TLcdISO19115Citation, 457
TLcdISO19115Code, 457
TLcdISO19115DataTypes, 458
TLcdISO19115Distribution, 458
TLcdISO19115Metadata, 457–459
TLcdISO19139MetadataDecoder, 460
TLcdJAIRasterModelDecoder, 138
 using, 138
TLcdJPLEllipsoidFactory, 419
TLcdJPLGeodeticDatumFactory, 417
TLcdJULLoggerFactory, 525
TLcdJULSimpleFormatter, 524
TLcdLambert1972BelgiumGridReference,
 424, 432
TLcdLambertConformal, 424
TLcdLambertFrenchGridReference, 424, 432
TLcdLoggerFactory, 523
TLcdLonLatBorderGridPainter, 287
TLcdLonLatGridPainter, 311
TLcdLonLatPoint, 40
TLcdLonLatPolyline, 40
TLcdMapGeorefGridLayer, 287
TLcdMap JPanel, 35
 world reference, 35
TLcdMapLonLatGridLayer, 287
TLcdMapTouchRulerController, 377
TLcdMapWorldMapWorldTransformation,
 433
TLcdMBeanPerformanceLogHandler, 525
TLcdMedianOp, 113
TLcdMercator, 424
TLcdMessageIcon, 309
TLcdMGRSGridLayer, 287
TLcdMIFModelEncoder, 143
TLcdMIFModelReferenceParser, 141
TLcdModelElementBalloonDescriptor, 161
TLcdModelList, 151
TLcdModelMetadata, 11
TLcdModelReferenceFactory, 432
TLcdModelTreeNode, 152
TLcdMoveLayerAction, 372
TLcdMultilevelGridCoordinateModel, 288
TLcdMultilevelRasterPainter, 298, 305
TLcdMultilevelRegularTiledModelDecoder,
 563
TLcdMultilevelTiledModelDecoder, 563
TLcdNIMA8350GeodeticDatumFactory, 417
TLcdNoWarpMultilevelRasterPainter, 305
TLcdNoWarpRasterPainter, 305
TLcdNullLoggerFactory, 525
TLcdObliqueMercator, 425
TLcdOpenAction, 371
TLcdOrthographic, 424
TLcdOrthorectifiedProjection, 443
TLcdPerspectiveProjection, 442
TLcdPixelRescaleOp, 113
TLcdPixelTransformOp, 113
TLcdPolarStereographic, 425
TLcdPolynomialRasterReferencer, 440
TLcdPrintComponentAction, 372
TLcdProjectionFactory, 424
TLcdProjectionPen, 308
TLcdProjectiveRasterReferencer, 440
TLcdRasterHeightProvider, 385
TLcdRasterModelDecoder, 139
 using, 139
TLcdRasterModelHeightProviderFactory, 386
TLcdRasterPainter, 298, 304
TLcdRationalRasterReferencer, 440
TLcdRD1918DutchGridReference, 424, 432
TLcdRectifiedPolynomialProjection, 439
TLcdRectifiedProjectiveProjection, 439
TLcdRectifiedRationalProjection, 439
TLcdRedoAction, 276, 379
TLcdRegularTiled2DBoundsIndexedModel,
 505
TLcdResizeOp, 113
TLcdRhumblineSegmentScanner, 436
TLcdSaveAction, 371
TLcdSelectionMediator, 171, 296
TLcdSemanticsOp, 113
TLcdShapeAnnotation, 90

TLcdShapeListAnnotation, 90
 TLcdSHPMModelEncoder, 143
 TLcdSimpleLoggerFactory, 525
 TLcdSingleGXYPainterProvider, 300
 TLcdSLDFeatureTypeStyle, 489
 TLcdSLDFeatureTypeStyleDecoder, 489
 TLcdSLDFeatureTypeStyleEncoder, 489
 TLcdSpatialReferenceParser, 141
 TLcdSphereUtil, 419, 435
 TLcdStereographic, 424
 TLcdStrokeLineStyle, 312
 TLcdStrokeLineStyleBuilder, 312
 TLcdSWAction, 265, 371
 TLcdSWIcon, 310
 TLcdSwipeOp, 113
 TLcdSwissGridReference, 432
 TLcdSymbol, 309
 TLcdTABRasterModelDecoder, 139
 TLcdTFWRasterModelDecoder, 140
 TLcdTopocentric2Grid, 433
 TLcdTopocentricCoordSysTransformation, 434
 TLcdTouchDevice.getTouchDeviceStatus(), 378
 TLcdTouchEvent, 378
 TLcdTouchPoint, 378
 TLcdTransformedHeightProvider, 385
 TLcdTransverseMercator, 424
 TLcdUnanchoredBalloonDescriptor, 161
 TLcdUndoAction, 276, 379
 TLcdUndoManager, 62, 276, 379
 TLcdUTMGrid, 424
 TLcdUTMGridSystem, 424
 TLcdVectorModel, 43
 TLcdVerticalPerspective, 425
 TLcdViewAnchoredBalloonDescriptor, 161
 TLcdViewHeightProvider, 388
 TLcdVVJPanel, 392
 TLcdVVTerrainModel, 392
 TLcdVVWithControllersJPanel, 393, 395
 TLcdWarpMultilevelRasterPainter, 305
 TLcdWarpRasterPainter, 305
 TLcdWKTReferenceParser, 141
 TLcdWMMModelDecoder, 563
 TLcdXLinkDataTypes, 492
 TLcdXMLBuiltInConstants, 492
 TLcdXMLBuiltInDataTypes, 492
 TLcdXMLDataModelBuilder, 493
 TLcdXMLDataObjectDecoderLibrary, 495
 TLcdXMLDataObjectSchemaHandler, 502
 TLcdXMLJavaClassResolver, 502
 TLcdXMLSchemaBasedDecoder, 469, 492, 496
 TLcdXMLSchemaBasedEncoder, 469, 496
 TLcdXMLSchemaBasedMapping, 492, 496
 TLcdXMLSchemaElement, 492
 TLcdXMLSchemaElementIdentifier, 492
 TLcdXMLSchemaMappingAnnotation, 494–496
 TLcdXMLSchemaType, 492
 TLcdXMLSchemaTypeIdentifier, 492
 TLcdXMLSchemaTypeMappingAnnotation, 494
 TLcdXMLTypeUnmarshallerProvider, 498
 TLcdXYGridLayer, 287
 TLcdZipModelListDecoder, 151
 TLsp2DPointListEditor, 248
 TLspAboveTerrainCameraConstraint3D, 158
 TLspArcEditor, 248
 TLspAWTView, 22
 TLspBalloonManager, 160
 TLspCircleEditor, 248
 TLspColorLookupTableFilterStyle, 194
 TLspComplexStrokedLineStyle, 185
 TLspCompositeDiscreteLabelingAlgorithm, 231
 TLspCompositeEditor, 248
 TLspCompositeLabelingAlgorithm, 231
 TLspCreateController, 255
 TLspCurvedPathLabelingAlgorithm, 231
 TLspCustomizableStyle, 197
 TLspCustomizableStyler, 198
 TLspEditableStyler, 198
 TLspEditController, 249, 251, 270
 TLspEditOperation, 251
 TLspExtrudedShapeEditor, 248
 TLspFillStyle, 183
 TLspFlickerController, 272
 TLspGeorefGridLayerBuilder, 162
 TLspGXYLayerAdapter, 409
 TLspIconStyle, 185
 TLspImageProcessingStyle, 115
 TLspInPathLabelingAlgorithm, 231
 TLspLabelingAlgorithm, 231
 TLspLabelPainter, 215
 TLspLabelPlacer, 231
 TLspLayer, 177
 TLspLineStyle, 182
 TLspLonLatGridLayerBuilder, 24
 TLspLookAtTrackingCameraConstraint2D, 158
 TLspLookAtTrackingCameraConstraint3D, 158

TLspLookFromTrackingCameraConstraint3D, 159
TLspMultiObjectTranslationHandle, 254
TLspObjectTranslationHandle, 252
TLspOnPathLabelingAlgorithm, 231
TLspOpenGLProfile, 547
TLspPaintRepresentation, 211
TLspPaintRepresentationState, 211
TLspPaintState, 211
TLspPlatformInfo, 546
TLspPointSetHandle, 252
TLspPointTranslationHandle, 252
TLspPortholeController, 272
TLspRasterLayer, 178
TLspRasterLayerBuilder, 24, 177
TLspRasterStyle, 193
TLspRulerController, 271
TLspSelectController, 268
TLspShapeEditor, 248
TLspShapeLayerBuilder, 24, 177
TLspShapePainter, 177
TLspStaticCreateHandle, 255
TLspStyler, 197
TLspSwipeController, 272
TLspTextStyle, 215
TLspVerticalLineStyle, 184
TLspViewBuilder, 408
TLspViewNavigationUtil, 166
TLspViewServices, 277
TLspViewTransformationUtil, 27
TLspViewXYZWorldTransformation2D, 164
TLspViewXYZWorldTransformation3D, 165
TLspWorldSizedLineStyle, 189
TLspXYGridLayerBuilder, 162
tone mapping, 103
toolbar 2D and 3D, 28
touch controller
 custom, 377
 customization, 274
 using, 376
touch controllers, 273
touch events
 creating, 378
 receiving, 378
 with interactive labels, 367
 with interactive lightspeed labels, 229
touch-based input device, 376
touched
 checking, 347
touchPointAvailable, 377
touchPointMoved, 377
touchPointWithDrawn, 377
transformation
 between coordinate systems, 433
 between model and world coordinates, 429
 between model coordinates and 2D world coordinates, 433
 between model coordinates and 3D world coordinates, 433
 between model references, 433
 between world and view coordinates, 429
 between world coordinates and view coordinates, 433
 between world references, 433
transformations, 427
transformations between geodetic and world coordinates, 421
transformations between view and model coordinates, 289
transformations in a non-rectified grid reference, 437
transformations in a rectified grid reference, 438
transforming a bounding box from view to model coordinates, 291
transforming bounding boxes, 433
transforming points, 433
translate2D, 92
TRANSLATING, 329
translating a shape, 92
transparency of rasters, 303
transparent access to a custom layer interface, 320
transverse cylindrical projection, 423
traversing
 object graph, 87
Troubleshooting, 543
 Graphics drivers, 543
tuning
 DMED/DTED fall-back, 507
 raster painter settings, 517
type information
 browsing, 82
typographical conventions, 5
undo and redo in Lightspeed view, 61
undo manager, 276
undo/redo support
 adding, 276, 379
unmarshalling, 491, 498
 custom, 495
XML child elements, 501

- XML simple types, 495
- unmarshalType, 498
- unprojected geodetic coordinates, 424
- up vector, 443
- updating selection state, 375
- UPS grid, 287
- useDeferredSubTileDecoding, 304
- user interface
 - freezing, 315
 - responsive, 315
- User styles, 197
- useSubTileImageCaching, 304
- using
 - a model decoder for image data, 104
 - a Properties file, 531
 - a TLcdGXYEditController2, 375
 - a TLcdGXYNewController2, 376
 - a TLcdGXYSelectController2, 373
 - an array of ILcdGXYPainterProvider objects, 301
 - an ILcdGXYPainter as ILcdGXYPainterProvider, 300
 - an ILcdGXYPainterProvider on a TLcdGXYLayer, 300
 - an ILcdGXYPainterProvider on composite shapes, 301
 - ILcdGXYEditor, 306
 - ILcdGXYEditorProvider, 307
 - ILcdGXYPainter, 298
 - ILcdGXYPainterProvider, 300
 - ILcdGXYPen, 307
 - ILcdIcon, 309
 - ILcdModelReferenceDecoder, 141
 - ILcdObjectIconProvider, 310
 - label locations, 363
 - listeners, 66
 - paint hints, 319
 - TLcdGXYIconPainter, 310
 - TLcdJAIRasterModelDecoder, 138
 - TLcdRasterModelDecoder, 139
 - touch controllers, 376
- UTM grid, 287
- UTM grid for polar areas, 425
- UTM grid system, 424
- vector data, 10
 - modeling, 40, 91
- vector layers
 - creation, 177
 - customization, 177
- vector styles, 182
- version
 - major, 539
 - minor, 539
 - patch, 539
- vertical datum, 415
- vertical line styles, 184
- vertical view
 - creating, 391
 - panel, 392
 - rendering the profile, 393
 - use case, 395
- vertical view profile
 - modeling, 391
- Video driver, 513
- Video memory allocation, 513
- view, 11
 - adding to a window, 38
 - centering, 158
 - constraining, 158
 - creating, 17, 35
 - defining, 17
 - fitting, 158
 - fitting to layer, 47, 50
 - limiting, 158
 - navigation, 166
 - positioning 2D, 164
 - positioning 3D, 165
- view bounds
 - adapting, 344
- View coordinate painting, 189
- view coordinates, 429
 - transformations with model coordinates, 289
- viewAWTBounds2worldSFCT, 290
- viewAWTPoint2worldSFCT, 289
- viewXYBounds2worldSFCT, 290
- viewXYPoint2worldSFCT, 289
- visual inspection, 272
- visualizing
 - business data, 39
 - contents of an ISO 19115 metadata model, 459
- visualizing handles, 253
- visualizing object elevation, 204
- visualizing terrain elevation, 203
- waiting paint queue, 321
- warpBlockSize, 304
- warped rasters, 305
- warping
 - avoiding, 517
- Well-Known Text format, 141
- WGS 1984

see *World Geodetic System 1984*, 18
WGS84, 35
WMM
 model decoder, 563
WMS reference information, 141
world and view coordinates
 transformation, 429
world coordinates, 421, 428
 transformation to view coordinates, 433
World Geodetic System 1984, 18, 42
World Magnetic Models see WMM, 563
world maps, 424
world reference, 11, 19, 428
 defining, 17
 TLcdMap JPanel, 35
worldBounds2modelSFCT, 290
worldPoint2modelSFCT, 164, 290

X and Y coordinates, 421
XLink references, 470
XLink schema, 492
XML
 streaming API, 491
XML 1.0 specification, 491
XML child elements
 unmarshalling, 501
XML complex types
 decoding, 496
XML data, 491
XML decoding
 custom, 495
XML document
 converting into Java content trees, 491
 decoding, 492
XML documents with unknown schema
 decoding, 502
XML runtime schema extension, 502
XML schema, 491
 creating a data model, 492
XML schema types
 complex, 496
 custom domain classes, 494
XML schemas, 465
XML simple types
 unmarshalling, 495
XML type extension, 498
XMLInputFactory, 492
XMLStreamReader, 498
XSD type extension, 498
XY grid, 162, 287
XY shapes, 92
XYHippodrome, 326