



JZY3D

Developer Documentation

API version: 0.9.1
Last update: 2013/10/20

<https://github.com/jzy3d/jzy3d-api/tree/0.9.1>

Summary

Chapter 1 : Charts	6
Main components	6
Quality	6
Running charts	7
Destroying charts	7
Chapter 2 : Drawables	8
Overview	8
Base features	8
Drawables	8
Wireframeables	9
Composites	9
Enlightables	9
Selectable	10
Pickable	10
Textured	10
Faster drawables with Vertex Buffer Objects	10
Text	11
Text renderers	11
Drawable text	11
Chapter 3 : Drawables for charts	12
Surfaces	12
Surface defined by a function	12
Surface defined by input points	14
Surface defined by polygons	15
Scatters	15
Histograms & bar charts	15
Chapter 4 : Colormaps	16
Overview	16
Apply a colormap to a drawable	17
Create a colorbar legend	17
Chapter 5 : Layout	18
View Layout	18
Maximize	18
Squarify	18
Text embedding	18
Orthogonal and perspective projections	19

View constraints.....	19
Viewpoint	19
Post renderers	19
Background	19
Tooltips	20
Chart layout.....	20
Axis Layout.....	21
Chapter 6 : Contour charts	22
Overview	22
Configuration	22
Contour lines	23
Filled Contour	23
Height Map Contour	23
Chapter 7 : Controllers.....	24
Mouse controllers	24
Controlling camera	24
Interaction with drawables	24
Threaded controllers	25
Keyboard controllers	25
Chapter 8 : Interactive objects	26
Overview	26
Picked objects	26
Selectable objects	27
Chapter 9 : Scene graph.....	28
Transforms	28
Transformations for Drawables.....	28
Animating a drawable transformation	28
Chapter 10 : Animations	29
Remapping surfaces	29
Adding and removing drawables dynamically	30
Chapter 11: Transparency.....	31
Handling transparency by sorting polygons	31
Handling transparency with a depth peeling algorithm	31
Hints with transparency	31
Chapter 12 : Events	32
View events	32
View point changed	32
View point reached top or bottom	32
Controller events	32

Drawable events	32
Chapter 13 : Maths and statistics package	33
Coordinates	33
Bounding boxes	33
Statistics and array processors	33
org.jzy3d.maths.Array	34
org.jzy3d.maths.Statistics	34
org.jzy3d.maths.Utils	34
Chapter 14 : Integrate the chart in your application	35
Understanding canvases	35
AWT applications	35
NEWT	35
Swing applications	35
SWT applications	35
Offscreen applications	36
Screenshots	36
Testing charts	36
Chapter 15 : Component injection	37
Chapter 16 : Rendering	38
Chapter 17 : Going further with OpenGL	39

Legend

Demo package or class

Warning

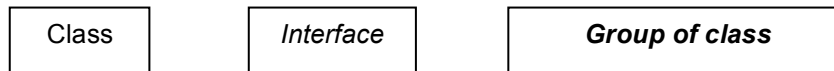
Note

Java code

Chapter reference

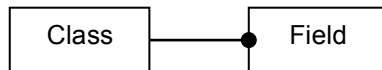
Schemas

We will use the following drawing convention for schemas

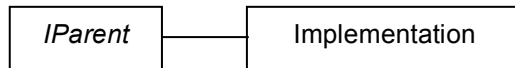


By group of class we mean a set of tool that will be detailed later in another schema of this manual.

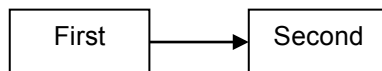
To explain the which building blocks are used by a class, we draw a line finished by a dot near the field class.



To explain inheritance or implementations, we use normal line, where the parent class or interface stand on top or on the left



To show a sequence of components, we use lines with arrow:

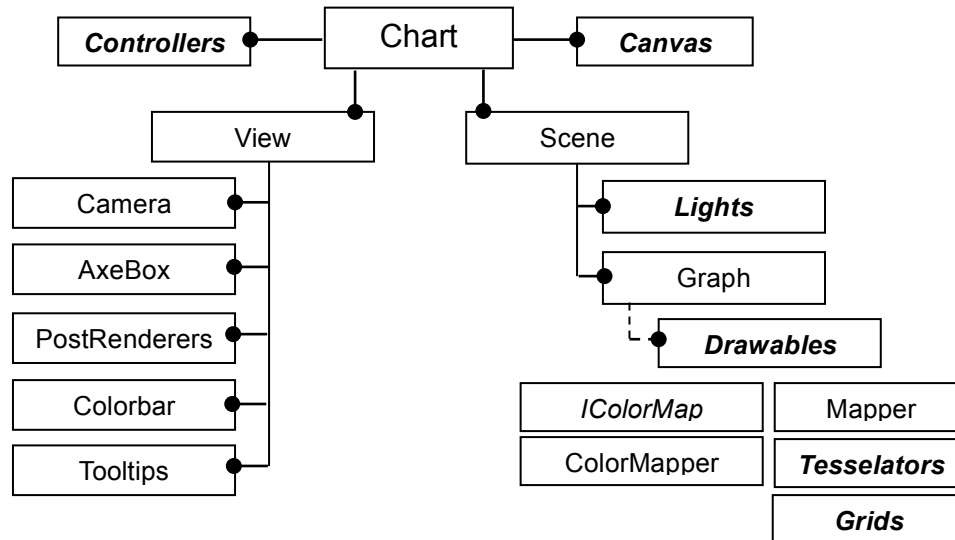


Chapter 1 : Charts

Main components

A Chart is a convenient object gathering what you need to plot and control 3d content easily:

- a Scene, backed by a scene Graph, that stores AbstractDrawable objects.
- a View, that handles the scene layout and annotations: axes, tooltips, post renderers, background image, etc. The default View has children classes that are also able to manage several image sources (Open GL scene, Java 2D colorbars, etc) to be laid out in a single Canvas.



To add drawable elements to the chart, simply call:

```
chart.getScene().getGraph().add(drawable);
```

To get the view and customize the chart appearance, call:

```
View view = chart.getView();
```

Quality

The chart's *Quality* let you configure the tradeoff between the performance and the quality of your chart. Chart's *Quality* is configured in its constructor and can't be changed at runtime. Four quality modes are available:

- **Fastest:** No transparency, no color shading, just uses the depth buffer.
- **Intermediate:** Fastest + Color shading, usefull for interpolated colors on polygons.
- **Advanced:** Intermediate + Transparency (Alpha blending + polygon ordering in scene graph). Note: depth buffer is desactivated.
- **Nicest:** Advanced + Anti aliasing on lines and polygons' wireframe.

Quality is also used to define wether the chart should *render continuously* or *on demand*. One can disable the default continuous rendering by calling:

```
quality.setAnimated(false);
```

One can trigger rendering manually using any of the following method which are equivalent:

```
chart.render();
chart.getView().shoot();
chart.getCanvas().forceRepaint();
```

Running charts

Chart initialization let you define its `Quality` and the target windowing toolkit of its canvas. Windowing toolkits may either be `awt`, `swing`, `newt`, or `offscreen`. To setup your desired mode, simply build the chart as follow:

```
Chart chart = AWTChartComponentFactory.chart(quality);
```

As it name suggests, the `SwingChartComponentFactory` delivers charts for `Swing`. At this time, the preferred canvas is `Newt`, and is obtained by initializing a chart as follow:

```
Chart chart = AWTChartComponentFactory.chart(quality, "newt");
```

Other factories deliver charts with dedicated components, such as `ContourChartFactory`, using a custom `ContourAxeBox` able to draw contour level on its ground face. See chapter [Component Injection](#) for more information on factories.

The utility class `ChartLauncher` provides static methods to open a ready to use chart window with configured controllers:

- `openChart(...)`: opens a chart frame with enabled mouse, keyboard and thread controller.
- `openStaticChart(...)`: open a chart frame with no controller.
- `configureControllers(...)` : enables controllers on a chart.
- `screenshot(...)` : save a screenshot to the given file.
- `frame(...)` : create a `Swing` or `AWT` frame according to the chart.
- `openLightEditors(...)` : opens light editor for a chart using lights.

However, you will most probably wish to integrate your chart in an existing application and select appropriate controllers. See dedicated chapters [“Controllers](#) and [Integrate the chart in your application](#).

Destroying charts

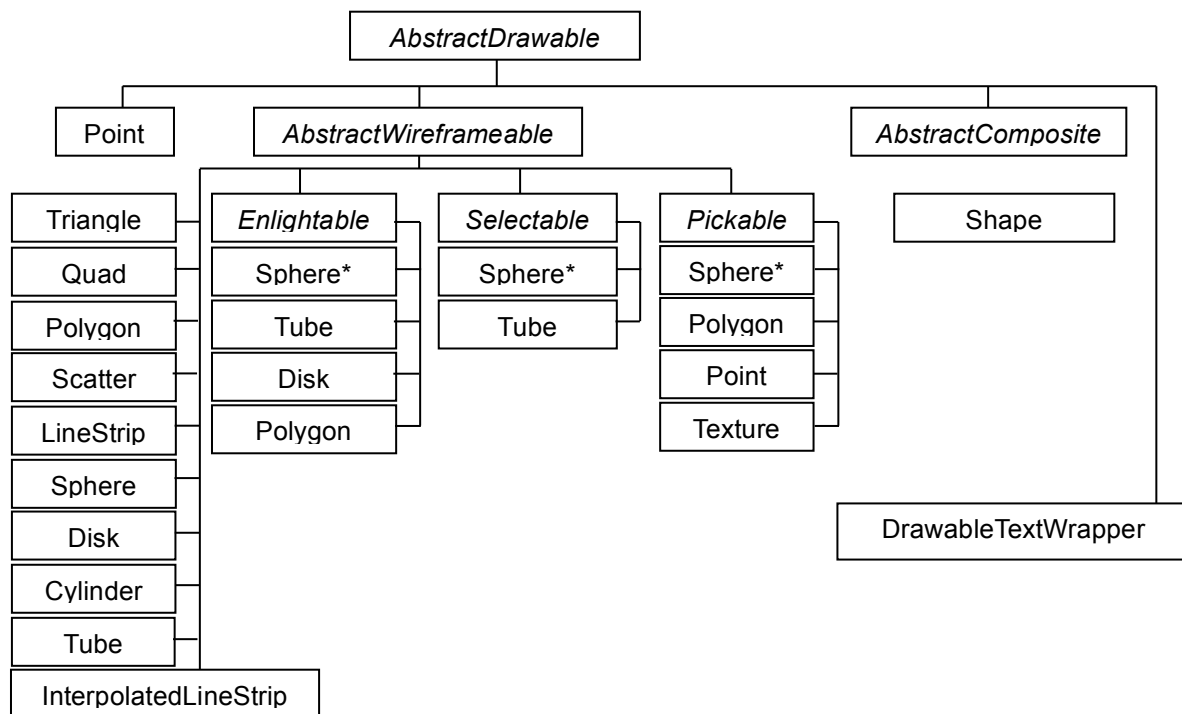
To safely destroy all resources that may be held by a chart, it is recommended to call the `dispose()` method of the chart that will in turn call the `dispose()` method of all its children. This ensures all listeners get unregistered, and all `OpenGL` or other native resource get destroyed.

Next section will discuss how to add basic drawable objects to the chart. If you want to immediately draw surface, scatter, or bar charts, then you may go to chapters [Drawable for charts](#).

Chapter 2 : Drawables

Overview

Drawable objects are organized into a hierarchy that reflects features they offer. The below section explain features provided by *Drawables*, *Wireframeables*, and *Composites* objects.



* Any shape (e.g. Sphere) implementing *Enlightable*, *Selectable*, or *Pickable*, will have as actual class name *EnlightableSphere*, *SelectableSphere*, or *PickableSphere*

Base features

Drawables

AbstractDrawable objects make use of OpenGL primitives. JOGL exposes a GL context allowing to call OpenGL native methods. Thus, a drawable object is expected to implement the method:

```
public void draw(GL gl, GLU glu, Camera camera)
where:
```

- `gl` is the GL context of the canvas in which you render your drawables.
- `glu` is a global GLU instance provided by the View.
- `camera` is a Camera instance provided by the View. It let you define rendering relative to the camera position.

Let's dive into the `draw()` method implementation of a `Point` object:

```
public void draw(GL gl, GLU glu, Camera cam){
    if(transform!=null)
        transform.execute(gl);

    gl.glPointSize(width);
    gl.glBegin(GL.GL_POINTS);
    gl.glColor4f(rgb.r, rgb.g, rgb.b, rgb.a);
    gl.glVertex3f(xyz.x, xyz.y, xyz.z);
    gl.glEnd();
}
```

As shown by the above piece of code, drawing imply first to apply a current transform to the object which is most frequently provided by the View. Then the implementation simply calls the required open GL functions to render the object, according to its properties (position, color, etc).

Objects implementing `AbstractDrawable` have the following properties available:

- Transform
- BoundingBox3d
- Legend
- Is object displayed
- Is legend displayed
- DrawableListener(s)

A drawable object may then be

- singled colored, and implementing `ISingleColorable`
- colored by a `Colormap`, and implementing `IMultiColorable`

The actual geometry of the object must be defined by its implementation. As each object may have its own datamodel, there is no interface method through which you should build your own drawable, but as a convention, Jzy3d uses `setData(...)`

Wireframeables

`AbstractWireframeable` objects have additional properties:

- Wireframe color
- Wireframe width
- Wireframe displayed or not
- Face displayed or not

Among the various wireframeable object, some of them of advanced GL features, especially Polygon that uses:

- A polygon mode: FRONT / BACK / FRONT_AND_BACK (default). When culling is enabled for rendering, FRONT polygons are rendered when they are facing camera only, whereas BACK polygons are rendered when there back is facing to the camera only.
- A polygon offset mode: (ON / OFF). Ability to very slightly change the polygon face and wireframe position at rendering to avoid pitfalls with the Open GL Z-Buffer when the chart is not enabled for alpha.

Composites

`AbstractComposite` are collections of other drawable object. The purpose of this class is to provide an implementation that delegates all above mentioned properties setting to its children. Among the different available composites, `CompilableComposite` is able to compile its GL work into a GL *display list* for faster rendering. It is especially usefull for large surface such as chromatograms (see [demos.surface.big](#)).

Enlightables

`AbstractEnlightable` objects basically hold as properties those provided by OpenGL concerning lights:

- Material emission color
- Material specular reflection color
- Material diffuse reflection color
- Material ambient reflection color
- Material shininess

Selectable

Drawable objects that implement `Selectable` are able to

- `process(...)` and hold the result of their own 2d projection on the screen
- Compute their convex hull

Projection of selectable objects is scheduled by a dedicated `SelectableView`. See chapter “[Interactive objects](#)” for more information.

Pickable

Drawable objects that implement `Pickable` are simply able to store an object ID that is used to identify which objects stand within a given small region around the mouse. See chapter “[Interactive objects](#)” for more information.

Textured

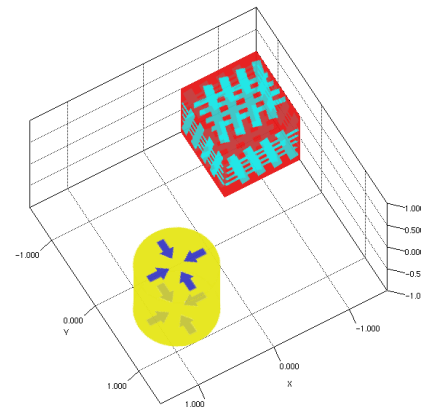
Texture support lets you build drawable objects based on X, Y, or Z planes holding a texture.

One can use a `TextureCube`, and a `TexturedCylinder` (that supports textures on top and bottom but not on the edge, as a cylinder edge is not a plane).

Masks

Masks are PNG images made of white pixels describing the pattern you want to draw, and translucent pixels elsewhere. Using a pair of masks, one can build a face symbol and its negative masks (the one having white pixels where the primary symbol mask has translucent pixels), and apply dynamic coloring on the masks. With that feature, you may edit a catalog of drawable skins, and inject them as `Jzy3d` drawables. Here is how one can setup a cube volume with masks and coloring:

```
SharedTexture t1 = TextureFactory.get(«data/textures/masks/sharp-bg-100.png»);
SharedTexture t2 = TextureFactory.get(«data/textures/masks/sharp-sym-100.png»);
MaskPair mask    = new MaskPair(t1, t2);
TexturedCube cube = new TexturedCube(coord, color, color.negative(), mask);
cube.setAlphaFactor(0.8f);
```



Faster drawables with Vertex Buffer Objects

All above mentioned primitives involve sending lot of geometrical instructions to the GPU each time the chart must be updated, which happens continuously. For simple objects such as medium size surface, it is not a problem at all, but when working with a 1 million polygon objects, the chart rendering is slowing down.

Vertex Buffer Objects are able to store the whole object geometry in the GPU memory at the beginning of the program, and then query transforms or rendering very efficiently without intense communication between CPU and GPU.

`DrawableVBO` is a base implementation allowing to send single colored objects out of `.OBJ` file or `.MAT` files. As reading such file must occur when a GL context is available, one must use a `IGLLoader` (such as `OBJFileLoader` or `MatlabVBOLoader`) that is able to *mount* the object geometry to the GPU once the program initialize the GL contexts.

See VBO in action in org.jzy3d.demos.vbo, and compare rendering speed with org.jzy3d.demos.matlab relying on same files but using non VBO rendering.

Text

Text renderers

The API provides few utility classes to deal with Strings to be rendered at a given 3d coordinate, that will always face the camera whatever the viewpoint.

On top of all text related classes hierarchy stands the `ITextRenderer` interface. It will show you that all text renderers support:

- a text color
- a horizontal alignment (left, right, or center)
- a vertical alignment (top, ground, center, bottom)
- an 2D offset, relative to the screen
- a 3D offset, relative to the scene

Looking at the interface, you may also notice that calling `drawText(...)` will let you retrieve a `BoundingBox3d`, that lets you know the actual space occupied by the text with the current scale and viewpoint.

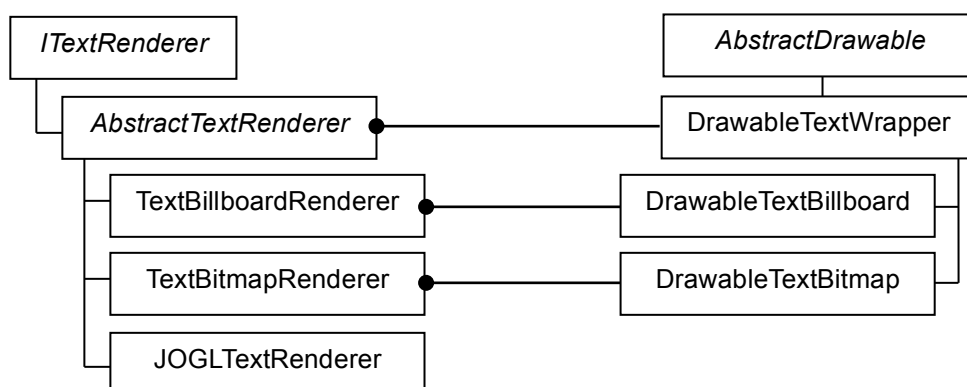
The two first text renderers, `TextBillboardRenderer` and `TextBitmapRenderer` share these common features:

- they do not resize when the scene scale changes.
- they are made of a single non resizable font
- they render ASCII characters (a square will be displayed for non supported characters).

The `JOGLTextRenderer` is different:

- text is resized when the scene scale changes.
- it supports any java Font
- it supports a `ITextStyle`.

Please note that `JOGLTextRenderer` is still in progress at version 0.9.



Drawable text

One may wish to treat texts as standard 3d drawable objects. The `DrawableTextWrapper` utility let you embed a renderer to setup text, alignment, offsets and color through properties.

See the dedicated `DrawableTextBitmap` and `DrawableTextBillboard` that wrap the text renderers mentioned by their name.

Chapter 3 : Drawables for charts

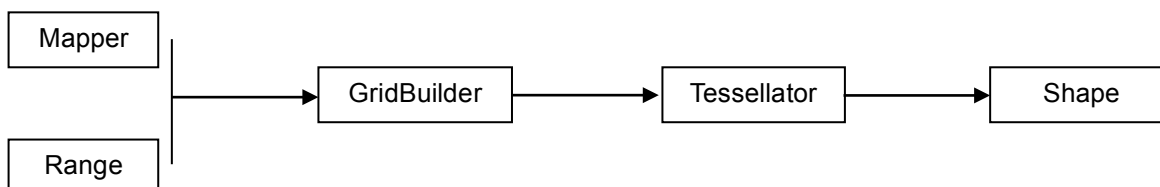
Surfaces

There are several ways to build a surface:

- providing its mathematical form, with an X and Y range.
- providing input points that should be used to build the surface polygons.
- providing input polygons that define the surface.

Surface defined by a function

The simplest way of providing data to plot is to implement a Mapper, which basically represents a mathematical function to be used by a GridBuilder. The mapper generates the input coordinates that allow a Tessellator to build polygons :



This sequence can be customized by any other strategy as soon as one is finally able to build a collection of *Drawable* object.

Mappers

A Mapper is an abstract class that requires its method `f(double x, double y)` to be implemented. The most simple mapper implementation may implement the mathematical function as follow:

```
Mapper mapper = new Mapper(){
    public double f(double x, double y) {
        return Math.sin(x)*Math.cos(y);
    }
};
```

Note that although a Mapper let you use double types for x, y and z, the Coord3d structure used by all drawables holds float values. The reason is that float precision is largely sufficient for the purpose of chart rendering, and uses less memory for large number of Coord3d.

The BufferedImageMapper is a Mapper implementation returning z values computed from the pixel colors of an image, where each {x,y} is a pixel index.

```
public double f(double x, double y) {
    if (x == Double.NaN || y == Double.NaN)
        return Double.NaN;
    int rgb = image.getRGB((int) x, (maxRow) - ((int) y));
    float red = (float) ((rgb >> 16) & 0xFF) / 255.0f;
    float green = (float) ((rgb >> 8) & 0xFF) / 255.0f;
    float blue = (float) ((rgb) & 0xFF) / 255.0f;
    return ((double) ((red * 0.3f) + (green * 0.59f) + (blue * 0.11f)));
}
```

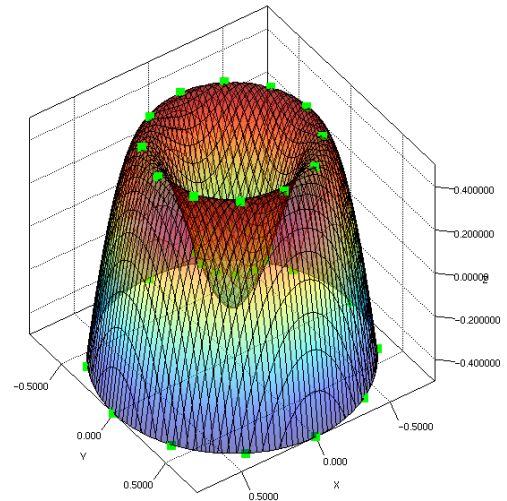
See the [ChromatogramDemo](#) in package [demos.surface.big](#).

Svm3d¹ is an extension allowing to define mappers through a set of training points. In other word, on can do 3d surface regression on a set of input points by using SvmMapper.

The green points on the right side picture form a training set for a SVM regression model. Once this model is trained, it is able to return a Z value for any point in {X,Y} space:

```
public double f(double x, double y) {  
    return svm.apply(x, y)[0];  
}
```

This mapper can be seen as an alternative as using the later mentioned Delaunay tessellator. SvmMapper as the advantage of delivering surface with a user choosen grid with possible high resolution and smooth curves. However it might require spending time on SVM parameter tuning.



Grids

A Grid is able to use Mapper to generate 3d coordinates. Their actual implementation can generate any mesh as soon as their exist a Tessellator able to handle their output. In the example below, we generate points standing on an orthogonal grid, for an identical X and Y range, with 50 grid steps:

```
Range range = new Range(-150, 150);  
int steps = 50;  
OrthonormalGrid grid = new OrthonormalGrid(range, steps, range, steps);
```

Next section explain how to build surfaces from grid-generated or non-grid-generated input points.

¹<https://github.com/jzy3d/jzy3d-svm-mapper>

Surface defined by input points

Tessellation

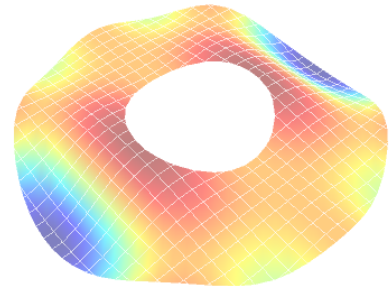
Tessellation is the process of creating polygons out of a set of input coordinates, Jzy3d supports several Tessellator strategies as explained hereafter.

Orthonormal tessellation

- `OrthonormalTessellator` is able to build surface polygons assuming the input data represents nodes of an orthogonal mesh.
- `RingTessellator` is an extension of `OrthonormalTessellator` that allows cutting surface according to a min and max radius.

Tessellating a surface based on the preceding mapper and grid is simply achieved by calling:

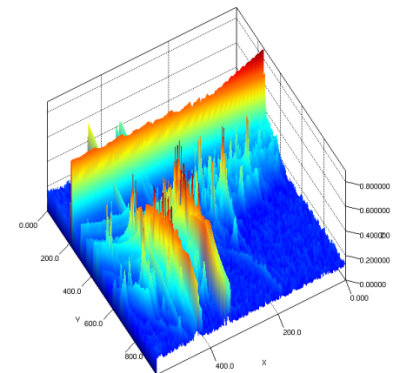
```
Shape surface = Builder.buildOrthonormal(grid, mapper);  
or  
Shape surface = Builder.buildRing(grid, mapper, 20f, 50f);
```



One may also build surfaces dedicated to large datasets. These are built using `CompileableComposite`, a drawable object able to compile itself as a GL *display list*, and then make use of this prebuilt GL code for all following rendering:

```
Shape surface = Builder.buildOrthonormalBig(grid, mapper);
```

See the [demos.surface.big](#) package.



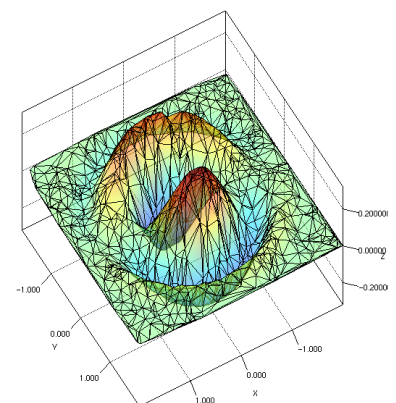
Delaunay: a constraint free surface tessellator

One may require to build a surface using a random set of points. For this use case, a Delaunay tessellator has been implemented using JDT² to compute the tessellation of an unordered set of points.

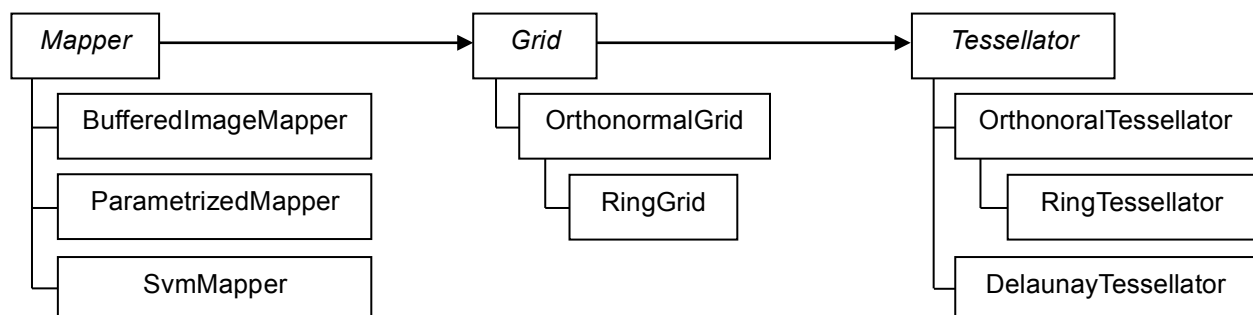
Delaunay algorithm, which details won't be covered here, is able to compute a good triangulation for points given as follow:

```
Shape surface = Builder.buildDeLaunay(coords);
```

See the [demos.surface.delaunay](#) package.



The below schema summarize all previously mentioned chainable components :



²<http://code.google.com/p/jdt/>

Surface defined by polygons

You can build any shape by defining raw polygons by yourself as shown in the bellow code:

```
double [][]mesh = new double[][] {{.25, .45, .20},
                                   {.56, .89, .45},
                                   {.6 , .3 , .7 }};

List<Polygon> polygons = new ArrayList<Polygon>();
for(int i = 0; i < mesh.length -1; i++){
    for(int j = 0; j < mesh[i].length -1; j++){
        Polygon polygon = new Polygon();
        polygon.add(new Point(new Coord3d(i, j, mesh[i][j])));
        polygon.add(new Point(new Coord3d(i, j+1, mesh[i][j+1])));
        polygon.add(new Point(new Coord3d(i+1, j+1, mesh[i+1][j+1])));
        polygon.add(new Point(new Coord3d(i+1, j, mesh[i+1][j])));
        polygons.add(polygon);
    }
}
Shape surface = new Shape(polygons);
```

It is important to have **only one Shape** referencing a given list of polygons. Indeed, each polygon point holds its own color data, so when a Shape is modified by a color tool, each polygon's point has its color field updated.

See [BuildSurfaceDemo](#)

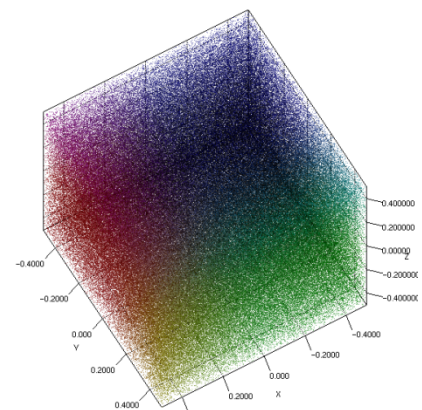
Scatters

Scatters are simple drawables coming in three flavors:

- Scatter: a scatter plot made of a single color, using Coord3d[] and Color[] as data model
- MultiColorScatter: a scatter plot painted by a Colormap, using Coord3d[] as data model
- MultiColorScatterList: a scatter plot painted by a Colormap, using List<Coord3d> as data model

The below code let you create a RGB scatter:

```
Coord3d[] points = new Coord3d[size];
Color[] colors = new Color[size];
for(int i=0; i<size; i++){
    x = (float) Math.random() - 0.5f;
    y = (float) Math.random() - 0.5f;
    z = (float) Math.random() - 0.5f;
    a = 0.25f;
    points[i] = new Coord3d(x, y, z);
    colors[i] = new Color(x, y, z, a);
}
Scatter scatter = new Scatter(points, colors);
```



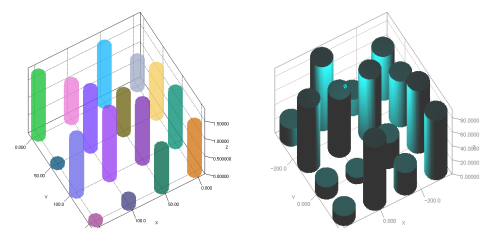
See [demos.scatter](#)

Histograms & bar charts

Bar charts can be build easily using HistogramBar, which is basically a Composite made of one Tube and two closing Disk. The below code shows how to create drawable bars:

```
Color color = Color.random();
HistogramBar bar = new HistogramBar();
bar.setData(new Coord3d(x, y, 0), height, 10, color);
bar.setWireframeDisplayed(false);
```

See [demos.simplebarchart](#)



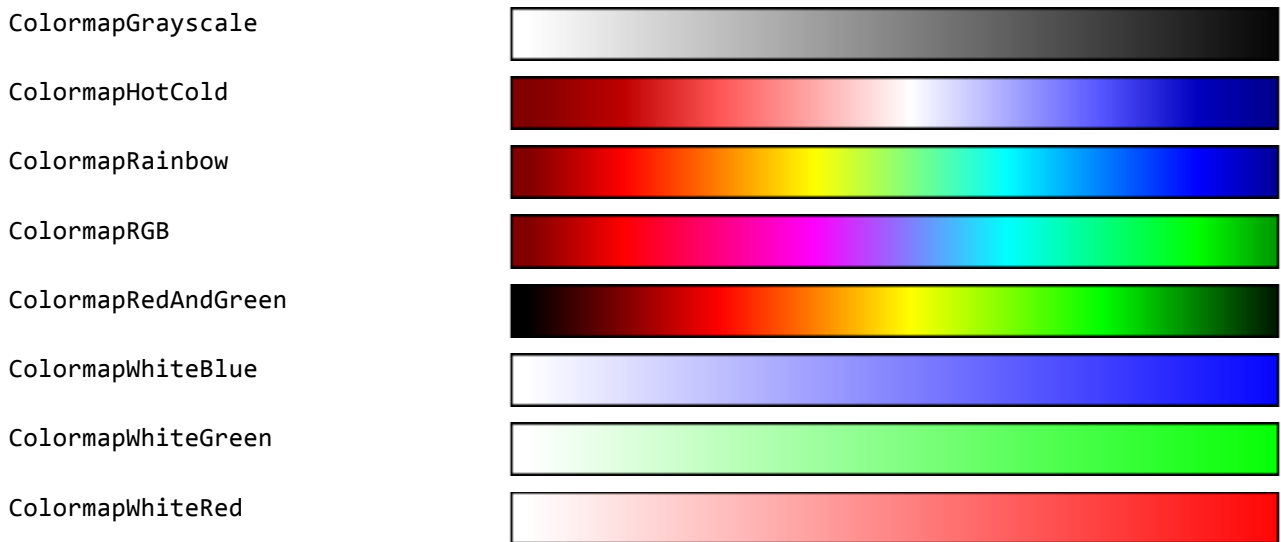
Chapter 4 : Colormaps

Overview

The API provides a set of Colormaps that can be used to paint objects that implement `IMultiColorable` (i.e. almost all primitives). There are a few objects with different roles:

- Colormaps are functions that generate a RGBA color for a given XYZ point.
- A `ColorMapper` applies a Colormap to an object with a given Z range.
- A `ColorBarLegend` draws the Colormap, next to the chart, with tick annotations.

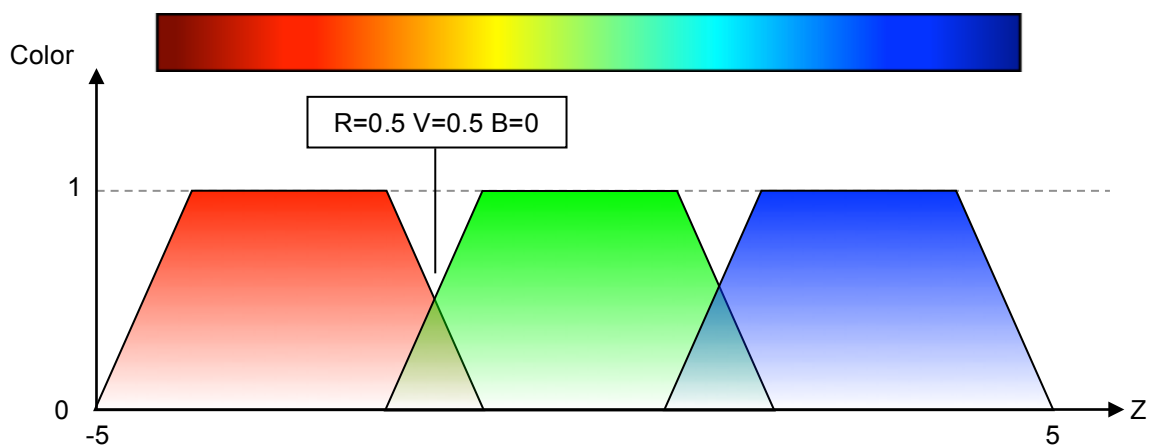
The framework already provides the following colormaps:



A colormap basically implement:

```
public Color getColor(IColorMappable colorable, float x, float y, float z)
```

The below schema shows how a colormap computes a color based on the three RGB components: each component exists in a given Z range and is added to all other components to produce a color.



Note that a colormap direction can be toggled by calling:

```
colormap.setDirection(false)
```

Apply a colormap to a drawable

A ColorMapper let you apply a Colormap to a MultiColorable object. It needs to know the Z range to be used in order to span the Colormap colors. Any point or polygon standing out of that range will have the «lowest color» under the minimum Z value, and «highest color» above the maximum Z value.

```
surface.setColorMapper(new ColorMapper(new ColorMapRainbow(), -5, 5));
```

A ColorMapper also supports a *color factor* enabling to multiply all colormap colors with a color. It is especially usefull to apply an alpha effect as follow:

```
ColorMapper mapper = new ColorMapper(new ColorMapRainbow(), -5, 5, new Color(1,1,1,.5f))
```

Create a colorbar legend

When painting an object with a Colormap and ColorMapper, you will most probably wish to have this colormap represented by a colorbar in a legend. This is the purpose of the ColorBarLegend object that keeps track of the object scale changes, and that remains cleanly layed out by the ChartView. To add a legend, just attach a colorbar to your drawable object as follow:

```
IBoxLayout layout = chart.getView().getAxe().getLayout();
ColorbarLegend colorbar = new ColorbarLegend(surface, layout);
colorbar.setMinimumSize(new Dimension(100, 600));
surface.setLegend(colorbar);
```

In its implementation, a ColorBarLegend uses a ColorbarImageGenerator that will create a Java2d BufferedImage according to current drawable's ColorMapper.

Additionally, the Colorbar can be given an ITickProvider and ITickRenderer (usually those already defined to configure the AxeBox) to generate a BufferedImage with tick annotation next to the colorbar.

```
ColorbarLegend(AbstractDrawable parent,
                ITickProvider provider, ITickRenderer renderer,
                Color foreground, Color background)
```

The roles of ITickProvider and ITickRenderer are given in the “[Axis layout](#)” chapter. Additional information on the ChartView layout are given in the “[View layout](#)” chapter.

Chapter 5 : Layout

The chart layout is controlled at three levels:

- the view layout, that lets you set various stretching policies, background settings, tooltips and post renderers for the chart object.
- the canvas layout, that lets you arrange several 3d and 2d views.
- the axe layout, that lets you define tick values and labels for each axis.

View Layout

Jzy3d provides controls that let you improve the rendering of the 3d scene in its parent canvas. For all following settings, you can activate the display of a debug grid to understand how the viewports (GL scene or GL image, e.g. ColorbarLegend) are actually displayed by OpenGL. Simply call:

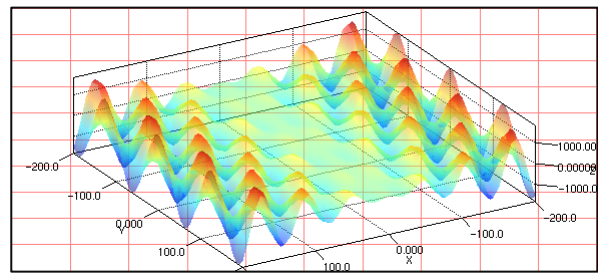
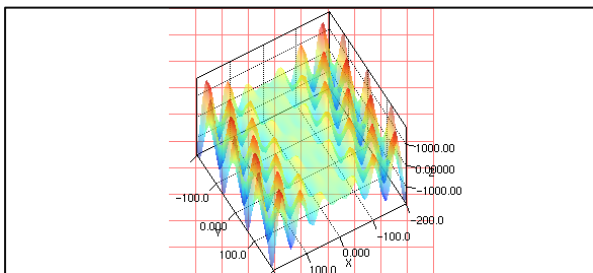
```
chart.getView().getCamera().setScreenGridDisplayed(true);
```

Maximize

The default GL scene window is a square that will try to occupy be the largest possible space. If the target panel is not a square, the scene will remain centered on the widest dimension.

To let the scene stretch over the complete panel as in the second chart, simply call:

```
chart.getView().setMaximized(true);
```

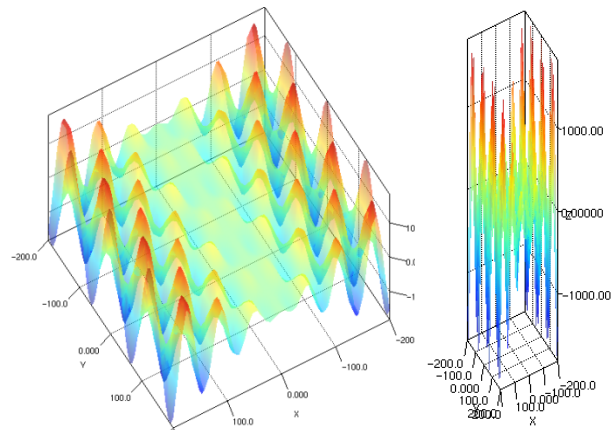


Squarify

Squarifying is an automatic scaling that lets a complete scene fit into a cube. This is the desired behavior in most cases.

One may wish to keep scales unchanged, as the picture standing on the right. To do so, just deactivate this setting through:

```
chart.getView().setSquared(false);
```



Text embedding

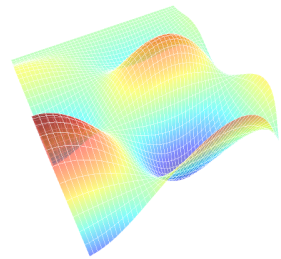
It is possible to force Jzy3d to always stretch the 3d scene so that all axes tick labels always stand IN the canvas. The feature is mainly useful for enhancing a 2d view, but rotating a 3d scene in this mode is uncomfortable since the scene scale may change in an undesired way. For working with that feature, edit:

```
View.MAINTAIN_ALL_OBJECTS_IN_VIEW = true;
```


Orthogonal and perspective projections

One can specify a projection mode, either orthogonal or perspective. Although most of the scientific plots use an orthogonal projection (default), plotting with a perspective lets you give more importance to what is in front of the viewpoint. To enable a specific projection:

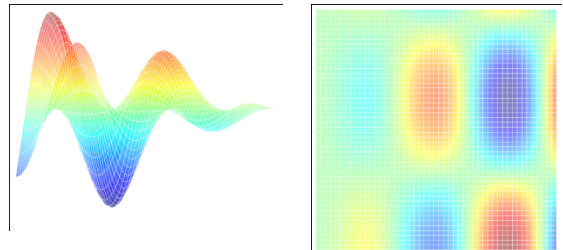
```
chart.getView().setCameraMode(CameraMode.{ORTHOGONAL  
|PROJECTION});
```



View constraints

The view can be constrained to only support specific viewpoint change. One may for example use the `ViewPositionMode.PROFILE` to allow rotation only around the Z axis when using the mouse, and keep X and Y axis rotation to 0.

`ViewPositionMode.TOP` is a mean to display 2d charts. In the top view, the camera's view direction stands on the Z axis at X,Y=0.



The default constraint is `FREE` to enable any viewpoint around the center of the scene. Changing the constraint mode is achieved by:

```
chart.getView().setViewPositionMode(ViewPositionMode.{FREE|TOP|PROFILE});
```

Viewpoint

In Jzy3d, the camera is always looking at the center of the scene, and rotate around this central point. The viewpoint, i.e. the position of the Camera is defined by a polar coordinate relative to the center of the scene.

- `x` is the horizontal angle in $[0;2\pi]$ (`x` can have any value, the view considers $x = x \% 2\pi$)
- `y` is the vertical angle in $[-\pi/2;\pi/2]$ (the view consider $y = -\pi/2$ if $y < -\pi/2$, $y = \pi/2$ if $y > \pi/2$)
- `z` is the distance from camera to center. It is adjusted by the View to let the scene fit in the display.

```
chart.getView().setViewpoint(new Coord3d(...));
```

Post renderers

Adding other kind of metadata on the main view can be done easily using the post renderers, which provide a `java2d Graphics` for drawing text, images, and 2d primitives, relative to the panel size.

As shown by the [Overlay demo](#), one accesses a `Graphics2d` this way:

```
chart.addRenderer(new Renderer2d() {  
    public void paint(Graphics g){  
        Graphics2D g2d = (Graphics2D) g;  
        g2d.setStroke(new BasicStroke(4.0f));  
        g2d.setColor(java.awt.Color.BLACK);  
        g2d.drawRect(10, 50, 100, 100);  
    }  
});
```

Background

It is possible to edit the view background by providing a background image:

```
BufferedImage i= FileImage.load(«data/bg-demo.jpg»);  
chart.getView().setBackgroundImage(i);
```

Background is only resized when it is too large to stand in the canvas, but not when the canvas is larger.

The background color can be changed as well:

```
chart.getView().setBackground-color(Color.BLACK);
```

Tooltips

You can accumulate tooltips on the view:

```
IntegerCoord2d screen = new IntegerCoord2d(10,10);
Coord3d world = new Coord3d(20,20,20);
CoordinateTooltipRenderer r = new CoordinateTooltipRenderer(«X»,«Y»,«Z»,
                                                             screen, world, true);

chart.getView().addTooltip(r);
```

and clear them:

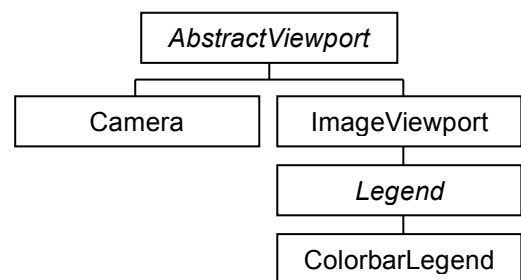
```
chart.getView().clearTooltips();
```

Chart layout

All the settings we have covered above are related to the root View object, which only handles the layout of a scene content. A Chart uses the extended class ChartView that is able to arrange several visual block in the same canvas.

Classes that implement AbstractViewport provide method to define where an object should be mapped in the current canvas. Let's mentioned:

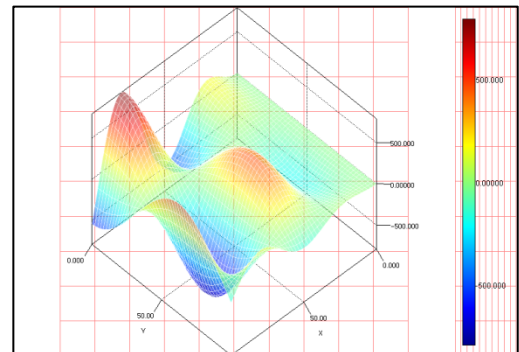
- A View's Camera is the object that let you define how to display the chart's scene in the canvas.
- A ColorbarLegend, relies on an ImageViewport, which is able to map a Java 2D BufferedImage on the canvas rendered by OpenGL.



As shown on the right, the ChartView simply computes horizontal slices for each AbstractViewport. One can debug each viewport by enabling:

```
viewport.setScreenGridDisplayed(true);
```

To inject a custom chart layout, see the chapter "[Component injection](#)".



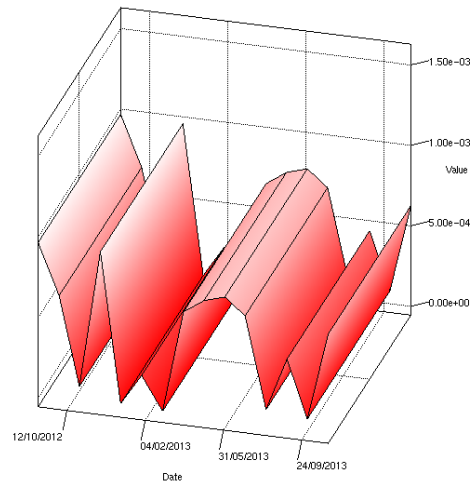
Axis Layout

The View comes with a box embedding the scene with value annotations. The box can be turned on/off by using:

```
chart.getView().setAxeBoxDisplayed(false);
```

One can access the `IAxeLayout` that gathers, for each X, Y, Z axis:

- all labeling policy settings (axis label and tick label properties) defined by `ITickRenderers`.
- all ticks to render defined by `ITickProvider`.
- grid and face colors.



A concrete `ITickProvider` must implement `generateTicks(float min, float max, int steps)` that returns an array of values indicating selected ticks for the axe range currently. Here are the already existing tick providers:

- `RegularTickProvider` generates a fixed number of ticks in any range.
- `SmartTickProvider` generates a readable number of ticks (between 3 to 5).
- `StaticTickProvider` holds a list of tick to display, whatever is the actual view range.

A concrete `ITickRenderer` must implement `format(float value)`. Here are the already existing tick renderers:

- `DateTickRenderer` renders a tick value as a date, according to a desired date format.
- `ScientificNotationTickRenderer` renders a tick value like "1.5e-03".
- `FixedDecimalTickRenderer` renders a tick value with a predefined precision.

One may for example create a `TrigonometricTickRenderer` with a joint `TrigonometricTickProvider` to draw ticks such as 0, $\pi/2$, $\pi/3$...

See the [demos.axelayout](#) package to see how to edit the whole `AxeBox` layout:

```
chart.getAxeLayout().setXAxeLabelDisplayed(false);  
chart.getAxeLayout().setXTickLabelDisplayed(false);  
chart.getAxeLayout().setYAxeLabel( «Date» );  
chart.getAxeLayout().setYTickRenderer( new DateRenderer( «dd/MM/yyyy» ) );  
chart.getAxeLayout().setZAxeLabel( «Ratio» );  
chart.getAxeLayout().setZTickRenderer( new ScientificNotationRenderer(2) );
```

Please note that you can set a global color for the entire axe by calling:

```
chart.getView().getAxe().getLayout().setMainColor(Color.GRAY);
```

Chapter 6 : Contour charts

Overview

The central tool to use is an `IContourPictureGenerator` that can compute various kinds of contour pictures:

- contour lines
- filled contour
- height maps

The `MapperContourPictureGenerator` is its primary implementation and works for surface defined by a `Mapper`. Thus, no contour function can be computed and drawn for surface built manually, or from a list of input points (e.g. Delaunay tessellated surfaces).

The contour can be computed according to:

- an array of level values
- a number of levels to draw

Once the contour is computed, its image is built and stored in a textured plane, in order to be added to a special axe box, `ContourAxeBox`, able to draw an image on the ground.

To paint contour lines or areas, one must use a `IContourColoringPolicy`, that returns an RGB color code according to a Z value. The provided implementation, `DefaultContourColoringPolicy`, supports a `ColorMapper` as input in order to be able to draw the contour function of a `IMultiColorable` object.

Configuration

Here is a sample code able to generate a contour chart:

```
JzyFactories.axe = new AxeFactory(){
    public IAxe getInstance(BoundingBox3d box, View view) {
        ContourAxeBox axe = new ContourAxeBox(box);
        axe.setView(view);
        return axe;
    }
};

IContourGenerator contour = new MapperContourGenerator(mapper, xrange, yrange);
DefaultContourColoringPolicy policy = new
    DefaultContourColoringPolicy(myColorMapper);

Chart chart = new Chart(quality, type);
ContourAxeBox cab = (ContourAxeBox)chart.getView().getAxe();
cab.setContourImg(contour.getContourImage(policy, 400, 400, 10),xrange,yrange);
```

Where

- 400 and 400 are the number of pixels of the output image.
- xrange and yrange the ranges used for the mapper.
- 10 the number of required levels.

It is also possible to specify a list of desired levels as follow:

```
double sortedContourLevels[]={-500.0,-200.0,0.0, 100.0, 300.0, 400.0};
BufferedImage img = contour.getContourImage(policy, 400, 400, sortedContourLevels);
```

Note that you can display the contour image in a separate window using:

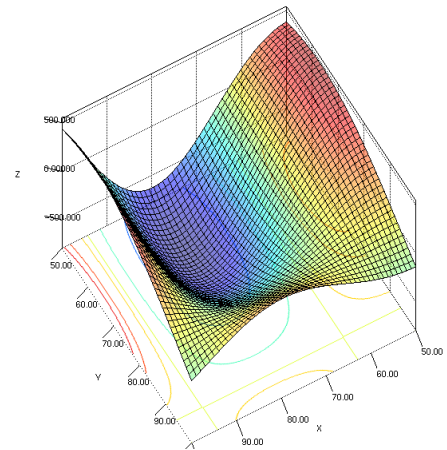
```
BufferedImage image=((ContourAxeBox)chart.getView().getAxe()).getContourImage();
ChartLauncher.openImagePanel( image, new Rectangle(800,600,400,400) );
```

Contour lines

Method `getContourImage(...)` returns a default contour image drawing contour lines.

```
cab.setContourImg(  
    contour.getContourImage(  
        new DefaultContourColoringPolicy(myColorMapper),  
        400, 400, 10),  
    xrange, yrange  
);
```

See: org.jzy3d.demos.contour.ContourPlotsDemo

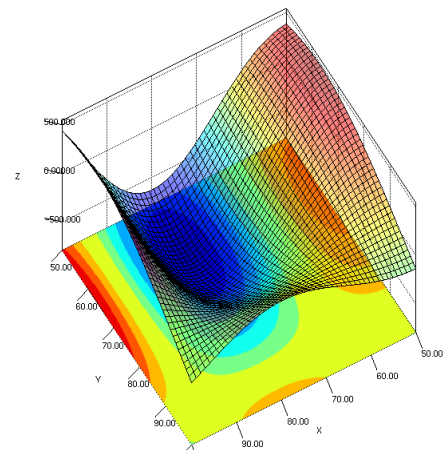


Filled Contour

Method `getFilledContourImage(...)` returns a contour region image.

```
cab.setContourImg(  
    contour.getFilledContourImage(  
        new DefaultContourColoringPolicy(myColorMapper),  
        400, 400, 10),  
    xrange, yrange  
);
```

See: org.jzy3d.demos.contour.FilledContourDemo

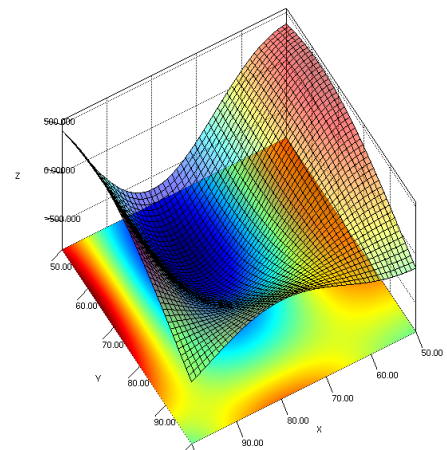


Height Map Contour

Method `getHeightMap(...)` applies the Z values returned by a mapper able to generate heatmap-like graphics.

```
cab.setContourImg(  
    contour.getHeightMap(  
        new DefaultContourColoringPolicy(myColorMapper),  
        400, 400, 10),  
    xrange, yrange  
);
```

See: org.jzy3d.demos.contour.HeightMapDemo



Chapter 7 : Controllers

Mouse controllers

Controlling camera

The CameraMouseController let you control the Camera eye position as follow:

- Rotate: Left click and drag mouse.
- Scale: Roll mouse wheel.
- Z Shift: Right click and drag mouse.
- Animate: Double click will start the thread controller that rotates the view.

To add a mouse controller, simply call:

```
chart.addMouseController();
```

You can still concurrently edit the viewpoint by yourself by calling:

```
chart.getView().setViewpoint(new Coord3d(...));
```

See the [Layout](#) chapter for more information about viewpoint value range.

Interaction with drawables

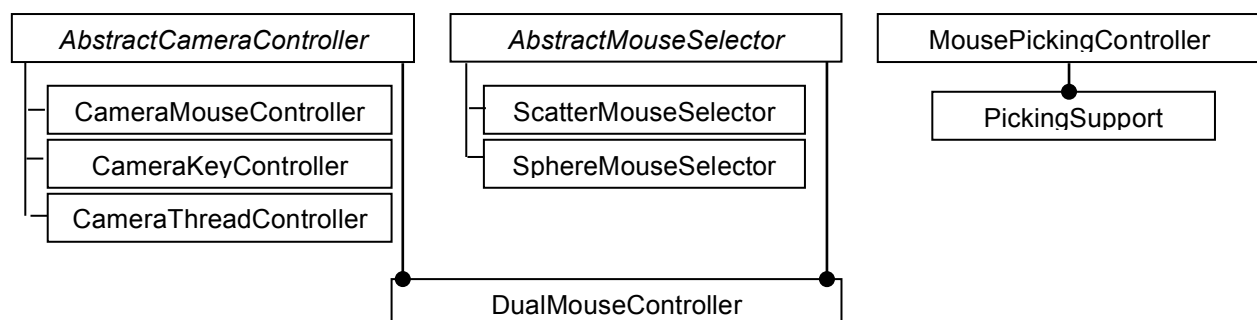
One may also interact with drawables (for example making a selection in a scatter), by using dedicated drawable *selectors*. There might be one mouse selector implementation per kind of Selectable object. This kind of mouse controllers have the following role:

- Processing selection according to a selection window.
- Drawing a selection rectangle.

Once a selector is built, one can still control the camera with mouse by using a DualModeMouseSelector that let you toggle between selection and rotation behavior by holding the C key:

```
ScatterMouseSelector selector = new ScatterMouseSelector(scatter);  
DualModeMouseSelector mouse = new DualModeMouseSelector(chart, selector);
```

One may also use a MousePickingController that let you enable a different strategy for processing mouse selection. See the chapter “[Interactive objects](#)” for more details.



Threaded controllers

You can add default threaded controllers that enable a continuous cube spin, and that are started or stopped by a mouse double-click. You can also make your own thread controller as follow:

```
CameraThreadController thread = new CameraThreadController();  
chart.addController(thread);
```

Keyboard controllers

Similar to the mouse controller, you can enable a keyboard controller to perform the following view changes:

- Rotate: Arrows
- Scale: Shift + arrows left & right
- Z Shift: Shift + arrows up & down

To add a keyboard controller, simply call:

```
chart.addKeyController();
```

Note that the key controllers can only receive key events if the canvas is focused. You need to make a first click on the chart, or to programmatically force the focus to your chart panel to have the key controller enabled.

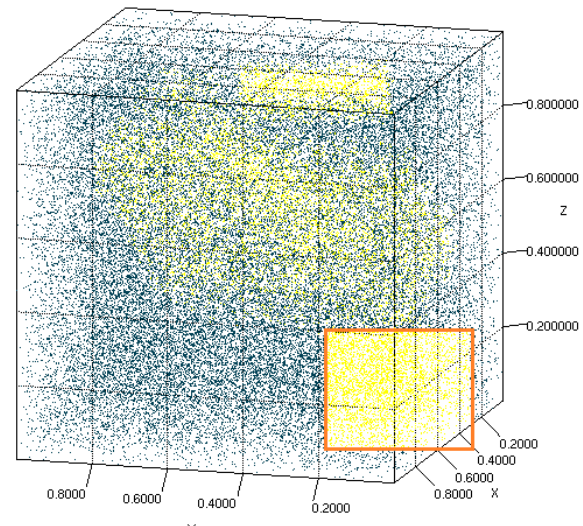
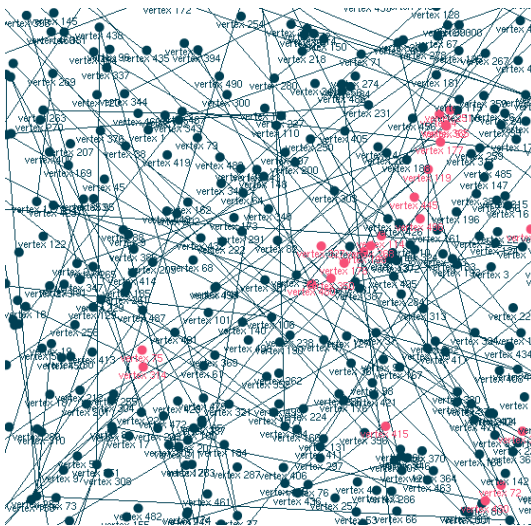
If you wish to implement your own controller see the [Component Injection](#).

Chapter 8 : Interactive objects

Overview

Interaction with 3d content can be achieved using two methods:

- The first consists in rendering a very small portion of the scene around the mouse pointer, and analysing which objects stand in that tiny window. This method is efficient for simple mouse crossing. We call objects supporting this method **Pickable**.
- The second consists in retrieving the 2d projection of the complete object and analyzing what the mouse should do with it. This method is suitable for global scene analysis and allows to deal with a rectangle selection. For example, selecting a rectangle region of a 3d scatter plot requires to know the complete projection in advance, and to decide which dots stand in or out of the mouse selection rectangle as soon as this rectangle dimension changes. We call objects supporting this method **Selectable**.



Picked objects

Picking requires to register every pickable polygon into a `PickingSupport` instance. Indeed, once the mouse is clicked, `PickingSupport` retrieves a collection of polygon IDs that can be found below the mouse pointer. `PickingSupport` defines a pixel area under which content is detected (default is a 10 x 10 area). To illustrate how to enable picking, we will discuss elements you may find in the demo [demos.graphs.PickableGraphDemo](#).

First, we deal with a set of `Pickable` objects: our graph is made of `PickablePoints` (`PickablePoint` is simply an extension of `Point` that implements `Pickable`, in other words it support `getId/setId`).

Second, we use a dedicated `MousePickingController` and override method `mousePressed()` to apply picking with the following lines of code (simplified):

```
public void mousePressed(MouseEvent e) {
    pick(e);
}
public void pick(MouseEvent e) {
    picking.pickObjects(gl, glu, view, graph, new IntegerCoord2d(e.getX(), yflipped));
}
```

To get notified by a successful picking, we then register a listener:

```
mouse.getPickingSupport().addObjectPickedListener(new IObjectPickedListener() {
    public void objectPicked(List<? extends Object> vertices, PickingSupport picking) {
        for(Object vertex: vertices)
            graph.setVertexHighlighted((String)vertex, true);
        chart.render();
    }
});
```

Last we have to register our PickablePoints to the PickingSupport instance:

```
public void setGraphModel(IGraph<V,E> graph, PickingSupport picking){
    for(V v: graph.getVertices()){
        PickablePoint p = newPoint(v);
        picking.registerDrawableObject(p, v);
    }
}
```

Selectable objects

Selectable objects let you easily get 2d projections of any geometry. You can then work the way you want with that projection. You may for example project a complete scatter plot, or only anchors of a shape model such as surfaces.

Such objects must implement:

```
public void project(GL gl, GLU glu, Camera cam)
```

The implementation will mainly consist in calling and caching the 2d projection. Let's see an example with SelectableScatter:

```
public void project(GLs gl, GLU glu, Camera cam) {
    projection = cam.modelToScreen(gl, glu, getData());
}
```

The scatter must have an associated mouse controller such as ScatterMouseSelector in charge of:

- calling a projection of all the instances of Selectable objects appearing in the scene graph
- retrieving the projected scatter, and verify how it matches the current selection rectangle
- changing highlighted status of each point to have them rendered differently.
- drawing the current selection rectangle

This is simply implemented as follow:

```
protected void processSelection(Scene scene, View view, int width, int height) {
    view.project();
    Coord3d[] projection = scatter.getProjection();
    for (int i = 0; i < projection.length; i++)
        if (matchRectangleSelection(in, out, projection[i], width, height))
            scatter.setHighlighted(i, true);
}

protected void drawSelection(Graphics2D g2d, int width, int height) {
    ...
    drawRectangle(g2d, in, out); // a simple rectangle
}
```

In the same fashion, a ScatterSphere has an associated SphereMouseSelector, that provides a selection renderer able to draw the convex hull of the selected sphere.

Keep in mind you remain responsible of calling project() once required. Having an ever up to date projection available for example requires customizing the view to perform such projection as soon as the viewpoint changes or the canvas dimensions change.

Projections must be used with care and optimally scheduled when working with large data. SelectableView is an extension of View able to update all projections at each any call to render(...). When using it take care to the chart rendering model (continuously/on demand, see section [Chart Quality](#)).

Chapter 9 : Scene graph

Jzy3d has a very minimal scene graph, mainly intended to:

- deal with polygon ordering for translucent objects (see chapter [Transparency](#))
- provide global scaling of drawable objects

You will most probably never have to deal with such details, unless you wish to apply custom transforms or polygon ordering methods.

Transforms

Transformations for Drawables

Every drawable object in jzy3d is globally transformed by calling `setTransform(Transform)`, that is commonly used by the View to let a scale and rotate a complete Scene.

Another transformation can be applied specifically to a drawable via `setTransformBefore(Transform)`,

Each call to the `draw(...)` method of a Drawable begins by executing the held transforms. Executing transformations:

- always start by reloading the identity matrix (thus ensuring any object can be freely transformed however you transform the other objects of the scene graph).
- then execute the sequence of Transformer, either a Scale, a Translate, or a Rotate.
 - Self transforms first
 - Global transforms

Animating a drawable transformation

The code below shows how to define a rotation axis to a specific shape, and how to start a rotation applying only to the object and not to the complete scene.

```
// Define rotation around Z axis
Rotate r = new Rotate(25, new Coord3d(0, 0, 1));
Transform t = new Transform();
t.add(r);
shape.setTransformBefore(t);
```

```
// Let the wheel rotate
Rotator rotator = new Rotator(10, r);
rotator.start();
```

Note that the bounding box of a drawable does not consider transforms. It only returns information about input geometry.

Chapter 10 : Animations

Remapping surfaces

Surfaces built with a mapper can be easily remapped, i.e. that having their Z coordinates updated due to a change in the $z=f(x,y)$ function defined by the Mapper.

The code below shows a `SingleParameterMapper` that evolves with `IncreaseParamRemapTask` :

```
SingleParameterMapper mapper;  
Shape surface;  
RemapTask remap;  
Thread thread;  
  
public void init(){  
    mapper = new SingleParameterMapper(0.999){  
        public double f(double x, double y) {  
            return 10*Math.sin(x*p)*Math.cos(y*p)*x;  
        }  
    };  
    surface = ...  
    remap = new IncreaseParamRemapTask(surface, mapper);  
    chart = AWTChartComponentFactory.chart(getCanvasType());  
    chart.getScene().getGraph().add(surface);  
    thread = new Thread(remap);  
    thread.start();  
}
```

Where the implementation of the remap task is defined with :

```
public class IncreaseParamRemapTask extends AbstractRemapTask {  
    ...  
    public void remap() {  
        mapper.setParam(mapper.getParam() + 0.0001);  
        mapper.remap(surface);  
    }  
}
```

And its abstract class defines

```
public void run() {  
    while (true) {  
        try {  
            Thread.sleep(1);  
        } catch (InterruptedException e) {}  
        time.tic();  
        remap();  
        time.toc();  
  
        info = Utils.num2str(time.elapsedSecond(), 4) + "s to remap surface";  
    }  
}
```

See: org.jzy3d.demos.animation.AnimatedSurfaceDemo

Adding and removing drawables dynamically

One can add and remove elements from the scenegraph by using its `add(...)` and `remove(...)` methods. These two methods can also be called with a boolean flag that indicates if the view bounds should be updated immediately or not.

Not updating the view immediately can be useful if you need to add a lot of new objects. It is indeed better to add them all, and to update the bounds a single time once all new objects are in the scenegraph.

```
chart.getScene().getGraph().add(line1);  
chart.getScene().getGraph().add(line2);  
chart.getScene().getGraph().add(line3);  
chart.getView().updateBounds();
```

or

```
chart.getScene().getGraph().add(line, true);
```

See: org.jzy3d.demos.animation.AddRemoveElementsDemo

Chapter 11: Transparency

Handling transparency in a chart deserve a few explanations. Indeed, transparency is not natively handled by OpenGL and is performed by various algorithms with various results.

Handling transparency by sorting polygons

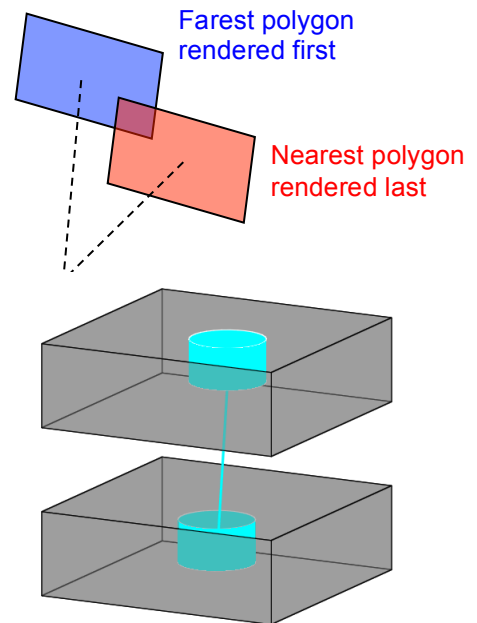
In this approach, blending is achieved by first rendering « what stands far », and then « what stands near ».

For that reason, the scene graph needs to know the camera position, and then rank all polygons to know their order relative to the eye. Guessing the appropriate order is a complicated task, and Jzy3d only computes the distance between the camera eye, and each polygon *barycenter*.

AbstractComposite objects, such as Shapes, are good primitives for such kind of rendering, since they can be decomposed into several atomic AbstractDrawable by the scene graph.

However, this method has two drawbacks:

- First, working with numerous polygons will lead to bad rendering performances, as the polygons' decomposition and ranking will be computed at each frame update.
- Second, the method is not suitable when working with large polygons, or with drawables of very different dimensions (see an example of visual cue on the right).

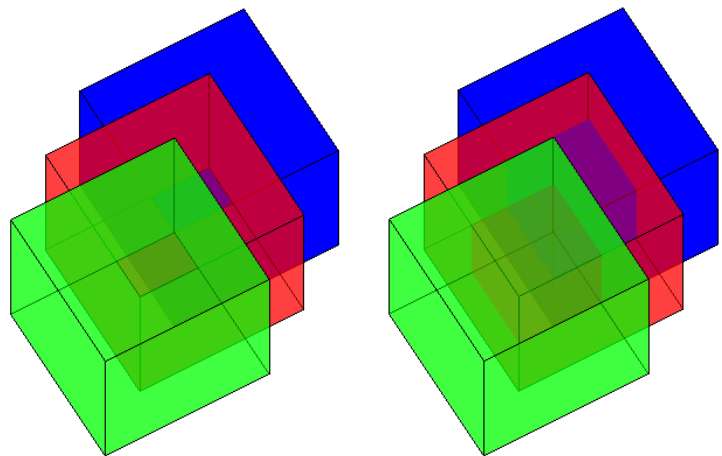


Handling transparency with a depth peeling algorithm

Depth peeling algorithms allow an order independent transparency of intersected translucent objects. The left picture shows translucent cubes that are rendered by ordering their distinct faces. The right picture shows the same cubes rendered by a dual depth peeling algorithm.

Our implementation of depth peeling is based on works made public by L. Bavoil (NVIDIA) proposing several variants:

- Dual depth peeling.
- Front to back peeling.
- Weighted average peeling.
- Weighted sum peeling.



Peeling is available as an extension³ of the API.

Hints with transparency

Remember that an object holding colors with $\alpha \neq 1$ will not appear translucent if you do not instantiate a Chart with a `Quality >= Advanced`.

When using a view configured with `CameraMode.PERSPECTIVE` in a translucent chart (i.e. initialized with `Quality.Advanced` or higher), it is required to disable a polygon property named "Polygon Offset Fill", otherwise the shape does not render properly. To do so, simply call this static method:

```
Polygon.setPolygonOffsetFillEnable(surface, false);
```

³ <https://github.com/jzy3d/jzy3d-depthpeeling>

Chapter 12 : Events

The API triggers its own events letting you know how the general settings of the chart changes.

View events

View point changed

One may register a `IViewPointListener` to get notified by a `ViewPointChangedEvent` once the viewpoint changes:

```
chart.getView().addViewPointChangeListener(new IViewPointChangeListener() {  
    public void viewPointChanged(ViewPointChangedEvent e) {  
        System.out.println("viewpoint changed to « + e.getViewPoint());  
    }  
});
```

View point reached top or bottom

To know that the user moved the chart to a top or bottom view. To get such notification:

```
chart.getView().addViewOnTopEventListener(new IViewIsVerticalEventListener() {  
    public void viewVerticalReached(ViewIsVerticalEvent e) {  
        System.out.println("view from top or bottom»);  
    }  
    public void viewVerticalLeft(ViewIsVerticalEvent e) {  
        System.out.println("left top or bottom»);  
    }  
});
```

Controller events

A mouse controller is able to notify its listeners that it performed a view change. Indeed, adding a `ControllerEventListener` let you receive events with informations on the kind of changed value, and the new value:

```
mouse.addControllerEventListener(new ControllerEventListener() {  
    public void controllerEventFired(ControllerEvent e) {  
        System.out.println(e.getType + « « + e.getValue());  
    }  
});
```

Where type is a `ControllerType` that may either be `ROTATE`, `ZOOM`, `SCALE`, or `PAN`.

Drawable events

One may receive notifications concerning a drawable object property change.

```
scatter.addDrawableListener(new IDrawableListener(){  
    public void drawableChanged(DrawableChangedEvent e) {  
        switch(e.what()){  
            case DrawableChangedEvent.FIELD_COLOR: ... break;  
            case DrawableChangedEvent.FIELD_DATA: ... break;  
            case DrawableChangedEvent.FIELD_DISPLAYED: ... break;  
            case DrawableChangedEvent.FIELD_METADATA: ... break;  
            case DrawableChangedEvent.FIELD_TRANSFORM: ... break;  
        }  
    }  
});
```

Chapter 13 : Maths and statistics package

The API provides a couple of math objects.

Coordinates

Coord3d uses floats and allows performing these operations:

`add(Coord3d)`, `sub(Coord3d)`, `mul(Coord3d)`, `div(Coord3d)` returning a new instance
`add(float value)`, `sub(float value)`, `mul(float value)`, `div(float value)`
`addSelf(Coord3d)`, `subSelf(Coord3d)`, `mulSelf(Coord3d)`, `divSelf(Coord3d)`
`distance(Coord3d)` compute the distance.
`dot(Coord3d)` compute the dot product.
`normalizeTo(Coord3d)` and `getNormalizedTo(Coord3d)` which return a new instance.
`interpolateTo(Coord3d, float ratio)`, performs a linear interpolation between current and given point.
`negative()` returns the negative coordinate.
`polar()` converts the current cartesian coordinate into a polar coordinate.
`cartesian()` converts the current polar coordinate into a cartesian coordinate.

The Coord2d object provides a subset of these 3d operations and is mainly here as a utility.

Bounding boxes

BoundingBox3d lets you append a list of coordinates (or also other bounding boxes) to get an overall bound. The object indeed holds values `xmin`, `xmax`, `ymin`, `ymax`, `zmin`, `zmax` that are updated over time.

Once created, a bounding box is in an invalid state until one adds a first coordinate or bound. Apart from providing final X, Y and Z min/max values and range, BoundingBox3d provides useful methods:

`valid()` returns true if the bounding box has been given at least one point or box to bound.
`margin(float value)` returns a new bounding box with an added margin on each side (x min, x max, etc)
`selfMargin(float value)` performs the same on the current bounding box.
`scale(Coord3d)` returns a new bounding box with each value multiplied by the given value.
`shift(Coord3d)` returns a new bounding box with each value incremented by the given value.
`intersect(BoundingBox3d)` returns true if both bounding boxes intersect.
`contains(BoundingBox3d)` returns true if this bounding box embed (or is equal to) the given box.
`getRadius()` returns the half diagonal of the box.
`getCenter()` returns the coordinates of the center of the box.
`getVertices()` returns the corners of this bounding box has a `List<Coord3d>`

The BoundingBox2d object provides a subset of these 3d operations and is mainly here as a utility.

Statistics and array processors

As Jzy3d first motivation was to clone the Matlab™ plot3d tool for java, you may find a couple of useful array processing methods that are inspired by Matlab™. We simply provide a listing of these methods, assuming their meaning should be clear by reading the methods' name. Please refer to the javadoc for more information.

org.jzy3d.maths.Array

```
append(int[], int)
atLeastOneNonNaN(double[])
clone(double[])
clone(double[], int)
clone(float[])
clone(float[], int)
clone(int[])
clone(int[], int)
countNaNs(double[])
filterNaNs(double[])
find(double[], double)
find(int[], int)
flatten(double[][][])
flatten(double[][][], boolean)
flatten(float[][][])
flatten(float[][][], boolean)
merge(double[], double[])
print(char[])
print(Coord3d[])
print(double[])
print(double[][][])
print(float[])
print(float[][][])
print(int[])
print(int[][][])
sortAscending(Date[])
sortAscending(double[])
sortAscending(float[])
sortAscending(int[])
sortDescending(Date[])
sortDescending(double[])
sortDescending(float[])
sortDescending(int[])
toColumnMatrix(double[])
toColumnMatrix(float[])
toColumnMatrixAsDouble(float[])
```

org.jzy3d.maths.Statistics

```
mad(double[])
max(double[])
max(float[])
max(float[][][])
max(int[][][])
maxId(int[])
mean(double[])
mean(float[])
median(double[], boolean)
min(double[])
min(float[])
min(float[][][])
min(int[][][])
minId(double[])
minId(float[])
minId(int[])
quantile(double[], double[])
quantile(double[], double[],
boolean)
std(double[])
variance(double[])
```

org.jzy3d.maths.Utils

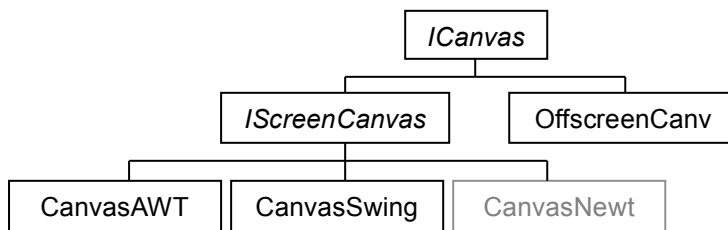
```
abs(double[])
blanks(int)
dat2num(Date)
dat2str(Date)
dat2str(Date, String)
max(Date[])
min(Date[])
num2dat(long)
num2str(char, double)
num2str(char, double, int)
num2str(double)
num2str(double, int)
sum(double[])
sum(int[])
vector(double, double)
vector(double, double,
int)
vector(int, int)
vector(int, int, int)
```

Chapter 14 : Integrate the chart in your application

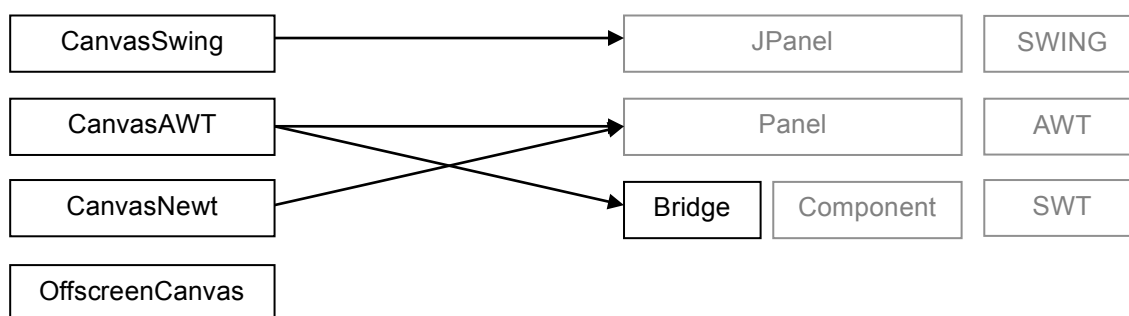
Understanding canvases

As mentioned in the [Chart](#) chapter, you can specify a canvas type in a chart constructor. Specifying either `awt`, `swing`, `newt` or `offscreen` will lead to the initialization of a different `ICanvas` able to work with a given windowing toolkit.

- `CanvasAwt` is an heavyweight AWT canvas (default)
- `CanvasSwing` is a lightweight swing canvas that supports other `JComponents` displayed on top of it (note that it is slower than `CanvasAwt`).
- `OffscreenCanvas` makes use of a `GLPBuffer` to render a chart without any displayed frame.
- `CanvasNewt` is an experimental `IScreenCanvas` relying on *Newt*, a promising windowing toolkit introduced by Sven Gothel in JOGL 2. Note that it has its own mouse and key controllers that are not compatible with other canvases.



The bellow schema shows how each canvas can be embedded in an existing application. Note that SWT charts simply rely on a `Bridge` class able to integrate a `CanvasAwt` into an `SWT Component`.



When working with the samples, you will use `ChartLauncher.openChart(...)`, but this won't let you include the chart in an existing application. To add the chart, retrieve the canvas as shown in the examples bellow.

AWT applications

This assumes you created your chart by calling `AWTChartComponentFactory.chart()`

```
Frame frame = new Frame();
frame.add((java.awt.Component)chart.getCanvas());
```

NEWT for AWT applications

Jzy3d supports Newt, a windowing toolkit introduced by JOGL, designed to be fast and offer more portability across platforms. The Newt Canvas can be enabled by calling the component factory as follow:

```
AWTChartComponentFactory.chart(«newt»)
```

As the method call suggests the underlying canvas will be a Newt Canvas for AWT.

Swing applications

This assumes you created your chart by calling `SwingChartComponentFactory.chart()`

```
JFrame frame = new JFrame();
frame.add((javax.swing.JComponent)chart.getCanvas());
```

SWT applications

This assumes you created your chart by calling `AWTChartComponentFactory.chart()`, as the bridge to SWT uses chart based on an AWT canvas:

```
Display display = new Display();
Shell shell = new Shell(display);
shell.setLayout(new FillLayout());
Bridge.adapt(shell, (java.awt.Component)chart.getCanvas());
```

Offscreen applications

No need to open a frame to have a working chart. One can create invisible charts by calling:

```
Chart chart = AWTChartComponentFactory.chart(quality, «offscreen»);
```

That creates a 800 x 600 chart. One can also specify custom chart dimensions by using the arguments:

```
Chart chart = AWTChartComponentFactory.chart(quality, «offscreen, 1200, 800»);
```

Afterwards, one can generate an image as shown in the next section.

Note that offscreen rendering does not mean that Jzy3d is able to run on headless computer. Indeed, Jzy3d relies on native Open GL libraries that may not be available on computer without GPU.

Screenshots

It is possible to retrieve a `BufferedImage` to do whatever you want with: serialize to disk, use for reporting, or include in a panel. The following piece of code shows how to save a screenshot as a PNG image file:

```
ImageIO.write( chart.screenshot(), «png», new File(«data/screenshot.png») );
```

Note that you can generate a screenshot with offscreen and onscreen charts.

Testing charts

The API package `org.jzy3d.junit` provides usefull tools for testing charts:

- `ChartTest` can run a chart and assert if it is similar to an image file pixel by pixel. If no comparison image exist, an image file is generated and will be used for the next test.
- The `Replay` package is a work in progress to record and replay macros with mouse and keyboard actions. Running interaction tests ensures mouse and keyboard command yield to the same visual result.

Chapter 15 : Component injection

You may need to customize some of library components. It is possible to easily inject your own implementations among the following kind of objects:

- Scene
- View
- Axe
- Camera
- ICanvas
- IMouseController
- IKeyController
- OrderingStrategy

Selecting the appropriate component implementations for a chart requires a `IChartComponentFactory`. This factory is the entry point for getting a new chart:

```
Chart chart = AWTChartComponentFactory.chart(getCanvasType());
```

As you might guess by this factory name, the component factory is also used to conveniently bundle components for a specific *windowing toolkit*. As of 0.9.1, the core API has no more dependency to AWT but only to JOGL. This let the library be buildable for Android applications. The framework provides the following component factories:

- AWTChartComponentFactory
- SwingChartComponentFactory
- ContourChartComponentFactory

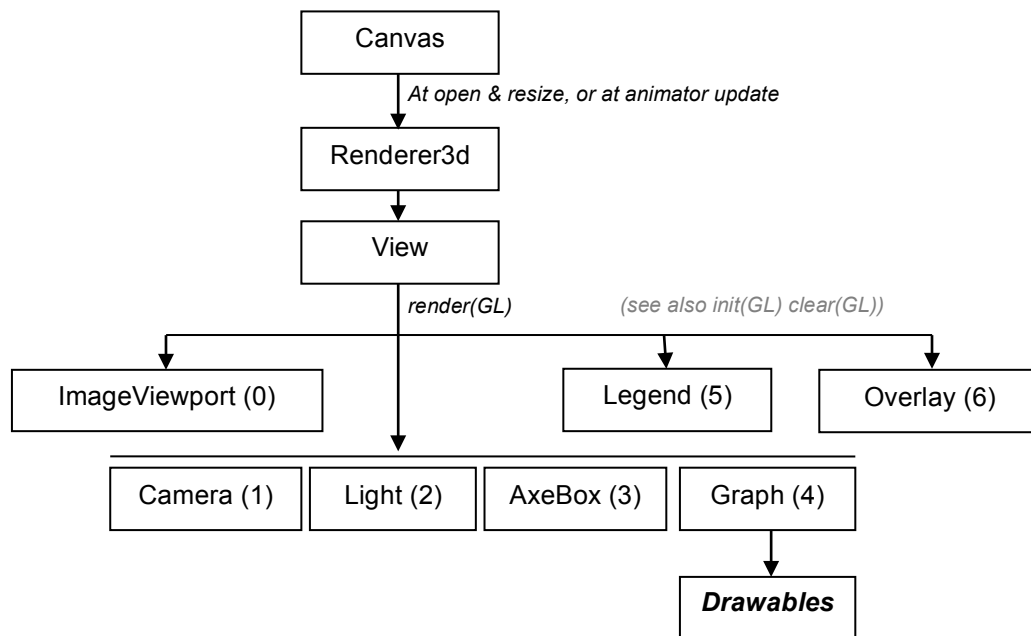
The NEWT canvas is deployed by using the AWT factory as follow:

```
AWTChartComponentFactory.chart("newt")
```

ContourChartComponentFactory extends the AWT factory to instantiate a specific Axe implementation able to draw the contours on the ground of the cube box. This initialization is done by the View that accesses the factory by being a child component of the Chart.

Chapter 16 : Rendering

From canvas to drawables, there are several cascading method calls and layers that are summarized in the schema below.



Thus, a typical rendering sequence:

0. Background ImageViewport rendering
1. Camera settings
2. Light settings
3. AxeBox rendering
4. Graph rendering
 - a. If activated, decomposition of composite into atomic polygons; then ranking
 - b. Rendering of each primitive
5. Legend rendering (ChartView only)
6. Overlay rendering
 - a. Tooltips (Java2d)
 - b. Post renderers (Java2d)

The schema shows rendering triggered by a Canvas, but to be complete, let's mention other components that may trigger rendering:

- Controllers generally update the view by calling `View.shoot()`
- The scene Graph provides `add(...)` and `remove(...)` methods with an option letting you choose whether you want to immediately update the view or not (default is true).

Chapter 17 : Going further with OpenGL

Those wishing to enhance Jzy3d using other OpenGL features might refer to JOGL documentation and mainly to the two official OpenGL books available online:

- <http://www.glprogramming.com/red/>
- <http://www.glprogramming.com/blue/>

The trunk contains a copy of all examples provided in these books that were ported to java by Kiet Le (http://www.opengl.org/code/detail/opengl_redbook_samples_ported_to_java_using_jogl/). This is a huge help for diving intuitively in low level Open GL.