

# 3D Geometry and Rendering ideas

## 3D Geometry and Rendering ideas

This page is supposed to hold ideas and goals for 3D rendering in GeoTools/GeoWidgets.

**This is a call for opinions and ideas. So please answer.** 😊

### Goals

1. Allow threedimensional display of 3D geographical data. Look at Google Earth, WorldWind, Grass, ArcScene and you know what I mean.
2. Ease of use. The API must be intuitive enough for "usual" geoscientists (as is usual a goal of mine).
3. Have the system running as soon as possible, i.e. utilize existing libraries where possible
4. Allow the same 1D/2D/3D objects to be used in 3D AND 2D environments, i.e. to be rendered "flat" if the user wishes this.
5. Be AWT-centric, i.e. paint onto a "Graphics2D" object. SWT adapting can be cared about at the side of the Graphics2D provider, which can convert this to SWT "GC" compatible images (class Graphics2DRenderer).
6. For the rendering side of this, rework the rendering pipeline to cope with 3D objects and implement ideas found on following [Rendering](#) RnD page.

### Existing geometry APIs and implementations

Anything geometry-related I could find.

Implementations:

- Java2D
- Java3D and Vecmath
- Java Topology Suite (JTS)
- GeoTools Geometry ... = JTS
- (GeoTools SLD styling)

APIs:

- Feature Geometry (aka OGC Topic 1 / ISO 19107): `org.opengis.spatial.schema.geometry`
- GO-1 Geometry (subset of the above ISO 19107)
- GO-1 Graphics (display representations of the above): `org.opengis.go.display.primitive`
- (GeoAPI SLD styling): `org.opengis.sld`
- (GO-1 styling, slightly different from above)

🔗 Any more known geometric-related libraries?

### Ideas

#### Geometry API

It would be favourable to have 1D-3D objects stored in an intuitive, geography-aware geometry API. The former means f.e. that objects can be constructed the way a geographer would expect it. A sphere would f.e. be defined by a center point  $[x,y,z,crs]$  and a radius  $[double,Unit?]$ . This means that 3D objects are stored with 3D coordinates and a 3D CRS that defines their exact location on earth.

The same API should also contain 1D and 2D objects: Coordinates(=Points), lines, arcs, flat polygons and the like.

2D and 3D objects should work together seamlessly: If no elevation data is available (as in shapefiles) "flat" 2D objects (elevation=0) could be created. The same objects could be derived from a 2D projection of 3D objects onto the earth surface. Also solids as f.e. boxes could deliver their faces in form of 2D objects (this time usually with elevation!= 0).

In the case of "flat" objects (with elevation=0) JTS operations could be used, possibly with the need to convert back and forth since the 2D objects would probably not be subclasses of JTS objects, which are "flat" 2D only.

Anyway, at a later stage 3D geographic functions would need to be added, such as 3D buffer, distance, intersections, splitting 3D objects a.s.o.

? Is there any API that fulfils all above idea or comes close to it? Something like JTS but in 3D maybe?

## General rendering

Above explained geometries should be rendered either on a 2D or 3D canvas. For this, Java2D resp. Java3D are the implementations of choice in the AWT-compatible Java world. They are both working.

? Are there working alternatives to Java3D for 3D rendering?

One of the goals is that the same geometries can be used in either context. In either case - 2D or 3D rendering - the geometries have to be converted into some objects understood by Java2D resp. Java3D. For example a sphere would need conversion to a `java.awt.geom.Ellipse2D` (or a subclass) respective to `com.sun.j3d.utils.geometry.Sphere` (or a subclass). These objects would then be reused every time the map/scene has to be rerendered.

Actually Java3D could as well be used to render a "flat" map as every conventional MapCanvas does. However I'd suggest having a Java2D-based alternative for people that don't need 3D functions and therefore don't want to have the Java3D libraries installed. (Which btw. are partly licensed under BSD and partly under Java Research License (JRL)/Java Distribution License (JDL) for noncommercial resp. commercial use.)

## Java3D and geography

In its current state, pure Java3D is neither intuitive to a geographer, nor is it geography-aware. It has no sense about "the earth" or coordinate reference systems and their meaning. However it would be possible to create a thin wrapping API that hides the Java2D internals (such as TransformGroups and the whole tree approach) and instead provides (geo)object oriented functions.

The geographer would just add 3D georeferenced, styled objects into the wrapped Java3D "universe" and define location and position of the viewer. The underlying implementation would care about:

- conversion of the coordinates from the object's CRS to the geocentric XYZ coordinate system as it is used in Java3D
- conversion of units to meter and so on
- conversion of SLD or other styling information to Java3D styling information

A point to check is how good Java3D could cope with SLD styling or styling in general. I expect this to be rather tricky!

? Who of the GT developers and other interested readers is familiar with Java3D? I am, but I am not an expert. Matt hias

? Have there been articles already about using Java3D for styled geobjects rendering? If so, please add here.

## Issues and problems to solve

## Coordinate Reference Systems and bounding boxes

As well known, Coordinate Reference Systems (CRS) include the typical GeographicCRS and ProjectedCRS, but also GeocentricCRS, TemporalCRS and Engineering CRS. So, in theory, when a 3D object constructor requires a `CoordinateReferenceSystem` object, any of these can be passed in. The implementation would need to either cope with them correctly or throw an exception (for example if a TemporalCRS is passed in).

Dealing with ProjectedCRS and GeographicCRS is fine: They usually have an axis pointing north-south and one pointing east-west. The third one (if exists) points up-down. Objects having such a CRS would produce a bounding box with these three axes.

Problems arise when an GeocentricCRS is passed in. Although it would define the location of coordinates as precisely and unambiguously as do ProjectedCRS and GeographicCRS, it would usually produce a bounding box in GeocentricCRS. This BBox would need conversion to some ProjectedCRS or GeographicCRS in order to get (for example) merged with BBoxes of other map layers or other features of other feature types of the same layer. However conversion from Geocentric BBox to Geographic BBox (or back) is very inaccurate.

Possible solutions:

1. Restrict CRS in constructors to `ProjectedCRS` or `GeographicCRS`. Problematic since there is no interface that covers both.
2. Check CRS argument for a list of supported types, correct if possible (i.e. convert 2D to 3D if possible) otherwise throw an exception ("Unsupported CRS type").
3. Let objects return Bounding boxes in any requested CRS. It is up to the implementation how to calculate the BBox as precise as possible in this requested CRS. So it might even be possible to have 3D objects stored in GeocentricCRS, as long as they give precise BBoxes in any other CRS type.
4. Combination of 2 and 3 (*Looks to me like this is the preferred approach.* Matthias)

## How to cope with GridCoverages?

GridCoverages are essentially flat, but with an elevation model quite things can happen...

## How to integrate elevation models, interpolated surfaces and the like?

Use coverages as elevation models...

## How to integrate TINs?

TINs are both geometrical objects (multiple 3D triangles) and coverages ...

Java3D could render them as lattice model or as actual surface. Nice to have.

## The time axis

Objects could move in time through the space or they could change their size over time. Some objects might have a start and end date between they exist only.

Java3D can cope with this. But the geometry API and the Java2D rendering API would need to deal with this too. Just something to think about...

## Specific implementation ideas and proposals

...go here. Feel free to add ideas how to best achieve the above.

 Or possibly there are better general approaches than the above?