

Projektpraktikum für das
Studium “Angewandte Informatik” in Salzburg

G2k-Plot

**A framework for the
visualization of cadastral data
for geodetical applications/systems**

Praktikant

Marc Strapetz
mstrap@cosy.sbg.ac.at
Institut für Computerwissenschaften,
Universität Salzburg

Universitärer Betreuer

Helmut Mayer
helmut@cosy.sbg.ac.at
Institut für Computerwissenschaften,
Universität Salzburg

Praxisbetreuer

Stefan Schiffer
schiffer@dcx-systeme.de
DCS Computer Systeme GmbH,
Ainring, Deutschland

A framework for the visualization of cadastral data for geodetical applications/systems

Marc Strapetz

October 17, 2002

Abstract

We will present a framework, the `plot.graphic` package, for the visualization of cadastral data which has evolved for the geodetical system “GEOi[2000]” (DCS Computer Systeme). The motivation to develop this framework were major shortcomings of the JAVA 1.1 graphics API (`Graphics`) in displaying vector oriented data and we will show, that our framework is a solution to this problem and is also a valuable extension to the JAVA 1.2 graphics API (`Graphics2D`). Please note, that this document is also intended to be a technical documentation of the framework, so it contains also information and links to the source code.

1 Motivation

The JAVA 1.1 graphics API provides basic services for drawing, which are all pixel-oriented. This means, that a function draws a line on a graphic device, like the screen, by drawing its pixels. Thereafter the pixels remain and the line is ‘forgotten’. I.e. graphic objects are only collections of unrelated pixels. For most applications, this API can’t be used directly, but serves as a good foundation for the construction of a more sophisticated framework which is specialized for the problem domain.

Geodetical software systems – like GEOi[2000] – are likely to work with vector-oriented data instead of pixel-oriented data. This mismatch between the JAVA graphics API and the requirements of such a software system was the main reason to develop our framework. We will now give a more detailed list of requirements on our framework:

1.1 Requirements

The main goal is to have a “visualization” component for geodetical objects. Geodetical objects are points, lines/polygons, but also geodetical calculations. All these objects consist of a set of attributes which define their graphical representation. For instance, a point has coordinates and is of a specific type; a circle-calculation can consist of two circles with a specific radius and has normally two intersection points.

For the user of GEOi[2000], the visualization component is a simple window, containing graphics. This window is part of the desktop frame. This window shows a specific view (a subset of all) of the geodetical objects. The user must be able to change this view, e.g. to zoom in a specific area, translate the view or to set a specific scale for the view.

For a comfortable integration, the visualization must also provide the selection of specific geodetical objects, like points. This is the foundation for Drag & Drop of an object (to a different window) or for object-specific actions, like removing the objects from the visualization, etc. To always give the user feedback about his

position within the visualization, the geodetical objects beyond the mouse cursor should also be marked by a (blue) rectangle.¹ Finally, the visualization must also be able to be printed with a predefined scale – and with high quality, of course.

The stated requirements suggest an object-oriented approach. The foundation will be ‘graphic objects’, which are related to the corresponding geodetical objects. A graphic object can draw itself to a graphic device. Its co-ordinates are specified for a world co-ordinate system and “views” define areas of this co-ordinate system, which are identified with a graphic device, like the visualization component window. This way, the graphic information is not bundled in pixels, but in objects which live longer than just for one display and are independent of a specific graphic device.

1.2 Cooperation with the University of Salzburg

To obtain a robust and stable framework which would fulfill the stated requirements, DCS decided to build this framework in cooperation with the University of Salzburg, where the author studies Applied Computer Science. This cooperation should guarantee a high quality based on scientific methods – which we think, it actually has.

2 Abbreviations and Notations

Following abbreviations are frequently used within the source code and within this document.

gob(s) means one or more objects of the class `G2GraphicObject`. It can also mean the class `G2GraphicObject`.

specified often appears within the class documentation created by javadoc. This word is used to talk about parameters of methods. Talking about the actual parameter names within javadoc documentation might be a bit confusing, because the parameter names are not shown in the short part of the method description. And because most methods have only one parameter, talking about ‘the specified so-and-so’ is unambiguous. We took over this notation from Java’s API documentation.

E.g. for the declaration `public Rectangle getContextBoundingBox(G2Context gc)` the ‘specified `G2Context`’ denotes the parameter called ‘gc’. Although it is more difficult to write ‘specified `G2Context`’ instead of ‘gc’, We think that it is not more difficult to read, but it is more robust against changes within the source code. We observed for many times, that parameter names change more often than their types.

this/these ² When documenting the source code, it is often necessary to talk about the current class. Sometimes one wants to talk about the class itself, sometimes about an object of this class. In nearly all cases the context of the description makes it unambiguous, what of the two things is meant. To shorten descriptions for both meanings ‘this’ is used. ‘This extends that’ means that the current class extends some other class, but ‘returns this’s height’ means the value of the attribute ‘height’ of objects of the current class. ‘These’ mostly means a set of objects of the current class but can also mean the current class and its subclasses.

¹Although, this seems to be a detail, we will see that it requires an efficient painting of the visualization and therefore has huge impact on the design of the framework.

²These words are my own creation. They express a relation to ‘this’ and ‘these’.

ties/tied Many objects within this framework work together much, although their classes are designed more independently. For example, there can exist many **G2PaintAreas**, but only one of them is used by a specific **G2View** and only this one is for the **G2View** important. But this **G2PaintArea** is not an aggregate of **G2View**, so it would be wrong to talk about the ‘**G2View**’s **G2PaintArea**’. On the other hand, simply talking about ‘**G2PaintArea**’ in relation with **G2View** is too general and doesn’t express, that only the one, important **G2PaintArea** is meant. To express exactly this kind of relation, we use ‘the tied **G2PaintArea**’. Normally, the reason that two objects are tied is a third object, for which these objects are aggregates. This third object ‘ties together’ the both objects.

graphic device is a device, that can hold a pixel-oriented graphic. For most occurrences the screen is meant, but also memory images and printers are graphic devices.

Graphic means all things, what this package is able to produce. The ‘Graphic’ is not just a screen shot, consisting of some pixels, but the whole thing, which is created³, when concrete objects of this framework are instantiated and run.

user is a person, who uses the program, where this framework is included. It is not a programmer, who uses this framework within his program.

programmer is a programmer, who uses this framework for within his program.

2.1 Notation for diagrams

For diagrams, the UML notation is used, when possible.

2.2 Notation of co-ordinates

Co-ordinates will be abbreviated as ‘coor’ or ‘coors’. Additionally, because within this framework different co-ordinate-systems are used, a character which represents the system type will be written directly before ‘coors’, like with **cCoors**, which means ‘context co-ordinates’, or ‘oCoors’, which means ‘original co-ordinates’.

When talking about a specific ordinate, like the x-ordinate, this can be abbreviated as ‘xCoor’, combined with a system this yields ‘oxCoor’, ‘cyCoor’ etc. The systems themselves are abbreviated as ‘O-System’ or ‘oSystem’ resp. ‘C-System’ or ‘cSystem’.

This is similar with geometric figures, like rectangles. ‘Rectangle’ can be abbreviated as ‘rect’, and can be combined with a system, like ‘cRect’. This is the same with ‘Polygon’ and ‘poly’ etc. ‘Bounding box’ is commonly abbreviated as ‘BBox’. E.g., combined with the **cSystem** this yields ‘cBBox’.

3 Main components

Unlike the Java API, where the single class **Graphics** includes all functionality, this framework contains a set of classes that work together. For the beginning, the main characteristics for the most important classes are outlined. Later, when discussing more sophisticated topics, these descriptions will be more and more completed.

³In a more poetic, but not less significant formulation: ‘comes to life’

G2GraphicObject **G2GraphicObjects** represent all sorts of geometric figures. They are primary characterised by their co-ordinates, maybe their colour, their label etc. Gobs can draw themselves to a graphic device. And most important of all, they have an identity. Gobs can be worked on, they can be added to the Graphic, they can be removed, they can be changed and they can be marked (and thereafter selected) by the user.⁴

G2PaintArea This is the most fundamental class. It represents the basic characteristics of an area that can be painted on. A very basic characteristic for such an area is its size.

Additionally **G2PaintArea** separates the area into an inner and an outer region. The inner region – also referenced as ‘active area’ – is the region, where gobs are primarily drawn to. The outer region – also referenced as ‘passive area’ or ‘border’ – is reserved for some special paintings. Gobs might be drawn to this region, but only in special situations. However, **G2PaintArea** doesn’t care what the regions are used for. So the second thing, what a **G2PaintArea** characterises is just the bordering rectangle between the inner and the outer region.

G2ObjectManager This class manages all gobs, that are part of the Graphic. It can be seen as a set of gobs, but its internal structure is more complex than the structure of a set. If the gobs are imagined to lie within a world co-ordinate system, this class could be said to manage the world co-ordinate system. Gobs can be added, removed and – what is very important – found quickly by their co-ordinates.

G2View This class represents an area of the co-ordinate system, or simply a ‘view’. A **G2View** is used to select a set of gobs in the world co-ordinate-system. Selected are those gobs, which lie within the **G2View**.

G2Context This class represents a kind of ‘graphic context’. It combines a **G2View** with a Java **Graphics**-object. Roughly spoken, this combination makes it possible to actually draw gobs to a graphic device, what is the main purpose of **G2Context**. Additionally, a **G2Context** has a specified outlook. For example, it can visualize the passive part of a **G2PaintArea**.

G2Painter This class represents things, that know what’s to be painted in which way. Painters are responsible for what a graphic devices actually shows. They decide which gobs are drawn, how the **G2-MainScreenContext**, which is introduced later, is painted etc.

G2VisibleManager This class is used to manage the visible gobs. Visible gobs are those, which can be seen on the graphic device. Which gobs are visible is defined by a **G2View**. **G2VisibleManager** provides methods to find gobs quickly. This is extremely useful to find all gobs that are drawn within a specified region of the graphic device. For example, finding gobs at specified cCoors for marking them is a time critical operation, where previously set up association between cCoors and gobs have to be used.

G2GraphicSettings This class contains general settings for many classes of the framework. E.g. characteristics of the **G2PaintArea** and the **G2Context** can be specified here. For the whole program there exists only one instance of this class, though there can be more than one Graphic.

G2ContextSettings This class contains specific settings for a **G2Context**. There might be more than one instance of this class.

⁴The user can use the mouse to mark a gob. When the mouse cursor is moved over a part of the screen, where a gob is painted on, this gob gets a blue rectangle around it to emphasise it. When the user clicks on a gob, the gob gets selected. Many operations (like translating gobs, removing gobs from the Graphic etc.) work on the currently selected gob(s).

G2MainScreenContext This class represents a graphic context for the screen. It can have some special features, like a ruler that shows the scale etc.

G2ObjectPainter This class is used to draw gobs for standard situations. It will be explained later, what standard and not-standard situations are.

G2SpecializedPainter This class is used to draw gobs for non-standard situations.

4 The co-ordinate systems

Within this framework two different co-ordinate systems are used.

4.1 The O-System

The O-System is the original (user, world) co-ordinate system. Gobs have O-Co-ordinates (oCoors) and they can be located anywhere in the O-System. oCoors have floating-point values. The **G2ObjectManager** manages this O-System. A **G2View** represents an area of this O-System. The user can change oCoors for gobs and for a **G2View**. There is no default unit for the O-System, because oCoors can only be interpreted by the user or programmer.

4.2 The C-System

The C-System is the context co-ordinate system. Context co-ordinates are the same as device co-ordinates (for graphic devices). For a given **G2View** and a given **G2PaintArea** context co-ordinates (cCoors) can be derived from oCoors. The mapping between oCoors and cCoors is done by a **G2cMapper**, which is provided by **G2View**. The user and the programmer have very limited access to cCoors.

The default unit for the C-System is pixels, if not differently specified. This is a very natural arrangement because all graphic devices normally have pixels as display unit. Also in the source code all quantities which are related to the context or to a graphic device have pixels as unit. Only sometimes, when such quantities can be set by the user, their unit is millimetres or metres.

5 Class G2Graphic

G2Graphic is the main class for all framework components. Programmers will mostly access methods of this class. This class provides methods for adding and removing gobs, setting the current **G2View**, drawing etc.

G2Graphic ties together most of the components mentioned above. It has

- one **G2GraphicSettings**
- one **G2PaintArea**
- one **G2View**
- one **G2ObjectManager**

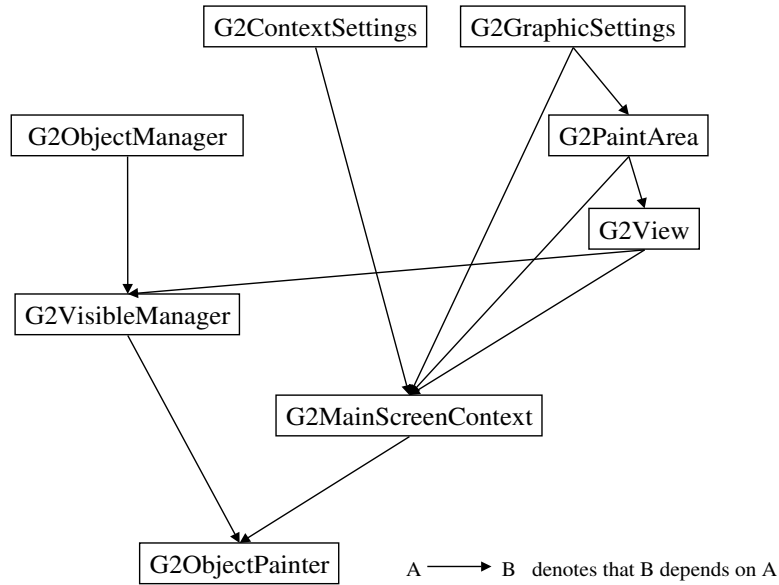


Figure 1: Dependencies of the components from **G2Graphic**

- one **G2ContextSettings**
- one **G2MainScreenContext**
- one **G2VisibleManager**
- one **G2ObjectPainter**
- one **G2SpecializedPainter**

5.1 The dependency of these components

Figure 1 shows the dependencies between these components. This diagram shows only the current situation. It will probably change when new features are added to the framework. Especially dependencies between **G2GraphicSettings** and other components are most likely to be added, when making the framework more configurable.

The figure shows the dependencies for those objects that **G2Graphic** ties together. This means for example, that not every **G2PaintArea** must depend on **G2GraphicSettings**, although most of the dependencies are generally applicable.

5.1.1 The cause for each dependency

- **G2GraphicSettings** - **G2PaintArea**: The border between the active and the passive region of the **G2PaintArea** and the size for one pixel in millimetres can be set within **G2GraphicSettings**.
- **G2PaintArea** - **G2View**: The **G2View** provides temporary **G2OcMappers**, which are dependent on the size of the **G2PaintArea**, especially on the size of the active area.
- **G2GraphicSettings** - **G2MainScreenContext**: Various settings, like fonts and features for the **G2MainScreenContext** can be set within **G2GraphicSettings**.

- **G2ContextSettings - G2MainScreenContext:** **G2ContextSettings** characterises some aspects of the **G2MainScreenContext**'s outlook.
- **G2PaintArea - G2MainScreenContext:** The **G2MainScreenContext** can visualize the passive and the active part of the **G2PaintArea**.
- **G2View - G2MainScreenContext:** The **G2OcMapper**, which the **G2MainScreenContext** uses for mapping **oCoors** to **cCoors**, is obtained from the **G2View**.
- **G2View - G2VisibleManager:** The **G2VisibleManager** manages all gobs within the **G2View**. These are exactly those gobs, which are called 'visible'. When the **G2View** changes, the set of visible gobs might change, but at least the size of the managed region changes. Because the **G2VisibleManager** is considered to have a very efficient implementation, it might need to know about these changes.
- **G2ObjectManager - G2VisibleManager:** When gobs are added, removed or changed within the **G2ObjectManager** and these gobs get, were or are visible, then the set of visible gobs has changed.
- **G2MainScreenContext - G2ObjectPainter:** Because the **G2ObjectPainter** draws gobs using the **G2MainScreenContext**, changes of the **G2MainScreenContext** will change the **G2ObjectPainter**'s painting. Co-ordinates or the outlook of the **G2MainScreenContext** might have changed.
- **G2VisibleManager - G2ObjectPainter:** The **G2ObjectPainter** might (and does indeed) use the **G2VisibleManager** to determine those gobs, which have to be drawn (because they are visible). The presumption, that every implementation of the **G2ObjectPainter** uses the **G2VisibleManager** might be criticised. When skipping these dependency, a **G2ObjectManager - G2ObjectPainter** dependency would have to be set up instead. When implementing these dependencies, one will see, that both solutions yield in the same result.

6 The dependency mechanism for the tied components

6.1 Theoretical aspects

There are several solutions to manage the dependencies between the tied components of **G2Graphic**. The most intuitive solution might be the following.

Each component (each class) knows, which components depend on it. When this component changes, it knows, that the depending components have to change too, because they rely on the state of this component, whereby the state is defined by its attribute values. This component could inform all depending components about the change and the depending components could update themselves. E.g. if a gob is added to the **G2ObjectManager**, the **G2ObjectManager** informs the **G2VisibleManager** and the **G2VisibleManager** checks if the added gob is visible and if it is, the **G2VisibleManager** includes this gob in its managing.

This solution fits not well for different reasons:

- The dependency mechanism shall not manage dependencies for all objects of a given class, but only for some, like those which **G2Graphic** ties together. For most other program-contexts this dependency mechanism is not necessary, because objects are set up once and do never change, like when printing the Graphic. To get rid of these unnecessary dependencies, it would be possible to let each component check, if it is part of the Graphic. Only if this is the case, it would inform its dependent components. This solution demands much knowledge within each component and this should be avoided due to the following reason.
- The tied components should know less about the other components. Why should a **G2PaintArea** have to know about a **G2MainScreenContext**, or even about a **G2Context** (the superclass of **G2MainScreenContext**).

- Most important, updates can be very (time-)expensive. E.g. `G2ObjectPainters` are likely to use images, which visualize the `G2MainScreenContext` and all gobs and can easily be painted to a graphic device. Painting this image is fast, but creating it takes relatively much time – Why `G2ObjectPainters` have to paint fast instead of update fast is discussed later. E.g. let five gobs be added to the Graphic and then the `G2ObjectPainter` should paint to the screen. The gobs are put to the Graphic, one after another. With each gob, the `G2ObjectManager` changes and informs the tied `G2VisibleManager`. The `G2VisibleManager` updates, has changed and again informs the `G2ObjectPainter`, that updates. So the `G2ObjectPainter` updates five times before the screen is repainted. The screen, then, shows only the last update of the `G2ObjectPainter`, the previous four updates have never been used and were useless.

To avoid these problems, we use a kind of observer pattern, namely a dependency manager with “delayed update”.

6.2 The implementation within package `plot.graphic.dependency`

Following classes are used within the dependency mechanism.

G2GraphicListenable An interface that makes an object able to listen for changes of objects.

G2GraphicObservable An interface that makes an object observable for changes of itself.

G2GraphicDependent This is the root class for all classes, that participate actively in the dependency mechanism. It implements both, the `G2GraphicListenable` and `G2GraphicObservable` interface. A `G2GraphicDependent` is listening and observing at the same time.

G2GraphicDependencyManager This class manages the dependencies. It is informed about changes of `G2GraphicObservables` and informs `G2GraphicListenable`s about changes of their dependent `G2GraphicObservables`.

6.3 The concept of `G2GraphicDependent`

`G2GraphicDependent` offers basic methods that derived classes can use to declare themselves changed and to update, when they are obsolete. Obsolete means, that an object might rely on an old state of objects it depends on. This can lead to a situation, where the object is requested to represent (visualize) the current situation, but it represents an old situation. What the representation of the current situation means for a specific object has to be decided by the author of the object.

For the following description, it is assumed, that objects are derived from `G2GraphicDependent`.

setChanged() When an object changes, it calls this method to declare itself changed. `G2GraphicDependent` will inform the `G2GraphicDependencyManager` about this change. `setChanged()` calls `preChange()` before the `G2GraphicDependencyManager` is informed and `postChange()` after the `G2GraphicDependencyManager` has been informed. These methods can be customised by derived classes if they want to execute additional code, when they are changed.

setObsolete() When an object *o*, on which object *l* depends, has changed, the **G2GraphicDependencyManager** informs object *l* about the change of object *o* by calling this method. This makes object *l* obsolete.

maybeUpdate() When an object *l* is dependent on another object *o*, it might fetch information from object *o* and store it internally. Always, when object *l* accesses this internal information, it wants to be guaranteed that this information is not obsolete. If it is, it would have to fetch the information again and maybe do some (more or less) expensive calculations. To ensure that the information is not obsolete, object *l* must always call **maybeUpdate()** before accessing such information. **maybeUpdate()** checks if object *l* is obsolete, and if it is the case, it calls **update()**.

update() Updates the object. E.g. this method is called, when **maybeUpdate()** notices that the object is obsolete. The object is set to be not obsolete anymore, because it has been updated. Because this method will execute the code to refresh the object, it can be called directly by the object, if needed.

doUpdate() This method is called by **update()**. By default this method does nothing. Derived classes have to overwrite this method to actually update the object. E.g. the **G2VisibleManager** can overwrite this method with recalculating the visible gobs, an expensive operation, which should not be executed too often. Derived classes should never access this method directly, but instead call **update()**, because **update()** resets the object's obsolescence-flag.

6.4 Deriving from G2GraphicDependent

The mechanism is quite simple, but when deriving from **G2GraphicDependent**, one must be very cautious for following reasons.

6.4.1 State changing methods

Each method must be analysed, if it can change the current object. If this is the case, **setChanged()** must be called.

By rule of thumb, set-methods in general change objects, but only if the new value, which is to be set, differs from the current value.

One must carefully distinguish between those changes, that can have an effect on dependent objects and other changes, that can't. If an object uses some attributes to store temporary values, changing these values doesn't change the object, when looked upon from outside the object. So this kind of change can't effect dependent objects and **setChanged()** should not be called.

6.4.2 State relying methods

Each method must be analysed, if it relies on the current state of the object, which depends on other objects. If this is the case, **maybeUpdate()** must be called.

For example, set-methods usually only write to attributes and need no update. get-methods read from attributes and are likely to need an update.

maybeUpdate() should only be called by the class itself and not for other classes. Each class should know best, for which situations updates are needed.

6.4.3 Conclusions

In fact, it's not as difficult to include a new class into the dependency mechanism as it seems. Generally spoken, it's better to call `setChanged()` and `maybeUpdate()` too often, than too seldom. But each call of `setChanged()` or `maybeUpdate()`, which is not necessary, should be skipped, because it might slow down the whole painting-process.

6.5 Case study: Adding to G2objectManager

The example problem is, what happens inside the dependency mechanism, when a gob is added to the tied `G2objectManager`. Following operations are executed - resp. results are seen - from outside the framework:

- A gob is added to the Graphic.
- Update is called, so the changes get painted to the screen.
- The screen is painted.

Figure 2 shows what happens inside the framework. Function calls that are not important for the understanding of the dependency activities are skipped. The executed actions are described by pseudo-function calls.

7 Detailed discussion of G2GraphicObject and its derivates

Gobs can represent all sorts of geometric figures. The base class for all gobs is `G2GraphicObject`.

7.1 Finite and infinite gobs

It can be distinguished between finite and infinite gobs. Finite gobs have a bounding box in the `oSystem`. This `oBBox` might be a single `oPoint`. Infinite gobs (those gobs, that implement `G2InfiniteGraphicObject`), have no bounding box, because they have at least one infinite co-ordinate.

7.2 Interactive gobs

Besides its location, it's an important characteristic of a gob, if it is interactive. Gobs which are not interactive, can only be painted. Interactive gobs can be associated with regions of the graphic device by providing a so-called 'context-fetch-polygon' by `getContextFetchPolygon()`. Interactive gobs can be highlighted and selected. So the user can work with these gobs, like dragging them etc.

When designing a gob, it's important to find a good context-fetch-polygon. It should not be too large, but not too small either. Context-fetch-polygons are used by the `G2VisibleManager`. They should be fast to calculate, because lots of them have to be calculated when the `G2Context` changes. A gob is not interactive by default.

7.3 The cBBox of a gob

Each gob must provide a bounding box for a specified `G2Context` by `calcContextBBox()`. This `cBBox` is used by the `G2VisibleManager` to manage the gob. It's further used by `G2SpecializedPainters` to

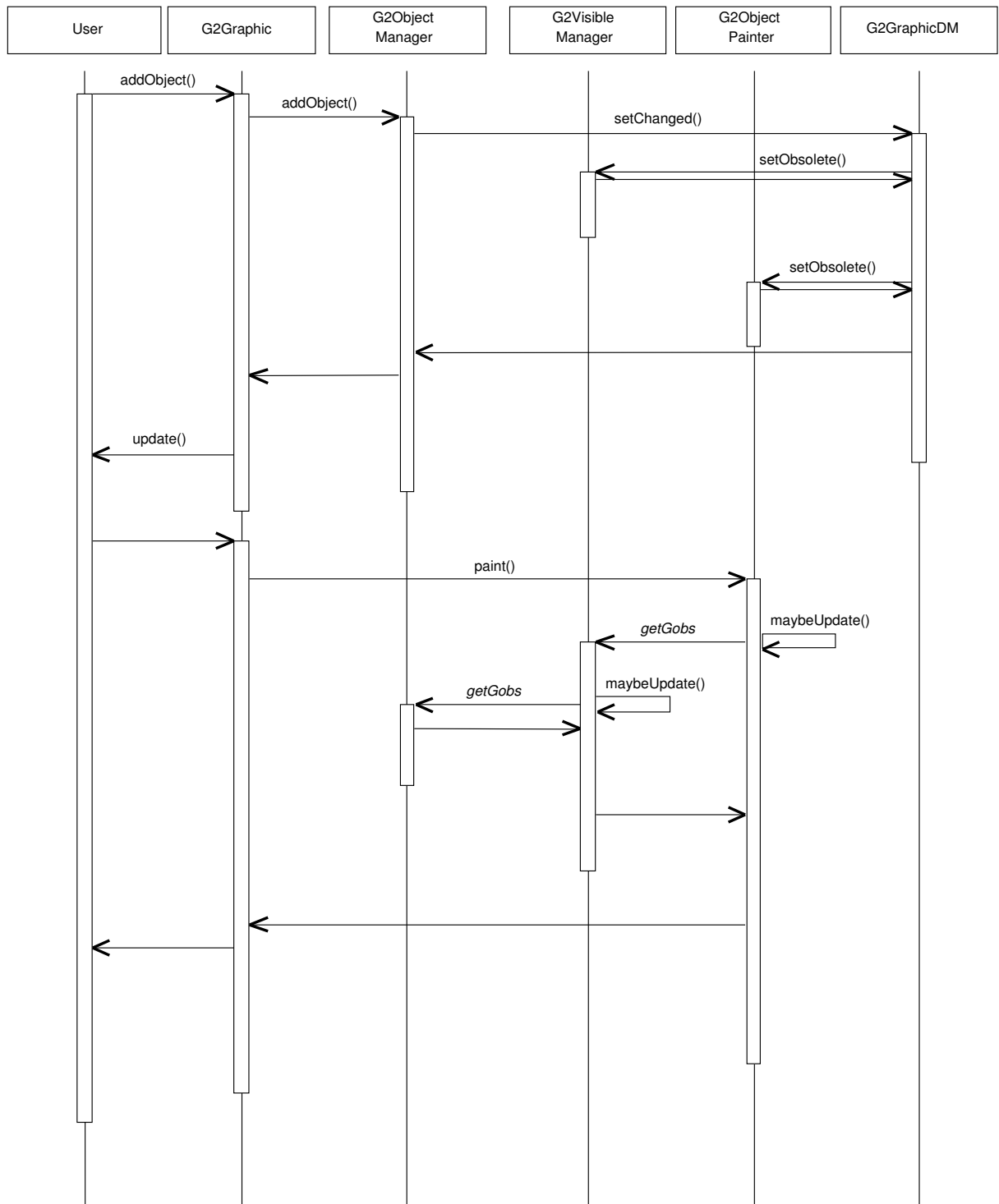


Figure 2: Adding a gob to the Graphic

calculate their dirty regions.

It's very important, that for a given `G2Context` with fixed parameters all possible drawing operations of the gob would lie within the gob's `cBBBox`. It's assured, that callers of `calcContextBBBox()` use the resulting `cBBBox` only until the `G2Context` has changed. Then, if necessary the system calls `calcContextBBBox()` again to ensure, that the `cBBBox` fits to the context's current parameters.

8 How `draw()`/`update()` work

When parameters of the Graphic are changed the Graphic must be updated to the graphic device. E.g. when adding gobs, changing the `G2View`, `G2GraphicSetting` etc. the screen needs to be updated.

There are two ways how the graphic device is updated.

- Java recognises a change of the graphic device from outside the framework (e.g. the Swing Component, which is painted on, changes).
- The programmer sent an 'update command'. E.g. a gob was added to the Graphic and then the screen should be updated to show this gob. Because the Graphic can't know how many changes are made at once, it can't send the update command by itself.

In general, updating is a very expensive operation, because the `G2ObjectPainter` has to be updated, the graphic device has to be repainted etc. So updates should be placed carefully and only if necessary.

`G2Graphic` provides the method `update()` for updating the Graphic. This method analyses, which regions of the graphic device have to be repainted and sends an appropriate `repaint()` to the tied graphic device.⁵

As one can see, `draw()` has a Java graphic context for its parameter. And exactly this is the reason why `G2Graphic.update()` can't draw by itself, because it is missing such a Java graphic context. Swing uses double buffering (painting to memory images) for painting `JComponents`, so the Java graphic context which is given to `draw()` changes from call to call and can't be cached for later draws to directly draw to the device.

8.1 From `update()` to the screen

To explain the way from `update()` to an updated screen, following example is used:

Example The programmer has added a gob to the Graphic. Adding a gob makes the `G2ObjectPainter` - among others - obsolete.

Now, `update()` is called. In general, `update()` checks, if an update is necessary and which regions of the graphic device have to be updated. There are following possibilities:

- The `G2ObjectPainter` is obsolete. In this case the whole device has to be repainted, because it doesn't reflect the current state of the Graphic anymore.
- If a `G2SpecializedPainter` has the control over the painting-process, it depends on this `G2SpecializedPainter`, if an update is needed. Again, the whole device or only specific regions of the device might have to be updated.

⁵Currently this is done by sending a `repaint()` to the `G2GraphicCanvas`, a `JComponent`. The graphic device manages the `repaints()` and decides if and what actually has to be repainted. Java does a repaint by calling `G2GraphicCanvas's paint()`, which results in a call of `G2Graphic.draw()`, which finally paints to the graphic device.

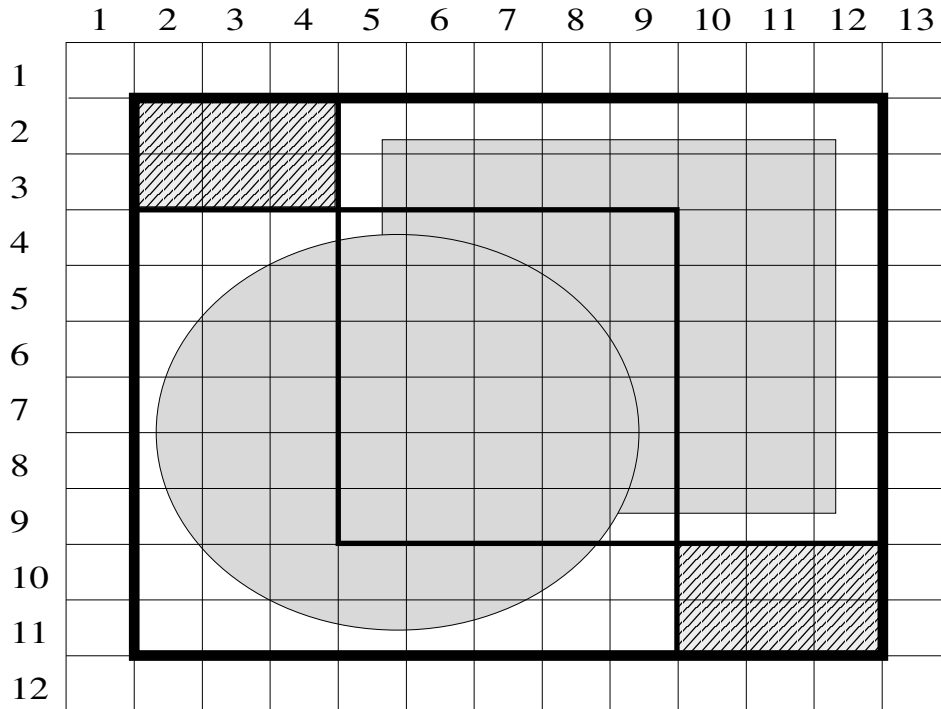


Figure 3: dirtyRegions

Updating the whole device is simple, but expensive. Updating only regions is more efficient but tricky. Those regions, which have to be updated are called ‘dirty regions’. Now, `update()` calls `repaint()` for these dirty regions and then `draw()` will be called by Java. For this example, the whole device must be updated.⁶

`draw()` is the main method for painting to the graphic device. First it has to assign the Java graphic context to all ‘device dependent’ components of `G2Graphic`.⁷

Now, it depends on the `G2Painter`, that paints the Graphic to the device. This can either be a `G2ObjectPainter`, or for special cases, a `G2SpecializedPainter` could have been set. When a `G2SpecializedPainter` has been set, it has more priority than the `G2ObjectPainter` and so it must paint to the device.

8.2 Dirty regions

Dirty regions and related processes are explained by an example of `G2MarkerPainter`, Figure 3.

Update has to determine the dirty regions, i.e. those regions which have to be repainted. Therefore `G2SpecializedPainters` have to manage their dirty regions and return it by the call of `getPaintRegions()`.

For the given example, an efficiently implemented `G2MarkerPainter` which is a subclass of `G2SpecializedPainter`, is likely to return two small rectangles (e.g. `[2,4]x[9,2]` and `[5,2]x[12,9]`) for its dirty regions, one for the previously marked (and now unmarked) rectangle gob and one for the now marked round gob.

⁶The reason is found in the source code.

⁷Currently, this is only the tied `G2MainScreenContext`.

Exactly these regions of the screen have been affected by changing the marker and exactly these and (only these) regions have to be repainted.

`update()` requests two repaints for the graphic device, one for the first and one for the second rectangle. In general, if a `G2SpecializedPainter` returns many dirty regions, `update` might request a total repaint of the device. Repaint-requests result in the call of `G2Graphic.draw()`, where the specified device context is clipped for the requested area of the repaint. E.g. if a repaint was requested for `[10,10]x[20,20]`, `draw()` can only paint within this rectangle. All painted pixels outside this rectangle will be ignored.

Java buffers `repaint()`-requests and it can occur, that many small requests are bundled to one large request. For the example, the distance between the two rectangles is small. Java will bundle this to only one request (`[2,2]x[12,11]`) and call `G2Graphic.draw()`. So the clip-bounds of the device context might be set to `[2,2]x[12,11]`.⁸ Java expects this region to be painted correctly.

This makes the situation a bit difficult. On the one hand, the `G2MarkerPainter` declared the two rectangles for its dirty regions and is only expected to be able to draw these dirty regions. At the other hand, Java expects the framework to paint the one, big rectangle. This problem is solved by the function `G2SpecializedPainter.canPaint()`. This function is called within `G2Graphic.draw()` and returns, if the `G2SpecializedPainter` is able to draw the specified region. By default `canPaint()` checks if the specified rectangle lies within the `G2SpecializedPainter`'s dirty regions. In this case it returns true, otherwise it returns false.

If `canPaint()` returns false, `draw()` first draws the `G2ObjectPainter` which always can paint the whole device, and then draws the `G2SpecializedPainter`. Efficiently implemented `G2SpecializedPainters`, like the `G2ImageMarkerPainter` overwrite `canPaint()`. The `G2ImageMarkerPainter` uses the `G2ObjectPainter` to draw the gobs and then just draws the marker. So it can overwrite `canPaint()`, signalling that it can paint everything, though its dirty regions are only the `cBoxes` of the marked/unmarked gobs.

In general, it's better to implement `G2SpecializedPainters` in a way, that their declared dirty regions are small, even if this means that the `G2ObjectPainter` has to paint larger regions - resulting from Java's way of bundling repaint-requests. The reason will be clear, when discussing `G2ObjectPainter`.

8.3 The 'inPaint-mode'

Within `draw()` the Graphic is set to the 'inPaint'-mode. This mode is set before the first painting occurs. While the 'inPaint'-mode is set, no object that is tied with the `G2Graphic` may be changed. This ensures that no inconsistencies in the visualization can occur. If this mode was not used, e.g. following problem could occur.

Example An old painter is destroyed within `draw()`, where `G2SpecializedPainter.destroy()` is called. Let the `G2SpecializedPainter` have overwritten this method with a code, that translates the Graphic. This action makes the `G2ObjectPainter` obsolete, which is assumed to have not been obsolete before. An obsolete `G2ObjectPainter` would result in a total repaint of the device, but because it was not obsolete when `update()` was called, only a small region of the device was planned to be repainted. The `G2ObjectPainter` would now paint this region translated, but the rest of the device remains not translated. This results in an inconsistency.

When the 'inPaint'-mode is set, a `G2SpecializedPainter` trying to change something causes an exception.

⁸We found no information, about how Java bundles the `repaint()` requests. Surely, it depends on the frequency of these requests, but might also depend on the VM and on the OS.

9 G2Painter and its derivatives

G2Painters are the uppermost objects in the paint-hierarchy. They use most of the classes of this framework to paint to a graphic device (especially to the screen).

As mentioned above, it can be distinguished between the ‘standard’ situation and ‘non-standard’ situations. The standard situation is that just the (visible) gobs of the Graphic shall be painted. E.g. the screen shows a set of gobs and a `repaint()` is sent. Then the set of gobs should be painted. For such a standard situation a **G2ObjectPainter** is used.

When the Graphic is zoomed in, translated, when gobs are marked or dragged etc., then this is a non-standard situation. Gobs should be painted, but they are in a kind of unstable state. For this kind of situations **G2SpecializedPainters** are used.

It would be thinkable to abandon **G2SpecializedPainter** and to use a **G2ObjectPainter** for all kinds of situations. This is possible, because normally, the state of the Graphic (respectively the state of the gobs) is not stored within a **G2Painter** (e.g. the translation of the **G2View** is stored within the **G2View**. If a gob is dragged, its current position is stored within the gob etc.). A **G2ObjectPainter** could also – as the **G2SpecializedPainters** do – use this information to paint non-standard situations. But because the **G2ObjectPainter** is a very general painter, the way it is painting is not very effective for special situation. And that was the motivation to introduce **G2SpecializedPainters**.

E.g. when the Graphic is translated the **G2ObjectPainter** would always paint all gobs for the current translation.⁹ A more efficient solution is to paint the gobs to a memory image and then just paint this image to the correct co-ordinates for the current translation. This is exactly what the **G2ImageTranslationPainter** does.

9.1 G2ObjectPainter

There’s only one major requirement to a **G2ObjectPainter**: It must be fast.

9.1.1 Why have G2ObjectPainters to be fast?

G2ObjectPainters are often used. Especially **G2SpecializedPainters** use a **G2ObjectPainter** to paint the standard part of the non-standard situation and then, additionally paint the non-standard part of the non-standard situation. E.g. the **G2ImageMarkerPainter** uses the **G2ObjectPainter** to draw the marked/unmarked gob and only the marker is drawn by the **G2ImageMarkerPainter** itself.

9.1.2 Where have G2ObjectPainters to be fast?

The process of painting can be separated into two phases, a preparation phase, that prepares the painter for a new situation, and an actual painting phase. It’s an empirical result, that the **G2ObjectPainter** often paints the same situation. This means that the preparation phase must be passed one time, the actual painting phase must be passed many times. So the conclusion for an efficiently implemented **G2ObjectPainter** is that it should optimise the duration of the actual painting phase on cost of the duration of the preparation phase.

A **G2ObjectPainter** should be fast in painting only specific regions, defined by rectangles, of the device. E.g. the **G2ImageMarkerPainter** uses the **G2ObjectPainter** to draw its gobs. Point gobs are usually small

⁹A translation of the Graphic happens, when the user drags on the graphic device.

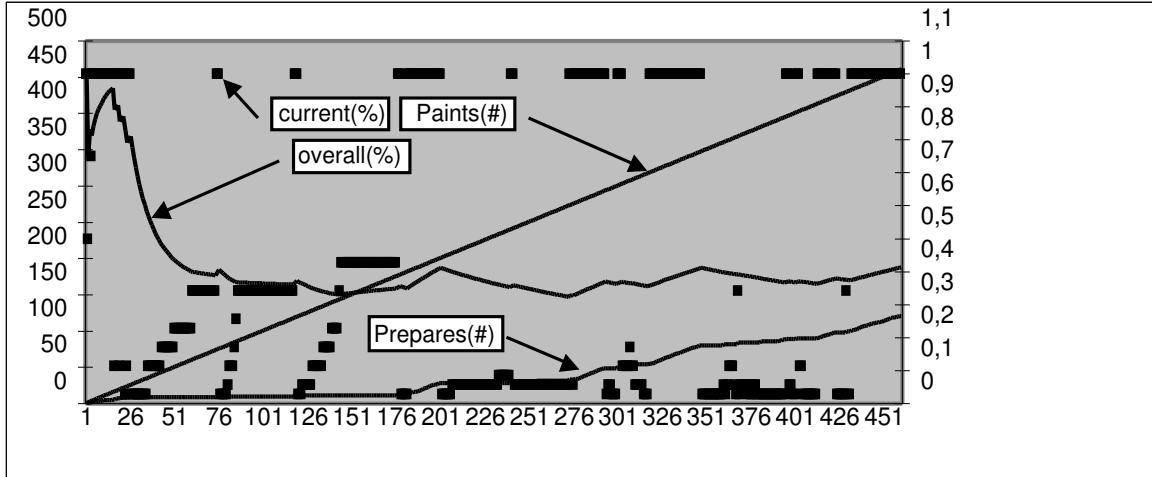


Figure 4: An experiment with G2TileObjectPainter

and therefore only a very small region of the whole device, which contains the marked gobs, has to be painted.

9.2 G2TileObjectPainter

The **G2TileObjectPainter** is an efficiently implemented **G2ObjectPainter**, which meets the mentioned criterions. It separates the whole device horizontally and vertically into tiles of the same size. For each tile a memory image is created and the corresponding regions of the device is painted to this image.

For the case of the **G2ImageMarkerPainter** the **G2TileObjectPainter** will mostly just have to paint one tile-image to the device. If the whole device has to repainted, the **G2TileObjectPainter** paints all tiles.

9.2.1 An experiment with the G2TileObjectPainter

Figure 4 shows statistics of an experiment with the **G2TileObjectPainter**. The experiment was set up as follows: The screen area for the Graphic was about 300x200 pixels. 243 point gobs were added to the Graphic. The goal of the experiment was that the user connects five points to a polygon. The x-axis of the shown statistics shows the number of paints.

As the left y-axis shows, the screen was painted 461 times ('Paints(#)'), the preparation phase of the **G2TileObjectPainter** was run through 121 times ('Prepares(#)'). For the left y-axis, 'Current(%)' shows for each paint, how many percent of the whole device were painted. 'Overall(%)' shows, how many percent of the whole device were painted during the whole experiment up to the current paint.

During the experiment, it was taken care, that only necessary gobs were marked. The more gobs are marked, the less tiles (relative to the whole device) have to be painted. One can imagine, that with increasing number of gobs in the Graphic, the number of 'erroneously' marked gobs increases and the advantages of the **G2TileObjectPainter** gains on importance.

9.3 G2SpecializedPainter

`G2SpecializedPainters` are used to paint the device in non-standard situations.¹⁰

The main philosophy for `G2SpecializedPainters` is, that the device must be fully reconstructable at any time. This means that a `G2SpecializedPainter` may not rely on previous paintings, that it had done, but instead must always be able to redo these previous paintings, if the device reflect show these paintings. The reason is – already mentioned – that Swing uses double buffering, which means, that for every call of `G2Graphic.draw()` the device must be considered to be empty.

`G2SpecializedPainters` are not only designed to show non-standard situation, but also to implement new features of the Graphic, which result in non-standard situations. E.g. the `G2ViewZoomingPainter` zooms the `G2Graphic`'s `G2View` in and out, what would not be possible without `G2ViewZoomingPainter`. Because `G2SpecializedPainters` are indeed specialised, they can also ensure, that `G2Graphic.update()` is called when necessary.

So, various `G2SpecializedPainters` plus `G2Graphic` make out the main interface to this framework.

10 Outlook

We have presented a framework for displaying cadastral data, which is based on the JAVA graphics API. It overcomes the APIs shortcomings of displaying vector-oriented data by a more object-oriented approach: Instead of pixels, graphical objects are managed by the framework. The visualization of these objects is based on a view of the world-coordinate system (corresponding to geodetical coordinate systems) which is mapped to a graphics device. The framework uses a painter-philosophy, which is both flexible and efficient, as specific painters can be implemented to handle specific situations and we have verified the efficiency in detail by a tiled painter. To reduce code complexity within the framework itself, components communicate indirectly via a dependency manager.

The presented framework is part of the geodetical software system “GEOi[2000]”, which displays points, lines/polygons and geodetical calculations. Also with a large number of objects, their display is still satisfyingly fast.

What the framework is currently missing, is a layered architecture. To have multiple layers for different sorts of objects is very common in geographic information systems and is also desirable – although not necessary – for GEOi[2000]. Another well-accepted concept are “renderers”, which are for instance used by Swing tables, etc. Instead of graphic objects, graphic renderers could be used to display geodetical objects directly. The advantage of using renderers would be to omit the construction of graphic objects and the maintainance of consistency between graphic object and geodetical object – although we want to note, that renderes can not always replace graphic objects, as there are graphic objects, which have pure and only graphical meaning and have no corresponding “renderable object”.

11 Acknowledgements

We want to thank the management of DCS Computer Systeme for giving us the chance and time to develop our framework; we want to thank our supervisors Helmut Mayer and Stefan Schiffer for keeping the project on course and for all their helpful comments and suggestions which finally led to the flexibility and robustness of the current solution.

¹⁰During the development of this framework, it was experienced that memory images are very fast and should be used if possible. So derivatives of the `G2SpecializedPainter`, like the `G2ImageMarkerPainter` and the `G2ImageTranslationPainter` were created.

A Class diagram (with relations) of `plot.graphic`

