I2G

INTERGEO

Deliverable Nº: D3.5

I2G **API Specification**

The Intergeo **Consortium**

Version: 1.0 of May 31, 2009
Based on: I2G Common File Format v1.0

Author:

Ulrich Kortenkamp (U. of Education Schwäbisch Gmünd, Cinderella)

Project co-funded by the European Community under the eContentplus Programme

Project ref.no.	ECP-2006-EDU-410016
Project title	Intergeo - Interoperable Interactive Geometry for Europe

Contractual date of delivery	M10
Actual date of delivery	June 1st, 2009
Deliverable title	API Specification
Type	Presentation
Status & version	submited 1.0 of May 31, 2009
Number of pages	18
WP contributing to the deliverable	WP3
WP/Task responsible	Daniel Marquès
Contributors	Ulrich Kortenkamp (U. of Education Schwäbisch Gmünd, Cinderella), Yves Kreis (Université du Luxembourg, GeoGebra), Paul Libbrecht (DFKI GmbH), Daniel Marquès (Maths for More & WIRIS)
EC Project Officer	Krister Olson
Keywords	dynamic geometric system, file format, open-math, API, standard
License	This document is licensed under the Creative Commons License *Namensnennung-Keine Bearbeitung 3.0 Deutschland*; see `http://creativecommons.org/licenses/by-nd/3.0/de`

Contents

1 Introduction

This document describes the specification of a common API (Application Programming Interface) for Interactive Geometry software. An API is used to programmatically influence the behavior of software, to exchange data between software, and to extend software with new functionality. This usage is beyond (asynchronous) data exchange that can be handled via a common file format, which is the reason why the Intergeo Consortium sees the need to specify such an API.

As the API is still evolving, this document should not be seen as a programming manual, but as the documentation of the design decisions and requirements as defined by the working group within the Intergeo project. The actual code and the accompanying source and API documentation is specified using an iterative software development process. All files necessary to implement the API are available through a SourceForge-hosted project at `http://intergeo.sourceforge.net/`. The specification follows a cycle with development and release builds. Thus, we can hope for the necessary sustainability of our approach.

1.1 Overview

We first describe some usage scenarios that guide the design of the API. Then we define a layer structure, which we further describe in the following sections. Finally, we describe the process that will be used to decide on the API specification that will be used during the lifetime of the project and also thereafter.

2 API Use Scenarios

In this section we describe possible applications of the Intergeo API. Most of these are inspired by existing uses of Interactive Geometry Software, however, all these approaches are currently realized using proprietary interfaces.

2.1 Applet Communication

Almost every Interactive Geometry Software can be embedded into web pages, using either Java, Flash, JavaScript/SVG, a dedicated plugin, or ActiveX. For the sake of simplicity, we refer to all these approaches as “Applets”, although this name is usually meant for Java applets only.

As soon as you have two applets on the same webpage, it might be useful to allow communication between them. Here, either both applets might show the same construction, but in different views (1A — Inter-Applet-Communication, same construction), or different constructions that share some data (1B — Inter-Applet-Communication, different construction). Views – the display of the construction – might differ in various ways. A construction may be rendered in various mathematical models (Euclidean Plane, Poincare Model, 3D embedding, textual description, . . .), or with different parameters (change of color, visibility of objects, zoom factor, . . .).

Note that it is not necessary for both applets to represent interactive geometry software; one (or both) might be replaced by any other software that implements the Intergeo API.

It is useful to distinguish a third scenario. As any modern browser includes a JavaScript engine, we also want to enable easy access to the applet from JavaScript (2 — Applet-Javascript-Communication). This can be used, for example, to create alternative user interfaces, or to create assessment facilities (Exercisers).

2.2 Standalone Communication

Most interactive geometry systems are also available in a stand-alone version that runs without the help of an enclosing browser. Stand-alone software has direct access to the computer's resources, in particular to the file system, works on- and offline, and might be better suited for some applications than a web-based system.

It is desirable to also have a standardized way of communicating with these stand-alone programs, for example when setting up script mechanisms to start up software in educational settings (3 — Standalone-Software-Control). Also, when running on a desktop computer further computational tools (a CAS like Maple or Mathematica) or visualization components (3D renderers like JReality) can be available. In order to use these, a bidirectional communication between the interactive geometry software and the other tool must be possible (4A — Standalone-Software-Communication). The communication can also be handled using manual interaction of the user through the system clipboard (4B — Standalone-Software-Communication (Clipboard)).

Another usage is the access to user interface and semantic events that are created by the interactive geometry software, for example when recording students' sessions for further analysis using a tool like Jacareto (5 — Event Recording).

2.3 Server Side Processing

The third major area of application for the Intergeo API is server-side processing of geometry content. On the I2Geo platform at `http://i2geo.net` we intend to use the API for at least three purposes: To retrieve static preview images for constructions (6 — Construction Processing), to convert proprietary files to the I2G format and vice-versa (7 – Construction Conversion) and to retrieve the necessary HTML code to embed an applet into a page (8 – Construction Information Retrieval). All these applications need a proper contextualization of the construction, that is, the server might have more information than just a plain file.

2.4 Summary

Documents can refer to the sketched scenarios above using the following codes:

1A Inter-Applet-Communication, same construction

1B Inter-Applet-Communication, different construction

2 Applet-Javascript-Communication

3 Standalone-Software-Control

4A Standalone-Software-Communication

4B Standalone-Software-Communication (Clipboard)

5 Event Recording

6 Construction Processing

7 Construction Conversion

8 Construction Information Retrieval

9 Other (has to be specified)

All APIs should include a reference to at least one of these scenarios.

3 API Layers

In order to be both flexible and easy to extend, we define a three-tier architecture, the transport layer which defines the possible ways to use the API; the command layer which defines the possible commands; and the data layer, which details the low-level data structures.

3.1 Transport Layer

The transport layer defines the possible ways to communicate with an interactive geometry software. This includes all local mechanisms (like direct calls into a Java application or Scripting support) as well as network-based mechanisms (like TCP/IP communication or a REST-API).

Included with every command layer is a description how commands are translated into the specific syntax or mechanisms used by the transport layer.

3.2 Command Layer

The command layer defines all possible commands that can be send to an application or applet. All commands may refer to an existing construction and can provide additional arguments. The transport layer should provide easy mechanisms to work with constructions as “resources” identifiable by unique IDs.

Software that implements the Intergeo API may implement only a subset of the commands, but some subsets of the commands are required to achieve Intergeo compliance. In particular, an “Intergeo Applet API,” an “Intergeo Application API,” and an “Intergeo Server API” will be defined that include the commands needed for Inter-Applet communication, for desktop applications, and for the I2G platform.

3.3 Data Layer

All data that is used together with a command needs to be encoded. The data layer defines the specifics of these encodings, in particular for constructions and mathematical objects.

3.4 Additional Information

All API implementations need three additional files that provide additional information on the implementation: `LICENSE`, `AUTHORS` and `README`. The files may have extensions like .txt or .html that specify their content format.

`LICENSE` has to clarify the license of the API implementation that has to be compatible with the license of the API definition as available on `http://intergeo.sourceforge.net`.

`AUTHORS` has to clarify who is responsible for this particular implementation, including a contact email.

`README` has to include documentation about partial implementations, limitations of the implementation, and any necessary deviations from the standard API semantics.

4 Transport Layers

In this section we suggest some transport layers. Not all of these have to be implemented by a software that wishes to conform to the Intergeo API. However, if the Java API described first is implemented, then the libraries provided by the consortium can be used to give access to the others.

4.1 Java Transport Layer

Many Interactive Geometry Systems are implemented in the platform independent language Java, which makes this a natural choice for the reference implementation of the API. Commands can be translated to either method calls in a singleton object[1] or instance methods on a proxy object that references a construction.

For example, the `loadFile` command described in Sec. 5.1.1 would translate to a method in an API object that implements the “static” part of the Intergeo API:

```
ConstructionProxy loadFile(java.net.URI filelocation)
```

This returns a proxy object for a construction, while the `writeFile` command described in Sec. 5.1.6 would translate to a method

```
void writeFile(java.net.URI filelocation, String mimeType)
```

in a class that implements the construction proxy interface (i.e. `ConstructionProxy`).

[1]This is necessary because it is not possible to inherit or overwrite static methods.

4.2 JavaScript Transport Layer

In particular for browser-embedded scenarios it is necessary to offer JavaScript-support for the Java API. We will provide a bridge between JavaScript and Java implementations of the API.

4.3 TCP/IP Transport Layer

Some software already offers remote protocols via TCP/IP (see `http://www.iana.org/assignments/port-numbers`, in particular port 3770). Again, we intend to offer a bridge between a generic TCP/IP transport and the Java transport.

4.4 REST Transport Layer

Recent webservices are usually realized using the REST architectural style. We can regard any construction as a resource in that sense, and use the commands in suitable URIs. While we do not define a discovery mechanism for software APIs, we still think that it is desirable to offer access to at least the consortium-provided software via the API. This is in particular useful for the server scenarios 6, 7 and 8 (see Sec. 2.3).

4.5 Clipboard Transport

Every operating system offers a concept like a system clipboard. While it is usually used to transfer data only, we can also put commands as defined in the command layer on the clipboard. This enables users to transfer actions from one software to the other using copy & paste.

4.6 Scripting Languages

In addition to the JavaScript transport layer other bindings to scripting languages (AppleScript, Ruby, . . .) could be useful, in particular in server scenarios (Sec. 2.3).

5 Command Layer

In this section we define groups of commands that the API should provide. This list is not definitive and has to be approved through the community process as defined in Sec. 7.

All commands have a unique name, given in the title of the subsection. The translation of this name into the syntax needed for the transport layer has to be specified by the transport layer. The final specification of parameters and return values will be available in the corresponding documentation at `http://intergeo.sourceforge.net`.

All commands that have an ID as first parameter should be local to a construction, as defined in the transport layer.

5.1 Startup and File Access

5.1.1 `ID` loadFile(`URI`)

Load a construction file given by a **`URI`** and return a unique identifier for further access. This should not open any views.

5.1.2 `ID` load(`i2g-data`)

Loads a construction file given by **`i2g-data`** and return a unique identifier for further access.

5.1.3 openView(ID, number)

Opens the view with the given number for the construction given by the **`ID`**. Views are specified in the `display`-section of the I2G Common File Format v1.0.

The construction has to be loaded in advance.

See 5.1.1.

5.1.4 closeView(`ID`, `index`)

Close the view of the construction identified by **`ID`** which is given by the 0. See 5.1.3.

5.1.5 addView(`ID`, `display-specification`)

Adds a view to the construction identified by **`ID`**. The views properties are defined by **`display-specification`**.

5.1.6 writeFile(`ID`, `URI`, `format`)

Write the construction given by **`ID`** to a file given by URI. The format can be specified by a MIME type given in **`format`**.

5.1.7 `image` renderViewToImage(`ID`, `index`, `format`)

Return the view with the given **`index`** to an image of the MIME type given by **`format`**.

5.2 Creation & Deletion

5.2.1 `ElementID` createElement(`ID`, `element-specification`)

Create an element for the construction with the given **`ID`**. If the specification includes an **`ElementID`**, this will be used for the element, unless an element with this **`ElementID`** already exists. The element does not have any constraints, these should be added using the addConstraint command.

Returns the **`ElementID`** for the new element.

See 5.2.2.

5.2.2 addConstraint(`ID`, `constraint-specification`)

Add the constraint(s) given by **`constraint-specification`**. Constraints should only use **`ElementID`**'s that are available in the construction. Use the data access methods given in Sec. 5.3 to find out about available elements, or use the createElement command.

See 5.2.1

5.2.3 deleteElement(`ID`, `ElementID`)

Removes the element given by **`ElementID`** *and all constraints* referring to this element.

5.2.4 deleteElements(`ID`, (`ElementID`, `ElementID`, `...`))

Convenience method to remove a list of elements and the associated constraints.[2]

Removes all elements given by the list of **`ElementID`**'s *and all constraints* referring to them.

5.3 Data Access

5.3.1 `element-specification` getElement(`ID`, `ElementID`)

Returns the **`element-specification`** of the element given by **`ElementID`** in the construction given by **`ID`**.

5.3.2 `constraint-specification` getConstraints(`ID`, `ElementID`)

Returns all **`constraint-specification`**'s available for the element **`ElementID`**.

[2]Remark: Currently (as of version 1 of the I2G file format), constraints do not have a unique identifier. So the only way to access constraints is via the referenced elements.

5.3.3 constraint-specification getDefinition(ID,ElementID)

Returns the **constraint-specification** for the element **ElementID** that list it as an output element, if there is one.

5.3.4 display-specification getView(ID, index)

Returns the **display-specification** for a view of **ID** given by **index**.

5.3.5 (ID, ...) getConstructions()

Returns a list of **ID**'s of all available constructions.

5.3.6 (index, ...) getViews(ID)

Returns a list of all available views.

5.3.7 double getX(ID, ElementID)

Convenience method to return the *x*-coordinate of a point given by **ElementID**.

5.3.8 double getY(ID, ElementID)

Convenience method to return the *y*-coordinate of a point given by **ElementID**.

5.4 Manipulation

5.4.1 modifyElement(ID, element-specification)

This general-purpose method can be used to change the properties of any element. If some of the properties are impossible to achieve due to constraints, then the behavior is unspecified.

5.4.2 modifyElementView(ID, display-specification)

Changes the view attributes of an element. The number of the view may be given in the **display-specification**.

5.4.3 movePoint(`ID`, `ElementID`, `double` x, `double` y)

This convenience command reflects the need for an easy way to change the (Euclidean) coordinates of a point. The point referenced by **`ElementID`** will move to coordinates (x, y), if possible.

5.5 Communication and Scripting

5.5.1 execute(`ID`, `script-language`, `code`)

Execute the script given by **`code`**. The script language is chosen by **`script-language`**.

Script languages may be proprietary. The access to elements of a construction and to the API from within the script is not defined, but see Sec. 4.6 for further information about access from scripting languages. A software may choose to enable access to the API for scripting languages.

6 Data Layer

In this section we define the various data types that are needed in the command layer. The encoding of the data is transport dependent.

All data types (in particular the **`ID`** data) may be encapsulated by objects. In that case the API has to provide access methods to the objects that take native data types of the underlying transport method as a parameter. We provide the native types used to refer to these encapsulations in the Java layer as an example. A JavaScript implementation may use JSON, for example.

6.1 `ID`

An **`ID`** identifies a construction temporarily, i.e. during the lifetime of the application or server that implements the API.

In Java, the **`ID`** can be referred to by any `String` value.

As many commands refer directly to a construction given by an **`ID`**, it is often more efficient to provide a proxy object to access these ‘ìnstance methods”. This proxy object is defined in the API implementation.

6.2 `index`

An index identifying a view of construction. These views are defined in the display specification of the I2G Common File Format v1.0.

In Java, **`index`** can be referred to by a positive integer value.

6.3 `format`

A MIME type. We recommend that every software implementing the API that also provides a proprietary or other standard file format registers a MIME type with IANA.

In Java, this can be specified as a `String`, for example "`application/vnd.cinderella`".

6.4 `URI`

A uniform resource identifier that locates data on the internet.

In Java, this can be specified using `java.net.URI`.

6.5 `ElementID`

An element identifier that refers to the name attribute of an element in the I2G Common File Format v1.0.

In Java, this can be specified using a `String` value.

6.6 `double`

A numerical floating point value in double precision. Depending on the transport layer, more precision may be available. It is not guaranteed that the software is able to handle the precision. The implementation description should provide details on this.

6.7 `i2g-data`

The XML data of an I2G file in I2G Common File Format v1.0.

In Java, this can be specified using `org.xml.sax.InputSource`.

6.8 `element-specification`

The XML data for an element, as specified in the elements part of the I2G Common File Format v1.0.

In Java, this can be specified using `org.xml.sax.InputSource`.

6.9 `constraint-specification`

The XML data for one or several constraints, as specified in the constraints part of the I2G Common File Format v1.0.

In Java, this can be specified using `org.xml.sax.InputSource`.

6.10 `display-specification`

The XML data for one or several specifications in the display part of the I2G Common File Format v1.0.

In Java, this can be specified using `org.xml.sax.InputSource`.

6.11 `script-language`

An identifier specifying a (proprietary or standard) script language. The identification should be unique to the application, using either a MIME-type like specification or the reverse domain name notation as customary in Java.

In Java, this can be specified using a `String` value.

6.12 `code`

A script, that is, program code in either a standard scripting language like JavaScript, Python, or Ruby, or in a proprietary language like Mathematica, Maple, or CindyScript.

In Java, this can be specified using a `String` value. For larger scripts, an alternative specification method can be provided. This should be detailed in the `README` section of the implementation. We urge all implementors to agree on a standard specification method.

6.13 `image`

Binary data describing an image.

In Java, this can be specified using a `java.awt.image.BufferedImage`.

6.14 Lists

All element types can potentially occur in form of lists of several objects of the *same* type. The base type has to be specified.

In Java, a collection subclassing the generic `java.awt.List`<T> can be used.

7 Approval Process

The ongoing development of both Interactive Geometry Software and the Intergeo platform requires a constant revision of the API specification. We do not expect this document to be the final revision of the API, but we foresee an evolution. This section specifies the continuing process that will be used to submit, revise, and approve new versions of the API.

7.1 Documents

The API is defined by the collection of files available at `http://intergeo.sourceforge.net`.

The svn repository version of this document at `http://svn.activemath.org/intergeo/Deliverables/WP3/D3.5/D3.5-API-specification.tex` contains the current draft for discussion of the document. Every time a new version is released, it will be added to the the sourceforge repository. All older released versions will be archived in the documentation tree of the sourceforge repository.

All references to the I2G format have to specify its version number.

7.2 Actors

WP3 Team. Those partners (consortium, associate and user) working actively in work package 3, i.e. take part in discussions on the wp3 mailing list and in video conferences of the wp3 group.

WP Leader. The work package leader as defined in the Description of Work of the Intergeo project. Responsible for the organization of meetings and discussions of the WP3 team.

API Authors. Those persons designated by the WP3 team to write this document.

API Implementors. All developers working on the SourceForge project at `http://intergeo.sourceforge.net`. New developers can join this project. The current SourceForge project leaders are Yves Kreis and Ulrich Kortenkamp.

7.3 Decision Rules

Until September 2010 all decisions about the API implementation are made by the API implementors in unanimous decisions. They can be overruled by decisions of the WP3 Team.

The decision rules after that date have yet to be determined.

7.4 Release Cycle

This document and the accompanying API implementation are in constant development. At least every 3 months the then current version will be finalized with a new version number. The document will be presented to the WP3 Team which will review it and, if it approves it, release it.

7.5 Licensing

The API and the accompanying libraries will be licensed using an open-source license that prohibits the distribution of altered versions under the same name, in order to maximize redistribution and interoperability.

The exact license of the API will be specified at `http://intergeo.sourceforge.net`.

8 Version History

Version 1.0 - May 31, 2009 - First version published.