Space Details

Key:	NDOC
Name:	Nunaliit Documentation
Description:	
Creator (Creation Date):	ahayes (Oct 20, 2006)
Last Modifier (Mod. Date):	ahayes (Oct 20, 2006)

Available Pages

0. Preface

This page last changed on Oct 20, 2006 by ahayes.

The Nunaliit Cybercartographic Atlas Framework is a powerful tool that enables people to create cybercartographic atlases. Cybercartography strives to go beyond traditional cartography to incorporate multisensory and multimedia data from multiple disciplines into a highly interactive and engaging representation based around maps. The Nunaliit framework is designed to capture the relationships between separate pieces of information and then take care of rendering an interactive interface without requiring content experts to be software developers.

Nunaliit is a product of the Cybercartography and the New Economy (CNE) project at Carleton University. The 4-year CNE research project is a multi-disciplinary endeavour funded by the Social Sciences and Humanities Research Council of Canada and makes use of infrastructure provided by the Canadian Foundation for Innovation, as well as by Carleton University. We greatly appreciate their support.

Many faculty members, partners, staff, and students from a wide variety of fields have contributed to the theory and practice of Cybercartography through this project, some of them making significant direct contributions to Nunaliit. Please see for the comprehensive list of project members who contributed their time and energy to the CNE project. Many thanks to all our contributors.

In addition, a number of third party open source projects have been included in Nunaliit. Please see the licensing information in the documentation folder for the list of included projects. Thank you to the communities behind these projects for making them available. Nunaliit is also developed with and depends on other open source projects. Our thanks to the Mozilla Foundation, the Eclipse Foundation, and the Subversion and Subclipse teams.

As of this writing, the framework is still maturing. Developers at the Geomatics and Cartographic Research Centre at Carleton University are actively working on Nunaliit, so if you see something unexpected, or fail to see something you would expect, please let us know. Amos Hayes <ahayes@gcrc.carleton.ca>

1. Introduction

This page last changed on Oct 20, 2006 by ahayes.

Goal of the Nunaliit Cybercartographic Atlas Framework

The aim of Nunaliit is to provide a means to capture all geographically linked information used to create an atlas module in such a way that it is independant from presentation.

The purpose of the Atlas Framework is to establish a language in XML, using an XML schema, that captures the essence of an atlas module. An atlas module developed that way establishes relationships between text, map (geographic object) and multi-media objects without imposing presentation restrictions. Once an atlas module is developed, tools can be used to transform it into various renderings, where it can be presented to an end user.

The benefits of this approach are:

- To liberate the atlas author from the technical details that slow down the production of an atlas;
- Multiple presentations of the same atlas module can be generated without requiring interventions from the author;
- Same look and feel can be established between various modules developed independently;
- Areas of concerns about the development, production, and hosting of the on-line atlas can be decoupled.

Other immediate goals with the Nunaliit project:

- Build a compiler that renders atlas modules already developed
- Build a rudimentary set of tools to help author create atlas modules

What is a module?

A module is the basic unit of an atlas. Conversly, an atlas is a collection of modules. Modules contain information about a certain subject and generally are the right size to be developed by a single or few authors. Modules can be authored with an end atlas in mind or not. If a module is not dependent, in content, on other module, it can then be migrated between atlases without any change.

In practice, a module is an XML document that complies with the schema imposed by the atlas framework. The content of the document can be thought of as a mark up language that establishes relationships between maps, text and multi-media entities. Module documents can be processed by an atlas compiler and transformed into an artifact with which an end-user can experience the information provided by the author.

Currently, Nunaliit offers an atlas compiler that transforms a set of atlas modules into a web application. This compiler is found in the Nunaliit-SDK. Once an atlas is compiled using the Nunaliit-SDK, it can be deployed on a web server. The following picture depicts the process from module creation to atlas deployment.

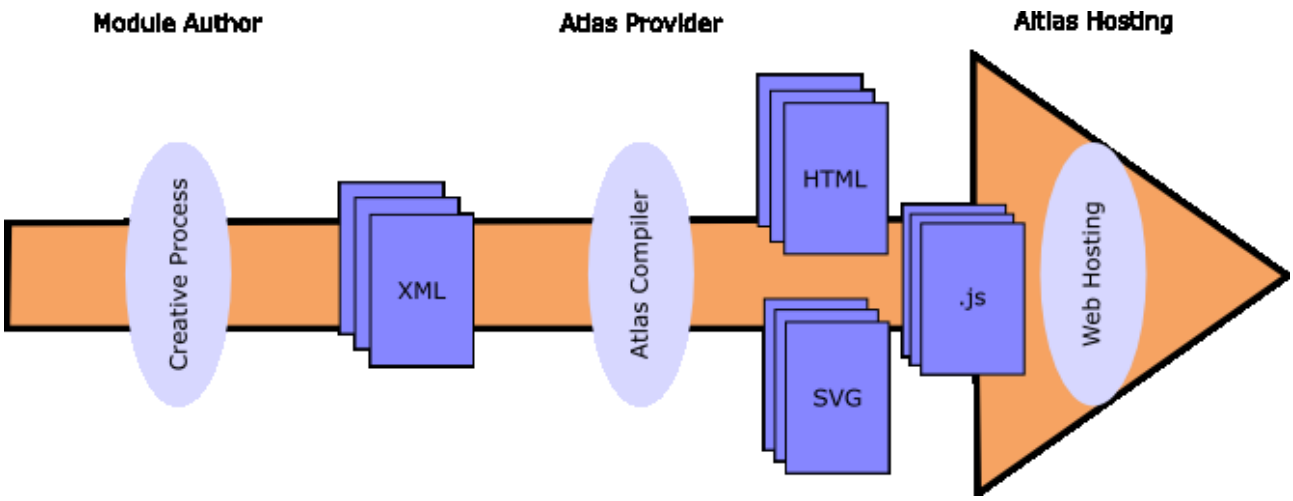

Figure 1.1. Atlas Framework Perpectives

The previous figure shows the three perspectives:

1. **Module Author**: The module author is left with the responsibility of creating XML documents that describe an atlas module. This document contains all the text that the author wants to publish. It also specifies the maps that should be displayed. Finally, it contains mark up elements that link features on the map to information found in the text.
2. **Atlas Provider**: Responsible for transforming the work of the authors into a set of packages that can be published on the web. The atlas provider makes decisions about the organization of the pages, the look and feel of the pages, the interaction level between the map and the text. The atlas provider produces a set of web files, including HTML, SVG, Javascript, and Servlets that can be deployed on a web site.
3. **Atlas Hosting**: In our model, this is the last step. The activity of atlas hosting entails making available all of the atlas provider packages with all the supporting applications and databases.

The Nunaliit Cybercartographic Atlas Framework is a project that helps the user with the first two perspectives: that of the module author and of the atlas provider. The project strives to take a module written by an author and create an atlas-like web page representing it. This way, the author is provided with early feedback as he/she progresses through the process of module development.

Atlas Modules' Inner Workings

The rendering of atlas modules in Nunaliit-SDK is based on DHTML (Dynamic HTML), JavaScript (also known as ECMAScript), and SVG. This provides the end users of the atlas a highly interactive experience while using a web browser. This kind of solution is known as "zero footprint", since the users install nothing specific to the application on their computers.

Therefore, the application runs solely on the end user's computer, within the browser, with supporting live data being sent to it by the server. Internally, the application saves all incoming data from the server in a model. It also creates a number of visual features that represent the data received. The user can interact with the visual features and affect the state of the model, which is then reflected in other visual features. This process provides the interactive elements of the module.

The paradigm of separating data and visual contraptions is known as the Model-View-Controller, or MVC,

where data is stored in a model, experienced via a viewer, and modified through a controller. In the atlas framework, we refer to viewers and controllers as "widgets" since most visual contraptions have the dual role of both viewer and controller. Also, since some of the widgets are providing sonification of data (basically, using sound to display or convey data), the inclusive term "sound viewer" would be too confusing.

2. Getting Started

This page last changed on Oct 20, 2006 by ahayes.

Requirements

Java

To successfully use this project, you need to have Java installed. Download and install the latest version of the Java Runtime Environment (JRE) from http://java.sun.com

Java2TM Standard Edition 5.0 is the version in use as of this writing.

Supported Web Browser

To view the generated atlas, you need a web browser that natively renders XHTML and SVG 1.1 in mixed namespaces (as in, one that does it directly without the use of a plug-in). As of this writing, the only supported web browser is Mozilla Firefox 1.5 or newer. You can obtain this free and open source browser at the Mozilla site at: http://www.mozilla.com/

It is also important to note that as of this writing, **Microsoft Internet Explorer is not able to render the atlas correctly**.

Optional Tools

XML Editor

Nunaliit is based on XML technology and authors using the nunaliit-sdk tools are required to edit XML files. The use of an XML editor is highly recommended; however, any plain text editor (don't use a word processor!) will do in a pinch. For Windows, Altova offers a free XML editor called XMLSpy. It can be obtained at http://www.altova.com. For Apple OS X, Bare Bones Software offers a great text editor that supports XML syntax highlighting at http://www.barebones.com/products/textwrangler/. Another excellent option is to simply use Eclipse as described below.

This project is developed using Eclipse. Therefore, the Eclipse IDE was used to create and maintain all XML files found in the project. If you feel more adventurous or already have Eclipse installed, you can use it to edit all XML files. The Eclipse IDE can be obtained at http://www.eclipse.org. XMLBuddy is an Eclipse plug-in which helps in the editing of XML files. A free version of XMLBuddy is available at http://xmlbuddy.com/.

Installing Nunaliit-SDK

You can obtain the latest version of the nunaliit-sdk at the official download page. Unzip the content of

the nunaliit-sdk-X.X.zip into a directory of your choice (where the Xs are version numbers). For example purposes, let's assume that the atlas framework is decompressed in c:\nunaliit-sdk-x.x.

To try it out, open a command line window and go to the directory you chose:

```
> c:
> cd \nunaliit-sdk-x.x
```

Then, generate the atlas from the modules:

```
C:\nunaliit-sdk-x.x> gen
```

Finally, look at the resulting atlas. To do so, start your web browser and open the document index.html. Follow the link titled "Jump to atlas pages".

Installing Nunaliit-Server

The nunaliit-server is not a compulsory component when a user first encounters the Nunaliit project. So, if you are an early user.

Producing a first atlas module

When it comes to atlas modules, it is always easier to start from an already made example and modify it to suit one's needs. To this end, a number of simple modules are provided with the SDK and can be found under the **modules** directory.

When adding a new module to the atlas, the following steps should be followed:

1. Create a new module file (or copy one already made and modify it);
2. Add an entry to the table of contents file (table_of_contents.xml);
3. Regenerate the atlas using the command line tool; and,
4. View the modifications using your browser.

Create a new module file

As described above, it is easier to copy and modify an existing module, starting from a module that already contains most of the behaviour you are attempting to achieve. Then, you must edit the module and modify it to suit your content. To edit the module, you require an editor. Since the module files are written in XML, an XML editor is ideal. However, since XML is stored in a plain text file, any text editor is sufficient. More information on XML editors can be found in the section called XML Editors.

Adding an entry to the table of contents file?

The stock table of contents file provided with the SDK is located at:

c:\nunaliit-sdk-x.x\modules\table_of_contents.xml

Using an XML editor, edit the table of contents file. The content of this file is structured to look like:

```
<?xml version="1.0" encoding="UTF-8"?>
<toc
xmlns="http://schemas.gcrc.carleton.ca/nunaliit/1.0/table_of_contents"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://schemas.gcrc.carleton.ca/nunaliit/1.0/table_of_contents
../schemas/1.0/table_of_contents.xsd">
    <level name="Welcome">
        <level module="intro.xml"/>
        <level module="hello/hello.xml"/>
        <level module="styling/styling.xml"/>
        <level module="topics/topics.xml"/>
    </level>
    <level module="model/model.xml">
        <level module="model/attributes/attributes.xml"/>
        <level module="model/dynamic/dynamic.xml"/>
        <level module="model/activate-on/activate-on.xml"/>
    </level>
    <level name="Maps" module="map/intro.xml">
        <level module="map/basic/basic.xml"/>
        <level module="map/text/text.xml"/>
        <level module="map/sound/sound.xml"/>
    </level>
    <level module="images/basic/basic.xml">
        <level module="images/animation/animation.xml"/>
        <level module="images/polygons/polygons.xml"/>
        <level module="images/active/active.xml"/>
    </level>
</toc>
```

A new <level> tag is required to include a new module. One attribute, called "module", is required to specify the location of the module file. Another attribute, which is optional and called "name", can be used to override the name that the module is referred to in the table of contents. For example, let's say a new module called "Climate" had a module file located in the modules/climate directory and was called climate_mod.xml, then the table of contents file should be modified to look this way:

```
<?xml version="1.0" encoding="UTF-8"?>
<toc
    xmlns="http://schemas.gcrc.carleton.ca/nunaliit/1.0/table_of_contents"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://schemas.gcrc.carleton.ca/nunaliit/1.0/table_of_contents
../schemas/1.0/table_of_contents.xsd">

    <level name="Welcome">
        <level module="intro.xml"/>
        <level module="hello/hello.xml"/>
        <level module="styling/styling.xml"/>
        <level module="topics/topics.xml"/>
    </level>
    <level module="model/model.xml">
        <level module="model/attributes/attributes.xml"/>
        <level module="model/dynamic/dynamic.xml"/>
        <level module="model/activate-on/activate-on.xml"/>
    </level>
    <level name="Maps" module="map/intro.xml">
        <level module="map/basic/basic.xml"/>
        <level module="map/text/text.xml"/>
        <level module="map/sound/sound.xml"/>
    </level>
    <level module="images/basic/basic.xml">
        <level module="images/animation/animation.xml"/>
        <level module="images/polygons/polygons.xml"/>
        <level module="images/active/active.xml"/>
    </level>
    <level name="Climate" module="climate/climate_mod.xml"/>
```

```
</toc>
```

Generate the atlas using the command line tool

A compiler is required to transform the module files into an atlas that can be viewed using a web browser. The process of using the compiler to create the atlas is known as generating the atlas. All components making up the compiler are available to users using the nunaliit-sdk project and the following paragraphs explain the steps required to generate the atlas.

The tools required to generate the atlas can be accessed via a command prompt. At the command prompt, type:

```
C:\nunaliit-sdk-x.x> gen
```

You should see an output like this:

```
C:\nunaliit-sdk-x.x>java \-classpath tools\apache-ant-1.6.5\lib\ant-launcher.jar
\-Dant.home=tools\apache-ant-1.6.5 org.apache.tools.ant.launch.Launcher build-atlas
Unable to locate tools.jar. Expected to find it in C:\Program
Files\Java\jre1.5.0_06\lib\tools.jar
Buildfile: build.xml

build-atlas:
    [NUNALIIT:echo] Building modules from module files
    [NUNALIIT:mkdir] Created dir: C:\nunaliit-sdk-x.x\cache

build-custom-routines:
build:
gen-xsl:
check-xsl:
generate-xsl:
    [NUNALIIT:java] Generate main script
build-xsl:
gen-xsl:
check-xsl:
generate-xsl:
    [NUNALIIT:java] Generate frame script
    [NUNALIIT:java] Insert scripts
    [NUNALIIT:java] Add template
    [NUNALIIT:java] Intra module navigation
    [NUNALIIT:java] Inter module navigation
build-xsl:
gen-xsl:
check-xsl:
generate-xsl:
    [NUNALIIT:java] Generate volumeBar script
build-xsl:
gen-xsl:
check-xsl:
generate-xsl:
build-xsl:
gen-xsl:
check-xsl:
generate-xsl:
    [NUNALIIT:java] Generate intra_nav script
build-xsl:
gen-xsl:
check-xsl:
generate-xsl:
build-xsl:
build-sound:
check-applet:
build-applet:
build:
```

```
build-atlas:
    [NUNALIIT:copy] Copying 8 files to C:\nunaliit-sdk-x.x\atlas
    [NUNALIIT:copy] Copying 24 files to C:\nunaliit-sdk-x.x\atlas\lib
    [NUNALIIT:java] Generating intro.xml.html
    [NUNALIIT:java] Generating intro.xml.svg
    [NUNALIIT:java] Generating hello/hello.xml.html
    [NUNALIIT:java] Generating hello/hello.xml.svg
    [NUNALIIT:java] Generating styling/styling.xml.html
    [NUNALIIT:java] Generating styling/styling.xml.svg
    [NUNALIIT:java] Generating styling/styling.xml.N65581.html
    [NUNALIIT:java] Generating topics/topics.xml.html
    [NUNALIIT:java] Generating topics/topics.xml.svg
    [NUNALIIT:java] Generating model/model.xml.html
    [NUNALIIT:java] Generating model/model.xml.svg
    [NUNALIIT:java] Generating model/attributes/attributes.xml.html
    [NUNALIIT:java] Generating model/attributes/attributes.xml.svg
    [NUNALIIT:java] Generating model/dynamic/dynamic.xml.html
    [NUNALIIT:java] Generating model/dynamic/dynamic.xml.svg
    [NUNALIIT:java] Generating model/activate-on/activate-on.xml.html
    [NUNALIIT:java] Generating model/activate-on/activate-on.xml.svg
    [NUNALIIT:java] Generating map/intro.xml.html
    [NUNALIIT:java] Generating map/intro.xml.svg
    [NUNALIIT:java] Generating map/basic/basic.xml.html
    [NUNALIIT:java] Generating map/basic/basic.xml.svg
    [NUNALIIT:java] Generating map/text/text.xml.html
    [NUNALIIT:java] Generating map/text/text.xml.svg
    [NUNALIIT:java] Generating map/sound/sound.xml.html
    [NUNALIIT:java] Generating map/sound/sound.xml.svg
    [NUNALIIT:java] Generating images/basic/basic.xml.html
    [NUNALIIT:java] Generating images/basic/basic.xml.svg
    [NUNALIIT:java] Generating images/animation/animation.xml.html
    [NUNALIIT:java] Generating images/animation/animation.xml.svg
    [NUNALIIT:java] Generating images/polygons/polygons.xml.html
    [NUNALIIT:java] Generating images/polygons/polygons.xml.svg
    [NUNALIIT:java] Generating images/active/active.xml.html
    [NUNALIIT:java] Generating images/active/active.xml.svg

default:
    [NUNALIIT:copy] Copying 1 file to C:\nunaliit-sdk-x.x\atlas\map\basic
    [NUNALIIT:copy] Copying 1 file to C:\nunaliit-sdk-x.x\atlas\map\sound

build:

default:

BUILD SUCCESSFUL
Total time: 19 seconds

C:\nunaliit-sdk-x.x>
```

If the build is not successful, then use the error message to determine what went wrong and fix the problem before trying again.

Viewing the new module

To view the generated atlas, you require a supported web browser. More information about supported web browsers is available in the Requirements section.

After starting a supported web browser, select the file index.html at the root directory of the nunaliit-sdk project. From there, there is a link to the atlas pages. If a custom target directory was specified, then open the **index.html** file located in the target directory.

3. Model

This page last changed on Oct 20, 2006 by ahayes.

Model

The model is the component of a rendered atlas where all the data is stored. This is where relationship in data can be established. It is also the area where most interactivity between end-user and data is implemented. It is called 'model' from the Model-View-Controller (MVC) popular software pattern where data is kept seperate from its various perspectives.

In an atlas module, there is a single model where data from all layers are stored. The model itself is made up of various smaller components. The following figure shows a hierarchical tree that represents how the model structures its data. There are four levels that make up the model tree:

1. **Model**. There is one object that represents the complete data model. The model is composed of multiple layers.
2. **Layers**. There is one layer object for each layer defined in data sources within a module document. The name of each layer is the one assigned in the module document, and each layer can be accessed by any widget in the module via its name. A layer is composed of a set of features. A layer can be broadly thought of as a table in a database.
3. **Features**. Features are the main organization unit in the model. They are made up of atoms, which contribute properties to a feature. The way features group atoms and the way features are grouped within layers is dependent on the data source defined in the module document.
4. **Atoms**. An atom contains a list of properties that can be assigned to one or multiple features. The supported features can be located in separate layers. An atom does not change after it is created. It represents one unit of knowledge/data that can be broadly viewed as a row in a database.

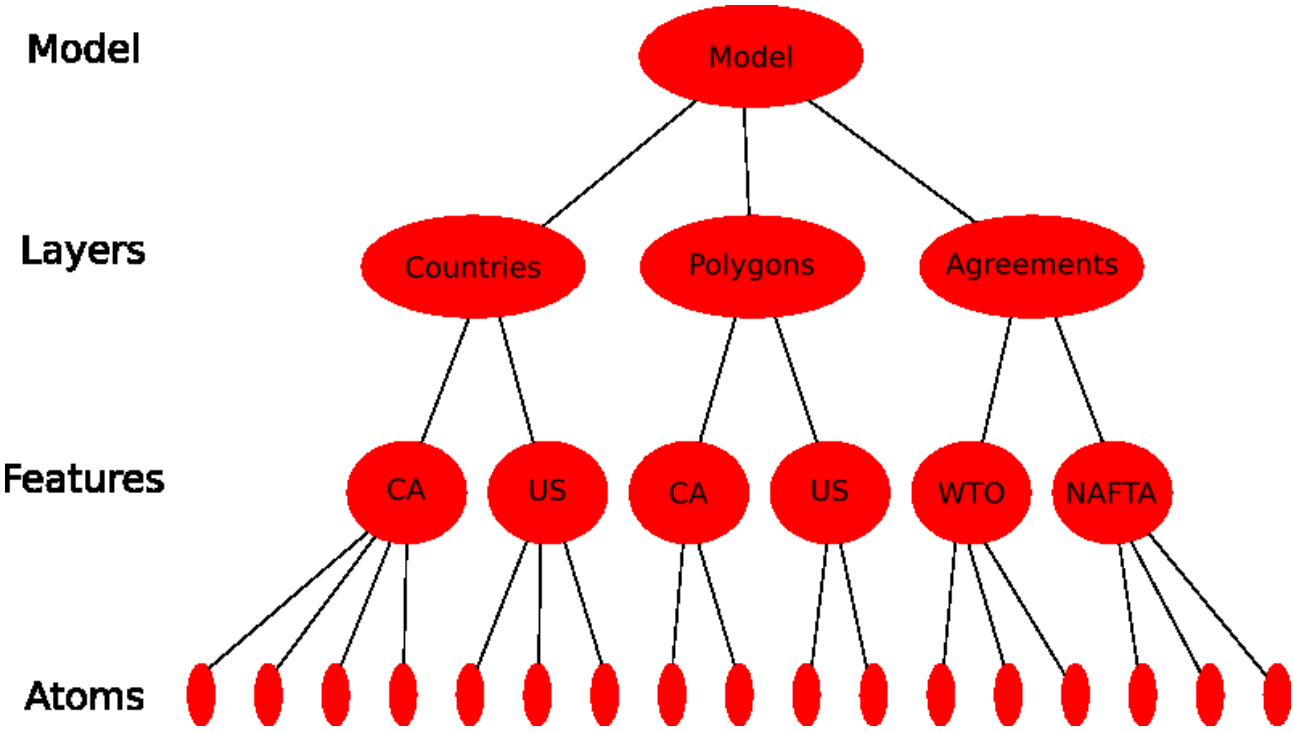

Figure 1.2. Atlas Module Model Tree

The model also dictates the contracts between it, the controllers, the viewers and widgets. It defines an interface where controllers and widgets can modify the model. It also defines a set of events that viewers and widgets can register for in order to be informed of changes in the model. These are the mechanisms that link the underlying model with the entities that the end-user interacts with.

From another perspective, the model can be looked at as an aggregation of a large set of data obtained from databases. Databases are abstracted into tables, and tables are composed of columns and rows. Each element of a database can be selected through a row/column pair identifiers. This simplification overlooks the majority of functions offered from a database, but serves well at illustrating the relationship between atoms and features in the model. An atom contains all the elements of a row obtained from a table in a database. It is a set of tuples where the column name is associated with the approriate element from the row.

Continuing with this perspective, a layer contains all the rows from a database table (in some circumstances, rows from multiple database tables). The information from all the rows are contained within atoms, that are accessible via features. Where operations are used in databases to obtain set of rows, features are employed in the model to associate atoms together. Therefore, within the model, it is possible to create a new layer based on the atoms from a previous layer where the features organizes the atoms in a completely different way. This approach keeps the bandwidth required to load multiple layers from tables to a minimum.

Layers

In a model, there are multiple layers. Each layer is accessible by name from the model. Each layer is uniquely named within the context of the model.

A layer is composed of multiple features. Each feature is accessible by name from a layer.

A layer is generally created from a data source. There are multiple data sources available in Nunaliit. One of the data source is a Web Feature Service (WFS) which is service where geospatial data can be obtained via a standard protocol. There are other data sources as well. The common point between all data sources is that they offer data with the same granularity as a database table. A database table is composed of multiple rows and coloumns, where each element can be accessed via a coordinate of row identifier and column name. All the information of a database table ends up being hosted within a layer.

In an atlas module, widgets are usually associated with a layer. A timeline, a map canvas or a graph is associated with a layer, where the individual features are displayed.

Features

In general, a layer has multiple features. Each feature is accessible by name from the layer (this name is also referred to as an identifier, or 'id' for short). Each feature is uniquely named within the context of a layer. Therefore, it is possible to have multiple features named similarly, as long as they are located on different layers.

Features have a collection of attributes. An attribute is a value that can be accessed via a name. These attributes are accessible by views and controllers. Therefore, feature attributes are usually 'experienced'

by an end-user, since these are the values displayed in the interface. For example, geometries made available via a feature are drawn on a map canvas. In this example, a map canvas would show all features of a layer seperately, each according to the geometries specified in the respective feature.

It stands to reason that the main level of interaction happens at the feature level. Widgets are created based on a layer, however the discreet units shown by a widget follows the number of features found in the layer. If a timeline is opened on a layer, the number of choices given in the timeline is directly proportional to the number of features found in the layer. Also, the labels associated with each choice of the timeline is drawn from the feature attributes.

Attributes are implemented based on variables and atoms, using the following algorithm:

1. if a variable exists with a name equivalent to the requested attribute, then the value of the variable is returned;
2. if an atom exists with a property name equivalent to the requested attribute, then the value of the first property found is returned; or,
3. null is returned (null, zero, empty string, etc.)

A variable is a place holder for a value which is associated with a feature based on a name. It can be modified at run-time. A variable can be set in the module file and modified using controllers installed on layers and features. There are no restriction on the naming of a variable and therefore, it is possible for a variable named identically to an atom property to hide this atom property.

As is discussed below, an atom does not changed once loaded into a module. An atom provides values for feature attributes, but the constance of an atom means that interactivity is not obtained via thiss vehicle.

Therefore, a feature contains a collection of atoms and a collection of variables. Both atoms and variables contribute to the attributes of the feature, and only variables can be modified at run-time.

Atoms

Atoms are named as such because they are an indivisble unit of information within an atlas module. While a feature contains a collection of atoms, an atom is not necessarily associated to a single feature. An atom can be shared between features of different layers. Internally, an atom has a unique identifier (which is unique throughout the model) but it is not visible from a module's perspective.

An atom is composed of a set of tuples, where a property name is associated to a property value. An atom mimicks a row found in a database table where a value is associated with each column. In this example, the property name is the column name. Generally, a relationship between multiple values is implied within a database table row, and the same applies to an atom. However, this implied relationship changes betwen applications.

Variable 'display_state'

All features are automatically augmented with a variable called: 'display_state'. This variable is observed by most widgets to style or otherwise emphasize a feature. There are four states associated with this variable:

1. normal;
2. inFocus;
3. selected; and,
4. inFocusAndSelected.

These states indicates the end-user interaction with a feature. For example, if an end-user moves a mouse over a portion of a widget that is associated with a feature, then the variable 'display_state' is changed from 'normal' to 'inFocus'. When this happens, all widgets that represent this feature updates the view to emphasize that this feature is now in focus.

A module author does not have to be preoccupied by this function since all developed widgets respect this convention.

Layer '_dynamic'

The model defines a special layer, named '_dynamic', with features where dynamic data is found. This data can be used as any data found on other layers. However, this data is populated and maintained by the model itself.

layer	feature	attribute	definition
_dynamic	global	currentSecCounter	Number of seconds elapsed since Janurary 1st, 1970
_dynamic	global	elapsedSecCounter	Number of seconds elapsed since the module was last loaded
_dynamic	global	lYear	Calendar year associated with the local time
_dynamic	global	lMonth	Numerical value that represents the calendar month associated with the local time (1-12)
_dynamic	global	lDay	Numerical value that represents the calendar day associated with the local time (1-31)
_dynamic	global	lHours	Hours portion of the local time expressed in 24-hour clock (0-23)
_dynamic	global	lMinutes	Minutes portion of the local time (0-59)
_dynamic	global	lSeconds	Seconds portion of the local time (0-59)

The values in this layer are updated to the reflect current state once every second.

Layer 'global'

This layer is used internally to control some aspects of the user interface. Module author shall refrain from accessing this layer.

Layer '_topics'

This layer is used internally to maintain information on all topics of an atlas. Module authors should refrain from using this layer.

4. Module Files

This page last changed on Oct 20, 2006 by ahayes.

Modules

Modules are the basic unit of an atlas. A module is generally self-referential and contains all information relating to an idea to be transmitted from an author to a population of users. A module defines all the entities presented and the relation between them. This way, it should include text, maps, images, movies, audio files, etc. It should also define all links between those entities. When rendered, a module presents the user with all the information provided by the author and emphasizes all the relevancy between the artifacts.

The term 'module' is quite abstract and can include many definitions. The concept of 'module' traverses all phases in the Nunaliit project, but may have very different concrete means for various people. For example, an author deals with 'module files', where he/she create and edit computer files that establish the content of an atlas module. At the other extreme, when an atlas is generated from various module files, an end-user experiences a module via one of its many instantiations. When referring to a module that has been generated and is ready for end-user consumption, the term 'module instance' should be used. When referring to the computer files that contains all the instructions to build an atlas module, the term 'module file' shall be employed.

The remaining of this section deals with module files.

Module Files

Module files are XML documents that respects the module schema developed in Nunaliit. The content of a module file describes all text, map, images, sounds and multi-media entities that composed an atlas module. A module file is used by the nunaliit-sdk compiler to generate all files required to view the module via a web browser.

A module is divided into two major section:

1. a section that describes the topics associated with the module; and,
2. a section that specifies all the data (model) used in the module.

Topics are themselves made up of other topics (sub-topics), text and maps. In general, a module file will have a structure as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<module xmlns="http://schemas.gcrc.carleton.ca/nunaliit/1.0/module"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://schemas.gcrc.carleton.ca/nunaliit/1.0/module
../schemas/1.0/module.xsd">

    [NUNALIIT:TOPICS SECTION]

    [NUNALIIT:MODEL SECTION]
```

```
</module>
```

The attributes on the module tag are important, and defined below:

1. xmlns="http://schemas.gcrc.carleton.ca/nunaliit/1.0/module"; describes the default name space used for remainder of the XML document. This name space is reserved for atlas module files defined in Nunaliit.
2. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"; assigns a name space to the "xsi" prefix, used below.
3. xsi:schemaLocation="http://schemas.gcrc.carleton.ca/nunaliit/1.0/module ../schemas/1.0/module.xsd"; specifies where the schema file that defines the Nunaliit module name space can be found. In this example, it is located in a file found through a relative path: ../schemas/1.0/module.xsd. If you change the location of the module file or the location of the schema file, this path must also be changed.

Topics

The topics section is dedicated in providing all the visual and audible elements that are experienced by a user of the atlas module. It deals with text, map, sound and other widgets. It defines the views that are presented to the user. The main unit of composition is a topic.

There is always a root topic to a module. All other topics must be made sub-topics of the root topics. Sub-topics may themselves have sub-topics of their own. A topic is structured as follows:

```
<topic id="*pecial_identifier">
   <title>The Title</title>

   [NUNALIIT:TOPIC ACTIVATION]

   <text>

      [NUNALIIT:TOPIC TEXT]

   </text>
   <map>

      [NUNALIIT:TOPIC MAP]

   </map>

   [NUNALIIT:SUB-TOPICS]

</topic>
```

Explanation of the different details are explained here:

1. **special_identifier**; This is a unique identifier, used across all atlas modules, to refer to this particualr topic. This is optional, and the **id** attribute can be ommited altogether. If used, the value of the identifier should be generated by the 'genid' utility, to ensure uniqueness and correct formatting.
2. **The Title**; This is the title associated with the topic. This is compulsory and must be provided. The content can be any string.
3. [NUNALIIT:TOPIC ACTIVATION]; This is a set of conditions under which the topic should be brought to the user's attention. This is discussed later and is optional.

4. [NUNALIIT:TOPIC TEXT]; This is the text portion of a topic. The text tag is required, but leaving the tag empty generates a topic without any text. The content of a text tag follows a grammar discussed in its own section.
5. [NUNALIIT:TOPIC MAP]; This is the graphical portion of a topic. The map tag is optional. If it is not provided, then a topic inherit the map from its parent topic. If the map tag is provided but left empty, the topic is generated without graphical elements. The content of the map tag follows special rules discussed in its own section.
6. [NUNALIIT:SUB-TOPICS]; This section may contains zero, one or multiple topic tags. Each of these topic tags must adhere to the rules explained here. This establishes a recursive pattern where topics can include sub-topics, which themselves have sub-topics and so on.

[NUNALIIT:TOPIC ACTIVATION]

TBD

[NUNALIIT:TOPIC TEXT]

Text that can be inserted in a <text> tag is rich and can be quite complex. At the top level, the text is a set of block tags that can be one of <h1>, <h2>, <h3> or <para>. Tags <h1> to <h3> are header tags, similar to the ones found in HTML. <para> tags are used to encapsulte paragraphs, similar to <p> tag in HTML. All of the block tags can be popuated with a mix of string and inline tags. Here is the list of inline tags:

1. <emphasis>: This tag is used to make some text in evidence
2. : This tag is used to style the given text with a stronger font
3.
: This tag breaks a line similar to the
 tag in HTML
4. <image>: This tag includes an image
5. <audio>: This tag includes an audio file
6. <movie>: This tag includes a movie file
7. <enumerate>: This tag contains a numbered list. The child tags must be <item>
8. <itemize>: This tag contains an unordered list. The child tags must be <item>
9. <peer>: This tag associates the included text with a feature in the model.
10. <attribute>: This tag inserts text obatined from a feature attribute.
11. <featureList>: This tag inserts a list of peer text elements. Each instance is associated to a feature from a layer.
12. <link>: This tag links to another topic or to an external link.
13. <attach-note>: Reference to a note
14. <quotation>: Captures a quotation
15. <caption>: Captures a caption

All those tags are discussed in greater details in another section.

[NUNALIIT:TOPIC MAP]

The topic map section defines all the visual and auditory widgets that are not directly related to the text. In practice, these widgets live in a pane seperate from the text. The module file language defined by the Nunaliit schema does not impose any layout restriction based on the location or order of the widgets within the map tag. It is possible that some implementations are influenced by the order, however an author should not rely on those side effects since a different atlas compiler might implement a different

layout.

The widgets that can be included in a topic map are:

1. <mapCanvas>: a map based on geometries found in the model
2. <graphoMap>: an interesting map cannvas replacement where multiple variables are explored versus the longitude of a phenomenon
3. <externalSvgWidget>: link to an external SVG file that can be customized and linked into the model's interactivity.
4. <soundBackgroundWidget>: audio widget that plays a background track
5. <soundPlayOnFeatureWidget>: audio widget that plays a track when a feature is in focus or selected
6. <soundGainOnAttributeWidget>: audio widget that adjust the gain of a track when a feature is in focus or selected
7. <selectionBar>: visual widget that selects a feature from a layer
8. <timeLine>: visual widget that updates feature attributes based on conditions

Model Section

The model section contains a collection of instructions to define layers and controllers. The following tags are available:

1. <ogcDataSource>: Controller that loads a layers from a WFS data source
2. <simpleDataSource>: Controller that populates a layer with data found within the tag
3. <externalDataSource>: Controller that reads an external file and loads the data in a layer
4. <layerFromLayer>: Controller that adds all the atoms found in a layer to another layer, but structured in different features
5. <copy>: Controller that mixes atoms from multiple layers into a single layer
6. <join>: Controller that adds all atoms found in a source layer into a target layer where selection is based on a named attribute
7. <link>: Controller that replicate the 'display_status' variable between features of different layers
8. <create-attribute>: Controller that assigns and update one or multiple feature variables based on an equation.

5. Customization

This page last changed on Oct 20, 2006 by ahayes.

Customizations

There exists a number of options when an atlas is compiled by nunaliit-sdk. Among others, there are options to dictate where the modules files are located, to specify where the generated atlas should be directed, what skin is in use, etc. All of those options can be specified in the **build.properties** file. You can find this file in the root directory of the nunaliit-sdk. It is a text file and can be edited using any text editor, such as notepad (Windows), vi (Linux,Unix) or TextEdit (Mac OS). Note that word processors are probably not recommended to edit this file.

When directory and file locations are entered in the build.properties file, they should adhere to the following conventions:

1. always provide the full path; and,
2. always use forward slashes ('/', a la Unix), even if located on a Windows machine.

The following code shows an example of a fictitious option assigned the name of a file on Windows:

```
example.option=c:/temp/file.txt
```

Custom location for module files

By default, the modules compiled by the framework are located in the **modules** directory located where nunaliit-sdk is installed. To use a different location for module files, the property **atlas.src.full** in the build.properties file can be modified, as shown in the following example:

```
atlas.src.full=c:/location-to-module-directory
```

Custom location for generated atlas

By default, the framework generated atlases are located in the **atlas** directory located where nunaliit-sdk is installed. To use a different target location, the property **atlas.dest.full** in the build.properties file can be modified, as shown in the following example:

```
atlas.dest.full=c:/directory-where-atlas-goes
```

Custom file name for table of contents file

Nunaliit-sdk uses a table of contents file (usually called table_of_contents.xml) to indicate which module files are used to create the atlas, as well as the relationship between modules in the table of contents.

This file is located in the same directory as the module files. The property **atlas.src.toc** in the build.properties file can be used to specify a different name for the table of contents file. Do not specify a full path for this option.

```
atlas.src.toc=toc.xml
```

In the previous example, a file named **toc.xml** will be used instead of **table_of_contents.xml**. This file is expected to be located in the same directory as the module files. The location of module files is dictated by the **atlas.src.full** option.

Debugging of Data Model

The modules generated by the nunaliit-sdk contain a fair amount of information organized in units (layers, features, and atoms) to drive the visual and audible representations of those modules. The collection of those information units is known as the data model (or model for short). To browse the model, the build.properties file can be modified so that the generated atlas include a model inspector. To do so, add the following line to the build.properties and regenerate the atlas:

```
atlas.custom.debugModel=1
```

Custom Title

The title of the atlas can be changed by editing the build.properties file. Changing the property named **atlas.custom.title** indicates to the compiler the new desired name. After this modification, a newly generated atlas should reflect the new name. Here is an example where the title of the atlas is changed to "Incredible Atlas"

```
atlas.custom.title=Incredible Atlas
```

Note that while using custom skins, the title option might not be available.

Custom Skin

Generated atlases are "skinnable". This means that the lokk-and-feel of an atlas can be changed, within reasonable extents. Currently, only one skin is distributed with Nunaliit SDK, which is known as the "generic" skin. The generic skin is used by the SDK compiler as a default.

However, it is possible to develop a new skin for the atlas where the colours and dimension are drastically different from what is found in the generic skin. To do so, save the file "generic.svg" found under the webCompiler/skin directory with a new name. For example, let's assume that the name of the new file is "my_skin.svg". Once the new skin file is ready, the property called "skin.name" in the build.properties file should be modified to reflect the name of the skin. The name of the skin is the skin file name without the extension (see example below). After these changes are applied, new generation
of the atlas produces pages that use the new skin.

```
skin.name=my_skin
```

Remember, when editing a skin file, great care must be taken to leave the XML identifiers as they are, since the compiler uses them to create the proper routines.

If it is not practical to have a custom skin file distributed with a copy of Nunaliit (located in the webCompiler/skins directory), it is possible to configure the compiler to pick a skin file from a different location. This might be desirable if it was deemed more appropriate to locate the skin file with the modules. In that case, the same procedures as outlined above must be followed with an extra step, where the property "skin.file" is added to "build.properties" to indicate the full path to the skin file. For example:

```
skin.name=my_skin
skin.file=c:/my_modules/my_skin.svg
```

6. References

This page last changed on Oct 20, 2006 by ahayes.

Frequently Asked Questions (FAQ)

This page last changed on Oct 20, 2006 by ahayes.

Where can I get Nunaliit SDK?

A link to the Nunaliit project and the latest nunaliit-sdk will be available at: http://gcrc.carleton.ca/. Special care should be paid to the version you obtain.

How can I use downloaded modules with the framework?

If you have obtained modules that are not included with the standard deployment of nunaliit-sdk, you can redirect the framework to produce an atlas based on those modules instead of the ones included with the distribution. Use the following steps to redirect nunaliit-sdk to the external modules:

1. Edit the file titled build.properties in the top directory of nunaliit-sdk. If nunaliit-sdk was installed in c:\nunaliit-sdk-x.x, then the file in question would be c:\nunaliit-sdk-x.x\build.properties
2. Add a line to the file that looks like: **atlas.src.full=c:/myModules** Instead of c:/myModules, use the full path to the modules that should be generated.
3. Regenerate the atlas.

What do I need to install before I can use the Nunaliit SDK? (Requirements)

The nunaliit-sdk framework requires two components:

1. An up-to-date version of Java installed on the platform (see How do I install Java?)
2. Mozilla Firefox version 1.5 or better. You can obtain Firefox at http://www.mozilla.com/

How do I install Java?

To successfully use nunaliit-sdk, you need to have Java installed. You have the choice between installing the Java2TM Standard Edition Runtime Environment (J2SE 5.0 JRE) or the Java Developer's Kit (J2SE 5.0 JDK). Nunaliit modules can be developed with the JRE alone; however, if you decide to use Eclipse as your editor or you want to help develop Nunaliit, then you'll need the JDK.
On Windows or Linux, download and install the latest version of Java JDK from http://java.sun.com. As of this writing, JDK 1.5 Update 6 and 7 have been tested on Windows. Choose the version appropriate for your platform and follow the instructions. If, during the installation, you are offered to install the JRE, accept and proceed with the installation of the JRE as well.
On the Mac, J2SE 5.0 Release 4 (as provided by Software Updates on Apple OS X Tiger) has been tested.
On a Windows machine, once the JDK has been installed, set the environment variable JAVA_HOME to the root directory where the JDK is installed: (These steps assume that the JDK was installed in c:\Program Files\Java\jdk1.5.0_06)

1. Open control panel: Start -> Control Panel
2. Open System panel: Double-click system icon (if you do not see a System icon, you might have to ask for a classic view of the control panel)
3. Select Advanced tab

4. Press the button titled: Environment Variables
5. Two panes are displayed with environment variables, associated with buttons. Use the top pane, meant for user variables (not system variables).
6. Browse pane and look for a variable named JAVA_HOME. If found, select it, press the button Edit and skip next step.
7. If the variable JAVA_HOME was not found, then create one by pressing the button titled **New**. At the dialog box, enter JAVA_HOME for variable name.
8. Enter the path to the JDK root directory as variable value. For example, c:\Program Files\Java\jdk1.5.0_06
9. Press OK to enter the variable.
10. Press OK to close the System panel.

How do I install Eclipse?

Eclipse is an Integrated Development Environment (IDE). It is an application where multiple tools used by developpers to build projects are combined. Eclipse requires that a JDK be installed on the platform. If not already done, follow the steps found in **How do I install Java?**. Use the following steps to install Eclipse:

1. Download the appropriate flavour of Eclipse for your platform from http://www.eclipse.org/. You will obtain a zip file roughly 110 MB in size
2. Unzip Eclipse in a directory (assume c:\Program Files)
3. In the top directory (c:\Program Files\eclipse), you will find an application called Eclipse. Start the application. On Windows, double-click.

The last step represents how to start Eclipse on subsequent uses. We recommend reading the tutorial. Once done, close the tutorial window. You need two plug-ins to make Eclipse useful for the purpose of nunaliit-sdk:

1. Subclipse (How do I install Subclipse in Eclipse?)
2. XmlBuddy (How do I install XMLBuddy in Eclipse?)

How do I install Subclipse in Eclipse?

Subclipse is a plug-in for Eclipse. It allows Eclipse to communicate directly with a Subversion repository. Subversion is the source control system used to develop the nunaliit-sdk framework and the GCRC modules (antarctic and trade). To install Subclipse:

1. Start Eclipse
2. Help -> Software Updates -> Find and Install...
3. Select "Search for new features to install" and press "Next"
4. Press New Remote Site... button, name is Subclipse and URL is http://subclipse.tigris.org/update_1.0.x Press OK
5. Select Subclipse and press Next button
6. Select all products and accept installation
7. Accept restarting of workspace, if requested

Once Subclipse (version 1.0.3 or greater) is installed, you have all reqquired tools to participate in the development of the nunaliit-sdk.

How do I install XMLBuddy in Eclipse?

XMLBuddy is an Eclipse plug-in that helps with editing XML files. Use the following steps to install XMLBuddy:

1. Close the Eclipse IDE application
2. Follow download instructions for XmlBuddy at http://xmlbuddy.com/. Use the free version appropriate to your version of Eclipse. You will end up with a ZIP file.
3. Open the ZIP file and extract all contents to the **plugins** directory of Eclipse. Assuming that Eclipse is installed in c:\Program Files\eclipse, the target directory would be c:\Program Files\eclipse\plugins
4. Once extracted successfully, you should observe a directory called com.objfac.xmleditor_2.0.72 (version numbers may differ) in the plugins directory.
5. Restart Eclipse

This page last changed on Oct 20, 2006 by ahayes.

Licenses

Nunaliit License

In order to facilitate the widest possible adoption of the concepts and code developed in a publicly funded research project, the Nunaliit Cybercartographic Atlas Framework has been placed under the "New BSD License". The text of this license follows:

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the Geomatics and Cartographic Research Centre, Carleton University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF SUCH DAMAGE.

Third Party Licenses

Many products are used and potentially distributed in the deployment of various Nunaliit projects. Great effort is taken within the Nuanliit project to respect all licenses that govern all imported projects and products to help in the deployment of on-line atlases. Users of the Nuanliit projects must comply with those licenses as well and should become familiar with them. The next sections discuss each product distributed with Nunaliit and a brief explanation of the associated license.

Apache ANT

Apache ANT (http://ant.apache.org) is covered by the Apache License Version 2.0 and published in January 2004.

ANT is used for most of the scripting required in the project. Please refer to the directory tools/apache-ant-1.6.5 for more details on the license.

Docbook XSD and XSL

Docbook is used for generating various documentation in the project. Two projects from the Docbook community are used: the Docbook schemas and the Docbook XSL project. More information on Docbook can be obtained at the following links:

http://docbook.org/

http://docbook.org/xsd/4.4/index.html

http://docbook.sourceforge.net/

ECMAScript Helper Functions

The ECMAScript Helper Functions employed by the generated atlas modules in Nunaliit were developed by Andreas Neumann and are licensed under the GNU Lesser General Public License version 2.1. Many thanks to Andreas for his inspiration through various projects he has developed.

Apache FOP

Apache FOP (http://xmlgraphics.apache.org/fop/) is covered under the The Apache Software License, Version 1.1.

FOP (formatting object processor) is used with the Docbook projects to create the PDF versions of the project's documentation. Please refer to the directory tools/fop-0.20.5 for more details on the license.

JLayer

JLayer is developed by Javazoom and is licensed under the GNU Library General Public License version 2. This is a Java library that decodes and plays MP3 bitstreams. It is used within the sound applet of Nunaliit to process MP3 audio sources.

JOrbis

JOrbis is developed by JCraft and licensed under GNU Library General Public License version 2. This is a Java library that accepts Ogg Vorbis bitstreams and decodes them to raw PCM. It is used within the

sound applet of Nunaliit to process Ogg Vorbis audio sources (via the Javazoom Ogg Vorbis SPI interface - see below).

Jsolait

Jsolait (or JavaScript O Lait) is developed by Jan-Klaas Kollhof and governed under the GNU Lesser General Public License version 2.1. Jsolait is used in the produced atlas modules to manage some of the Javascript code. This is an excellent library.

Saxon

Nunaliit uses Saxon, The Saxon XSLT and XQuery Processor from Saxonica Limited and available at http://www.saxonica.com/. We are grateful to Michael Kay, the initial developer of the original code; the web would not be the same without him.

Two open source versions are distributed with Nunaliit: 6.5.3 and B8-3. Both versions are subject to the license Mozilla Public License Version 1.0

Saxon is a XSLT processor. It transforms XML documents into useful computer artifacts. Version 6 supports the first version of the XSLT language. Version B8 supports version 2.0 of the XSLT specifications.

Tritonus

Tritonus: Open Source Java Sound is copyrighted to Matthias Pfisterer and licensed under GNU Library General Public License version 2. It provides an implementation of the Java Sound API. The audio decoders used by Nunaliit to decode MP3 and Ogg Vorbis audio (JOrbis, Ogg Vorbis SPI, and JLayer) are dependent on the Tritonus libraries.

Ogg Vorbis SPI

VorbisSPI from Javazoom is a Java Service Provider Interface that adds OGG Vorbis audio format support to Java platform. It supports icecast streaming. It is based on JOrbis Java libraries and the Tritonus Java Sound libraries. The VorbisSPI package is licensed under the GNU Library General Public License version 2. Nunaliit uses Ogg Vorbis SPI to decode and convert Ogg Vorbis encoded audio sources.

XS3P

Xs3p is a schema documentation generator written as an XSLT. It is used to generate documentation of the schemas in Nunaliit. Xs3p is licensed under the DSTC Public License (DPL) Version 1.1. The copyright holder is the University of Queensland in Australia. Please refer to the directory tools/xs3p for more details on the license.

This page last changed on Nov 27, 2006 by jpfiset.

Release Notes

Release 1.0.5

Changes since last revision:

- Fix a problem where the compiler fails to create an atlas with an error: "XTRE1500: Cannot read a document that was written during the same transformation:" is reported
- Support for path data. In a <mapCanvas> tag, a layer can be added to paint path data (linear strings) on the canvas. Use the <layer> tag with an attribute named "path" equal to "solid". For example:

```
<mapCanvas id="svgmap1">
       <extent minX="-73.985" minY="40.775" maxX="-73.975" maxY="40.785"/>
       <layer name="paths" path="solid" colour="#000000" size="0.0001"/>
</mapCanvas>
```

- Added ability to choose a geometry attribute other than "topp:the_geom" when drawing a canvas. When specifying a layer in a map canvas, use the "geometryAttribute" to specify the attribute to be used as geometry. For example:

```
<layer name="paths" path="solid" colour="#000000" size="0.0001"
geometryAttribute="tiger:the_geom"/>
```

Release 1.0.4

Changes since last revision:

- Include documentation found in Confluence with SDK releases.
- External HTML pages are available in the map tag. They use the same syntax as External SVG pages.
- Added a servlet project that implements a proxy
- Migrated to using Saxon B8.8j

Release 1.0.3

Changes since last revision:

- Added a tag to text element: *<featureList>*.
  This tag creates a dynamic list of all features in a layer. It is similar to combining a list of *<peer>* and *<attribute>* tags. The advantage of this tag, over the latter less convenient approach, is that

the list reflect all the features currently in the layer, in a dynamic way.

- Added a new processing tag to the external SVG widget:

```
<atlas-text name="attributeName" feature="featureName" layer="layerName"/>
```

This processing instruction informs the widget that the content of the parent node should be replaced by the text found in the attribute associated with the name, feature and layer.

Release 1.0.2

Changes since last revision:

- Make text pane optional in the generated atlas module. To do so, one must provide an empty text tag (<*text/*>). The generated module will then contain a module where artifacts, other than text, occupy the full view port.

- Added ability to link between topics, accross multiple modules. Linking to a topics is a two steps process. First, one must assign a unique identifier to a topic. Second, a topic link tag must be used in the text that includes the unique identifier.
  To obtain a unique topic identifier, use the utility program called 'genId'. It is located in the same directory as the command 'gen' to create the atlas.
  > genId
  Generated id: p31fWtc3YIto92kn2k7JupIq1xE
  The unique identifier should be used only once and is assigned to a topic via the XML identifier, as shown in this example:

```
<topic id="p31fWtc3YIto92kn2k7JupIq1xE"> ...
```

The second step, linking to a topic from the text, amounts to adding the tag <topicLink> to the text, as demonstrated in the next example:

```
<topic id="4HwbqsRv5tmfVoPSCfIWZOFFzao">
       <title>Topic Links</title>
       <text>
               <h1>A Topic</h1>
               <para>
                       This module demonstrates the use of topic link, within a module and
between modules.
                       <topicLink ref="p31fWtc3YIto92kn2k7JupIq1xE"/>
               </para>
       </text>
```

- Added the ability to import an external SVG file as a map. Within the <map> tag, an author can use the <externalSvgWidget> tag, as shown below:

```
<map>
       <externalSvgWidget src="test.svg"/>
</map>
```

In the simplest form, importing an external SVG file is equivalent to displaying a static image. However, it is possible to augment the SVG file with processing instructions that links the content to the atlas model. The following is a list of supported processing instructions:

```
<atlas-animate name="<svgAttrName>" layer="<layerName>" feature="<featureName>"
attribute="<variableName>"/>
```

This processing instruction indicates that the SVG element including the instruction (the parent element) should be modified according to an attribute found in the model. The attribute (variable) found in the model via the specified layer, feature and attribute names is observed. For every change in the value of the observed attribute, the parent element has a SVG attribute modified with the new value.

```
<atlas-controller layer="<layerName>" feature="<fatureName>"/>
```

This processing instruction indicates that the parent SVG element should be observed for mouse events. Mouse events captured for the parent element are rerouted to the model feature indicated via the layer and feature names.

```
<atlas-display layer="<layerName>"
       feature="<featureName>" onNormal="(true|false)"
       onFocus="(true|false)" onSelected="(true|false)"
       onFocusAndSelected="(true|false)"/>
```

This processing instruction indicates that the parent element should be displayed only at certain times, according to behaviour found in the model. A feature is selected via layer and feature names. The selected feature is monitored for the four selection states: normal, in focus, selected and focus while selected. When the selection state of the monitored feature is changed, then the parent SVG element is rendered visible or invisible, according to the states chosen in the processing instructions.
The following example shows how some processing instructions can be combined to achieve interactions between an external SVG file and the atlas framework model:

```
<circle
       cx="100" cy="50" r="40"
       style="stroke:black;stroke-width:2;fill:yellow;">
       <atlas-display layer="testLayer" feature="testFeature" onFocusAndSelected="true"/>
</circle>
<circle
       cx="100" cy="50" r="40"
       style="stroke:black;stroke-width:2;fill:black;">
       <atlas-display layer="testLayer" feature="testFeature" onFocus="true"/>
</circle>
       <circle
       cx="100" cy="50" r="40"
       style="stroke:black;stroke-width:2;fill:blue;">
       <atlas-display layer="testLayer" feature="testFeature" onSelected="true"/>
</circle>
<circle
       cx="100" cy="50" r="40"
       style="stroke:black;stroke-width:2;fill:red;">
       <atlas-display layer="testLayer" feature="testFeature" onNormal="true"/>
</circle>
<circle
       cx="100" cy="50" r="40"
       style="stroke:black;stroke-width:2;fill:white;opacity:0">
       <atlas-controller layer="testLayer" feature="testFeature"/>
```

```
</circle>
```

In this example, 5 identical circles are drawn atop one another. The yellow circle is located on a layer below all others, while the white circle is above. In this scheme, the white circle (the one above) is given an opcity of 0, meaning it is transparent. It is also associated with the processing instruction 'atlas-controller'. This means that when the mouse is entering/exiting/clicking the circle, the appropriate events are routed to the feature named 'testFeature' located on the layer named "testLayer'. These events affect the display state of the feature. The other four circles are associated with processing instructions 'atlas-display'. They are all associated with the same feature as the controller circle (the white one). However, the circles are displayed only at certain values of the display state of the feature. In fact, in the default state (normal), only the red circle is displayed. When the feature is selected, only the blue circle is displayed. Therefore, at any one time, only two circles are displayed with one, the controller, being transparent. The visual effect to the end user is to see a circle changing colour with the display state of a feature.

Release 1.0.1

Changes since last revision:

Renamed the compiler option "atlas.src.nav" to "atlas.src.toc".

Fixed timeline widget where times are sorted according to time values, not label content

Release 1.0

Initial public release. Rename project to Nunaliit.

Selected Data Sources

This page last changed on Oct 20, 2006 by ahayes.

Service Provider	Access URL	Type	Data
Atlas of Canada	http://atlas.gc.ca/cgi-bin/atlas/wms_en?VERSION=1.1.1&request=GetCapabilities	*WMS*	•*Road network (1:2 000 000)* •*Drainage (1:2 000 000)* •*Boundaries*
Land Information Ontario	http://wfs.mnr.gov.on.ca/wfs?request=GetCapabilities&service=WFS	*WFS*	•*Settlements* •*Roads* •*Water*
Municipality of Truro, Nova Scotia	http://142.176.62.108/cgi-bin/mapserv.exe?map=d:\inetpub\wwwroot\maps\TRURO_WMS.map&	*WMS*	•*Streets* •*Parks*
Environment Canada Pacific and Yukon Region	http://excise.pyr.ec.gc.ca/cgi-bin/mapserv.exe?map=/data/mapsurfer/PYRWQMP.map	*WMS*	•*Water Quality*
Canadian Geographical Names Service	http://cgns.nrcan.gc.ca/wfs/wfsbeserv.cgi?request=getCapabilities	*WFS*	•*Place Names*
Meteorological Service of Canada	http://map.ns.ec.gc.ca/stsarmse/map.aspx?service=WMS&request=GetCapabilities	*WMS*	•*Atmospheric (historical and current)*