

Introduction to the CMGUI Zinc API v2.8

Richard Christie, Auckland Bioengineering Institute, University of Auckland, New Zealand

13 October 2011

The CMGUI Zinc Library

The CMGUI Zinc library, developed as part of the CMISS project, is primarily designed for building interactive graphical interfaces to mathematical field models. It provides functionality in several 'layers':

1. Representations of mathematical fields, including finite element, image-based, CAD, plus fields derived by mathematical, image processing and other operators applied to source fields. Fields are created in hierarchical namespaces called 'regions'.
2. Graphics conversions for making 3-D graphical 'renditions' of the fields in each region, built from primitives including points, lines, surfaces, iso-surfaces and streamlines.
3. OpenGL rendering for several UI systems including win32, gtk+, Carbon(Mac), wxWidgets. Also includes interaction such as picking and highlighting.
4. Utilities including non-linear optimisation.

CMGUI is the name of the standalone application built using the library; the combined home page is: www.cmiss.org/cmgui. Note we are transitioning to calling the library the 'CMISS Zinc library' to distinguish it from the CMGUI application.

API Concepts and Standards

ANSI C

The underlying API is presented in ANSI C headers to ease integration into the widest variety of languages.

Not Threadsafe

The API is not threadsafe. Calling an API method in one thread while another API method is in progress on another thread is expected to cause uncertain behaviour or a crash. There are however plans to use multiple threads internally to parallelise graphics generation, which may involve making some APIs thread-safe in controlled situations.

Include Files

Public C API headers are in the 'api' folder of the CMGUI Zinc source tree. Headers are named for the main object or category of objects for which API is provided, so in many cases several related object types will be in the same header. Definitions of handle types and enumerations needed by multiple API headers are in the subfolder 'api/types'; it should not be necessary to include these directly. The latest C API can be viewed on our subversion viewer at:

<https://svn.physiomeproject.org/svn/cmiss/cmgui/trunk/source/api/>

Namespaces

To avoid function name collisions with other libraries, all types and their methods are prefixed by "Cmiss_", all enumerated constants are written in uppercase prefixed by "CMISS_", and all API header file names begin with "cmiss_". After the prefix, C API function names consist of unabbreviated words separated by underscores ('_').

Object/Interface-based

The overall design of the API is based on calling methods on an object or interface. Only the initial 'context' object is created without reference to an existing object:

```
Cmiss_context_id context = Cmiss_context_create("Instance1");
```

All other objects are created or obtained by calling methods on objects/interfaces you have obtained:

```
Cmiss_region_id region = Cmiss_context_get_default_region(context);
```

Here, a Cmiss_region_id is a handle to a Cmiss_region object (see later). All methods on an object start with the type name and take the object as the first argument.

Note that all required state must be passed as function arguments: there are no global singleton objects from which data is obtained.

Object Handles and Clean-Up

All objects are returned by the API as 'handles', and handle types always take the name of the object type with the suffix '_id'. The handle type is currently just a typedef for a pointer, but it is possible future implementations will change this:

```
struct Cmiss_region;  
typedef struct Cmiss_region * Cmiss_region_id;
```

With few exceptions, all handles returned by the API must be destroyed by a type-specific destroy function, which takes the address of the handle and clears it to prevent it being used again:

```
Cmiss_region_destroy(&region);
```

The only exceptions are 'base_cast' functions for derived types which return a temporary handle to the base class for calling methods on it to simulate object-oriented polymorphism.

Excepting deliberate misuse of the API, while you have a valid handle to an object it is guaranteed to remain in existence, or at least in a safe state.

Reference Counting and Object Lifetimes

Internally, most objects are reference counted. Destroying the handle reduces the reference count, and, usually, when there are no external references held the object is destroyed.

Objects which are reference counted expose additional methods to obtain new references (i.e. increment reference count), which clients should use whenever additional handles must be held for an uncertain time by multiple objects:

```
Cmiss_region_id extra_region_handle = Cmiss_region_access(region);
```

Handles to non-reference-counting API objects (those without 'access' methods) should be carefully destroyed *once* when finished with them. It is possible these will be switched to being reference counted in future.

Most main reference counted objects are managed in sets by their owning objects to facilitate change messaging (see below).

Reference counted objects fall into 3 categories according to how their lifetimes are managed:

1. Objects which can be discarded via methods on their owning object, including element (owner: mesh), node (owner: nodeset), graphic (owner: rendition), time_notifier (owner: time_keeper). There may be restrictions, for example, nodes may not be destroyed while in use by elements in the region. Calling the owner's method to discard the object leaves it in a permanently orphaned, but safe state; the only practical action that can be done with them is to destroy any handles held.

2. Objects whose lifetime management is switchable within their owning object, including major objects owned by field_module (field) and graphics_module (graphics_material, graphics_filter, tessellation, spectrum, scene). By default these objects are destroyed when the last external reference to them is removed. This strategy can be changed on a per-object basis by setting their 'IS_MANAGED' flag to 1 which ensures that the object lives indefinitely in its owning object:

```
Cmiss_field_set_attribute_integer(field, CMISS_FIELD_ATTRIBUTE_IS_MANAGED, 1);
```

The default case supports applications which want to do their own management of object lifetimes, and cases when an object is needed only temporarily. The common reason to 'manage' objects is so they remain available for users to select to use: all objects read in from data files (e.g. fields) are automatically managed, otherwise they would never be available for use. Some field expressions require the use of intermediate fields which are often best left as 'unmanaged' so they live or die with the final field.

3. Simply reference counted objects, including region but many support objects which are not expected to be held after use. These are simply destroyed when no references are held to them.

A further exception is that each region always has exactly 3 'master' mesh objects (for 1, 2 and 3 dimensions) and 2 nodesets (for nodes and 'data'). These can be obtained from the region's field module, but new ones cannot be created, nor can they be destroyed. These are limitations of the current data model.

Change Messaging

The library automatically updates graphics when changes are made to the fields and other objects which they are based on. It does this via internal change messages which are sent from their owning object to clients. Normally a change message is sent by every method call which modifies the state of an object, which is very inefficient if you are making many changes. Hence if more than one API is to be called, you should call the begin/end_change API on the owning object. The following example is for making many field changes to a region's field module:

```
Cmiss_field_module_begin_change(field_module);  
// add or modify multiple fields from field_module here  
Cmiss_field_module_end_change(field_module);
```

It is very important that you call the matching end_change for each begin_change. In code capable of throwing exceptions it is best to construct an object which calls begin_change on construction and end_change in its destructor so both are guaranteed to be called.

Region objects have hierarchical begin/end_hierarchical_change methods which prevent change messages for the entire region tree.

API Method Details

All constructors and functions for finding or getting existing objects return a new handle to the object on success, or NULL on failure:

```
Cmiss_field_id field = Cmiss_field_module_find_field_by_name(field_module,
"coordinates");
```

Constructors (“Create” functions) always create the object in a safe, fully-constructed state. The primary reason for a constructor to fail is if arguments are missing or invalid. In most cases a different factory object is used to create new objects, e.g.:

String getter functions return allocated strings which must be freed using a supplied deallocate function:

```
char *name = Cmiss_field_get_name(field);
// use name
Cmiss_deallocate(name);
```

API methods for setting or getting arrays of values require the size of the array to be passed as an argument, and it is up to the caller to provide a memory buffer of sufficient size:

```
int Cmiss_field_assign_real(Cmiss_field_id field, Cmiss_field_cache_id cache, int
number_of_values, const double *values);

int Cmiss_field_evaluate_real(Cmiss_field_id field, Cmiss_field_cache_id cache, int
number_of_values, double *values);
```

Other functions return an integer 'status code' which will be CMISS_OK on success, any other value on failure. We plan to return more useful error codes in future.

The context has a special method to execute any of the legacy 'gfx' command strings from the standalone CMGUI application (but note there is no perl interpreter acting on the string), and these only work after the user interface has been enabled:

```
Cmiss_context_execute_command(context, "gfx define field bob constant 1");
Cmiss_context_execute_command(context, "gfx modify g_element heart lines coordinate
coordinates material default");
```

Some objects have methods taking partial commands, and do not need a user interface enabled:

```
Cmiss_field_module_define_field(field_module, "bob", "constant 1");
Cmiss_rendition_execute_command(rendition, "lines coordinate coordinates material
default");
```

Use of any methods taking these legacy commands is discouraged, but occasionally necessary when direct API for a feature is unavailable. Always prefer direct API to partial legacy commands, and both over the general context execute command (which should be treated as deprecated: it is already unavailable in the Zinc plugin).

The Zinc Plugin

The Zinc plugin allows the library to be used from web pages including rendering into a canvas in the browser window with a custom GUI, all controlled by calling JavaScript bindings to the API. For the security of the client machine, the Zinc plugin omits all methods for writing to local files, and the context 'execute command' method is not available due to the many file writing commands it permits, plus our intention to remove it.

Some boilerplate code is needed to get the plugin up and running in a web page, and also to obtain data files over the web. These can be seen in examples, plus other documentation on the website.

The Zinc JavaScript API is conceptually identical to the C API, but there are some key differences in appearance.

The Javascript bindings presents all API functions as methods on objects i.e. `object.method(args)`, and the method name uses camelCase starting with little letters for the verb. Hence the C call:

```
Cmiss_field_id field = Cmiss_field_module_find_field_by_name(field_module,
"coordinates");
```

becomes, in JavaScript:

```
var field = field_module.findFieldByName("coordinates")
```

Not only is the format much nicer to read, but you don't have to clean-up any object handles as this is automatically done by the system when variables go out of scope, become unused or are reassigned.

Be aware that the Zinc plugin may be lag behind the C API, but it is a straight-forward task to bring missing functionality into the plugin, so do enquire. Also, we currently don't have a good means for communicating what API is available, so it will be necessary to look at the full C API and translate the relevant function to the JavaScript format.

Making an Application

To start your application using the library, you will need to construct the main `Cmiss_context` object using function `Cmiss_context_create`. The context provides access to all the other major objects in the data model, including the root region, and the graphics module. Current safest practice is to ensure that the handle to the `Cmiss_context` is destroyed after all other handles to CMGUI Zinc objects are destroyed.

Model data is stored as fields, which belong to regions. Regions can be arranged into hierarchical tree structures, and provide namespaces for fields and some hierarchical relationships between them. Regions and fields can be read in from data files, including the native EX format and FieldML 0.4. Image fields can be defined from most common image formats. API is provided for creating finite element meshes and defining interpolation with basis functions, and a large number of field types are available to define new fields by applying operators to existing fields.

To create graphics, first we must get a handle to the graphics module by calling `Cmiss_context_get_default_graphics_module`. All objects responsible for graphics generation are accessed from this module, including scenes, renditions, graphic objects, graphics materials, graphics filters, tessellation objects, spectrums.

Within the graphics module, each region can have a graphical representation called a 'rendition' but before handles to each rendition can be created, renditions must be enabled for the region tree by calling `Cmiss_graphics_module_enable_renditions` on the root region. Thereafter you can call `Cmiss_graphics_module_get_rendition` to get a handle to the rendition for a region, and are then able to create graphics for it.

A rendition consists of a list of `Cmiss_graphic` objects, each of which has a type which sets the operation it uses to convert fields into graphics: points (element points, node points, data points, single point), lines, surfaces, iso-surfaces and streamlines. One of the key strengths of the CMGUI Zinc library is that all aspects of graphics conversion are controlled by fields including the coordinates source, data values (for colouring with a spectrum to create 'contour plots'), iso-scalar for iso-surfaces, stream vector for streamlines, glyph orientation and scaling for points. Combining this with the ability to define fields by applying operators on existing fields mean there are few

limitations to what can be visualised.

Currently only limited API is available for constructing and modifying the many settings for each graphic type, so at this time it is usually necessary to pass the data for a graphic using the format from the legacy cmgui 'gfx' commands. For example, to create a point graphic called "bob" which draws the scaled "axes" glyph at the origin in the supplied rendition, use the following:

```
Cmiss_rendition_execute_command(rendition, "point as bob glyph axes size 1.2 centre 0,0,0");
```

Refer to the cmgui website for examples of how to set up graphics with 'gfx' commands.

To visualise the graphics using OpenGL in a window, you must first enable the user interface by calling a variant of `Cmiss_context_enable_user_interface` for the UI widget set in use. This also enables other functionality such as timers and idle updates. The `wxWidgets` variant of the library has additional API for passing an external main loop into it.

After the user interface is enabled, the `scene_viewer_package` can be obtained from the context and used to create a `Cmiss_scene_viewer` object in a drawing canvas for the respective UI layer, currently `wxWidgets`, `gtk`, `win32` or `Carbon`. A scene viewer renders the graphics from a scene; a scene essentially specifies the top region for which graphics are to be output and a graphics filter object controlling which graphics within that region tree are to be shown. The graphics module always has a default scene, and new scenes can also be created and managed. The default graphics filter, used by the default scene, filters according to the visibility flag for each graphic and each rendition.

To get the scene viewer to show all visible graphics within the scene at a default viewing angle, call the `Cmiss_scene_viewer_view_all` function.

Examples

The `simple_axis_viewer` example demonstrates how to make a simple application using the CMGUI Zinc library and API:

http://cmiss.bioeng.auckland.ac.nz/development/examples/a/simple_axis_viewer/index.html

The `time_api` example demonstrates how to set up a rendition and also how to receive time callbacks with the user interface is enabled:

http://cmiss.bioeng.auckland.ac.nz/development/examples/a/testing/cmiss_time_api/index.html

The `cmiss_field_image_api` example demonstrates how an image field is created and modified via the API:

http://cmiss.bioeng.auckland.ac.nz/development/examples/a/testing/cmiss_field_image_api/index.html

The `meshing_api` example shows how to create finite element meshes and interpolated fields:

http://cmiss.bioeng.auckland.ac.nz/development/examples/a/meshing_api/index.html

The `optimisation/mesh-alignment-c` example shows how to use the optimisation API to optimise an objective function:

<http://cmiss.bioeng.auckland.ac.nz/development/examples/a/optimisation/mesh-alignment-c/index.html>

Main CMGUI Zinc Objects

Following is a table describing the main classes of objects used in the CMGUI Zinc interface, and their relationships to each other. Note that Field and to a lesser extent Mesh and Graphics filter objects have derived types with specialised behaviours and type-specific APIs; these are not listed here.

Class	Purpose	Created or obtained from class.	Main objects created or obtained from it.
Context	Main object (instance) from which other main objects are obtained.	-	Region Graphics module Time keeper Scene viewer package
Region	Hierarchical container of fields and child regions.	Context Region (as child)	Field module
Field module	Factory object for managing fields and related objects in a region.	Region Field	Field (many types) Mesh Nodeset Time Sequence Optimisation
Field	Instance of concrete field type which can be evaluated over elements, at nodes etc.	Field module	-
Mesh	Owning container of elements over which finite element fields are defined.	Field module	Element Element template Element basis
Element	A single finite element chart over which fields can be interpolated.	Mesh	-
Element template	Object storing information for creating new elements and defining element fields.	Mesh	-
Element basis	Basis function description.	Mesh	-
Nodeset	Owning container of Nodes	Field module	Node Node template
Node	A point object, capable of having parameters stored at it.	Nodeset	-

Introduction to the CMGUI Zinc API v2.8

Class	Purpose	Created or obtained from class.	Main objects created or obtained from it.
Node template	Object storing information for creating new nodes and defining node fields.	Nodeset	-
Time sequence	Stores an increasing list of time values for defining time-varying fields.	Field module	-
Time keeper	Keeps track of system time and calls time notifiers to support animation.	Context	Time notifier
Time notifier	Describes pattern of timer notification and callbacks for a client.	Time keeper	-
Graphics module	Container/owner of all graphics-related objects.	Context	Rendition Graphics material Spectrum Graphics filter Tessellation Scene
Rendition	Collection of graphic objects visualising fields in a region.	Graphics module	Graphic
Graphics material	Object giving colour, lighting and material shader properties to a graphic.	Graphics module	-
Spectrum	Object which modifies colours of graphics depending on values of the graphic's data field.	Graphics module	-
Graphics filter	Boolean operator determining visibility of graphics	Graphics module	-
Tessellation	Controls discretization of elements into line segments for graphics.	Graphics module	-
Scene	Controls which graphics are visible by combining a top region and a graphics filter.	Graphics module	-

Introduction to the CMGUI Zinc API v2.8

Class	Purpose	Created or obtained from class.	Main objects created or obtained from it.
Graphic	Instance of a conversion operator making graphics from fields: points, lines, surfaces, isosurfaces, streamlines	Rendition	-
Scene viewer package	Factory object for creating scene viewers.	Context	Scene viewer
Scene viewer	Object for rendering a CMGUI Zinc scene into a UI-layer specific drawing canvas.	Scene viewer package	-
Optimisation	Describes and performs non-linear optimisation problems on Fields.	Field module	-