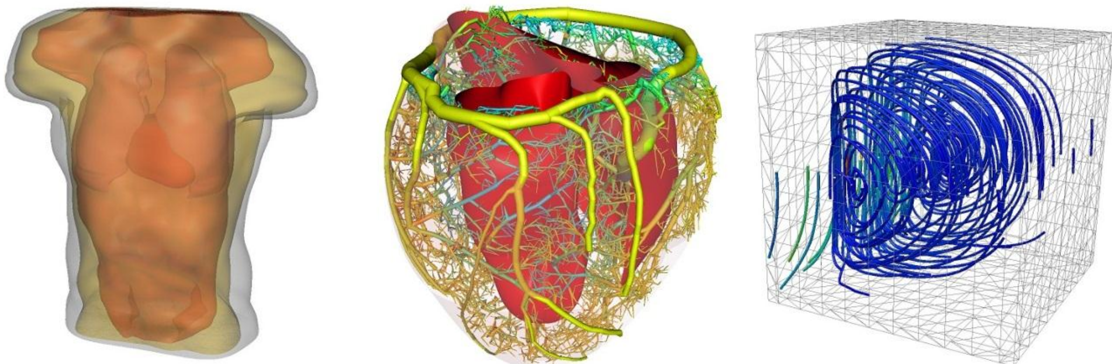
Introduction to OpenCMISS-Zinc v3.0

A Library for Interactive Modelling and Visualisation

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

7 March 2014



AUCKLAND
BIOENGINEERING INSTITUTE
THE UNIVERSITY OF AUCKLAND
NEW ZEALAND
Te Whare Wānanga o Tāmaki Makaurau

Contents

1 Overview 4
2 The Zinc Library and API 4
2.1 Library 4
2.2 Object method interface 5
2.3 Reference counting and object lifetimes 5
2.4 Change notification 5
2.5 Languages 6
2.5.1 C 6
2.5.2 C++ 6
2.5.3 Python via PyZinc 6
2.5.4 Examples 7
3 Zinc model objects 9
3.1 Context 9
3.2 Region 9
3.2.1 Model Input/Output 9
3.3 Fieldmodule 9
3.4 Mesh and Element 9
3.5 Nodeset and Node 10
3.6 Field 10
3.6.1 Finite Element Field 11
3.6.2 Image Field and Imagefilters 11
3.6.3 Group Fields 11
3.7 Timekeeper and Timenotifier 12
4 Zinc graphics objects 12
4.1 Scene 12
4.2 Graphics 12
4.2.1 Points 13
4.2.2 Lines 13
4.2.3 Surfaces 13
4.2.4 Contours 13
4.2.5 Streamlines 14
4.3 Font 14
4.4 Glyph 14

4.5 Material ... 14
4.6 Scenefilter ... 14
4.7 Scenepicker ... 15
4.8 Sceneviewer ... 15
4.9 Spectrum ... 15
4.10 Tessellation ... 15
5 Miscellaneous Zinc objects ... 16
5.1 Optimisation ... 16

1 Overview

The OpenCMISS-Zinc Library (opencmiss.org/zinc), or 'Zinc', is a software library for building interactive graphical modelling and visualisation applications.

Models are represented in Zinc as mathematical fields defined over domains, including finite elements with support for high-order basis functions, complex parameter mappings and time variation, and image-based fields. Further fields can be defined by mathematical expressions and algorithms on existing fields, including image processing filters. Zinc's model data structures are dynamic, supporting interactive applications which programmatically create, destroy and modify parts of models.

In Zinc, visualisations of models are created by graphics algorithms which assign fields to attributes, including 3-D coordinates, texture coordinates, data/colouring, and specific attributes such as iso-scalar field for contours, vector field for streamlines, and orientation, scale and label fields for points. The graphics approach combined with the general field expression capability means almost any derived result can be visualised.

Zinc graphics are rendered using OpenGL into the client window or canvas, and built-in picking, selection groups and automatic highlighting support the easy development of interactive applications. The Zinc library is UI-independent which means that additional client code is needed to set up render windows, timers and user events for interactive software: examples for doing this are listed later in this document. Zinc also includes utilities such as non-linear optimisation.

Zinc was created from the core engine of the Cmgui visualisation application (cmiss.org/cmgui), now a client of the library. Zinc v3.0 is the first release of the library with a full API for controlling its functionality without legacy Cmgui commands. Zinc and Cmgui are co-developed with the distributed parallel solver OpenCMISS-Iron (opencmiss.org/iron) which with its predecessor CMISS-cm (cmiss.org/cm) both produce output suitable for visualisation with Cmgui and Zinc.

The source code for OpenCMISS-Zinc is released under the Open Source Initiative approved Mozilla Public License 2.0.

2 The Zinc Library and API

This section describes technical details of the Zinc library including general patterns for how to use its Application Programming Interface (API). New users may prefer to first look at examples in Section 2.5.4 and Table 1, refer to descriptions of Zinc objects in later sections, and return here later. Many of the Zinc objects will be familiar to Cmgui users, and Cmgui documentation can provide further information; see: cmiss.org/cmgui/wiki/UsingCmgui.

2.1 Library

Zinc is delivered as a single dynamic linked library or shared object for Windows, Mac OS X and Linux with both a C and C++ API. It can also be used from Python via the 'PyZinc' bindings. C API headers (.h) and C++ API headers (.hpp) are named for the type or category of types they define the interface to.

The Zinc library is not thread safe: you must not call API functions on objects from the same Zinc context (see later) from different threads concurrently.

2.2 Object method interface

The Zinc API consists of types and methods (functions) that act on 'handles' to objects of those types. Many methods return handles to objects (which must be cleaned up: see the following section), and failure of these methods is indicated by the return of 0/NULL or an invalid handle.

Other API methods return an integer status code indicating the result of the operation (OK, ERROR_ARGUMENT etc.) and these constants must be compared against if subsequent processing is conditional on the result.

2.3 Reference counting and object lifetimes

Zinc uses reference counting to manage object lifetimes, ensuring that objects are not destroyed while in use. If a handle to an object is returned by an API method the reference count is incremented. In C++ and Python, smart handles automate the release of references, but in C it is very important to call a matching destroy function for each returned handle.

Usually, objects are destroyed when their reference count drops to zero, but two other strategies are employed in Zinc to manage lifetimes, depending on the class of object:

1. Objects organised in modules (field, material, tessellation, spectrum etc.) can individually have their 'managed' flag set (default is false i.e. not managed) so they exist in their module even when no external references are held. This is needed for fields read in from file, which can later be found from their module by name or other attributes. Clearing the managed flag returns to the default lifetime which destroys the object when no external references to it are held.
2. A few objects can be discarded via methods on their owning object, including Element (owner: Mesh), Node (owner: Nodeset), Graphics (owner: Scene). If the discard functions are called on these objects, handles to them remain safe but all methods on them fail except the destroy method. Note that nodes cannot be destroyed while in use by elements.

A small number of default objects cannot be destroyed, some of which are described later.

2.4 Change notification

Internal to the Zinc library, objects communicate messages informing downstream objects that they have been modified. This is what makes interactive applications built on Zinc appear dynamic: field changes are sent to scenes, if graphics are affected scene changes are sent to scene viewers which notify clients to redraw in windows, for example.

Users of the Zinc API should minimise the number of messages by enclosing multiple changes to objects in calls to 'begin change' and the matching 'end change' method, which means at most one message is sent (no message is sent when these calls are nested, nor if no changes have occurred). For a number of objects, particularly those organised in modules (fields, materials etc.) the parent object or module is responsible for messaging for all the objects it owns and its begin/end change methods must be called. Region objects also have hierarchical begin/end change methods which disable change messages for the entire region tree.

The Zinc API includes several external ‘notifier’ objects for requesting callbacks for changes to fields (from Fieldmodule, with node and element changes), selection (from Scene), Sceneviewer (from itself, for redraw and view change) and time (from Timekeeper). These assist building interactive applications and modular dialogs.

2.5 Languages

Full access to the functionality in Zinc is offered in the three languages listed in the following sections. Zinc has previously been offered as a browser plugin with JavaScript API, however with plugins deprecated by all browser vendors, appropriate demand and effort will be needed to update it or offer an alternative web interface.

2.5.1 C

To avoid name clashes with other libraries, all OpenCMISS-Zinc C API types start with the prefix "cmzn_", and all object methods start with the type name including prefix, with the object as first argument to simulate the object-method pattern. C object handle types always end in "_id" (and are currently typedef’d to a pointer-to-struct). Types and functions use lowercase letters with underscores separating words. Constants start with the prefix "CMZN_" and are capitalised with underscores separating words.

Since C does not support polymorphism this is simulated in the Zinc C API by offering "base_cast" functions which cast a derived type into its base type (without incrementing the reference count – the one exception to this rule) for passing to base type methods. Another feature needed only in C is the provision of "access" methods which increment the reference count for when an additional handle to an object needs to be kept.

Array arguments to C API methods are preceded by the integer size of the buffer being supplied, except where size is guaranteed to be fixed. This is shown in the C code example listed in Table 1. Strings returned by API functions are always allocated and must be freed using cmzn_deallocate(). Notifier callbacks in C pass user data via non-type-safe void pointers.

2.5.2 C++

The Zinc C++ API is defined in namespace OpenCMISS::Zinc, and uses CamelCase for class names and methods, but these names can be systematically converted to the C equivalent. It is created ‘inline’ over the C API, and restores polymorphism so base class methods can be called for derived classes.

Zinc object handles have value semantics in C++, and automatically release reference counts when the object is destroyed as it goes out of scope. For efficiency they are passed by reference to API methods. All handle classes implement isValid() methods for checking whether the handle is for a valid object. Passing arrays and returning function results including strings are the same as in C. Notifier callbacks in C++ must be derived from base functor objects and are thus type-safe.

Code written using the Zinc C++ API is smaller and safer than the C API and with a modern compiler should be as efficient.

2.5.3 Python via PyZinc

Python bindings to Zinc are built from its C++ API by the PyZinc module using SWIG (www.swig.org). Consequently, coding in Python is nearly identical to C++ but there are some small language differences.

Python arrays/lists know their size so this is omitted when passing arrays into Zinc. However, Python does not permit arguments to be modified, so all 'out' arrays are added to the list of function return values, and if a variable sized array was in the C++ method, the number of values requested must be supplied as an argument. The example code in Table 1 on the following page shows these differences. String values returned by Zinc are cleaned up by Python. Callbacks from Zinc are set up by passing Python callable objects to notifier objects.

Like many scripting languages, Python maintains its own shared references to objects. Assigning a Zinc object to another Python variable does not make another Zinc object handle as it would in C++: it only adds another reference to the same handle. Also be aware that releasing of Zinc handles may not be as predictable as in C++ with possible consequences for object lifetimes.

2.5.4 Examples

Table 1 shows the differences between coding with Zinc in C, C++ and Python with an example which loads a model and iterates over the 3-D elements in the root region evaluating the "coordinates" field in each, assuming these all exist. The example is in the style of script which runs from start to finish, for example to batch process results.

To write an interactive example with graphics rendered into a window requires linking to a User Interface library, event-driven programming, and additional client code for rendering (for details see Section 4.8 Sceneviewer). The documentation section of the OpenCMISS-Zinc webpage has several introductory examples using Zinc from Python including graphical applications with a Qt interface (via PySide). More are available at:

https://svn.physiomeproject.org/svn/cmiss/zinc/bindings/examples/trunk/python/

Several of these examples use the 'Zinc Widget' which takes care of basic interactive tasks including rendering, easing the development of bespoke applications. Code for this can be found at:

https://github.com/OpenCMISS-Zinc/ZincPythonUtilities/

Table 1 Comparison of Zinc code in C, C++ and Python. Examples load a model and iterate over all elements in the default 3-D mesh in the default region, evaluating field "coordinates". For brevity, details such as the main() function in C and C++ have been omitted. Error checking is also omitted; critical code should check validity of inputs and/or success of function calls.

<table>
<tr><th>C (C99)</th><th>C++</th><th>Python</th></tr>
<tr><td>

```c
#include "zinc/context.h"
#include "zinc/element.h"
#include "zinc/field.h"
#include "zinc/fieldcache.h"
#include "zinc/fieldmodule.h"
#include "zinc/region.h"

cmzn_context_id context =
    cmzn_context_create("Example");
cmzn_region_id region =
    cmzn_context_get_default_region(context);
cmzn_region_read_file(region, "example.exfile");
cmzn_fieldmodule_id fieldmodule =
    cmzn_region_get_fieldmodule(region);
cmzn_field_id field =
    cmzn_fieldmodule_find_field_by_name(
    fieldmodule, "coordinates");
cmzn_mesh_id mesh =
    cmzn_fieldmodule_find_mesh_by_dimension(
    fieldmodule, 3);
cmzn_fieldcache_id cache =
    cmzn_fieldmodule_create_fieldcache(
    fieldmodule);
const double xi[3] = { 0.5, 0.5, 0.5 };
double outValues[3];
cmzn_elementiterator_id el_iter =
    cmzn_mesh_create_elementiterator(mesh);
cmzn_element_id element = 0;
while (0 != (element =
    cmzn_elementiterator_next(el_iter)))
{
    cmzn_fieldcache_set_mesh_location(cache,
        element, 3, xi);
    cmzn_field_evaluate_real(field, cache,
        3, outValues);
    /* use outValues */
    cmzn_element_destroy(&element);
}
cmzn_elementiterator_destroy(&el_iter);
cmzn_fieldcache_destroy(&cache);
cmzn_mesh_destroy(&mesh);
cmzn_field_destroy(&field);
cmzn_fieldmodule_destroy(&fieldmodule);
cmzn_region_destroy(&region);
cmzn_context_destroy(&context);
```

</td><td>

```cpp
#include "zinc/context.hpp"
#include "zinc/element.hpp"
#include "zinc/field.hpp"
#include "zinc/fieldcache.hpp"
#include "zinc/fieldmodule.hpp"
#include "zinc/region.hpp"

using namespace OpenCMISS::Zinc;

Context context("Example");
Region region = context.getDefaultRegion();
region.readFile("example.exfile");
Fieldmodule fieldmodule = region.getFieldmodule();
Field field =
  fieldmodule.findFieldByName("coordinates");
Mesh mesh = fieldmodule.findMeshByDimension(3);
Fieldcache cache = fieldmodule.createFieldcache();
const double xi[3] = { 0.5, 0.5, 0.5 };
double outValues[3];
Elementiterator el_iter =
    mesh.createElementiterator();
Element element;
while ((element = el_iter.next()).isValid())
{
  cache.setMeshLocation(element, 3, xi);
  field.evaluateReal(cache, 3, outValues);
  // use outValues
}
```

</td><td>

```python
from opencmiss.zinc.context import Context

context = Context("Example")
region = context.getDefaultRegion()
region.readFile("example.exfile")
fieldmodule = region.getFieldmodule()
field = fieldmodule.findFieldByName("coordinates")
cache = fieldmodule.createFieldcache()
xi = [0.5, 0.5, 0.5]
mesh = fieldmodule.findMeshByDimension(3)
el_iter = mesh.createElementiterator()
element = el_iter.next()
while element.isValid():
    cache.setMeshLocation(element, xi)
    result, outValues = field.evaluateReal(cache,3)
    # Check result, use outValues
    element = el_iter.next()
```

</td></tr>
</table>

3 Zinc model objects

This section describes the main object types used for starting up and building models in Zinc.

3.1 Context

This is the first object created when using the Zinc library (and the only object not created by a method on another object), and from it all other objects are created or obtained. Multiple Context objects can be created, but API users must ensure that only Zinc objects stemming from the same context are used together!

The Context provides methods to get the default region which is typically used as the root of a model tree, but independent root regions can also be created from it. The Context also provides methods to get the module objects which own all materials, fonts, spectrums, tessellations and other graphics-related objects as described later.

3.2 Region

A Region acts as hierarchical namespace since each region can have any number of uniquely named child regions, and most region API consists of methods to create and navigate around the region tree. The first region is obtained from the Context.

The Region API provides a method to obtain its Fieldmodule which provides the interface for creating and working with domains and fields making up the model in that region. Each region also has a Scene which owns Graphics for visualising parts of the model in the owning region.

3.2.1 Model Input/Output

The Region offers simple APIs for reading and writing model files by name. The most full-featured file format supported by Zinc is the EX format (used by Cmgui; exported from OpenCMISS-Iron, CMISS-cm), documented in the /docs folder of the Zinc source and on the Cmgui web page (see: cmiss.org/cmgui/wiki/TheCmguiEXFormatGuideExnodeAndExelemFiles).

For more customised I/O, the Region can create a Streaminformation object which allows multiple 'Streamresources' including files and memory blocks to be defined and read or written with greater efficiency, and for additional options to be specified globally or per resource.

3.3 Fieldmodule

A Fieldmodule is an interface for creating and working with the model content of a Region. Its primary job is to manage the list of fields defined in the region, and it provides API for creating fields, finding fields by name, creating Fieldcache objects for evaluating fields, Fielditerators and more. It has methods to obtain the domain objects (Mesh, Nodeset) which describe the spaces over which fields are defined. The Fieldmodule has a method to define face and line elements for its meshes if needed.

3.4 Mesh and Element

The Zinc Library is currently limited to having exactly three meshes per region, one for containing three dimensional elements, one for 2-D and one for 1-D. If elements have face elements defined, they exist in the next lower dimension mesh in the same region. Hence meshes can be found by dimension from the Fieldmodule.

The Mesh API provides methods for creating and destroying elements, finding elements by identifier, creating Elementiterator objects for iterating over elements and more. Specialised 'group fields' allow subsets of domains to be defined and from these derived 'Mesh Group' objects can be obtained which support all the base Mesh API for iteration, query and so on.

Each element in a mesh is of the same dimension as the mesh, has a unique non-negative integer identifier, and a shape consistent with its dimension. Elements may have finite element fields defined on them using an Elementbasis and various parameter maps, commonly involving a mapping of parameters from local nodes listed in the element. Elements are created from Elementtemplate objects which describe the shape and how fields are defined on it. An Elementtemplate can also be merged into an existing element to define additional fields on it.

3.5 Nodeset and Node

A Nodeset is the zero dimensional equivalent of a mesh, the owner of a set of Nodes which are typically interpreted as points in space. Nodesets are obtained from a Region's Fieldmodule. Zinc is currently limited to having 2 nodesets per region, one called "nodes" used as points from which element parameters are obtained for interpolation, and one called "datapoints" for other point data.

The Nodeset API provides methods for creating and destroying nodes, finding nodes by identifier, creating Nodeiterator objects for iterating over nodes and more. Specialised 'group fields' allow subsets of domains to be defined and from these derived 'Nodeset Group' objects can be obtained which support all the base Nodeset API for iteration, query and so on.

Each Node has a unique non-negative integer identifier and can have finite element field parameters stored at it, including real values, derivatives, versions. Strings can be stored at nodes using 'stored string' fields, and 'stored mesh location' fields allow nodes to store locations in a mesh. Node field parameters can be arrays indexed by time (using a Timesequence) for time-varying fields.

Nodes are created in the parent Nodeset using Nodetemplate objects which describe the fields to be defined and what types of parameters to store for them. A Nodetemplate can also be merged into an existing Node to define or undefine fields on it.

3.6 Field

A key feature of the Zinc library is its consistent use of Field objects to describe quantities which vary over domains, including in space and time. Zinc Fields include interpolated 'finite element' fields, images and around 100 other types of fields including those defined by mathematical operators (arithmetic, trigonometric, logical, conditional, vector, matrix, derivatives, integration, constants), algorithms (ITK image processing filters, find mesh location) and other specialised types.

Methods on the Fieldmodule API are used to create fields of particular types corresponding to the operator or method each is defined with. For simple mathematical operators the arguments in their create method are enough to fully define the field. For example an 'add' field requires 2 source fields; in C++ you can write `Field c = fieldModule.createFieldAdd(a, b);` or use operator overloading to write just `Field c = a + b;` assuming fields a and b are already defined. With the exception of alias and group field types, fields may only be defined in terms of domains and fields from the same region.

Field API includes methods to get and set name, coordinate system, is-managed state and other attributes. It offers methods to evaluate and assign field values (of real, string and mesh location type) which require a Fieldcache object, obtained from the owning Fieldmodule. The Fieldcache has methods to set the domain location including time at which to evaluate or assign fields, and it also stores intermediate values and per-field caches which make multiple evaluations more efficient.

More complex field types offer type-specific APIs or require other objects to complete their definition. The C++ and Python APIs correctly present the 'polymorphism' of field types, meaning all the methods of the base Field type can be called for a derived field object. The C API requires the use of 'base cast' functions to get a base Field handle to pass to base field functions. Cast methods on the base Field are used to obtain handles to derived types, which are valid if the field is of that type.

Following are some details on more complex field types.

3.6.1 Finite Element Field

Zinc Finite Element Fields allow stored values or parameters at nodes and elements, and interpolation over elements using a variety of basis functions including tensor products of functions over each element coordinate direction. These are most commonly defined by reading in from files (see Section 3.2.1), but can also be created and modified programmatically.

When a finite element field is first constructed it is not defined anywhere; only the number of components is declared. As described earlier, Node, Nodetemplate and Nodeset API methods are used to define the field value/parameter storage at nodes which can then be evaluated into, while Element, Elementtemplate and Mesh API methods are used to define interpolation of finite element fields over new and existing elements.

Note the Zinc API does not yet offer full capability to define element interpolation involving derivatives, versions and scale factors (thus cubic Hermite basis functions) but these can be defined by reading in from file or memory resources.

3.6.2 Image Field and Imagefilters

A Zinc Image Field stores a 2-D or 3-D image giving field values for texture coordinates given on its domain field. 8 and 16 bit images are supported with 1 to 4 components.

The image field has type specific API for reading image files individually, or using a Streaminformation object a stack of 2-D images can be read to create a 3-D image. Zinc supports reading and writing most well-known image file formats including jpg, png, bmp, dicom, raw uncompressed data and others. Image fields can be attached to Materials for graphics texturing, bump mapping and as general sources of data in material shader programs.

Zinc Imagefilter field types implement a selection of ITK filters (itk.org) to perform image processing on input images. Note that image filters describe operations but don't make a stored image as a result: for that there is a special 'Image-from-source' field type which is an image field that is automatically generated from the source field.

3.6.3 Group Fields

Zinc Group fields are a collection of field types used to represent subsets of domains in a region tree. As fields they evaluate to 1 (true) at domain locations in the group, and 0 (false) outside, allowing

them to be combined in logical operations with other field expressions. However, the subdomains can also be interrogated by other high level functions.

The basic Group Field has a flag indicating whether the owning Region is in the group, which *excludes* its sub-regions. It also maintains a list of related Group fields from child regions, and subobject group fields for describing parts of domains from the same region. Node Group and Element Group fields keep track of a subset of a master Nodeset or Mesh, respectively, and offer the ability to get a handle to a Nodeset Group or Mesh Group for modifying or iterating over.

Each Scene has an optional 'selection group' attribute which can take a Group field, and supplies the currently selected domain objects for highlighting in graphics.

3.7 Timekeeper and Timenotifier

The Zinc library maintains a Timekeeper object, obtained from its module (in turn obtained from the Context or any Scene) for synchronising time across the model and graphics, and Timenotifier objects for requesting notification at specified times or frequencies.

While physical timing is left to client UI code, the Timekeeper can be queried for the next optimal time to redraw. The client is responsible for setting the current time in the Timekeeper to make it available to Zinc objects, for example 'Time Value' Fields which return the time from a Timekeeper.

4 Zinc graphics objects

This section describes the main object types used to build visualisations of Zinc models, and also perform rendering and picking. The key Scene and Graphics types are described first, followed by supporting types in alphabetical order.

4.1 Scene

Each Zinc Region has a Scene which contains the graphical representation of its domains and fields, turning the Region tree into a 'scene graph'. The Scene consists of a list of Graphics in the order they are to be drawn by a Sceneviewer. The Scene API has methods for creating Graphics of each type, and for iterating over and modifying the list. Scene begin/end change methods should be called for multiple changes to the Scene or any of its Graphics.

The Scene maintains a visibility flag which graphics can be filtered by, and its selection group field can be set to automatically highlight graphics for selected parts of the model in the Scene and child Scenes. For convenience the modules containing graphics-related objects (materials, spectrums, fonts etc.) can be obtained from any Scene.

4.2 Graphics

Each Zinc Graphics object generates a set of 3-D graphics primitives (points, lines, surfaces in 3-D space) from domains and fields of its Scene's owning Region. There are 5 types of Graphics as described below, which differ in algorithm and by the dimension of domain they work with.

Each Graphics type has a number of attributes controlling it. Attributes valid for all types include:

- Coordinate field: supplies the coordinates of the graphics. Does not need to be geometric; e.g. temperature-pressure, strain space.

- Scene coordinate system: specifies whether graphics are drawn in local, world or a Sceneviewer/window-relative coordinate system for overlay effects.
- Tessellation: controls approximation of curves by line segments.
- Field DomainType: the domain to visualise, as appropriate to algorithm: a Mesh (MESH1D, MESH2D, MESH3D), Nodeset (NODES, DATAPOINTS) or a single point per region (POINT).
- Subgroup field: specifies subset of domain to visualise.
- Material, selected material: specify colouring/shading of unselected and selected objects.
- Texture coordinates field: map to coordinates range of Material texture.
- Data field, Spectrum: for colouring graphics by a field.
- Name, visibility flag: metadata for finding graphics or filtering with a Scenefilter.

All Field attributes are cleared for new Graphics, so essential fields such as the coordinate field must be set in order to generate graphics. Other attributes including materials, tessellation, fixed values and flags have standard defaults.

Some attributes are bundled into separate objects and apply to only some Graphics types:

- Graphicspointattributes: how points are visualised including the glyph, orientation scale fields and values, label field and font.
- Graphicslineattributes: how lines are visualised including shape, scaling fields and values.
- Graphicssamplingattributes: how discrete points are sampled in elements.

Graphics primitives are generated on-demand when rendering and are automatically marked for update when fields and attributes are changed.

4.2.1 Points

Points Graphics visualise discrete locations in the model with oriented and scaled glyphs and text labels as specified by the Graphicspointattributes. Points can be generated on any field DomainType. For mesh domains, points are sampled in elements according to the Graphicssamplingattributes and Tessellation. The single point domain is used to draw a single glyph such as axes or colour bar (and is the only case not requiring a coordinate field as it defaults to the origin).

4.2.2 Lines

Lines visualise 1-D elements in the model, which currently requires 1-D line (and 2-D face) elements to be read in, or defined via the Fieldmodule, for higher dimensional elements. Lines are visualised according to the Graphicslineattributes, and can be displayed as lines or extruded circles.

4.2.3 Surfaces

Surfaces visualise 2-D elements in the model. To view faces of 3-D elements, Zinc currently requires 2-D face elements to be read in or defined via the Fieldmodule.

4.2.4 Contours

Contours generate surfaces (for 3-D domains) or lines (for 2-D domains) where its required Isoscalar field equals particular values. Isovalues can be specified as a list, or a number and range. These attributes are settable from the Contours derived-type API.

4.2.5 Streamlines

Streamlines visualise the path of a fluid particle tracking along a Stream Vector field specified via the Streamlines derived-type API. 2-D and 3-D mesh domains are supported. Seed points for streamlines are sampled from elements according to the Graphicssamplingattributes and Tessellation. Streamlines are drawn as lines, scalable ribbons or extruded circles or squares, as specified by the Graphicslineattributes. The curl of the stream vector field, or fibre sheet and normal, are visualised by the rotation or lateral orientation of the streamline when viewed with non-line shapes.

4.3 Font

Each Zinc Font is a particular OpenType typeface with a size and RenderType (bitmap, polygon, outline etc.), and is used to control the appearance of labels on Points Graphics. Fonts are managed by the Fontmodule which has methods to find and create fonts, and set the default font for new Graphics.

4.4 Glyph

Glyphs are simple graphics objects that can be drawn at each point in a Points Graphics, with its scaling and orientation. Glyphs are managed by the Glyphmodule which has methods to find them by name or by their Glyph ShapeType. It also has a method to create a collection of standard glyphs: point, sphere, cylinder, cone, cube, arrows and 3-D axes, which currently must be called on start-up to use glyphs – after defining standard materials needed for coloured glyphs.

The Glyphmodule also permits creation of two specialised glyph types: Axes (set of 3 orthogonal axes with labels) and ColourBar (a scale showing the range and colours of a Spectrum, with labels). The ColourBar automatically updates to show the current state of its Spectrum.

Some Glyphs inherit properties from the Points Graphics they are used in. Axes and colour bar use the font, and circular glyphs (cylinder, cone, sphere, arrow solid) are drawn with the Circle Divisions from the Tessellation.

4.5 Material

Zinc Materials specify colouring of Graphics similarly to the original OpenGL shading model with diffuse, ambient, emission and specular colours, shininess and alpha/opacity. Image fields can be attached for texturing (and will be used by OpenGL shaders once enabled in a future release).

Materials are managed by the Materialmodule, which is obtainable from the Context or a Scene. It provides methods to create materials, define a collection of standard materials (recommended when starting up Zinc), and set default materials for new Graphics.

4.6 Scenefilter

Zinc Scenefilter objects are Boolean functions determining which Graphics are drawn on a Sceneviewer, or processed by a Scenepicker or other tool. Scenefilters are managed by the Scenefiltermodule which has methods for creating several types of filters. The initial default Scenefilter is for ‘visibility flags’ which returns true i.e. shows Graphics whose visibility flag is set AND whose Scene and all parent Scenes have their visibility flags set. Other types filter by Region/Scene, Graphics name, Graphics Type, Field DomainType, and logical and/or operators allow expressions combining multiple filters.

4.7 Scenepicker

A Scenepicker is used to pick Graphics and domain objects in a Scene. The picking volume can be set to a rectangle in a Sceneviewer in any window-relative coordinate system, including pixel coordinates matching those in UI mouse events. The nearest Graphics, Node or Element can be queried for, and convenience methods allow all Nodes or Elements in the volume to be added to a Group Field. The Scenepicker has a Scenefilter attribute, often set to an expression combining the Sceneviewer's filter with other filters to make picking more precise and efficient. A Scenepicker is created by a method on the Scene.

4.8 Sceneviewer

The Zinc Sceneviewer is responsible for rendering the graphical Scene using OpenGL. It has methods to set its top Scene and Scenefilter, and to get and set attributes controlling the view orientation, field of view, clipping planes and more. Its renderScene() method tells the Sceneviewer to render the Scene with OpenGL. The Sceneviewer is created from the Sceneviewermodule, obtained from the Context or any Scene.

Since Zinc is UI-independent, client code has these additional responsibilities:

- Create the OpenGL-capable canvas/window in their UI library, with a Sceneviewer.
- Set the Sceneviewer viewport size to match the canvas, including on resize events.
- Make the OpenGL Context current for the canvas and tell the Sceneviewer to render.

The Sceneviewer offers a Sceneviewernotifier object which notifies of any changes to the graphics or view requiring a redraw, with flags indicating whether transformation or content has changed. Also, to get Zinc to handle rotating, panning and zooming of the window, mouse events can be converted into Sceneviewerinput objects and passed to the Sceneviewer to process. Examples and reusable code for performing these UI-specific tasks are listed in Section 2.5.4.

The Sceneviewer has a default 'head light' pointing into the screen and slightly down. API for controlling lighting will be offered in a future release.

4.9 Spectrum

A Zinc Spectrum maps values of a Graphics data field to colours. It consists of a list of Spectrumcomponent objects each of which maps a single component of the data field to one of several colour ramps, rainbow, alpha ramp, contour bands or a step function. Multiple components add to give the overall colouring. Spectrums are managed by the Spectrummodule.

The Scene getSpectrumDataRange() method gets the ranges of data field components in use by Graphics with a particular Spectrum.

4.10 Tessellation

Zinc Tessellation objects control the number of polygons or line segments used to draw element surfaces and lines, and circular forms in Graphics. Its attributes include the Minimum Divisions to use on each element coordinate direction and the Refinement Factors which multiply the minimums for a coordinate field with non-linear bases or a curvilinear coordinate system. The Circle Divisions attribute gives the number of segments used to draw circular glyphs and circle extrusions. Changing a Tessellation causes all graphics using it to be updated, giving global control of quality.

The Tessellationmodule manages the list of Tessellations and has methods for creating new Tessellations, and getting and setting default Tessellations for new Graphics. It has a separate default for Points Graphics with a single division in each element coordinate.

5 Miscellaneous Zinc objects

5.1 Optimisation

Zinc offers highly flexible non-linear optimisation built on its arbitrary field expression capabilities. An Optimisation object is created from a Fieldmodule, and its API has methods to set up the optimisation problem including Independent fields, Objective fields, solution method, tolerances and maximum iterations. Supported solution methods are Quasi-Newton and Least Squares Quasi-Newton. Once the problem is set up, the optimise() method is called to run the optimisation until its stopping criteria are met, and it can be called again if needed.

The optimisation problem requires specifying one or more Independent fields whose parameters will be changed to minimise the objective function. These can be Constant or Finite Element type. One or more Objective fields must be specified to give terms of the objective function that is to be minimised. Objective fields are typically scalar valued sums or integrals over the domain of interest, hence not spatially varying. For example, the Objective function for mesh fitting problems could be the sum (over all data points) of squared error between point coordinates and their projected coordinates on the mesh, which can be described by a NodesetSumSquares field type. Additional Objective fields allow constraints and penalty functions to be easily added.