1

Overview

This chapter presents basic information and a functional description for Granite modeling.

Topic	Page
Introduction	1 - 2
Granite Interfaces	1 - 4
Application Integration	1 - 10

Introduction

Granite is the PTC feature-based geometric modeling kernel.

Granite consists of an application integration framework built over the same core geometry libraries that are at the heart of Pro/ENGINEER. These core geometry libraries provide modeling, analysis, and data translation functionality. The core libraries also support serialization of Granite models to files in the same format used as Pro/ENGINEER. The application integration framework provides feature-based undo and redo, API journaling, and tight integration with an application's own data storage and graphical systems.

Granite provides a highly portable object-oriented API for parametric solid modeling. The Granite API has specific language bindings for C++, Java, and COM.

Granite is a solution for CAD vendors to create associatively interoperable applications. Granite enables applications to read other application files natively, and to evaluate and create geometry.

Granite enables concurrent engineering through an associative, heterogeneous design processes. Granite includes associative modeling libraries and a development environment for rapid prototyping and debugging of CAD applications.

Granite defines a gPlug architecture and protocol to permit other applications to directly read serialized Granite data.

Granite Structure

The Granite library consists of a single DLL (or shared library). This library can be linked into an application program and accessed using the C++ interface. To use the C++ interface, `#include` the supplied C++ header files in the C++ source code of your application.

The Java interface to Granite is a separate JNI DLL. To use the Java interface, include the Granite class definition file *granite.jar* in your Java compiler's CLASSPATH.

The COM interface to Granite is a windows DLL built on top of the main Granite DLL. To use the COM interface, register the Granite COM DLL, for example, using the Windows command `regsrvr32 granite_com.dll`. If you are using COM from C++, `#import` the supplied type library in your application's C++ source.

Note: All of the examples in this guide refer to the C++ language binding unless otherwise specified.

Application Integration

Getting started with Granite modeling requires only a single top-level function call. **KGetSession()** returns a **KSession** object, and methods on this object provide access to all Granite modeling capabilities.

To display a Granite model using an application's own graphics system, the application accesses triangulation, wire-frame or other geometric data directly through the Granite API.

Tighter integration with an application may require saving and loading Granite data to and from an application's own files. To achieve this, the application creates an object which implements the **KCStream** interface, and then passes this object as input to the methods **KSession::LoadModel()** and **KModel::Save()**.

Granite handles its own memory management and external file access with no further support from an application.

Prototyping with gStudio

To begin using Granite without writing a host application, use the gStudio Development Environment. gStudio is a Java-based application that provides interactive access to the Granite API. gStudio loads, displays, and saves Granite models. It also supports selection and interaction with models. gStudio includes a Java interpreter with full access to the Granite API, both interactively and through script files.

You can use gStudio to prototype new code and algorithms quickly, to use journal files from your application, or as a neutral application platform for Granite.

Modeling

Granite uses a feature- and history- based paradigm to build boundary representation (B-rep) solid models. Most of the modeling operations provided in the API correspond closely to high-level tasks (such as rounding a set of edges). Granite also supports low-level geometry and topology manipulations.

Granite Interfaces

This section describes Granite interface categories and naming conventions. It also explains how to use specific interfaces to modify a model.

Interface Categories

Granite supports five types of interfaces:

- Top-Level Functions
- Granite Basic Objects
- Granite Common Objects
- Granite Exceptions
- Granite Client Interfaces

The following sections describe each interface category in detail.

Top-Level Functions

Most Granite functionality is accessed by calling methods on interface objects. However, Granite also provides a small number of top-level functions. These functions are provided for creating top-level objects. For example, use **KGetSession()** to get the Granite session object. This function is a starting point for Granite.

Granite provides the following top-level functions:

- **KGetSession**()—Get a Granite session object. The first call to this function starts a new session.
- **KGetAnalysis()**—Get an Analysis object to access analysis methods.
- **KGetTranslator()**—Get a Translator object for data translation.
- **KGetHlrRenderer()**—Get an HlrRenderer object to perform hidden line rendering.
- **KGetTriangulator()**—Get a Triangulator object to perform triangulation and tessellation.
- **KCreateVector3D()**—Create a KVector3D object
- **KCreatePoint3D()**—Create a KPoint3D object.
- **KCreateMatrix3D()**—Create a KMatrix3D object.
- **KCreatePointUV()**—Create a KPointUV object.

- **KCreateVectorUV()**—Create a KVectorUV object

Granite Basic Objects

Granite basic objects are fundamental objects used for simple data in many methods. Methods on basic objects are limited to getting and setting object properties. Granite supports the following basic object types:

- KPoint3D
- KVector3D
- KPointUV
- KVectorUV
- KMatrix3D
- KBoxUV
- KCurveEvalData
- KSurfaceEvalData
- KCsys
- KSingularity
- KBox3D

Opaque Object Reference

An application that interfaces with Granite references basic objects through opaque reference. The application uses Granite methods to get or set basic object properties. For example, method **KCreatePoint3D()** creates a point, then **KPoint3D::Set()** sets the X, Y, and Z coordinates of the new point. Granite methods manipulate basic objects at the interface between the user application and Granite.

Value Semantics

Granite basic objects have value semantics, that is, Granite never returns two references to the same basic object. For example, if a Granite evaluation method returns a basic object, then calling the same method again returns a new object with the same property values as the first. Each subsequent call creates a new basic object copy in the memory.

Granite Common Objects

Granite common objects are higher level objects used for complex data in many methods. Methods on common objects can perform geometric queries and manipulations on the model. Examples of common objects are:

- KModel
- KFeature
- KBody
- KFace
- KSurface

See appendix Common Objects for a complete list of these objects.

Reference Semantics

Unlike basic objects, common Granite objects support reference semantics. Multiple calls to a method can return a reference to the same object from the Granite model.

Object States

Common objects that support or inherit from the **KObj** interface can have different states depending on how the application handled them. To determine the state of an object use the method **KObj::GetObjState()**.

> **Note:** **KObj::GetObjState()** is the only method valid for all **KObj** objects regardless of state.

Valid object states are:

- Active (KSTATE_ACTIVE)—For example, a face that has geometry and topology. All methods on an active object are valid.

  Some methods are available only for active objects, for example **KFace::GetSurface()**.

- Dormant (KSTATE_DORMANT)—For example, a face from a feature that is suppressed. Only a subset of methods on a dormant item are valid. In this example, **KFace::GetSurface()** is not valid and throws a **KXInvalidState** exception.

  Some methods are valid for both active and dormant objects, for example **KItem::GetHistory()**.

- Deleted (KSTATE_DELETED)—For example, a face from a suppressed, deleted feature. Only one method is valid for a deleted object, **KObj::GetObjState()**. The application may still hold a pointer to a deleted object and **KObj::GetObjState()** may be called through this pointer. Granite method calls do not usually return deleted objects to the application.

Granite Exceptions

Any method on any object can throw an exception. Exceptions are objects that signal errors and contain additional data on the error condition. Methods on exception objects return detailed data on the error.

All exception object names have the prefix **KX**.

If an application creates a section that has curves that intersect other than at their endpoints, the method **KWFeature::CreateSection()** throws a **KXIntersectingCurves** exception.The detailed error data is then the pair of curves that intersect and the parameters of the intersection point.

Exceptions also arise from argument checking. If an application passes a bad argument to a method, the method throws a **KXInvalidArg** exception. The **KXInvalidArg** exception has a property **ArgIndex**, which is the index of the bad argument. The first argument is index 1, the second argument is index 2.

Granite also uses exceptions to report invalid object states. If an application performs an invalid operation on a dormant or deleted object, then the method throws the **KXInvalidState** exception. The **KXInvalidState** exception has a property called **Obj** that references the deleted or dormant object.

Granite Client Interfaces

Client interfaces are interfaces to objects implemented by the client application, not by Granite. Names of interfaces have the prefix **KC**. Client interface objects are implemented to support tight integration between the application and Granite. Some examples of client interfaces are:

- **KCStream**—For example, as an input argument to **KSession::LoadModel()**
- **KCHlrSink**—For example, as an input argument to **KHlrRenderer::SetHlrSink()**
- **KCFaceAction, KCEdgeAction**—For example, as an input argument to **KFeature::Visit()**

Query and Modify Interfaces

Most Granite interfaces just provide read-only methods. These read-only interfaces do not change the model. Read-only methods support:

- Model traversal
- Topological query
- Geometric evaluation and analysis
- Triangulation and tessellation for graphical output
- Hidden Line Rendering
- Data translation

To create or modify model geometry and topology, use methods on writable interfaces. Names of writable interfaces begin with KW, for example, **KWBody** or **KWFeature**.

There are four ways to obtain writable interfaces. Interfaces that support modification of models are:

- Methods on **KWFeature**
- Objects created from **KWFeature**
- Feature extensions
- Casting or querying for a writable interface on an existing object

KWFeature Methods

Granite supports modification of model objects with methods on **KWFeature**. To obtain a **KWFeature** interface, use the method **KWSession::OpenFeature()**.

The method **KWFeature::Extrude()** creates a body within the model. The input to the method is a *section*, the output is a *body*.

> **Note:** **KWFeature** methods do not change the model features, they change model geometry and topology. This can also be seen as creating items in a feature, or modifying sets of items owned by other features.

Objects from WFeature

Some methods on **KWFeature** return new writable objects. **KWFeature::OpenBody()** returns a **KWBody**. Granite allows modification of model objects through methods on the created writable objects. For example, the methods **KWBody::MergeWithBody()**, **KWFace::CreateLoop()**, or **KWEdge::SetFace**().

Extension Features

Granite supports modification of model objects with methods on extension features. You can create extension features with methods on **KWModel**. For example, **KWModel::CreateTweakFeature()** takes a **KWFeature** as input and returns a **KWFeatureTweak**.

Having created the extension feature, use the methods **KWFeatureTweak::Shell()** or **KWFeatureTweak::OffsetFaces()** to create or modify model geometry and topology.

Casting or Querying

Creating an object using a method on **KWSession** provides a read only interface to that object. **KWSession::CreateArc()** returns a **KArc** interface. The **KArc** interface supports properties such as traversal, endpoints, and radius. **KArc** inherits from **KCurveDef**, which supports methods such as **KCurveDef::Copy()** and **KCurveDef::IsSameCurve()**. **KCurveDef** inherits from **KCurveEvaluator**, which supports evaluation, conversion, and other methods.

To enable modification of new geometric objects, Granite supports casting to writable interfaces. You can cast geometric primitives such as **KArc**, **KLine**, **KTorus**, to **KWArc**, **KWLine**, **KWTorus,** and so on. You can also cast a **KSession** to a **KWSession**, and **KModel** to **KWModel**.

Figure 1-1 shows how to cast to, or query for, a **KWArc** interface, given a **KArc** object called `arc`.

Figure 1-1: Cast or Query to **KWArc** Interface

```
//C++ Version
//=========
KWArc_ptr warc = KWArc::cast(arc);
```

```
//Java Version
//=========
 com.ptc.granite.KWGeometry.WArc warc =
(com.ptc.granite.KWGeometry.WArc) arc;


//COM Version
//==========
IKWArc *warc;
warc->QueryInterface(IID_IKWArc, &warc);
```

Application Integration

This section describes how to use the facilities Granite provides for integration with an application. It describes:

- Application Structure
- Saving and Loading Models
- Item UserData and UserId
- API Journaling

Application Structure

This section describes how to structure a Granite application.

Session Objects (KSession, KWSession)

A Granite application must reference a single **KSession** object throughout the lifetime of the application. The application creates the reference (and initializes Granite) by calling **KGetSession()** once. Subsequent calls to **KGetSession()** return the same **KSession** object. The application must maintain the first reference to the **KSession** object, since when the last reference to the session object is dropped, Granite is closed down. Granite does not support accessing the session object again (through **KGetSession()**) after Granite is closed down.

> **Note:** Granite does *not* check for attempts to access the session object after Granite has been closed down. The results of continuing after dropping the last reference to the session object are undefined.

The Granite session object also supports the **KWSession** interface. To obtain the **KWSession** interface from a **KSession** object, cast to or query for the **KWSession** interface.

Methods on the **KSession** interface allow an application to load models from files and to control session-wide API journaling. Methods on the **KWSession** interface allow an application to create new models, to create temporary geometry, and to obtain interfaces for feature modeling. Methods on the **KWSession** interface also allow an application to create flat assemblies. These are very simple assembly structures, for assembly HLR and analysis operations. They cannot be saved with models, and they are not intended to provide typical application assembly functionality.

Model Objects

A Granite application that is doing analysis or translation of existing models, rather than building its own geometry, interacts with Granite model objects (**KModel**) that it obtains through **KSession::LoadModel()** or from one of the translator interfaces. From the model object the application accesses the solid geometry and topology using **KModel::GetSolidBody()** and then **KBody::GetFaceList()** and so on.

The Granite model remains in memory while the model, or any of its contents, are referenced from the application. To delete the model from memory, ensure that the application no longer references the model, any of its features, or any faces or edges.

Feature Objects

A Granite application that is authoring CAD data creates solid geometry as a sequence of features, using Granite feature objects.

To build a parametric feature model that can be regenerated, an application keeps the “recipe” for each feature in its own data structures, along with a **KFeature** interface reference to the corresponding Granite feature object.

When an application feature is to be regenerated, the application suppresses the required Granite feature (the application identifies the feature to Granite by the reference it holds). Use the method **KModel::SuppressFeature**s**()** to suppress the feature. The application then opens the Granite feature for regeneration with **KWSession::OpenFeature()**. It can then re-interpret its feature recipe, making Granite method calls to rebuild the feature geometry.

After the feature geometry is rebuilt, the applications calls **KWFeature::Commit()** to commit the feature changes and prepare Granite for the next operation.

Translator Interfaces

An application can use Granite to read geometry defined in another 3D data format such as IGES, STEP or SAT.

After a Granite session is established, the application obtains a **KTranslator** interface by calling the top-level function **KGetTranslator()**. This interface provides the required methods for reading foreign data and creating a new Granite model.

An application may make any number of calls to **KGetTranslator()** during a Granite session. The **KTranslator** interface returned by each call is a new one, but the method functions identically.

There is no need for an application to keep a reference to a **KTranslator** interface. Any application code that needs such an interface can obtain one, use it temporarily, and then dispose of it.

Graphical Output Interfaces

Hidden line rendering is available through the **KHlrRenderer** interface. Obtain a **KHlrRenderer** interface by calling the top-level function **KGetHlrRenderer()** after a Granite session is established.

An application may make any number of calls to **KGetHlrRenderer()** during a Granite session. The interface returned by each call is a new one, but the methods function identically. There is no need for an application to keep a reference to a **KHlrRenderer** interface. Any application code that needs such an interface can obtain one, use it temporarily, and then dispose of it.

Triangulation and tessellation of models are available through methods on the **KTriangulator** interface. Obtain a **KTriangulator** interface by calling the top-level function **KGetTriangulator()** after a Granite session is established.

An application may make any number of calls to **KGetTriangulator()** during a Granite session. Each call returns a new object interface. The new object has default values for the properties **AngularTolerance**, **PlanarTolerance**, **TessellationTolerance**. To avoid having to reset these properties each time, an application should make a single call to **KGetTriangulator()** and use the same interface pointer for the lifetime of the Granite session.

Analysis Interfaces

Obtain a **KAnalysis** interface by calling the top-level function **KGetAnalysis** after a Granite session is established.

An application may make any number of calls to **KGetAnalysis** during a Granite session. The interface returned each time is new, but the methods functions identically.

Saving and Loading Models

Granite models can be saved to a stream and loaded from a stream. The stream is a client interface object **KCStream** (implemented by the client), so that Granite model can be stored in the application stream. Granite also provides a default implementation of **KCStream**, where it reads or writes to or from a disk file. Use method **KSession::CreateFileStream()** to create a default output stream, or **KSession::OpenFileStream()** to create a default input stream.

An example of a client implementation of the **KCStream** interface can be found in section Task 8: Saving a Model of the Task Guide chapter.

Item UserData and UserId

Granite items, that is, feature faces, edges, curves, and quilts, have User IDs. IDs are allocated to Granite items and are unique within each Granite model. Granite also provides a way to store arbitrary application data on Granite items. This ability allows applications to associate application objects with Granite items.

Granite offers the following two ways for storing data with Granite items:

- Every Granite item can store an integer value in the UserId field.
- Granite items can store a reference to an object implementing a **KCUserData** interface in the UserData field. The application can implement this interface on any of the application objects and store a pointer to this interface with the Granite item.

Use methods **KItem::GetUserData()**, **KItem::SetUserData()**, or **KItem::DeleteUserData()** to get, set, or delete item user data.

Use methods **KItem::GetUserId()** and **KItem::SetUserId()** to get or set item user IDs.

Note: There is no method to delete User IDs. An unset User ID has a value of -1.

Applications are guaranteed to get the IDs and user data back even if they drop a reference to Granite item and query for the same item later. Applications also get the user ID and user data back after regeneration.

User IDs are managed by Undo. When an application undoes model changes, user IDs are reset to their former values.

Granite objects are reference counted. When the user data object is set on any Granite item, the item holds a reference to the object. It is important to drop the reference for the object to be deleted. This can be achieved either by calling **KItem::SetUserData()** with a NULL argument or calling **KItem::DeleteUserData()**. Otherwise, the user data object is held onto as long as the model is active.

API Journaling

All Granite API calls can be journaled to a file during program execution. The calls are journaled in a neutral format that can later be converted to Java, C++, or any other language of choice.

Journal files are useful to application writers for accomplishing the following tasks:

- Tracing lifetimes and data flow of Granite object interfaces
- Moving code between gStudio and other Granite applications in either directions
- Discussing application and Granite interaction with the PTC Granite customer support organization

To begin journaling, use method **KSession::StartJournaling()**. To end journaling, use method **KSession::EndJournaling()**.

Journaling can be stopped and restarted at any time during a Granite session. To pause journaling temporarily and restart it to the same output stream it is more efficient to use the methods **KSession::PauseJournaling()** and **KSession::ResumeJournaling()**. This technique is useful around application code that is reliable and that generates a large quantity of journaling output, for example, a graphics operations.

Note: Suspend journaling for graphics operations and also stream read/write operations.

To annotate a journal file with application-specific comments, use the method **KSession::JournalComment()**.