Djinn. A Geometric Interface for Solid Modelling

Article · May 2002

Source: CiteSeer

CITATIONS
15

READS
89

11 authors, including:

Cecil G Armstrong
Queen's University Belfast
187 PUBLICATIONS **5,178** CITATIONS
SEE PROFILE

Jonathan R. Corney
University of Strathclyde
174 PUBLICATIONS **1,298** CITATIONS
SEE PROFILE

Jonathan Charles Salmon
10 PUBLICATIONS **66** CITATIONS
SEE PROFILE

Stephen Cameron
University of Oxford
109 PUBLICATIONS **1,943** CITATIONS
SEE PROFILE

Djinn

A Geometric Interface for Solid Modelling

Specification and Report

Djinn

A Geometric Interface for Solid Modelling

Specification and Report

Cecil Armstrong
Adrian Bowyer
Stephen Cameron
Jonathan Corney
Graham Jared
Ralph Martin
Alan Middleditch
Malcolm Sabin
Jonathan Salmon

THE GEOMETRIC MODELLING SOCIETY

INFORMATION GEOMETERS LTD

First published December 2000

Information Geometers Ltd
47 Stockers Avenue
Winchester
SO22 5LB
UK

Subject to availability, further copies of this book
can be obtained direct from the publisher.

ISBN 1-874728-13-5

British Library Cataloguing-in-Publication Data.
A catalogue record for this book is available
from the British Library.

Typeset and designed by
Ralph Martin, John Woodwark, and Stephen Cameron.

Printed in Great Britain by
Redwood Books
Kennet Way
Trowbridge
BA14 8RN

Contents

Part I
A
Preface

This book is about D̈jinn, a representation-independent application programming interface (API) for solid modelling. This Preface is an informal explanation of the significance of a representation-independent API, and explains why and how this one was created. It also describes the domain of modelling addressed by D̈jinn and gives a guide to the contents of the remainder of this book.

The need for Djinn

Many researchers create application software for specific purposes in computing, engineering and the sciences, using solid modelling as an underpinning representation of shape. Recently, many commercial modellers have acquired documented APIs which allow such application writers direct access to modeller functions. However, by their very nature such APIs expose the structures used in the underlying modellers and thus encourage the creation of application software which is tied to a particular modelling paradigm (e.g. CSG or boundary models), a specific product or, at worst, a particular version of that product.

Previous efforts to develop modeller-independent APIs have similar limitations. Most assume a boundary model (e.g. [Shah 1997]), some are available only on specific operating systems [DMAC], and some have been designed for particular applications such as analysis modelling.

Whilst the development of distributed computing techniques such as CORBA [CORBA], DCOM [DCOM], Java RMI [RMI] and DCE [DCE] may facilitate the mechanics of creating, querying and modifying engineering objects in a heterogeneous environment, [PDES, OMG], there is still a strong need for an interface to geometric modelling that is truly independent of the underlying representation.

The D̈jinn API specification has been developed to provide applications researchers with a means of interfacing to modellers without locking themselves in to any particular paradigm or system. Furthermore, the D̈jinn specification has been targeted especially at the plethora of new research modellers still being developed around the world. This is because one of the primary goals of the development of D̈jinn is to allow applications researchers to access the best and latest modelling technology at any one time. It is intended that D̈jinn should make it easy for applications to switch from one modeller to another, even when they are written in different languages, in pursuit of appropriate functionality.

The demand to support many applications areas led to the compilation of a list of 'all necessary functions' to be included in D̈jinn. The goal of independence from any particular modelling paradigm drove the search in the D̈jinn project for a 'lingua franca' in which to specify the necessary modelling functions. Put another way, D̈jinn had to be a 'representation independent API'. In the event, a means has been devised to specify D̈jinn in a representation-independent way using the language of point-sets.

Innovative aspects of Djinn

It appears that specifying an API for modellers in terms of operations on point-sets is a novel departure, although, as will become clear in later sections, some of the relevant mathematics has been known at least since the 1950s. D̈jinn supports a version of what is called 'cellular modelling' in which there are no boundary representation or set-theoretic (a.k.a. CSG) structures visible at the API. D̈jinn has a 'modern' implementation style making use of the encapsulation property of object orientation. Thus the D̈jinn API is specified purely in terms of the required functionality and data abstraction needed to support it; it is left to the underlying implementation to define how that requirement is met. Thus, while a particular implementation of D̈jinn may well be based on CSG or boundary modelling technology, this will not be obvious at the API nor will the user need to know such information about the underlying modeller.

What sort of implementations comply?

As a result of the abstract, and thus more general, level at which the functionality provided by the D̈jinn API is specified, the performance of compliant implementations under the interface can vary. So can the accuracy of the results and even the likelihood of obtaining correct answers: these are properties of the implementation and not of the API. However

the D̈jinn API specification does indicate the nature of the *incorrect* answers that may be produced. The D̈jinn API is specified in a way that assumes 'perfect geometry' is being used, but then gives the application the means, using D̈jinn API functions, to discover the quality of the results returned by the implementation.

Inevitably, individual research modelling systems will not support all D̈jinn functionality and also some implementations will be better or worse in some way than others in the performance of particular tasks. For these reasons, although all *functions* in the API must exist in every implementation, partial implementations are allowed, which have limited *functionality*. The D̈jinn specification includes ways to communicate responses through the API of the type: "cannot do that", "failed on this occasion to do this", or "I believe this answer is correct to x%". In the extreme case a D̈jinn-compliant implementation could simply return an indication of failure to most function calls, but it seems unlikely that application writers would find too many good uses or be willing to pay very much for such a specimen!

In designing the D̈jinn API some account was taken of the paradigms that were likely to be used, either singly or in combination, in compliant implementations. The list of those considered is as follows: boundary, set-theoretic, oct-tree and also faceted representations. In case there should be any misunderstanding, let it be said again that the geometric entities visible at the D̈jinn API are only point-sets. This paragraph merely makes the point that the designers of the API took account of the fact that the invisible modeller embodying the implementation would be quite likely to employ one or more of these modelling paradigms.

How the Djinn specification was created

As already stated, the D̈jinn API is intended to support a range of application areas. It is designed to deal with the geometric computations which underpin conventional computer-aided design and manufacture, constraint-based design, feature-based design, finite-element mesh generation, molecular modelling and other applications [Han 1996, Rossignac 1997]. The present specification is the result of an extended consultation exercise in which a committee of representative applications writers drew up a 'wish list' of desired modeller functions. Another group of modelling researchers (drawn from the UK Geometric Modelling Society) drew up an internal document responding to the wish list. The specification that forms the remainder of this book is the result of some considerable further joint effort to create a consensus view of what the D̈jinn API should be.

Limits of the domain of the Djinn API

The D̈jinn API specification describes functions that will support the integrated modelling of points, curves, surfaces and solids all embedded in (one-, two- or) three-dimensional space. Put another way, D̈jinn implementations work in a world where objects of various dimensionality 'from zero to three dimensions' can each have a separate existence and can be made to interact to form more complicated shape models. Objects in this D̈jinn world can also have their own internal cellular structure and the specification provides functions for manipulating them in these terms. A term also often used to describe this sort of modelling environment is 'non-manifold modelling', but this is not a strictly correct label because it blurs the boundary between two distinct aspects of D̈jinn: handling non-manifold geometry and representing objects with cellular structure.

Even within the domain of geometric modelling that D̈jinn addresses there are a multitude of possible variations on themes such as, for example, the representation of free-form or sculptured geometry. Thus the D̈jinn API could potentially be expected to include all the functions necessary to allow applications to use a variety of curve and surface formulations. The impossible task of including all these curve and surface types has been avoided by specifying functions to import and export approximate geometry, and by providing direct support for NURBS.

From the beginning, D̈jinn was conceived as an API which would cover a domain where the technology was sufficiently well established that a consensus could be arrived at in the actual design. It was also clear that covering the domain of geometric modelling would be a sufficiently large task in itself whether measured by time taken or the size of the resulting documentation. Thus it is intended that areas such as, for example, variational modelling and parameterized models will be implemented as further layers on top of the D̈jinn API. In fact there is already a specification for a 'features' layer over D̈jinn [Middleditch 97].

As mentioned earlier, D̈jinn makes use of encapsulation so that the actual representation of objects in the D̈jinn implementation world is hidden from the application at the API. However, the specification includes functions for exporting data out of, and (re)importing it (back) into, the encapsulated internal world under the API. This allows D̈jinn to put some appropriate responsibilities on to the application. Thus, for example, there are D̈jinn functions to allow the export of models as a triangular, faceted representation such as would be used in creating rendered images. Such 'mundane' tasks as 'journalling' (i.e. keeping a history of D̈jinn function calls and their parameters) are also left to the application writer to implement. The D̈jinn specification actually contains some simple data types with a representation that is visible at the API in order to facilitate the

application's implementation in such areas.

The D̈jinn specification also attempts to avoid pre-empting design decisions which applications writers may well wish to control. For example, D̈jinn does not prescribe how an application should choose to deal with assemblies of parts, which could either be represented as a single object with a cellular structure or as many independent objects.

Finally, it is common for applications to need to make associations between entities in its domain and geometric entities which will exist in the D̈jinn world, encapsulated within the underlying D̈jinn implementation and thus invisible at the API. The specification includes mechanisms—'labels' and 'attributes'—for making such associations. These are basic but adequate for the purpose and again involve 'a fair division of labour' between application and D̈jinn implementation.

Content of this book

Like Gaul, the remainder of this book is divided into three parts. Part I sets out a detailed description of the D̈jinn API. It introduces the rationale for the API, the basic concepts and the entities used in the specification, and then proceeds to describe the functionality available in detail. Part II is a semi-formal specification of all the functions making up the D̈jinn API. Part III describes a C language binding of the API. Both Parts II and III have their own introductory chapters which describe their content in more detail. The chapter that follows this one gives a general introduction to all aspects of D̈jinn. A more detailed analysis of the requirements is set out. This is followed by a definition of point-sets used in D̈jinn and an overview of all the types of entities used in the API specification, including: geometric objects, transformations, labels and attributes. The general principles adopted in designing the API are documented together with a description of how they are applied, the exceptions made and the limitations to their application. Finally, issues determining the visibility of data at the API and the representation of the few visible entities are discussed.

The remaining sections of Part I describe the D̈jinn API functions in detail.

Acknowledgements

The authors would like to thank various other researchers in the UK Geometric Modelling Society who helped formulate the D̈jinn API, especially John Woodwark and Patrick Campbell-Preston. Pat Fothergill,

Tony Medland, Frank Mill and Ken Swift also helped by provding users' perspectives on the required functionality.

Various companies and other bodies, including EDS Unigraphics, Perspective Design, XOX Shapes, and the US Army Ballistics Research Laboratory, freely supplied software and documentation which assisted us in developing D̈jinn by giving us insight into existing solid modelling APIs.

We also gratefully acknowledge support from the UK Engineering and Physical Sciences Research Council under grant GR/K 00841.

Part I
B
Introduction

This chapter starts with a description of the philosophy underlying the design of the D̈jinn API, followed by a summary of the requirements considered in the design. The last part of the chapter briefly introduces the concepts used at the interface. (A more formal mathematical definition of these concepts is given in Chapter C.)

Philosophy

Djinn and the application

It is assumed that some major piece of software, known in this book as the application, is using a library of geometric functions, called D̈jinn, to build geometric objects, to carry out geometric operations on them, and to enquire about their geometric properties. It is necessary to distinguish carefully between the definition of the geometric functions, called the D̈jinn API, and the underpinning executable code, which will be referred to as the D̈jinn implementation.

The application is normally expected to be a non-trivial piece of software, possibly many times the size of the D̈jinn implementation, written by competent programmers. The latter assumption implies that it is not necessary to provide every possible facility within D̈jinn. Thus the main concern in the design of D̈jinn has been to ensure that the facilities which the application actually requires can be realized by appropriate programming, using the API. Indeed several aspects of the D̈jinn API are influenced by a guiding principle of minimality, including considerations such as:

- The exposure at the API of only the minimum number of concepts necessary to support all of the intended functionality.

- The number of interface functions is kept to the minimum possible without recourse to so-called gateway procedures and switches.

Following the first of these considerations, the D̈jinn API design exploits, as far as practical, ideas of encapsulation (data hiding) and generalization recently popularized in object-oriented programming environments.

A particular instance of the application of minimality is the lack of 'logging' or 'journalling' facilities in the D̈jinn API. As far as the application is concerned, D̈jinn functions are atomic, indivisible operations and thus the calling program can easily log calls to them. A log file maintained by the application can also contain data relevant to other important stages in its operation rendering simple logging of function invocations at the D̈jinn API rather less useful by comparison. Furthermore, it seems counterproductive to encumber the D̈jinn API with generic functions for logging which are likely to be already available to applications in any general programming environment.

The D̈jinn API has also been designed with the expectation that the application will have significant data structures in its own right. Provision has been made for associations to be created and maintained between application data and that hidden inside the objects maintained by D̈jinn.

Having said all this, however, the minimality principle is a weak one in that, where the provision of a function which is not strictly needed would give significant convenience, it is included in the API.

Representation-independence

It is taken as axiomatic that the application programmer using a modelling API such as D̈jinn is concerned with the functions provided and their efficiency, and not with the internal representation of geometry or the particular algorithms used. It is therefore necessary that the D̈jinn API must be defined in abstract terms that are not specific to any of the modelling paradigms that may be used by the underpinning implementations (which may be boundary, set-theoretic, oct-tree or faceted representations as already stated). In other words, an 'open' geometric modelling kernel API must be representation-independent.

In the D̈jinn API, representation-independence is achieved by couching definitions in terms of point-sets and operations on them, as will be explained in more detail later in this chapter. By this device, applications can manipulate geometric abstractions rather than boundary graphs, set-expressions or any other structure specific to a particular modelling paradigm. Thus neither the underlying modelling paradigm used nor any particular language implementation is reflected in the API definition, but issues of accuracy, degeneracy, robustness and exception handling are dealt

with explicitly. Indeed, implementations are free to use multiple representations and to employ conversions internally if necessary; the Djinn API only defines what is to be computed without reflecting how it should be done. In designing Djinn, however, consideration has been given to 'implementability'; it would be foolish, for example, to build in the assumption of the existence of a general solution to the historic problem of converting models from boundary to set-theoretic representation.

State and side-effects

In designing any API, it is important to decide whether the implementation behind it will maintain any information about its internal state between function invocations across the interface. In the case of Djinn, it is clear that the geometric objects handled by the implementation contain state to record their shape, and possibly other related information. However, an important Djinn design aim has been to make this state (and its limitations) extremely clear.

It has been decided that all Djinn functions are to be re-entrant, so that all manipulations of different pieces of geometry can be interleaved (or even concurrent, if the implementation permits that) without interaction. This means that there is no state in Djinn outside the data objects which are passed in and out of the functions as arguments. One practical result of this, for example, is that a desired computational accuracy has to be supplied to each interrogation function rather the Djinn implementation keeping any 'accuracy state' information. The only exceptions are an indication that Djinn has been correctly initialized, and other data relating to the reading and writing of information to files for persistence or archiving purposes. Writing an object to a file does have a side-effect on the file, and reading an object does alter the position of the 'read-pointer' within it. This exception has to be accepted, as to have outlawed it would have prevented any data from lasting for more than one 'session'.

Persistence

Once the Djinn interface is used in significant applications, end-users will start to invest in data held in Djinn objects. It has therefore been accepted that there is a need for object data to have considerable persistence. The save-restore cycle, for which a single mechanism is defined in Djinn, is intended to cover both the saving of data and reloading it into a subsequent session using the same executable, and also the reloading of data into new versions of the application.

However, achieving persistence where an application has migrated from one Djinn implementation to another would have demanded that this

specification should lay down the formats in which all objects should be archived. It was felt that this was unlikely to be achieved in a way that was independent of the internal details of D̈jinn implementations and thus any attempt to do so would undermine the essential representation independence of the API. It is therefore assumed that any migration of data between different D̈jinn implementations will be done by applications' logging of calls to D̈jinn API functions.

Language binding

Applications are written in many different languages. Indeed, some have parts within them which take advantage of different languages for different purposes and thus D̈jinn calls may need to be made from more than one such part.

Any D̈jinn implementation may therefore need to be called from a number of different languages. A D̈jinn implementation therefore needs to be callable from any of the widely-used, compiled, procedural languages: C, C++, Fortran and Pascal come immediately to mind. It is therefore necessary that D̈jinn implementations be written in a language which can be called from these.

In the light of earlier design decisions about state and side-effects, it would be convenient to place further restrictions on language bindings by outlawing the use of certain language features. Examples of these would be: functions as arguments, call-back function registration and functions on data structures since they provide a means of circumvention. Unfortunately, D̈jinn can do without the last two of these, but not the first. In order to allow a convenient specification of functions when computing integrals over D̈jinn objects (as defined later in Chapter 8), it is necessary to use functions as arguments.

In order to make it as easy as possible for applications to migrate from one D̈jinn implementation to another, there needs to be a language binding for each such language, so that widespread editing of the application is not necessary. As a result of the limitations on resources of the D̈jinn project only a C binding has been defined, but in principle others should follow.

Each language binding defines the data types with which the application should communicate arguments to D̈jinn, and the names to be used for each of the functions.

Requirements

The following is a summary of application requirements in so far as they have affected the D̈jinn API specification.

Multi-dimensional models

Surfaces can be used to represent iso-valued point-sets of scalar fields, boundaries of solids and, together with a thickness attribute, solid shells. Thin-sheet representations are widely used in the shipbuilding, automotive and aerospace industries. Curves can be used to represent streamlines in fluid flow, NC machine-tool cutter paths, the edges of sheet and solid objects and, together with a cross-section, sheet or solid objects. Most existing geometric modelling systems are concerned with objects of a single dimensionality embedded in two- or three-dimensional space: for instance, drafting systems (one-dimensional arcs in two-dimensional space) and three-dimensional design systems (two-dimensional surfaces or three-dimensional solids in three-dimensional space). Such systems do not meet the needs of applications that require the ability to model geometric entities of different dimensions simultaneously [Middleditch 1994]. For example:

Design: A car body modelled as a surface rather than a solid must admit the subtraction of a solid to create the hole for a head lamp.

Analysis: Objects are often simplified for analysis. For example, some parts of a solid component may be analysed as a shell (i.e. a surface with a thickness property), and a bolt hole may be treated as a line segment with loads and constraints. A yacht might be modelled as a solid with the sail, represented by a sheet, embedded in a block of fluid and a block of air, secured by one-dimensional ropes. The final shape of the sail in this example may only be defined by an elastic analysis of the deformation of a faceted model of the sail shell in response to wind pressure, gravity loading and the elastic deflection of supporting structures like the mast and ropes. Similar considerations apply to objects produced by manufacturing processes like thermoforming or forging: the final shape or properties like sheet thickness may be only available on a faceted representation following an analysis.

Manufacturing: An end-mill traverses a curve to create a slot and that curve must be derived from representations of the solid cutter and the solid void to be created [Stacey 1986].

Cellular models

Many applications require the ability to partition an object model into a set of disjoint cells. It must be possible to manipulate such cells individually, without losing the ability to operate on the object as a whole. For example:

- Models of assemblies require composite object models with disjoint pieces that can be separately interrogated and manipulated.
- Objects composed of different materials require object models to include a partitioning. For example, such models are useful for semiconductors in which defined regions of the original material have been doped to create different electrical properties. An aerospace composite laminate may superficially appear as a thin sheet but be composed of multiple layers with different properties. It should be possible to model such a structure with a void or crack at a given location between two layers. There are similar requirements in geological modelling and information systems. A reinforced concrete structure might be represented as a solid with embedded one-dimensional wires to represent the steel reinforcing bars. It must be possible to discover that there are embedded partitions and to find if and where they penetrate the surface.
- Finite-element systems involve elements that are subdivisions of wire, sheet or solid objects (i.e. objects to be modelled as a set of disjoint cells). The shape of an object is often changed for analysis. For example, insignificant chamfers and fillets may be removed and tangencies may be replaced to enable the use of an element with reasonable shape. The identity of a cell must persist through such shape modification and also through subdivision. For example, it must be possible to assign a material property to an entire cell that has been partitioned and for a material property to remain associated with a cell that is subsequently partitioned. Sometimes it is necessary for the cells of an object to share their common geometry together with its properties, e.g. in finite element systems. In contrast, the bounding entities of cells that represent components in an assembly should be distinct, merely having the same coordinate values.

Non-manifold models

Industrial applications may involve non-manifold objects [Charlesworth 1995] (e.g. three-dimensional solids with an edge incident to more than two faces—see Figure 1). For example, a manifold three-dimensional region to be machined may require the cutter to traverse a non-manifold region. Many existing boundary representation geometric modelling systems do not accommodate such objects.

Boundary representation

Sub-regions of the boundary of a wire, sheet or solid object must be separately accessible in order to associate application data with them.

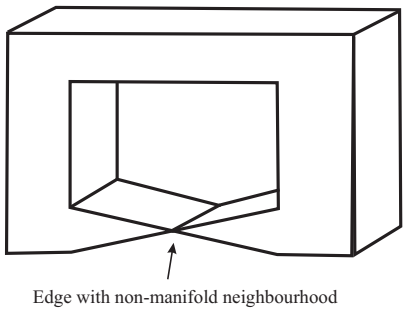


Figure 1: Non-manifold object.

For example, finite-element systems require different parts of an object boundary to be assigned different loads and constraints. These edges and faces constitute an application dependent cellular structure for the object boundary. 'Natural' faces of a solid are bounded by edges defined by tangent discontinuities, but there are several alternative face definitions [Silva 1981]. Applications sometimes require faces that are subdivisions of 'natural' faces, to represent an area of a face where a load or restraint is applied in an analysis, or where a different surface finish is prescribed in a machining application. Applications may also require adjacent 'natural' faces to be considered as a single face, if for example the geometric feature is below the scale of interest in an analysis or visualization. Similar considerations apply to edges.

Collections

Some applications require lists of parts of an object (e.g. all red faces). Elements of such collections could belong to more than one collection.

Object creation and modification

It must be possible to create and modify objects of any dimension. A useful set of primitive shapes is required. It should be possible to create new objects by combining others with Boolean operations , sweeping into new shapes [Martin 1990], filling a closed lower dimensional boundary or 'emptying' the interior of a higher dimensional object. Local modifications such as tweaking and blending are desirable. Facilities to convert an object to other approximate forms which are more easily used by the application are needed. These include bounding boxes, facetings, oct-trees or voxels. The medial axis transform (related to the Voronoi diagram) is a well-defined alternative representation which is potentially valuable for applications such as proximity determination (pairs of opposite faces), feature recognition (identification of thin walls for machining, thin sheets

for analysis) and robotic path planning (using the MAT as a lower dimensional skeleton). Convex hulls have applications in manufacturing to determine stability for fixturing.

Methods are needed to create the cellular structure of partitioned models and to allow modifications of this structure.

Association with applications data

A critical requirement of the modeller is that it should be possible to associate application data (attributes) with particular parts of the geometry and for applications to identify or label particular parts of the geometry in a modeller-independent way [Braid 1985, Subrahmanyam 1995]. For applications such as feature-based design, it would be highly desirable to have repeatable labelling, where the identifier of a particular part of the geometry is the same if, for example, the history of construction is the same. It should be possible to discover what elements of the object boundary or partitioning have changed as a result of a particular geometric operation, so that the application can modify its own data structures. Thus, a structural optimization which was attempting find the best position of a hole should be told that the axis of the hole had been moved too far if the topology of the faces penetrated by the hole had changed.

Geometric enquiries

Local geometric enquiries such as closest point, edge tangent, curvature and torsion, face normal, and principal curvatures and directions are needed. Tests for point containment in an edge, face or volume are often necessary. Derived measures such as the concavity or convexity of an edge along its length are important in feature recognition. Ray-casting is useful for picking, testing access directions and visualization. Integral properties such as edge length, face area and solid volume are used in many applications. Centre of gravity and moments of inertia enquiries could usefully be supplemented with the ability to integrate a specified function over the domain. This would allow for example the computation of resultant forces due to a given pressure distribution over a surface, or a given gravitational or acceleration loading over a volume. Testing for the occurrence of intersection is usually more efficient than computing the exact Boolean intersection of a pair of solids, faces or edges and thus tests of this sort are also useful. One class of geometric enquiry which is incompatible with the D̈jinn approach is that which depends on an underlying parameter space for curves and surfaces, since not all geometry classes have such a description.

Topological enquiries

Conventionally, boundary modellers provide information such as the vertices bounding an edge, the edges bounding a face and the faces bounding a solid. With a mixed dimensional, cellular model more information is required. Finite-element stress analysis of an oil rig typically involves three-dimensional solid representation of the detailed shape, including blend radii, where several cylindrical legs are joined. At a distance of the order of the tube wall thickness away from the three-dimensional blends, a cylindrical shell is sufficient. Further than a few diameters away from the intersection, the legs are accurately and economically represented by a one-dimensional line with section properties.

It must be possible to identify the dimensionality of a given part, so that it is clear that a particular face attached to a given edge of a solid is a surface model, not a face of the solid. Similarly for an edge attached to a vertex: is it part of the boundary of the body or a one-dimensional body attached to the exterior of a higher-dimensional body at this point, or a reinforcing bar through the interior of the body? It must be possible unambiguously to identify adjacencies such as all the faces attached to a given vertex of a given body in a cellular model, where a given face may be shared between two bodies. Other higher-level topological enquiries might be to find the common boundary between a pair of solids, or whether a given vertex and edge are part of the boundary of a given face: have they a common parent of a given dimensionality?

Product data models

To derive the full benefit from precise geometric modelling of solids, there is great emphasis in commercial systems on having all data associated with the design, analysis and manufacturing of the product held in association with the geometric models. D̈jinn cannot possibly address all the issues associated with product data modelling so the philosophy has been to limit support to geometrical operations and interrogations, but to provide facilities whereby higher level applications can query the underlying geometric modeller for information which may be of significance to it. D̈jinn should at least provide a mechanism whereby an application can discover what changes have been made to an object as a result of a D̈jinn call. Thus for example, while feature-based design is an important application of geometric modelling and feature-based modelling usually requires constraints to be enforced on relationships between features, these facilities are not provided by the D̈jinn implementation.

Features and constraints

Mechanical components, buildings and other objects are typically designed in terms of features (e.g. a bearing with a flange containing a circle of bolt holes). Such features may also be designed in terms of other features, thus leading to a feature hierarchy [Mäntylä 1996]. Each sub-feature must be defined in a coordinate system located with respect to its parent feature. Such feature hierarchies also are useful for modelling the human and other biological forms. A process-plan for component manufacture consists of a sequence of machining operations, each removing some material from a workpiece. This requires the final component to be modelled as the initial workpiece together with a set of voids. Such voids are features of an object that define a region outside that object rather than a part of it (e.g. a T-slot in a machine-tool fixture). Adjacent features may overlap (e.g. some machine-tool fixtures have orthogonal slots that cross each other).

Constraint-based computer-aided design systems solve a set of equations that relate geometric parameters, then change the geometry to reflect new values of those parameters. There is thus a requirement for modifications such as:

- Vertex or face translation.
- Hole diameter modification.
- Translation of one feature with respect to another.

Constraint-based design imposes relationships between features. Sets of such relationships often relate a feature to more than one other feature. This often leads to the incorrect deduction that it should be possible for a feature to have more than one parent and thus that a hierarchy is an inappropriate structure for feature-based design. It is the constraint relationships for which a strict hierarchy is inadequate and for which a general graph is necessary. Features remain hierarchical irrespective of the constraints between them: a flea cannot be on two dogs simultaneously.

However, in line with the philosophy of keeping D̈jinn simple, it is left to the external application to keep track of hierarchies or other groupings of D̈jinn cells. The cellular structure provided by D̈jinn is an adequate basis for feature description within applications.

Furthermore, D̈jinn supports local modifications to objects which do not change adjacency relationships in the cellular structure. The proper specification of modifications which require changes in adjacency is still an open research issue [Ragothama 1998].

Summary

A geometric modelling system for use as a kernel in industrial applications of design, analysis and manufacture should support geometric objects con-

sisting of isolated points, arcs, surface patches and solids, i.e. zero-, one-, two- and three-dimensional geometric entities, in three-dimensional space. The API must provide for the following requirements:

- The ability to deal with boundaries and to be able to handle various subsets of them independently.
- Have facilities which allow feature-based application layers to be built on top of the API.
- Allow objects with internal faces, edges and vertices which may or may not subdivide the solid into disjoint pieces.
- Allow two way association between application data and entities held by the D̈jinn implementation.

Figure 2 provides an example of a two-dimensional object that exhibits these characteristics. The dashed boundary segment indicates that the associated boundary points of the rectangle are not included, and the dashed tram-line indicates a crack in the rectangle (i.e. a missing set of points on a one-dimensional line segment). Figure 2 also includes lines external to the rectangle which are isolated or connected to it, and lines internal to the rectangle, only one of which separates two cells. It also includes an isolated external point, a point within the rectangle, and a 'missing' point in the rectangle.

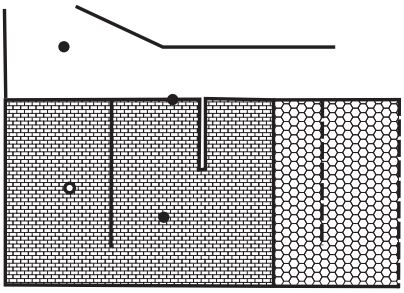


Figure 2: Two-dimensional object exhibiting cellular structure and non-manifold geometry.

It must be possible to intersperse geometric modification with cell creation and modification, without losing cellular structure unaffected by the modification. Also it must be possible to unite two touching objects so that they either coalesce or remain as two distinct disjoint cells.

Djinn concepts

Context

D̈jinn deals with three-dimensional geometry, although the actual specification of the majority of the functions allows them to be used as if the

Djinn world were two or even one-dimensional. The geometric objects may be zero-, one-, two- or three-dimensional within the modelling space.

As stated earlier, the Djinn API exposes the minimum number of concepts to the application. Those that it does use fall into two categories: those with a representation that is visible and those for which it is hidden.

Visible types

Although Djinn largely deals with objects with hidden internal representations, because their representation may differ greatly between implementations, there has to be some level at which the application may transmit actual data into Djinn and receive results.

Djinn therefore allows the representation of function arguments of the following types to be visible to the application:

Booleans	
Character strings	
Integers	
Reals	
Logic variables	With values true, unknown or false
Vectors	A sequence of reals of arbitrary length
Matrices	A sequence of vectors of arbitrary length
Facetings	
Strings of conics	

In languages which support type definitions, the implementation itself will declare the above types: in languages which do not, the language binding will state their relationship to the types of the host language.

Hidden types

The four concepts with hidden representation are now introduced. These are Djinn objects (the geometric abstraction used by Djinn which is based on point-sets), a means of dealing with collections of entities, transformations and a way of describing the precision of geometric calculations.

Object representation

In this section only a few salient points about the representation of objects are explained: a fully detailed mathematical presentation is provided in Chapter C.

Any Djinn object more complicated than a point, unbounded line or plane is a collection of cells. These may be viewed either in set-theoretic

terms, as a partitioning of space into disjoint point-sets, or in boundary representation terms as a collection of boundary elements with adjacency (bounding) relationships.

Each D̈jinn cell of a complex is an oriented, manifold, semi-analytic, open point-set, homogeneous in dimension. The cells of a D̈jinn object are mutually disjoint.

A D̈jinn object can represent a cellularly partitioned, non-manifold object containing cells of differing dimensionality. The point-set of an object is just the union of its cells.

Stratification: An object is not just a random collection of disjoint cells, however. The boundary of each cell must be exactly the union of some other cells also in the object. This is achieved trivially in the setting up of primitives, since each starts as life with an interior cell and a boundary cell. Either or both of these cells can be partitioned by subsequent operations. However, since all D̈jinn operations maintain this stratification, the application does not need to be concerned with it.

Labels (and OUT*):* Individual cells within an object can be identified by their labels. When an object is created, the function always returns an access path to at least one label, so that the application can navigate through the object. For compatibility with boundary modelling and with conventional set-theoretic modelling there is a reserved label value, called OUT, which always acts as an identifier to the part of space outside the object.

Labels are of hidden type, and either the implementation or the language binding informs the application of the value of OUT.

Orientation: Although a point-set is just a point-set, it has proved extremely convenient in the programming of both modellers and applications to be able to distinguish reliably between, for example, the two sides of a face.

This is expressed in the ability to enquire a tangent vector of a curve cell, and the normal vector of a surface cell, at any point thereof. Application programmers will normally expect that enquiries at nearby points will give results of consistent sense, and that the data abstraction of D̈jinn gives exactly that assurance. This means that a Möbius strip cannot be held as a single cell (with its single edge boundary). There has to be an additional edge cell lying across the threshold where the orientation of the surface normal flips.

Relative orientation: In the cellular context, a face separating two solid cells cannot have its normal pointing out of both of them. The orientation of a cell with respect to another which it bounds is important and can be determined by enquiry.

Manifoldness: An open point-set is manifold if every point has a neighbourhood homeomorphic to a ball of appropriate dimension.

Semi-analyticity: Not all point-sets are useful for the representation of geometric entities, but semi-analytic point-sets provide a reasonable domain [Requicha 1980]. These are defined in detail in Chapter C, where a full description of their significance is also given.

Openness: An open point-set does not contain its boundary. This is important in geometric complexes, since the boundary will be made up of distinct cells, which must be disjoint from the bounded cell. The point-set of the complete D̈jinn object, however, is closed because all boundaries must be included.

Homogeneous in dimension: All parts of a cell must have the same manifold dimension. A single cell cannot combine pieces of wire, sheets and solid parts. To achieve multi-dimensionality requires a D̈jinn object to be made up of more than one cell.

Connectedness: D̈jinn cells are not required to be connected. A single cell can consist of a number of disconnected pieces, but they must all be of the same dimension.

If an application requires each of the cells of its objects to be a connected point-set, a D̈jinn function may be invoked to split all disconnected cells into sets of cells, each of which is connected.

Attributes: The need to support sophisticated associations between geometric and non-geometric data of the application has already been mentioned. There are two mechanisms for this. Access can be gained from application data to the individual cells of an object by storing the values of those cells' labels (see above) in application data. Access can be gained from the cells of an object to application data by the association of attributes with each cell.

A D̈jinn cell can have an arbitrary number of attributes, each of which is a key-value pair. It is necessary to devise a means of preventing clashes between keys created by different users in different applications with different D̈jinn implementations; how this is achieved in D̈jinn will be described later.

Many of the more powerful modelling operations in D̈jinn alter the cellular structure of an object, as well as its overall shape. After any of these operations it is necessary, in principle, to update the associations between cells and application data. In the interests of security and provability D̈jinn avoids the use of call-back routines for this purpose. Instead, an object has, as part of its state, information about the parentage of each of its cells (including OUT). There are D̈jinn enquiries to access this data so that an application can carry out the necessary updating.

Parentage map As already mentioned, D̈jinn objects contain a parentage map which allows an application to track what happens to the cells of the object across D̈jinn function calls. From this map, an application can determine the ancestors of any newly created cells, and what has happened to any cells that existed before a particular function call.

Modification record Each D̈jinn object also keeps track of the number of times that its point-set has been modified, together with a representation of a volume enclosing the region of change. This data applies to operations performed since the modification record was last reset and, together with the parentage map, allows applications to work more efficiently.

Other hidden types

As well as D̈jinn objects used for representing geometric entities, other hidden types are also provided by D̈jinn for various purposes. These will now be described.

Frequently used geometries: It is often convenient, particularly for purposes of articulating enquiries, to be able to deal with frequently used simple linear entities: points and hyper-planes (i.e. lines and planes) without all the connotations which the use of D̈jinn objects (q.v.) would imply. D̈jinn therefore provides two extra hidden data types and a set of functions specifically to ease manipulation of these geometries.

Label sets: Many D̈jinn enquiries take the form of "tell me all the cells which ..." (for example, all the cells which bound this cell, all the cells from which this cell is derived, all the one-dimensional cells, ...) and D̈jinn provides a mechanism for applications to manipulate collections of cells.

The facilities provided in D̈jinn for handling collections of cells are based around two abstractions: sets and multi-sets. These are called,

respectively, label-sets and label-sequences and D̈jinn provides functions for mutual conversion between them. A sophisticated set of manipulation functions is provided for label-sets, whereas only those functions needed to scan through a label-sequence are provided.

Transformations: There are many geometric transformations required, both in modelling and in making sense of models.

D̈jinn deals with these as $1:1$ invertible, continuous maps from points to points. Under such maps, geometric complexes map into geometric complexes, each cell mapping compatibly into its own image of the same dimensionality, and so the effect of a transformation on any object is well-defined.

For certain purposes, D̈jinn transformations are also regarded, not as single maps, but as being parameterized by a scalar parameter. The transformation value when the parameter is zero is the identity map; the value when the parameter is one is the nominal transformation, and intermediate values of parameter give intermediate 1:1 invertible maps.

For convenience when D̈jinn is being used in a lower-dimensional context, each transform has a spatial dimension. A translation along the x-axis (but in no other direction) may be regarded as being of spatial dimension 1. A translation in the xy-plane, or a rotation about the x-axis, may be regarded as being of spatial dimension 2. Most transformations will be of spatial dimension 3.

Precisions: Many geometric enquiries may give results with precision dependent on the representations used by the particular implementation. Faceted and voxel modellers, in particular, may be expected to return approximate answers, but even with the nominally 'exact' geometry of set-theoretic and precise boundary representation modellers, the real arithmetic used by computers may cause the actual result of an enquiry to be an approximation.

There are several possible approaches to dealing with the 'inevitable' geometric approximations of an implementation and the issue of how much they should be exposed to the application. These are discussed at greater length in Chapter C. Here it will suffice to say that D̈jinn nominally assumes exact geometry, but allows the application to specify desired interrogation accuracy and also to enquire the achieved accuracy of results.

There are a number of different measures of precision which might be returned, and in order to provide for those most likely to be wanted, D̈jinn provides a type of hidden structure which holds the achieved

precision. A variable of this kind may be further enquired to give specific measures of the accuracy obtained.

A more searching theoretical basis for many of the concepts introduced above is provided in the following chapter.

Part I
C
Necessary theory

Mathematical background

This section summarizes the relevant theoretical background and provides definitions and notation. Together with the requirements described previously, discussion of this theory leads to a choice of D̈jinn object properties relating to space and coordinate systems, orientation, boundaries and continuity. These properties are then used to define D̈jinn objects formally.

Space and coordinate system

Each point of a solid object can be modelled as a coordinate triple with respect to some implicit coordinate system[1] (i.e. a point in $\mathcal{R}^3$). The objects we wish to model can be represented in three-dimensional Euclidean space $\mathcal{E}^3$ (i.e. $\mathcal{R}^3$ with the standard Euclidean distance metric defined on pairs of points). Since some applications manipulate one and two-dimensional objects, the subspaces $\mathcal{E}^1$ and $\mathcal{E}^2$ are also relevant.

Models can be constructed from intermediate models that contain points at infinity. Projective transformations which transform points to infinity can be used to generate perspective images and as distortions for design purposes (see Chapter 4). However, projective spaces ($\mathcal{P}^n$) lack a distance metric and other properties important to the applications mentioned previously. For example, $\mathcal{P}^2$ is not orientable.

D̈jinn geometry is defined in Euclidean space $\mathcal{E}^n$, $1 \leq n \leq 3$. Projective transformations are supported, but their domain is restricted to avoid

[1] (Appropriate) sets of coordinates also can be used to represent geometric entities other than points. Planes and lines each have a variety of sets of numerical coordinates, but Djinn treats them all as point-sets, subordinate in a philosophical sense to the point coordinate view.

points at infinity.

Point-sets

Set-theory has been strongly associated with CSG modelling systems, but the geometric entities manipulated by boundary modellers can also be defined as point-sets. Boundary modellers operate on closed surfaces by exploiting topological concepts such as Euler operators, whilst naive set-theoretic modellers apply set-operators to point-sets. However, since closed surfaces bound a solid, correct topology follows from solidity, not the other way around. Therefore, D̈jinn defines points, curves, surfaces and solids, and operations on them, in terms of point-sets.

These point-sets are equipped with topological structure (i.e. form topological spaces). Except where stated otherwise this will be the topology induced by $\mathcal{E}^n$ as described below.

Topology

A set R is a *topological space* if every element (point) x of R has an associated set of neighbourhoods (subsets of R) and the following conditions are satisfied:

- The point x is a member of each of its neighbourhoods.
- Any subset of R containing a neighbourhood of x is also a neighbourhood of x.
- The intersection of any two neighbourhoods of x is also a neighbourhood of x.
- If N is a neighbourhood of x, then N contains a neighbourhood M of x such that N is a neighbourhood of every element of M.

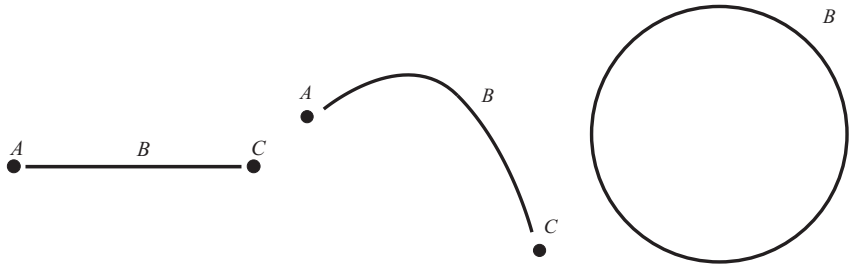
It is only necessary to consider Hausdorff topological spaces where for any two different points each has a neighbourhood not containing the other.

In Euclidean space, an (open) *n-ball* centred on point $P \in \mathcal{E}^n$ is defined as the set of points $\{Q \in \mathcal{E}^n \mid \|P - Q\| < r\}$ for some $r \in \mathcal{R} - \{0\}$. The unit n-ball centred at the origin is written B^n.

The standard topology for $\mathcal{E}^n$ results from defining a point-set N to be a neighbourhood of point P if N contains an n-ball centred on P.

If S is a subset of points of a topological space R, then S can be considered a topological space where the neighbourhoods of P in S are the neighbourhoods of P in R intersected with S. This is called the induced or relative topology on S with respect to R.

For a topological space R and subset S of the points in R, the following properties are now defined (see Figure 3):



	a) 1D cell embedded in ϵ^1	b) 1D cell embedded in ϵ^2	c) 1D cell embedded in ϵ^2
boundary	$\partial B = A \cup C$	$\partial B = A \cup B \cup C$	$\partial B = B$
interior	$interior(B) = B$	$interior(B) = \emptyset$	$interior(B) = \emptyset$
closure	$closure(B) = A \cup B \cup C$	$closure(B) = A \cup B \cup C$	$closure(B) = B$
frontier	$front(B) = A \cup C$	$front(B) = A \cup C$	$front(B) = \emptyset$

Figure 3: Topological relationships.

- The *boundary* ∂S is the set of points in R, every neighbourhood of which contains both points in S and points not in S.
- The *interior* $Interior\,(S)$ is the set of points in S that are not boundary points:

$$Interior\,(S) = S - \partial S.$$

- The *closure* $Closure(S)$ is the union of S and its boundary:

$$Closure(S) = S \cup \partial S.$$

- S is *closed* if it contains its boundary (i.e. iff $S = Closure(S)$).
- S is *open* if it does not contain any of its boundary (i.e. iff $S = Interior\,(S)$, or equivalently if $R - S$ is closed).

A set S can be both open and closed (e.g. the empty set) and a set can be neither open nor closed (e.g. a set containing part of its boundary but not all of it).

- A subset S of $\mathcal{E}^n$ is *bounded* if it is contained in an open n-ball of finite radius. (This should not be confused with S having a boundary.)

  A concept which is particularly valuable for arcs embedded in $\mathcal{E}^2$ and patches embedded in $\mathcal{E}^3$ is that of a frontier.
- The *frontier* $Frontier(S)$ of S (a subspace of R) is that part of the boundary not in S.

$$Frontier(S) = Closure(S) - S.$$

  The frontier is equivalent to taking the boundary relative to the subspace which is the closure of S rather than R.

When S is an open set in R, then the frontier coincides with the boundary. Thus, for the one-dimensional cell embedded in $\mathcal{E}^1$ in Figure 3(a), both the frontier and the point-set B are its end-points $A \cap C$. However, for a one-dimensional cell embedded in $\mathcal{E}^2$, the boundary of B is $A \cup B \cup C$, since every point in B has a neighbourhood which contains points in B and not in B. In this case, it is the frontier of B, $A \cup C$, which contains the end-points of the arc (Figure 3(b)). Note that the frontier of the closed circular arc in Figure 3(c) is empty.

As another example, the frontier of an open 3-ball in $\mathcal{E}^3$ is its surface sphere. The frontier of this sphere in $\mathcal{E}^3$ is empty whereas its boundary is the sphere itself. If a closed disc is removed from the spherical surface, then the frontier of the remaining spherical surface is the edge of the disc, whereas its boundary is the union of the edge of the disc and the partial spherical surface itself.

- A space R is *connected* if, for any non-empty subsets A and B such that $R = A \cup B$, either $A \cap \mathit{Closure}(B)$ or $B \cap \mathit{Closure}(A)$ is non-empty.
- A space R is *path-connected* if any two points in R can be connected by a continuous path in R. (Path-connection implies connection, but not vice versa).
- If X and Y are topological spaces, then a function f from the point-set of X to the point-set of Y is a *continuous transformation* from X to Y if for every $x \in X$, and N is a neighbourhood of $f(x)$ in Y, and $f^{-1}[N]$ is a neighbourhood of x in X.
- Two topological spaces are *topologically equivalent* (or *homeomorphic*) if there exists a continuous transformation between them that has a continuous inverse.

Manifolds

An *n-manifold* is a (Hausdorff) topological space where each point has a neighbourhood topologically equivalent to B^n.

An n-manifold can be embedded in any $\mathcal{E}^m$ provided $m \geq n$. The *manifold dimension* n may or may not be equal to m, which is the dimension of the space in which the manifold is embedded. For example, a 3-ball is a 3-manifold in $\mathcal{E}^3$ and its surface sphere is a 2-manifold in $\mathcal{E}^3$. The surface of a torus in $\mathcal{E}^3$ is an example of a 2-manifold which is not itself homeomorphic to B^2. A 2-manifold is more usually called a *surface*.

Let P be a p-manifold and let Q be a q-manifold, both in $\mathcal{E}^n$, where $p, q \in \{0, \ldots n\}$, and let C be a closed set in $\mathcal{E}^n$.

- $P - C$ is a (possibly empty) p-manifold.
- Front(P) is a closed set in $\mathcal{E}^n$.

- P is open in $\mathcal{E}^n$ if and only if $p = n$ or P is empty.
- $P \cap Q$ and $P \cup Q$ need not be manifolds even if $p = q$ (for examples see Chapter 5).

Topological spaces are often partitioned into manifolds of different dimensions for different mathematical reasons. For example, regular stratification is concerned with partitioning certain spaces into manifolds which are smooth (derivatives of arbitrary order exist) in order to handle discontinuities in a space. In contrast CW-complexes provide a different form of partitioning which is not concerned with smoothness but with reasoning about the topological structure of a space. Both these forms of partitioning are described below.

Stratification

A *stratification* of a space M (a sub-space of $\mathcal{E}^m$) is a partition of M into disjoint manifolds, such that the union of the manifolds of a given dimension and lower is a closed subset of M [Wall 1971]. A *stratum* is a path-connected component of one of the manifolds in the stratification.

A *Whitney* (or regular) *stratification* [Whitney 1965] also satisfies the following conditions:

- The stratification is *locally finite* (i.e. each point of M has a neighbourhood of points contained in a finite number of strata.
- For each pair of strata X and Y, for which $x \in X \cap \mathit{Closure}(Y)$, Y is *(Whitney) regular* over X at x.

This notion of regularity is concerned with singular points where the local topological structure could be different to that of neighbouring points in the same stratum (relative to a neighbouring stratum). Whitney regularity ensures that such points are in separate strata.

It is known that these two conditions imply the *frontier condition* [Mather 1970]: the frontier of each stratum is a union of lower-dimensional strata.

Semi-analytic sets

Not all point-sets are useful for the representation of geometric entities, but semi-analytic point-sets provide a reasonable domain [Requicha 1980].

A subset X of $\mathcal{E}^n$ is *semi-analytic* if for every point u of $\mathcal{E}^n$ there is a neighbourhood U of u, such that $X \cap U$ is in the Boolean algebra of sets generated from sets of the form $\{P \in U | f(P) > 0\}$ for $f : U \rightarrow \mathcal{R}$ analytic. ('In the Boolean algebra' means sets which can be formed using finite unions, finite intersections and complements.)

D̈jinn uses semi-analytic sets because they include a large class of sets which are useful in geometric modelling, and it has been proved that any semi-analytic set has a Whitney stratification into smooth manifolds. This provides a way to manage discontinuities. Furthermore, any semi-analytic set is known to be triangulable (i.e. it can be partitioned as a simplicial cell-complex [Lojasiewicz 1965]).

D̈jinn does not use point-sets defined by arbitrary functions: they may not have a frontier of lower dimension, and thus they may not be stratifiable or orientable. This makes it impossible rigorously to define the effect of geometric operations such as set-combination.

CW-complexes

- A *cell* (in a space R) is a sub-space homeomorphic to an open ball (more specifically, an *n-cell* is homeomorphic to B^n) of which the boundary is a union of cells (of smaller dimension). (This standard topological definition of a cell is not the same as the definition of the D̈jinn cell given later).
- A *cell-complex* is the union of disjoint cells, such that the boundary of each n-cell is contained in the union of lower dimensional cells.

This definition admits infinite complexes (infinite numbers of cells). Such complexes are more tractable if they are restricted by further conditions identified by Whitehead [Whitehead 1949]: a *CW-complex* is a cell-complex which is *closure-finite* (the boundary of each cell is the *finite* union of lower dimensional cells) and has *weak topology* (the closed sets in the complex coincide with the sets which have a closed intersection with every cell).

'Closure-finite' neither implies, nor is implied, by local finiteness: that is, the situation in which each point in the complex is in a finite number of closures of cells. Finite cell-complexes are automatically CW-complexes which are also locally finite.

Orientation

Any surface (2-manifold) can be partitioned as a cell-complex by a process of triangulation. 1-cells and 2-cells can each be oriented (in one of two ways). Two adjacent 2-cells are consistently oriented in a cell-complex if they impose opposite orientations on the 1-cells where they meet. If it is possible to orient all the 2-cells of cell-complex in a consistent way, then the cell-complex is said to be orientable. The orientability of any surface can be determined by the orientability of any cell-complex into which it can be decomposed. Clearly each path component of a 1-manifold is

orientable, so 1-manifolds are always orientable. A Möbius band is an example of a 2-manifold that is not orientable.

For smooth 1-manifolds in $\mathcal{E}^2$ and $\mathcal{E}^3$, an orientation can be represented by a tangent vector at each point in the manifold. For orientable smooth 2-manifolds in $\mathcal{E}^3$, an orientation can be represented by a vector normal to the tangent plane at each point in the manifold. These normals constitute a continuous vector field[2] (i.e. the function from points on the surface to their associated surface normal is continuous). Note that, for non-manifold curves and surfaces, there is a problem at points of self-intersection where it is not possible to give such a vector consistently. For non-smooth but manifold point-sets which are orientable, an orientation vector can be constructed for all points in a piecewise fashion from consistent orientations of smooth sub-manifolds produced by stratification and inducing a vector for points with derivative discontinuity by limits from either side.

Therefore, since it is often necessary to sort intersection points along a curve and applications often need tangent directions, D̈jinn curves are manifold and have a unique tangent direction at each point that is consistent with an orientation. This does not prevent the concatenation of curve strata, provided the result is manifold and orientable. The orientation is one of two possible orientations and is defined by a tangent direction at any point.

It is often useful to identify the two sides of a sheet object, and applications often need surface normals. Therefore, D̈jinn surfaces are manifold and have a unique normal direction at each point that is consistent with an orientation.

Surface strata may be joined, provided the result is manifold and orientable, thus admitting an orientation specified by a normal direction at any point. This eliminates the representation of a Möbius band as a single face, but it can be represented if the continuous loop is broken by two coincident frontier edges.

Surface normals are traditionally directed out of solid objects, but this is not an argument in favour of oriented surfaces because the inside of a solid can be identified by point classification.

Bounded sets

Some geometric entities are unbounded in any manifold embedding of the same dimension (e.g. a spherical surface and a plane). The traditional approach in the boundary modelling community is to contrast such unbounded constructional geometry with bounded faces and edges of topo-

[2]It is not possible to create a continuous frame field on all surfaces [O'Neill 1966].

logical structures that represent solid objects. In fact, constructional entities need not be unbounded: it is both useful and elegant to omit topological entities that represent artificial boundaries. For example, a spherical face does not require the bounding vertices and edges traditionally used for its representation by boundary modelling systems.

D̈jinn abandons the distinction between construction and topological entities and any curve, surface or solid may or may not be bounded. This necessitates the bounding operation, which instantiates any subset of a defined curve or surface as an independent entity. Bounding operations include the trimming of a parametric surface patch and the set-intersection of a surface and some (probably solid) bounding region.

Regularization (taking closures of interiors—see Chapter 5) is traditionally used to eliminate unwanted geometry (e.g. isolated edges and faces, and cracks in solids represented as closed sets [Requicha 1980]), but such entities are sometimes required. Regularization also coalesces point-sets with their frontier points. Separate identification of frontiers is achieved in D̈jinn by using geometry based on single points, arcs, surface patches and solids, each defined as a manifold:

- Three-dimensional solids that do not include their bounding faces, edges or vertices (3-manifolds).
- Two-dimensional surface patches that do not include their bounding edges or vertices (2-manifolds).
- One-dimensional arcs that do not include their end-points (1-manifolds).
- Zero-dimensional points which have an empty frontier (0-manifolds).

Applications require curve networks, surface and solid objects and their boundaries to be subdivided into independently accessible vertices, edges, faces and solids. Although boundaries and isolated point-sets of the same dimension have significantly different properties, they are defined in the same way in D̈jinn. Differences are captured by adjacency relationships (see Chapter 3) and within application data associated with D̈jinn geometry (see Chapter 3). Thus, vertices, edges, and faces of solids are subsets of points, arcs and surface patches respectively.

Continuity

Applications require the separate identification of vertices, edges and faces, but such entities may be defined in several ways. For example, an edge or face may be constrained to be connected or to have tangent continuity everywhere. However, a manifold point-set does not prescribe any degree of continuity. For example, it allows disjoint pieces and creases or cusps

in the interior of a face (i.e. lack of tangent continuity). Therefore, the definitions of an arc and a surface patch could reflect any of the following constraints:

- Analytical continuity, with each edge and face defined by a single equation.
- Topological (or C^0) continuity.
- Tangent plane (or C^1) continuity.
- Principal curvature (or C^2) continuity.
- Smooth (or C^∞) continuity.

Analytical continuity does not necessarily achieve any degree of derivative continuity. A single equation can define a surface with two sheets (e.g. $xy = 1$), or a cuspidal discontinuity, usually called an edge (e.g. $y^3 = x^2$). Surface patches may be defined in terms of geometry (e.g. the centre and radius of a sphere). Such patches inevitably have a single equation but since the D̈jinn interface hides specific representations from the application, this is not evident to a D̈jinn user. Patches may also be constructed from other patches using D̈jinn operations; the result is not necessarily a composite patch easily defined by a single equation. Therefore, nothing is to be gained from restricting patches to analytical continuity. A similar argument holds for arcs.

More than C^0 continuity forces each facet of a faceted object created in D̈jinn by an application (not by the D̈jinn faceting of Chapter 9) to be treated as a separate face. Restriction to higher degrees of continuity may make some interrogations more robust, but the D̈jinn specification does not presuppose any particular implementation techniques; it is the responsibility of each implementation to avoid algorithms sensitive to singularities. Higher degrees of continuity also eliminate patches with cuspidal edges that disappear within the patch, and patches with isolated points of discontinuity (e.g. a cone apex or points that satisfy $z = \log(x^2 + y^2)$ or $z = (x^2 + y^2)^{0.25}$).

Many existing representations of real objects (e.g. cars) have derivative discontinuities. For example, when an application moves a numerical representation from a Bézier-based CAD system to a system based on B-splines, degenerate B-splines are often used to ease the import of collections of badly matched patches with only C^0 continuity. This implies the need for D̈jinn to support (composite) patches with at least small discontinuities of slope.

Imprecise arithmetic could unexpectedly lead to entities consisting of disjoint pieces. Therefore, there is no restriction on the degree of derivative discontinuity of D̈jinn arcs, surface-patches or solids. They can contain cusps and can consist of disjoint pieces. The potential ambiguity (e.g. a

cube could be considered to have six faces, or just one) is avoided by D̈jinn functions to subdivide faces and edges into regions bounded by derivative discontinuities (see Chapter 3). Other D̈jinn functions combine across such boundaries to form a single entity that includes the discontinuities; or subdivide the boundaries themselves to form multiple entities, each having no internal discontinuities (see Chapter 2).

D̈jinn does not require even C^0 continuity, since significant gains in efficiency are sometimes possible by allowing a point-set to have multiple connected components.

Djinn geometry

This section summarizes the required properties of a D̈jinn object and thus defines the abstraction manipulated by the application. Applications can access object properties only if they form part of that abstraction or can be computed from it.

From software engineering considerations, it is preferable to implement a hierarchy of features or a graph representing constraints between pieces of geometry in terms of objects without such structure. However, this approach demands appropriate functions for modifying objects. D̈jinn provides such functions and does not explicitly support feature hierarchies or constraint graphs.

D̈jinn extends the traditional concepts of set-theoretic modelling systems to zero, one, two or three dimensions and admits non-manifold objects and objects of mixed dimension.

A point-set can consist of multiple disjoint components. Homogeneous point-sets are adequate for some applications that involve multiple objects, but they are inadequate for assemblies or partitions of a single object, because it is not possible to identify boundaries between subdivisions. Applications could provide support for partitions, but that would necessitate application structures that mimic those of the D̈jinn implementation. Therefore operations on partitioned objects are an integral part of D̈jinn.

Thus, mixed-dimensional objects with a cellular structure can be represented by D̈jinn objects consisting of a set of separately identifiable disjoint cells, each of which can have user-defined attributes.

Approximate geometry

Before providing formal definitions of D̈jinn cells and objects, it is necessary to consider the visibility to an application of the inevitable geometric approximations of an implementation.

One point-set may be used as an approximation to another in order to make a computation more efficient or easier to perform. For example, a set of planar *facets* might be used to represent a quadric surface or the boundary of a quadric solid. Each facet is represented by a polygonal boundary with implicit spanning geometry. For triangular facets the spanning geometry is planar, but for facets with more sides, it is planar only if the vertices are co-planar. In general, spanning geometry must be generated by transfinite interpolation.

An alternative approximation uses a voxel representation (i.e. a collection of lattice cells). Such an approximation of a solid is bounded by a collection of lattice cell faces. Individual surfaces can also be represented by such collections, and their boundaries approximated by a network of lattice edges. Approximate models may be used to assist computations on exact models (e.g. a voxel model could be used to increase efficiency by localizing operations on a more accurate model). Such approximations cause larger numerical errors than those inherent in exact models due to floating-point approximations of real numbers, but the form of the geometric results is unaffected.

This is not the case when approximate models are used as the primary representation of geometry; in that case, computations can be performed on an entity of the approximate model, an approximation to the corresponding entity of the exact model, or a reconstruction of that exact entity. For instance, a face of a voxel approximation to a solid bounded by quadric faces could be a voxel face, a set of voxels corresponding to an exact face, or a reconstruction of that face, made by some fitting process. From the application viewpoint there is no distinction, because all these data-types are hidden. Questions such as "What is a face?" apply to the abstraction visible to the application, not to the internal approximation of a D̈jinn implementation.

Thus, approximations are not visible at the D̈jinn interface, except indirectly when their effect on the precision of numerical computation exceeds that expected from limited floating-point precision. A D̈jinn implementation may compute a property directly from the approximate model or it may use a fitting procedure, but such operations are invisible to an application. Approximations become directly visible only when geometry is explicitly exported. For example, D̈jinn exports surface facets in order to allow an application to exploit graphics hardware.

The application must be able to control the accuracy of D̈jinn computations and exported geometry. Accuracy specified by the application when an object is created by D̈jinn only affects implementations that use approximate primary representations. Such approximation is reflected by subsequent interrogation for numerical properties such as volume. The

alternative taken by D̈jinn is to create nominally exact geometry and to allow the application to specify the desired interrogation accuracy. This may require the implementation to refine the approximate model until the result is available to the appropriate accuracy, but it hides the approximation more effectively and provides control of accuracy for individual interrogations.

When a D̈jinn implementation only uses approximate object representations, that approximation may or may not reference the implicit true geometry. For example, a model represented by voxels or facets might have a label for the true face associated with each voxel or facet. References to the true geometry enable the labels of that geometry to be returned as function results, for example the label of the closest face to a given point. If there are no such references in a particular implementation, then the D̈jinn functions that normally return a label of true geometry provide an error return. Approximations are not considered in the following definitions because D̈jinn object representations are hidden and are thus irrelevant, from the application viewpoint. Approximations that are exported for manipulation by external software (see Chapter 3) are specified in terms of exact geometry.

Djinn cells

A cell is the simplest part of an object that may be identified and manipulated at the D̈jinn interface. The following table provides some terminology for D̈jinn cells of different dimension:

Dimension	Cells with non-empty frontier	Cells with empty frontier	Frontier cells
0		point	vertex
1	arc	curve	edge
2	patch	surface	face
3	solid	solid (the whole space)	

A D̈jinn cell has the following properties:

- A D̈jinn cell does not contain its frontier. Arcs do not contain their end-points, patches do not contain bounding edges, and solids do not contain any bounding surface. Such entities are often called *relatively open*.
- A D̈jinn cell may or may not have an empty frontier. For example, a line segment has frontier points at its ends, but a complete spherical surface and a set of discrete points have empty frontiers.
- A D̈jinn cell can be a universal set $\mathcal{E}^n$ $(0 \leq n \leq 3)$, or it may be empty.

- □ Djinn curve and surface cells are oriented; a Möbius strip is not a valid cell.
- □ A Djinn cell need not be connected, but its components are homogeneous in manifold dimension. Djinn provides a function to subdivide disconnected cells into connected components (see Chapter 3).
- □ A cell can be piecewise analytic. Non-uniform rational B-spline (NURBS) surface patches and trimmed patches are valid cells, provided there are no self-intersections.
- □ There is no restriction on the degree of continuity of a cell. Thus an arc or surface patch may have derivative discontinuities in its interior (i.e. have a composite piecewise structure such as a spline curve).

Thus, Djinn cells are manifold point-sets in $\mathcal{E}^n$ $(0 \leq n \leq 3)$, and each cell has a well-defined (manifold) dimension. Djinn cells are not the same as the simple topological cells defined earlier because they need not be homeomorphic to an n-ball: for example, the surface of a torus could be a single Djinn cell but it is not homeomorphic to a sphere. Nor are Djinn cells equivalent to Whitney strata, because they may contain derivative discontinuities and thus violate the regularity condition.

Points are defined by coordinate triples, but cells of higher dimension are point-sets of infinite cardinality. The latter must be specified by a small finite set of parameters, such as the centre and radius of a sphere. These parameters constitute a (possibly non-linear) point mapping from a standard primitive point-set, such as a unit sphere at the origin (see Chapter 1):

$$\text{Parameters} \subset \mathcal{E}^3 \rightarrow \mathcal{E}^3.$$

If interactive, parametric and constraint-based design systems are to modify the geometry of a cell by changing the specific map that captures its shape and parameters, such maps must form part of the cell data abstraction at the Djinn interface. This implies externally perceived maintenance of the map over set-combination, i.e. maintenance of the set-expression used for construction. Since this would restrict implementations to the set-theoretic representation, the parameter map is not part of the Djinn cell definition and shape modification facilities are independent of cell construction sequence. Thus, shape modification can only be achieved by transformation (see Chapter 4) and tweaking (see Chapter 7).

In order that no particular cell representation is prescribed, Djinn cells are defined as a data-type *Cell* that captures the required properties, i.e. a type that is specified in terms of observer functions to provide a cell's spatial dimension (*SDim*), manifold dimension (*MDim*), point-set and orientation (*Orient*)[3]. These functions satisfy the following constraints:

[3]This does not imply that Djinn provides such functions as part of the interface.

$\forall C \in \mathit{Cell}$

$\mathit{SDim}(C) \in \{0, 1, 2, 3\}$.

$\mathit{MDim}(C) \in \{0, 1, 2, 3\}$.

$\mathit{MDim}(C) \leq \mathit{SDim}(C)$.

$\mathit{Point\text{-}set}(C) \subset \mathcal{E}^i$, where $i = \mathit{SDim}(C)$.

$\forall p \in \mathit{Point\text{-}set}(C) \bullet \exists N$, a neighbourhood of p in $\mathcal{E}^i \bullet$
$(\mathit{Point\text{-}set}(C) \cap N)$ is homeomorphic to B^j {manifold}
where $j = \mathit{MDim}(C)$, $i = \mathit{SDim}(C)$.

$\mathit{Point\text{-}set}$(C) is an oriented, semi-analytic manifold point-set.[4]

$\mathit{Orient}\,(C) \in (D \rightarrow \mathcal{R}^i)$ (continuous almost everywhere)
where $D = \mathcal{E}^i \cap \mathit{Point\text{-}set}(C)$ and $i = \mathit{SDim}(C)$.

$(0 < \mathit{MDim}(C) < \mathit{SDim}(C)) \Rightarrow (\forall v \in \mathit{Range}(\mathit{Orient}\,(C)) \bullet (\|v\| \neq 0))$.

The orientation vector is arbitrarily considered to be the zero vector for point cells and cells with equal manifold and spatial dimension, e.g. three-dimensional solids in three-dimensional space. Orientation has no meaning for empty point-sets.

Djinn objects

The point-set of an object is the union of the point-sets of its cells. Objects with non-manifold point-sets are created by the union of point-sets with common frontier points. Unlike the point-set of a cell, the point-set of an object can be non-manifold (i.e. a wire, sheet, solid, or a combination of these, containing points with a neighbourhood not topologically equivalent to an open ball of any dimension).

Since D̈jinn cell point-sets are manifold, the union of two disjoint cells that share a common frontier point is disconnected and manifold[5]. Therefore, a D̈jinn object consists of cells of which the union is a closed point-set. Thus, all frontier points of a cell's point-set are included in the union of point-sets of other cells in the same object. This prevents the explicit D̈jinn representation of objects with cracks, but the application can model cracks by assigning appropriate user-defined attribute to rank-deficient cells at the crack locations. Similarly, in an assembly represented as a single D̈jinn object, the contact area between two solid cells would have to be represented as a single face cell with an attribute.

Objects with non-manifold point-sets are created also by the union of rank-deficient point-sets with common points (e.g. two surfaces that cross;

[4]This does not imply that Djinn implementations must represent cells in terms of finite unions and intersections of analytic functions

[5]In practice, interrogation using floating-point arithmetic could indicate either a non-manifold cell or a cell with two disjoint components.

in D̈jinn, such operations create multiple cells each with a manifold point-set—see Chapter 5). Thus, a D̈jinn object consists of a collection of D̈jinn cells with disjoint point-sets.

Thus, a D̈jinn object is a stratification of a semi-analytic point-set that satisfies the local finiteness and frontier conditions; its cells are the strata. The strata are manifold but not necessarily connected nor smooth.

The regularity condition of a Whitney stratification [Whitney 1965] does not necessarily hold for D̈jinn objects, but such a stratification is theoretically possible because the point-sets of D̈jinn objects are semi-analytic sets. Formally, D̈jinn objects comprise a data-type *Object*[6] that admits the existence of observer functions to provide an object's cells (*Cells*), spatial dimension (*SDim*) and point-set. These functions satisfy the following constraints:

$\forall F \in Object$.

$Cells(F) \in Fin(Cell)$ (finite collection).

$SDim(F) \in \{0, 1, 2, 3\}$ (spatial dimension).

$\forall a \in Cells(F) \bullet SDim(a) = SDim(F)$ (consistent cell dimension).

$\forall a, b \in Cells(F) \bullet Point\text{-}set(a) \cap Point\text{-}set(b) = \emptyset$ (cells disjoint).

$\forall a \in Cells(F) \bullet \exists G \subset Cells(F) \bullet Frontier(Point\text{-}set(a)) =$
$\quad \cup_{c \in G} Point\text{-}set(c)$.

$Point\text{-}set(F) = \cup_{c \in Cells(F)} Point\text{-}set(c)$.

Labels and attributes

The identification of cells with labels is captured formally by requiring some interrogation functions in addition to those of the previous section. It is necessary to be able to access: the labels of all the cells of an object (cell-labels); maps from the labels to the cells (cell maps); the attributes of cells; and the cells from which the cells derive (parentage map). These functions must satisfy the following constraints:

$\forall F \in Object$.

$Cell\text{-}labels(F) \subset Label$.
$Cell\text{-}map(F) \in Cell\text{-}labels(F) \rightarrow Cells(F)$ (one-one map).
$Attribute\text{-}map(F) \in Cell\text{-}labels(F) \rightarrow (String \mapsto \mathcal{Z})$.
$Parentage\text{-}map(F) \in Cell\text{-}labels(F) \rightarrow$
$\quad \{((Label, Orient), (Label, Orient))\}$.

Cell attributes are integers accessed by means of a character string and a cell's parentage is a set of cell label pairs, each indicating cells from

[6]This type definition is extended in the next sub-section to accommodate cell labels (i.e. identifiers) and attributes.

which a cell was derived by a D̈jinn binary function. Derivation by unary functions yield the OUT label as the second element in all pairs.

Only the results of the most recent modification to the cell structure are recorded in the parentage map; they will be overwritten by subsequent D̈jinn operations which modify geometry, and the application must record more complex histories if required. This minimal interface does provide the data necessary to construct complicated histories.

Summary

A D̈jinn object may be regarded as having the following properties:

- A finite collection of disjoint cells.
- A point-set which is the union of the point-sets of its cells.
- A spatial dimension $\{0, 1, 2, 3\}$.
- A $1:1$ correspondence between labels and its cells.
- A map from its labels to its attributes.
- A parentage map from labels to sets of label-orientation pairs.

D̈jinn transformations are a slight generalization of the those already widely used. D̈jinn transformations may be viewed as movements, and they are therefore parameterized; they also have a spatial dimension, just as objects do. When a transformation of low spatial dimension is applied to an object of higher dimension, the leading coordinates of every point in the object are transformed: the remainder are invariant. Thus a D̈jinn transformation may be regarded as having the following properties:

- A spatial dimension $\{0, 1, 2, 3\}$.
- An invertible point map $\mathcal{E}^i \times [0.0 \ldots 1.0] \leftrightarrow \mathcal{E}^i \times [0.0 \ldots 1.0]$.

Part I
1
Primitive objects and copying

Visibility of geometric representations

The data types described in this chapter are divided into *simple data*, with a visible geometric representation, and *hidden data types*, with a representation that is only known to the modeller under the API. There is also externally defined geometry, but this is dealt with in Chapter 9.

Points and displacements

Both *points* and *displacements* are represented by tuples of n real numbers (where n corresponds to the dimensionality of the space), but they have different semantics to each other; for example, they behave differently under projective transformation, and points cannot be summed. This behaviour might be captured by distinct hidden types for points and displacements, together with a set of manipulation functions such as:

- Create a point or displacement from n real coordinates.
- Extract the real coordinates from a point or displacement.
- Addition: $\mathit{Point} \times \mathit{Displacement} \rightarrow \mathit{Point}$.
  $\mathit{Displacement} \times \mathit{Displacement} \rightarrow \mathit{Displacement}$.
- Subtraction: $\mathit{Point} \times \mathit{Point} \rightarrow \mathit{Displacement}$.
  $\mathit{Point} \times \mathit{Displacement} \rightarrow \mathit{Point}$.
  $\mathit{Displacement} \times \mathit{Displacement} \rightarrow \mathit{Displacement}$.
- Compute the inner and outer products of displacements.
- Multiply a displacement by a scalar.
- Compute the magnitude of a displacement.

However, this set of operations is inevitably inadequate; the required set is extremely large and unpredictable, including operations such as finding

the distance of a point from a line etc. Therefore, in order that the writers of applications can create the set of operations that they need, as well as for efficiency and simplicity, point and displacement representations are made visible at the D̈jinn interface.

Thus, the manipulation of points and displacements becomes primarily the responsibility of the application, although similar facilities will inevitably be found embedded within D̈jinn implementations. D̈jinn operations on points and displacements are restricted to:

- The use of points to interrogate D̈jinn objects (e.g. to compute the closest vertex to a given point—see Chapter 8).
- Special cases of more general D̈jinn functions. For example, the Minkowski sum (see Chapter 2) of a D̈jinn point object with a D̈jinn line object, which creates a parallel line through the point.
- The creation of a D̈jinn object consisting of a single point cell. (See the section on *simple primitives* later in this chapter.)

It is necessary to distinguish carefully between this last entity, which is a D̈jinn object with all the accompanying cell, label and other structures, and a simple point represented by three real numbers.

However, as a result of the decision to allow D̈jinn to deal with geometry at infinity, a further level of complexity has to be introduced. D̈jinn also has a hidden representation of points (and planes) which handles points at infinity, and functions are provided to convert between the visible and hidden representations. Going from the visible representation to the hidden one is achieved using the function:

Create point	Given the Euclidean coordinates of a point, create a point of hidden type from them; an extra argument allows the construction of points at infinity.

The inverse of this function enables the coordinates to be recovered from a point of hidden type:

Get point coordinates	Given a point of hidden type, return its Euclidean coordinates; an extra argument is used to indicate a point at infinity.

Unbounded straight lines and line segments

If the representations of lines were hidden, the intersection of a line with a D̈jinn object would require functions to create two points from their coordinates, to create a *line segment* from the two points, and to extract an *unbounded line* as the embedding of the line segment. Since the hiding

of line-representations incurs the same disadvantages as hiding points, D̈jinn allows line segments and unbounded lines to be created indirectly from pairs of hidden D̈jinn points. There is no other hidden D̈jinn line representation.

In other words, lines are created and interrogated by means of points with hidden representations; but the application can convert these from and to visible representations, as necessary. This leaves the application responsible for most operations on straight lines. D̈jinn operations on lines are restricted to:

- The use of lines to interrogate D̈jinn objects (e.g. to compute the intersection of a ray with a D̈jinn object—see Chapter 8).
- The creation of a D̈jinn object consisting of a single line-segment and a Minkowski sum extrusion (see page 369).

Planes

Since the hiding of plane representations would incur the same disadvantages as the hiding of point types, D̈jinn allows a *plane* (and, more generally, a hyper-plane) to be specified in terms of its coefficients, represented as a vector of dimensionality appropriate to the space in which the plane is embedded. (Actually, these are planar half-spaces, rather than planes: see the following section on simple primitives, and also Part II.)

D̈jinn operations on planes are restricted to:

- The use of planes to interrogate D̈jinn objects (e.g. to compute a planar section—see Chapter 8).
- The creation of a D̈jinn object consisting of a single planar half-space.

The hidden representation of a plane (or, more generally, a hyper-plane) may be constructed from its visible representation using:

Create plane	Given a set of Euclidean plane coordinates, create a hidden representation of the linear open half-space bounded by this plane.

The inverse of this function allows the recovery of the Euclidean coordinate description of a plane (or hyper-plane) from its hidden representation:

Get plane coordinates	Given the hidden representation of a planar half-space, return the visible representation of the Euclidean coordinates of its bounding plane.

Summary of points, lines and planes

In order to allow applications to build the range of functions they need, there is a visible representation of points, lines (as two points) and planes. There is also a hidden representation of points and planes with conversion functions between the visible and hidden types. This allows convenient handling of geometry at infinity.

Djinn functions which need points, lines and planes explicitly as arguments, such as the intersection of a ray with an object, use the hidden representation of points and hyper-planes, as appropriate. Apart from these cases, general Djinn objects must be created from hidden points and planes if they are to be used further; the functions for doing this will be discussed shortly (see the section on *simple primitives* later in this chapter).

Creation and destruction of Djinn objects

Variables of the type *DjObject* are assigned values when new objects are created by Djinn functions. This destroys previous values of those variables, but since the data representation is hidden, a Djinn implementation cannot avoid memory loss unless variables are assigned initial values on declaration. Since this is not possible in all languages, the following function is provided for applications to reclaim memory in a representation and language independent way:

Create empty object	Changes a Djinn object into an object with no cells.

Simple primitives

Simple primitives are created in general position and there are no special-purpose shape-modification functions required or available.

Points

A *point* object is a Djinn object created by the function:

Create point object	Given an n-dimensional point (of hidden type), create a single-celled Djinn object, consisting of the same point, and return the cell label.

Unbounded straight lines and line-segments

A *line-segment* is created as a D̈jinn object consisting of a single cell, by the function:

Create line object — Given two distinct points (of hidden type), create a two-celled D̈jinn object, consisting of one cell containing the end-points and a second cell which is a single n-dimensional open straight line-segment, and return the cell label of the latter.

The manifold dimension of the cell of which the label is returned is always 1.

Hyper-planes (planes, lines and semi-infinite intervals)

An open linear half-space bounded by a *hyper-plane* with coordinates H is the set of points with homogeneous coordinates satisfying the dot-product inequality $p \cdot H < 0$. Such half-spaces include a three-dimensional half-space bounded by a plane, a two-dimensional half-plane bounded by a straight line and a semi-infinite interval of the real line. D̈jinn provides a general-purpose function to create all such half-spaces embedded in three-dimensional space:

Create plane object — Given an n-dimensional hyper-plane (of hidden type), create a two-celled D̈jinn object consisting of the boundary plane and a second cell which is an n-dimensional half-space bounded by the given hyper-plane and return the cell label of the latter.

The spatial dimension of the object is that of the half-space, and this dimension is also the manifold dimension of the cell returned.

Enquiries

For each type of object introduced above, an enquiry function is also provided to find the point, line or plane in the case where a D̈jinn object is a simple primitive of this type:

Get point — Given a single-celled D̈jinn object which contains just a single point, return that point as a point (of hidden type).

Get line segment — Given a D̈jinn object which only contains a line-segment, return the end-points of that line as points (of hidden type).

Get plane Given a D̈jinn object which only contains a planar (one-, two- or three-dimensional) half-space, return the defining plane as a plane (of hidden type).

Canonical primitives

Canonical primitives have 'natural' unit forms that define geometry in a space of any dimension. They are created in a standard position and orientation and there are no special-purpose shape-modification functions. Instead, their shape, like that of any any D̈jinn object, may be transformed (see Chapter 4) and interrogated (see Chapter 8).

Simplices (tetrahedra, triangles and intervals)

Simplices can have any dimension and be defined in a space of any dimension, but the use of standard position reduces the number of parameters required. Thus, D̈jinn provides a general-purpose function to create a solid tetrahedron, a triangular region in the xy-plane, and a segment of the x-line, with one of its vertices at the origin and the others at unit points on each of the n coordinate axes:

Create simplex Given the dimension n of a simplex, create a two-celled D̈jinn object consisting of, firstly, a unit simplex and, secondly, its boundary and return L the cell label of the former.

The dimension of the simplex defines its point-set: the object has spatial dimension n, which is also the manifold dimension of the simplex cell L. Arbitrary simplices may be created by subsequent transformation.

A triangle and a tetrahedron are obviously specified by their vertices; the intersection of planes is the dual specification. It is easy for an application to use such specifications by creating affine transformations from such data and transforming the canonical simplex (see Chapter 4).

Simplex vertices and faces can be retrieved using D̈jinn boundary interrogation functions (see Chapter 3).

Cubes, squares and intervals

Cuboids are useful as primitive objects and, when aligned with the coordinate axes, as enclosure geometry (bounding boxes).

D̈jinn provides a general canonical function for creating n-dimensional unit *'cubes'* centred on the origin (including xy-rectangles and x-intervals):

Create cube Given the dimension n of a hyper-cube, create a two-celled D̈jinn object consisting of a unit hyper-cube in one cell and its boundary in the other, and return the cell label of the former.

The dimension of the cube defines its point-set (i.e. n is the spatial dimension of the object and also the manifold dimension of the returned cell). Rectangular boxes with sides of different lengths are created by subsequent differential scaling (see Chapter 4).

Spheres, circular discs and intervals

D̈jinn provides a general-purpose function to create a unit *hyper-sphere* (i.e. a solid sphere in three dimensions, a disc in the plane, a line-segment or a point):

Create sphere Given the dimension n of a hyper-sphere, create a two-celled D̈jinn object, consisting of a unit-radius hyper-sphere centred at the origin in one cell and its boundary in the other, and return the cell label L of the former.

The dimension of the sphere defines its point-set (i.e. n is the dimensionality of of the object and also the manifold dimension of the returned cell). Spheres of arbitrary size and hyper-ellipsoids may be created by subsequent differential scaling.

Special-purpose primitives of fixed dimensionality

Special-purpose primitives do not generalize to varying numbers of dimensions, and are common in current solid modelling systems. These primitives are created in a standard position and orientation and are of unit size; nevertheless, they also have shape parameters.

The shape of special-purpose primitives may be changed, as before, using transformations (see Chapter 4) and interrogated (see Chapter 8). But changes to shape parameters require special-purpose shape-modification functions, which can only be applied if the primitive is the sole constituent of an object and has not already been transformed (otherwise it cannot be guaranteed that the primitive remains the same type of object).

Cylinders and cones

A finite right circular *cone*, possibly truncated, is conveniently defined by the tangent of the apex half-angle. This formulation supports the cylinder

as an obvious special case; however it does not prevent the creation of a double cone extending on both sides of the apex. D̈jinn provides a function to create a unit conical frustum using this parameter.

The axis of the cone is the z-coordinate axis and the top and bottom planes of the frustum are equidistant from the xy-plane. The height of the cone and the diameter of its xy-plane cross-section are both unit length. The free shape parameter is the tangent of the apex half-angle.

Create cone — Given c, the tangent of the cone apex half-angle, create a multi-celled D̈jinn object, consisting of a conical frustum in one cell and its boundary in another and, when appropriate, its apex in a third cell; and return the cell label of the frustum.

The returned cell has the point-set:

$$\{\langle x, y, z\rangle \in \mathcal{R}^3 | (4(x^2 + y^2 < (1 - 2cz)^2) \wedge (-1 < 2z < 1)\},$$

where c is the tangent of the cone apex half-angle. The shape will of course vary depending on the value of c, and it is also possible that a third cell may be created:

$c = 0$	the cone is a right circular cylinder.
$\|c\| = 1$	the cone apex is part of the boundary cell.
$\|c\| > 1$	the cone apex constitutes a third cell.
$0 < c < 1$	the top diameter is less than the bottom diameter.
$0 > c > -1$	the top diameter is greater than the bottom diameter.

Cones of arbitrary size and those with elliptical cross-section can be created by differential scaling.

Wedges, prisms and pyramids

A regular *pyramid* can be inscribed by a right circular cone. For consistency, it is specified using parameters analogous to those of the primitive cone, but with an additional parameter determining the number of sides. Therefore, D̈jinn provides a function to create a unit pyramid inscribed in a D̈jinn unit cone:

Create pyramid — Given c, the tangent of the cone apex half-angle, and n, the number of sides of a three-dimensional regular pyramid, create a multi-celled D̈jinn object, consisting of a pyramidal frustum in one cell and its boundary in another and, optionally, the apex in a third cell; and return the cell label of the frustum.

Regular *prisms* can be created as pyramids with zero apex angle. Triangular prisms can be also be generated by sweeping (see Chapter 6) a

triangular simplex.

The pyramid cell has the point-set defined by the set-intersection of planes:

$$\bigcap_{i=0}^{n-1}\{\langle x,y,z\rangle \in \mathcal{R}^3 | (-1 < 2z < 1) \wedge (x\cos\frac{2\pi i}{n} + y\sin\frac{2\pi i}{n} < 1 - 2cz)\}.$$

Certain non-regular prisms can be created by differential scaling.

Tori and cyclides

A torus is the volume formed by sweeping a sphere about an axis. If the sphere changes radius as it is swept, such that the sphere remains just touching two given circles in the plane through its centre and perpendicular to the axis, the swept volume is a *cyclide* [Dutta 1993].

If one of the circles lies inside the sphere, the cyclide is a *spindle cyclide*; its surface is self-intersecting and there is no central hole. Points in the region of overlap inside the 'spindle' do not constitute part of the solid in the point-set of a Djinn cyclide.

If the two circles intersect, the cyclide is a *horned cyclide*, also with two points of self-intersection.

These two circles are not a particularly convenient way of specifying a cyclide. D̈jinn uses instead that the centre of the moving sphere lies in the xy-plane, with symmetry about the xz-plane. The cross section of the cyclide by the xz-plane then consists of two circles whose centres are at [1,0,0] and [-1,0,0]. The shape of the cyclide is then defined by the diameters, d and e, of these two cross-section circles.

If the two circles overlap ($d + e > 4$), the cyclide is a spindle cyclide; a horned cyclide can be defined by specifying one of the diameters to be negative. If d and e are equal, the cyclide is a torus.

This specification form is convenient for defining torus-like cyclides. It does not support the creation of degenerate cyclides (i.e. planes, spheres, cylinders and cones) or the third order cyclides.

Create cyclide — Given the diameters of the two zx-sections, create a multi-celled D̈jinn object consisting of a cyclide and its boundary and return the cell label of the former.

The point-set of the cyclide is given by:

$$\{\langle x,y,z\rangle \in \mathcal{E}^3 | \exists b,c,m \in \mathcal{R}\bullet \\ (x^2+y^2+z^2-m^2+b^2)^2 < 4(x-cm)^2 + 4b^2y^2 \wedge \\ (4m = d+e) \wedge (4c = d-e) \wedge (1 = b^2+c^2)\}.$$

Helices

The volume created by sweeping (see Chapter 6) an orthogonal circular, triangular or square cross-section along a *helix* is ubiquitous in engineering components for shapes such as screw threads and springs. Therefore, D̈jinn provides a function to create, in two dimensions, one turn of a plane spiral or, in three dimensions, one turn of a tapered helix on the surface of a D̈jinn conical frustum (i.e. a helix about the z-coordinate axis of unit pitch and diameter in the xy-plane):

Create helix Given a number of dimensions, n, and c, the tangent of half the apex angle of a three-dimensional conical frustum, create a two-celled D̈jinn object consisting of: one cell containing a one turn of a spiral (in two dimensions), or one turn of a tapered helical curve of unit pitch and diameter in the xy-plane, rotated about the z-coordinate axis (in three dimensions); and a second cell which is the boundary of the first. Return the cell label of the first cell.

A spiral or helix with an arbitrary number of turns can be created using D̈jinn functions to copy, scale, translate and union D̈jinn objects. Differential scaling can be used also to change pitch and diameter.

The point-set of the helical cell is given by:

$$(n = 2) \Rightarrow \{\langle x, y, z\rangle \in \mathcal{R}^2 | \exists\theta \in (-\pi \ldots \pi) \bullet \\ (2x = (1 - c\theta/\pi)\cos\theta) \wedge (2y = (1 - c\theta/\pi)\sin\theta)\}.$$

$$(n = 3) \Rightarrow \{\langle x, y, z\rangle \in \mathcal{R}^3 | \exists\theta \in (-\pi \ldots \pi) \bullet \\ (2x = (1 - 2cz)\cos\theta) \wedge (2y = (1 - 2cz)\sin\theta) \wedge (2z\pi = \theta)\}.$$

NURBS

Nearly all current commercial CAD systems support non-uniform rational B-spline curves and surfaces (*NURBS*) [Piegl 1997]. Therefore, for pragmatic reasons, D̈jinn cannot omit them. The definition here is based on the data model to be found in ISO 10303 (STEP), although to be consistent with the D̈jinn object concept the following changes have had to be made:

- Closure and self-intersection are geometric properties, which can (and must) be determined by the point-set of the curve, which is itself determined entirely by the knots, control-points and weights. The 'flags' specified in STEP are therefore omitted; even within the STEP document these are labelled as being "for information only".
- D̈jinn does not have list structures distinct from arrays.

In fact, D̈jinn uses sequences as the relevant abstraction: arrays and lists are relevant at the binding level only. Sequences are therefore used rather than lists as the primary structure for the control points, weights and knots. Furthermore, in order to conform to the philosophy of minimality, the default variants for uniform, quasi-uniform and multi-Bézier forms are not provided by D̈jinn. However, it is easily possible for applications to build such functionality using D̈jinn functions.

Note that only the underlying curves and surfaces are specified in the STEP format. D̈jinn has its own approach to defining trimming, which applies to NURBS in exactly the same ways as to any other curves or surfaces.

Two D̈jinn functions are defined:

Create NURBS curve — Given the knot, control point and weight data according to *ISO 10303 (STEP)*, create a D̈jinn object usually consisting of two cells: a *NURBS* curve in one and (usually) its end-points in another, and return the cell label of the curve. If the curve is closed, the resulting D̈jinn object has only one cell.

Create NURBS surface — Given the knot, control point and weight data according to *ISO 10303 (STEP)*, create a D̈jinn object usually consisting of two cells: a *NURBS* surface in one and (usually) its boundary curve in another, and return the cell label of the surface. If the surface is closed, the resulting D̈jinn object has only one cell.

Maintaining the modification record

The modification record contains the number of times that an object has been updated, and the region affected by those updates. In order for this facility to be fully useful, the application must be able to reinitialize the modification record.

Reset modification record — Given an object, reset its modification count to zero and its update region to the empty set.

Copying objects

The functions for copying objects, for creating shells for solids, and solids from shells may be considered as the most basic operations after the cre-

ation of primitives. The simplest of these functions makes a copy of an existing object:

Copy	Given an object, create another object with the same cellular structure, labels and attributes.

The copy is a replica in the sense that no data is shared between the new and original objects. That is all there is to it.

Part I
2
Partitioning and related operators

The cellular structures of D̈jinn objects are recognized by D̈jinn functions, and operations such as set-combination, sweeping or tweaking should modify them appropriately. It is often, however, necessary for applications to be able to modify the cell structure, using explicit partitioning and departitioning operations.

Initially, we discuss the rationale for the D̈jinn operators which partition simple objects into multiple cells, and which dissolve partitions so as to reconstitute homogeneous objects with a single interior and boundary cell. We continue by defining the D̈jinn partitioning and departitioning operators.

The rest of the chapter describes the range of functions available for manipulating the cell structure of D̈jinn objects. In general, D̈jinn objects consist of multiple cells when they are created. Most, but not all, primitive objects (see Chapter 1) are created with two cells: one corresponding to the interior and the other to the boundary. Other D̈jinn operations create objects with more sophisticated divisions into many cells. Here, a function is described which allows an application to partition boundaries using geometric continuity as the segmentation criterion.

Partitioning cellular objects

Any partitioning of a D̈jinn object must maintain the frontier condition. A convenient way of articulating such splitting is to use a second D̈jinn object as a 'pastry cutter', which imprints its own structure on to the first object. The resulting object has all cell boundaries from both operands in their region of overlap.

Using operators $\cap^\circ$ and $-^f$ from Chapter 5, the partitioning operator

can be defined as:

$$
\begin{aligned}
\#\in & Object \times Object \rightarrowtail Object \bullet \\
& \forall F, G \in Object \bullet SDim(F) = SDim(G) \bullet \\
& (SDim(F\#G) = SDim(F)) \\
& \wedge (Cells(F\#G) = Cells(F \cap^{\circ} G) \cup (F -^{f} G).
\end{aligned}
$$

Informally, the mathematical description above is of an operator #, which takes two objects and produces another object, such that for any objects F and G of the same spatial dimension, the spatial dimension of $F\#G$ is also that dimension and the cells of $F\#G$ are the cells created by the Djinn object intersection $F \cap^{\circ} G$, defined in Chapter 5, united with the collection of cells (not an object) produced by the difference $F -^{f} G$, also defined in Chapter 5. $F -^{f} G$ is formed by subtracting all cells of G from each cell of F. Using this expression, the polygonal object with a single interior cell in Figure 4(a) is subdivided by the curved object to create the six-celled object shown in Figure 4(b), with two interior cells, two boundary cells, the separation cell and one cell with two unconnected components containing its end-points.

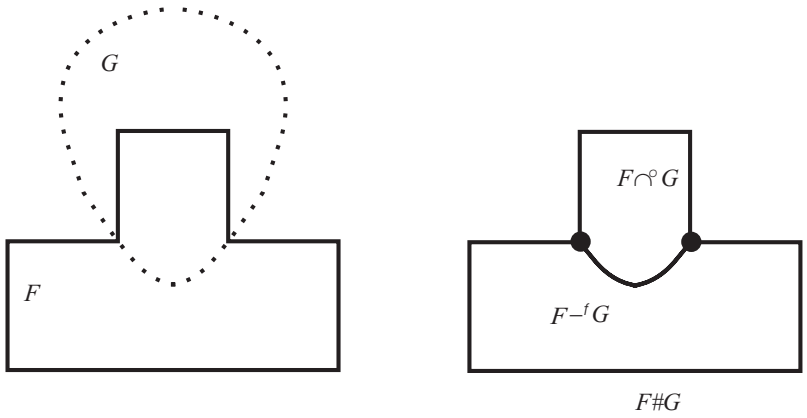


Figure 4: Subdivision of one object by another.

If the partitioning object in Figure 4 were a single one-dimensional elliptical cell, then the result of the partitioning would have the same separation cell; but the only cells in the object would be the original polygon, the internal curved edge which separates the two disjoint pieces, the polygon's boundary, (now also consisting of two disjoint pieces), and the end-points of the elliptical edge.

An object such as that with three interior cells in Figure 5(a) would be partitioned by the dashed object with three interior cells to give the object with seven interior cells in Figure 5(b). Note that in this case $F -^{f} G$ is empty.

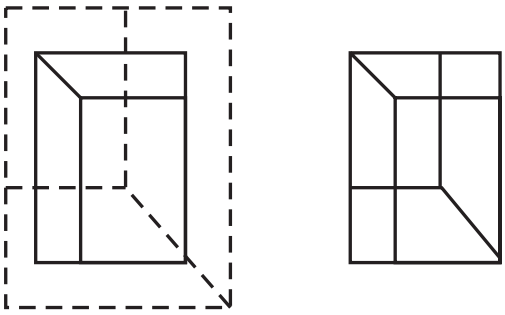


Figure 5: Example partitioning of one 2D cellular object by another.

Departitioning cellular objects

In order to support the application's detailed control over the cellular structure of an object, this function allows a number of cells to be replaced by a single cell, whose point-set is the union of the point-sets of the selected cells.

Clearly the new cell must satisfy all the conditions for cellhood, being open and manifold and bounded by the union of other cells of the object. Its creation must also not violate the frontier condition of any cells which its original components bounded.

Thus for example, two cells separated by a cell of one dimension lower can be combined into a single cell, provided that they originally bounded only the same cells of higher dimension (see Figure 6(a)).

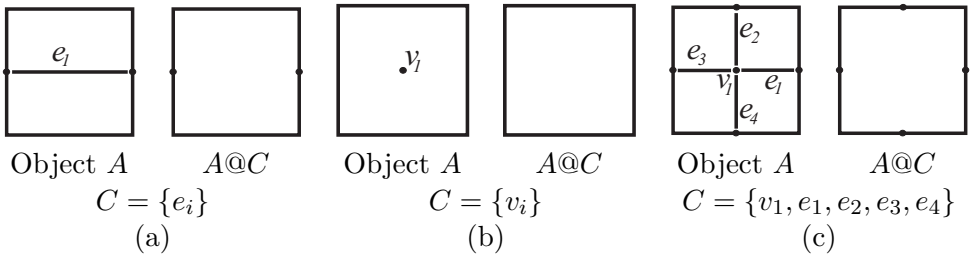


Figure 6: Departition of cellular objects.

Also an internal bound of dimension two or more lower can be combined with the cell that surrounds it by putting both into the input set (see Figure 6(b)).

The cells in the given set are processed in descending sequence of manifold dimension, so that a set consisting of an edge, and the faces and solids which meet at it can be combined into one. Logically the faces are removed first, combining with the solids to form a single solid surrounding the edge, which is then in turn combined into the final cell (see

Figure 6(c)).

This functionality is denoted by the departitioning operator @. Thus $A@C$ denotes the object in which the collection of cells identified by the set of cell labels C is replaced by a single cell in so far as that is consistent with the rules for cellularity.

Note that it is possible within the rules for cellularity to construct cells with more than one connected component. If several departitionings are intended to result in separate components as separate cells, the function should be called separately for each.

Partitioning and departitioning functions

There are two functions provided for explicitly changing the subdivision of the point-set of a D̈jinn object into cells whilst leaving the point-set itself unchanged. They are as follows:

Partition — Given two objects, partition the cells of the first using the cells of the second.

This function implements the partitioning operations described above, and thus it provides a means of explicitly controlling the subdivision into cells of the first object.

The reverse effect can be achieved using the function:

Departition — Given an object and a set of cell labels, remove the partitioning of the point-set of the object, under the control of the cell labels supplied in the second operand.

This function allows partitioning of the object to be selectively removed.

As explained in Chapter 1, the definition of D̈jinn objects incorporates a map which allows applications to track the fate of cells across the invocation of D̈jinn functions. The following table gives the parent sets for a cell immediately after each operations:

Operation	*Condition*	*Parentage set*
Partition (A, B)	OUT	$\{(\mathsf{OUT}, \mathsf{OUT}), \{(\mathsf{OUT}, i_B)\}\}$
	in A out of B	$\{(i_A, \mathsf{OUT})\}$
	in A in B	$\{(i_A, j_B)\}$
Departition (A, L)	OUT	$\{(\mathsf{OUT}, *)\}$
	in a merged cell	$\{(i_A, *), (j_A, *)\}$
	in an original cell	$\{(i_A, *)\}$

The notation $\{(i_A, j_B)\}$ implies that a given cell was derived from cell i of object A and cell j of object B, whilst $*$ indicates that no cell is relevant since the operation had only a single parent object. Note that the identifier OUT represents the space outside the object. The parents of this cell are all the cells that were destroyed by the previous point-set modification operation.

Partitioning boundaries

'Natural' partitionings of the boundary

A primitive D̈jinn object is created with two cells, the object interior and its complete boundary. However, applications require the ability to separately identify faces, edges and vertices. Boundaries can be subdivided by derivative discontinuities to give edge and face subdivisions that have such discontinuities at their boundaries and no internal discontinuities. Thus, a *geometric* face (or 'natural' face) is a maximal connected region of a surface that is either unbounded or bounded by discontinuities of tangent plane. A *geometric edge* (or 'natural' edge) is a maximal connected region of a curve that is either unbounded or bounded by discontinuities of tangent direction. If a curve is divided geometrically into edges, those edges are separated by geometric vertices; boundaries of surface patches can also consist of isolated vertices.

Depending on the application, the criterion for separation may be discontinuity of tangent plane, or discontinuity of some higher derivative. For example, for some purposes a pair of plane faces with a fillet rounding the edge between them may be considered as one face, for other purposes as three. D̈jinn therefore allows this partitioning process to be controlled by the specification of the order of derivative to be used. As a useful convention, the same operation is used to separate cells into their connected components by specifying a derivative order of zero.

D̈jinn partitioning by derivative discontinuity applies equally to cells forming internal membranes, isolated faces and edges.

Application partitionings of the boundary

Applications often require faces, edges and vertices that do not necessarily correspond one-to-one with the geometric faces, edges and vertices defined by tangent discontinuities. For example, a face cell might be partitioned into areas requiring different surface treatment. To indicate a point on a surface where a load is to be applied, an application can construct a surface cell which includes the point of loading as a cell in its frontier. A distinct

cell of an object can consist of any bounding point-set of homogeneous dimension. Such cells can be subdivided and combined using the D̈jinn partitioning and set-operations (see above and Chapter 5).

Boundary partitioning at discontinuities

A single D̈jinn function is provided to perform all variations of boundary partitioning using continuity as a criterion:

Subdivide boundary	Given an object and a set of labels of cells of that object and the required degree of discontinuity, divide up the cells in the set given and possibly others as described below.

The action of this function is powerful but complicated. Firstly, it only operates directly on cells in the label-set given, although this may result in consequential changes to other cells of the object. Secondly, its effect is determined by the degree of discontinuity specified. The minimum effect on an object will occur for the case where the level of discontinuity is set to zero; in this case only those specified cells of the object which span two (or more) connected components will be partitioned. Those cells which formed boundaries of these cells may also need to be subdivided as a result. Where a higher level of discontinuity is specified, further subdivisions may take place: for example, subdivision of specified face cells at tangent discontinuities. Again, further subdivision of other cells will be performed as necessary to keep the object correctly formed.

Clearly, the functions described in this chapter allow applications to perform a wide range of sophisticated manipulations of the way an object is subdivided into cells.

Draining and filling

Two simple functions are concerned only with the removal or instantiation of the interior of objects. Thus the *Drain* function removes all cells from an object that are not boundaries of another cell. For instance, given a spherical primitive object (which has two cells) *Drain* would remove the interior solid cell leaving a spherical shell.

The *Fill* function has the opposite effect, namely that of creating cells from their boundaries. Given a set of face cells within a three-dimensional object, *Fill* creates, and adds to the object, a single new cell. Its point-set includes all those connected components that are completely bounded by face cells which are in the set, and which do not go to infinity.

Drain Given an object, remove all cells that do not bound others and return the labels of the remaining cells.

Fill Given an object and the labels of the relevant cells, create a new cell bounded by the indicated cells and return its label.

An example of filling is shown in Figure 7. The original model consists of the union of three straight-line segments. The Figure shows the cells resulting from that union. The *Fill* operation creates a new triangular cell and also splits the centre sections out of the original lines and into new cells, so that the frontier condition can be maintained.

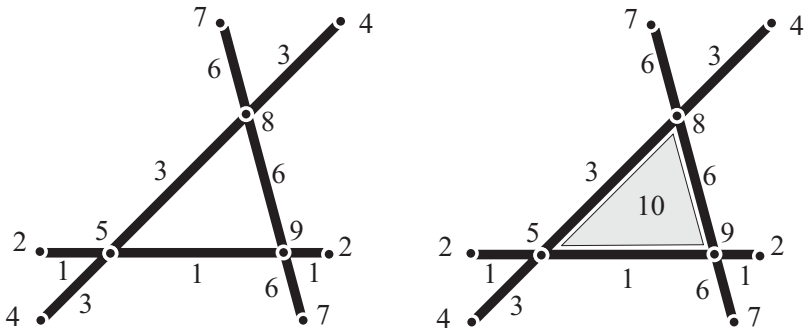


Figure 7: An example of filling.

Part I
3
Managing cells

This chapter describes the facilities in D̈jinn for managing the cell structure of D̈jinn objects: labels, attributes, and methods for selecting cells.

Labels

Labels provide the way in which an application can distinguish between the cells of an object. Each D̈jinn cell in an object has a label which identifies it uniquely among the cells of that object. Although much of the following is written as though the label belonged to the cell, it is probably nearer the truth to think of the cell belonging to its label. Since D̈jinn implementations have no perceived state, it is not possible for an implementation to ensure that the same label is not used for cells of different objects. A cell must therefore be identified by label 'x' of object 'y'.

Label assignment

Each object-creation function (see Chapter 1) assigns a value (the created object) to an object variable. During both the creation and modification of objects, labels are assigned by the D̈jinn implementation to the cells when they are created[1].

If label representations were visible to the application, it might be possible to assign label values implicitly. For example, integer labels could be assigned to the vertices of a cube using a specific geometric pattern. This approach would:

[1]The moment at which a label is *logically* created is specified as part of the Djinn interface, but this does not prevent lazy evaluation (i.e. implementation assignment of labels at the first enquiry).

- Reduce the need to obtain labels by object navigation and geometric queries.
- Provide related labels for similar parts of different objects, because labels would reflect the construction process. This property could be exploited to represent a family of parameterized parts.

This approach is not followed in D̈jinn because it requires the scheme for assigning labels to primitive objects to be published, together with the algorithm for the assignment of labels to derived cells (e.g. edges created during intersection). This, in turn, requires the label of a solid boundary to encapsulate the identifier of the solid which it bounds. For more complicated constructions, labelling would have to rely on some structure of objects that is visible to the application; candidates include the history of D̈jinn function invocations and a geometric order of new cells. However, either approach would create different labels in identical objects constructed in different ways [Kripac 1995].

The fact that labels are assigned by the D̈jinn implementation does not prevent the use of D̈jinn in applications involving families of parts. Parts of a single family can be defined in data-structures within the application, using application-specified identifiers that are invariant over that family in some appropriate way (e.g. from geometric queries). Alternatively, D̈jinn functions can be used to copy and modify the master part of the family.

Many D̈jinn functions access cells in the boundary of an object by navigation (Chapter 3) from the object's exterior. Therefore, this exterior is treated as a special case, with the label OUT.

Cell disruption

D̈jinn functions may cause the point-set of a cell to disappear or the cell itself may become redundant (i.e. it is certain that it will not be referenced again).

Even if D̈jinn functions destroy cells only when it is obvious that they should be destroyed, it may not be convenient for an application to update its own data-structures immediately destruction occurs. To allow for this, a 'cell exists' pre-condition might be attached to all functions that access cells, with an additional function provided to check the existence of a cell. However, for ease of use, this approach has been rejected in favour of error returns from access functions. Thus, all functions that access cells have a 'cell does not exist' error return.

The ability of an application to manage its own identifiers also requires that, if a cell's point-set disappears during geometric modification, that cell should become an empty cell with the same label. Note that labels

identifying empty cells are not the same as labels that do not identify any cell at all.

When a single cell is subdivided into multiple cells and that initial cell is destroyed, new and unique labels are assigned to the resulting cells. However, an application may want to access the new cells from the label of the initial cell. For example, if a face is subdivided into smaller faces, it may be convenient to use the label of the initial face to access the collection of sub-faces. Djinn supports such operations by maintaining a map from which the parentage of cells can be discovered.

The parentage map is useful when two objects are combined (e.g. by set-operations). A cell derived from a single operand cell inherits that operand's label, if possible (i.e. unchanged cells remain accessible with the same label); but, if necessary, a new unique label is assigned to each such cell to avoid label conflict. A new label is always assigned to any cell of the resulting object which is derived from two cells, one from each operand.

Since a cell is often the parent or child of more than one other cell, it is necessary to introduce the concept of *collections of labels* to support the functions for handling parentage maps. The other necessary concepts, of label-sets and label sequences, are explained later in this chapter.

In fact, several functions are provided to discover the effects of Djinn operations on cells of objects. The first two of these give access to the mapping between cell labels before and after a Djinn function call. They are:

Child cells — Given a Djinn object, a label and a Boolean, return a set of labels which refer to cells derived from the given cell.
The Boolean identifies a given cell as part of the first or second operand. If the original cell has disappeared, the returned set contains the single special label `OUT`.

Parent cells — Given a Djinn object and a label, return the two sets of labels of cells from which the given cell is derived.

This function enables the application to determine the parent cell or cells of the given cell, even though those cells are no longer part of the object. Again, in appropriate circumstances, `OUT` may appear as a parent.

Two related functions enable the application to ascertain which cells have been modified:

Modified cells Given a Djinn object, return the set of labels of cells that have been geometrically modified by the most recent call to a Djinn geometry-modification function. The definition of 'most recent' excludes functions which never modify geometry (i.e. read-only functions and functions which change only labels or attributes).

It is also possible to determine from which of the operands of the last function (for example, a Boolean combination) the cells of an object were derived:

Modifying cells Given a Djinn object, return two sets of cell labels, one for each operand of the most recently called geometry-modification function, from which the geometrically modified cells of the object are derived.

The above set of Djinn functions provides the application with a comprehensive set of tools to track the fate of the cells of an object across Djinn function invocations.

Label representation

Labels are cell identifiers with values assigned by Djinn implementations. Maps from labels to cells are stored as part of a Djinn object. Labels may also be stored in application variables and may be copied between those variables. There is no need for an application to know the value a of cell label; only its association with a cell is necessary. Therefore, the label representation is hidden from the application and thus becomes a matter to be resolved in a particular implementation.

It might be desirable from the application programmer's point of view to use 'meaningful' character strings as labels. This cannot be done because it prevents label assignment beneath the API. The effect desired in applications can however be achieved since such strings can be accessed through cell attributes, by means of a symbol table or some other data-structure maintained by the application.

The need for Djinn data to persist over a cycle of storage and retrieval precludes the use of data addresses (pointers) for labels. A unique (information-preserving) cast from pointers to integers could be used for storage, but the integers would need to be mapped to different pointers after retrieval. That implies that maps must be used for all accesses to cells and negates the advantage of pointers (i.e. direct access to data). Since the representation of labels is hidden, Djinn includes functions to

retrieve individual labels from label sets.

Label-sets and sequences

D̈jinn provides two concepts for handling collections of labels: these are sets and sequences. A *label-set* is a collection of (zero or more) labels without any repeats and thus it is a set in the mathematical sense. A *label sequence* is a collection of (zero of more) labels within which a particular label value may occur an arbitrary number of times. A label may be added to a label set:

Add cell to label set — Given a label-set, add the given label to it.

D̈jinn provides functions to convert between sets and sequences:

Label set to sequence — Given a label-set, return the equivalent label sequence.

In essence, this function merely provides type-conversion, while the following function also removes any repeated values.

Label sequence to set — Given a label sequence, return the equivalent label-set.

Both label-sets and label sequences are of hidden type and thus D̈jinn includes functions to manipulate them. Functions are provided to perform set-operations on labels sets:

Unite label sets — Compute the union of two label-sets.

Intersect label sets — Compute the intersection of two label-sets.

Subtract label set — Compute the difference of two label-sets.

A label-set or label sequence may be 'emptied':

Delete label set — Empty a label-set.

Delete label sequence — Empty a label-sequence.

The number of labels in a label-set may be determined:

Get cardinality — Get the number of labels in a label-set.

To support label sequences in addition to label sets, the capability is needed to access a label at a given position in the sequence:

Get label — Get the label at a given position in a given position in a label sequence.

Attributes

The need to associate application information with geometric models has long been recognized, and today it is common to find models which have values for properties such as density, colour and surface finish incorporated in their data-structures. These values are known as *attributes*. Since any element of a model may have many attributes, it is customary to view them as key-value pairs (i.e. the value `green` might be associated with the key `colour`).

The classic difficulty with attributes is defining how they persist under modifications of cells. However, the map of changes in a Djinn object that have occurred during the immediately preceding operation, gives the application the opportunity to update its attributes.

In principle, Djinn could maintain a single attribute for each cell, leaving the application to handle the mapping from a single Djinn attribute to multiple-attribute key-value pairs (application data). However, in a world of Djinn-compliant modellers, it is inevitable that researchers will use application modules from different sources. If two different application modules in the same integrated system required multiple attributes, both applications (which might have been written by different implementors) would independently attempt to maintain the two sets of attributes. For this reason a single attribute scheme, that loads the management of multiple attributes on to the application, is not acceptable.

There are further difficulties with attributes that are peculiar to Djinn and which arise from its lack of state and the required independence from any implementation or developer. Proprietary modellers can maintain a registry of attribute identifiers (i.e. attribute keys assigned to developers) thus ensuring their uniqueness. Since no central certification body will exist for Djinn, it cannot embody this approach. Indeed the problems of registering attribute keys in Djinn are compounded by its lack of state. Not only do instances of Djinn applications not know of the attributes of other applications, but instances of Djinn objects within an application are unaware of the attribute keys in use within other objects.

It is possible to envisage a Djinn function which takes a set of objects and returns a new key unique to that set of objects. Problems arise when the application requires further objects with independently assigned

attributes to be added to that set. Djinn's lack of state suggests that all the attribute key-value pairs would have to be removed from the existing set and regenerated using a key derived from the new set. This solution would be cumbersome; it would also prohibit the import and export of Djinn objects between applications. Furthermore, for the attributes to be portable outside a single Djinn instance, some central key registration (explicit or implicit) would again be required.

The solution adopted in Djinn is that all attributes are assigned by the application developer who also assigns plain-text keys, each key consisting of a 'developer code' and an 'attribute name'. Although the choice of developer code is not part of the Djinn function specifications, it is sensible to base it on some existing global identification scheme. Possibilities that were considered include:

1. E-mail address (e.g. `Djinn.User@ed.ac.uk:SurfaceFinish`).
2. Machine IP address (e.g. `192.41.103.174:colour`).
3. Reversed domain name (e.g. `uk.ac.ed.mech:density`).
4. Machine and domain specifier (e.g. `vader.mech.ed.ac.uk:nearest_face`).
5. International phone number (e.g. `44-1895-203390:roughness`).

The third possibility appears to be the most satisfactory; the domain name (reversed to simplify sorting) is sufficient to identify a small group of people who may be assumed to communicate closely enough to decide on their own scheme for unique attribute names. This convention obviates the need for a central register of developer keys. While there is no absolute guarantee that two attributes will not share the same key, the damage will be 'local'.

The issue of representation of attribute values must now be considered. Potentially, attributes can be used for any number of purposes. Some requirements can be met simply by using an integer as an index to a lookup table (e.g. the colour of a face might be one of 16 standard colours). Other types of data, such as density, might be better represented by floating-point values, and yet others may be pointers to application data. Such a wide variety of uses suggests a user-defined data-type; but that might lead to difficulties with the language binding.

However, the number of cells created by a Djinn implementation is necessarily finite, and the number of different attribute values required by any attribute type is also finite (and often substantially smaller than

the number of different cells). So attributes are defined to be integers, providing an index into a table defined by the application.

In summary:

- Every D̈jinn cell can have an arbitrary number of attributes.
- Every attribute is a key-value pair.
- An attribute key is a character string that should consist of a 'developer code' and an 'attribute name'; the developer code is the reversed Internet domain name of the attribute developer.
- The value of an attribute is an integer.

Attribute manipulation

When a cell is created, it has no application data associated with it. Applications may subsequently associate attributes with cells using the D̈jinn function:

Set attribute — Given a D̈jinn object, the label of a constituent cell, and an attribute type and value, associate the attribute with the given cell.

Attributes are deleted by assigning the null attribute, value 0. The attributes of a cell may be retrieved by:

Get attribute — Given a D̈jinn object and the label of a constituent cell and an attribute type, return the attributes of the given type associated with the given cell.

The types of attribute associated with a given object are discovered by:

Get attribute types — Given a D̈jinn object, return the set of attribute types associated with the object.

Attributes of a give type are eliminated by:

Delete attribute type — Given a D̈jinn object and an attribute type, delete all attributes of the given type from that object.

Cell disruption

D̈jinn's mechanism for attributes must cope with the disruption of cells, just like the label mechanism. A sensible solution is particularly important for the case when a D̈jinn operation results in the creation of new cells. Often, properties referenced by the attributes of new cells might be derived

from the properties of their parent cells, but this is not always what is required.

Many attributes reference simple intrinsic or extrinsic scalar properties. An intrinsic property (e.g. temperature) applies equally to all points of the cell with which it is associated. Such attributes should be copied when a cell is subdivided. An extrinsic scalar property (e.g. a distributed load on a face) is a value which must be shared among the subdivisions, usually weighted by a geometric measure such as surface area. Weighted sums are sometimes appropriate for extrinsic properties of disjoint unions, and weighted averages for intrinsic properties. In general, however, there is no self-evident approach to selecting the appropriate computation during subdivision; and there is certainly no obvious computation to use in the context of operations such as set-intersection.

Since attributes are not properties, but identifiers of properties, Djinn functions can easily copy attributes, but they do not have access to the data to perform any other computation on them. It is therefore not possible to prescribe how attributes are combined for each Djinn function that combines objects. In general, properties for new cells can only be computed by the application. Such computations could be initiated using 'call-backs' to functions supplied as part of the application, or through additional parameters of Djinn functions. Call-back functions might be invoked many times (e.g. in applications that treated every edge and face as a cell) and dummy functions would need to be provided if an application did not require attribute combination. But, in any case, functions provided by the application cannot be invoked while Djinn objects are being combined, because cells may be in an intermediate state and are therefore undefined, as explained in Chapter B.

Often, the objects resulting from Djinn functions will have few cells that are straight copies of cells in one operand. However, the labels of all cells remain accessible from the originals using the label map and, after a geometric function has been performed, the application can take the opportunity to compute correct attributes for the new cells based on their parentage.

When a Djinn function destroys a cell, it may destroy the only path to application data, which then becomes redundant. This problem is avoided because labels and thus attributes of cells that disappear during an operation such as departitioning remain accessible by means of the parentage map, as cells which are parents of OUT.

This provides an opportunity for the application to retrieve attributes from condemned cells and process the corresponding application data appropriately.

Cell selection

This section is about the facilities in the D̈jinn API for making sense of the cell structure of objects. Since, by design, the internal structure of D̈jinn objects is hidden, cell selection must replace the traversal of representation-dependent data-structures familiar to users of, for example, boundary or set-theoretic solid modellers.

Even if an application has partitioned a D̈jinn object so that it has cells corresponding to, say, faces and edges, D̈jinn functions will do no more than discover whether one cell 'bounds' another. However, since D̈jinn geometry includes the concept of orientation, it is subsequently possible to enquire about the relative orientation of cells and those that bound them.

Selection by type

Small cells — Given an object, a length and a dimension set return a set of labels for all cells with a dimension in the given set that have a length, area or volume (as appropriate) less than the given length raised to the power of their dimension. All of an object's cells of the required dimension may be obtained by using a sufficiently large length.

Selection by proximity

Cells can be selected by their proximity to a given point using the function:

Point proximate cells — Given an object, the coordinates of a point, and a set of dimensions, return the labels of the cell (or cells) of any required dimensionality closest to the given point.

Analogous functions use lines and planes for selection:

Line proximate cells — Given an object, two points defining a line, and a set of dimensions, return the labels of a cell (or cells) of any required dimensionality closest to the given line.

Plane proximate cells — Given an object, the coordinates of a plane, and a set of dimensions, return the label of the cell (or cells) of any required dimensionality closest to the given plane.

If the line or plane intersects multiple cells, the result includes all intersected cells whose frontiers are not intersected. For example, if a line intersects a solid sphere, only its surface cell is returned.

Selection by bounding relations

Two bounding enquiry functions answer the questions, "What bounds this cell?" and "What does this cell bound?"

Get boundary cells	Given an object and a cell label, return the set of labels of cells which bound the given cell.
Get bounded cells	Given an object and a cell label, return the set of labels of cells which are bounded by the given cell.

These two functions only recognize 'direct' bounding relationships. For instance, if cell A bounds cell B and cell B bounds cell C, then cell C also bounds cell A because of the frontier condition (Chapter C); but the label of cell C will not be in the set returned when the boundary cells of cell A are retrieved. Thus the edge cells bounding a face cell will be returned, but not the vertex cells which bound those edge cells. Direct bounding cells are usually one dimension lower then the cell they bound, but this is not always the case. For example, the apex of a cone is in the direct boundary of a conical surface cell, as is the interior point of a face cell that is in contact with some other part of a non-manifold object.

Some extra nuances are introduced because the Djinn specification allows the label of OUT to be passed to and returned by these two functions. Thus OUT will be returned as a label of one of the cells bounded by a boundary cell of a Djinn object, and boundary cells will be returned if the cell labelled OUT is interrogated for its boundary. The relative orientation of cells which have a bounding relationship may be discovered by the function:

Get orientation	Given a Djinn object and two cell labels, return a logical variable which indicates the relative orientation of the cells.

This function operates on two cells which differ in dimension by one. Thus it can be used on a point and an edge cell to determine whether the edge is directed away from an end-point. Given an edge and a face cell, it will determine whether the cross product of the edge direction and the face normal points into the exterior of the face; and in the case of a face and a solid cell *Get orientation* will determine whether or not the normal to the face points out of the solid.

Part I
4
Transformations

The philosophy underlying the D̈jinn approach to transformations and transforming objects has already been explained in Chapter B. This chapter fills in the detail by listing all the transformation types that can be created by calls to D̈jinn functions.

As indicated in Chapter B, D̈jinn allows a single transformation to be applied to an object, or a series of transformations to be concatenated before use. This second approach is supported by the function:

Concatenate transformations	Given two transformations, create the transformation that is equivalent to their sequential application.

This function can be applied to all (linear, quadratic and non-polynomial) transformations, because they are all represented by the same D̈jinn type.

Representation

D̈jinn uses geometry to create transformations with a representation that is not visible to the application. Thus, D̈jinn transformation arguments are not given as conventional transformation matrices. D̈jinn supports simple geometric construction functions to create transformations such as rotation about a coordinate axis etc. (see below). Although these functions can be concatenated to provide more general transformations, which is often convenient, general linear transformations can be created directly by D̈jinn, as follows.

Suppose that a general 4×4 linear transformation matrix Q maps a point $\mathbf{p}$ to point $\mathbf{q}$ so that their homogeneous coordinates have the relationship $\langle q_x, q_y, q_z, 1\rangle = \langle p_x, p_y, p_z, 1\rangle Q$. Such a transformation may be turned into hidden D̈jinn form using the function:

Create linear transformation — Given a 4×4 linear transformation matrix, create the corresponding Djinn transformation.

Object distortions such as taper, bend and twist can be specified using quadratic transforms. These specific transforms will be dealt with later in this section, but an arbitrary user-defined quadratic transform can be created in hidden Djinn form using the function:

Create quadratic transformation — Given four 4×4 symmetric matrices Q_1, create the Djinn transformation by which each homogeneous coordinate of a result point $\mathbf{q}_i$ is given by the quadratic form $\mathbf{p}Q_i\mathbf{p}^{\mathrm{T}}$, where the elements of $\mathbf{p}$ are the homogeneous coordinates of the untransformed point.

Sometimes, the application may need to use a Djinn transformation. Since transformations are hidden, Djinn provides functions to convert them into the conventional linear and quadratic forms:

Get linear transformation — Given a linear transformation, compute the corresponding 4×4 transformation matrix.

Get quadratic transformation — Given a quadratic transformation, compute the four corresponding quadratic forms.

Transforming objects

The following function applies a transformation to an object:

Apply transformation — Given an object and a transformation, apply the transform to the point-sets of the object and of all its cells.

Degeneracy

Since all Djinn entities reside in $\mathcal{E}^3$, transformations are three-dimensional (i.e. they map between points in $\mathcal{E}^3$). Degenerate transformations (e.g. differential scaling by 0 in the x direction) require special consideration.

- They could cause Djinn transformation functions to create a set of lower dimension.
- They could cause Djinn transformation functions to execute a null operation with an error return.

- They might be disallowed by pre-conditions that prohibit the invocation of D̈jinn transformation functions with degenerate transformations.

The second approach has been taken in D̈jinn, so as to relieve the application of the burden of satisfying the pre-condition and to ensure that all transformations have an inverse with which the original geometry can be retrieved. (How projections are still possible will be discussed later.) The detection of degenerate transformations is difficult in floating-point implementations, but transformations that are nearly degenerate are also of little practical use. Therefore D̈jinn rejects transformations that reduce the bounding box of an object to a shape with a minimum diameter less than a value supplied by the application. (N.B. this degeneracy control parameter and the error return parameter are not explicitly listed in the function summaries in the rest of this chapter.)

Inversion

All non-degenerate D̈jinn transformations are one-to-one maps; their inverse can be obtained using the D̈jinn function:

Invert transformation	Given a transformation, create the transformation that reverses its effect.

If the inverse cannot be computed because the transformation is degenerate or nearly degenerate, the result is a null transformation. When a null transformation is applied, it will fail due to the minimum diameter test. Thus, the effect is identical to the application of a nearly degenerate transformation for which it is possible to compute an inverse. The concatenation of a transformation and its inverse gives the identity transformation only if its computed inverse is not null.

Linear transformations

For convenience, D̈jinn provides functions to create the usual simple linear transformations.

Rigid-body transformations

Create translation	Given two points specifying a displacement, create the corresponding translation transformation.

Create axial rotation — Given a rotation angle, and a plane of rotation formed by choosing two of the coordinate axes, create the transformation that rotates one axis towards the other[1].

Create alignment — Given two defining points, create the transformation that rotates about a line through the origin such that the vector from the origin to the first point is rotated so as to have the same direction as the vector from the origin to the second point.

Create rotation — Given two points defining a line, and a rotation angle, create the transformation that rotates clockwise about the given line.

Affine transformations

D̈jinn provides four functions for creating more general affine transformations.

Create scale — Given three scale factors in the directions of the coordinate axes, create the corresponding unequal scaling transformation.

Zero scale factors are not permitted, since they result in a degenerate transformation without an inverse. Negative scale factors define a transformation that mirrors an object in a coordinate plane. More general mirror transformations are created by the function:

Create mirror — Given a general plane, create the transformation to mirror in the given plane.

A shear transformation translates points in a shear direction by an amount proportional to their coordinates with respect to an orthogonal shear axis. For example, a point $\langle x, y, z\rangle$ sheared in the x direction by factor s with respect to shear axis y becomes $\langle x + sy, y, z\rangle$. A shear transformation defined by any pair of coordinate axes is created by:

Create axial shear — Given two of the coordinate axes and a shear scale factor, create the shear transformation to shear in the direction of the first coordinate axis by an amount proportional to the coordinate value with respect to the second.

[1]Unlike some other definitions, this way of defining axial rotations also applies to rotations in spaces of dimension greater than three.

For an arbitrary shear axis $\mathbf{u}$, and a (usually orthogonal) shear direction $\mathbf{v}$, the point $\mathbf{p}$ becomes $(\mathbf{p}.\mathbf{u})\mathbf{u} + [\mathbf{p}.(\mathbf{v} + s\mathbf{u})]\mathbf{v} + [\mathbf{p}.(\mathbf{u} \times \mathbf{v})](\mathbf{u} \times \mathbf{v})$. The transformation to achieve this operation is created by the Djinn function:

Create general shear	Given a shear axis vector, a shear direction and a shear scale factor, create the corresponding shear transformation.

Projective transformations

Djinn does not support graphics directly, but has functions to facilitate ray-casting, for use by graphics and other applications (see Chapter 8). Creating perspective views efficiently by ray-casting requires the object to be projectively distorted to allow rays to be cast parallel to a coordinate axis. Such a transformation can be achieved by a rigid-body transformation that maps from object space to a view space followed by a projectively distorting transformation to the image space [Foley 1990].

The viewing transformation can be created using the Djinn function:

Create view transformation	Given the centre of interest, the view plane normal and view up vector, create the corresponding rigid-body viewing transformation.

When the viewing transformation is applied to an object, it is transformed such that parallel views from the positive z-direction are equivalent to views in the opposite direction to the view plane normal. The centre of interest becomes the origin and vectors in the view-up (VUP) and view plane normal (VPN) directions become parallel to the y and z view-space coordinate axes respectively.

The view transformation allows the creation of parallel orthographic projections. Perspective images and specific projections require the further transformation into image space by a transformation created using the Djinn function:

Create image transformation	Given the direction of projection and the reciprocal of the view distance, create the corresponding image transformation.

When the image transformation is applied to an object, it is transformed such that the origin and the z-coordinate axis are unchanged. Perspective effects are obtained when the viewing distance is finite.

Usually, the direction of projection (DOP) of the image transformation is opposite to the view plane normal (VPN) of the view transformation, but alternative orientations are required for specific projections. For example:

	VPN	VUP	DOP	R
Parallel orthographic view from $+x$	$\langle 1,0,0\rangle$	$\langle 0,0,1\rangle$	$\langle 0,0,-1\rangle$	0
Isometric projection	$\langle 1,1,1\rangle$	$\langle 0,0,1\rangle$	$\langle -1,-1,-1\rangle$	0
Front cabinet projection	$\langle 1,0,0\rangle$	$\langle 0,0,1\rangle$	$\langle \frac{-c}{2}, \frac{-s}{2}, -1\rangle$	0
One-point perspective of paraxial box	$\langle 1,0,0\rangle$	$\langle 0,0,1\rangle$	$\langle -1,-1,-r\rangle$	r

The cosine and sine of the angle between x and y-coordinate axes after projection are c and s respectively, and r is the reciprocal of the viewing distance.

Quadratic transformations

Taper

It is often convenient to construct an object by tapering or bending another object. For example, simple objects such as cuboids and cylinders can be tapered to become wedges, pyramids and cones. Although such simple tapers can be achieved by a linear projective transformation, this approach often causes unwanted effects. For example, a stepped cylinder becomes a stepped cone, but the relative lengths of the pieces are also changed.

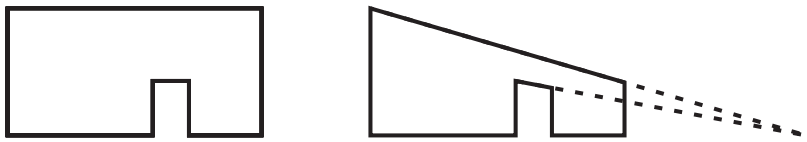


Figure 8: A tapering operation.

In general, quadratic distortions are necessary for a useful tapering operation. For example, a wedge with a $\pi/4$-radian ridge through the origin can be created by scaling the y-coordinate in proportion to the z-coordinate, as shown in Figure 8. Thus, a point $\langle x, y, z\rangle$ becomes $\langle x, yz, z\rangle$. A cube can be converted into a pyramid and a cylinder into a cone by the concatenation of two such transformations, affecting both x and y-coordinates. Such transformations are created from a geometric specification by the function:

Create taper — Given two taper factors, create the transformation to taper in proportion to the third coordinate.

For example, if taper factors a and b were given for x and y, then this would map point coordinates $\langle x, y, z\rangle$ to coordinates $\langle x(1+az), y(1+bz), z\rangle$.

Geometric inversion

Geometric constructions are often simplified by the use of geometric inversion. A point $\mathbf{p}$ that is inverted to a point $\mathbf{q}$ satisfies $(\mathbf{c}-\mathbf{p}).(\mathbf{c}-\mathbf{q}) = r^2$, where $\mathbf{c}$ and r are the centre and radius of inversion respectively: the points $\mathbf{c}$, $\mathbf{p}$ and $\mathbf{q}$ lie on the same straight line. Geometric inversion maps spheres (including planes) to spheres. For example, a sphere of diameter r, in the positive-z half-space and resting on the xy-plane at the origin, is transformed into the entire negative-z half-space, if it is inverted using the D̈jinn function defined below.

Geometric inversion is achieved by creating a transformation, which is quadratic in two dimensions and cubic in three dimensions, using the D̈jinn function:

Create inversion — Given the radius of inversion r, create a transformation which is quadratic in two dimensions and cubic in three dimensions, with the given radius of inversion and a centre of inversion at coordinate r on the z-axis.

Bend

It is often convenient to construct an object by bending another object. Inversion can be used to bend objects but parallel faces do not remain parallel. Quadratic transformations can be used to create the parabolic bends of Figure 9. Thus, a point $\langle x, y, z\rangle$ becomes $\langle x, y, z + x^2\rangle$ and the horizontal xy-plane shown is bent into the dotted parabola. A general parabolic bend is created by the D̈jinn function:

Create bend — Given two bend factors, create the corresponding bend transformation.

The bend factors a and b map point coordinates $\langle x, y, z\rangle$ to coordinates $\langle x, y, z + ax^2 + by^2\rangle$.

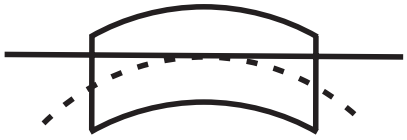


Figure 9: A quadratic bend.

Non-polynomial transformations

Twist

Useful distortions for the construction of solid objects include twists that cannot be achieved with polynomial maps. Twist maps involve sines and cosines [Barr 1984]. Thus, a twist about the z-coordinate axis, with pitch p, maps a point $\langle x, y, z \rangle$ to a point:

$$\left\langle x \cos \frac{2\pi z}{p} - y \sin \frac{2\pi z}{p}, x \sin \frac{2\pi z}{p} + y \cos \frac{2\pi z}{p}, z \right\rangle .$$

Points on the z-coordinate axis and those on the xy-plane remain unchanged. Other points are rotated about the z-axis in proportion to their z-coordinate, such that points with z-coordinates equal to the pitch are rotated by a complete revolution. D̈jinn provides the following function to create a twist transformation:

Create twist	Given the pitch, create the transformation to twist about the z-coordinate axis.

Similar but not identical effects are produced by sweeping two-dimensional profiles along a helical curve (see Chapter 2).

Part I
5
Object combination

Most solid modellers, whether they are based on set-theoretic , boundary, or other representations, have functions to combine solids. Commonly, these are set-union, set-intersection and set-difference; they are illustrated in two dimensions in Figure 10. These operations are normally regularized, as explained below, so that the functions take solids as their operands and produce only solids as results.

In the D̈jinn environment, objects of different dimension may co-exist and interact, and thus operations for combining sets must be able to deal with all the possible permutations of operands: a solid combined with a solid, a face, an edge or a vertex; a face combined with another face, an edge, or a vertex; and so on. Additionally, the D̈jinn operations have to work on point-sets that may contain internal membranes; and they must also allow point-sets and their boundaries to co-exist while remaining separate.

At the start of this chapter some modified point-set operations are defined, together with the notation necessary for considering set-combination operations for D̈jinn cells and objects. Various situations are described, and these are classified according to the dimension of the operands and the dimension of any intersection between them; fortunately there are only three different cases that must be distinguished.

We next discuss issues more specific to set-combinations of D̈jinn cells and objects: Firstly, it is shown how the manifold dimension of the results of set-combination depend on the operand dimensions. Then, since D̈jinn objects have a cellular structure, the cellular structure resulting from the set-combination of two D̈jinn objects is considered. After that, operations for the set-combination of cellular objects of heterogeneous dimension are defined.

We conclude the description of set-operations in D̈jinn by describing the

actual functions provided to implement the 'primary' set-operations, and indicating how they affect the D̈jinn objects that they take as operands.

Set-operations on point-sets

Figure 10 illustrates the conventional set-operations $\cup$, $\cap$ and $-$, defined in terms of the point-sets P and Q, and points $\mathbf{w}$, as follows:

$$P \cup Q = \{\mathbf{w} \in \mathcal{E}^3 | (\mathbf{w} \in P) \vee (\mathbf{w} \in Q)\},$$
$$P \cap Q = \{\mathbf{w} \in \mathcal{E}^3 | (\mathbf{w} \in P) \wedge (\mathbf{w} \in Q)\},$$
$$P - Q = P \cap Q',$$

where Q' is the complement of Q.

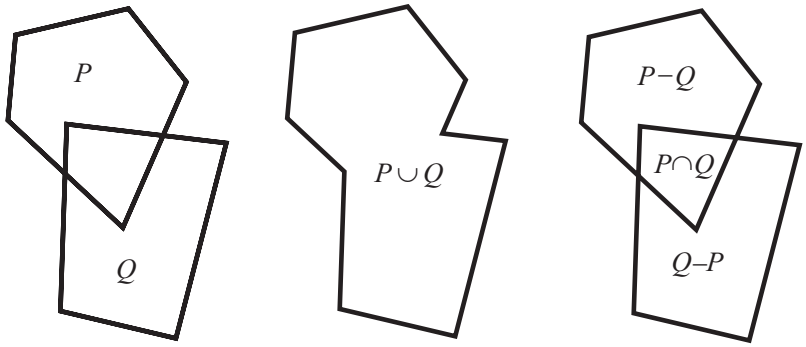


Figure 10: The conventional set-operators.

Regularization

As already mentioned, conventional set-theoretic and boundary modellers usually provide functions for the *regularized* set-combination of solids represented by closed point-sets. These operators apply interior (*Interior*) followed by closure (*Closure*) operations to the result of conventional set-operations in order to remove unwanted isolated edges and faces that result from the set-intersection of touching objects [Middleditch 1974, Requicha 1980]. For open point-sets, the analogous regularized set-operators are more relevant:

$$P \cup^* Q = \mathit{Interior}\,(\mathit{Closure}(P \cup Q)),$$
$$P \cap^* Q = \mathit{Interior}\,(\mathit{Closure}(P \cap Q)),$$
$$P -^* Q = \mathit{Interior}\,(\mathit{Closure}(P - Q)).$$

The results of these operations are open sets. Unfortunately, regularization with respect to $\mathcal{E}^3$ always reduces rank-deficient point-sets (i.e.

'hanging' faces, edges and vertices) to the empty point-set. This is incompatible with the requirements for D̈jinn to handle objects of mixed dimension (see Chapter B). Furthermore, it was also shown in Chapter B that rank-deficient manifold point-sets are not open subsets of the space in which they are embedded. The regularized operators are appropriate only for manifolds of dimension equal to the common spatial dimension and, of course, when it is in fact required that all isolated parts are to be removed, which is not always the case.

Manifold interiors

We define the k-manifold interior of a (not necessarily manifold) set embedded in $\mathcal{E}^n$ as follows:

For P, a subset of $\mathcal{E}^n$, the (possibly empty) *k-manifold interior* ($0 \leq k \leq n$), which will be written $Interior_k(P)$, is defined as the set of points in P which have a neighbourhood in P homeomorphic to the ball B^k.

If P is a k-manifold, then $Interior_k(P) = P$; if P is an i-manifold ($i \leq k$), then $Interior_k(P)$ is empty.

In order to combine manifolds and produce point sets which are also manifolds, D̈jinn defines the set-operators $\cap_k^{\#}$ and $-^{\#}$. Consider first the effect of the subtraction operator $-^{\#}$ on P and Q of $\mathcal{E}^n$ (see Figure 11(b)):

$$P -^{\#} Q = P - Closure(Q).$$

The result of this difference operation is a manifold set because the set that is removed is closed (see Chapter B). In contrast, simple intersection may not result in a manifold set, but a manifold set can be retrieved by taking a manifold interior:

$$P \cap_k^{\#} Q = Interior_k(P \cap Q).$$

Operands of the same dimension with no lower-dimension intersection

The first case to be considered is the set-combination of manifold point-sets of the same manifold dimension, that have no set-intersection of lower dimension. In this case, the conventional set-union is manifold and both conventional set-union and set-intersection have the same manifold dimension as the operands.

Operands of different dimension with no lower-dimension intersection

Now, consider set-combinations of manifold point-sets of different manifold dimension that have no set-intersection of lower dimension than that of either operand.

Cracks can be created using the $-^{\#}$ subtraction of one manifold point-set from an overlapping manifold set of higher dimension; the result is a manifold set. The intersection $\cap^{\#}$ of operands of different dimension yields the expected manifold point-set with the lower of the operand dimensions.

Operands with an intersection of lower dimension than either operand

Now consider the set-combination of rank-deficient manifold point-sets that intersect to yield a point-set of manifold dimension lower than that of either operand (e.g. two surface patches intersecting in a common edge or two edges intersecting at a point). This intersection is $P \cap^{\#}_{k-1} Q$, where k is the minimum manifold dimension of P and Q.

Figure 11: Set-operations on open manifold point-sets.

Unfortunately, when one operand is also incident to the other in a point-set of the same manifold dimension (see Figure 11(a)), set-intersection produces disjoint subsets of different manifold dimension. Djinn cells must be point-sets of homogeneous dimension, and in that case the result of set-intersection must be considered as two distinct disjoint subsets of such type:

$$\{P \cap^{\#}_{k} Q,\ P \cap^{\#}_{k-1} Q\},$$

where k is the is the minimum manifold dimension of P and Q. The first point-set has the same dimension as the operand of lower dimension (the line segment $P \cap^{\#}_{1} Q$ of Figure 11(b), and the second point-set is an intersection $P \cap^{\#}_{0} Q$ of dimension one lower than that operand (the point of intersection in Figure 11(b).

The definition above also applies in the case of operands without lower-dimension intersections, discussed above. However, if there is no intersection of lower dimension than each operand, then the second component is the empty set; the first component is the empty set when there is *only* an intersection of lower dimension.

Since D̈jinn cells must be manifold point-sets of homogeneous dimension, the result of the union operator must be considered as the following set of disjoint manifold point-sets:

$$\{P -^{\#} Q, Q -^{\#} P,\ Q \cap^{\#}_{k} P,\ Q \cap^{\#}_{k-1} P\},$$

where k is the minimum manifold dimension of P and Q. These separate components are identified in Figure 11(b).

D̈jinn, therefore, uses the following modified set-operations that combine open manifold point-sets of any dimension to give a set of open manifold results (see Figure 11):

Difference	$\{P -^{\#} Q\}$,
Intersection	$\{P \cap^{\#}_{k} Q, P \cap^{\#}_{k-1} Q\}$,
Union	$\{P -^{\#} Q, Q -^{\#} P, Q \cap^{\#}_{k} P, Q \cap^{\#}_{k-1} P\}$,

where k is the minimum manifold dimension of P and Q.

These results exclude subsets of intersection of lower dimension than normal. For example, they exclude isolated points of intersection between two surfaces, since it is difficult to detect these reliably.

Manifold dimension of set-combinations of cells

In addition to its spatial dimension, each D̈jinn cell has a well-defined manifold dimension; from the application viewpoint D̈jinn geometry is exact (see Chapter B) and inaccuracy occurs only when numerical results are computed. The dimension of a D̈jinn set-combination of cells is defined in terms of the operand dimensions and incidence relationships. Thus, despite the ambiguities sometimes implied by the point-set interrogations discussed in Chapter 8, which are caused by the vagaries of real arithmetic, an application can rely on results being of the correct dimension.

Pieces of surface and curve segments arise only from operations which explicitly create them. For example, since solid cells are open sets, a face cannot be created by the set-intersection of solid cells that meet at common pieces of boundary. In practice, such operations are sensitive to computational errors; subsequent interrogation might indicate that the set is empty, very thin or thin with random holes. The result remains

three-dimensional because its manifold dimension is defined to be that of the operands. It is not dependent on the computation of touching contact.

This principle applies to cells of any manifold dimension that are set-intersected in a space of the same dimension. For example, the intersection of two subsets of a surface has a manifold dimension of 2.

In cases where the manifold dimensions of the operands are not equal to their spatial dimensions, only intersections of dimension equal to the minimum manifold dimension among the operands, and intersections of (minimum dimension − 1) are of interest. For example, the intersection of an arc and a surface in $\mathcal{E}^3$ yields segments of the arc lying in the surface together with discrete points, or the empty set. When computing intersections of operands with different manifold and spatial dimensions, but which are operands incident to a common manifold of the same dimension, the result is either indeterminate, or (due to computational errors) an arbitrary point.

Unions of point-sets of different homogeneous dimension consist of multiple disjoint manifold point-sets, as described earlier in the chapter. The difference between two point-sets has the dimension of the first operand. If the (possibly empty) set-intersection of manifold point-sets with manifold dimension a and b respectively also has homogeneous dimension, then that dimension is $\min(a, b)$ or $\min(a, b) - 1$.

For example, if there is no common manifold superset of the same dimension, then the intersection between two surfaces is a set of curves, the intersection between two curves is a set of points, and so is the intersection between a curve and a surface. This is all true by definition. Thus:

- The intersection of two surface patches with no two-dimensional manifold superset is an arc: of dimension 1.
- The intersection of an arc and patch with no two-dimensional manifold superset is a point: of dimension 0.
- The intersection of two arcs in a common manifold surface is a point: of dimension 0.
- The intersection of two patches in a common manifold surface is a patch: of dimension 2.

Partial coincidence of two surface patches can be detected from the existence of a common surface that contains those patches. This is a special case of the determination of the lowest-dimension manifold superset of two arbitrary point-sets. The dimensions of such supersets can be used to distinguish the situations elaborated earlier. Thus the dimension of the set-intersection is defined, and that assists in the choice of algorithms to locate it.

Inexact computation

Unfortunately, the determination of a lowest-dimension common manifold superset of two point-sets is generally subject to arithmetic vagaries in floating-point implementations. However, when a superset of the same dimension is detected, the set-intersection can be reliably computed. When such a superset exists but is not detected, the rank-deficient set-intersection is indeterminate. For example, the intersection of two partially coincident surface patches without a detected common surface is an indeterminate curve, not another patch.

In floating-point implementations, the set-intersection of touching or nearly touching point-sets may or may not be found to be disjoint during the interrogations described in Chapter 8, and their set-union may or may not be found to be disjoint. If objects are intended to be disjoint, they should be explicitly modelled by the application as distinct cells that are processed independently by D̈jinn. Similarly, a region of touching contact between two solid objects should be modelled as a distinct face cell.

The topological uncertainty of set-union is irrelevant for many evaluation algorithms (e.g. the computation of volume integrals). When topological uncertainty is relevant, the application can usually resolve the difficulty. For example, a computer-aided manufacturing program would not attempt to drive a machine-tool cutter through cracks between the union of two nearly touching objects because the objects would be dilated (see Chapter 6) before the union operation. If the difficulties cannot be resolved in a pragmatic way, then the application should explicitly model the cellular structure of the object to reflect the desired topology.

An application is still able to use D̈jinn to detect the possibility of touching contact—taking account of computation errors—and a region of possible overlap can be explicitly converted into a lower-dimensional point-set. For example, two point-sets could be dilated (see Chapter 6) before set-intersection followed by extraction of the medial surface (see Chapter 8). Fortunately, such complicated operations should usually be unnecessary.

The computation of the dimension of a set-combination entirely from the dimensions of the operands and minimum dimension supersets is imperative in floating-point implementations, because reliable detection of touching contact is impossible. Detection may be possible using arbitrary-precision rational arithmetic or symbolic manipulation, but these approaches are extremely slow. Furthermore, for many applications, the explicit declaration and predictable deduction of point-set dimension is desirable. Unfortunately, because dimension involves the computation of supersets of minimum dimension, it sometimes depends on the reliable

detection of coincidence between rank-deficient point-sets.

Set-operations on Djinn cells

It is now appropriate to specify set-combination operators for the Djinn cells defined in Chapter B (i.e. oriented manifold semi-analytic point-sets of homogeneous dimension). These operators are derived from the new set-operators on manifolds ($\cap^{\#}$ etc.) already introduced.

The set-intersection of any pair of cells consists of two components, which are dimensionally homogeneous, and either of which may be the empty set. The component with the same dimension as the operand of lower dimension is given by the following operator:

$$
\begin{aligned}
&\cap^{c_1} \in \mathit{Cell} \times \mathit{Cell} \rightarrowtail \mathit{Cell} \bullet \\
&\quad \forall A, B \in \mathit{Cell} \bullet \mathit{SDim}(A) = \mathit{SDim}(B) \bullet \text{Let } C = A \cap^{c_1} B \bullet \\
&\qquad (\mathit{SDim}(C) = \mathit{SDim}(A)) \\
&\qquad \wedge (\mathit{MDim}(A) \leq \mathit{MDim}(B)) \Rightarrow (\mathit{MDim}(C) = \mathit{MDim}(A)) \\
&\qquad \wedge (\mathit{Point\text{-}set}(C) = \mathit{Point\text{-}set}(A) \cap^{\#}_{\mathit{MDim}(C)} \mathit{Point\text{-}set}(B)) \\
&\qquad \wedge (\mathit{Orient}\,(C) = \mathit{Orient}\,(A) \triangleright \mathit{Point\text{-}set}(C)) \\
&\qquad \wedge (\mathit{MDim}(A) > \mathit{MDim}(B)) \Rightarrow (A \cap^{c_1} B = B \cap^{c_1} A).
\end{aligned}
$$

The orientation of the result and its manifold dimension is that of the lower dimension operand ($\triangleright$ restricts $\mathit{Orient}\,(A)$ to the domain of $\mathit{Point\text{-}set}(C)$).

The second component of lower dimension than either operand is given by the operator:

$$
\begin{aligned}
&\cap^{c_2} \in \mathit{Cell} \times \mathit{Cell} \rightarrowtail \mathit{Cell} \bullet \\
&\quad \forall A, B \in \mathit{Cell} \bullet \mathit{SDim}(A) = \mathit{SDim}(B) \bullet \text{Let } C = A \cap^{c_2} B \bullet \\
&\qquad (\mathit{SDim}(C) = \mathit{SDim}(A)) \\
&\qquad \wedge (\mathit{MDim}(A) \leq \mathit{MDim}(B)) \Rightarrow (\mathit{MDim}(C) = \mathit{MDim}(A) - 1) \\
&\qquad \wedge (\mathit{Point\text{-}set}(C) = \mathit{Point\text{-}set}(A) \cap^{\#}_{\mathit{MDim}(C)} \mathit{Point\text{-}set}(B)) \\
&\qquad \wedge (\mathit{Orient}\,(C) = \mathit{Orient}\,(A) \triangleright \mathit{Point\text{-}set}(C)) \\
&\qquad \wedge (\mathit{MDim}(A) > \mathit{MDim}(B)) \Rightarrow (A \cap^{c_2} B = B \cap^{c_2} A).
\end{aligned}
$$

The orientation of the result is that of the operand of lower dimension, and its manifold dimension is one less than that of the operand of lower dimension. (Note the "-1" in the definition of $\mathit{MDim}(C)$.)

Set-subtraction of one cell from another cell of higher dimension either has no effect or creates cracks. Subtraction of one cell from another of the same or lower dimension clips the latter:

$$
\begin{aligned}
&-^c \in \mathit{Cell} \times \mathit{Cell} \rightarrowtail\!\!\!\!\!\!+ \mathit{Cell} \bullet \\
&\quad \forall A, B \in \mathit{Cell} \bullet \mathit{SDim}(A) = \mathit{SDim}(B) \bullet \text{Let } C = A -^c B \bullet \\
&\qquad (\mathit{SDim}(C) = \mathit{SDim}(A)) \\
&\qquad \wedge (\mathit{MDim}(C) = \mathit{MDim}(A)) \\
&\qquad \wedge (\mathit{Point\text{-}set}(C) = \mathit{Point\text{-}set}(A) - \mathit{Point\text{-}set}(B)) \\
&\qquad \wedge (\mathit{Orient}\,(C) = \mathit{Orient}\,(A) \triangleright \mathit{Point\text{-}set}(C)).
\end{aligned}
$$

Thus, the orientation and the manifold dimension of the result is that of the first operand.

Set-operations on Djinn objects

The cell-operators just defined are now used to define operators with operands and results that are sets of cells (i.e. constituents of cellular objects). The operators are specified in terms of D̈jinn objects, as defined in Chapter B, but without the augmentation of attributes and labels. The point-set of a cellular object is the conventional union of the point-sets of its cells.

Most solid modelling systems use regularized set-combinations in order to avoid the creation of cracks during the union of touching objects, or an isolated face resulting from intersection. When touching cellular objects are united, their common boundary can be eliminated. However, if the internal cellular structures are preserved, they may not be compatible. Therefore, D̈jinn treats interior and boundary cells in the same way during set-combination. If necessary, the application can eliminate common boundaries or discard degenerate intersections using other D̈jinn functions, thus achieving appropriate regularization.

In regions where the point-sets of the operands do not overlap, object union and intersection do not affect the cellular structure. Possible ways to treat the region of overlap include retaining all original cell boundaries, no cell boundaries, or boundaries from only one operand (see Figure 12).

The set-intersection $\cap^\circ$ of two objects is thus the set of all cells created by the set-intersection of a cell from each operand. Both dimensionally homogeneous components of the intersection (i.e. $\cap^{c_1}$ and $\cap^{c_2}$) are used:

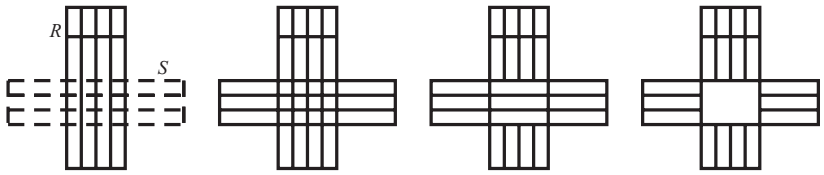


Figure 12: Interpretations of a set-union operation between cellular objects.

$$
\begin{aligned}
&\cap^\circ \in \mathit{Object} \times \mathit{Object} \rightarrowtail \mathit{Object} \\
&\quad \forall F, G \in \mathit{Object} \bullet \mathit{SDim}(F) = \mathit{SDim}(G) \bullet \\
&\qquad (\mathit{SDim}(F \cap^\circ G) = \mathit{SDim}(F)) \\
&\qquad \wedge (\mathit{Cells}(F \cap^\circ G) = \left(\cup_{A \in \mathit{Cells}(F)} \cup_{B \in \mathit{Cells}(G)} \{A \cap^{c_1} B\} \right) \\
&\qquad\quad \bigcup \left(\cup_{A \in \mathit{Cells}(F)} \cup_{B \in \mathit{Cells}(G)} \{A \cap^{c_2} B\} \right).
\end{aligned}
$$

This set-intersection can create cells with disjoint pieces, but if necessary such cells may be subsequently subdivided to ensure that all cells are connected (see Chapter 2).

Consider the point-set of one object (i.e. the union of all its cells' point-sets), subtracted from another object. The result is a set of cells, but it may not constitute a D̈jinn object because the boundary points of those cells are not contained in a cell of the set. The same result is obtained when the cell difference operator $-^c$ is used to subtract all cells of the second operand from each cell of the first:

$$
\begin{aligned}
&-^f \in \mathit{Object} \times \mathit{Object} \rightarrowtail P(\mathit{Cell}) \\
&\quad \forall F, G \in \mathit{Object} \bullet \mathit{SDim}(F) = \mathit{SDim}(G) \bullet \\
&\qquad F -^f G = \cup_{A \in \mathit{Cells}(F)} \mathit{Subtract}(A, G), \\
&\qquad \text{where } \mathit{Subtract}(A, G) = A_{r+1}, \\
&\qquad\qquad \mathit{Cells}(G) = \{B_1, \ldots, B_r\} \\
&\qquad \text{and } A_0 = A, A_{i+1} = A_i -^c B_i.
\end{aligned}
$$

The D̈jinn set-difference $-^\circ$ between two objects augments this operation to complete the boundary whilst preserving the frontier condition defined in Chapter B:

$$
\begin{aligned}
&-^\circ \in \mathit{Object} \times \mathit{Object} \rightarrowtail \mathit{Object} \bullet \\
&\quad \forall F, G \in \mathit{Object} \bullet \mathit{SDim}(F) = \mathit{SDim}(G) \bullet \\
&\qquad (\mathit{SDim}(F -^\circ G) = \mathit{SDim}(F)) \\
&\qquad \wedge (\mathit{Cells}(F -^\circ G) = (F -^f G) \bigcup (\cup_{A \in \mathit{Cells}(F)} \cup_{B \in \mathit{Cells}(H)} \{A \cap^c_1 B\}) \\
&\qquad\quad \text{where } H = \{C \in \mathit{Cells}(G) | \mathit{Point\text{-}set}(C) \subseteq \partial(\mathit{Point\text{-}set}(G))\}.
\end{aligned}
$$

This object set-difference enables faces to be subtracted from a solid to create internal cells possibly representing either a crack or an internal membrane (see Figures 13(a1) and 13(a2)), and the subtraction of an object with two touching solid cells leaves an imprint of its boundary cells (see Figures 13(b1) and 13(b2)). Note that, since all Djinn cells are disjoint, the two faces of a crack cannot be coincident and the application must use an attribute to denote an infinitesimally narrow crack.

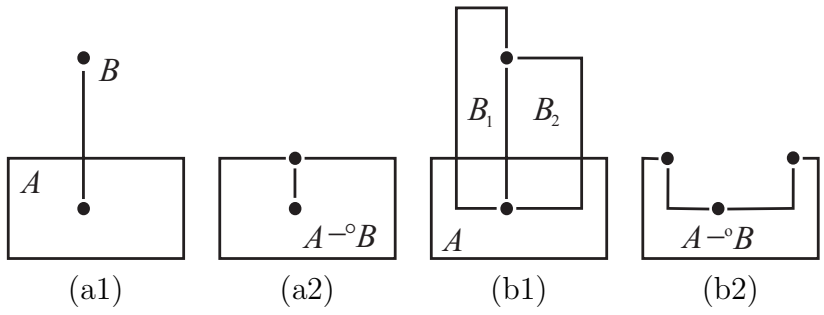


Figure 13: Set-difference operations between cellular objects.

The Djinn object union is obtained by uniting the results of intersection with those of difference, without boundary completion:

$$
\begin{aligned}
&\cup^\circ \in \mathit{Object} \times \mathit{Object} \rightarrowtail \mathit{Object} \bullet \\
&\quad \forall F, G \in \mathit{Object} \bullet \mathit{SDim}(F) = \mathit{SDim}(G) \bullet \\
&\qquad (\mathit{SDim}(F \cup^\circ G) = \mathit{SDim}(F)) \\
&\qquad \wedge (\mathit{Cells}(F \cup^\circ G) = (F -^f G) \cup \mathit{Cells}(F \cap^\circ G) \cap (G -^f F).
\end{aligned}
$$

Thus, an object union consists of:

- Those parts of the first operand's cells outside the point-set of the second operand.
- Cells created by the pairwise set-intersection of cells from each operand and subsequent stratification, and

□ Those parts of the second operand's cells outside the point-set of the first operand.

Following [Middleditch 1992] and [Rossignac 1990], when two cells are combined by a set-operation, the new geometric entities of reduced rank that are defined by derivative discontinuities are instantiated by D̈jinn as new cells. For example, the union of two solid objects includes cells of face intersection. In another example (see Figure 14), the union of overlapping objects F and G, each with a single interior and boundary cell, is an eight-celled object with three face cells, three edge cells and two vertex cells.

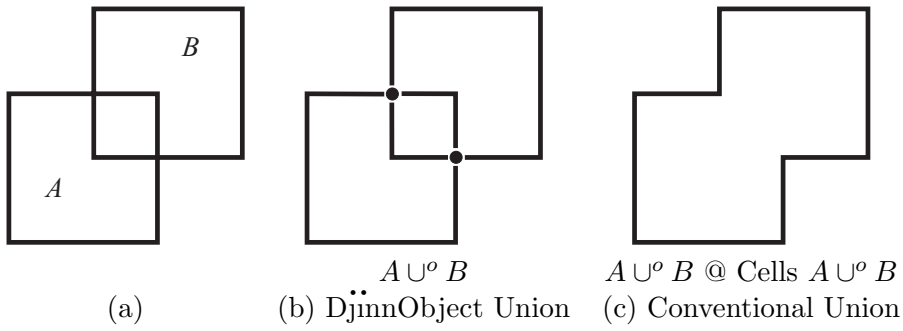


Figure 14: D̈jinn object union using departitioning to obtain the conventional, non-cellular, regularized union.

Applications do not always require the union of two single-celled objects to consist of multiple cells. Given a set which contains all the cells in the D̈jinn object union $A \cup^\circ B$ as *Cells*$(A \cup^\circ B)$,the departitioning operator @ defined in Chapter 2 can be used to obtain the conventional non-cellular regularized union as $(A \cup^\circ B)$@*Cells*$(A \cup^\circ B)$ (see Figure 14).

Figure 15: D̈jinn object union, intersection and difference.

Figure 15 shows the effect of the D̈jinn set-operations on two one-dimensional objects in $\mathcal{E}^3$. Note the difference between this figure and the corresponding operations on the manifold point-sets in Figure 11. The

frontier condition implies that one-dimensional edges must have cells containing their end-points.

Implementation of set-operations

D̈jinn objects also incorporate a 'modification record' (see Chapter C) which logs the number of times that an object has been modified (since the record's contents were last reset) and also a bounding box which indicates a 'cumulative region of change'. Clearly, all the functions described in this chapter will update this modification record (see the definitions in Part II for precise details of the effects of each function).

The D̈jinn set-operations all share the common convention that the result of the operation is left in the object that is given as its first operand, and the object given as the second operand remains unaffected by the operation.

Primary set-operations

The three primary D̈jinn set-operations are as follows:

Unite — Given two objects, modify the first object to be the D̈jinn object which is the union $\cup^\circ$ of the two.

This function modifies the first object such that its point-set (i.e. the union of its cells' point-sets) becomes the conventional (non-regularized) union of the point-sets of its operands. The point-sets of the cells of the result are:

- Those parts of the first object's cells outside the point-set of the second.
- Cells created by the pairwise set-intersection of cells from each operand (i.e. the cells computed by the D̈jinn intersection function).
- Those parts of the second object's cells outside the point-set of the first.

Intersect — Given two objects, modify the first object to be the D̈jinn object intersection $\cap^\circ$ of the two.

This function modifies the first object such that the union of its cells' point-sets becomes the conventional (non-regularized) intersection of the point-sets of the operands. The point-set of each cell of the result is the intersection of the point-set of a cell from each operand.

Subtract Given two objects, modify the first object to be the closure of the Djinn object set-difference $-^\circ$ of the two.

The point-sets of the cells of the result are:

- Those parts of the first object's cells outside the point-set of the second.
- Additional cells from the boundary of the second object, added where necessary to complete cell boundaries, whilst preserving the frontier condition described in Chapter C.

The definition of D̈jinn objects incorporates a map which allows applications to determine which cells of the parent objects contributed to the point-sets of cells in the current object. Naturally, this mechanism is particularly relevant to the D̈jinn set-operations. The following table gives the set of parents for a cell after each operation. The notation $\{(i_A, j_B)\}$ indicates that a given cell was derived from cell i of object A and cell j of object B.

Operation	*Condition*		*Parentage set*
Unite (A, B)	OUT		$\{(\mathsf{OUT}, \mathsf{OUT})\}$
	in A	out of B	$\{(i_A, \mathsf{OUT})\}$
	out of A	in B	$\{(\mathsf{OUT}, i_B)\}$
	in A	in B	$\{(i_A, j_B)\}$
Intersect (A, B)	OUT		$\{(\mathsf{OUT}, \mathsf{OUT})\}, \{(i_A, \mathsf{OUT})\}, \{(\mathsf{OUT}, i_B)\}$
	in A	in B	$\{(i_A, j_B)\}$
Subtract (A, B)	in A	OUT of B	$\{(i_A, \mathsf{OUT})\}$
	OUT		$\{(\mathsf{OUT}, \mathsf{OUT})\}, \{(i_A, j_B)\}, \{(\mathsf{OUT}, i_B)\}$

Part I
6
Swept objects

In general, a *sweep* is the continuous union of a family of point-sets. An *extrusion* is a simple sweep in three dimensions, defined as the region swept by a plane lamina moving along a perpendicular straight line. A *volume of revolution* is created by a lamina perpendicular to the circumference of a circle, moving around it. (Many books about modelling are short of material in this area, but some [Hoffmann 1989, Hoschek 1989] do cover it.)

If G is a finite collection of point-sets, then their union S contains all points that belong to any of them:

$$S = \{p | \exists R \in G \bullet p \in R\}.$$

The continuous union is similar, but applies to a transfinite collection of point-sets. Since it is not possible to define each member of a transfinite set of sets separately, they must be defined as a family; the taxonomy of sweeps is a taxonomy of such families.

Thus, a general swept region is the union of a family of regions defined in terms of a generator region and a directrix[1] transformation that operates on the generator to produce the entire family. The generator is allowed to be an arbitrary manifold of any dimension. In three dimensions that means it may be a point, an edge, a face or a solid.

The directrix is often defined indirectly, using geometry along which the generator is swept, possibly with changing orientation. Generator and directrix geometry of dimension g and d respectively create a swept region of dimension s, where $g, d \leq s \leq g + d, 3$.

[1] We have generalized the conventional use of *directrix* (meaning a curve along which the generator is swept) to mean any continuous family (possibly multivariate) of transformations which map the generator's points into new positions.

Possible sweeps in three-dimensional space are:

g	d	s						
1	1	1	a curve	traverses	a curve	to sweep	a curve
1	1	2	”	”	a curve	”	a face
1	2	2	”	”	a face	”	”
1	2	3	”	”	”	”	a solid
1	3	3	”	”	a solid	”	”
2	1	2	a face	”	a curve	”	a face
2	1	3	”	”	”	”	a solid
2	2	2	”	”	a face	”	a face
2	2	3	”	”	”	”	a solid
2	3	3	”	”	a solid	”	”
3	1	3	a solid	”	a curve	”	”
3	2	3	”	”	a face	”	”
3	3	3	”	”	a solid	”	”

Complementary sweeps

If we view a sweep as a continuous union, then that suggests an analogous continuous intersection operation, in which the intersection S of a set of point-sets G contains all points that belong to all sets:

$$S = \{\mathbf{p} | \forall R \in G \bullet \mathbf{p} \in R\}.$$

Such operations are defined, like sweeps themselves, in terms of a generator region and a directrix. The complement of the generator is swept through the directrix to provide the complement of the result, which is why these are called complementary sweeps [Ilies 1999].

Swept objects may be used to determine if a moving object collides with another object, or indeed to define two objects that do *not* collide: for instance, a rotational sweep of the rotor defines the combustion chamber of a Wankel engine. Conversely, the complementary sweep of the combustion chamber by the inverse rotation defines the rotor.

Directrix transformations

A directrix (i.e. a continuous sequence of point maps) can be represented by a parameterized transformation. For example, a translational sweep of region P in direction $\mathbf{v}$ is defined by the set of points that can be reached from a point in P by a translation of vector $\mathbf{v}$ or part of it:

$$\{\mathbf{r} | \exists \mathbf{p} \in P \bullet \exists t \in [0 \ldots 1] \bullet \mathbf{r} = \mathbf{p} + t\mathbf{v}\}.$$

In general, transformation sweeps and complementary sweeps take the following forms, respectively:

$$\{\mathbf{r} | \exists \mathbf{p} \in P \bullet \exists t \in [0 \ldots 1] \bullet \mathbf{r} = \mathit{Transformation}\,(\mathbf{p}, t)\},$$

$$\{\mathbf{r} | \forall t \in [0 \ldots 1] \bullet \exists \mathbf{p}_t \in P \bullet \mathbf{r} = \mathit{Transformation}\,(\mathbf{p}_k, t)\},$$

where *Transformation* is an arbitrary point map and t is multivariate. Such a transformation directrix is only useful for specifying swept regions if it is defined by geometrically meaningful sets of constant and variable parameters. For example, a translational sweep is defined by a constant translation vector $\mathbf{v}$ and the relative distance t along that vector.

Although any point map can be used to generate a swept region, only the linear transformations appear to have geometrically meaningful definitions for sweeping. Geometrical control is reflected by the following classification of linear transformations:

Translation	Preserves direction
Rotation	Preserves distances from the axis of rotation
Rigid body motion (translation and rotation)	Preserves distances between points
Similarity	Preserves shape
Shear	Preserves distance in a specific direction
Affine transformation (all the above)	Preserves distance ratios on straight lines
Projective transformation	Preserves cross-ratios of distances along a straight line

Bivariate and trivariate translational sweeps require two and three constant-direction vectors respectively; but the same effect may be achieved with three successive univariate translational sweeps: so the merit of bivariate and trivariate transformations is not obvious. Furthermore, a transformation must be parameterized for use in sweeping, and geometrically meaningful univariate parameterizations of general affine and projective transformations do not exist. Useful sweep transformations are therefore limited to the simple linear transformations that have an obvious univariate parameterization:

Translation	Distance in a given direction
Rotation	Angle
Differential scaling	Multiplier for three scale factors

General affine sweeps can be created by concatenating such transformations and using the same parameter for each. For example, rotation and translation can be used to create a swept region with a helical spine. In combination with a second rotation, a twist can be imparted to the cross-section. Alternatively, a tapered region may be obtained by combining translation and scaling.

Djinn therefore supports sweeps and complementary sweeps defined with a directrix that is a univariate translation, rotation, or scaling; or an affine combination of these transformations.

Minkowski operators

A translational sweep may be defined in an alternative way, as:

$$\{\mathbf{r} | \exists \mathbf{p} \in P \bullet \exists \mathbf{t} \in T \bullet \mathbf{r} = \mathbf{p} + \mathbf{t}\},$$

where T is the set of vectors $\{k\mathbf{v} | 0 \leq k \leq 1\}$. This is a special case of the less familiar Minkowski sum operator for combining two sets of points[2]:

$$P \oplus Q = \{\mathbf{r} | \exists \mathbf{p} \in P \bullet \exists \mathbf{q} \in Q \bullet \mathbf{r} = \mathbf{p} + \mathbf{q}\}.$$

For example, an extrusion is the Minkowski sum of an interval of the z-coordinate axis and a two-dimensional region of the xy-plane. A Minkowski sum R of two-dimensional regions P and Q is illustrated in Figure 16. Thus, a Minkowski sum of two point-sets is a sweep of one point-set by a directrix that is a translation by a vector that ranges over all points in the second set.

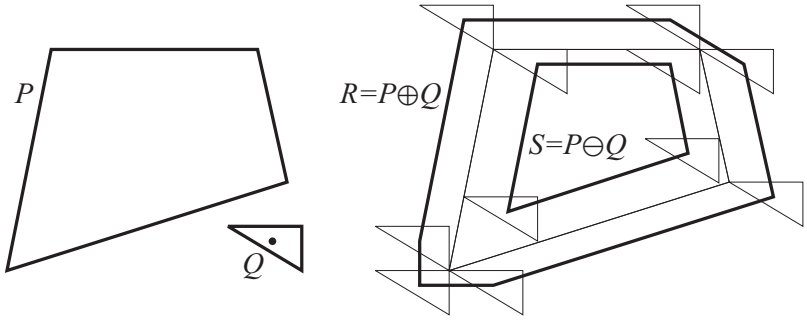


Figure 16: $R = P \oplus Q, S = (P \ominus Q)$.

The Minkowski sum, which has been called the set sum and the vector sum [Yaglom 1961, Grunbaum 1976, Lyusternic 1963], may also be defined

[2]Since the difference between two points is a direction vector, not a point, the equality for $\mathbf{r}$ in this equation should really be $\mathbf{r} = \mathbf{p} + (\mathbf{q} - \mathbf{s})$, where $\mathbf{s}$ is the origin.

as the continuous union of a region P shifted by point vectors contained within region Q:

$$P \oplus Q = \cup_{\text{all } \mathbf{q} \text{ in } Q}\{\mathbf{r} | \exists \mathbf{p} \in P \bullet \mathbf{r} = \mathbf{p} + \mathbf{q}\}.$$

This definition suggests an analogous intersection operation, $\cap_{\text{all } \mathbf{q} \text{ in } Q}\{P \oplus \mathbf{q}\}$, which may also be written in terms of the Minkowski sum of the sets P and Q [Middleditch 1988]:

$$(P' \oplus Q)' = P \ominus Q = \cap_{\text{all } \mathbf{q} \text{ in } Q}\{\mathbf{r} | \exists \mathbf{p} \in P \bullet \mathbf{r} = \mathbf{p} + \mathbf{q}\}.$$

This operator $\ominus$ is a special case of the general complementary sweep; here it will be called the complementary Minkowski sum.

The Minkowski sum has no unique inverse, but the complementary Minkowski sum provides an inverse for some operands. The complementary Minkowski sum S for polygonal sets P and Q is also illustrated in Figure 16.

Directrix geometry

Sweeps have traditionally been defined in terms requiring univariate geometric objects as directrices. This definition is included within the approach described above by regarding the geometric object as a way of defining linear transformations[3].

For example, an extrusion is a sweep along a straight line with a transformation which is a translation from the start of the line to its general point, parameterized by distance along the line. A spring may be described by a disc sweeping along a helix, while the disc remains orthogonal to that helix. Here again the translation is to the general point on the curve, but there is also an orientation derived from the differential geometry of the curve. One may also wish the scale of the image of the generator to vary with position along the curve.

Therefore, for convenience, D̈jinn also provides sweeps and complementary sweeps over a point-set with an orientation transformation that depends on the geometry of that point-set, augmented by transformations that define a taper and twist.

Frenet frames

The obvious orientation at a point on a curve is defined by its Frenet frame [O'Neill 1966], $F = [\mathbf{t}, \mathbf{n}, \mathbf{b}]^T$, where $\mathbf{t}$, $\mathbf{n}$ and $\mathbf{b}$ are the tangent vector,

[3]This approach can, in principle, be extended to directrix objects of manifold dimensions greater than one, but Djinn does not incorporate this extension.

principal normal and binormal respectively:

$\mathbf{t} = d\mathbf{q}/ds$, at point $\mathbf{q}$ as a function of arc length s.
$k\mathbf{n} = d\mathbf{t}/ds$.
$\mathbf{b} = \mathbf{t} \times \mathbf{n}$.

Unfortunately, the Frenet frame is defined only for curves with non-zero first and second derivatives (i.e. non-zero tangent vector and curvature). Thus, it is not defined on straight-line segments nor at inflection points. In the latter case, even if the derivatives are continuous, the Frenet frame is a discontinuous function of position at the inflection; it rotates through 180°. It is also discontinuous if either the tangent direction or the curvature are discontinuous. If either changes rapidly, the Frenet frame can change through a large angle over an arbitrarily small segment of a curve.

Swept regions may be defined by a directrix which is a translation to a point on a curve followed by the Frenet frame rotation about that point. Non-smooth swept regions are to be expected when the curve has tangent discontinuities but, unfortunately, smooth curves also yield non-smooth sweeps at discontinuities in the Frenet frame as a function of position.

A modified Frenet frame has been used by [Klok 1986] to extend the range of smooth sweep trajectories to include non-linear plane curves with points of zero curvature and curves without a continuous second derivative. This approach also accommodates curves that are piecewise planar, when the modified frame is discontinuous where the pieces of plane join. Klok also describes an alternative moving frame which has minimum total rotation around the curve's tangent vector. Sweeps based on this frame are smooth, provided that the directrix curve is not closed and that it consists of twice-differentiable segments with non-zero first derivative and continuous tangent at the segment boundaries. If the curve is closed, only symmetric cross-sections yield a smooth sweep.

Since smooth sweeps should result from smooth directrix curves, D̈jinn uses a frame that depends only on the differential properties of the Frenet frame at points where it is continuous. At the start of the curve, this frame is the same as the Frenet frame. It agrees with the Frenet frame until a point of zero curvature and does not change at that point. Immediately after such a point the D̈jinn frame F' differs from the Frenet frame F by the magnitude of the Frenet frame's discontinuity at that point:

$$F' = \sum_{i=1}^{n} \int_{a_i}^{b_i} \frac{dF}{ds} ds,$$

where $(a_i \ldots b_i)$ are intervals of continuous Frenet frame function of arc length s, and n is the number of such intervals between the start of the

edge, and the point at which the length $s = b_n$. In this way, smooth directrix curves result in smooth sweeps, if they are not closed; closed curves may give a sweep with a single discontinuity, but this occurs only when the pure Frenet frame orientation has discontinuities.

This approach also accommodates directrix curves with straight segments; the frame merely remains unchanged along such segments. If the start of the directrix is straight, the first well-defined Frenet frame is used over that segment. Thus, the frame is undefined only if the directrix is entirely straight. In that case, the frame of the cross-section is used, and the sweep is equivalent to a translational sweep and a Minkowski sum.

Cellular structure of sweeps

Consider two objects with point-sets that are independent, in the sense that all vectors between points in one object are linearly independent of those between points in the other. An example is a two-dimensional object in the xy-plane and a tangent-continuous three-dimensional curve with no tangent direction parallel to that plane. The Minkowski sum of two point-sets, each comprising a cell from one of the objects, is disjoint from all other such Minkowski sums. Performing the Minkowski sum cell-wise for two linearly independent cellular objects can sensibly reflect the internal structure of both the operands. For example, the cells of a partitioned lamina can be extruded independently to create a partitioned extrusion.

A separate directrix, derived from appropriate geometry, can be applied to each cell of that geometry. Thus, a general cellular sweep creates point-sets, each of which is derived from a cell of the generator and a cell of the directrix. The object-independence condition can be generalized for such arbitrary sweeps. Thus, if all swept point-sets are disjoint, they can be used as cells of the result.

Unfortunately, when the objects are not linearly independent, the fact that the Minkowski sum is a symmetric continuous union renders a sensible interpretation of cellular structure much less obvious. Nevertheless, applications may require cellular structure to remain intact during simple operations such as offsetting. In three dimensions, this may be considered as the Minkowski sum of an object with a sphere centred at the origin. The two-dimensional analogue of this operation is shown in Figure 17: the point-set of object C is the Minkowski sum of the point-sets of objects A and B. In this figure there is a plausible internal cellular structure for C; the structure of A is replicated in the interior of C and a consistent cellular structure is shown in the offset region.

Since the Minkowski sum of two objects is larger than either operand, self-intersection may occur, as shown in Figure 18. In this case, a rea-

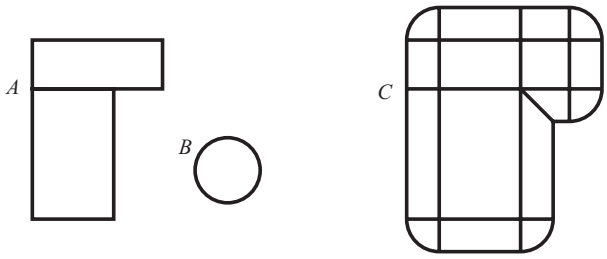


Figure 17: Simple offsetting.

sonable approach to the cellular structure of the result is to treat the region of overlap in the same way as the union of two distinct objects (see Chapter 5). Thus, each separate two-dimensional region of Figure 18 is a cell, each edge common to the same pair of two-dimensional cells is a one-dimensional cell, and each point common to the same pair of one-dimensional cells is a zero-dimensional cell.

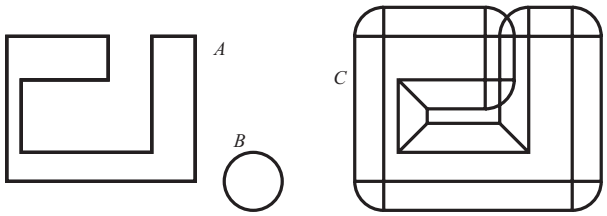


Figure 18: Self-intersection during offsetting.

Similar problems occur with general sweeps that self-intersect (i.e. when a swept region contains any point generated from more than one generator point, or more than one directrix parameter value, or both). Self-intersections can result from distinct or adjacent parts of the sweep directrix as shown in Figures 19(a) and 19(b) respectively, where a straight-line segment is swept orthogonal to a curve (not shown in the figure).

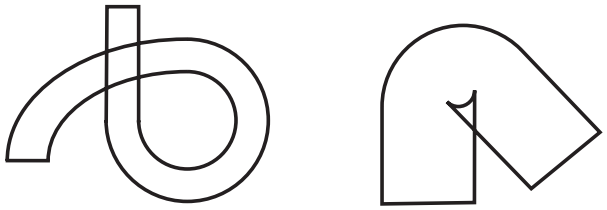


Figure 19: Self-intersections caused by a) distinct or b) distant parts of a sweep.

The problems exemplified by Figures 17, 18 and 19 may all be resolved by generalizing the approach taken for set-union (see Chapter 5). Thus, each point in the swept region is derived from a set of pairs, each consisting

of a generator point and a directrix parameter (or point). If there are no self-intersections, a single point and a parameter pair defines the resulting points. Points in the region of overlap in Figure 19 are defined by two pairs and an infinite set of pairs define most points within Figures 17 and 18. A (possibly infinite) set of generator point-directrix parameter pairs corresponds to a finite multi-set of generator cell-directrix cell pairs. D̈jinn creates a cell from each set of points in the swept region that result from the same multi-set of generator cell-directrix cell pairs. Thus, for example, the lower left rectangle with four two-dimensional cells in Figure 20 is swept along the horizontal line-segment to create the upper rectangle with ten two-dimensional cells. See the table in Chapter 3 for details of the appropriate parentage map for cells in the result.

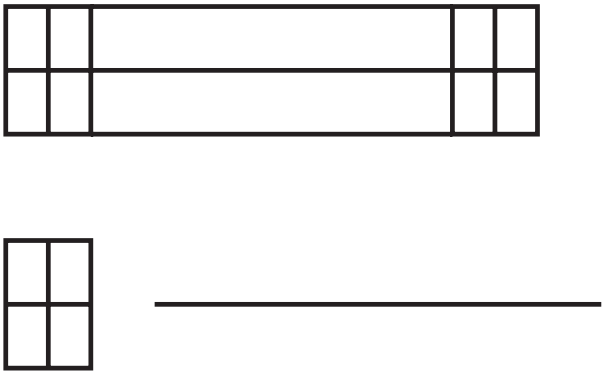


Figure 20: Sweeping a cell structure.

In Figure 19, the generator is a straight-line segment with two cells, its interior and the pair of end-points; the directrix consists of an open curve cell and a cell with two end-points. The interior of the generator gives rise to four swept cells: its start position, its end position, the region of self-intersection and the remainder of the sweep (disjoint in Figure 19(a)). The generator points give rise to five cells: their start positions, their end positions, the parallel curves outside the self-intersection, and the four curve segments, and four vertices, bounding the self-intersection.

The same approach to regions of self-intersection applies to the complementary sweeps. It is evident from the continued intersection formulation of the complementary Minkowski sum,

$$P \ominus Q = \cap_{\text{all } \mathbf{q} \in Q} \{\mathbf{r} | \exists \mathbf{p} \in P \bullet \mathbf{r} = \mathbf{p} + \mathbf{q}\},$$

that all points of the result are derived from all points of Q (i.e. from all cells of Q and possibly some cells of P). No cell of P is evident in the complementary Minkowski sum of two linearly independent point-sets because it is empty. If they are dependent, the first operand is eroded by

the second and cells of the result satisfy the condition for general sweep cellularity.

Djinn sweep functions

Sweep creation

General and complementary sweeps can be created by the D̈jinn functions:

General USweep — Given a generator object, an open directrix curve and an affine transformation T, compute the object created by the union of this transformation of the generator object for each point of the directrix curve.

General ISweep — Given a generator object, an open directrix curve and an affine transformation T, compute the object created by the intersection of this transformation of the generator object for each point of the directrix curve.

The transformation consists of a translation to the point of the directrix, preceded by a transformation $T(\mathbf{p})$ that depends on the coordinates of the directrix point $\mathbf{p}$, such that $T(\mathbf{p}) = I$ (the identity transformation) at the start of the curve, $T(\mathbf{p}) = T$ at the end of the curve, and $T(\mathbf{p})$ varies linearly with length along the curve. The curve must be open so that it has a well-defined start and end.

D̈jinn provides two Minkowski combination functions for objects with internal cellular structure (i.e. *Dilate*—Minkowski sum, and *Erode*—complementary Minkowski sum):

Dilate — Given two objects, dilate the first object by the second using the Minkowski sum operator as already described.

Erode — Given two objects, erode the first object by the second using the complementary Minkowski sum operator as described previously.

The sweep and complementary sweep of an arbitrary object along a curve, controlled using the Frenet frame rather than a general transformation, are created by the functions:

Frenet USweep Given a generator object, a one-dimensional directrix object, a total twist angle and a taper scale factor, compute the object created by continuous union of the generator object swept over the directrix, using the Frenet frame of the directrix to define the orientation of the generator as specified above. The generator object also is rotated about the curve tangent and scaled, linearly with distance along the curve.

Frenet ISweep Given a generator object, a one-dimensional directrix object, a total twist angle and taper scale factor, compute the object created by continuous intersection of the generator object swept over the directrix, using the Frenet frame of the directrix to define the orientation of the generator as defined previously. The generator object also is rotated about the curve tangent and scaled, linearly with distance along the curve.

The most common swept volumes are extrusions, volumes of revolution and volumes with helical spines. Extrusions can be created by the general sweep function, the Frenet sweep function or the Minkowski sum. Volumes of revolution and screws can be created by the general or Frenet sweep functions. However, D̈jinn provides two specialist functions, essentially for convenience:

Extrude Given a one-dimensional cross-section (in x) or a two-dimensional cross-section (in x and y), and a draft angle, extrude the one or two-dimensional cross-section in the z- or y-direction respectively to create an object of unit height with sides at a constant draft taper angle.

Spin Given a two-dimensional cross-section (in x and y) sweep the cross-section around the z-axis by 2π to create a complete volume of revolution.

Labels and attributes

Either or both operands of all the D̈jinn functions for creating sweeps may have internal cellular structure, resulting in a swept region with internal structure as described previously. All cells that result from a sweep operation are derived from at least one cell of each operand and the parentage map can be used to discover these parent cells.

No attribute is assigned to cells resulting from a sweep operation. The application can subsequently compute appropriate attributes for each new cell by navigating to all Djinn cells (see Chapter 3), retrieving their attributes (see Chapter 1) and interrogating the cells (see Chapter 8) to obtain appropriate geometric measures: for example, to compute attributes that depend on cell volume.

Part I
7
Local changes of shape

Tweaking

Tweaks are changes to the shape of an object which are caused by changes to some of the geometric entities that comprise it. For example, translating the top face of a cuboid changes its height; translating an edge changes its cross-section; but if a vertex is moved the edges of the former cuboid may no longer be compatible with six planar faces. Tweaks of this sort cannot be interpreted without some further information. And, further, if the vertex is moved through a face, there will be topological as well as geometrical problems to resolve.

Tweaks have traditionally been used as a convenient way to edit boundary models when the changes required are small enough to leave at least the topological structure of the object unchanged. However, in order that more complicated changes may be achieved, some modellers permit tweaks in which a part of the model may temporarily collide with or even overlap another part, on the assumption that the subsequent tweaks in the sequence will restore topological integrity.

The alteration of individual half-spaces in CSG modelling also has something of the flavour of a tweak about it, although topological problems are not encountered in this case.

D̈jinn incorporates a generalization of these concepts, and supports well-defined small changes in a way suitable for implementation by parametric and boundary-feature-based modelling. Of necessity, it has been accepted that the exact geometry resulting from a tweak may not be easy to pin down in a way that is genuinely independent of the underlying modelling technology. Therefore tweaks are defined only as far as setting criteria which the chosen geometry should satisfy; and thus different modellers may produce different results within a defined equivalence class.

Basic concepts

A D̈jinn tweak is specified as the application of a transformation to a collection of cells within an object. In general, because the transformation is not applied to all the cells, some distortion of the object occurs: which is, hopefully, the change required.

This collection of cells is called the *move-set*, because the cells to be modified are placed in it and moved according to the transformation. Additionally, a minimal number of adjacent cells are modified so that the adjacency relationships between cells are the same after the tweak as before. These adjacent cells are called the *border set*, because they provide a border between the move-set and the *static set*, which contains the cells that do not move at all.

In order to describe the behaviour of the cells under a tweak, two further concepts are required: the *embedding* of a cell, which is a point-set within which the cell lies, and the *extent*, which is the amount of the embedding which the cell occupies.

Valid tweaks are those which do not disrupt the cellular structure of the object. It is often the case that a small tweak is valid, but a larger tweak of the same sort (i.e. a transformation through a larger distance or angle) would be invalid. D̈jinn distinguishes between *small tweaks* and *large tweaks*.

The change of geometry that occurs during the tweak need not be viewed as a discrete jump, but as a smooth modification starting with the original model, and finishing with one that incorporates the required change. This approach permits the concept of the *amount* of transformation to be introduced. Typically, a large tweak which is invalid implies the possibility of a small tweak which would be valid, using a smaller amount of the same transformation and the same collection of cells to be tweaked.

The concepts introduced here are closely-related to those of BR-variance introduced and elaborated by [Ragothama 1998]. Their ideas could well be implemented in terms of the parentage maps returned by D̈jinn after operations altering the bounding relations between cells; but this has not been explored in sufficient detail to be included here.

The basic concepts introduced above will now be refined.

Embeddings

The *embedding* of a cell is a point-set of the same dimensionality, containing—at least—the points of the cell and of its bounding cells. In order that cells can stretch during a tweak, where necessary, it must be possible to extrapolate the embedding. The exact extrapolation is not

constrained by D̈jinn, except that, if a cell is initially flat (solid, planar or linear), the extrapolation of its embedding remains flat.

The embedding of a particular cell depends only on the current state of that cell. It does not depend on the tweak being carried out, or on any other cell, and it does not depend on any history. This means that a truncation followed by an extrapolation need not recreate its original point-set exactly; that is an inevitable corollary of having no hidden state in D̈jinn objects.

The embedding is independent of transformation (at least as far as linear transformations are concerned): thus the embedding of a transformed cell is the transformation of the embedding of the original.

Recall from Chapter 3 that the *bounders* (or *bounds*) of a cell are those cells which together form its frontier: the *boundeds* of a cell are those cells of which the given one is a bounded.

Extents

The *extent* of a cell is the amount of the embedding which the cell occupies. The extent may change either by truncation of the cell within the embedding or by the extrapolation of the embedding to permit the cell to be expanded.

Transformations

D̈jinn transformations are $1:1$ maps of space to itself. Such maps have a Jacobian of the same sign at every point. Transformations which have a positive Jacobian can be viewed as resulting from a continuous motion of every point in the space as a function of a scalar parameter.

The exact motion is not constrained by D̈jinn. However an implementation would have to be particularly perverse not to use the obvious paths for translations and for small rotations.

Because a transformation does not always define a path uniquely, the path to be used for the tweak must be specified by a transformation sufficiently small that the path is unambiguous. For example, the 180° rotation could be expressed as two 90° rotations, or four 45° rotations, or four 22.5° rotations, and so on.

Collections

The concept of a *collection* is used here in exactly the same way as elsewhere in D̈jinn. The collection of cells is given to the tweaking function by passing a collection of cell labels (a label-set) as an argument.

The collection used for a tweak is normally a proper subset of the entire object; a tweak could be applied to all the cells in the object, but the effect of this would be identical to a transformation of the object as a whole.

A careful choice of which cells to include in the collection will allow the application to select the most appropriate outcome of the tweak.

The difference between small and large tweaks

The amount of a tweak which can be applied, while remaining valid, may be limited by a number of mechanisms.

1. Part of the move-set or the border set may collide with part of the border set or the static set.
2. Two embeddings, with an intersection that defines the embedding of a cell, may reach a point where that intersection changes in topology.
3. An implementation may be unable to extrapolate an embedding beyond a certain amount.

A small tweak is one which hits none of these limits.

In some cases, no movement at all may be possible. This can happen when, for example, the specification of a tweak requires the movement of just one of a large number of faces incident on a single edge, while the edge itself remains unchanged.

Small tweaks

As outlined above, small tweaks are those which can be achieved without any change to the number or adjacency of the cells of the object.

Small tweak — Given a Djinn object, a set of cells to be transformed, a transformation, and a real fraction, try to apply the specified fraction of the transformation to the cells. Return the fraction that was applied. If this is less than the required fraction then the object is unchanged.

Thus the small tweak function, *STweak*, is a map from an original object F, a set of cells S within it, a transformation T and a real a to a new object F'. A collection of input arguments is within the domain of *STweak* only if the new object has exactly the same cell adjacencies as the original.

The smallness of a tweak can be clearly be assured if there exists a continuous 1 : 1 mapping of the entire space which carries the transformed cells into their new positions and keeps the untransformed cells static. One possibility would be to specify tweaks in terms of such transformations, but then it would be too hard to specify many of the useful cases. Note that the Schoenfliess theorem [Cairns 1951, Brown 1960] guarantees the inverse: namely that such a transformation exists if the deformations of the cells do not cause any changes in adjacency. It is not necessary to construct the transformation explicitly.

Move set

The move-set is defined as the minimal set that contains:

1. All the cells in the input collection.
2. Those cells with all bounds lying in the move-set.
3. Those cells with all boundeds lying in the move-set.

The last two conditions need to be applied transitively.

Also included in the move-set are any cells where two bounded cells have coincident embeddings which bound both a cell of the move-set and also one not in the move-set. This is because the normal rule (see below) becomes indeterminate in this case. This is particularly relevant when a cell of full dimension is in the move-set. Such a cell always shares its flat embedding with all other solid cells.

The embeddings of all cells in the move-set are transformed during the tweak. The extent of any of these cells may change, depending on what happens to its neighbours. Clearly if a cell in the move-set has all its bounds in the move-set also, its extent will not change.

Static set

The static set contains those cells which are not part of the move-set or the border set (see below).

The embedding of a cell in the static set is left unchanged by the tweak, but its extent may change if it has a bound in the move or border sets.

Border set

The border set contains cells which must be moved in a way compatible with both the move-set and the static set. The cells comprising the

border set are found by moving outwards from the move-set, through the bounding relation. Moving through bounding relations in one direction increases dimensionality, and in the other decreases it.

The following rules are set up to minimize the number of cells in the border set.

Increasing dimension: If a cell A, in the move-set or already in the border set, bounds a cell B not in the move-set, then the embedding of B must change in such a way that it continues to contain A. The extent of B also changes so that A remains a bound.

This requirement is naturally transitive through the dimensions. A vertex in the move-set causes the edges which it bounds directly, and the faces which it bounds indirectly, and also the solid cells which it bounds very indirectly, all to have their embeddings modified. There is no possibility of these modifications becoming incompatible.

The same rule applies when a vertex bounds a face directly, for example at the vertex of a cone.

The only exception of note is that when the new embedding of a cell A lies in the original embedding of B, the cell B changes only in extent. This applies in particular when B is of full dimension, so that solid cells only ever need to change in extent, not in embedding.

This process clearly terminates at the end of the transitive ascent through the dimensions.

Decreasing dimension: Here the dimensions must be considered separately.

1. It has already been shown that, if a solid cell B is in the move-set, all its bounds need to be placed in the move-set also.
2. If a face cell B, that is in the move-set or already in the border set, is bounded by an edge cell A not in the move-set, then the new embedding for A depends on the number of face cells which it bounds:
   (a) If there are no other face cells, the embedding of A is transformed as if it were in the move-set. Indeed, the earlier rules put A into the move-set for exactly this reason.
   (b) If there is one other face cell (or more than one but all with coincident embeddings), then the new embedding for A is the intersection of the new embedding of B with the original embedding of the other face, or faces; these need to change only in extent, not embedding, and the propagation in that direction terminates.
   (c) If A bounds more than one other face cell, no small tweak is possible, and the permissible amount of tweak is zero.

3. If a face cell B in the move-set is bounded directly by a vertex cell A, exactly similar rules apply, except that any edge cells which A bounds are now considered too.
4. If an edge cell B, in the move-set or already in the border set, is bounded by a vertex cell A not in the move-set, then the faces bounded by B will be compatible with B by previous rules, and will therefore be compatible with any position of A matching B. In the situation that A bounds three edge cells and indirectly three face cells, the new embedding for A is chosen to be the intersection of the embedding of B with the embedding of the face not bounded by B. This convention terminates the propagation most quickly. An exception occurs when the new embedding of B lies in the embedding of that third face. That gives an indeterminate result; it is handled exactly like the coincident embeddings mentioned above, by adding A into the move-set.

Summary

In summary, individual cells may be modified in five distinct ways in order to achieve a desired tweak:

1. They may have the tweak transformation applied to them and to all their bounds.
2. They may be 'bent', to achieve compatibility with their bounds.
3. They may be redefined as the intersection of modified boundeds.
4. They may be truncated.
5. They may be extrapolated.

Cases 1 and 4 are well-defined, and require no further comment. The only complication in Case 3 is that the intersection may be of extrapolated embeddings. Case 5 has already been dealt with: the detail of the extrapolation is permitted to be implementation-dependent, provided that the geometry of extrapolation should give at least continuity of tangent between the original and the extrapolation.

That leaves Case 2, in which we must define how cells can be bent. Again, the detail of the modification is permitted to be implementation-dependent provided that:

- □ The embedding of a bent cell must interpolate all the bounds of the cell.

- The perturbation of the embedding inside the bounds should be no larger than the largest distance of the bounds from the original embedding.

The perturbation between two point-sets A and B is defined as the smaller of:

$$\max_A \min_B Dist\,(P \in A; Q \in B)$$
$$\max_B \min_A Dist\,(P \in A; Q \in B),$$

which is not quite the Hausdorff distance, but is more relevant to this context.

Examples

A simple example (see Figure 21) shows how including different cells in the collection leads to different outcomes. In each case the transformation is a small move to the left.

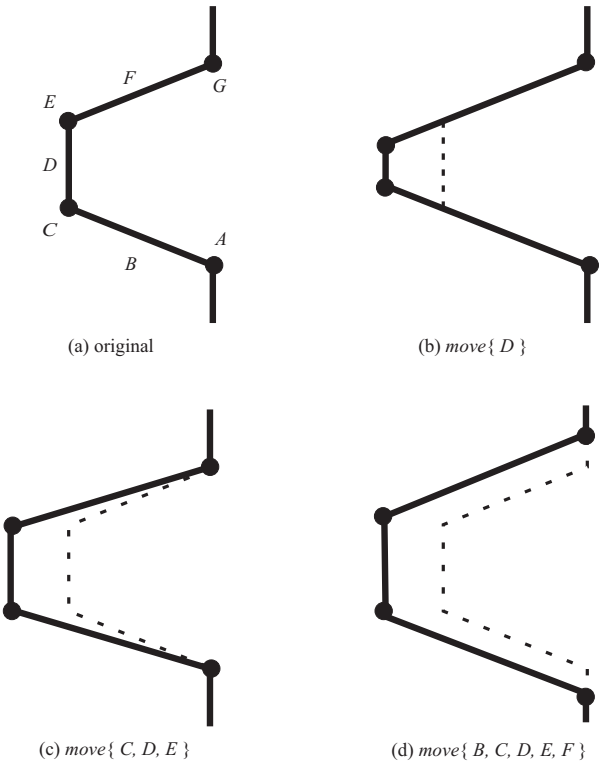


Figure 21: Small tweaks.

At the top right only D is moved. Vertices C and E are computed to lie on the intersections of B and F (extrapolated) with the new position of D. D is truncated to its new end-points.

At the bottom left, D and its end-points C and E have all been moved. This time, B and F have been modified to run between the transformed C and E and the untransformed A and G respectively.

Finally, at the bottom right, the collection transformed includes B and F as well as C, D and E. Now the points A and G have to move to stay on B and F (extrapolated), and they also stay on the original vertical. The vertical edges bounded by A and G are truncated.

Large tweaks

Djinn does not handle large tweaks except to provide feedback to the application when such a tweak is detected. In that case, a scalar value is returned, which is the smallest value of amount that would cause a tweak to be rejected. Any smaller value of amount would have produced a small tweak which would have succeeded. This slightly awkward definition is necessary because the interval of amount which corresponds to a small tweak is an open interval in amount-space, which does not have a maximum member.

Blends

Blends are used to smooth the edges of solid or sheet objects, as shown in Figure 22(a), but they can also be used to join two disjoint objects, as in Figure 22(b). The joining operation is useful for modelling people and animals as 'soft' objects, which are created by constructing scalar fields from key points and lines. The contours of the resulting composite field define the blended objects [Wyvill 1986].

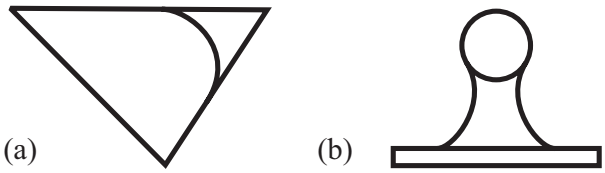


Figure 22: A blend (a) used for smoothing and another (b) used for joining.

Desirable properties

Different applications may require the point-set that comprises a blend to satisfy various different properties, such as the following:

- The bounding surface of the blend point-set should join that of the unblended point-set, or sets, with a prescribed degree of derivative continuity.
- The blend surface is required to have a circular cross-section with a prescribed constant or variable radius.
- The shape of the cross-section of the blend surface[1] should be controllable by one or more parameters.
- The bounding surface of the blend and that of the unblended point-set or sets must meet along one or more prescribed curves.
- The blend is required to exist only within prescribed distances of the unblended point-sets.

Note that it is not possible for all of the conditions listed above to be satisfied simultaneously. Indeed, part of the difficulty of finding a useful but precise definition for blend point-sets is that different applications have such different requirements.

Definitions

The literature is replete with statements to the effect that the actual form of a blend is not significant provided that it is 'smooth' and joins the unblended object 'correctly'. This implies a looseness in the blend specification, because there is a range of suitable results that will satisfy requirements. The implementors of some commercial systems apparently take this to mean that the definition of the blending algorithm itself can be imprecise.

In contrast, the D̈jinn philosophy demands that any blend is a well-defined point-set; or, at least, the point-set must be a member of a well-defined equivalence class (i.e. a class of point-sets which satisfy explicit properties, such as tangency with other point-sets). That is necessary because the point-set has to be combined, by union or subtraction, with the original object during the blending operation. It is not acceptable to define blends by vague imperatives such as: "blend all edges with a constant-radius blend and deal appropriately with all resulting cliff edges". Definitions of blend point-sets should ideally have the following properties:

- Blends depend only on the unblended geometry and shape-control parameters.
- The definition of symmetric multi-face blends is independent of the order in which the faces are blended[2]. For example, take three faces

[1] This is sometimes called the 'pinch factor', 'stiffness', 'softness', or 'thumbweight'.

[2] Asymmetric blends of multiple objects are order-dependent by definition.

A, B and C; if they are blended symmetrically, then the same shape should result from blending A and B then blending the result with C, as by blending A and C then blending the result with B.

- The extent of the blend is explicitly defined. If the user cannot define where a blend ends, it is unreasonable to expect a D̈jinn implementation to do so.

However, in practice, it has proved very difficult to formulate a suitable definition for blending which:

- Results in a well defined point-set.
- Has practically useful results.
- Modifies cellular structure in an intuitively acceptable way.

Thus, well-defined blends are a topic which requires further research, and there can be no completely satisfactory definition for blending in this book.

On the other hand, it would be a serious shortcoming not to provide functions for blending in D̈jinn. Therefore, as a compromise, two different types of blends with known shortcomings are defined in D̈jinn. The first type is based on Minkowski sums, and produces a well-defined point-set, but has the principal disadvantage that its operation is localized in three dimensions, rather than in the embedding of the object. The second type of blend offers more direct user control but, as in the case of tweaks, the corresponding functions are only defined to produce any point-set in an equivalence class defined by its properties.

Cellular structure, attributes and labels

Two distinct possibilities exist for modifying the cellular structure of a two- or three-dimensional solid when blending is performed.

The first possibility, which is probably the more natural, is to regard the blend as locally distorting the shape of the object, in which case the interior cells of the object should be distorted too. In principle, the effects on the three-dimensional cellular structure could be defined by regarding the blend as performing a local mapping of $\mathcal{R}^3 \rightarrow \mathcal{R}^3$, but the precise formulation for such a mapping is elusive.

It is even less clear what should happen to boundary cells of blended solids. For example, the edges of a face intersection should intuitively be replaced by blend faces, which meet the original faces in new edges; but it is not obvious what correspondences should be made. These problems become even more difficult near vertices of the original object.

The second possibility is to regard the effect of a two- or three-dimensional solid blend as the simple union or subtraction of a blend

point-set (depending on whether a concave or convex blend is being created) and to define the effect on the cells from the definition of changes to cellular structure during D̈jinn union and subtraction operations. For this purpose the blend volume is taken to consist of a single cell and its lower-dimensional boundaries as the minimum number of cells that satisfy the frontier-condition invariant of D̈jinn objects (see Chapter C).

The definitions of blend point-sets and corresponding effects on cells and attributes are weaknesses in the current definition of D̈jinn, and therefore also offer fertile areas for further research.

Constant radius rolling-sphere blends

A constant-radius rolling-sphere blend of a single convex edge of a solid object is typically defined procedurally as follows:

A sphere of given radius is rolled inside the solid such that it remains in simultaneous contact with the two faces adjacent to an edge. If more than one contact point is possible, the point closest to the edge is used. The blend point-set to be removed from the object is the connected point-set between the edge and the canal surface formed by the envelope of the sphere in all possible positions. A two-dimensional solid blend of a vertex is created similarly, by placing a disc in contact with the two edges adjacent to that vertex.

The blend point-set to be added in the case of a concave three-dimensional edge (or two-dimensional vertex) is similarly defined by rolling (placing) the sphere in the space outside the object.

Such three-dimensional blends could be terminated in the direction of the edge by the plane containing the sphere centre, and the great circle through the contact points when the sphere reaches a bounding edge of one of the guiding faces. Unfortunately, several disadvantages (that are reminiscent of blend definitions in commercial modelling software) remain:

- Each edge must be uniquely classifiable as convex or concave along its entire length.
- During the motion of the sphere, that part of the canal surface bounding the region to be removed must not intersect any other face of the object.
- If the blend terminates when the sphere reaches a concave edge of a guiding face, blends on adjacent edges join without tangent plane continuity.
- If the blend terminates when the sphere reaches a convex edge of a guiding face, blends on adjacent edges are disjoint.

The last disadvantage is avoided if the sphere motion continues 'in much the same direction as before' while remaining in touching contact with the other face. For example, its centre could be constrained to lie in a plane containing the point of contact with the second face, which has its normal orthogonal to the direction of travel. When the sphere ceases to be in contact with both faces, its motion could be continued in the same direction. This is an *ad hoc* solution and exhibits tangent-plane discontinuity between adjacent edge blends.

These difficulties are typically overcome in the blend definitions of commercial modelling systems by restrictions on the configurations that can be blended: the result is a user manual describing many special-case blends. Situations not covered by such descriptions can usually be found: the remedy is usually to run the software, to see what happens, and then to add another special case to the manual.

In contrast, the following alternative definition for blend point-sets for three-dimensional convex edges and two-dimensional convex vertices is precise:

- Define a region of influence, F(i.e. a point-set that delimits that part of the original object to be blended—see Figure 23).
- Use the complementary Minkowski-sum operation (defined in Chapter 6) to offset the object R inwards by the blend radius r (see Figure 23(a)).
- Similarly offset the result outwards by the blend radius (see Figure 23(a)).
- Use that part of the overall result within the region of influence (i.e. eliminate the dashed blends of Figure 23(a)) to get the result (see Figure 23(b)).

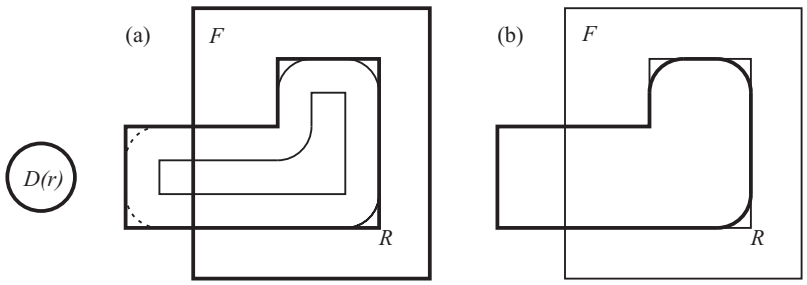


Figure 23: A blend precisely defined for convex vertices.

This leads to a blended object defined[3] by

$$(R - F) \cup (MxBlend\,(R, r) \cap F) = R - (F - MxBlend\,(R, r)),$$

[3]This definition specifies blend geometry. It may or may not reflect the implementation.

where

$MxBlend\,(R, r) = ((R \ominus D(r)) \oplus D(r))$,
$D(r)$ is a disc or spherical point-set of radius r centred at the origin, and $\oplus$ and $\ominus$ are the Minkowski sum and complementary sum operators.

Thus the blend point-set to be subtracted is $(R - MxBlend\,(R, r)) \cap F$.

Blends of concave edges are created by reversing the order of the offset operations:

$$(R - F) \cup (MvBlend\,(R, r) \cap F) = R \cup (F \cap MvBlend\,(R, r)),$$

where $MvBlend\,(R, r) = ((R \oplus D(r)) \ominus D(r))$ and the blend point-set to be added is $(MvBlend\,(R, r) - R) \cap F$.

Care must be taken to construct the region of influence such that unwanted blends do not occur near its boundary. While this is usually possible in two dimensions, it is significantly more difficult in three. In either case, the region of interest must usually be defined in terms of the geometry of the unblended object.

Unfortunately, Minkowski-sum blends of convex edges eliminate object protrusions with a medial surface of radius less than the blend radius [Middleditch 1988], and concave blends eliminate small voids. If both convex and concave edges are blended, the result sometimes depends on the order in which the convex and concave blends are performed [Middleditch 1988] (see Figure 24). Despite these disadvantages, Djinn supports Minkowski blends for the reasons already given.

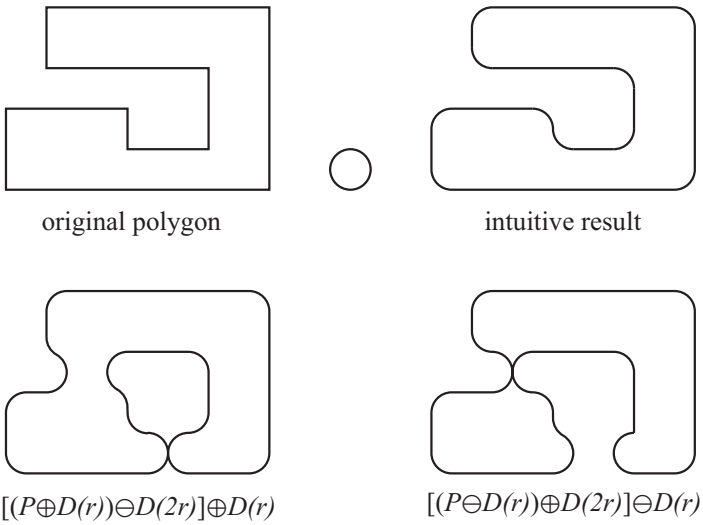


Figure 24: Minkowski-sum blends.

Constant-radius blends on the edges of sheet objects can be achieved in much the same way, although if the region of influence is chosen incautiously, the sheet may become disjoint in cases where a solid part bounded by the same sheet would remain connected.

The constant-radius blend at a bivalent vertex of a wire object could be obtained by using a circular curve segment tangent to the original edges adjoining that vertex. However, it is not entirely clear what to do in the case of short edges adjacent to a vertex, or when more than two edges meet at a vertex.

It is even less clear what properties blends between objects of different dimensionality should have; further work is again needed to assess their potential uses and modes of definition.

Thus, D̈jinn is currently restricted to constant-radius blends between those parts of two- or three-dimensional objects with, respectively, a two- or three-dimensional manifold dimension. This is not as restrictive as it sounds. If it is necessary to blend a three-dimensional sheet object, the user may add extra pieces of sheet to form a closed boundary, fill it, blend the resulting solid, and extract the relevant piece of sheet from the resulting boundary. Blending wire objects can readily be achieved by direct use of procedures to construct new wire geometry and to cut existing curves.

D̈jinn provides the following constant-radius blending functions:

Convex round — Given a solid object R, a region of influence I and a blend radius r, blend those parts of the two-dimensional and three-dimensional object of maximal manifold dimension within the region of influence by subtracting the blend point-set defined previously.

Concave round — Given a solid object R, a region of influence I and a blend radius r, blend those parts of the two- and three-dimensional object of maximal manifold dimension within the region of influence, by uniting the blend point-set defined previously.

In both cases the blend point-set consists of a single two-dimensional solid cell and its bounding cell, or a single three-dimensional solid cell and its bounding cell. Subtraction and union conform to the cellular object set-operations of Chapter 5.

The blend operation primarily affects the point and orientation maps of a D̈jinn object (see Chapter C). Since the surface cells of blend point-sets have their orientation well defined by the outward-directed normals, and the solid cells have the null orientation vector, the Minkowski blends

described previously are captured by an equivalence class of partial functions convex round (*XRound*) and concave round (*VRound*), each with domain and range:

$$((Object) \times P(\mathcal{R}^3) \times \mathcal{R}) \rightarrow (Object).$$

The first parameter of these functions is the D̈jinn object to be blended; the second parameter is the region of influence and the third is the blend radius. The result is the object corresponding to the blend point-set. Each D̈jinn implementation embodies a member of each of these equivalence classes.

Variable-radius sphere and other rolling blends

The Minkowski-sum definition of a constant-radius rolling-sphere blend cannot be adapted for a varying radius. In contrast, the *ad hoc* procedural 'definition' described previously merely requires the radius of the sphere to change as a function of the distance travelled by the sphere's centre.

In the absence of a precise definition, D̈jinn does not support variable-radius rolling-sphere blends. This may not be the serious deficiency it seems. When the exact shape of such a blend is important it is likely to be a relatively simple shape, for example part of a cone or cyclide, which are provided as primitives in D̈jinn. More complicated variable-radius rolling-sphere blends are less likely to need to be precise, and approximations are available using the general blend D̈jinn function described later.

The *ad hoc* procedural definition of constant-radius rolling-sphere blends applies to any shape that can be moved relative to another while remaining in with point contact (e.g. an elliptical disc constrained to be orthogonal to its direction of motion). Once again however, in the absence of a precise definition, D̈jinn does not support such blends.

General edge blends

The difficulties of the *ad hoc* procedural definition of constant-radius rolling-sphere blends arise because the definition is based on surfaces rather than point-sets of maximum manifold dimension. Similar problems arise with more general three-dimensional blends defined by the curves where the blend should join the unblended geometry, usually with tangent-plane continuity. Such ambiguities can be avoided by using three-dimensional regions to contain the blend.

Blends along concave edges of solid objects are formed by uniting a uniformly three-dimensional blend point-set. Each joining curve of a single edge blend is defined by the intersection of a face to be blended and the

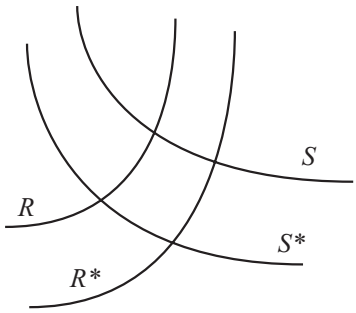


Figure 25: Organization of a general edge blend.

bounding surface of a region of influence derived from the other face. Each blended face is defined in terms of a half-space bounded by its embedding geometry. Thus, in the cross-section of Figure 25, the blend B of a single concave edge of the set-union of half-spaces R and S, with regions of influence R^* and S^* respectively, is a member of the equivalence class defined by the following properties:

- Where the surface of the blend, $\partial(B)$, intersects the surface $\partial(R)$, it does so along the curve $\partial(R) \cap \partial(S^*)$ with a prescribed degree of tangent-plane continuity. The intersection between the surfaces $\partial(B)$ and $\partial(S)$ along the curve $\partial(S) \cap \partial(R^*)$ is similarly defined.
- $R' \cap S^* \supseteq B$, and $S' \cap R^* \supseteq B$. The choice of R^* and S^* can thus be used to control blend termination along the edge. For example, they could be chosen to reduce the size of the blend gradually as it approaches the end-points of the edge.
- The bulk of the blend point-set depends on a shape-control parameter in the range $[0 \ldots 1]$.
- The blend point-set $B \cap R^* \cap S^* - R - S$ is continuous with respect to the shape-control parameter. A value of zero defines an empty point-set and a value of 1 defines the 'fullest' blend that a particular implementation can construct (i.e. if u and u' are two values of the shape parameter, and $u < u'$, then $B(u) \subset B(u')$).
- The bounding surface of the blend point-set is a non-self-intersecting point-set.

In a similar way, the blend along a convex edge is defined by means of a blend point-set to be subtracted from the unblended object. Such point-sets satisfy the equivalent set of properties obtained by consideration of the complement of the object.

This definition looks suspiciously like Liming's method [Hoffmann 1986, Middleditch 1985], but Liming blends are only a subset of the equivalence class defined. Liming blends can be obtained by using, for both R^* and S^*,

an offset of a half-space with a bounding surface that contains the edge to be blended. The equivalence class also includes convolution blends [Bloomenthal 1985], 'soft' objects [Wyvill 1986] and more general blends than any produced by existing techniques. 'Convolution and approximate convolution' is probably the best description of this general Djinn blend definition. It also covers blends between parametric surfaces of solids, provided that the tangency curves for the blend are specified as three-dimensional point-sets.

An equivalence class of blend point-sets for an arbitrary number of half-spaces is defined merely by replacing R and S in the above definitions by a set of half-spaces. Thus, for example, the blend point-set for a 'suitcase-corner' blend is defined using the set of half-spaces that contain the solid in the neighbourhood of the corner vertex; the bounding surfaces of the half-spaces are the embedding geometries of the faces at that vertex.

So-called 'soft' objects [Wyvill 1986], formed from two key points R and S, are created by the union of solids bounded by iso-surfaces of those kernels with a blend point-set that satisfies the previous definition. The regions of influence R^* and S^* usually contain the iso-surfaces of S and R respectively (i.e. there are no blend-joining curves). In this case, the first and last properties of the equivalence class are irrelevant.

Blends on a connected or disjoint network of convex edges of solids, with or without a prescribed degree of continuity between the blends, are defined as two pairs of virtual objects. Intersection edges between the bounding surfaces of these objects contain the edges to be blended and, in the neighbourhood of those edges, the intersection of the objects is equal to the object to be blended. The objects replace the half-spaces R, S, R^* and S^* in the definition above and their regions of influence select the edges to be blended. A similar approach is used for networks of connected or disjoint concave edges, and also for vertex blends by using multiple virtual objects.

Note that the user must choose whether convex or concave edges are being blended during any one operation; material is either added or removed, never both. This is an unfortunate limitation of the current approach, and further research is needed.

Again, note that these operations are only defined for two- or three-dimensional solid objects. The remarks about rounding made above are also valid in this case.

Djinn thus provides the following blending functions for two-dimensional or three-dimensional objects:

Convex blend Given an object, a set of virtual objects each with a region of influence, a degree of continuity and a blend shape control parameter in the range $[0 \ldots 1]$, return a blended object which is the original object minus a blend point-set with the properties defined above.

Concave blend Given an object, a set of virtual objects each with a region of influence, a degree of continuity and a blend shape control parameter in the range $[0 \ldots 1]$, return a blended object which is the original object united with a blend point-set with the properties defined above.

In both cases the blend point-set consists of a single two-dimensional solid cell and its bounding cell, or a single three-dimensional solid cell and its bounding cell. Subtraction and union conform to the set-operations defined for cellular objects in Chapter 5.

The blend operation primarily affects the point and orientation maps of a D̈jinn object (see Chapter C). Since the orientation of the surface cells of blend point-sets is well defined by the outward-directed normals, and the solid cells have the null orientation vector, the Minkowski blends described previously are captured by an equivalence class of partial functions *XBlend* and *VBlend*, each with domain and range:

$$((Object) \times P(P(\mathcal{R}^3)^2) \times \mathcal{R} \times Z \rightarrow (Object).$$

The first parameter of these functions is the D̈jinn object to be blended; the second parameter is the set of pairs of virtual point-sets; the third parameter is the blend fullness; the last parameter is the degree of continuity required. The result is the object corresponding to the blend point-set. Each D̈jinn implementation embodies a member of each of these equivalence classes.

Summary

D̈jinn provides two well-defined equivalence classes of blend formulations for two- and three-dimensional solid objects of maximum manifold dimension:

- Constant-radius blends defined using the Minkowski-sum operator.
- Convolution blends and their approximations.

In contrast to *ad hoc* procedural blends, these blend definitions apply equally well to non-manifold objects (e.g. when more than two faces meet along an edge). Also, since edges are not involved in the definitions, edges

not at tangent-plane discontinuities are of no concern, nor are vertices in the middle of faces.

Part I
8
Interrogation

This chapter describes the facilities for interrogating objects provided in the D̈jinn API, and the rationale for them. Most of the interrogations are geometric enquiries about the point-sets of cells.

Geometric relationships

Object proximity

The distance between two point-sets is the minimum of the distances between a point in each set. When the sets intersect, this distance is zero. Negative distances could be used for overlapping point-sets, but a precise definition such as minimum diameter of the set-intersection is misleading when the objects are concave.

The smallest distance between point-sets is usually measured between one point from each, but sometimes multiple point-pairs have the same separation distance (e.g. concentric spheres). Sometimes there are several distinct regions of equal proximity. Thus, in general, proximity between two point-sets is represented by pairs of connected subsets. All points in one subset are the same distance from the closest point in the other element of the pair.

Since the point-set proximity subsets are surfaces of arbitrary complexity, in general they must be represented by two D̈jinn cellular objects. Each pair of subsets is represented by a cell in each object; correspondence is established by use of the same label. Minimum and maximum distances are computed for object point-sets by the function:

Distance — Given two objects and a Boolean indicating whether the smallest or largest distance is required, compute the coordinates of the points closest together, or furthest apart, or a D̈jinn object that represents regions of closest proximity, or furthest distance, between cells.

The D̈jinn object is computed whenever the closest point-pair is not unique. In this case, point coordinates are obtained by interrogation of the new object. However, since this approach is unnecessarily complicated in most situations, closest points are computed directly whenever they are unique.

The directional distance between two point-sets is the minimum of the distances between a point in each set such that one point is displaced from the other in a given direction.

For objects:

Directional distance — Given two objects, a direction vector and a Boolean indicating whether the smallest or largest distance is required, compute the coordinates of the points closest together, or furthest apart, or a D̈jinn object that represents regions of closest proximity, or furthest distance, in the given direction between cells.

If one object is a point at the origin, this function can be used to compute the two support planes of the second object in a given direction. Such planes can be used by the application to compute the convex hull to any precision, although this may not be very efficient.

Distances between sets of cells in different objects can be computed by suppressing the irrelevant cells.

Object interference

There are four containment relations between two point-sets:

1. Point-sets A and B are disjoint.
2. Point-set A contains point-set B.
3. Point-set B contains point-set A.
4. The point-sets A and B intersect but neither contains the other.

D̈jinn does not provide a single function to compute these results because applications are often interested in the truth of only one condition. Since

the existence of intersection can be computed much more quickly than the other conditions, the four relationships are computed by the following functions in combination:

Interference Given two objects, return TRUE only if the objects intersect.

Containment Given two objects that intersect, return TRUE if the first object contains the second and FALSE otherwise.

Again, relationships between chosen cells can be obtained by suppressing all other cells.

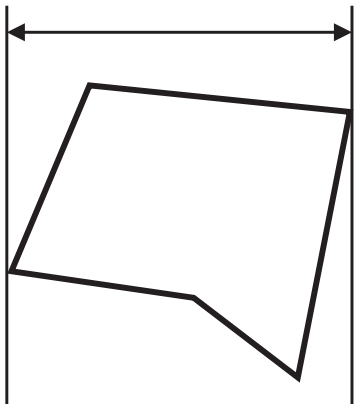


Figure 26: Horizontal diameter of a polygon.

Overlap distances for intersecting point-sets may be characterized in terms of the diameters of their set-intersection. The diameter of a point-set in a given direction is the distance between its two distinct support planar half-space boundaries orthogonal to that direction. A supporting planar half-space contains the point-set and is bounded by a plane that shares at least one common point with the boundary of the point-set. Figure 26 shows the horizontal diameter of a polygon and the two corresponding vertical support lines. Object or cell diameter is computed by the function:

Diameter Given an object, the label of a cell, a direction and a desired accuracy, compute the end-points of the diameter in the given direction.

As before, the normal result is two sets of point coordinates, but if the diameter is not unique, multiple diameter end-points are represented by two corresponding Djinn objects.

The minimum (maximum) overlap of two point-sets is the minimum (maximum) diameter of their set-intersection. For object and cell point-sets it is computed by:

Extreme diameter — Given an object, the label of a cell, a Boolean indicating whether the shortest or longest diameter is required, and a desired accuracy, compute the end-points of the appropriate diameter.

The vector between the end-points of the shortest diameter of the set-intersection of two intersecting convex point-sets defines a translation to separate those sets. If either set is not convex, such translation may cause other parts of the sets to overlap.

Topology

The genus of the point-set of a D̈jinn cell or object is computed by the function:

Genus — Given an object and the label of a cell, return the genus of its point-set.

It is often useful to know if a solid cell is (globally) convex or concave, or if the neighbourhood of a vertex or edge is convex or concave. This information is provided by the function:

Vexity — Given an object and two cell labels, return the vexity of the first cell in the neighbourhood of the second.

If the first cell label is null, the vexity of the complete object is computed; if the second cell label is null, then the vexity of the first cell is computed; if the second cell label is not null, then vexity in the neighbourhood of the second cell is computed. For example, if the first cell is solid and the second is a vertex or edge of that solid, the result indicates the convexity or concavity of the solid in the neighbourhood of that vertex or edge. (In the latter case of an edge, the vexity could vary along it.) If the first cell is an edge or face, vexity is computed with respect to the 'natural' embedding geometry. For example, a face is convex if all geodesics between points in the face lie completely within the face.

Object geometry

Get shape — Given an object and a cell label, return an indication of various characteristics of the cell as detailed below.

This function gets information about the boundedness and dimension of the cell indicated. It also indicates whether the cell corresponds to certain types of primitive shape. Thus the spatial and manifold dimensions of the

cell are returned together with a logical variable indicating if the cell is bounded. A further output shows whether the cell is a point, line, sphere, cylinder, cone, torus or cyclide; all other categories of cell are regarded as 'complicated'.

Bounds

It is often useful to know whether the point-set of an object is empty. Since D̈jinn implementations may use sampling techniques and low accuracy reduces computation time, emptiness is computed to a tolerance:

Empty	Given an object and a desired accuracy, return TRUE only if its point-set is empty.

A TRUE result indicates that no connected component of the object's point-set has a diameter greater than the given accuracy. The efficiency of applications can often be improved by using a simple region that bounds a cell. Such enclosures are most effective if they allow simple computation and provide tight bounds. Spheres are useful because of their invariance under rotation, but they are can make computation more difficult; they are less effective than cuboids parallel to the coordinate axes because they have been found in practice to provide bounds of larger volume. Ideally, bounding enclosures are the smallest possible enclosures, but how near they are depends on the implementor of the D̈jinn function:

Get bounds	Given an object, return an axially aligned cuboid which contains its point-set.

When small changes are made to an object, it may not be necessary to re-compute properties of the entire object. D̈jinn therefore provides facilities to record regions of change within objects defined by the further enhanced definition:

$$\textit{Object} < \textit{Label or cell map} \times \textit{Modification count} \times \textit{Update region}.$$

The update region is a set of points bounded by an axially aligned cuboid. It is accessed by the following function:

Get modification record	Given an object, return the number of modifications and an axially aligned cuboid which bounds the point-sets of cells of the given object that have been modified since the modification record was last reset.

Convex hull

There are no known algorithms to compute the convex hull of an arbitrary point-set, and those for plane-faced polyhedra are quite complicated. Therefore, the following D̈jinn function is designed to enable simple implementations: even the bounding box is a valid result.

Convex hull — Given an object and a desired accuracy, compute an object with a convex point-set that encloses the point-set of the given object.

Integral properties

D̈jinn computes integral properties for the union of the point-sets of a set of cells. This allows an application, for example, to compute the surface area for face cells alone or for surface or patch cells that do not bound solids. A set of labels is used to select cells for property computation; inclusion of the null-cell identifier NULL in the set selects the entire object. The label-set manipulations of Chapter 3 thus allow the computation of properties for sets of cells relevant to the application.

Integral property functions require the application to supply a desired accuracy and their results include an indication of the precision to which those results are computed. This precision has a hidden type due its complexity and to allow implementations to use measures of precision not covered by this specification. Such measures require access functions in addition to those of D̈jinn:

Error bounds — Given a precision, return the upper and lower bound on a scalar result.

Error estimate — Given a precision, return an estimate of the error bound on a scalar result.

D̈jinn also provides the following integral properties of cells and entire objects:

Volume — Given an object, a set of cell labels and a desired accuracy, return the volume of the given cells and the precision of the result.

Naturally, face and edge cells have no effect on the computed volume.

Face area — Given an object, a set of cell labels and a desired accuracy, return the surface area of the given cells and the precision of the result.

Only *one* side of isolated edge cells and face cells shared by adjacent solid cells contributes to the surface area. For example, if the above function were given the labels of all an object's cells and (nominally) zero, the result would be its surface area plus the area of the internal boundaries between cells. As a solid becomes thinner and approaches a sheet, its surface area approaches twice that of the sheet with the same point-set. This constitutes another good reason for associating an explicit dimension with each cell (see Chapter 1).

Edge length — Given an object, a set of cell labels and a desired accuracy, return the sum of the lengths of the given cells and the precision of the result.

Isolated edge cells and edge cells that bound multiple face cells contribute once to the edge length. No contribution is made to the length by edges at discontinuities of surface normal that are not instantiated as cells.

Centroid — Given an object, a set of cell labels and a desired accuracy, return the coordinates of the centroid or centre of gravity of the given cells, and the precision of the result.

Principal moments — Given an object, a Boolean and a desired accuracy, return the principal axes and corresponding principal moments of inertia of the given cells, for unit or true density and the precision of the result.

If any of the principal moments (second moments about the principal axes through the centroid) are equal, the corresponding principal axes remain orthogonal but are otherwise arbitrary. Centroids and the corresponding principal axes depend only on the cells of the greatest dimension in the selected set. If the selected set includes solid cells and isolated sheets, the centre of gravity depends only on the solids.

More general integral properties can be computed using D̈ïnn functions to integrate a user-supplied function over solid, surface, or edge cells:

Scalar integral — Given an object, a set of cell labels, a scalar function F of position and a desired accuracy, compute the integral of $F * dV$ over the volume V of the given cells.

Face integral — Given an object, a set of cell labels, a vector function F of position and a desired accuracy, compute the integral of the dot product $F.dA$ over the directed surface area A of the given cells.

Edge integral Given an object, a set of cell labels, a vector function F of position and a desired accuracy, compute the integral of the dot product $F.ds$ over the directed length s of all the edges of the given cells.

Medial geometry

Some applications require the medial surface of a solid object, medial axis on a sheet object or face of a solid, or the mid-point of edges (see [Blum 1967] for an exposition in two dimensions, [Sheehy 1996, Sherbrooke 1996] for the extension to three dimensions, and [Lavender 1992] for the Voronoi diagram, a closely related property).

Medial surface

The medial surface of a manifold solid point-set is a connected set of faces, inside that point-set, such that all points in those faces (or degenerate edges and points) are equidistant from at least two of its bounding faces. Medial surfaces thus depend on the definition of a face. Djinn defines the medial surface of solid cells in terms of their bounding face cells. Therefore, edges of tangent discontinuity in a face of a solid do not influence its medial-axis topology unless they are instantiated as cells. They do however give rise to derivative discontinuities in the medial surface.

Medial axis

Face cells may have tangent-plane discontinuities but are manifold and thus uniformly bi-variate without bifurcations. The medial axis of a face is a connected set of arcs on that face, such that all points on those arcs (or degenerate points) are equidistant from at least two bounding edges of the face. Distances are measured along geodesic curves in the face. Since Djinn medial axes are defined in terms of cells, edges of tangent discontinuity in a face and points of tangent discontinuity in an edge of a face do not influence its medial axis topology unless they are instantiated as cells. They do however give rise to derivative discontinuities in the medial axis.

Mid-point

Edge cells may have tangent discontinuities but are manifold and thus uniformly univariate without bifurcations. The mid-point of an edge is equidistant from its end-points, as measured along the edge. Mid-points

are defined in terms of edge cells and tangent discontinuities within the cell have no effect.

In order to provide a rich set of medial geometry enquiries, D̈jinn provides functions to create the medial geometry of an object or a set of cells as independent D̈jinn objects:

Medial geometry — Given an object and a set of cell labels, create an object that consists of cells representing the medial geometry of the given cells.

The resulting object contains cells that represent the medial surface, medial axis and mid-points of the solid, surface and face cells respectively. Since boundaries of D̈jinn objects are always included in cells of those objects (see Chapter 1), the medial object has cells in common with the object from which it was derived. These cells are assigned the same label.

If the set of cell labels is empty, the medial geometry of the complete object is calculated.

Relationships between an object and its medial geometry is captured by the parentage map of the medial object (see Chapter 1). The parentage map is a label-to-label map that relates each medial object cell to the closest cells in the original object.

Medial geometry can be used to define interrogations of an object's local properties:

Medial radius — Given an object, a corresponding medial geometry object and the coordinates of a point, return the radius of the medial axis transform at the medial point closest to the given point.

If the medial point lies in a solid cell, the radius is the length of the straight line segment from the medial point through the given point to the cell boundary and orthogonal to that boundary. The radius is thus half the thickness of the solid cell at the given point. If the cell is a face, the radius is the half face width (i.e. the length of a geodesic segment through the given point) orthogonal to an edge. If the cell is an edge, the computed radius is its half of its length.

Differential geometry

D̈jinn provides functions to compute differential properties of curves and surfaces. Such properties could be computed given only the point of concern, but D̈jinn functions also allow the application to provide a cell identifier. Since the application has probably already determined the cell of

interest, this enables faster implementations without penalty. If the cell label is unknown, the null identifier is provided by the application and differential properties of the entire object are computed. This allows differential properties to be provided by implementations that do not support cells.

Differential properties are computed for curves and surfaces that are cells or boundaries of cells. Acceptance of boundary enquiries is more flexible for the application and allows differential properties to be provided by implementations that do not support boundaries.

Since differential properties are not defined at derivative discontinuities, (i.e. tangent-plane surface discontinuities and tangent direction curve discontinuities) the following functions indicate when their results are undefined:

Face normal — Given an object, the label of a cell (or NULL) and the coordinates of a point on the surface of that cell, return the surface normal vector at that point or an indication that it is undefined.

Face curvature — Given an object, the label of a cell (or NULL) and the coordinates of a point on the surface of that cell, return the principal curvatures and the associated direction vectors and the surface normal at that point or an indication that the results are undefined.

The three vectors are returned as an orthonormal surface frame. At spherical points on a surface (including points on a plane), the principal curvatures are equal and the directions are orthogonal but otherwise arbitrary.

Edge direction — Given an object, the label of a cell (or NULL) and the coordinates of a point on an edge of that cell, return the tangent direction vector at that point or an indication that it is undefined.

Edge curvature — Given an object, the label of a cell (or NULL) and the coordinates of a point on an edge of that cell, return the curvature, torsion and tangent, normal and binormal vectors or an indication that the results are undefined.

The three vectors are returned as an ortho-normal Frenet frame. At points which are (instantaneously) on a straight line, the curvature is zero and the normals are orthogonal but otherwise arbitrary.

Applications may need to trace curves (i.e. to obtain a sequence of points on a curve). This can be done using the differential properties and closest

point enquiries or alternatively by the faceting described in Chapter 9.

Sections

Sectioning operations include the computation of the intersection curve between two surfaces and the intersection points between a curve and surface. Although such intersections can be computed by the set-intersection of two objects (see Chapter 5), for convenience D̈jinn provides special-purpose functions for linear intersections (i.e. point-membership tests, ray-casting and plane sectioning). Special-purpose functions are not provided for non-linear sections.

Sections of zero dimensions: point membership tests

D̈jinn provides a function to classify a given point as outside an object or inside a particular cell:

Point membership	Given an object and the coordinates of a point, compute the label of the cell within which it lies or the null identifier.

Sections of one dimension: ray-classification

The intersection of a ray with an object consists of a sequence of disjoint ray segments, each classified as outside the object or inside a particular cell.

Explicit knowledge of cell dimension (see Chapter 1) allows ray-classification algorithms to regularize only when necessary (see Chapter 5) (i.e. to eliminate zero-length ray segments only when executing a regularized set-operator). The segments adjacent to the intersection of the ray with a face cell have the same classification. Unfortunately, this fact cannot be used to represent such intersections because membranes and cracks in a solid both have adjacent segments inside the same cell. Ray-classification data must therefore explicitly include zero length segments with the appropriate classification at face intersections.

Classified rays could be represented by a D̈jinn object or a variable of a special type. With a D̈jinn object, surface penetration points become endpoint cells of the disjoint line segments. These points can be obtained by navigation (see Chapter 3), but it is not possible to access the penetrated cells of the original object directly, nor to obtain surface neighbourhood data at the intersection points. Using proximity enquiries (see the beginning of this chapter) for such data would be inefficient. In addition, a

D̈jinn object does not contain an explicit sequence of ray-segments. A new type could have an explicit sequence and contain penetration point neighbourhood geometry, but that would exclude the wealth of D̈jinn object operations.

Since many applications have no requirement for complete penetration-point neighbourhood information, D̈jinn captures neighbourhoods by the label of the penetrated cell. Since these labels are inherent in the parentage map of object set-intersections, a new type is avoided by a ray-classification object that represents the penetrated cells and has cell labels that represent the sequence of segments in the ray direction:

Ray classification — Given an object, two points defining a ray (an unbounded directed line), and a desired accuracy, compute a new object that represents the classification of the ray with respect to the object (i.e. their set-intersection).

Accuracy control allows an application to specify a low accuracy in the coordinates of the surface penetration points of a ray. This enables the D̈jinn implementation to make object simplifications and thus to accelerate computation. The parentage map of the result provides labels of the penetrated faces and thus access to surface neighbourhood geometry and face attributes (see Chapter 6).

Ray-classification could be used to support ray-casting graphics applications, but the following function is provided for convenience and efficiency:

Surface points — Given an object, a regular array of directed rays parallel to the z-axis, a desired accuracy and an associated array of coordinate, surface normal, attribute triples, update the triples if the first point where the ray penetrates the surface has a more positive coordinate in the given direction.

Edge and point cells have no effect on the result. Each surface point array triple consists of the most positive coordinate of a ray-penetration point in the given direction, the surface normal at that point and the associated face attribute. Surface normals reflect face cell orientation[1]. This function implements depth-buffering because external combination of surface point arrays for multiple objects is significantly less efficient.

[1]Such normals need to be reversed into a positive z-direction for perceived surface colour computations.

Sections of two dimensions: plane sections

Analogous to ray-classification, D̈jinn provides a procedure that intersects an unbounded plane with solid, face and edge cells of an object:

Plane section — Given an object, the coordinates of an unbounded plane and a desired accuracy, compute a new object that represents the classification of the plane with respect to the object (i.e. their set-intersection).

Objects returned by *Plane section* have a cellular structure that reflects that of the sectioned object. They consist of a set of disjoint two-dimensional regions, edge and point cells, each associated with a sectioned cell. The parentage map of the new object provides access to the labels of the sectioned cells and thus to their attributes and to the neighbourhood geometry of the intersection. The explicit dimension of the new cells reflects that of the sectioned cells, as described in Chapter 5.

Part I
9
External data

Saving and retrieving data

This chapter considers two related issues: how to read and write entire Djinn models for the purposes of saving and retrieving them between invocations of the application program; and how to import and export geometric data across the Djinn API for use by other parts of the application (e.g. display).

Persistence of Djinn objects

For reasons already explained in Chapter B, there are definite limitations to the save-and-retrieve cycle provided by the two Djinn functions defined below; which are only intended to provide persistence of Djinn objects between different sessions of the same application and Djinn implementation. Thus, for example, any migration of Djinn objects between different Djinn implementations will have to be achieved by logging calls to Djinn API functions in one application, and subsequently replaying them in the other.

Application data

It is necessary for application-specific data and related Djinn data to be stored together in non-volatile store and for the composite data objects to be retrieved subsequently. Thus the Djinn specification allows application data and Djinn objects to be contained in the same file.

However, it is the application's responsibility to write or read its data. The Djinn read and write functions therefore operate on opened files, leaving the application to control opening and closing of files and the organization of application data and objects within the file. Djinn does,

however, provide some assistance to the application; mechanisms are provided for identifying object cells, for associating user-defined attributes with cells, and for differentiating application data from different sources. These facilities are explained in Chapter 5.

Save-and-retrieve functions

Two very simple functions are provided, these are:

Write object	Given a D̈jinn object, and a text file open for writing, write the object to the file.
Read object	Given a text file open for reading, read and return the next D̈jinn object on the file.

Text files are used for portability, but the text may not be readable by people. It should be noted that the requirement for no loss of numerical precision across a save-and-retrieve cycle is only an 'ideal post-condition' in the function specifications in Part II.

It was stated in earlier chapters that *all* results of a D̈jinn function are returned through that function's parameters: they do not remain as part of D̈jinn state. Thus D̈jinn functions do not produce any graphical output (but they may return data suitable for graphics—see later in this chapter), neither do they normally write to any output stream. By their very nature, the functions described in this section must be exceptions to this rule; but they are the only ones.

Geometry import and export

Import and export requirements

Applications such as constraint-based design, engineering analysis, simulation and computer-aided manufacture all require geometric computations. Most of the support necessary for such applications is supplied by the D̈jinn object-creation and geometric query functions, but it is inevitable that some applications will need to create geometry and to compute properties not directly available at the D̈jinn interface.

For example, applications may want to use D̈jinn geometry to generate a finite-element mesh, a numerical control (NC) program, an engineering drawing, or some other graphical display. Manipulation of geometry by an application demands that D̈jinn should be able to export geometry with a visible representation. For example, the surface normals and interference checks required for NC cutter-path generation can be obtained by means of

Djinn queries, but explicit two-dimensional curvilinear polygons may also be required for cutter paths. The cellular operations provided by Djinn are useful for mesh generation, but cell geometry must also be exported. Ideally, this should be in some simple format that provides, at least, an approximation of known accuracy to the real geometry.

Similarly, it may be desirable to create geometry using well-established two-dimensional drafting or curve and surface design systems, and then to apply the geometric modelling capability of Djinn to that data. Curve and surface geometry can be classified as:

- Simple (e.g. natural conics, natural quadrics and cyclides).
- Complicated (i.e. free-form curves and surfaces).

Simple geometry is well-defined independently of its representation. In contrast, free-form curves and surfaces are usually defined *by* their representations. Even though complicated geometry is usually created from well-defined parameters such as points and normals, together with incidence and continuity conditions, each representation gives a different result.

Curve and surface designers often use a free-form representation that suits their application, and a mature software system to create, modify and interrogate their geometry. Djinn cannot compete directly with established interactive systems; however, such systems do not provide all the functions usually associated with solid modelling. Djinn is not intended to replace surface design systems, but to interface with them. Implementors should be free to select appropriate curve and surface software, even perhaps using proprietary methods which may not be generally divulged, and then transfer the results to a Djinn application. So the Djinn API must support the import of geometry created elsewhere.

Curve, surface and half-space paradigms

There are three important curve paradigms:

A *parametric curve* is represented by the image of an interval (usually $[0 \ldots 1]$) under a partial map[1] $\mathcal{R} \mapsto \mathcal{R}^3$.

An *implicit curve* is represented a set of simultaneous equations with a single degree of freedom. Such equations can represent two intersecting surfaces, a surface and viewpoint defining a silhouette curve, two surfaces with common tangent curves, or two surfaces and a rolling

[1]Such curves include intersection curves represented by a sequence of points defining a piecewise straight line, any point of which can be dropped accurately on to the true intersection.

element such as a plane or a sphere defining loci of rolling contact. Implicit curves are a superset of parametric curves.

A *Cayley curve* [Cayley 1860, Sabin 1976] is represented by a mapping $\mathcal{R}^3 \times \mathcal{R}^3 \to \mathcal{R}$. In principle, Cayley curves are a more 'explicit' form of implicit curves[2].

There are three main surface paradigms [Sabin 1994]:

A *parametric surface* is represented by the image of a two-dimensional parameter plane under a partial map $\mathcal{R}^2 \hookrightarrow \mathcal{R}^3$ (e.g. NURBS, trigonometrically parameterized cylinders, and surfaces offset from parametric definitions). Such surfaces are typically bounded by restricting the domain of the map to a rectangle or triangle of the parameter plane (usually $[0 \ldots 1] \times [0 \ldots 1]$), but there is no reason why the entire parameter plane should not be used to define unbounded parametric surfaces.

An *implicit surface* is represented by a set of simultaneous equations with two degrees of freedom, usually a single equation (i.e. the zero-set of a mapping $\mathcal{R}^3 \to \mathcal{R}$). The typical polynomial mappings define algebraic surfaces. Because the zero-set is the boundary of the positive set, it cannot itself have a frontier; additional machinery must be used to obtain surfaces with a frontier.

A *recursive subdivision surface* is defined as the limit set of a polyhedron under a refinement operator [Catmull 1978, Doo 1978, Stollnitz 1996].

There are two main half-space paradigms:

An *implicit half-space* is represented by that part of the domain of a mapping $\mathcal{R}^3 \to \mathcal{R}$ that gives a positive (or negative) image. It is bounded by an implicit surface.

A *parametric half-space* is represented by the image of a three-dimensional parameter space under a partial map $\mathcal{R}^3 \hookrightarrow \mathcal{R}^3$. Such half-spaces are typically bounded by restricting the domain of the map to a rectilinear region of the parameter space (usually $[0 \ldots 1] \times [0 \ldots 1] \times [0 \ldots 1]$).

There are well-understood relationships between the different paradigms, and many curves, surfaces and half-spaces can be represented equally well by any of them. Some conversion routes exist and others do

[2]They have not been found to be particularly useful in practical systems. For example, a construction for a piecewise Cayley curve is not obvious.

not, and some conversion routes are restricted to a particular paradigm subset. For example, all implicit cubic cylinders can be parameterized if a square root is permitted in the parametric mapping, but not if the mapping is limited to rational polynomials. Import and export of all possible curve and surface paradigms is therefore impractical and runs counter to the basic D̈jinn philosophy of a representation-independent API.

Imported geometry

Surfaces can be imported into D̈jinn as infinitesimally thin sheet objects. If necessary these objects can subsequently be swept (see Chapter 6) to create sheets with a thickness, or filled (see Chapter 1) for use as boundaries of solid objects. Similarly, curves are imported as infinitesimally thin wire objects.

Simple curves and surfaces are treated at the D̈jinn interface as point-sets of infinite cardinality with well-defined geometry but unknown representation (e.g. a cone is specified in standard form by one shape parameter—see Chapter 1). The D̈jinn philosophy of a representation-independent interface suggests that the definitions of free-form curves and surfaces should be based on geometry. This remains an incomplete research topic, although some progress has been made with a point-based system following the suggestion of Ball [Ball 1994]. Therefore, free-form curve and surface geometry can be well-defined at the D̈jinn interface only if its finite representation (e.g. equation coefficients) is visible.

Explicit imported geometry

There are many candidate representations for free-form geometry, but D̈jinn support for a large number of representations is impractical. The most popular free-form curves and surfaces are non-uniform rational B-splines (NURBS), but there are other valuable representations: for example, some current research provides good reason for the use of lower-degree algebraic surfaces [Middleditch 1994, Bajaj 1993]. Due to their popularity, D̈jinn provides functions for the explicit creation of NURBS (see Chapter 1); other free-form curves and surfaces may be imported only by approximation.

Curves, surfaces and solids may be approximated by linear polytopes or geometry of higher degree. For example, two-dimensional geometry could be imported as:

- Straight-line segments.
- Circular arcs.

- Conic arcs.
- Splines.

There is a balance to be struck between the number of imported entities required for a given accuracy and the complexity of those entities.

Free-form geometry can be imported only by approximation. In contrast, most geometry created by two-dimensional drafting software can be specified precisely by a few geometric parameters, and that precision should not be lost. Circular arcs are ubiquitous in engineering drawings, but arcs of degree greater than two usually appear only as free-form curves. Since conics are only marginally more complicated than circular arcs and quadratic splines can be represented by isolated conic arcs, D̈jinn allows the import of two-dimensional conic arcs in addition to the direct creation of NURBS curves.

D̈jinn also imports faceted approximations to both two- and three-dimensional objects:

- A facet of a point is itself.
- A faceted edge is a sequence of straight-line segments which approximate that edge.
- A faceted face is a set of planar triangles which approximate that face.
- A faceted solid is a set of plane-faced tetrahedra which approximate that solid.

D̈jinn imports point, line, surface and solid triangulations for consistency with the exported approximations to be discussed below.

Exported geometry

Applications can acquire D̈jinn object geometry piecemeal by navigating the structure and making detailed geometric queries for properties such as curve direction. This is time-consuming, but high-level queries for geometry such as surface-intersection curves amount to the explicit export of representations and defeat the purpose of the D̈jinn interface: since exported geometry has a visible representation, it must be either simple (e.g. a point), or a simple approximation to a more complicated entity. Such approximations need to be generated only when required and may or may not be the approximations used internally as the primary representation.

Although derived geometry such as the convex hull or the medial surface may be considered to be an approximation to D̈jinn geometry, these constructions are not primarily intended for export: they are nominally exact properties of a D̈jinn object, and may be interrogated by D̈jinn functions.

However, like other geometry, they may be approximated and exported if required.

Linear geometry alone is inadequate for many applications; additional derivative data often avoids the need for higher-order approximation (e.g. surface normal vectors are sufficient for smooth Phong or Gouraud shading of faceted surfaces). Derivative data also helps the application to reconstruct smooth curves and surfaces (e.g. Bézier quadratic triangles). Therefore, D̈jinn exports faceted approximations to both two- and three-dimensional D̈jinn objects, and derivatives may be obtained using D̈jinn queries or by approximation from point data. Unfortunately, since accurate circular arcs are required for both NC and engineering drawing, it is not adequate to restrict exported two-dimensional geometry to linear approximations; efficiency is improved by the export of two-dimensional conic arcs.

It is easy for an application to reduce convex regions to triangles, but it is not trivial to partition non-convex regions. It is also difficult for an application to use triangles to construct polygons suitable for its own purposes; for example, finite-element mesh generation may require a quadrilateral surface approximation. Fortunately, the object-partitioning function of D̈jinn enables the application to create object subdivisions with a topology that simplifies such reconstructions. Therefore, since they the simplest representations that are generally useful, D̈jinn supports the import and export of curve, surface and solid triangulations in preference to more general polygons and polyhedra.

Some computations external to D̈jinn involve multiple distinct exported geometric entities (e.g. the superposition of edges and faces in a single display). Therefore, D̈jinn exports linear approximations in which all exported cells are consistent. Thus an exported faceting of a face contains the same line segments as the exported facets of its bounding edges and the exported faceting of a solid contains the same triangular faces as the exported facets of its bounding faces.

Approximate geometry representations

Data sharing

The D̈jinn functions to import and export facetings utilize a shared array of vertex positions. Specification of a D̈jinn object or cell is accomplished by collections of facets, defined by indices into this shared array. (The approach is similar to that used to specify facetings for graphical display in OpenGL [Woo 1997].) Using indices into a single vertex array leads to a substantial reduction in the volume of data, compared with supplying the

vertex coordinates with every facet; and it simplifies the reconstruction of facet adjacencies if these are required. The use of indices also facilitates the import and export of the cellular structure of the approximated model, since all cell facetings share the same vertices.

Adjacency

Facets can be represented independently, as a strip of adjacent facets [Woo 1997] or as a mesh with links between all adjacent facets [George 1998, Weatherill, 1992].

Independent facets are the simplest approach, but facets sharing a vertex or an edge are more difficult for an application to find and they require more data. Using a full mesh, there is a path from each facet to the adjoining facets. Unfortunately, a full mesh requires even more data and, for a surface, amounts to a boundary representation of triangular-faced polyhedra; complicated traversal algorithms are then required in the application. Since adjacent triangles in a strip of triangular face facets share two vertices, only one vertex is required on average for each triangle in a long strip approximating a smooth surface. Therefore, a face-facet strip can be represented by a sequence of vertices, each consecutive triple of which defines a facet.

Strips of facets provide a compromise between independent facets and a full mesh structure and cope easily with non-manifold and cellular objects. Independent facets can be found from each successive sub-sequence of indices into the array of facet vertices, and two of the facet's neighbours are implicitly available from the strip sequence. Therefore, although the selection of an efficient set of strips of facets is non-trivial, D$\ddot{\text{j}}\ddot{\text{i}}$nn approximates a cell of a D$\ddot{\text{j}}\ddot{\text{i}}$nn object by sets of strips of facets, each strip represented by a sequence of points. This does not prevent a D$\ddot{\text{j}}\ddot{\text{i}}$nn implementation from importing or exporting strips that each contain a single facet.

In an edge strip, each pair of facets in the sequence represents an edge segment. In a face strip, each triple of points in the strip represents a triangular face. In Figure 27 the surface of a sphere truncated by planes at its poles is approximated by four strips of surface facets: one for each truncating plane and two for the remaining spherical surface. One of the latter strips is represented by the sequence of vertices $1, 2, \ldots 12$, representing facets $(1, 2, 3)$, $(2, 3, 4)$, $(3, 4, 5)$ etc. Vertices may be duplicated in the same strip; see, for instance, the centre of the truncating face in Figure 27.

For facetings of manifold dimension 2 or 3, it may not be possible to fill the cell with facets representable by a sequence in which every subsequence of $d + 1$ consecutive vertices is a facet. In these circumstances

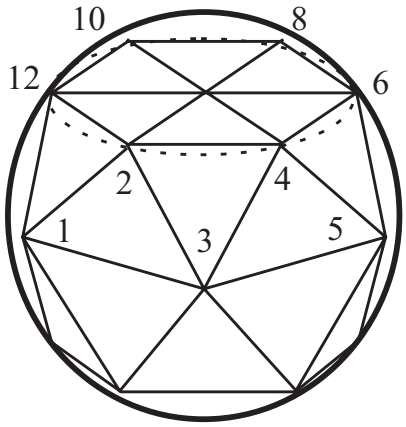


Figure 27: A faceted sphere.

vertices will be repeated within the element of *Facs*, using the convention that a sequence of $d + 1$ vertices containing a repetition is not a facet. For example, a triangulation containing the sequence $ABCDDEFG$ will represent the facets ABC, BCD, DEF and EFG. Similarly, the sequence $ABCDDEEFG$ will represent the facets ABC, BCD and EFG. where there are such breaks in the strip, the convention that triangular and tetrahedral facets alternate in orientation. The same convention is used to represent the faceting of a cell with more than one connected component.

In a solid strip, each sub-sequence of four points represents a tetrahedral facet, with adjacent tetrahedra sharing a triangular face.

The D̈jinn faceting defined for import and export is therefore a sequence of vertices. Given the vertices and the manifold dimension of the cell to be represented, the sequence can be interpreted. Independent facets or connected meshes of simplices can be reconstructed from the sequence in a straightforward manner.

Smooth geometry reconstruction

Face reconstruction can be improved by using surface normals of the true face at the approximated points; edge reconstruction can be improved by using edge tangents at the approximated points. Since normals or tangents can be retrieved if the cell label is known (see Chapter 8), each faceting exported by D̈jinn is associated with the label of the cell that it approximates. The import of free-form curves or surfaces with true discontinuities can be accomplished by importing vertex or edge facetings to identify these discontinuities, followed by departitioning operators to remove the explicit boundary thus created.

Orientation

Facets are coherently orientated by their vertex sequence. Although alternate triangular facets in face strips have opposite orientations, the orientation of all facets in a strip approximating a cell of any dimension can be taken from the vertex sequence of the first facet in that strip. For geometry import, all strips in a given faceting must have a consistent orientation. An attempt to import a cell which is not orientable will fail. Thus, importing a collection of surface facets defining a Möbius strip requires an edge faceting to break the face loop.

Two-dimensional curved geometry

By analogy with edge facets, D̈jinn can also approximate two-dimensional edge cells for export by sequences of connected conic arcs. Each directed arc is represented by its start point and shape parameters: precise details of this representation are documented in Part II of this book, but it amounts to using a start point, the intersection point of the tangent vectors and a bulge factor.

Approximation accuracy

Vertex facets

The accuracy specified by the application is greater than the distance between the single point defining a vertex facet and the vertex cell.

Edge facets

Approximation accuracy could be the maximum distance between a point on a facet edge and a point on the true edge which it approximates, or the maximum inter-facet angle. (At discontinuities of tangent direction, the latter measure becomes the maximum angular deviation from the true angle.) Facet accuracy can also be controlled indirectly by varying the number of facets generated. This is useful to guarantee computation speed, but indirect control of speed is provided by the direct accuracy measures. In order to handle variations of curvature on a true edge in a sensible manner, D̈jinn uses the maximum point deviation as the measure of accuracy.

There is no need for D̈jinn to ensure that each tangent discontinuity of a true edge corresponds to an end-point in a facet edge (i.e. that points on such discontinuities are within the prescribed accuracy of a facet edge end-point). If necessary, an application can improve the facet accuracy to

make them arbitrarily close. Therefore, the specified accuracy is greater than the distance between:

- Each facet point and a point on the approximated edge cell and vice versa.
- Each point cell bounding an edge cell and a facet vertex.

Surface facets

Since surface facets approximate face cells, their bounding edges and vertices are approximated to the same accuracy by facet edges and vertices respectively. Consistency between the edge facets and face facets of any given object is thus assured because face cell boundaries are edge and point cells.

Closed face cells and solid cells with a single boundary cell have no edges. This implies a need for the export of approximations to curves with discontinuities in surface normal (e.g. to display a wire-frame). However, D̈jinn does not ensure that points on such discontinuities are within the prescribed accuracy of a D̈jinn facet edge, because an application can instantiate discontinuities as edge cells or improve the facet accuracy to make them arbitrarily close. The latter is no more awkward than the display of silhouette curves as wire-frames using faceted approximations.

For consistency with edge facets, D̈jinn uses the maximum point deviation as the measure of accuracy. Thus, the specified accuracy is greater than the distance between:

- Each surface-facet point and a point on the approximated face cell and vice versa.
- Each bounding edge cell of the approximated face cell and a point on an edge of a surface facet.
- Each bounding point cell of the approximated face cell and a vertex of a surface facet.

Solid facets

For consistency between surface and solid facets of the same object, D̈jinn applies the maximum point deviation as the measure of accuracy to the tetrahedron faces that approximate the face cells bounding each solid cell. Thus, the specified accuracy is greater than the distance between:

- Each solid-facet point and a point in the approximated solid cell and vice versa.
- Each bounding face cell of the approximated solid cell and a point on a face of a solid facet.

- Each bounding edge cell of the approximated solid cell and a point on an edge of a solid facet.
- Each bounding point cell of the approximated solid cell and a vertex of a solid facet.

Two-dimensional curved geometry

For consistency, D̈jinn applies the maximum point deviation as the measure of accuracy of the exported conics that approximate edge cells. Quadratic edge cells are exported precisely and imported conics become edge cells without geometric change.

Export functions

Facets

When a D̈jinn object and some of its cells are exported in the course of a single call to the export function, the faces of the solid-facet approximation that is created are a superset of the solid's boundary. Similarly, the edges of a face-facet approximation are a superset of the edge facets of the face's boundary, and the vertices of an edge-facet approximation are a superset of the vertices returned as an approximation to the edge vertices.

Selected cells from two- or three-dimensional D̈jinn objects can be exported with the single function:

Export facets — Given an object, a sequence of cells and an accuracy, return a sequence of facetings that approximate the selected cells.

Each faceting in the sequence returned corresponds to a cell at the appropriate location in the input cell sequence. This defines the relationship between the cell and its faceting.

Exported edge facetings are an approximation of an object's edge cells. They do not include the edges of facets of higher dimensions because the latter may just as easily be obtained directly from such facets. Each faceting is a series of facet strips. Each facet strip is a sequence of vertices, each pair of which represents an oriented line-segment approximating part of the edge cell.

Exported face facets are an approximation of an object's face cells that does not include the surfaces of facets of higher dimension, because the latter may just as easily be obtained directly from such facets.

Sometimes an application requires a faceted approximation, but does not need the facets associated with any internal partitioning surfaces. For

example, the elimination of hidden lines for display requires surface facets, but not those in the interior of the solids comprising the scene. Thus, a complete faceting of all surfaces often causes many redundant facets to be produced and subsequently processed by the application. When only the exterior surface is required, the faceted approximation may be limited to the appropriate face cells.

Each triangular facet in a facet strip is represented by a triple of vertices. The first vertex triple in a face strip gives the orientation of all cells approximated by that strip. Edge and vertex boundary cells are approximated by facet edges and vertices respectively.

Solid facets are an approximation of solid cells. Each tetrahedron in a faceting is represented by a sequence of four vertices. Face, edge and vertex boundary cells are approximated by facet faces, edges and vertices respectively.

Two-dimensional curved geometry

Exported conics are an approximation of an object's edge cells:

Export 2D conics — Given an object, a sequence of cells and an accuracy, return a sequence of directed two-dimensional piecewise conic approximations to the selected edge cells, or indicate that the given object is not a two-dimensional object in the xy-plane.

Each piecewise conic can therefore be associated with the label of the cell from which it derives.

An arbitrary plane section can be exported by transforming it to the xy-plane before export. This can be done by a single rotation about a line through the origin that aligns the plane section normal with the positively directed z-axis (see Chapter 2) and the subsequent translation of any point of the section to the xy-plane.

Import functions

Facets

The D̈jinn facet import function is the inverse of the export function. Some or all of the cells of the D̈jinn object can be imported explicitly as facetings of the appropriate dimension. D̈jinn creates the smallest number of additional cells necessary to bound or separate the imported cells, such that each cell is bounded by the union of the imported cells and these additional cells. Thus, if a single solid cell is imported as strips of

tetrahedra, the returned D̈jinn object has two cells: the imported solid cell and the 'skin' formed by the surface facets of the tetrahedra. If part of the surface had been explicitly imported as a face faceting, then the returned object would have two face cells: the explicitly imported face and the remaining surface skin. The orientation of each of the extra cells is arbitrary, although their orientation relative to cells that they bound, or by which they are bounded, can be determined.

A D̈jinn object and cells selected from it can be imported with the single function:

Import facets — Given a sequence of facetings that approximates the cells of the object, return the object and a corresponding sequence of cells.

The faceting includes a specification of the manifold dimension of the cell to be imported.

The correspondence between the sequence of input facetings and the returned cell label allows applications to identify the imported cells.

Vertex cells are imported by an index to a single facet vertex.

Curves are imported with a similar representation to exported edge facets. An oriented edge cell is created for each directed line-segment faceting, and point cells are created to bound or separate those edge cells if they are not explicitly imported.

Three-dimensional surfaces and two-dimensional space partitions are imported with a similar representation to exported face facets. An orientated face cell is created for each input faceting, and the smallest number of additional edge cells needed to bound or separate the face cells are created, if these are not explicitly imported.

Solids are imported with a similar representation to exported solid facets. A solid cell is created for each input solid faceting and the smallest number of additional cells are created to bound or separate the solid cells, such that each solid cell is bounded by a union of such cells. The orientation of cell faces is deduced from the order of the first three vertices in each strip that represents a face cell.

Two-dimensional curved geometry

D̈jinn provides facilities to create D̈jinn objects from imported two-dimensional curves with a similar representation to those exported:

Import 2D conics — Given a piecewise sequence of directed two-dimensional conic arcs, create a two-dimensional object in the xy-plane.

Each conic is represented by its start point and shape parameters. An orientated edge cell is created for each directed piecewise conic and point cells are created to bound or separate the edge cells.

Export followed by import

The import of a previously exported triangulation will usually create a different object because both import and export functions compute a member of an equivalence class. Similarly, the export of a previously imported triangulation will usually create a different triangulation.

Implicit imported geometry

An additional method for accessing existing geometry which was considered but ultimately rejected for D̈jinn was the import of external geometry implicitly by reference. Explicit importation involves a choice between many candidate representations. Implicit importation prevents access to the representation, and D̈jinn implementations would then need to exploit external functions of the appropriate curve or surface package in order to manipulate referenced geometry. Such functions could also be exploited by a D̈jinn implementation for free-form geometry created by sweeping, tweaking and blending (see Chapters 3 and 6). The purpose of this section is to record some of the difficulties that were encountered in attempting to define a generic 'bottom interface' to external geometry.

References to external geometry

Some existing solid modelling systems use external software for archetypal queries such as 'find the point on a (parametric) surface at the given parameter value' or 'is the given point inside, outside, or on the boundary of, an (implicit) half-space'. Functions of this sort expose the paradigm and are thus limited to accessing external geometry with a single paradigm. The D̈jinn philosophy leads to a search for geometric modelling kernel that uses only paradigm-independent geometric queries of external functions. However, this does not prevent an application from using paradigm-dependent functions for curve and surface creation and manipulation.

Many algorithms for interrogating free-form surfaces are iterative, whether they (normally) provide a single result, such as the closest point on a surface to a given point, or a sequence of results, such as points on the intersection curve between two surfaces. Where an iterative algorithm is performed by external software, there are several implementation choices:

- All iterations are performed by the kernel, with calls to the external software at each step.
- The external software performs a single iteration, but any sequence must be performed by successive calls.
- The entire sequence is performed by the external software as a single result.

Performing iteration within the kernel is the only approach able to support operations that involve both D̈jinn internal and external geometry because the external computations that it requires depend only on geometric parameters. Unfortunately, it is far less efficient than the use of paradigm-dependent parameters because coherence between iterations cannot be exploited.

Efficiency of the second approach requires external software to retain paradigm-dependent coherence data (e.g. parameter position or implicit surface function and derivative values) as state[3], and function arguments must indicate when the coherence is relevant.

Results from the third approach would be in D̈jinn's hidden representation, and hence require further external functions for their interrogation.

Since the D̈jinn interface provides access to geometric properties such as curvature (see Chapter 8), external functions are required to compute such properties for imported geometry with hidden representation. All D̈jinn functions can be implemented for imported geometry if external functions are provided to compute the properties from which a curve, surface or half-space can be reconstructed:

- Two-dimensional curves: tangent direction.
- Three-dimensional curves: tangent, normal, binormal, curvature and torsion.
- Three-dimensional surfaces: principal curvatures and directions, and surface normal.
- Half-spaces: point-membership.

Unfortunately, exclusive use of such properties is prohibitively inefficient for D̈jinn functions such as Boolean operations and further support is required for the following computations:

Curves: sequence of sample points, closest point on the curve to a given point, silhouette points, reflection points, bounding box and curve integration.

[3]Or equivalently, to pass it out as a result to be re-introduced at the next call, perhaps as part of the hidden representation of the sequence being traversed.

Surfaces: sample points (i.e. faceting), umbilic points, closest point on the surface to a given point, silhouette curves, reflection lines, geodesic curves, lines of curvature, intersection points between a curve and a surface (including ray intersections), intersection curve between two surfaces (including plane sections), bounding box and surface integration.

Half-spaces: ray-classification (giving inside and outside segments) and all the surface operations on the bounding surface.

Surface intersections: it is possible for geometric intersections to consist of point-sets of different dimension (e.g. the intersection of two surfaces may consist of isolated points, curves and pieces of surface). Since external intersection functions return results by reference to a hidden representation, external interrogation functions are required for pieces of intersection of different dimension and the intersection functions must provide the distinct pieces.

Singularities: there are two types of singular point on a curve or surface:

- Singularities caused by the implementation paradigm breaking down locally (e.g. the pole of a sphere parameterized by latitude and longitude).
- Essential geometric singularities (e.g. the apex of a cone).

The former create computation difficulties, but are irrelevant for specification of the external geometry interface; external software must be able to compute correct results. Geometric singularities of curves include isolated points, cusps and self-intersections. Singularities for surfaces are even more complicated. At geometric singularities, the required result may not exist (e.g. the surface normal at a cusp) or there may be multiple or an infinite number of correct results (e.g. all points on a sphere are equally near to its centre). External functions must report such situations and provide a complete and consistent result. For example, if the result is an infinite number of vectors in the same plane, then the plane must be returned. A particular specimen solution vector is inappropriate because it may not be repeatable or compatible with the solution of neighbouring problems.

When external computations produce results by reference to a hidden representation, singular points are merely unevaluated properties of that geometry. External interrogation software must be robust in the presence of such singularities and report their existence when appropriate.

Transformation: if surface transformation is contained within a kernel implementation, the intersection of two transformed external surfaces

must be performed by the kernel using external geometric queries. Since the transformation type is hidden, external transformation requires the conversion of D̈jinn transformations into a sequence of visible representations such as the general linear and quadratic forms allowed by D̈jinn (see Chapter 2). Such transformations must be supported by the external software.

Implications of imported geometry for Djinn

The D̈jinn interface does not restrict the paradigms that can be used by an implementation. Therefore, if D̈jinn implementations are to use the same external functions, intersections between internal and external geometry require the use of geometric queries alone. This is extremely inefficient. For efficiency, intersections between external geometry are computed by external functions that provide results of hidden type. Intersections between internal geometry can be done completely by the D̈jinn implementation.

External intersection functions must indicate indeterminate results and (possibly an infinite number of) multiple results. They must also distinguish all pieces of different dimension. External geometric query functions are required for all hidden types of the latter. External queries must be robust in the presence of geometric singularities and report their presence. The alternative is for external software to subdivide geometry to isolate singularities (i.e. to create a stratification [Whitney 1965]). The use of external intersection functions demands provision for the transformation of external geometry by transformations equivalent to those of D̈jinn. The more esoteric of these transformations are unlikely to be supported by most free-form software packages.

Free-form geometry created by D̈jinn when sweeping, tweaking and blending (see Chapters 6 and 7) can be manipulated internally, entirely by the external free-form surface software, or by using geometric queries provided by that software. The use of geometric queries is prohibitively inefficient, but requiring external software to support all D̈jinn geometry severely restricts the external software that can be exploited.

Thus, D̈jinn requirements restrict the external curve and surface packages that might be used with D̈jinn and some operations remain highly inefficient. Some of the inefficiency could be eliminated by exposing the external paradigm (i.e. by D̈jinn import functions for each paradigm), but this would require the external provision of query functions for each paradigm.

Therefore, D̈jinn interfaces to external free-form curve and surface packages only by the import and export of explicit geometry.

Part II
A
Introduction

This part of the D̈jinn specification provides more detailed specifications for the functions described in Part I, but nevertheless it is a reference document that stands by itself. There is a one-to-one correspondence between the chapters in Parts I and II. The specifications of Part II form the basis for the language binding of Part III. The function specifications of this part of the D̈jinn specification reflect the design guidelines of Part I Chapter B, summarized as follows:

- A minimal number of data abstractions (see Part II, Chapter B) is exposed at the interface.
- Interface functions are restricted to the fundamental operations from which more complicated functions can be built, except where significant convenience or computation speed may be gained.
- Function parameters are checked for validity in preference to limiting their range by means of pre-conditions.
- There is no externally perceived state (i.e. no externally perceived data that persists within a D̈jinn implementation between function invocations).
- D̈jinn functions indicate exceptional results by one of the following:
  - A variable with a value corresponding to unsuccessful computation.
  - A special value of the normal output variable.
  - An error variable (of hidden type).
- Although all functions must exist, their primary operation need not be supported. Thus, any D̈jinn function may return an error result instead of any useful computation. Although this error may indicate one of several exceptional situations, they always can be distinguished by further D̈jinn interrogations. That may not be a simple approach,

but it is to be preferred to the inclusion of a specific function parameter to indicate that a function is not supported.

- Function return parameters are chosen to minimize the number of functions that cannot be supported by an implementation when a particular D̈jinn concept (e.g. cells) is absent from the implementation.

Function definitions

D̈jinn *functions* are defined in terms of their input and output parameters, together with pre-conditions and post-conditions on these parameters. Most of these conditions are presented informally, but formal predicate calculus expressions are used when doing so enhances the likelihood of correct interpretation.

- A *pre-condition* for a function must be satisfied by its input parameters. It is a logical function of those parameters which must evaluate to TRUE. The action of a function is undefined if its parameters do not satisfy every pre-condition.
- A *post-condition* for a function defines the computed results. A post-condition is a logical function of both input and output parameters. It is always satisfied by the parameter values and answer returned. When a parameter (P) is modified by a function, its symbol is decorated to distinguish the input (P) and output (P') values. A post-condition is often formulated to allow a range of results, or alternatives, e.g. the expected answer or an error indication.

The D̈jinn specification provides both *ideal* and *actual* post-conditions. Due to the finite precision of floating-point numbers, an ideal post-condition cannot always be satisfied. The implementation has an obligation to meet such a post-condition as far as possible. In contrast, actual post-conditions are always satisfied.

Values of variables with representations inaccessible at the D̈jinn interface can be found by functions that provide data of a visible type. Thus, approximations are perceived by the application only after such interrogation has taken place. The function definitions imply that hidden representations are exact and that all approximations occur on interrogation. This is reflected in the actual and ideal post-conditions.

The normal ideal and actual post-conditions apply only when the input parameters have normal values and the computation is supported by the implementation. D̈jinn function specifications also provide post-conditions for exceptional input parameter values (e.g. a point might unexpectedly be produced from the intersection of two coincident lines in three-dimensional space).

Even if an implementation does not support a particular operation, the associated D̈jinn function must nevertheless exist. D̈jinn function specifications provide post-conditions for such unsupported functions. For example, normal post-conditions may demand that an output flag is TRUE, but unsupported functions may set such a flag to FALSE. Alternatively, an unsupported function may return a special value of an output variable, such as an object with no cells.

D̈jinn implementations may weaken a function's pre-conditions or strengthen its post-conditions, or both, but not strengthen the pre-conditions or weaken the post-conditions. Thus, subject to perfect arithmetic, implementations must satisfy the post-conditions of this specification (i.e. $(I \wedge A) \vee E \vee U$ must hold where I, A, E and U are the ideal, actual, exceptional and unsupported post-conditions respectively). This shows that any function can at least satisfy the unsupported post-condition.

Partially supported functions could be defined by stronger pre-conditions, but the D̈jinn approach is different. Pre-conditions are always satisfied, but the 'unsupported function' result is returned. Thus, functions may be unsupported for certain particular input parameter values. This situation is reflected in a strengthened *unsupported post-condition*, which implies particular input parameter values.

Both pre- and post-conditions usually consist of several conditions to be satisfied simultaneously. For clarity, D̈jinn function specifications provide these conditions as a list rather than a logical conjunction.

Part II
B
Djinn object types

Visible exported data types

The following data types are exported in a form in which their representation is visible for use by application software.

DjBoolean **:** a standard type of the implementation language supporting $\wedge$, $\vee$ and $\neg$, acting on logical values.

DjString **:** a standard type of the implementation language supporting sequences of characters.

DjInteger **:** a standard type of the implementation language supporting $+$, $-$ and $\times$, acting on integers.

DjReal **:** a standard type of the implementation language supporting $+$, $-$, $\times$ and $/$, acting on real numbers.

Thus, an application can manipulate variables of such exported types without regard for the particular data type being used. For example, if rational integers were a standard type, they could be used by a Djinn implementation to represent and export real numbers, by equating *DjReal* to that type. If, on the other hand, rationals were not a standard type, they could still be used in the Djinn implementation, but results must be converted to a different *DjReal* type for export. All real parameters of Djinn functions have the same type, as do all integer parameters. The following types have visible representation but are unlikely to be handled by standard operators of the implementation language:

DjLogic **:** the set $\{DjTrue, DjUnknown, DjFalse\}$.

DjVector **:** an arbitrary sized sequence of *DjReal*.

DjMatrix **:** an arbitrary sized sequence of *DjVector*.

Finally, we need to be able to pass certain geometric items efficiently between D̈jinn and the application. We have chosen to provide two kinds of objects, commonly used in CAD systems and in graphics.

DjFaceting **:** a sequence of consecutive facets of edges, faces or solids, encoded by a sequence of vertices where $d+1$ successive vertices define each facet, d being the manifold dimension of the cell being imported or exported. Each *DjFaceting* represents one connected component of a cell.

DjConics **:** A sequence of conic arcs, together approximating a curve of spatial dimension 2. Each piecewise conic is encoded by a sequence of triples, each triple containing two points and a real shape parameter. Each conic in the sequence uses the first point of a triple as its start-point, the second point of the triple as its tangent intersection point, and the first point of the next triple as its end-point. Thus n conics are represented by $n+1$ triples, the second point and the real of the last triple being unused.

If the curve to be represented has more than one connected component, the convention followed is that a null arc joins the last point of the first component with the first point of the next. A null arc is recognizable by the tangent intersection point being coincident with the start-point. The real of a null arc is also unused.

The real is a shape parameter varying from near 0, when the conic is a hyperbola going very close to the tangent point, to near 1, when the conic is an ellipse remaining close to the chord between start- and end-points.

Exported constants

All D̈jinn implementations export definitions of the constants *DjMaxInteger*, *DjMaxReal*, *DjSmallReal* and *DjRealPrecision*. The constant *DjSmallReal* is the smallest positive real number distinct from zero that can be represented by the type *DjReal*; it is not $-$*DjMaxReal*. The constant *DjRealPrecision* is the smallest real number which, summed with unity, differs from unity, when both are represented by *DjReals*.

Hidden exported data types

The following data types are exported, but their representation is hidden from the application software. Manipulation of variables of these types

can only be performed by Djinn functions. In contrast to the visible exported types discussed above, hidden types are defined in terms of their abstraction. In addition to their primary range, some types include 'indeterminate' as a possible value.

Indeterminate variables represent an entity of the primary range that cannot be computed (e.g. 0/0 is an indeterminate real number). In contrast, a variable with a value that is undefined either has an unknown value or does not represent an entity in the primary range.

DjLabel: $\exists F \in Z \leftrightarrow DjLabel$.
(Cell label; *DjOut* (referred to as OUT in function definitions) represents *not a cell* and thus the region outside all cells.)

DjLabelSet: $\mathrm{P}(DjLabel)$.
(Set of cell identifiers.)

DjLabelSequence: $\sum_{k \in Z^+} Seq_k(DjLabel)$.
(Arbitrary length 1-ordered multi-set of cell identifiers[1].)

DjPoint: $\mathcal{E}^1 \cup \mathcal{E}^2 \cup \mathcal{E}^3 \cup Indeterminate$ (afterwards written $\mathcal{E}^n$).
(Euclidean point of arbitrary dimension.)

DjPlane: $\mathrm{P}(\mathcal{E}^1) \cup \mathrm{P}(\mathcal{E}^2) \cup \mathrm{P}(\mathcal{E}^3) \cup Indeterminate \bullet \forall H \in DjPlane \bullet \forall \langle x_1, \ldots x_n \rangle \in H \bullet \exists \langle k_0, \ldots k_n \rangle \in \mathcal{R} \bullet k_0 + k_1 x_1 + \ldots + k_n x_n < 0$.
(Linear Euclidean half-space.)

DjTransform: $\mathcal{R} \rightarrowtail ((\mathcal{E}^3 \rightarrow \mathcal{E}^3) \cup (\mathcal{E}^2 \rightarrow \mathcal{E}^2) \cup (\mathcal{E}^1 \rightarrow \mathcal{E}^1) \cup Indeterminate)$.
(Continuous map from $[0 \ldots 1]$ to a one-to-one continuous transform. Note that 0 maps to the identity transform, and 1 maps to the 'full extent' of the transform.)

DjPrecision: $\mathcal{R}^3 \cup Indeterminate$.
(For a variable P of type *DjPrecision*, $P.lb$ and $P.ub$, and $P.est$, are the lower and upper error bounds, and error estimate, respectively. *DjPrecision* is used to represent errors in computations of real numbers.)

Although a Djinn object may consist of a single point or plane, distinct data types also are used because many geometric operations that involve points and planes do not require their augmentation with cells, attributes, and the other items that constitute an object. Djinn functions are specified in terms of mappings from the hidden types. Thus, the Euclidean

[1] Seq_k is an ordered multi-set of its k arguments.

coordinates of a non-indeterminate D̈jinn point are given by the partial function:

$$Point \in DjPoint \rightarrowtail DjVector\,.$$

If a point is infinite, the Euclidean coordinates of its (not necessarily unit) direction vector are given by the partial function:

$$Vector \in DjPoint \rightarrowtail DjVector\,.$$

Similarly, the plane coordinates of a (non-indeterminate) D̈jinn plane **Pl** are given by the partial function:

$$Plane \in DjPlane \rightarrowtail DjVector \bullet \forall\ \mathbf{Pl} \in DjPlane \bullet \\ \forall \langle x_1, \ldots x_n \rangle \in \mathbf{Pl} \bullet \langle 1, x_1, \ldots x_n \rangle \cdot Plane(\mathbf{Pl}) < 0.$$

The formal definition of a D̈jinn object developed in Part 1, Chapter B consists of its cells and their auxiliary data. That definition in terms of functions which provide properties is thus realized by the abstraction:

$$Object\ \subset \mathrm{P}(Cell).$$

This is enhanced to include a count of modifications to object geometry, and the corresponding region of change; they are included to facilitate interrogations that enhance application efficiency. The spatial dimension of an object is a property of the point-sets of its cells, but the definitions of Part I are augmented by an explicit dimension field, to enhance the clarity of function specification. The previous abstraction thus becomes:

$$Object \subset \mathrm{P}(Cell) \times \mathcal{Z} \times \mathcal{Z} \times \mathrm{P}(\mathcal{E}_n).$$

When D̈jinn changes the point-set of an object, it increments the modification count and augments the update region with the region of change. A null object with no cells is represented by a cell map with an empty domain. The functionality of the cells of an object, as defined in Part I, is realized by a partial function from labels to those cells, their attributes and parentage:

$$Cell\ \subset DjLabel \rightarrowtail ((\mathcal{E}^n \rightarrowtail \mathcal{R}^n) \times (String \rightarrowtail \mathcal{Z}) \times \mathrm{P}(DjLabel^2)).$$

The first item on the right-hand side of this definition represents both its point-set, and the mapping to an orientation vector at each point.

The manifold dimension of a cell is a property of its point-set, but as with the spatial dimension of an object, an explicit dimension field is added to the formal definition of Part I to enhance the clarity of function specifications:

$$Cell\ \subset DjLabel \rightarrowtail ((\mathcal{E}^n \rightarrowtail \mathcal{R}^n) \times \mathcal{Z} \times (String \rightarrowtail \mathcal{Z}) \times \mathrm{P}(DjLabel^2)).$$

Thus, the cell identifier i of object O maps to $O(i) = \langle S, n, At, An\rangle$ (i.e. to a cell with a relatively open semi-analytic manifold point-set $\mathit{Domain}(S)$ of homogeneous dimension n and orientation $S(\mathbf{p})$ at each point $\mathbf{p}$, a set of attributes At, and a set An of all parent cell pairs from which it derives, if any). The attributes have a type and value (i.e. they are character string and integer pairs). Implementations may or may not reflect the manifold dimension field in their data structures, but explicit implementation could be used to resolve ambiguities that arise due to errors in computation. For example, the set-intersection of two touching solids is an empty or thin solid (see Part I, Chapter 5), but the geometry computed might indicate a sheet object, perhaps with random holes; an explicitly computed manifold dimension admits only the correct result. This can be exploited to improve subsequent computations and non-numeric interrogations.

In preference to the notation above, the clarity of pre- and post-conditions in the subsequent function definitions is enhanced by a notation reminiscent of Z [Spivey 1989] for data abstractions. Thus, the subsequent schema show field names and data invariants. For example, the modification count of object Obj is $\mathit{Obj}.\mathit{ModCnt}$ with invariant $\mathit{Obj}.\mathit{ModCnt} \geq 0$.

Summing up the previous definitions, we may now give the following description of what a Djinn object is:

$\mathit{DjObject}$ $\mathrm{P}(\mathit{Cell}) \times \mathcal{Z} \times \mathcal{Z} \times \mathrm{P}(\mathcal{E}^n)$.

Cell: $\mathit{DjLabel} \rightarrow \mathit{DjCell}$ (cell geometry and associated data).

Dim: $\mathit{DjInteger}$ (spatial dimension).

ModCnt: $\mathit{DjInteger}$ (modification count).

$\mathit{ModRegion}$: $P(\mathit{DjPoint})$ (region of change).

$\mathit{ModCnt} \geq 0$.
Spatial dimension of $\mathit{ModRegion} = \mathit{Dim}$.
Bounding points of a cell in $\mathit{Range}(\mathit{Cell})$ = union of cells in that range.

DjCell $(\mathcal{E}^n \mathrel{+\!\!\!\rightarrow} \mathcal{R}^n) \times \mathcal{Z} \times (\mathit{String} \mathrel{+\!\!\!\rightarrow} \mathcal{Z}) \times \mathrm{P}(\mathit{DjLabel}^2)$.

Orient: $\mathit{DjPoint} \mathrel{+\!\!\!\rightarrow} \mathit{DjVector}$ (mapping from point-set to orientation vector).

Dim: $\mathit{DjInteger}$ (manifold dimension of point-set).

Atr: $\mathit{DjString} \mathrel{+\!\!\!\rightarrow} \mathit{DjInteger}$ (string, integer attribute identifiers).

$\mathit{Parents}$: $\mathrm{P}(\mathit{DjLabel}^2)$ (parent cell pairs).

$\mathit{Domain}(\mathit{Orient})$ is a semi-analytic relatively open manifold

point-set.
Dim is the homogeneous dimension of *Domain*(*Orient*).
$0 \leq Dim \leq 3$.
Each element of *Range*(*Atr*) ≥ 0.

Part II
1
Primitive objects and copying

Creation and interrogation of simple geometry

Dj Create Point

INPUT

n	:	*DjInteger*	(dimension).
$\mathbf{P}$	:	*DjVector*	(Euclidean point coordinates).
B	:	*DjBoolean*	(FALSE for an ideal point).

OUTPUT

$\mathbf{Pt}$	:	*DjPoint*	(D̈j̈inn point).

DESCRIPTION

This function creates the point $\mathbf{Pt}$ of hidden D̈j̈inn type in a space of dimension n from its n Euclidean coordinates given in $\mathbf{P}$. When B is FALSE, the point $\mathbf{Pt}$ is at infinity in the direction of vector $\mathbf{P}$ (points at infinity in opposite directions are the same point).

PRE-CONDITIONS

- ☐ The point is one-, two- or three-dimensional: $1 \leq n \leq 3$.
- ☐ When B is FALSE, the vector $\mathbf{P}$ is not the zero vector.

ACTUAL POST-CONDITIONS

- ☐ When B is TRUE, the point $\mathbf{Pt}$ is finite: $Point(\mathbf{Pt}) = \mathbf{P}$.
- ☐ When B is FALSE, the point $\mathbf{Pt}$ is at infinity in direction $\mathbf{P}$: $\exists k \in \mathcal{R}^+ \bullet Vector(\mathbf{Pt}) = k \times \mathbf{P}$.

IDEAL POST-CONDITIONS

- ☐ TRUE (there are no ideal postconditions).

Exceptional post-conditions

□ FALSE (exceptional input parameter values do not exist).

Post-conditions for the unsupported function

□ **Pt** has the value *indeterminate* and is thus not in the domain of the function *Point* or the function *Vector*, i.e. *Point*(**Pt**) and *Vector*(**Pt**) are undefined.

Dj Get Point Coordinates

Input

Pt : *DjPoint* (D̈jinn point).

Output

n : *DjInteger* (dimension).
P : *DjVector* (Euclidean point coordinates).
B : *DjLogic* (FALSE if **Pt** is an ideal point).

Description

This function extracts the n Euclidean coordinates **P** from the point **Pt** of hidden D̈jinn type. The logical result B indicates whether the point is finite, infinite or indeterminate.

Pre-conditions

□ TRUE (there are no pre-conditions).

Actual post-conditions

□ n is the spatial dimension of point **Pt**.

Ideal post-conditions

□ $B = DjTrue$ when the point **Pt** is finite.

□ $B = DjFalse$ when the point **Pt** is on the line at infinity.

□ The dimension n and coordinates of **P** are defined only when B is not *DjUnknown*:

$$\begin{aligned} &(B = DjTrue) \Leftrightarrow \\ &\quad (Point(\mathbf{Pt}) = \mathbf{P}) \wedge (n = Dim(\mathbf{Pt})) \\ \wedge\ &(B = DjFalse) \Leftrightarrow \\ &\quad (Vector(\mathbf{Pt}) = \mathbf{P}) \wedge (n = Dim(\mathbf{Pt})). \end{aligned}$$

Exceptional post-conditions

□ If the point **Pt** is indeterminate, $B = DjUnknown$.

Post-conditions for the unsupported function

□ The logical variable B takes the value *DjUnknown*.

Dj Create Plane

Input

n : *DjInteger* (dimension).
H : *DjVector* (Euclidean plane coordinates).

Output

Pl : *DjPlane* (D̈jinn plane).

Description

This function creates the open linear half-space **Pl**, of hidden D̈jinn type, from its $n + 1$ hyper-plane coordinate vector **H**. The half-spaces that can be created are a three-dimensional half-space bounded by a plane, a two-dimensional half-plane bounded by a straight line and a semi-infinite interval of the real line. If all coordinates of **H** except the first are zero, then **Pl** is either the empty set or the entire n-dimensional space excluding points at infinity, depending on the sign of the first coordinate.

Pre-conditions

□ The coordinates **H** must be one-, two- or three-dimensional: $1 \leq n \leq 3$.

Actual post-conditions

□ The point-set of the plane **Pl** with coordinates **H** is given by the inner-product inequality: $\mathbf{Pl} = \{\langle x, y, \ldots\rangle \in \mathcal{R}^n | \langle 1, x, y, \ldots\rangle \cdot \mathbf{H} < 0\}$.

Ideal post-conditions

□ TRUE (there are no ideal conditions).

Exceptional post-conditions

□ If **H** is the zero vector, then **Pl** is indeterminate and thus not in the domain of the function (i.e. plane coordinates *Plane*(**Pl**) are undefined).

Post-conditions for the unsupported function

□ The plane **Pl** is indeterminate.

Dj Get Plane Coordinates

Input

Pl : *DjPlane* (D̈jinn plane).

OUTPUT

n	:	*DjInteger*	(dimension).
H	:	*DjVector*	(Euclidean plane coordinates).
B	:	*DjLogic*	(FALSE if ideal plane).

DESCRIPTION

This function extracts the $n + 1$ plane coordinates **H** from the n-dimensional half-space **Pl**, bounded by a plane (see the specification of *Dj Create Plane*). The logical result B indicates whether that plane is finite, at infinity, or indeterminate.

PRE-CONDITIONS

□ TRUE (there are no pre-conditions).

ACTUAL POST-CONDITIONS

□ The spatial dimension of half-space **Pl** is n.

IDEAL POST-CONDITIONS

□ $B = DjTrue$ when the plane is finite:
$(B = DjTrue) \Leftrightarrow (\emptyset^n \subset \mathbf{Pl} \subset \mathcal{R}^n)$.

□ $B = DjFalse$ when the plane is at infinity, i.e. **Pl** is the empty or universal point-set: $(B = DjFalse) \Leftrightarrow (\mathbf{Pl} = \emptyset^n) \vee (\mathbf{Pl} = \mathcal{R}^n)$.

□ The coordinates of **H** are defined only when B is not $DjUnknown$:
$(B \neq DjUnknown) \Leftrightarrow (Plane(\mathbf{Pl}) = H)$.

EXCEPTIONAL POST-CONDITIONS

□ When the plane **Pl** is indeterminate, $B = DjUnknown$.

POST-CONDITIONS FOR THE UNSUPPORTED FUNCTION

□ $B = DjUnknown$.

Object creation

Dj Create Empty Object

INPUT AND OUTPUT

Obj : *DjObject* (empty D̈jinn object).

DESCRIPTION

This function changes the well-defined D̈jinn object *Obj* into an object with no cells. Storage previously occupied by *Obj* is reclaimed.

PRE-CONDITIONS

□ TRUE (there are no pre-conditions).

Actual post-conditions

□ Object fields.

No cells:	$Obj'.Cell = \{\mathsf{OUT} \rightarrow \{\{p \in \mathcal{R}^3 \rightarrow \langle 0,0,0\rangle\}, 3, \emptyset, \emptyset\}\}$.
Spatial dimension:	$Obj'.Dim = 3$.
Modification count:	$Obj'.ModCnt = 0$.
Update region:	$Obj'.ModRegion = \emptyset$.

Ideal post-conditions

□ TRUE (there are no ideal conditions).

Exceptional post-conditions

□ FALSE (exceptional input parameter values do not exist).

Post-conditions for the unsupported function

□ FALSE (this function is always supported).

Simple primitive objects

Dj Create Point Object

Input

Pt : *DjPoint* (D̈jinn point).

Output

Obj : *DjObject* (D̈jinn point object).
L : *DjLabel* (label of point cell).

Description

This function creates the D̈jinn object *Obj* consisting of a single cell *L* with a point-set consisting of the single point **Pt**, possibly at infinity. The point **Pt** and thus the object *Obj* may exist in any spatial dimension but the manifold dimension of the cell is always zero.

Pre-conditions

□ TRUE (there are no pre-conditions).

Actual post-conditions

□ Object fields.

$Obj.ModCnt = 0$.
$Obj.ModRegion = \{Point(\mathbf{Pt})\}$.
$Obj.Dim = Dim(\mathbf{Pt})$.
$Domain(Obj.Cell) = \{\mathsf{OUT}, L\}$ (there is one cell).

□ Cell fields.

$Obj.Cell(L).Atr = \emptyset$ (there are no attributes).
$Obj.Cell(L).Parents = \emptyset$ (there are no parents).
$Obj.Cell(L).Orient =$
$\{\mathbf{Pt} \to \langle 0, \ldots \rangle\}$ (one point).
$Obj.Cell(L).Dim = 0$ (manifold dimension is 0).

□ Region outside the object.

$Obj.Cell(\mathsf{OUT}).Dim = Obj.Dim$.

$\cup_{i \in D} Domain(Obj.Cell(i).Orient) = \mathcal{R}^{Obj.Dim}$.
$D = Domain(Obj.Cell)$.

$Obj.Cell(\mathsf{OUT}).Atr = \emptyset$.

$Obj\,Cell(\mathsf{OUT}).Parents = \emptyset$.

$\forall \mathbf{p} \in Domain(\mathbf{Ut}) \bullet \mathbf{Ut}(\mathbf{p}) = \langle 0, \ldots \rangle$,
where $\mathbf{Ut} = Obj.Cell(\mathsf{OUT}).Orient$.

Ideal post-conditions

□ TRUE (there are no ideal conditions).

Exceptional post-conditions

□ If **Pt** is indeterminate, L is undefined and object Obj has no cells.

$Obj.Cell = \{\mathsf{OUT} \to \{\{p \in \mathcal{R}^n \to \langle 0, 0, 0 \rangle\}, n, \emptyset, \emptyset\}\}$,
where n = dimension of given point.
Spatial dimension: $Obj.Dim = n$.
Modification count
incremented: $Obj.ModCnt = Obj.ModCnt + 1$.
Update region: $Obj.ModRegion = Point\text{-}set(Obj)$.

Post-conditions for the unsupported function

□ L is undefined and object Obj has no cells: therefore this condition is the same as the exceptional post-condition.

Dj Create Line Object

Input

pt, **qt** : *DjPoint* (line-segment).

Output

Obj : *DjObject* (line-segment Djinn object).
L : *DjLabel* (label of line-segment cell).

DESCRIPTION

This function creates the D̈jinn object *Obj* with two cells: cell L with a point-set that is the open straight-line segment from point **pt** to point **qt**, and cell B consisting of points **pt** and **qt**. Point **pt** and **qt**, or both of them, may be at infinity and in a space of any dimension, but the manifold dimension of cell L is always one.

PRE-CONDITIONS

□ TRUE (there are no pre-conditions).

ACTUAL POST-CONDITIONS

□ Object fields.

$Obj.ModCnt = 0.$

$Obj.ModRegion = Point\text{-}set(Obj).$

$Obj.Dim = 1.$

$\exists B \in DjLabel.$

$Domain(Obj.Cell) = \{\mathsf{OUT}, L, B\}.$

□ Cell fields.

$Obj.Cell(L).Atr = Obj.Cell(B).Atr = \emptyset.$

$Obj.Cell(L).Parents = Obj.Cell(B).Parents = \emptyset.$

$Obj.Cell(L).Dim = 1.$

$Obj.Cell(B).Dim = 0.$

$Domain(Obj.Cell(L).Orient) =$
$\{s \in \mathcal{R}^n | \forall t \in (0 \ldots 1) \bullet s = t\mathbf{pt}' + (1-t)\mathbf{qt}'\}$
where $\mathbf{pt}'$ and $\mathbf{qt}'$ are **pt** and **qt**, augmented with 0-valued coordinates to bring their dimensionality up to n, the larger spatial dimension of **pt** and **qt**.

$Domain(Obj.Cell(B).Orient)$
$= \{\mathbf{pt}', \mathbf{qt}'\} \forall v \in Range(Obj.Cell(L).Orient) \bullet$
$v = Point(\mathbf{qt}') - Point(\mathbf{pt}').$
$\forall v \in Range(Obj.Cell(B).Orient) \bullet v = \langle 0, \ldots \rangle.$

□ Region outside the object.

$Obj.Cell(\mathsf{OUT}).Dim = Obj.Dim.$
$\cup_{i \in D} Domain(Obj.Cell(i).Orient) = \mathcal{R}^{Obj.Dim},$
where $D = Domain(Obj.Cell).$

$Obj.Cell(\mathsf{OUT}).Atr = \emptyset.$

$Obj.Cell(\mathsf{OUT}).Parents = \emptyset.$

$\forall \mathbf{p} \in Domain(\mathbf{Ut}) \bullet \mathbf{Ut}(\mathbf{p}) = \langle 0, \ldots \rangle,$
where orientation $\mathbf{Ut} = Obj.Cell(\mathsf{OUT}).Orient.$

Ideal post-conditions

□ TRUE (there are no ideal conditions).

Exceptional post-conditions

□ If **pt** or **qt** is indeterminate, or **pt** = **qt**, L and B are undefined and object Obj has no cells.

$Obj.Cell = \{\mathsf{OUT} \rightarrow \{\{p \in \mathcal{R}^n \rightarrow \langle 0,0,0\rangle\}, n, \emptyset, \emptyset\}\}$.
where n = dimensionality of given points.

Spatial dimension: $Obj.Dim = n$.

Modification count incremented:
$Obj.ModCnt = Obj.ModCnt + 1$.

Update region: $Obj.ModRegion = Point\text{-}set(Obj)$.

Post-conditions for the unsupported function

□ L and B are undefined and object Obj has no cells (same as exceptional post-condition).

Dj Create Plane Object

Input

Pl : *DjPlane* (D̈jinn planar half-space).

Output

Obj : *DjObject* (D̈jinn object).
L : *DjLabel* (label of plane half-space cell).

Description

This function creates the D̈jinn object Obj consisting of two cells; a cell L with a point-set that is the open planar half-space **Pl**, and a cell B consisting of its bounding plane. The spatial dimension of object Obj is that of half-space **Pl**, and this dimension is also the manifold dimension of cell L. The half-space **Pl** may be empty or it may consist of the entire space, less points at infinity.

Pre-conditions

□ TRUE (there are no pre-conditions).

Actual post-conditions

□ Object fields:

$Obj.ModCnt = 0$.

$Obj.ModRegion = Closure(\mathbf{Pl})$.

$Obj.Dim = Dim(\mathbf{Pl})$.

$\exists B \in DjLabel \bullet Domain(Obj.Cell) = \{\mathsf{OUT}, L, B\}$.

□ Cell fields.

$Obj.Cell(L).Atr = Obj.Cell(B).Atr = \emptyset.$

$Obj.Cell(L).Parents = Obj.Cell(B).Parents = \emptyset.$

$Obj.Cell(L).Dim = Dim(\mathbf{Pl}).$

$Obj.Cell(B).Dim = Dim(\mathbf{Pl}) - 1.$

$Domain(Obj.Cell(L).Orient) = \mathbf{Pl}\ (Point\text{-}set(L)).$
$Domain(Obj.Cell(B).Orient) = \partial(\mathbf{Pl}).$
$\forall v \in Range(Obj.Cell(L).Orient) \bullet v = \langle 0, \ldots \rangle$
Surface normal of boundary cell B is directed out of the solid cell L:

$\forall v \in Range(Obj.Cell(B) \bullet Orient)$
$\bullet \exists r \in \mathcal{R} \bullet \exists k \in Z \bullet k\langle r, v\rangle = Plane(\mathbf{Pl}).$

□ Region outside the object.

$Obj.Cell(\mathsf{OUT}).Dim = Obj.Dim.$

$\cup_{i \in D} Domain(Obj_1.Cell(i).Orient) = \mathcal{R}^{Obj.Dim},$
where $D = Domain(Obj.Cell),$
$Obj.Cell(\mathsf{OUT}).Atr = \emptyset.$
$Obj.Cell(\mathsf{OUT}).Parents = \emptyset.$

$\forall \mathbf{p} \in Domain(\mathbf{Ut}) \bullet \mathbf{Ut}(\mathbf{p}) = \langle 0, \ldots \rangle,$
where $\mathbf{Ut} = Obj.Cell(\mathsf{OUT}).Orient.$

Ideal post-conditions

□ TRUE (there are no ideal conditions).

Exceptional post-conditions

□ If **Pl** is indeterminate or the empty set, L and B are undefined and object Obj has no cells.

$Obj.Cell = \{\mathsf{OUT} \rightarrow \{\{p \in \mathcal{R}^n \rightarrow \langle 0, 0, 0\rangle\}, n, \emptyset, \emptyset\}\},$
where $n = Dim(\mathbf{Pl}).$

Spatial dimension: $Obj.Dim = n.$

Modification count incremented:
$Obj.ModCnt = Obj.ModCnt + 1.$

Update region: $Obj.ModRegion = Point\text{-}set(Obj).$

Post-conditions for the unsupported function

□ L and B are undefined and object Obj has no cells. (This is the same as then exceptional post-condition).

Simple object interrogation

Dj Get Point

INPUT

Obj : *DjObject* (D̈jinn point object).

OUTPUT

Pt : *DjPoint* (point of hidden D̈jinn type).

DESCRIPTION

This function extracts the point **Pt** of hidden D̈jinn type from the object *Obj*.

PRE-CONDITIONS

□ Object *Obj* consists of a single point cell.

ACTUAL POST-CONDITIONS

□ **Pt** is the single point in the point-set of the single cell of object *Obj*.

IDEAL POST-CONDITIONS

□ TRUE (there are no ideal conditions).

EXCEPTIONAL POST-CONDITIONS

□ FALSE (there are no exceptional input parameter values that satisfy the pre-condition).

POST-CONDITIONS FOR THE UNSUPPORTED FUNCTION

□ The point **Pt** is indeterminate.

Dj Get Line Segment

INPUT

Obj : *DjObject* (line-segment D̈jinn object).

OUTPUT

pt, **qt** : *DjPoint* (D̈jinn points).

DESCRIPTION

This function extracts the end-points from the line-segment object *Obj*.

PRE-CONDITIONS

□ The point-set of object *Obj* is a line-segment. It may consist of two or more cells: a cell for the segment interior and one or two cells comprising its end-points.

Actual post-conditions

- **pt** and **qt** are the boundary points of the line-segment.
- The orientation of line-segment interior cell bounded by point **pt** is $Point(\mathbf{qt}) - Point(\mathbf{pt})$.

Ideal post-conditions

- TRUE (there are no ideal conditions).

Exceptional post-conditions

- FALSE (there are no exceptional input parameter values that satisfy the pre-condition).

Post-conditions for the unsupported function

- Points **pt** and **qt** are indeterminate.

Dj Get Plane

Input

Obj : *DjObject* (D̈jinn plane object).

Output

Pl : *DjPlane* (D̈jinn plane).

Description

This function extracts the half-space **Pl** of hidden D̈jinn type from the object *Obj*. The half-space **Pl** may be one-, two- or three-dimensional; it may possibly be the universal set without points at infinity, but it may not be the empty set.

Pre-conditions

- The point-set of object *Obj* is a planar half-space. It may consist of two or more cells: cells for the half-space interior and cells for its plane boundary.

Actual post-conditions

- The half-space **Pl** is equal to the open regularized union of the point-sets of its interior cells.

Ideal post-conditions

- TRUE (there are no ideal conditions).

Exceptional post-conditions

- FALSE (there are no exceptional input parameter values that satisfy the pre-condition).

Post-conditions for the unsupported function

□ Plane **Pl** is indeterminate.

Canonical primitive objects

Dj Create Simplex

Input

n : *DjInteger* (dimension).

Output

Obj : *DjObject* (simplex Djinn object).
L : *DjLabel* (cell label).

Description

This function creates the Djinn object *Obj* consisting of two cells: a cell L, with an open point-set that is a solid simplex of dimension n, i.e. a tetrahedron, triangle or interval; and a cell B with a point-set that is the entire boundary of that simplex. Object *Obj* and the simplex cell L both have manifold dimension n. Its vertices are the origin and unit points on the n coordinate axes. Arbitrary simplices are created by subsequent transformation.

Pre-conditions

□ $1 \leq n \leq 3$.

Actual post-conditions

□ Object fields.

$Obj.ModCnt = 0$.

$Obj.ModRegion = Point\text{-}set(Obj)$.

$Obj.Dim = n$.

$\exists B \in DjLabel \bullet Domain(Obj.Cell) = \{\mathsf{OUT}, L, B\}$.

□ Cell fields.

$Obj.Cell(L).Atr = Obj.Cell(B).Atr = \emptyset$.

$Obj.Cell(L).Parents = Obj.Cell(B).Parents = \emptyset$.

$Obj.Cell(L).Dim = n$.

$Obj.Cell(B).Dim = n - 1$.

$Domain(Obj.Cell(L).Orient) =$
$\{\langle x_1, \ldots x_n\rangle \in \mathcal{R}^n | (\forall i \in [1 \ldots n] \bullet 0 < xi) \wedge (\sum_{1 \le i \le n} x_i < 1)\}$.

$Domain(Obj.Cell(B).Orient) = \partial Domain(Obj.Cell(L).Orient)$
$\forall v \in Range(Obj.Cell(L).Orient) \bullet v = \langle 0, \ldots\rangle$.

$Obj.Cell(B).Orient$: points map to outward plane normal.

□ Region outside the object.

$Obj.Cell(\mathsf{OUT}).Dim = Obj.Dim$.

$\cup_{i \in D} Domain(Obj.Cell(i).Orient) = \mathcal{R}^{Obj.Dim}$,
where $D = Domain(Obj.Cell)$,

$Obj.Cell(\mathsf{OUT}).Atr = \emptyset$.

$Obj.Cell(\mathsf{OUT}).Parents = \emptyset$.

$\forall \mathbf{p} \in Domain(\mathbf{Ut}) \bullet \mathbf{Ut}(\mathbf{p}) = \langle 0, \ldots\rangle$,
where $\mathbf{Ut} = Obj.Cell(\mathsf{OUT}).Orient$.

Ideal post-conditions

□ TRUE (there are no ideal conditions).

Exceptional post-conditions

□ FALSE (there are no exceptional input parameter values that satisfy the pre-condition).

Post-conditions for the unsupported function

□ Object *Obj* has no cells.

$Obj.Cell = \{\mathsf{OUT} \rightarrow \{\{\mathbf{p} \in \mathcal{R}^n \rightarrow \langle 0, 0, 0\rangle\}, n, \emptyset, \emptyset\}\}$.

Spatial dimension: $Obj.Dim = n$.

Modification count incremented:
$Obj.ModCnt = Obj.ModCnt + 1$.

Update region: $Obj.ModRegion = Point\text{-}set(Obj)$.

Dj Create Cube

Input

n	:	*DjInteger*	(dimension).

Output

Obj	:	*DjObject*	(hyper-cube D̈jinn object).
L	:	*DjLabel*	(cell label).

DESCRIPTION

This function creates the D̈jinn object *Obj* consisting of two cells: cell L with an open point-set that is a solid unit hyper-cube of dimension n, centred on the origin; and a cell B with a point-set that is the entire boundary of that hyper-cube. The object may be a cube, a square, or a one-dimensional interval, with three-, two- and one-dimensional points respectively, i.e. n is the spatial dimension of the object and the manifold dimension of cell L. Rectangular boxes of arbitrary size are created by subsequent differential scaling.

PRE-CONDITIONS

□ $1 \leq n \leq 3$.

ACTUAL POST-CONDITIONS

□ Object fields.

$Obj.ModCnt = 0$.

$Obj.ModRegion = Point\text{-}set(Obj)$.

$Obj.Dim = n$.

$\exists B \in DjLabel \bullet Domain(Obj.Cell) = \{\mathsf{OUT}, L, B\}$.

□ Cell fields.

$Obj.Cell(L).Atr = Obj.Cell(B).Atr = \emptyset$.

$Obj.Cell(L).Parents = Obj.Cell(B).Parents = \emptyset$.

$Obj.Cell(L).Dim = n$.

$Obj.Cell(B).Dim = n - 1$.

$Domain(Obj.Cell(L).Orient)$
$= \{\langle x_1, \ldots x_n\rangle \in \mathcal{R}^n | \forall i \in [1 \ldots n] \bullet (-1 < 2x_i < 1)\}\}$.

$Domain(Obj.Cell(B).Orient)$
$= \partial Domain(Obj.Cell(L).Orient)$.
$\forall v \in Range(Obj.Cell(L).Orient) \bullet v = \langle 0, \ldots\rangle$.

$Obj.Cell(B).Orient$: points map to outward plane normal.

□ Region outside the object.

$Obj.Cell(\mathsf{OUT}).Dim = Obj.Dim$.

$\cup_{i \in D} Domain(Obj.Cell(i).Orient) = \mathcal{R}^{Obj.Dim}$,
where $D = Domain(Obj.Cell)$.

$Obj.Cell(\mathsf{OUT}).Atr = \emptyset$.

$Obj.Cell(\mathsf{OUT}).Parents = \emptyset$.

$\forall \mathbf{p} \in Domain(\mathbf{Ut}) \bullet \mathbf{Ut}(\mathbf{p}) = \langle 0, \ldots\rangle$,
where $\mathbf{Ut} = Obj.Cell(\mathsf{OUT}).Orient$.

IDEAL POST-CONDITIONS

□ TRUE (there are no ideal conditions).

Exceptional post-conditions

□ FALSE (there are no exceptional input parameter values that satisfy the pre-condition).

Post-conditions for the unsupported function

□ Object *Obj* has no cells.

$Obj.Cell = \{\mathsf{OUT} \rightarrow \{\{\mathbf{p} \in \mathcal{R}^n \rightarrow \langle 0,0,0\rangle\}, n, \emptyset, \emptyset\}\}$.

Spatial dimension: $Obj.Dim = n$.

Modification count incremented:
$Obj.ModCnt = Obj.ModCnt + 1$.

Update region: $Obj.ModRegion = Point\text{-}set(Obj)$.

Dj Create Sphere

Input

n	:	*DjInteger*	(dimension).

Output

Obj	:	*DjObject*	(hyper-sphere D̈jinn object).
L	:	*DjLabel*	(cell label).

Description

This function creates the D̈jinn object *Obj* consisting of two cells; the cell L with an open point-set that is a solid unit-diameter hyper-sphere of dimension n, centred on the origin, and a cell B with a point-set that is the entire boundary of the hyper-sphere. The object may be a three-dimensional sphere, a two-dimensional circular disc, or a one-dimensional interval, with three-, two- and one-dimensional points respectively, i.e. n is the spatial dimension of the object and the manifold dimension of cell L. Spheres of arbitrary size and hyper-ellipsoids may be created by subsequent differential scaling.

Pre-conditions

□ $1 \leq n \leq 3$.

Actual post-conditions

□ Object fields.

$Obj.ModCnt = 0$.

$Obj.ModRegion = Point\text{-}set(Obj)$.

$Obj.Dim = n$.

$\exists B \in DjLabel \bullet Domain(Obj.Cell) = \{\mathsf{OUT}, L, B\}$.

□ Cell fields.

$Obj.Cell(L).Atr = Obj.Cell(B).Atr = \emptyset.$

$Obj.Cell(L).Parents = Obj.Cell(B).Parents = \emptyset.$

$Obj.Cell(L).Dim = n.$

$Obj.Cell(B).Dim = n - 1.$

$Domain(Obj.Cell(L).Orient) =$
$\{\langle x_1, \ldots x_n\rangle \in \mathcal{R}^n | \sum_{1 \le i \le n} x_i^2 < \frac{1}{4}\}.$

$Domain(Obj.Cell(B).Orient) =$
$\partial Domain(Obj.Cell(L).Orient).$

$\forall v \in Range(Obj.Cell(L).Orient) \bullet$
$v = \langle 0, \ldots\rangle.$

$Obj.Cell(B).Orient$: points map to outward sphere normal.

□ Region outside the object.

$Obj.Cell(\mathsf{OUT}).Dim = Obj.Dim.$

$\cup_{i \in D} Domain(Obj.Cell(i).Orient) = \mathcal{R}^{Obj.Dim},$
where $D = Domain(Obj.Cell),$

$Obj.Cell(\mathsf{OUT}).Atr = \emptyset.$

$Obj.Cell(\mathsf{OUT}).Parents = \emptyset.$

$\forall \mathbf{p} \in Domain(\mathbf{Ut}) \bullet \mathbf{Ut}(\mathbf{p}) = \langle 0, \ldots\rangle,$
where $\mathbf{Ut} = Obj.Cell(\mathsf{OUT}).Orient.$

Ideal post-conditions

□ TRUE (there are no ideal conditions).

Exceptional post-conditions

□ FALSE (there are no exceptional input parameter values that satisfy the pre-condition).

Post-conditions for the unsupported function

□ Object Obj has no cells.

$Obj.Cell = \{\mathsf{OUT} \to \{\{\mathbf{p} \in \mathcal{R}^n \to \langle 0, 0, 0\rangle\}, n, \emptyset, \emptyset\}\}.$
Spatial dimension: $Obj.Dim = n.$

Modification count incremented:
$Obj.ModCnt = Obj.ModCnt + 1.$

Update region: $Obj.ModRegion = Point\text{-}set(Obj).$

Special-purpose primitive objects

Dj Create Cone

INPUT

c : *DjReal* (tangent of half the cone apex angle).

OUTPUT

Obj : *DjObject* (cylinder or cone Djinn object).
L : *DjLabel* (cell label).

DESCRIPTION

This function creates the Djinn object *Obj*, usually consisting of two cells: cell L with an open point-set that is a solid frustum of a unit right circular cone; and a cell B with a point-set that is the entire boundary of that cone. The cone axis is the z-coordinate axis and the top and bottom planes of the frustum are equidistant from the xy-plane. Its height and the xy-plane cross-section diameter are unity. The tangent of half the cone apex angle is c.

$c = 0$	Cone is a right circular cylinder.
$\|c\| = 1$	Cone apex is part of the boundary cell.
$\|c\| > 1$	Cone apex constitutes a third cell C.
$0 < c < 1$	Top diameter is less than bottom diameter.
$0 > c > -1$	Top diameter is greater than bottom diameter.

Cones of arbitrary size and those with elliptical cross-section may be created by differential scaling.

PRE-CONDITIONS

□ TRUE (there are no pre-conditions).

ACTUAL POST-CONDITIONS

□ Object fields.

$Obj.ModCnt = 0.$

$Obj.ModRegion = Point\text{-}set(Obj).$

$Obj.Dim = 3.$

$\exists B \in DjLabel \bullet$
$Domain(Obj.Cell) = \{\mathsf{OUT}, L, B\}.$

$\exists B, C \in DjLabel \bullet$
$((|c| > 1) \Rightarrow Domain(Obj.Cell) = \{L, B, C\}$ (3 cells) $\wedge$
$((|c| \leq 1) \Rightarrow Domain(Obj.Cell) = \{L, B\}$ (2 cells).

□ Cell fields.

$\forall H \in Domain(Obj.Cell) \bullet$
$Obj.Cell(H).Atr = \emptyset,$
$Obj.Cell(H).Parents = \emptyset.$

$Obj.Cell(L).Dim = 3.$
$Obj.Cell(B).Dim = 2.$
If cell C exists, then $Obj.Cell(C).Dim = 0.$

The point-set of cell L:

$Domain(Obj.Cell(L).Orient) =$
$\{\langle x, y, z\rangle \in \mathcal{R}^3 | (4(x^2 + y^2) < (1 - 2cz)^2) \wedge (-1 < 2z < 1)\}.$

If cell C exists, its point-set is:

$Domain(Obj.Cell(C).Orient) = \{\langle 0, 0, 1/2c\rangle\}.$

The point-set of cell B:

$Domain(Obj.Cell(B).Orient) =$
$\partial Domain(Obj.Cell(L).Orient) - \{\langle 0, 0, 1/2c\rangle\}.$

$\forall v \in Range(Obj.Cell(L).Orient) \bullet v = \langle 0, \ldots\rangle.$

$Obj.Cell(B).Orient$: points map to outward cone normal.

□ Region outside the object.

$Obj.Cell(\mathsf{OUT}).Dim = Obj.Dim.$

$\cup_{i \in D} Domain(Obj.Cell(i).Orient) = \mathcal{R}^{Obj.Dim},$
where $D = Domain(Obj.Cell),$

$Obj.Cell(\mathsf{OUT}).Atr = \emptyset.$

$Obj.Cell(\mathsf{OUT}).Parents = \emptyset.$

$\forall \mathbf{p} \in Domain(\mathbf{Ut}) \bullet \mathbf{Ut}(\mathbf{p}) = \langle 0, \ldots\rangle,$
where $\mathbf{Ut} = Obj.Cell(\mathsf{OUT}).Orient.$

Ideal post-conditions

□ TRUE (there are no ideal conditions).

Exceptional post-conditions

□ FALSE (there are no exceptional input parameter values that satisfy the pre-condition).

Post-conditions for the unsupported function

□ Object Obj has no cells.

$Obj.Cell = \{\mathsf{OUT} \rightarrow \{\{\mathbf{p} \in \mathcal{R}^3 \rightarrow \langle 0, 0, 0\rangle\}, 3, \emptyset, \emptyset\}\}.$

Spatial dimension: $Obj.Dim = 3.$

Modification count incremented: $Obj.ModCnt = Obj.ModCnt + 1.$

Update region: $Obj.ModRegion = Point\text{-}set(Obj).$

Dj Create Pyramid

INPUT

n	:	*DjInteger*	(number of sides).
c	:	*DjReal*	(tangent of the inscribed cone's apex half-angle).

OUTPUT

Obj	:	*DjObject*	(D$\ddot{\text{j}}$inn object).
L	:	*DjLabel*	(cell label).

DESCRIPTION

This function creates the D$\ddot{\text{j}}$inn object *Obj* consisting of two cells; cell L with an open point-set is a solid pyramid and a cell B with a point-set that is the entire boundary of that pyramid. If $|c| > 1$ the vertex becomes a third cell. The pyramid created is the one with an inscribed cone specified by the parameter c, as defined for *Dj Create Cone*. Triangular and other regular prisms can be created as pyramids with zero apex angle. If non-regular prisms are required, certain ones can be created by differential scaling. Triangular prisms can be also created by sweeping a triangular simplex.

PRE-CONDITIONS

□ $n > 2$.

ACTUAL POST-CONDITIONS

□ Object fields.

$Obj.ModCnt = 0$.

$Obj.ModRegion = Point\text{-}set(Obj)$.

$Obj.Dim = 3$.

$\exists B \in DjLabel \bullet Domain(Obj.Cell) = \{\mathsf{OUT}, L, B\}$.

□ Cell fields.

$Obj.Cell(L).Atr = Obj.Cell(B).Atr = \emptyset$.

$Obj.Cell(L).Parents = Obj.Cell(B).Parents = \emptyset$.

$Obj.Cell(L).Dim = 3$.

$Obj.Cell(B).Dim = 2$.

The cell's point-set is the set-intersection of planes:

$Domain(Obj.Cell(L).Orient) =$
$\cap_{i=0}^{n-1}\{\langle x, y, z\rangle \in \mathcal{R}^3 | (-1 < 2z < 1) \wedge$
$(x \cos 2\pi i/n + y \sin 2\pi i/n + z < 1/2c)\}$,
where c is the tangent of the half apex angle of the inscribed cone.

$Domain(Obj.Cell(B).Orient) =$
$\partial Domain(Obj.Cell(L).Orient)$.

$\forall v \in Range(Obj.Cell(L).Orient) \bullet v = \langle 0, \ldots\rangle$.

where $Obj.Cell(\mathrm{B}).Orient$ points the map to the outward normal of the plane face.

□ Region outside the object.

$Obj.Cell(\mathsf{OUT}).Dim = Obj.Dim$.

$\cup_{i \in D} Domain(Obj.Cell(i).Orient) = \mathcal{R}^{Obj.Dim}$,
where $D = Domain(Obj.Cell)$,

$Obj.Cell(\mathsf{OUT}).Atr = \emptyset$,

$Obj.Cell(\mathsf{OUT}).Parents = \emptyset$.

$\forall \mathbf{p} \in Domain(\mathbf{Ut}) \bullet \mathbf{Ut}(\mathbf{p}) = \langle 0, \ldots \rangle$,
where $\mathbf{Ut} = Obj.Cell(\mathsf{OUT}).Orient$.

Ideal post-conditions

□ TRUE (there are no ideal conditions).

Exceptional post-conditions

□ FALSE (there are no exceptional input parameter values that satisfy the pre-condition).

Post-conditions for the unsupported function

□ Object Obj has no cells.

$Obj.Cell = \{\mathsf{OUT} \rightarrow \{\{p \in \mathcal{R}^3 \rightarrow \langle 0,0,0 \rangle\}, 3, \emptyset, \emptyset\}\}$.

Spatial dimension: $Obj.Dim = 3$.

Modification count incremented:
$Obj.ModCnt = Obj.ModCnt + 1$.

Update region: $Obj.ModRegion = Point\text{-}set(Obj)$.

Dj Create Cyclide

Input

d, e : *DjReal* (diameters of two zx-sections).

Output

Obj : *DjObject* (D̈jinn object).
L : *DjLabel* (cell label).

Description

This function creates the D̈jinn object *Obj* consisting of two or three cells: a cell L with a point-set that is a solid cyclide and a cell B with a point-set that is the boundary of that cyclide less the singular points of its boundary. If they exist, these points constitute the third cell: they only occur on spindle or horned cyclides. The cyclide has xy and xz as planes of symmetry, and the xz-section consists of two circles, centred at [1,0,0]

and [-1,0,0]. This cross-section defines the cyclide, and is itself specified by the diameters d and e of these two circles. More general cyclides may be created by scaling.

Pre-conditions

□ $n > 2$.

Actual post-conditions

□ Object fields.

$Obj.ModCnt = 0.$

$Obj.ModRegion = Point\text{-}set(Obj).$

$Obj.Dim = 3.$

$\exists B, C \in DjLabel \bullet ((d + e > 2) \Rightarrow$
$Domain(Obj.Cell) = \{\mathsf{OUT}, L, B, C\} \wedge$
$((d + e \leq 2) \Rightarrow Domain(Obj.Cell) = \{\mathsf{OUT}, L, B\}.$

□ Cell fields.

$\forall H \in Domain(Obj.Cell) \bullet$
$Obj.Cell(H).Atr = \emptyset$
$Obj.Cell(H).Parents = \emptyset.$

$Obj.Cell(L).Dim = 3.$
$Obj.Cell(B).Dim = 2.$
$Obj.Cell(C).Dim = 0.$

The point-set of cell L:

$Domain(Obj.Cell(L).Orient) = \{\mathbf{p} \in \mathcal{R}^3 | F(\mathbf{p}) < 0\},$
where $F(\mathbf{p} = \langle x, y, z \rangle) =$
$(x^2 + y^2 + z^2 - m^2 + b^2)^2 - 4(x - cm)^2 + 4b^2y^2$
and $4m = d + e, 4c = d - e, 1 = b^2 + c^2.$

The point-set of cell C, if cell C exists:

$Domain(Obj.Cell(B).Orient) = S,$
where $S =$ singular points of $\{\partial Obj.Cell(L).Point\text{-}set\}.$

The point-set of cell B:

$Domain(Obj.Cell(B).Orient) =$
$\partial Domain(Obj.Cell(L).Orient) - S.$

$\forall v \in Range(Obj.Cell(L).Orient) \bullet v = \langle 0, \ldots \rangle.$

The orientation of cell B: points map to outward normal of the cyclide face:

$\forall v \in Domain(Obj.Cell(B).Orient) \bullet v = \nabla F.$

□ Region outside the object.

$Obj.Cell(\mathsf{OUT}).Dim = Obj.Dim$.
$\cup_{i \in D} Domain(Obj.Cell(i).Orient) = \mathcal{R}^{Obj.Dim}$,
where $D = Domain(Obj.Cell)$.
$Obj.Cell(\mathsf{OUT}).Atr = \emptyset$.
$Obj.Cell(\mathsf{OUT}).Parents = \emptyset$.
$\forall \mathbf{p} \in Domain(\mathbf{Ut}) \bullet \mathbf{Ut}(\mathbf{p}) = \langle 0, \ldots \rangle$,
where $\mathbf{Ut} = Obj.Cell(\mathsf{OUT}).Orient$.

Ideal post-conditions

□ TRUE (there are no ideal conditions).

Exceptional post-conditions

□ FALSE (there are no exceptional input parameter values that satisfy the pre-condition).

Post-conditions for the unsupported function

□ Object *Obj* has no cells.

$Obj.Cell = \{\mathsf{OUT} \rightarrow \{\{p \in \mathcal{R}^3 \rightarrow \langle 0,0,0 \rangle\}, 3, \emptyset, \emptyset\}\}$.

Spatial dimension: $Obj.Dim = 3$.

Modification count incremented:
$Obj.ModCnt = Obj.ModCnt + 1$.

Update region: $Obj.ModRegion = Point\text{-}set(Obj)$.

Dj Create Helix

Input

n	:	*DjInteger*	(dimension).
c	:	*DjReal*	(tangent of half the apex angle of the cone around which the tapered helix is wrapped).

Output

Obj	:	*DjObject*	(D̈jinn object).
L	:	*DjLabel*	(cell label).

Description

This function creates the D̈jinn object *Obj* consisting of two cells; the cell L with a point-set that is one turn of a plane spiral ($n = 2$) or a tapered helix on the surface of a conical frustum ($n = 3$) and a cell B consisting of its two end-points. The axis of the spiral or helix is the z-coordinate axis and its start point is negative in x and z and zero in y, while the radius varies from $\frac{1-c}{2}$ to $\frac{1+c}{2}$. Since the frustum is specified by c, as described in the specification of *Dj Create Cone*, the helix has unit pitch and unit diameter in the xy-plane. A spiral or helix with an arbitrary number of

turns can be created using Djinn functions to copy, scale, translate and union Djinn objects. Differential scaling can be used to change pitch and diameter.

Pre-conditions

□ $2 \leq n \leq 3$.

Actual post-conditions

□ Object fields.

$Obj.ModCnt = 0$.

$Obj.ModRegion = Point\text{-}set(Obj)$.

$Obj.Dim = 3$.

$\exists B \in DjLabel \bullet Domain(Obj.Cell) = \{\mathsf{OUT}, L, B\}$.

□ Cell fields.

$Obj.Cell(L).Atr = Obj.Cell(B).Atr = \emptyset$.

$Obj.Cell(L).Parents = Obj.Cell(B).Parents = \emptyset$.

$Obj.Cell(L).Dim = 1$.

$Obj.Cell(B).Dim = 0$.

Point-set $(n = 2)$:

$Domain(Obj.Cell(L).Orient)$
$= \{< x, y, z > \in \mathcal{R}^2 | \exists \theta \in (-\pi \ldots \pi) \bullet$
$(2x = (1 - c\theta/\pi)\cos\theta) \wedge (2y = (1 - c\theta/\pi)\sin\theta)\}$.

$Domain(Obj.Cell(B).Orient) = \{< -1 - c, 0 >, < -1 + c, 0 >\}$.

Point-set $(n = 3)$:

$Domain(Obj.Cell(L).Orient) =$
$\{< x, y, z > \in \mathcal{R}^3 | \exists \theta \in (-\pi \ldots \pi)$
$\bullet (2x = (1 - 2cz)\cos\theta) \wedge (2y = (1 - 2cz)\sin\theta) \wedge (2z\pi = \theta)\}$.

$Domain(Obj.Cell(B).Orient) =$
$\{< -1 - c, 0, -\frac{1}{2} >, < -1 + c, 0, \frac{1}{2} >\}$
$\forall v \in Range(Obj.Cell(B).Orient) \bullet v = \langle 0, \ldots \rangle$.

$Obj.Cell(B).Orient$: points map to tangent direction with increasing θ.

□ Region outside the object.

$Obj.Cell(\mathsf{OUT}).Dim =$
$n \cup_{i \in D} Domain(Obj.Cell(i).Orient) = \mathcal{R}^n$,
where $D = Domain(Obj.Cell)$.

$Obj.Cell(\mathsf{OUT}).Atr = \emptyset$.

$Obj_1.Cell(\mathsf{OUT}).Parents = \emptyset$.

$\forall \mathbf{p} \in Domain(\mathbf{Ut}) \bullet \mathbf{Ut}(\mathbf{p}) = \langle 0, \ldots \rangle$,
where $\mathbf{Ut} = Obj.Cell(\mathsf{OUT}).Orient$.

Ideal post-conditions

□ TRUE (there are no ideal conditions).

Exceptional post-conditions

□ FALSE (there are no exceptional input parameter values that satisfy the pre-condition).

Post-conditions for the unsupported function

□ Object *Obj* has no cells.

$Obj.Cell = \{\mathsf{OUT} \rightarrow \{\{p \in \mathcal{R}^n \rightarrow \langle 0,0,0\rangle\}, n, \emptyset, \emptyset\}\}$.

Spatial dimension: $Obj.Dim = n$.

Modification count incremented:

$Obj.ModCnt = Obj.ModCnt + 1$.

Update region: $Obj.ModRegion = Point\text{-}set(Obj)$.

Dj Create NURBS Curve

Input

d	:	*DjInteger*	(degree of curve).
p	:	n-sequence of *DjVector*	(control points).
t	:	m-sequence of *DjReal*	(knots).
w	:	n-sequence of *DjReal*	(weights).

Output

Obj	:	*DjObject*	(D̈j̈inn object).
L	:	*DjLabel*	(cell label).

Description

This function creates the D̈j̈inn object *Obj*, usually consisting of two cells; the cell L with a point-set that is a NURBS curve of degree d; and cell B, usually consisting of its two end-points. If the NURBS is a closed loop, then there are no point cells. If the curve intersects itself, then additional cells are created to ensure that each cell is manifold. The curve is specified by n weighted control points p and a parameter knot m-vector, possibly with repeated elements. Its coordinate dimension is that of points p.

Pre-conditions

□ There are at least two control points p: $n \geq 2$.

□ There are (degree+1) more knots than control points: $m = n + d + 1$.

ACTUAL POST-CONDITIONS

□ Object fields.

$Obj.ModCnt = 0.$

$Obj.ModRegion = Point\text{-}set(Obj).$

$Obj.Dim =$ dimension of control points.

$\exists B \in DjLabel \bullet Domain(Obj.Cell) = \{\mathsf{OUT}, L, B\}.$

□ Cell fields.

$Obj.Cell(L).Atr = Obj.Cell(B).Atr = \emptyset.$

$Obj.Cell(L).Parents = Obj.Cell(B).Parents = \emptyset.$

$Obj.Cell(L).Dim = 1.$

$Obj.Cell(B).Dim = 0.$

If points $p = \mathbf{p}_0 \dots \mathbf{p}_{n-1}$ and knots $t = t_0 \dots t_{m-1}$, then the remaining cell fields are:

$$Domain(Obj.Cell(L).Orient) = \left\{ \mathbf{q} \in \mathcal{R}^d | \exists t \in (t_d \dots t_n) \bullet \mathbf{q} = \frac{\sum_{i=0}^{n-1} w_i \mathbf{p}_i N_i^d(t)}{\sum_{i=0}^{n-1} w_i N_i^d(t)} \right\},$$

where $N_i^d(t)$ is the ith normalized B-spline basis function of degree d, defined on the knot subset $t_i \dots t_{i+d}$.

$$Domain(Obj.Cell(B).Orient) = \left\{ \frac{\sum_{i=0}^{n-1} w_i \mathbf{p}_i N_i^d(t_d)}{\sum_{i=0}^{n-1} w_i N_i^d(t_d)}, \frac{\sum_{i=0}^{n-1} w_i \mathbf{p}_i N_i^d(t_n)}{\sum_{i=0}^{n-1} w_i N_i^d(t_n)} \right\}.$$

$\forall v \in Range(Obj.Cell(B).Orient) \bullet v = \langle 0, \dots \rangle.$

$Obj.Cell(L).Orient(q) = \frac{\partial q}{\partial t},$
for all q in the point-set of cell L, as defined above.

□ Region outside the object.

$Obj.Cell(\mathsf{OUT}).Dim = Obj.Dim$
$\cup_{i \in D} Domain(Obj.Cell(i).Orient) = \mathcal{R}^{Obj.Dim},$
where $D = Domain(Obj.Cell),$

$Obj.Cell(\mathsf{OUT}).Atr = \emptyset.$

$Obj.Cell(\mathsf{OUT}).Parents = \emptyset.$

$\forall \mathbf{p} \in Domain(\mathbf{Ut}) \bullet \mathbf{Ut}(\mathbf{p}) = \langle 0, \dots \rangle,$
where $\mathbf{Ut} = Obj.Cell(\mathsf{OUT}).Orient.$

IDEAL POST-CONDITIONS

□ TRUE (there are no ideal conditions).

EXCEPTIONAL POST-CONDITIONS

□ When the curve is closed, i.e. when the two points in the above definition of the point-set of cell B are coincident, that point becomes a member of the point-set of cell L, and object Obj contains only one cell.

- When the curve degenerates to a single point, that point constitutes the point-set of the single cell L of object *Obj* with dimension 0 and no orientation.
- When two parameter values in the definition above give the same point and exactly one of those parameters is in the interior of the parametric domain, the object is stratified by moving that point from cell L to cell B.
- When at least one pair of interior parameter values give the same point, the object is stratified, such point or points constituting a third cell of manifold dimension 0 (see Part I, Chapter B).

Post-conditions for the unsupported function

- Object *Obj* has no cells.

  $Obj.Cell = \{\mathsf{OUT} \to \{\{\mathbf{p} \in \mathcal{R}^n \to \langle 0,0,0\rangle\}, n, \emptyset, \emptyset\}\}$.
  where n = the dimension of the control points.
  Spatial dimension:
  $Obj.Dim$ = the dimension of the control points.
  Modification count incremented:
  $Obj.ModCnt = Obj.ModCnt + 1$.
  Update region: $Obj.ModRegion = Point\text{-}set(Obj)$.

Dj Create NURBS Surface

Input

a, b	:	*DjInteger*	(parametric degrees of surface).
p	:	n-sequence of m-sequence of *DjVector*	(control points).
u	:	r-sequence of *DjReal*	(knots).
v	:	s-sequence of *DjReal*	(knots).
w	:	n-sequence of m-sequence of *DjReal*	(weights).

Output

Obj	:	*DjObject*	(D$\ddot{\text{j}}$inn object).
L	:	*DjLabel*	(cell label).

Description

This function creates the D$\ddot{\text{j}}$inn object *Obj* usually consisting of two cells; the cell L with a point-set that is a NURBS surface of parametric degrees a and b and a cell consisting of its boundary curve (unless it is a closed surface). If the surface intersects itself, then additional cells are created to ensure that each cell is manifold. The surface is specified by $n \times m$ weighted control points p, and a parameter $r \times s$ knot array possibly with repeated elements. Its spatial dimension is that of the points p.

PRE-CONDITIONS

- □ There are at least 2×2 control points p: $m, n \geq 2$.
- □ There are (degree+1) more knots than control points in each parametric direction:

  $m = r + a + 1$
  $n = s + b + 1$.

ACTUAL POST-CONDITIONS

- □ Object fields.

  $Obj.ModCnt = 0$.

  $Obj.ModRegion = Point\text{-}set(Obj)$.

  $Obj.Dim =$ Dimension of control points.
  $\exists B \in DjLabel \bullet Domain(Obj.Cell) = \{\mathsf{OUT}, L, B\}$.
- □ Cell fields.

  $Obj.Cell(L).Atr = Obj.Cell(B).Atr = \emptyset$.

  $Obj.Cell(L).Parents = Obj.Cell(B).Parents = \emptyset$.

  $Obj.Cell(L).Dim = 2$.

  $Obj.Cell(B).Dim = 1$.

  If points $p = \mathbf{p}_{0,0} \dots \mathbf{p}_{n-1,m-1}$ and knots $u = u_0 \dots u_{r-1}$, and $v = v_0 \dots v_{s-1}$, then:

  $Domain(Obj.Cell(L).Orient)$
  $= \{ \mathbf{q} \in \mathcal{R}^d | \exists u \in (u_a \dots u_n) \bullet \exists v \in (v_b \dots v_m) \bullet$
  $\mathbf{q}(u,v) = \frac{\sum_{i=0}^{n-1} \sum_{j=0}^{m-1} w_{i,j} \mathbf{p}_{i,j} N_i^a(u) N_j^b(v)}{\sum_{i=0}^{n-1} \sum_{j=0}^{m-1} w_{i,j} N_i^a(u) N_j^b(v)} \}$,

  where $N_i^d(u)$ is the ith normalized B-spline basis function of degree d, defined on the knot subset $u_i \dots u_{i+a}$.

  Using the previous definition for $\mathbf{q}(u,v)$,

  $Domain(Obj.Cell(B).Orient) =$
  $\{\mathbf{q} \in \mathcal{R}^d | \exists u \in [u_a \dots u_n] \bullet \exists v \in [v_b \dots v_m] \bullet$
  $(\mathbf{q} = \mathbf{q}(u_a, v)) \vee (\mathbf{q} = \mathbf{q}(u_n, v)) \vee$
  $(\mathbf{q} = \mathbf{q}(u, v_b)) \vee (\mathbf{q} = \mathbf{q}(u, v_m))\}$.

  $Obj.Cell(B).Orient =$
  $\{\mathbf{q}(u_a, v_b \dots v_m) \rightarrow \partial\mathbf{q}\partial v, \mathbf{q}(u_n, v_b \dots v_m) \rightarrow$
  $\partial\mathbf{q}\partial v, \mathbf{q}(u_a \dots u_n, v) \rightarrow \partial\mathbf{q}\partial u, \mathbf{q}(u_a \dots u_n, v_m) \rightarrow \partial\mathbf{q}\partial u\}$.

  $Obj.Cell(L).Orient(\mathbf{q}) = \partial\mathbf{q}\partial u \times \partial\mathbf{q}\partial v$,
  for all $\mathbf{q}$ in the point-set of cell L defined above.
- □ Region outside the object.

$Obj.Cell(\mathsf{OUT}).Dim = Obj.Dim$.

$\cup_{i \in D} Domain(Obj.Cell(i).Orient) = \mathcal{R}^{Obj.Dim}$,
where $D = Domain(Obj.Cell)$,

$Obj.Cell(\mathsf{OUT}).Atr = \emptyset$.

$Obj.Cell(\mathsf{OUT}).Parents = \emptyset$.

$\forall \mathbf{p} \in Domain(\mathbf{Ut}) \bullet \mathbf{Ut}(\mathbf{p}) = \langle 0, \ldots \rangle$,
where $\mathbf{Ut} = Obj.Cell(\mathsf{OUT}).Orient$.

Ideal post-conditions

□ TRUE (there are no ideal conditions).

Exceptional post-conditions

□ When the boundary curves at $u = u_a$ and $u = u_n$, or $v = v_b$ and $v = v_m$ in the above definition are equal, points in those curves become members of point-set of cell L.

□ When the surface patch is closed, points in the boundary curves at $u = u_a$, $u = u_n$, $v = v_b$, $v = v_m$ in the above definition become members of the point-set of cell L, and object Obj contains only one cell.

□ When the surface patch degenerates to a curve, points on that curve constitute the point-set of cell L, and object Obj contains a cell of manifold dimension 1 and, if it is not a closed curve, its endpoints form another cell.

□ When two parameter values in the definition above give the same point and at least one of those parameters is in the interior of the parametric domain, the object is stratified into up to six cells (see Part I, Chapter B).

Post-conditions for the unsupported function

□ Object Obj has no cells.

$Obj.Cell = \{\mathsf{OUT} \rightarrow \{\{\mathbf{p} \in \mathcal{R}^n \rightarrow \langle 0,0,0 \rangle\}, n, \emptyset, \emptyset\}\}$,
where n = dimensionality of control points.

Spatial dimension:
$Obj.Dim$ = dimensionality of control points.

Modification count incremented:
$Obj.ModCnt = Obj.ModCnt + 1$.

Update region: $Obj.ModRegion = Point\text{-}set(Obj)$.

Dj Reset Modification Record

Input and output

Obj : *DjObject* (Djinn object).

Description

This function resets the modification count of object *Obj* to zero and its update region to empty.

Pre-conditions

□ TRUE (there are no pre-conditions).

Actual post-conditions

□ The modification count of *Obj* is reset, so that $Obj.ModCnt = 0$.

□ The update region of *Obj* is reset, so that $Obj.ModRegion = \emptyset$.

□ All other fields of *Obj* and its cells are unaffected.

Ideal post-conditions

□ TRUE (there are no ideal conditions).

Exceptional post-conditions

□ FALSE (the function performs normally even if the object has no cells).

Post-conditions for the unsupported function

□ FALSE (the function has no effect).

Dj Copy

Input

Obj : *DjObject* (Djinn object).

Output

ObjR : *DjObject* (Djinn object).

Description

This function creates a new object *ObjR*, with the same cellular structure, labels and attributes as *Obj*. No data is shared between the new object and the original one.

Pre-conditions

□ TRUE (there are no pre-conditions).

Actual post-conditions

- □ Modification count of $ObjR$ is inherited:
  $ObjR.ModCnt = Obj.ModCnt$.
- □ Update region of $ObjR$ is inherited:
  $ObjR.ModRegion = Obj.ModRegion$.
- □ Dimension of $ObjR$ is inherited: $ObjR.Dim = Obj.Dim$.
- □ Each cell L of $ObjR$ is identical (i.e. it has the same attribute, parentage, point-set, orientation, and dimension) to a cell of Obj; and it inherits its label: $ObjR.Cell = Obj.Cell$.

Ideal post-conditions

- □ TRUE (there are no ideal conditions).

Exceptional post-conditions

- □ If object Obj has no cells, $ObjR$ also has no cells: $ObjR = Obj$.

Post-conditions for the unsupported function

- □ Object $ObjR$ has no cells.
  $ObjR.Cell = \{\mathsf{OUT} \to \{\{p \in \mathcal{R}^n \to \langle 0,0,0\rangle\}, n, \emptyset, \emptyset\}\}$,
  where $n = Obj.Dim$.
  Spatial dimension: $ObjR.Dim = Obj.Dim$.
  Modification count: $ObjR.ModCnt = 0$.
  Update region: $ObjR.ModRegion = \emptyset$.

Part II
2
Partitioning and related operators

Dj Partition

INPUT AND OUTPUT

Obj_1 : *DjObject* (D̈jinn object).

INPUT

Obj_2 : *DjObject* (D̈jinn object).

DESCRIPTION

This function modifies an object Obj_1 by partitioning its cells with the cells of Obj_2; the point-set of Obj_1 remains unchanged and Obj_2 is unaffected. The point-sets of the cells of the result are:

- Those parts of the cells of Obj_1 which are outside the point-set of Obj_2.
- Cells created by the pair-wise set-intersection of cells from each operand, i.e. the cells computed by the D̈jinn intersection function.

PRE-CONDITIONS

- TRUE (there are no pre-conditions).

ACTUAL POST-CONDITIONS

- Object fields.

  Modification count is incremented:
  $Obj'_1.ModCnt = Obj_1.ModCnt + 1$.

  Update region: $Obj'_1.ModRegion =$
  $Obj_1.ModRegion \cup (Point\text{-}set(Obj_1) \cap Point\text{-}set(Obj_2))$.

  Spatial dimension: $Obj'_1.Dim = Obj_1.Dim$.

- □ Cell point-sets and orientation: $F' = F\#E$, where F and E are sets of point/orientation maps for cells of objects Obj_1 and Obj_2 (see Part I, Chapter 2).
- □ Remaining cell fields of each cell L of Obj'_1 outside Obj_2 .

  L partially outside Obj_2 . . . one parent cell:
  $Obj'_1.Cell(L).Parents = \{\langle L, \mathsf{OUT}\rangle\}$.

  L equal to cell of Obj_1 . . . no parents:
  $Obj'_1.Cell(L).Parents = \emptyset$.

  Inherited label and attributes:
  $Obj'_1.Cell(L).Atr = Obj_1.Cell(L).Atr$.

  Inherited manifold dimension:
  $Obj'_1.Cell(L).Dim = Obj_1.Cell(L).Dim$.
- □ Remaining cell fields of each cell L with point-set inside cells A and B of Obj_1 and Obj: Same as cells created by the function *Dj Intersect*.
- □ Each undivided cell of Obj_1 retains its label.
- □ Region outside the object.

  $Obj'_1.Cell(\mathsf{OUT}).Dim = Obj'_1.Dim$.

  $\cup_{i\in D} Domain(Obj'_1.Cell(i).Orient) = \mathcal{R}^{Obj'.Dim}$,
  where $D = Domain(Obj'_1.Cell)$.

  $Obj'_1.Cell(\mathsf{OUT}).Atr = Obj_1.Cell(\mathsf{OUT}).Atr$.
  $Obj'_1.Cell(\mathsf{OUT}).Parents = \{\langle \mathsf{OUT}, \mathsf{OUT}\rangle\}$.
  $\forall \mathbf{p} \in Domain(Obj'_1.Cell(\mathsf{OUT}).Orient)\bullet$
  $Obj'_1.Cell(\mathsf{OUT}).Orient(\mathbf{p}) = \langle 0, \ldots\rangle$.

Ideal post-conditions

- □ TRUE (there are no ideal conditions).

Exceptional post-conditions

- □ If Obj_1 or Obj_2 has no cells, then the function has no effect:
  $Obj'_1 = Obj_1$.

Post-conditions for the unsupported function

- □ The function has no effect: $Obj'_1 = Obj_1$.

Dj Departition

Input and output

Obj : *DjObject* (Djinn object).

Input

S : *DjLabelSet* (cell labels).

DESCRIPTION

This function combines the cells of Object Obj in the set S into a single cell, thus leaving the point-set of Obj unchanged.

PRE-CONDITIONS

□ TRUE (there are no pre-conditions).

ACTUAL POST-CONDITIONS

□ Object fields.

Modification count is incremented:
$Obj'.ModCnt = Obj.ModCnt + 1$.

Update region:
$Obj'.ModRegion = Obj.ModRegion \cup Point\text{-}set(Obj)$.

Spatial dimension: $Obj'.Dim = Obj.Dim$.

□ Cell point-sets and orientation: $F' = F@E$, where F is the set of point/orientation maps for the cells of Obj and E is the set of cells identified by the labels S (see Part I, Chapter 2).

□ Remaining cell fields of each cell L not in S.

Inherited attributes: $Obj'.Cell(L).Atr = Obj.Cell(L).Atr$.

Inherited label; one parent cell:
$Obj'.Cell(L).Parents = \{\langle L, \mathsf{OUT}\rangle\}$.

Inherited manifold dimension:
$Obj'.Cell(L).Dim = Obj.Cell(L).Dim$.

□ Remaining cell fields of each cell L derived from a set of cells R, all in S.

No attributes: $Obj'.Cell(L).Atr = \emptyset$.

Multiple parent cells:
$Obj'.Cell(L).Parents = \{\langle A, \mathsf{OUT}\rangle | A \in R\}$.

Manifold dimension:
$Obj'.Cell(L).Dim = \max(Obj.Cell(A).Dim | A \in R)$.

□ Region outside the object.

$Obj'.Cell(\mathsf{OUT}).Dim = Obj'.Dim$.

$\cup_{i \in D} Domain(Obj'.Cell(i).Orient) = \mathcal{R}^{Obj'.Dim}$,
where $D = Domain(Obj'.Cell)$.

$Obj'.Cell(\mathsf{OUT}).Atr = Obj.Cell(\mathsf{OUT}).Atr$.

$Obj'.Cell(\mathsf{OUT}).Parents = \{\langle \mathsf{OUT}, \mathsf{OUT}\rangle\}$.

$\forall \mathbf{p} \in Domain(Obj'.Cell(\mathsf{OUT}).Orient) \bullet$
$Obj'.Cell(\mathsf{OUT}).Orient(\mathbf{p}) = \langle 0, \ldots\rangle$.

IDEAL POST-CONDITIONS

□ TRUE (there are no ideal conditions).

Exceptional post-conditions

- If Obj has no cells, the function has no effect: $Obj' = Obj$.
- If set S contains no cell of object Obj, the function has no effect: $Obj' = Obj$.

Post-conditions for the unsupported function

- The function has no effect: $Obj' = Obj$.

Dj Subdivide Boundary

Input and output

Obj : *DjObject* (Djinn object).

Input

S : *DjLabelSet* (cell labels).
d : *DjInteger* (continuity).

Description

This function subdivides each cell of object Obj in the set S into component cells with geometric continuity of order C^d. All boundaries common to the same set of these new cells also become a new cell, similarly for boundaries common to sets of all these new cells and so on.

For example, if $d = 0$, each connected component of each cell in S becomes a new cell and boundaries of the original cell that now bound distinct cells are also subdivided, i.e. the frontier condition is preserved (see Part I, Chapter B). The same effect occurs when $d = 1$ but in addition, each line of tangent plane discontinuity within a face cell becomes a new edge cell that separates new cell subdivisions of the original cell. Also, each point of tangent discontinuity within an original or new edge cell becomes a new vertex cell that separates new cell subdivisions of that edge.

Pre-conditions

- Continuity is positive ($d \geq 0$).

Actual post-conditions

- Object fields.

$$Obj'.ModCnt = Obj.ModCnt + 1.$$
$$Obj'.ModRegion = Obj.ModRegion \cup \{Domain(Obj.Cell(C).Orient) | C \in S\}.$$
$$Obj'.Dim = Obj.Dim.$$

- □ Fields of each cell $L \notin S$ and cells in S of 0 or maximal manifold dimension (this includes the region outside object Obj').

  $Domain(Obj'.Cell(L).Orient) =$
  $\quad Domain(Obj.Cell(L).Orient)$.

  $Obj'.Cell(L).Orient = Obj.Cell(L).Orient$.

  $Obj'.Cell(L).Dim = Obj.Cell(L).Dim$.

  $Obj'.Cell(L).Atr = Obj.Cell(L).Atr$.

  $Obj'.Cell(L).Parents = \{\langle L, \mathsf{OUT}\rangle\}$.

- □ Fields of each cell L of Obj' with a point-set that is a subset of that of a cell $H \in S$.

  $Domain(Obj'.Cell(L).Orient)$ is the maximal subset with C^d continuity that satisfies the frontier condition:

  $\partial Domain(Obj'.Cell(L).Orient) =$
  $\quad \cup_{i \in D} Domain(Obj'.Cell(i).Orient)$,
  $\quad\quad$ where $D \subset Domain(Obj'.Cell)$.

  $Obj'.Cell(L).Orient$ is derived from its point-set.
  $Obj'.Cell(L).Dim$ is derived from its point-set.
  $Obj'.Cell(L).Atr = Obj.Cell(H).Atr$ (inherited attributes).
  $Obj'.Cell(L).Parents = \{\langle H, \mathsf{OUT}\rangle\}$.

Ideal post-conditions

- □ TRUE (there are no ideal conditions).

Exceptional post-conditions

- □ If Obj has no cells, then it remains unchanged ($Obj' = Obj$).
- □ If no element of set S identifies a cell of Obj, then Obj remains unchanged ($Obj' = Obj$).

Post-conditions for the unsupported function

- □ FALSE (the object Obj remains unchanged).

Dj Drain

Input and output

Obj : *DjObject* (Djinn object).

Input

S : *DjLabelSet* (cell labels).

DESCRIPTION

This function removes each cell in set S from object Obj, provided it is not the boundary of another cell. For example, this function could remove the interior solid cell from a primitive sphere with two cells, its interior and its bounding surface.

PRE-CONDITIONS

□ TRUE (there are no pre-conditions).

ACTUAL POST-CONDITIONS

□ Object fields.

$Obj'.ModCnt = Obj.ModCnt + 1.$

$Obj'.ModRegion =$
$\quad Obj.ModRegion \cup (Point\text{-}set(Obj') - Point\text{-}set(Obj)).$

$Obj'.Dim = Obj.Dim.$

$Domain(Obj'.Cell) = Domain(Obj.Cell) - S'.$
where set S' is set S less those cells that bound a cell of Obj not in S.

□ Fields of each retained cell L.

$\forall L \notin S' \bullet$
$\quad Obj'.Cell(L).Orient = Obj.Cell(L).Orient.$
$\quad Obj'.Cell(L).Dim = Obj.Cell(L).Dim.$
$\quad Obj'.Cell(L).Atr = Obj.Cell(L).Atr.$
$\quad Obj'.Cell(L).Parents = \{\langle L, \mathsf{OUT}\rangle\}.$

□ Region outside the object.

$Obj'.Cell(\mathsf{OUT}).Dim = Obj.Cell(\mathsf{OUT}).Dim.$
$\cup_{i \in D} Domain(Obj_1.Cell(i).Orient) = \mathcal{R}^{Obj.Dim},$
$\quad$ where $D = Domain(Obj_1.Cell).$
$Obj'.Cell(\mathsf{OUT}).Atr = Obj.Cell(\mathsf{OUT}).Atr.$
$Obj'.Cell(\mathsf{OUT}).Parents =$
$\quad \{\langle L, \mathsf{OUT}\rangle | L \in S'\} \cup \{\langle \mathsf{OUT}, \mathsf{OUT}\rangle\}.$
$\forall \mathbf{p} \in Domain(\mathbf{Ut}) \bullet \mathbf{Ut}(\mathbf{p}) = \langle 0, \ldots \rangle,$
$\quad$ where $\mathbf{Ut} = Obj'.Cell(\mathsf{OUT}).Orient.$

IDEAL POST-CONDITIONS

□ TRUE (there are no ideal conditions).

EXCEPTIONAL POST-CONDITIONS

□ If Obj has no cells, it remains so: $Obj' = Obj.$

□ If no element of sequence S identifies a cell of Obj, then Obj remains unchanged: $Obj' = Obj.$

Post-conditions for the unsupported function

□ FALSE (the object *Obj* remains unchanged).

Dj Fill

Input and output

Obj : *DjObject* (D̈jinn object).

Input

S : *DjLabelSet* (cell labels).

Output

C : *DjLabel* (cell label).

Description

This function creates a (possibly disconnected) bounded cell C with boundary points lying in cells of set S. The new cell C has a manifold dimension equal to the spatial dimension of the object *Obj*. Original cells that (partially) bound the new cell are subdivided into connected components if necessary to preserve the frontier condition, i.e. the boundary points of C are the union of other cells' point-sets. All cells of the original object are retained intact or subdivided in the modified object. Thus, some cells may bound two disconnected pieces of the new cell, or bound the new cell on each side without disconnecting it. For example, if *Obj* is three-dimensional, a three-dimensional solid is created; and if *Obj* is two-dimensional, a two-dimensional plane region is created. For instance, a solid sphere would be created from a spherical shell, and a triangular region from three intersecting line-segments.

Pre-conditions

□ TRUE (there are no pre-conditions).

Actual post-conditions

□ Object fields.

$$Obj'.ModCnt = Obj.ModCnt + 1.$$

$$Obj'.ModRegion = Obj.ModRegion \cup (Point\text{-}set(Obj') - Point\text{-}set(Obj)).$$

$$Obj'.Dim = Obj.Dim.$$

□ Fields of cell C.
$Domain(Obj'.Cell(C).Orient)$ = a maximal bounded point-set with boundary points in cells of S that satisfy:

$Domain(Obj'.Cell(C).Orient) =$
$\quad Point\text{-}set(Obj') - Point\text{-}set(Obj)$.
$Obj'.Cell(C).Orient = \langle 0, \ldots 0 \rangle$.
$Obj'.Cell(C).Dim = Obj.Dim$.
$Obj'.Cell(C).Atr = \emptyset$.
$Obj'.Cell(C).Parents = \{\langle \mathsf{OUT}, \mathsf{OUT} \rangle\}$.

- Fields of each cell L of Obj', with a point-set that bounds cell C.

  Frontier condition satisfied:

  $\partial Domain(Obj'.Cell(C).Orient) =$
  $\quad \cup_{i \in D} Domain(Obj'.Cell(i).Orient)$,
  $\quad\quad$ where $D \subset Domain(Obj'.Cell)$.
  $\exists H \in Domain(Obj.Cell) \bullet$
  $Domain(Obj'.Cell(H).Orient)) \supseteq$
  $\quad Domain(Obj'.Cell(L).Orient) \bullet$
  $Obj'.Cell(L).Orient = Obj.Cell(H).Orient$.
  $Obj'.Cell(L).Dim = Obj.Cell(H).Dim$.
  $Obj'.Cell(L).Atr = Obj.Cell(H).Atr$.
  $Obj'.Cell(L).Parents = \{\langle H, \mathsf{OUT} \rangle\}$.

- Fields of each cell L of Obj' with a point-set equal to that of a cell of Obj that does not intersect the closure of the point-set of cell C (this includes the region outside object Obj'):

  $Domain(Obj'.Cell(L).Orient) = Domain(Obj.Cell(L).Orient)$.
  $Obj'.Cell(L).Orient = Obj.Cell(L).Orient$.
  $Obj'.Cell(L).Dim = Obj.Cell(L).Dim$.
  $Obj'.Cell(L).Atr = Obj.Cell(L).Atr$.
  $Obj'.Cell(L).Parents = \{\langle L, \mathsf{OUT} \rangle\}$.

Ideal post-conditions

- TRUE (there are no ideal conditions).

Exceptional post-conditions

- If Obj has no cells, it remains so: $Obj' = Obj$.
- If no element of set S identifies a cell of Obj, Obj remains unchanged ($Obj' = Obj$).

Post-conditions for the unsupported function

- FALSE (the object Obj remains unchanged).

Part II
3
Managing cells

Cell ancestry

Dj Child Cells

INPUT

Obj	:	*DjObject*	(D̈jinn object).
L	:	*DjLabel*	(cell label).
B	:	*DjBoolean*.	

OUTPUT

S : *DjLabelSet* (set of cell labels)

DESCRIPTION

This function returns a set S of labels of cells derived from a cell with label L during the most recent object geometry modification (e.g. a set-operation or partitioning). L is a cell of the first operand of the modification function if B is TRUE, and the second if it is FALSE. If cell L disappeared during that operation, then S contains the single label OUT. If L is OUT, then S contains labels of all cells entirely derived from one operand.

PRE-CONDITIONS

□ TRUE (there are no pre-conditions).

ACTUAL POST-CONDITIONS

□ TRUE (there are no actual conditions).

IDEAL POST-CONDITIONS

□ Each element of S is a parent of L:

$$(M \in S) \Leftrightarrow$$
$$(\exists P \in Obj.Cell(M).Parents \bullet \exists N \in DjLabel \bullet$$
$$(B \Rightarrow (P = \langle L, N \rangle))$$
$$\wedge(\neg B \Rightarrow (P = \langle N, L \rangle))).$$

Cell L from the first object and cell N from the second are the parents of M.

Exceptional post-conditions

- If cell L did not exist in the appropriate operand immediately prior to the previous D̈jinn geometry modification (e.g. a set-operation or partitioning), then S is the empty set.
- If the object Obj has no cells, then S is the label OUT if cell L disappeared during the most recent cell geometry modification.

Post-conditions for the unsupported function

- S is the empty set.

Dj Parent Cells

Input

Obj	:	*DjObject*	(D̈jinn object).
L	:	*DjLabel*	(cell label).

Output

R, S	:	*DjLabelSequence*	(set of cell labels).

Description

This function returns sequences R and S of labels of cells from which cell L derives during the most recent object geometry modification. Corresponding elements of R and S correspond to cells of the first and second operands respectively that were combined to form cell L.

If the most recent operation had one object operand, then S contains only the label OUT. If object Obj was created by that operation, then R and S are empty. If L is OUT, then R and S contain labels of all cells destroyed by that operation.

Pre-conditions

- TRUE (there are no pre-conditions).

Actual post-conditions

- TRUE (there are no actual conditions).

Ideal post-conditions

□ Each output pair consists of parents of L:

$$\forall i \in \{1 \ldots \text{Cardinality}(S)\} \bullet \\ \langle R_i, S_i \rangle \in Obj.Cell(L).Parents.$$

Exceptional post-conditions

□ if no cell of object Obj has label L, R and S are the empty sequences.

Post-conditions for the unsupported function

□ S is the empty set.

Dj Modified Cells

Input

Obj : $DjObject$ (D̈jinn object).

Output

S : $DjLabelSet$ (Set of cell labels).

Description

This function computes all cells S of object Obj whose geometry was changed by the most recent geometry modification function.

Pre-conditions

□ TRUE (there are no pre-conditions).

Actual post-conditions

□ Each element of S is the label of a cell of object Obj.

□ Each cell of S was modified:

$$(L \in S) \Leftrightarrow \\ (Obj.Cell(L).Parents \neq \langle L, \mathsf{OUT} \rangle) \\ \wedge\ (Obj.Cell(L).Parents \neq \langle \mathsf{OUT}, L \rangle).$$

Ideal post-conditions

□ TRUE (there are no ideal conditions).

Exceptional post-conditions

□ If object Obj has no cells, S is empty.

Post-conditions for the unsupported function

□ S is empty.

Dj Modifying Cells

INPUT

Obj : *DjObject* (Djinn object).

OUTPUT

R, *S* : *DjLabelSequence* (sequences of Djinn cells).

DESCRIPTION

This function computes the sequences R and S of cells of the objects that were operands of the most recent geometry modification function. Cells of object *Obj* which were changed by that function were derived from the corresponding cells in R and S.

PRE-CONDITIONS

□ TRUE (there are no pre-conditions).

ACTUAL POST-CONDITIONS

□ Each element of R is the label of a cell of the first object which was an operand of the previous object geometry modification function.

□ Each element of S is the label of a cell of the second object which was an operand of the previous object geometry modification function.

□ Each cell of R and S was modified:

□ Each output pair consists of parents of a modified cell:

$$\forall \in \text{Cardinality}(S) \bullet$$
$$\exists\ L \in \mathit{Domain}(\mathit{Obj}.\mathit{Cell}) \bullet (R_i \neq L \neq S_i) \bullet$$
$$\langle R_i, S_i \rangle \in \mathit{Obj}.\mathit{Cell}(L)\mathit{Parents}.$$

IDEAL POST-CONDITIONS

□ TRUE (there are no ideal conditions).

EXCEPTIONAL POST-CONDITIONS

□ If object *Obj* has no cells, S is empty.

POST-CONDITIONS FOR THE UNSUPPORTED FUNCTION

□ Label set S is empty.

Label-set manipulation

Dj Add Cell to Label Set

INPUT

C : *DjLabel* (Djinn label).

Input and output

S : *DjLabelset* (set of Djinn labels).

Description

This function adds the label C to the label set S.

Pre-conditions

□ TRUE (there are no pre-conditions).

Actual post-conditions

□ $S = S \cup \{C\}$.

Ideal post-conditions

□ TRUE (there are no ideal conditions).

Exceptional post-conditions

□ TRUE (there are no exceptional input parameter values).

Post-conditions for the unsupported function

□ $S =$ *DjUnknown*.

Dj Label Set to Sequence

Input

R : *DjLabelSet* (set of Djinn labels).

Output

T : *DjLabelSequence* (sequence of Djinn labels).

Description

This function converts the set of labels R to the sequence T containing the same labels.

Pre-conditions

□ TRUE (there are no pre-conditions).

Actual post-conditions

□ Each element of R is contained once in T.

□ Each element of T is contained in R.

Ideal post-conditions

□ TRUE (there are no ideal conditions).

Exceptional post-conditions

□ T is empty if R is empty.

Post-conditions for the unsupported function

□ The sequence of labels T is empty.

Dj Label Sequence to Set

Input

R : *DjLabelSequence* (sequence of D̈jinn labels).

Output

T : *DjLabelSet* (set of D̈jinn labels).

Description

This function converts the sequence of labels R to the set T containing the same labels.

Pre-conditions

□ TRUE (there are no pre-conditions).

Actual post-conditions

□ Each element of R is contained once in T.

□ Each element of T is contained in R.

□ Elements of R that are duplicated in R appear only once in T.

Ideal post-conditions

□ TRUE (there are no ideal conditions).

Exceptional post-conditions

□ If R is empty, then the sequence of labels T is empty.

Post-conditions for the unsupported function

□ The set of labels T is empty.

Dj Unite Label Sets

Input

R, S : *DjLabelSet* (sets of D̈jinn labels).

Output

T : *DjLabelSet* (set of D̈jinn labels).

Description

This function unites sets of labels R and S to form set T.

Pre-conditions

□ TRUE (there are no pre-conditions).

Actual post-conditions

□ $T = R \cup S$.

Ideal post-conditions

□ TRUE (there are no ideal conditions).

Exceptional post-conditions

□ TRUE (there are no exceptional input parameter values).

Post-conditions for the unsupported function

□ The set of labels T is empty.

Dj Intersect Label Sets

Input

R, S : *DjLabelSet* (sets of Djinn labels).

Output

T : *DjLabelSet* (set of Djinn labels).

Description

This function creates set T from the common labels of sets R and S.

Pre-conditions

□ TRUE (there are no pre-conditions).

Actual post-conditions

□ $T = R \cap S$.

Ideal post-conditions

□ TRUE (there are no ideal conditions).

Exceptional post-conditions

□ TRUE (there are no exceptional input parameter values).

Post-conditions for the unsupported function

□ The set of labels T is empty.

Dj Subtract Label Sets

INPUT

R, S : *DjLabelSet* (sets of D̈jinn labels).

OUTPUT

T : *DjLabelSet* (set of D̈jinn labels).

DESCRIPTION

This function creates a set T from those labels of set R which are not in set S.

PRE-CONDITIONS

□ TRUE (there are no pre-conditions).

ACTUAL POST-CONDITIONS

□ $T = R - S$.

IDEAL POST-CONDITIONS

□ TRUE (there are no ideal conditions).

EXCEPTIONAL POST-CONDITIONS

□ TRUE (there are no exceptional input parameter values).

POST-CONDITIONS FOR THE UNSUPPORTED FUNCTION

□ The set of labels T is empty.

Dj Delete Label Set

INPUT AND OUTPUT

S : *DjLabelSet* (set of D̈jinn labels).

DESCRIPTION

This function removes all labels from set S.

PRE-CONDITIONS

□ TRUE (there are no pre-conditions).

ACTUAL POST-CONDITIONS

□ $S = \emptyset$.

IDEAL POST-CONDITIONS

□ TRUE (there are no ideal conditions).

Exceptional post-conditions

□ TRUE (there are no exceptional input parameter values).

Post-conditions for the unsupported function

□ The set of labels S remains unchanged.

Dj Delete Label Set

Input and output

S : *DjLabelSequence* (sequence of D̈jinn labels).

Description

This function removes all labels from sequence S.

Pre-conditions

□ TRUE (there are no pre-conditions).

Actual post-conditions

□ S is an empty label sequence.

Ideal post-conditions

□ TRUE (there are no ideal conditions).

Exceptional post-conditions

□ TRUE (there are no exceptional input parameter values).

Post-conditions for the unsupported function

□ The sequence of labels S remains unchanged.

Dj Get Label

Input

S : *DjLabelSequence* (sequence of D̈jinn labels).
I : *DjInteger*.

Output

L : *DjLabel* (D̈jinn cell label).
T : *DjLogic* (Ith label exists).

Description

This function returns the Ith label L of sequence S.

Pre-conditions

□ I is strictly positive ($I > 0$).

Actual post-conditions

- If there are at least I labels in sequence S, then $T = DjTrue$, otherwise $T = DjFalse$.
- L is the Ith label of sequence S.

Ideal post-conditions

- TRUE (there are no ideal conditions).

Exceptional post-conditions

- TRUE (there are no exceptional input parameter values).

Post-conditions for the unsupported function

- $T = DjUnknown$.

Dj Get Cardinality

Input

S : *DjLabelSet* (set of Djinn labels).

Output

N : *DjInteger* (cardinality of the set S).

Description

This function returns the number N of labels in set S.

Pre-conditions

- TRUE (there are no pre-conditions).

Actual post-conditions

- $N = Cardinality(S)$.

Ideal post-conditions

- TRUE (there are no ideal conditions).

Exceptional post-conditions

- TRUE (there are no exceptional input parameter values).

Post-conditions for the unsupported function

- TRUE (there are no unsupported postconditions).

Attribute manipulation

Dj Set Attribute

INPUT AND OUTPUT

Obj : *DjObject* (Djinn object).

INPUT

L : *DjLabel* (cell label).
T : *DjString* (attribute type).
S : *DjInteger* (attribute value).

DESCRIPTION

This function assigns the value S to attribute T of cell L of object *Obj*. If S is 0, then the attribute of type T is removed from cell L. The string T may not be empty.

PRE-CONDITIONS

- Attribute type T is not the empty string.

ACTUAL POST-CONDITIONS

- If $S \neq 0$, then:
  - If necessary an attribute of type T is assigned to cell L:
    $Domain(Obj'.Cell(L).Atr) =$
    $T \cup Domain(Obj.Cell(L).Atr)$.
  - Cell L's attribute of type T is assigned value S:
    $Obj'.Cell(L).Atr(T) = S$.
  - Attributes of types other than T, and non-attribute fields of *Obj*, remain unchanged.
- If $S = 0$, then:
  - Remove the attribute of type T from cell L:
    $Domain(Obj'.Cell(L).Atr) =$
    $Domain(Obj.Cell(L).Atr) - T$.

IDEAL POST-CONDITIONS

- TRUE (there are no ideal conditions).

EXCEPTIONAL POST-CONDITIONS

- TRUE (The function has no effect if
  No cell of object *Obj* has label L, or if
  The object *Obj* has no cells.)

POST-CONDITIONS FOR THE UNSUPPORTED FUNCTION

- TRUE (the function has no effect).

Dj Get Attribute

INPUT

Obj	:	*DjObject*	(D̈jinn object).
L	:	*DjLabel*	(cell label).
T	:	*DjString*	(attribute type).

OUTPUT

S	:	*DjInteger*	(attribute).

DESCRIPTION

This function returns the value S of attribute T of cell L of object Obj, or zero. If $L = \mathsf{OUT}$, then the attribute of the region outside the object is returned.

PRE-CONDITIONS

- □ Attribute type T is not the empty string.

ACTUAL POST-CONDITIONS

- □ If an attribute of type T exists, then S is attribute of type T of cell L:
  $(T \in \mathit{Domain}(\mathit{Obj}\,\mathit{Cell}\,(L).\mathit{Atr}) \Rightarrow$
  $(S = \mathit{Obj}\,.\mathit{Cell}\,(L).\mathit{Atr}\,(T))$.
- □ If an attribute of type T does not exist, then S is zero:
  $(T \notin \mathit{Domain}(\mathit{Obj}\,.\mathit{Cell}\,.\mathit{Atr}\,)) \Rightarrow (S = 0)$.

IDEAL POST-CONDITIONS

- □ TRUE (there are no ideal conditions).

EXCEPTIONAL POST-CONDITIONS

- □ S is the null attribute zero, if no cell of object Obj has label L.

POST-CONDITIONS FOR THE UNSUPPORTED FUNCTION

- □ S is the null attribute zero.

Dj Get Attribute Types

INPUT

Obj	:	*DjObject*	(D̈jinn object).

OUTPUT

S	:	Set of *DjString*	(attribute types).

DESCRIPTION

This function returns all attribute types S of object Obj.

Pre-conditions

□ TRUE (there are no pre-conditions).

Actual post-conditions

□ $(T \in S) \Leftrightarrow$
$(\exists m \in \mathit{Domain}(\mathit{Obj}.\mathit{Cell}) \bullet T \in \mathit{Domain}(\mathit{Obj}.\mathit{Cell}(m).\mathit{Atr})$.

Ideal post-conditions

□ TRUE (there are no ideal conditions).

Exceptional post-conditions

□ If *Obj* has no cells, S is empty.

Post-conditions for the unsupported function

□ The set S is empty.

Dj Delete Attribute Type

Input and output

Obj : *DjObject* (Djinn object).

Input

T : *DjString* (Attribute type).

Description

This function removes all attributes of type T from object *Obj*. The attribute of type T can be removed from a single cell using the function *Dj Set Attribute* to assign the null attribute zero.

Pre-conditions

□ TRUE (there are no pre-conditions).

Actual post-conditions

□ No cell L of object *Obj* has an attribute of type T:
$\forall L \in \mathit{Domain}(\mathit{Obj}'.\mathit{Cell}) \bullet T \notin \mathit{Domain}(\mathit{Obj}'.\mathit{Cell}(L).\mathit{Atr})$.

□ All non-attribute fields of *Obj* remain unchanged.

Ideal post-conditions

□ TRUE (there are no ideal conditions).

Exceptional post-conditions

□ TRUE (The function has no effect if
No cell of object *Obj* has label L, or if
Cell L does not have an attribute of type T, or if
Object *Obj* has no cells.)

Post-conditions for the unsupported function

□ The function has no effect.

Cell selection

Dj Small Cells

Input

Obj	:	*DjObject*	(Djinn object).
d	:	*DjReal*	(length).
K	:	Set of *DjInteger*	(dimension).

Output

S	:	*DjLabelSet*	(set of Djinn cell labels).

Description

This function selects all cells S of object Obj whose dimension is in the set K, and which are smaller than length d. If d is sufficiently large, then all cells with a dimension in K are selected.

Pre-conditions

□ Each element of K is in $[0 \dots 3]$.

Actual post-conditions

□ Each element of S is the label of a cell of object Obj.

□ The dimension of each cell of S is in K:

$$(L \in S) \Leftrightarrow (Obj.Cell(L).Dim \in K).$$

Ideal post-conditions

□ The volume of each solid cell in S is less than d^3.

□ The area of each face cell in S is less than d^2.

□ The length of each edge cell in S is less than d.

□ No solid cell of Obj, which is not in S, has a volume less than d^3.

□ No face cell of Obj, which is not in S, has an area less than d^2.

□ No edge cell of Obj, which is not in S, has a length less than d.

Exceptional post-conditions

□ If object Obj has no cells, then the set S is empty.

Post-conditions for the unsupported function

□ The set S is empty.

Dj Point Proximate Cells

INPUT

Obj	:	*DjObject*	(D̈jinn object).
p	:	*DjPoint*	(D̈jinn point).
K	:	Set of *DjInteger*	(dimension).
tol	:	*DjReal*	(tolerance).

OUTPUT

S	:	*DjLabelSequence*	(sequence of D̈jinn labels).
q	:	*DjPoint*	(D̈jinn point).

DESCRIPTION

This function computes the cells S that are closest to point **p** whose dimension is in the set K. Point **q** is the point in the first cell of S closest to point **p**. The distance of **p** to each cell of S is greater than the distance between **p** and **q** and within *tol* of it. When a closest cell is a frontier of another cell, only the former is included in S, unless all points of the latter are within tolerance.

PRE-CONDITIONS

- ☐ Each element of K is in $[0 \ldots 3]$.
- ☐ $tol \geq 0$.

ACTUAL POST-CONDITIONS

- ☐ Each element of S is the label of a cell of object *Obj*.
- ☐ The dimension of each cell of S is in K:

  $(L \in S) \Leftrightarrow (Obj.Cell(L).Dim \in K)$.

IDEAL POST-CONDITIONS

- ☐ Point **q** is in the point-set of the first cell of S.
- ☐ No point r in the point-set of a cell of object *Obj* is closer to point **p** than point **q**: $||p - r|| \geq ||p - q||$.
- ☐ There is a point **r** in the point-set of each cell in S that is within tolerance:

  $||\mathbf{p} - \mathbf{r}|| \leq ||\mathbf{p} - \mathbf{q}|| + tol$.
- ☐ If two distinct cells are equidistant from point **p** and have the same closest point, i.e. one bounds the other, only the cell of lower dimension is in S, unless all points of the other cell are within tolerance.

EXCEPTIONAL POST-CONDITIONS

- ☐ If object *Obj* has no cells, S is empty.

- □ If each cell of object *Obj* has an empty point-set, then S contains the singleton OUT, i.e. the label of the object exterior, and **q** is an indeterminate point.
- □ If there is a multiple finite number of closest points in the same cell, then S contains repeated cell labels.
- □ If there are an infinite number of closest points in the first cell of S (e.g. when **p** is the centre of a spherical surface cell), then **q** is indeterminate.

Post-conditions for the unsupported function

- □ S is empty.

Dj Line Proximate Cells

Input

Obj	:	*DjObject*	(Djinn object).
pt, **qt**	:	*DjPoint*	(straight line).
K	:	Set of *DjInteger*	(dimension).
tol	:	*DjReal*	(tolerance).

Output

S	:	*DjLabelSequence*	(sequence of Djinn cell labels).
q	:	*DjPoint*	(Djinn point).

Description

This function computes the cells S that are closest to a point on the unbounded line through points **pt** and **qt**, and whose dimension d is in set K. The point **q** is a closest point in the first cell of S. The distance from the line to each cell of S is greater than the distance from the line to **q** and within *tol* of it. When a closest cell is a frontier of another cell, only the former is included in S, unless all points of the latter are within tolerance. Thus, for example, if the line penetrates a solid, only its surface cells usually appear in S.

Pre-conditions

- □ Each element of K is in the domain $[0 \ldots 3]$.
- □ $tol \geq 0$.

Actual post-conditions

- □ Each element of S is the label of a cell of object *Obj*.
- □ The dimension of each cell of S is in K:

$$(L \in S) \Leftrightarrow (Obj.Cell(L).Dim \in K).$$

Ideal post-conditions

- □ Point **q** is in the point-set of the first cell of S.
- □ No point in the point-set of a cell of object *Obj* is closer to the unbounded line **pt** ... **qt** than point **q**.
- □ There is a point **r** in the point-set of each cell in S that is within tolerance:

$$||(\mathbf{qt} - \mathbf{pt}) \times (\mathbf{q} - \mathbf{pt})|| \leq ||(\mathbf{qt} - \mathbf{pt}) \times (\mathbf{r} - \mathbf{pt})|| + tol||\mathbf{qt} - \mathbf{pt}||.$$

- □ If two distinct cells are equidistant from the unbounded line **pt** ... **qt** and have the same closest point(s), i.e. one bounds the other, only the cell of lower dimension is in S, unless all points of the other cell are within tolerance.

Exceptional post-conditions

- □ If object *Obj* has no cells, then S is empty.
- □ If each cell of object *Obj* has an empty point-set, then S contains the singleton OUT, i.e. the label of the object exterior, and **q** is an indeterminate point.
- □ If there are a multiple finite number of closest points in the same cell, then S contains repeated cell labels.
- □ If there are an infinite number of closest points in the same cell, then **q** is indeterminate.

Post-conditions for the unsupported function

- □ S is empty.

Dj Plane Proximate Cells

Input

Obj	:	*DjObject*	(Djinn object).
Pln	:	*DjPlane*	(half-space).
K	:	Set of *DjInteger*	(dimension).
tol	:	*DjReal*	(tolerance).

Output

S	:	*DjLabelSequence*	(sequence of Djinn labels).
q	:	*DjPoint*	(Djinn point).

Description

This function computes the cells S that are closest to a point on the boundary of the planar half-space **Pln**. Point **q** is the closest point in the first cell of S. The distance of the plane to each cell of S is greater than the distance between **p** and **q** and within *tol* of it. When a closest cell is a frontier of another cell, only the former is included in S, unless all

points of the latter are within tolerance. Thus, for example, if the plane intersects a solid, only its edge cells usually appear in S.

Pre-conditions

- ☐ Each element of K is in the domain $[0 \ldots 3]$.
- ☐ $tol \geq 0$.

Actual post-conditions

- ☐ Each element of S is the label of a cell of object Obj.
- ☐ The dimension of each cell of S is in K:
  $(L \in S) \Leftrightarrow (Obj.Cell(L).Dim \in K)$.

Ideal post-conditions

- ☐ Point **q** is in the point-set of the first cell of S.
- ☐ No point in the point-set of a cell of object Obj is closer to the unbounded plane **Pln** than point **q**.
- ☐ There is a point **r** in the point-set of each cell in S that is within tolerance:
  $||N.(\mathbf{q} - \mathbf{pt})|| \leq ||N.(\mathbf{r} - \mathbf{pt})|| + tol||N||$,
  where **pt** is a point on the plane and N is its surface normal.
- ☐ If two distinct cells are equidistant from the unbounded plane **Pln** and have the same closest point(s), i.e. one bounds the other, then only the cell of lower dimension is in S, unless all points of the other cell are within tolerance.

Exceptional post-conditions

- ☐ If object Obj has no cells, S is empty.
- ☐ If each cell of object Obj has an empty point-set, then S is the singleton OUT, i.e. the label of the object exterior, and **q** is an indeterminate point.
- ☐ If there are a multiple finite number of closest points in the same cell, then S contains repeated elements.
- ☐ If there are an infinite number of closest points in the same cell, then **q** is indeterminate.

Post-conditions for the unsupported function

- ☐ S is empty.

Dj Get Boundary Cells

Input

Obj	:	*DjObject*	(Djinn object).
C	:	*DjLabel*	(cell label).

OUTPUT

S : *DjLabelSet* (set of D̈jinn cell labels).

DESCRIPTION

This function computes the set of labels S of cells that bound cell C of object Obj. However, bounding cells are excluded from S if they bound other bounding cells. For example, if Obj is a solid cube with its faces and edges represented as distinct cells, and C is its interior, then only the faces are returned in S. If C is OUT, it identifies the object exterior and S contains boundary cells of object Obj.

PRE-CONDITIONS

□ TRUE (there are no pre-conditions).

ACTUAL POST-CONDITIONS

□ Each element of S is the label of a cell of Obj.

IDEAL POST-CONDITIONS

□ If C identifies a cell of object Obj, then each element of S identifies a cell with a point-set in the frontier of the point-set of cell C.

□ If C is OUT, then each element of S identifies a cell with a point-set in the boundary of the object Obj.

□ No element of S identifies a cell with a point-set in the frontier of the point-set of another cell identified in S.

EXCEPTIONAL POST-CONDITIONS

□ S is empty if one of the following conditions is met:
  - Cell C is bounded by no other cell.
  - $C \neq$ OUT and cell C does not exist in object Obj.
  - the point-set of cell C is empty.
  - object Obj has no cells.

POST-CONDITIONS FOR THE UNSUPPORTED FUNCTION

□ S is empty.

Dj Get Bounded Cells

INPUT

Obj : *DjObject* (D̈jinn object).
C : *DjLabel* (cell label).

OUTPUT

S : *DjLabelSet* (set of cell labels).

DESCRIPTION

This function computes the set of labels S of cells that are (partially) bounded by cell C of object Obj. However, bounded cells are excluded from S if they are bounded by other bounded cells. For example, if Obj is a solid cube with its faces and edges represented as distinct cells, and C is an edge, then only the adjacent faces are returned in S, but *not* the cube interior. If C is a boundary cell of object Obj, then S includes OUT to represent the object exterior.

PRE-CONDITIONS

- TRUE (there are no pre-conditions).

ACTUAL POST-CONDITIONS

- Each element of S is either OUT or the label of a cell of Obj.

IDEAL POST-CONDITIONS

- Each element of S identifies a cell (possibly the object exterior) with a point-set whose frontier contains the point-set of cell C.
- No element of S identifies a cell with a point-set, whose frontier contains the point-set of another cell identified in S.

EXCEPTIONAL POST-CONDITIONS

- S is empty if one of the following conditions is met:
  - Cell C bounds no other cell nor the object exterior.
  - Cell C does not exist in object Obj.
  - the point-set of cell C is empty.
  - object Obj has no cells.

POST-CONDITIONS FOR THE UNSUPPORTED FUNCTION

- S is empty.

Dj Get Orientation

INPUT

Obj	:	*DjObject*	(Djinn object).
C, D	:	*DjLabel*	(cell identifiers).

OUTPUT

R	:	*DjLogic*	(orientation).

DESCRIPTION

This function determines whether the orientation of a cell C is consistent with that of cell D.

Pre-conditions

- TRUE (there are no preconditions).

Ideal post-conditions

- If D is a frontier cell of cell C, then let $\mathbf{u}$ and $\mathbf{v}$ be the normal (tangent) vectors of face (edge) cells C and D respectively, at an arbitrary common point $\mathbf{p}$, such that those vectors are compatible with the cell orientations.
- If cell C is an edge and cell D is the singleton end-point $\mathbf{p}$, then R is
  DjTrue, if vector $\mathbf{u}$ at point $\mathbf{p}$ is directed away from $\mathbf{p}$ and cell D is a source vertex, or $\mathbf{u}$ is directed toward a sink vertex.
  DjFalse, if the previous condition is not satisfied.
- If cell C is a face and cell D is an edge, then R is
  DjTrue, if the cross product $\mathbf{u} \times \mathbf{v}$ at all common points is directed into the interior of cell C.
  DjFalse, if the cross product $\mathbf{u} \times \mathbf{v}$ at all common points is directed to the exterior of cell C.
- If cell C is a solid and cell D is a face, then R is
  DjTrue, if $\mathbf{v}$ at all common points is directed to the exterior of C.
  DjFalse, if $\mathbf{v}$ at all common points is directed to the interior of C.

Exceptional post-conditions

- R is *DjUnknown* if any of the following conditions are met:
  - Cell C or cell D does not exist in object *Obj*.
  - Cell D does not bound cell C, i.e. the point-set of D is not in the frontier of the point-set of cell C.
  - The dimension of cell D is not one less than that of cell C.
  - The point-set of cell C or D is empty.
  - The object *Obj* has no cells.

Post-conditions for the unsupported function

- The orientation R is *DjUnknown*.

Part II
4
Transformations

Dj Concatenate Transformations

INPUT

R, **S** : *DjTransformation* (D̈jinn transformations).

OUTPUT

T : *DjTransformation* (D̈jinn transformation).

DESCRIPTION

This function creates the parameterized transformation $\mathbf{T}(k)$ which has an effect equivalent to that of the sequential application of transformations $\mathbf{R}(k)$ and $\mathbf{S}(k)$. If **R** has a lower dimension than **S** (or vice versa), it affects only the first coordinates of points with the dimension of **S**, and **T** has the higher dimension.

PRE-CONDITIONS

□ TRUE (there are no pre-conditions).

ACTUAL POST-CONDITIONS

□ Let a, b and c be the spatial dimensions of transformations **R**, **S** and **T** respectively.

□ Transformation **T** has dimension $c = \max(a, b)$.

□ The effect $\mathbf{T}(k)(\mathbf{p})$ of transformation **T** is equivalent to the sequential application of the transformations $\mathbf{R}(k)$ and $\mathbf{S}(k)$ respectively:

$$\forall \mathbf{p} = \langle x_1, \ldots x_c \rangle \in \mathcal{R}^c \bullet \forall k \in [0 \ldots 1] \bullet \exists \mathbf{R}', \mathbf{S}' \in (\mathcal{R}^c \to \mathcal{R}^c) \bullet$$
$$(\mathbf{R}'(\mathbf{p}) = \langle \mathbf{R}(\langle x_1, \ldots x_a \rangle), x_{a+1} \ldots x_c \rangle) \wedge$$
$$(\mathbf{S}'(\mathbf{p}) = \langle \mathbf{S}(\langle x_1, \ldots x_b \rangle), x_{b+1} \ldots x_c \rangle) \wedge$$
$$(\mathbf{S}'(k)(\mathbf{R}'(k)(\mathbf{p})) = \mathbf{T}(k)(\mathbf{p})).$$

Ideal post-conditions

□ TRUE (there are no ideal conditions).

Exceptional post-conditions

□ If either transformation **R** or transformation **S** is indeterminate, **T** is indeterminate.

Post-conditions for the unsupported function

□ Transformation **T** is indeterminate.

Dj Create Linear Transformation

Input

n : *DjInteger* (spatial dimension).
M : *DjMatrix* (homogeneous linear transformation matrix).

Output

T : *DjTransformation* (Djinn transformation).

Description

This function creates the n-dimensional linear transformation **T** which has net effect equivalent to that of the homogeneous coordinate transformation matrix **M**. **T** is parameterized over $[0 \ldots 1]$ to a continuous sequence of transformations equivalent to a matrix sequence from the identity to **M**.

Pre-conditions

□ The matrix **M** has at least $n + 1$ rows and $n + 1$ columns.

□ $1 \leq n \leq 3$.

Actual post-conditions

□ The transformation **T** is a map $[0 \ldots] \to (\mathcal{R}^n \to \mathcal{R}^n)$.

□ The effect $\mathbf{T}(k \in [0 \ldots 1])(\mathbf{p})$ of the linear transformation **T** on n-dimensional points **p** is equivalent to the post-multiplication of their homogeneous coordinates by the upper left $n + 1$ square sub-matrix $\mathbf{M}'$ of matrix **M**:

$$\forall \mathbf{p} = \langle x_1, \ldots x_n \rangle \in \mathcal{R}^n \bullet \forall k \in [0 \ldots 1] \bullet \exists w \in \mathcal{R} \bullet$$
$$(\langle 1, x_1, \ldots x_n \rangle (I(1-k) + \mathbf{M}'k) = w \langle 1, \mathbf{T}(k)(\mathbf{p}) \rangle.$$

Ideal post-conditions

□ TRUE (there are no ideal conditions).

Exceptional post-conditions

□ FALSE (there are no exceptional input parameter values).

Post-conditions for the unsupported function

□ Transformation **T** is indeterminate.

Dj Create Quadratic Transformation

Input

n	:	*DjInteger*	(dimension).
$\mathbf{Q}_0 \ldots \mathbf{Q}_n$	:	*DjMatrix*	(quadratic transformation matrices).

Output

T	:	*DjTransformation*	(D$\ddot{\text{j}}$inn transformation).

Description

This function creates the n-dimensional quadratic transformation **T** whose effect on an arbitrary point is equivalent to generating the homogeneous point coordinates of the result using the quadratic forms $\mathbf{q}\mathbf{Q}_i\mathbf{q}^{\mathrm{T}}$, $0 \leq i \leq n$, where **q** are the homogeneous point coordinates. The transformation **T** is parameterized over $[0 \ldots 1]$ to a continuous sequence of transformations equivalent to a matrix sequence from the Identity to **Q**.

Pre-conditions

□ $1 \leq n \leq 3$.

□ Each matrix $\mathbf{Q}_i$ has at least $n+1$ rows and $n+1$ columns.

Actual post-conditions

□ The transformation **T** is a map $\mathcal{R}^n \rightarrow \mathcal{R}^n$.

□ **T** is the quadratic transformation given by:

$$\begin{aligned}
&\forall \mathbf{p} = \langle x_1, \ldots x_n \rangle \in \mathcal{R}^n \bullet \forall k \in [0 \ldots 1] \bullet \exists \langle c_0, \ldots c_n \rangle \in \mathcal{R}^{n+1} \bullet \\
&\quad \exists w \in \mathcal{R} \bullet \forall i \in [0 \ldots n] \bullet \\
&\qquad (c_i = \langle 1, x_1, \ldots x_n \rangle (I(1-k) + kQ'_i) \langle 1, x_1, \ldots x_n \rangle^{\mathrm{T}}) \wedge \\
&\qquad (\langle c_0, \ldots c_n \rangle = w \langle 1, T(k)(\mathbf{p}) \rangle),
\end{aligned}$$

where Q'_i is the upper left $n+1$ square sub-matrix of $\mathbf{Q}_i$.

Ideal post-conditions

□ TRUE (there are no ideal conditions).

Exceptional post-conditions

□ FALSE (there are no exceptional input parameter values).

Post-conditions for the unsupported function

□ Transformation **T** is indeterminate.

Dj Get Linear Transformation

INPUT

T : *DjTransformation* (D̈jinn transformation).

OUTPUT

n : *DjInteger* (spatial dimension).
Q : *DjMatrix* (homogeneous linear transformation matrix).

DESCRIPTION

This function creates a matrix **Q** of size $(n+1) \times (n+1)$ which, when applied in the sense of conventional computer graphics techniques, has the same effect on a point as $\mathbf{T}(1)$.

PRE-CONDITIONS

□ **T** must be either a linear transformation or a concatenation of linear transformations.

ACTUAL POST-CONDITIONS

□ n is the dimension of **T**.
□ **Q** is an $(n+1) \times (n+1)$ matrix.
□ $\forall \mathbf{p} \in R^n \bullet \exists w \in R \bullet \langle 1, \mathbf{p} \rangle \mathbf{Q} = w \langle 1, \mathbf{T}(1)(\mathbf{p}) \rangle$.

IDEAL POST-CONDITIONS

□ TRUE (there are no ideal conditions).

EXCEPTIONAL POST-CONDITIONS

□ $n = 0$.
□ **Q** is indeterminate.

POST-CONDITIONS FOR THE UNSUPPORTED FUNCTION

□ $n = 0$.
□ **Q** is indeterminate.

Dj Get Quadratic Transformation

INPUT

T : *DjTransformation* (D̈jinn transformation).

OUTPUT

n : *DjInteger* (spatial dimension).
$Q_0 \ldots Q_n$: *DjMatrix* (homogeneous linear transformation matrix).

DESCRIPTION

This function creates $n+1$ matrices $Q_0 \ldots Q_n$ of size $(n+1) \times (n+1)$ which, when applied as a quadratic transformation, have the same effect on the coordinates of any point as $\mathbf{T}(1)$.

PRE-CONDITIONS

- $\mathbf{T}$ is either a linear or a quadratic transformation, or a concatenation of a quadratic transformation (i.e. defined by *Dj Create Quadratic Transformation*, by *Dj Create Bend*, or by *Dj Create Taper*) with linear transformations.

ACTUAL POST-CONDITIONS

- n is the dimension of $\mathbf{T}$.
- each $\mathbf{Q}_i$ is an $(n+1) \times (n+1)$ matrix.
- $\forall \mathbf{p} \in R^n \bullet \exists w \in R \bullet \exists \langle c_0, \ldots c_n \rangle \in R^{n+1} \bullet$
  $(c_i = \langle 1, \mathbf{p} \rangle \mathbf{Q}_i \langle 1, \mathbf{p} \rangle^{\mathrm{T}}) \wedge (\langle c_0, \ldots c_n \rangle = w \langle 1, \mathbf{T}(1)(\mathbf{p}) \rangle)$.

IDEAL POST-CONDITIONS

- TRUE (there are no ideal conditions).

EXCEPTIONAL POST-CONDITIONS

- $n = 0$.
- $\mathbf{Q}_i$ is indeterminate.

POST-CONDITIONS FOR THE UNSUPPORTED FUNCTION

- $n = 0$.
- $\mathbf{Q}_i$ is indeterminate.

Dj Apply Transformation

INPUT

$\mathbf{T}$: *DjTransformation* (Djinn transformation).

INPUT AND OUTPUT

Obj : *DjObject* (Djinn object).

DESCRIPTION

This function uses the m-dimensional transformation $\mathbf{T}$ to change the point-sets of object *Obj* and all its cells; only the first m Euclidean point coordinates are affected. If the spatial dimension of *Obj* is less than m, it is increased. Bounding relations between cells and cell manifold dimension are unaffected by the transformation.

Pre-conditions

□ TRUE (there are no pre-conditions).

Actual post-conditions

□ Object fields.

$Obj'.ModCnt = Obj.ModCnt + 1.$

$Obj'.ModRegion = Point\text{-}set(Obj) \cup Point\text{-}set(Obj').$

$Obj'.Dim = \max(Dim(T), Obj.Dim).$

$Domain(Obj'.Cell) = Domain(Obj.Cell).$

□ Fields of each cell L (including the region $Obj'.Cell(\mathsf{OUT})$ outside the object).

$Obj'.Cell(L).Atr = Obj.Cell(L).Atr.$

$Obj'.Cell(L).Parents = \{\langle L, \mathsf{OUT}\rangle\}.$

$Obj'.Cell(L).Dim = Obj.Cell(L).Dim.$

□ Point-set.

$(\mathbf{p} \in Domain(Obj.Cell(L).Orient) \Leftrightarrow$
$(T'(\mathbf{p}) \in Domain(Obj'.Cell(L).Orient),$
where $\mathbf{T}' = \mathbf{T}(1)$, if $\mathbf{T}(1) \in \mathcal{R}^m \rightarrow \mathcal{R}^m$ and $m = Obj.Dim.$

If $m > Obj.Dim$, the points of Obj are appended, with 0 coordinates and $\mathbf{T}' = \mathbf{T}(1)$.

If $m < Obj.Dim$, $\mathbf{T}'$ has the same effect as $\mathbf{T}(1)$ on the first m coordinates, and no effect on the remainder.

□ $\mathbf{T}'$ also transforms the orientation $Obj.Cell(\mathrm{L}).Orient.$

Ideal post-conditions

□ TRUE (there are no ideal conditions).

Exceptional post-conditions

□ The dimension of cells is invariant under transformation. Thus, if transformation $\mathbf{T}$ causes a reduction in dimension, e.g. the projection of a three-dimensional object on to a plane, object Obj becomes the null object with no cells:

$Obj'.ModCnt = Obj.ModCnt + 1.$

$Obj'.ModRegion = Point\text{-}set(Obj) \cup Point\text{-}set(Obj').$

$Obj'.Dim = \max(Dim(\mathbf{T}), Obj.Dim).$

$Obj'.Cell = \emptyset.$

□ If transformation $\mathbf{T}$ is not 1-to-1 in the domain of the point-set of object Obj, the result of the transformation is a self-intersecting object. Cells in the region of overlap are created in the same way as cells in the union of two distinct intersecting objects. Thus, the point-set

of each cell in the region of overlap is the maximal set that contains image points of the same two cells of the original object *Obj*. Both original cells are parents of the new cell.

- □ If transformation **T** maps an interior point of a cell to infinity, the result of the transformation is two unbounded pieces.
- □ If *Obj* has no cells, it remains so.

Post-conditions for the unsupported function

- □ Object *Obj* remains unchanged.

Dj Invert Transformation

Input

R : *DjTransformation* (D̈jinn transformation).

Output

T : *DjTransformation* (D̈jinn transformation).

Description

This function creates the parameterized transformation **T** with an effect that is the inverse of the effect of transformation **R**.

Pre-conditions

- □ TRUE (there are no pre-conditions).

Actual post-conditions

- □ If $\mathbf{R}(k) \in [0 \ldots 1]$ is a map $\mathcal{R}^m \rightarrow \mathcal{R}^m$, so is **T**.
- □ $\forall \mathbf{p} \in \mathcal{R}^m \bullet \forall k \in [0 \ldots 1] \bullet \mathbf{p} = \mathbf{R}(k)(\mathbf{T}(k)(\mathbf{p}))$.

Ideal post-conditions

- □ TRUE (there are no ideal conditions).

Exceptional post-conditions

- □ If transformation **R** is not 1-to-1, e.g. a projection to a lower dimension, then **T** is indeterminate.
- □ If **R** is indeterminate, then **T** is indeterminate.

Post-conditions for the unsupported function

- □ Transformation **T** is indeterminate.

Rigid-body transformations

Dj Create Translation

INPUT

u, **v** : *DjPoint* (displacement).

OUTPUT

T : *DjTransformation* (D̈jinn transformation).

DESCRIPTION

This function creates the parameterized linear transformation **T** with an effect on n-dimensional points that is equivalent to their translation by part of the vector from the n-dimensional point **u** to the n-dimensional point **v**.

PRE-CONDITIONS

□ TRUE (there are no pre-conditions).

ACTUAL POST-CONDITIONS

□ If n is the maximum dimension of the two points **u** and **v**, then the point of lower dimension is augmented with zero coordinates to give n-dimensional points $\mathbf{u}'$ and $\mathbf{v}'$ and:

$$\forall \mathbf{p} \in \mathcal{R}^n \bullet \forall k \in [0 \ldots 1] \bullet \\ \mathbf{p} + k(\mathit{Point}(\mathbf{v}') - \mathit{Point}(\mathbf{u}')) = T(k)(\mathbf{p}).$$

IDEAL POST-CONDITIONS

□ TRUE (there are no ideal conditions).

EXCEPTIONAL POST-CONDITIONS

□ If either point **u** or point **v** is indeterminate, then **T** is indeterminate.

POST-CONDITIONS FOR THE UNSUPPORTED FUNCTION

□ Transformation **T** is indeterminate.

Dj Create Axial Rotation

INPUT

n, i, j	:	*DjInteger*	(dimension and rotation plane).
θ	:	*DjReal*	(angle in radians).

OUTPUT

Output **T** : *DjTransformation* (D̈jinn transformation).

Description

This function creates the parameterized n-dimensional transformation $\mathbf{T}$ to rotate points about a line through the origin perpendicular to coordinate axes i and j. Small positive values of θ correspond to rotations of a vector, originally lying along axis i, towards axis j.

Pre-conditions

- $1 < n \leq 3$.
- $0 \leq i < j \leq n$.

Actual post-conditions

- The effect $\mathbf{T}(\mathrm{k})(\mathbf{p})$ of linear transformation $\mathbf{T}(k \in [0 \ldots 1]) \in \mathcal{R}^n \rightarrow \mathcal{R}^n$ on n-dimensional points $\mathbf{p}$ is equivalent to their rotation by $k\theta$ radians about a line through the origin perpendicular to coordinate axes i and j.

Ideal post-conditions

- TRUE (there are no ideal conditions).

Exceptional post-conditions

- FALSE (there are no exceptional input parameter values).

Post-conditions for the unsupported function

- Transformation $\mathbf{T}$ is indeterminate.

Dj Create Alignment

Input

$\mathbf{p}$, $\mathbf{q}$: *DjPoint* (D̈jinn points).

Output

$\mathbf{T}$: *DjTransformation* (D̈jinn transformation).

Description

This function creates the parameterized rotation $\mathbf{T}(k)$. $\mathbf{T}(1)$ rotates the vector in the direction of point $\mathbf{p}$ to that in the direction of point $\mathbf{q}$. The parameter k is proportional to the angle of rotation. The points need not be the same distance from the origin and either or both points may be infinite.

Pre-conditions

- TRUE (there are no pre-conditions).

Actual post-conditions

- □ $\mathbf{T}(k \in [0 \ldots 1])$ is an n-dimensional linear map $\mathcal{R}^n \rightarrow \mathcal{R}^n$, where n is the maximum dimension of points **p** and **q**.
- □ Transformation $\mathbf{T}(k)$ rotates about the origin in the plane that contains the origin and points $\mathbf{p}'$ and $\mathbf{q}'$, which are points **p** and **q** increased to the maximum of their dimensions with zero coordinate values.
- □ Transformation $\mathbf{T}(1)$ maps the unit vector from the origin in the direction of point **p** to the unit vector from the origin in the direction of point **q**.
- □ If transformation $\mathbf{T}(1)$ rotates by an angle q, then $\mathbf{T}(k)$ rotates by an angle kq.

Ideal post-conditions

- □ TRUE (there are no ideal conditions).

Exceptional post-conditions

- □ If the unit vectors from the origin in the direction of points **p** and **q** are equal, then $\mathbf{T}(k)$ is the identity transformation.
- □ If the unit vector from the origin in the direction of point **p** is the reverse of that in the direction of point **q**, then transformation $\mathbf{T}(1)$ is an arbitrary rotation by π radians.

Post-conditions for the unsupported function

- □ Transformation **T** is indeterminate.

Dj Create Rotation

Input

pt, **qt** : *DjPoint* (straight line).
θ : *DjReal* (angle).

Output

T : *DjTransformation* (D̈jinn transformation).

Description

This function creates the parameterized transformation **T** to rotate by a fraction of θ radians about the unbounded directed line from point **pt** to point **qt**.

Pre-conditions

- □ TRUE (there are no pre-conditions).

Actual post-conditions

- □ $\mathbf{T}(k \in [0 \ldots 1])$ is an n-dimensional linear map $\mathcal{R}^n \rightarrow \mathcal{R}^n$ where n is the maximum dimension of points **pt** and **qt**.
- □ $\mathbf{T}(k)$ rotates points by $k\theta$ radians in the direction of a right-handed screw with respect to the direction of the line from **pt** to **qt**. If one of the points has lower dimension than the other, it is augmented with zero coordinates.

Ideal post-conditions

- □ TRUE (there are no ideal conditions).

Exceptional post-conditions

- □ If points **pt** and **qt** are coincident, then the transformation **T** is indeterminate.
- □ If either point **pt** or point **qt** is indeterminate, transformation **T** is indeterminate.

Post-conditions for the unsupported function

- □ Transformation **T** is indeterminate.

Affine transformations

Dj Create Scale

Input

n : *DjInteger* (dimension).
v : *DjVector* (scale factors).

Output

T : *DjTransformation* (D̈jinn transformation).

Description

This function creates the parameterized n-dimensional linear transformation **T** to differentially scale by the elements of vector **v** or fractions thereof. Negative scale factors define a transformation that mirrors in a coordinate plane.

Pre-conditions

- □ $1 \leq n \leq 3$.
- □ Each element of **v** is non-zero.

Actual post-conditions

- □ $\mathbf{T}(k \in [0 \ldots 1])$ is an n-dimensional linear map $\mathcal{R}^n \rightarrow \mathcal{R}^n$.
- □ The effect of $\mathbf{T}$(k) on any point $\mathbf{p}$ is equivalent to the multiplication of its Euclidean coordinates by the corresponding elements of vector $k\mathbf{v}$.

Ideal post-conditions

- □ TRUE (there are no ideal conditions).

Exceptional post-conditions

- □ If $\mathbf{v}$ is the zero vector, then $\mathbf{T}$ is the zero transformation that maps all points to the origin.

Post-conditions for the unsupported function

- □ Transformation $\mathbf{T}$ is indeterminate.

Dj Create Mirror

Input

Pl : *DjPlane* (displacement).

Output

T : *DjTransformation* (D̈jinn transformation).

Description

This function creates the parameterized linear transformation $\mathbf{T}$ that mirrors points in the oriented plane **Pl**.

Pre-conditions

- □ TRUE (there are no pre-conditions).

Actual post-conditions

- □ $\mathbf{T}(k \in [0 \ldots 1])$ is a n-dimensional linear map $\mathcal{R}^n \rightarrow \mathcal{R}^n$, where n is the dimension of plane **Pl**.
- □ The straight line containing points $\mathbf{p}$ and $\mathbf{T}$(k)($\mathbf{p}$) is orthogonal to plane **Pl**.
- □ The signed distance of $\mathbf{T}$(k)($\mathbf{p}$) to the plane **Pl** is equal to -1 times the signed distance of $\mathbf{p}$ to that plane.

Ideal post-conditions

- □ TRUE (there are no ideal conditions).

Exceptional post-conditions

- □ If plane **Pl** is indeterminate, $\mathbf{T}$ is indeterminate.

Post-conditions for the unsupported function

□ Transformation **T** is indeterminate.

Dj Create Axial Shear

Input

n, i, j	:	*DjInteger*	(dimension and axes).
s	:	*DjReal*	(shear factor).

Output

T	:	*DjTransformation*	(D̈jinn transformation).

Description

This function creates the parameterized n-dimensional linear shear transformation **T** to translate points in the direction of ith coordinate axis by a fraction of the product of s and their coordinate value with respect to the jth coordinate axis.

Pre-conditions

□ $1 \leq i, j \leq n \leq 3$.

Actual post-conditions

□ $\mathbf{T}(k \in [0 \ldots 1])$ is a n-dimensional linear map $\mathcal{R}^n \to \mathcal{R}^n$.

□ Points $\mathbf{p}$ and $\mathbf{T}(k)(\mathbf{p})$ differ only in their ith Euclidean coordinate: $\mathbf{T}(k)(\mathbf{p})_i = (1 + ks\mathbf{p}_j)\mathbf{p}_i$.

Ideal post-conditions

□ TRUE (there are no ideal conditions).

Exceptional post-conditions

□ FALSE (there are no exceptional input parameter values).

Post-conditions for the unsupported function

□ Transformation **T** is indeterminate.

Dj Create General Shear

Input

n	:	*DjInteger*	(spatial dimension).
u, v	:	*DjVector*	(direction and axis).
s	:	*DjReal*	(shear factor).

Output

T	:	*DjTransformation*	(D̈jinn transformation).

Description

This function creates the parameterized n-dimensional linear shear transformation **T** to translate points in the direction of vector **v** by a fraction of the product of s and their distance from the origin in direction **u**. Vectors **u** and **v** need not be unit vectors.

Pre-conditions

- $1 \leq n \leq 3$.

Actual post-conditions

- $\mathbf{T}(k \in [0 \ldots 1])$ is a n-dimensional linear map $\mathcal{R}^n \rightarrow \mathcal{R}^n$.
- If $\mathbf{u}'$ and $\mathbf{v}'$ are unit vectors in the directions of **u** and **v** respectively, then:

$$\mathbf{T}(k)(\mathbf{p}) = \mathbf{p}.\mathbf{u}'\mathbf{u}' + \mathbf{p}.(\mathbf{v}' + ks\mathbf{u}')\mathbf{v}' + \mathbf{p}.(\mathbf{u}' \times \mathbf{v}')(\mathbf{u}' \times \mathbf{v}').$$

Ideal post-conditions

- TRUE (there are no ideal conditions).

Exceptional post-conditions

- FALSE (there are no exceptional input parameter values).

Post-conditions for the unsupported function

- Transformation **T** is indeterminate.

Dj Create View Transformation

Input

COI	:	*DjPoint*	(centre of interest).
VPN, **VUP**	:	*DjVector*	(view plane normal, view up vector).

Output

T	:	*DjTransformation*	(Djinn transformation).

Description

This function creates the parameterized three-dimensional rigid-body transformation **T**, such that **T**(1) transforms points from object space to view space. The centre of interest is the point **COI**, and the view plane normal and view up vector are provided by the, not necessarily unit, orthogonal vectors **VPN** and **VUP**.

Pre-conditions

- **VPN** and **VUP** are orthogonal.

Actual post-conditions

- □ $\mathbf{T}(k \in [0 \ldots 1])$ is a three-dimensional rigid linear map $\mathcal{R}^n \to \mathcal{R}^n$.
- □ $\mathbf{T}(k \in [0 \ldots 1])$ are a continuous sequence of transformations from the identity $\mathbf{T}(0)$ to $\mathbf{T}(1)$ defined as follows:
  - ○ $\mathbf{T}(1)(\mathbf{COI})$ is the origin.
  - ○ Vectors in the view-up (**VUP**) direction are mapped by $\mathbf{T}(1)$ to the y view-space coordinate direction.
  - ○ Vectors in the view plane normal (**VPN**) direction are mapped by $\mathbf{T}(1)$ to the z view-space coordinate direction.

Ideal post-conditions

- □ TRUE (there are no ideal conditions).

Exceptional post-conditions

- □ If the point **COI** is indeterminate, or vector **VPN** or **VUP** is zero, **T** is indeterminate.

Post-conditions for the unsupported function

- □ **T** is indeterminate.

Projective transformations

Dj Create Image Transformation

Input

DOP	:	*DjVector*	(direction of projection).
R	:	*DjReal*	(reciprocal of view distance).

Output

T : *DjTransformation* (D̈jinn transformation).

Description

This function creates the parameterized three-dimensional projective transformation **T**, such that $\mathbf{T}(1)$ transforms points from view space to image space. The direction of projection is given by the vector **DOP** and R is the reciprocal of the distance from the centre of projection to the image plane, which is perpendicular to **DOP** and passes through the origin.

Perspective projections and various parallel projections such as isometric and cabinet projections can be produced by the concatenation of transformations created using *Dj Create View Transformation* and *Dj Create Image Transformation*. (These transformations conform to the PHIGS

standard.) When R is zero, image space represents a parallel projection and when R is not zero, image space represents a perspective projection. Further details of specific projections may be found in Part I, Chapter 4.

Pre-conditions

- □ Vector **DOP** is non-zero.
- □ $R \geq 0$.

Actual post-conditions

- □ $\mathbf{T}(k \in [0 \ldots 1])$ is a three-dimensional linear projective map $\mathcal{R}^3 \rightarrow \mathcal{R}^3$.
- □ $\mathbf{T}(k \in [0 \ldots 1])$ are a continuous sequence of transformations from the identity $\mathbf{T}(0)$ to $\mathbf{T}(1)$.
- □ The origin and z-coordinate axis are unchanged by the map $\mathbf{T}(1)(k)$.

Ideal post-conditions

- □ TRUE (there are no ideal conditions).

Exceptional post-conditions

- □ FALSE (there are no exceptional input parameter values).

Post-conditions for the unsupported function

- □ **T** is indeterminate.

Quadratic transformations

Dj Create Taper

Input

m	:	*DjInteger*	(axis).
v	:	*DjVector*	(taper factors).

Output

T	:	*DjTransformation*	(D̈jinn transformation).

Description

This function creates the parameterized quadratic transformation **T** that modifies the Euclidean coordinates of points in proportion to fractions of the taper factors **v**. The transformation **T** causes a taper orthogonal to the mth coordinate axis.

Pre-conditions

- □ The mth element of **v** is zero.

ACTUAL POST-CONDITIONS

- □ $\mathbf{T}(k \in [0 \ldots 1])$ is a quadratic map $\mathcal{R}^n \rightarrow \mathcal{R}^n$, where n is the dimension of vector $\mathbf{v}$.
- □ $\exists i \in [1 \ldots n] \bullet \mathbf{v}_i = 0 \Rightarrow$
  $\forall k \in [0 \ldots 1] \bullet \forall \mathbf{p} \in \mathcal{R}^n \bullet \forall j \in [1 \ldots n] \bullet$
  $\mathbf{T}(\mathbf{p})_j = \mathbf{p}_j(1 + k\mathbf{v}_j\mathbf{p}_i)$.

IDEAL POST-CONDITIONS

- □ TRUE (there are no ideal conditions).

EXCEPTIONAL POST-CONDITIONS

- □ FALSE (there are no exceptional input parameter values).

POST-CONDITIONS FOR THE UNSUPPORTED FUNCTION

- □ $\mathbf{T}$ is indeterminate.

Dj Create Inversion

INPUT

m	:	*DjInteger*	(axis).
r	:	*DjReal*	(radius of inversion).

OUTPUT

$\mathbf{T}$	:	*DjTransformation*	(D$\ddot{\text{j}}$inn transformation).

DESCRIPTION

This function creates the parameterized n-dimensional transformation $\mathbf{T}$, such that $\mathbf{T}(1)$ geometrically inverts in a centre of inversion at coordinate r on the mth coordinate axis. Geometric inversion maps spheres (including planes) to spheres.

PRE-CONDITIONS

- □ $r > 0$.

ACTUAL POST-CONDITIONS

- □ $T(k \in [0 \ldots 1])$ is a quadratic map $\mathcal{R}^n \rightarrow \mathcal{R}^n$.
- □ $T(k \in [0 \ldots 1])$ are a continuous sequence of transformations from the identity $\mathbf{T}(0)$ to $\mathbf{T}(1)$.
- □ $||\mathbf{c} - \mathbf{p}||||\mathbf{c} - \mathbf{T}(1)(\mathbf{p})|| = r^2$, where $\mathbf{c}_i = 0, i \neq m$ and $\mathbf{c}_m = r$.

IDEAL POST-CONDITIONS

- □ TRUE (there are no ideal conditions).

EXCEPTIONAL POST-CONDITIONS

□ FALSE (there are no exceptional input parameter values).

POST-CONDITIONS FOR THE UNSUPPORTED FUNCTION

□ **T** is indeterminate.

Dj Create Bend

INPUT

m : *DjInteger* (axis).
v : *DjVector* (bend factors).

OUTPUT

T : *DjTransformation* (D̈jinn transformation).

DESCRIPTION

This function creates the parameterized quadratic transformation **T** that bends objects in proportion to fractions of the bend factors v. The transformation **T** causes planes orthogonal to the mth coordinate axis to become paraboloids.

PRE-CONDITIONS

□ The mth element of **v** is zero.

ACTUAL POST-CONDITIONS

□ $\mathbf{T}(k \in [0 \ldots 1])$ is a quadratic map $\mathcal{R}^n \rightarrow \mathcal{R}^n$.

□ **T** only affects the Euclidean point coordinate corresponding to the zero element of **v**.

□ $\exists i \in [1 \ldots n] \bullet \mathbf{v}_i = 0 \Rightarrow$
$\forall k \in [0 \ldots 1] \bullet \forall \mathbf{p} \in \mathcal{R}^n \bullet \mathbf{T}(k)(\mathbf{p})_i = \mathbf{p}_i + \sum_{j=1}^{n} k\mathbf{v}_j\mathbf{p}_j^2$.

IDEAL POST-CONDITIONS

□ TRUE (there are no ideal conditions).

EXCEPTIONAL POST-CONDITIONS

□ FALSE (there are no exceptional input parameter values).

POST-CONDITIONS FOR THE UNSUPPORTED FUNCTION

□ **T** is indeterminate.

Non-polynomial transformations

Dj Create Twist

INPUT

p : *DjReal* (pitch).

OUTPUT

T : *DjTransformation* (D̈jinn transformation).

DESCRIPTION

This function creates the parameterized transformation **T** that twists objects about the z-coordinate axis in proportion to a fraction of their coordinate in that direction.

PRE-CONDITIONS

□ $p > 0$.

ACTUAL POST-CONDITIONS

□ **T** is an n-dimensional map $\mathcal{R}^n \to \mathcal{R}^n$.

□ **T** does not affect the z-coordinates of transformed points.

□ The effect of **T** on any point is equivalent to the changes to its Euclidean coordinates:

$$\langle x, y, z \rangle \text{ to } \left\langle x \cos \tfrac{2\pi kz}{p} - y \sin \tfrac{2\pi kz}{p}, x \sin \tfrac{2\pi kz}{p} + y \cos \tfrac{2\pi kz}{p}, z \right\rangle.$$

IDEAL POST-CONDITIONS

□ TRUE (there are no ideal conditions).

EXCEPTIONAL POST-CONDITIONS

□ FALSE (there are no exceptional input parameter values).

POST-CONDITIONS FOR THE UNSUPPORTED FUNCTION

□ **T** is indeterminate.

Part II
5
Object combination

Dj Unite

Input and output

Obj_1 : *DjObject* (D̈jinn object).

Input

Obj_2 : *DjObject* (D̈jinn object).

Description

This function modifies object Obj_1 so that its point-set is the conventional (non-regularized) union of the point-sets of operands Obj_1 and Obj_2; object Obj_2 is unaffected. The point-sets of the cells of the result are:

- Those parts of the cells of Obj_1 which are outside the point-set of Obj_2.
- Cells created by the pair-wise set-intersection of cells from each operand, i.e. the cells computed by the D̈jinn intersection function.
- Those parts of the cells of Obj_2 which are outside the point-set of Obj_1.

Pre-conditions

- TRUE (there are no pre-conditions).

Actual post-conditions

- Object fields.

  Modification count is incremented:
  $Obj'_1.ModCnt = Obj_1.ModCnt + 1$.

  Update region:
  $Obj'_1.ModRegion = Obj_1.ModRegion \cup Point\text{-}set(Obj_2)$.

  Spatial dimension: $Obj'_1.Dim = \max(Obj_1.Dim, Obj_2.Dim)$.

- □ Cell point-sets and orientation: $F' = F \cup^{\circ} E$, where F and E are sets of point/orientation maps for cells of objects Obj_1 and Obj_2(see Part I, Chapter 5).
- □ Remaining cell fields of each cell L of Obj'_1 outside Obj_2.

  L derived from a cell partially outside Obj_2...one parent cell:
  $Obj'_1.Cell(L).Parents = \{\langle L, \mathsf{OUT}\rangle\}$.

  L equal to cell of Obj_1...no parents:
  $Obj'_1.Cell(L).Parents = \emptyset$.

  Inherited label and attributes:
  $Obj'_1.Cell(L).Atr = Obj_1.Cell(L).Atr$.

  Inherited manifold dimension:
  $Obj'_1.Cell(L).Dim = Obj_1.Cell(L).Dim$.
- □ Remaining fields of each cell L of Obj'_1 outside Obj_1 with point-set in that of cell H of Obj_2.

  No attributes: $Obj'_1.Cell(L).Atr = \emptyset$.

  One parent cell: $Obj'_1.Cell(L).Parents = \{\langle \mathsf{OUT}, H\rangle\}$.

  Inherited manifold dimension:
  $Obj'_1.Cell(L).Dim = Obj_2.Cell(H).Dim$.
- □ Remaining fields of each cell L of Obj'_1 with point-set in those of cells A and B of Obj_1 and Obj_2:

  Same as cells created by the function *Dj Intersect*.
- □ Region outside the object:

  $Obj'_1.Cell(\mathsf{OUT}).Dim = Obj'_1.Dim$.

  $\cup_{i \in D} Domain(Obj'_1.Cell(i).Orient) = \mathcal{R}^{Obj'.Dim}$,
  where $D = Domain(Obj'_1.Cell)$.

  $Obj'_1.Cell(\mathsf{OUT}).Atr = Obj_1.Cell(\mathsf{OUT}).Atr$.

  $Obj'_1.Cell(\mathsf{OUT}).Parents = \{\langle \mathsf{OUT}, \mathsf{OUT}\rangle\}$.

  $\forall \mathbf{p} \in Domain(Obj'_1.Cell(\mathsf{OUT}).Orient)\bullet$
  $Obj'_1.Cell(\mathsf{OUT}).Orient(\mathbf{p}) = \langle 0, \ldots\rangle$.

Ideal post-conditions

- □ TRUE (there are no ideal conditions).

Exceptional post-conditions

- □ If Obj_1 has no cells: $Obj'_1 = Obj_2$.
- □ If Obj_2 has no cells, the function has no effect: $Obj'_1 = Obj_1$.

Post-conditions for the unsupported function

- □ The function has no effect: $Obj'_1 = Obj_1$.

Dj Intersect

INPUT AND OUTPUT

Obj_1 : *DjObject* (D̈jinn object).

INPUT

Obj_2 : *DjObject* (D̈jinn object).

DESCRIPTION

This function modifies object Obj_1 such that its point-set is the conventional (non-regularized) intersection of the point-sets of operands Obj_1 and Obj_2; object Obj_2 is unaffected. The point-set of each cell of the result is the intersection of the point-set of a cell from each operand.

PRE-CONDITIONS

□ TRUE (there are no pre-conditions).

ACTUAL POST-CONDITIONS

□ Object fields.

Modification count is incremented:
$Obj'_1.ModCnt = Obj_1.ModCnt + 1$.

Update region:
$Obj'_1.ModRegion = Obj_1.ModRegion \cup Point\text{-}set(Obj_1)$.

Spatial dimension: $Obj'_1.Dim = \min(Obj_1.Dim, Obj_2.Dim)$.

□ Cell point-sets and orientation: $F' = F \cup^\circ E$, where F and E are sets of point/orientation maps for cells of objects Obj_1 and Obj_2 (see Part I, Chapter 5).

□ Remaining fields of each cell L of Obj'_1 outside Obj_1 with point-set in that of cell H of Obj_2.

Inherited attributes: $Obj'_1.Cell(L).Atr = Obj_1.Cell(A).Atr$.

Two parent cells: $Obj'_1.Cell(L).Parents = \{\langle A, B\rangle\}$.

Inherited manifold dimension: $Obj'_1.Cell(L).Dim = \min(Obj_1.Cell(A).Dim, Obj_2.Cell(B).Dim)$.

□ Each cell of Obj'_1 that is identical to a cell of Obj_1 inherits its label.

□ Region outside the object.

$Obj'_1.Cell(\mathsf{OUT}).Dim = Obj'_1.Dim$.

$\cup_{i\in D} Domain(Obj'_1.Cell(i).Orient) = \mathcal{R}^{Obj'.Dim}$,
where $D = Domain(Obj'_1.Cell)$.

$Obj'_1.Cell(\mathsf{OUT}).Atr = Obj_1.Cell(\mathsf{OUT}).Atr$.

$Obj'_1.Cell(\mathsf{OUT}).Parents = \{\langle \mathsf{OUT}, \mathsf{OUT}\rangle\}$.

$\forall \mathbf{p} \in Domain(Obj'_1.Cell(\mathsf{OUT}).Orient)\bullet$
$Obj'_1.Cell(\mathsf{OUT}).Orient(\mathbf{p}) = \langle 0, \ldots\rangle$.

Ideal post-conditions

□ TRUE (there are no ideal conditions).

Exceptional post-conditions

□ If Obj_1 has no cells, the function has no effect: $Obj'_1 = Obj_1$.

□ If Obj_2 has no cells: $Obj'_1 = Obj_2$.

Post-conditions for the unsupported function

□ The function has no effect: $Obj'_1 = Obj_1$.

Dj Subtract

Input and output

Obj_1 : *DjObject* (D̈jinn object).

Input

Obj_2 : *DjObject* (D̈jinn object).

Description

This function modifies object Obj_1 such that its point-set is the closure of the conventional (non-regularized) difference of the point-sets of operands Obj_1 and Obj_2; object Obj_2 is unaffected. The point-sets of the cells of the result are:

□ Those parts of the cells of Obj_1 which are outside the point-set of Obj_2.

□ Additional cells added where necessary to complete cell boundaries, whilst preserving the frontier condition.

Pre-conditions

□ TRUE (there are no pre-conditions).

Actual post-conditions

□ Object fields:

Modification count is incremented:
$Obj'_1.ModCnt = Obj_1.ModCnt + 1$.

Update region: $Obj'_1.ModRegion =$
$Obj_1.ModRegion \cup (Point\text{-}set(Obj_1) \cap Point\text{-}set(Obj_2))$.

Spatial dimension: $Obj'_1.Dim = Obj_1.Dim$.

□ Cell point-sets and orientation: $F' = F -^{\circ} E$, where F and E are sets of point/orientation maps for cells of objects Obj_1 and Obj_2 (see Part I, Chapter 5).

□ Remaining cell fields of each cell L of Obj'_1 outside Obj_2.

L derived from a cell partially outside Obj_2 ... one parent cell:
$Obj'_1.Cell(L).Parents = \{\langle L, \mathsf{OUT}\rangle\}$.

L equal to cell of Obj_1 ... no parents:
$Obj'_1.Cell(L).Parents = \emptyset$.

Inherited label and attributes:
$Obj'_1.Cell(L).Atr = Obj_1.Cell(L).Atr$.

Inherited manifold dimension:
$Obj'_1.Cell(L).Dim = Obj_1.Cell(L).Dim$.

□ Remaining cell fields of each cell L with point-set inside cells A of Obj_1 and boundary cells B of Obj_2.

No attributes: $Obj'_1.Cell(L).Atr = \emptyset$.

Two parent cells: $Obj'_1.Cell(L).Parents = \{\langle A, B\rangle\}$.

Inherited manifold dimension:
$Obj'_1.Cell(L).Dim = \min(Obj'_1.Cell(A).Dim, Obj_2.Cell(B).Dim)$.

□ Region outside the object.

$Obj'_1.Cell(\mathsf{OUT}).Dim = Obj'_1.Dim$.

$\cup_{i\in D} Domain(Obj'_1.Cell(i).Orient) = \mathcal{R}^{Obj'.Dim}$,
where $D = Domain(Obj'_1.Cell)$.

$Obj'_1.Cell(\mathsf{OUT}).Atr = Obj_1.Cell(\mathsf{OUT}).Atr$.

$Obj'_1.Cell(\mathsf{OUT}).Parents = \{\langle \mathsf{OUT}, \mathsf{OUT}\rangle\}$.

$\forall \mathbf{p} \in Domain(Obj'_1.Cell(\mathsf{OUT}).Orient)\bullet$
$Obj'_1.Cell(\mathsf{OUT}).Orient(\mathbf{p}) = \langle 0 \ldots\rangle$.

Ideal post-conditions

□ TRUE (there are no ideal conditions).

Exceptional post-conditions

□ If Obj_1 or Obj_2 has no cells, the function has no effect:
$Obj'_1 = Obj_1$.

Post-conditions for the unsupported function

□ The function has no effect: $Obj'_1 = Obj_1$.

Part II
6
Swept objects

Dj General USweep

INPUT AND OUTPUT

Obj_1	:	*DjObject*	(D̈jinn object).

INPUT

Obj_2	:	*DjObject*	(D̈jinn objects).
T	:	*DjTransformation*	(D̈jinn transformation).
Q	:	label of start point.	

DESCRIPTION

This function modifies object Obj_1 such that its point-set is the continuous union of a family of rigid-body transformations of its original point-set; object Obj_2 is unaffected. Each transformation is a translation to a point of Obj_2 preceded by a transformation **T**, each parameterized by the fractional distance from Q along Obj_2 to that point (see Part I, Chapter 4). If all points of the result are derived from a single point of both Obj_1 and Obj_2, each cell of the result is the sweep of a cell from Obj_1 over another from Obj_2. If such independence does not exist between Obj_1 and Obj_2, then each cell of the result contains points that derive from the same multi-set of Obj_1, Obj_2 cell pairs (see Part I, Chapter 6).

PRE-CONDITIONS

- □ Obj_2 must consist of a single chain of edges.
- □ Q must be the label, within Obj_2,of an end-point of the chain.

ACTUAL POST-CONDITIONS

- □ Object fields.

Modification count is incremented:
$Obj'_1.ModCnt = Obj_1.ModCnt + 1.$

Update region of Obj_1 is its point-set:
$Obj'_1.ModRegion = Obj_1.ModRegion \cup Point\text{-}set(Obj'_1).$

Spatial dimension:
$Obj'_1.Dim = \max(Obj_1.Dim, Obj_2.Dim).$

□ Cell fields of each cell L.

No attributes: $Obj'_1.Cell(L).Atr = \emptyset.$

Parent cells: $Obj'_1.Cell(L).Parents = \{\langle A_1, B_1\rangle\langle A_2B_2\rangle \ldots\}.$
where each point of cell L derives from point pairs in each cell pair $\langle A_i, B_i\rangle$.

If Obj_1 and Obj_2 are linearly independent, then there is only one pair of parent cells.

□ Object point-set.

$Point\text{-}set(Obj'_1) =$
$\{p(\mathbf{a}, \mathbf{b}) | \exists \mathbf{a} \in Point\text{-}set(Obj_1) \bullet \exists \mathbf{b} \in Point\text{-}set(Obj_2)\}.$
Each point in the point-sets of Obj_1 and Obj_2 is appended with zero coordinates to the maximum of their dimensions, and each modified point $\mathbf{a}$ of Obj_1 is swept by each modified point $\mathbf{b}$ of Obj_2 to a point $\mathbf{p}$ of the point-set of Obj'_1:

$\mathbf{p}(\mathbf{a}, \mathbf{b}) = \mathbf{b} + \mathbf{Tr}(\mathbf{a}),$
where $\mathbf{Tr}$ is the transformation parameterized by the relative distance along Obj_2 from Q to $\mathbf{b}$.

□ The point-set of cell L.

$Domain(Obj'_1.Cell(L).Orient) =$
$\{p(\mathbf{a}, \mathbf{b}) | \exists S \in PC \bullet \forall \langle M, N\rangle \in S \bullet \exists \mathbf{a} \in M \bullet \exists \mathbf{b} \in N\},$
where PC is the power set of (Obj_1 cell point-set, Obj_2 cell point-set) pairs, and $p(\mathbf{a}, \mathbf{b})$ is defined above.

□ Manifold dimension.
Each point of cell L derives from point pairs in each cell pair $\langle A_i, B_i\rangle$:
$Obj'_1.Cell(L).Dim = \min_i(\max(Obj'_1.Dim, D_i),$
where $D_i = Obj_1.Cell(A_i).Dim + Obj_2.Cell(B_i).Dim.$

□ Orientation.

$Obj'_1.Cell(L).Dim \in \{0, Obj'.Dim\} \Rightarrow$
orientation is the zero vector,

$0 < Obj'_1.Cell(L).Dim < Obj'.Dim\} \Rightarrow$
orientation derived from the point-set.

□ Region outside the object.

$Obj'_1.Cell(\mathsf{OUT}).Dim = Obj'_1.Dim$
$\cup_{i \in D} Domain(Obj'_1.Cell(i).Orient) = \mathcal{R}^{Obj'.Dim}$,
where $D = Domain(Obj'_1.Cell)$,
$Obj'_1.Cell(\mathsf{OUT}).Atr = \emptyset$.
$Obj'_1.Cell(\mathsf{OUT}).Parents = \{\langle \mathsf{OUT}, \mathsf{OUT} \rangle\}$.
$\forall \mathbf{p} \in Domain(\mathbf{Ut}) \bullet \mathbf{Ut}(\mathbf{p}) = \langle 0, \ldots \rangle$,
where $\mathbf{Ut} = Obj'_1.Cell(\mathsf{OUT}).Orient$.

Ideal post-conditions

□ TRUE (there are no ideal conditions).

Exceptional post-conditions

□ If Obj_1 or Obj_2 has no cells,

Spatial dimension: $Obj'_1.Dim = \max(Obj_1.Dim, Obj_2.Dim)$.

Object Obj'_1 has no cells:
$Obj'_1.Cell = \{\mathsf{OUT} \rightarrow \{\{p \in \mathcal{R}^n \rightarrow \langle 0,0,0 \rangle\}, n, \emptyset, \emptyset\}\}$,
where $n = Obj'_1.Dim$.

Modification count incremented:
$Obj'_1.ModCnt = Obj_1.ModCnt + 1$.

Update region: $Obj'_1.ModRegion = Point\text{-}set(Obj_1)$.

Post-conditions for the unsupported function

□ Same as exceptional post-condition.

Dj General ISweep

Input and output

Obj_1 : *DjObject* (Djinn object).

Input

Obj_2	:	*DjObject*	(Djinn objects).
T	:	*DjTransformation*	(Djinn transformation).
Q	:	label of start point.	

Description

This function is the same as *Dj General USweep*, but with intersection replacing union. It modifies object Obj_1 such that its point-set is the continuous intersection of a family of rigid-body transformations of its original point-set; object Obj_2 is unaffected. The point-set of the result contains those points not swept by the complement of the point-set of Obj_1. Each transformation is a translation to a point of Obj_2 preceded by a transformation $\mathbf{T}$, each parameterized by the fractional distance from

Q along Obj_2 to that point (see Part I, Chapter 4). Each cell of the result contains points that derive from the same multi-set of Obj_1, Obj_2 cell pairs (see Part I, Chapter 5). If the point-sets of the objects are independent, the result has an empty point-set.

PRE-CONDITIONS

- Obj_2 must consist of a single chain of edges.
- Q must be the label, within Obj_2,of an end-point of the chain.

ACTUAL POST-CONDITIONS

- Object fields. They are the same as *Dj General USweep*.
- Cell fields of each cell L. They are the same as *Dj General USweep*, except the point-set:

  $$\textit{Point-set}(Obj'_1) = \{p(\mathbf{a},\mathbf{b}) | \exists \mathbf{a} \in \textit{Point-set}(Obj_1) \bullet \forall \mathbf{b} \in \textit{Point-set}(Obj_2)\},$$

  where $p(\mathbf{a},\mathbf{b})$ are defined as in *Dj General USweep*. Each cell's point-set and orientation satisfy the same post-conditions as *Dj General USweep*.
- Region outside the object: same as *Dj General USweep*.

IDEAL POST-CONDITIONS

- TRUE (there are no ideal conditions).

EXCEPTIONAL POST-CONDITIONS

- These are the same as for *Dj General USweep*.

POST-CONDITIONS FOR THE UNSUPPORTED FUNCTION

- These are the same as the exceptional post-condition.

Dj Dilate

INPUT AND OUTPUT

Obj_1 : *DjObject* (Djinn object).

OUTPUT

Obj_2 : *DjObject* (Djinn object).

DESCRIPTION

This function modifies object Obj_1 such that its point-set is the Minkowski sum of the point-sets of objects Obj_1 and Obj_2; object Obj_2 is unaffected. If all points of the result are derived from a single point of both Obj_1 and Obj_2, each cell of the result is the Minkowski sum of a cell from Obj_1 and another from Obj_2. If such independence does not exist between Obj_1

and Obj_2, each cell of the result contains points that derive from the same multi-set of Obj_1, Obj_2 cell pairs (see Part I, Chapter 6).

Pre-conditions

□ TRUE (there are no pre-conditions).

Actual post-conditions

□ Object fields.

Modification count is incremented:
$Obj'_1.ModCnt = Obj_1.ModCnt + 1$.

Update region of Obj_1 is its point-set:
$Obj'_1.ModRegion = Obj_1.ModRegion \cup Point\text{-}set(Obj'_1)$.

Spatial dimension: $Obj'_1.Dim = \max(Obj_1.Dim, Obj_2.Dim)$.

□ The point-set of Obj'_1 is the Minkowski sum of the point-set of Obj_1 with that of Obj_2.

Let $A = \cup_{i \in I} Domain(Obj_1.Cell(i).Orient)$,
where $I = Domain(Obj_1.Cell) - \{\mathsf{OUT}\}$.
$B = \cup_{i \in I} Domain(Obj_2.Cell(i).Orient)$,
where $I = Domain(Obj_2.Cell) - \{\mathsf{OUT}\}$.
$R = \cup_{i \in I} Domain(Obj'_1.Cell(i).Orient)$
where $I = Domain(Obj'_1.Cell) - \{\mathsf{OUT}\}$.

Then $(p \in R) \Rightarrow (\exists \mathbf{a} \in A \bullet \exists \mathbf{b} \in B \bullet \mathbf{p} = \mathbf{a} + \mathbf{b})$.

□ Cell fields of each cell L.

No attribute: $Obj'_1.Cell(L).Atr = \emptyset$.

Parent cells: $Obj'_1.Cell(L).Parents = \{\langle A_1, B_1\rangle\langle A_2, B_2\rangle \ldots\}$,
where each point of cell L derives from point pairs in each cell pair $\langle A_i, B_i\rangle$. If Obj_1 and Obj_2 are linearly independent, there is only one parent cell pair.

- Point-set.

$Domain(Obj'_1.Cell(L).Orient) =$
$\{\mathbf{p} | \exists S \in PC \bullet$
$\forall \langle M, N\rangle \in S \bullet \exists \mathbf{a} \in M \bullet \exists \mathbf{b} \in N \bullet \mathbf{p} = \mathbf{a} + \mathbf{b}\}$,
where PC is the power set of point-set pairs from the point-sets of Obj_1 and Obj_2.

- Manifold dimension.
Each point of cell L derives from point pairs in each cell pair $\langle A_i, B_i\rangle$,
$Obj'_1.Cell(L).Dim = \min_i(\max(Obj'_1.Dim, D_i))$,
where $D_i = Obj_1.Cell(A_i).Dim + Obj_2.Cell(B_i).Dim$.

- Orientation.

$Obj'_1.Cell(L).Dim \in \{0, Obj'.Dim\} \Rightarrow$
orientation is the zero vector.
$0 < Obj'_1.Cell(L).Dim < Obj'.Dim\} \Rightarrow$
orientation derived from the point-set.

□ Region outside the object.

$Obj'_1.Cell(\mathsf{OUT}).Dim =$
$Obj'_1.Dim \cup_{i \in D} Domain(Obj'_1.Cell(i).Orient) = \mathcal{R}^{Obj'.Dim}$,
where $D = Domain(Obj'_1.Cell)$.

$Obj'_1.Cell(\mathsf{OUT}).Atr = \emptyset$.

$Obj'_1.Cell(\mathsf{OUT}).Parents = \{\langle \mathsf{OUT}, \mathsf{OUT} \rangle\}$.

$\forall \mathbf{p} \in Domain(\mathbf{Ut}) \bullet \mathbf{Ut}(\mathbf{p}) = \langle 0, \ldots \rangle$,
where $\mathbf{Ut} = Obj'.Cell(\mathsf{OUT}).Orient$.

Ideal post-conditions

□ TRUE (there are no ideal conditions).

Exceptional post-conditions

□ If Obj_1 or Obj_2 has no cells,

Spatial dimension: $Obj'_1.Dim = \max(Obj_1.Dim, Obj_2.Dim)$.

Object Obj'_1 has no cells:
$Obj'_1.Cell = \{\mathsf{OUT} \rightarrow \{\{p \in \mathcal{R}^n \rightarrow \langle 0,0,0 \rangle\}, n, \emptyset, \emptyset\}\}$.
where $n = Obj'_1.Dim$.

Modification count incremented:
$Obj'_1.ModCnt = Obj_1.ModCnt + 1$.

Update region: $Obj'_1.ModRegion = Point\text{-}set(Obj_1)$.

Post-conditions for the unsupported function

□ Same as exceptional post-condition.

Dj Erode

Input and output

Obj_1 : *DjObject* (Djinn object).

Input

Obj_2 : *DjObject* (Djinn object).

Description

This function is the same as *Dj Dilate*, with intersection replacing union. It modifies object Obj_1 such that its point-set is the complementary Minkowski sum of the point-sets of objects Obj_1 and Obj_2; object Obj_2 is unaffected.

Each cell of the result contains points that derive from the same multi-set of Obj_1, Obj_2 cell pairs (see Part I, Chapter 6). If the point-sets of the objects are independent, the result has an empty point-set.

Pre-conditions

- TRUE (there are no pre-conditions).

Actual post-conditions

- Object fields must meet the same criteria as in *Dj Dilate*.
- Cell fields of each cell L must meet the same criteria as in *Dj Dilate*, except the point-set.
- The point-set of Obj'_1 is the complementary Minkowski sum of the point-set of Obj_1 with that of Obj_2:

  $(\mathbf{p} \in R) \Rightarrow (\exists \mathbf{a} \in A \bullet \forall \mathbf{b} \in B \bullet \mathbf{p} = \mathbf{a} + \mathbf{b})$,

  where A, B and R are defined as in *Dj Dilate*.
- The point-set of each cell L:

  $Domain(Obj'_1.Cell(L).Orient) =$
  $\{\mathbf{p} | \exists S \in PC \bullet$
  $\forall \langle M, N \rangle \in S \bullet \exists \mathbf{a} \in M \bullet \forall \mathbf{b} \in N \bullet \mathbf{p} = \mathbf{a} + \mathbf{b}\}$,

  where PC is the power set of point-set pairs from the point-sets of Obj_1 and Obj_2.
- Region outside the object: same criteria as *Dj Dilate*.

Ideal post-conditions

- TRUE (there are no ideal conditions).

Exceptional post-conditions

- If Obj_1 or Obj_2 has no cells, or
- All vectors between points in the point-set of object Obj_1 are independent from those in the point-set of object Obj_2. For example, a two-dimensional Obj_1 and an edge Obj_2 with no tangent direction parallel to the xy-plane:

  Spatial dimension: $Obj'_1.Dim = \max(Obj_1.Dim, Obj_2.Dim)$.

  Object Obj'_1 has no cells:
  $Obj'_1.Cell = \{\mathsf{OUT} \rightarrow \{\{p \in \mathcal{R}^n \rightarrow \langle 0,0,0 \rangle\}, n, \emptyset, \emptyset\}\}$.
  where $n = Obj'_1.Dim$.

  Modification count incremented:
  $Obj'_1.ModCnt = Obj_1.ModCnt + 1$.

  Update region: $Obj'_1.ModRegion = Point\text{-}set(Obj_1)$.

Post-conditions for the unsupported function

- Same as exceptional post-condition.

Dj Frenet USweep

INPUT AND OUTPUT

Obj_1	:	*DjObject*	(Djinn object).

INPUT

Obj_2	:	*DjObject*	(Djinn object).
ρ, θ	:	*DjReal*.	
Q	:	label	(start-point).

DESCRIPTION

This function is a special case of *Dj General USweep*. It modifies object Obj_1 such that its point-set is the continuous union of a family of rigid-body transformations of its point-set; object Obj_2 is unaffected. Obj_2 must consist of a single chain of edges. Each transformation is a translation to a point of Obj_2 preceded by a rotation about its tangent direction, an orientation by the modified Frenet frame at that point and a scaling. The rotation and scaling are proportional to the distance along the chain. Each point of the result is derived from a single point of both Obj_1 and Obj_2, each cell of the result is the sweep of a cell from Obj_1 over another from Obj_2. If such independence does not exist between Obj_1 and Obj_2, each cell of the result contains points that derive from the same multi-set of Obj_1, Obj_2 cell pairs (see Part I, Chapter 6).

PRE-CONDITIONS

- Obj_2 must consist of a single open chain of edges.
- Q must be the label within Obj_2 of an end-point of the chain.

ACTUAL POST-CONDITIONS

- Object fields.

  Modification count is incremented:
  $Obj'_1.ModCnt = Obj_1.ModCnt + 1$.

  Update region of Obj_1 is its point-set:
  $Obj'_1.ModRegion = Obj_1.ModRegion \cup Point\text{-}set(Obj'_1)$.

  Spatial dimension: $Obj'_1.Dim = \max(Obj_1.Dim, Obj_2.Dim)$.

- Each point $\mathbf{a}$ in the point-set of Obj_1 is swept by each point $\mathbf{b}$ of the edge point-set of an Obj_2, to a point $\mathbf{p}$ in Obj'_1, as follows:

  $\mathbf{p}(\mathbf{a}, \mathbf{b}) = \mathbf{b} + \mathbf{F}(\mathbf{R}(\mathbf{S}(\mathbf{a})))$,

  where $\mathbf{S}$, $\mathbf{R}$ and $\mathbf{F}$ depend on point $\mathbf{b}(s)$ as a function of the arc length s.

  $\mathbf{S}$ uniformly scales by $\rho\frac{s}{S}$, where S is the total length of Obj'_2, $\mathbf{R}$ rotates by $\theta\frac{s}{S}$ in a right-hand sense about the z-axis, and $\mathbf{F}$ is a transformation that rotates the coordinate axes to the modified Frenet frame at point $\mathbf{b}$. The Frenet frame, F, is a rotation about the origin

that maps the x-axis to $\mathbf{N}$, the y-axis to $\mathbf{B}$ and the z-axis to $\mathbf{T}$, where $\mathbf{T}$, $\mathbf{N}$ and $\mathbf{B}$ are the tangent vector, principal normal and binormal respectively:

$\mathbf{T} = \nabla b(s)$, at point $\mathbf{b}$ as a function of arc length s.

$kN = \frac{\partial \mathbf{T}}{\partial s}$, where curvature $k = \left|\frac{\partial T}{\partial s}\right|$, and $\mathbf{B} = \mathbf{T} \times \mathbf{N}$.

The rotation $\mathbf{F}$(s) is equivalent to the orientation matrix for point coordinate post-multiplication:

$\sum_{i=1}^{n} \int_{a_i}^{b_i} \frac{\partial F}{\partial s} \partial s$,

where $(\mathbf{a}_i \ldots \mathbf{b}_i)$ are intervals of continuous Frenet-frame function of arc length, n is the number of such intervals from the start of the edge to edge length $s = \mathbf{b}_n$.

□ Cell fields of each cell L.

No attributes: $Obj'_1.Cell(L).Atr = \emptyset$.

Parent cells: $Obj'_1.Cell(L).Parents = \{\langle A_1, B_1\rangle\langle A_2, B_2\rangle \ldots\}$.

where each point of cell L derives from point pairs in each cell pair $\langle A_i, B_i\rangle$.

If Obj_1 and Obj_2 are linearly independent, there is only one parent cell pair.

□ Object point-set.

$Point\text{-}set(Obj'_1) =$
$\{p(a,b) | \exists a \in Point\text{-}set(Obj_1) \bullet \exists b \in Point\text{-}set(Obj_2)\}$.

□ Cell point-set.

$Domain(Obj'_1.Cell(L).Orient) =$
$\{p(a,b) | \exists S \in PC \bullet \forall \langle M, N\rangle \in S \bullet \exists \mathbf{a} \in M \bullet \exists \mathbf{b} \in N\}$.

where PC is the power set of Obj_1 cell point-set, Obj_2 cell point-set pairs, and $\mathbf{p}(\mathbf{a},\mathbf{b})$ is defined above.

□ Manifold dimension.

Each point of cell L derives from point pairs in each cell pair $\langle A_i, B_i\rangle$,

$Obj'_1.Cell(L).Dim = \min_i(\max(Obj'_1.Dim, D_i)$,
where $D_i = Obj_1.Cell(A_i).Dim + Obj_2.Cell(B_i).Dim$.

□ Orientation:

$Obj'_1.Cell(L).Dim \in \{0, Obj'.Dim\} \Rightarrow$
orientation is the zero vector.

$0 < Obj'_1.Cell(L).Dim < Obj'.Dim\} \Rightarrow$
orientation is derived from the point-set.

□ Region outside the object.

$Obj'_1.Cell(\mathsf{OUT}).Dim = Obj'_1.Dim$
$\cup_{i \in D} Domain(Obj'_1.Cell(i).Orient) = \mathcal{R}^{Obj'.Dim}$,
where $D = Domain(Obj'_1.Cell)$.
$Obj'_1.Cell(\mathsf{OUT}).Atr = \emptyset$.
$Obj'_1.Cell(\mathsf{OUT}).Parents = \{\langle \mathsf{OUT}, \mathsf{OUT} \rangle\}$.
$\forall \mathbf{p} \in Domain(Ut) \bullet \mathbf{Ut}(\mathbf{p}) = \langle 0, \ldots \rangle$,
where $\mathbf{Ut} = Obj'_1.Cell(\mathsf{OUT}).Orient$.

Ideal post-conditions

□ TRUE (there are no ideal conditions).

Exceptional post-conditions

□ If Obj_1 or Obj_2 has no cells.

Spatial dimension: $Obj'_1.Dim = \max(Obj_1.Dim, Obj_2.Dim)$.

Object Obj'_1 has no cells:
$Obj'_1.Cell = \{\mathsf{OUT} \rightarrow \{\{p \in \mathcal{R}^n \rightarrow \langle 0,0,0 \rangle\}, n, \emptyset, \emptyset\}\}$,
where $n = Obj'_1.Dim$.

Modification count incremented:
$Obj'_1.ModCnt = Obj_1.ModCnt + 1$.

Update region: $Obj'_1.ModRegion = Point\text{-}set(Obj_1)$.

Post-conditions for the unsupported function

□ Same as exceptional post-condition.

Dj Frenet ISweep

Input and output

Obj_1	:	*DjObject*	(D̈jinn object).

Input

Obj_2	:	*DjObject*	(D̈jinn object).
θ, ρ	:	*DjReal*.	
Q	:	label	(start point).

Description

This function is a special case of *Dj General ISweep* and the same as *Dj Frenet USweep* with intersection replacing union. It modifies object Obj_1 such that its point-set is the is the continuous intersection of a family rigid-body transformations of its point-set; object Obj_2 is unaffected. Obj_2 must be a single open chain of edges. The point-set of the result contains those points not swept by the complement of the point-set of Obj_1. Each transformation is a translation to a point of an edge cell of

Obj_2 preceded by a rotation about its tangent direction, an orientation by the modified Frenet frame at that point and a scaling. The rotation and scaling are proportional to the distance along the edge cell. Each cell of the result contains points that derive from the same multi-set of Obj_1, Obj_2 cell pairs (see Part I, Chapter 6). If the point-sets of the objects are independent, the result has an empty point-set.

Pre-conditions

- Obj_2 must consist of a single open chain of edges.
- Q must be the label within Obj_2 of an end-point of the chain.

Actual post-conditions

- The object fields must fulfil the same criteria as in *Dj Frenet USweep*, except the point-set:

  $$Domain(Obj'_1.Cell(L).Orient) = \{p(a,b) | \exists S \in PC \bullet \forall \langle M, N\rangle \in S \bullet \exists \mathbf{a} \in M \bullet \forall \mathbf{b} \in N\}.$$

  where PC is the power set of Obj_1 cell point-set, Obj_2 cell point-set pairs, and $\mathbf{p}(\mathbf{a},\mathbf{b})$ is defined above.
- The cell fields of each cell L must fulfil the same criteria as in *Dj General USweep*.
- The region outside the object must satisfy the same criteria as in *Dj Frenet USweep*.

Ideal post-conditions

- TRUE (there are no ideal conditions).

Exceptional post-conditions

- If Obj_1 or Obj_2 has no cells, the criteria are the same as in *Dj Frenet USweep*.

Post-conditions for the unsupported function

- Same as exceptional post-condition.

Dj Extrude

Input and output

Obj : *DjObject* (D̈jinn object).

Input

k : *DjReal*.

DESCRIPTION

This function extrudes object Obj of spatial dimension two to unit height in the z-direction with sloping sides. Side slopes represent a scale factor for the x and y point coordinates that varies linearly from 1 to k; the sides do not have a constant draft taper angle. Each cell of Obj gives rise to two cells: a cell consisting of disjoint pieces at each end of the extrusion; and a cell between those pieces.

PRE-CONDITIONS

□ The spatial dimension $Obj.Dim = 2$.

ACTUAL POST-CONDITIONS

□ Object fields.

Modification count is incremented:
$Obj'.ModCnt = Obj.ModCnt + 1$.

Update region of Obj is its point-set:
$Obj'.ModRegion = Obj.ModRegion \cup Point\text{-}set(Obj')$.

Spatial dimension: $Obj'.Dim = 3$.

□ Cell fields.

Each cell H of Obj gives rise to a cell A, consisting of disjoint pieces at each end of the extrusion, and a cell B between those pieces. Each cell A inherits its

Label: $A = H$.

Attributes: $Obj'.Cell(A).Atr = Obj.Cell(A).Atr$
$Obj'.Cell(B).Parents = \{\langle H, \mathsf{OUT}\rangle\}$.

Each cell B has no attributes:

$Obj'.Cell(B).Atr = \emptyset$.

$Obj'.Cell(A).Parents = \{\langle H, \mathsf{OUT}\rangle\}$.

□ Point-sets:

$Domain(Obj'.Cell(A).Orient) =$
$R \cup \{\mathbf{p} | \exists \mathbf{a} \in R \bullet \mathbf{p} = \langle k\mathbf{a}_x, k\mathbf{a}_y, 0\rangle + \langle 0, 0, 1\rangle\}$.

$Domain(Obj'.Cell(B).Orient) = \{\mathbf{p} | \exists \mathbf{a} \in R \bullet \exists t \in (0 \ldots 1) \bullet$
$vcp = \langle kz\mathbf{a}_x, kz\mathbf{a}_y, 0\rangle + \langle 0, 0, t\rangle\}$,
where $R = Domain(Obj.Cell(H).Orient)$.

□ Manifold dimension.

$Obj.Cell(A).Dim = Obj.Cell(H).Dim$.

$Obj.Cell(B).Dim = 1 + Obj.Cell(H).Dim$.

□ Orientation of each cell L.

$(Obj'.Cell(L).Dim = 3) \Rightarrow$
orientation is the zero vector.
$(Obj'.Cell(L).Dim < 3) \Rightarrow$
orientation is derived from the point-set.

□ Region outside the object.

$Obj'.Cell(\mathsf{OUT}).Dim = 3.$

$\cup_{i \in D} Domain(Obj'.Cell(i).Orient) = \mathcal{R}^3,$
where $D = Domain(Obj'.Cell)$.

$Obj'.Cell(\mathsf{OUT}).Atr = \emptyset.$

$Obj'.Cell(\mathsf{OUT}).Parents = \{\langle \mathsf{OUT}, \mathsf{OUT} \rangle\}.$

$\forall \mathbf{p} \in Domain(\mathbf{Ut}) \bullet \mathbf{Ut}(\mathbf{p}) = \langle 0, \ldots \rangle,$
where $\mathbf{Ut} = Obj'.Cell(\mathsf{OUT}).Orient$.

Ideal post-conditions

□ TRUE (there are no ideal conditions).

Exceptional post-conditions

□ If Obj has no cells.

Spatial dimension: $Obj'.Dim = 3.$

Object Obj' has no cells:
$Obj'.Cell = \{\mathsf{OUT} \rightarrow \{\{\mathbf{p} \in \mathcal{R}^3 \rightarrow \langle 0, 0, 0 \rangle\}, 3, \emptyset, \emptyset\}\}.$

Modification count incremented:
$Obj'.ModCnt = Obj.ModCnt + 1.$

Update region: $Obj'.ModRegion = Point\text{-}set(Obj)$.

Post-conditions for the unsupported function

□ Same as exceptional post-condition.

Dj Spin

Input and output

Obj : *DjObject* (Djinn objects).

Description

This function sweeps object *Obj* of spatial dimension two around the y-axis by a complete revolution. Each cell of *Obj* is swept to give a cell one dimension higher.

Pre-conditions

□ The spatial dimension $Obj.Dim = 2$.

ACTUAL POST-CONDITIONS

□ Object fields.

Modification count is incremented:
$Obj'.ModCnt = Obj.ModCnt + 1.$

Update region of Obj is its point-set:
$Obj'.ModRegion = Obj.ModRegion \cup Point\text{-}set(Obj').$

Spatial dimension: $Obj'.Dim = 3.$

□ Cell fields.

Each cell L retains its label:
$Obj'.Cell(L).Parents = \{\langle L, \mathsf{OUT}\rangle\}$

Each cell L retains its attributes:
$Obj'.Cell(L).Atr = Obj.Cell(L).Atr.$

□ Point-set.

$Domain(Obj'.Cell(L).Orient) =$
$\{\mathbf{p} | \exists \langle \mathbf{a}, \mathbf{b}\rangle \in R \bullet (\mathbf{p}_x{}^2 + \mathbf{p}_z{}^2 = \mathbf{a}^2) \wedge (\mathbf{p}_y = \mathbf{b})\}$
where $R = Domain(Obj.Cell(L).Orient)$

$Obj'.Cell(L).Dim = 1 + Obj.Cell(L).Dim.$

Orientation of each cell L:

$(Obj'.Cell(L).Dim = 3)$	$\Rightarrow$	orientation is the zero vector.
$(Obj'.Cell(L).Dim < 3)$	$\Rightarrow$	orientation is derived from the point-set.

□ Region outside the object.

$Obj'.Cell(\mathsf{OUT}).Dim = 3.$

$\cup_{i \in D} Domain(Obj'.Cell(i).Orient) = \mathcal{R}^3,$
where $D = Domain(Obj'.Cell).$

$Obj'.Cell(\mathsf{OUT}).Atr = \emptyset.$

$Obj'.Cell(\mathsf{OUT}).Parents = \{\langle \mathsf{OUT}, \mathsf{OUT}\rangle\}.$

$\forall \mathbf{p} \in Domain(\mathbf{Ut}) \bullet \mathbf{Ut}(\mathbf{p}) = \langle 0, \ldots\rangle,$
where $\mathbf{Ut} = Obj'.Cell(\mathsf{OUT}).Orient.$

IDEAL POST-CONDITIONS

□ TRUE (there are no ideal conditions).

EXCEPTIONAL POST-CONDITIONS

□ If Obj has no cells.

Spatial dimension: $Obj'.Dim = 3.$

Object Obj' has no cells:
$Obj'.Cell = \{\mathsf{OUT} \rightarrow \{\{\mathbf{p} \in \mathcal{R}^3 \rightarrow \langle 0, 0, 0\rangle\}, 3, \emptyset, \emptyset\}\}.$

Modification count incremented:
$Obj'.ModCnt = Obj.ModCnt + 1.$

Update region: $Obj'.ModRegion = Point\text{-}set(Obj).$

Post-conditions for the unsupported function

□ Same as exceptional post-condition.

Part II
7
Local changes of shape

Dj Small Tweak

INPUT AND OUTPUT

Obj : *DjObject* (D̈jinn object to be modified).

INPUT

S	:	*DjLabelSet*	(labels of cells to be transformed).
$\mathbf{T}$	:	*DjTransformation*	(tweak transformation path).
a	:	*DjReal*	(fraction of transformation path required).

OUTPUT

b : *DjReal* (indication of maximum fraction of transformation path possible).

DESCRIPTION

If possible, this function transforms the cells of object *Obj* in set S by the parametric transformation $\mathbf{T}(a)$ and modifies adjacent cells to maintain bounding relations as described in Part I Chapter 7. The object is not modified if the specified modification can not be achieved whilst maintaining all cell adjacency relations. If modification is performed, $b = a$, otherwise b is a lower bound on transformation parameters corresponding to violation of such an adjacency-preserving modification (See Part I, Chapter 7).

PRE-CONDITIONS

□ $0 < a \leq 1$.

ACTUAL POST-CONDITIONS

- $(\langle F, S, \mathbf{T}, a\rangle \notin \mathit{Domain}(\mathit{STweak})) \Rightarrow$
  $(\forall c < b \bullet \langle F, S, \mathbf{T}, c\rangle \in \mathit{Domain}(\mathit{STweak}))\wedge$
  $(\langle F, S, \mathbf{T}, b\rangle \notin \mathit{Domain}(\mathit{STweak}))$,
  where F = a set of point/orientation maps for cells of the object Obj. (See Part I, Chapter 7 for the definition of function STweak).
- $(\langle F, S, \mathbf{T}, a\rangle \in \mathit{Domain}(\mathit{STweak})) \Leftrightarrow (b = a)$.
- $(\langle F, S, \mathbf{T}, a\rangle \in \mathit{Domain}(\mathit{STweak})) \Rightarrow$
  - Object fields.
    Modification count is incremented:
    $\mathit{Obj}'.\mathit{ModCnt} = \mathit{Obj}.\mathit{ModCnt} + 1$.
    Update region: $\mathit{Obj}'.\mathit{ModRegion} =$
    $\mathit{Obj}.\mathit{ModRegion} \cup \mathit{Point\text{-}set}(\mathit{Obj}) \cup \mathit{Point\text{-}set}(\mathit{Obj}')$.
    Spatial dimension: $\mathit{Obj}'.\mathit{Dim} = \mathit{Obj}.\mathit{Dim}$.
  - Cell point-sets and orientation: $F' = \mathit{STweak}(F, S, \mathbf{T}, a)$.
  - Parents of each cell L of Obj' with a point-set that is cell H of Obj.
    Cell not modified ... no parents: $\mathit{Obj}'.\mathit{Cell}(L).\mathit{Parents} = \emptyset$.
    Cell modified (including outside region) ... one parent cell:
    $\mathit{Obj}'.\mathit{Cell}(L).\mathit{Parents} = \{\langle L, \mathsf{OUT}\rangle\}$.
  - Remaining fields of each cell L of Obj'_1 (including region outside object).
    Inherited label and attributes:
    $\mathit{Obj}'.\mathit{Cell}(L).\mathit{Atr} = \mathit{Obj}.\mathit{Cell}(L).\mathit{Atr}$.
    Inherited manifold dimension:
    $\mathit{Obj}'.\mathit{Cell}(L).\mathit{Dim} = \mathit{Obj}.\mathit{Cell}(L).\mathit{Dim}$.

IDEAL POST-CONDITIONS

- TRUE (there are no ideal conditions).

EXCEPTIONAL POST-CONDITIONS

- If Obj has no cells or set S is empty, the function has no effect: $\mathit{Obj}' = \mathit{Obj}$.

POST-CONDITIONS FOR THE UNSUPPORTED FUNCTION

- The function has no effect: $\mathit{Obj}' = \mathit{Obj}$.

Blends

Dj Convex Round

INPUT AND OUTPUT

Obj : *DjObject* (Djinn object to be rounded).

INPUT

I : *DjObject* (Djinn object giving region of influence of round).
r : *DjReal* (Djinn real giving radius of rounding).

DESCRIPTION

This function uses a radius of r to round the convex regions of that part of object *Obj* which have a manifold dimension equal to its spatial dimension and lie inside the point-set of object I. The operation is the convex Minkowski sum blend defined in Part I, Chapter 7.

PRE-CONDITIONS

□ $r > 0$.

ACTUAL POST-CONDITIONS

□ $Obj' = Obj -^{\circ} XRound(H@S, E, r)$,

where H = subset of Obj...
cells of maximum manifold dimension $Obj.Dim$.

E = point-set of object I.

S = set all cells of Obj.

$H@S$ is the departitioned object *Obj* with rank-deficient non-boundary cells eliminated, i.e. an object with a cell of maximum manifold dimension and a cell constituting its boundary.
See Part I, Chapters 5, 2 and 7 for definitions of operators @ and $-^{\circ}$ and function *XRound* respectively. These definitions define only the cell point-sets and orientation, the remaining fields follow:

□ Object fields.

Modification count is incremented:
$Obj'.ModCnt = Obj.ModCnt + 1$.

Update region: $Obj'.ModRegion =$
$Obj.ModRegion \cup (Point\text{-}set(Obj) \cap Point\text{-}set(I))$.

Spatial dimension: $Obj'.Dim = Obj.Dim$.

□ Remaining fields of each cell L of Obj'_1 with a point-set that is part of cell H of *Obj* and new boundary.

Inherited label, one parent cell:
$Obj'.Cell(L).Parents = \{\langle H, \mathsf{OUT}\rangle\}$.
Inherited attributes: $Obj'.Cell(L).Atr = Obj.Cell(H).Atr$.
Manifold dimension: $Obj'.Cell(L).Dim = Obj.Dim - 1$.

□ Remaining fields of each cell L of Obj'_1 not part of new boundary (including region outside the object):

L blended ... one parent cell:
$Obj'_1.Cell(L).Parents = \{\langle L, \mathsf{OUT}\rangle\}$.
L equal to cell of Obj_1 ... no parents:
$Obj'_1.Cell(L).Parents = \emptyset$.
Inherited label and attributes:
$Obj'.Cell(L).Atr = Obj.Cell(L).Atr$.
Inherited manifold dimension:
$Obj'.Cell(L).Dim = Obj.Cell(L).Dim$.

Ideal post-conditions

□ TRUE (there are no ideal conditions).

Exceptional post-conditions

□ If Obj does not intersect I, then the function has no effect:
$Obj' = Obj$.

□ If Obj or I has no cells, then the function has no effect:
$Obj' = Obj$.

Post-conditions for the unsupported function

□ The function has no effect: $Obj' = Obj$.

Dj Concave Round

Input and output

Obj : *DjObject* (D̈jinn object to be rounded).

Input

I : *DjObject* (D̈jinn object giving region of influence of round).
r : *DjReal* (D̈jinn real giving radius of rounding).

Description

This function uses a radius of r to round the concave regions of that part of object Obj which have a manifold dimension equal to its spatial dimension and lie inside the point-set of object I. The operation is the concave Minkowski sum blend defined in Part I, Chapter 7.

Pre-conditions

□ $r > 0$.

Actual post-conditions

□ $Obj' = Obj \cup^{\circ} VRound\,(H@S, E, r)$,

H = subset of Obj . . .
the cells of maximum manifold dimension $Obj.Dim$.

E = point-set of object I.

S = set all cells of Obj.

$H@S$ is the departitioned object Obj with rank-deficient non-boundary cells eliminated, i.e. an object with a cell of maximum manifold dimension and a cell constituting its boundary.
See Part I, Chapters 5, 2 and 7 for definitions of operators @ and $\cup^{\circ}$ and function *VRound* respectively. These definitions define only the cell point-sets and orientation, the remaining fields follow:

□ Object fields.

Modification count is incremented:
$Obj'.ModCnt = Obj.ModCnt + 1$.

Update region: $Obj'.ModRegion =$
$Obj.ModRegion \cup Point\text{-}set(Obj')$.

Spatial dimension: $Obj'.Dim = Obj.Dim$.

□ Remaining fields of each cell L of Obj'_1 with a point-set that is part of the new region.

Inherited label, one parent cell:
$Obj'.Cell\,(L).Parents = \{\langle \mathsf{OUT}, \mathsf{OUT}\rangle\}$.

Inherited attributes: $Obj'.Cell\,(L).Atr = \emptyset$.

Manifold dimension:
Non-boundary cells . . . $Obj'.Cell\,(L).Dim = Obj.Dim$.
Boundary cells . . . $Obj'.Cell\,(L).Dim = Obj.Dim - 1$.

□ Remaining fields of each cell L of Obj'_1 not part of new region (including region outside object):

No parents: $Obj'_1.Cell\,(L).Parents = \emptyset$.

Inherited attributes: $Obj'.Cell\,(L).Atr = Obj.Cell\,(L).Atr$.

Inherited manifold dimension:
$Obj'.Cell\,(L).Dim = Obj.Cell\,(L).Dim$.

Ideal post-conditions

□ TRUE (there are no ideal conditions).

Exceptional post-conditions

- If Obj does not intersect I, then the function has no effect: $Obj' = Obj$.
- If Obj or I has no cells, then the function has no effect: $Obj' = Obj$.

Post-conditions for the unsupported function

- The function has no effect: $Obj' = Obj$.

Dj Convex Blend

Input and output

Obj : *DjObject* (Djinn object to be blended).

Input

V : Set of (*DjObject* $\times$ *DjObject*)
(virtual object pairs defining edges or vertices to blend).
c : *DjInteger* (integer giving degree of continuity of blend).
r : *DjReal* (real parameter controlling fullness of blend).

Description

This function subtracts from Obj a blend point-set along those convex edges or vertices of that part of Obj with maximum manifold dimension which are selected by the virtual objects. If there are two pairs of virtual objects, a network of convex edges are blended; if there are more than two, convex vertices are blended. Blended edges or vertices are those that are contained by the intersection of the surfaces of all first virtual objects; the second object in each pair defines the region of influence of the first. The blend point-set has fullness parameter r and geometric continuity of order c, with each face adjacent to the blended edge or vertex (see Part I, Chapter 7).

Pre-conditions

- $c \geq 0$.
- $0.0 \leq r \leq 1.0$.
- Where the virtual objects V_i intersect each other at edges of Obj, they only do so at convex edges of Obj. (These edges must be convex along their entire lengths.)

Actual post-conditions

- $Obj' = Obj -^{\circ} XBlend\,(H@S, P, r, c)$,

where H = subset of Obj with only the cells of maximum manifold dimension $Obj.Dim$.
P = the set of pairs of point-sets of object pairs V.
S = the set all cells of Obj.

$H@S$ is the departitioned object Obj with rank-deficient non-boundary cells eliminated, i.e. an object with a cell of maximum manifold dimension and a cell constituting its boundary. (See Part I, Chapters 5, 2 and 7 for definitions of operators @ and $-^\circ$ and function *XBlend* respectively.) These definitions define only the cell point-sets and orientation, the remaining field follow:

- Object fields.
  Modification count is incremented:
  $Obj'.ModCnt = Obj.ModCnt + 1$.
  Update region: $Obj'.ModRegion =$
  $Obj.ModRegion \cup (Point\text{-}set(Obj) \cap_{A \in V} Point\text{-}set(A))$.
  Spatial dimension: $Obj'.Dim = Obj.Dim$.
- The remaining cell fields must meet the same criteria as in function *Dj Convex Round*.

Ideal post-conditions

- TRUE (there are no ideal conditions).

Exceptional post-conditions

- If Obj does not intersect all V, then the function has no effect:
  $Obj' = Obj$.
- If Obj or any V has no cells, then the function has no effect:
  $Obj' = Obj$.

Post-conditions for the unsupported function

- The function has no effect: $Obj' = Obj$.

Dj Concave Blend

Input and output

Obj : *DjObject* (Djinn object to be blended).

Input

V : Set of (*DjObject* × *DjObject*)
(virtual object pairs defining edges or vertices to blend).
c : *DjInteger* (integer giving degree of continuity of blend).
r : *DjReal* (real parameter controlling fullness of blend).

DESCRIPTION

This function unites with Obj a blend point-set along those concave edges or vertices of that part of Obj with maximum manifold dimension which are selected by the virtual objects. If there are two pairs of virtual objects, a network of concave edges are blended; if there are more than two, concave vertices are blended. Blended edges or vertices are those that are contained by the intersection of the surfaces of all first virtual objects; the second object in each pair defines the region of influence of the first. The blend point-set has fullness parameter r and geometric continuity of order c with each face adjacent to the blended edge or vertex (see Part I, Chapter 7).

PRE-CONDITIONS

- $c \geq 0$.
- $0.0 \leq r \leq 1.0$.
- Where the virtual objects V_i intersect each other at edges of Obj, they only do so at concave edges of Obj. (These edges must be convex along their entire lengths.)

ACTUAL POST-CONDITIONS

- $Obj' = Obj \cup^{\circ} VBlend(H@S, P, r, c)$,

  where H = subset of Obj,

  with only the cells of maximum manifold dimension $Obj.Dim$.

  P = set of pairs of point-sets of object pairs V.

  S = set all cells of Obj.

  $H@S$ is the departitioned object Obj with rank-deficient non-boundary cells eliminated, i.e. an object with a cell of maximum manifold dimension and a cell constituting its boundary. (See Part I, Chapter 5, 2 and 7 for definitions of operators @ and $\cup^{\circ}$ and function *VBlend* respectively.) These definitions define only the cell point-sets and orientation, the remaining field follow:
- Object fields.

  Modification count is incremented:
  $Obj'.ModCnt = Obj.ModCnt + 1$.

  Update region: $Obj'.ModRegion =$
  $Obj.ModRegion \cup Point\text{-}set(Obj')$.

  Spatial dimension: $Obj'.Dim = Obj.Dim$.
- The remaining cell fields must meet the same criteria as in function *Dj Concave Round*.

IDEAL POST-CONDITIONS

- TRUE (there are no ideal conditions).

Exceptional post-conditions

- □ If Obj does not intersect all V, then the function has no effect: $Obj' = Obj$.
- □ If Obj or any V has no cells, then the function has no effect: $Obj' = Obj$.

Post-conditions for the unsupported function

- □ The function has no effect: $Obj' = Obj$.

Part II
8
Interrogation

Geometric relationships

Dj Distance

INPUT

Obj_1, Obj_2	:	*DjObject*	(D$\ddot{\text{j}}$inn objects).
B	:	*DjBoolean*	(nearest/furthest selector).

OUTPUT

$\mathbf{p}$, $\mathbf{q}$	:	*DjPoint*	(D$\ddot{\text{j}}$inn points).
Obj_3	:	*DjObject*.	
Tl	:	*DjLogic*.	

DESCRIPTION

This function computes the closest (B = TRUE) or furthest (B = FALSE) points $\mathbf{p}$ and $\mathbf{q}$ in objects Obj_1 and Obj_2 respectively.

PRE-CONDITIONS

- □ TRUE (there are no pre-conditions).

ACTUAL POST-CONDITIONS

- □ TRUE (there are no actual conditions).

IDEAL POST-CONDITIONS

- □ No pair of points in the point-sets of objects Obj_1 and Obj_2 respectively have a separation distance less than or equal to that between points $\mathbf{p}$ and $\mathbf{q}$ when B is TRUE, and greater than or equal to that between points $\mathbf{p}$ and $\mathbf{q}$ when B is FALSE.
- □ Obj_3 is null.
- □ Tl is *DjTrue*.

Exceptional post-conditions

- If there are a multiple finite or an infinite number of closest or furthest point pairs, **p** and **q** are one such pair, Tl is *DjUnknown*, and Obj_3 is a D̈jinn object representing the regions of closest proximity or furthest distance.
- Object fields.

  $Obj_3.ModCnt = 0.$

  $Obj_3.ModRegion = Point\text{-}set(Obj_3).$

  $Obj.Dim = \max(Obj_1.Dim, Obj_2.Dim).$
- If Obj_1 or Obj_2 has an empty point-set, or they both do, then point **p**, point **q**, or both, are correspondingly indeterminate, and Tl is *DjFalse*.
- If the point-sets of Obj_1 and Obj_2 intersect, **p** and **q** are equal and Tl is *DjFalse*.
- If Obj_1 or Obj_2 has no cells, or neither of them do, then point **p**, point **q**, or both, are correspondingly indeterminate, and Tl is *DjUnknown*.

Post-conditions for the unsupported function

- Tl is *DjUnknown*.

Dj Directional Distance

Input

Obj_1, Obj_2	:	*DjObject*	(D̈jinn objects).
v	:	*DjVector*	(direction).
B	:	*DjBoolean*	(nearest/furthest selector).

Output

p, **q**	:	*DjPoint*	(D̈jinn points).
Obj_3	:	*DjObject*.	
Tl	:	*DjLogic*.	

Description

This function computes the closest ($B =$ TRUE) or furthest ($B =$ FALSE) points **p** and **q** in objects Obj_1 and Obj_2 respectively in the direction of vector **v**.

Pre-conditions

- TRUE (there are no pre-conditions).

Actual post-conditions

- TRUE (there are no actual conditions).

Ideal post-conditions

- □ No pair of points in the point-sets of objects Obj_1 and Obj_2 respectively have a directional separation distance $(\mathbf{p} - \mathbf{q}) \bullet \mathbf{v}$ less (greater) than or equal to that between points $\mathbf{p}$ and $\mathbf{q}$ when B is TRUE (FALSE).
- □ Obj_3 is null.
- □ Tl is *DjTrue*.

Exceptional post-conditions

- □ If there are a multiple finite or an infinite number of closest point pairs, $\mathbf{p}$ and $\mathbf{q}$ are one such pair, Tl is *DjUnknown*, Tl is *DjUnknown*, and Obj_3 is a D̈jinn object representing the regions of closest proximity or furthest distance.
- □ Object fields.

  $Obj_3.ModCnt = 0.$

  $Obj_3.ModRegion = Point\text{-}set(Obj_3).$

  $Obj_3.Dim = \max(Obj_1.Dim, Obj_2.dim).$
- □ If Obj_1 or Obj_2 has an empty point-set, or both of them do, then point $\mathbf{p}$, point $\mathbf{q}$, or both, are correspondingly indeterminate and Tl is *DjFalse*.
- □ If the point-sets of Obj_1 and Obj_2 overlap in direction $\mathbf{v}$, $\mathbf{p}$ and $\mathbf{q}$ are equal and Tl is *DjFalse*.
- □ If Obj_1 or Obj_2 has no cells, or neither has any, then point $\mathbf{p}$, point $\mathbf{q}$, or both, are correspondingly indeterminate, and Tl is *DjUnknown*.

Post-conditions for the unsupported function

- □ Tl is *DjUnknown*.

Dj Interference

Input

Obj_1, Obj_2	:	*DjObject*	(D̈jinn objects).
tol	:	*DjReal*	(tolerance).

Output

Tl : *DjLogic*.

Description

This function determines interference between the objects Obj_1 and Obj_2. It should be significantly faster than the function to compute separation distance and computation speed should increase with the magnitude of *tol*.

Pre-conditions

□ $tol > 0$.

Actual post-conditions

□ TRUE (there are no actual conditions).

Ideal post-conditions

□ The logical variable Tl is
DjTrue, if the point-sets of Obj_1 and Obj_2 intersect.
DjFalse, if the point-sets of Obj_1 and Obj_2 are disjoint.
DjUnknown, if tol is an upper bound on the diameter of the overlap region.

Exceptional post-conditions

□ If either Obj_1 or Obj_2 has an empty point-set, then Tl is *DjFalse*.

□ If either object Obj_1 or object Obj_2 has no cells, then Tl is *DjUnknown*.

Post-conditions for the unsupported function

□ Tl is *DjUnknown*.

Dj Containment

Input

Obj_1, Obj_2	:	*DjObject*	(Djinn objects).
tol	:	*DjReal*	(tolerance).

Output

Tl : *DjLogic*.

Description

This function determines whether or not object Obj_1 contains Obj_2. In conjunction with the function *Dj Interference*, all such relationships between two objects can be determined.

Pre-conditions

□ $tol > 0$.

Actual post-conditions

□ TRUE (there are no actual conditions).

Ideal post-conditions

- The logical variable Tl is
  DjTrue, if the point-set of Obj_1 contains that of Obj_2.
  DjFalse, if the point-set of Obj_1 does not contain that of Obj_2.
  DjUnknown, if *tol* is an upper bound on the diameter of the point-set of $Obj_2 - Obj_1$.

Exceptional post-conditions

- If either Obj_1 or Obj_2 has an empty point-set, then Tl is *DjFalse*.
- If either object Obj_1 or object Obj_2 has no cells, then Tl is *DjUnknown*.

Post-conditions for the unsupported function

- Tl is *DjUnknown*.

Dj Diameter

Input

Obj_1	:	*DjObject* (D̈jinn object).
v	:	*DjVector*.

Output

p, **q**	:	*DjPoint* (diameter end-points).
Obj_2	:	*DjObject*.
Tl	:	*DjLogic*.

Description

This function determines end-points **p** and **q** of the diameter of the point-set of object Obj_1, in direction **v**.

Pre-conditions

- TRUE (there are no pre-conditions).

Actual post-conditions

- TRUE (there are no actual conditions).

Ideal post-conditions

- Points **p** and **q** are both in the point-set of Obj_1.
- No two points in the point-set of Obj_1 are closer in direction **v** than **p** and **q**.
- Obj_2 is null.
- Tl is *DjTrue*.

Exceptional post-conditions

- ☐ If there are a multiple finite, or an infinite number of, diameters with the same length, then **p** . . . **q** is one such diameter, *Tl* is *DjUnknown*, and Obj_2 is a D̈jinn object representing the region containing those diameters.
- ☐ Object fields.

  $Obj_2.ModCnt = 0.$

  $Obj_2.ModRegion = Point\text{-}set(Obj_2).$

  $Obj_2.Dim = Obj_1.Dim.$
- ☐ If Obj_1 has an empty point-set, then **p** and **q** are indeterminate and *Tl* is *DjFalse*.
- ☐ If Obj_1 has no cells, then **p** and **q** are indeterminate and *Tl* is *DjUnknown*.

Post-conditions for the unsupported function

- ☐ *Tl* is *DjUnknown* and points **p** and **q** are undefined.

Dj Extreme Diameter

Input

Obj_1	:	*DjObject*	(D̈jinn object).
B	:	*DjBoolean*.	

Output

p, **q**	:	*DjPoint*	(diameter end-points).
Obj_2	:	*DjObject*.	
Tl	:	*DjLogic*.	

Description

This function determines end-points **p** and **q** of the extreme diameter of the point-set of Obj_1. The shortest diameter is computed when *B* is TRUE and the longest when it is FALSE.

Pre-conditions

- ☐ TRUE (there are no pre-conditions).

Actual post-conditions

- ☐ TRUE (there are no actual conditions).

Ideal post-conditions

- □ Points **p** and **q** are both in the point-set of Obj_1.
- □ When B = TRUE, no two points in the point-set of Obj_1 are closer than **p** and **q**.
- □ When B = FALSE, no two points in object Obj_1's point-set are further apart than **p** and **q**.
- □ Obj_2 is null.
- □ Tl is *DjTrue*.

Exceptional post-conditions

- □ If there are multiple finite or an infinite number of extreme diameters with the same length, then **p** . . . **q** is one such diameter, Tl is *DjUnknown*, and Obj_2 is a Djinn object representing the regions containing those diameters.
- □ Object fields.

  $Obj_2.ModCnt = 0.$

  $Obj_2.ModRegion = Point\text{-}set(Obj_3).$

  $Obj_2.Dim = Obj_1.Dim.$
- □ If Obj_1 has an empty point-set, then **p** and **q** are indeterminate and Tl is *DjFalse*.
- □ If Obj_1 has no cells, then **p** and **q** are indeterminate and Tl is *DjUnknown*.

Post-conditions for the unsupported function

- □ Tl is *DjUnknown* and points **p** and **q** are undefined.

Topology

Dj Genus

Input

Obj : *DjObject* (Djinn object).

Output

G : *DjInteger* (genus).

Description

This function determines genus G of object *Obj*. For example, a sphere has genus 0 and a torus has genus 1.

Pre-conditions

- □ TRUE (there are no pre-conditions).

Actual post-conditions

□ TRUE (there are no actual conditions).

Ideal post-conditions

□ G is the genus of the point-set of object Obj.

Exceptional post-conditions

□ If the point-set of Obj is empty, then G is -1.

□ If the object Obj has no cells, then G is -2.

Post-conditions for the unsupported function

□ The genus G is -3.

Dj Vexity

Input

Obj	:	*DjObject*	(Djinn object).
$C1$, $C2$	:	*DjLabel*	(cell identifiers).

Output

vex	:	*DjLogic*	(vexity).

Description

This function determines convexity or concavity of cell $C1$ in the neighbourhood of cell $C2$. If $C1$ is OUT, the vexity of the entire object Obj is determined. If $C2$ is OUT, vexity is determined with respect to the natural embedding of $C1$. For example

□ If $C1 = \mathsf{OUT}$ and $C2 = \mathsf{OUT}$, then *vex* is the vexity of Obj.

□ If $C1 = \mathsf{OUT}$ and $C2$ is a face cell, then *vex* is the vexity of Obj in the neighbourhood of its face $C2$.

□ If $C1$ is a solid cell and $C2$ is a face cell, then *vex* is the vexity of cell $C1$ in the neighbourhood of its face $C2$.

□ If $C1$ is a planar faced cell and $C2 = \mathsf{OUT}$, then *vex* is the vexity of the face with respect to the plane, i.e. TRUE.

Pre-conditions

□ Cells $C1$ and $C2$ enjoy a bounding relation; more precisely, point-set R intersects point-set S, where:
If $C1 = \mathsf{OUT}$, R is the point-set of object Obj:
$R = \cup_{i \in \mathit{Domain}(\mathit{Obj}.\mathit{Cell})} \mathit{Domain}(\mathit{Obj}.\mathit{Cell}(i).\mathit{Orient})$.
If $C1 \neq \mathsf{OUT}$, R is the point-set of cell $C1$:
$R = \mathit{Domain}(\mathit{Obj}.\mathit{Cell}(C1).\mathit{Orient})$.

If $C2 = \mathsf{OUT}$, then $S = R$.

If $C2 \neq \mathsf{OUT}$, then S is an arbitrarily small region around cell $C2$:

$S = \cap_{A \subset B \subseteq P^3} B$, where $A = \mathit{Domain}(\mathit{Obj}.\mathit{Cell}(C2).\mathit{Orient})$.

Actual post-conditions

□ TRUE (there are no actual conditions).

Ideal post-conditions

□ *vex* is:
DjTrue. If point-set $R \cap S$ is convex in the natural embedding of point-set R(P^3 for solids, unbounded straight lines for line-segments etc.
DjFalse. If point-set $R \cap S$ is concave in the natural embedding of point-set R.

Exceptional post-conditions

□ If the point-set of either $C1$ or $C2$ is empty, then *vex* is *DjUnknown*.

□ If object *Obj* has no cells, then *vex* is *DjUnknown*.

Post-conditions for the unsupported function

□ The vexity *vex* is *DjUnknown*.

Object geometry

Dj Get Shape

Input

Obj	:	*DjObject*	(Djinn object).
L	:	*DjLabel*	(cell label).

Output

s, *m*	:	*DjInteger* (Dimension).
sh	:	{empty, point, line, sphere, cylinder, cone, torus, cyclide, complicated} (Shape).
bnd	:	*DjLogic*.

Description

This function returns the spatial dimension *s*, manifold dimension *m* and shape *sh* of cell *L* of object *Obj*. If and only if the cell is bounded, *bnd* is *DjTrue*. For example, *bnd* is *DjTrue* for a straight-line segment and *DjFalse* for a planar half-space or unbounded straight line. The shape *sh* is complicated when the cell is none of the other categories, all of which apply only to rank-deficient cells, i.e. boundary cells. For example,

a spherical surface and two-dimensional or three-dimensional circle or a part thereof are all spherical, but a solid sphere or a part thereof is not. Linear and spherical shapes are independent of spatial dimension. For example, a straight line in any space and a plane in three dimensions are all linear.

Pre-conditions

- TRUE (there are no pre-conditions).

Actual post-conditions

- $s = Obj.Dim$.
- $m = Obj.Cell(\mathrm{L}).Dim$.
- $(bnd = DjFalse) \Rightarrow$ *Point-set*, i.e. $Domain(Obj.Cell(L).Orient)$, of cell L contains at least one point at infinity.
  $(bnd = DjTrue) \Rightarrow$ point-set of cell L contains no point at infinity.

Exceptional post-conditions

- If cell L has an empty point-set, then *sh* is empty and the remaining results are arbitrary.
- If *Obj* has no cells, then *bnd* is *DjUnknown* and the remaining results are arbitrary.
- If L is not a cell of *Obj*, then *bnd* is *DjUnknown* and the remaining results are arbitrary.

Post-conditions for the unsupported function

- The logical variable *bnd* is *DjUnknown* and the remaining results are arbitrary.

Dj Empty

Input

Obj	:	*DjObject*	(Djinn object).
tol	:	*DjReal*	(tolerance).

Output

Emp	:	*DjLogic*.

Description

This function determines whether or not the point-set of object *Obj* is empty.

Pre-conditions

- $tol > 0$.

Actual post-conditions

□ TRUE (there are no actual conditions).

Ideal post-conditions

□ *Emp* is *DjTrue* only if no connected component of the point-set of object *Obj* has a diameter greater than *tol*.

Exceptional post-conditions

□ If the object *Obj* has no cells, then *Emp* is *DjFalse*.

Post-conditions for the unsupported function

□ The logical variable *Emp* is *DjUnknown*.

Dj Get Bounds

Input

Obj : *DjObject* (Djinn object).

Output

p, **q** : *DjPoint*.
succ : *DjLogic* (success flag).

Description

This function computes an axially aligned cuboid that tightly contains the point-set of object *Obj*. The points **p** and **q** are the vertices of the cuboid with minimum and maximum Cartesian coordinates.

Pre-conditions

□ TRUE (there are no pre-conditions).

Actual post-conditions

□ TRUE (there are no actual conditions).

Ideal post-conditions

Let the point-set of the bounding cuboid be:

$$Cuboid\,(\mathbf{p},\mathbf{q}) = \{r = \langle r_1, \ldots r_n\rangle | (\mathbf{p}_1 \leq r_1 \leq q_1) \wedge \ldots \wedge (p_n \leq r_n \leq q_n)\}.$$

□ *Cuboid* $(\mathbf{p},\mathbf{q})$ contains the point-set of object *Obj*.

□ No smaller cuboid contains the point-set of object *Obj*.

□ The success flag *succ* is *DjTrue*.

Exceptional post-conditions

- □ If the point-set of object *Obj* is empty, then *succ* is *DjFalse* and **p** and **q** are undefined.
- □ If object *Obj* has no cells, then *succ* is *DjFalse* and **p** and **q** are undefined.

Post-conditions for the unsupported function

- □ The logical variable *succ* takes the value *DjUnknown*, and **p** and **q** are indeterminate.

Dj Get Modification Record

Input

Obj : *DjObject* (Djinn object).

Output

mc : *DjInteger*.
p, **q** : *DjPoint*.
succ : *DjLogic* (success flag).

Description

This function returns the number of modifications *mc* that have been made to object *Obj* since the modification record was reset using the function *Dj Reset Modification Record*. It also returns an axis-aligned cuboid that tightly contains the region of change of object *Obj* due to those modifications. This includes point-sets of all cells with changed point-sets, previous point-sets of those cells, and point-sets of cells that no longer exist. The points **p** and **q** are the vertices of the cuboid with minimum and maximum Cartesian coordinates.

Pre-conditions

- □ TRUE (there are no pre-conditions).

Actual post-conditions

- □ TRUE (there are no actual conditions).

Ideal post-conditions

Let the point-set of the bounding cuboid be

$$Cuboid(\mathbf{p},\mathbf{q}) = \{r = \langle r_1, \ldots r_n\rangle | (p_1 \le r_1 \le q_1) \wedge \ldots \wedge (p_n \le r_n \le q_n)\}.$$

- □ *Cuboid*(**p**, **q**) contains *Obj*.*ModRegion*.
- □ No smaller cuboid contains *Obj*.*ModRegion*.
- □ The success flag *succ* is *DjTrue*.

Exceptional post-conditions

- If object *Obj* has no cells, or its point-set is empty, then *succ* is *DjFalse* and **p** and **q** represent all space occupied since the modification record was reset.

Post-conditions for the unsupported function

- The logical variable *succ* takes the value *DjUnknown*, and **p** and **q** are indeterminate.

Dj Convex Hull

Input

Obj : *DjObject* (D̈jinn object).
tol : *DjReal* (tolerance).

Input and output

Hull : *DjObject* (D̈jinn object).

Description

This function computes the smallest convex object *Hull* that contains object *Obj* to a resolution *tol*. Points of *Hull* have the same coordinate dimension as those with the highest dimension in *Obj*. Thus, if *Obj* consists entirely of points defined in the *xy*-plane, then *Hull* represents the 2-manifold convex hull in that plane. If, however, such a two-dimensional object has been moved into three-dimensional space, then *Hull* is a 3-manifold.

Pre-conditions

- $tol > 0$.

Actual post-conditions

- Modification count of *Hull* is zero: $Hull.ModCnt = 0$.
- Update region of *Hull* is its point-set:
  $Hull.ModRegion = Point\text{-}set(Hull)$.
- Spatial dimension : $Hull.Dim = Obj.Dim$.
- *Hull* has two cells with a bounding relation between them. One cell has a manifold dimension equal to the spatial dimension of *Obj*, and its boundary cell is of one lower dimension.
- The boundary cell of *Hull* has an orientation directed away from *Obj*.
- Each cell L of *Hull* has a no attributes: $Hull.Cell(L).Atr = \emptyset$.
- Cell parentage maps are empty: $Hull.Cell(L).Parents = \emptyset$.

Ideal post-conditions

- ☐ The point-set of object *Hull* is convex.
- ☐ The point-set of object *Hull* contains that of the convex hull of object *Obj*.
- ☐ No point of *Hull* lies further than *tol* from the convex hull of *Obj*.

Exceptional post-conditions

- ☐ If the point-set of the object *Obj* is empty, then *Hull* has no cells.
- ☐ If object *Obj* has no cells, neither does object *Hull*.

Post-conditions for the unsupported function

- ☐ The object *Hull* is undefined.

Integral properties

Dj Error Bounds

Input

Pr : *DjPrecision* (Djinn precision).

Output

a, *b* : *DjReal* (upper and lower bound).
succ : *DjLogic* (success flag).

Description

This function computes the upper bound *a* and lower bound *b* for a Djinn computation that yielded the precision measure *Pr*.

Pre-conditions

- ☐ TRUE (there are no pre-conditions).

Actual post-conditions

- ☐ *a* is the lower bound: $a = Pr.lb$.
- ☐ *b* is the upper bound: $b = Pr.ub$.
- ☐ $succ = DjTrue$.

Ideal post-conditions

- ☐ TRUE (there are no ideal conditions).

Exceptional post-conditions

- ☐ If the precision *Pr* is indeterminate, then the success flag *succ* takes the value *DjUnknown* and the bounds *a* and *b* are undefined.

Post-conditions for the unsupported function

□ The success flag *succ* takes the value *DjUnknown*, and bounds a and b are undefined.

Dj Error Estimate

Input

Pr : *DjPrecision* (Djinn precision).

Output

err : *DjReal* (error estimate).
succ : *DjLogic* (success flag).

Description

This function computes the error estimate *err* for a Djinn computation that yielded the precision measure *Pr*.

Pre-conditions

□ TRUE (there are no pre-conditions).

Actual post-conditions

□ The success flag *succ* = *DjTrue*.

Ideal post-conditions

□ The error estimate *err* = *Pr.est*.

Exceptional post-conditions

□ If precision *Pr* is indeterminate, then the success flag *succ* takes the value *DjUnknown* and the error estimate *err* is undefined.

Post-conditions for the unsupported function

□ The success flag *succ* takes the value *DjUnknown* and the error estimate *err* is undefined.

Dj Volume

Input

Obj : *DjObject* (Djinn object).
S : *DjLabelSet* (Djinn cell identifiers).
acc : *DjReal* (desired relative error bound).

Output

Vol : *DjReal* (volume).
Pr : *DjPrecision* (Djinn precision).

DESCRIPTION

This function computes the volume *Vol* of those three-dimensional cells of object *Obj* with labels in the set S. The desired relative error bound is *acc* and the accuracy of the result is defined by the precision *Pr* that captures error bounds and expected error.

PRE-CONDITIONS

- $acc > 0$.
- $\mathsf{OUT} \notin S$.

ACTUAL POST-CONDITIONS

- The computed relative error estimate is within the computed relative error bounds: $Pr.lb \leq Pr.est \leq Pr.ub$.
- The computed relative error estimate is less than the required relative accuracy *acc*: $Pr.est \leq acc$.

IDEAL POST-CONDITIONS

- Let the true volume be $\int_{p \in B} dxdydz$, where $p = \langle x, y, z \rangle$, and B is the union of the point-sets $Domain(Obj.Cell(C).Orient)$ of all cells C of object *Obj* in set S.
- The computed volume *Vol* is within the required relative accuracy *acc*:

  $|Vol - (true\ volume)| \leq acc(true\ volume)$.
- The error in the computed volume *Vol* is within the computed error bounds:

  $Pr.lb(true\ volume) \leq$
  $|Vol - (true\ volume)| \leq$
  $Pr.ub(true\ volume)$.
- The error estimate for the computed volume *Vol* is correct:

  $Pr.est(true\ volume) = Vol - (true\ volume)$.

EXCEPTIONAL POST-CONDITIONS

- If object *Obj* has no cells, then *Vol* is zero and the errors constituting precision *Pr* are zero.
- If the point-set of *Obj* is empty, then *Vol* and the errors constituting precision *Pr* are not necessarily zero.

POST-CONDITIONS FOR THE UNSUPPORTED FUNCTION

- The precision *Pr* is indeterminate and the volume *Vol* is undefined.

Dj Face Area

INPUT

Obj	:	*DjObject*	(D̈jinn object).
S	:	*DjLabelSet*	(D̈jinn cell identifiers).
acc	:	*DjReal*	(desired relative error bound).

OUTPUT

Area	:	*DjReal*	(surface area).
Pr	:	*DjPrecision*	(D̈jinn precision).

DESCRIPTION

This function computes the surface area *Area* of those two-dimensional cells of object *Obj* with labels in the set S. The desired relative error bound is *acc* and the accuracy of the result is defined by the precision *Pr* that captures error bounds and expected error.

PRE-CONDITIONS

- $acc > 0$.
- $\mathsf{OUT} \notin S$.

ACTUAL POST-CONDITIONS

- The computed relative error estimate is within the computed relative error bounds:

  $Pr.lb \leq Pr.est \leq Pr.ub$.
- The computed relative error estimate is less than the required relative accuracy *acc*:

  $Pr.est \leq acc$.

IDEAL POST-CONDITIONS

- Let the true surface area be: $\mathit{true\ area} = \sum_{B \in C} \int_{r \in R} \sqrt{|G|} du dv$, where C is the set of all two-dimensional cells of object *Obj* with labels in the set S. R is a region[1] of UV-space, such that for all $r = \langle u, v \rangle$ in R, $p(u, v)$ is in the point-set of the face cell B, and G is the first fundamental matrix of B:

  $$G = \begin{bmatrix} \frac{\partial p}{\partial u} \cdot \frac{\partial p}{\partial u} & \frac{\partial p}{\partial u} \cdot \frac{\partial p}{\partial v} \\ \frac{\partial p}{\partial v} \cdot \frac{\partial p}{\partial u} & \frac{\partial p}{\partial v} \cdot \frac{\partial p}{\partial v} \end{bmatrix}.$$
- The computed surface area *Area* is within the required relative accuracy *acc*:

  $|Area - (\mathit{true\ area})| \leq acc\,(\mathit{true\ area})$.

[1] This notional parameterization does not imply that a parametric representation exists, nor that a parameterization is practical.

- □ Error in computed surface area *Area* is within the computed error bounds:

  $Pr.lb(true\ area) \leq |Area - (true\ area)| \leq Pr.ub(true\ area)$.
- □ The error estimate for the computed surface area *Area* is correct:

  $Pr.est\,(true\ area) = |Area - (true\ area)|$.

Post-conditions for the unsupported function

- □ Precision *Pr* is indeterminate and *Area* is undefined.

Dj Edge Length

Input

Obj	:	*DjObject*	(D̈jinn object).
S	:	*DjLabelSet*	(D̈jinn cell identifiers).
acc	:	*DjReal*	(desired relative error bound).

Output

Len	:	*DjReal*	(length).
Pr	:	*DjPrecision*	(D̈jinn precision).

Description

This function computes the sum *Len* of the lengths of those one-dimensional cells of object *Obj* with labels in the set *S*. The desired relative error bound is *acc* and the accuracy of the result is defined by the precision *Pr* that captures error bounds and expected error.

Pre-conditions

- □ $acc > 0$.
- □ $\mathsf{OUT} \notin S$.

Actual post-conditions

- □ The computed relative error estimate is within the computed relative error bounds:

  $Pr.lb \leq Pr.est \leq Pr.ub$.
- □ The computed relative error estimate is less than the required relative accuracy *acc*:

  $Pr.est \leq acc$.

Ideal post-conditions

- □ Let the true length be: $true\ length = \int_{p \in B} ds$, where $p = \langle x, y, z \rangle$, $ds^2 = dx^2 + dy^2 + dz^2$, and B is the union of the point-sets $Domain(Obj.Cell(C).Orient)$ of all one-dimensional cells C of object *Obj* in set *S*.

- □ Computed length *Len* is within the required relative accuracy *acc*:
  $|len - (true\ length)| \leq acc\,(true\ length)$.
- □ The error in computed length *Len* is within the computed error bounds:
  $Pr.lb(true\ length) \leq |Len - (true\ length)| \leq Pr.ub(true\ length)$.
- □ The error estimate for the computed length *Len* is correct:
  $Pr.est\,(true\ length) = |Len - (true\ length)|$.

Exceptional post-conditions

- □ If object *Obj* has no cells, then *Len* is zero and the errors constituting precision *Pr* are zero.
- □ If the point-set of *Obj* is empty, then *Len* and the errors constituting the precision *Pr* are not necessarily zero.

Post-conditions for the unsupported function

- □ The precision *Pr* is indeterminate and the length *Len* is undefined.

Dj Centroid

Input

Obj	:	*DjObject*	(D̈jinn object).
S	:	*DjLabelSet*	(D̈jinn cell identifiers).
acc	:	*DjReal*	(desired relative error bound).

Output

Cen	:	*DjPoint*	(centroid).
Pr	:	*DjPrecision*	(D̈jinn precision).

Description

This function computes the centroid *Cen* of those cells of object *Obj* with labels in the set *S*. The desired relative error bound is *acc* and the accuracy of the result is defined by the precision *Pr* that captures error bounds and expected error.

The dimension of the centroid *Cen* is the highest dimension among point-sets of cells of *Obj*. Thus, for example, if all cells are defined in the *xy*-plane, *Cen* is a two-dimensional point.

The centroid *Cen* depends only on the highest dimension cells in set *S*. Thus, for example, if *S* contains faces but no solid cells, then *Cen* is the centroid of those faces. Cells of lower dimension have no influence.

Pre-conditions

- □ $acc > 0$.
- □ $\mathsf{OUT} \notin S$.

ACTUAL POST-CONDITIONS

□ The computed relative error estimate is within the computed relative error bounds:

$Pr.lb \leq Pr.est \leq Pr.ub.$

□ The computed relative error estimate is less than the required relative accuracy *acc*:

$Pr.est \leq acc.$

IDEAL POST-CONDITIONS

□ The true volume (area or length) of cells in three (two or one) dimensions is:

$true\ volume = \int_{p \in B} dx_1 \ldots dx_n.$

and the true first moments are:

$I_i = \int_{p \in B} x_i dx_1 \ldots dx_n,\ i \in [1 \ldots n],$

where n is the highest dimensionality among points of cells of *Obj*,

$p = \langle x_1, \ldots x_n \rangle,$

and B is the union of the point-sets $Domain(Obj.Cell(C).Orient)$ of all cells C of object *Obj* in set S.

If S contains no cells of dimension n, the previous integral is zero. In general:

$true\ volume = \sum_{B \in S} \int_{r \in R} \sqrt{|G|} du_1 \ldots du_m,$

where m is the highest manifold dimension among the cells of S, R is a region of $U_1 \ldots U_m$-space such that for all $r = \langle u_1, \ldots u_m \rangle$ in R, $p(u_1 \ldots u_m)$ is in the point-set of cell B, and G is the first fundamental matrix of cell B:

$$\begin{bmatrix} \frac{\partial p}{\partial u_1} \cdot \frac{\partial p}{\partial u_1} & \cdots & \cdots & \frac{\partial p}{\partial u_1} \cdot \frac{\partial p}{\partial u_m} \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial p}{\partial u_m} \cdot \frac{\partial p}{\partial u_1} & \cdots & \cdots & \frac{\partial p}{\partial u_m} \cdot \frac{\partial p}{\partial u_m} \end{bmatrix}.$$

Similarly, the true first moments are:

$I_i = \sum_{B \in S} \int_{r \in R} x_i(u_1 \ldots u_m) \sqrt{|G|} du_1 \ldots du_m,$

□ Each coordinate of the computed centre *Cen* is within the required relative accuracy *acc*. (This condition requires zero coordinates to be computed accurately; it probably will be violated by all implementations.)

$\forall i \in [1 \ldots n] \bullet |I_i - Cen_i(true\ volume)| \leq acc \times I_i.$

□ The error in each coordinate of the computed centre *Cen* is within the computed error bounds:

$Pr.lb \times I_i \leq |I_i - Cen_i(true\ volume)| \leq Pr.ubI_i.$

- The error estimate for each coordinate of the computed centre *Cen* is correct:
  $\forall i \in [1 \ldots n] \bullet Pr.est\, I_i = |I_i - Cen_i(true\ volume)|$.

Exceptional post-conditions

- If the point-set of *Obj* is empty, or it has no cells of dimension equal to that of their point-sets, then *Pr* is indeterminate and *Cen* is undefined.

Post-conditions for the unsupported function

- Precision *Pr* is indeterminate and *Cen* is undefined.

Dj Principal Moments

Input

Obj	:	*DjObject*	(D̈jinn object).
S	:	*DjLabelSet*	(D̈jinn cell identifiers).
acc	:	*DjReal*	(desired relative error bound).

Output

n	:	*DjInteger*	(number of principal moments).
Ax	:	*DjMatrix*	(principal axes).
Mom	:	*DjVector*	(principal moments).
Pr1 , *Pr2*	:	*DjPrecision*	(D̈jinn precision).

Description

This function computes the n principal moments *Mom* and the corresponding mutually orthogonal unit principal-axis vectors *Ax* of those cells of object *Obj* with labels in the set *S*. The desired relative error bound is *acc* and the accuracy of the results is defined by the precisions *Pr1* and *Pr2* that capture error bounds and expected error.

The number of principal moments n is the highest dimension among the point-sets of all cells in set *S*. Thus, for example, if all cells are defined in the *xy*-plane, *n* is 2.

When *n* is 3, if two of the principal moments are equal, the corresponding principal axes are orthogonal to each other and to the third principal axis, but their directions are otherwise arbitrary. If all three principal moments are equal, the principal axes are all mutually orthogonal, but otherwise arbitrary.

Pre-conditions

- $acc > 0$.
- $\mathsf{OUT} \notin S$.

ACTUAL POST-CONDITIONS

- ☐ The computed maximum relative error estimates of principal moments and principal axis directions are within the computed relative error bounds:

  $Pr1.lb \leq Pr1.est \leq Pr1.ub.$

  $Pr2.lb \leq Pr2.est \leq Pr2.ub.$
- ☐ The computed relative error estimates of principle moments and principal axis directions are less than the required relative accuracy acc:

  $Pr1.est \leq acc.$

  $Pr2.est \leq acc.$
- ☐ n is the highest dimension among points of cells of S.
- ☐ The principal moments Mom are in increasing order:

  $\forall i, j \in [1 \ldots n] \bullet (i < j) \Rightarrow (Mom_i \leq Mom_j).$

IDEAL POST-CONDITIONS

- ☐ Let the true inertia matrix be I with elements I_{ij}:

  $\forall i \in [1 \ldots n] \bullet I_{ii} = \int_{p \in B} \left(\sum_{1 \leq k \leq n}^{k \neq i} x_k^2 \right) dx_1 \ldots dx_n$
  (moments of inertia).

  $\forall i, j \in [1 \ldots n], i \neq j \bullet I_{ij} = \int_{p \in B} -x_i x_j dx_1 \ldots dx_n$
  (products of inertia).

  where $p = \langle x_1 \ldots dx_n \rangle$,

  and B is the union of the point-sets $Domain(Obj.Cell(C).Orient)$ of all cells C of object Obj in set S.

  Let the true principal moments and axes be *true moments* and *true axis* respectively:

  $\forall i, j \in [1 \ldots n] \bullet true\ axis_i I = true\ moments_i true\ axis_i.$

  $(i \neq j) \Rightarrow (true\ axis_i \cdot true\ axis_j = 0)$
  (mutual orthogonality).
- ☐ Each computed principal moment Mom_i is within the required relative accuracy acc:

  $\forall i \in [1 \ldots n] \bullet true\ moments_i - Mom_i \leq acc(true\ moments_i).$
- ☐ The angle between each computed (Ax_i) and true ($true\ axis_i$) principal axis is within the required relative accuracy $2\pi.acc$:

  $\forall i \in [1 \ldots n]$ such that $true\ axis_i \cdot x_i \leq \cos(2\pi.acc)$.
- ☐ The error in each computed principal moment Mom_i is within the computed error bounds:

  $\forall i \in [1 \ldots n] \bullet$
  $Pr1.lb(true\ moments_i) \leq$
  $|true\ moments_i - Mom_i| \leq$
  $Pr1.ub(true\ moments_i).$

- The angle between each computed (Ax_i) and true($true\ axis_i$) principal axis is within the computed error bounds:
  $\forall i \in [1 \ldots n] \bullet \cos(Pr2.lb) \leq true\ axis_i \cdot x_i \leq \cos(Pr2.ub)$.
- The error estimate for each computed principal moment Mom_i is correct:
  $\forall i \in [1 \ldots n] \bullet$
  $Pr1.est\,(true\ moments_i) = |true\ moments_i - Mom_i|$.
- The error estimate for the angle between each computed (Ax_i) and true ($true\ axis_i$) principal axis is correct:
  $\forall i \in [1 \ldots n] \bullet \cos(Pr2.est\,) = true\ axis_i \cdot x_i$.

Exceptional post-conditions

- If the point-set of *Obj* is empty, or it has no cells of dimension equal to that of their point-sets, then *Pr* is indeterminate and *Ax* and *Mom* are undefined.

Post-conditions for the unsupported function

- Precisions *Pr1* and *Pr2* are indeterminate and *Ax* and *Mom* are undefined.

Dj Scalar Integral

Input

Obj	:	*DjObject*	(D̈jinn object).
S	:	*DjLabelSet*	(D̈jinn cell identifiers).
acc	:	*DjReal*	(desired relative error bound).
Fnc	:	$P^n \rightarrow \mathcal{R}$	(scalar-valued function of position).

Output

Igrl	:	*DjReal*	(volume integral).
Pr	:	*DjPrecision*	(D̈jinn precision).

Description

This function computes the integral *Igrl* of the scalar function *Fnc* over those cells of object *Obj* with labels in the set S. The desired relative error bound is *acc* and the accuracy of the result is defined by the precision *Pr* that captures error bounds and expected error.

The dimension n of points in the domain of *Fnc* is the highest dimension among the points of the point-set of *Obj*. Thus, for example, if all cells are defined in the xy-plane, then *Fnc* maps two-dimensional points to a scalar.

The dimension m of integration is the highest manifold dimension among the cells of S. Thus, for example, if S contains face cells but

no solid cells, a double integration is performed over the surfaces; cells of lower dimension contribute nothing to the integral.

Pre-conditions

- $acc > 0$.
- OUT $\notin S$.
- The dimension n of points in the domain of *Fnc* is the highest dimension among the point-sets of all cells of *Obj*.
- The scalar function *Fnc* must not invoke any D̈jinn functions.

Actual post-conditions

- The computed relative error estimate is within the computed relative error bounds:

  $Pr.lb \leq Pr.est \leq Pr.ub$.
- The computed relative error estimate is less than the required relative accuracy *acc*:

  $Pr.est \leq acc$.

Ideal post-conditions

The true integral is:

$true\ integral = \sum_{B \in S} \int_{r \in R} Fnc(\mathbf{p}) \sqrt{|G|} du_1 \ldots du_m$,

where m is the highest manifold dimension among the cells of S.

R is a region of $U_1 \ldots U_m$-dimensional space such that:
$\forall r = \langle u_1, \ldots u_m \rangle$ in R, $p(u_1 \ldots u_m)$
is in the point-set of cell B.

G is the first fundamental matrix of cell B:

$$G = \begin{bmatrix} \frac{\partial p}{\partial u_1} \cdot \frac{\partial p}{\partial u_1} & \cdots & \cdots & \frac{\partial p}{\partial u_1} \cdot \frac{\partial p}{\partial u_m} \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial p}{\partial u_m} \cdot \frac{\partial p}{\partial u_1} & \cdots & \cdots & \frac{\partial p}{\partial u_m} \cdot \frac{\partial p}{\partial u_m} \end{bmatrix}.$$

If the highest manifold dimension m is equal to the coordinate dimension n of *Obj* then,

$true\ integral = \int_{p \in B} Fnc(\mathbf{p}) dx_1 \ldots dx_n$,

where $p = \langle x_1, \ldots x_n \rangle$, and B is the union of the point-sets $Domain(Obj.Cell(C).Orient)$ of all cells C of set S in *Obj*.

- The computed integral *Igrl* is within the required relative accuracy *acc*:

  $|Igrl - (true\ integral)| \leq acc\,(true\ integral)$.
- The error in the computed integral *Igrl* is within the computed error bounds:

$Pr.lb(\textit{true integral}) \leq$
$|Igrl - (\textit{true integral})| \leq$
$Pr.ub(\textit{true integral})$.

- The error estimate for the computed integral *Igrl* is correct:
  $Pr.est\ (\textit{true integral}) = |Igrl - (\textit{true integral})|$.

Exceptional post-conditions

- If object *Obj* has no cells, then *Igrl* is zero and the errors constituting precision *Pr* are zero.
- If the point-set of *Obj* is empty, then *Igrl* and the errors constituting precision *Pr* are not necessarily zero.

Post-conditions for the unsupported function

- The precision *Pr* is indeterminate and the integral *Igrl* is undefined.

Dj Face Integral

Input

Obj	:	*DjObject*	(Djinn object).
S	:	*DjLabelSet*	(Djinn cell identifiers).
acc	:	*DjReal*	(desired relative error bound).
Fnc	:	$P^n \rightarrow \mathcal{R}^n$	(vector-valued function of position).

Output

Igrl	:	*DjReal*	(surface integral).
Pr	:	*DjPrecision*	(Djinn precision).

Description

This function computes the integral *Igrl* of the inner product of the function *Fnc* and the unit 'surface' normal of the 'face' cells of object *Obj* in set *S*. The desired relative error bound is *acc* and the accuracy of the result is defined by the precision *Pr* that captures error bounds and expected error. The dimension n of points in the domain of *Fnc* is the highest dimension among the points in the point-set of *Obj*. Integration is performed over the cells of dimension $n - 1$. Thus, for example, if all cells are defined in the xy-plane, *Fnc* maps two-dimensional points to a two-dimensional vector and integration is performed over edges.

Pre-conditions

- $acc > 0$.
- $\mathsf{OUT} \notin S$.
- The dimension n of points in the domain of *Fnc* is the highest dimension among the point-sets of all cells of *Obj*.
- The function *Fnc* must not invoke any Djinn functions.

Actual post-conditions

- ☐ The computed relative error estimate is within the computed relative error bounds:

  $Pr.lb \leq Pr.est \leq Pr.ub.$

- ☐ The computed relative error estimate is less than the required relative accuracy *acc*:

  $Pr.est \leq acc.$

Ideal post-conditions

- ☐ The true integral is:

  $true\ integral = \sum_{B \in S} \int_{r \in R} Fnc(\mathbf{p}) \cdot N(\mathbf{p}) \sqrt{|G|} du_1 \ldots du_{n-1},$
  where $p = \langle x_1, \ldots x_n \rangle$,

  $\mathrm{N}(\mathbf{p})$ is the correctly oriented unit surface normal n-dimensional vector at point $\mathbf{p}$, and R is a region of $U_1 \ldots U_{n-1}$-dimensioned space such that, for all $r = \langle u_1, \ldots u_{n-1} \rangle$ in R, $\mathbf{p}(u_1 \ldots u_{n-1})$ is in the point-set of cell B, and G is the first fundamental matrix of cell B:

  $$G = \begin{bmatrix} \frac{\partial p}{\partial u_1} \cdot \frac{\partial p}{\partial u_1} & \cdots & \cdots & \frac{\partial p}{\partial u_1} \cdot \frac{\partial p}{\partial u_{n-1}} \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial p}{\partial u_{n-1}} \cdot \frac{\partial p}{\partial u_1} & \cdots & \cdots & \frac{\partial p}{\partial u_{n-1}} \cdot \frac{\partial p}{\partial u_{n-1}} \end{bmatrix}.$$

  If $n = 3$, $true\ integral = \int_{\mathbf{p} \in B} Fnc(\mathbf{p}) \cdot dA$,
  where A is a two-dimensional element of surface,
  and B is the union of the point-sets $Domain(Obj.Cell(C).Orient)$ of all cells C of set S in Obj.

- ☐ Computed integral *Igrl* is within the required relative accuracy *acc*:

  $|Int - (true\ integral)| \leq acc(true\ integral).$

- ☐ Error in computed integral *Igrl* is within the computed error bounds:

  $Pr.lb(true\ integral) \leq$
  $|Igrl - (true\ integral)| \leq$
  $Pr.ub(true\ integral).$

- ☐ The error estimate for the computed integral *Igrl* is correct:

  $Pr.est(true\ integral) = |Igrl - (true\ integral)|.$

Exceptional post-conditions

- ☐ If object *Obj* has no cells, *Igrl* is zero and the errors constituting precision *Pr* are zero.
- ☐ If the point-set of *Obj* is empty, then *Igrl* and the errors constituting precision *Pr* are not necessarily zero.

Post-conditions for the unsupported function

- ☐ Precision *Pr* is indeterminate and *Igrl* is undefined.

Dj Edge Integral

INPUT

Obj	:	*DjObject*	(D̈jinn object).
S	:	*DjLabelSet*	(D̈jinn cell identifiers).
acc	:	*DjReal*	(desired relative error bound).
Fnc	:	$P^n \to \mathcal{R}^n$	(vector-valued function of position).

OUTPUT

Igrl	:	*DjReal*	(line integral).
Pr	:	*DjPrecision*	(D̈jinn precision).

DESCRIPTION

This function computes the integral *Igrl* of the inner product of the function *Fnc* and the unit edge direction of the edge cells of object *Obj* in set *S*. The desired relative error bound is *acc* and the accuracy of the result is defined by the precision *Pr* that captures error bounds and expected error. The dimension n of points in the domain of *Fnc* is the highest dimension among the points in the point-set of *Obj*. Integration is performed over the cells of dimension 1.

PRE-CONDITIONS

- □ $acc > 0$.
- □ $\mathsf{OUT} \notin S$.
- □ The dimension n of points in the domain of *Fnc* is the highest dimension among the point-sets of all cells of *Obj*.
- □ *Fnc* must not invoke any D̈jinn functions.

ACTUAL POST-CONDITIONS

- □ The computed relative error estimate is within the computed relative error bounds:

  $Pr.lb \leq Pr.est \leq Pr.ub.$
- □ The computed relative error estimate is less than the required relative accuracy *acc*:

  $Pr.est \leq acc.$

IDEAL POST-CONDITIONS

- □ The true integral is: $true\ integral = \int_{\mathbf{p}\in B} Fnc(\mathbf{p}) \cdot Tds$
  where $\mathbf{p} = \langle x_1, \ldots x_n \rangle$,
  $ds^2 = \sum_{1\leq i\leq n} dx_i^2$,
  The vector $\mathbf{T}$ is the correctly oriented unit edge direction at point $\mathbf{p}$, and B is the union of the point-sets $Domain(Obj.Cell(C).Orient)$ of all one-dimensional cells C of set S in *Obj*.

- □ The computed integral *Igrl* is within the required relative accuracy *acc*:

  $|Igrl - (true\ integral)| \leq acc\,(true\ integral)$.

- □ The error in the computed integral *Igrl* is within the computed error bounds:

  $Pr.lb(true\ integral) \leq$
  $|Igrl - (true\ integral)| \leq$
  $Pr.ub(true\ integral)$.

- □ The error estimate for the computed integral *Igrl* is correct:

  $Pr.est\,(true\ integral) = |Igrl - (true\ integral)|$.

Exceptional post-conditions

- □ If object *Obj* has no cells, then *Igrl* is zero and the errors constituting the precision *Pr* are zero.
- □ If the point-set of *Obj* is empty, then *Igrl* and the errors constituting the precision *Pr* are not necessarily zero.

Post-conditions for the unsupported function

- □ The precision *Pr* is indeterminate, and *Igrl* is undefined.

Medial geometry

Dj Medial Geometry

Input

Obj	:	*DjObject*	(Djinn object).
S	:	*DjLabelSet*	(Djinn identifiers).

Output

ObjR	:	*DjObject*	(medial object).

Description

This function computes the medial geometry *ObjR* of each cell of object *Obj* in set *S*, i.e. the medial surface, medial axis and mid-point of solid, face and edge cells respectively. The coordinate dimension of *ObjR* is also that of *Obj*. If *S* contains OUT, the medial geometry of the space exterior to *Obj* is computed. Medial axes and mid-points lie in their faces and edges respectively. Tangent discontinuities in cell boundaries are reflected in the medial geometry, but such a discontinuity in medial geometry constitutes a distinct cell only if the boundary discontinuity constitutes a distinct cell.

Pre-conditions

- TRUE (there are no pre-conditions).

Actual post-conditions

- The modification count of *ObjR* is zero ($ObjR.ModCnt = 0$).
- The update region of *ObjR* is its point-set:
  ($ObjR.ModRegion = Point\text{-}set(ObjR)$).
- Each cell m of *ObjR* has the null attribute ($ObjR.Cell(m).Atr = 0$).

Ideal post-conditions

- The point-set of each cell of *ObjR* is either interior to the medial geometry of the point-set of a single cell of *Obj* (or the object exterior), or it bounds such geometry.
- The point-set of each n-dimensional cell *ObjR* that is interior to a cell (or the object exterior) c of *Obj* is:
  - Contained strictly within the point-set of cell c.
  - Contains points which are equidistant from more than n point-sets of bounding cells of c.
  - Maximal: no other cell of *ObjR* contains points which are equidistant from the same cells of *Obj*.
- Each n-dimensional cell m of *ObjR* is associated with the set $ObjR.Cell(m).Parents$ of the bounding cells of *Obj* from which it is equidistant.

Exceptional post-conditions

- If object *Obj* has no cells, neither does object *ObjR*.

Post-conditions for the unsupported function

- Object *ObjR* has no cells.

Dj Medial Radius

Input

Obj, *ObjM*	:	*DjObject*	(Djinn objects).
p	:	*DjPoint*	(Djinn point).

Output

r	:	*DjReal*	(medial radius).
succ	:	*DjBoolean*	(success flag).

Description

If *ObjM* is the medial geometry object of *Obj*, this function computes the medial radius r at point **p**.

Pre-conditions

- □ *ObjM* is the medial geometry object of *ObjR*.
- □ Point **p** is in the point-set of a cell m of *ObjM*: $\mathbf{p} \in Domain(ObjR.Cell(m).Orient)$.

Actual post-conditions

- □ $r \geq 0$.
- □ $succ = DjTrue$.

Ideal post-conditions

- □ If point **p** lies in cell m of *ObjM*, it is a distance r from each of the cells of *Obj* in the set $ObjR.Cell(\mathrm{m}).Parents$.

Exceptional post-conditions

- □ If object *Obj* or object *ObjM* has no cells, or point **p** is indeterminate, $succ = DjFalse$ and r is undefined.

Post-conditions for the unsupported function

- □ $succ = DjUnknown$ and r is undefined.

Differential geometry

Dj Face Normal

Input

Obj	:	*DjObject*	(Djinn object).
p	:	*DjPoint*	(Djinn point).
L	:	*DjLabel*	(face cell label).

Output

N	:	*DjVector*	(surface normal).
succ	:	*DjLogic*	(success flag).

Description

This function computes the unit normal vector **N** at the point **p** of face cell *L* of object *Obj*. The dimension of **N** is that of *Obj*.

Actual pre-conditions

- □ Label *L* is the label of a face cell of object *Obj*.

Ideal pre-conditions

- □ Point **p** is in the point-set of face cell *L*.

Actual post-conditions

- □ the vector **N** is not null.
- □ The success flag *succ* = *DjTrue*.

Ideal post-conditions

- □ The vector **N** is a unit vector.
- □ The vector **N** is a normal to face cell L of *Obj* at point **p**.
- □ The vector **N** conforms to the orientation of face cell L of *Obj* at point **p**.

Exceptional post-conditions

- □ If object *Obj* has no cells or point **p** is indeterminate, the success flag *succ* takes the value *DjFalse*, and vector **N** is undefined.

Post-conditions for the unsupported function

- □ The success flag *succ* takes the value *DjUnknown*, and vector **N** is undefined.

Dj Face Curvature

Input

Obj	:	*DjObject*	(D̈jinn object).
p	:	*DjPoint*	(D̈jinn point).
L	:	*DjLabel*	(face cell label).

Output

k_1, k_2	:	*DjReal*	(principal face curvatures).
$\mathbf{D}_1$, $\mathbf{D}_2$, **N**	:	*DjVector*	(principal face directions).
succ	:	*DjLogic*	(success flag).

Description

This function computes the principal curvatures k_1 and k_2 ($k_1 \leq k_2$), the mutually orthogonal principal face directions $\mathbf{D}_1$ and $\mathbf{D}_2$ and the face normal **N** at the point **p** of face cell L of object *Obj*.

The curve of intersection between the face and an arbitrary normal plane has a curvature k which lies between k_1 and k_2. The sign of the curvature reflects convexity. For example, spheres with outward directed normals have negative curvatures.

Actual pre-conditions

- □ Label L is the label of an face cell of object *Obj*.

Ideal pre-conditions

- □ Point **p** is in the point-set of face cell L.

ACTUAL POST-CONDITIONS

- □ None of the vectors $\mathbf{D}_1$, $\mathbf{D}_2$ and $\mathbf{N}$ are null.
- □ The success flag *succ* = *DjTrue*.

IDEAL POST-CONDITIONS

- □ Vectors $\mathbf{D}_1$ and $\mathbf{D}_2$ are unit vectors tangent to face cell L at point $\mathbf{p}$.
- □ Vectors $\mathbf{D}_1$, $\mathbf{D}_2$ and $\mathbf{N}$ form a right-handed orthonormal set:
  $\mathbf{D}_1 \times \mathbf{D}_2 = \mathbf{N}$.
- □ Vector $\mathbf{N}$ conforms to the face orientation.
- □ k_1 and k_2 are the external normal curvatures.
- □ The plane containing point $\mathbf{p}$, the face normal $\mathbf{N}$ at point $\mathbf{p}$ and vector $\mathbf{D}_1$ ($\mathbf{D}_2$) intersects the face in a curve with curvature k_1 (k_2):
  $k_1 = -\nabla_{\mathbf{D}_1}\mathbf{N} \cdot \mathbf{D}_1$.
  $k_2 = -\nabla_{\mathbf{D}_2}\mathbf{N} \cdot \mathbf{D}_2$.

EXCEPTIONAL POST-CONDITIONS

- □ If object *Obj* has no cells, or point $\mathbf{p}$ is indeterminate, then the success flag *succ* takes the value *DjFalse* and the other results are undefined.
- □ If point $\mathbf{p}$ is an umbilic of the face L, $k_1 = k_2$ and the directions $\mathbf{D}_1$ and $\mathbf{D}_2$ are orthogonal but otherwise arbitrary.

POST-CONDITIONS FOR THE UNSUPPORTED FUNCTION

- □ *succ* = *DjUnknown* and the other results are undefined.

Dj Edge Direction

INPUT

Obj	:	*DjObject*	(D̈jinn object).
$\mathbf{p}$	:	*DjPoint*	(D̈jinn point).
L	:	*DjLabel*	(edge cell label).

OUTPUT

$\mathbf{T}$	:	*DjVector*	(tangent vector).
succ	:	*DjLogic*	(success flag).

DESCRIPTION

This function computes the unit tangent direction vector $\mathbf{T}$ at the point $\mathbf{p}$ of edge cell L of object *Obj*. The dimension of $\mathbf{T}$ is that of *Obj*.

ACTUAL PRE-CONDITIONS

- □ Label L is the label of an edge cell of object *Obj*.

IDEAL PRE-CONDITIONS

- □ Point $\mathbf{p}$ is in the point-set of edge cell L.

Actual post-conditions

□ The vector **T** is not null.

□ *succ* = *DjTrue*.

Ideal post-conditions

□ The vector **T** is a unit vector.

□ The vector **T** is tangent to the edge cell L of *Obj* at point **p**.

□ Vector **T** conforms to the orientation of edge cell L of *Obj* at point **p**.

Exceptional post-conditions

□ If object *Obj* has no cells or point **p** is indeterminate, then the success flag *succ* takes the value *DjFalse*, and the vector **T** is undefined.

Post-conditions for the unsupported function

□ The success flag *succ* takes the value *DjUnknown*, and the vector **T** is undefined.

Dj Edge Curvature

Input

Obj	:	*DjObject*	(Djinn object).
p	:	*DjPoint*	(Djinn point).
L	:	*DjLabel*	(edge cell label).

Output

k, t	:	*DjReal*	(curvature and torsion).
T, **N**, **B**	:	*DjVector*	(tangent, normal and binormal vectors).
succ	:	*DjLogic*	(success flag).

Description

This function computes the curvature k, torsion t, and the Frenet frame **T**, **N**, **B** at the point **p** of edge cell L of object *Obj*. The Frenet frame consists of the mutually orthogonal unit tangent **T**, normal **N** and binormal **B** vectors.

Actual pre-conditions

□ Label L is the label of an edge cell of object *Obj*.

Ideal pre-conditions

□ Point **p** is in the point-set of edge cell L.

Actual post-conditions

□ None of the vectors **T**, **N** and **B** are null.

Ideal post-conditions

- ☐ Vector **T** is a unit vector tangent to the edge cell L of *Obj* at point **p**. It conforms to the edge orientation.
- ☐ The curvature k of edge cell L is the magnitude of the derivative of the tangent vector **T** with respect to arc length s:

  $k = \left|\frac{\partial \mathbf{T}}{\partial s}\right|$.
- ☐ Normal vector **N** is the unit vector in the direction of the derivative of the tangent vector **T** with respect to arc length s in the edge direction:

  $k\mathbf{N} = \frac{\partial \mathbf{T}}{\partial s}$.
- ☐ The binormal vector **B** is the unit vector that completes the orthonormal Frenet frame:

  $\mathbf{B} = \mathbf{T} \times \mathbf{N}$.
- ☐ The magnitude of the torsion t of the edge is equal to the magnitude of the derivative of the binormal vector **B** with respect to arc length s. It is positive only if that derivative and the normal are in opposite directions:

  $-tN = \frac{\partial B}{\partial s}$.
- ☐ *succ* = TRUE.

Exceptional post-conditions

- ☐ If the object *Obj* has no cells or the point **p** is indeterminate, then *succ* = *DjFalse* and the other results are undefined.
- ☐ If the coordinate dimension of the object *Obj* is less than 3, then the success flag *succ* takes the value *DjFalse*, and the other results are undefined.
- ☐ If the edge has zero curvature at point **p**, then **N** is orthogonal to **T**, but otherwise arbitrary.

Post-conditions for the unsupported function

- ☐ The success flag *succ* takes the value *DjUnknown* and the other results are undefined.

Sections

Dj Point Membership

Input

Obj	:	*DjObject*	(Djinn object).
p	:	*DjPoint*	(Djinn point).

OUTPUT

c : *DjLabel* (cell label).

DESCRIPTION

This function computes the label c of the cell of object *Obj* within which point **p** lies. If **p** is outside the object, then c takes the value OUT.

PRE-CONDITIONS

□ TRUE (there are no pre-conditions).

ACTUAL POST-CONDITIONS

□ TRUE (there are no actual conditions).

IDEAL POST-CONDITIONS

□ Point **p** lies in a cell c of *Obj*: $\mathbf{p} \in \mathit{Domain}(\mathit{Obj}.\mathit{Cell}(c).\mathit{Orient})$.

EXCEPTIONAL POST-CONDITIONS

□ If object *Obj* has no cells, then c takes the value OUT.

POST-CONDITIONS FOR THE UNSUPPORTED FUNCTION

□ The cell label c takes the value OUT.

Dj Ray Classification

INPUT

Obj : *DjObject* (Djinn object).
p, **q** : *DjPoint* (Djinn points).

OUTPUT

ObjR : *DjObject* (Djinn object).

DESCRIPTION

This function computes the object *ObjR* that is the set-intersection of object *Obj* with the unbounded directed line from point **p** to point **q**. The effect is the same as that of the function *Dj Intersect* operating on *Obj* and an unbounded line object.

PRE-CONDITIONS

□ TRUE (there are no pre-conditions).

ACTUAL POST-CONDITIONS

□ Object fields.

$ObjR.ModCnt = 0.$

$ObjR.ModRegion = Point\text{-}set(ObjR).$

$ObjR.Dim = Obj.Dim.$

□ Fields of each cell L of $ObjR$ with a point-set inside cell H of Obj.

$ObjR.Cell(L).Atr = Obj.Cell(H).Atr.$

$ObjR.Cell(L).Parents = \{\langle H, \mathsf{OUT}\rangle\}.$

$ObjR.Cell(L).Dim = 0$ or $1.$

Point-set $\ldots Domain(ObjR.Cell(L).Orient) =$
$PQ \cap Domain(Obj.Cell(H).Orient),$
where PQ is the unbounded straight line through points **p** and **q**.

Orientation: Point cells have no orientation.
Edge cells are directed from point **p** to point **q**.

□ Region outside the object.

$ObjR.Cell(\mathsf{OUT}).Dim = ObjR.Dim$
$\cup_{i \in D} Domain(ObjR.Cell(i).Orient) = \mathcal{R}^{Obj.Dim},$
where $D = Domain(ObjR.Cell),$
$ObjR.Cell(\mathsf{OUT}).Atr = \emptyset.$
$ObjR.Cell(\mathsf{OUT}).Parents = \emptyset.$

Points of external cells have no direction.

IDEAL POST-CONDITIONS

□ TRUE (there are no ideal conditions).

EXCEPTIONAL POST-CONDITIONS

□ Object $ObjR$ has no cells if

- ◦ object Obj has no cells,
- ◦ points **p** and **q** are equal, or
- ◦ either point **p** or **q** is indeterminate.

POST-CONDITIONS FOR THE UNSUPPORTED FUNCTION

□ Object $ObjR$ has no cells.

Dj Surface Points

INPUT

Obj	:	*DjObject*	(Djinn object).
p, *dp*	:	*DjVector*	(Djinn point and grid spacing).
m, *n*	:	*DjInteger*	(number of rays in each direction).
acc	:	*DjReal*	(required accuracy).

INPUT AND OUTPUT

Buf : Array of $\langle$**pnt** : *Vector*3, **nrm** : *Vector*3, *Atr* : *Integer*$\rangle$ (z-buffer).

DESCRIPTION

This function intersects a regular array of $m \times n$ parallel rays with the object *Obj*. The rays are parallel to the z-coordinate axis and directed in the negative z-direction. Point **p** is the origin of the grid and the spacing is defined by the coordinates of *dp*. If necessary, each element of the z-buffer **Buf** is replaced with the first penetrated surface point **pnt** of the corresponding ray, the unit surface normal **nrm** and the face attribute *Atr*.

ACTUAL PRE-CONDITIONS

- $|\mathrm{dp}| \neq 0 \quad (m > 1) \Rightarrow (|\mathrm{dp}_x| > 0)$
  $(n > 1) \Rightarrow (|\mathrm{dp}_y| > 0)$.
- $acc > 0$.
- $m, n > 0$.

IDEAL PRE-CONDITIONS

- All surface points in **Buf** lie on the grid.

ACTUAL POST-CONDITIONS

- TRUE (there are no actual post-conditions).

IDEAL POST-CONDITIONS

- For each ray that intersects object *Obj*, if the point where the ray first penetrates the surface has a greater z-coordinate than the corresponding element of the z-buffer **Buf**, then the corresponding element of the z-buffer **Buf** is replaced with:
  - The first penetrated surface point **pnt**.
  - The corresponding unit surface normal **nrm** of the penetrated face of *Obj* on which the point **pnt** lies.
  - The attribute *Atr* of the face cell.
- Each replaced point **pnt** in **Buf** is within *acc* of a point of ray intersection with a face cell of object *Obj*.

Exceptional post-conditions

□ If object *Obj* has no cells, the *z*-buffer **Buf** is unchanged.

Post-conditions for the unsupported function

□ The *z*-buffer **Buf** is unchanged.

Dj Plane Section

Input

Obj : *DjObject* (D̈jinn object).
P : *DjPlane* (D̈jinn plane).

Output

ObjR : *DjObject* (D̈jinn object).

Description

This function computes the object *ObjR* that is the set-intersection of object *Obj* with the unbounded oriented plane **P**. The effect is the same as that of the function *Dj Intersect*.

Pre-conditions

□ TRUE (there are no pre-conditions).

Actual post-conditions

□ Object fields.

$ObjR.ModCnt = 0.$
$ObjR.ModRegion = Point\text{-}set(ObjR).$
$ObjR.Dim = Obj.Dim.$

□ Fields of each cell L of $ObjR$ with a point-set inside cell H of Obj.

$ObjR.Cell(L).Atr = Obj.Cell(H).Atr.$
$ObjR.Cell(L).Parents = \{\langle H, \mathsf{OUT}\rangle\}.$
$ObjR.Cell(L).Dim = 0, 1 \text{ or } 2.$
Point-set $\ldots Domain(ObjR.Cell(L).Orient) =$
$Domain(Obj.Cell(H).Orient) \cap \partial P.$

Orientation: Point cells $\ldots \langle 0, \ldots\rangle$ no orientation.
Edge cells ... arbitrary.
Face cells ... surface normal of plane **P**.

□ Region outside the object.

$ObjR.Cell(\mathsf{OUT}).Dim = ObjR.Dim$
$\cup_{i \in D} Domain(ObjR.Cell(i).Orient) = \mathcal{R}^{Obj.Dim}$,
where $D = Domain(ObjR.Cell)$,
$ObjR.Cell(\mathsf{OUT}).Atr = \emptyset$.
$ObjR.Cell(\mathsf{OUT}).Parents = \emptyset$.

Points of external cell have no orientation.
$\forall \mathbf{p} \in Domain(\mathbf{Ut}) \bullet \mathbf{Ut}(\mathbf{p}) = \langle 0, \ldots \rangle$,
where $\mathbf{Ut} = ObjR.Cell(\mathsf{OUT}).Orient$.

Ideal post-conditions

- TRUE (there are no ideal conditions).

Exceptional post-conditions

- Object $ObjR$ has no cells if
  - object Obj has no cells, or
  - plane $\mathbf{P}$ is indeterminate.

Post-conditions for the unsupported function

- The object $ObjR$ has no cells.

Part II
9
External data

Dj Write Object

INPUT

Obj : *DjObject* (Djinn object).

INPUT AND OUTPUT

Txt : *TextFile* (text file).

DESCRIPTION

This function writes a textual representation of the (possibly null) object *Obj* to the text file *Txt*. This representation may or may not be humanly readable. Data associated with user definable attributes of object *Obj* must be explicitly written by application software.

PRE-CONDITIONS

- The file *Txt* is open for writing; although possibly it is the standard output stream.

ACTUAL POST-CONDITIONS

- *Txt* contains the data it had on entry, to which is appended a textual representation of object *Obj*.
- The file *Txt* remains open for writing.

EXCEPTIONAL POST-CONDITIONS

- Objects with no cells are written to the file *Txt*.

POST-CONDITIONS FOR THE UNSUPPORTED FUNCTION

- Nothing is written to the file *Txt*.

Dj Read Object

INPUT AND OUTPUT

Txt : *TextFile* (text file).

OUTPUT

Obj : *DjObject* (D̈jinn object).

DESCRIPTION

This function reads a textual representation of the next (possibly null) object *Obj* from the text file *Txt*. The user-defined integer attributes described in Chapter 5 are restored, but the associated data is the responsibility of the application software.

PRE-CONDITIONS

- ☐ The file *Txt* is open for reading; although possibly it is the standard input stream.
- ☐ The next characters to be read form the textual representation of a D̈jinn object, previously written by the function *Dj Write Object*.

ACTUAL POST-CONDITIONS

- ☐ *Txt* contains the data it had on entry.
- ☐ The read pointer is advanced past the textual representation of one object.
- ☐ The text between the input and output values of the read pointer represents object *Obj*.

IDEAL POST-CONDITIONS

- ☐ An object retrieved after a write/read cycle behaves exactly the same as the object written; numerical data incurs a write/read cycle without loss of precision.

EXCEPTIONAL POST-CONDITIONS

- ☐ Objects with no cells are read from the file.

POST-CONDITIONS FOR THE UNSUPPORTED FUNCTION

- ☐ Nothing is read from the file.

Geometry import and export

Dj Export Facets

INPUT

Obj	:	*DjObject*	(D̈jinn object).
S	:	*DjLabelSequence*	(D̈jinn cell identifiers).
acc	:	*DjReal*	(approximation accuracy).

OUTPUT

Facs	:	Sequence of $\langle$*DjFaceting*$\rangle$	(faceting of corresponding cells).

DESCRIPTION

This function computes a set *Facs* of D̈jinn facetings that approximate those cells of object *Obj* with labels in S. The order of the facetings *Facs* is the same as the order of the cells in S, so the faceting which corresponds to a given cell can be determined.

Each element of *Facs* consists of a faceting of the same manifold and spatial dimensions as the corresponding cell.

PRE-CONDITIONS

- $acc > 0$.

ACTUAL POST-CONDITIONS

- For each label L in S that identifies a cell of *Obj* which is associated with a corresponding element f in the sequence of *Facs*:
  $\forall L \in S \exists (L, f) \in Facs \bullet f.Dim = Obj.Cell(L).Dim$.

IDEAL POST-CONDITIONS

- For each vertex in S, the facet vertex approximates the position of the vertex cell to an accuracy *acc*.
- For each edge in S, each straight-line segment between two adjacent points in an edge facet strip approximates part of the edge cell to an accuracy *acc* greater than the distance between:
  - each segment point and the nearest point on the approximated edge cell, and
  - each approximated edge cell bounding point and the nearest point on the piecewise straight-line boundary.
- For each face in S, the plane triangle defined by three consecutive points in a face facet strip approximates part of the face cell whose label is associated with that faceting to an accuracy *acc* greater than the distance between:
  - each facet point and the nearest point on the approximated face cell,

  - each point on an edge cell that bounds an approximated face cell and the nearest point on a bounding edge of a faceting, and
  - a vertex cell that bounds an approximated face cell and the nearest bounding vertex of a facet.
- For each solid cell in S, the tetrahedron defined by four consecutive points in a solid facet strip approximates part of the solid cell to an accuracy *acc* greater than the distance between:
  - each facet point and the nearest point in the approximated solid cell,
  - each point on a face cell that bounds an approximated solid cell and the nearest point on the bounding face of a faceting,
  - each point on an edge cell that bounds an approximated solid cell and the nearest point on the bounding edge of a faceting, and
  - a vertex cell that bounds an approximated solid cell and the nearest bounding vertex of a faceting.
- For each cell in S, every point on the cell is within a distance *acc* of the nearest point on the associated faceting in *Facs*.

EXCEPTIONAL POST-CONDITIONS

- TRUE (there are no exceptional postconditions).

POST-CONDITIONS FOR THE UNSUPPORTED FUNCTION

- The set *Facs* is empty.

Dj Export 2D Conics

INPUT

Obj	:	*DjObject*	(Djinn object).
S	:	*DjLabelSequence*	(Djinn identifiers).
acc	:	*DjReal*	(approximation accuracy).

OUTPUT

Facs : Sequence of *DjConics* for corresponding cells.

DESCRIPTION

This function computes a sequence *Facs* of sequences of conic arcs that approximate the two-dimensional edge cells of *Obj* with labels in sequence S; point, three-dimensional edge, face and solid cells are ignored.

Each element of *Facs* represents a sequence of conic arcs, approximating an edge cell; each element in the sequence consists of two points and a shape parameter. The first point of an element is normally the start of a

conic arc that approximates part of the cell; the arc end-point is the start of the next element in the sequence, and the intersection of the arc's end-point tangent lines is the second point of the sequence element. Where a cell has more than one connected component, the convention is used that a null arc joins the last point of the first component with the first point of the next. A null arc is recognizable by the tangent intersection point being coincident with the start point.

Arc shape is defined by a shape parameter usually between 0 and 1, but a special value of -1 is used to indicate a circular arc. As the parameter increases from zero to unity, the conic changes continuously from a pair of straight segments joined at the intersection of tangents to a single straight-line segment. Each limiting case has singular point(s). Consider the line-segment from the tangent intersection point to the bisector of the arc end-points. The distance between the intersection of the arc with that line-segment and the tangent intersection point increases linearly with the shape parameter. Thus, when the parameter s is in the domain $[0 \ldots 1]$, that conic arc is contained in the unbounded conic containing points $\mathbf{p}$ that satisfy

$$(1-k)(\mathbf{p}\cdot\mathbf{p}_1\times\mathbf{p}_2)(\mathbf{p}\cdot\mathbf{p}_3\times\mathbf{p}_2)+k(\mathbf{p}\cdot\mathbf{p}_1\times\mathbf{p}_3)^2=0,$$

where p_1, p_2 and p_3 are homogeneous coordinates of the arc start, tangent intersection and arc end respectively and

$$k=\frac{-s^2}{(3s-2)(s-2)}.$$

Pre-conditions

- $acc > 0$.

Actual post-conditions

- Each label in S that identifies a two-dimensional edge cell of Obj is associated with the corresponding element of *Facs*.

Ideal post-conditions

- Each conic arc between two adjacent points in an element of a piecewise conic in a *Facs* approximates part of the corresponding edge cell to an accuracy *acc* greater than the distance between
  - each arc point and a point on the approximated edge cell and vice versa, and
  - each approximated edge cell bounding point cell and a piecewise conic bounding point.
- Each two-dimensional (circular) conic edge cell of S is identical to a (circular) arc in the corresponding element of *Facs*.

- Each maximal tangent continuous segment of a two-dimensional edge cell of S is approximated by a tangent continuous sequence of conic arcs.

Exceptional post-conditions

- If object *Obj* has no two-dimensional edge cells in set S, then *Facs* is empty.

Post-conditions for the unsupported function

- The set *Facs* is empty.

Dj Import Facets

Input

Facs : Sequence of *DjFaceting* (facet data).

Output

Obj : *DjObject* (D̈jinn object).
S : *DjLabelSequence* (cell for corresponding faceting).

Description

This function creates an object *Obj* consisting of zero- one- two- or three-dimensional cells. A cell is created from a piecewise faceted approximation represented by an element of *Facs*, and its label is returned at the corresponding position in the sequence S. This allows an application to associate user-defined attributes with the imported cells. All such cells are combined in the same way as in the D̈jinn object set-union function. This ensures that extra cells are created where facetings intersect. Extra cells are also created to accommodate self-intersecting facetings or facet strips.

Pre-conditions

- The facets of a *DjFaceting* must be consistent with a single orientation of the resulting cell.

Actual post-conditions

- Object fields:

  $Obj.ModCnt = 0$.

  $Obj.ModRegion = Point\text{-}set(Obj)$.

  $Obj.Dim$ = maximum spatial dimensionality of all *DjFaceting*s in *Facs*.

- □ For each cell L of object Obj corresponding to faceting f of $Facs$:
  $Obj.Cell(L).Atr = \emptyset$.
  $Obj.Cell(L).Parents = \emptyset$.
  $Obj.Cell(L).Dim = f.Dim$.
- □ The boundary of the point-set of each edge or face cell is the union of the point-sets of the corresponding edge and vertex cells.
- □ Each cell is maximal (i.e. cells cannot be united without violating the previous condition).

Ideal post-conditions

- □ The orientation of each component of Obj is taken from the first facet in the corresponding *DjFaceting*.

Exceptional post-conditions

- □ If all the facetings $Facs$ are empty, then Obj is the null object with no cells.
- □ If all the strips in a given faceting do not have a consistent orientation, then Obj is the null object with no cells.

Post-conditions for the unsupported function

- □ Object Obj is the null object with no cells.

Dj Import 2D Conics

Input

Facs	:	Sequence of *DjConics*.	

Output

Obj	:	*DjObject*	(D̈jinn object).
S	:	*DjLabelSequence*	(cell for corresponding piecewise conic).

Description

This function creates an object Obj consisting of two-dimensional edge cells and their associated end-point cells. An edge cell is created from *DjConics* in $Facs$ and its label is returned at the corresponding position in S. All such cells are combined as in the D̈jinn set-union function. Bounding cells of these edge cells constitute vertex cells that may bound several edge cells. Cells with tangent discontinuities can be created subsequently by departitioning.

Pre-conditions

- □ TRUE (there are no pre-conditions).

Actual post-conditions

- Object fields:

  $Obj.ModCnt = 0.$

  $Obj.ModRegion = Point\text{-}set(Obj).$

  $Obj.Dim = 2.$
- Cell fields:

  $Obj.Cell(L).Atr = 0.$

  $Obj.Cell(L).Parents = 0.$

  $Obj\,Cell(L).Dim = 1.$
- $Obj.Cell(L).Orient$ reflects the sequence $Facs$.

Ideal post-conditions

- TRUE (there are no ideal post-conditions).

Exceptional post-conditions

- If $Facs$ or all its sequences are empty, object Obj is the null object with no cells.

Post-conditions for the unsupported function

- Object Obj is the null object with no cells.

Part III
A
Introduction

The first two parts of this book have covered the rationale behind D̈jinn as an API for solid modelling, independent of any specific modellers or programming languages. In this section the practicality of the D̈jinn approach is illustrated by presenting an example language binding. The choice of the language used, namely C, should not to be taken as a particular endorsement for that language when using D̈jinn; indeed, it was the opinion of the majority of the authors of this book that there are definite advantages to writing applications in a more modern language such as C++. Rather, the use of C allows us to illustrate the way bindings for D̈jinn could work in a range of languages, making the link to languages like Fortran that are a legacy from the past as well as modern languages like C++ or Java; furthermore, in this part we make comments where appropriate to point out particular issues that would occur in the production of good bindings in other languages.

Languages like C allow long names for identifiers. Thus, rather than invent a new set of names for the D̈jinn function calls in C, we have essentially used the names given for the generic definition of D̈jinn functions in this C binding; the only difference is that the hyphen and space characters are replaced by underscores. Languages that impose a more stringent limit on the lengths of identifiers will require a new set of names to be defined; a possible encoding scheme for Fortran is given below as an example.

Readers interested in a range of general issues involved in the production of bindings should consult, for example, [Martin 1985], who presents the issues in the context of graphics software.

General issues

Despite recent advances in the design of programming languages, we recognize that the most likely use of D̈jinn will be with systems developed in other, older programming languages. After much discussion it was decided that the main example language binding for D̈jinn would be for the C language, as:

- C is now a popular language for solid modelling applications.
- Although recent moves have been towards object-oriented languages, such as C++ or Java, there is still a considerable amount of legacy code (and indeed, new programs) written in Fortran.

Thus C is a compromise: the Fortran programmer may reasonably regard C as a language with a similar expressive power to Fortran, whilst the C++ programmer should be able to see readily how to map the C language binding into C++.

We have also addressed the problem of application programmers porting their applications between two languages both of which have a D̈jinn interface available. It was felt that completely automatic translation from one application language to another was unlikely to be feasible, as—if nothing else—the control structures of the languages will differ. However, by keeping the general style of bindings for D̈jinn in different languages the same, semi-automatic translation should be possible. For example, for two languages with similar control structures (such as C and Pascal) the conversion of D̈jinn calls should be achievable largely by the direct translation of function and type names.

We now discuss the general issues involved in defining D̈jinn bindings for languages other than C.

Fortran

Fortran now comes in several dialects, and, although many applications use later versions, we comment here on the problems in producing a binding for Fortran77 and earlier, referred to as *classic Fortran* in what follows.

Variable names

Classic Fortran only allows identifiers of up to six characters. A translation is therefore required from the full binding identifiers given in this document to shortened ones. We do not attempt to define this translation here, but suggest that the following rules should be applied in the formulation of a translation table:

- The first two characters should be `DJ`.
- If there is only one further part to the D̈jinn function name, up to four characters from that part may be used to make the intent of the function clear.
- If there are two further parts to the D̈jinn function name, two letters from each part should be used to form the name.
- If there are three further parts to the D̈jinn function name, use two letters from the first part and one each from the next two parts.
- If there are four further parts to the D̈jinn function name, use one letter from each part.

Normally the first letters of each part would be used. Using these rules then *Dj Copy* becomes `DJcopy`, and *Dj Create General Shear* becomes `DJcrgs`. However, even with these rules ambiguities can still arise, for example, *Dj Get Plane Coordinates* and *Dj Get Point Coordinates*. These problematic cases would have to be resolved by convention; for instance to `DJgplc` and `DJgptc`.

Data types

Modern languages encourage the use of a data aggregation feature like C structures; classic Fortran lacks such a feature. The C D̈jinn binding makes extensive use of structures and references to structures (by means of C pointers), and so a formal Fortran binding for D̈jinn will therefore have to define a set of style rules for replacing these constructions. For example, C pointers will have to be replaced by integer indices. This has the consequence that it will be much more difficult to enforce *information hiding* in Fortran implementations than in C; a lot more will be left to the discipline of the D̈jinn-in-Fortran programmer.

Recursion

Classic Fortran does not support recursive procedures. This is not, in itself, a problem for a Fortran binding, as the D̈jinn interface is not naturally used in a recursive style. Rather, it is likely to be a problem for an implementor of D̈jinn in classic Fortran.

C++

Although the clean features of C form a subset of C++, and therefore the C binding of D̈jinn is *de facto* a binding in C++ also, there are some features of C++ which could be used to advantage in an advanced binding.

Classes

C++ is an object-oriented programming language, and as such supports the concepts of *object methods*. It does this through the use of *classes*, which are an extension of the notion of a C structure. Classes allow data aggregation like C structures, but also allow (and encourage) the definition of methods that operate on the data. Thus, the C Djinn function call:

```
Dj_Create_Point(n, coords, 0, &pt);
```

that sets the Djinn object `pt` to be a point, could have an alternative form in C++ such as:

```
pt->Dj_Create_Point(n, coords, 0);
```

which uses a more object-oriented style. Unlike C, C++ allows *polymorphism* in the sense that the same function name can be used for different functions, provided that the functions can be distinguished by the types of their arguments. Thus a C++ binding could allow both the form given by the C binding and the object-oriented form, and similar alternative forms could exist for other Djinn functions that naturally affect a single (program) object.

Call by reference

C uses only the calling convention known as *call by value*; the values of function arguments are always passed to functions. As a result, the only way of changing the value of a variable in the calling code is to pass the address of the variable, and this is commonly done in C programs (and is used extensively in the C binding).

C++, on the other hand, also allows *call by reference*. The subtleties of the differences between these calling conventions are not important here, but they do mean that it would be possible to allow the C call to `Dj_Create_Point` given above to look like:

```
Dj_Create_Point(n, coords, 0, pt);
```

This alternative form does not explicitly take the address of the Djinn object `pt`.

The use of call by reference can help avoid the need for an applications programmer to deal with the concept of pointers; on the other hand, it may make it more difficult for the compiler to detect logical errors by the application programmer. We therefore remain neutral on whether the use of call by reference should be encouraged in a C++ binding.

Overloading of in-line operators

The C Djinn binding includes a function for computing the set-union of two objects, `Dj_Unite()`. To set an object variable `a` so that it is the set-union of existing object variables `b` and `c` we would write in C the code fragment:

```
Dj_Copy(b, &a); Dj_Unite(c, &a);
```

However C++ supports the overloading of in-line operators; in-line operators can call a particular function, depending on the type of the arguments supplied. This is a feature that is open to abuse, but in some circumstances it can help the programmer by making the intent of a program clearer. Thus in a C++ binding of Djinn it might make sense to overload the `|` operator[1] so that the same set-union operation could be written as:

```
a = b | c;
```

(or even '`a = b; a |= c;`'). However this form has the disadvantage that it is no longer possible for the overloaded operator to return an error code directly. The Djinn API was designed to allow future use of such operators in some cases by not having the appropriate functions return an error value; instead they set the result (say, the object `a` in the above) to some special state (e.g. in this case, to *not-an-object*).

An alternative might be to allow the in-line operators to raise a suitable exception. A C++ binding for Djinn could thus be developed at several levels:

1. A base level, equivalent to the C binding.
2. A primary level, using the object-oriented features of C++ but not in-line operators.
3. A final, optional, level that would overload in-line operations and add new functions to inquire the error returned by the last operation.

Other more advanced features of C++ could also be used to make a C++ binding seem more natural to C++ programmers. For example, namespaces could be used instead of `Dj_` to denote that Djinn functions are intended, rather than those of another library.

[1] There is an obvious correspondence between bitwise OR and set-union; the other Boolean operators have equivalents in set-theory too.

Other languages

Here are some brief comments for a further range of languages.

Java

Java has gained great popularity over the last few years. It is a C++-like language which has been 'cleaned up' and designed with a set of libraries so that it is especially useful for platform-independent and network programming. As the language shares the look of C, a Java binding for Djinn could seem very similar to the C binding. However Java is completely object-oriented, to the extent that there are no raw function calls as in C (and most other procedural languages); everything that looks like a function call is in fact a method call. It would probably make most sense then to make the entire Djinn API available as an imported Java library; say with the name `Djinn`. Then the archetypal C Djinn API call:

```
Dj_Foo_Bar();
```

might be written in a Java-like style as

```
Djinn.Foo_Bar();
```

That means a call to the `Foo_Bar()` method of the `Djinn` package (cf. the remark about namespaces at the end of the previous section). There is also scope to use the more object-oriented-looking form for operations on program objects; for example, the set-union call which in C is:

```
Dj_Unite(c, &a);
```

might be written in Java as:

```
a.Unite(c);
```

after `a`, etc., have been defined to be of type `Djinn.object`.

The designers of Java took the view that the overloading of in-line operators was to be avoided, and so this feature of C++ does not apply.

Some other interesting features of Java include the following:

Garbage collection: Languages like C and C++ leave memory management to the programmer; in particular, if a piece of storage is obtained from the operating system (by a C call to `malloc`, or the use of `new` in C++) then it is up to the programmer to return that storage to the system when it is no longer in use (or risk running out of space because of the resulting memory leak). Programs that have to do these

operations have proved to be hard to get right and to maintain, and so some modern languages, like Java, use a technique called *garbage collection* that automates the process of reclaiming unused storage.

D̈jinn was designed to work with languages that do not have garbage collection, and so has an explicit method for returning storage. This function has no effect in languages with automated garbage collection.

Threads: A Java program can have several different threads of program execution running concurrently. This should not be a problem owing to the stateless nature of the D̈jinn interface, although synchronization issues will need careful consideration.

Exceptions: Like C++, Java uses the *throw-catch* model of exception handling. This could be used as an alternative method of handling errors and exceptions in a D̈jinn binding.

Lisp

Lisp was invented in the late 1950's as a language for artificial intelligence research, and is based on the mathematical abstraction known as lambda calculus. In its pure form Lisp is a *functional* programming language, in which there is no concept of state and therefore no concept of assignment. However pure functional languages tend to have limited appeal, as if there is no state then it is difficult to pass on results! Thus most functional languages tend to relax this restriction to some extent or other, and in particular Lisp has explicit assignment of values to variables.

In Lisp the standard program construct is a list, written as the collection of list elements in brackets. In turn, when a Lisp program is run the outermost list is evaluated and an answer returned. For example, the mathematical expression $a * (5 + b)$ could be written in Lisp as:

```
(* a (+ 5 b))
```

Note from this that Lisp doesn't permit the standard infix notation of mathematics; in fact, a list is treated as something that can be evaluated, with the first element of the list being the name of the function, and the rest of the elements of the list being the arguments to that function. In the above expression '`*`', '`+`' and '`5`' would be recognized as known functions and constants by the Lisp system, whereas '`a`' and '`b`' would be variables defined already, and with values that can easily be looked up. All function applications look like this in Lisp, which makes it easy to write Lisp functions with results that are Lisp expressions which can be evaluated in turn. (This last property made Lisp much suited to work in artificial intelligence.)

Although Lisp has a very simple set of syntactic rules, the standard Lisp libraries contain many pre-defined functions that implement most of the features found in modern programming languages, including arrays, aggregates (cf. C structures), garbage collection, and even objects. Lisp suffers from some problems in standardization, partly due to the ease with which programmers can extend the core language. This problem was addressed in the 1980's with the definition of a standard: *Common Lisp*; however there are a small number of other dialects of the language that have a popular following. Of these *Scheme* is probably the most well known; being somewhat smaller and more logically defined than Common Lisp it is easier to learn. In fact, Scheme-based APIs exist for two of the most common geometric modelling systems, AutoCAD and ACIS.

Given the very different control structures of the two languages, mechanical translation of application programs between C and Lisp is likely to be difficult.

Prolog

Prolog is the classic example of what is known as a *logic programming* language. Whereas the standard procedural languages (like C) work by altering the values of program variables, and (pure) functional languages work by computing results, logic programming languages are designed to elicit facts from a database of rules and other facts. Calculation tends to occur within a logic program as a side-effect, and assignment of values to variables occurs by proclaiming that they have some new value. Prolog implementations tend to be weak in numerical calculation, and therefore it seems unlikely that a modeller supporting D̈jinn would ever be implemented in Prolog. However that does not exclude the possibility that a Prolog program might use a D̈jinn interface to communicate with a solid modelling system.

Details of the C binding

The next chapter describes the representation of D̈jinn objects. That is followed by chapters listing the functions in the C binding, giving the meanings and types of the arguments and values returned. The layout follows Parts I and II, and the descriptions from Part II are also briefly recapitulated, but with additional comments where appropriate. The pre-conditions and post-conditions from Part II are not repeated here. The function arguments also essentially follow the order given in Part II; input arguments are listed first, followed by input/output arguments, and then output arguments. Input arguments are declared constant (`const`) to

enable compilers to apply appropriate optimizations. For consistency, all input/output arguments and output arguments are passed by address, even in the cases where this is not strictly necessary (because the argument is itself a suitable pointer).

Each D̈jinn function returns a value of type `DjError`; this is not made explicit in the remainder of Part III, and the function names appear at the *start* of each appropriate line, for the sake of emphasis.

Writers of D̈jinn-compliant geometric modellers can obtain a template version of the file `djinn.c` containing null versions[2] of all these functions from the D̈jinn web site at `http://www.bath.ac.uk/~ensab/GMS/Djinn`.

[2] A null function is one that returns the error 'not implemented'.

Part III
B
Djinn objects

A C D̈jinn implementation must export to the application program a number of data-types, constants, and function definitions. This is done by means of a single standard header file, `djinn.h`, which can be read into an application program in the usual way (i.e. by including the line '`#include <djinn.h>`' within the program code). A version of `djinn.h` intended for writers of D̈jinn-compliant geometric modellers can be obtained from the D̈jinn web site at `http://www.bath.ac.uk/~ensab/GMS/Djinn`.

The `djinn.h` file, and all other D̈jinn files that it includes must start and end with include guards:

```
#ifndef DJ_DJINN
#define DJ_DJINN
     /* Rest of the djinn.h header file */
#endif
```

The names of the guards that are used by any other included files should consist of the characters `DJ_` followed by the name of the included file (without the `.h`), all in capital letters.

Exported constants

D̈jinn implementations must export definitions of the constants `DjMaxInteger`, `DjMaxReal`, `DjSmallReal` and `DjRealPrecision`. The C names for these constants are as above.

An application programmer using an implementation of D̈jinn in C should be able to assume that these numbers are at least as good as would be found if the implementation used 16-bit signed integers for `DjInteger`,

and used IEEE arithmetic with normalized 32-bit numbers for `DjReal`. Integers are assumed to be signed unless explicitly declared as unsigned.

Visible data types

Djinn requires there to be four exported data types with visible appearance. These map to C data types with the same names as follows:

`DjBoolean:` The C Djinn binding requires that this is one of the standard integer types, or an enumerated type. Following the standard rules for C, a value of this type corresponds to logical false if it is zero, and logical true otherwise. Thus the normal C arithmetic and logical operations can be applied in the usual way.

`DjInteger:` The C Djinn binding requires that this is one of the standard integer types. As mentioned above, `DjMaxInteger` $\geq 2^{15} - 1$, and it is also required that a value of type `DjInteger` must be capable of being assigned to a variable of type `unsigned long` on any machine on which Djinn is implemented in C without the value being disturbed.

`DjReal:` It is expected that the type `DjReal` will correspond to the C type `double` in any implementation of Djinn in C, but this is *not* a requirement. (This is to allow the use of high-precision non-ANSI C floating-point types where they exist, although such use would clearly not be portable.) In any case, the assumption that the underlying arithmetic is as good as that given by the IEEE standard with 32-bit normalized quantities applies. (Most modelling applications will use at least a 64-bit word length for modelling purposes and thus far exceed this minimum requirement.)

`DjString:` C does not really have a natural built-in type for representing strings. The nearest it comes is with the type `char *`, which provides a pointer to the beginning of an array of characters, and this is commonly used for passing strings by using arrays of characters that are terminated with a null (ASCII 0) character. `DjString` is therefore defined to be a pointer type in exactly this way. Note that all strings that are inputs to C Djinn functions are owned by the calling code: if Djinn needs to store the string it will make its own copy. Similarly, all strings that are output are owned by Djinn data structures, and if calling programs want to keep a copy they must make their own copy.

There are also three types that do not operate using the standard arithmetic operators:

`DjLogic`: In C D̈jinn this is an integer or enumerated type with predefined constant values `DjTrue`, `DjUnknown` and `DjFalse`. The application programmer should make no assumption about the numerical values of the constants; therefore, if `test` is a variable of type `DjLogic`, then the application programmer should not write (say) '`if ( test ) ...`' but '`if ( test==DjTrue ) ...`' instead.

`DjVector`: The type `DjVector` is an arbitrary sequence of not more than $n+1$ numbers of type `DjReal`, where n is the dimension of the space in which D̈jinn is working. This is supplied in the C D̈jinn binding by a structure containing (at least) a count of how many elements are in the sequence, and an array of a suitable number of `DjReal`s, all `typedef`ed to the type `DjVector`. Thus, the header file `djinn.h` contains a type definition like

```
typedef struct {
     DjInteger size;
     /* number of elements in the vector */
     DjReal v[DjVMSize];
     } DjVector;
```

Here `DjVMSize` is a small constant (given in `djinn.h`) that defines how many elements the vector can contain. Few applications should need to directly alter the data fields of a `DjVector`, and those that do should not assume that these are the only fields in the structure, or that they follow this order.

`DjMatrix`: The type `DjMatrix` is defined in Part I to be a sequence of `DjVector`s of arbitrary length. In C D̈jinn this is achieved by treating the elements as a rectangular matrix, and storing them in a structure which contains a two-dimensional array:

```
typedef struct {
     DjInteger size1, size2;
     /* number of elements in the matrix */
     DjReal m[DjVMSize][DjVMSize];
     } DjMatrix;
```

Again, application programs should not assume that these are the only fields in the structure, or that they follow this order.

The D̈jinn definition is written in such a way as to encourage elegant implementations in languages that have a more elegant treatment of arrays than C. We could have defined `DjMatrix` to be exactly a sequence of `DjVector`, but this would imply a performance penalty that is likely to be unacceptable to many application programmers.

`DjFaceting`: The C implementation of this type makes a number of changes from the simplistic, in order to achieve reasonable efficiency. We note that facetings always occur in sequences and therefore implement {`sequence of DjFaceting`} as a number of arrays, rather than providing a variable length `struct` for a single `DjFaceting`.

First, the coordinates are stored in a separate array, so that points which are referenced more than once can be stored without duplication. This wastes a little space for edge facetings, but saves a great deal for face and solid facetings.

The sequences of vertices thus consist of sequences of subscripts into this `coords` array. These sequences are concatenated into a `vertices` array.

Lastly, an `index` array contains the final subscript of each sequence. Because the sequences are concatenated, we know that the first sequence starts at the beginning of the `vertices` array and that thereafter each sequence starts at the slot after the end of the previous one. Because the set may contain facetings of different manifold dimensions `Mdims`, we put an array of manifold dimensions beside the index, accessed using the same subscript value.

```
DjInteger Mdims[];
DjInteger index[];
DjInteger vertices[];
DjReal coords[][DjVMSize];
```

Where we need to implement a {`sequence of DjLabel`} to associate with the facetings, an additional array

```
DjLabel labels[];
```

is accessed by the same subscript as the `index` and `Mdims` arrays.

Figure 28 illustrates a pyramid whose surface has two cells, one for the base and one for all the sloping faces taken together. There are therefore just four edge cells and four vertex cells.

The data for the faceting of this object might be

```
DjInteger Mdims[] = {0,0,0,0,1,1,1, 1, 2, 2, 3};

DjInteger index[] = {0,1,2,3,5,7,9,11,15,23,28};

DjInteger vertices[] = {
0, 1, 2, 3, 0,1, 1,2, 2,3, 3,0,
0,3,1,2,
0,1,4,2,3,3,4,0,
```

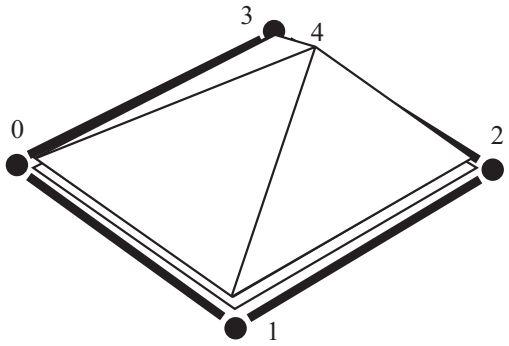


Figure 28: Faceting of a pyramid.

```
0,1,4,3,2 };

DjReal coords[][DjVMSize] = {
{0, 0, 0},
{1, 0, 0},
{1, 1, 0},
{0, 1, 0},
{.5,.5,1}};
```

This illustrates that the list of vertices may hold points which are not point cells, and the use of repeated entries to get the list of vertices round a corner.

`DjConics`: In this case, there is no advantage in accessing the points indirectly, because there is very little sharing. However, the use of an `index` array is still relevant, and the same conventions apply. We know that the manifold dimension of each cell is 1 and that the spatial dimension is 2, and so there is no need to represent either of these explicitly.

```
DjInteger index[];
DjReal conicdata[][5];
```

The five reals of each row of the `conicdata` array are

x_v, y_v	the coordinates of the start point.
x_t, y_t	the coordinates of the tangent intersection point.
s	the shape parameter.

Again, when we require a `sequence of DjLabel` to associate with the `DjConics`, the $1:1$ correspondence is achieved by using the same subscript into an additional array

```
DjLabel labels[];
```

as into the `index` array.

Hidden data types

The D̈jinn API defines a number of data-types with an internal structure hidden from the applications programmer.

DjObject

The D̈jinn type `DjObject` is used to store all geometric information within D̈jinn. In C, `DjObject` is a pointer type, so that when a variable of this type is declared it will not be associated with any data. To associate data with a `DjObject` variable we can use any function that creates an object, such as

```
Dj_Create_Point(3, x, 0, &a);
```

where `a` is of type `DjObject`. C D̈jinn assumes that the implementation will include sensible storage management, so that if `a` had already pointed to the data for some D̈jinn object, then the storage for that original data will be reclaimed by such a call.

Note carefully that C does not guarantee that a variable, such as `a` in the example above, will have any particular value when declared. Thus the example above may fail in mysterious ways if `a` is not set to some safe value before use. Therefore all variables of the type `DjObject` should be initialized to the value 0 on declaration, as in:

```
DjObject a = 0;   Dj_Create_Point(3, x, 0, &a);
```

C D̈jinn implementations may wish to declare a C macro for forming a variable of type `DjObject`, such as:

```
#define Djc_Declare_Object(v)     DjObject v = 0;
```

Languages such as C++, which allow default initializations of data, will not need such approaches.

Labels

In C, D̈jinn variables of the type `DjLabel` are guaranteed to be relatively short, in the sense that it is always possible to cast a label value to the C type `void *` without loss of precision. (This is so that common C libraries for creating object databases and the like may be used on values of these types.)

On the other hand, as they are of unbounded size the types `DjLabelSet` and `DjLabelSequence` are pointers. It is not necessary for an applications programmer to manipulate directly the data structures pointed to

by variables of these types, as they are generated by the relevant Djinn calls. However the Djinn implementation has no way of knowing when any storage for this data is no longer required, and so it is up to the application programmer to tell Djinn (by calling `Dj_Delete_Label_Set` or `Dj_Delete_Label_Sequence`).

A variable of type `DjLabelSet` can be made to point to the empty set or sequence by setting its value to one of the constants `DjEmptyLabelSet` or `DjEmptyLabelSequence`. For the same reasons as mentioned above, one should not pass an unassigned variable of these types to any C Djinn function.

Other hidden types

The Djinn types `DjError`, `DjPoint`, `DjPlane`, `DjTransform`, and `DjPrecision` are defined in C Djinn to be structures of hidden type.

Structures of these hidden types will be passed to C Djinn functions as entire structures when they are input variables. This may seem wasteful; however modern computers are quite good at passing (relatively) large structures to functions; many modern compilers will automatically detect that the data is not going to be altered, and pass a pointer across the calling stack instead. All input variables in `djinn.h` are declared as `const` to help compilers optimize in this way.

Other types

A few Djinn functions require parameters that are not any of the Djinn types given above. One is an enumerated type, which forms an output parameter to the function `Dj_Get_Shape`, and that is simply the type of a primitive shape. For this the structure `DjShapeType` is defined (in `djinn.h`), and can contain the values `DjEmptyType`, `DjPointType`, `DjLineType`, `DjPlaneType`, `DjSphereType`, `DjCylinderType`, `DjConeType`, `DjTorusType`, `DjCyclideType`, or `DjComplicatedType`. Several others are simply lists (or sets) of one of the standard Djinn types to be used as input to a function; this is done by first giving the length of the list (as a parameter of type `DjInteger`) and then following it by a C array of the elements of the list or set. (In the standard C fashion, this array is identified by the address of its first element.)

Output arguments are more complicated; the calling program passes an array into which the elements to be output are written. However the calling program does not necessarily know *a priori* how many elements will be returned. To deal with this the calling program passes an integer

to the routine as an input/output argument—that is, the calling program passes the *address* of the integer. By setting the integer to the length of the list for the output arguments the called routine can ensure that it does not overwrite the array bounds, and the called routine can indicate how many elements were available to be written in total. Thus the calling program can then, if necessary, call the function again with a bigger array.

There are also some functions that pass back not just a single array, but an array of arrays. For these the calling program has to pass two arrays; one to hold the addresses of the individual arrays, and one to hold the data for the arrays. With each array is associated a size, which the calling program sets to indicate maximum size, and the called routine updates it to indicate the quantity of data.

Finally, there are four classes of functions that require different data. One of these is a pair of functions for reading and writing to a stream, for which the C standard library type `FILE *` is used. Another is the class of integration functions, which require a function to be integrated over some geometric entity, for which a standard C function type suffices. The third is the 'compute z-buffer function', `Dj_Surface_Points`, which requires arrays to be passed. However as these arrays are just rectangular arrays of given dimensions these can be passed in the usual C fashion. Finally, the data types `DjFaceting` and `DjConic` are not explicitly defined in the C binding, but are represented by suitable arrays and other variables holding the necessary data.

Part III
1
Primitive objects and copying

Creation and interrogation of simple geometry

`Dj_Create_Point` `(const DjInteger n, const DjVector coords, const DjBoolean finite, DjPoint *pt)`

This function creates the point `*pt` of hidden D̈jinn type in a space of dimension `n` from its Euclidean coordinates `coords`. When `finite` is false, the point is created at infinity in the direction of vector `coords`. (Points at infinity in opposite directions are the same point.)

`Dj_Get_Point_Coordinates` `(const DjPoint pt, DjInteger *n, DjVector *coords, DjLogic *b)`

This function extracts the Euclidean coordinates `*coords` from `pt` of hidden D̈jinn type. The logical result `*b` indicates whether the point is finite (`*b = DjTrue`), infinite (`*b = DjFalse`) or indeterminate (`*b = DjUnknown`). The dimension of `pt` (i.e. the number of coordinates) is given by `*n`.

`Dj_Create_Plane` `(const DjInteger n, const DjVector coords, DjPlane *pln)`

This function creates the open linear halfspace `*pln` of hidden D̈jinn type from its `n + 1` hyper-plane coordinate vector `coords`. Such half-spaces include a three-dimensional half-space bounded by a plane, a two-dimensional half-plane bounded by a straight line and a semi-infinite interval of the real line. If all coordinates of `coords` except the first are zero,

`*pln` is either the empty set or the entire `n`-dimensional space excluding points at infinity.

`Dj_Get_Plane_Coordinates` `(const DjPlane pln, DjInteger *n, DjVector *coords, DjLogic *b)`

This function extracts the `*n + 1` plane coordinates `*coords` from the `*n`-D halfspace `pln` bounded by a plane (see specification of `Dj_Create_Plane`). The logic result `*b` indicates whether that plane is finite, the plane is at infinity, or indeterminate.

Object creation

`Dj_Create_Empty_Object` `(DjObject *obj)`

This function changes the well-defined Djinn object `*obj` into an object with no cells.

Simple primitive objects

`Dj_Create_Point_Object` `(const DjPoint pt, DjObject *obj, DjLabel *l)`

This function creates the Djinn object `*obj` consisting of a single cell, labelled `*l`, with a point-set consisting of the single point `point`, possibly at infinity. The point `point` and thus the object `*obj` may exist in any spatial dimension but the manifold dimension of the cell is always zero.

`Dj_Create_Line_Object` `(const DjPoint pt, const DjPoint qt, DjObject *obj, DjLabel *l)`

This function creates the Djinn object `*obj` with two cells: the cell `*l` with a point-set that is the open straight-line segment from point `pt` to point `qt`, and a cell consisting of points `pt` and `qt`. Points `pt` and/or `qt` may be at infinity and in a space of any dimension, but the manifold dimension of cell `*l` is always one.

`Dj_Create_Plane_Object` `(const DjPlane pln, DjObject *obj, DjLabel *l)`

This function creates the D̈jinn object `*obj` consisting of two cells; cell `*l` with a point-set that is the open planar halfspace `pln`, and a cell consisting of its bounding plane. The spatial dimension of object `*obj` is that of halfspace `pln`, and this dimension is also the manifold dimension of cell `*l`. Halfspace `*pln` may be empty or the entire space less points at infinity.

Simple object interrogation

`Dj_Get_Point` `(const DjObject obj, DjPoint *pt)`

This function extracts the point `*pt` of hidden D̈jinn type from the object `obj`.

`Dj_Get_Line_Segment` `(const DjObject obj, DjPoint *pt, DjPoint *qt)`

This function extracts the end-points from the line-segment object `obj`.

`Dj_Get_Plane` `(const DjObject obj, DjPlane *pln)`

This function extracts the halfspace `*pln` of hidden D̈jinn type from the object `obj`. `*pln` may be a halfspace of 1, 2 or 3 dimensions, possibly the universal set without points at infinity, but it may not be the empty set.

Canonical primitive objects

`Dj_Create_Simplex` `(const DjInteger n, DjObject *obj, DjLabel *l)`

This function creates the D̈jinn object `*obj` consisting of two cells; cell `*l` with an open point-set that is a standard solid simplex of dimension `n` (i.e. a tetrahedron, triangle or interval) and a cell with a point-set that is the entire boundary of that simplex. Object `*obj` has spatial dimension `n` which is also the manifold dimension of the simplex cell `*l`. Its vertices are the origin and unit points on the `n` coordinate axes. Arbitrary simplices may be created by subsequent transformation.

`Dj_Create_Cube (const DjInteger n, DjObject *obj, DjLabel *l)`

This function creates the D̈jinn object `*obj` consisting of two cells; cell `*l` with an open point-set that is a solid unit hyper-cube of dimension `n` and a cell with a point-set that is the entire boundary of that hyper-cube. The object may be a cube, a square, or a one-dimensional interval, with three- two- and one-dimensional points respectively, i.e. `n` is the spatial dimension of the object and the manifold dimension of cell `*l`. Rectangular boxes of arbitrary size may be created by subsequent differential scaling.

`Dj_Create_Sphere (const DjInteger n, DjObject *obj, DjLabel *l)`

This function creates the D̈jinn object `*obj` consisting of two cells; the cell `*l` with an open point-set that is a solid unit-diameter hyper-sphere of dimension `n`, centred on the origin, and a cell with a point-set that is the entire boundary of the hyper-sphere. The object may be a solid sphere (i.e. a ball), a circular disc, or a one-dimensional interval, with three-, two- and one-dimensional points respectively, i.e. `n` is the spatial dimension of the object and the manifold dimension of cell `*l`. Balls of arbitrary size and hyper-ellipsoids may be created by subsequent differential scaling.

Special-purpose primitive objects

`Dj_Create_Cone (const DjReal c, DjObject *obj, DjLabel *l)`

This function creates the D̈jinn object `*obj` usually consisting of two cells; cell `*l` with an open point-set that is a solid frustum of a unit right circular cone, and a cell with a point-set that is the entire boundary of that cone. The cone axis is the z-axis and the top and bottom planes of the frustum are equidistant from the xy-plane. The cone's height and the xy-plane cross-section diameter are unity. The tangent of half the cone apex angle is `c`. Its value determines the type of object created:

$\mathtt{c} = 0$	Cone is a right circular cylinder.
$\lvert\mathtt{c}\rvert = 1$	Cone apex is part of the boundary cell.
$\lvert\mathtt{c}\rvert > 1$	Cone apex constitutes a third cell.
$0 < \mathtt{c} < 1$	Top diameter is less than bottom diameter.
$0 > \mathtt{c} > 1$	Top diameter is greater than bottom diameter.

Cones of arbitrary size and those with elliptical cross-sections may be created by differential scaling.

`Dj_Create_Pyramid` `(const DjInteger n, const DjReal c, DjObject *obj, DjLabel *l)`

This function creates the D̈jinn object `*obj` consisting of two cells; cell `*l` with an open point-set that is a solid pyramid and a cell with a point-set that is the entire boundary of that pyramid. The number of sides of the base is `n`. The pyramid is the cell of which the inscribed cone is specified by the variable `c` as defined for `Dj_Create_Cone`. Regular prisms can be created as pyramids with zero apex angle. Certain non-regular prisms can be created by differential scaling. Triangular prisms can be also created by sweeping a triangular simplex.

`Dj_Create_Cyclide` `(const DjReal d, const DjReal e, DjObject *obj, DjLabel *l)`

A torus is a volume of revolution formed by revolving a disc about an axis in its plane. If the disc changes radius as it rotates such that each point on the circumference of the disc traces a circle, the swept volume is a cyclide. If two circular cross-sections of a cyclide overlap, then the cyclide is a spindle cyclide; its surface is self-intersecting and there is no central hole. Points in the region of overlap inside the "spindle" do not constitute part of the solid D̈jinn cyclide's point-set. This function creates the stratified D̈jinn object `*obj` consisting of two or three cells; cell `*l` with a point-set that is a solid cyclide, and a cell with a point-set that is the boundary of that cyclide less the singular points of its boundary. These points, if they exist, constitute the third cell. The locus of the centre of the circular cross-section is the unit diameter circle in the xy-plane centred at the origin. The cross-section is defined by the diameters `d` and `e` of the two zx-sections. More general cyclides can be created by scaling.

`Dj_Create_Helix` `(const DjInteger n, const DjReal c, DjObject *obj, DjLabel *l)`

This function creates the D̈jinn object `*obj` consisting of two cells; the cell `*l` with a point-set that is a one turn of a planar spiral (`n` $= 2$) or a tapered helix on the surface of a conical frustum (`n` $= 3$) and a cell consisting of its two end-points. The axis of the spiral or helix is the z-coordinate axis and its start point is negative in x and z and zero in y, the radius varying from $(1 - c)/2$ to $(1 + c)/2$. Since the frustum is specified by `c`, as described in the specification of `Dj_Create_Cone`, the helix has unit pitch and unit diameter in the xy-plane. A spiral or helix with an arbitrary number of turns can be created using D̈jinn functions to copy, scale, translate and

union D̈jinn objects. Differential scaling can be used also to change pitch and diameter.

`Dj_Create_NURBS_Curve` `(const DjInteger d, const DjInteger n, const DjInteger m, const DjVector p[], const DjReal t[], const DjReal w[], DjObject *obj, DjLabel *l, DjLogic *b)`

This function creates the D̈jinn object `*obj`, usually consisting of two cells (i.e. wherever the curve has no singular points); the cell `*l` with a point-set that is a NURBS curve of degree `d` and a cell consisting of its two end-points. The curve is specified by `n` weighted control points `p` and a parameter knot vector of `m` elements, possibly with repeated elements. Its coordinate dimension is that of the points `p`.

`Dj_Create_NURBS_Surface` `(const DjInteger a, const DjInteger b, const DjInteger n, const DjInteger m, const DjInteger r, const DjInteger s, const DjVector *p[], const DjReal u[], const DjReal v[], const DjReal w[], DjObject *obj, DjLabel *l)`

This function creates the D̈jinn object `*obj` usually consisting of two cells (i.e. wherever the surface has no singular points); the cell `*l` with a point-set that is a NURBS surface of parametric degrees `a` and `b` and a cell consisting of its boundary curve unless it is a closed surface. The surface is specified by `n`×`m` weighted control points `p` and a parameter $r \times s$ knot array possibly with repeated elements. Its spatial dimension is that of the points `p`.

Copying objects

`Dj_Reset_Modification_Record` `(DjObject *obj)`

This function resets the modification count of object `*obj` to zero and the update region to empty.

`Dj_Copy` `(const DjObject in, DjObject *out)`

This function creates object `*out` with the same cellular structure, labels and attributes as object `in`. No data is shared between the new and original objects.

Part III
2
Partitioning and related operators

Dj_Partition `(const DjObject obj2, DjObject *obj1)`

This function modifies object `*obj1` by partitioning its cells with the cells of `obj2`; the point-set of `*obj1` remains unchanged and `obj2` is unaffected. The point-sets of the cells of the result are:

- Those parts of `*obj1`'s cells outside the point-set of `obj2`.
- Cells created by the pairwise set-intersection of cells from each operand, i.e. the cells computed by the Djinn intersection function.

Dj_Departition `(const DjLabelSet s, DjObject *obj)`

This function removes all cells of object `*obj` in the set `s` that separate other cells of `*obj`. The separated cells are coalesced together and with their separators, thus leaving the point-set of `*obj` unchanged.

Dj_Subdivide_Boundary `(const DjLabelSet s, const DjInteger d, DjObject *obj)`

This function subdivides each cell of object `*obj` in the set `s` into component cells with geometric continuity of order C^{d}. All boundaries common to the same set of these new cells also become a new cell, similarly for boundaries common to sets of all these new cells and so on. For example, if `d` = 0, each connected component of each cell in `s` becomes a new cell and boundaries of the original cell that now bound distinct cells are also subdivided (i.e. the frontier condition is preserved—see Part I, Chapter B). The same effect occurs when `d` = 1 but, in addition, each line of tangent plane discontinuity within a face cell becomes a new edge cell

that separates new cell subdivisions of the original cell. Also, each point of tangent discontinuity within an original or new edge cell becomes a new vertex cell that separates new cell subdivisions of that edge.

Dj_Drain `(const DjLabelSet s, DjObject *obj)`

This function removes each cell in set `s` from object `*obj`, provided it is not the boundary of another cell. For example, this function could remove the interior solid cell from a primitive sphere with two cells, its interior and its bounding surface, leaving just the surface.

Dj_Fill `(const DjLabelSet s, DjObject *obj, DjLabel *c)`

This function creates a (possibly disconnected) bounded cell `*c` with boundary points lying in cells of set `s`. The new cell `*c` has a manifold dimension equal to the spatial dimension of the object `*obj`. Original cells that (partially) bound the new cell are subdivided into their connected components, if that is necessary to preserve the frontier condition, i.e. the boundary points of `*c` are the union of other cells' point-sets. All cells of the original object are retained intact or subdivided in the modified object. Thus, some cells may bound two disconnected pieces of the new cell, or bound the new cell on each side without disconnecting it. For example, if `*obj` is three-dimensional, a three-dimensional solid is created, and if `*obj` is two-dimensional, a two-dimensional planar region is created; a solid sphere is created from a spherical shell, and a triangular region is created from three intersecting line segments.

Part III
3
Managing cells

Cell ancestry

`Dj_Child_Cells` `(const DjObject obj, const DjLabel l, const DjBoolean b, DjLabelSet *s)`

This function returns a set `*s` of labels of cells derived from a cell with label `l` during the most recent object geometry modification. The label `l` is a cell of the first operand of the modification function if `b` is TRUE, and the second if it is FALSE. If cell `l` disappeared during that operation, `*s` contains the single label OUT. If `l` is OUT, `*s` contains labels of all cells entirely derived from one operand.

`Dj_Parent_Cells` `(const DjObject obj, const DjLabel l, DjLabelSequence *r, DjLabelSequence *s)`

This function returns sequences `*r` and `*s` of labels of cells from which cell `l` derives during the most recent object geometry modification. Corresponding elements of `*r` and `*s` correspond to cells of the first and second operands respectively that were combined to form cell `l`. If the most recent operation had one object operand, `*s` contains only the label OUT. If object `obj` was created by that operation, `*r` and `*s` are empty. If `l` is OUT, then `*r` and `*s` contain labels of all cells destroyed by that operation.

`Dj_Modified_Cells` `(const DjObject obj, DjLabelSet *s)`

This function computes all cells `*s` of object `obj` whose geometry was changed by the most recent object geometry modification function.

```
Dj_Modifying_Cells  (const DjObject obj,
   const DjLabelSequence *r, const DjLabelSequence *s)
```

This function computes the sequences `*r` and `*s` of cells of the objects that were operands of the most recent geometry modification function. Cells of the object `obj` that were changed by that function were derived from the corresponding cells in `*r` and `*s`.

Label-set manipulation

```
Dj_Add_Cell_to_Label_Set  (const DjLabel c,
   DjLabelSet *s)
```

This function adds the label `c` to the set `*s`.

```
Dj_Label_Set_to_Sequence  (const DjLabelSet r,
   DjLabelSequence *t)
```

This function converts the set of labels `r` to the sequence `*t` containing the same labels.

```
Dj_Label_Sequence_to_Set  (const DjLabelSequence r,
   DjLabelSet *t)
```

This function converts the sequence of labels `r` to the set `*t` containing the same labels.

```
Dj_Unite_Label_Sets  (const DjLabelSet r,
   const DjLabelSet s, DjLabelSet *t)
```

This function unites sets of labels `r` and `s` to form set `*t`.

```
Dj_Intersect_Label_Sets  (const DjLabelSet r,
   const DjLabelSet s, DjLabelSet *t)
```

This function creates set `*t` from the common labels of sets `r` and `s`.

`Dj_Subtract_Label_Sets` `(const DjLabelSet r, const DjLabelSet s, DjLabelSet *t)`

This function creates the set `*t` from those labels of set `*r` that are not in set `*s`.

`Dj_Delete_Label_Set` `(DjLabelSet *s)`

This function removes all labels from set `*s`.

`Dj_Delete_Label_Sequence` `(DjLabelSet *s)`

This function removes all labels from sequence `*s`.

`Dj_Get_Label` `(const DjLabelSequence s, const DjInteger i, DjLabel *l, DjLogic *t)`

This function returns the `i`th label `*l` of sequence `s`. The logical variable `t` indicates whether an `i`th label exists.

`Dj_Get_Cardinality` `(const DjLabelSet s, DjInteger *n)`

This function returns the number `*n` of labels in set `s`.

Attribute manipulation

`Dj_Set_Attribute` `(const DjLabel l, const DjString t, const DjInteger s, DjObject *obj)`

This function assigns the value `s` to attribute `t` of cell `l` of object `*obj`. If `s` is zero, then the attribute of type `t` is removed from cell `l`.

`Dj_Get_Attribute` `(const DjObject obj, const DjLabel l, const DjString t, DjInteger *s)`

This function returns the value `*s` of attribute `t` of cell `l` of object `obj`, or zero if that attribute does not exist. If `l` = OUT, then the attribute of the region outside the object is returned.

`Dj_Get_Attribute_Types` `(const DjObject obj, DjInteger *n, DjString *s[])`

This function returns the attribute types `*s` of object `obj`. The variable `*n` is set by the calling program to the size of the array `*s[]`, and at most that many types are returned in `*s[]`. (The function sets `*n` to the total number of attributes associated with `obj`. Thus the calling program can determine whether it needs to call the routine again with a larger sized array `*s[]`.)

`Dj_Delete_Attribute_Type` `(const DjString t, DjObject *obj)`

This function removes all attributes of type `t` from object `*obj`. The attribute of type `t` can be removed from a single cell using the function `Dj_Set_Attribute` to assign the null attribute zero.

Cell selection

`Dj_Small_Cells` `(const DjObject obj, const DjReal d, const DjInteger k[], const DjInteger n, DjLabelSet *s)`

This function selects all cells `*s` of object `obj` whose dimensionality is one of the integers in `k[]`, and which are smaller than length `d`.

`Dj_Point_Proximate_Cells` `(const DjObject obj, const DjPoint p, const DjInteger k[], const DjInteger n, const DjReal tol, DjLabelSequence *s, DjPoint *q)`

This function computes the cells `*s` that are closest to point `p` whose dimensionality is one of the integers in `k[]`. The point `*q` is the point in the first cell of `*s` closest to point `p`. The distance of `p` to each cell of `*s` is greater than the distance between `p` and `*q`, but within `tol` of it.

`Dj_Line_Proximate_Cells` `(const DjObject obj, const DjPoint pt, const DjPoint qt, const DjInteger k[], const DjInteger n, const DjReal tol, DjLabelSequence *s, DjPoint *q)`

This function computes the cells `*s` that are closest to a point on the unbounded line through points `pt` and `qt`, whose dimensionality is one of the integers in `k[]`. The point `*q` is the closest point in the first cell of `*s`. The distance of the line to each cell of `*s` is greater than the distance from the line to `*q` but within `tol` of it.

`Dj_Plane_Proximate_Cells` `(const DjObject obj, const DjPlane pln, const DjInteger k[], const DjInteger n, const DjReal tol, DjLabelSequence *s, DjPoint *q)`

This function computes the cells `*s` that are closest to a point on the unbounded plane `pln`, whose dimensionality is one of the integers in `k[]`. The point `*q` is the closest point in the first cell of `*s`. The distance of the plane to each cell of `*s` is greater than the distance between `pln` and `*q` but within `tol` of it.

`Dj_Get_Boundary_Cells` `(const DjObject obj, const DjLabel c, DjLabelSet *s)`

This function computes the set of labels `*s` of cells that bound cell `c` of object `obj`. However, bounding cells are excluded from `*s` if they bound other boundary cells. For example, if `obj` is a solid cube with its faces and edges represented as distinct cells, and C is its interior, then only the faces are returned in S. If `c` is OUT, it identifies the object exterior and `*s` contains boundary cells of object `obj`.

`Dj_Get_Bounded_Cells` `(const DjObject obj, const DjLabel c, DjLabelSet *s)`

This function computes the set of labels `*s` of cells that are (possibly partially) bounded by cell `c` of object `obj`. For example, if `obj` is a solid cube with its faces and edges represented as distinct cells, and `c` is an edge, then only the adjacent faces are returned in `*s`, but *not* the cube interior. If `c` is a boundary cell of object `obj`, `*s` includes OUT to represent the object exterior.

`Dj_Get_Orientation` `(const DjObject obj, const DjLabel c, const DjLabel d, DjLogic *r)`

This function computes `*r` to indicate whether the orientation of cell `c` is consistent with that of cell `d`.

Part III
4
Transformations

`Dj_Concatenate_Transformations` `(const DjTransform r, const DjTransform s, DjTransform *t)`

This function creates the parameterized transformation `*t` which has an effect equivalent to the sequential application of transformations `r` and `s`. If `r` has a lower dimension than `s` (or vice versa), it affects only the first coordinates of points with the dimension of `s`, and `*t` has the higher dimension.

`Dj_Create_Linear_Transformation` `(const DjInteger n, const DjMatrix m, DjTransform *t)`

This function creates the n-dimensional linear transformation `*t` which has an effect equivalent to the homogeneous coordinate transformation matrix `m`. The matrix `*t` is parameterized over [0...1] to a continuous sequence of transformations equivalent to the matrix sequence from the identity to `m`.

`Dj_Create_Quadratic_Transformation`
`(const DjInteger n, const DjMatrix q[], DjTransform *t)`

This function creates the n-dimensional quadratic transformation `*t`, which has an effect on an arbitrary point equivalent to generating the homogeneous point coordinates of the result using the quadratic forms $\mathbf{p}\mathbf{q}[i]\mathbf{p}^{\mathrm{T}}$, $0 \leq i \leq n$, where $\mathbf{p}$ are the homogeneous point coordinates. The matrix `*t` is parameterized over [0...1] to a continuous sequence of transformations equivalent to the matrix sequence from the identity to `q`.

`Dj_Get_Linear_Transformation (const DjTransform t, DjInteger *n, DjMatrix *q)`

This function extracts `*n` and an $n \times n$ matrix `*q` the data which, if used as input to `Dj_Create_Linear_Transformation` would create a transformation identical to `t`.

`Dj_Get_Quadratic_Transformation (const DjTransform t, DjInteger *n, DjMatrix *q[])`

This function extracts `*n` and `*q` which, if used as input to `Dj_Create_Quadratic_Transformation` would create a transformation identical to `t`.

`Dj_Apply_Transformation (const DjTransform t, DjObject *obj)`

This function uses the n-dimensional transformation `t` to change the point-sets of object `*obj` and all its cells; only the first n Euclidean point coordinates are affected. If the spatial dimension of `*obj` is less than n, then it is increased. Bounding relations between cells and cell manifold dimension are unaffected by the transformation.

`Dj_Invert_Transformation (const DjTransform r, DjTransform *t)`

This function creates the parameterized transformation `*t` with the inverse of the effect of transformation `r`.

Rigid-body transformations

`Dj_Create_Translation (const DjPoint u, const DjPoint v, DjTransform *t)`

This function creates the parameterized linear transformation `*t` with an effect on n-dimensional points which is equivalent to their translation along the vector from n-dimensional point `u` to n-dimensional point `v`.

`Dj_Create_Axial_Rotation` `(const DjInteger n, const DjInteger i, const DjInteger j, const DjReal ang, DjTransform *t)`

This function creates the parameterized n-dimensional transformation `*t` to rotate points about a line through the origin perpendicular to coordinate axes `i` and `j`.

`Dj_Create_Alignment` `(const DjPoint p, const DjPoint q, DjTransform *t)`

This function creates the parameterized rotation `*t`, which rotates the vector in the direction of point `p` to that in the direction of point `q`. The parameter is proportional to the angle of rotation. The points need not be the same distance from the origin and either or both points may be infinite.

`Dj_Create_Rotation` `(const DjPoint p, const DjPoint q, const DjReal ang, DjTransform *t)`

This function creates the parameterized transformation `*t` to rotate by a fraction of `ang` radians about the unbounded directed line from point `p` to point `q`.

Affine transformations

`Dj_Create_Scale` `(const DjInteger n, const DjVector v, DjTransform *t)`

This function creates the parameterized n-dimensional linear transformation `*t` to scale differentially by the elements of vector `v` or fractions thereof. Negative scale factors define a transformation that mirrors in a coordinate plane.

`Dj_Create_Mirror` `(const DjPlane pl, DjTransform *t)`

This function creates the parameterized linear transformation `*t` that mirrors points in the oriented plane `pl`.

`Dj_Create_Axial_Shear` `(const DjInteger n, const DjInteger i, const DjInteger j, const DjReal s, DjTransform *t)`

This function creates the parameterized n-dimensional linear shear transformation `*t` to translate points in the direction of `i`th coordinate axis by a fraction of the product of `s` and their coordinate value with respect to the `j`th coordinate axis.

`Dj_Create_General_Shear` `(const DjInteger n, const DjVector u, const DjVector v, const DjReal s, DjTransform *t)`

This function creates the parameterized n-dimensional linear shear transformation `*t` to translate points in the direction of vector `v` by a fraction of the product of `s` and their distance from the origin in direction `u`. Vectors `u` and `v` need not be unit vectors.

`Dj_Create_View_Transformation` `(const DjVector coi, const DjVector vpn, const DjVector vup, DjTransform *t)`

This function creates the parameterized three-dimensional rigid-body transformation `*t`, such that `*t` transforms points from object space to view space. The centre of interest is the point `coi`, and the view plane normal and view up vector are provided by the (not necessarily unit) orthogonal vectors `vpn` and `vup`.

Projective transformations

`Dj_Create_Image_Transformation` `(const DjVector dop, const DjReal r, DjTransform *t)`

This function creates the parameterized three-dimensional projective transformation `*t`, such that `*t` transforms points from view space to image space. The direction of projection is given by the vector `dop` and `r` is the reciprocal of the distance from the centre of projection to the image plane. Perspective projections and various parallel projections such as isometric and cabinet projections can be produced by the concatenation of transformations created using `Dj_Create_View_Transformation` and `Dj_Create_Image_Transformation`. When `r` is zero, image space represents a parallel projection and when `r` is not zero, image space represents

a perspective projection. Further details of specific projections may be found in Part I, Chapter 4.

Quadratic transformations

`Dj_Create_Taper (const DjInteger n, const DjVector v, DjTransform *t)`

This function creates the parameterized quadratic transformation `*t` that modifies the Euclidean coordinates of points in proportion to fractions of the taper factors `v`. The transformation `*t` causes a taper orthogonal to the `n`th coordinate axis.

`Dj_Create_Inversion (const DjInteger n, const DjReal r, DjTransform *t)`

This function creates the parameterized n-dimensional quadratic transformation `*t`, such that `*t` geometrically inverts in a centre of inversion at coordinate `r` on the `n`th coordinate axis. Geometric inversion maps spheres (including planes) to spheres.

`Dj_Create_Bend (const DjInteger n, const DjVector v, DjTransform *t)`

This function creates the parameterized quadratic transformation `*t` that bends objects in proportion to fractions of the bend factors `v`. The transformation `*t` causes planes orthogonal to the `n`th coordinate axis to become paraboloids.

Non-polynomial transformations

`Dj_Create_Twist (const DjReal p, DjTransform *t)`

This function creates the parameterized transformation `*t` that twists objects about the z-coordinate axis in proportion to a fraction of their coordinate in that direction.

Part III
5
Object combination

Dj_Unite `(const DjObject obj2, DjObject *obj1)`

This function modifies object `*obj1` such that its point-set is the conventional (non-regularized) union of the point-sets of operands `*obj1` and `obj2`; object `obj2` is unaffected. The point-sets of the cells of the result are:

- Those parts of `*obj1`'s cells outside the point-set of `obj2`.
- Cells created by the pair-wise set-intersection of cells from each operand, i.e. the cells computed by the Djinn intersection function.
- Those parts of `obj2`'s cells outside the point-set of `*obj1`.

Dj_Intersect `(const DjObject obj2, DjObject *obj1)`

This function modifies an object `*obj1` such that its point-set is the conventional (non-regularized) intersection of the point-sets of operands `*obj1` and `obj2`; object `obj2` is unaffected. The point-set of each cell of the result is the intersection of the point-set of a cell from each operand.

Dj_Subtract `(const DjObject obj2, DjObject *obj1)`

This function modifies an object `*obj1` such that its point-set is the closure of the conventional (non-regularized) difference of the point-sets of operands `*obj1` and `obj2`; object `obj2` is unaffected. The point-sets of the cells of the result are:

- Those parts of `*obj1`'s cells outside the point-set of `obj2`.
- Additional cells added where necessary to complete cell boundaries, whilst preserving the frontier condition discussed in Part I, Chapter B.

Part III
6
Swept objects

`Dj_General_USweep` `(const DjObject obj2, const DjTransform t, const DjLabel q, DjObject *obj1)`

This function modifies object `*obj1` such that its point-set is the continuous union of a family of rigid-body transformations of its original point-set; object `obj2` is unaffected. Each transformation is a translation to a point of `obj2` preceded by the transformation `t`, parameterized by the distance from the point labelled `q` along `obj2` (see Part I, Chapter 4). If all points of the result are derived from a single point of both `*obj1` and `obj2`, then each cell of the result is the sweep of a cell from `*obj1` over another from `obj2`. If such independence does not exist between `*obj1` and `obj2`, each cell of the result contains points that result from the same multi-set of `*obj1`, `obj2` cell pairs (see Part I, Chapter 6).

`Dj_General_ISweep` `(const DjObject obj2, const DjTransform t, const DjLabel q, DjObject *obj1)`

This function is the same as `Dj_General_USweep` with intersection replacing union.

`Dj_Dilate` `(const DjObject obj2, DjObject *obj1)`

This function modifies object `*obj1` such that its point-set is the Minkowski sum of the point-sets of objects `*obj1` and `obj2`; object `obj2` is unaffected. If each point of the result is derived uniquely from a single point of both `*obj1` and `obj2`, each cell of the result is the Minkowski sum of a cell from `*obj1` and another from `obj2`. If such independence does not exist between `*obj1` and `obj2`, each cell of the result contains

points that result from the same multi-set of `*obj1`, `obj2` cell pairs (see Part I, Chapter 6).

Dj_Erode `(const DjObject obj2, DjObject *obj1)`

This function is the same as `Dj_Dilate` with intersection replacing union in the definition of the Minkowski sum. It modifies object `*obj1` such that its point-set is the complementary Minkowski sum of the point-sets of objects `*obj1` and `obj2`; object `obj2` is unaffected. Each cell of the result contains points that result from the same multi-set of `*obj1`, `obj2` cell pairs (see Part I, Chapter 6). If the point-sets of the objects are independent, then the result has an empty point-set.

Dj_Frenet_USweep `(const DjReal ta, const DjReal ts, const DjObject obj2, const DjLabel l, DjObject *obj1)`

This function is a special case of `Dj_General_USweep`. It modifies an object `*obj1` such that its point-set is the is the continuous union of a family of rigid-body transformations of its point-set; object `obj2` is unaffected. Each transformation is a translation to a point of an edge cell of `obj2`, preceded by a rotation about its tangent direction, an orientation by the modified Frenet frame at that point and a scaling. The rotation and scaling are proportional to the distance along the edge cell. If each point of the result is uniquely derived from a single point of both `*obj1` and `obj2`, then each cell of the result is the sweep of a cell from `*obj1` over another from `obj2`. If such independence does not exist between `*obj1` and `obj2`, then each cell of the result contains points that result from the same multi-set of `*obj1`, `obj2` cell pairs (see Part I, Chapter 6).

Dj_Frenet_ISweep `(const DjReal ta, const DjReal ts, const DjObject obj2,const DjLabel l, DjObject *obj1)`

This function is a special case of `Dj_General_ISweep` and is the same as `Dj_Frenet_USweep`, with intersection replacing union.

Dj_Extrude `(const DjReal k, DjObject *obj)`

This function extrudes an object `*obj` of spatial dimension two to unit height in the z-direction with sloping sides. Side slopes represent a scale factor for the x and y point coordinates that varies linearly from 1 to

`k`; the sides do not have a constant draft taper angle. Each cell of `*obj` gives rise to two cells, a cell consisting of disjoint pieces at each end of the extrusion and a cell between those pieces.

`Dj_Spin` `(DjObject *obj)`

This function sweeps an object `*obj` of spatial dimension two around the y-axis by a complete revolution. Each cell of `*obj` is swept to give a cell one dimension higher.

Part III
7
Local changes of shape

Tweaks

`Dj_Small_Tweak` `(const DjLabelSet s, const DjTransform t, const DjReal a, DjReal *b, DjObject *obj)`

If possible, this function transforms the cells belonging to object `*obj` and in set `s` by the parametric transformation `t`, and modifies adjacent cells to maintain bounding relations as described in Part I, Chapter 7. The object is not modified if the specified modification cannot be achieved whilst maintaining all cell adjacency relations. If modification is performed, then `*b` = `a`, otherwise it is an upper bound on the transformation parameter corresponding to such an adjacency-preserving modification.

Blends

`Dj_Convex_Round` `(const DjObject i, const DjReal r, DjObject *obj)`

This function uses a radius of `r` to round the convex regions of that part of object `*obj` which have a manifold dimension equal to its spatial dimension and lie inside the point-set of object `i`. The operation is the convex Minkowski Sum blend defined in Part I, Chapter 7

`Dj_Concave_Round` `(const DjObject i, const DjReal r, DjObject *obj)`

This function uses a radius of `r` to round the concave regions of that part of object `*obj` which have a manifold dimension equal to its spatial dimension and lie inside the point-set of object `i`. The operation is the concave Minkowski sum blend defined in Part I, Chapter 7.

Dj_Convex_Blend `(const DjObject v[], const DjInteger n, const DjInteger c, const DjReal r, DjObject *obj)`

This function subtracts from `*obj` a blend point-set along those convex edges or vertices of that part of `*obj` with maximum manifold dimension which are selected by the `n` pairs of virtual objects `v`. If there are two pairs of virtual objects, a network of convex edges is blended; if there are more than two, convex vertices are blended. Blended edges or vertices are those that are contained by the intersection of the surfaces of all first virtual objects; the second object in each pair defines the region of influence of the first. The blend point-set has fullness parameter `r` and geometric continuity of order `c` with each face adjacent to the blended edge or vertex (see Part 1, Chapter 7).

Dj_Concave_Blend `(const DjObject v[], const DjInteger n, const DjInteger c, const DjReal r, DjObject *obj)`

This function unites with `*obj` a blend point-set along those concave edges or vertices of that part of `*obj` with maximum manifold dimension which are selected by the `n` pairs virtual objects `v`. If there are two pairs of virtual objects, a network of concave edges is blended; if there are more than two, concave vertices are blended. Blended edges or vertices are those that are contained by the intersection of the surfaces of all first virtual objects; the second object in each pair defines the region of influence of the first. The blend point-set has fullness parameter `r` and geometric continuity of order `c` with each face adjacent to the blended edge or vertex (see Part 1, Chapter 7).

Part III
8
Interrogation

Geometric relationships

`Dj_Distance` `(const DjObject obj1, const DjObject obj2, const DjBoolean b, DjPoint *p, DjPoint *q, DjObject *obj3, DjLogic *t)`

This function computes the closest (`b` = TRUE) or furthest (`b` = FALSE) points `*p` and `*q` in objects `obj1` and `obj2` respectively. In the case where multiple closest or furthest points exist (maybe infinitely many), `obj3` represents the result.

`Dj_Directional_Distance` `(const DjObject obj1, const DjObject obj2, const DjVector v, const DjBoolean b, DjPoint *p, DjPoint *q, DjObject *obj3, DjLogic *t)`

This function computes the closest (`b` = TRUE) or furthest (`b` = FALSE) points `*p` and `*q` in objects `obj1` and `obj2` respectively in the direction of vector `v`. In the case where multiple closest or furthest points exist (maybe infinitely many), `obj3` represents the result.

`Dj_Interference` `(const DjObject obj1, const DjObject obj2, const DjReal tol, DjLogic *t)`

This function determines whether interference exists between the objects `obj1` and `obj2` to within a distance tolerance `tol`. It should be significantly faster than the function to compute distance, and computation speed should increase with the magnitude of `tol`.

`Dj_Containment` `(const DjObject obj1, const DjObject obj2, const DjReal tol, DjLogic *t)`

This function determines whether or not object `obj1` contains `obj2`, to within a distance tolerance `tol`. Using this function in conjunction with Dj_Interference, all such relationships between two objects can be determined.

`Dj_Diameter` `(const DjObject obj, const DjVector v, DjPoint *p, DjPoint *q, DjObject *obj2, DjLogic *t)`

This function determines endpoints `*p` and `*q` of the diameter of object `obj`'s point-set in direction `v`. If multiple (possibly infinitely many) diameters exist, the result is returned in `obj2`.

`Dj_Extreme_Diameter` `(const DjObject obj1, const DjBoolean b, DjPoint *p, DjPoint *q, DjObject *obj2, DjLogic *t)`

This function determines endpoints `*p` and `*q` of the extreme diameter of object `obj1`'s point-set. The shortest diameter is computed when `b` is TRUE and the longest when it is FALSE. If multiple (possibly infinitely many) diameters exist, the result is returned in `obj2`.

Topology

`Dj_Genus` `(const DjObject obj, DjInteger *g)`

This function determines genus `*g` of object `obj`. For example, a sphere has genus 0 and a torus has genus 1.

`Dj_Vexity` `(const DjObject obj, const DjLabel c1, const DjLabel c2, DjLogic *vex)`

This function determines convexity or concavity of cell `c1` in the neighbourhood of cell `c2`. If `c1` is OUT, the vexity of the entire object `obj` is determined. If `c2` is OUT, vexity is determined with respect to the natural embedding of `c1`. For example:

- `c1` = OUT and `c2` = OUT ... `*vex` is vexity of `obj`.

- `c1` = OUT and face cell `c2` ... `*vex` is the vexity of `obj` in the neighbourhood of its face `c2`.
- Solid cell `c1` and face cell `c2` ... `*vex` is the vexity of cell `c1` in the neighbourhood of its face `c2`.
- Plane face cell `c1` and `c2` = OUT ... `*vex` is the vexity of the face with respect to the plane.

Here `DjTrue` represents convex, `DjFalse` represents concave.

Object geometry

`Dj_Get_Shape` `(const DjObject obj, const DjLabel l, DjInteger *s, DjInteger *m, DjShape *sh, DjLogic *bnd)`

This function returns the spatial dimension `*s`, manifold dimension `*m` and shape `*sh` of cell `l` of object `obj`. If and only if the cell is bounded, `*bnd` is true. For example, `*bnd` is true for a straight-line segment and false for a planar half-space or unbounded straight line.

The shape `*sh` is complicated when the cell is none of the other categories, all of which apply only to rank deficient cells, i.e. boundary cells. For example, a spherical surface and a two- or three-dimensional circle or a part thereof are all spherical, but a solid sphere or a part thereof is not. Linear and spherical shapes are independent of spatial dimension. For example, a straight line in any space and a plane in three dimensions are all linear.

`Dj_Empty` `(const DjObject obj, const DjReal tol, DjLogic *emp)`

This function determines whether or not the point-set of object `obj` is empty to within a tolerance `tol`.

`Dj_Get_Bounds` `(const DjObject obj, DjPoint *p, DjPoint *q, DjLogic *succ)`

This function computes an axially aligned cuboid that tightly contains the point-set of the object `obj`. The variables `*p` and `*q` are the vertices of the cuboid with minimum and maximum Euclidean coordinates.

`Dj_Get_Modification_Record` `(const DjObject obj, DjInteger *mc, DjPoint *p, DjPoint *q, DjLogic *succ)`

This function returns the number of modifications `*mc` that have been made to the object `obj` since the modification record was reset using the function `Dj_Reset_Modification_Record`. It also returns an axially aligned cuboid that tightly contains object `obj`'s region of change due to those modifications. This includes the point-sets of all cells with changed point-sets, previous point sets of those cells and point-sets of cells that no longer exist. The variables `*p` and `*q` are the vertices of the cuboid with minimum and maximum Euclidean coordinates.

`Dj_Convex_Hull` `(const DjObject obj, const DjReal tol, DjObject *hull)`

This function computes the smallest convex object `*hull` that contains the object `obj` to a resolution `tol`. Points of `hull` have the same coordinate dimension as those with the highest dimension in `obj`. Thus, if `obj` consists entirely of points defined in the xy-plane, `*hull` represents the 2-manifold convex hull in that plane. In contrast, if such a two-dimensional object has been moved into three-dimensional space, then `*hull` is a 3-manifold.

Integral properties

`Dj_Error_Bounds` `(const DjPrecision pr, DjReal *a, DjReal *b, DjLogic *succ)`

This function computes the upper bound `*a` and lower bound `*b` for a Djinn computation that yielded the precision measure `pr`.

`Dj_Error_Estimate` `(const DjPrecision pr, DjReal *err, DjLogic *succ)`

This function computes the error estimate `*err` for a Djinn computation that yielded the precision measure `pr`.

`Dj_Volume` `(const DjObject obj, const DjLabelSet s, const DjReal acc, DjReal *vol, DjPrecision *pr)`

This function computes the volume `*vol` of those cells of object `obj` with labels in the set `s`. The desired relative error bound is `acc` and the accuracy of the result is defined by the precision `*pr` that captures error bounds and expected error.

`Dj_Face_Area` `(const DjObject obj, const DjLabelSet s, const DjReal acc, DjReal *area, DjPrecision *pr)`

This function computes the surface area `*area` of those cells of object `obj` with labels in the set `s`. The desired relative error bound is `acc` and the accuracy of the result is defined by the precision `*pr` that captures error bounds and expected error.

`Dj_Edge_Length` `(const DjObject obj, const DjLabelSet s, const DjReal acc, DjReal *len, DjPrecision *pr)`

This function computes the sum `*len` of the lengths of those one-dimensional cells of object `obj` with labels in the set `s`. The desired relative error bound is `acc` and the accuracy of the result is defined by the precision `*pr` that captures error bounds and expected error.

`Dj_Centroid` `(const DjObject obj, const DjLabelSet s, const DjReal acc, DjPoint *cen, DjPrecision *pr)`

This function computes the centroid `*cen` of those cells of object `obj` with labels in the set `s`. The desired relative error bound is `acc` and the accuracy of the result is defined by the precision `*pr` that captures error bounds and expected error. The dimension of the centroid `*cen` is the highest dimension among point-sets of cells of `obj`. Thus, for example, if all cells are defined in the xy-plane, `*cen` is a two-dimensional point. The centroid `*cen` depends only on the highest dimension cells in set `s`. Thus, for example, if `s` contains faces but no solid cells, `*cen` is the centroid of those faces. Cells of lower dimension have no influence.

`Dj_Principal_Moments` `(const DjObject obj, const DjLabelSet s, const DjReal acc, DjInteger *n, DjMatrix *ax, DjVector *mom, DjPrecision *pr1, DjPrecision *pr2)`

This function computes the `*n` principal moments `*mom` and the corresponding mutually orthogonal unit principal axis vectors `*ax` of those cells of object `obj` with labels in the set `s`. The desired relative error bound is `acc` and the accuracy of the results is defined by the precisions `*pr1` and `*pr2`, that capture error bounds and expected error. The number of principal moments `*n` is the highest dimension among the point-sets of all cells in set `s`. Thus, for example, if all cells are defined in the xy-plane, `*n` is 2. When `*n` is 3, if two of the principal moments are equal, the corresponding principal axes are orthogonal to each other and to the third principal axis, but their directions are otherwise arbitrary. If all three principal moments are equal, the principal axes are all mutually orthogonal, but otherwise arbitrary.

Dj_Scalar_Integral

```
(const DjObject obj,
   const DjLabelSet s, const DjReal acc,
   DjReal *F(const DjPoint), DjReal *igrl, DjPrecision *pr)
```

This function computes the integral `*igrl` of the scalar function `F(const DjPoint )` over those cells of object `obj` with labels in the set `s`. The desired relative error bound is `acc` and the accuracy of the result is defined by the precision `*pr` that captures error bounds and expected error. The dimension n of points in the domain of `*F(const DjPoint)` is the highest dimension among the points of `obj`'s point-set. Thus, for example, if all cells are defined in the xy-plane, `*F(const DjPoint)` maps two-dimensional points to a scalar. The dimension m of integration is the highest manifold dimension among the cells of `s`. Thus, for example, if `s` contains face cells but no solid cells, a double integration is performed over the surfaces; cells of lower dimension contribute nothing to the integral.

Dj_Face_Integral

```
(const DjObject obj, const DjLabelSet s,
   const DjReal acc, DjVector *F(const DjPoint),
   DjReal *igrl, DjPrecision *pr)
```

This function computes the integral `*igrl` of the inner product of the function `*F(const DjPoint)` and the unit 'surface' normal of the 'face' cells of object `obj` in set `s`. The desired relative error bound is `acc` and the accuracy of the result is defined by the precision `*pr` that captures error bounds and expected error. The dimension n of points in the domain of `*F(const DjPoint)` is the highest dimension among the points of `obj`'s point-set. Integration is performed over the cells of dimension $n - 1$. Thus, for example, if all cells are defined in the xy-plane, then

`F(const DjVector )` maps two-dimensional points to a two-dimensional vector and integration is performed over edges.

```
Dj_Edge_Integral (const DjObject obj, const DjLabelSet s,
   const DjReal acc, DjVector *F(const DjPoint),
   DjReal *igrl, DjPrecision *pr)
```

This function computes the integral `*igrl` of the inner product of the function `*F(const DjPoint)` and the unit edge direction of the edge cells of object `obj` in set `s`. The desired relative error bound is `acc` and the accuracy of the result is defined by the precision `*pr` that captures error bounds and expected error. The dimension n of points in the domain of `*F(const DjPoint)` is the highest dimension among the points of `obj`'s point-set. Integration is performed over the cells of dimension 1.

Medial geometry

```
Dj_Medial_Geometry (const DjObject obj,
   const DjLabelSet s, DjObject *objr)
```

This function computes the medial geometry `*objr` of each cell of object `obj` in set `s` (i.e. the medial surface, medial axis and mid-point of solid, face and edge cells respectively). The coordinate dimension of `*objr` is that of `obj`. If `s` contains OUT, then the medial geometry of the space exterior to `obj` is computed. Medial axes and mid-points lie in their faces and edges respectively. Tangent discontinuities in cell boundaries are reflected in the medial geometry, but such a medial geometry discontinuity constitutes a distinct cell only if the boundary discontinuity constitutes a distinct cell.

```
Dj_Medial_Radius (const DjObject obj,
   const DjObject objm, const DjPoint p, DjReal *r,
   DjBoolean *succ)
```

If `objm` is the medial geometry object of `obj`, this function computes the medial radius `*r` at point `p`.

Differential geometry

```
Dj_Face_Normal (const DjObject obj, const DjPoint p,
   const DjLabel l, DjVector *n, DjBoolean *succ)
```

This function computes the unit normal vector `*n` at the point `p` of face cell `l` of object `obj`. The dimension of `*n` is that of `obj`.

`Dj_Face_Curvature` `(const DjObject obj, const DjPoint p, const DjLabel l, DjReal *k1, DjReal *k2, DjVector *d1, DjVector *d2, DjVector *n, DjBoolean *succ)`

This function computes the principal curvatures `*k1` and `*k2` (`*k1` $\leq$ `*k2`), the mutually orthogonal principal directions `*d1` and `*d2`, and the face normal `*n` at the point `p` of face cell `l` of object `obj`. The sign of the curvature reflects convexity: for example, spheres with outward directed normals have negative curvatures.

`Dj_Edge_Direction` `(const DjObject obj, const DjPoint p, const DjLabel l, DjVector *t, DjBoolean *succ)`

This function computes the unit tangent direction vector `*t` at the point `p` of edge cell `l` of object `obj`. The dimension of `*t` is that of `obj`.

`Dj_Edge_Curvature` `(const DjObject obj, const DjPoint p, const DjLabel l, DjReal *k, DjReal *tr, DjVector *t, DjVector *n, DjVector *b, DjBoolean *succ)`

This function computes the curvature `*k`, torsion `*tr`, and the Frenet frame `*t`, `*n`, `*b` at the point `p` of edge cell `l` of object `obj`. The Frenet frame consists of the mutually orthogonal unit tangent `*t`, normal `*n` and binormal `*b` vectors.

Sections

`Dj_Point_Membership` `(const DjObject obj, const DjPoint p, DjLabel *c)`

This function computes the label `*c` of the cell of object `obj` within which point `p` lies. If `p` is outside the object, `*c` is OUT.

`Dj_Ray_Classification` `(const DjObject obj, const DjPoint p, const DjPoint q, DjObject *objr)`

This function computes the object `*objr` that is the set-intersection of object `obj` with the unbounded directed line from point `p` to point `q`. The effect is the same as that of the function `Dj_Intersect` operating on `obj` and an unbounded line object.

`Dj_Surface_Points` `(const DjObject obj, const DjVector p, const DjVector dp, const DjInteger m, const DjInteger n, const DjReal acc, DjVector *buffer, DjVector *normals, DjVector *fa)`

This function intersects a regular array of `m` by `n` parallel rays with the object `obj`. The rays are parallel to the z-coordinate axis and directed in the negative z-direction. Point `p` is the origin of the grid and the spacing is defined by the coordinates of `dp`. The last three arguments define optional arguments that are used to return information about `obj` where it is cut by the grid of rays. Each of these arguments can be set to zero, in which case it is ignored, otherwise the argument is treated as the base address of an `m` by `n` array of the appropriate type.

The elements in `buffer` are replaced with the first penetrated surface point of the corresponding ray, and similarly the surface normal and the face attribute for `normals` and `fa`. [1]

`Dj_Plane_Section` `(const DjObject obj, const DjPlane pln, DjObject *objr)`

This function computes the object `*objr` that is the set-intersection of object `obj` with the unbounded oriented plane `pln`. The effect is the same as that of the function `Dj_Intersect`.

[1] It would be a very strange call that set all three optional arguments to zero, as the function would have no external effect.

Part III
9
External data

`Dj_Write_Object` `(const DjObject obj, FILE *txt)`

This function writes a textual representation of the (possibly null) object `obj` to the text file `*txt`. This representation may or may not be humanly readable. Data associated with user-definable attributes of object `obj` must be explicitly written by application software.

`Dj_Read_Object` `(FILE *txt, DjObject *obj)`

This function reads a textual representation of the next (possibly null) object `*obj` from the text file `*txt`. User-definable attributes are restored but the associated data is the responsibility of the application software.

Geometry import and export

```
Dj_Export_Facets  (const DjObject obj,
   const DjLabel labels[], const DjReal acc,
   DjInteger *objdim, DjInteger *ncells, DjInteger Mdims[],
   DjInteger index[], DjInteger *nvertices,
   DjInteger vertices[], DjInteger *npts,
   DjReal coords[][DjVMSize])
```

This function computes the representation of a faceting of the cells of `obj` whose labels are in `labels[]`, creating visibly exported data.

On entry, the available size `*npts` of the `coords[]` array, the available size `*nvertices` of the `vertices[]` array and the available size `*ncells`

of the `Mdims[]` and `index[]` arrays are specified. On exit the sizes actually required are returned, so that the calling program can, if necessary, call the function again with bigger arrays.

```
Dj_Export_2D_Conics (const DjObject obj,
   const DjLabel labels[], const DjReal acc,
   DjInteger *ncells, DjInteger *index[], DjInteger *nrows,
   DjInteger *conicdata[][5])
```

This function computes a visible representation of the piecewise conic arcs which approximate the edge cells of `obj` whose labels are in `labels[]` to a precision `acc`.

On entry, the available size `*nrows` of the `conicdata[][5]` array and the available size `*ncells` of the `labels[]` and `index[]` arrays are specified. On exit, the actually required sizes are returned, so that the calling program can, if necessary, call the function again with bigger arrays.

```
Dj_Import_Facets (const DjInteger objdim,
   const DjInteger ncells, const DjInteger Mdims[],
   const DjInteger index[], const DjInteger nvertices,
   const DjInteger vertices[], const DjInteger npts,
   const DjReal coords[][DjVMSize], DjObject *obj,
   DjLabel labels[])
```

This function returns an object `*obj`, in a space of dimension `objdim`, consisting of cells defined by the sequence of facetings in the input data.

The output argument `label[]`, of size `ncells`, returns the labels of the cells corresponding to the facetings in the `Mdims[]` and `index[]` arrays. Other cells are created to ensure that `*obj` is a valid Djinn object.

```
Dj_Import_2D_Conics (const DjInteger ncells,
   const DjInteger index[], const DjInteger nrows,
   const DjInteger conicdata[][5], DjObject *obj,
   DjLabel labels[])
```

This function creates an object `*obj` consisting of edge cells in two-dimensional space together with those end-points necessary to form a valid Djinn object. If the data provided represents a closed loop, no end-points will be necessary. If the data provided represents a self-intersecting curve,

additional points will be constructed at the self-intersections. The output argument `labels[]` of size `ncells` supplies the label of the piecewise conic identified at the corresponding in the `index[]` array.

Cells with tangent discontinuities can be created by importing adjacent cells and then departitioning at the point of discontinuity.

References

Bajaj 1993 C. Bajaj, "The emergence of algebraic curves and surfaces in geometric design", in *Directions in Geometric Computing*, R.R. Martin, ed. (1–29), Information Geometers, 1993.

Ball 1994 A.A. Ball, "Towards a geometric characterization of shape", in *Design and Application of Curves and Surfaces*, R.B. Fisher, ed. (91–98), Oxford University Press, 1994.

Bloomenthal 1995 J. Bloomenthal, "Bulge elimination in implicit surface blends", in *Proc. Implicit Surfaces '95* (Grenoble, France), B.L.M. Wyvill and M. Gascuel, eds. (7–20), 1995.

Blum 1967 H. Blum, "A transformation for extracting new descriptions of shape", in *Models for the Perception of Speech and Visual Form*, W. Wathen-Dunn, ed. (362–380), MIT Press, 1967.

Braid 1985 I.C. Braid, "From geometric to product modelling", in *Software for Discrete Manufacturing*, J.P. Crestin and J.F. McWaters, eds. (123–133), Elsevier, 1986.

Brown 1960 M. Brown, "A proof of the generalized Schoenflies theorem", *Bulletin of the American Mathematical Society* **66** (74–76), 1960.

Cairns 1951 S.S. Cairns, "An elementary proof of the Jordan-Schoenflies theorem", *Proceedings of the American Mathematical Society* **2** (860–867), 1951.

Catmull 1978 E. Catmull and J. Clark, "Recursively generated B-spline surfaces on arbitrary topological meshes", *Computer-Aided Design* **10**, 6 (350–355), 1978.

Cayley 1860 A. Cayley, "A new analytical representation of curves in space", *Quarterly Journal of Pure and Applied Mathematics* **3** (225–236), 1860.

Charlesworth 1995 W.W. Charlesworth and D.C. Anderson, "Applications of non-manifold topology", in *Proc. ASME Computers in Engineering*, A.A. Busnaian and R. Rangan, eds. (103–112), ASME Press, 1995.

CORBA `http://www.corba.org`

DCE `http://www.opengroup.org/dce`

DCOM `http://channels.microsoft.com/ntserver/appservice/techdetails/overview/dcomtec.asp`

Doo 1978 D. Doo and M.A. Sabin, "Behaviour of recursive division surfaces near extraordinary points", *Computer-Aided Design* **10**, 6 (356–360), 1978.

DMAC `http://www.dmac.org`

Dutta 1993 D. Dutta, R.R. Martin and M.J. Pratt, "Cyclides in surface and solid modelling", *IEEE Computer Graphics and Applications* **13**, 1 (53–59), 1993.

Foley 1990 J.D. Foley, A. van Dam, S.K. Feiner and J.F. Hughes, *Computer Graphics Principles and Practice*, Addison Wesley, 1990.

George 1998 P.-L. George and H. Borouchaki, *Delaunay Triangulation and Meshing: Application to Finite Elements*, Éditions Hermès, 1998.

Grunbaum 1976 B. Grunbaum, *Convex Polytopes*, John Wiley Interscience, 1976.

Haimes 1998 R. Haimes and G.J. Follen, "Computational analysis programming interface", in *Numerical Grid Generation in Computational Field Simulations*, M. Cross, B.K. Soni, J.F. Thompson, J. Hauser and P.R. Eiserman, eds. (663–672), International Society of Grid Generation, 1998.

Han 1996 J. Han, and A.A.G. Requicha, "Modeler-independent procedural interfaces for solid modeling", in *Proc. Computer Graphics International* (176–183), IEEE Computer Society Press, 1996.

Hoffmann 1986 C.M. Hoffmann and J. Hopcroft, "Quadratic blending surfaces", *Computer-Aided Design* **18** (301–306), 1986.

Hoffmann 1989 C.M. Hoffmann, *Geometric and Solid Modeling: An Introduction*, Morgan Kaufmann, 1989.

Hoschek 1989 J. Hoschek and D. Lasser, *Fundamentals of Computer Aided Design*, A.K. Peters, Wellesley, 1989.

Ilies 1999 H.T. Ilies and V. Shapiro, "The dual of sweep", *Computer-Aided Design* **31**, 3 (185–201), 1999.

Klok 1986 F. Klok, "Two moving co-ordinate frames for sweeping along a 3D trajectory", *Computer Aided Geometric Design* **3** (217–229), 1986.

Kripac 1995 J. Kripac, "Mechanism for persistently naming topological entities in history-based parametric solid models (topological ID system)", in *Proc. Solid Modeling '95, Third ACM Symposium on Solid Modeling and Applications*, C.M. Hoffmann and J.R. Rossignac, eds. (21–30), ACM Press, 1995.

Lavender 1992 D.A. Lavender, A. Bowyer, J. Davenport, A.F. Wallis and J.R. Woodwark, "Voronoi diagrams of set-theoretic solid models", *IEEE Computer Graphics & Applications* **12**, 5 (69–77), September 1992.

Lojasiewicz 1965 S. Lojasiewicz, *Ensembles Semi-Analytiques*, Lecture notes, IHES, Bures-sur-Yvette, 1965.

Lyusternic 1963 L.A. Lyusternic, *Convex Figures and Polyhedra*, Dover Publications, 1963.

Mäntylä 1996 M.A. Mäntylä, D.S. Nau and J.J. Shah, "Challenges in feature-based manufacturing research", *Communications of the ACM* **39**, 2 (77–85) February 1996.

Martin 1985 R.R. Martin and C. Anderson, "A proposal for an ALGOL68 binding of GKS", *Computer Graphics Forum* **4**, 1 (43–57), 1985.

Martin 1990 R.R. Martin and P.C. Stephenson, "Sweeping of three-dimensional objects," *Computer-Aided Design* **22** (223–234), 1990.

Mather 1970 J. Mather, *Notes on Topological Stability*, Harvard, 1970.

Middleditch 1974 A.E. Middleditch, *Some Notes on Solid Object Boundary Evaluation*, Internal Technical Memorandum PCL/CAE/74:3, Polytechnic of Central London, 1974.

Middleditch 1985 A.E. Middleditch and K.H. Sears, "Blend Surfaces for set-theoretic volume modelling systems", *Computer Graphics* **19**, 3 (161–170), 1985.

Middleditch 1988 A.E. Middleditch, "The representation and manipulation of convex polygons", *Theoretical Foundations of Computer Graphics and CAD* (NATO ASI Series F, Volume 40), R.A. Earnshaw, ed. (211–252) Springer-Verlag, 1988.

Middleditch 1992 A.E. Middleditch, *Cellular Models of Mixed Dimension*, Internal Technical Memorandum, BRU/CAE/92:3, Brunel University, 1992.

Middleditch 1994 A.E. Middleditch and E.E. Dimas, "Solid models with piecewise algebraic free-form faces", in *Proc. CSG 94, Set-theoretic Solid Modelling: Techniques and Applications* (133–148), Information Geometers, 1994.

Middleditch 1997 A.E. Middleditch and C. Reade, "A kernel for geometric features", in *Proc. 4th ACM Symposium on Solid Modeling and Applications*, C.M. Hoffmann and W. Bronsvoort, eds. (131–140), ACM Press, 1997.

OMG `http://www.omg.org/homepages/mfg`

O'Neill 1966 B. O'Neill, *Elementary Differential Geometry*, Academic Press, 1966.

PDES `http://pdesinc.scra.org`

Piegl 1997 L. Piegl and W. Tiller, *The NURBS Book* (2nd edn.), Springer Verlag, 1997.

Ragothama 1998 S. Ragothama and V. Shapiro, "Boundary representation deformation in parametric solid modeling", *ACM Transactions on Graphics* **17**, 24 (259–296), October 1998.

Requicha 1980 A.A.G. Requicha, "Representations for rigid solids: theory, methods and systems", *ACM Computing Surveys* **12**, 4 (437–464), December 1980.

RMI `http://www.javasoft.com/docs/books/tutorial/rmi/index.html/overview.html`

Rossignac 1990 J.R. Rossignac and and M.A. O'Connor, "SGC: A dimension-independent model for pointsets with internal structures and incomplete boundaries", in *Geometric Modeling for Product Engineering* (Proc. IFIP WG 5.2/NSF Working Conf. on Geometric Modeling, Rensselaerville, September 1988), M.J. Wozny, J.U. Turner and K. Preiss, eds. (145–180), North-Holland, 1990.

Rossignac 1997 J.R. Rossignac, "Structured topological complexes: a feature-based API for non-manifold topologies", in *Proc. 4th Symposium on Solid Modeling and Applications*, C.M. Hoffmann and W. Bronsvoort, eds. (1–9), ACM Press, 1997.

Sabin 1976 M.A. Sabin, "A method for displaying the intersection curve of two quadric surfaces," *Computer Journal* **19**, 4 (336–338), November 1976.

Sabin 1994 M.A. Sabin, "Numerical geometry of surfaces", in *Acta Numerica*, Iserles, ed. (411–466), Cambridge University Press, 1994.

Shah 1997 J.J. Shah, "Dynamic interfacing of applications to geometric modeling services via modeler neutral protocol", *Computer-Aided Design* **29**, 12 (811–824), 1997.

Sheehy 1996 D.J. Sheehy, C.G. Armstrong and D.J. Robinson, "Shape description by medial surface construction", *IEEE Transactions on Visualization and Computer Graphics* **2** (62–72), 1996.

Sherbrooke 1996 E.C. Sherbrooke, N.M. Patrikalakis and E. Brisson, "An algorithm for the medial axis transform of 3-D polyhedral solids", *IEEE Transactions on Visualization and Computer Graphics* **2** (44–61), 1996.

Silva 1981 C.E. Silva, *Alternative definitions of faces in boundary representations of solid objects*, Technical Memorandum No. 36, Production Automation Project, University of Rochester, 1981.

Spivey 1989 J.M. Spivey, *The Z Notation: A Reference Manual*, Prentice Hall, 1989.

Stacey 1986 T.W Stacey and A.E. Middleditch, "The geometry of machining for computer aided manufacture" *Robotica* **4** (83–91), 1986.

Stollnitz 1996 E.J. Stollnitz, T.D. DeRose and D.H. Salesin, *Wavelets for Computer Graphics*, Morgan Kaufmann, 1996.

Subramanyam 1995 S. Subramanyam, W. DeVries and M.J. Pratt, "Feature attributes and their role in product modeling", in *Proc. Solid Modeling '95, Third ACM Symposium on Solid Modeling and Applications*, C.M. Hoffmann and J.R. Rossignac, eds. (115–124), ACM Press, 1995.

Wall 1971 C.T.C. Wall, "Stratified sets: a survey", in *Proc. Liverpool Singularities Symposium I*, C.T.C. Wall, ed. (133–140), Lecture Notes in Mathematics Vol. 192, Springer 1971.

Weatherill 1992 N.P. Weatherill, "Delaunay triangulation in computational field dynamics", *Computers and Mathematical Applications* **24** (129–150), 1992.

Whitehead 1949 J.H.C. Whitehead, "Combinatorial homotopy I", *Bulletin of the American Mathematical Society* **54** (213–245), 1949.

Whitney 1965 H. Whitney, "Local properties of analytical varieties", in *Differential and Combinatorial Topology*, S.S. Cairns, ed. (205–244), Princeton University Press, 1965.

Woo 1997 M. Woo, J. Neider and T. Davis, *OpenGL Programming Guide: the Official Guide to Learning OpenGL* (2nd edn.), Addison Wesley Longman, 1997.

Wyvill 1986 G. Wyvill, C. McPheeters and B.L.M. Wyvill, "Data structures for soft objects", *The Visual Computer* **2** (227–234), 1986.

Yaglom 1961 I.M. Yaglom and V.G. Boltyanski, *Convex Figures*, Holt Rinehart & Winston, 1961.

Symbols and abbreviations

$\bullet$	bullet: previous $\forall$ and $\exists$ clauses apply to predicates following the $\bullet$ symbol.
$\times$	Cartesian product.
$\triangleright$	domain restriction: $f \triangleright C$ restricts the function f to a sub-domain C.
$\exists$	there exists.
$\forall$	for all.
$: \rightarrow$	from, to, e.g. $f : U \rightarrow R$ means that f is a function from the set U to the set R.
$\Rightarrow$	$A \Rightarrow B$ means that A implies B.
$\Leftrightarrow$	implies bidirectionally: $A \Leftrightarrow B$ means that A implies B and B implies A.
$\in$	in, e.g. $a \in s$ means that a is a member of set s.
$\leftrightarrow$	mapping: $A \leftrightarrow B$ is a one-to-one mapping between the elements of sets A and B.
$\nrightarrow$	partial function: $\mathit{String} \nrightarrow \mathcal{Z}$ is a partial function from string to integer.
$\mid$	such that, e.g. $a \in \mathcal{R} \mid a \leq 0$ is the set of all negative real numbers.
$\langle\,\rangle$	$\langle a, b, c \rangle$ is a vector.
$\{\,\}$	the set delimiter. $\{a, b, c\}$ is the set containing elements a, b and c. $\{0 \ldots n\}$ is the set of integers between 0 and n inclusive.
$\wedge$	logical and.
$\vee$	logical or.
$\neg$	logical negation.

∂	∂s is the boundary of point-set s.
∂^n	$\partial^0 A$, $\partial^1 A$ and $\partial^2 A$ are zero-, one- and two- dimensional point-sets in the boundary of A, separated by tangent-plane discontinuities.
$\emptyset$	the empty set.
$\supseteq$	superset: $A \supseteq B$ means that A is a proper superset of B or $A = B$.
$\subseteq$	subset: $A \subseteq B$ means that A is a proper subset of B or $A = B$.
$\subset$	$A \subset B$ means that A is a proper subset of B.
$'$	the set Q' is the complement of set Q.
$'$	Obj' is the Djinn object Obj after it has been processed by a Djinn function.
$\cap$	the intersection of two sets.
$\cap^*$	regularized set-intersection: conventionally used in solid modelling to avoid creating objects with dangling faces and edges, or internal cracks.
$\cap^\circ$	D$\ddot{\text{j}}$inn object intersection: result is always an object.
$\cap_k^\#$	set-intersection of manifolds: the subscript k selects the k-manifold in the set-intersection.
$\cap^{c_1}$	set-intersection and selection (1): that part of the set-intersection of D$\ddot{\text{j}}$inn cells that has the same dimension as the lowest dimension among the operands.
$\cap^{c_2}$	set-intersection and selection (2): that part of the set-intersection of D$\ddot{\text{j}}$inn cells that is one dimension lower than the lowest dimension among the operands.
$\bigcap_{i=1}^n$	the intersection of n sets.
$\cup$	union of two sets.
$\cup^*$	regularized set-union: conventionally used in solid modelling to avoid creating objects with dangling faces and edges, or internal cracks.
$\cup^\circ$	D$\ddot{\text{j}}$inn object union: result is always an object.
$-$	the difference of two sets.

$-^*$	regularized set-difference: conventionally used in solid modelling to avoid creating objects with dangling faces and edges, or internal cracks.
$-^\circ$	Djinn object difference: the result is always an object.
$-^c$	the difference of two Djinn cells.
$-_k^\#$	set-difference of manifolds: the subscript k selects the k-manifold in the set-intersection.
$\oplus$	the Minkowski sum of two point-sets.
$\ominus$	the complementary Minkowski sum of two point-sets.
$\#$	the Djinn partitioning operator. $A\#B$ subdivides each cell of object A, such that each subdivision is contained within a single cell of the object B.
@	the Djinn departitioning operator: $A@C$ removes the collection of cells, identified by the set of cell labels C, from the object A.
Atr( )	the notation used in the function specifications in Part II for *Attribute-map*( ). $Obj.Cell(L).Atr(S)$ is the user-defined integer associated with string S. This attribute is attached to the cell labelled L of object Obj.
Attribute-map( )	*Attribute-map*(F) is a map for object F that relates cells to their attributes. See Part I, Chapter C.
B^n	unit n-dimensional ball, centred at the origin.
C^n	continuity of order n.
Cardinality( )	the cardinality of a set is the number of elements that it contains.
Cell( )	a function from a label to the corresponding Djinn cells, which include their point-sets, orientations, parentage and attributes.
Closure( )	*Closure*(S) is the union of set S and its boundary, i.e. $Closure(S) = S \cup \partial S$.
Dim( )	the manifold dimension of a Djinn cell or the spatial dimension of a Djinn object.
Dist( )	$Dist(P, Q)$ is the Euclidean distance between points P and Q.
DjCell	the set of all Djinn cells: the Djinn cell type.

DjObject	the set of all D̈jinn objects: the D̈jinn object type.
DjPrecision	the set of all D̈jinn precisions: the D̈jinn precision type.
Domain( )	the set of independent variables of a function, e.g. *Domain*($Cell$) is the set containing the labels of all cells in the object.
$\mathcal{E}^n$	Euclidean space of n dimensions.
est	the error estimate field of a *DjPrecision*. See Part II, Chapter B.
Fin( )	finite power set: *Fin*($Cell$) is the set of all finite sets of cells.
Frontier( )	frontier of a set: *Frontier*(S) is *Closure*(S) $- S$.
Interior ( )	the interior of a set, e.g. *Interior* $(S) = S - \partial S$.
Interior$_k$( )	function that extracts a manifold: *Interior*$_k(S)$ is that part of the interior of point-set S that is a k-manifold.
Inverse	the transformation which when concatenated with an original transformation gives the identity.
Inversion	the transformation which maps each point to a new point in the same direction from the origin but at a reciprocal distance.
Label	cell label, the independent variable in the function *Cell* ( ).
Label^2	the Cartesian product of two universal label-sets: *Label*^{2} = *Label* $\times$ *Label*, and the elements of this product are all possible pairs of labels.
lb	the lower error bound field of a *DjPrecision*. See Part II, Chapter B.
MDim( )	manifold dimension: if C is an edge of a three-dimensional solid model, then *MDim*(C) = 1.
ModCnt	the modification count, which indicates how many times a D̈jinn object has been changed since the count was reset.
ModRegion	a region containing modifications to a D̈jinn object.
MvBlend ( )	concave blending function, based on the Minkowski sum, which adds blend regions to a point-set.

MxBlend( )	convex blending function, based on the Minkowski sum, which subtracts blend regions from a point-set.
NULL	the null-cell identifier.
Orient( )	cell orientation: $Obj.Cell(L).Orient(P)$ is the orientation vector of the cell labelled L of object Obj at point P.
OUT	the entire region of space outside an object.
$\mathcal{P}^n$	projective space of n dimensions.
$\mathcal{P}(\,)$	power set: for example, if $\mathcal{R}$ is the set of all real numbers, then $\mathcal{P}(\mathcal{R})$ is the set of all sets of real numbers.
Parentage-map( )	*Parentage-map*(F) is a map for object F from each cell to the cells from which it was derived in the latest geometric modification. See Part I, Chapter C.
Parents	the notation used in the function specification of Part II for *Parentage-map*. $Obj.Cell(L).Parents$ is a set of cell pairs from which the cell labelled L of object Obj was derived.
Point-set( )	*Point-set*(C) is the point-set of cell C.
$\mathcal{R}$	the set of all real numbers.
$\mathcal{R}^n$	real space of n dimensions.
Range( )	the set of possible results of a function, e.g. *Range*(*Cell*) is all the cells of an object.
SDim( )	spatial dimension.
Seq( )	sequence.
STweak( )	a small tweak: a function which maps a set of cells in an original object under a given transformation and distance to produce a new object
ub	the upper error bound field of a *DjPrecision*. See Part II, Chapter B.
VBlend( )	concave blending function, based on *MvBlend*( ), which adds blend cells to a Djinn object.
Vector( )	converts a *DjPoint* to a *DjVector*.
VRound( )	convex rounding function.

XBlend( )	convex blending function, based on *MxBlend*( ), which subtracts blend cells from a Djinn object.
XRound( )	concave rounding function.
$\mathcal{Z}$	the set of all the integers.
$\mathcal{Z}^+$	the set of all positive integers.

Index