Copy No. _____

Defence Research and Development Canada
Recherche et développement pour la défense Canada

DEFENCE R&D DÉFENSE

C++ classes for representing curves and surfaces

Part III: Reading and writing in IGES format

David Hally

Defence R&D Canada – Atlantic

Technical Memorandum
DRDC Atlantic TM 2006-256
January 2007

Canada

This page intentionally left blank.

C++ classes for representing curves and surfaces

Part III: Reading and writing in IGES format

David Hally

Defence R&D Canada – Atlantic

Technical Memorandum

DRDC Atlantic TM-2006-256

January 2007

Principal Author

Original signed by David Hally

David Hally

Approved by

Original signed by R. Kuwahara

R. Kuwahara
Head/Signatures

Approved for release by

Original signed by K. Foster

K. Foster
Chair/Document Review Panel

Abstract

The Initial Graphics Exchange Specification (IGES) is in widespread use as a means of exchanging graphical information between computer programs. It can also be used to exchange the definitions of geometrical objects, regardless of how they are displayed. C++ classes in the CurveLib library allow geometry to be represented as differentiable functions within computer programs written in C++. This document describes a library of C++ classes which allow the geometry described by an IGES file to be represented by the CurveLib classes. Conversely, geometry represented by the CurveLib classes can be stored in an IGES file so that it can be used by other applications.

Résumé

Le format Initial Graphics Exchange Specification (IGES) est largement utilisé pour transmettre des informations graphiques d'un logiciel à un autre. On peut également l'utiliser pour transmettre des définitions d'objets géométriques, quelle que soit la façon dont ils sont affichés. Dans les logiciels écrits en C++, les classes de la bibliothèque CurveLib permettent de représenter la description géométrique sous la forme de fonctions différentiables. Le présent document décrit une bibliothèque de classes C++ qui permet aux informations géométriques contenues dans un fichier IGES d'être représentées par des classes CurveLib. Inversement, les informations géométriques représentées par les classes CurveLib peuvent être sauvegardées dans un fichier IGES et utilisées par d'autres logiciels.

This page intentionally left blank.

Executive summary

C++ classes for representing curves and surfaces: Part III: Reading and writing in IGES format

David Hally; DRDC Atlantic TM-2006-256; Defence R&D Canada – Atlantic; January 2007.

Background: The flow around ships and propellers affects their performance in many ways. Defence R&D Canada – Atlantic uses Computational Fluid Dynamics (CFD) to calculate these flows so that the performance of the hull and propellers can be evaluated and improved. Before the flow can be calculated, the geometry of the ship or propeller must be represented in a fashion that can be used by the CFD applications. The CurveLib library of C++ classes has been developed as a means of representing such geometry. The current document describes C++ classes which can be used to store the geometry in files conforming to the Initial Graphics Exchange Specification (IGES). The geometry can then be used by a wide variety of commercial applications. Conversely, geometry which has been defined in another application and stored in the IGES format can be represented using the CurveLib classes.

Results: A library of C++ classes has been developed which use files in the IGES format to provide an interface between CurveLib geometry and commercial applications. The overlap between the CurveLib library and the IGES specification is not perfect: there are many IGES entities that cannot be represented by a CurveLib class and there are many curves representable by CurveLib classes which have no representation by IGES entities. However, the ability to approximate geometric objects with splines means that at least a close approximation of most geometric objects can be represented both in an IGES file and as a CurveLib class.

Significance: The C++ classes are a useful tool which allow the geometry of ship hulls and propellers, generated using DRDC applications, to be used by commercial applications. Conversely, they allow geometry generated by commercial applications to be represented as CurveLib classes and used within DRDC applications.

Sommaire

C++ classes for representing curves and surfaces: Part III: Reading and writing in IGES format

David Hally ; DRDC Atlantic TM-2006-256 ; R & D pour la défense Canada – Atlantique ; janvier 2007.

Contexte : L'écoulement de l'eau autour des navires et de leurs hélices influence leur comportement de différentes manières. R & D pour la défense Canada – Atlantique utilise la dynamique numérique des fluides pour calculer ces écoulements et ainsi évaluer et améliorer le comportement des carènes et des hélices. Pour calculer l'écoulement, on doit toutefois pouvoir représenter la géométrie du navire ou de l'hélice d'une manière compatible avec les logiciels de dynamique numérique des fluides. La bibliothèque de classes C++ CurveLib a été créée afin de pouvoir représenter cette géométrie. Dans le présent document, nous décrivons les classes C++ que l'on peut utiliser pour conserver les informations géométriques dans des fichiers conformes au format Initial Graphics Exchange Specification (IGES). Ces informations géométriques peuvent ensuite être utilisées par une vaste gamme de logiciels commerciaux. En contrepartie, des informations géométriques définies par un autre logiciel conservées dans le format IGES peuvent être représentées avec les classes CurveLib.

Résultats : Nous avons élaboré une bibliothèque de classes C++ qui utilisent des fichiers en format IGES et assurent donc l'interface entre les informations géométriques de CurveLib et des logiciels commerciaux. Or, la bibliothèque CurveLib et les spécifications IGS ne se recoupent pas parfaitement : de nombreuses entités IGES ne peuvent pas être représentées par une classe CurveLib et plusieurs courbes représentables par une Classe CurveLib ne peuvent être représentées par une entité IGES. Grâce à la capacité d'approximer les objets géométriques avec des splines, on pourra toutefois utiliser une approximation proche de la plupart des objets géométriques et, ce, à la fois dans un fichier IGES et par une classe CurveLib.

Importance : Les classes C++ sont un outil précieux permettant à des logiciels commerciaux d'utiliser les informations géométriques sur les carènes et les hélices produites avec des logiciels de RDDC. Inversement, les informations géométriques produites avec les logiciels commerciaux pourraient être représentées comme des classes CurveLib et utilisées par les logiciels de RDDC.

Table of contents

List of figures

1 Introduction

The Initial Graphics Exchange Specification (IGES)[1,2] is in widespread use as a means of exchanging graphical information between computer programs. It can also be used to exchange the definitions of geometrical objects, regardless of how they are displayed. C++ classes in the CurveLib library[3] allow geometry to be represented as differentiable functions within computer programs written in C++. This document describes a library of C++ classes which allow the geometry described by an IGES file to be represented by the CurveLib classes. Conversely, geometry represented by the CurveLib classes can be stored in an IGES file so that it can be used by other applications.

The overlap between the CurveLib library and the IGES specification is far from perfect: there are many IGES entities that cannot be represented by a CurveLib class and there are many curves representable by CurveLib classes which have no representation by IGES entities. However, the ability to approximate geometric objects with splines means that at least a close approximation of most geometric objects can be represented both in an IGES file and as a CurveLib class.

The CurveLib classes were originally designed for representing the geometry of complex shapes (e.g. ship hulls and propellers) for use in Computation Fluid Dynamics programs. The ability to represent this geometry in an IGES format means that it can be used in a wide variety of commercial software. For example, a representation of a propeller using the CurveLib classes can be stored as an IGES file, loaded into a grid generation program such as GridGen^{TM} to generate a computational grid suitable for use in a flow calculation program such as TRANSOM[4] or CFX^{TM}.

2 Overview

The basis of the interface between an IGES file and representation of geometry by the CurveLib classes is the IGES model. It is a collection of geometrical entities each of which can be represented both as an entity in an IGES file and as a CurveLib curve. To store CurveLib curves in an IGES file an IGES model must first be created, the appropriate CurveLib curves are converted to IGES entities and added to it, then the model is written to the file. To read an IGES file an IGES model is created, the file is read to create the geometry within the model, then the model can be queried to extract the geometry for whatever purpose it is required.

Not all IGES entities are supported. The following is the list of IGES entities which can be represented using the CurveLib classes.

- 100: Circular Arc Entity
- 102: Composite Curve Entity
- 104: Conic Arc Entity
- 108: Plane Entity
- 110: Line Entity
- 112: Parametric Spline Curve Entity
- 114: Parametric Spline Surface Entity
- 116: Point Entity
- 118: Ruled Surface Entity
- 120: Surface of Revolution Entity
- 122: Tabulated Cylinder Entity
- 124: Transformation Matrix Entity
- 126: Rational B-spline Curve Entity
- 128: Rational B-spline Surface Entity
- 130: Offset Curve Entity
- 140: Offset Surface Entity
- 141: Boundary Entity
- 142: Curve on a Parametric Surface Entity
- 143: Bounded Surface Entity
- 144: Trimmed (Parametric) Surface Entity
- 308: Subfigure Definition Entity
- 408: Singular Subfigure Instance Entity

Conversely, not all CurveLib curves and surfaces will have a direct representation by IGES entities. When this is the case, the curve or surface will usually have to be approximated by a spline first.

3 Basic classes and types

All the classes described in this report belong to the namespace `IGES`.

Classes used by the IGES classes but described elsewhere include the following:

`template<class F>`
`Angle<F>`
: Represents an angle: described in Reference 3, Annex E. `F` is the type of the floating point number used to store the value of the angle.

`template<unsigned N, class F>`
`VecMtx::VecN<N,F>`
: A vector of length `N` each of whose elements is of type `F`: described in Reference 3, Annex B.

`template<unsigned N, class F>`
`VecMtx::MtxN<N,F>`
: An `N`×`N` matrix each of whose elements is of type `F`: described in Reference 3, Annex C.

`template<unsigned N, class V, class F>`
`CurveLib::Curve<N,V,F>`
: An `N` parameter differentiable function which returns a value of type `V`; each function parameter is of type `F`: described in Reference 3, Section 2.

`template<unsigned N, class V, class F>`
`CurveLib::RangeCurve<N,V,F>`
: An `N` parameter differentiable function which returns a value of type `V`; each function parameter is of type `F` and has a specified range of values: described in Reference 3, Section 10.

The IGES namespace also defines the following generic types.

`typedef double Float`
: A floating point number. Though the CurveLib classes allow the representation of floating point numbers in an arbitrary way via template arguments, the classes in the namespace `IGES` all use `Float`. `Float` is defined in the header file `IGESFloat.h`.

`typedef VecMtx::VecN<3U,Float> Point`
: A point in space.

`typedef VecMtx::VecN<3U,Float> Normal`
: A normal to a surface.

`typedef CurveLib::RangeCurve<1U,Point,Float> CurveType`
: The type of a curve in three-dimensional space which has a fixed parameter range.

`typedef CurveLib::RangeCurve<2U,Point,Float> SurfaceType`
: The type of a parametric surface in three-dimensional space which has a fixed ranges for each of its two parameters.

`typedef CurveLib::LimitedSurface<Point,Float> LimitedSurfaceType`
: The type of limited surfaces: see Section 5.

`typedef VecMtx::VecN<3U,Float> TransVec`
: The type of a translation vector for a point in space.

`typedef VecMtx::MtxN<3U,Float> RotMtx`
: The type of a rotation matrix for a point in space.

`typedef std::string Str`
: The type of a character string.

3.1 Exceptions and warnings

All exceptions thrown by the classes in the namespace `IGES` are derived from the base class `Error`. It contains an error message and has member functions for appending or prepending to the message should it be necessary to rethrow the exception. It is wise to enclose code using the IGES classes with a try block which catches any `Error`:

```
try{
  // Code using classes from namespace IGES
}
catch(Error &e) {
  // Handle the exception e.
}
```

Detailed prototypes of the `Error` member functions are given in Reference 3, Annex F.

One important exception that is derived from `Error` is `ProgError`. It is thrown when an error is encountered that can clearly be identified as a programming error rather than an error by the user of the application.

In addition to exceptions, the IGES classes will sometimes issue warnings to indicate that something was not quite right but that some remedial action has been taken. By default, warnings are simply written to `std::cerr`. However, it is possible to intercept warnings in order to display them in other ways. To intercept warning messages you must define a warning handler. The procedure for doing this is described in Annex A.

The warning messages issued by the IGES classes are often long and no attempt has been made to break them into lines (because it is impossible to tell what a suitable line length is). Therefore it is usually best to substitute the default warning handler with one that will break the message into lines. The following code substitutes the default warning handler with one that sends the warning to `std::cerr`, but breaks it into lines of fewer than 80 characters. It is appropriate for output to most terminals.

```
#include "FormattedWarning.h"

int main()
{
  FormattedOStream out(std::cerr);
  FormattedWarningHandler whandler(out);
  set_warning_handler(whandler);

  // Remaining program code
}
```

The class `FormattedOStream`, defined in the header file `FormattedWarning.h`, is an output stream that will format the output stream into lines. The maximum length of the lines defaults to 80 but can be set using the member function `num_columns(int)`. For example, if only 60 columns were available on the screen for the warning message, then one could add

```
out.num_columns(60);
```

to the code above.

4 Coordinate transformations

IGES entities are often defined in one coordinate system, the definition space, and displayed in another, the model space; a coordinate transformation is used to transform the entity from its definition space to model space. Each coordinate transformation consists of a scale represented by a `Float`, a rotation represented by a `RotMtx`, and a translation represented by a `TransVec`. The translation is applied last. Thus, if $\mathbf{x}$ is a point in definition space, then the corresponding point in model space is

$$\mathbf{x}' = \mathbf{T} + s\mathbf{R}\mathbf{x} \tag{1}$$

where s is the scaling factor, $\mathbf{R}$ is the rotation matrix, and $\mathbf{T}$ is the translation vector.

Coordinate transformations are represented by the class `Transformation` which has the following member functions:

`Transformation()`
: Default constructor. The transformation is the identity: i.e. a scaling factor of 1, an identity rotation matrix and a zero translation matrix.

`Transformation(const RotMtx &m, const TransVec &v, Float s = 1.0)`
: Makes a transformation with rotation `m`, translation `v` and scale `s`.

`void set_scale(Float s)`
: Sets the scaling factor to `s`.

`Float get_scale() const`
: Returns the scaling factor.

`void set_rotation(const RotMtx &m)`
: Sets the rotation matrix to `m`.

`RotMtx get_rotation() const`
: Returns the rotation matrix.

`void set_translation(const TransVec &v)`
: Sets the translation vector to `v`.

`TransVec get_translation() const`
: Returns the translation vector.

`Transformation operator*(const Transformation &trans) const`
: Combines the transformation with `trans`.

`template<class Type> Type apply(const Type &obj) const`
: Applies the transformation to the geometric object `obj`. The arithmetic operations `RotMtx*Type`, `Float*Type` and `TransVec+Type` must all be defined and return a `Type`.

  Most commonly `Type` will be `LimitedSurfaceType`, `SurfaceType`, `CurveType` or `Point`.

5 Limited surfaces

A trimmed surface is a surface whose domain is limited by a set of boundary curves. Only the portion of the surface lying inside or outside each boundary curve is accepted as part of the trimmed surface. IGES provides three entities for representing trimmed surfaces: Trimmed (Parametric) Surface Entity, Bounded Surface Entity and a bounded Plane Entity. To avoid confusion in nomenclature, we will call any type of trimmed surface a *limited surface* and will reserve the term trimmed surface for limited surfaces represented by a Trimmed (Parametric) Surface Entity.

None of the CurveLib classes described in Reference 3 is capable of describing a limited surface since `RangeCurve<2U,Point,Float>` is the only class that limits the domain of a surface and it simply limits the domain to a rectangle. Therefore, to be able to represent the limited surfaces defined in an IGES file, CurveLib must be must extended. Two additional classes are defined in the header file `LimitedSurface.h`: `LimitedSurface<V,F>` and `SurfaceCurve<V,F>`. Since neither of these classes has any direct dependence on IGES, they are put in the namespace `CurveLib`. The template argument `V` is the type of the returned value of the surface curve and the template argument `F` is the type of each surface parameter. When used with the IGES classes `V` and `F` will be `Point` and `Float` respectively.

The class `SurfaceCurve<V,F>` represents a curve embedded in a surface. It is derived from `RangeCurve<1U,V,F>` in namespace `CurveLib` (see Reference 3, Section 10.2). A `SurfaceCurve<V,F>` stores two different representations of the curve:

1. A direct representation which returns a value of type `V`. This representation will some times be called the space representation. In IGES it is also called the model space representation.

2. A composed representation which defines the curve as $\mathbf{S}\big(\mathbf{P}(u)\big)$ where $\mathbf{S}(x, y)$ is the surface in which the curve is embedded and $\mathbf{P}(u)$ is a curve in the parameter space of $\mathbf{S}$: i.e. for each u, $\mathbf{P}(u)$ returns a parametric coordinate (x, y).

`SurfaceCurve<V,F>` has member functions which dictate which of these representations is to be used. Neither of these representations is mandatory.

The use of two different representations has been largely driven by the requirements of IGES. If $\mathbf{S}$ and $\mathbf{P}$ have representations as CurveLib curves, then their composition does too. Therefore there is no real need for the direct representation of the curve in CurveLib. However, it is not true that if $\mathbf{S}$ and $\mathbf{P}$ have representations as IGES entities, that their composition will also have a representation as an IGES entity. At best one can only define an approximation to the composed curve.

However, even within the confines of CurveLib, the direct representation can have advantages. For simple surfaces it is often possible to define a direct representation that will evaluate considerably more efficiently than the composed representation.

The principal use for a surface curve is to define a boundary for trimming the surface. In this application the curve often has discontinuities in derivative (imagine a square hole being trimmed from a flat plate; the curve describing the square has discontinuities in derivative at each of its corners). For each of the two representations of the curve, `SurfaceCurve<V,F>` maintains a list of parameters at which the curve may have discontinuous derivatives.

In fact, the parameter curve may not only have discontinuous derivatives but may also be completely discontinuous at some points. In general, due to round-off errors and the finite tolerance of the IGES model (see Section 8), it is usually impossible to tell with certainty whether a curve is really discontinuous rather than simply inaccurate when evaluated. Therefore `SurfaceCurve<V,F>` has a member function, `get_parametric_mismatch()` that returns the largest mismatch in values of the parameter curve at each point of discontinuity: i.e. if x_i are the points of discontinuity of the parameter curve $\mathbf{p}(x)$, then

$$\epsilon \equiv \max_i \left| \lim_{x \to x_i +} \mathbf{p}(x) - \lim_{x \to x_i -} \mathbf{p}(x) \right| \tag{2}$$

is returned. If the value of ϵ is larger than the tolerance of the model, then the parametric curve can be considered to be discontinuous.

`SurfaceCurve<V,F>` has the following public members.

`typedef V ValueType`
: The type of the value of the curve.

`typedef RangeCurve<2U,V,F> SurfaceType`
: The type of the surface in which the curve is embedded.

`typedef RangeCurve<1U,V,F> SpaceCurveType`
: The type of the direct representation of the curve.

`typedef typename SpaceCurveType::ParamType ParamType`
: The type of the curve parameter list.

`typedef typename SurfaceType::ParamType SurfParamType`
: The type of the parameter list for the surface.

`typedef RangeCurve<1U,SurfParamType,F> ParamCurveType`
: The type of the curve $\mathbf{P}(u)$ in the parameter space of the surface.

`typedef Spline::KnotSeq<F> DiscontinuityList`
: The type of a list of parameters at which the curve might have discontinuous derivatives. The parameters in the list must be strictly increasing. The class `KnotSeq<F>` is described in Reference 5, Section 3.

`SurfaceCurve()`
: Default constructor. The curve remains undefined.

`SurfaceCurve(SurfaceType s, SpaceCurveType c)`
: Makes a curve embedded in surface `s` whose direct representation is `c`. The composed representation of the curve will be undefined.

`SurfaceCurve(SurfaceType s, ParamCurveType p)`
: Makes a curve embedded in surface `srf` whose curve in parameter space is `p`. The direct representation of the curve will be identical to the composed curve.

`SurfaceCurve(SurfaceType s, SpaceCurveType c, ParamCurveType p)`
: Makes a curve embedded in surface `s` whose curve in space is `c` and whose curve in parameter space is `p`.

`bool has_composed_representation() const`
: Returns true if the curve has a composed representation.

`bool has_space_representation() const`
: Returns true if the curve has a direct representation.

`bool is_composed() const`
: True if the composed representation is being used.

`void use_composed_curve()`
: Causes the composed representation of the curve to be used. Ignored if no composed representation is defined.

`void use_space_curve()`
: Causes the space representation of the curve to be used. Ignored if no space representation is defined.

`SurfaceType get_surface() const`
: Returns the surface in which the curve is embedded.

`void set_surface(const SurfaceType &s)`
: Sets the surface in which the curve is embedded to `s`.

`SpaceCurveType get_space_curve() const`
: Returns the model space representation of the curve.

`void set_space_curve(const SpaceCurveType &sc)`
: Sets the model space representation of the curve to `sc`. The model curve discontinuities remain unchanged.

`ParamCurveType get_parametric_curve() const`
: Returns the curve in the parameter space of the surface.

`void set_parametric_curve(const ParamCurveType &pc) const`
: Sets the curve in the parameter space of the surface to `pc`. The parametric curve discontinuities remain unchanged.

`const DiscontinuityList& get_discontinuities() const`
: Returns the list of parameters at which the curve may have discontinuous derivatives. It will contain at least the parameters for the end points, so the discontinuities can also be used to get a range for the curve.

`void set_space_curve_discontinuities(const DiscontinuityList &d)`
: Sets the list of parameters at which the direct representation curve may have discontinuous derivatives.

`void set_parametric_curve_discontinuities(const DiscontinuityList &d)`
: Sets the list of parameters at which the parameter curve may have discontinuous derivatives.

`Float get_parametric_mismatch(bool end_points) const`
: Returns the value of ϵ defined above. If `end_points` is true, the mismatch between the end-points of the parametric curve is included.

`void reverse()`
: Reverses the direction of the curve but retains the same parameter range: i.e. as the parameter of the the curve is increased from its lowest to highest allowed value, the curve is traced from its end-point to its start point. The list of possible discontinuities in the curve is also reversed.

  The principal use of a `SurfaceCurve<V,F>` is as a boundary of a limited surface represented as a `LimitedSurface<V,F>` (see below). The latter requires that the boundary curve is oriented counter-clockwise (relative to the surface normal) around the portion of the surface to be retained after trimming. When applied to a curve used as a `LimitedSurface<V,F>` boundary, this function changes the portion of the surface retained from the interior to the exterior or vice versa.

The class `LimitedSurface<V,F>` reresents a limited surface. It stores a CurveLib representation of the surface whose domain is to be limited as well as a list of the boundary curves which limit the domain of the surface. Each boundary curve is represented as a `SurfaceCurve<V,F>`.

The surface whose domain is to be limited, $\mathbf{S}$, must have a well-defined normal at each point in its interior. If its parameters are (u, v), the normal is defined by

$$\mathbf{n} = \frac{\partial \mathbf{S}}{\partial u} \times \frac{\partial \mathbf{S}}{\partial v}; \qquad \hat{\mathbf{n}} = \frac{\mathbf{n}}{|\mathbf{n}|} \tag{3}$$

Each boundary curve must be continuous and closed: its start point and end point must match. The portion of the surface retained lies on the left of the boundary curve: i.e. the portion in the direction of the cross product of the surface normal with the tangent vector to the boundary curve.

There is no requirement that the parameter curves for a boundary be either continuous or closed in the surface parameter space. For example, consider the cylindrical surface defined by

$$\mathbf{S}(\theta, z) = \hat{x}\cos\theta + \hat{y}\sin\theta + z\hat{z}; \qquad \theta \in [0, \pi]; z \in [-1, 1] \tag{4}$$

We could define a boundary where the cylinder intersects the xy plane using the single parameter curve

$$\mathbf{C}(u) = (2\pi u, 0); \quad u \in [0, 1] \tag{5}$$

This curve is not closed in parameter space since $\mathbf{C}(0) \neq \mathbf{C}(1)$. However, it still generates a valid boundary curve as the space curve $\mathbf{S}\big(\mathbf{C}(u)\big)$, which is the unit circle in the xy plane centred at the origin, *is* closed.

`LimitedSurface<V,F>` has no base class and has the following public members:

`typedef V ValueType`
: The value of the surface and the boundary curves.

`typedef SurfaceCurve<V,F> SurfaceCurveType`
: The type of each of the boundary curves.

`typedef typename SurfaceCurveType::SurfaceType SurfaceType`
: The type of the surface to be trimmed.

`typedef typename SurfaceCurveType::ParamCurveType ParamCurveType`
: The type of the representation of a boundary curve in the parameter space of the surface.

`typedef std::list<SurfaceCurveType> BoundaryList`
: The type of a list of boundary curves.

`LimitedSurface()`
: Default constructor. The surface and the boundary curves are undefined.

`LimitedSurface(SurfaceType s)`
: Makes a limited surface which limits the domain of `s`.

`SurfaceType get_surface() const`
: Returns the surface whose domain is to be limited.

`void set_surface(SurfaceType s)`
: Sets the surface whose domain is to be limited to `s`.

`void clear()`
: Removes all the boundary curves.

`void add_boundary(SurfaceCurveType c)`
: Adds `c` to the list of boundaries. If no surface has been defined, it will be set to the surface in which `c` is embedded. If a surface has been defined, it must be the same as the surface in which `c` is embedded.

`void add_boundary(SurfaceCurveType c, bool interior)`
: Adds c to the list of boundaries. If ınterior is true, the orientation of the curve will be adjusted, if necessary, so that the interior of the curve is retained; otherwise the orientation will be adjusted, if necessary, so that the exterior of the curve is retained.

  If no surface has been defined, it will be set to the surface in which `c` is embedded. If a surface has been defined, it must be the same as the surface in which `c` is embedded.

  This function requires that `c` has a parametric representation.

`const BoundaryList& get_boundaries() const`
: Returns the list of boundary curves.

`bool is_parametric() const`
: Returns true if every boundary curve has a parametric representation.

`bool is_defined() const`
: Returns true if the surface whose domain is to be limited has been defined.

Arithmetic operators have also been defined for `LimitedSurface<V,F>` to make it easy to transform a limited surface between coordinate systems. The following operators each return a transformed `LimitedSurface<V,F>` provided that `s` is a `LimitedSurface<V,F>`, `t` is of type `V`, and `a` is of type `V1` such that `V1*V` and `V*V1` are defined and return a `V`:

`-s` returns `s` scaled by -1;

`a*s` returns `s` multiplied by `a`;

`s*a` returns `s` multiplied by `a`;

`s+t` returns `s` translated by `t`;

`t+s` returns `s` translated by `t`;

`s-t` returns `s` translated by `-t`;

`t-s` returns `s` scaled by -1 then translated by `t`.

Note that these operators are sufficient to allow a `LimitedSurface<Point,Float>` to be used as the argument to the `Transformation` member function `apply()`: see Section 4.

6 Entities

The abstract base class `Entity` is used to represent an arbitrary IGES entity. It provides the connection between an entity in an IGES file and the representation of that entity using CurveLib classes. Figure 1 is an inheritance diagram for `Entity` and all the classes derived from it.

Every IGES entity has the following attributes:

- An Entity Type Number which specifies the type of entity.
- A level indicating where the entities lies in the hierarchy of the IGES model.
- An optional Transformation Matrix Entity (see Section 6.1) which transforms the entity from its definition space to its model space.
- A flag which indicates whether the entity is visible.
- A flag which indicates whether the entity is physically dependent on another entity. Physically dependent means that another entity (the parent) refers to the entity; the child cannot exist unless the parent exists.
- A flag which indicates whether the entity is logically dependent on another entity. Logically dependent means that the entity can exist alone but belongs to some sort of logical grouping.
- A Hierarchy flag which indicates whether entities subordinate to the current entity inherit its visibility. If the visibility is inherited, the visibility attributes of the subordinate entities (see below) will be ignored.
- An Entity Use Flag which indicates the intended use of the entity.
- A Form Number: an integer which has different meanings depending on the type of the entity.
- An optional label of at most eight characters.
- An optional subscript number associated with the label and used to indicate that the entity is part of an aggregate of entities.

Other entity attributes defined within IGES (e.g. line font pattern or colour number) are ignored.

When any entity is first created it will normally be visible, have visibility inherited by its subordinate entities, be physically and logically independent, and have no Transformation Matrix entity, label or subscript number.

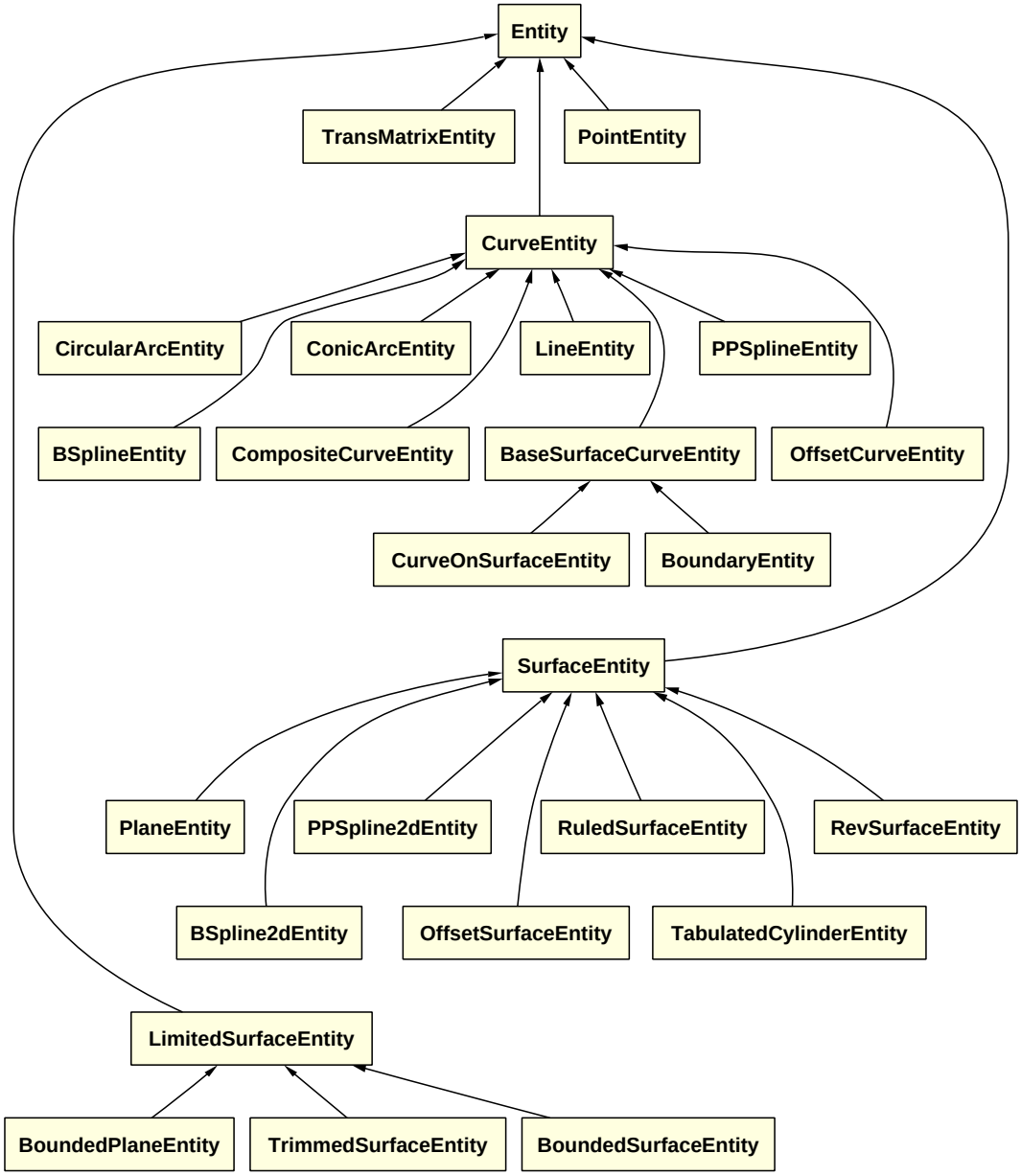

Figure 1: *The inheritance relations between the entity classes. The arrows point from a derived class to its base class.*

Each entity also maintains a list of pointers to each of its instantiated subordinate entities. An example of a subordinate entity is a Circular Arc Entity which is a component of a Composite Curve Entity (see Sections 6.3.2 and 6.3.6). The subordinate entities of the subordinate entities are not included in the list. For example, if the Composite Curve Entity is included in a Singular Subfigure Instance Entity (see Section 6.9), then the list of subordinate entities for the Singular Subfigure Instance Entity includes the Composite Curve Entity but not the Circular Arc Entity. Neither is the Transformation Matrix Entity, if there is one, included in the list. Note, too, that the list of subordinate entities of a Subfigure Definition Entity (see Section 6.8) is empty; it does *not* include the entities used to define the subfigure because they are not instantiated.

An `Entity` is a component of a `Model`, the class that represents the full geometric model described by an IGES file (see Section 8). A `Model` takes over responsibility for deleting all entities that it contains. This has the following consequences:

- An `Entity` should always be allocated from the heap.
- An `Entity` should never be deleted explicitly unless it is never added to a `Model`.
- An `Entity` should be added to at most one `Model`.

When an `Entity` is added to a `Model`, its transformation matrix and all its subordinate entities are also added to the `Model` provided that they do not already belong to that `Model`. If the transformation matrix or one of the subordinate entities already belongs to a different `Model`, a `ProgError` exception will be thrown.

`Entity` has the following public member functions:

`int get_entity_number() const`
: Returns entity number.

`int get_form_number() const`
: Returns the Form Number for the entity. A value of -1 indicates that no form number has been specified.

`bool is_visible() const`
: Returns true if the entity is meant to be visible.

`bool is_dependent() const`
: Returns true if the entity is physically or logically dependent on another.

`bool is_physically_dependent() const`
: Returns true if this entity is physically dependent on another.

`bool is_logically_dependent() const`
Returns true if this entity is logically dependent on another.

`bool visibility_inherited() const`
Returns true if the visibility of the entity is inherited by subordinate entities.

`UseFlag intended_use() const`
Returns the Entity Use Flag specifying the intended use of the entity. The type `UseFlag` is an `enum` having the following possible values:

`geometry`
The entity is used to define geometry.

`annotation`
The entity is used to add annotation or description to the file.

`definition`
The entity is used in definition of structures in the file.

`other`
The entity is being used for other purposes.

`logical_positional`
The entity is used as a logical and/or positional reference by other entities.

`parametric_2D`
The entity has values in two-dimensional parametric space considered as a subset of three-dimensional (x, y, z) space by ignoring the z coordinate. Intended for use in defining curves on surfaces.

`construction_geometry`
The entity is used only for convenience in preparing the model or drawing, *not* for defining the geometry of the structure of the product. An example is the two lines intersected to find the center of a rectangle.

The entities implemented by the classes described here will have an intended use flag of `definition` (the Subfigure Definition Entity; class `SubFigDefEntity`), `parametric_2D` (any curve entity used as the parametric curve in a `BaseSurfaceCurveEntity`) or `geometry` (all others). Entities read from an input file having any of the other entity use flags will normally be ignored.

`void make_visible()`
Makes the entity visible. If the subordinate entities inherit visibility, then they are made visible too.

`void make_invisible()`
Makes the entity invisible. If the subordinate entities inherit visibility, then they are made invisible too.

`void make_physically_dependent()`
Makes the entity physically dependent on another (unspecified) entity.

`void make_logically_dependent()`
Makes the entity logically dependent on another (unspecified) entity.

`bool make_visibility_inherited()`
Makes the visibility of subordinate entities inherited from the visibility of this entity.

`bool make_visibility_independent()`
Makes the visibility of subordinate entities independent of the visibility of this entity.

`void make_intended_use(UseFlag u)`
Sets the intended use of the entity to `u`. See the description of `intended_use()` above.

`const TransMatrixEntity* get_transformation_entity() const`
Returns the Transformation Matrix Entity for the entity: see Section 6.1.

`void set_transformation_entity(TransMatrixEntity *m)`
Sets a Transformation Matrix Entity for the entity.

`virtual Transformation get_transformation() const`
Returns the transformation from the entity definition space to the entity model space. For most entities this is the same as the combined transformation of the Transformation Matrix Entity (see Section 6.1); however, for some entities, Singular Subfigure Instances in particular, there may be additional transformations that are included in the returned transformation.

`void set_label(const Str &s)`
Sets the label for the entity. If `s` has more than eight characters it will be truncated.

`Str get_label() const`
Returns the label for this entity.

`int get_subscript_number() const`
Returns the subscript number for this entity.

`void set_subscript_number(int n)`
Sets the subscript number for this entity.

`const EntityCollection& get_children() const`
Returns the subordinate entities (children). The class `EntityCollection` is described in Section 6.7.

6.1 Transformation Matrix Entity

Every IGES entity is defined in its own definition space. In order for the entity to be correctly placed in model space it may have to be rotated and/or translated. Therefore each entity has an optional pointer to a transformation, represented by a Transformation Matrix Entity which has Entity Type Number 124. When this pointer is present, the entity is subjected to the transformation before being placed in model space. The transformation represented by the Transformation Matrix Entity is called the defining transformation for the entity.

The defining transformation consists of a rotation, implemented using a 3×3 matrix $\mathbf{R}$, and a translation, implemented using a 3-vector $\mathbf{T}$. A point in the entity definition space, $\mathbf{x}_d$ is transformed to a point in model space $\mathbf{x}_m$ by

$$\mathbf{x}_m = \mathbf{R}\mathbf{x}_d + \mathbf{T} \tag{6}$$

The rotation matrix $\mathbf{R}$ must be orthogonal: its transpose must be the same as its inverse.

Since a Transformation Matrix Entity is itself an Entity, it may also have a defining transformation. When a Transformation Matrix Entity has a defining transformation, the two transformations are combined, with the Transformation Matrix Entity being applied first followed by its defining transformation. In fact, the defining transformation may itself have a defining transformation, so we define the combined transformation to be the Transformation Matrix entity transformation applied first followed by the combined transformation of the defining transformation. This recursive definition accounts for the complete hierarchy of transformations.

The Form Number for a Transformation Matrix Entity must be either 0 or 1 (IGES also defines the values 10, 11 and 12 for use in finite element geometry but they are not used here). The value 0 indicates that the determinant of the rotation matrix is 1, so that the handedness of the coordinate system is not changed by the transformation. The value 1 indicates that the determinant of the rotation matrix is -1, so that the handedness of the coordinate system is reversed by the transformation.

In the IGES classes a Transformation Matrix Entity is represented by the class `TransMatrixEntity`; the transformation itself is represented by a `Transformation`: see Section 4. `TransMatrixEntity` has the following public members.

`TransMatrixEntity()`
: Makes a transformation matrix entity. The transformation is the identity.

`TransMatrixEntity(const RotMtx &m, const TransVec &v)`
: Makes a transformation matrix entity. The rotation matrix and translation vector are set to `m` and `t`.

`TransMatrixEntity(const Transformation &t)`
Makes a transformation matrix entity using the transformation `t`.

`void set_rotation(const RotMtx &m)`
Sets the rotation matrix to `m`.

`RotMtx get_rotation() const`
Returns the rotation matrix.

`void set_translation(const TransVec &v)`
Sets the translation vector to `v`.

`TransVec get_translation() const`
Returns the translation vector.

`virtual Transformation get_transformation() const`
Returns the combined transformation.

`bool maintains_handedness() const`
Returns true if the matrix maintains the handedness of the coordinate system: i.e. if the determinant of the rotation matrix is positive.

`bool maintains_combined_handedness() const`
Returns true if the matrix, when combined with the rotation of the defining transformation, maintains the handedness of the coordinate system.

6.2 Point Entity

An IGES Point Entity describes a single point in space as well as an optional Subfigure Definition (see Section 6.8) used as a display symbol for the point. The IGES Point Entity has Entity Type Number 116.

`PointEntity` has the following public members.

```
PointEntity(const Point &p = Point(0.0,0.0,0.0),
            SingSubFigInstEntity *ds = 0)
```
Makes a point at `p` with display symbol `ds`. If `ds` is null, there is no display symbol defined. If `ds` is not null, then it becomes a physically dependent subordinate entity.

`void set_point(const Point &p)`
Sets the point to `p`.

`Point get_point() const`
Returns the point in definition space.

`Point get_transformed_point() const`
: Returns the point transformed to model space using the combined transformation.

`void set_display_symbol(SubFigDefEntity *sf)`
: Sets the Subfigure Definition Entity (see Section 6.8) defining the display symbol. The display symbol is made physically dependent.

6.3 Curve entities

A curve entity is an entity that describes a curve in space. Each curve entity can be represented as an IGES Entity as well as a CurveLib curve having a single parameter and returning a point in space (a `Point`).

Curve entities are represented by the class `CurveEntity` which is derived from `Entity`. Each `CurveEntity` stores a curve of type `CurveType` (see Section 3) which returns the value of the curve in the entity definition space.

`CurveEntity` has the following public members in addition to those inherited from `Entity`:

`CurveType get_curve() const`
: Returns a CurveLib representation of the curve. The value of the curve is a point in the entity definition space.

`CurveType get_transformed_curve() const`
: Returns the curve transformed by the transformation matrix: i.e. a curve whose value is a point in the entity model space.

6.3.1 Line Entity

The IGES Line Entity describes a straight line between two points. It has Entity Type Number 110.

Line entities are represented by the class `LineEntity`, a derivation of `CurveEntity`, which has the following public members in addition to those inherited from its base classes.

`LineEntity()`
: Makes a line entity. The end points are undefined.

`LineEntity(const Point &y0, const Point &y1)`
: Makes a line entity having end points `y0` and `y1`.

`void set_end_points(const Point &y0, const Point &y1)`
Sets the end points of the line.

`void get_end_points(Point &y0, Point &y1) const`
Returns the end points of the line.

The parameter of the line varies between 0 and 1: i.e. if $\mathbf{p}_1$ and $\mathbf{p}_2$ are the two points, then the curve returned by `get_curve()` (inherited from `CurveEntity`) is:

$$\mathbf{c}(t) = \mathbf{p}_1 + t(\mathbf{p}_2 - \mathbf{p}_1) \tag{7}$$

6.3.2 Circular Arc Entity

The IGES Circular Arc Entity describes a circular arc lying in a plane with z constant. It has Entity Type Number 100.

Circular Arc entities are represented by the class `CircularArcEntity` which is derived from `CurveEntity` and has the following public members in addition to those inherited from its base classes.

`CircularArcEntity()`
Default constructor. The entity remains undefined.

```
CircularArcEntity(const Point &c, Float r,
                  Angle<Float> theta1 = Angle<Float>(0.0),
                  Angle<Float> theta2 = Angle<Float>(0.0))
```
Makes a circular arc of radius `r` with center at `c` starting at `theta1` and proceeding to `theta2`, where `theta1` and `theta2` are the angles that a radius vector makes with the x axis. The curve proceeds counterclockwise around the $\hat{z}$ axis in the xy plane. If `theta1` and `theta2` are identical, the arc will be a full circle.

```
void define(const Point &c, Float r,
            Angle<Float> theta1 = Angle<Float>(0.0),
            Angle<Float> theta2 = Angle<Float>(0.0))
```
Defines the circular arc. The arguments are similar to the arguments of the constructor.

The curve returned by the function `get_curve()` (inherited from `CurveEntity`) has the following parameterization:

$$\mathbf{x}(\theta) = \mathbf{x}_c + r(\hat{x}\cos\theta + \hat{y}\sin\theta) \tag{8}$$

Its parameter range is $[\theta_1, \theta_2]$ where θ_1 is the angle equivalent to `theta1` in the range $[0, 2\pi)$ and θ_2 is the angle equivalent to `theta2` in the range $(\theta_1, \theta_1 + 2\pi]$.

6.3.3 Conic Arc Entity

The IGES Conic Arc Entity describes a conic arc lying in a plane with z constant. It has Entity Type Number 104. The conic arc is represented by an equation of the form

$$ax^2 + bxy + cy^2 + dx + ey + f = 0 \tag{9}$$

where at least one of a, b and c is non-zero. For simplicity in the following calculations, we will also require that $a + c \geq 0$. If not, all the coefficients can simply be negated.

In order to simplify the representation, we rotate the (x, y) coordinate system so that the principal axes of the conic are aligned with the axes of the coordinates. To do this rewrite Equation (9) as

$$\mathbf{x}^T\mathbf{M}\mathbf{x} + \mathbf{x}^T\mathbf{V} + f = 0; \quad \mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix}; \quad \mathbf{M} = \begin{bmatrix} a & b/2 \\ b/2 & c \end{bmatrix}; \quad \mathbf{V} = \begin{bmatrix} d \\ e \end{bmatrix} \tag{10}$$

The eigenvalues of $\mathbf{M}$ are

$$\lambda_\pm = \tfrac{1}{2}\left[a + c \pm \sqrt{(a-c)^2 + b^2}\right] \tag{11}$$

Note that $\lambda_+ > 0$, since we have required that $a + c \geq 0$.

Define

$$\mathbf{A} = \frac{2}{\sqrt{b^2 + 4(a - \lambda_2)^2}} \begin{bmatrix} c - \lambda_1 & -b/2 \\ b/2 & \lambda_2 - a \end{bmatrix} \tag{12}$$

where λ_1 is one of λ_+ or λ_- and λ_2 is the other one. Then it is easily verified that $\mathbf{A}$ is unitary and that

$$\mathbf{M}' \equiv \mathbf{A}\mathbf{M}\mathbf{A}^T = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} \tag{13}$$

Therefore the equation for the conic can be rewritten:

$$\mathbf{x}'^T\mathbf{M}'^T\mathbf{x}' + \mathbf{x}'^T\mathbf{V}' + f = 0; \qquad \mathbf{x}' \equiv \mathbf{A}\mathbf{x}; \qquad \mathbf{V}' \equiv \mathbf{A}\mathbf{V} \equiv \begin{bmatrix} d' \\ e' \end{bmatrix} \tag{14}$$

or simply

$$\lambda_1 x'^2 + \lambda_2 y'^2 + d'x' + e'y' + f = 0 \tag{15}$$

If λ_2 is zero, then the conic is a parabola which can be represented by:

$$y' = -\frac{\lambda_1 x'^2 + d'x' + f}{e'} \tag{16}$$

with x' varying from x'_1 to x'_2. Note that if $e' = 0$, the conic degenerates. Similarly if λ_1 is zero, we have

$$x' = -\frac{\lambda_2 y'^2 + e'y' + f}{d'} \tag{17}$$

with y' varying from y'_1 to y'_2.

If λ_1 and λ_2 are both positive, then the conic is an ellipse.

$$\lambda_1 \left(x' + \frac{d'}{2\lambda_1} \right)^2 + \lambda_2 \left(y' + \frac{e'}{2\lambda_2} \right)^2 = f'; \quad f' \equiv \frac{(d')^2}{4\lambda_1} + \frac{(e')^2}{4\lambda_2} - f \tag{18}$$

Note that in this case the right hand side is required to exceed zero. The curve is parameterized as follows:

$$x' = r_1 \cos(t) - \frac{d'}{2\lambda_1}; \quad y' = r_1 \sin(t) - \frac{e'}{2\lambda_2}; \quad r_1 = \sqrt{\frac{f'}{\lambda_1}}; \quad r_2 = \sqrt{\frac{f'}{\lambda_2}}; \tag{19}$$

The parameter t varies between t_1 and t_2 such that $0 \leq t_1 < 2\pi$ and $t_2 - t_1 \leq 2\pi$.

If one of the eigenvalues is negative – we can choose it to be λ_2 – then the conic is a hyperbola which is represented parametrically by

$$x' = r_1 \sec(t) - \frac{d'}{2\lambda_1}; \quad y' = r_1 \tan(t) - \frac{e'}{2\lambda_2}; \tag{20}$$

The parameter t varies between t_1 and t_2 such that both t_1 and t_2 are in the range $[-\pi/2, \pi/2]$.

Conic arcs entities are represented by the class `ConicArcEntity` which is derived from `CurveEntity` and has the following public members in addition to those inherited from its base classes.

`enum Type { unknown, parabola, ellipse, hyperbola }`
: Used to indicate the type of conic.

`ConicArcEntity()`
: Default constructor. The conic entity remains undefined.

```
ConicArcEntity(Float a, Float b, Float c, Float d, Float e,
               Float f, Float x1, Float y1, Float x2, Float y2,
               Float z, Float acc = 1.0e-04)
```
: Makes a conic arc with defining coefficients `a`, `b`, `c`, `d`, `e` and `f`: i.e. the conic is defined by

$$ax^2 + bxy + cy^2 + dx + ey + f = 0 \tag{21}$$

: The curve is traced from `(x1,y1,z)` to `(x2,y2,z)`; a warning is sent if these two points do not lie on the conic to within `acc`. An appropriate value for `acc` is the minimum resolution of the model to which the conic arc entity belongs.

```
void define(Float a, Float b, Float c, Float d, Float e,
            Float f, Float x1, Float y1, Float x2, Float y2,
            Float z, Float acc = 1.0e-04)
```

Redefines the conic arc. The arguments are similar to those of the constructor above.

```
void define_parabola(Float a, Float x1, Float x2)
```

Defines the conic arc to be the parabola $y = ax^2$ in the plane $z = 0$ for x between `x1` and `x2`.

```
void define_ellipse(Float a, Float b,
                    Angle<Float> theta1 = Angle<Float>(0.0),
                    Angle<Float> theta2 = Angle<Float>(0.0))
```

Defines the conic arc to be the ellipse

$$x = |a| \cos\theta; \qquad y = |b| \sin\theta; \qquad z = 0 \tag{22}$$

for $\theta \in [\theta_1, \theta_2]$ where θ_1 is the angle in $[0, 2\pi)$ that corresponds to `theta1` and θ_2 is the angle in $(\theta_1, \theta_1 + 2\pi]$ that corresponds to `theta2`. If the two angles are the same, the full ellipse will be traced. The coefficients `a` and `b` must not be zero; if they are a `ProgError` is thrown.

The defining equation for the ellipse is

$$\left(\frac{x}{a}\right)^2 + \left(\frac{y}{b}\right)^2 = 1 \tag{23}$$

Its principal axes have lengths a and b.

```
void define_hyperbola(Float a, Float b,
                      Angle<Float> theta1, Angle<Float> theta2)
```

Defines the conic arc to be the hyperbola

$$x = a \sec\theta; \qquad y = b \tan\theta; \qquad z = 0 \tag{24}$$

for θ between `theta1` and `theta2`. Both `theta1` and `theta2` must be in the range $[-\pi/2, \pi/2]$. The defining equation for the hyperbola is

$$\left(\frac{x}{a}\right)^2 - \left(\frac{y}{b}\right)^2 = 1 \tag{25}$$

```
Type get_type() const
```

Returns the type of conic.

6.3.4 Parametric Spline Curve Entity

The IGES Parametric Spline Curve Entity describes a curve represented as a PP-spline (see Reference 5, Section 4 for a description of PP-splines). It has Entity Type Number 112.

Parametric spline curve entities are represented by the class `PPSplineEntity` which is derived from `CurveEntity` and has the following public members in addition to those inherited from its base classes.

`typedef Spline::PPSpline<Point,Float> SplineType`
: The type of the spline curve: described in Reference 5, Section 5.

`PPSplineEntity(const SplineType &s)`
: Makes a `PPSplineEntity` for the spline surface `s`. The parameter range is determined from the first and last knot of `s`.

`void define(const SplineType &s)`
: Defines the spline using spline curve `s`. The parameter range is determined from the first and last knot of `s`.

The class `HermiteSpline<Point,Float>` in namespace `Spline` (described in Reference 5, Section 8.2) provides a means for approximating any differentiable curve by a PP-spline. Since it is derived from `PPSpline<Point,Float>`, it can also be used to generate a `PPSplineEntity` which approximates any curve. For example, suppose that `c` is a `CurveType` representing some curve. Then a `PPSplineEntity` approximating `c` to accuracy `tol` can be created using:

```
using namespace Spline;
Float xlo = c.low_range(0), xhi = c.high_range(0);
HermiteSpline<Point,Float> spline(c,xlo,xhi,tol);
PPSplineEntity pp_ent = new PPSplineEntity(spline);
```

Note that a `HermiteSpline<Point,Float>` is smooth but has discontinuous second derivatives. If continuity of second derivatives is important, it is better to use a `BSplineEntity` to represent to approximated curve: see Section 6.3.5. Reference 5, Section 11 contains more information about approximating curves with splines.

6.3.5 Rational B-spline curve entities

The IGES Rational B-Spline Curve Entity describes a curve represented as a B-spline curve or a non-uniform rational B-spline (NURB) curve (see Reference 5, Section 7 and 8 for descriptions of B-spline and NURB curves). It has Entity Type Number 126.

Rational spline curve entities are represented by the class `BSplineEntity` which is derived from `CurveEntity` and has the following public members in addition to those inherited from its base classes.

`typedef CurveType::RangeType RangeType`
: The type of a parameter range for the curve. This definition makes `RangeType` equivalent to a `ParamRange<1U,Float>` in namespace `CurveLib`; this class is discussed in Reference 3, Section 10.1.

`typedef Spline::BSpline<Point,Float> SplineType`
: The type of the curve if it is a simple B-spline curve: described in Reference 5, Section 8.1.

`typedef Spline::NURBSpline<Point,Float> NURBType`
: The type of the curve if it is a NURB curve: described in Reference 5, Section 9.1.

`BSplineEntity(const SplineType &b)`
: Makes a `BSplineEntity` for the spline curve `b`. The parameter range is determined from the knot sequence; if k is the order of `b` and its knots are x_i, then the parameter range is $[x_{k-1}, x_{N_k-k}]$ where N_k is the number of knots.

`BSplineEntity(const SplineType &b, const RangeType &r)`
: Makes a `BSplineEntity` for the spline curve `b` with parameter range `r`.

`BSplineEntity(const NURBType &b)`
: Makes a `BSplineEntity` for the NURB curve `b`. The parameter range is determined from the knot sequence; if k is the order of `b` and its knots are x_i, then the parameter range is $[x_{k-1}, x_{N_k-k}]$ where N_k is the number of knots.

`BSplineEntity(const NURBType &b, const RangeType &r)`
: Makes a `BSplineEntity` for the NURB curve `b` with parameter range `r`.

`void define(const SplineType &b, const RangeType &r)`
: Defines the B-spline curve entity using curve `b` and range `r`.

`void define(const NURBType &b, const RangeType &r)`
: Defines the B-spline curve entity using curve `b` and range `r`.

The class `BSpline<Point,Float>` in namespace `Spline` (described in Reference 5, Section 9.1) provides a means for approximating any differentiable curve using a B-spline curve. Therefore a `BSplineEntity` can be used to approximate any curve. For example, suppose that `c` is a `CurveType` representing some curve. Then a `BSplineEntity` approximating `c` to accuracy `tol` can be created using:

```
using namespace Spline;
unsigned k = 4;
BSpline<Point,Float> spline(k,c,c.low_range(0),c.high_range(0),tol);
BSplineEntity pp_ent = new BSplineEntity(spline);
```

Here `k` is the order of the B-spline which we have chosen to be 4 (a cubic spline). In general, a B-spline will have two fewer continuous derivatives than its order: e.g. the spline defined above will have continuous second derivatives (`k-2 = 2`). Reference 5, Section 11 contains more information about approximating curves with splines.

6.3.6 Composite Curve Entity

The IGES Composite Curve Entity describes a continuous curve generated by concatenating several sub-curves. It has Entity Type Number 102. A Composite Curve Entity maintains an ordered list of the sub-curves of which it is comprised. Point Entities may also be added to the composite curve with the following caveats:

1. two points cannot appear as consecutive elements of the composite curve;
2. the location of a point must agree with the end point of a sub-curve that immediately precedes it and with the start point of a sub-curve that follows it.

The IGES standard also allows a Connect Point Entity to be part of a Composite Curve Entity but they have not been implemented here.

Let $\mathbf{c}_i(u)$ be the i^{th} sub-curve and let its parameter range be $a_i \leq u \leq b_i$. Define an increasing sequence of parameter values, u_i by:

$$u_0 = a_0 \tag{26}$$

$$u_{i+1} = u_i + b_i - a_i; \qquad \text{for } i \geq 0 \tag{27}$$

Then if there are N sub-curves, the composite curve, $\mathbf{c}(u)$ is defined on the range $[u_0, u_N]$ by:

$$\mathbf{c}(u) = \mathbf{c}_i(u - u_i + a_i); \qquad \text{for } u_i \leq u \leq u_{i+1} \tag{28}$$

IGES requires that the concatenated curve be continuous: i.e. the end-point of a sub-curve must be the same as the start-point of the following sub-curve to within the minimum resolution of the `Model` which contains the `CompositeCurveEntity`.

Composite Curve Entities are represented by the class `CompositeCurveEntity` which is derived from `CurveEntity`. The curve described by the Composite Curve Entity is represented by a `GeneralSpline<Point,Float>` in namespace `Spline`: see Reference 5, Section 4. `CompositeCurveEntity` has the following public members in addition to those inherited from its base classes.

`typedef Spline::GeneralSpline<Point,Float> SplineType`
: The type of the spline used to implement the composite curve.

`CompositeCurveEntity()`
: Default constructor. The list of sub-curves and points is empty.

`void clear()`
: Clears the list of entities which comprise the composite curve.

`void add(Entity *e)`
: Adds an entity to the curve. The entity must be a `PointEntity` or an entity derived from `CurveEntity` but not including `CompositeCurveEntity`. The entity becomes a physically dependent subordinate entity of the composite curve. If the entity is a point, its location will be adjusted to be the end of the curve that precedes it (or the beginning of the curve that follows it if the point is the first entity).

  The order of the sub-curves is set by the order in which they are added to the Composite Curve Entity: the first curve added will be the first curve in the list of sub-curves.

`SplineType get_spline() const`
: Returns the general spline used to implement the composite curve. The knots of the spline (available via the member function `SplineType::get_knots()`) mark the locations where the sub-curves have been joined.

`Float get_mismatch()`
: Returns the maximum mismatch between the ends of the sub-curves. This function can be used to check the requirement that the end-points of contiguous sub-curves match to within the minimum resolution of the model.

6.3.7 Offset Curve Entity

The IGES Offset Curve Entity describes a curve generated by specifying an offset from another curve. The curve to be offset must lie in a plane and have continuous first derivative. If $\mathbf{c}(u)$ denotes the curve to be offset, parameterized by u, then the offset curve is given by:

$$\mathbf{x}(u) = \mathbf{c}(u) + f(s)\,\hat{\mathbf{n}} \times \hat{\mathbf{t}}(u); \qquad \text{for } u_1 \le u \le u_2 \tag{29}$$

where $\hat{\mathbf{n}}$ is a normal to the plane in which the curve lies and $\hat{\mathbf{t}}(u)$ is a unit tangent to the curve. The function $f(s)$ gives the offset distance. Its parameter s can be interpreted in two ways:

1. It is equivalent to the parameter of the curve to be offset, u; or

2. It is equivalent to arclength along $\mathbf{c}(u)$ starting at $\mathbf{c}(u_1)$.

The Offset Curve Entity has Entity Type Number 130.

Offset Curve Entities are represented by the class `OffsetCurveEntity` which is derived from `CurveEntity` and has the following public members in addition to those inherited from its base classes.

`OffsetCurveEntity()`
: Default constructor. The entity remains undefined.

`OffsetCurveEntity(CurveEntity *c, Float d, const Normal &n)`
: Makes an Offset Curve Entity in which `c` is offset uniformly by `d`. The curve in `c` should lie in a plane whose unit normal is `n`.

  The curve entity `c` becomes a physically dependent subordinate entity of the offset curve.

`OffsetCurveEntity(CurveEntity *c, Float d1, Float d2, const Normal &n)`
: Makes an Offset Curve Entity in which the offset of `c` varies linearly from `d1` at the start of `c`, to `d2` and the end of `c`. The parameterization of the offset is the same as the parameterization of `c`. The curve in `c` should lie in a plane whose unit normal is `n`.

  The curve entity `c` becomes a physically dependent subordinate entity of the offset curve.

```
OffsetCurveEntity(CurveEntity *c, Float p1, Float d1,
                  Float p2, Float d2,
                  bool use_arclength, const Normal &n)
```
: Makes an Offset Curve Entity in which the offset of `c` varies linearly from `d1` at `p1` to `d2` at `p2`. If `use_arclength` is true, `p1` and `p2` are arclengths along `c`; otherwise they are parameters of `c`. The curve in `c` should lie in a plane whose unit normal is `n`.

  The curve entity `c` becomes a physically dependent subordinate entity of the offset curve.

```
OffsetCurveEntity(CurveEntity *c, CurveEntity *ce, int comp,
                  bool use_arclength, const Normal &n)
```
: Makes an Offset Curve Entity in which the curve define by `c` is offset by the function $f(s)$, where $f(s)$ is component `comp` (0, 1 or 2) of the curve defined by curve entity `ce`. If `use_arclength` is true, the value of this component is interpreted as the arclength along the curve; otherwise it is interpreted as the parameter of `c`.

  The curve in `c` should lie in a plane whose unit normal is `n`.

The curve entities `c` and `ce` become physically dependent subordinate entities of the offset curve.

`void define(CurveEntity *c, Float d, const Point &n)`
Defines the Offset Curve Entity in a manner similar to the constructor with the same arguments.

`void define(CurveEntity *c, Float d1, Float d2, const Normal &n)`
Defines the Offset Curve Entity in a manner similar to the constructor with the same arguments.

```
void define(CurveEntity *c, Float p1, Float d1, Float p2, Float d2,
            bool use_arclength, const Normal &n)
```
Defines the Offset Curve Entity in a manner similar to the constructor with the same arguments.

```
void define(CurveEntity *c, CurveEntity *ce, int comp,
            bool use_arclength, const Normal &n)
```
Defines the Offset Curve Entity in a manner similar to the constructor with the same arguments.

`void set_range(Float u1, Float u2)`
Sets the parameter range for the offset curve. The range must be contained in the range of the curve being offset.

6.3.8 Curves on surfaces

IGES provides two entities for specifying a curve which is embedded in a surface: Curve on a Parametric Surface Entity and Boundary Entity. The common features of these two entities are included in the base class `BaseSurfaceCurveEntity` derived from `CurveEntity`.

The surface containing the curve will be called $\mathbf{S}$. IGES does not require $\mathbf{S}$ to be parametric (i.e. $\mathbf{S} \equiv \mathbf{S}(u, v)$, where u and v are scalar parameters) but in the classes defined here all surfaces are parametric; the only non-parametric IGES surface entity that is supported is the Plane Entity, and that is given a parameterization so that it can be represented in CurveLib (see Section 6.4.1).

The curve is represented using the class `SurfaceCurve<Point,Float>` from namespace `CurveLib` (see Section 5) which contains a CurveLib representation of the surface as well as model space and composed representations of the curve. The surface is also represented as a `SurfaceEntity`. The surfaces stored by the `SurfaceEntity` and the `SurfaceCurve<Point,Float>` are the same.

The `SurfaceEntity` used to represent the surface is *not* subordinate to the `BaseSurfaceCurveEntity`; therefore the visibility of the surface is independent of the visibility of the curve.

`BaseSurfaceCurveEntity` has member functions which determine which representation of the curve will be returned by the function `get_curve()` inherited from `CurveEntity`. Entities derived from `BaseSurfaceCurveEntity` also have entries in the IGES file which specify which representation is preferred by the system that generated the file.

`BaseSurfaceCurveEntity` has the following public members in addition to those inherited from its base classes:

`typedef CurveLib::SurfaceCurve<Point,Float> SurfaceCurveType`
: The type of the surface curve containing both model space and composed representations.

`enum PrefRep { unknown, composed, model_space, equal }`
: The type of a flag which indicates which representation of the curve is preferred in the system that generated the curve (see Section 5 for a description of the two curve representations).

  `unknown`
  : The preferred representation is unspecified.

  `composed`
  : The composed curve is preferred.

  `model_space`
  : The model space curve is preferred.

  `equal`
  : The composed and model space representation have equal preference.

`PrefRep get_preferred_representation() const`
: Returns the preferred representation of the curve.

`void set_preferred_representation(PrefRep pr)`
: Sets the preferred representation of the curve to `pr`. If `pr` is `unknown` or `equal`, then the curve returned by `get_curve()` will not change. If `pr` is `composed`, the curve returned by `get_curve()` will be the composed curve. If `pr` is `model_space`, the curve returned by `get_curve()` will be the model space curve.

`const SurfaceEntity *get_surface_entity() const`
: Returns the entity used to specify the surface in which the curve is embedded.

`SurfaceCurveType get_surface_curve() const`
Returns the surface curve.

`SurfaceCurveType get_transformed_surface_curve() const`
Returns the surface curve transformed into model space using the combined transformation for the entity.

6.3.8.1 Curve on a Parametric Surface Entity

The IGES Curve on a Parametric Surface Entity describes a curve embedded in a parametric surface $\mathbf{S}(u, v)$. The curve is required to have a composed representation. The Curve on a Parametric Surface Entity has Entity Type Number 142.

A Curve on a Parametric Surface Entity is represented by `CurveOnSurfaceEntity` which is derived from `BaseSurfaceCurveEntity` and has the following public members in addition to those inherited from its base classes.

`CurveOnSurfaceEntity()`
Default constructor. The entity remains undefined.

```
CurveOnSurfaceEntity(SurfaceEntity *s, CurveEntity *c,
                     CurveEntity *dc = 0, Float tolerance = 1.0e-04)
```
Makes a Curve on a Parametric Surface Entity defined by the parametric curve `c` in the surface `s`. The surface entity `s` may not be a Plane Entity; IGES does not consider Plane Entities to be parametric.

The curve defined by `c` is three-dimensional. To reduce it to the two-dimensional parameter space of the surface, `s`, the third component is ignored.

If `dc` is non-null, it defines the model space representation of the curve. If `dc` is null, a model space representation of the curve is made by approximating the composed representation of the curve with a PP-spline. The difference of the spline curve from the composed curve will not exceed `tolerance`. The value of `tolerance` should be set in accordance with the minimum resolution of the `Model` to which the `CurveOnSurfaceEntity` is assigned.

The entities `c` and `dc` become physically dependent subordinate entities of the Curve on a Parametric Surface entity; `c` is also made invisible and its intended use is set to `parametric_2D`. The entity `s` is *not* subordinate to the `CurveOnSurfaceEntity`.

The method of generation of the curve is `unspecified` (see below) and its preferred representation is `composed`.

```
CurveOnSurfaceEntity(SurfaceEntity *s, int param, Float val,
                     Float tolerance = 1.0e-04)
```
Makes a Curve on a Parametric Surface Entity which is an isoparametric line

of `s`: parameter `param` (either 0 or 1) has a constant value of `val`. The surface entity `s` must not be a Plane Entity.

The model space representation of the curve is made by approximating the composed representation with a PP-spline. The difference of the spline curve from the composed curve should not exceed `tolerance`. The value of `tolerance` should be set in accordance with the minimum resolution of the `Model` to which the `CurveOnSurfaceEntity` is assigned. The model space representation of the curve will inherit its visibility from the Curve on a Parametric Surface Entity.

The entity `s` is *not* subordinate to the `CurveOnSurfaceEntity`.

There is no `define()` function corresponding to this constructor; if `define()` were called after the entity was added to a `Model`, the entities defining composed and model space representations of the curve (defined implicitly by the constructor) would never be added to the model.

The method of generation of the curve is `iso_parametric` (see below) and its preferred representation is `composed`.

```
CurveOnSurfaceEntity(SurfaceEntity *s,
                     const SurfaceType::ParamType &p1,
                     const SurfaceType::ParamType &p2,
                     Float tolerance = 1.0e-04)
```

Makes a Curve on a Parametric Surface Entity which is straight line between the points `p1` and `p2`. As with the previous constructor, there is no `define()` function corresponding to this constructor.

The model space representation of the curve is made by approximating the composed representation with a PP-spline to an accuracy of `tolerance`. The model space representation of the curve will inherit its visibility from the Curve on a Parametric Surface Entity.

The entity `s` is *not* subordinate to the `CurveOnSurfaceEntity`.

The method of generation of the curve is `unspecified` (see below) and its preferred representation is `composed`.

```
void define(SurfaceEntity *s, CurveEntity *c, CurveEntity *dc = 0,
            Float tolerance = 1.0e-04)
```

Defines the Curve on a Parametric Surface Entity in a manner similar to the constructor with the same arguments.

```
HowFlag get_method_of_generation() const
```

Returns the method of generation of the curve. `HowFlag` is an `enum` local to `CurveOnSurface` having the following values:

`unspecified`
: The method of generation is unspecified.

`projection`
: The curve is a projection onto the surface.

`intersection`
: The curve is an intersection of two surfaces.

`iso_parametric`
: Either the first or second parameter of the surface is held constant.

`void set_method_of_generation(HowFlag h)`
: Sets the method of generation of the curve to `h`.

`const CurveEntity* get_parametric_entity() const`
: Returns the IGES entity defining the curve in the parameter space of the surface.

`const CurveEntity* get_model_space_entity() const`
: Returns the IGES entity defining the model space representation of the curve.

`void reverse()`
: Reverses the direction of the CurveLib curve used to represent the curve on a surface, but retains the same parameter range: i.e. as the parameter of the the curve is increased from its lowest to highest allowed value, the curve is traced from its end-point to its start point. The list of possible discontinuities in the curve is also reversed.

  One of the primary uses of a Curve On a Parametric Surface Entity is as a boundary curve of a Trimmed Surface Entity (see Section 6.5.1). To meet the requirements of the base class `LimitedSurfaceEntity` (see Section 6.5) from which the class `TrimmedSurfaceEntity` is derived, the boundary curve must be oriented counter-clockwise (relative to the surface normal) around the portion of the surface to be retained after trimming. However, IGES does not require this of the Curve On a Parametric Surface entities used in a Trimmed Surface Entity, so a Curve On a Parametric Surface Entity read from an IGES file may be incorrectly oriented. The purpose of this function is to allow the orientation of the curve to be reversed so that it is correctly oriented when used by a `LimitedSurfaceEntity`.

`bool is_reversed() const`
: True if the orientation of the curve has been reversed: see `reverse()`.

6.3.8.2 Boundary Entity

The IGES Boundary Entity, which has Entity Type Number 141, defines a surface boundary consisting of a set of parametric curves on the surface. The curves are combined using a `GeneralSpline<Point,Float>` from namespace `Spline` (see Reference 5, Section 5) to form a single closed curve representing the boundary. We will

refer to each of the curves comprising the boundary as a segment. The segments are ordered and, when concatenated, must generate a closed curve in model space.

The segments are concatenated into a single curve in the same manner as the sub-curves of a `CompositeCurveEntity`: the parameter ranges of the segments are also concatenated to form the parameter range of the curve. Let $\mathbf{c}_i(u)$ be the curve for the i^{th} segment and let its parameter range be $a_i \leq u \leq b_i$. Define an increasing sequence of parameter values, u_i by:

$$u_0 = a_0 \tag{30}$$

$$u_{i+1} = u_i + b_i - a_i; \qquad \text{for } i \geq 0 \tag{31}$$

Then if there are N sub-curves, the boundary curve, $\mathbf{c}(u)$ is defined on the range $[u_0, u_N]$ by:

$$\mathbf{c}(u) = \mathbf{c}_i(u - u_i + a_i); \qquad \text{for } u_i \leq u \leq u_{i+1} \tag{32}$$

Each segment must include a model space representation. It may also specify an ordered collection of curves in the surface parameter space such that, when composed in order with the surface $\mathbf{S}$, they generate the segment. The parameter space curves from all the segments are concatenated to generate a single curve in parameter space. This is composed with the CurveLib representation of the surface to generate a single composed representation of the boundary. The complete concatenated curve is represented by a `SurfaceCurve<Point,Float>` from namespace `CurveLib`. There is no requirement that the parameter curves form a closed region in parameter space (see Section 5 for more on this point).

The struct `BdyCurveSegment` is used to represent a single segment of one of the boundary curves. It has the following public members.

`CurveEntity *space_curve_entity`
: An entity representing the curve for the segment in model space.

`bool reversed`
: A flag indicating whether the model space curve must be reversed when included in the boundary.

`std::list<CurveEntity*> param_curves`
: A list of parametric curves which generate the model space curve. If `reversed` is true, the parametric curves generate the reversed model space curve: therefore, the parametric curves always generate the curve as it is included in the boundary.

  If the list of parameter curves is empty, no composed representation of the segment has been specified.

Boundary entities are represented by the class `BoundaryEntity` derived from the base class `BaseSurfaceCurveEntity` and having the following public members in addition to those inherited from its base classes.

`typedef std::list<BdyCurveSegment> BdySegmentList`
: The type of the list of segments which comprise the curve.

`BoundaryEntity(SurfaceEntity *s = 0)`
: Makes a Boundary Entity which bounds a region of the surface defined by `s`: `s` must be either a `PlaneEntity` or a `SurfaceEntity`. The curves comprising the boundary curve remain undefined.

`void clear()`
: Removes all segments from the curve.

`void add_segment(BdyCurveSegment seg, Float tolerance = 1.0e-04)`
: Adds a segment to the boundary curve. If there are already segments added, the first point of `seg` must match the last point of the last segment added.

  The entity defining the model space curve and all the entities defining the parameter curve will be made physically dependent.

  If `seg.space_curve_entity` is null (i.e. the segment has no model space curve entity defined), then a model space curve entity will be generated using a PP-spline approximation of the composed representation of the segment. The difference of the spline curve from the composed curve will not exceed `tolerance`. The value of `tolerance` should be set in accordance with the minimum resolution of the `Model` to which the `BoundaryEntity` is assigned.

  If the `SurfaceEntity` representing the surface (argument `s` in the constructor) is a `PlaneEntity`, `seg.param_curves` must be empty: i.e. no parametric representation of the segment is given. However, in the `SurfaceCurveType` returned by `get_surface_curve()`, a parametric representation of the curve *will* be defined; it is generated automatically from the model space representation of the segment and the definition of the plane.

`const BdySegmentList& get_segments() const`
: Returns a list of segments comprising the boundary.

`Float get_mismatch()`
: Returns the maximum mismatch between the ends of the segments in model space. This should not exceed the minimum resolution of the `Model` to which the `BoundaryEntity` is assigned.

`Float get_parametric_mismatch() const`
: Returns the maximum mismatch in parameter space of the end-points of the

parametric curves for each of the segments. There is no requirement that the parametric mismatch be small.

6.4 Surface entities

A surface entity is an entity that describes a parametric surface in space. Each surface entity can be represented as an IGES Entity as well as a CurveLib surface having a two parameters and returning a point in space (a `Point`).

Surface entities are represented by the class `SurfaceEntity` which is derived from `Entity`. Each `SurfaceEntity` stores a surface of type `SurfaceType` (see Section 3) which returns the value of the surface in the entity definition space.

`SurfaceEntity` has the following public members in addition to those inherited from `Entity`:

`SurfaceType get_surface() const`
: Returns the surface. The value of the surface is a point in the entity definition space.

`SurfaceType get_transformed_surface() const`
: Returns the surface transformed into model space using the combined transformation for the entity.

6.4.1 Unbounded Plane Entity

In IGES a plane is defined by an equation of the form

$$ax + by + cz = d \tag{33}$$

It also requires the coordinates of a symbol to mark the location of the plane and a single floating point number to indicate the size of the symbol. The IGES Plane Entity has Entity Type Number 108.

An IGES Plane Entity can be either unbounded (the Form Number is 0) or bounded (the Form Number is 1 or -1). An unbounded plane is a simple surface while a bounded plane is a type of limited surface; therefore they are represented by different classes. This section describes the class `PlaneEntity` used to represent unbounded planes. Bounded planes are represented by the class `BoundedPlaneEntity` described in Section 6.5.3.

IGES does not specify a parameterization for the plane. We define one here so that the plane can be represented by a CurveLib surface.

If $a = b = 0$, define the parameterized plane by

$$\mathbf{P}(u, v) = \begin{bmatrix} u \\ \text{sgn}(c)v \\ d/c \end{bmatrix} \tag{34}$$

where $\text{sgn}(x)$ is 1 if x is positive, -1 if x is negative.

If at least one of a and b is non-zero, we rewrite Equation (33) as

$$\hat{n} \cdot \mathbf{x} = \alpha \tag{35}$$

where

$$\hat{n} = \frac{1}{\sqrt{a^2 + b^2 + c^2}} \begin{bmatrix} a \\ b \\ c \end{bmatrix}; \quad \mathbf{x} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}; \quad \alpha = \frac{d}{\sqrt{a^2 + b^2 + c^2}} \tag{36}$$

Define two unit vectors by

$$\hat{x}_1 = \hat{n} \times \hat{z}; \quad \hat{x}_2 = \hat{n} \times \hat{x}_1 \tag{37}$$

Then the three unit vector $\hat{n}$, $\hat{x}_1$ and $\hat{x}_2$ are orthonormal. Define the parameterized plane by

$$P(u, v) = \alpha\hat{n} + u\hat{x}_1 + v\hat{x}_2 = \begin{bmatrix} \alpha n_x + \dfrac{un_y + vn_x n_z}{\sqrt{1 - n_z^2}} \\ \alpha n_y - \dfrac{un_x - vn_y n_z}{\sqrt{1 - n_z^2}} \\ \alpha n_z - v\sqrt{1 - n_z^2} \end{bmatrix} \tag{38}$$

A normal, $\mathbf{s}$, to the parameterized plane can be calculated using

$$\mathbf{s} = \frac{\partial \mathbf{P}}{\partial u} \times \frac{\partial \mathbf{P}}{\partial v} \tag{39}$$

For each of the parameterizations defined above, we get

$$\hat{s} \equiv \frac{\mathbf{s}}{|\mathbf{s}|} = \hat{n} \tag{40}$$

(This was the purpose of the $\text{sgn}(c)$ term in the parameterization when $a = b = 0$.)

The class `PlaneEntity` is derived from the base class `SurfaceEntity` and has the following public members in addition to those it inherits from its base classes:

`PlaneEntity()`
: Makes a plane through the origin with normal along the z axis. The display symbol has size 1 and is placed at the origin.

`PlaneEntity(Float a, Float b, Float c, Float d)`
: Defines an unbounded plane by specifying the coefficients of Equation (33). The display symbol has size 1 and is placed at the origin in parameter space: i.e. at $\mathbf{P}(0,0)$. The bounding curve must be set by a call to `set_bounding_curve()`.

`void set_equation(Float a, Float b, Float c, Float d)`
: Defines the plane by specifying the coefficients of the defining equation.

`void get_equation(Float &a, Float &b, Float &c, Float &d) const`
: Gets the coefficients of the equation defining the plane.

`void set_symbol_location(Float x, Float y, Float z)`
: Sets the location for the symbol marking the plane.

`void set_symbol_size(Float s)`
: Sets the size for the symbol marking the plane.

`Point get_normal() const`
: Returns a unit normal to the plane.

6.4.2 Parametric Spline Surface Entity

The IGES Parametric Spline Surface Entity describes a surface represented as a two-parameter PP-spline (see Reference 5, Section 4.1 for a description of two-parameter PP-splines). It has Entity Type Number 114.

Parametric spline surface entities are represented by the class `PPSpline2dEntity` which is derived from `SurfaceEntity` and has the following public members in addition to those inherited from its base classes.

`typedef Spline::PPSpline2d<Point,Float> SplineType`
: The type of the spline surface: described in Reference 5, Section 5.1.

`PPSpline2dEntity(const SplineType &s)`
: Makes an `PPSpline2dEntity` for the spline surface `s`. The range of the spline is determined from the knot sequences.

`void define(const SplineType &s)`
: Defines the spline surface using `s`. The range of the spline is determined from the knot sequences.

`SplineType get_spline_surface() const`
: Returns the spline surface.

6.4.3 Ruled Surface Entity

The IGES Ruled Surface Entity describes a surface represented as a ruled surface between two bounding curves. It has Entity Type Number 118.

Let $\mathbf{c}_i(t_i)$, for $i = 1$ or 2, represent the two bounding curves. Their parameter ranges are $a_i \leq t_i \leq b_i$. The parameters of these curves are mapped to a new parameter, u, which varies between 0 and 1. The ruled surface is then defined by

$$\mathbf{x}(u, v) = (1 - v)\,\mathbf{c}_1\big(t_1(u)\big) + v\,\mathbf{c}_2\big(t_2(u)\big) \tag{41}$$

The mappings $t_i(u)$ can be defined in two ways:

1. If the Form Number is 0, $t_i(u)$ is the fractional arclength along curve i.
2. If the Form Number is 1, the parameter range is mapped linearly onto $[0, 1]$ so that $t_i(u) = a_i + (b_i - a_i)u$.

A Ruled Surface Entity also allows the direction of parameterization of the second curve to be reversed so that the first point of $\mathbf{c}_1$ is joined to the last point of $\mathbf{c}_2$ and vice versa. In this case the ruled surface is defined by

$$\mathbf{x}(u, v) = (1 - v)\,\mathbf{c}_1\big(t_1(u)\big) + v\,\mathbf{c}_2\big(t_2(1 - u)\big) \tag{42}$$

Ruled surface entities are represented by the class `RuledSurfaceEntity` which is derived from `SurfaceEntity` and has the following public members in addition to those inherited from its base classes.

`RuledSurfaceEntity(CurveEntity *c1, CurveEntity *c2, bool rev = false)`
: Makes a ruled surface between the two curves `c1` and `c2`. The first parameter of the ruled surface is u (the parameter along the bounding curves); the second parameter is v (the parameter between the bounding curves). If `rev` is true, the first point of `c1` is joined to the last point of `c2`. The ruled surface uses arclength parameterization by default.

  The curve entities `c1` and `c2` become physically dependent subordinate entities of the Ruled Surface Entity.

```
void set_bounding_curves(CurveEntity *c1, CurveEntity *c2,
                         bool rev = false)
```
: Sets the bounding curves to `c1` and `c2`. If `rev` is true, the first point of `c1` is joined to the last point of `c2`.

  The curve entities `c1` and `c2` become physically dependent subordinate entities of the Ruled Surface Entity.

`void use_parameters()`
: Sets the surface so that $t_i(u) = a_i + (b_i - a_i)u$.

`void use_arclength()`

Sets the surface so that $t_i(u)$ is the fractional arclength along $\mathbf{c}_i$.

Each `RuledSurfaceEntity` uses a `CurveLib::RuledCurve<2U,Point,Float>` to represent the surface; it is described in Reference 3, Section 9.1. The bounding curves are obtained from the arguments of type `CurveEntity` passed to the constructor or to `set_bounding_curves()`. If arclength parameterization is used, each of the bounding curves is composed with an `Distrib::ArcLengthDistribution<Float>`: see Reference 6, Section 15.

6.4.4 Surface of Revolution Entity

The IGES Surface of Revolution Entity describes a surface defined by rotating a curve (the generatrix) around a line (the axis). It has Entity Type Number 120.

Let $\mathbf{T}(\theta)$ be the transformation that causes a rotation through angle θ counterclockwise about the axis (the direction of the axis is defined by the direction of the parameterization of the Line Entity defining the axis). The surface of revolution is defined by

$$\mathbf{x}(u,\theta) = \mathbf{T}(\theta)\mathbf{c}(u) \tag{43}$$

for $\theta_1 \le \theta \le \theta_2$ where $\mathbf{c}(u)$ is the generatrix. The range of the parameter u is defined by the curve entity which represents the generatrix.

Surface of revolution entities are represented by the class `RevSurfaceEntity` which is derived from `SurfaceEntity` and has the following public members in addition to those inherited from its base classes.

```
RevSurfaceEntity(LineEntity *line, CurveEntity *c,
                 Angle<Float> a1 = Angle<Float>(0.0),
                 Angle<Float> a2 = Angle<Float>(0.0))
```

Makes a surface of revolution with axis `line`, generatrix `c`, start angle `a1`, and end angle `a2`. The end angle, `a2`, is modified so that it lies between `a1` and `a1` $+ 2\pi$. If `a1` and `a2` are the same, then `a2` is increased to `a1` $+ 2\pi$.

The entities `line` and `c` both become physically dependent subordinate entities of the Surface of Revolution Entity.

```
void define(LineEntity *line, CurveEntity *c,
            Angle<Float> a1 = Angle<Float>(0.0),
            Angle<Float> a2 = Angle<Float>(0.0))
```

Defines the surface in a manner similar to the constructor with the same arguments.

6.4.5 Tabulated Cylinder Entity

The IGES Tabulated Cylinder Entity describes a surface defined by moving a line segment (the generatrix) parallel to itself along a curve (the directrix). It has Entity Type Number 120.

If $\mathbf{C}(u)$ is the directrix and $\mathbf{p}$ is the end-point of the generatrix when it is positioned at $\mathbf{C}(0)$, then the surface may be parameterized by:

$$\mathbf{X}(u, v) = \mathbf{C}(u) + v\big(\mathbf{p} - \mathbf{C}(0)\big) \tag{44}$$

The range of u is the range of the directrix and the range of v is $[0, 1]$.

Tabulated cylinder entities are represented by the class `TabulatedCylinderEntity` which is derived from `SurfaceEntity` and has the following public members in addition to those inherited from its base classes.

`TabulatedCylinderEntity(CurveEntity *crv, const Point &p)`
: Makes a tabulated cylinder with directrix `crv` and end point `p`. The entity `crv` becomes a physically dependent subordinate entity.

`void define(CurveEntity *crv, const Point &p)`
: Defines the tabulated cylinder in a manner similar to the constructor with the same arguments.

6.4.6 Rational B-spline Surface Entity

The IGES Rational B-Spline Surface Entity describes a curve represented as a two-parameter B-spline curve or a two-parameter non-uniform rational B-spline (NURB) curve (see Reference 5, Section 8.1 and 9.1 for descriptions of two-dimensional B-spline and NURB curves). It has Entity Type Number 128.

Rational spline surface entities are represented by the class `BSpline2dEntity` which is derived from `SurfaceEntity` and has the following public members in addition to those inherited from its base classes.

`typedef SurfaceType::RangeType RangeType`
: The type of a parameter range for the curve. This definition makes `RangeType` equivalent to a `ParamRange<2U,Float>` in namespace `CurveLib`; this class is discussed in Reference 3, Section 10.1.

`typedef Spline::BSpline2d<Point,Float> SplineType`
: The type of the surface if it is a simple B-spline surface: described in Reference 5, Section 8.1.

`typedef Spline::NURBSpline2d<Point,Float> NURBType`
: The type of the surface if it is a NURB surface: described in Reference 5, Section 9.1.

`BSpline2dEntity(const SplineType &b)`
: Makes an `BSpline2dEntity` for the spline surface `b`. The parameter ranges are determined from the knot sequences of `b`; if k_x and k_y are the orders of `b` in each parametric direction and its knots are $\{x_i : i \in [0, N_x - 1]\}$ and $\{y_j : j \in [0, N_y - 1]\}$, then the parameter ranges are $x \in [x_{k_x-1}, x_{N_x-k_x}]$ and $y \in [y_{k_y-1}, y_{N_y-k_y}]$.

`BSpline2dEntity(const SplineType &b, const RangeType &r)`
: Makes an `BSpline2dEntity` for the spline surface `b` with parameter range `r`.

`BSpline2dEntity(const NURBType &b)`
: Makes an `BSpline2dEntity` for the NURB surface `b`. The parameter range is determined from the knot sequences.

`BSpline2dEntity(const NURBType &b, const RangeType &r)`
: Makes an `BSpline2dEntity` for the NURB surface `b` with parameter range `r`.

`void define(const SplineType &b, const RangeType &r)`
: Defines the B-spline surface entity using surface `b` and range `r`.

`void define(const NURBType &b, const RangeType &r)`
: Defines the B-spline surface entity using surface `b` and range `r`.

6.4.7 Offset Surface Entity

The IGES Offset Surface Surface Entity describes a surface by offsetting another surface along its normals by a constant distance. Suppose that $\mathbf{S}(u, v)$ is the original surface and the offset distance is d. Then the offset surface is defined by

$$\mathbf{x}(u, v) = \mathbf{S}(u, v) + d\,\hat{\mathbf{n}}(u, v) \tag{45}$$

where $\hat{\mathbf{n}}(u, v)$ is the unit normal defined by

$$\mathbf{n}(u, v) = \frac{\partial \mathbf{S}}{\partial u} \times \frac{\partial \mathbf{S}}{\partial v}; \qquad \hat{\mathbf{n}}(u, v) = \frac{\mathbf{n}(u, v)}{|\mathbf{n}(u, v)|} \tag{46}$$

An Offset Surface has Entity Type Number 140.

Offset Surface entities are represented by the class `OffsetSurfaceEntity` which is derived from `SurfaceEntity` and has the following public members in addition to those inherited from its base classes.

`OffsetSurfaceEntity(SurfaceEntity *s, Float d)`
: Makes an offset surface using surface `s` offset by the distance `d`. The entity `s` becomes a physically dependent subordinate entity of the Offset Surface Entity.

`void define(SurfaceEntity *s, Float d)`
: Defines the offset surface in a manner similar to the constructor with the same arguments.

6.5 Limited surface entities

IGES has three entities which can be used to specify a limited surface: Trimmed (Parametric) Surface Entity, Bounded Surface Entity and a bounded Plane Entity. The class `LimitedSurfaceEntity` is a base class which encapsulates the common features of all classes used to specify limited surfaces. It stores the following:

1. An `SurfaceEntity` representing the surface whose domain is to be limited, **S**.
2. A list of entities defining the boundaries of the surface. Each of these entities is a `CurveEntity`.
3. A `LimitedSurfaceType` representing the limited surface.

The surface must conform to the requirements of any surface stored by an instance of the CurveLib class `LimitedSurface<Point,Float>`: see Section 5.

`LimitedSurfaceEntity` is a specialization of `Entity` and has the following public members in addition to those it inherits from `Entity`.

`typedef std::list<const CurveEntity*> BdyEntityList`
: The type of a list of curve entities describing the boundary curves.

`const SurfaceEntity* get_surface_entity() const`
: Returns the entity defining the surface whose domain is to be limited.

`const LimitedSurfaceType &get_limited_surface() const`
: Returns the limited surface in the entity definition space.

`LimitedSurfaceType get_transformed_limited_surface() const`
: Returns the limited surface transformed to model space using the entity's combined transformation.

`const BdyEntityList& get_boundary_entities() const`
: Returns a list of the boundary entities.

6.5.1 Trimmed (Parametric) Surface Entity

An IGES Trimmed (Parametric) Surface Entity represents a limited surface each of whose bounding curves is a Curve on a Parametric Surface Entity. It defines the surface to be trimmed, $\mathbf{S}(u, v)$, a closed outer boundary curve, $\mathbf{C}_0(t)$, and any number of closed inner boundary curves, $\mathbf{C}_n(t)$. The trimmed curve is the same as $\mathbf{S}(u, v)$ but has domain restricted to the intersection of the interior of the outer boundary and the exterior of all the inner boundaries. If the outer boundary is not specified, the boundary of the domain of $\mathbf{S}(u, v)$ is used instead.

IGES requires that $\mathbf{S}(u, v)$ have continuous normals throughout its domain, and that it has no coordinate singularities: i.e. the vectors $\partial\mathbf{S}/\partial u$ and $\partial\mathbf{S}/\partial v$ are linearly independent. Moreover, the inner boundaries, as well as their interiors, must be disjoint and must lie wholly within the interior of the outer boundary.

Trimmed (Parametric) Surface Entity places no restrictions on the direction of the boundary curves; the portion of the surface to be retained is determined solely from whether the boundary is designated as an inner or outer curve.

Each of the inner boundaries must be specified using a Curve on a Parametric Surface Entity (class `CurveOnSurfaceEntity`: see Section 6.3.8.1). If the outer boundary is specified explicitly, it must also be represented by a Curve on a Parametric Surface Entity.

A Trimmed (Parametric) Surface has Entity Type Number 144.

Trimmed Surface entities are represented by the class `TrimmedSurfaceEntity` which is derived from `LimitedSurfaceEntity` and has the following public members in addition to those inherited from its base classes.

`typedef std::list<const CurveOnSurfaceEntity*> CurveEntityList`
: The type of a list of the entities defining the inner boundaries.

`TrimmedSurfaceEntity(SurfaceEntity *s)`
: Makes a trimmed surface using surface `s`: `s` must not be a Plane Entity because IGES does not consider planes to be parametric surfaces. The entity `s` becomes a physically dependent subordinate entity and will inherit the visibility of the trimmed surface.

  The outer curve is defined to be the edge of the domain of `s` but can be changed using `set_outer_curve()`. The transformation matrix is the identity and the surface is independent and visible.

`void set_surface_entity(SurfaceEntity *s)`
: Sets the entity defining the surface to `s`: `s` must not be a Plane Entity. The

entity `s` becomes a physically dependent subordinate entity of the Trimmed Surface Entity.

`void set_outer_curve_entity(CurveOnSurfaceEntity *c)`
Sets the outer boundary to the curve defined by `c`. The entity `c` becomes a physically dependent subordinate entity of the Trimmed Surface Entity. If this function is never called, the outer boundary will be the edge of the domain of `s`.

`const CurveOnSurfaceEntity* get_outer_curve_entity() const`
Returns the entity defining the outer curve. Returns null if there is no outer boundary defined explicitly.

`void add_inner_curve_entity(CurveOnSurfaceEntity *c)`
Adds `c` to the list of inner boundaries. The entity `c` becomes a physically dependent subordinate entity of the Trimmed Surface Entity.

`CurveEntityList get_inner_curve_entities() const`
Returns the list of entities defining the inner boundaries.

6.5.2 Bounded Surface Entity

An IGES Bounded Surface Entity represents a limited surface whose boundaries are Boundary Entities. It defines the surface to be trimmed, $\mathbf{S}$, and any number of closed boundary curves, $\mathbf{C}_n(t)$. The bounded surface is the portion of $\mathbf{S}(u, v)$ that lies to the left (i.e. in the direction of the cross-product of the surface normal and a tangent to the boundary curve, $d\mathbf{C}_n/dt$) of all its boundaries. The bounded surface is required to have finite area. As well, it must be possible to connect any two points on the bounded surface by a path lying in the bounded surface. For more information on the restrictions of bounded surfaces see Reference 1, Section 4.31.

Note that, unlike the Trimmed (Parametric) Surface Entity, the direction of the boundary curves is important as it defines which portion of the surface is retained. Bounded Surface Entity also differs from Trimmed (Parametric) Surface Entity in that the entity specifying the surface is allowed to be a Plane Entity and the surface is allowed to have coordinate singularities at the limits of its domain.

Each of the boundary curves must be specified using a Boundary Entity (represented by the class `BoundaryEntity`): see Section 6.3.8.2.

A Bounded Surface Entity has Entity Type Number 143.

Bounded Surface entities are represented by the class `BoundedSurfaceEntity` which is derived from `LimitedSurfaceEntity` and has the following public members in addition to those inherited from its base classes.

`BoundedSurfaceEntity(SurfaceEntity *s)`

Makes a bounded surface entity which uses surface `s`. The transformation matrix is the identity and the surface is independent and visible.

The entity `s` becomes a physically dependent subordinate entity and will inherit the visibility of the bounded surface.

`void add_boundary(BoundaryEntity *bent)`

Adds `bent` to the list of boundary curves. The entity `bent` becomes a physically dependent subordinate entity and will inherit the visibility of the bounded surface.

6.5.3 Bounded Plane Entity

A bounded plane is represented in IGES by a Plane Entity (Entity Type Number 108) with Form Number equal to 1 (the interior of the bounding curve is retained) or -1 (the exterior of the bounding curve is retained). It is a limited surface with a single boundary curve.

A bounded plane is represented by the class `BoundedPlaneEntity` derived from `LimitedSurfaceEntity`. The plane itself is specified and parameterized in the same manner as the plane in a `PlaneEntity`: see Section 6.4.1. `BoundedPlaneEntity` has the following public members in addition to those it inherits from its base classes:

`BoundedPlaneEntity()`

Makes a plane through the origin with normal along the z axis. The display symbol has size 1 and is placed at the origin. The bounding curve must be set by a call to `set_bounding_curve()`.

`BoundedPlaneEntity(Float a, Float b, Float c, Float d)`

Makes a bounded plane defined by $ax + by + cz = d$. The display symbol has size 1 and is placed at the origin. The bounding curve must be set by a call to `set_bounding_curve()`.

```
BoundedPlaneEntity(Float a, Float b, Float c, Float d,
                   CurveEntity *c, bool interior)
```

Makes a bounded plane defined by $ax + by + cz = d$ and bounded by the curve defined by `c`. If `interior` is true, the interior of `c` is the portion of the plane that is retained; otherwise the exterior of `c` is retained. The display symbol has size 1 and is placed at the origin in parameter space: i.e. at $\mathbf{P}(0, 0)$.

The entity `c` becomes a physically dependent subordinate entity and will inherit the visibility of the bounded plane.

`void set_equation(Float a, Float b, Float c, Float d)`
Defines the plane in the same manner as the constructor with the same arguments.

`void get_equation(Float &a, Float &b, Float &c, Float &d) const`
Gets the coefficients of the equation defining the plane.

`void set_bounding_curve(CurveEntity *c, bool interior)`
Makes the plane bounded by `c`. If `interior` is true, the interior of the `c` is the portion of the plane that is retained; otherwise the exterior of `c` is retained. The entity `c` becomes a physically dependent subordinate entity and will inherit the visibility of the bounded plane.

`void set_symbol_location(Float x, Float y, Float z)`
Sets the location for the symbol marking the plane.

`void set_symbol_size(Float s)`
Sets the size for the symbol marking the plane.

`Point get_normal() const`
Returns a unit normal to the plane.

6.6 Entity predicates

The Standard Template Library (STL)[7] concept of a Predicate is a unary function which returns a boolean value expressing the truth or falsehood of some condition. An entity predicate, represented by the abstract class `EntityPredicate`, is a model of an STL Predicate which takes a `const Entity` pointer as its argument and evaluates some condition of the entity. Classes derived from `EntityPredicate` evaluate different conditions. Entity predicates are used by entity collections (see Section 6.7) to restrict lists of entities.

The boolean operators `&&`, `||` and `!` are defined for entity predicates: i.e. if `p1` and `p2` are both instances of classes derived from `EntityPredicate`, then:

- `p1 && p2` is an `EntityPredicate` that returns true if and only if both `p1` and `p2` return true;
- `p1 || p2` is an `EntityPredicate` that returns true if either `p1` or `p2` return true; false otherwise; and
- `!p1` is an `EntityPredicate` that returns true if and only if `p1` returns false.

Entity predicates having the following constructors are defined in the namespace `IGES`:

`TruePredicate()`
: Always returns true.

`FalsePredicate()`
: Always returns false.

`AndPredicate(const EntityPredicate &p1, const EntityPredicate &p2)`
: Returns true if and only if `p1` and `p2` return true. This predicate is used to implement the operator `&&`.

`OrPredicate(const EntityPredicate &p1, const EntityPredicate &p2)`
: Returns true if either `p1` or `p2` return true; otherwise returns false. This predicate is used to implement the operator `||`.

`NotPredicate(const EntityPredicate &p)`
: Returns true if and only if `p1` returns false. This predicate is used to implement the operator `!`.

`IsVisible()`
: Returns true if the entity is visible.

`IsPhysicallyDependent()`
: Returns true if the entity is physically dependent.

`HasTypeNumber(int n)`
: Returns true if the Entity Type Number is `n`.

`IsPoint()`
: Returns true if the entity is a Point Entity. It is safe to cast the entity to a `PointEntity`.

`IsCurve()`
: Returns true if the entity is a curve entity. It is safe to cast the entity to a `CurveEntity`.

`IsSurface()`
: Returns true if the entity is a surface entity. It is safe to cast the entity to a `SurfaceEntity`.

`IsLimitedSurface()`
: Returns true if the entity is a limited surface entity. It is safe to cast the entity to a `LimitedSurfaceEntity`.

`HasLabel(const Str &label)`
: Returns true if the entity has label equal to `label`.

`HasUse(UseFlag use)`
: Returns true if the entity's Entity Use Flag is equal to `use`.

6.7 Entity collections

The class `EntityCollection` is a base class for classes that contain a collection of pointers to entities. It has the following public members.

`typedef std::list<const Entity*> EntityList`
: The type of a list of entities of any type.

`typedef std::list<const PointEntity*> PointEntityList`
: The type of a list of Point entities.

`typedef std::list<const CurveEntity*> CurveEntityList`
: The type of a list of curve entities.

`typedef std::list<const SurfaceEntity*> SurfaceEntityList`
: The type of a list of surface entities.

`typedef std::list<const LimitedSurfaceEntity*>`
`LimitedSurfaceEntityList`
: The type of a list of limited surface entities.

`typedef std::list<Point> PointList`
: The type of a list of points.

`typedef std::list<CurveType> CurveList`
: The type of a list of curves.

`typedef std::list<SurfaceType> SurfaceList`
: The type of a list of surfaces.

`typedef std::list<LimitedSurface> LimitedSurfaceList`
: The type of a list of limited surfaces.

`EntityCollection()`
: Creates an empty collection of entities.

`unsigned size() const`
: Returns the number of entities in the collection.

`void clear()`
: Empties the collection of all entities.

`void add(const Entity *e)`
: Add an entity to the collection.

`void remove(const Entity *e)`
: Removes an entity from the collection. A `ProgError` is thrown if the entity is not in the collection.

```
const EntityList& get_entity_list() const
```
Returns a list of pointers to all the entities in the collection.

```
EntityCollection get_entities(const EntityPredicate &pred) const
```
Returns an `EntityCollection` which contains only those entities which satisfy the entity predicate `pred`: i.e. if `e` is a `const` entity pointer in the current collection for which `pred(e)` is true, then `e` will be in the `EntityCollection` which is returned.

```
template<class Ent>
list<const Ent*> get_entities_of_type(
                const EntityPredicate &pred = TruePredicate()) const
```
Returns a list of all the entities of type `Ent` for which `pred` is true. This function is defined only if the compiler supports template member functions.

```
PointEntityList get_point_entity_list() const
```
Returns a list of pointers to all the Point Entities.

```
virtual PointList get_transformed_point_list(
                            const EntityPredicate &pred, bool nested,
                            bool only_if_independent = false) const
```
Returns a list of the points in the entity collection, each transformed into model space by its combined transformation. Only points defined by Point Entities satisfying the entity predicate `pred` will be included.

If `nested` is true, points nested within entities in the collection will also be included in the list provided that they satisfy `pred`. Note that children of parent entities that do not satisfy `pred` are also considered; thus a visible child of an invisible parent can be in the list when `pred` is `IsVisible`.

If `only_if_independent` is true, then points at the top level are included only if they are physically independent; this flag does not affect the inclusion of nested points (e.g. in a subfigure) as they are always physically independent. Typically `only_if_independent` will be true if the entity collection is a `Model`, false otherwise.

```
CurveEntityList get_curve_entity_list() const
```
Returns a list of pointers to all the curve entities.

```
virtual CurveList get_transformed_curve_list(
                            const EntityPredicate &pred, bool nested,
                            bool only_if_independent = false) const
```
Similar to `get_transformed_point_list` but returns a list of curves, each of type `CurveType`: see Section 3.

```
SurfaceEntityList get_surface_entity_list() const
```
Returns a list of pointers to all the surface entities.

```
virtual SurfaceList get_transformed_surface_list(
                          const EntityPredicate &pred, bool nested,
                          bool only_if_independent = false) const
```

Similar to `get_transformed_point_list` but returns a list of surfaces, each of type `SurfaceType`: see Section 3.

```
SurfaceEntityList get_limited_surface_entity_list() const
```

Returns a list of pointers to all the limited surface entities.

```
virtual SurfaceList get_transformed_limited_surface_list(
                          const EntityPredicate &pred, bool nested,
                          bool only_if_independent = false) const
```

Similar to `get_transformed_point_list` but returns a list of limited surfaces, each of type `LimitedSurfaceType`: see Section 3.

For example, if `ecoll` is an `EntityCollection`, then you can obtain a list of all the visible Composite Curve Entities using:

```
std::list<const CompositeCurveEntity*> comp_curves =
  ecoll.get_entities_of_type<CompositeCurveEntity>(IsVisible());
```

To get a list of all visible curves with label `"Blade"`, whether at the top level or nested inside other entities, each being transformed into model space, use:

```
CurveList curves = get_transformed_curve_list(
                     IsVisible() && HasLabel("Blade"),true);
```

6.8 Subfigure Definition Entity

An IGES Subfigure Definition Entity supports the concept of an aggregation of entities that can be used more than once. It contains a set of pointers to the IGES entities that comprise the aggregation. It has Entity Type Number 308.

Each subfigure definition also has a name: a character string of arbitrary length. Note that this is independent of the label for the entity.

Subfigure definition entities are represented by the class `SubFigDefEntity` which is derived from both `Entity` and `EntityCollection`. It has the following public members in addition to those inherited from its base classes.

```
SubFigDefEntity(const Str &n)
```

Makes a subfigure definition with name `n`. The list of subfigure entities is empty.

```
const Str& get_name() const
```

Returns the name of the subfigure.

`unsigned get_depth() const`
: Returns the depth of the subfigure. The depth is an indication of the level of nesting within the subfigure; a depth of n indicates that the subfigure refers to a subfigure instance whose definition has depth $n-1$. Note that the depth may change if a subfigure instance is added to the list of entities.

`void add(const Entity *e)`
: Add entity `e` to the list of entities. The entity `e` is made physically dependent but is not considered a subordinate entity. Instead it will become a child of any instantiations of the subfigure definition: see Section 6.9.

`SubFigDefEntity` redefines the member function `get_transformed_point_list()` to return points which have been transformed into the model space of the Subfigure Definition: that is, the combined transformation of the Subfigure Definition has been applied to each of the points. Similarly `get_transformed_curve_list()` returns curves and `get_transformed_surface_list()` returns surfaces which have been transformed into the model space of the Subfigure Definition.

6.9 Singular Subfigure Instance Entity

An IGES Subfigure Instance Entity is a single instance of a subfigure defined by a Subfigure Definition Entity. It has Entity Type Number 308.

Subfigure instance entities are represented by the class `SubFigInstEntity` which is derived from `Entity` and has the following public members in addition to those inherited from its base classes.

`typedef SubFigDefEntity::PointList PointList`
: The type of a list of points in the sub-figure.

`typedef SubFigDefEntity::CurveList CurveList`
: The type of a list of curves in the sub-figure.

`typedef SubFigDefEntity::SurfaceList SurfaceList`
: The type of a list of surfaces in the sub-figure.

`typedef SubFigDefEntity::LimitedSurfaceList LimitedSurfaceList`
: The type of a list of limited surfaces in the sub-figure.

`SingSubFigInstEntity(const SubFigDefEntity *sf)`
: Makes a singular subfigure instance entity which uses Subfigure Definition Entity `sf`.

`SingSubFigInstEntity(const SubFigDefEntity *sf, const Point &x, Float s)`

Makes a singular subfigure instance entity which uses Subfigure Definition Entity `sf` translated by `x` and scaled by `s`.

`const SubFigDefEntity* get_subfigure_definition() const`

Returns the Subfigure Definition Entity.

`const TransVec& get_translation() const`

Returns the translation of the subfigure instance.

`void set_translation(const TransVec &x)`

Sets the translation of the subfigure.

`Float get_scale() const`

Returns the scaling factor of the subfigure instance.

`void set_scale(Float s)`

Sets the scaling factor of the subfigure.

`virtual Transformation get_transformation() const`

Returns the transformation to be applied to convert from the model space of the subfigure definition to the model space of the subfigure instance. This is a combination of the Transformation Matrix Entity associated with the subfigure instance and the translation and scale defined by the subfigure instance.

`PointList get_transformed_point_list(const EntityPredicate &pred, bool nested) const`

Returns a list of all the points in the subfigure instance, each transformed by its own transformation matrix, the transformation matrix of the subfigure definition, the transformation of the subfigure instance and its scaling factor and translation. Only points defined by entities which satisfy `pred` are included in the list.

If `nested` is true, points nested within entities in the subfigure will also be included in the list provided that they satisfy `pred`. Note that children of parent entities that do not satisfy `pred` are also considered; thus a visible child of an invisible parent can be in the list when `pred` is `IsVisible`.

`CurveList get_transformed_curve_list(const EntityPredicate &pred, bool nested) const`

Similar to `get_transformed_point_list()` but returns a list of curves in the subfigure instance.

`SurfaceList get_transformed_surface_list(const EntityPredicate &pred, bool nested) const`

Similar to `get_transformed_point_list()` but returns a list of surfaces in the subfigure instance.

```
LimitedSurfaceList get_transformed_limited_surface_list(
                                          const EntityPredicate &pred,
                                          bool nested) const
```

Similar to `get_transformed_point_list()` but returns a list of limited surfaces in the subfigure instance.

The subordinate entities of a Singular Subfigure Instance Entity are all the entities contained by its Subfigure Definition Entity. The Subfigure Definition Entity itself is not a child of the Singular Subfigure Instance as it does not represent instantiated geometry.

6.10 Null Entity

IGES defines a Null Entity as a convenient means for editing an IGES file so that an entity is ignored. The Entity Type Number of a Null Entity is zero. If the Entity Type Number of an entity in an IGES file is changed to zero in both its Directory Entry and its Parameter Data, then the entity can be ignored.

The IGES classes do not provide a representation of a Null Entity as a class; therefore there is no way to include a Null Entity as an entity in a `Model`. However, if a Null Entity is encountered when an IGES file is being read by a `Model`, it will be ignored as required by the specification.

7 Units

Each IGES file must use a consistent set of units. The units are stored both as an integer flag and as a character string.

The units used are represented by the class `Units`. It has the following public members.

```
enum Type { inches, millimetres, unknown, feet, miles, metres,
            kilometres, mils, microns, centimetres, microinches }
```

The allowed types for the units. If the type is `unknown`, the string version of the units is used as the specifier.

```
Units(Type f = metres)
```

Creates units with flag `f`. `f` should not be of type `unknown`.

```
Units(const Str &s)
```

Creates units using `s` as a specifier. If `s` is recognized as one of the standard

units, the unit flag will be set accordingly. Otherwise the unit flag will be `unknown`. The character string version of the units is set to `s`.

The following case-independent strings are recognized for each of the possible units:

`inches`
: `"inch"`, `"inches"`, `"in"`, `"in."`.

`millimetres`
: `"millimetre"`, `"millimetres"`, `"millimeter"`, `"millimeters"`, `"mm"`, `"mm."`.

`feet`
: `"feet"`, `"ft"`, `"ft."`.

`miles`
: `"mile"`, `"miles"`, `"mi"`, `"mi."`.

`metres`
: `"metre"`, `"metres"`, `"meter"`, `"meters"`, `"m"`, `"m."`.

`kilometres`
: `"kilometre"`, `"kilometres"`, `"kilometer"`, `"kilometers"`, `"km"`, `"km."`.

`mils`
: `"mils"`, `"mil"`.

`microns`
: `"micron"`, `"microns"`, `"micrometre"`, `"micrometres"`, `"micrometer"`, `"micrometers"`, `"um"`, `"um."`.

`centimetres`
: `"centimetre"`, `"centimetres"`, `"centimeter"`, `"centimeters"`, `"cm"`, `"cm."`.

`microinches`
: `"microinch"`, `"microinches"`, `"uin"`, `"uin."`.

`Type unit_flag() const`
: Returns the unit flag.

`const Str& string() const`
: Returns the units as a normal string.

`Str hollerith() const`
: Returns the units as a hollerith string.

8 The IGES model

An IGES model is represented by the class `Model` which is derived from the base class `EntityCollection`. It has the following public members in addition to those inherited from the base class.

`Str header`
: The contents of the start section of the IGES file. The start section can be used to write comments describing the contents of the file. Empty by default.

`Str pident_send`
: The product identification from the sending system. Empty by default.

`Str pident_rec`
: The product identification from the receiving system. Empty by default.

`Str system_id`
: An identification code which should identify the software that generated the file. The default value for this string is `DRDC C++ IGES classes`.

`Str preprocessor`
: The version or release date of the pre-processor that created the file. Empty by default.

`double model_scale`
: The ratio of model space to real world space. The default value is 1.

`Units units`
: The units used: see Section 7.

`int max_num_line_weights`
: The maximum number of line weight gradations. The default value is 1.

`double max_line_width`
: The actual size of the thickest line possible in the (scaled) file. The default value is 0.

`double min_resolution`
: The smallest distance in model space units that should be considered as discernible. The default value is 10^{-6}.

`double max_coord`
: The approximate maximum coordinate value occurring in the model expressed in `units`. The default value is 0 which, in accordance with IGES, means that the maximum coordinate is unspecified.

`Str author`
: The name of the person responsible for creating the data in the file. Empty by default.

`Str organization`
: The author's organization. The default value is `DRDC Atlantic`.

`Str protocol`
: A descriptor indicating the Mil-specification, application protocol, application subset, or user-defined protocol or subset, if any.

`Model()`
: Creates a `Model` containing no entities.

`const Str& get_time_stamp() const`
: Returns the time stamp for the file in the format `YYYYMMDD.HHNNSS`. The time stamp is not defined until the model is either read from a file or written to one. If it is not defined, an empty string is returned.

`const Str& get_modification_time_stamp() const`
: Similar to `get_time_stamp()` but returns the time at which the file was last modified.

`Str get_standard() const`
: Returns a string describing the drafting standard in compliance to which the data encoded in the file was generated. The string may be empty (no standard specified).

`void clear()`
: Empties the model of all entities.

`void add(Entity *e)`
: Add an entity to the model. The model takes over responsibility for deleting the entity. If `e` has subordinate entities, they will also be added to the model.

`void write(const Str &filename)`
: Writes the model to file `filename` in IGES format. Floating point numbers are written with eight significant figures.

`void read(const Str &filename)`
: Defines the model by reading data from file `filename` in IGES format. Any IGES entities currently defined are discarded.

If an entity `e` has subordinate entities they will be added to the model when `e` is added. However, if the subordinate entities are not defined until after `e` is added, then the subordinate entities must be added to the model explicitly. For example, consider the following code:

```
using namespace IGES;
LineEntity *line1 = new LineEntity(Point(0,0,0),Point(1,0,0));
LineEntity *line2 = new LineEntity(Point(1,0,0),Point(1,1,0));
CompositeCurveEntity *ccrv = new CompositeCurveEntity;
ccrv->add(line1);
ccrv->add(line2);

Model model;
model.add(ccrv);
```

There is no need to add `line1` and `line2` to `model` explicitly since they are added when `ccrv` is added. However, if `line1` and `line2` are not added to `ccrv` until after `ccrv` is added to `model`, then `line1` and `line2` must be added explicitly:

```
using namespace IGES;
CompositeCurveEntity *ccrv = new CompositeCurveEntity;

Model model;
model.add(ccrv);

LineEntity *line1 = new LineEntity(Point(0,0,0),Point(1,0,0));
LineEntity *line2 = new LineEntity(Point(1,0,0),Point(1,1,0));
ccrv->add(line1);
ccrv->add(line2);

model.add(line1); // Need to add line1 and line2 explicitly
model.add(line2)
```

9 Examples

This section provides some brief examples of the use of the class `Model` for reading and writing IGES files.

9.1 Writing to an IGES file

We present three different examples of creating a `Model` to represent geometric objects, then writing it to an IGES file.

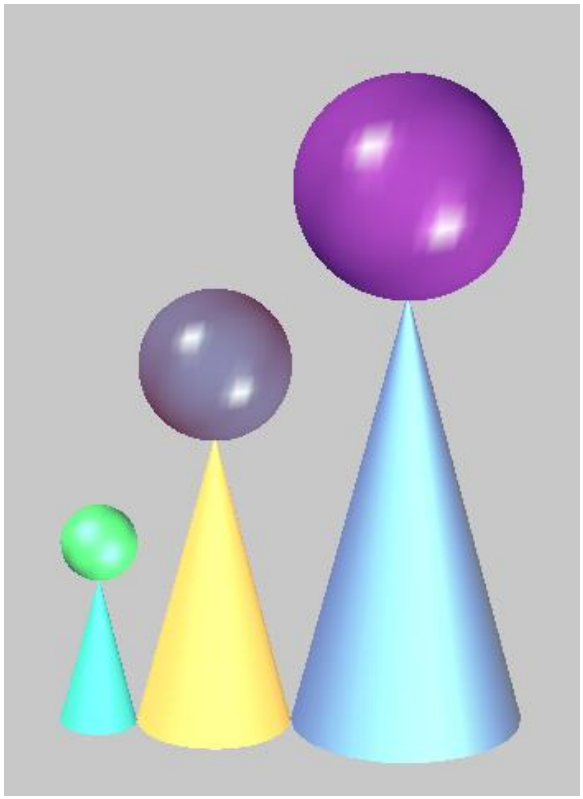

Figure 2: *A display of the geometry defined in* `test.igs`

9.1.1 Example 1: Cones and spheres

The following program defines a model consisting of three cones and three spheres located side by side: see Figure 2. The model is written to the file `test.igs`. This example illustrates the use of sub-figures to replicate similar geometric objects.

```
#include "IGES.h"
#include "FormattedWarning.h"
using namespace IGES;

int main()
{
  FormattedOStream out(std::cout);
  FormattedWarningHandler whandler(out);
  set_warning_handler(whandler);
  try {
    // Make a model
    Model model;
    model.header = "Example IGES file.";
```

```
// Make a sphere as a surface of revolution with an arc as the
// generatrix.
Point centre(0.0,0.0,0.0);
LineEntity *axis = new LineEntity(Point(1.0,0.0,0.0),centre);
axis->make_invisible();

Angle<Float> zero(0.0), pi = Angle<Float>::straight_angle();
CircularArcEntity *arc =
  new CircularArcEntity(centre,1.0,zero,pi);
arc->make_invisible();
RevSurfaceEntity *sphere = new RevSurfaceEntity(axis,arc);
sphere->set_label("Sphere");

// Move the sphere to (0,0,5)
TransMatrixEntity *tmtx = new TransMatrixEntity;
tmtx->set_translation(TransVec(0.0,0.0,5.0));
sphere->set_transformation_entity(tmtx);

// Define a cone as a surface of revolution with a line as
// generatrix.
Point tip(0.0,0.0,4.0);
axis = new LineEntity(centre,tip);
axis->make_invisible();

LineEntity *line = new LineEntity(Point(1.0,0.0,0.0),tip);
line->make_invisible();
RevSurfaceEntity *cone = new RevSurfaceEntity(axis,line);
cone->set_label("Cone");

// Define a subfigure from the sphere and cone
SubFigDefEntity *subfig_def = new SubFigDefEntity("Subfigure");
subfig_def->add(sphere);
subfig_def->add(cone);

// Define three subfigure instances
SingSubFigInstEntity
  *sf_inst1 = new SingSubFigInstEntity(subfig_def),
  *sf_inst2 = new SingSubFigInstEntity(subfig_def,
                                       TransVec(3.0,0.0,0.0),2.0),
  *sf_inst3 = new SingSubFigInstEntity(subfig_def,
                                       TransVec(8.0,0.0,0.0),3.0);

// Add the subfigure instances to the model.  This will also add
// all the other subordinate entities.
model.add(sf_inst1);
model.add(sf_inst2);
model.add(sf_inst3);
```

```
    // Now write the model to the file test.igs
    model.write("test.igs");
  }
  catch(Error &e) {
    out << e.get_msg() << '\n';
    return 1;
  }
  return 0;
}
```

The file `test.igs` is shown in Figure 3.

9.1.2 Example 2: A trimmed surface

This example illustrates the use of the Trimmed (Parametric) Surface Entity to trim a surface. In this case the geometric object consists of a square in the xy-plane, centred on the origin, having side of length 4.0. Two circles are cut from the square: one centred at $(2.8, 2.8, 0.0)$ with radius 0.8, the other centred at $(-0.8, -0.8, 0.0)$ and having radius 1.0. A display of the trimmed square is shown in Figure 4.

The square is generated as a `RuledSurfaceEntity`. A `TrimmedSurfaceEntity` is created to represent the trimmed square. It has two inner boundaries, each represented using a `CurveOnSurfaceEntity`. A parametric representation of each the inner boundaries must also be generated. Since the parameter space of the ruled surface is the unit square, the parameter curves for the two circles are also circles, one centred at $(0.7, 0.7)$ with radius 0.2 and the other centred at $(0.3, 0.3)$ with radius 0.25.

The following program defines the model and writes it to the file `tsquare.igs` shown in Figure 5.

```
#include "IGES.h"
#include "FormattedWarning.h"
using namespace IGES;

int main()
{
  FormattedOStream out(std::cout);
  FormattedWarningHandler whandler(out);
  set_warning_handler(whandler);
  try {
    // Make a model
    Model model;
    model.header = "Example IGES file.";
```

```
Example IGES file.                                                      S      1
1H,,1H;,                                                                G      1
,                                                                       G      2
8Htest.igs,                                                             G      3
21HDRDC C++ IGES classes,,32,38,6,308,15,,1.0000000e+00,6,1HM,1,,       G      4
15H20061024.104834,1.0000000e-05,1.0000000e+00,,13HDRDC Atlantic,11,0,  G      5
15H20061201.134008,;                                                    G      6
     110       1                                              01010000D      1
     110                      2                                      D      2
     100       3                                              01010000D      3
     100                      2                                      D      4
     124       5                                                     D      5
     124                      4                                      D      6
     120       9                                    5         00010000D      7
     120                      1                       Sphere        D      8
     110      10                                              01010000D      9
     110                      2                                      D     10
     110      12                                              01010000D     11
     110                      2                                      D     12
     120      14                                              00010000D     13
     120                      1                         Cone         D     14
     308      15                                              00000200D     15
     308                      2                                      D     16
     408      17                                                     D     17
     408                      2                                      D     18
     408      19                                                     D     19
     408                      2                                      D     20
     408      21                                                     D     21
     408                      2                                      D     22
110,1.0000000e+00,0.0000000e+00,0.0000000e+00,                        1P      1
0.0000000e+00,0.0000000e+00,0.0000000e+00;                            1P      2
100,0.0000000e+00,0.0000000e+00,0.0000000e+00,                        3P      3
1.0000000e+00,0.0000000e+00,-1.0000000e+00,1.2246468e-16;             3P      4
124,                                                                  5P      5
1.0000000e+00,0.0000000e+00,0.0000000e+00,0.0000000e+00,              5P      6
0.0000000e+00,1.0000000e+00,0.0000000e+00,0.0000000e+00,              5P      7
0.0000000e+00,0.0000000e+00,1.0000000e+00,5.0000000e+00;              5P      8
120,1,3,0.0000000e+00,6.2831853e+00;                                  7P      9
110,0.0000000e+00,0.0000000e+00,0.0000000e+00,                        9P     10
0.0000000e+00,0.0000000e+00,4.0000000e+00;                            9P     11
110,1.0000000e+00,0.0000000e+00,0.0000000e+00,                       11P     12
0.0000000e+00,0.0000000e+00,4.0000000e+00;                           11P     13
120,9,11,0.0000000e+00,6.2831853e+00;                                13P     14
308,0,9HSubfigure,2,                                                 15P     15
7,13;                                                                15P     16
408,15,                                                              17P     17
0.0000000e+00,0.0000000e+00,0.0000000e+00,1.0000000e+00;             17P     18
408,15,                                                              19P     19
3.0000000e+00,0.0000000e+00,0.0000000e+00,2.0000000e+00;             19P     20
408,15,                                                              21P     21
8.0000000e+00,0.0000000e+00,0.0000000e+00,3.0000000e+00;             21P     22
S      1G      TD     22P     22                                      T      1
```

Figure 3: *The file* `test.igs`

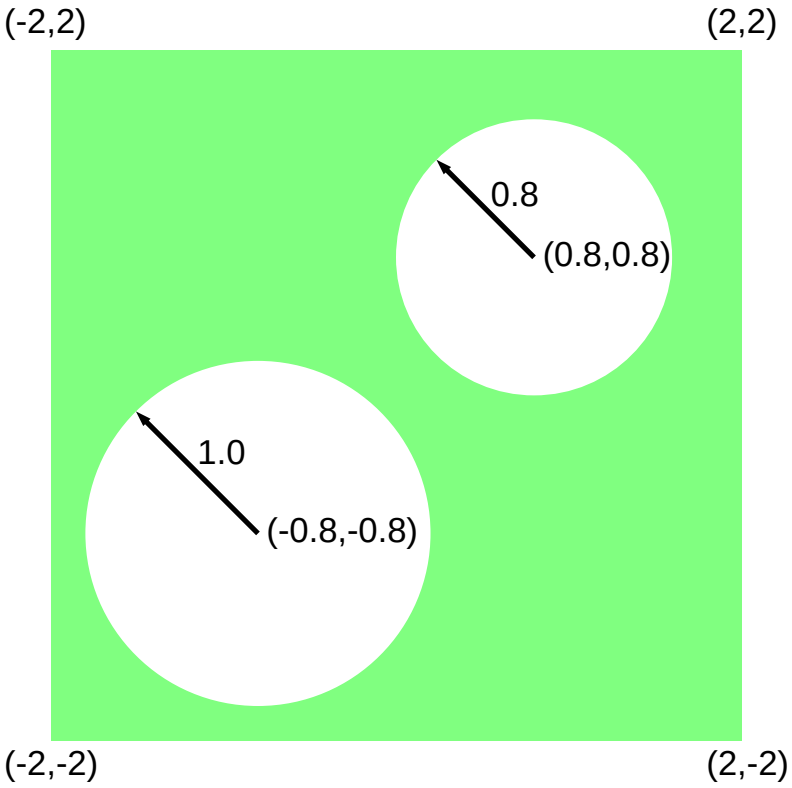

Figure 4: *A display of the geometry defined in* `tsquare.igs`

```
//=================================================
// Make a square surface

LineEntity *line1 = new LineEntity(Point(-2.0,-2.0,0.0),
   Point( 2.0,-2.0,0.0));
LineEntity *line2 = new LineEntity(Point(-2.0, 2.0,0.0),
   Point( 2.0, 2.0,0.0));
RuledSurfaceEntity *rsrf = new RuledSurfaceEntity(line1,line2);
rsrf->make_invisible();
rsrf->set_label("Square");

//=================================================
// Define the circles in model and parameter space.

Point pcentre1(0.7,0.7,0.0), centre1(0.8,0.8,0.0);
CircularArcEntity *pcirc1 = new CircularArcEntity(pcentre1,0.2);
CircularArcEntity *circ1 = new CircularArcEntity(centre1,0.8);
circ1->make_invisible();
```

```
    CurveOnSurfaceEntity *cinner1 =
      new CurveOnSurfaceEntity(rsrf,pcirc1,circ1);
    cinner1->make_invisible();

    Point pcentre2(0.3,0.3,0.0), centre2(-0.8,-0.8,0.0);
    CircularArcEntity *pcirc2 = new CircularArcEntity(pcentre2,0.25);
    CircularArcEntity *circ2 = new CircularArcEntity(centre2,1.0);
    circ2->make_invisible();

    CurveOnSurfaceEntity *cinner2 =
      new CurveOnSurfaceEntity(rsrf,pcirc2,circ2);
    cinner2->make_invisible();

    //================================================
    // Trim the square with the circles

    TrimmedSurfaceEntity *trim = new TrimmedSurfaceEntity(rsrf);
    trim->add_inner_curve_entity(cinner1);
    trim->add_inner_curve_entity(cinner2);
    trim->set_label("TSquare");

    model.add(trim);

    // Write the model to a file.
    model.write("trim.igs");
  }
  catch(Error &e) {
    out << e.get_msg() << '\n';
    return 1;
  }
  return 0;
}
```

We could also have generated the trimmed square using a Bounded Surface Entity. In that case we would have to be careful when using a `CircularArcEntity` to define a boundary curve. A Circular Arc Entity always traverses the circle counter-clockwise around $\hat{z}$; therefore, if used as a parametric curve for a Boundary Entity, a Circular Arc Entity will normally pick the interior of the circle as the active area. To select the outside of the circle the direction of traversal must be reversed.

This can be done by applying a transformation to the Circular Arc Entity, rotating through 180° about the x-axis, then translating it so that its centre remains the same. However, after this is done the orientation of the model space curve is reversed relative to the composed curve, so we must show this when defining the segments for the boundaries.

```
Example IGES file.                                                      S      1
1H,,1H;,                                                                G      1
,                                                                       G      2
8Htrim.igs,                                                             G      3
21HDRDC C++ IGES classes,,32,38,6,308,15,,1.0000000e+00,6,1HM,1,,       G      4
15H20061201.143600,1.0000000e-05,1.0000000e+00,,13HDRDC Atlantic,11,0,  G      5
15H20061201.143600,;                                                    G      6
     110       1                                              01010000D      1
     110                       2                                    D      2
     110       3                                              01010000D      3
     110                       2                                    D      4
     118       5                                              01010000D      5
     118                       1       0                  Square    D      6
     100       6                                              01010500D      7
     100                       2                                    D      8
     100       8                                              01010000D      9
     100                       2                                    D     10
     142      10                                              01010000D     11
     142                       1                                    D     12
     100      11                                              01010500D     13
     100                       2                                    D     14
     100      13                                              01010000D     15
     100                       2                                    D     16
     142      15                                              01010000D     17
     142                       1                                    D     18
     144      16                                                   D     19
     144                       2                          TSquare   D     20
110,-2.0000000e+00,-2.0000000e+00,0.0000000e+00,                       1P      1
2.0000000e+00,-2.0000000e+00,0.0000000e+00;                            1P      2
110,-2.0000000e+00,2.0000000e+00,0.0000000e+00,                        3P      3
2.0000000e+00,2.0000000e+00,0.0000000e+00;                             3P      4
118,1,3,0,0;                                                           5P      5
100,0.0000000e+00,7.0000000e-01,7.0000000e-01,                         7P      6
9.0000000e-01,7.0000000e-01,9.0000000e-01,7.0000000e-01;               7P      7
100,0.0000000e+00,8.0000000e-01,8.0000000e-01,                         9P      8
1.6000000e+00,8.0000000e-01,1.6000000e+00,8.0000000e-01;               9P      9
142,0,5,7,9,0;                                                        11P     10
100,0.0000000e+00,3.0000000e-01,3.0000000e-01,                        13P     11
5.5000000e-01,3.0000000e-01,5.5000000e-01,3.0000000e-01;              13P     12
100,0.0000000e+00,-8.0000000e-01,-8.0000000e-01,                      15P     13
2.0000000e-01,-8.0000000e-01,2.0000000e-01,-8.0000000e-01;            15P     14
142,0,5,13,15,0;                                                      17P     15
144,5,0,2,0,                                                          19P     16
11,17;                                                                19P     17
S      1G      6D     20P     17                                       T      1
```

Figure 5: *The file* `tsquare.igs`

The following code fragment shows how to define a Bounded Surface Entity which generates the surface shown in Figure 4.

```
//=================================================
// Make a Bounded Surface Entity to trim the square.

BoundedSurfaceEntity *bsrf = new BoundedSurfaceEntity(rsrf);
bsrf->set_label("Bounded");

//=================================================
// Define a rotation matrix to rotate through 180
// degrees about the x axis.

RotMtx mtx;
identity(mtx);
mtx[1][1] = mtx[2][2] = -1;

//=================================================
// Define the model space and parameter curve for
// the first circle

Point pcentre1(0.7,0.7,0.0), centre1(0.8,0.8,0.0);
TransVec v1(0.0,2.0*pcentre1[1],0.0);
TransMatrixEntity *tmtx1 = new TransMatrixEntity(mtx,v1);

CircularArcEntity *pcirc1 = new CircularArcEntity(pcentre1,0.2);
pcirc1->set_transformation_entity(tmtx1);

CircularArcEntity *circ1 = new CircularArcEntity(centre1,0.8);
circ1->make_invisible();

//=================================================
// Make a Boundary Entity for the first circle
// having a single segment.

BdyCurveSegment bcseg1;
bcseg1.reversed = true;
bcseg1.space_curve_entity = circ1;
bcseg1.param_curves.push_back(pcirc1);

BoundaryEntity *bent1 = new BoundaryEntity(rsrf);
bent1->add_segment(bcseg);

//=================================================
// Add the boundary to the bounded square

bsrf->add_boundary(bent1);

//=================================================
// Make the boundary for the second circle in a
// similar way.
...
```

9.2 Reading from an IGES file

Suppose we wish to display all the visible curves defined in an IGES file call `model.igs` (we could easily add points, surfaces and limited surface but we restrict this example to curves for simplicity). We assume that a function, `display_curve`, is available that will display a curve of type `CurveType`. The following code will do this.

```
try {
  using namespace CurveLib;
  using namespace IGES;

  // Make a model and read the input file model.igs.
  Model model;
  model.read("model.igs");

  // Get a list of all visible curves, each transformed to model
  // space. Ignore physically dependent curves at the top level
  // but include all visible curves nested within other entities.
  CurveList curves =
    model.get_transformed_curve_list(IsVisible(),true,true);

  // Display the curves
  CurveList::const_iterator pcrv = curves.begin();
  while (crv != curves.end()) {
    display_curve(*pcrv);
    ++pcrv;
  }
}
catch (ProgError &pe) {
  // Handle the programming error pe. Write the error message
  // and quit.
  std::cerr << pe.get_msg() << '\n';
  exit(1);
}
catch (Error &e) {
  // Handle the exception e.
  ...
}
```

10 Concluding remarks

This document has described a library of C++ classes that provide an interface between geometry specified using the CurveLib library of C++ classes and geometry specified using the Initial Graphics Exchange Specification. Although the overlap

between these two methods of describing geometry is not perfect, it is large enough that adequate representations of most geometrical objects can be made in a way that can be represented by both methods. Therefore the classes provide a useful mechanism for converting geometry from one method of representation to the other.

The CurveLib classes were originally designed for representing the geometry of ship hulls and propellers use in Computation Fluid Dynamics programs. They provide a means for transferring geometry generated using the CurveLib classes to commercial grid generation and flow computation software.

References

[1] (1988), Initial Graphics Exchange Specification (IGES) Version 4.0, US Dept. of Commerce, National Bureau of Standards. Document No. NBSIR 88-3813.

[2] (1998), The Initial Graphics Exchange Specification (IGES) Version 6.0 (DRAFT) (online), IGES 5.x Preservation Society, http://www.iges5x.org (Access Date: May 2006). This site provides an unofficial copy of the IGES Version 6.0 draft in LaTeX format.

[3] Hally, D. (2006), C++ classes for representing curves and surfaces: Part I: Multi-parameter differentiable functions, (DRDC Atlantic TM 2006-254) Defence R&D Canada – Atlantic.

[4] Hally, D. (1997), TRANSOM: A Multi-method Navier-Stokes Solver: Overall Design, (DREA TM 97-231) Defence R&D Canada – Atlantic.

[5] Hally, D. (2006), C++ classes for representing curves and surfaces: Part II: Splines, (DRDC Atlantic TM 2006-255) Defence R&D Canada – Atlantic.

[6] Hally, D. (2006), C++ classes for representing curves and surfaces: Part IV: Distribution functions, (DRDC Atlantic TM 2006-257) Defence R&D Canada – Atlantic.

[7] Standard Template Library Programmer's Guide (online), Silicon Graphics, Inc., http://www.sgi.com/tech/stl (Access Date: November 2006).

[8] Stroustrup, B. (1997), The C++ Programming Language, 3rd ed, Addison-Wesley Publishing Co.

Annex A: Warnings

The IGES classes make use of a warning message facility that can be used to issue warnings when something is not quite right but some natural remedial action can be taken. In these cases one does not want to interrupt the flow of the program by throwing an exception, but one still wants to inform the user that the remedial action has been taken.

The warning message facility is invoked by including the header file `Warning.h`. It defines the following function which can be used to send a warning message:

```
void warning(const Str &msg)
```

All warning messages are sent to a global warning message handler. It is an instance of a class derived from the base class `WarningHandler` having the following member function:

```
virtual void handle(const Str &msg)
```
Handles the warning `msg`.

The `handle` function for the base class `WarningHandler` simply writes the message to `std::cerr`.

The global warning handler can be replaced using the function:

```
WarningHandler& set_warning_handler(WarningHandler &wh);
```
Sets the global warning handler to `wh`. Returns the previous warning handler.

In addition, `Warning.h` defines the sentry class `WarningSentry`. It changes the global warning handler when the sentry is constructed and restores it to its previous value when the sentry is destroyed. The sentry ensures that the warning handler is returned to its previous state even if an exception is thrown (see Stroustrup[8, Section 21.3.8]). `WarningSentry` has the following member functions:

```
WarningSentry(WarningHandler &wh)
```
Make a format sentry for `s` using the current format flags.

```
~WarningSentry()
```
Resets the global warning handler to its original state.

`Warning.h` also defines the class `IgnoreWarningHandler`, a warning handler that simply ignores all warning messages.

The header file `FormattedWarning.h` defines the class `FormattedWarningHandler`, derived from `WarningHandler`, which caters for warning messages which are very long.

A `FormattedWarningHandler` writes the warning message to a `FormattedOStream`, an output stream which breaks the output into lines having a given indentation and a maximum length. To use a global warning handler that sends the warning messages to `std::cout` which each line indented by two spaces and no line longer than 60 characters, use the following:

```
FormattedOStream out(std::cout);
out.indent(2);
out.num_columns(60);
FormattedWarningHandler whandler(out);
set_warning_handler(whandler);
```

Index

This page intentionally left blank.

Distribution list

DRDC Atlantic TM-2006-256

Internal distribution

1 Author

5 Library

Total internal copies: 6

External distribution

Department of National Defence

1 DRDKIM

2 DMSS 2

Others

2 Canadian Acquisitions Division
National Library of Canada
395 Wellington Street
Ottawa, Ontario
K1A ON4
Attn: Government Documents

1 Director-General
Institute for Marine Dynamics
National Research Council of Canada
P.O. Box 12093, Station A
St. John's, Newfoundland
A1B 3T5

1 Director-General
Institute for Aerospace Research
National Research Council of Canada
Building M-13A
Ottawa, Ontario
K1A OR6

1 Transport Development Centre
Transport Canada
6th Floor
800 Rene Levesque Blvd, West
Montreal, Que.
H3B 1X9
Attn: Marine R&D Coordinator

1 Canadian Coast Guard
Ship Safety Branch
Canada Building, 11th Floor
344 Slater Street
Ottawa, Ontario
K1A 0N7
Att: Chief, Design and Construction

MOUs

6 Canadian Project Officer ABCA-02-01 (C/SCI, DRDC Atlantic – 3 paper copies, 3 PDF files on CDROM)

Total external copies: 15

Total copies: 21

DOCUMENT CONTROL DATA

(Security classification of title, body of abstract and indexing annotation must be entered when document is classified)

1. ORIGINATOR (The name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g. Centre sponsoring a contractor's report, or tasking agency, are entered in section 8.)

   Defence R&D Canada – Atlantic
   PO Box 1012, Dartmouth NS B2Y 3Z7, Canada

2. SECURITY CLASSIFICATION (Overall security classification of the document including special warning terms if applicable.)

   UNCLASSIFIED

3. TITLE (The complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S, C or U) in parentheses after the title.)

   C++ classes for representing curves and surfaces: Part III: Reading and writing in IGES format

4. AUTHORS (Last name, followed by initials – ranks, titles, etc. not to be used.)

   Hally, D.

5. DATE OF PUBLICATION (Month and year of publication of document.)

   January 2007

6a. NO. OF PAGES (Total containing information. Include Annexes, Appendices, etc.)

   90

6b. NO. OF REFS (Total cited in document.)

   8

7. DESCRIPTIVE NOTES (The category of the document, e.g. technical report, technical note or memorandum. If appropriate, enter the type of report, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered.)

   Technical Memorandum

8. SPONSORING ACTIVITY (The name of the department project office or laboratory sponsoring the research and development – include address.)

   Defence R&D Canada – Atlantic
   PO Box 1012, Dartmouth NS B2Y 3Z7, Canada

9a. PROJECT NO. (The applicable research and development project number under which the document was written. Please specify whether project or grant.)

   11cj18

9b. GRANT OR CONTRACT NO. (If appropriate, the applicable number under which the document was written.)

10a. ORIGINATOR'S DOCUMENT NUMBER (The official document number by which the document is identified by the originating activity. This number must be unique to this document.)

   DRDC Atlantic TM-2006-256

10b. OTHER DOCUMENT NO(s). (Any other numbers which may be assigned this document either by the originator or by the sponsor.)

11. DOCUMENT AVAILABILITY (Any limitations on further dissemination of the document, other than those imposed by security classification.)

   (X) Unlimited distribution
   ( ) Defence departments and defence contractors; further distribution only as approved
   ( ) Defence departments and Canadian defence contractors; further distribution only as approved
   ( ) Government departments and agencies; further distribution only as approved
   ( ) Defence departments; further distribution only as approved
   ( ) Other (please specify):

12. DOCUMENT ANNOUNCEMENT (Any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution (beyond the audience specified in (11)) is possible, a wider announcement audience may be selected.)

13. ABSTRACT (A brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual.)

The Initial Graphics Exchange Specification (IGES) is in widespread use as a means of exchanging graphical information between computer programs. It can also be used to exchange the definitions of geometrical objects, regardless of how they are displayed. C++ classes in the CurveLib library allow geometry to be represented as differentiable functions within computer programs written in C++. This document describes a library of C++ classes which allow the geometry described by an IGES file to be represented by the CurveLib classes. Conversely, geometry represented by the CurveLib classes can be stored in an IGES file so that it can be used by other applications.

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (Technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus. e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus identified. If it is not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title.)

IGES
CurveLib
splines
B-splines
NURBs
differentiable functions
C++
computer programs

This page intentionally left blank.

Defence R&D Canada

Canada's leader in defence
and National Security
Science and Technology

R & D pour la défense Canada

Chef de file au Canada en matière
de science et de technologie pour
la défense et la sécurité nationale

www.drdc-rddc.gc.ca