Chapter 37.
Classes Sr thru Sz

Topic: Ignore

standard_error_info

Class: Debugging

Purpose: Returns standard error information.

Derivation: standard_error_info : error_info : ACIS_OBJECT : –

SAT Identifier: None

Filename: kern/kernel/kernutil/errorsys/stde_info.hxx

Description: This class contains standard error information, which is sufficient for use in many circumstances. For example, when working in the Local Operations Component, if unable to solve for an EDGE, the error LOP_TWK_NO_EDGE will produce an outcome that points to a standard error info object. The argument entity0 will be set to the EDGE in question.

This class also contains flags that warn the user if the returned ENTITY will be deleted on roll back.

Two overloaded versions of the function sys_error set a global pointer to an error_info object. One version is passed an error_info object, and the other creates a standard_error_info object when sys_error is passed one or two ENTITYs. The standard_error_info class is derived from error_info, which provides error data that is adequate in a majority of cases, such as local operations and blending.

In the Local Ops, Remove Faces, and Shelling Components, the error_info object returns an ENTITY that further specifies where the local operation first fails, when such information is available. A standard_error_info object is adequate for use in these components, and more detailed information could be returned, if necessary, by deriving a new class.

Limitations: None

References: KERN ENTITY

Data:

```
public ENTITY *entity0;
```
First entity with standard error information.

```
public ENTITY *entity1;
```
Second entity with standard error information.

```
public logical entity0_dead;
```
Flag that warns whether the returned entity0 will be deleted on rollback.

```
public logical entity1_dead;
```
Flag that warns whether the returned entity1 will be deleted on rollback.

Constructor:

```
public: standard_error_info::standard_error_info (
    ENTITY* e0              // Pointer to first
        = NULL,             // entity with error
    ENTITY* e1              // Pointer to second
        = NULL              // entity with error
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

Destructor:

```
public: virtual
    standard_error_info::~standard_error_info ();
```

C++ destructor, deleting a standard_error_info.

Methods:

```
public: static int standard_error_info::id ();
```

Identifies the standard error object.

```
public: virtual int
    standard_error_info::type () const;
```

Returns the string “standard_error_info”.

Related Fncs:

None

STRAIGHT

Class: Model Geometry, SAT Save and Restore

Purpose: Defines an infinite line as an object in the model.

Derivation: STRAIGHT : CURVE : ENTITY : ACIS_OBJECT : –

SAT Identifier: “straight”

Filename: kern/kernel/kerndata/geom/straight.hxx

Description: STRAIGHT is a model geometry class that contains a pointer to a (lowercase) straight, the corresponding construction geometry class. In general, a model geometry class is derived from ENTITY and is used to define a permanent model object. It provides model management functionality, in addition to the geometry definition.

STRAIGHT is one of several classes derived from CURVE to define a specific type of curve. The straight class defines an infinite line by a point (SPAposition) on the line and its direction (SPAunit_vector).

Along with the usual CURVE and ENTITY class methods, STRAIGHT has member methods to provide access to specific implementations of the geometry. For example, methods are available to set and retrieve the root point and direction of a line.

A use count allows references to multiple STRAIGHTs. The construction of a new STRAIGHT initializes the use count to 0. Methods are provided to increment and decrement the use count, and after the use count returns to 0, the entity is deleted.

Limitations: None

References: KERN straight

Data:

None

Constructor:

```
public: STRAIGHT::STRAIGHT ();
```

C++ allocation constructor requests memory for this object but does not populate it. The allocation constructor is used primarily by restore. Applications should call this constructor only with the overloaded new operator, because this reserves the memory on the heap, a requirement to support roll back and history management.

```
public: STRAIGHT::STRAIGHT (
    SPAposition const&,      // position
    SPAunit_vector const&    // unit vector
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments. Applications should call this constructor only with the overloaded new operator, because this reserves the memory on the heap, a requirement to support roll back and history management.

Creates a STRAIGHT that passes through a given SPAposition and in the direction of a given unit vector.

```
public: STRAIGHT::STRAIGHT (
    straight const&          // straight object
    );
```

C++ copy constructor requests memory for this object and populates it with the data from the object supplied as an argument. Applications should call this constructor only with the overloaded new operator, because this reserves the memory on the heap, a requirement to support roll back and history management.

Destructor:

```
public: virtual void STRAIGHT::lose ();
```

Posts a delete bulletin to the bulletin board indicating the instance is no longer used in the active model. The lose methods for attached attributes are also called.

```
protected: virtual STRAIGHT::~STRAIGHT ();
```

This C++ destructor should never be called directly. Instead, applications should use the overloaded lose method inherited from the ENTITY class, because this supports history management. (For example, x=new STRAIGHT(...) then later x–>lose.)

Methods:

```
protected: virtual logical
     STRAIGHT::bulletin_no_change_vf (
    ENTITY const* other,         // other entity
    logical identical_comparator// comparator
    ) const;
```

Virtual function for comparing subclass data – called by bulletin_no_change. For identical_comparator to be TRUE requires an exact match when comparing doubles, and returns the result of memcmp as a default (for non–overridden subclasses). FALSE indicates tolerant compares and returns FALSE as a default.

```
public: virtual void STRAIGHT::debug_ent (
    FILE*                   // file pointer
    ) const;
```

Prints the type and address of this object, roll back pointer, attributes, and any unknown subtype information to the specified file. Refer to the ENTITY class for more details.

```
public: SPAunit_vector const&
    STRAIGHT::direction () const;
```

Returns the SPAunit_vector defining the direction of the line.

```
public: curve const& STRAIGHT::equation () const;
```

Returns the curve's equation for reading only.

```
public: curve& STRAIGHT::equation_for_update ();
```

Returns the address of the curve's equation for update operations. Before performing a change it checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

```
public: virtual int STRAIGHT::identity (
    int                     // level
        = 0
    ) const;
```

If level is unspecified or 0, returns the type identifier STRAIGHT_TYPE. If level is specified, returns STRAIGHT_TYPE for that level of derivation from ENTITY. The level of this class is defined as STRAIGHT_LEVEL.

```
public: virtual logical STRAIGHT::is_deepcopyable (
    ) const;
```

Returns TRUE if this can be deep copied.

```
public: void STRAIGHT::operator*= (
    SPAtransf const&        // transform
    );
```

Transforms a STRAIGHT. Before performing a change it checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

```
public: void STRAIGHT::restore_common ();
```

The RESTORE_DEF macro expands to the restore_common method, which is used in reading information from a SAT file. This method is never called directly. It is called by a higher hierarchical function if an item in the SAT file is determined to be of this class type. An instance of this class will already have been created through the allocation constructor. This method then populates the class instance with the appropriate data from the SAT file.

straight::restore_data	Low-level geometry definition for a line.

```
public: SPAposition const&
    STRAIGHT::root_point () const;
```

Returns a SPAposition defining a point on the line.

```
public: void STRAIGHT::set_direction (
    SPAunit_vector const&   // unit vector
    );
```

Sets the STRAIGHT's direction to the given SPAunit_vector. Before performing a change it checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

```
public: void STRAIGHT::set_root_point (
    SPAposition const&      // root point
    );
```

Sets the STRAIGHT's root point to the given SPAposition. Before performing a change it checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

```
public: curve* STRAIGHT::trans_curve (
    SPAtransf const&            // transform
        = * (SPAtransf* ) NULL_REF,
    logical                     // reversed flag
        = FALSE
    ) const;
```

Transforms the curve's equation. If the logical is TRUE, the curve is reversed.

```
public: virtual const char*
    STRAIGHT::type_name () const;
```

Returns the string "straight".

Internal Use: full_size

Related Fncs: is_STRAIGHT

straight

Class: Construction Geometry, SAT Save and Restore

Purpose: Defines an infinite straight line represented by a point and a unit vector specifying the direction.

Derivation: straight : curve : ACIS_OBJECT : –

SAT Identifier: "straight"

Filename: kern/kernel/kerngeom/curve/strdef.hxx

Description: This class defines an infinite straight line represented by a point and a unit vector specifying the direction. A straight also has a scale factor for the parameterization, so the parameter values can be made invariant under transformation.

A straight line is an open curve that is not periodic. It is parameterized as:

```
point = root_point + t* param_scale* direction
```

where *t* is the parameter.

Limitations: None

References:

by KERN	STRAIGHT
BASE	SPAposition, SPAunit_vector

Data:

```
public SPAposition root_point;
```
A point through which the straight line passes.

```
public double param_scale;
```
The scaling factor for parameterization that allows fixed parameters despite transformation.

```
public SPAunit_vector direction;
```
The tangent along the line. A NULL unit vector in an uninitialized straight line.

Constructor:

```
public: straight::straight ();
```

C++ allocation constructor requests memory for this object but does not populate it.

```
public: straight::straight (
    SPAposition const&,      // point
    SPAunit_vector const&,   // direction
    double                   // parameter scaling
        = 1.0
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

Creates a straight using a point, a direction, and a parameter scaling.

```
public: straight::straight (
    straight const&          // straight line
    );
```

C++ copy constructor requests memory for this object and populates it with the data from the object supplied as an argument.

Destructor:

```
public: straight::~straight ();
```

C++ destructor, deleting a straight.

Methods:

```
public: virtual int straight::accurate_derivs (
    SPAinterval const&              // portion of
        = * (SPAinterval*) NULL_REF // the curve
    ) const;
```

Returns the number of derivatives that evaluate can find accurately. For a straight, any number of derivatives can be calculated (which are all zero after the first) and so the value ALL_CURVE_DERIVATIVES is returned.

```
public: SPAbox straight::bound (
    double start,               // start point
                                // on the line
    double end,                 // end point
                                // on the line
    SPAtransf const& t          // transformation
        = * (SPAtransf* ) NULL_REF
    ) const;
```

Returns a box enclosing the two given points on the straight line. This function is retained for historical reasons.

```
public: virtual SPAbox straight::bound (
    SPAbox const&,              // box
    SPAtransf const&            // transformation
        = * (SPAtransf* ) NULL_REF
    ) const;
```

Returns a box surrounding that portion of the curve within the given box.

```
public: virtual SPAbox straight::bound (
    SPAinterval const&,         // two given points
                                // represented as an
                                // interval
    SPAtransf const&            // transformation
        = * (SPAtransf* ) NULL_REF
    ) const;
```

Return a box enclosing the two given points on the straight line.

```
public: virtual SPAbox straight::bound (
    SPAposition const&,         // first position
    SPAposition const&,         // second position
    SPAtransf const&            // transformation
        = * (SPAtransf* ) NULL_REF
    ) const;
```

Return a box enclosing the two given points on the straight line.

```
public: virtual logical straight::closed () const;
```

Indicates if a curve is closed. This function joins itself (smoothly or not) at the ends of its principal parameter range. This function should always return TRUE if periodic does.

```
public: virtual void straight::debug (
    char const*,            // title line
    FILE*                   // file name
        = debug_file_ptr
    ) const;
```

Outputs a title line and details about the straight line to the debug file or to the specified file.

```
public: virtual curve* straight::deep_copy (
    pointer_map* pm         // list of items within
        = NULL              // the entity that are
                            // already deep copied
    ) const;
```

Creates a copy of an item that does not share any data with the original. Allocates new storage for all member data and any pointers. Returns a pointer to the copied item.

```
public: virtual curve_boundcyl
    straight::enclosing_cylinder (
    const SPAinterval&          // interval
        = * (SPAinterval*) NULL_REF
    ) const;
```

Returns a cylinder that encloses the portion of the curve bounded by the interval.

```
public: virtual void straight::eval (
    double,                       // parameter value
    SPAposition&,                 // position
    SPAvector&                    // first derivative
        = * (SPAvector* ) NULL_REF,
    SPAvector&                    // second derivative
        = * (SPAvector* ) NULL_REF,
    logical                       // take advantage of
                                  // report orders, so
                                  // values don't
                                  // have to be
                                  // recomputed?
        = FALSE,
    logical                       // approx ok?
        = FALSE
    ) const;
```

Evaluate the curve at a given parameter value, giving the position, and first and second derivatives, all of which are optional.

```
public: virtual int straight::evaluate (
    double,                       // parameter
    SPAposition&,                 // point on curve at
                                  // given parameter
    SPAvector**                   // array of ptrs to
        = NULL,                   // vectors, size nd.
                                  // any of the ptrs
                                  // may be null, =>
                                  // corresponding
                                  // derivative won't
                                  // be returned
    int                           // # derivatives
        = 0,                      // required (nd)
    evaluate_curve_side           // the evaluation
        = evaluate_curve_unknown// location – above,
                                  // below or don't
                                  // care
    ) const;
```

Calculates derivatives, of any order up to the number requested, and store them in vectors provided by the user. This function returns the number it was able to calculate; this will be equal to the number requested in all but the most exceptional circumstances. A certain number will be evaluated directly and (more or less) accurately; higher derivatives will be automatically calculated by finite differencing; the accuracy of these decreases with the order of the derivative, as the cost increases.

```
public: virtual SPAvector straight::eval_curvature (
    double,                 // parameter
    logical                 // repeat order?
        = FALSE,
    logical                 // approx ok?
        = FALSE
    ) const;
```

Returns the curvature at a given parameter value. This is always zero, so all the arguments are ignored.

```
public: virtual double straight::eval_deriv_len (
    double,                 // point
    logical                 // repeat order?
        = FALSE,
    logical                 // approx ok?
        = FALSE
    ) const;
```

Finds the magnitude of the derivative at the given parameter value on the curve.

```
public: virtual curve_extremum*
    straight::find_extrema (
    SPAunit_vector const&   // direction
    ) const;
```

Finds the extrema of the curve in the given direction. A straight line has no extrema, so this method returns NULL.

```
public: virtual int straight::high_curvature (
    double k,               // maximum curvature
    SPAinterval*& spans     // interval list
) const;
```

Finds regions of high curvature of the curve. This method stores an array of intervals in spans argument over which the curvature exceeds k. It returns the number of intervals stored.

```
public: law* straight::law_form ();
```

Returns the law form of the straight entity.

```
public: virtual double straight::length (
    double,                 // start parameter
    double                  // end parameter
    ) const;
```

Return the algebraic distance along the curve between the given parameters, the sign being positive if the parameter values are given in increasing order, and negative if they are in decreasing order. The result is undefined if either parameter value is outside the parameter range of a bounded curve. For a periodic curve the parameters are not reduced to the principal range, and so the portion of the curve evaluated may include several complete circuits. This function is therefore always a monotonically increasing function of its second argument if the first is held constant, and a decreasing function of its first argument if the second is held constant.

```
public: virtual double straight::length_param (
    double,                 // datum parameter
    double                  // arc length
    ) const;
```

Returns the parameter value of the point on the curve at the given algebraic arc length from that defined by the datum parameter. This method is the inverse of the length method. The result is not defined for a bounded nonperiodic curve if the datum parameter is outside the parameter range, or if the length is outside the range bounded by the values for the ends of the parameter range.

```
public: virtual curve* straight::make_copy () const;
```

Makes a copy of this straight on the heap, and returns a pointer to it.

```
public: virtual curve& straight::negate ();
```

Negates this straight line (i.e., this method negates the direction).

```
public: virtual curve& straight::operator*= (
    SPAtransf const&        // transformation
    );
```

Transforms this straight line by the given transformation.

```
public: straight straight::operator– () const;
```

Returns a straight line with the opposite sense from this line.

```
public: virtual logical straight::operator== (
    curve const&            // curve
    ) const;
```

Tests two curves for equality. This method does not guarantee equality for effectively-equal curves, but it is guaranteed to determine inequality if the two curves are not equal. Use this result for optimization.

```
public: virtual double straight::param (
    SPAposition const&,             // position
    SPAparameter const&             // param guess
        = * (SPAparameter* ) NULL_REF
    ) const;
```

Finds the parameter value at the given point on the curve. If the point is not on the curve, a plane is constructed perpendicular to the line, and the parameter value for its intersection with the line returns.

```
public: virtual double
    straight::param_period () const;
```

Returns the parameter period, 0 in this case because a straight line is not periodic

```
public: virtual SPAinterval straight::param_range (
    SPAbox const&           // box
        = * (SPAbox* ) NULL_REF
    ) const;
```

Returns the principal parameter range, or what is inside the given box, if one is indeed given. If there is no box, the range is unbounded, and we return the empty interval.

```
public: virtual logical straight::periodic () const;
```

Indicates if the curve is periodic and joins itself smoothly at the ends of its principal parameter range, so that edges may span the seam.

```
public: virtual SPAvector straight::point_curvature (
    SPAposition const&,             // point
    SPAparameter const&             // param guess
        = * (SPAparameter* ) NULL_REF
    ) const;
```

Returns the curvature, which is 0 for a straight line. It is immaterial whether the point is on or off the curve. The SPAparameter argument is ignored.

```
public: virtual
    SPAunit_vector straight::point_direction (
    SPAposition const&,             // position
    SPAparameter const&             // param guess
        = * (SPAparameter* ) NULL_REF
    ) const;
```

Returns the direction of the curve at a point on it. This is a constant for a straight line, so it is immaterial whether the given point is on or off the curve.

```
public: virtual void straight::point_perp (
    SPAposition const&,             // position
    SPAposition&,                   // foot
    SPAunit_vector&,                // curve
                                    // direction at
                                    // foot
    SPAvector&,                     // curvature at
                                    // foot
    SPAparameter const&             // param guess
        = * (SPAparameter* ) NULL_REF,
    SPAparameter&                   // actual param
        = * (SPAparameter* ) NULL_REF,
    logical f_weak          // weak flag
        = FALSE
    ) const;
```

Finds the foot of the perpendicular from the given point to the curve, and tangent to the curve at that point, and its parameter value. If an input parameter value is supplied (as argument 5), the perpendicular found is the one nearest to the supplied parameter position, otherwise it is the one at which the curve is nearest to the given point. Any of the return value arguments may be a NULL reference, in which case it is simply ignored.

```
public: void straight::point_perp (
    SPAposition const& pos,         // position
    SPAposition& foot,              // foot
    SPAparameter const& guess       // param guess
        = * (SPAparameter* ) NULL_REF,
    SPAparameter& actual            // actual param
        = * (SPAparameter* ) NULL_REF,
    logical f_weak          // weak flag
        = FALSE
    ) const;
```

Find the foot of the perpendicular from the given point to the curve, and tangent to the curve at that point, and its parameter value

```
public: void straight::point_perp (
    SPAposition const& pos,         // position
    SPAposition& foot,              // foot
    SPAunit_vector& foot_dt,        // tangent to
                                    // curve at foot
    SPAparameter const& guess       // param guess
        = * (SPAparameter* ) NULL_REF,
    SPAparameter& actual            // actual param
        = * (SPAparameter* ) NULL_REF,
    logical f_weak           // weak flag
        = FALSE
    ) const;
```

Drop a perpendicular from the given point to the line, returning the foot of the perpendicular, and the curve direction there.

```
public: void straight::restore_data ();
```

Restore the data for a straight from a save file.

read_position	Root point
read_vector	The direction of straight
curve::restore_data	Restore the rest of the curve data.

```
public: virtual void straight::save () const;
```

Saves the type or id, then calls save_data.

```
public: void straight::save_data () const;
```

Save the information for the straight to a save file.

```
public: virtual curve_tancone
    straight::tangent_cone (
    SPAinterval const&,          // line interval
    logical,                     // approx results OK?
    SPAtransf const&             // transformation
        = * (SPAtransf* ) NULL_REF
    ) const;
```

Returns a cone bounding the tangent direction of a curve. The cone has its apex at the origin, and it has a given axis direction and positive half-angle. If logical is TRUE, then approximate results are found. The approximate result may lie wholly within or wholly outside the guaranteed bound (obtained with the FALSE argument), but it may not cross from side to outside. Flags in the returned object indicate whether the cone is the best available, and if the result is inside or outside the best cone.

```
public: virtual logical straight::test_point_tol (
    SPAposition const&,             // point
    double                          // tolerance
        = 0,
    SPAparameter const&             // param guess
        = * (SPAparameter* ) NULL_REF,
    SPAparameter&                   // actual param
        = * (SPAparameter* ) NULL_REF
    ) const;
```

Tests a point on the curve, returning the parameter value if it is required.

```
public: virtual int straight::type () const;
```

Returns the type of straight.

```
public: virtual char const*
    straight::type_name () const;
```

Returns a string “straight”.

```
public: logical straight::undef () const;
```

Determines whether this curve is undefined.

Internal Use: full_size

Related Fncs:

restore_straight

```
friend: straight operator* (
    straight const&,         // input straight
    SPAtransf const&         // transform to use
    );
```

Transforms the given straight by the transform given.

StreamFinder

Class: History and Roll

Purpose: Used by the DistributeStates function below to find the HISTORY_STREAM corresponding to a given entity.

Derivation: StreamFinder : ACIS_OBJECT : –

SAT Identifier: None

Filename: kern/kernel/sg_husk/history/history.hxx

Description: This class is used by the DistributeStates function below to find the HISTORY_STREAM corresponding to a given entity. It gives a great deal of flexibility and supports a variety of usage scenarios. At the same time user needs to be careful during usage. This is a class, instead of function pointers, hence it can cache data or arguments without using the global data hack. Examples of implementations are:
. Find the top level owner and get the stream from an ATTRIB_HISTORY found there.
. Find the PART for the ENTITY and return the stream associated with the PART.
. Always return the same stream. This is useful when the application has some knowledge about what has transpired and knows all BULLETINs go on the same stream.

Limitations: None

References: by KERN HISTORY_MANAGER

Data:

```
public enum LookStrategy;
```

Type of look strategy.

Constructor:

```
public: StreamFinder::StreamFinder ();
```

C++ allocation constructor requests memory for this object.

Destructor:

None

Methods:

```
public: virtual HISTORY_STREAM*
StreamFinder::findStream (
    ENTITY*,                //Entity Object
    LookStrategy strategy   //Type of look strategy
        = LookNormal        //
    )= 0;
```

Function to return the HISTORY_STREAM for any entity.

```
public: virtual HISTORY_STREAM*
StreamFinder::quick_findStream (
    ENTITY*                 //Entity Object
    );
```

Function to return the HISTORY_STREAM for any entity whose stream needs to be quickly determined. This check is very fast.

Related Fncs:

None

SUBSHELL

Class: Model Topology, SAT Save and Restore

Purpose: Represents a subdivision of a SHELL or SUBSHELL.

Derivation: SUBSHELL : ENTITY : ACIS_OBJECT : –

SAT Identifier: “subshell”

Filename: kern/kernel/kerndata/top/subshell.hxx

Description: A subshell represents a subdivision of a shell or superior subshell. The efficiency of many-to-many comparisons is improved by allowing quantities of faces to be excluded by a single box test. The subdivision is determined by the system and may change at any time. Thus, the subshell has no significance to the user, although the application program may find the implied spatial subdivision useful.

Limitations: None

References:

KERN	FACE, WIRE
by KERN	FACE, SHELL, WIRE

Data:

None

Constructor:

```
public: SUBSHELL::SUBSHELL ();
```

C++ allocation constructor requests memory for this object but does not populate it. The allocation constructor is used primarily by restore. Applications should call this constructor only with the overloaded new operator, because this reserves the memory on the heap, a requirement to support roll back and history management.

```
public: SUBSHELL::SUBSHELL (
    FACE*,                  // list of FACEs
    SUBSHELL*,              // list of SUBSHELLs
    SUBSHELL*               // sister SUBSHELLs list
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments. Applications should call this constructor only with the overloaded new operator, because this reserves the memory on the heap, a requirement to support roll back and history management.

Constructs a SUBSHELL, initializing the record and interacting with the bulletin board. It sets back pointer in child SUBSHELLs and FACEs. The first two arguments define the starts of lists of FACEs and SUBSHELLs contained within the SUBSHELL, and the last is a list of sibling SUBSHELLs already in the current BODY. The arguments initialize face_ptr, child_ptr, and sibling_ptr respectively. The calling routine must set parent_ptr and if desired, bound_ptr, using set_parent and set_bound.

Destructor:

```
public: virtual void SUBSHELL::lose ();
```

Posts a delete bulletin to the bulletin board indicating the instance is no longer used in the active model. The lose methods for attached attributes are also called.

```
protected: virtual SUBSHELL::~SUBSHELL ();
```

This C++ destructor should never be called directly. Instead, applications should use the overloaded lose method inherited from the ENTITY class, because this supports history management. (For example, x=new SUBSHELL(...) then later x–>lose.)

Methods:

```
public: SPAbox* SUBSHELL::bound () const;
```

Returns a pointer to the geometric bounding region (box) that contains the entire SUBSHELL (with respect to the internal coordinate system of the BODY). This function returns NULL if the bounding box was not calculated since the SUBSHELL was last modified.

```
protected: virtual logical
     SUBSHELL::bulletin_no_change_vf (
    ENTITY const* other,        // other entity
    logical identical_comparator// comparator
    ) const;
```

Virtual function for comparing subclass data – called by bulletin_no_change. For identical_comparator to be TRUE requires an exact match when comparing doubles, and returns the result of memcmp as a default (for non–overridden subclasses). FALSE indicates tolerant compares and returns FALSE as a default.

```
public: SUBSHELL* SUBSHELL::child () const;
```

Returns a pointer to the start of list of SUBSHELLs contained within the current SUBSHELL. This data scheme allows a hierarchy of subdivisions.

```
public: virtual void SUBSHELL::debug_ent (
    FILE*                   // file pointer
    ) const;
```

Prints the type and address of this object, roll back pointer, attributes, and any unknown subtype information to the specified file. Refer to the ENTITY class for more details.

```
public: FACE* SUBSHELL::face () const;
```

Returns the first FACE in a complete enumeration of all the FACEs in the owning SHELL, continued by repeated use of FACE::next.

```
public: FACE* SUBSHELL::face_list () const;
```

Return a pointer to the first FACE in a list of FACEs contained in this SUBSHELL. Each FACE in a SHELL must be in the face list of the SHELL, or exactly one of its SUBSHELLs.

```
public: virtual int SUBSHELL::identity (
    int                     // level
        = 0
    ) const;
```

If level is unspecified or 0, returns the type identifier SUBSHELL_TYPE. If level is specified, returns SUBSHELL_TYPE for that level of derivation from ENTITY. The level of this class is defined as SUBSHELL_LEVEL.

```
public: virtual logical SUBSHELL::is_deepcopyable (
    ) const;
```

Returns TRUE if this can be deep copied.

```
public: ENTITY* SUBSHELL::owner () const;
```

Returns a pointer to the owning parent.

```
public: SUBSHELL* SUBSHELL::parent () const;
```

Returns a pointer to the next superior SUBSHELL. The return is NULL if the SUBSHELL belongs directly to a SHELL.

```
public: void SUBSHELL::restore_common ();
```

The RESTORE_DEF macro expands to the restore_common method, which is used in reading information from a SAT file. This method is never called directly. It is called by a higher hierarchical function if an item in the SAT file is determined to be of this class type. An instance of this class will already have been created through the allocation constructor. This method then populates the class instance with the appropriate data from the SAT file.

read_ptr	Pointer to record in save file for parent SUBSHELL
read_ptr	Pointer to record in save file for next SUBSHELL belonging to parent (sibling)
read_ptr	Pointer to record in save file for first child SUBSHELL
read_ptr	Pointer to record in save file for first FACE in subshell
if (restore_version_number >= WIREBOOL_VERSION)	
read_ptr	Pointer to record in save file for first WIRE in subshell
else	Pointer for first WIRE in subshell is set to NULL.

```
public: void SUBSHELL::set_bound (
    SPAbox*                 // new bounding box
    );
```

Sets the SUBSHELL's bounding box pointer to the given box. Before performing a change, it checks whether the data structure is posted on the bulletin board. If not, the method calls backup to put an entry on the bulletin board.

```
public: void SUBSHELL::set_child (
    SUBSHELL*               // child SUBSHELLs list
    );
```

Sets the SUBSHELL's child pointer to the given SUBSHELL starting a list of child SUBSHELLs. Before performing a change, it checks whether the data structure is posted on the bulletin board. If not, the method calls backup to put an entry on the bulletin board.

```
public: void SUBSHELL::set_face (
    FACE*                   // list of FACEs
    );
```

Sets the SUBSHELL's FACE pointer to the given FACE at the start of a list of FACEs. Before performing a change, it checks whether the data structure is posted on the bulletin board. If not, the method calls backup to put an entry on the bulletin board.

```
public: void SUBSHELL::set_parent (
    SUBSHELL*                // parent SUBSHELL
    );
```

Sets the SUBSHELL's parent pointer to the given SUBSHELL. Before performing a change, it checks whether the data structure is posted on the bulletin board. If not, the method calls backup to put an entry on the bulletin board.

```
public: void SUBSHELL::set_sibling (
    SUBSHELL*                // sister SUBSHELL
    );
```

Sets the SUBSHELL's sibling pointer to the given SUBSHELL. Before performing a change, it checks whether the data structure is posted on the bulletin board. If not, the method calls backup to put an entry on the bulletin board.

```
public: void SUBSHELL::set_wire (
    WIRE*                    // wire
    );
```

Sets the SUBSHELL's WIRE pointer to the given WIRE at the start of a list of WIREs. Before performing a change, it checks whether the data structure is posted on the bulletin board. If not, the method calls backup to put an entry on the bulletin board.

```
public: SUBSHELL* SUBSHELL::sibling () const;
```

Returns a pointer to the next SUBSHELL in the list contained by a superior SHELL or SUBSHELL.

```
public: virtual const char*
    SUBSHELL::type_name () const;
```

Returns the string "subshell".

```
public: WIRE* SUBSHELL::wire () const;
```

Returns the first WIRE in a complete enumeration of all the WIREs in the owning SHELL, continued by repeated use of WIRE::next.

```
public: WIRE* SUBSHELL::wire_list () const;
```

Return a pointer to the first WIRE in a list of WIREs contained in this SUBSHELL. Each WIRE in a SHELL must be in the face list of the SHELL, or exactly one of its SUBSHELLs.

Internal Use: first_face, save, save_common

Related Fncs: is_SUBSHELL

subtrans_object

Class: Construction Geometry

Purpose: Defines a shared-subtype class that is subject to transformations.

Derivation: subtrans_object : subtype_object : ACIS_OBJECT : –

SAT Identifier: None

Filename: kern/kernel/kernutil/subtype/subtrans.hxx

Description: This class defines a shared-subtype class that is subject to transformations, and is designed to ensure that transforming several references to the same object with the same transformation retains the sharing.

Limitations: None

References: None

Data: None

Constructor:

```
protected: subtrans_object::subtrans_object ();
```

C++ constructor, creating a subtrans_object.

```
protected: subtrans_object::subtrans_object (
    subtrans_trans*,         // transform list
    subtrans_object*         // parent object
    );
```

This constructor transforms the given parent to make a new child, and hooks up all the pointers. Constructor to be used when a linked list of extensions is made. Initially the linked list is of length 1 and parent points to itself.

Destructor:

```
protected: subtrans_object::~subtrans_object ();
```

C++ destructor, deleting a subtrans_object. Removes references in the trans_list and also the one referring to this in the parent's list.

Methods:

```
public: void subtrans_object::clear_trans ();
```

Clears any references to this object using transformation lists, before incompatible changes.

```
protected: virtual subtrans_object*
    subtrans_object::copy () const = 0;
```

Duplicates this object. This method is virtual so that the true derived object is copied.

```
public: subtrans_object* subtrans_object::get_next ()
const;
```

Returns the next object in the list.

```
public: subtrans_object*
    subtrans_object::make_trans (
    SPAtransf const&        // transformation
    );
```

Transforms the subtrans_object on an external level. This method searches the transform list to find a match. It returns the corresponding transformed object if it is found; otherwise, it constructs a new transformed object, enters it into the transformation list for future reference, and returns the new object. If this object is only singly-referenced and has no transformed counterpart, then it transforms directly without copying, and "this" returns.

```
protected: virtual void subtrans_object::operator*= (
    SPAtransf const&        // transformation
    ) = 0;
```

Transforms the subtrans_object by the given transformation.

```
public: void subtrans_object::set_next (
    subtrans_object*        // object
    );
```

Sets the next object in the list.

Internal Use: full_size

Related Fncs:

None

subtype_object

Class: Construction Geometry

Purpose: Defines the master object from which all subtype objects must be derived.

Derivation: subtype_object : ACIS_OBJECT : –

SAT Identifier: None

Filename: kern/kernel/kernutil/subtype/subtype.hxx

Description: This class defines the master object from which all subtype objects must be derived. This object contains a use count (in case the object is shareable) and defines two virtual functions and a destructor.

Limitations: None

References: None

Data:

None

Constructor:

```
public: subtype_object::subtype_object ();
```

C++ constructor, creating a subtype_object.

Destructor:

```
public: virtual subtype_object::~subtype_object ();
```

C++ destructor, deleting a subtype_object. and all those objects that need it.

Methods:

```
public: void subtype_object::add_ref ();
```

Use count manipulation.

```
public: virtual void subtype_object::debug (
    char const*,            // leader for second and
                            // subsequent lines of
                            // output
    logical,                // TRUE for brief output
                            // FALSE for full output
    FILE*                   // file name
        = debug_file_ptr
    ) const = 0;
```

Writes the subtype_object in readable form for debugging. If logical is TRUE, brief output is produced; if logical is FALSE, long output is produced.

```
public: logical subtype_object::mult_ref () const;
```

Returns TRUE if there is more than one reference; otherwise, it returns FALSE.

```
public: logical subtype_object::operator!= (
    subtype_object const& rhs   // subtype-object
    ) const;
```

TRUE if two subtype objects are not the same. Use this method for sharing on restoration of old save files.

```
public: virtual logical subtype_object::operator== (
    subtype_object const&   // subtype-object
    ) const;
```

Determines if two subtype objects are the same. Use this method for sharing on restoration of old save files.

```
public: int subtype_object::ref_count ();
```

Returns the use count.

```
public: void subtype_object::remove_ref ();
```

Removes a reference.

```
public: virtual void subtype_object::save () const;
```

Saves a subtype object with identifier and brackets, and enters it in the current tag table. If it is already there, just puts out a reference.

```
public: virtual void
    subtype_object::save_data () const = 0;
```

Saves the information for the subtype_object to a save file.

```
public: virtual int
    subtype_object::type () const = 0;
```

Returns the type of subtype_object.

```
public: virtual char const*
    subtype_object::type_name () const = 0;
```

Returns a pointer to a static string, which is the externally-meaningful type name for this subtype.

```
public: virtual logical
    subtype_object::unknown_type () const;
```

Returns TRUE if this object type is unknown to the system; otherwise, it returns FALSE. This method returns FALSE as the default, so except for the system unknown subtype, the method should be omitted from derived class definitions.

Internal Use: full_size

Related Fncs: None

sum_spl_sur

Class: Construction Geometry, SAT Save and Restore

Purpose: Represents a linear sum of two curves.

Derivation: sum_spl_sur : spl_sur : subtrans_object : subtype_object : ACIS_OBJECT : –

SAT Identifier: “sumsur”

Filename: kern/kernel/kerngeom/splsur/sum_spl.hxx

Description: This class represents a surface that is a linear sum of two curves. This is derived from the class spl_sur, which is used by the spline surface class to contain the surface descriptions. The surface is defined primarily by two curves that are assumed not parallel, and the parameter ranges over which the surface is defined.

Parametric Representation

If the curves are represented as:

$x = c_1(t)$ and $x = c_2(t)$

the surface is:

$x = s(u, v) = c_1(u) + c_2(v) - p$

where p is a constant position, normally initialized to be the value of c_2 at the start of the parameter range.

Limitations: None.

References:

KERN	curve
BASE	SPAposition

Data:

None

Constructor:

```
public: sum_spl_sur::sum_spl_sur (
    const sum_spl_sur&      // input curve
    );
```

C++ copy constructor requests memory for this object and populates it with the data from the object supplied as an argument.

```
public: sum_spl_sur::sum_spl_sur (
    curve const&,                   // 1st curve
    curve const&,                   // 2nd curve
    SPAinterval const&              // 1st curve
        = * (SPAinterval* ) NULL_REF,// param range
                                    // (u–param)
    SPAinterval const&              // 2nd curve
        = * (SPAinterval* ) NULL_REF,// param range
                                    // (v–param)
    SPAposition const&              // datum position
        = * (SPAposition* ) NULL_REF
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

The u-parameter range defaults to the full first curve and the *v*-parameter range defaults to the full second curve. If either curve is unbounded, this constructor returns an error. The datum position is subtracted from the sum of the curves to give a position for the surface and is normally initialized to be the start of cur2.

Destructor:

None

Methods:

```
protected: virtual SPAbox sum_spl_sur::bound (
    SPApar_box const&           // parameter box of
        = * (SPApar_box* )NULL_REF// defining curves
    );
```

Returns a bounding box by boxing the defining curves and combining them.

```
public: virtual spl_sur* sum_spl_sur::deep_copy (
    pointer_map* pm         // list of items within
        = NULL              // the entity that are
                            // already deep copied
    ) const;
```

Creates a copy of an item that does not share any data with the original. Allocates new storage for all member data and any pointers. Returns a pointer to the copied item.

```
public: virtual curve*
    sum_spl_sur::get_path () const;
```

Returns the sweep path curve.

```
public: virtual sweep_path_type
    sum_spl_sur::get_path_type () const;
```

Returns the sweep path type

```
public: virtual curve*
    sum_spl_sur::get_profile (
    double                  // param
) const;
```

Returns the sweep profile on the sum_spl_sur, which is the *u*-curve.

```
public: virtual law*
    sum_spl_sur::get_rail () const;
```

Returns the rail law for the sum_spl_sur.

```
public: static int sum_spl_sur::id ();
```

Returns the ID for the sum_spl_sur list.

```
public: virtual logical sum_spl_sur::is_extendable (
    sum_spl_sur const& other,   // other surface
    SPApar_box& new_domain      // new domain
    ) const;;
```

This method will return true if the sum_spl_sur can be extended to include the given sum_spl_sur. It also returns what the new domain of the “this” surface should be to include the other surface. This method returns false in any case that is not covered.

```
protected: virtual void sum_spl_sur::make_approx (
    double fit,             // fit tolerance
    const spline& spl       // pointer to output
        = * (spline*) NULL_REF,// spline approx.
    logical force           // flag for forcing
        = FALSE
    ) const;
```

Makes or remakes an approximation of the surface, within the given tolerance.

```
private: void sum_spl_sur::restore_data ();
```

Restores the information for the sum_spl_sur from a save file.

```
 restore_curve                          u-curve
restore_curve                           v-curve
read_position                           Datum point
if ( restore_version_number < APPROX_SUMMARY_VERSION )
     read_interval                      u-range
     read_interval                      v-range
     if ( restore_version_number >= DISCONTINUITY_VERSION )
          discontinuity_info::restore   u discontinuities
          discontinuity_info::restore   v discontinuities
else
     spl_sur::restore_common_data       restore the generic surface data
```

```
public: virtual void sum_spl_sur::save_data () const;
```

Save the information for the sum_spl_sur to a save file.

```
public: virtual int sum_spl_sur::type () const;
```

Returns the type of sum_spl_sur.

```
public: virtual char const*
    sum_spl_sur::type_name () const;
```

Returns the string “sumsur”.

Internal Use: full_size

Related Fncs:

---

restore_sum_spl_sur

SURFACE

Class: Model Geometry, SAT Save and Restore

Purpose: Defines a generic surface as an object in the model.

Derivation: SURFACE : ENTITY : ACIS_OBJECT : –

SAT Identifier: “surface”

Filename: kern/kernel/kerndata/geom/surface.hxx

Description: SURFACE is a model geometry class that contains a pointer to a (lowercase) surface, the corresponding construction geometry class. In general, a model geometry class is derived from ENTITY and is used to define a permanent model object. It provides model management functionality, in addition to the geometry definition.

A SURFACE provides the basic framework for the range of surface geometries implemented in the modeler. Additional classes are derived from SURFACE to define specific types of surfaces, such as CONE, MESHSURF, PLANE, SPHERE and TORUS.

Along with the usual ENTITY class methods, SURFACE has member methods to provide access to specific implementations of the geometry. For example, a surface can be transformed by a given transform operator, resulting in another surface.

A use count allows multiple references to a SURFACE. The construction of a new SURFACE initializes the use count to 0. Methods are provided to increment and decrement the use count, and after the use count returns to 0, the entity is deleted.

Limitations: None

References:

KERN	ENTITY
by KERN	FACE, pattern_holder

Data:

None

Constructor:

```
public: SURFACE::SURFACE ();
```

C++ allocation constructor requests memory for this object but does not populate it. The allocation constructor is used primarily by restore. Applications should call this constructor only with the overloaded new operator, because this reserves the memory on the heap, a requirement to support roll back and history management.

Destructor:

```
public: virtual void SURFACE::lose ();
```

Posts a delete bulletin to the bulletin board indicating the instance is no longer used in the active model. The lose methods for attached attributes are also called.

```
protected: virtual SURFACE::~SURFACE ();
```

This C++ destructor should never be called directly. Instead, applications should use the overloaded lose method inherited from the ENTITY class, because this supports history management. (For example, x=new SURFACE(...) then later x–>lose.)

Methods:

```
public: virtual void SURFACE::add ();
```

Increments the use count. Before performing a change it checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

```
public: void SURFACE::add_owner (
    ENTITY* owner,              // owner
    logical increment_use_count // increment use
        = TRUE                  // count or not
    );
```

Adds owner argument to the list of owners.

```
protected: virtual logical
     SURFACE::bulletin_no_change_vf (
    ENTITY const* other,        // other entity
    logical identical_comparator// comparator
    ) const;
```

Virtual function for comparing subclass data – called by bulletin_no_change. For identical_comparator to be TRUE requires an exact match when comparing doubles, and returns the result of memcmp as a default (for non–overridden subclasses). FALSE indicates tolerant compares and returns FALSE as a default.

```
public: virtual void SURFACE::debug_ent (
    FILE*                   // file pointer
    ) const;
```

Prints the type and address of this object, roll back pointer, attributes, and any unknown subtype information to the specified file. Refer to the ENTITY class for more details.

```
public: virtual logical SURFACE::deletable () const;
```

Indicates whether this entity is normally destroyed by lose (TRUE), or whether it is shared between multiple owners using a use count, and so gets destroyed implicitly when every owner has been lost (FALSE). The default for SURFACE is FALSE.

```
public: virtual surface const&
    SURFACE::equation () const;
```

Returns the surface's equation.

```
public: virtual surface&
    SURFACE::equation_for_update ();
```

Returns a pointer to surface equation for update operations. Before performing a change it checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

```
public: int SURFACE::get_owners (
    ENTITY_LIST& list       // list of owners
) const;
```

Copies the list of owners from this object to the list argument. The method returns the number of owners copied.

```
public: virtual int SURFACE::identity (
    int                     // level
        = 0
    ) const;
```

If level is unspecified or 0, returns the type identifier SURFACE_TYPE. If level is specified, returns SURFACE_TYPE for that level of derivation from ENTITY. The level of this class is defined as SURFACE_LEVEL.

```
public: virtual logical SURFACE::is_deepcopyable (
    ) const;
```

Returns TRUE if this can be deep copied.

```
public: virtual logical SURFACE::is_use_counted (
) const;
```

Returns TRUE if the entity is use counted.

```
public: virtual SPAbox SURFACE::make_box (
    LOOP*,                  // list of LOOPs
    SPAtransf const* t = NULL,  // for future use
    logical tight_box       // for future use
        = FALSE,
    SPAbox* untransformed_box   // for future use
        = NULL
    ) const;
```

Constructs a bounding box for a FACE. Although the generic record type should never exist, this function is defined for it, to return a box enclosing all the edges of the given list of LOOPs. This is sufficient for any ruled surface type, for which any point in a FACE must be a linear combination of some two points on its boundary.

```
public: virtual void SURFACE::operator*= (
    SPAtransf const&        // transform
    );
```

Transforms the SURFACE equation. Before performing a change it checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

```
public: virtual void SURFACE::remove (
    logical lose_if_zero    // flag for lose
        = TRUE
    );
```

Decrements the use count. If the use count reaches 0, the SURFACE is deleted. Before performing a change it checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

```
public: void SURFACE::remove_owner (
    ENTITY* owner,          // owner
    logical                 // decrement use
        = TRUE,             // count flag
    logical                 // lose if
        = TRUE              // zero flag
    );
```

Removes the owner argument from the list of owners.

```
public: void SURFACE::restore_common ();
```

The RESTORE_DEF macro expands to the restore_common method, which is used in reading information from a SAT file. This method is never called directly. It is called by a higher hierarchical function if an item in the SAT file is determined to be of this class type. An instance of this class will already have been created through the allocation constructor. This method then populates the class instance with the appropriate data from the SAT file.

```
if (restore_version_number < PATTERN_VERSION
    read_ptr                          APATTERN index
        if (apat_idx != (APATTERN*)(–1)))
        restore_cache();
if ( !get_standard_save_flag() )
    read_int                          use count data
// Nothing to copy or restore under normal circumstances.
```

```
public: virtual void SURFACE::set_use_count (
    int val                 // value to set
    );
```

Sets the reference use count of the SURFACE.

```
public: virtual surface* SURFACE::trans_surface (
    SPAtransf const&            // transform
        = * (SPAtransf* ) NULL_REF,
    logical                     // reversed
        = FALSE
    ) const;
```

Returns the transformed surface. If the logical is TRUE the surface is reversed.

```
public: virtual const char*
    SURFACE::type_name () const;
```

Returns the string “surface”.

```
public: virtual int SURFACE::use_count () const;
```

Returns the use count for the SURFACE.

Internal Use: full_size, save, save_common

Related Fncs:

is_SURFACE

surface

Class: Construction Geometry, SAT Save and Restore

Purpose: Base class for all ACIS surface types that defines the basic virtual functions that are supplied for all specific surface classes.

Derivation: surface : ACIS_OBJECT : –

SAT Identifier: None

Filename: kern/kernel/kerngeom/surface/surdef.hxx

Description: The surface class is the base class that all ACIS surface types (plane, cone, sphere, torus, and spline) are derived. The surface class defines the basic virtual functions that are supplied for all specific surface classes. Some of these functions are pure; i.e., the derived classes must define their own version; others have default definitions that can be used by the derived classes.

All ACIS surfaces have a parameterization scheme defined for them; however, the analytic surfaces (plane, cone, sphere, and torus) are not considered parametric surfaces. The only true parametric surface is the spline surface.

The parameterization of any ACIS surface maps a rectangle within a 2D vector space (*u*,*v*-parameter space) into a 3D real vector space *(xyz* object space). A surface is closed in *u* (or *v*) if the opposite sides of the rectangle map into identical curves in object space. If the derivatives also match at these boundaries, the surface is periodic in that parameter. If one side of this rectangle maps into a single point in object space, this point is a parametric singularity. If the surface normal is not continuous at this point, the point is a surface singularity.

The parameterization can be either right-handed; i.e., the surface normal is the cross product of *u* and *v,* or left-handed; i.e., the normal is the cross product of *v* and *u*.

Limitations: None

References: by KERN blend_support, exp_par_cur, int_cur, law_par_cur, off_spl_sur, skin_spl_sur, stripc, surf_surf_int, surface_law_data, taper_spl_sur

BASE SPApar_box

Data:

```
protected SPApar_box subset_range;
```

Any surface may be subset to a given parameter range.

Constructor:

```
public: surface::surface ();
```

C++ allocation constructor requests memory for this object but does not populate it.

Destructor:

```
public: virtual surface::~surface ();
```

C++ destructor, deleting a surface.

Methods:

```
public: virtual int surface::accurate_derivs (
    SPApar_box const&           // parameter box
        = * (SPApar_box* ) NULL_REF
    ) const;
```

Returns the number of derivatives that evaluate finds accurately and directly, rather than by finite differencing, over the given portion of the surface. If there is no limit to the number of accurate derivatives, this method returns the ALL_SURFACE_DERIVATIVES value.

```
public: virtual const double*
    surface::all_discontinuities_u (
    int& n_discont,          // number of disc.
    int order                // max order
    );
```

Returns (in a read-only array) the number and parameter values of discontinuities of the surface up to the given order (maximum three).

```
public: virtual const double*
    surface::all_discontinuities_v (
    int& n_discont,          // # of disc.
    int order                // max order
    );
```

Returns (in a read-only array) the number and parameter values of discontinuities of the surface up to the given order (maximum three).

```
public: virtual SPAbox surface::bound (
    SPAbox const&,              // box
    SPAtransf const&            // transformation
        = * (SPAtransf* ) NULL_REF
    ) const;
```

Returns a box around a surface bounded in object space. This box need not be the smallest box that contains the specified portion of the surface, but it must balance the tightness of the bound against the cost of the evaluation.

```
public: virtual SPAbox surface::bound (
    SPApar_box const&               // parameter box
        = * (SPApar_box* ) NULL_REF,
    SPAtransf const&                // transformation
        = * (SPAtransf* ) NULL_REF
    ) const;
```

Returns a box around a surface bounded in parameter space. This box need not be the smallest box that contains the specified portion of the surface, but it must balance the tightness of the bound against the cost of the evaluation.

```
public: virtual void surface::change_event ();
```

Notifies the derived type that the surface has been changed (e.g., the subset_range has changed) so that it can update itself. The default version of the function does nothing.

```
public: virtual check_status_list* surface::check (
    const check_fix& input          // flags for
        = * (const check_fix*)      // allowed
        NULL_REF,                   // fixes
    check_fix& result               // fixes applied
        = * (check_fix*) NULL_REF,
    const check_status_list*        // checks to be
        = (const check_status_list*)// made, default
        NULL_REF                    // is none
    );
```

Check for any data errors in the surface, and correct the errors if possible. The various arguments provide control over which checks are made, which fixes can be applied and which fixes were actually applied. The function returns a list of errors that remain in the surface on exit.

The default for the set of flags which say which fixes are allowable is none (nothing is fixed). If the list of checks to be made is null, then every possible check will be made. Otherwise, the function will only check for things in the list. The return value for the function will then be a subset of this list.

```
public: virtual logical surface::closed_u () const;
```

Determines whether the surface is closed, smoothly or not, in the *u*-parameter direction. A closed method always returns TRUE if the corresponding periodic method returns TRUE.

```
public: virtual logical surface::closed_v () const;
```

Determines whether the surface is closed, smoothly or not, in the *v*-parameter direction. A closed method always returns TRUE if the corresponding periodic method returns TRUE.

```
public: surface* surface::copy_surf () const;
```

Makes a copy of the given surface. This method calls make_copy.

```
public: virtual void surface::debug (
    char const*,            // title line
    FILE*                   // file name
        = debug_file_ptr
    ) const = 0;
```

Prints out a title line and details about the surface to the debug file or to the specified file.

```
public: virtual surface* surface::deep_copy (
    pointer_map* pm         // list of items within
        = NULL              // the entity that are
                            // already deep copied
    ) const = 0;
```

Creates a copy of an item that does not share any data with the original. Allocates new storage for all member data and any pointers. Returns a pointer to the copied item.

In a *deep* copy, all the information about the copied item is self-contained in a new memory block. By comparison, a *shallow* copy stores only the first instance of the item in memory, and increments the reference count for each copy.

The pointer_map keeps a list of all pointers in the original object that have already been deep copied. For example, a deep_copy of a complex model results in self contained data, but identical sub-parts within the model are allowed to share a single set of data.

```
public: virtual const double*
    surface::discontinuities_u (
    int& n_discont,          // number of discont.
    int order                // curve order
    ) const;
```

Returns in a read–only array the number and parameter values of discontinuities of the surface, of the given order (maximum three).

```
public: virtual const double*
    surface::discontinuities_v (
    int& n_discont,          // number of discont.
    int order                // curve order
    ) const;
```

Returns in a read–only array the number and parameter values of discontinuities of the surface, of the given order (maximum three).

```
public: virtual int surface::discontinuous_at_u (
    double u                 // parameter
    ) const;
```

Returns whether a particular parameter value is a discontinuity.

```
public: virtual int surface::discontinuous_at_v (
    double v                 // parameter
    ) const;
```

Returns whether a particular parameter value is a discontinuity.

```
public: virtual void surface::eval (
    SPApar_pos const&,       // parameter
    SPAposition&,            // point position
    SPAvector*               // first derivatives
        = NULL,              // array of length 2 in
                             // the order Xu, Xv
    SPAvector*               // second derivatives
        = NULL               // array of length 3 in
                             // the order Xuu, Xuv,
                             // Xvv
    ) const;
```

Finds the point on a parametric surface with given parameter values, and optionally the first and second derivatives as well.

```
public: virtual int surface::evaluate (
    SPApar_pos const&,              // pt on surface
    SPAposition&,                   // at given param
                                    // values
    SPAvector**                     // array of ptrs
        = NULL,                     // to array of
                                    // vectors
    int                             // number of
        = 0,                        // derivatives
    evaluate_surface_quadrant       // eval. location
        = evaluate_surface_unknown
    ) const;
```

Calculates derivatives, of any order up to the number requested, and stores them in vectors provided by the user. This method returns the number of derivatives it was able to calculate; usually, this equals the requested number. A certain number are evaluated directly and accurately; higher derivatives are automatically calculated by finite differencing. The accuracy of these decreases with the order of the derivative as the cost increases. Any of the pointers may be NULL in which case the corresponding derivatives will not be returned. Otherwise they must point to arrays long enough for all the derivatives of that order.

```
public: virtual int surface::evaluate_iter (
    SPApar_pos const&,              // pt on surface
    surface_evaldata*,              // data supplying
                                    // initial
                                    // values, and
                                    // set to reflect
                                    // the results of
                                    // this
                                    // evaluation
    SPAposition&,                   // at given param
                                    // values
    SPAvector**                     // array of ptrs
        = NULL,                     // to array of
                                    // vectors
    int                             // number of
        = 0,                        // derivatives
    evaluate_surface_quadrant       // eval. location
        = evaluate_surface_unknown
    ) const;
```

The evaluate_iter function is just like evaluate, but is supplied with a data object which contains results from a previous close evaluation, for use as initial values for any iteration involved.

```
public: virtual double surface::eval_cross (
    SPApar_pos const&,      // given parameter values
    SPAunit_vector const&   // given normal
    ) const;
```

Finds the curvature of a cross-section curve of the parametric surface at the point with the given parameter values. The cross-section curve is given by the intersection of the surface with a plane passing through the point and with the given normal.

```
public: virtual SPAunit_vector surface::eval_normal (
    SPApar_pos const&       // parameter values
    ) const;
```

Finds the normal to a parametric surface at a point with the given parameter values.

```
public: virtual SPAunit_vector surface::eval_outdir (
    SPApar_pos const&       // parameter values
    ) const;
```

Finds the outward direction from the surface at a point with the given parameter values. This method usually returns the normal, but if the nearest point is a singularity (like the apex of a cone), this method still returns an outward direction.

```
public: virtual SPAposition surface::eval_position (
    SPApar_pos const&       // parameter values
    ) const;
```

Finds the point on a parametric surface with the given parameter values.

```
public: surf_princurv surface::eval_prin_curv (
    SPApar_pos const&       // parameter values
    ) const;
```

Finds the principle aces of curvature of the surface at a point with the given parameter values.

```
public: virtual void surface::eval_prin_curv (
    SPApar_pos const&,      // parameter values
    SPAunit_vector&,        // first axis direction
    double&,                // 1st dir. curvature
    SPAunit_vector&,        // second axis direction
    double&                 // 2nd direction crvtr.
    ) const;
```

Finds the principle aces of curvature of the surface at a point with the given parameter values and the curvatures in those directions.

```
protected: int
    surface::finite_difference_derivatives (
    SPApar_pos const&,          // parameter
    SPAposition&,               // point on surface
                                // at given parameter
    SPAvector**,                // array of ptrs to
                                // array of vectors
    int,                        // # derivatives
    int,                        // # derivatives
                                // evaluated
    double,                     // finite diff.
                                // step to
                                // use in u param.
    double,                     // finite diff.
                                // step to
                                // use in v param.
    evaluate_surface_quadrant   // evaluation
                                // quadrant
    ) const;
```

Evaluates higher derivatives than are available accurately in evaluate by finite differencing. This method is available to any derived class for use in its own evaluate. It calls back the evaluate function for adjacent points to evaluate a number of derivatives, so evaluate must ensure that this does not cause a further call to this method.

```
public: virtual const discontinuity_info&
    surface::get_disc_info_u() const;
```

Return read-only access to discontinuity_info objects, if they exist. The default version of the functions return null.

```
public: virtual const discontinuity_info&
    surface::get_disc_info_v() const;
```

Return read-only access to discontinuity_info objects, if they exist. The default version of the functions return null.

```
public: virtual curve*
    surface::get_path () const;
```

Returns the path curve.

```
public: virtual sweep_path_type
    surface::get_path_type () const;
```

Returns the sweep path type for this surface.

```
public: virtual curve* surface::get_profile (
    double                  // parameter
    ) const;
```

Returns the sweep profile on the surface.

```
public: virtual law*
    surface::get_rail () const;
```

Returns the rail law for the swept surface.

```
public: virtual logical
    surface::left_handed_uv () const;
```

Indicates whether the parameter coordinate system of the surface is right-handed or left-handed. With a right-handed system, the output normal at any point is given by the cross product of the increasing *u*-direction with the increasing *v*-direction, in that order. With a left-handed system, the outward normal is in the opposite direction from this cross product.

```
public: void surface::limit (
    SPApar_box const&       // parameter box
    );
```

Limits a subset to the given parameter box.

```
public: void surface::limit_u (
    SPAinterval const&      // u interval
    );
```

Limits a subset to the given *u* interval.

```
public: void surface::limit_v (
    SPAinterval const&      // v interval
    );
```

Limits a subset to the given *v* interval.

```
public: virtual surface*
    surface::make_copy () const = 0;
```

Makes a copy of this surface on the heap, and returns a pointer to it.

```
public: virtual surface_evaldata*
    surface::make_evaldata () const;
```

Construct a data object to retain evaluation information across calls to evaluate_iter. This is to allow subsidiary calls within an iterative evaluator to start iteration much closer to the required result than is possible just using the curve information itself.

```
public: virtual surface& surface::negate () = 0;
```

Reverses the sense of a surface.

```
public: virtual surf_normcone surface::normal_cone (
    SPApar_box const&,          // parameter box
    logical                     // approx results OK?
        = FALSE,
    SPAtransf const&            // transformation
        = * (SPAtransf* ) NULL_REF
    ) const;
```

Returns a cone bounding the normal direction of the surface. The cone has its apex at the origin, and it has a given axis direction and positive half-angle. If logical is TRUE, then this method finds a quick approximation. The approximate result may lie wholly within or wholly outside the guaranteed bound (obtained with a FALSE argument), but it may not cross from inside to outside. Flags in the turned object indicate whether the cone is the best available, and if this result is inside or outside the best cone.

```
public: logical surface::operator!= (
    surface const& rhs      // surface
    ) const;
```

Tests two surfaces for equality. This method does not guarantee equality for effectively-equal surfaces, but it is guaranteed to determine inequality if the two surfaces are not equal. Use this result for optimization.

```
public: virtual surface& surface::operator*= (
    SPAtransf const&        // transformation
    ) = 0;
```

Transforms the surface by the given transformation.

```
public: virtual logical surface::operator== (
    surface const&          // surface
    ) const;
```

Tests two surfaces for inequality. This method does not guarantee equality for effectively-equal surfaces, but it is guaranteed to determine inequality if the two surfaces are not equal. Use this result for optimization.

```
public: virtual SPApar_pos surface::param (
    SPAposition const&,         // given point
    SPApar_pos const&           // parameter guess
        = * (SPApar_pos* ) NULL_REF
    ) const;
```

Finds the parameter values of a point on a surface given an optional first guess.

```
public: virtual logical surface::parametric () const;
```

Returns TRUE if the surface is a parametric surface; otherwise, it returns FALSE. This reflects the fundamental nature of the surface and it is used by the ACIS kernel to avoid using parameters when they are not necessary. For a surface, the ACIS kernel does not use any of the parameter-based functions, though they are implemented for the sake of those components and applications that prefer parameter-based representations of every surface.

Whether to declare a surface parametric or not depends on the implementation of the point-based evaluations. If the availability of a good approximation to the point’s parameter values makes no significant improvement to the speed of these functions, then the surface is nonparametric; otherwise, it is parametric. A plane is an obvious nonparametric surface; a B-spline is a parametric surface.

```
public: SPApar_dir surface::param_dir (
    SPAunit_vector const&,  // object-space direction
    SPApar_pos const&       // parameter value
    ) const;
```

Finds the direction in parameter space of a given object-space direction on a surface at a given parameter value.

```
public: virtual double
    surface::param_period_u () const;
```

Returns the period of a periodic parametric surface. If the surface is not parametric or is not periodic in the *u*-direction, this method returns 0.

```
public: virtual double
    surface::param_period_v () const;
```

Returns the period of a periodic parametric surface. If the surface is not parametric or is not periodic in the *v*-direction, this method returns 0.

```
public: virtual SPApar_box surface::param_range (
    SPAbox const&           // box
        = * (SPAbox* ) NULL_REF
    ) const;
```

Returns the principle parameter range of a surface. A periodic surface is defined for all parameter values in the periodic direction by reducing the given parameter modulo the period into this principle range. For a surface that is open or nonperiodic in the chosen direction, the surface evaluation functions are defined only for the parameter values in the returned range.

If a box is provided, the parameter range returned may be restricted to a portion of the surface that is guaranteed to contain all portions of the surface that lie within the region of interest. If none is provided and the parameter range in some direction is unbounded, then this method returns an empty interval.

```
public: virtual SPAinterval surface::param_range_u (
    SPAbox const&           // box
        = * (SPAbox* ) NULL_REF
    ) const;
```

Returns the principle parameter range of a surface in the *u*-parameter direction. A periodic surface is defined for all parameter values in the periodic direction by reducing the given parameter modulo the period into this principle range. For a surface that is open or nonperiodic in the chosen direction, the surface evaluation functions are defined only for the parameter values in the returned range.

If a box is provided, the parameter range returned may be restricted to a portion of the surface that is guaranteed to contain all portions of the surface that lie within the region of interest. If none is provided and the parameter range in some direction is unbounded, then this method returns an empty interval.

```
public: virtual SPAinterval surface::param_range_v (
    SPAbox const&           // box
        = * (SPAbox* ) NULL_REF
    ) const;
```

Returns the principle parameter range of a surface in the *v*-parameter direction. A periodic surface is defined for all parameter values in the periodic direction by reducing the given parameter modulo the period into this principle range. For a surface that is open or nonperiodic in the chosen direction, the surface evaluation functions are defined only for the parameter values in the returned range.

If a box is provided, the parameter range returned may be restricted to a portion of the surface that is guaranteed to contain all portions of the surface that lie within the region of interest. If none is provided and the parameter range in some direction is unbounded, then this method returns an empty interval.

```
public: virtual SPApar_vec surface::param_unitvec (
    SPAunit_vector const&,  // object space direction
    SPApar_pos const&       // parameter position
    ) const;
```

Finds the rate of change in the surface parameter corresponding to a unit velocity in a given object-space direction at a given position in parameter space.

```
public: virtual logical surface::periodic_u () const;
```

Determines if the surface is periodic in the *u*-parameter direction (i.e., it is smoothly closed so that faces can run over the seam).

```
public: virtual logical surface::periodic_v () const;
```

Determines if the surface is periodic in the *v*-parameter direction (i.e., it is smoothly closed so that faces can run over the seam).

```
public: virtual logical surface::planar (
    SPAposition&,           // point on the surface
    SPAunit_vector&         // axis direction
    ) const;
```

Reports if a surface is planar.

```
public: virtual double surface::point_cross (
    SPAposition const&,         // point
    SPAunit_vector const&,      // given normal
    SPApar_pos const&           // parameter guess
        = * (SPApar_pos* ) NULL_REF
    ) const;
```

Finds the curvature of a cross-section curve of the surface at the given point. The cross-section curve is given by the intersection of the surface with a plane passing through the given point and with the given normal.

```
public: virtual SPAunit_vector surface::point_normal
(
    SPAposition const&,         // given point
    SPApar_pos const&           // parameter guess
        = * (SPApar_pos* ) NULL_REF
    ) const = 0;
```

Finds the normal to the surface at a point on the surface nearest to the given point.

```
public: virtual SPAunit_vector surface::point_outdir
(
    SPAposition const&,         // given point
    SPApar_pos const&           // parameter guess
        = * (SPApar_pos* ) NULL_REF
    ) const;
```

Finds the outward direction from the surface at a point with the given parameter values. This method usually is the normal, but if the nearest point is a singularity (like the apex of a cone), this method still returns an outward direction. The base class definition returns point_normal, which is used by default on by simple surfaces.

```
public: virtual void surface::point_perp (
    SPAposition const&,             // given point
    SPAposition&,                   // foot
    SPAunit_vector&,                // normal to
                                    // surface
    surf_princurv&,                 // principle
                                    // curvature
    SPApar_pos const&               // param guess
        = * (SPApar_pos* ) NULL_REF,
    SPApar_pos&                     // actual param
        = * (SPApar_pos* ) NULL_REF,
    logical f_weak                  // weak flag
        = FALSE                     // internal use
    ) const = 0;
```

Finds the point on the surface nearest to the given point and optionally, the normal to and the principle curvatures of the surface at that point. If the surface is parametric, this method also returns the parameter values at the found point.

```
public: void surface::point_perp (
    SPAposition const& pos,         // given point
    SPAposition& foot,              // foot
    SPApar_pos const&               // param position
        param_guess                 // param guess
        = * (SPApar_pos* ) NULL_REF,
    SPApar_pos& param_actual        // parameter
        = * (SPApar_pos* ) NULL_REF,
    logical f_weak                  // weak flag
        = FALSE                     // internal use
    ) const;
```

Finds the point on the surface nearest to the given point and optionally, the normal to and the principle curvatures of the surface at that point. If the surface is parametric, this method also returns the parameter values at the found point.

```
public: void surface::point_perp (
    SPAposition const& pos,          // given point
    SPAposition& foot,               // foot
    SPAunit_vector& norm,            // normal to
                                     // surface
    SPApar_pos const&                // param position
        param_guess                  // param guess
        = * (SPApar_pos* ) NULL_REF,
    SPApar_pos& param_actual         // actual param
        = * (SPApar_pos* ) NULL_REF,
    logical f_weak                   // weak flag
        = FALSE                      // internal use
    ) const;
```

Finds the point on the surface nearest to the given point and optionally, the normal to and the principle curvatures of the surface at that point. If the surface is parametric, this method also returns the parameter values at the found point.

```
public: surf_princurv surface::point_prin_curv (
    SPAposition const&,          // point
    SPApar_pos const&            // parameter guess
        = * (SPApar_pos* ) NULL_REF
    ) const;
```

Finds the principle axes of curvature of the surface at a given point and the curvatures in those directions in a structure defined for the purpose.

```
public: virtual void surface::point_prin_curv (
    SPAposition const&,          // position
    SPAunit_vector&,             // 1st axis direction
    double&,                     // 1st dir. curvature
    SPAunit_vector&,             // 2nd axis direction
    double&,                     // 2nd dir. curvature
    SPApar_pos const&            // parameter guess
        = * (SPApar_pos* ) NULL_REF
    ) const = 0;
```

Finds the principle axes of curvature of the surface at a given point and the curvatures in those directions.

```
public: void surface::restore_data ();
```

Restore the data for a surface from a save file.

```
if (restore_version_number >= BNDSUR_VERSION)
    read_interval                   subset u interval
    read_interval                   subset v interval
```

```
public: virtual void surface::save () const = 0;
```

Calls the virtual save method for the particular type of surface.

```
public: void surface::save_data () const;
```

Save the information for the surface to a save file.

```
public: void surface::save_surface () const;
```

Saves the surface if the surface is of an unknown type or NULL. It checks for NULL then calls the save method.

```
public: virtual logical surface::singular_u (
    double                  // u-parameter value
    ) const;
```

Determines whether the surface parameterization is singular at the specified *u*-parameter value. The only singularity recognized is where every value of the nonconstant parameter generates the same object-space point, and these can only occur at the ends of the parameter range.

```
public: virtual logical surface::singular_v (
    double                  // v-parameter value
    ) const;
```

Determines whether the surface parameterization is singular at the specified *v*-parameter value. The only singularity recognized is where every value of the nonconstant parameter generates the same object-space point, and these can only occur at the ends of the parameter range.

```
public: virtual int surface::split_at_kinks_u (
    spline**& pieces,       // split pieces
    double curvature = 0.0  // curvature
    ) const;
```

Divides a surface into separate pieces which are smooth (and therefore suitable for offsetting or blending). The surface is split if the curvature exceeds the minimum curvature argument. If it is closed after this, it is then split into two. The split pieces are stored in the pieces argument. The function returns the count of split pieces. Only implemented for splines and elliptical cones.

```
public: virtual int surface::split_at_kinks_v (
    spline**& pieces,        // split pieces
    double curvature = 0.0   // curvature
    ) const;
```

Divides a surface into separate pieces which are smooth (and therefore suitable for offsetting or blending). The surface is split if the curvature exceeds the minimum curvature argument. If it is closed after this, it is then split into two. The split pieces are stored in the pieces argument. The function returns the count of split pieces. Only implemented for splines and elliptical cones.

```
public: surface* surface::subset (
    SPApar_box const&        // parameter box
    ) const;
```

Constructs a subset copy within the given parameter box.

```
public: logical surface::subsetted () const;
```

Determines whether the surface has a significant subset range.

```
public: logical surface::subsetted_u () const;
```

Determines whether the surface has a significant subset range in the *u* direction.

```
public: logical surface::subsetted_v () const;
```

Determines whether the surface has a significant subset range in the *v* direction.

```
public: SPApar_box surface::subset_box () const;
```

Returns a subset of the surface.

```
public: surface* surface::subset_u (
    SPAinterval const&      // u interval
    ) const;
```

Constructs a subset copy within the given *u* interval.

```
public: SPAinterval surface::subset_u_interval ()
const;
```

Returns a subset interval.

```
public: surface* surface::subset_v (
    SPAinterval const&      // v interval
    ) const;
```

Constructs a subset copy within the given *v* interval.

```
public: SPAinterval surface::subset_v_interval ()
const;
```

Returns a subset interval.

```
public: logical surface::test_point (
    SPAposition const& pos,         // point
    SPApar_pos const& uv_guess      // param guess
        = * (SPApar_pos* ) NULL_REF,
    SPApar_pos& uv_actual           // actual param
        = * (SPApar_pos* ) NULL_REF
    ) const;
```

Determines if a point lies on the surface to the system precision.

```
public: virtual logical surface::test_point_tol (
    SPAposition const&,             // point
    double                          // tolerance
        = 0,                        // (defaults to
                                    // SPAresabs)
    SPApar_pos const&               // param guess
        = * (SPApar_pos* ) NULL_REF,
    SPApar_pos&                     // actual param
        = * (SPApar_pos* ) NULL_REF
    ) const = 0;
```

Determines whether a point lies on the surface to the given tolerance.

```
public: virtual int surface::type () const = 0;
```

Returns the type of surface.

```
public: virtual char const*
    surface::type_name () const = 0;
```

Returns the string “surface”.

```
public: virtual logical surface::undef () const;
```

Determines whether a surface is defined or undefined. A NULL or generic surface are always undefined; other surfaces depend on their contents.

```
public: logical surface::undefined () const;
```

Determines whether a surface is defined or undefined. A NULL or generic surface are always undefined; other surfaces depend on their contents.

```
public: void surface::unlimit ();
```

Removes the subsetting from this surface.

```
public: void surface::unlimit_u ();
```

Removes the subsetting from this surface in the *u* direction.

```
public: void surface::unlimit_v ();
```

Removes the subsetting from this surface in the *v* direction.

```
public: surface* surface::unsubset () const;
```

Constructs a copy of the unbounded surface underlying this one.

```
public: surface* surface::unsubset_u () const;
```

Constructs a copy of the unbounded surface underlying this one in the *u* direction

---

```
public: surface* surface::unsubset_v () const;
```

Constructs a copy of the unbounded surface underlying this one in the *v* direction.

```
public: virtual curve* surface::u_param_line (
    double                  // v parameter
    ) const;
```

Constructs a parameter line on the surface. A *u*-parameter line runs in the direction of increasing *u* parameter, at constant *v.* The parameterization int he nonconstant direction matches that of the surface, and it has the range obtained by the use of param_range_u. If the supplied constant parameter value is outside the valid range for the surface, or if it is at a singularity, this method returns NULL.

The new curve is constructed in free storage, so it is the responsibility of the caller to ensure that it is correctly deleted.

```
public: virtual curve* surface::v_param_line (
    double                  // u parameter
    ) const;
```

Constructs a parameter line on the surface. A *v*-parameter line runs in the direction of increasing *v* parameter, at constant *u*. The parameterization int he nonconstant direction matches that of the surface, and it has the range obtained by the use of param_range_v. If the supplied constant parameter value is outside the valid range for the surface, or if it is at a singularity, this method returns NULL.

The new curve is constructed in free storage, so it is the responsibility of the caller to ensure that it is correctly deleted.

Internal Use: full_size

Related Fncs: restore_surface, surf_deriv_to_curv

surface_law_data

Class: Laws, Geometric Analysis, SAT Save and Restore

Purpose: Creates a wrapper to an ACIS surface class.

Derivation: surface_law_data : base_surface_law_data : law_data : ACIS_OBJECT : –

SAT Identifier: “SURF#”

Filename: kern/kernel/kernutil/law/law.hxx

Description: This is a law data class that holds a pointer to a surface.

Limitations: None

References:

KERN	surface
BASE	SPAinterval, SPApar_pos, SPAposition

Data:

`protected SPAinterval u_domain;`
This holds the *u* parameter range of the given surface.

`protected SPAinterval v_domain;`
This holds the *v* parameter range of the given surface.

`protected int *which_cached;`
This holds the time tags.

`protected int point_level;`
This holds the size of tvalue.

`protected SPApar_pos *tvalue;`
This holds the parameter values.

`protected SPAposition *cached_f;`
This holds the positions.

`protected surface *acis_surface;`
This holds a pointer to the ACIS surface class.

Constructor:

```
public: surface_law_data::surface_law_data (
    surface const& in_acis_surface, // surface
    SPAinterval const& in_u_domain, // u parameter
                                    // range
    SPAinterval const& in_v_domain  // v parameter
                                    // range
    );
```

C++ constructor, creating a sruface_law_data which is a wrapper for the ACIS surface.

```
public: surface_law_data::surface_law_data (
    surface const& in_acis_surface  // surface
    );
```

C++ constructor, creating a sruface_law_data which is a wrapper for the ACIS surface.

Destructor:

```
public: surface_law_data::~surface_law_data ();
```

Applications are required to call this destructor for their law data types.

Methods:

```
public: SPAposition surface_law_data::bs3_eval (
    SPApar_pos const& in_par_pos// uv parameters
    ) const;
```

Returns the position of the *u*,*v* parameters on the spline approximating surface.

```
public: virtual law_data*
    surface_law_data::deep_copy (
    base_pointer_map* pm    // list of items within
        = NULL              // the entity that are
                            // already deep copied
    ) const;
```

Creates a copy of an item that does not share any data with the original. Allocates new storage for all member data and any pointers. Returns a pointer to the copied item.

```
public: void surface_law_data::eval (
    SPApar_pos& uv,          // parameter to evaluate
    SPAposition& pos,        // output surface
                             // position
    SPAvector* dpos,         // array of 1st
                             // derivatives
    SPAvector* ddpos         // array of second
                             // derivatives
    );
```

This takes in a *uv* parameter value and returns the corresponding *xyz* position on the surface; a vector array, which holds the derivative with respect to *u* and the derivative with respect to *v*; and an array with three vectors, which correspond to the second derivative with respect to *u*, the derivative with respect to *u* and then to *v*, and the second derivative with respect to *v*.

```
public: double
    surface_law_data::eval_gaussian_curvature (
    SPApar_pos const& in_par_pos// parameter position
    ) const;
```

Finds the Gaussian curvature at the given parameter value on the curve.

```
public: double surface_law_data::eval_max_curvature (
    SPApar_pos const& in_par_pos// parameter position
    ) const;
```

Finds the maximum curvature at the given parameter value on the curve.

```
public: double
    surface_law_data::eval_mean_curvature (
    SPApar_pos const& in_par_pos// parameter position
    ) const;
```

Finds the mean curvature at the given parameter value on the curve.

```
public: double surface_law_data::eval_min_curvature (
    SPApar_pos const& in_par_pos// parameter position
    ) const;
```

Finds the minimum curvature at the given parameter value on the curve.

```
protected: void surface_law_data::init (
    surface const& in_acis_surface, // surface
    SPAinterval const& in_u_domain, // u parameter
                                    // range
    SPAinterval const& in_v_domain  // v parameter
                                    // range
    );
```

Initializes this object.

```
public: SPApar_pos surface_law_data::point_perp (
    SPAposition in_point    // given position
    );
```

Finds the parameter position on the surface perpendicular to the given position outside of the surface.

```
public: SPApar_pos surface_law_data::point_perp (
    SPAposition in_point,   // given position
    SPApar_pos in_par_pos   // given par pos
    );
```

Finds the parameter position on the surface perpendicular to the given position outside of the surface.

```
public: virtual void surface_law_data::save ();
```

Calls the specific surface save method, and saves the *u* domain interval and *v* domain interval.

```
public: void surface_law_data::set_levels (
    int in_point_level      // number of tvalue
        = 4,
    int in_derivative_level // number of deriv
        = 2
    );
```

Establishes the number of positions stored in tvalue in preparation for starting over. In addition, it clears out cached arrays for the positions and their derivatives.

```
public: char const* surface_law_data::symbol (
    law_symbol_type type    // type of law symbol
    );
```

Returns the string that represents this law class's symbol. The symbol is used for parsing the law and for saving and restoring law-based geometry. For a law to be saved and restored, it must have or inherit this method.

The default law symbol for this class is SURF.

```
public: logical surface_law_data::term_domain (
    int which,              // term to bound
    SPAinterval& answer     // bounds for term
    );
```

Establishes the domain of a given term in the law.

Internal Use: grid

Related Fncs: restore_law, restore_law_data, save_law

surf_int_cur

Class: Construction Geometry, SAT Save and Restore

Purpose: Represents spline curves on a surface within the given fit tolerance.

Derivation: surf_int_cur : int_cur : subtrans_object : subtype_object : ACIS_OBJECT : –

SAT Identifier: "surfcur"

Filename: kern/kernel/kerngeom/intcur/surf_int.hxx

Description: This class, derived from int_cur, represents spline curves on a surface within a given fit tolerance.

Limitations: None

References: None

Data: None

Constructor:

```
public: surf_int_cur::surf_int_cur (
    bs3_curve,                    // spline curve
    double,                       // fit tolerance
    surface const&,               // 1st surface
                                  // for curve
    surface const&,               // 2nd surface
                                  // for curve
    bs2_curve,                    // 1st curve
                                  // param space
    bs2_curve,                    // 2nd curve
                                  // param space
    logical                       // surface for
        = TRUE,                   // true curve
    const SPAinterval&            // safe range
        = * (SPAinterval*) NULL_REF// for curve
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

```
public: surf_int_cur::surf_int_cur (
    bs3_curve,                     // spline curve
    double,                        // fit tolerance
    surface const&                 // surface where
        = * (surface* ) NULL_REF,  // curve lies
    bs2_curve                      // curve in param
        = NULL,                    // space
    logical                        // surface for
        = TRUE,                    // true curve
    const SPAinterval&             // safe range
        = * (SPAinterval*) NULL_REF // for curve
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

```
public: surf_int_cur::surf_int_cur (
    const surf_int_cur&     // input curve
    );
```

C++ copy constructor requests memory for this object and populates it with the data from the object supplied as an argument.

Destructor:

None

Methods:

```
public: virtual int_cur* surf_int_cur::deep_copy (
    pointer_map* pm          // list of items within
        = NULL               // the entity that are
                             // already deep copied
    ) const;
```

Creates a copy of an item that does not share any data with the original. Allocates new storage for all member data and any pointers. Returns a pointer to the copied item.

```
public: static int surf_int_cur::id ();
```

Returns the ID for the surf_int_cur list.

```
private: void surf_int_cur::restore_data ();
```

Restores the information for the surf_int_cur from a save file.

int_cur::restore_data	Generic curve data
if (restore_version_number >= PARCUR_VERSION)	
read_logical	Either “surf2” or “surf1”

```
public: virtual void
    surf_int_cur::save_data () const;
```

Save the information for the surf_int_cur to a save file.

```
public: virtual int surf_int_cur::type () const;
```

Returns the type of surf_int_cur.

```
public: virtual char const*
    surf_int_cur::type_name () const;
```

Returns the string “surfcur”.

Internal Use: full_size

Related Fncs:

restore_surf_int_cur

surf_normcone

Class: Construction Geometry, Mathematics

Purpose: Provides a return value for normal_cone which returns a cone bounding the surface normal.

Derivation: surf_normcone : ACIS_OBJECT : –

SAT Identifier: None

Filename: kern/kernel/kerngeom/surface/surdef.hxx

Description: Defines a bound on the surface normal. It is used in intersection code to eliminate the possibility of tangencies and common normals.

Limitations: None

References:

by KERN	mesh_tree
BASE	SPAunit_vector

Data:

```
public double angle;
```
Positive half angle defining the bounding cone.

```
public logical approx;
```
TRUE if this is only an approximation to the best cone available.

```
public logical oversize;
```
If approx is TRUE, this flag is TRUE if this cone is entirely outside the best available cone and FALSE if it is inside.

```
public surface_vardir vardir;
```
Classifies whether the normal direction varies more in the *u*-parameter direction, more in the *v*-parameter direction, or in neither. This item is to be used only as a hint.

```
public SPAunit_vector axis;
```
Axis direction for the cone. The cone is deemed to have its apex at the origin.

Constructor:

```
public: surf_normcone::surf_normcone (
    SPAunit_vector const& ax,   // axis direction
    double ang,             // positive half angle
    surface_vardir vdir,    // variation direction
    logical app,            // approx results OK?
    logical over            // inside or outside?
    );
```

C++ constructor, creating a surf_normcore using the specified parameters.

Destructor:

None

Methods:

None

Related Fncs:

surf_deriv_to_curv

surf_princurv

Class: Construction Geometry, Mathematics

Purpose: Provides the return value for the principle curvature functions, returning two directions and two curvatures for a surface.

Derivation: surf_princurv : ACIS_OBJECT : –

SAT Identifier: None

Filename: kern/kernel/kerngeom/surface/surdef.hxx

Description: This class provides the return value for the principle curvature functions, returning two directions and two curvatures for a surface.

Limitations: None

References: BASE SPAunit_vector

Data:

```
public double curv1;
```
The first curvature.

```
public double curv2;
```
The second curvature.

```
public SPAunit_vector dir1;
```
First direction vector.

```
public SPAunit_vector dir2;
```
The second direction vector.

Constructor:

```
public: surf_princurv::surf_princurv ();
```

C++ allocation constructor requests memory for this object but does not populate it.

```
public: surf_princurv::surf_princurv (
    SPAunit_vector const& d1,// 1st vector direction
    double c1,               // first curvature
    SPAunit_vector const& d2,// 2nd vector direction
    double c2                // second curvature
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

Destructor:

None

Methods:

None

Related Fncs:

surf_deriv_to_curv

surf_surf_int

Class: Intersectors

Purpose: Represents the intersection of two face surfaces and returns zero or more curves.

Derivation: surf_surf_int : ACIS_OBJECT : –

SAT Identifier: None

Filename: kern/kernel/kernint/intsfsf/sfsfint.hxx

Description: This class holds the details of the intersection of two face surfaces and returns zero or more curves. Every edge of each face is assumed to have been intersected with the other surface, so the intersection points may be used to assist (for example, if the surfaces are parametric).

Limitations: None

References: KERN curve, pcurve, surf_surf_term, surface

Data:

```
public curve *cur;
```

It indicates the face-face coincidence, and it may be NULL. In this case, all face-body relationships are either surf_symmetric or surf_antisymmetric, and this is the only CURVE_LIST record in the list.

```
public double *split_param;
```
An array of parameter values flagging bounded regions of the curve where it lies outside the region of interest. Each value is in a typical parameter value within the portion outside the box. If no box is specified, or if the intersection curve lies wholly within the box, or if it is unbounded but only enters the box in one interval, then this pointer is NULL; otherwise it must point to an array on the heap.

```
public int nsplit;
```
The number of values in the array split_param. This is 0 if split_param is NULL.

```
public pcurve *pcur1;
```
The first pcurve, it provides the parametric-space intersection curve with respect to the intersection surfaces, if they are parametric. It may be NULL even if cur is not NULL.

```
public pcurve *pcur2;
```
The second pcurve, it provides the parametric-space intersection curve with respect to the intersection surfaces, if they are parametric. It may be NULL even if cur is not NULL.

```
public double end_param;
```
The parameter value of end_point, which is meaningless if the end_point is NULL.

```
public double start_param;
```
The parameter value of start_point, which is meaningless if the start_point is NULL.

```
public surf_int_type int_type;
```
The classification of the intersection type (normal, tangent, or antitangent).

```
public surf_int_type left_int_type[2];
```
Intersection type (with respect to other face) of the portions of each face to the left of the intersection curve. Only used for mesh surface intersections (otherwise the single int_type above is sufficient). Set to int_unknown if not used.

```
public surf_int_type right_int_type[2];
```
Intersection type (with respect to other face) of the portions of each face to the left of the intersection curve. Only used for mesh surface intersections (otherwise the single int_type above is sufficient). Set to int_unknown if not used.

```
public surf_surf_int *next;
```
For chaining surface-surface intersections.

```
public surf_surf_rel aux_left_rel[ 2 ];
```
For each of the surfaces, it specifies the relationship on the left side of the intersection curve to the auxiliary surface. Because this is always a clean “inside” or “outside,” the right relationship is always the converse, so it does not need to be recorded.

```
public surf_surf_rel left_surf_rel[ 2 ];
```
The relationship (with respect to the other face) of the portions of each face to the left of the intersection curve.

```
public surf_surf_rel right_surf_rel[ 2 ];
```
The relationship (with respect to the other face) of the portions of each face to the right of the intersection curve.

```
public surf_surf_term *end_term;
```
The terminator point at the end of the curve. It is NULL if the curve is not bounded at the end.

```
public surf_surf_term *start_term;
```
The terminator point at the start of the curve. It is NULL if the curve is not bounded at the start.

```
public surface *aux_surf;
```
Normally NULL, this points to a surface containing the intersection curve and is roughly perpendicular to the subject surfaces if they are tangential or near tangential, which allows for clean intersections with the edges of the faces.

Constructor:

```
public: surf_surf_int::surf_surf_int (
    curve*,                 // curve
    surf_surf_int*          // surf-surf intersection
        = NULL,
    surf_surf_term*         // surf-surf termination
        = NULL,
    surf_surf_term*         // surf-surf termination
        = NULL
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

The default type is normal, and the relationships are suitable for the conventional curve direction being the cross products of the surface normals in the given order.

```
public: surf_surf_int::surf_surf_int (
    SPAposition const&,      // position
    surf_surf_int*           // next surf-surf
        = NULL               // intersection
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

Constructor for an intersection curve representing an isolated point. The default type is normal and the relationships are set to unknown.

Destructor:

```
public: surf_surf_int::~surf_surf_int ();
```

C++ destructor, deleting a surf_surf_int.

Methods:

```
public: void surf_surf_int::debug (
    FILE*                    // file name
        = debug_file_ptr
    ) const;
```

Writes debug information about surf_surf_int to the printer or to the specified file.

Related Fncs:

None

sweep_spl_sur

Class: Sweeping, Construction Geometry, SAT Save and Restore

Purpose: Defines the perpendicular sweep of a planar profile curve along a path curve.

Derivation: sweep_spl_sur : spl_sur : subtrans_object : subtype_object : ACIS_OBJECT : –

SAT Identifier: “sweepsur”

Filename: kern/kernel/sg_husk/sweep/swp_spl.hxx

Description: This class defines the perpendicular sweep of a planar profile curve along a path curve. The start of the path is in the plane of the shape curve.

The evaluator for the “sweep” surface type has been redone to remove the restriction on the profile and path curves.

The new equation S*(u,v)* for the sweep surface is defined below:

```
S (u,v) = k (u) *M (v)
```

where

k (u) = w (u) * transpose (M (start v))
p (u) = the position of the profile at u
w (u) = p (u) – (the start of the sweep path)

M (v) = the 3 x 3 matrix
with rows { mx (v), my (v), mz (v) }

mx (v) = rail_law (v)
mz (v) = normalized (sweep path tangent at v)
my (v) = mz (v) X mx (v)

M is a frame at v on the sweep path, and k is the vector in the starting frame’s coordinate system that points from the start of the sweep path to the profile at u.

When we consider scale, the equation becomes

```
S (u,v) = k (u) *s (v) * M (v)
```

where

s (v) = the 3 x 3 matrix
with diagonal equal to { xs (v), ys (v), zs (v) }

and where

“_s” is the scale in the frame’s “_” direction

When we consider draft, the equation becomes

```
S (u,v) = k (u,v) * s (v) * M (v)
```

where

k (u,v) = p (u,v) – (the start of the sweep path)

and where

p (u,v) is the u position of the profile offset by the draft_law (v)

The profile must be planar if draft is used.

Limitations: None

References:

KERN	curve
BASE	SPAinterval, SPAmatrix, SPAposition, SPAunit_vector
LAW	law

Data:

`protected bs3_curve path_bs;`
The spline representation of a path curve.

`protected bs3_curve profile_bs;`
The spline representation of a profile curve.

`protected curve *path_curve;`
A copy of the path curve.

`protected curve *profile_curve;`
A copy of the profile curve.

`protected SPAinterval draft_domain;`
The draft range.

`protected law *draft_law;`
This is a law that represents the draft angle as a function of the length of the wire, starting at wire length equal to zero for the beginning of the wire.

`protected law *rail_law;`
This is a law that represents a vector orientation as a function of the length of the wire, starting at wire length equal to zero for the beginning of the wire.

`protected law *scale_law;`
This is a law that represents *xyz* scaling as a function of the length of the wire, starting at wire length equal to zero for the beginning of the wire.

`protected logical is_path_normal;`
The variables to switch on the path type.

`protected logical sweep_nor;`
A logical variable that is TRUE if the shape is in the plane formed by local_x and local_y; otherwise, it is FALSE.

`protected SPAmatrix start_frame;`
Start of frame used by the path and rail law. *x* is the rail, *z* is the path direction at the start. $y = z \times x$.

```
protected SPAmatrix start_frame_t;
```
The transpose of the start_frame.

```
protected SPAposition path_start;
```
Start the Frenet Frame, which is defined by the path and the rail curves.

```
protected SPAunit_vector profile_nor;
```
The normal vector to the profile.

Constructor:

```
protected: sweep_spl_sur::sweep_spl_sur ();
```

C++ allocation constructor requests memory for this object but does not populate it.

```
public: sweep_spl_sur::sweep_spl_sur (
    curve const&,               // shape curve
    curve const&,               // path curve
    law*,                       // rail law
    law*,                       // draft law
    law*,                       // scale law
    SPAinterval const&,         // shape range
    SPAinterval const&,         // path range
    logical,                    // shape perp. to
                                // path?
    bs3_curve help_profile_bs   // rail curve
        = (bs3_curve) NULL,
    bs3_curve help_path_bs      // rail curve
        = (bs3_curve) NULL,
    SPAunit_vector const&       // unit vector
        in_profile_nor          // normal in shape
        = * (SPAunit_vector const* ) NULL_REF
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

This sweep surface is formed by sweeping the shape along the path curve, with an optional rail curve to specify the twist of the shape curve when it is swept along the path. If the path curve is normal, the path normal must not be NULL. If the path is not planar, a rail curve must be specified.

Destructor:

```
protected: virtual sweep_spl_sur::~sweep_spl_sur ();
```

C++ destructor, deleting a sweep_spl_sur.

Methods:

```
public: int sweep_spl_sur::accurate_derivs (
    SPApar_box const&           // range for derivs
        = * (SPApar_box*) NULL_REF
    ) const;
```

Calculates the derivatives for the surface.

```
public: void
sweep_spl_sur::check_for_approx () const;
```

This function should be used after creating a sweep_spl_sur, to ensure the surface will be valid for downstream operations.

```
protected: virtual subtrans_object*
    sweep_spl_sur::copy () const;
```

Constructs a duplicate sweep_spl_sur in free storage of this object, with a zero use count.

```
protected: void
    sweep_spl_sur::curve_add_discontinuities ();
```

Calculates discontinuity information from the generating curves and adds it to the sweep surface.

```
protected: virtual void sweep_spl_sur::debug (
    char const*,            // class-identifying line
    logical,                // brief output?
    FILE*                   // file name
    ) const;
```

Prints out a class-specific identifying line to standard output or to the specified file.

```
protected: void sweep_spl_sur::debug_data (
    char const*,            // class-identifying line
    logical,                // brief output?
    FILE*                   // file name
    ) const;
```

Prints out the details. The debug_data derived class can call its parent's version first, to put out the common data. If the derived class has no additional data it need not define its own version of debug_data and use its parent's instead. A string argument provides the introduction to each displayed line after the first, and a logical sets brief output (normally removing detailed subsidiary curve and surface definitions).

```
public: virtual spl_sur* sweep_spl_sur::deep_copy (
    pointer_map* pm          // list of items within
        = NULL               // the entity that are
                             // already deep copied
    ) const;
```

Creates a copy of an item that does not share any data with the original. Allocates new storage for all member data and any pointers. Returns a pointer to the copied item.

```
protected: virtual int sweep_spl_sur::evaluate (
    SPApar_pos const& uv,        // parameter
    SPAposition& xyz,            // point on surface
                                 // at given parameter
    SPAvector** derivs,          // array of points
                                 // to vector
    int number_of_derivs,        // no. of derivatives
                                 // required
    evaluate_surface_quadrant    // location to
        quadrant                 // evaluate
    ) const;
```

Calculates derivatives of any order up to the number requested, and stores them in vectors provided by the user. It returns the number of derivatives it was able to calculate. (In most circumstances, this will be the number of derivatives requested.) Lower order derivatives will be evaluated directly and with reasonable accuracy; higher derivatives will be automatically calculated by finite differencing. The accuracy of the higher derivatives decreases with the order of the derivative, while the computing cost increases.

Any of the pointers may be NULL, in which case the corresponding derivatives will not be returned. Otherwise they must point to arrays large enough to contain all the derivatives of that order; i.e., 2 for the first derivatives, 3 for the second, etc.

```
protected: virtual double sweep_spl_sur::eval_cross (
    SPApar_pos const&,       // parametric position
    SPAunit_vector const&    // curve direction
    ) const;
```

Finds the curvature of a cross-section curve of the surface at the point on the surface with the specified parameter values. The cross-section curve is defined as the intersection of the surface with a plane passing through the point on the surface and normal to the specified direction, which must lie in the surface.

```
protected: virtual SPAunit_vector
    sweep_spl_sur::eval_normal (
    SPApar_pos const&        // parametric position
    ) const;
```

Finds the normal to the surface at the point with the specified parameter values.

```
protected: virtual void
    sweep_spl_sur::eval_prin_curv (
    SPApar_pos const&,       // parametric position
    SPAunit_vector&,         // first curvature axis
                             // direction
    double&,                 // curvature in first
                             // direction
    SPAunit_vector&,         // second curvature axis
                             // direction
    double&                  // curvature in the 2nd
                             // direction
    ) const;
```

Finds the principal axes of curvature of the surface at a point with the specified parameter values and curvatures in those directions.

```
public: virtual law*
    sweep_spl_sur::get_draft () const;
```

Returns the draft law for the sweep.

```
public: virtual curve*
    sweep_spl_sur::get_path () const;
```

Returns the sweep path curve.

```
public: virtual sweep_path_type
    sweep_spl_sur::get_path_type () const;
```

Returns the sweep path type.

```
public: virtual curve* sweep_spl_sur::get_profile (
    double param            // parameter location
    ) const;
```

Evaluates the curve at the given parameter location.

```
public: virtual law*
    sweep_spl_sur::get_rail () const;
```

Returns a pointer to the sweep rail law.

```
public: virtual law*
    sweep_spl_sur::get_scale () const;
```

Returns a pointer to the sweep scale law.

```
public: logical sweep_spl_sur::helix (
    spline const& owner,    // starting surface
    SPAvector& axis         // axis start direction
        = * (SPAvector* )NULL_REF,
    SPAposition& root       // axis start position
        = * (SPAposition* )NULL_REF,
    double& pitch           // thread pitch
        = * (double* )NULL_REF,
    double& radius          // radius of helix
        = * (double* )NULL_REF,
    logical& right_handed   // TRUE is right handed
        = * (logical* )NULL_REF
    ) const;
```

Sweeps a helical surface.

```
public: static int sweep_spl_sur::id ();
```

Returns the ID for the sweep_spl_sur list.

```
protected: virtual void sweep_spl_sur::make_approx (
    double fit,             // fit tolerance
    const spline& spl       // pointer to output
        = * (spline*) NULL_REF,// spline approx.
    logical force           // flag for forcing
        = FALSE
    ) const;
```

Makes or remakes an approximation of the surface, within the given tolerance.

```
protected: virtual void sweep_spl_sur::operator*= (
    SPAtransf const&        // transformation
    );
```

Transforms the sweep by the specified transform.

```
protected: logical sweep_spl_sur::operator== (
    subtype_object const&   // sweep_spl_sur
    ) const;
```

Tests for equality. This does not guarantee to find all effectively equal surfaces, but it does guarantee that different surfaces are correctly identified as different.

```
protected: virtual SPApar_pos sweep_spl_sur::param (
    SPAposition const&,         // given point
    SPApar_pos const&           // parameter guess
        = * (SPApar_pos* ) NULL_REF
    ) const;
```

Finds the parameter values of a point on the surface.

```
protected: virtual SPApar_vec
    sweep_spl_sur::param_unitvec (
    SPAunit_vector const&,  // direction
    SPApar_pos const&       // parametric position
    ) const;
```

Finds the change in the surface parameter corresponding to a unit offset in a given direction at a given *uv*, the direction lying in the surface.

```
public: virtual logical
    sweep_spl_sur::planar_profile () const;
```

Checks whether the sweep profile is planar.

```
public: virtual logical
    sweep_spl_sur::planar_profile (
        SPAunit_vector& norm // normal to profile
        ) const;
```

Checks whether the sweep profile is planar and returns a vector normal to the profile.

```
protected: virtual SPAunit_vector
    sweep_spl_sur::point_normal (
    SPAposition const&,          // point
    SPApar_pos const&            // parameter guess
        = * (SPApar_pos* ) NULL_REF
    ) const;
```

Finds the normal to the surface at a point on the surface nearest to the specified point.

```
protected: virtual void
    sweep_spl_sur::point_prin_curv (
    SPAposition const&,          // given point
    SPAunit_vector&,             // 1st axis direction
    double&,                     // curvature in first
                                 // direction
    SPAunit_vector&,             // 2nd axis direction
    double&,                     // curvature in 2nd
                                 // direction
    SPApar_pos const&            // parametric guess
        = * (SPApar_pos* ) NULL_REF// supplied
    ) const;
```

Finds the principle axes of curvature of the surface at a specified point and the curvatures in those directions.

```
public: curve const& sweep_spl_sur::profile () const;
```

Returns the sweep profile curve.

```
protected: void sweep_spl_sur::restore_data ();
```

Restore the data for a sweep_spl_sur from a save file.

```
read_logical                          // path normal, ”angled” or
                                      ”normal”
restore_curve                         // profile curve
restore_curve                         // path curve
read_logical                          // sweep normal, ”angled” or
                                      ”normal”
read_vector                           // profile normal
read_position                         // path start position
read_vector                           // x–vector
read_vector                           // y–vector
read_vector                           // z–vector
if ( restore_version_number < APPROX_SUMMARY_VERSION )
     read_real                        // low point of u interval
     read_real                        // high point of u interval
     read_real                        // low point of v interval
     read_real                        // high point of v interval
else                                  // else
     read_real                        // low point
     read_real                        // high point
read_real                             // start draft distance
read_real                             // end draft distance
if(restore_version_number >= LAW_VERSION)
     restore_law                      // rail law
     restore_law                      // draft law
     restore_law                      // scale law
restore_common_data                   // common surface data
```

```
public: virtual void sweep_spl_sur::save () const;
```

Saves an approximation of the surface, or calls the subtype object’s save method.

```
public: virtual void
    sweep_spl_sur::save_data () const;
```

Save the information for the sweep_spl_sur to a save file.

```
protected: virtual void sweep_spl_sur::shift_u (
    double                  // shift value
    );
```

Adjusts the spline surface to have a parameter range increased by the shift value, which may be negative. This method is only used to move portions of a periodic surface by integral multiples of the period, so shift_v is never used.

```
protected: virtual void sweep_spl_sur::shift_v (
    double                  // shift value
    );
```

This method is never used because shift_u and shift_v are designed to move portions of a periodic surface by integral multiples of the period. Refer to shift_u for more information.

```
protected: virtual void sweep_spl_sur::split_u (
    double,                 // u-parameter value
    spl_sur* [ 2 ]          // returned split
                            // surfaces
    );
```

Divides a surface into two pieces at the specified parameter value. It the split occurs at the end of the parameter range, the spl_sur returns as the appropriate half (in increasing parameter order), and the other is NULL; otherwise a new spl_sur is used for one part, and the old one is modified for the other part.

```
protected: virtual void sweep_spl_sur::split_v (
    double,                 // v-parameter value
    spl_sur* [ 2 ]          // returned split
                            // surfaces
    );
```

Divides a surface into two pieces at the specified parameter value. It the split occurs at the end of the parameter range, the spl_sur returns as the appropriate half (in increasing parameter order), and the other is NULL; otherwise a new spl_sur is used for one part, and the old one is modified for the other part.

```
protected: virtual logical
    sweep_spl_sur::test_point_tol (
    SPAposition const&,             // given point
    double,                         // tolerance
    SPApar_pos const&               // param guess
        = * (SPApar_pos* ) NULL_REF,
    SPApar_pos&                     // actual param
        = * (SPApar_pos* ) NULL_REF
    ) const;
```

Tests whether a point lies on the surface.

```
public: virtual int sweep_spl_sur::type () const;
```

Returns the type of sweep_spl_sur.

```
public: virtual char const*
    sweep_spl_sur::type_name () const;
```

Returns a string “sweepsur”.

Internal Use: full_size

Related Fncs: is_path_planar, restore_sweep_spl_sur