Chapter 34.
Classes Pa thru Pz

Topic: Ignore

par_int_cur

Class: Construction Geometry, SAT Save and Restore

Purpose: Represents an exact spline curve in the parameter space of a surface.

Derivation: par_int_cur : int_cur : subtrans_object : subtype_object : ACIS_OBJECT : –

SAT Identifier: “parcur”

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

Description: This class represents a 3D spline curve as a 2D parameter curve on a spline surface. The spline surface is used to map the 2D parameter curve from (*u*,*v*) parameter space into (*x*,*y*,*z*) euclidean space. The approximate parameter curve is everywhere within the fit tolerance of the exact parameter curve.

Limitations: None

References: None

Data:

None

Constructor:

```
public: par_int_cur::par_int_cur (
    bs3_curve,                        // spline curve
    double,                           // fit tolerance
    surface const&,                   // surface where
                                      // curve lies
    bs2_curve,                        // surface curve
                                      // in parameter
                                      // space
    logical                           // surface
        = TRUE,
    const discontinuity_info&         // discontinuity
        = * (discontinuity_info*) NULL_REF
    );
```

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

Constructs a general parameter curve, given an exact bs2_curve and an approximate bs3_curve. If logical is TRUE, the surface is made the first surface of the int_cur; if it is FALSE, the surface is made the second surface of the int_cur.

```
public: par_int_cur::par_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 on
                                        // surface
    bs2_curve,                          // 2nd curve on
                                        // surface
    logical                             // surface
                                        // surface
        = TRUE,
    const discontinuity_info&           // discontinuity
        = * (discontinuity_info*) NULL_REF
    );
```

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

Constructs a general parameter curve, with and additional surface and parameter curve. If logical is TRUE, the surface is made the first surface of the int_cur; if it is FALSE, the surface is made the second surface of the int_cur.

```
public: par_int_cur::par_int_cur (
    spline const&,                  // surface where
                                    // curve lies
    par_int_cur_dir,                // select u or v
                                    // direction
    double,                         // constant u or
                                    // v parameter
    logical                         // surface
        = TRUE,
    const discontinuity_info&       // discontinuity
        = * (discontinuity_info*) NULL_REF
    );
```

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

Constructs a constant parameter curve. If logical is TRUE, the surface is made the first surface of the int_cur; if it is FALSE, the surface is made the second surface of the int_cur.

```
public: par_int_cur::par_int_cur (
    const par_int_cur&      // par_int_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:

```
protected: virtual int par_int_cur::accurate_derivs (
    SPAinterval const&              // part of curve
        = * (SPAinterval*)NULL_REF  // to evaluate
    ) const;
```

Returns the number of derivatives which evaluate() can find “accurately” (and fairly directly), rather than by finite differencing, over the given portion of the curve. If there is no limit to the number of accurate derivatives, returns the value ALL_CURVE_DERIVATIVES.

```
public: virtual void
    par_int_cur::calculate_disc_info ();
```

Calculates the discontinuity information if it was never stored. This function is intended to support restore of old versions of int_curs.

```
protected: virtual check_status_list*
    par_int_cur::check (
    const check_fix& input          // flags for
        = * (const check_fix*)      // the 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 curve, 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 curve 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.

```
protected: virtual void par_int_cur::debug (
    char const*,            // title line
    logical,                // brief: sets amount
                            // of debug returned
    FILE*                   // debug file
    ) const;
```

Outputs a title line and the details of the class for inspection to standard output or to the specified file.

```
protected: void par_int_cur::debug_data (
    char const*,            // title line
    logical,                // brief: sets amount
                            // of debug returned
    FILE*                   // debug file
    ) const;
```

Debug printout. As for save and restore we split the operation into two parts; the virtual function “debug” prints a class–specific identifying line, then calls the ordinary function “debug_data” to put out the details. It is done this way so that a derived class’ debug_data can call its parent’s version first, to put out the common data. Indeed, 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 int_cur* par_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.

```
protected: virtual void par_int_cur::eval (
    double,                  // parameter
    SPAposition&,            // position
    SPAvector&,              // 1st derivative
    SPAvector&,              // 2nd derivative
    logical                  // needs precise eval?
    ) const;
```

Finds the position and first and second derivative on a curve at the given parameter value.

```
protected: virtual int par_int_cur::evaluate (
    double,                       // parameter
    SPAposition&,                 // point on curve at
                                  // given parameter
    SPAvector**                   // array of pointers
        = NULL,                   // to vectors
    int                           // no. of derivatives
        = 0,                      // required
    evaluate_curve_side           // eval. location:
        = evaluate_curve_unknown  // above, below or
                                  // don’t care
    ) const;
```

This function calculates derivatives, of any order up to the number requested, and stores them in vectors provided by the user. It 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.

Any of the pointers in the array of pointers to vectors may be null, in which case the corresponding derivative will not be returned.

```
public: virtual int par_int_cur::evaluate_surfs (
    double,                          // parameter
    SPAposition&,                    // point on curve
                                     // at given
                                     // parameter
    SPAvector*,                      // derivatives of
                                     // off_int_cur
    int& nd_cu,                      // no. of curve
                                     // derivatives
                                     // required/calc.
    int& nd_sf,                      // no. of surface
                                     // derivatives
                                     // required/calc.
    evaluate_curve_side              // the evaluation
        = evaluate_curve_unknown,    // location
                                     // above, below
                                     // or don't care
    SPAposition&                     // point on
        = * (SPAposition*) NULL_REF,// support
                                     // surface 1
    SPAvector*                       // derivatives of
        = NULL,                      // 1st support
                                     // surface
    SPAposition&                     // point on
        = * (SPAposition*) NULL_REF,// support
                                     // surface 2
    SPAvector*                       // derivatives of
        = NULL,                      // 2nd support
                                     // surface
    SPApar_pos&                      // Parameters on
        = * (SPApar_pos*) NULL_REF,  // surface 1
    SPApar_vec*                      // derivatives of
```

```
        = NULL,                       // parameters on
                                      // surface 1
    SPApar_pos&                       // Parameters on
        = * (SPApar_pos*) NULL_REF,   // surface 2
    SPApar_vec*                       // derivatives of
        = NULL,                       // parameters on
                                      // surface 2
    SPApar_pos const&                 // optional guess
        = * (SPApar_pos* )NULL_REF,   // value for 1st
                                      // par_pos
    SPApar_pos const&                 // optional guess
        = * (SPApar_pos* )NULL_REF    // value for 2nd
                                      // par_pos
    ) const;
```

An evaluator that takes surface arguments in addition to the usual arguments. As well as returning curve position and derivatives, it returns the derivatives of the surface wrt t (these will often but not always be equal to the curve derivs) and also the derivatives of the surface parameters wrt t. The array of vectors to return the curve derivatives must be of length at least nd_cu, and the various arrays of vectors to return the surface data can either be null, indicating that this particular derivative is not required, or be of length at least nd_sf.

Unlike the other evaluators, this function OVERWRITES the integer arguments specifying the numbers of derivatives required, with the number actually obtained. The function itself returns information about the surface data that was calculated:

0 => no surface data (e.g. exact_int_cur)
1 => data for first surface only
2 => data for second surface only
3 => data for both surfaces

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

Returns the ID for the par_int_cur list.

```
protected: virtual void par_int_cur::make_approx (
    double fit,                  // fit
    const intcurve& ic           // intcurve
        = * (intcurve*) NULL_REF
    ) const;
```

Make or remake the approximating curve. The intcurve argument 'ic' may be null but if it is supplied the function may be a little faster. The function returns the approximating curve but does not store it, so that it can be a const function.

```
protected: virtual curve_evaldata*
    par_int_cur::make_evaldata () const;
```

Constructs 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.

```
protected: virtual double par_int_cur::param (
    SPAposition const&,      // position
    SPAparameter const&      // parameter
    ) const;
```

Parameter value for a given point on a curve.

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

Restores the information for the par_int_cur from the save 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.

int_cur::restore_common_data	Restore the underlying interpolated curve
if (restore_version_number >= PARCUR_VERSION)	
read_logical	Either "surf2" or "surf1"

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

Saves the information for the par_int_cur to the save file. Stores the information from this class to the save 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.

```
protected: virtual void par_int_cur::split (
    double,                 // parameter value
    SPAposition const&,     // position of split
    int_cur* [ 2 ]          // 2 pieces after split
    );
```

Divides an intersection curve into two pieces at a given parameter value, possibly adjusting the spline approximations to an exact value at the split point.

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

Returns the type of par_int_cur.

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

Returns the string “parcur”.

Internal Use: evaluate_iter, full_size

Related Fncs: restore_par_int_cur

par_int_interp

Class: Construction Geometry

Purpose: Fits a 3D curve to the parameter curve.

Derivation: par_int_interp : curve_interp : ACIS_OBJECT : –

SAT Identifier: None

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

Description: This class fits a 3D curve to the parameter curve in a surface given a parameter interval on the defining curve and the number of (equally-spaced) points int he interval to be fitted.

Limitations: None

References: KERN pcurve

Data:

None

Constructor:

```
public: par_int_interp::par_int_interp (
    pcurve const&,          // parameter-space curve
    int,                    // number of points
    SPAinterval const&,     // curve parameter range
    double,                 // fit tolerance
    logical                 // projected surface
        = TRUE
    );
```

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

Constructs an interpolated curve, given all the necessary information. If logical is TRUE then the first surface is projected; if it is FALSE, then the second surface is projected.

Destructor:

```
public: par_int_interp::~par_int_interp ();
```

C++ destructor, deleting a part_int_interp.

Methods:

```
public: int_cur* par_int_interp::make_int_cur ();
```

Constructs the appropriate int_cur subclass object (in this case, a par_int_cur) from the data in this object after curve interpolation.

```
public: void par_int_interp::true_point (
    double,                 // tolerance
    point_data&             // point data
    ) const;
```

Finds the true-point in 3D for a given parameter value. The input position, direction, and parameter values are approximate; the exact values are provided as output.

Related Fncs:

None

pattern

Class: Patterns, SAT Save and Restore

Purpose: Provides all information necessary to generate a regular or irregular pattern of entities from a single, “seed” entity.

Derivation: pattern : ACIS_OBJECT : –

SAT Identifier: None

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

Description: Refer to Purpose.

Limitations: None

References:

KERN	APATTERN, pattern_datum
by KERN	APATTERN, pattern_datum, pattern_holder
BASE	SPAinterval, SPAtransf
LAW	law

Data:

None

Constructor:

```
public: pattern::pattern (
    const pattern& in_pattern   // pattern to copy
    );
```

C++ copy constructor.

```
public: pattern::pattern (
    const SPAposition* in_positions// array of
        = NULL,             // pattern positions
    int in_list_size        // number of pattern
        = 0                 // transforms
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments. The pattern transforms are relative to pattern coordinates.

```
public: pattern::pattern (
    const SPAtransf* in_transfs,// array of pattern
                            // transforms
    int in_list_size        // number of pattern
                            // transforms
    );
```

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

```
public: pattern::pattern (
    law* in_trans_vec,       // translation law
    law* in_x_vec            // x-axis orientation
        = NULL,              // law
    law* in_y_vec            // y-axis orientation
        = NULL,              // law
    law* in_scale            // scaling law
        = NULL,
    law* in_z_vec            // z-axis orientation
        = NULL,              // law
    law* in_keep             // keep filter law
        = NULL,
    const SPAtransf& in_root_transf// root
        = * (SPAtransf*)NULL_REF// transformation
    );
```

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

Destructor:

```
protected: pattern::~pattern ();
```

C++ destructor, deleting a pattern.

Methods:

```
public: void pattern::add () const;
```

This is not usually called by an application directly. In order to preserve a copy of this pattern, an application calls this method. It is also called by the pattern constructors for the pattern being constructed, as well as for all of its sublaws. It increments the use_count.

```
public: logical pattern::add_element (
    int index                // index
    );
```

Adds the element indexed by index to the pattern. If this element was not formerly suppressed (via a keep law or list entry), this method has no effect. Unlike the restore_element methods, this method adds the element whether or not it already has an entry in the pattern list.

```
public: void pattern::compose (
    const pattern& in_pat    // pattern to be composed
                             // with this pattern
    );
```

Composes the pattern with the pattern referenced by in_pat.

```
public: void pattern::concatenate (
    const pattern& cat_pat,  // pattern to be
                             // concatenated with
                             // this pattern
    const SPAtransf& cat_trans// transformation to be
        = * (SPAtransf*)NULL_REF //applied to cat_pat
    );
```

Concatenates the pattern with the pattern referenced by cat_pat. If cat_trans is given, it is applied to cat_pat prior to concatenation.

```
public: pattern* pattern::deep_copy (
    pointer_map* pm          // list of items
        = NULL               // 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.

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: int pattern::domain_size (
    int index                    // index
    ) const;
```

Returns the size of the pattern domain in the dimension specified by the zero–based index.

```
public: int
    pattern::first_included_element () const;
```

Returns the internal index of the first element included in the pattern (i.e., the first element not suppressed by a keep law or list entry).

```
public: APATTERN* pattern::get_APATTERN ();
```

Returns the pattern as an entity.

```
public: void pattern::get_coords (
    int index,              // index
    double*& out_coords     // coordinates
    ) const;
```

Returns the pattern coordinates of an element, given its index.

```
public: int pattern::get_index (
    const double* in_coords // coordinates
    ) const;
```

Returns the index of the pattern element, given its coordinates.

```
public: law* pattern::get_keep ();
```

Returns a pointer to the pattern keep law.

```
public: const pattern_datum* const*
    pattern::get_list () const;
```

Returns a pointer to the list of pattern_datum pointers.

```
public: int pattern::get_list_size () const;
```

Returns the size of the list of pattern_datum pointers.

```
public: const SPAtransf*
    pattern::get_root_transf () const;
```

Returns a pointer to the root transformation of the pattern.

```
public: law* pattern::get_scale ();
```

Returns a pointer to the scale law of the pattern.

```
public: law* pattern::get_trans ();
```

Returns a pointer to the translation law of the pattern.

```
public: logical pattern::get_transf (
    int from,               // index of the existing
                            // element
    int to,                 // index of new element
    SPAtransf& out_trans,   // associated relative
                            // transform (output)
    logical use_map         // index flag
        = TRUE
    ) const;
```

Returns the relative transformation used to generate the pattern element indexed by from from the element indexed by to. The argument use_map is a flag to map the indices to skip over any suppressed elements. The function returns FALSE if either from or to are invalid indices.

```
public: logical pattern::get_transf (
    int index,              // index
    SPAtransf& out_trans,   // transform
    logical use_map         // use map
        = TRUE
    ) const;
```

Returns the transformation used to generate the pattern element indexed by index. The argument use_map is a flag to map index to skip over any suppressed elements. The function returns FALSE if index is invalid.

```
public: void pattern::get_visible_coords (
    int index,                  // index
    double*& out_coords         // coordinates
    ) const;
```

Behaves as does get_coords, except that suppressed pattern elements are ignored.

```
public: int pattern::get_visible_index (
    const double* in_coords     // coordinates
    ) const;
```

Behaves as does get_index, except that suppressed pattern elements are ignored.

```
public: law* pattern::get_x ();
```

Returns a pointer to the x–axis law of the pattern.

```
public: law* pattern::get_y ();
```

Returns a pointer to the y–axis law of the pattern.

```
public: law* pattern::get_z ();
```

Returns a pointer to the z–axis law of the pattern.

```
public: logical pattern::has_z_vec () const;
```

Returns TRUE if the pattern has a non–NULL z_vec member.

```
public: logical pattern::is_circular (
    SPAposition root,       // circle's center
    SPAvector axis          // direction of
                            // circle's axis
    ) const;
```

Returns TRUE if the pattern exhibits circular symmetry with respect to the specified circle.

```
public: logical pattern::is_cylindrical (
    FACE* in_face           // cylindrical face
    ) const;
```

Returns TRUE if the pattern exhibits cylindrical symmetry with respect to the specified cylinder.

```
public: logical pattern::is_cylindrical (
    SPAposition root,       // position on axis
    SPAvector axis          // direction of axis
    ) const;
```

Returns TRUE if the pattern exhibits cylindrical symmetry with respect to the specified cylinder.

```
public: logical pattern::is_included_element (
    int index               // index
    ) const;
```

Returns TRUE if the element referred to by the internal value index is not suppressed by the pattern keep law or a list entry.

```
public: logical pattern::is_planar (
    SPAposition root,       // position on plane
    SPAvector normal        // normal to plane
    ) const;
```

Returns TRUE if the pattern exhibits planar symmetry with respect to the specified plane.

```
public: logical pattern::is_planar (
    FACE *in_face           // planar face
    ) const;
```

Returns TRUE if the pattern exhibits planar symmetry with respect to the specified plane.

```
public: logical pattern::is_spherical (
    FACE* in_face           // spherical face
    ) const;
```

Returns TRUE if the pattern exhibits spherical symmetry with respect to the specified sphere.

```
public: logical pattern::is_spherical (
    SPAposition root        // center of sphere
    ) const;
```

Returns TRUE if the pattern exhibits spherical symmetry with respect to the specified sphere.

```
public: int pattern::make_element_index_law (
    law*& index_law         // law
    ) const;
```

Creates a scalar index law of the same size as the pattern.

```
public: void pattern::mirror (
    const SPAposition& root, // a point on the
                             // reflecting plane
    const SPAvector& normal  // vector normal to the
                             // reflecting plane
    );
```

Replaces this pattern by its concatenation with its reflection, using root and normal to define the reflecting plane.

```
public: logical pattern::move_element (
    const double* in_coords, // coordinates of the
                             // affected element
    const SPAtransf& move    // applied transform
    );
```

Transforms the element referenced by the coordinates in_coords according to the transform move.

```
public: logical pattern::move_element (
    int index,               // index of the affected
                             // element
    const SPAtransf& move    // applied transform
    );
```

Transforms the element referenced by index according to the transform move.

```
public: int pattern::num_elements () const;
```

Returns the number of elements in the pattern.

```
public: logical pattern::operator!= (
    const pattern& pat       // pattern to compare
    );
```

Returns TRUE if pat is not identical to this pattern.

```
public: const pattern& pattern::operator*= (
    const pattern& in_pat    // pattern to compose
                             // with this pattern
    );
```

Replaces this pattern with its composition with in_pat.

```
public: const pattern& pattern::operator+= (
    const pattern& in_pat    // pattern to concatenate
                             // with this pattern
    );
```

Replaces this pattern with its concatenation with in_pat.

```
public: logical pattern::operator== (
    const pattern& pat       // pattern to compare
                             // with this pattern
    );
```

Returns TRUE if pat is identical to this pattern.

```
public: void pattern::orient_vec (
    const SPAvector& in_vec, // vector to be oriented
    SPAvector& out_vec       // resulting vector
                             // (output)
    ) const;
```

Orients in_vec according to the axes defined for the first element of this pattern and returns the result as out_vec.

```
public: void pattern::reflect (
    const SPAposition& root, // a point on the
                             // reflecting plane
    const SPAvector& normal  // a vector normal to the
                             // reflecting plane
    );
```

Replaces this pattern by its reflection, using root and normal to define the reflecting plane.

```
public: void pattern::remove ();
```

Applications should call remove instead of the tilde (~) destructor to get rid of a pattern. Decrements the use_count. This is called by the pattern destructors for the pattern being destructed, as well as for all of its sublaws. The remove method calls the destructor if use_count falls to zero. Used for memory management.

```
public: logical pattern::remove_element (
    const double* in_coords // coordinates of the
                            // affected element
    );
```

Suppresses the element referenced by the coordinates in_coords, returning FALSE if the element has already been removed. The element may be restored using the restore_element method.

```
public: logical pattern::remove_element (
    int index               // index of the affected
                            // element
    );
```

Suppresses the element referenced by index, returning FALSE if the element has already been removed. The element may be restored using the restore_element method.

```
public: logical pattern::restore_element (
    const double* in_coords // coordinates of the
                            // affected element
    );
```

Restores to the pattern the element referenced by the coordinates in_coords, returning FALSE if the element is already present in the pattern. Unlike the add_element method, this method does nothing unless there is already an entry for the element in the pattern list.

no data	This method does not store any data.

```
public: logical pattern::restore_element (
    int index               // index of the
                            // affected element
    );
```

Restores to the pattern the element referenced by index, returning FALSE if the element is already present in the pattern. Unlike the add_element method, this method does nothing unless there is already an entry for the element in the pattern list.

no data	This method does not store any data.

```
public: void pattern::rotate (
    law* rx_law,              // x rotation
    law* ry_law,              // y rotation
    law* rz_law,              // z rotation
    logical merge             // merge flag
        = TRUE
    );
```

Rotates the orientation laws of the pattern by applying the rotation laws rx_law, ry_law, and rz_law when merge is TRUE; otherwise, these three laws replace the existing orientation laws.

```
public: void pattern::save () const;
```

Saves the pattern to a SAT file.

```
public: logical pattern::scale_element (
    const double* in_coords,     // coordinates
    double in_scale,             // scale factor
    const SPAposition& root,     // point about which
                                 // scaling is applied
    logical merge                // flag for
        = TRUE                   // merge/replace
);
```

Uniformly scales the element referenced by in_coords by a factor equal to in_scale, about the position root. The existing scaling is either factored in or replaced, depending upon the setting of the flag merge. A value of FALSE is returned if in_scale is unity.

```
public: logical pattern::scale_element (
    int index,                  // index
    double in_scale,            // scale factor
    const SPAposition& root,    // point about which
                                // scaling is applied
    logical merge               // flag for
        = TRUE                  // merge/replace
    );
```

Uniformly scales the element referenced by index by a factor equal to in_scale, about the position root. The existing scaling is either factored in or replaced, depending upon the setting of the flag merge. A value of FALSE is returned if in_scale is unity.

```
public: logical pattern::scale_element (
    const double* in_coords,    // coordinates of the
                                // affected element
    const SPAvector& in_scale,  // vector of scale
                                // factors
    const SPAposition& root,    // root position for
                                // scaling
    logical merge               // flag for
        = TRUE                  // merge/replace
    );
```

Scales the element referenced by the coordinates in_coords by the factors taken from in_scale, about the position root. The existing scaling is either factored in or replaced, depending upon the setting of the flag merge. A value of FALSE is returned if all in_scale components are unity.

```
public: logical pattern::scale_element (
    int index,                  // index of the
                                // affected element
    const SPAvector& in_scale,  // vector of scale
                                // factors
    const SPAposition& root,    // root position for
                                // scaling
    logical merge               // flag for
        = TRUE                  // merge/replace
    );
```

Scales the element referenced by index by the factors taken from in_scale, about the position root. The existing scaling is either factored in or replaced, depending upon the setting of the flag merge. A value of FALSE is returned if all in_scale components are unity.

```
public: void pattern::set_keep (
    law* keep_law,          // new keep law
    logical merge           // flag for merge/replace
        = TRUE
    );
```

Sets the keep law, if absent, or modifies the existing one. In the latter case, the merge flag determines whether or not keep_law is factored into the existing one or completely replaces it.

```
public: void pattern::set_list (
    pattern_datum** in_dl,  // pointer list
    int in_list_size        // list size
    );
```

Sets the list of pattern_datum pointers to in_dl, and updates the list size stored with the pattern.

```
public: void pattern::set_root_transf (
    const SPAtransf& in_transf// transform to be used
                              // as the root transform
    );
```

Sets the root transform to that given by in_transf.

```
public: void pattern::set_scale (
    law* scale_law,         // new scale law
    const SPAposition& root // point about which
        = SPAposition(0,0,0),// scaling is applied
    logical merge           // flag for merge/replace
        = TRUE
    );
```

Sets the scale law, if absent, or modifies the existing one. In the latter case, the merge flag determines whether or not scale_law is factored into the existing one or completely replaces it. The position root specifies the point about which the scaling is applied.

```
public: int pattern::take_dim () const;
```

Returns the dimension of the pattern laws' domain (input).

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

Establishes the domain of a given term in the pattern laws.

```
public: void pattern::transform (
    const SPAtransf& in_transf  // transform used
    );
```

Transforms this pattern using in_transf.

```
public: void pattern::translate (
    law* disp_law           // displacement law used
    );
```

Translates the pattern by merging the displacement law disp_law with the pattern.

```
public: logical pattern::undo_move_element (
    const double* in_coords // coordinates of the
                            // affected element
    );
```

Restores the pattern element referenced by the coordinates in_coords to the location defined by the pattern laws, by removing the element from the pattern list. This method returns FALSE if the element is not present in the list.

```
public: logical pattern::undo_move_element (
    int index               // index of the affected
                            // element
    );
```

Restores the pattern element referenced by index to the location defined by the pattern laws, by removing the element from the pattern list. This method returns FALSE if the element is not present in the list.

```
friend: pattern* operator* (
    const pattern& pat1,     // first pattern
    const pattern& pat2      // second pattern
    );
```

Creates a pattern that is the composition of pat1 and pat2 and returns its pointer.

MAC NT UNIX platforms only.

```
friend: pattern* operator+ (
    const pattern& pat1,     // first pattern
    const pattern& pat2      // second pattern
    );
```

Creates a pattern that is the concatenation of pat1 and pat2 and returns its pointer.

MAC NT UNIX platforms only.

Internal Use: root_to_first_element

Related Fncs: restore_law_data, restore_pattern, restore_pattern_datum

PCURVE

Class: Model Geometry, SAT Save and Restore

Purpose: Defines a 2D parameter-space approximation to a curve as an object in the model.

Derivation: PCURVE : ENTITY : ACIS_OBJECT : –

SAT Identifier: “pcurve”

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

Description: The purpose of a PCURVE is to provide a persistent, logable, savable object to manage pcurve information associated with a COEDGE.

PCURVE is a model geometry class that provides a (lowercase) pcurve, 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 PCURVE provides a procedural 2D parameter-space representation of a CURVE lying on a parameterized SURFACE. The representation may be a private copy(i.e., contained within a lowercase pcurve associated with the PCURVE), or it may refer to information contained within an intcurve. In either case, it may be negated from the underlying parameter space representation. Because there is only one such representation, this class does not need to have derived classes for specific geometries.

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

A use count allows multiple references to a PCURVE. The construction of a new PCURVE 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	CURVE, ENTITY, pcurve
by KERN	COEDGE, pattern_holder
BASE	SPApar_vec

Data:

None

Constructor:

```
public: PCURVE::PCURVE ();
```

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: PCURVE::PCURVE (
    CURVE*,                     // existing CURVE
    int,                        // definition
                                // type index
    logical                     // negated
        = FALSE,
    SPApar_vec const&           // parameter
        = * (SPApar_vec* ) NULL_REF// space 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.

Make a PCURVE to point to an existing PCURVE (via a CURVE). The index is positive 1 or 2, representing the two surfaces in order. A logical value of TRUE means the PCURVE referenced via the CURVE is considered negated, and SPApar_vec offsets the spline surface in parametric space.

```
public: PCURVE::PCURVE (
    PCURVE*                 // existing PCURVE
    );
```

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.

```
public: PCURVE::PCURVE (
    pcurve const&           // pcurve object
    );
```

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.

Destructor:

```
public: virtual void PCURVE::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 PCURVE::~PCURVE ();
```

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 PCURVE(...) then later x–>lose.)

Methods:

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

Increments the value of the use count for the PCURVE. 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 PCURVE::add_owner (
    ENTITY* owner,              // owner
    logical increment_use_count // increment use
        = TRUE                  // flag
    );
```

Adds owner argument to list of owners.

```
protected: virtual logical
     PCURVE::bulletin_no_change_vf (
    ENTITY const* other,        // other pointer in
                                // change bulletin
    logical identical_comparator// comparator
    ) const;
```

A virtual compare function. Compare this object with its change bulletin partner to see if the two entities are really the same.

```
public: virtual void PCURVE::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: pcurve const& PCURVE::def_pcur () const;
```

Returns the definition pcurve, or NULL if the pcurve is not private.

```
public: virtual logical PCURVE::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 PCURVE is FALSE.

```
public: pcurve PCURVE::equation () const;
```

Returns the CURVE equation, for reading only.

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

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

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

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

```
public: int PCURVE::index () const;
```

Returns the definition type of the PCURVE. A 0 value indicates a private pcurve. A positive 1 or 2 represents the first or second pcurve in an intcurve definition while a negative 1 or 2 represents the reverse of the corresponding pcurve.

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

Returns TRUE if this can be deep copied.

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

Returns TRUE if the entity is use counted.

```
public: void PCURVE::negate ();
```

Negates the pcurve, either by reversing the pcurve or by reversing the value of a nonzero def_type. 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: SPApar_vec PCURVE::offset () const;
```

Returns the SPApar_vec parameter space vector offset. Offset is the displacement in parameter space between the “fit” definition and this PCURVE. This allows the PCURVE to be positioned in the infinite parameter space of a periodic surface, so continuous curve sequences in object space are continuous in parameter space. The components of this vector should always be integer multiples of the corresponding surface parameter period, zero if it is not periodic in that direction.

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

Transforms the PCURVE. If the definition is a CURVE reference, it assumes that the curve will be transformed as well, so it does nothing. 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* PCURVE::ref_curve () const;
```

Returns the reference CURVE.

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

Decrements the value of the use count for the PCURVE. If the use count reaches 0, the record is deleted using lose. 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 PCURVE::remove_owner (
    ENTITY*,                // owner
    logical                 // decrement use
        = TRUE,             // count flag
    logical                 // lose if
        = TRUE              // zero flag
    );
```

Removes the owner argument from the list of owners.

```
public: void PCURVE::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();
read_int                              Type of pcurve.
if (def_type == 0)
    pcurve::restore_data              Save the data from the underlying
                                      low-level geometry definition.
else
    read_ptr                          Pointer to the CURVE definition.
    if (restore_version_number < PCURVE_VERSION)
        // Set off = SPApar_vec(0, 0)
    else
        read_real                     du
        read_real                     dv
        // Set off = SPApar_vec(du, dv)
    if ( !std_acis_save_flag )
        read_int                      use count data
```

```
public: void PCURVE::set_def (
    CURVE*,                          // CURVE
    int,                             // nth pcurve
    logical                          // negate
        = FALSE,
    SPApar_vec const&                // parameter
        = * (SPApar_vec* ) NULL_REF// space vector
    );
```

Sets set_def to the *nth* pcurve of an existing CURVE, where n as a positive integer. The logical negate is TRUE for a reversed pcurve. Removes any previous reference in cur to a CURVE, and increments the use-count for the given CURVE. Makes a NULL pcurve and sets it in def. 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 PCURVE::set_def (
    pcurve const&           // pcurve
    );
```

Checks that this PCURVE has been backed up, then zeros def_type, removes any curve referred to by cur (which is set to NULL), sets def_type to zero, and puts the given pcurve in def. 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 PCURVE::set_use_count (
    int val                 // use count
    );
```

Sets the use count to the given value.

```
public: void PCURVE::shift (
    SPApar_vec const&       // parameter space vector
    );
```

Shifts the PCURVE in parameter space by integral multiples of the period on a periodic surface. 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: pcurve* PCURVE::trans_pcurve (
    SPAtransf const&            // transform
        = * (SPAtransf* ) NULL_REF,
    logical                     // negate
        = FALSE
    ) const;
```

Construct a transformed pcurve. The logical is TRUE if the pcurve is considered to be reversed. 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 const char*
    PCURVE::type_name () const;
```

Returns the string “pcurve”.

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

Returns the value of the use count.

Internal Use: full_size

Related Fncs: is_PCURVE

pcurve

Class: Construction Geometry, SAT Save and Restore

Purpose: Defines a 2D curve defined in the parameter space of a parametric surface.

Derivation: pcurve : ACIS_OBJECT : –

SAT Identifier: None

Filename: kern/kernel/kerngeom/pcurve/pcudef.hxx

Description: The pcurve class represents parameter-space curves that map an interval of the real line into a 2D real vector space (parameter space). This mapping is continuous, and one-to-one except possibly at the ends of the interval whose images may coincide. It is differentiable twice, and the direction of the first derivative with respect to the parameter is continuous. This direction is the positive sense of the curve.

A parameter-space curve is always associated with a surface, that maps the parameter-space image into 3D real space (object space); therefore, the two mappings together can be considered to be a single mapping from a real interval into object space. Most of the properties of a parameter-space curve relate in fact to this combined mapping.

If the two ends of the curve are different in object space, the curve is open. If they are the same, it is closed. If the curve joins itself smoothly, the curve is periodic, with its period being the length of the interval that it is primarily defined. A periodic curve is defined for all parameter values, by adding a multiple of the period to the parameter value so the result is within the definition interval, and evaluating the curve at that resultant parameter. The point at the ends of the primary interval is known as the seam. If the surface is periodic, a closed or periodic parameter-space curve cannot in fact be closed in parameter space, but its end values can differ by the surface parameter period in one or both directions.

Also, a parameter-space curve is always associated with an object-space curve lying in (or fitted to) the surface. This curve is used to assist in the determining of the surface parameter values corresponding to object-space points on the 3D curve, by using the parameter value on the 3D curve to evaluate the 2D curve for an approximation to the surface parameter values for iterative refinement. For this reason, a parameter-space curve must always have the same parameter range as its associated object-space curve, and its internal parameterization must be similar, though not necessarily identical, to that of the object-space curve. A parameter-space curve can have the same sense as its associated object-space curve, or be opposite. In the latter case, the parameterization is negated one to the other.

In general, it is not necessary to have *u* or *v* continuity between two pcurves on a periodic face or on a singular face, because evaluation of a parameter value outside the parameter range is mapped back into the evaluation range by adding or subtracting the period from the parameter value to be evaluated (periodic face), or by adding or subtracting the parameter value changes while the actual 3-space location is coincident (singular face).

In the case of a spline converted sphere, face 0 ranges from u = –pi/2 to pi/2 and v = 0 to pi; face 1 ranges from u = –pi/2 to pi/2 and v = pi to 2*pi. Coedges 0 and 2 belong to face 0, so their param ranges should match the corresponding face param ranges. Coedge 0 has param values: u = –pi/2 to pi/2 and v = 0. Coedge 2 has param values: u = pi/2 to –pi/2 and v = pi.

The param values are correct, because Coedge 0 travels from top vertex to bottom vertex, and Coedge 2 travels from bottom vertex to top vertex on the sphere; as a result, u values are continuous (–pi/2 to pi/2 and pi/2 to –pi/2) moving along Coedge 0, onto Coedge 2, and along Coedge 2. For the V direction (since face 0 is singular in v), v ranges from 0 to pi along both singular poles of the hemispherical face, so v value may be any value between 0 and pi for the given u param values, and should still evaluate to the same 3–space point; at u = –pi/2 or u = pi/2, all v values map to (0, 0, 20) and (0, 0, –20) respectively. Since the pcurves map to surface param values, they could use any value at endpoints. However, since pcurve 0 must travel along entire edge of face 0, the v value must be pi along the entire length of pcurve 0 (only valid value at u values not at singularity points in v). Similarly, pcurve 2 must have v value of 0 along entire length of pcurve 2. Transition from pcurve 0 to pcurve 2 is, then, continuous in u, and discontinuous in v, but discontinuity in v matches singularity in v at poles of hemispherical face 0; in addition, evaluation of any value of v (0 to pi) at poles gives correct 3–space position. Similar discussion results if data for Coedge 1 and 3 is examined.

A pcurve consists of pointer to a par_cur that holds the data defining the 2D parameter space curve and a logical flag indicating reversal of the pcurve from the underlying spline curve. In addition, a parameter space vector is stored that represents the displacement of this pcurve in the parameter space of the surface the pcurve lies in. By having a nonzero vector for a periodic surface, a continuous sequence of object space curves (3D curves) can have a continuous sequence of parameter space curves (2D curves).

The par_cur in turn consists of a pointer to a bs2_curve (a 2D spline curve), a fitting tolerance, and a pointer to the surface where the parameter space curve lies.

Limitations: None

References:

KERN	par_cur
by KERN	BDY_GEOM_CIRCLE, BDY_GEOM_PCURVE, PCURVE, exp_par_cur, imp_par_cur, law_par_cur, par_cur, par_int_interp, pcur_int_cur, pcurve_law_data, skin_spl_sur, stripc, surf_surf_int
BASE	SPApar_vec

Data:

None

Constructor:

```
public: pcurve::pcurve ();
```

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

```
public: pcurve::pcurve (
    law*,                   // law
    double,                 // start parameter
    double,                 // end parameter
    surface const&,         // surface
    double                  // fit tolerance
        = SPAresabs,
    bs2_curve               // underlying curve
        = NULL
    );
```

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

```
public: pcurve::pcurve (
    bs2_curve,              // underlying bs2 curve
    double,                 // parameter
    surface const&,         // surface
    int                     // knots on curve
        = -1,               // default is unknown
    int                     // bs2_curve hull
        = -1,               // contains true curve
    int                     // hull angles ok
        = -1,               // default is unknown
    int                     // hull self intersects
        = -1                // default is unknown
    );
```

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

Creates a pcurve as a bs2_curve in the parameter-space of a surface and having a specified fit tolerance.

```
public: pcurve::pcurve (
    curve const&,           // curve name
    surface const&,         // surface name
    double                  // parameter
    );
```

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

Creates a pcurve fitting it to the given object-space curve lying on the given surface.

```
public: pcurve::pcurve (
    intcurve const&,         // intcurve name
    int                      // integer
    );
```

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

```
public: pcurve::pcurve (
    par_cur*                 // parameter curve
    );
```

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

Creates a pcurve by promoting a par_cur to a full pcurve.

```
public: pcurve::pcurve (
    pcurve const&            // parameter curve
    );
```

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

```
public: pcurve::pcurve (
    logical,                 // u direction flag
    double,                  // parameter
    surface const&,          // surface
    curve const&             // curve
        = *(curve*) NULL_REF
    );
```

C++ initialize constructor requests memory for this object and makes an iso pcurve (uses the curve if supplied).

Destructor:

```
public: pcurve::~pcurve ();
```

C++ destructor, deleting a PCURVE.

Methods:

```
public: int pcurve::accurate_knot_tangents () const;
```

Returns 1 if the bs2_curve tangents are within SPAresnor of the true curve tangent vectors at all of the knots that lie within the checked_range, or 0 if this is not satisfied. Returns –1 if this property has not yet been checked.

```
public: int pcurve::add_bs2_knot (
    double new_knot_param,          // knot to add
    int mult,                       // multiplicity
    SPApar_pos& new_knot_uv         // knot uv
        = * (SPApar_pos* ) NULL_REF,
    SPApar_vec& new_knot_deriv_below// derivative
        = * (SPApar_vec* ) NULL_REF,
    SPApar_vec& new_knot_deriv_above// derivative
        = * (SPApar_vec* ) NULL_REF
    );
```

Adds a knot to the bs2_curve with the supplied multiplicity. If the new SPApar_pos and derivative vectors are supplied, set the new control points so that the bs2_curve has the supplied position and derivatives at the new knot. Return the multiplicity of the new knot. It is assumed that the caller will take care of any changes in the underlying par_cur properties, (self intersection, for example), that the addition of knots generates.

```
public: void pcurve::add_knots_at_discontinuities (
    const curve* cu,        // curve
    int num_knots           // number of knots
        = 0,
    double* knots           // knots
        = NULL
    );
```

Adds knots to the pcurve's bs2_curve, at any of the corresponding curve discontinuities. The pcurve knots can be supplied to this function in an array, but if they are not, they will be extracted from the pcurve.

```
public: int pcurve::bezier_form ();
```

Returns 1 if the underlying bs2_curve is known to be Bezier form, and 0 if it is not in this form, and –1 if this property has not yet been checked.

```
public: SPApar_box pcurve::bound (
    double,                 // first parameter
    double,                 // second parameter
    const curve* true_cu    // curve
        = NULL
    ) const;
```

Returns a box around the curve.

```
public: SPApar_box pcurve::bound (
    SPAinterval const&,     // interval name
    const curve* true_cu    // curve
        = NULL
    ) const;
```

Returns a box around the curve.

```
public: SPAinterval pcurve::checked_range () const;
```

Returns the range over which checking occurs.

```
public: logical pcurve::check_bezier_form ();
```

Checks if the underlying bs2_curve is in Bezier form.

```
public: logical pcurve::check_hull_curve_enclosure (
    const curve* true_cu,   // curve
    BOUNDED_SURFACE* bsf    // bounded surface
        = NULL,
    int deg                 // degree
        = 0,
    int num_knots           // number of knots
        = 0,
    double* knots           // knots
        = NULL,
    int num_ctrlpts         // number of control
        = 0,                // points
    SPApar_pos* ctrlpts     // control points
        = NULL
    );
```

Checks if the pcurve bs2_curve hull contains the bounded true curve, and if not, to find the 2–space distance that the true curve moves outside the hull by. The bounded surface, degree, knots, and control points can be supplied to this function in an array, but if they are not, they will be extracted from the underlying par_cur. The function returns TRUE if the hull contains the true curve, or FALSE otherwise.

```
public: logical pcurve::check_hull_intersection (
    int deg                 // degree
        = 0,
    int num_knots           // number of knots
        = 0,
    double* knots           // knots
        = NULL,
    int num_ctrlpts         // number of control
        = 0,                // points
    SPApar_pos* ctrlpts     // control points
        = NULL
    );
```

Checks if the convex hull of a pcurve self–intersects. The degree and control points can be supplied to this function in an array, but if they are not, they will be extracted from the underlying par_cur. The function returns TRUE if the hull self–intersects, or FALSE otherwise.

```
public: logical pcurve::check_hull_turning_angle (
    const curve* true_cu,   // curve
    BOUNDED_SURFACE* bsf,   // bounded surface
    int deg,                // degree
    int& num_knots,         // number of knots
    double*& knots,         // knots
    int& num_ctrlpts,       // number of control
                            // points
    SPApar_pos*& ctrlpts    // control points
    );
```

Checks that the pcurve bs2_curve hull does not turn by more than the minimum turning angle. If it does, then the hull points are projected onto a parameter line and the order of the points is checked, to test if the pcurve has kinked. The bounded surface and the knots and control points can be supplied to this function, or if they are NULL, they will be extracted from the underlying par_cur. If the number of knots and control points, and the knot and control point arrays are supplied, these are supplied as references, as more knots and control points may be added to correct the hull. If the true curve is supplied, this is used to find the positions for any new knots that are added to the bs2_curve. The function returns TRUE if the hull does not turn by more than the defined minimum angle, or FALSE otherwise.

```
public: logical pcurve::check_knots_on_true_curve (
    const curve* true_cu,    // curve
    BOUNDED_SURFACE* bsf,    // bounded surface
    int deg,                 // degree
    int& num_knots,          // number of knots
    double*& knots           // knots
    );
```

Checks that all the knots of the pcurve bs2_curve lie on the supplied true curve. The bounded surface and the knots can be supplied to this function, but if they are not, they will be extracted from the underlying par_cur. If the number of knots and the knot array is supplied, these are supplied as references, as more knots may be added to correct the hull. The function returns TRUE if all of the knot points lie on the true curve, or FALSE otherwise.

```
public: logical pcurve::check_knot_tangents (
    const curve* true_cu,    // curve
    BOUNDED_SURFACE* bsf     // bounded surface
        = NULL,
    int num_knots            // number of knots
        = 0,
    double* knots            // knots
        = NULL
    );
```

Checks that the pcurve bs2_curve tangent directions agree with the true curve tangent directions at each of the knots. The bounded surface and the knots can be supplied to this function, but if they are not, they will be extracted from the underlying par_cur. The function returns TRUE if the tangent directions agree at all of the knot points, or FALSE otherwise.

```
public: bs2_curve pcurve::cur () const;
```

Returns the underlying 2D NURBS defining the parameter curve.

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

Outputs details of a pcurve.

```
public: virtual pcurve* pcurve::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.

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: void pcurve::eval (
    double,                          // parameter
    SPApar_pos&,                     // param position
    SPApar_vec&                      // 1st derivative
        = * (SPApar_vec* ) NULL_REF,
    SPApar_vec&                      // 2nd derivative
        = * (SPApar_vec* ) NULL_REF
    ) const;
```

Evaluates a pcurve at a parameter value, giving position and optionally first and second derivatives.

```
public: SPApar_vec pcurve::eval_deriv (
    double                   // parameter
    ) const;
```

Evaluates a pcurve at a parameter value, to give the derivative with respect to its parameter.

```
public: SPApar_pos pcurve::eval_position (
    double                   // parameter
    ) const;
```

Evaluates a pcurve at a parameter value.

```
public: SPApar_pos pcurve::eval_position (
    SPAposition const&,      // position name
    double,                  // parameter
    logical point_perp       // use point_perp if
                             // true, param if false
        = FALSE
    ) const;
```

Evaluates a pcurve at a parameter value.

```
public: double pcurve::fitol () const;
```

Returns the fit tolerance of the parameter curve.

```
public: double pcurve::hull_distance () const;
```

Returns the distance from the true curve to the underlying bs2_curve convex hull, if the true curve comes outside of the hull. Set to 0 if the true curve is known to lie completely within the hull or –1 if this property has not been checked yet.

```
public: int pcurve::hull_self_intersects () const;
```

Returns 1 if the bs2_curve hull self–intersects, within the checked_range, or to zero if it does not self–intersect within this range. Returns –1 if this property has not yet been checked.

```
public: int pcurve::hull_turning_angles_ok () const;
```

Returns the value for hull turning angle checking. Set to 1 if the bs2_curve hull does not turn too sharply at any point within the checked_range, or set to 0 if the hull does turn too sharply at a point. Set to –1 if this property has not yet been checked.

```
public: int pcurve::knots_on_true_curve () const;
```

Returns 1 if all of the bs2_curve knots that are within the checked_range lie on the associated true curve (to within SPAresabs), or to 0 if this is not the case. Returns –1 if this property has not yet been checked.

```
public: logical pcurve::make_bezier_form ();
```

Makes the underlying bs2_curve have Bezier form by adding knots as necessary.

```
public: BOUNDED_SURFACE* pcurve::
    make_bounded_surface () const;
```

Makes up a bounded surface from the underlying surface. This does not have to accurately bound the pcurve, but is just used to make up SVECs for the other member functions.

```
public: pcurve& pcurve::negate ();
```

Negates a pcurve in place.

```
public: SPApar_vec pcurve::offset () const;
```

Returns the offset.

```
public: pcurve& pcurve::operator*= (
    SPAtransf const&        // transformation
    );
```

Transforms a pcurve in object space.

```
public: pcurve pcurve::operator+ (
    SPApar_vec const&       // parameter vector
    ) const;
```

Displaces the curve in surface parameter space.

```
public: pcurve const& pcurve::operator+= (
    SPApar_vec const& pv    // parameter vector
    );
```

Add a SPApar_vec to a pcurve's offset.

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

Makes a negated curve.

```
public: pcurve pcurve::operator– (
    SPApar_vec const&       // parameter vector
    ) const;
```

Makes a negated curve given a SPApar_vec.

```
public: pcurve const& pcurve::operator–= (
    SPApar_vec const& pv    // parameter vector
    );
```

Subtract a SPApar_vec to a pcurve's offset.

```
public: pcurve& pcurve::operator= (
    pcurve const&           // pcurve to be assigned
    );
```

An assignment operator that copies the pcurve record, and adjusts the use counts of the underlying information.

```
public: double pcurve::param (
    const SPApar_pos& uv    // parameter position
    );
```

Returns the parameter of the pcurve at the given SPApar_pos.

```
public: double pcurve::param_period () const;
```

Returns the parameter period - the length of the parameter range if periodic, 0 otherwise.

```
public: SPAinterval pcurve::param_range () const;
```

Returns the principal parameter range of a pcurve.

```
public: void pcurve::reparam (
    double,                 // first parameter
    double                  // second parameter
    );
```

Performs a linear transformation on the curve parameterization, so that it starts and ends at the given values (that must be in increasing order).

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

Restore the data for a pcurve from a save file.

```
read_logical                        Sense of the pcurve: either
                                    “forward” or “reversed”.
if (restore_version_number < PCURVE_VERSION)
    // Restore as an explicit pcurve.
    par_cur* restore_exp_par_cur    Restore the appropriate pcurve
else
    // Switch to the right restore routine, using the standard
    // system mechanism.
    (par_cur *)dispatch_restore_subtype
    read_real                       du; u offset
    read_real                       dv; v offset
```

```
public: logical pcurve::reversed () const;
```

Inquires whether the parameter space curve is in the same or opposite direction of the underlying 2D NURBS curve.

```
public: void pcurve::save () const;
```

Saves the id the calls save_data.

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

Saves the information for a pcurve to the save file. Function to save a pcurve of a known type, where the context determines the pcurve type, so no type code is necessary.

```
public: void pcurve::save_pcurve () const;
```

Function to be called to save a pcurve of unknown type, or NULL. Just checks for null, then calls save. This separation is not really necessary for pcurves at present, as there are no subtypes, but we retain it for consistency with curve and surface.

```
public: void pcurve::set_accurate_knot_tangents (
    int acc_tangents        // value
    );
```

Set to 1 if the bs2_curve tangents are within SPAresnor of the true curve tangent vectors at all of the knots that lie within the checked_range, or to 0 if this is not satisfied. Set to –1 if this property has not yet been checked.

```
public: void pcurve::set_checked_range (
    const SPAinterval& new_range,   // interval
    int num_knots                   // number of
        = 0,                        // knots
    double* knots                   // knots
        = NULL
    );
```

Sets the checked range of the pcurve.

```
public: void pcurve::set_hull_distance (
    double dist             // value
    );
```

Sets the distance from the true curve to the underlying bs2_curve convex hull, if the true curve comes outside of the hull. Set to 0 if the true curve is known to lie completely within the hull or –1 if this property has not been checked yet.

```
public: void pcurve::set_hull_self_intersects (
    int self_ints           // value
    );
```

Set to 1 if the bs2_curve hull self–intersects, within the checked_range, or to zero if it does not self–intersect within this range. Set to –1 if this property has not yet been checked.

```
public: void pcurve::set_hull_turning_angles_ok (
    int angles_ok           // value
    );
```

Sets the value for hull turning angle checking. Set to 1 if the bs2_curve hull does not turn too sharply at any point within the checked_range, or set to 0 if the hull does turn too sharply at a point. Set to –1 if this property has not yet been checked.

```
public: void pcurve::set_knots_on_true_curve (
    int knots_on_cu         // value
    );
```

Set to 1 if all of the bs2_curve knots that are within the checked_range lie on the associated true curve (to within SPAresabs), or to 0 if this is not the case. Set to –1 if this property has not yet been checked.

```
public: void pcurve::set_surface(
    surface const&          // surface to set
    );
```

Used to set the surface of a pcurve after a space warp.

```
public: pcurve* pcurve::split (
    double,                     // parameter
    SPApar_pos const&           // parameter position
        = * (SPApar_pos* ) NULL_REF,
    SPApar_vec const&           // parameter position
        = * (SPApar_vec* ) NULL_REF
    );
```

Divides a pcurve into two pieces at a parameter value. This function creates a new pcurve on the heap, but either one of the pcurves may have a NULL actual curve. The supplied curve is modified to be the latter section, and the initial section is returned as value.

```
public: surface const& pcurve::surf () const;
```

Returns the surface that the parameter space curve is defined.

```
public: SPAinterval pcurve::trim (
    const SPAinterval& new_range,   // trim to range
    const SPApar_pos& start_uv      // start param
        = * (SPApar_pos* ) NULL_REF,
    const SPApar_vec& start_duv     // slope
        = * (SPApar_vec* ) NULL_REF,
    const SPApar_pos& end_uv        // end param
        = * (SPApar_pos* ) NULL_REF,
    const SPApar_vec& end_duv       // slope
        = * (SPApar_vec* ) NULL_REF
    );
```

Trims the pcurve to the supplied range, using the supplied end par_pos's, if any. The new pcurve range is returned.

```
public: SPAinterval pcurve::trim_to_curve_range (
    const curve* cu         // trimming curve
    );
```

Trims the pcurve to the range of the supplied curve. The new pcurve range is returned.

```
public: char const* pcurve::type_name () const;
```

Returns a string identifying the pcurve type. For exp_par_cur and its derived types, this method returns "exppc". For imp_par_cur and its derived types, this method returns "impcc".

```
public: void pcurve::validity_checks (
    const curve* cu,                // curve
    const SPAinterval& chkd_range,  // interval
    logical& knots_on_true_cur,     // knots of true
                                    // curve
    logical& knot_tangents_ok,      // knot tangents
                                    // ok
    logical& hull_turn_angles_ok,   // hull turning
                                    // angles ok
    logical& hull_intersects,       // hull self
                                    // intersections
    logical& hull_contains_true_cu  // hull contains
                                    // true curve
    );
```

Carries out all the validity checks for a pcurve, over the supplied interval. If a logical is supplied as a NULL reference, the corresponding test is not carried out. This function extracts the knots and control points from the pcurve and supplies them to the validity checking functions, so that they only have to be extracted once.

Internal Use: full_size, get_par_cur

Related Fncs:

restore_pcurve

```
friend: pcurve operator* (
    pcurve const&,          // pcurve
    SPAtransf const&        // transformation
    );
```

Transforms a pcurve in object space.

```
friend: pcurve operator+ (
    SPApar_vec const& pv,   // parameter vec
    pcurve const& pc        // pcurve
    );
```

Adds a SPApar_vec to a pcurve's offset.

pcurve_law_data

Class: Laws, Spline Interface, Construction Geometry, SAT Save and Restore

Purpose: Creates a wrapper for pcurve classes for passing as arguments to laws.

Derivation: pcurve_law_data : base_pcurve_law_data : path_law_data : law_data : ACIS_OBJECT : –

SAT Identifier: PCURVE

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

Description: This is a wrapper to handle specific ACIS pcurve classes. These wrapper classes are used by api_str_to_law. These are returned by the law method string_and_data.

Limitations: None

References:

KERN	pcurve
BASE	SPAposition, SPAvector

Data:

```
protected double *tvalue;
```
Holds the parameter values.

```
protected int *which_cached;
```
Holds the time tags.

```
protected int derivative_level;
```
Holds how many derivatives are cached.

```
protected int point_level;
```
Size of tvalue.

```
protected pcurve *acis_pcurve;
```
The actual ACIS pcurve.

```
protected SPAposition *cached_f;
```
Holds the positions.

```
protected SPAvector *cached_ddf;
```
Holds the second derivatives at the position.

```
protected SPAvector *cached_df;
```
Holds the first derivatives at the position.

Constructor:

```
public: pcurve_law_data::pcurve_law_data (
    pcurve const& in_acis_pcurve,   // pointer to
                                    // ACIS pcurve
    double in_start                 // starting
        = 0,                        // parameter
    double in_end                   // ending
        = 0                         // parameter
    );
```

C++ initialize constructor requests memory for an instance of pcurve_law_data and populates it with the data supplied as arguments. This sets the use_count to 1 and increments the how_many_laws. It sets dlaw, slaw, and lawdomain to NULL.

Destructor:

```
public: pcurve_law_data::~pcurve_law_data ();
```

C++ destructor, deleting a pcurve_law_data.

Methods:

```
public: double pcurve_law_data::curvature (
    double para             // parameter for calc
    );
```

This calculates the curvature at the given parameter value on the pcurve.

```
public: virtual law_data*
    pcurve_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: SPAvector pcurve_law_data::eval (
    double para,            // parameter for eval
    int deriv,              // which derivative
    int side                // which side
        = 0                 //
    );
```

Evaluates the pcurve at the given input parameter value and takes the respective derivative.

```
public: pcurve* pcurve_law_data::pcurve_data ();
```

Returns a pointer to the actual ACIS pcurve.

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

Saves the law to the SAT file.

```
public: void pcurve_law_data::set_levels (
    int in_point_level      // input point level
        = 4,
    int in_derivative_level // number of derivatives
        = 2
    );
```

Changes the number of points and derivative levels to cache.

```
public: int pcurve_law_data::singularities (
    double** where,         // where singularity is
    int** type,             // type of singularity
    double start,           // start value
    double end              // end value
    );
```

This specifies where in the given law there might be singularities.

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

This method is a pure virtual method for all classes derived from this one. The definition of this virtual method forces derived classes to have this method defined. This method is called from the derived class and not from this abstract class.

Related Fncs:

restore_law, restore_law_data, save_law

pcur_int_cur

Class: Construction Geometry, SAT Save and Restore

Purpose: Defines an interpolated curve subtype that is the 3D extension of the parameter curve representing a curve on a surface.

Derivation: pcur_int_cur : int_cur : subtrans_object : subtype_object : ACIS_OBJECT : –

SAT Identifier: “pcurcur”

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

Description: This class defines an interpolated curve subtype that is the 3D extension of the parameter curve representing a curve on a surface. This is used internally by ACIS during point-in-face testing on a parametric surface, and certain member functions that are not required by ACIS are disabled, to simplify the implementation. It should not be used by an application.

Limitations: None

References: KERN curve, pcurve

Data:

None

Constructor:

```
public: pcur_int_cur::pcur_int_cur (
    const pcur_int_cur&     // pcur_int_curve
    );
```

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

```
public: pcur_int_cur::pcur_int_cur (
    curve const&,           // 2D curve
    pcurve const&           // 3D pcurve
    );
```

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

Destructor:

None

Methods:

```
public: virtual int_cur* pcur_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 pcur_int_cur::id ();
```

Returns the ID for the pcur_int_cur list.

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

Saves the information for a pcur_int_cur to the save file.

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

Returns the type of pcur_int_cur.

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

Returns the string “pcurcur”.

Internal Use: full_size

Related Fncs:

None

pick_ray

Class: Picking

Purpose: Maps a 2D graphic pick on an entity position defined in model space.

Derivation: pick_ray : ACIS_OBJECT : –

SAT Identifier: None

Filename: kern/kernel/geomhusk/pick_ray.hxx

Description: A pick_ray is a combination of a position and a direction. It typically allows mapping a 2D graphic pick on an ENTITY position, such as an EDGE, that is defined in model space. In this context, the position is the 2D pick location mapped into 3D space, and the direction is the direction the user looks at (on the model and in the view) when the pick is executed.

Limitations: None

References:

by KERN	entity_with_ray
BASE	SPAposition, SPAunit_vector

Data:

None

Constructor:

```
public: pick_ray::pick_ray ();
```

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

```
public: pick_ray::pick_ray (
    const pick_ray&          // pick ray
    );
```

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

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

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

```
public: pick_ray::pick_ray (
    const SPAposition&,      // position
    const SPAvector&         // vector
    );
```

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

Destructor:

None

Methods:

```
public: SPAunit_vector pick_ray::direction () const;
```

Gets the direction (SPAunit_vector) from the pick_ray.

```
public: double pick_ray::distance_to (
    const SPAposition& pt    // position
    ) const;
```

Computes the distance from the pick_ray to a position.

```
public: pick_ray pick_ray::operator* (
    const SPAtransf&        // transformation
    ) const;
```

Returns a new pick_ray that is the result of applying a transformation to this pick_ray.

```
public: pick_ray& pick_ray::operator*= (
    const SPAtransf&        // transformation
    );
```

Applies a transformation to the pick_ray.

```
public: SPAposition pick_ray::point () const;
```

Gets the position from the pick_ray.

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

Sets the direction with a SPAunit_vector.

```
public: void pick_ray::set_direction (
    const SPAvector&        // vector
    );
```

Sets the direction with a vector.

```
public: void pick_ray::set_point (
    const SPAposition&      // position
    );
```

Sets the position.

Related Fncs:

None

PLANE

Class: Model Geometry, SAT Save and Restore

Purpose: Defines a plane as an object in the model.

Derivation: PLANE : SURFACE : ENTITY : ACIS_OBJECT : –

SAT Identifier: “plane”

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

Description: PLANE is a model geometry class that contains a pointer to a (lowercase) plane, 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.

PLANE is one of several classes derived from SURFACE to define a specific type of surface. The plane class defines a plane by a point on the plane and its unit normal.

Along with the usual SURFACE and ENTITY class methods, PLANE 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 normal of a plane.

A use count allows multiple references to a PLANE. The construction of a new PLANE 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 plane

Data:

None

Constructor:

```
public: PLANE::PLANE ();
```

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: PLANE::PLANE (
    plane const&            // plane 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.

```
public: PLANE::PLANE (
    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.

Makes a plane that passes through the given SPAposition with the given SPAunit_vector normal.

Destructor:

```
public: virtual void PLANE::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 PLANE::~PLANE ();
```

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 PLANE(...) then later x–>lose.)

Methods:

```
protected: virtual logical
     PLANE::bulletin_no_change_vf (
    ENTITY const* other,         // other pointer
                                 // in change bulletin
    logical identical_comparator// comparator
    ) const;
```

Compare this object with its change bulletin partner to see if the two entities are really the same.

```
public: virtual void PLANE::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: surface const& PLANE::equation () const;
```

Returns the surface equation of the PLANE.

```
public: surface& PLANE::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: virtual int PLANE::identity (
    int                     // level
        = 0
    ) const;
```

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

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

Returns TRUE if this can be deep copied.

```
public: SPAunit_vector const& PLANE::normal () const;
```

Returns the normal defining the PLANE.

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

Transforms the PLANE. 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 PLANE::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.

plane::restore_data	low-level plane geometry definition.

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

Returns the point defining the PLANE.

```
public: void PLANE::set_normal (
    SPAunit_vector const&   // normal
    );
```

Sets the PLANE's normal 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 PLANE::set_root_point (
    SPAposition const&      // root point
    );
```

Sets the PLANE'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: surface* PLANE::trans_surface (
    SPAtransf const&            // transform
        = * (SPAtransf* ) NULL_REF,
    logical                     // reversed
        = FALSE
    ) const;
```

Returns the transformed surface equation of the PLANE. If the logical is TRUE, the surface is reversed.

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

Returns the string “plane”.

Internal Use: full_size

Related Fncs: is_PLANE

plane

Class: Construction Geometry, SAT Save and Restore

Purpose: Defines a planar surface.

Derivation: plane : surface : ACIS_OBJECT : –

SAT Identifier: “plane”

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

Description: A plane class defines a plane with a point and a unit vector normal to the plane. Usually, the point chosen to define the plane is near the center of interest. The normal represents the outside of the surface. This is important when a plane is used to define a FACE of a shell or solid.

Four data members describe the parameterization of the plane. For more information about data members, see “Data.”

To find the object-space point corresponding to a given (*u, v*) pair, first find the cross product of the plane normal with u_deriv, negate it if reverse_v is TRUE, and call it v_deriv. Then the evaluated position is:

```
pos = root_point + u* u_deriv + v* v_deriv
```

When the plane is transformed, u_deriv is transformed in the usual way, along with the root point and normal, and reverse_v is inverted if the transform includes a reflection. When the plane is negated, the direction of the normal is reversed, and reverse_v is inverted.

When a plane is constructed, u_deriv is automatically generated to be a fairly arbitrary unit vector perpendicular to the normal, and reverse_v is set FALSE. If the normal is of zero length, or if the plane is constructed using the raw constructor with no normal, u_deriv is set to be a zero vector, and the arbitrary direction is generated whenever a parameter-based function is called. Whenever an application changes the normal directly, it should also ensure that u_deriv is perpendicular to it.

In summary, planes are:

– Not true parametric surfaces.
– Open in *u* and *v*.
– Not periodic in either *u* or *v*.
– Not singular at any *u* or *v*.

Limitations: None

References:

by KERN	PLANE
BASE	SPAposition, SPAunit_vector, SPAvector

Data:

```
public logical reverse_v;
```
By default the *v*-direction is the cross product of normal with u_dir. If this is TRUE, the *v*-direction must be negated. This is set to TRUE if the parameterization is left-handed with respect to the surface normal, or FALSE if it is right-handed. A right-handed parameterization is such as make the surface normal the direction of the cross product of the *u* and *v*-directions, respectively.

```
public SPAposition root_point;
```
A point though which the plane passes.

```
public SPAunit_vector normal;
```
The normal to the plane. Conventionally set to a NULL unit vector to indicate that the plane is undefined.

```
public SPAvector u_deriv;
```
The direction in the plane of constant *v*-parameter lines, with a magnitude to convert dimensionless parameter values into distances. This vector gives the direction and scaling of *u*-parameter lines.

Constructor:

```
public: plane::plane ();
```

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

```
public: plane::plane (
    plane const&            // plane name
    );
```

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

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

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

Destructor:

```
public: plane::~plane ();
```

C++ destructor, deleting a plane.

Methods:

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

Returns the number of derivatives that evaluate can find accurately (and directly), rather than by finite differencing, over the given portion of the surface. For a plane, all surface derivatives can be obtained accurately.

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

Returns a box around the surface.

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

Returns a box around the surface.

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

Reports if the surface is closed, smoothly or not, in the *u*-parameter direction. A plane is open in both directions.

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

Reports if the surface is closed, smoothly or not, in the *v*-parameter direction. A plane is open in both directions.

```
public: virtual void plane::debug (
    char const*,            // debug leader string
    FILE*                   // file pointer
        = debug_file_ptr
    ) const;
```

Prints out details of plane.

```
public: virtual surface* plane::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 void plane::eval (
    SPApar_pos const&,       // parameter position
    SPAposition&,            // position
    SPAvector*               // first derivatives –
        = NULL,              // array of length 2,
                             // in order xu, xv
    SPAvector*               // second derivatives –
        = NULL               // array of length 3,
                             // in order xuu, xuv, xvv
    ) const;
```

Finds the point on the plane corresponding to the given parameter values. It may also return the first and second derivatives at this point.

```
public: virtual int plane::evaluate (
    SPApar_pos const&,              // param position
    SPAposition&,                   // position
    SPAvector**                     // array of ptrs
        = NULL,
    int                             // number of
        = 0,                        // derivatives
                                    // required (nd)
    evaluate_surface_quadrant       // the evaluation
        = evaluate_surface_unknown  // location,
                                    // which
                                    // is not used
    ) const;
```

Finds the principal axes of curvature of the surface at a point with given parameter values. The function also determines curvatures in those directions. 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 - i.e. 2 for the first derivatives, 3 for the second, etc.

```
public: virtual double plane::eval_cross (
    SPApar_pos const&,       // parameter position
    SPAunit_vector const&    // direction
    ) const;
```

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

```
public: virtual SPAunit_vector plane::eval_normal (
    SPApar_pos const&       // parameter position
    ) const;
```

Returns the surface normal at a given point on the surface.

```
public: surf_princurv plane::eval_prin_curv (
    SPApar_pos const& param // parameter position
    ) const;
```

Finds the principal axes of curvature of the surface at a point with given parameter values. The function also determines curvatures in those directions.

```
public: virtual void plane::eval_prin_curv (
    SPApar_pos const&,      // parameter position
    SPAunit_vector&,        // first axis direction
    double&,                // curvature in first
                            // direction
    SPAunit_vector&,        // second axis direction
    double&                 // curvature in second
                            // direction
    ) const;
```

Finds the principal axes of curvature of the surface at a point with given parameter values. The function also determines curvatures in those directions.

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

Gets the curve used as a sweeping path. This is NULL for a plane, but the method is included for compatibility with other geometry classes.

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

Gets the type of sweeping path used for sweeping a plane.

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

Returns the sweeping rail for the plane. This is normal to the plane.

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

Indicates whether the parameter coordinate system of the surface is right or left-handed. With a right-handed system, at any point the outward normal 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: virtual surface* plane::make_copy () const;
```

Returns a copy of the plane.

```
public: virtual surface& plane::negate ();
```

Negates this plane; i.e. reverses the surface normal.

```
public: virtual surf_normcone plane::normal_cone (
    SPApar_box const&,          // parameter bounds
    logical                     // approximate
        = FALSE,                // results OK?
    SPAtransf const&            // plane transform
        = * (SPAtransf* ) NULL_REF
    ) const;
```

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

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

Transforms this plane by the given transform.

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

Returns a plane being (a copy of) this plane negated; i.e., with opposite normal.

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

Tests two surfaces for equality. This is not guaranteed to state equal for effectively equal surfaces, but is guaranteed to state not equal if the surfaces are not equal. The result can be used for optimization.

```
public: virtual SPApar_pos plane::param (
    SPAposition const&,         // position
    SPApar_pos const&           // parameter position
        = * (SPApar_pos* ) NULL_REF
    ) const;
```

Finds the parameter values corresponding to a point on a surface.

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

Determines if a plane is parametric. A plane is not a parametric surface, as surface characteristics are easy to calculate without and independent of parameter values.

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

Returns the period of a periodic parametric surface, or 0 if the surface is not periodic in the *u*-parameter or not parametric. A plane is not periodic in both directions.

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

Returns the period of a periodic parametric surface, or 0 if the surface is not periodic in the *v*-parameter or not parametric. A plane is not periodic in both directions.

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

Returns the parameter range of a portion of the surface in the bounding box. If a box is provided, the parameter range returned is 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 conventionally an empty interval is returned.

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

Returns the principal parameter range of a surface in the *u*-parameter direction.

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 conventionally an empty interval is returned.

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

Returns the principal parameter range of a surface in the *v*-parameter direction.

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 conventionally an empty interval is returned.

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

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

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

Reports whether the surface is periodic in the *u*-parameter direction; i.e., it is smoothly closed, so faces can run over the seam. A plane is not periodic in the *u*-direction.

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

Reports whether the surface is periodic in the *v*-parameter direction; i.e., it is smoothly closed, so faces can run over the seam. A plane is not periodic in the *v*-direction.

```
public: virtual logical plane::planar (
    SPAposition&,           // Point through which
                            // the plane passes
    SPAunit_vector&         // axis direction
    ) const;
```

Reports on whether the plane is planar.

```
public: virtual double plane::point_cross (
    SPAposition const&,         // position name
    SPAunit_vector const&,      // direction
    SPApar_pos const&           // parameter
        = * (SPApar_pos* ) NULL_REF
    ) const;
```

Returns the curvature of a curve in the surface through a given point normal to a given direction in the surface. The curvature of any curve on a plane is always zero.

```
public: virtual SPAunit_vector plane::point_normal (
    SPAposition const&,          // position
    SPApar_pos const&            // parameter position
        = * (SPApar_pos* ) NULL_REF
    ) const;
```

Returns the surface normal at a given point on the surface.

```
public: virtual void plane::point_perp (
    SPAposition const&,          // given position
    SPAposition&,                // position on
                                 // plane
    SPAunit_vector&,             // normal
                                 // direction at
                                 // point on plane
    surf_princurv&,              // surface principle
                                 // curve
    SPApar_pos const&            // param position
        = * (SPApar_pos* ) NULL_REF,
    SPApar_pos&                  // actual position
        = * (SPApar_pos* ) NULL_REF,
    logical f_weak               // weak flag
        = FALSE
    ) const;
```

Finds the point on the surface nearest to the given point. It may optionally return the normal to and principal curvatures of the surface at that point. Also returns the parameter values at the found point, if desired.

```
public: void plane::point_perp (
    SPAposition const& pos,      // given position
    SPAposition& foot,           // position on plane
    SPApar_pos const& param_guess   // param position
        = * (SPApar_pos* ) NULL_REF,
    SPApar_pos& param_actual     // actual position
        = * (SPApar_pos* ) NULL_REF,
    logical f_weak               // weak flag
        = FALSE
    ) const;
```

Finds the point on the surface nearest to the given point. It may optionally return the parameter value at that point.

```
public: void plane::point_perp (
    SPAposition const& pos,  // given position
    SPAposition& foot,       // position on plane
    SPAunit_vector& norm,    // normal direction at
                             // point on plane
    SPApar_pos const&        // param position
        param_guess          // possible position
        = * (SPApar_pos* ) NULL_REF,
    SPApar_pos& param_actual// actual position
        = * (SPApar_pos* ) NULL_REF,
    logical f_weak           // weak flag
        = FALSE
    ) const;
```

Finds the point on the surface nearest to the given point. It may optionally return the parameter value at that point.

```
public: surf_princurv plane::point_prin_curv (
    SPAposition const& pos,      // position
    SPApar_pos const&            // parameter position
        param_guess              //possible parameter
        = * (SPApar_pos* ) NULL_REF
    ) const;
```

Returns the principal directions and magnitudes of curvature at a given point on the surface. The curvature is zero everywhere on a plane, so the principal directions are rather arbitrary in this case.

```
public: virtual void plane::point_prin_curv (
    SPAposition const&,          // position
    SPAunit_vector&,             // direction
    double&,                     // parameter
    SPAunit_vector&,             // direction
    double&,                     // parameter
    SPApar_pos const&            // parameter position
        = * (SPApar_pos* ) NULL_REF
    ) const;
```

Returns the principal directions and magnitudes of curvature at a given point on the surface. The curvature is zero everywhere on a plane, so the principal directions are rather arbitrary in this case.

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

Restore the data for a plane from a save file. The restore_data function for each class is called in circumstances when the type of surface is known and there is one available to be filled in.

read_position	Root point
read_unit_vector	Normal vector to plane
if (restore_version_number < SURFACE_VERSION)	
// Old style	
else	
read_vector	*u* derivative vector
read_logical	reverse v measured with respect to right hand rule: either “forward_v” or ”reversed_v”
surface::restore_data	Generic surface data

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

Saves the id then calls save_data.

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

Saves the information for a plane to the save file.

```
public: virtual logical plane::singular_u (
    double                  // constant u parameter
    ) const;
```

Reports whether the surface parameterization is singular at the specified *u*-parameter value. The only singularity recognized is where every value of the non-constant parameter generates the same object-space point, and these can only occur at the ends of the parameter range as returned by the functions above. A plane is non-singular in both directions.

```
public: virtual logical plane::singular_v (
    double                  // constant v parameter
    ) const;
```

Reports whether the surface parameterization is singular at the specified *v*-parameter value. The only singularity recognized is where every value of the non-constant parameter generates the same object-space point, and these can only occur at the ends of the parameter range as returned by the functions above. A plane is non-singular in both directions.

```
public: virtual logical plane::test_point_tol (
    SPAposition const&,             // position
    double                          // tolerance
        = 0,                        // value
    SPApar_pos const&               // param position
        = * (SPApar_pos* ) NULL_REF,
    SPApar_pos&                     // param position
        = * (SPApar_pos* ) NULL_REF
    ) const;
```

Tests whether a point lies on the surface, to user-supplied precision. The function may optionally return the parametric position of the nearest point.

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

Returns the type of plane.

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

Returns string “plane”.

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

Classification of a plane.

```
public: SPAvector plane::u_axis () const;
```

Gets the *u*-parameter direction.

```
public: virtual curve* plane::u_param_line (
    double                  // parameter value
    ) 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 in the non-constant direction matches that of the surface, and has the range obtained by use of param_range_u or param_range_v appropriately. If the supplied constant parameter value is outside the valid range for the surface, or singularity, a NULL is returned.

Because the new curve is constructed in free store, it is the responsibility of the caller to ensure that it is correctly deleted.

```
public: virtual curve* plane::v_param_line (
    double                  // parameter value
    ) const;
```

Constructs a parameter line on the surface.

A *v*-parameter line runs in the direction of increasing *v*, at constant *u*. The parameterization in the non-constant direction matches that of the surface, and has the range obtained by use of param_range_u or param_range_v appropriately. If the supplied constant parameter value is outside the valid range for the surface, or singularity, a NULL is returned.

Because the new curve is constructed in free store, it is the responsibility of the caller to ensure that it is correctly deleted.

Internal Use: full_size

Related Fncs:

restore_cone

```
friend: plane operator* (
    plane const&,           // given plane
    SPAtransf const&        // transform
    );
```

Returns a plane that is (a copy of) the given plane transformed by the given transform.

position_array

Class: Mathematics

Purpose: Creates dynamic arrays of positions.

Derivation: position_array : ACIS_OBJECT : –

SAT Identifier: None

Filename: kern/kernel/geomhusk/posarray.hxx

Description: This class creates dynamic arrays of positions. There is an operator to cast a position_array to a SPAposition* so it can be used when a SPAposition* is needed, but it automatically grows as positions are added to it.

Limitations: None

References: BASE SPAposition

Data:

```
protected int m_nArraySize;
```
The initial size of the array.

```
protected int m_nNumPositions;
```
The number of positions in the initial size of the array.

```
protected SPAposition* m_pPositions;
```
The new position in the initial size of the array.

Constructor:

```
public: position_array::position_array ();
```

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

```
public: position_array::position_array (
    const position_array&   // position_array
    );
```

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

```
public: position_array::position_array (
    int initialSize         // initial size
    );
```

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

Destructor:

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

C++ destructor, deleting a position_array.

Methods:

```
public: int position_array::Add (
    const SPAposition& pos  // position
    );
```

Adds a SPAposition as the last position in the array, which expands the array as needed. Use the Add and RemoveLast methods for convenience to add positions to the array without keeping track of the current index.

```
public: SPAposition*
    position_array::CopyBuffer () const;
```

Copies the position_array. The user must delete the returned buffer when it is no longer needed.

```
public: SPAposition& position_array::ElementAt (
    int nIndex              // index value
    );
```

Gets the element at a given index in the array, which expands the size of the array if needed. The element returns as a reference so that it can be used on the left side of an assignment.

```
public: void position_array::Empty ();
```

Empties the array. This sets the number of positions in the array to 0, but it does not free up allocated storage.

```
public: SPAposition* position_array::GetBuffer ()
const;
```

Casts a position_array to a SPAposition* so that it can be used as an argument in procedures that require a SPAposition*. The returned pointer is only valid as long as the size of the array is not changed.

```
public: SPAposition position_array::GetLast () const;
```

Gets the last position in the array that is set.

```
public: int position_array::GetSize () const;
```

Returns the size of the position_array.

```
public: SPAposition& position_array::operator[] (
    int nIndex              // index value
    );
```

Makes the operator look like an array. Because this method returns a position, it can be used on the left side of an assignment. The array is expanded, if needed, if the index is too large.

```
public: operator position_array::SPAposition* ()
const;
```

Returns a copy of the position. The user must must delete the returned buffer when it is no longer needed.

```
public: SPAposition position_array::PositionAt (
    int nIndex              // index value
    ) const;
```

Provides a copy of the position at a given index. This method also provides access to a constant position_array. The index must be within the range.

```
public: int position_array::RemoveLast ();
```

Removes the last position in the array. Use this with the Add method.

```
protected: void position_array::SetMinSize (
    int size                // initial size
    );
```

Sets the size of the array so that it contains at least size positions. If the array is already the right size, the size does not change.

```
public: void position_array::SetSize (
    int size                // size
    );
```

Sets the size of the array., which expands the array, if needed. If size is smaller than the current size, this sets the number of positions, but it does not change the amount of allocated memory.

```
public: void position_array::Shrink ();
```

Truncates the size of the internal position buffer so that it is exactly big enough to hold the specified number of positions.

Related Fncs:

None