Chapter 36.
Classes Sa thru Sq

Topic: Ignore

SabFile

Class: SAT Save and Restore

Purpose: Performs save and restore to stream files.

Derivation: SabFile : BinaryFile : FileInterface : ACIS_OBJECT : –

SAT Identifier: None

Filename: kern/kernel/kernutil/fileio/sabfile.hxx

Description: This class performs ACIS save and restore to stream files using the new binary format that supports unknown ENTITY data.

Limitations: None

References: None

Data:

None

Constructor:

```
public: SabFile::SabFile (
    FILE*                   // file pointer
    );
```

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

Destructor:

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

C++ destructor, deleting a SabFile.

Methods:

```
public: virtual FilePosition SabFile::goto_mark (
    FilePosition            // file position
    );
```

Moves the file pointer to the specified position in the SabFile.

```
protected: virtual size_t SabFile::read (
    void* buf,              // buffer for data
    size_t length,          // memory size
    logical swap            // support byte swapping
    );
```

Reads data from a stream file in binary format.

```
public: virtual FilePosition SabFile::set_mark ();
```

Returns the current file position within the SabFile.

```
protected: virtual void SabFile::write (
    const void* data,       // pointer to data
    size_t len,             // memory size
    logical swap            // support byte swapping
    );
```

Writes data to a stream file in binary format.

Related Fncs:

None

SatFile

Class: SAT Save and Restore

Purpose: Defines the SatFile class for doing ACIS save and restore to stream files in text format.

Derivation: SatFile : FileInterface : ACIS_OBJECT : –

SAT Identifier: None

Filename: kern/kernel/kernutil/fileio/satfile.hxx

Description: Defines the SatFile class for doing ACIS save and restore to stream files in text format.

Limitations: None

References: None

Data:

None

Constructor:

```
public: SatFile::SatFile (
    FILE*                   // filename
    );
```

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

Destructor:

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

C++ destructor, deleting a SatFile.

Methods:

```
public: virtual FilePosition SatFile::goto_mark (
    FilePosition            // file position
    );
```

Moves the file pointer to the specified pointer in the SatFile.

```
public: virtual char SatFile::read_char ();
```

Reads a character. Written with C printf format “%c”.

```
public: virtual TaggedData* SatFile::read_data ();
```

Reads the data for an unknown ENTITY until the end of record terminator is reached.

```
public: virtual double SatFile::read_double ();
```

Reads a double. Written with C printf format “%g ”.

```
public: virtual int SatFile::read_enum (
    enum_table const&       // enumeration table
    );
```

Read an enumeration table. The <identifier> specifies which enumeration is active and its valid values. The <identifier> is not written to the file. A valid value only is written to the file. This is a character string or a long value from the enumeration <identifier> written with C printf format “%s”.

```
public: virtual float SatFile::read_float ();
```

Reads a float. Written with C printf format “%g ”.

```
public: virtual logical SatFile::read_header (
    int&,                   // first integer
    int&,                   // second integer
    int&,                   // third integer
    int&                    // fourth integer
    );
```

Reads a header. The first record of the ACIS save file is a header, such as:
200 0 1 0

First Integer: An encoded version number. In the example, this is “200”. This value is 100 times the major version plus the minor version (e.g., 107 for ACIS version 1.7). For point releases, the final value is truncated. Part save data for the .sat files is not affected by a point release (e.g., 105 for ACIS version 1.5.2).

Second Integer: The total number of saved data records, or zero. If zero, then there needs to be an end mark.

Third Integer: A count of the number of entities in the original entity list saved to the part file.

Fourth Integer: The least significant bit of this number is used to indicate whether or not history has been saved in this save file.

```
public: virtual int SatFile::read_id (
    char*,                  // title
    int                     // integer
        = 0
    );
```

Reads an identifier. The save identifier written with C printf format “%s ”.

```
public: virtual logical SatFile::read_logical (
    const char* f           // title
        = ”F”,
    const char* t           // title
        = ”T”
    );
```

Reads a logical. (false_string, true_string {or any_valid_string}): Appropriate string written with C printf format “%s ”.

```
public: virtual long SatFile::read_long ();
```

Reads a long. Written with C printf format “%ld”.

```
public: virtual void* SatFile::read_pointer ();
```

Reads a pointer. Pointer reference to a save file record index. Written as “$” followed by index number written as a long.

```
public: virtual int SatFile::read_sequence ();
```

Reads a sequence. Written as “–” followed by the entity index written as long.

```
public: virtual short SatFile::read_short ();
```

Reads a short. Written with C printf format “%d”.

```
public: virtual char* SatFile::read_string (
    int&                    // integer
    );
```

Reads a string, allocates memory for it, and the argument returns the length of the string. Length written as long followed by string written with C printf format “%s”.

```
public: virtual size_t SatFile::read_string (
    char* buf,              // buffer
    size_t maxlen           // maximum length
        = 0
    );
```

Reads a string into a supplied buffer of a given size, maxlen.

```
public: virtual logical SatFile::read_subtype_end ();
```

Reads subtype end. Braces around the subtypes, written as “} ”.

```
public: virtual logical
    SatFile::read_subtype_start ();
```

Reads subtype start. Braces around the subtypes, written as “{ ”.

```
public: virtual FilePosition SatFile::set_mark ();
```

Returns the current file position within the SatFile.

```
public: virtual void SatFile::write_char (
    char                    // character
    );
```

Writes a character. Written with C printf format “%c”.

```
public: virtual void SatFile::write_double (
    double                  // parameter
    );
```

Writes a real. Written with C printf format “%g ”.

```
public: virtual void SatFile::write_enum (
    int,                    // number in
    enum_table const&       // enumeration table
    );
```

Writes enumeration table. The <identifier> specifies which enumeration is active and its valid values. The <identifier> is not written to the file. A valid value only is written to the file. This is a character string or a long value from the enumeration <identifier> written with C printf format “%s”.

```
public: virtual void SatFile::write_float (
    float                   // float
    );
```

Writes a float. Written with C printf format “%g ”.

```
public: virtual void SatFile::write_header (
    int,                    // first integer
    int,                    // second integer
    int,                    // third integer
    int                     // fourth integer
    );
```

Writes a header. The first record of the ACIS save file is a header, such as: 200 0 1 0

First Integer: An encoded version number. In the example, this is “200”. This value is 100 times the major version plus the minor version (e.g., 107 for ACIS version 1.7). For point releases, the final value is truncated. Part save data for the .sat files is not affected by a point release (e.g., 105 for ACIS version 1.5.2).

Second Integer: The total number of saved data records, or zero. If zero, then there needs to be an end mark.

Third Integer: A count of the number of entities in the original entity list saved to the part file.

Fourth Integer: The least significant bit of this number is used to indicate whether or not history has been saved in this save file.

```
public: virtual void SatFile::write_id (
    const char*,            // character
    int                     // integer
    );
```

Writes an identifier. The save identifier written with C printf format “%s ”.

```
public: virtual void SatFile::write_literal_string (
    const char*,            // character
    size_t len              // length
        = 0
    );
```

Writes a literal string.

```
public: virtual void SatFile::write_logical (
    logical,                // logical
    const char* f           // character
        = "F",
    const char* t           // character
        = "T"
    );
```

Writes a logical. *(false_string, true_string, {or any_valid_string})*: Appropriate string written with C printf format "%s ".

```
public: virtual void SatFile::write_long (
    long                    // long
    );
```

Writes a long. Written with C printf format "%ld".

```
public: virtual void SatFile::write_newline (
    int                     // number of newlines
        = 1
    );
```

Writes a new line.

```
public: virtual void SatFile::write_pointer (
    void*                   // parameter
    );
```

Writes a pointer. Pointer reference to a save file record index. Written as "$" followed by index number written as a long.

```
public: virtual void SatFile::write_sequence (
    int                     // integer
    );
```

Writes a sequence. Written as "–" followed by the entity index written as long.

```
public: virtual void SatFile::write_short (
    short                   // short
    );
```

Writes a short. Written with C printf format “%d”.

```
public: virtual void SatFile::write_string (
    const char*,            // character
    size_t len              // length
        = 0
    );
```

Writes a string. Length written as long followed by string written with C printf format “%s”.

```
public: virtual void SatFile::write_subtype_end ();
```

Writes a subtype end. Braces around the subtypes, written as “} ”.

```
public: virtual void SatFile::write_subtype_start ();
```

Writes a subtype start. Braces around the subtypes, written as “{ ”.

```
public: virtual void SatFile::write_terminator ();
```

Writes a terminator. Written as “#”.

Related Fncs:

None

SHELL

Class: Model Topology, SAT Save and Restore

Purpose: Represents the external boundary of a LUMP, or the internal boundary of a void (unoccupied space) within a LUMP.

Derivation: SHELL : ENTITY : ACIS_OBJECT : –

SAT Identifier: “shell”

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

Description: The shell is a connected portion of a lump’s boundary. It has no physical or topological connection with any other shell. It is not possible to traverse the topological structure of one shell and end up on another shell. If a lump has no voids, then exactly one shell gives its overall extent. Any other shells bound voids wholly within the lump. There is no distinction made in the data structure between peripheral and void shells. In this context a shell is closed and bounded.

It is technically possible for a shell to be open and bounded or unbounded. If bounded, the containing lump (and body) is considered *incomplete*, or more accurately, *incompletely bounded*. It interacts with other bodies only so far as the defined portions of their shells interact. There are configurations of that interaction that are disallowed. If the shell is unbounded, it can be semi-infinite (e.g., a plane bounded by a single infinite straight line) or infinite (e.g., two half-infinite planes joined at their boundaries). If the shell is semi-infinite, the body is incomplete, while an infinite shell is completely defined, though of infinite extent.

The concepts of *peripheral* and *void* shells, and of *connected* and *disjoint* bodies have no meaning when applied to incomplete lump or body.

A shell is constructed from a collection of "faces" and "wires." Large collections may be subdivided into a hierarchy of "subshells," each containing a proper subcollection. A shell subdivided into subshells may also contain faces and wires directly; in this case, these entities are not contained in any subshell.

Limitations: None

References:

KERN	FACE, LUMP, SUBSHELL, WIRE
by KERN	FACE, LUMP, pattern_holder

Data:

None

Constructor:

```
public: SHELL::SHELL ();
```

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

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

Creates a SHELL, initializes all the class data, and records the creation in the bulletin board. The first two arguments are the starts of lists of FACEs and SUBSHELLs contained, and the last is a list of sister SHELLs already in the current LUMP. The calling routine must set lump_ptr and if desired, bound_ptr, using set_lump and set_bound.

Destructor:

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

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 SHELL(...) then later x–>lose.)

Methods:

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

Returns a pointer to a geometric bounding region (a box), within which the entire SHELL lies (with respect to the internal coordinate system of the BODY). The return may be NULL if the bound was not calculated since the SHELL was last changed.

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

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

```
public: logical SHELL::copy_pattern_down (
    ENTITY* target              // target
    ) const;
```

Copies the pattern through all children of the target entity.

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

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

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

Returns the first FACE in a complete enumeration of all the FACEs in the SHELL, continued by repeated use of FACE::next_face. The undefined order of SUBSHELLs fluctuates with each change of the SUBSHELL subdivision.

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

Returns a pointer to the first FACE of a list of FACEs immediately contained in this SHELL.

```
public: void SHELL::get_all_patterns (
    VOID_LIST& list             // list
    );
```

Returns all patterns in the list.

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

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

```
public: logical SHELL::is_closed () const;
```

Determine if the shell is closed or not. This method considers only single–sided faces. It ignores all double–sided faces and wires.

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

Returns TRUE if this can be deep copied.

```
public: LUMP* SHELL::lump () const;
```

Returns a pointer to the owning LUMP (SHELLs in separate LUMPs are entirely separate).

```
public: SHELL* SHELL::next (
    PAT_NEXT_TYPE next_type // shell type
        = PAT_CAN_CREATE    // for patterns
) const;
```

Returns a pointer to the next SHELL in the list of SHELLs contained in a BODY.

The next_type argument controls how the next method treats patterns, and can take any one of three values:

PAT_CAN_CREATE: if the next shell is to be generated from a pattern, create it if it doesn't yet exist and return its pointer.

PAT_NO_CREATE: if the next shell is to be generated from a pattern, but hasn't yet been created, bypass it and return the pointer of the next already–created shell (if any).

PAT_IGNORE: behave as though there is no pattern on the shell.

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

Returns a pointer to the owning LUMP.

```
public: logical SHELL::patternable () const;
```

Returns TRUE.

```
public: logical SHELL::remove_from_pattern ();
```

Removes the pattern element associated with this entity from the pattern. Returns FALSE if this entity is not part of a pattern element, otherwise TRUE.

Note *The affected entities are not destroyed, but are merely made independent of the pattern. The pattern itself is correspondingly modified to "drop out" the newly disassociated element.*

```
public: logical SHELL::remove_from_pattern_list ();
```

Removes this entity from the list of entities maintained by its pattern, if any. Returns FALSE if no pattern is found, otherwise TRUE.

```
public: logical SHELL::remove_pattern ();
```

Removes the pattern on this and all associated entities. Returns FALSE if no pattern is found, otherwise TRUE.

```
public: void SHELL::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.

In versions before 1.6, the next tag will be for a body, but put it in the lump pointer for now anyway, and fix it up later (in fix_pointers).

```
if (restore_version_number >= PATTERN_VERSION
    read_ptr                          Pointer to record in save file for
                                      APATTERN on loop
        if (apat_idx != (APATTERN*)(–1)))
        restore_cache();
read_ptr                              Pointer to record in save file for
                                      next SHELL in lump
read_ptr                              Pointer to record in save file for
                                      first SUBSHELL in shell
read_ptr                              Pointer to record in save file for
                                      first FACE in shell
if (restore_version_number >= WIREBOOL_VERSION)
    read_ptr                          Pointer to record in save file for
                                      first WIRE in shell
else                                  Pointer for first WIRE in shell is
                                      set to NULL
    read_ptr                          Pointer to record in save file for
                                      body owning the LUMP containing
                                      shell
```

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

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

```
public: void SHELL::set_face (
    FACE*                    // new FACE
    logical reset_pattern    // reset or not
        = TRUE
    );
```

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

```
public: void SHELL::set_lump (
    LUMP*                   // new owning LUMP
    logical reset_pattern   // reset or not
        = TRUE
    );
```

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

```
public: void SHELL::set_next (
    SHELL*                  // new sister SHELL
    logical reset_pattern   // reset or not
        = TRUE
    );
```

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

```
public: void SHELL::set_pattern (
    pattern* in_pat             //
    logical reset_pattern   // reset or not
        = TRUE
    );
```

Set the current pattern.

```
public: void SHELL::set_subshell (
    SUBSHELL*               // new SUBSHELL
    logical reset_pattern   // reset or not
        = TRUE
    );
```

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

```
public: void SHELL::set_wire (
    WIRE*                   // wire
    logical reset_pattern   // reset or not
        = TRUE
    );
```

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

```
public: SUBSHELL* SHELL::subshell () const;
```

Returns a pointer to the first SUBSHELL in a list of SUBSHELLs immediately contained within this SHELL.

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

Returns the string "shell".

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

Returns the first WIRE in a complete enumeration of all the WIREs in the SHELL, continued by repeated use of WIRE::next. The undefined order of SUBSHELLs fluctuates with each change of the SUBSHELL subdivision.

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

Returns a pointer to the first WIRE of a list of WIREs immediately contained in this SHELL.

Internal Use: first_face, save, save_common

Related Fncs: is_SHELL

skin_spl_sur

Class: Skinning and Lofting, Construction Geometry, SAT Save and Restore

Purpose: Defines a skin surface between a list of curves.

Derivation: skin_spl_sur : spl_sur : subtrans_object : subtype_object : ACIS_OBJECT : –

SAT Identifier: “skinsur”

Filename: kern/kernel/sg_husk/skin/skin_spl.hxx

Description: This class defines a skin surface between a list of curves.

Surface Parameterization

The surface parameterization is the *u*-direction corresponds to the parameterization of the curves to be skinned and the *v*-direction corresponds to the cubic Bezier between the skin-curves.

The input to this surface class are the curves to be skinned (all the curves are reparameterized to lie in [0.0 – 1.0] range), optional tangents (the magnitude of the curves’ tangents have to match on the ends) in *u*-direction, and the optional surfaces on which the curves lie. If surfaces containing the curves are provided, these determine the tangent directions in *v*.

Evaluation Process

The evaluation process is a three-step process, as described below.

Step 1

If any matching tangent magnitudes are given, the section curves (curves to be skinned) are reparameterized as follows: parameter *t* is the parameter on the original curve. Parameter *u* on the skin surface is determined such that the *u*-partial at each end of the skin surface is equal to the matching tangent magnitude.

$$t = f(u) = t_s * H_0(u) + m_0 * H_1(u) + m_1 * H_2(u) + t_e * H_3(u)$$

In the above expression, H_n are the *cubic* Hermite polynomials and t_s and t_e are the start and end parameter values of the original curves to be skinned, which here are 0 and 1, respectively.

So, *ds*/*du* on the ends are:

$$dc/du = dc/dt * dt/du$$

where the *dt*/*du* values on the ends are m_0 and m_1.

So by choosing the values m_0 and m_1 such that the *dc*/*du* on the left surface is equal to *dc*/*du* on the right surface (provided that the curves are G1), a C1 continuous surface is achieved even when skinning G1 continuous curves.

Step 2

The tangent directions for the *v* are determined by fitting a circle through the points corresponding to the same *u*-value on the adjacent section curves to the left and right. The scheme followed is similar to the way Bessel tangents are computed. If there are only two section curves, the circle radius is chosen to be infinity. If the surfaces are given for any section, the tangent direction in *v* when on that curve is obtained by the cross product of surface normal and the section curve tangent at that point. The direction also has an optional scalar value that can be applied. The surface is called a loft surface when such a surface is provided.

Step 3

Now the skin/loft surface is defined using cubic Hermite interpolants between sections that join each other C1 continuously. To evaluate the surface $s(u,v)$ at a particular *v*-parameter, the first step is to find the segment to which this parameter corresponds. Then a local parameter v_i is computed, which ranges from 0 to 1. The section curves c_i and c_{i+1}, and the *v*-tangents t_i and t_{i+1} are also obtained. The surface is defined as:

$$s(u,v) = c_i(u)*H_0(v_i) + t_i*H_1(v_i) + t_{i+1}*H_2(v_i) + c_{i+1}*H_3(v_i)$$

The parametric derivatives of this surface are obtained by differentiating the above equation algebraically.

Limitations: None

References:

KERN	pcurve, surface
BASE	SPAvector
LAW	law

Data:

```
protected VOID_LIST curves;
```
The list of curves to be skinned.

```
protected VOID_LIST path_curves;
```
Array of curves used to define the path for skinning.

```
protected int no_path_crv;
```
Value specifies how many path curves are in entity list.

```
protected law **laws;
```
Array of laws used to define how skinning is performed.

```
protected double *deriv_cache;
```
Section mapping information.

```
protected double *matching_derivs;
```
Section mapping information.

```
protected double *tan_factors;
```
An array of factors applied to the cross boundary tangents.

```
protected double *v_knots;
```
An array of reals indicating the knot values at each section that is being interpolated.

```
protected int no_crv;
```
The number of curves to be skinned.

```
protected pcurve **pcurves;
```
Pcurves corresponding to the curves and surfaces are stored.

```
protected logical arc_length_param;
```
Flag for using arc-length parameterization. When set, arc-length parameterization is used. When clear, arc-wise parameterization is used.

```
protected logical perpendicular_option;
```
Flag for the loft direction. When set, it is perpendicular to the curve; when clear, it is in the curve direction.

```
protected logical skin_2p1_surface;
```
Flag to indicate this is an old skin surface.

```
protected surface **surfaces;
```
An array of pointers to the surface. The curves that are skinned lie on these surfaces and the surfaces are used to obtain cross boundary tangents. If this is NULL, the cross boundary tangents are calculated on the fly.

```
protected skin_data *surface_data;
```
Data cache for computing the optimal tangent factors for the skin surface.

```
protected SPAvector *tangents;
```
An array of cross-boundary tangents. If these are NULL, the tangents at each point are calculated on the fly.

Constructor:

```
protected: skin_spl_sur::skin_spl_sur ();
```

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

```
public: skin_spl_sur::skin_spl_sur (
    const skin_spl_sur&     // surface to copy
    );
```

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

Destructor:

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

C++ destructor, deleting a skin_spl_sur.

Methods:

```
public: int skin_spl_sur::accurate_derivs (
    SPApar_box const&           // area for deriv
        = * (SPApar_box* ) NULL_REF
    ) const;
```

Calculates the derivatives within the given parameter box.

```
public: void skin_spl_sur::add_path_data (
    int no_curves,          // number of curves in
                            // path
    curve** curves          // curve paths to add
    );
```

Adds the path data to the skin_spl_sur object.

```
protected: virtual void
 skin_spl_sur::calculate_disc_info ();
```

Calculates the discontinuity information from the defining curves.

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

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

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

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

```
protected: virtual void skin_spl_sur::debug (
    char const*,            // leader string
    logical,                // brief output OK?
    FILE*                   // output pointer
    ) const;
```

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

```
protected: void skin_spl_sur::debug_data (
    char const*,            // leader string
    logical,                // brief output ok?
    FILE*                   // output pointer
    ) const;
```

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

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

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

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: double
    skin_spl_sur::estimate_min_rad_curv ();
```

This function estimates the minimum radius of curvature of the skin surface for a given set of tangent factor values.

```
public: void skin_spl_sur::estimate_tanfacs_scale (
    SPAinterval& tan_range      // range to use
    );
```

Estimates the scaling factor range by which the tangent factors controlling the skin surface should be scaled so as to get the surface with as large a radius of curvature as possible.

```
protected: virtual void skin_spl_sur::eval (
    SPApar_pos const& uv,    // param space location
    SPAposition& pos,        // returned point
    SPAvector* dpos,         // first derivatives
    SPAvector* ddpos         // second derivatives
    ) const;
```

Finds the position and the first and second derivatives of the surface at a specified point. dpos is of length 2, ddpos is of length 3. Upon return, dpos contains x_u and x_v. ddpos contains x_{uu}, x_{uv}, x_{vv}.

```
protected: virtual int skin_spl_sur::evaluate (
    SPApar_pos const&,              // parameter on
                                    // surface point
    SPAposition&,                   // for deriv
    SPAvector**                     // first
        = NULL,                     // derivative
    int                             // second
        = 0,                        // derivative
    evaluate_surface_quadrant       // quadrant of
        =  evaluate_surface_unknown // discontinuity
                                    // to evaluate
    ) const;
```

The evaluate 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.

```
protected: void skin_spl_sur::eval_2p1_skin (
    SPApar_pos const& uv,    // param space location
    SPAposition& pos,        // returned point
    SPAvector* dpos,         // first derivatives
    SPAvector* ddpos         // second derivatives
    ) const;
```

Finds the position and first and second derivatives of the ACIS 2.1 skin surface at a given point.

```
protected: void skin_spl_sur::eval_skin (
    SPApar_pos const& uv,    // parameter on surface
    SPAposition& pos,        // point for deriv
    SPAvector* dpos,         // first deriv. array of
                             // length 2 in order
                             // xu, xv
    SPAvector* ddpos,        // second deriv. array of
                             // length 3 in order
                             // xuu, xuv, xvv
    SPAvector* dddpos,       // third deriv. array of
                             // length
    evaluate_surface_quadrant   // which quadrant to
        quadrant                // evaluate
    ) const;
```

Finds the position and first and second derivatives of the skin surface at the given parameter position value.

```
public: void skin_spl_sur::get_curves (
    int& no_crv,             // number of curves
    curve** *curves          // output array pointer
    ) const;
```

Returns the surface curves.

```
public: void skin_spl_sur::get_laws (
    int& no_laws,            // number of laws
    law**&  laws             // list of laws pointer
    ) const;
```

Returns a list of laws used by the skin_spl_sur. The use count of the laws is incremented by one.

```
public: void skin_spl_sur::get_surfaces (
    int& no_surfaces,        // number of surfaces
    surface**&  surf_arr     // surface array pointer
    ) const;
```

Returns the surfaces. The array of surfaces need to be deleted by the calling routine.

```
public: void skin_spl_sur::get_tanfacs (
    double* tangents         // tangent factors
    );
```

Get the tangent factors to determine optimal values for them.

```
public: void skin_spl_sur::get_v_knots (
    double u,                // u parameter
    int& out_no_knots,       // number of knots
    double** out_vknots      // output array pointer
    ) const;
```

Returns the v_knot sequence for a given parameter value.

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

Returns the ID for the skin_spl_sur list.

```
protected: void skin_spl_sur::initialize ();
```

Initializes the member data for this class.

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

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

```
public: static spl_sur*
    skin_spl_sur::make_skin_spl_sur (
    logical,                // arc-length option
    int,                    // number of curves
    curve**,                // array of curves
    double*,                // array of knot values
    double*,                // array of left tangents
    double*,                // array of rt tangents
    SPAvector*              // array of tangent dirs.
        = (SPAvector* ) NULL,
    closed_forms            // Flag for periodicity
        = OPEN,             // of surface in u
    closed_forms            // Flag for periodicity
        = OPEN              // of surface in v
    );
```

Constructs a skin surface from the given section curves and the optional matching tangents.

```
public: static spl_sur*
    skin_spl_sur::make_skin_spl_sur (
    logical,                // arc-length flag
    logical,                // perpendicular flag
    int,                    // # section curves
    curve**,                // array of curves
    double*,                // array of knot values
    double*,                // array of left tangents
    double*,                // right tangents array
    surface**,              // array of surfaces
    double*,                // cr-bnd tangents array
    law**,                  // array of laws
    closed_forms            // Flag for periodicity
        = OPEN,             // of surface in u
    closed_forms            // Flag for periodicity
        = OPEN              // of surface in v
    );
```

Constructs a loft surface from the given section curves and the corresponding surfaces on which the sections lie and the optional tangent factors that should be applied to cross-boundary tangents. The surfaces are passed in are owned by the skin_surface, so the user should pass in a copy. All arrays are the size of int, the number of sections.

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

Transforms this surface by the specified transform.

```
protected: logical skin_spl_sur::operator== (
    subtype_object const&   // object sub-type
    ) const;
```

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

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

Finds the parameter values of a point on a 3D B–spline surface, iterating from the given parameter values, if supplied.

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

Restore the data for a skin_spl_sur from a save file.

```
if (restore_version_number >= ARCWISE_SKIN_VERSION )
    read_logical                    // skin_2p1_surface, ”FALSE” or
                                    ”TRUE”
    read_logical                    // arc length parameter, ”ISO” or
                                    ”ARC”
    read_logical                    // perpendicular option, ”SKIN” or
                                    ”PERPENDICULAR”
read_int                            // number of curves
for( int i = 0; i < no_crv; i ++ ) {    // for each curve
    read_real                       // tangent length at start of curve or
                                    –1
    read_real                       // tangent length at end of curve or
                                    –1
    read_real                       // matching tangent length at start
                                    of curve or –1
    read_real                       // matching tangent length at end
                                    of curve or –1
    read_real                       // v knots
    restore_curve                   // restore the underlying curve
    read_vector                     // tangent vector
    restore_surface                 // underlying surface
    read_real                       // tangent factor
    if (restore_version_number >= LOFT_LAW_VERSION )
        restore_law                 // restore law definition if available
    if (restore_version_number >= LOFT_PCURVE_VERSION )
        restore_pcurve              // restore pcurve definition if
                                    available
    if (restore_version_number >= LOFT_LAW_VERSION ) {
        read_int                    // number of path curves
    for(int i = 0; i < no_path_crv; i ++ ) // for each path curve
        restore_curve               // restore path curve to be skinned
                                    or lofted
spl_sur::restore_common_data        Restore the rest of the surface
```

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

Saves the skin_spl_sur as an approximation if there is a need to approximate.

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

Saves the information for skin_spl_sur to the save file.

```
public: void skin_spl_sur::set_tanfacs (
    double* tangents,        // tangent factors
    logical remake_approx    // remake enabled
        = TRUE               // or not
    );
```

Set the tangent factors and get them for purposes of determining optimal values for them.

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

Adjusts the spline surface to have a parameter range increased by the shift value, which may be negative. This method is only used to move portions of a periodic surface by integral multiples of the period.

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

Adjusts the spline surface to have a parameter range increased by the shift value, which may be negative. This method is only used to move portions of a periodic surface by integral multiples of the period.

```
protected: virtual void skin_spl_sur::split_u (
    double,                  // u-parameter value
    spl_sur* [ 2 ]           // two spline surfaces
    );
```

Divides a surface into two pieces at the *u*-parameter value.

```
protected: virtual void skin_spl_sur::split_v (
    double,                  // v-parameter value
    spl_sur* [ 2 ]           // two spline surfaces
    );
```

Divides a surface into two pieces at the specified *v*-parameter value.

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

Returns the type of skin_spl_sur.

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

Returns the string of the given spline surface type, which is “skinsur” for a skin_spl_surf.

Internal Use: arclength_index_end, arclength_index_general, arclength_index_start, calculate_arcwise_data, calculate_iso_data, compute_bernstein_polynomials, deep_copy_elements_skin, full_size, remap_and_eval, sg_calculate_surface_normal_dervs, sg_recalculate_vknots_and_dervs

Related Fncs: restore_skin_spl_sur

SPHERE

Class: Model Geometry, SAT Save and Restore

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

Derivation: SPHERE : SURFACE : ENTITY : ACIS_OBJECT : –

SAT Identifier: “sphere”

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

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

SPHERE is one of several classes derived from SURFACE to define a specific type of surface. The sphere class defines a sphere by its center point and radius.

Along with the usual SURFACE and ENTITY class methods, SPHERE has member methods to provide access to specific implementations of the geometry. For example, methods are available to set and retrieve the center and radius of a sphere.

A use count allows multiple references to a SPHERE. The construction of a new SPHERE 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 sphere

Data:

None

Constructor:

```
public: SPHERE::SPHERE ();
```

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: SPHERE::SPHERE (
    SPAposition const&,      // center point
    double                   // radius
    );
```

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.

```
public: SPHERE::SPHERE (
    sphere const&            // sphere
    );
```

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

Destructor:

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

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 SPHERE(...) then later x–>lose.)

Methods:

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

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

```
public: SPAposition const& SPHERE::centre () const;
```

Returns the center of the SPHERE.

```
public: virtual void SPHERE::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& SPHERE::equation () const;
```

Returns the surface equation of a SPHERE.

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

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

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

Returns TRUE if this can be deep copied.

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

Makes a bounding box for this surface. The box contains the complete underlying surface and ignores the bounding EDGEs, unless the tight_sphere_box option is on.

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

Transforms a SPHERE. 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: double SPHERE::radius () const;
```

Returns the radius of the SPHERE.

```
public: void SPHERE::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.

sphere::restore_data	low-level geometry definition for sphere data.

```
public: void SPHERE::set_centre (
    SPAposition const&      // center point
    );
```

Sets the SPHERE's center 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: void SPHERE::set_radius (
    double                  // radius
    );
```

Sets the SPHERE's radius to the given value. 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* SPHERE::trans_surface (
    SPAtransf const&        // transform
        = * (SPAtransf* ) NULL_REF,
    logical                 // reversed
        = FALSE
    ) const;
```

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

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

Returns the string "sphere".

Internal Use: full_size

Related Fncs: is_SPHERE

sphere

Class: Construction Geometry, SAT Save and Restore

Purpose: Defines a spherical surface.

Derivation: sphere : surface : ACIS_OBJECT : –

SAT Identifier: “sphere”

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

Description: A sphere is defined by a center point and radius. A positive radius indicates an outward pointing surface normal. A negative radius indicates an inward pointing surface normal.

Five data members define the parameterization of the sphere and they are described in “Data.”

The *u*-parameter is the latitude metric, running from –pi/2 at the south pole through 0 at the equator to pi/2 at the north pole. The *v*-parameter is the longitude metric, running from –pi to pi, with 0 on the meridian containing ori_dir, and increasing in a clockwise direction around pole_dir, unless reverse_v is TRUE.

Let *P* be pole_dir and *Q* ori_dir, and let *R* be *P* x *Q*, negated if reverse_v is TRUE. Let *r* be the absolute value of the sphere radius. Then:

```
pos = center + r* sin(u)* P + r* cos(u)*
        (cos(v)* Q + sin(v) R)
```

This parameterization is left-handed for a convex sphere and right-handed for a hollow one, if reverse_v is FALSE, and reversed if it is TRUE.

When the sphere is transformed, the sense of reverse_v is inverted if the transform includes a reflection. No special action is required for a negation.

In summary, spheres are:

– Not true parametric surfaces.
– Periodic in *v* (–pi to pi with period 2 * pi) but not in *u*.
– Closed in *v* but not in *u*.
– Singular in *u* at the poles; nonsingular everywhere else.

Limitations: None

References:

by KERN	SPHERE
BASE	SPAposition, SPAunit_vector

Data:

```
public logical reverse_v;
```
Constant *u*-parameter lines are circles around pole_dir, normally clockwise, but counterclockwise if this is TRUE.

```
public SPAposition centre;
```
The center of the sphere.

```
public double radius;
```
The radius of a sphere. If negative, the surface normal points inward to the center of the sphere.

```
public SPAunit_vector pole_dir;
```
Direction normal to uv_oridir that points from the center to the “north pole” of the sphere; i.e., the maximum-*u* singularity.

```
public SPAunit_vector uv_oridir;
```
Direction from the center of the sphere to the origin of parameter space.

Constructor:

```
public: sphere::sphere ();
```

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

```
public: sphere::sphere (
    SPAposition const&,      // position
    double                   // radius
    );
```

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

```
public: sphere::sphere (
    sphere const&            // given sphere
    );
```

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

Destructor:

```
public: sphere::~sphere ();
```

C++ destructor, deleting a sphere.

Methods:

```
public: virtual int sphere::accurate_derivs (
    SPApar_box const&       // parameter box name
        = * (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 curve. If there is no limit to the number of accurate derivatives, returns the value ALL_SURFACE_DERIVATIVES. This is the case with a sphere.

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

Returns a box that encloses the surface in object space.

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

Returns a box that encloses the surface in parameter space.

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

Reports whether the surface is closed, smoothly or not, in the *u*-parameter direction. A sphere is open in the *u*-direction.

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

Reports whether the surface is closed, smoothly or not, in the *v*-parameter direction. A sphere is closed in the *v*-direction

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

Prints out details of sphere.

```
public: virtual surface* sphere::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 sphere::eval (
    SPApar_pos const&,      // parameter position
    SPAposition&,           // position
    SPAvector*              // 1st derivatives array
        = NULL,             // length 2, in order xu,
                            // xv
    SPAvector*              // second derivatives –
        = NULL              // array of length 3, in
                            // order xuu, xuv, xvv
    ) const;
```

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

```
public: virtual int sphere::evaluate (
    SPApar_pos const&,              // param position
    SPAposition&,                   // pt on surface
                                    // at given
                                    // param position
    SPAvector**                     // array of ptrs
        = NULL,                     // to arrays of
                                    // vectors
    int                             // number of
        = 0,                        // derivatives
                                    // required
    evaluate_surface_quadrant       // the evaluation
                                    // location
                                    // above, below
                                    // for each
                                    // parameter
                                    // dir., or don't
        = evaluate_surface_unknown  // care.
    ) const;
```

Calculates derivatives, of any order up to the number requested, and stores them in vectors provided by the user. The function returns the number of derivatives calculated. 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 sphere::eval_cross (
    SPApar_pos const&,      // parameter position
    SPAunit_vector const&   // direction
    ) const;
```

Finds the curvature of a cross-section curve of the parametric surface 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.

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

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

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

Finds the principle axes of curvature and the curvatures in those directions of the surface at a point with given parameter values. For a sphere, the curvature in every direction is a constant, so the direction of the principle axes is arbitrary.

```
public: virtual void sphere::eval_prin_curv (
    SPApar_pos const&,      // parameter
    SPAunit_vector&,        // first axis direction
    double&,                // curvature in the first
                            // direction
    SPAunit_vector&,        // second axis direction
    double&                 // curvature in the 2nd
                            // direction
    ) const;
```

Finds the principle axes of curvature and the curvatures in those directions of the surface at a point with given parameter values. For a sphere, the curvature in every direction is a constant, so the direction of the principle axes is arbitrary.

```
public: logical sphere::hollow () const;
```

Determines if a sphere is hollow.

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

Indicates whether the parameter coordinate system of the surface is right-handed 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* sphere::make_copy () const;
```

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

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

Negates this sphere.

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

Returns a cone bounding the normal direction of a curve.

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

Transforms this sphere by the given transform.

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

Returns a copy of this sphere negated; i.e., with normal reversed.

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

Tests two surfaces for equality.

This, like testing floating point numbers for equality, is not guaranteed to say equal for effectively equal surfaces, but is guaranteed to say not equal if they are indeed not equal. The result can be used for optimization, but not where it really matters.

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

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

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

Determines if a sphere is parametric. A sphere is not considered to be parametric.

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

Return the period of a periodic parametric surface, or 0 if the surface is not periodic in the *u*-parameter or not parametric. A sphere is not periodic in the *u*-direction.

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

Return the period of a periodic parametric surface, or 0 if the surface is not periodic in the *v*-parameter or not parametric. A sphere has a period of 2 * pi in the *v*-direction.

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

Returns the parameter ranges of the portion of a surface lying within the given box.

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

Returns the parameter ranges of the portion of a surface that lies within the given box in a *u*-parameter direction.

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

Returns the parameter ranges of the portion of a surface that lies within the given box in a *v*-parameter direction.

```
public: virtual SPApar_vec sphere::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 sphere::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 sphere is not periodic in the *u*-direction.

```
public: virtual logical sphere::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 sphere is not periodic in the *v*-direction.

```
public: virtual double sphere::point_cross (
    SPAposition const&,     // position
    SPAunit_vector const&,  // direction
    SPApar_pos const&       // parameter position
        = * (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 a sphere is 1/radius in all directions, at all locations.

```
public: virtual SPAunit_vector sphere::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 sphere::point_perp (
    SPAposition const&,      // first position
    SPAposition&,            // second position
    SPAunit_vector&,         // direction
    surf_princurv&,          // surf. principle curve
    SPApar_pos const&        // parameter position
        = * (SPApar_pos* ) NULL_REF,
    SPApar_pos&              // parameter position
        = * (SPApar_pos* ) NULL_REF,
    logical f_weak           // weak flag
        = FALSE
    ) const;
```

Finds the point on the surface nearest to the given point. Optionally, the function may determine the normal to and principal curvatures of the surface at that point. If the surface is parametric, also return the parameter values at the found point.

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

Finds the point on the surface nearest to the given point. If the surface is parametric, also return the parameter values at the found point.

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

Finds the point on the surface nearest to the given point. Optionally, the function may determine the normal to the surface at that point. If the surface is parametric, also return the parameter values at the found point.

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

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

```
public: virtual void sphere::point_prin_curv (
    SPAposition const&,      // position
    SPAunit_vector&,         // first axis direction
    double&,                 // curvature in first
                             // direction
    SPAunit_vector&,         // second axis direction
    double&,                 // curvature in second
                             // direction
    SPApar_pos const&        // parameter position
        = * (SPApar_pos* ) NULL_REF
    ) const;
```

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

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

Restore the data for a sphere from a save file.

```
read_position                       Center of sphere
read_real                           Radius of sphere
if (restore_version_number < SURFACE_VERSION)
    // Old style – that is all to read.
else
    read_unit_vector                uv x-axis
    read_unti_vector                pole direction (z-axis)
    read_logical                    Reverse v; either “forward_v” or
                                    “reversed_v”
surface::restore_data               Restore remainder of surface data
```

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

Saves the type or id, then calls save_data.

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

Saves the information for sphere in the save file.

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

Reports whether the surface parameterization is singular at the specified *u*-parameter value. A sphere is singular in *u* at both poles.

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

Reports whether the surface parameterization is singular at the specified *v*-parameter value. A sphere is not singular in *v* anywhere.

```
public: virtual logical sphere::test_point_tol (
    SPAposition const&,     // position
    double                  // parameter
        = 0,
    SPApar_pos const&       // parameter position
        = * (SPApar_pos* ) NULL_REF,
    SPApar_pos&             // parameter
        = * (SPApar_pos* ) NULL_REF
    ) const;
```

Tests whether a point lies on the surface, to user-supplied precision.

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

Returns the type of sphere.

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

Returns the string “sphere”.

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

Verifies if the sphere is properly defined.

```
public: virtual curve* sphere::u_param_line (
    double                  // constant v-parameter
    ) const;
```

Constructs a parameter line on the surface.

A *u*-parameter line runs in the direction of increasing *u*-parameter, at constant v. The parameterization in the nonconstant direction matches that of the surface, and has the range obtained by use of param_range_u. The new curve is constructed in free store, so it is the responsibility of the caller to ensure that it is correctly deleted.

```
public: virtual curve* sphere::v_param_line (
    double                  //constant u-parameter
    ) 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 nonconstant direction matches that of the surface, and has the range obtained by use of param_range_v. The new curve is constructed in free store, so it is the responsibility of the caller to ensure that it is correctly deleted.

Internal Use: full_size

Related Fncs: restore_cone

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

Transforms a sphere surface.

SPLINE

Class: Model Geometry, SAT Save and Restore

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

Derivation: SPLINE : SURFACE : ENTITY : ACIS_OBJECT : –

SAT Identifier: “spline”

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

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

SPLINE is one of several classes derived from SURFACE to define a specific type of surface. The spline class holds a pointer to a spl_sur and a logical denoting the sense of the stored surface. A spl_sur also contains a use count and a detailed parametric surface description.

Along with the usual SURFACE and ENTITY class methods, SPLINE has member methods to provide access to specific implementations of the geometry. For example, a spline can be transformed, resulting in another surface. All access to the surface data is through methods for the spline class.

A use count allows multiple references to a SPLINE. The construction of a new SPLINE 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 spline

Data:

None

Constructor:

```
public: SPLINE::SPLINE ();
```

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: SPLINE::SPLINE (
    spline const&           // spline object
    );
```

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

Destructor:

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

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 SPLINE(...) then later x–>lose.)

Methods:

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

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

```
public: virtual void SPLINE::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& SPLINE::equation () const;
```

Returns the surface equation if the SPLINE for reading only.

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

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

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

Returns TRUE if this can be deep copied.

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

Creates a bounding box for this surface that is surrounded by a loop of EDGEs. The box contains the complete underlying surface, and ignores the bounding EDGEs. If the surface is kept minimal, this is sufficient.

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

Transforms the SPLINE in place. 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 SPLINE::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 < SURFACE_VERSION)	
// Old style – the reverse bit was read explicitly, and	
// forgotten when reading a lower–case spline normally.	
read_int	Reverse bit
spline::restore_data	Low-level spline surface geometry definition.
if (reverse bit)	
spline::negate	Change definition of underlying spline.

```
public: void SPLINE::set_def (
    spline const&           // spline
    );
```

Sets the definition spline to the given spline. 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* SPLINE::trans_surface (
    SPAtransf const&        // transform
        = * (SPAtransf* ) NULL_REF,
    logical                 // reversed
        = FALSE
    ) const;
```

Returns a new surface that is the spline of the SPLINE. If the SPAtransf is non-NULL, it is transformed. If logical is TRUE, it is reversed in sense.

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

Returns the string “spline”.

Internal Use: full_size

Related Fncs: is_SPLINE

spline

Class: Construction Geometry, SAT Save and Restore

Purpose: Records a B–spline surface.

Derivation: spline : surface : ACIS_OBJECT : –

SAT Identifier: “spline”

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

Description: The spline class represents a parametric surface that maps a rectangle within a 2D real vector space (parameter space) into a 3D real vector space (object space). This mapping must be continuous, and one-to-one except possibly at the boundary of the rectangle in parameter space. It is differentiable twice, and the normal direction is continuous, though the derivatives need not be. The positive direction of the normal is in the sense of the cross product of the partial derivatives with respect to *u* and *v* in that order. The portion of the neighborhood of any point on the surface that the normal points to is outside the surface, and the other part is inside.

Opposite sides of the rectangle can map into identical lines in object space, in which case the surface is closed in the parameter direction normal to those boundaries. If the parameterization and derivatives also match at these boundaries, the surface is periodic in this parameter direction. The line in object space corresponding to the coincident boundaries is known as the seam of a periodic surface.

If a surface is periodic in one parameter direction, it is defined for all values of that parameter. A parameter value outside the domain rectangle is brought within the rectangle by adding a multiple of the rectangle's width in that parameter direction, and the surface evaluated at that value. If the surface is periodic in both parameters, it is defined for all parameter pairs (*u*,*v*), with reduction to standard range happening with both parameters.

One side of the rectangle can map into a single point in object space. This point is a parametric singularity of the surface. If the surface normal is not continuous at this point, it is a surface singularity.

The spline contains a "reversed" bit together with a pointer to another structure, a spl_sur or something derived from it, that contains the bulk of the information about the surface.

Providing this indirection serves two purposes. First, when a spline is duplicated, the copy simply points to the same spl_sur and does not copy the bulk of the data. The system maintains a use count in each spl_sur. This allows automatic duplication if a shared spl_sur is to be modified, and deletes any spl_sur no longer accessible.

Second, the spl_sur contains virtual functions to perform all spline operations defined that depend on the method of definition of the true surface. Therefore, new surface types can be defined by declaring and implementing derived classes. The spline and everything using it require no changes to make use of the new definition.

Limitations: None

References:
KERN discontinuity_info, spl_sur
by KERN SPLINE, spl_sur, sub_spl_sur

Data:

None

Constructor:

```
public: spline::spline ();
```

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

```
public: spline::spline (
    bs3_surface             // bs3 surface
    );
```

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

Construct a spline from a bs3_surface, which is the type that represents the fundamental parametric surface. The resulting spline surface is taken to be exactly the bs3_surface supplied. After construction, the bs3_surface is owned by the spline object, so should not be reused or deleted by the caller.

```
public: spline::spline (
    spline const&           //given spline
    );
```

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

```
public: spline::spline (
    spl_sur*                // spline surface
    );
```

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

Construct a spline from a pointer to an underlying spl_sur (usually in fact a class derived from it). This adds one new reference to the spl_sur, for the purposes of eventual deletion.

Destructor:

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

C++ destructor, deleting a spline.

Methods:

```
public: virtual int spline::accurate_derivs (
    SPApar_box const&       // default to the surface
        = * (SPApar_box* ) NULL_REF
    ) const;
```

Return the number of derivatives that evaluate can find accurately and fairly directly, rather than by finite differencing, over the given portion of the surface. If there is no limit to the number of accurate derivatives, returns the value ALL_SURFACE_DERIVATIVES.

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

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

```
public: virtual const double*
    spline::all_discontinuities_v (
    int& n_discont,          // number of disc
    int order                // order
    );
```

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

```
public: SPAbox spline::bound (
    SPAtransf const&,        // transform
    SPApar_box const&        // parameter range
        = * (SPApar_box* ) NULL_REF
    ) const;
```

Return a box around the spline. This is retained for historical reasons—it exactly parallels the previous virtual function.

```
public: virtual SPAbox spline::bound (
    SPAbox const&,           // object space box
    SPAtransf const&         // transform
        = * (SPAtransf* ) NULL_REF
    ) const;
```

Return a box that encloses the portion of the surface that lies within the given box after transformation.

```
public: virtual SPAbox spline::bound (
    SPApar_box const&        // parameter range
        = * (SPApar_box* ) NULL_REF,
    SPAtransf const&         // transform
        = * (SPAtransf* ) NULL_REF
    ) const;
```

Return a box that encloses the portion of the surface within the given range after transformation.

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

Notifies the derived type that the surface has been changed (e.g. the subset_range has changed) so that it can update itself.

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

Check for any data errors in the 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.

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

Report whether the surface is closed, smoothly or not, in the *u*-parameter direction.

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

Report whether the surface is closed, smoothly or not, in the *v*-parameter direction.

```
public: logical spline::contains_pipe () const;
```

Returns TRUE if this spline depends on a pipe surface.

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

Print out details of a spline.

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

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

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

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

```
public: virtual int spline::discontinuous_at_u (
    double u                // location
    ) const;
```

Returns whether a particular parameter value is a discontinuity.

```
public: virtual int spline::discontinuous_at_v (
    double v                // location
    ) const;
```

Returns whether a particular parameter value is a discontinuity.

```
public: virtual void spline::eval (
    SPApar_pos const& uv,    // parameter
    SPAposition& pos,        // position
    SPAvector* dpos          // 1st derivatives array
        = NULL,              // length 2 in order xu,
                             // xv
    SPAvector* ddpos         // second derivatives -
        = NULL               // array of length 3 in
                             // order xuu, xuv, xvv
    ) const;
```

Find the position and first and second derivatives of the surface at given parameter values.

```
public: virtual int spline::evaluate (
    SPApar_pos const&,              // param value
    SPAposition&,                   // pt on surface
                                    // at given
                                    // parameter
    SPAvector**                     // array of ptrs
        = NULL,                     // to arrays
                                    // of vectors.
    int                             // # derivatives
        = 0,                        // required (nd)
    evaluate_surface_quadrant       // the evaluation
                                    // loc. above,
                                    // below for each
                                    // parameter
                                    // direction,
        = evaluate_surface_unknown  // or don't care.
    ) const;
```

Calculates derivatives, of any order up to the number requested, and stores them in vectors provided by the user. 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 int spline::evaluate_iter (
    SPApar_pos const&,          // parameter position
    surface_evaldata*,          // data supplying
                                // initial values,
                                // and set to reflect
                                // the results of
                                // this evaluation
    SPAposition&,               // point on curve at
                                // given parameter
    SPAvector**                 // array of pointers
        = NULL,                 // to vectors, of
                                // size nd. Any of
                                // the pointers may
                                // be null, in which
                                // case the
                                // corresponding
                                // derivative will
                                // not be returned
    int                         // number of
        = 0,                    // derivatives
                                // required (nd)
    evaluate_surface_quadrant   // evaluation
                                // location – above,
                                // below, don’t care
        = evaluate_surface_unknown
    ) const;
```

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

```
public: virtual double spline::eval_cross (
    SPApar_pos const&,      // parameter
    SPAunit_vector const&   // curve normal
    ) const;
```

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

```
public: virtual SPAunit_vector spline::eval_normal (
    SPApar_pos const&       // parameter value
    ) const;
```

Finds the normal to the spline at the point with given parameter values.

```
public: virtual SPAunit_vector spline::eval_outdir (
    SPApar_pos const&       // parameter value
    ) const;
```

Find an outward direction from the surface at a point with given parameter values.

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

Finds the point on the spline with given parameter values.

```
public: surf_princurv spline::eval_prin_curv (
    SPApar_pos const& param // parameter value
    ) const;
```

Finds the principal axes of curvature of the surface at a point.

```
public: virtual void spline::eval_prin_curv (
    SPApar_pos const&,      // parameter value
    SPAunit_vector&,        // first axis direction
    double&,                // 1st direction
                            // curvature
                            // direction
    SPAunit_vector&,        // second axis direction
    double&                 // second direction
                            // curvature
    ) const;
```

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

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

Returns the fit tolerance of the bs3_curve to the true spline surface.

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

Returns read–only access to the disc_info objects.

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

Returns read-only access to the disc_info objects.

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

Returns the sweep path curve for this spline.

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

Returns the sweep path type for this spline.

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

Returns the sweep profile curve for this spline.

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

Returns the sweep rail law for this spline.

```
public: spl_sur const& spline::get_spl_sur () const;
```

Returns defining spline surface and should only be used when absolutely necessary.

```
public: virtual logical
    spline::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* spline::make_copy () const;
```

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

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

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

```
public: void spline::make_single_ref ();
```

Makes a single reference to this spline.

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

Negates this spline.

```
public: virtual surf_normcone spline::normal_cone (
    SPApar_box const&,      // parameter bounds
    logical                 // approx. ok?
        = FALSE,
    SPAtransf const&        // transformation
        = * (SPAtransf* ) NULL_REF
    ) const;
```

Return 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 wholly within or wholly 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& spline::operator*= (
    SPAtransf const&        // transformation
    );
```

Transforms this spline by the given transformation.

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

Returns a surface with a reversed sense.

```
public: spline& spline::operator= (
    spline const&           // spline name
    );
```

Copies the spline record, and adjust the use counts of the underlying information to suit.

```
public: virtual logical spline::operator== (
    surface const&          // surface to be compared
    ) const;
```

This, like testing floating point numbers for equality, is not guaranteed to say *equal* for effectively equal surfaces, but is guaranteed to say *not equal* if they are indeed not equal. The result can be used for optimization, but not where it really matters. The default always says *not equal*, for safety.

```
public: virtual SPApar_pos spline::param (
    SPAposition const&,     // position
    SPApar_pos const&       // initial guess
        = * (SPApar_pos* ) NULL_REF
    ) const;
```

Finds the parameter values of a point on a 3D B–spline surface, iterating from the given parameter values (if supplied).

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

Indicates if the surface is parametric. Always TRUE for splines.

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

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

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

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

```
public: virtual SPApar_box spline::param_range (
    SPAbox const&           // region of interest
        = * (SPAbox* ) NULL_REF
    ) const;
```

Returns the principal parameter range of a parametric surface in a chosen parameter direction. For a nonparametric surface, the range is returned as the empty interval or box. A periodic surface is defined for all parameter values in the periodic direction, by reducing the given parameter modulo the period into this principal range. For a surface open or nonperiodic in the chosen direction the surface evaluation functions are defined only for parameter values in the returned range. If a box is provided, the parameter range returned may be restricted to a portion of the surface that is guaranteed to contain all portions of the surface that lie within the region of interest. If none is provided, and the parameter range in some direction is unbounded, then conventionally an empty interval is returned.

```
public: virtual SPAinterval spline::param_range_u (
    SPAbox const&           // region of interest
        = * (SPAbox* ) NULL_REF
    ) const;
```

Refer to previous description.

```
public: virtual SPAinterval spline::param_range_v (
    SPAbox const&           // region of interest
        = * (SPAbox* ) NULL_REF
    ) const;
```

Refer to previous description.

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

Finds the change in surface parameter corresponding to a unit offset in a given direction at a given position.

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

Reports whether a parametric surface is periodic in the *u*-parameter direction; i.e., it is smoothly closed, so faces can run over the seam.

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

Reports whether a parametric surface is periodic in the *v*-parameter direction; i.e., it is smoothly closed, so faces can run over the seam.

```
public: virtual logical spline::planar (
    SPAposition&,            // location
    SPAunit_vector&          // unit vector
    ) const;
```

Determines whether spline is planar.

```
public: virtual double spline::point_cross (
    SPAposition const&,      // position
    SPAunit_vector const&,   // direction
    SPApar_pos const&        // initial param guess
        = * (SPApar_pos* ) NULL_REF
    ) const;
```

Finds the curvature of a cross-section curve of the surface at the point on the surface closest to the given point, iterating from the given parameter values (if supplied). The cross-section curve is determined by the intersection of the surface with a plane passing through the point on the surface and with given normal.

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

Finds the normal to the surface at the given point.

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

Finds an outward direction from the surface at a point on the surface nearest to the given point. Normally just the normal, but nonzero at a singularity.

```
public: virtual void spline::point_perp (
    SPAposition const&,      // given position
    SPAposition&,            // position on a surface
    SPAunit_vector&,         // surface normal
    surf_princurv&,          // principal curvature
    SPApar_pos const&        // parameter guess
        = * (SPApar_pos* ) NULL_REF,
    SPApar_pos&              // actual parameter
        = * (SPApar_pos* ) NULL_REF,
    logical f_weak           // weak flag
        = FALSE
    ) const;
```

Finds the point on the surface nearest to the given point and the normal to and principal curvatures of the surface at that point. If the surface is parametric, also return the parameter values at the found point.

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

Finds the point on the surface nearest to the given point. If the surface is parametric, also return the parameter values at the found point.

```
public: void spline::point_perp (
    SPAposition const& pos,  // given position
    SPAposition& foot,       // position on a surface
    SPAunit_vector& norm,    // surface normal
    SPApar_pos const&        // parameter position
        param_guess          // parameter guess
        = * (SPApar_pos* ) NULL_REF,
    SPApar_pos& param_actual// actual parameter
        = * (SPApar_pos* ) NULL_REF,
    logical f_weak           // weak flag
        = FALSE
    ) const;
```

Finds the point on the surface nearest to the given point. If the surface is parametric, also return the parameter values at the found point.

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

Find the principal curvatures at a given point, returning the values in a struct. Just uses the other (virtual) principal curvature function.

```
public: virtual void spline::point_prin_curv (
    SPAposition const&,      // position
    SPAunit_vector&,         // first axis direction
    double&,                 // curvature in first
                             // direction
    SPAunit_vector&,         // second axis direction
    double&,                 // curvature in second
                             // direction
    SPApar_pos const&        // parameter guess
        = * (SPApar_pos* ) NULL_REF
    ) const;
```

Find the principal axes of curvature of the surface at a given point, and the curvatures in those directions.

```
public: void spline::reparam (
    double,                  // new start u parameter
    double,                  // new end u parameter
    double,                  // new start v parameter
    double                   // new end v parameter
    );
```

Reparameterizes the curve.

```
public: void spline::reparam_u (
    double,                  // new start u parameter
    double                   // new end u parameter
    );
```

Reparameterizes the curve in *u*.

```
public: void spline::reparam_v (
    double,                  // new start v parameter
    double                   // new end v parameter
    );
```

Reparameterizes the curve in *v*.

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

Restore the data for a spline from a save file.

```
if (restore_version_number < SPLINE_VERSION )
    // Just restore as an exact spline.
    (spl_sur *)dispatch_restore_subtype( ”sur”, ”exactsur” )
else
    read_logical                 Reverse flag; either “forward” or
                                 “reversed”
    // Switch to the right restore routine, using the standard
    // system mechanism. Note that the argument is to enable
    // the reader to distinguish old–style types where ”exact”
    // was both an int_cur and a spl_sur. They are now ”exactcur”
    // and ”exactsur”.
    (spl_sur *)dispatch_restore_subtype( ”sur” )
surface::restore_data            Fix the underlying surface
```

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

Determines if the underlying sculptured (spline) surface is in the opposite direction of the ACIS spline surface. This function returns TRUE if spline surface is opposite.

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

Saves the type or id, then calls save_data.

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

Saves the information for the spline in the save file.

```
public: const eval_sscache_entry*
    spline::search_eval_cache (
    const SPAposition& pos  // position to evaluate
    ) const;
```

Searches the underlying cache for an entry at the given position. Returns the matching eval entry if this is found, or NULL otherwise.

```
public: void spline::set_sur (
    bs3_surface,            // surface data
    double fitol            // fit tolerance
        = -1.0
    );
```

Sets the surface information.

```
public: virtual logical spline::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 nonconstant 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 nonsingular in both directions.

```
public: virtual logical spline::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 nonconstant 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 nonsingular in both directions.

```
public: spline* spline::split_u (
    double                  // parameter
    );
```

Divide a surface into two pieces at a *u*-parameter value. Returns a new surface for the low-parameter side, and change the old one to represent the high-parameter side.

```
public: spline* spline::split_v (
    double                  // parameter
    );
```

Divide a surface into two pieces at a *v*-parameter value. Returns a new surface for the low-parameter side, and change the old one to represent the high-parameter side.

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

Divide a surface into separate pieces which are smooth (and therefore suitable for offsetting or blending). The surface is split at its non–G1 discontinuities, and if it is closed after this, it is then split into two. The split pieces are stored in the the pieces argument. The function returns the count of split pieces.

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

Divide a surface into separate pieces which are smooth (and therefore suitable for offsetting or blending). The surface is split at its non–G1 discontinuities, and if it is closed after this, it is then split into two. The split pieces are stored in the the pieces argument. The function returns the count of split pieces.

```
public: spline* spline::subset (
    SPApar_box const&       // parameter range
    ) const;
```

Constructs a new spline that is a copy of the part of the given one within given parameter bounds.

```
public: bs3_surface spline::sur (
    double tol              // tolerance
        = -1.0
) const;
```

Returns (a pointer to) the underlying surface, or NULL if none.

```
public: logical spline::sur_present () const;
```

Returns TRUE if there is underlying surface data.

```
public: virtual logical spline::test_point_tol (
    SPAposition const&,     // position
    double                  // parameter
        = 0,
    SPApar_pos const&       // parameter guess
        = * (SPApar_pos* ) NULL_REF,
    SPApar_pos&             // actual parameter
        = * (SPApar_pos* ) NULL_REF
    ) const;
```

Tests whether a point lies on the surface, to user-defined tolerance.

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

Returns the type of spline.

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

Returns string “spline_xxx” where xxx is replaced with type_names of the underlying spl_sur.

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

Indicates if the spline is improperly defined.

```
public: virtual curve* spline::u_param_line (
    double                  // u-parameter
    ) const;
```

Constructs a *u*-parameter line on the surface. A *u*-parameter line runs in the direction of increasing *u*-parameter, at constant *v*. The parameterization in the nonconstant direction matches that of the surface, and has the range obtained by use of param_range_u. The new curve is constructed in free store, so it is the responsibility of the caller to ensure that it is correctly deleted.

```
public: virtual curve* spline::v_param_line (
    double                  // v-parameter
    ) const;
```

Constructs a *v*-parameter line on the surface. A *v*-parameter line runs in the direction of increasing *v*, at constant *u*. The parameterization in the nonconstant direction matches that of the surface, and has the range obtained by use of param_range_v. The new curve is constructed in free store, so it is the responsibility of the caller to ensure that it is correctly deleted.

Internal Use: full_size

Related Fncs:

restore_spline

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

Transforms a spline surface.

spl_sur

Class: Construction Geometry, SAT Save and Restore

Purpose: Defines an abstract base class from which spline surface definitions are derived.

Derivation: spl_sur : subtrans_object : subtype_object : ACIS_OBJECT : –

SAT Identifier: spl_sur

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

Description: In ACIS a sculptured surface is represented by the class spline, which contains a pointer to an internal description called spl_sur. The spl_sur further contains a bs3_surface that is a pointer to a rational or nonrational, nonuniform B–spline surface in the underlying surface package.

To support various types of surface construction, ACIS uses classes derived from the internal representation spl_sur. Also, surface classes can be derived from the derived class to construct more complicated surfaces. This section covers the base class spl_sur along with the methods used to create derived classes, rewritten per their specifications. The section also presents the classes derived from spl_sur and the construction method for them.

This class contains the mathematical definition for a spline surface. It uses use counts to limit copying, and it allows derivation to construct surfaces that are only approximated by the bs3_surface. The base class spl_sur contains the following information for defining the surface:

– A use count indicating the number of times this spl_sur is used.
– A pointer to a bs3_surface, that represents the spline surface.
– A fitting tolerance representing the precision of the spline approximation to the true surface.

Classes derived from spl_sur can contain additional information, and can record the creation method of the true spline surface.

All functions defined for the spline class are supported by virtual functions that depend on the true definition of the surface. The functionality is made virtual to allow the derived surfaces to implement the functionality on their own. For surfaces that have an exact bs3_surface, there is no need to implement the functionality because the methods written for the base class are sufficient.

Limitations: None

References:

KERN	discontinuity_info, summary_bs3_surface
by KERN	spline, summary_bs3_surface
BASE	SPAinterval

Data:

```
protected bs3_surface sur_data;
```
Object-space approximation to true surface.

`protected closed_forms closed_in_u;`
Takes the values OPEN, CLOSED or PERIODIC (or unset if the spl_sur is undefined). If an approximating surface is present (in sur_data), the closure of the approximating surface will be consistent.

`protected closed_forms closed_in_v;`
Takes the values OPEN, CLOSED or PERIODIC (or unset if the spl_sur is undefined). If an approximating surface is present (in sur_data), the closure of the approximating surface will be consistent.

`protected discontinuity_info u_disc_info;`
Storage for the discontinuities, if there are any.

`protected discontinuity_info v_disc_info;`
Storage for the discontinuities, if there are any.

`protected double fitol_data;`
The precision that the spline approximates the true surface.

`protected SPAinterval u_range;`
The full range of the spl_sur, as returned by param_range_u. If an approximating surface is present (in sur_data), this range should be identical to that of the approximating surface.

`protected SPAinterval v_range;`
The full range of the spl_sur, as returned by param_range_u. If an approximating surface is present (in sur_data), this range should be identical to that of the approximating surface.

`protected logical calling_make_approx;`
Prevents recursive calls to the method make_approx.

`protected singularity_type u_singularity;`
Records whether the surface is singular in *v*. If an approximating surface is present (in sur_data), the singularities of the approximating surface will be consistent.

`protected singularity_type v_singularity;`
Records whether the surface is singular in *u*. If an approximating surface is present (in sur_data), the singularities of the approximating surface will be consistent.

`protected summary_bs3_surface* summary_data;`
bs3_surface data in summary form. This field may be set on restore, if the full surface is not available. It may be used to make the actual bs3_surface.

Constructor:

```
protected: spl_sur::spl_sur ();
```

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

```
public: spl_sur::spl_sur (
    bs3_surface,            // approximation surface
    double                  // fit tolerance
        = 0
    );
```

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

```
public: spl_sur::spl_sur (
    const spl_sur&          // spline surface
    );
```

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

```
public: spl_sur::spl_sur (
    SPAinterval,            // u range
    SPAinterval,            // v range
    closed_forms,           // type of closure in u
    closed_forms,           // type of closure in v
    singularity_type,       // singularity type for u
    singularity_type        // singularity type for v
    );
```

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

Destructor:

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

C++ destructor, deleting a spl_sur.

Methods:

```
protected: virtual int spl_sur::accurate_derivs (
    SPApar_box const&       // parameter box
        = * (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 curve. If there is no limit to the number of accurate derivatives, this method returns the value, ALL_SURFACE_DERIVATIVES.

```
protected: virtual void spl_sur::append_u (
    spl_sur&                // given surface
    );
```

Concatenates the contents of two surfaces into one. the given surface appends to the existing one along *u*. The surfaces are guaranteed to be the same base or derived type and to have contiguous parameter ranges (“this” is the beginning part of the combined surface; i.e., lower parameter values, the argument gives the end part).

```
protected: virtual void spl_sur::append_v (
    spl_sur&                // given surface
    );
```

Concatenates the contents of two surfaces into one. the given surface appends to the existing one along *v*. The surfaces are guaranteed to be the same base or derived type and to have contiguous parameter ranges (“this” is the beginning part of the combined surface; i.e., lower parameter values, the argument gives the end part).

```
protected: virtual SPAbox spl_sur::bound (
    SPApar_box const&       // parameter box
        = * (SPApar_box* ) NULL_REF
    );
```

Returns a box around the surface. This need not be the smallest box which contains the specified portion of the surface, but needs to balance the tightness of the bound against the cost of evaluation.

```
protected: virtual void
    spl_sur::calculate_disc_info ();
```

Calculates the discontinuity information for the surface.

```
protected: virtual check_status_list*
    spl_sur::check (
    const check_fix& input          // flags for
        = * (const check_fix*)      // allowed
        NULL_REF,                   // fixes
    check_fix& result               // fixes
        = * (check_fix*) NULL_REF,  // applied
    const check_status_list*        // checks to
        = (const check_status_list*)// be made
        NULL_REF                    // default 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.

```
public: logical
    spl_sur::closed_u () const;
```

Determines if the surface is closed, smoothly or not, in the *u*-parameter direction.

```
public: logical
    spl_sur::closed_v () const;
```

Determines if the surface is closed, smoothly or not, in the *v*-parameter direction.

```
protected: virtual logical
    spl_sur::contains_pipe () const;
```

Returns TRUE if this spl_sur depends on a pipe surface.

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

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

```
protected: virtual void spl_sur::debug (
    char const*,            // leader
    logical,                // brief
    FILE*                   // output file
    ) const = 0;
```

Prints the definition of a spline surface to standard output or to the specified file. As for save and restore the operation is split 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.

```
protected: void spl_sur::debug_data (
    char const*,            // leader
    logical,                // brief
    FILE*                   // output file
    ) const;
```

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

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

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

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

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

```
protected: void spl_sur::delete_summary_data ();
```

Allows derived classes to delete summary_data when it goes out of date.

```
protected: save_approx_level
    spl_sur::enquire_save_approx_level () const;
```

Returns the default level at which the approximating surface should be stored.

```
public: virtual void spl_sur::eval (
    SPApar_pos const& uv,    // given parameter
    SPAposition& pos,        // returned point
    SPAvector* dpos,         // first derivative
    SPAvector* ddpos         // second derivative
    ) const;
```

Finds the position and the first and second derivatives of the surface at a specified point.

```
protected: virtual int spl_sur::evaluate (
    SPApar_pos const&,              // parameter
    SPAposition&,                   // pt on surface
                                    // at a given
                                    // parameter
    SPAvector**                     // Array of ptrs
        = NULL,                     // to arrays of
                                    // vectors size
                                    // nd
    int                             // number of
        = 0,                        // derivatives
                                    // required (nd)
    evaluate_surface_quadrant       // eval. location
        = evaluate_surface_unknown
    ) const;
```

Calculates position and derivatives. Once calculated the derivatives are stored in vectors provided by the user. This method returns the number it was able to calculate; this equals the number requested in all but the most exceptional circumstances. A certain number are evaluated directly and accurately; higher derivatives are automatically calculated by finite differencing; the accuracy of these decreases with the order of the derivative, as the cost increases. Any of the pointers may be NULL, in which case the corresponding derivatives will not be returned. Otherwise they must point to arrays long enough for all the derivatives of that order; i.e., 2 for the first derivatives, 3 for the second, etc.

```
protected: virtual int spl_sur::evaluate_iter (
    SPApar_pos const&,          // parameter position
    surface_evaldata*,          // data supplying
                                // initial values,
                                // and set to reflect
                                // the results of
                                // this evaluation
    SPAposition&,               // point on surface
                                // at given parameter
    SPAvector**                 // array of pointers
        = NULL,                 // to vectors, of
                                // size nd. Any of
                                // the pointers may
                                // be null, in which
                                // case the
                                // corresponding
                                // derivative will
                                // not be returned
    int                         // number of
        = 0,                    // derivatives
                                // required (nd)
    evaluate_surface_quadrant   // evaluation
                                // location – above,
                                // below, don’t care
        = evaluate_surface_unknown
    ) const;
```

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

```
protected: int spl_sur::evaluate_iter_with_cache (
    SPApar_pos const&,       // parameter
    surface_evaldata*,       // data supplying initial
                             // values, and set to
                             // reflect results
    SPAposition&,            // point on curve at
                             // parameter
    SPAvector**              // size nd array of
                             // pointers to arrays of
                             // vectors.
        = NULL,
    int                      // Number of deriv's req.
        = 0,
    evaluate_surface_quadrant    // evaluation
                                 // location – above,
                                 // below, don't care
        = evaluate_surface_unknown
    ) const;
```

This non-virtual function looks in the cache for position and nd derivatives at the given parameter value. If found it returns them. Otherwise it computes them, puts them in the cache, and returns them. The evaluate_iter_with_cache method, rather than evaluate_iter, should be called by classes derived from spl_sur, so as to get the benefit of caching.

```
protected: int spl_sur::evaluate_with_cache (
    SPApar_pos const&,       // parameter
    SPAposition&,            // position at parameter
    SPAvector**              // derivates at position
        = NULL,
    int                      // nd number of deriv's
        = 0,
    evaluate_surface_quadrant    // evaluation
                             // location above, below
                             // for each parameter
                             // direction
        = evaluate_surface_unknown
    ) const;
```

This non-virtual function looks in the cache for position and nd derivatives at the given parameter value. If found it returns them. Otherwise it computes them, puts them in the cache, and returns them. The evaluate_with_cache method, rather than evaluate, should be called by classes derived from spl_sur, so as to get the benefit of caching.

```
protected: virtual double spl_sur::eval_cross (
    SPApar_pos const&,       // given parameter
    SPAunit_vector const&    // given plane normal
    ) const;
```

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

```
protected: virtual SPAunit_vector
spl_sur::eval_normal (
    SPApar_pos const&        // given parameter
    ) const;
```

Finds the normal to the surface at a given parameter.

```
protected: virtual SPAunit_vector
spl_sur::eval_outdir (
    SPApar_pos const&        // given parameter
    ) const;
```

Return a direction which points outward from the surface. This should be the outward normal if the point is not singular, otherwise a fairly arbitrary outward direction.

```
public: virtual SPAposition spl_sur::eval_position (
    SPApar_pos const&        // given parameter
    ) const;
```

Finds the point on the spline with the given parameter value.

```
protected: virtual void spl_sur::eval_prin_curv (
    SPApar_pos const&,       // given parameter
    SPAunit_vector&,         // first axis direction
    double&,                 // 1st direction
                             // curvature
    SPAunit_vector&,         // second axis direction
    double&                  // 2nd direction
                             // curvature
    ) const;
```

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

```
protected: void spl_sur::eval_with_cache (
    SPApar_pos const&,       // parameter
    SPAposition&,            // position at parameter
    SPAvector*,              // 1st deriv
    SPAvector*               // 2nd deriv
    ) const;
```

This non-virtual function looks in the cache for point perpendicular at the given parameter value. If found it returns them. Otherwise it computes them, puts them in the cache, and returns them. The eval_with_cache method, rather than eval, should be called by classes derived from spl_sur, so as to get the benefit of caching.

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

Returns the fit tolerance for the approximating bs3_surface.

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

Returns the sweep path curve for this spl_sur.

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

Returns the sweep path type for this spl_sur.

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

Returns the sweep profile curve for this spl_sur.

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

Returns the sweep rail law for this spl_sur.

```
protected: virtual void
spl_sur::incremental_make_approx (
    double fit              // tolerance value for
                            // approx. surface
                            // to be made
    );
```

Makes an approximating surface for an extended spl_sur incrementally given an approximating surface for the original spl_sur. The extension is done first for the u/v direction, which is a smaller percentage of the original range.

```
public: void spl_sur::invalidate_cache ();
```

Method to be called by any user who modifies the surface in an external process, to ensure that stale evaluation results are discarded.

```
protected: logical spl_sur::iterate_perp (
    SPAposition const&,     // given position
    surface_evaldata*,      // surface
    SPAposition&,           // position on surface
    SPAunit_vector&,        // normal to surface
    surf_princurv&,         // principle curvature
    SPApar_pos const&,      // guess parameter
    SPApar_pos&,            // actual parameter
    logical                 // TRUE to iterate to a
                            // (local) near-point
                            // rather than any
                            // perpendicular.
    ) const;
```

Support function for point_perp (and bs3_surface_perp). This method finds a true perpendicular given an initial parameter guess, and avoiding oscillations. It may be set to iterate to the nearest perpendicular of any sort (minimum or maximum distance, or inflexion), or to find only minima (which is sometimes more reliable when there are inflexions), and it returns a success or failure indication.

```
public: logical spl_sur::iterate_perp (
    SPAposition const&,      // given position
    SPAposition&,            // position on surface
    SPAunit_vector&,         // normal to surface
    surf_princurv&,          // principle curvature
    SPApar_pos const&,       // guess parameter
    SPApar_pos&,             // actual parameter
    logical                  // TRUE to iterate to a
                             // (local) near–point
                             // rather than any
                             // perpendicular.
    ) const;
```

Support function for point_perp (and bs3_surface_perp). This method finds a true perpendicular given an initial parameter guess, and avoiding oscillations. It may be set to iterate to the nearest perpendicular of any sort (minimum or maximum distance, or inflexion), or to find only minima (which is sometimes more reliable when there are inflexions), and it returns a success or failure indication.

```
protected: virtual logical
    spl_sur::left_handed_uv () const;
```

Indicates whether the parameter coordinate system of the surface is right-handed 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

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

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

```
protected: virtual surface_evaldata*
    spl_sur::make_evaldata () const;
```

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

```
protected: virtual surf_normcone
    spl_sur::normal_cone (
    SPApar_box const&       // parameter box
        = * (SPApar_box* ) NULL_REF,
    logical                 // approx. results OK
        = FALSE
    );
```

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

```
protected: virtual void spl_sur::operator*= (
    SPAtransf const&        // transform
    );
```

Transforms this spline by the specified transform.

```
protected: virtual logical spl_sur::operator== (
    subtype_object const&   // object sub-type
    ) const;
```

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

```
protected: virtual SPApar_pos spl_sur::param (
    SPAposition const&,     // given point
    SPApar_pos const&       // guess parameter
        = * (SPApar_pos* ) NULL_REF
    ) const = 0;
```

Finds the parameter values of a point on a 3D B–spline surface, iterating from the given parameter values, if supplied.

```
public: double
    spl_sur::param_period_u () const;
```

Returns the *u* period of a periodic parametric surface, zero if the surface is not periodic in the *u* direction.

```
public: double
    spl_sur::param_period_v () const;
```

Returns the *v* period of a periodic parametric surface, zero if the surface is not periodic in the *v* direction.

```
public: SPApar_box spl_sur::param_range (
    SPAbox const&           // object space box
        = * (SPAbox* ) NULL_REF
    ) const;
```

Return the principal parameter range of a parametric surface in both *u* and *v*-parameter directions. For a nonparametric surface, the range is returned as the empty interval or box.

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

If a box is provided, the parameter range returned may be restricted to a portion of the surface which 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: SPAinterval spl_sur::param_range_u (
    SPAbox const&           // object space box
        = * (SPAbox* ) NULL_REF
    ) const;
```

Return the principal parameter range of a parametric surface in the *u*-parameter direction. For a nonparametric surface, the range is returned as the empty interval or box. A periodic surface is defined for all parameter values in the periodic direction, by reducing the given parameter modulo the period into this principal range. For a surface open or nonperiodic in the chosen direction the surface evaluation functions are defined only for parameter values in the returned range.

```
public: SPAinterval spl_sur::param_range_v (
    SPAbox const&           // object space box
        = * (SPAbox* ) NULL_REF
    ) const;
```

Return the principal parameter range of a parametric surface in the *v*-parameter direction. For a nonparametric surface, the range is returned as the empty interval or box. A periodic surface is defined for all parameter values in the periodic direction, by reducing the given parameter modulo the period into this principal range. For a surface open or nonperiodic in the chosen direction the surface evaluation functions are defined only for parameter values in the returned range.

```
protected: virtual SPApar_vec spl_sur::param_unitvec
(
    SPAunit_vector const&,  // given unit offset
    SPApar_pos const&       // given parameter
    ) const;
```

Finds the change in the surface parameter corresponding to a unit offset in a given direction at a given position. The position and direction must both lie in the surface.

```
protected: SPApar_pos spl_sur::param_with_cache (
    SPAposition const&,     // given position
    SPApar_pos const&       // return parameter
        = * (SPApar_pos* )NULL_REF
    );
```

This non–virtual function looks in the cache for a given position. If found it returns the parameter, otherwise it finds the parameter using param, places it in the cache, and returns it. The param_with_cache method, rather than param, should be called by classes derived from int_cur, so as to get the benefit of caching.

```
public: logical spl_sur::periodic_u () const;
```

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

```
public: logical spl_sur::periodic_v () const;
```

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

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

Reports whether a surface is planar.

```
protected: virtual double spl_sur::point_cross (
    SPAposition const&,      // given point
    SPAunit_vector const&,   // normal to plane
    SPApar_pos const&        // guess parameter
        = * (SPApar_pos* ) NULL_REF
    ) const;
```

Finds the curvature of a cross-section curve of the surface at the point on the surface closest to the given point, iterating from the given parameter values, if supplied. The cross-section is determined by the intersection of the surface with a plane passing through the point on the surface and with the given normal.

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

Finds the normal to the surface at the given point. This method returns exactly 0 if the point is a singularity of the surface where there is no well-defined normal.

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

Returns a direction that points outward from the surface. This should be the outward normal if the point is not singular, otherwise a fairly arbitrary outward direction.

```
protected: virtual void spl_sur::point_perp (
    SPAposition const&,      // given point
    SPAposition&,            // point returned
    SPAunit_vector&,         // normal returned
    surf_princurv&,          // principal curvature
    SPApar_pos const&        // guess parameter
        = * (SPApar_pos* ) NULL_REF,
    SPApar_pos&              // parameter returned
        = * (SPApar_pos* ) NULL_REF,
    logical f_weak           // weak flag
        = FALSE
    ) const;
```

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

```
protected: void spl_sur::point_perp_with_cache (
    SPAposition const&,      // given point
    SPAposition&,            // point returned
    SPAunit_vector&,         // normal returned
    surf_princurv&,          // principle curvature
    SPApar_pos const&        // guess parameter
        = * (SPApar_pos* )NULL_REF,
    SPApar_pos&              // parameter returned
        = * (SPApar_pos* )NULL_REF,
    logical f_weak           // weak flag
        = FALSE
    ) const;
```

This non-virtual function looks in the cache for point perpendicular at the given parameter value. If found it returns them. Otherwise it computes them, puts them in the cache, and returns them. The point_perp_with_cache method, rather than point_perp, should be called by classes derived from spl_sur, so as to get the benefit of caching.

```
protected: virtual void spl_sur::point_prin_curv (
    SPAposition const&,      // given point
    SPAunit_vector&,         // first axis direction
    double&,                 // 1st curvature
                             // direction
    SPAunit_vector&,         // second axis direction
    double&,                 // 2nd curvature
                             // direction
    SPApar_pos const&        // surface
        = * (SPApar_pos* ) NULL_REF
    ) const;
```

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

```
protected: virtual void spl_sur::reparam (
    double,                  // new start u parameter
    double,                  // new end u parameter
    double,                  // new start v parameter
    double                   // new end v parameter
    );
```

Reparameterizes the curve.

```
protected: virtual void spl_sur::reparam_u (
    double,                  // new start u parameter
    double                   // new end u parameter
    );
```

Reparameterizes the curve in u.

```
protected: virtual void spl_sur::reparam_v (
    double,                  // new start v parameter
    double                   // new end v parameter
    );
```

Reparameterizes the curve in v.

```
protected: void spl_sur::restore_common_data ();
```

Restore the data for a spl_sur from a save file.

```
if ( restore_version_number >= APPROX_SUMMARY_VERSION )
    read_enum                       Restore enumeration for
                                    save_approx_level.
if ( level == save_approx_full )
    bs3_surface_restore             Restore the surface data
    if (restore_version_number < SPLINE_VERSION)
        // No fit tolerance to read
    else
        read_real                   Fit tolerance data
else if ( level == save_approx_summary )
    summary_bs3_surface::restore    Restore the surface data
    read_real                       Fit tolerance data
    read_enum                       Restore enumeration for
                                    closed_forms for u.
    read_enum                       Restore enumeration for
                                    closed_forms for v.
    read_enum                       Restore enumeration for
                                    singularity_type for u.
    read_enum                       Restore enumeration for
                                    singularity_type for v.
else
    read_interval                   Restore u range
    read_interval                   Restore v range
    read_enum                       Restore enumeration for
                                    closed_forms for u.
    read_enum                       Restore enumeration for
                                    closed_forms for v.
    read_enum                       Restore enumeration for
                                    singularity_type for u.
    read_enum                       Restore enumeration for
                                    singularity_type for v.
if ( restore_version_number >= DISCONTINUITY_VERSION )
    // Restore the discontinuity information
    discontinuity_info::restore     u discontinuities
    discontinuity_info::restore     v discontinuities
```

```
protected: void spl_sur::save_as_approx () const;
```

Saves an approximation of the spl_sur.

```
protected: void spl_sur::save_common_data (
    save_approx_level       // level that spl_sur
                            // is to be stored
    ) const;
```

Saves data common to all spl_surs.

```
protected: virtual void spl_sur::save_data () const;
```

Save the information for the spl_sur to a save file.

```
protected: const eval_sscache_entry*
    spl_sur::search_eval_cache (
    const SPAposition&      // position to evaluate
    ) const;
```

Searches the underlying cache for an entry at the given position. Returns the matching eval entry if this is found, or NULL otherwise.

```
protected: void spl_sur::set_sur (
    bs3_surface,            // spline surface
    double tol              // fit tolerance
        = -1.0
    );
```

Sets the particular spline surface.

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

Adjusts the spline surface to have a parameter range increased by the shift value, which may be negative. This method is only used to move portions of a periodic surface by integral multiples of the period.

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

Adjusts the spline surface to have a parameter range increased by the shift value, which may be negative. This method is only used to move portions of a periodic surface by integral multiples of the period.

```
public: logical spl_sur::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 nonconstant 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 nonsingular in both directions.

```
public: logical spl_sur::singular_v (
    double                  // constant u-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 nonconstant 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 nonsingular in both directions

```
protected: virtual int spl_sur::split_at_kinks (
    spl_sur**& pieces,      // pieces
    logical udir,           // u direction or not
    double curvature = 0.0  // curvature
    ) const;
```

Divide a surface into separate pieces which are smooth (and therefore suitable for offsetting or blending). The surface is split at its non–G1 discontinuities, and if it is closed after this, it is then split into two. The split pieces are stored in the pieces argument. The function returns the count of split pieces.

```
protected: logical spl_sur::split_spl_sur_u (
    double approx_par,      // approx. param. value
    double real_par,        // real param. value
    spl_sur*,               // spare spl_sur
    spl_sur* [ 2 ]          // resulting pieces
    );
```

Divides a surface into two pieces at the specified parameter value, except that it provides an empty spl_sur in case it is needed. This method returns TRUE if the spl_sur is used; otherwise, it returns FALSE. This method is not externally called, but it is available for use by the derived class implementation of split_u.

Typically, the derived class implementations of split_u/v call these functions with a slightly different parameter to the one that they were originally called with (the new parameter is obtained by relaxing from the split point to the approximating surface, and so can be regarded as the parameter on the approximating surface). This method takes both versions of the parameter.

```
protected: logical spl_sur::split_spl_sur_v (
    double approx_par,      // approx. param. value
    double real_par,        // real param. value
    spl_sur*,               // spare spl_sur
    spl_sur* [ 2 ]          // resulting pieces
    );
```

Divides a surface into two pieces at the specified parameter value, except that it provides an empty spl_sur in case it is needed. This method returns TRUE if the spl_sur is used; otherwise, it returns FALSE. This method is not externally called, but it is available for use by the derived class implementation of split_v.

Typically, the derived class implementations of split_u/v call these functions with a slightly different parameter to the one that they were originally called with (the new parameter is obtained by relaxing from the split point to the approximating surface, and so can be regarded as the parameter on the approximating surface). This method takes both versions of the parameter.

```
protected: virtual void spl_sur::split_u (
    double,                 // parameter value
    spl_sur* [ 2 ]          // resulting pieces
    ) = 0;
```

Divides a surface into two pieces at the specified parameter value. This method returns a new surface for the low-parameter side, and changes the old one to represent the high-parameter side.

```
protected: virtual void spl_sur::split_v (
    double,                 // parameter value
    spl_sur* [ 2 ]          // resulting pieces
    ) = 0;
```

Divides a surface into two pieces at the specified parameter value. This method returns a new surface for the low-parameter side, and changes the old one to represent the high-parameter side.

```
protected: virtual spl_sur* spl_sur::subset (
    SPApar_box const&       // parameter box
    );
```

Constructs a new spline that is a copy of the part of the original within the given parameter bounds, unless this is not a proper subset, when returning the *this* pointer, or there is no overlap of the ranges, when returning NULL. The ranges should not overlap at a single point in either parameter direction—if they do, the whole parameter range in that direction is assumed. This cannot be const because it sometimes returns *this* as a non-const pointer.

```
protected: int spl_sur::summary_nuknots () const;
```

Provides read-only access to the nuknots summary_data for derived classes.

```
protected: int spl_sur::summary_nvknots () const;
```

Provides read-only access to the nvknots summary_data for derived classes.

```
protected: const double*
    spl_sur::summary_uknots () const;
```

Provides read-only access to the uknots summary_data for derived classes.

```
protected: const double*
    spl_sur::summary_vknots () const;
```

Provides read-only access to the vknots summary_data for derived classes.

```
public: bs3_surface spl_sur::sur () const;
```

Returns the bs3_surface approximation.

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

Tests whether a point lies on the surface, within a user-defined tolerance.

```
protected: virtual char const*
    spl_sur::type_name () const = 0;
```

Returns the string “spl_sur”.

```
protected: void spl_sur::update_data (
    bs3_surface              // surface for update
    );
```

Updates the range, closure and singularity information from a bs3_surface.

```
protected: virtual curve* spl_sur::u_param_line (
    double,                  // constant v-parameter
    spline const&            // owning surface
    ) const;
```

Constructs an iso-parameter line on the surface. A *u*-parameter line runs in the direction of increasing *u*-parameter, at constant *v*. The parameterization in the nonconstant direction matches that of the surface, and it has the range obtained by the use of param_range_u.

```
protected: virtual curve* spl_sur::v_param_line (
    double,                  // constant v-parameter
    spline const&            // owning surface
    ) const;
```

Constructs an isoparameter line on the surface. A *v*-parameter line runs in the direction of increasing *v*-parameter, at constant *u*. The parameterization in the nonconstant direction matches that of the surface, and it has the range obtained by the use of param_range_v.

Internal Use: deep_copy_elements, full_size

Related Fncs:

None