Chapter 4.
Classes

Topic: Ignore

The class interface is a set of C++ classes, including their public and protected data and methods (member functions), that an application can use directly to interact with ACIS. Developers may also derive their own classes from these classes to add application–specific functionality and data. Refer to the *3D ACIS Online Help User's Guide* for a description of the fields in the reference template.

SWEEP_ANNOTATION

Class: Feature Naming, Sweeping, SAT Save and Restore

Purpose: Implements the base class for sweeping annotations.

Derivation: SWEEP_ANNOTATION : ANNOTATION : ENTITY : ACIS_OBJECT : –

SAT Identifier: "sweep_annotation"

Filename: swp/sweep/sg_husk/sweep/swp_anno.hxx

Description: This is the base (organization) class for sweeping annotations. More specific sweep annotation classes are derived from this class.

Limitations: None

References: KERN ENTITY

Data:

`protected ENTITY* ents[e_num_datums];`
Array of entities.

`protected enum;`
Can have the values e_path, e_profile, and e_num_datums.

`protected static annotation_descriptor descriptors[];`
Array of descriptors for the sweep annotation.

`protected static const int num_ents;`
Number of entities in the array.

```
protected logical unhooked_out_is_ee[e_num_datums];
```
This is a logical for internal use only indicating when the unhooked output entity ents[i] is an EE_LIST. It has no meaning when the ANNOTATION members are hooked (when the ANNOTATION logical members_are_hooked is TRUE) and when ents[i] is an input to the ANNOTATION.

Constructor:

```
public: SWEEP_ANNOTATION::SWEEP_ANNOTATION (
    ENTITY* path            // sweep path
        = NULL,
    ENTITY* profile         // sweep profile
        = NULL
    );
```

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

Destructor:

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

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 SWEEP_ANNOTATION(...) then later x–>lose.)

Methods:

```
public: void SWEEP_ANNOTATION::add_path (
    ENTITY* e               // path entity
    );
```

Adds a pointer to the entity representing the sweep path to the annotation structure.

```
public: void SWEEP_ANNOTATION::add_profile (
    ENTITY* e               // profile entity
    );
```

Adds a pointer to the entity representing the sweep profile to the annotation structure.

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

Outputs the entity debug information to the specified file.

```
public: virtual ENTITY*& SWEEP_ANNOTATION::
    find_entity_ref_by_name(
    const char* name,       // name of entity
    logical& isInput        // flag to tell if item
                            // is input
    );
```

Implements get_entity_by_name and set_entity_by_name so they don't have to be virtual stacks.

```
public: virtual ENTITY*
    SWEEP_ANNOTATION::get_entity_by_name (
    const char* name        // name of entity
    );
```

Returns a pointer to entity specified by the given name.

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

If level is unspecified or 0, returns the type identifier ATTRIB_SYS_TYPE. If level is specified, returns <class>_TYPE for that level of derivation from ENTITY. The level of this class is defined as ATTRIB_SYS_LEVEL.

```
public: virtual void SWEEP_ANNOTATION::inputs (
    ENTITY_LIST& list,       // input list
    logical no_tags          // tags or not
        = TRUE
    ) const;
```

Returns a list of entities that are the inputs to the modeling operation.

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

Returns TRUE if this can be deep copied.

```
public: virtual logical SWEEP_ANNOTATION::
    is_entity_by_name (
    const char* name,        // name of entity
    ENTITY* entity           // pointer to entity
    );
```

Returns TRUE if the named entity is the current entity.

```
public: virtual const char*
    SWEEP_ANNOTATION::member_name (
    const ENTITY* entity     // entity with name
    ) const;
```

Returns the name of the member.

```
public: virtual void SWEEP_ANNOTATION::outputs (
    ENTITY_LIST& list        // output list
    ) const;
```

Returns an entity list that is the output results of the modeling operation.

```
public: ENTITY* SWEEP_ANNOTATION::path () const;
```

Returns a pointer to the entity representing the sweep path.

```
public: EDGE* SWEEP_ANNOTATION::path_edge () const;
```

Returns a pointer to an edge belonging to the sweep path.

```
public: VERTEX*
    SWEEP_ANNOTATION::path_vertex () const;
```

Returns a pointer to the vertex of the path.

```
public: ENTITY* SWEEP_ANNOTATION::profile () const;
```

Returns a pointer to the entity that represents the sweep profile.

```
public: EDGE*
    SWEEP_ANNOTATION::profile_edge () const;
```

Returns a pointer to an edge of the profile.

```
public: VERTEX*
    SWEEP_ANNOTATION::profile_vertex () const;
```

Returns a pointer to a vertex of the profile.

```
public: void SWEEP_ANNOTATION::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.

No data	This class does not save any data.

```
public: virtual void
    SWEEP_ANNOTATION::set_entity_by_name (
    const char* name,        // name for entity
    ENTITY* value            // entity pointer
    );
```

Specifies the name for the entity.

```
public: void SWEEP_ANNOTATION::set_profile (
    ENTITY* e                // profile entity
    );
```

Specifies the entity that represents the sweep profile.

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

Returns the string “sweep_annotation”.

Internal Use: hook_members, lose_input_tags, lose_lists, member_lost, member_lost_internal, merge_member, split_member, save, save_common, unhook_members

Related Fncs:

is_SWEEP_ANNOTATION

SWEEP_ANNO_EDGE_LAT

Class: Feature Naming, Sweeping, SAT Save and Restore

Purpose: Defines the annotation class for the lateral topology corresponding to an edge of the profile which becomes a face in sweeping.

Derivation: SWEEP_ANNO_EDGE_LAT : SWEEP_ANNOTATION : ANNOTATION : ENTITY : ACIS_OBJECT : –

SAT Identifier: “sweep_anno_edge_lat”

Filename: swp/sweep/sg_husk/sweep/swp_anno.hxx

Description: The lateral topology consists of the faces and edges filling the space between the profile and the top topology. This identifies the lateral (side) topology created from a profile edge during sweeping.

Inputs:

Profile – edge

Path – edge

Outputs:

Lateral_face – face

Limitations: None

References: KERN ENTITY

Data:

```
protected ENTITY* ents[e_num_datums];
```

Array of entities.

```
protected enum;
```

Can have the values e_lateral_face, and e_num_datums.

```
protected static annotation_descriptor descriptors[];
```
Array of descriptors for the sweep annotation.

```
protected static const int num_ents;
```
Number of entities in the array.

```
protected logical unhooked_out_is_ee[e_num_datums];
```
This is a logical for internal use only indicating when the unhooked output entity ents[i] is an EE_LIST. It has no meaning when the ANNOTATION members are hooked (when the ANNOTATION logical members_are_hooked is TRUE) and when ents[i] is an input to the ANNOTATION.

Constructor:

```
public: SWEEP_ANNO_EDGE_LAT::SWEEP_ANNO_EDGE_LAT (
    ENTITY* path              // sweep path
        = NULL,
    ENTITY* profile           // sweep profile
        = NULL,
    FACE* lateral_face        // pointer to lateral
        = NULL                // face
    );
```

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

Destructor:

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

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 SWEEP_ANNO_EDGE_LAT(...) then later x–>lose.)

Methods:

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

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

```
public: virtual ENTITY*& SWEEP_ANNO_EDGE_LAT::
    find_entity_ref_by_name(
    const char* name,       // name of entity
    logical& isInput        // flag to tell if item
                            // is input
    );
```

Implements get_entity_by_name and set_entity_by_name so they don't have to be virtual stacks.

```
public: virtual ENTITY*
    SWEEP_ANNO_EDGE_LAT::get_entity_by_name (
    const char* name        // name of entity
    );
```

Returns a pointer to entity specified by the given name.

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

If level is unspecified or 0, returns the type identifier SWEEP_ANNO_EDGE_LAT_TYPE. If level is specified, returns <class>_TYPE for that level of derivation from ENTITY. The level of this class is defined as BLEND_ANNONATION_LEVEL.

```
public: virtual void SWEEP_ANNO_EDGE_LAT::inputs (
    ENTITY_LIST& list,      // input list
    logical no_tags         // tags or not
        = TRUE
    ) const;
```

Returns a list of entities that are the inputs to the modeling operation.

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

Returns TRUE if this can be deep copied.

```
public: virtual logical SWEEP_ANNO_EDGE_LAT::
    is_entity_by_name (
    const char* name,        // name of entity
    ENTITY* entity           // pointer to entity
    );
```

Returns TRUE if the named entity is the current entity.

```
public: ENTITY*
    SWEEP_ANNO_EDGE_LAT::lateral_face () const;
```

Returns a pointer to the created lateral face from sweeping.

```
public: virtual const char*
    SWEEP_ANNO_EDGE_LAT::member_name (
    const ENTITY* entity     // entity with name
    ) const;
```

Returns the name of the entity.

```
public: virtual void SWEEP_ANNO_EDGE_LAT::outputs (
    ENTITY_LIST& list        // output list
    ) const;
```

Returns an entity list that is the output results of the modeling operation.

```
public: void SWEEP_ANNO_EDGE_LAT::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.

No data — This class does not save any data.

```
public: virtual void
    SWEEP_ANNO_EDGE_LAT::set_entity_by_name (
    const char* name,       // name of entity
    ENTITY* value           // entity pointer
    );
```

Specifies the name for the entity.

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

Returns the string “sweep_anno_edge_lat”.

Internal Use: hook_members, lose_input_tags, lose_lists, member_lost, member_lost_internal, merge_member, save, save_common, split_member, unhook_members

Related Fncs: is_SWEEP_ANNO_EDGE_LAT, is_SWEEP_ANNO_END_CAPS

SWEEP_ANNO_EDGE_TOP

Class: Feature Naming, Sweeping, SAT Save and Restore

Purpose: Defines the top topology from a swept edge, which is another edge.

Derivation: SWEEP_ANNO_EDGE_TOP : SWEEP_ANNOTATION : ANNOTATION : ENTITY : ACIS_OBJECT : –

SAT Identifier: “sweep_anno_edge_top”

Filename: swp/sweep/sg_husk/sweep/swp_anno.hxx

Description: This class identifies the top (ending) topology created from a profile edge during sweeping.

Inputs:

Profile – edge

Path – edge

Outputs:

Top_edge – edge

Limitations: None

References: KERN ENTITY

Data:

```
protected ENTITY* ents[e_num_datums];
```
Array of entities.

```
protected enum;
```
Can have the values e_top_edge, and e_num_datums.

```
protected static annotation_descriptor descriptors[];
```
Array of descriptors for the sweep annotation.

```
protected static const int num_ents;
```
Number of entities in the array.

```
protected logical unhooked_out_is_ee[e_num_datums];
```
This is a logical for internal use only indicating when the unhooked output entity ents[i] is an EE_LIST. It has no meaning when the ANNOTATION members are hooked (when the ANNOTATION logical members_are_hooked is TRUE) and when ents[i] is an input to the ANNOTATION.

Constructor:

```
public: SWEEP_ANNO_EDGE_TOP::SWEEP_ANNO_EDGE_TOP (
    ENTITY* path            // sweep path
        = NULL,
    ENTITY* profile         // sweep profile
        = NULL,
    EDGE* top_edge          // pointer to top edge
        = NULL
    );
```

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

Destructor:

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

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 SWEEP_ANNO_EDGE_TOP(...) then later x–>lose.)

Methods:

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

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

```
public: virtual ENTITY*& SWEEP_ANNO_EDGE_TOP::
    find_entity_ref_by_name(
    const char* name,       // name of entity
    logical& isInput        // flag to tell if item
                            // is input
    );
```

Implements get_entity_by_name and set_entity_by_name so they don't have to be virtual stacks.

```
public: virtual ENTITY*
    SWEEP_ANNO_EDGE_TOP::get_entity_by_name (
    const char* name        // name of entity
    );
```

Returns a pointer to entity specified by the given name.

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

If level is unspecified or 0, returns the type identifier SWEEP_ANNO_EDGE_TOP_TYPE. If level is specified, returns <class>_TYPE for that level of derivation from ENTITY. The level of this class is defined as SWEEP_ANNO_EDGE_TOP_LEVEL.

```
public: virtual void SWEEP_ANNO_EDGE_TOP::inputs (
    ENTITY_LIST& list,       // input list
    logical no_tags          // tags or not
        = TRUE
    ) const;
```

Returns a list of entities that are the inputs to the modeling operation.

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

Returns TRUE if this can be deep copied.

```
public: virtual logical SWEEP_ANNO_EDGE_TOP::
    is_entity_by_name (
    const char* name,        // name of entity
    ENTITY* entity           // pointer to entity
    );
```

Returns TRUE if the named entity is the current entity.

```
public: logical SWEEP_ANNO_EDGE_TOP::is_top (
    const ENTITY* entity     // entity to test
    ) const;
```

Determines whether or not the given edge is a top edge of the sweep operation.

```
public: virtual const char*
    SWEEP_ANNO_EDGE_TOP::member_name (
    const ENTITY* entity     // entity with name
    ) const;
```

Returns the name of the member.

```
public: virtual void SWEEP_ANNO_EDGE_TOP::outputs (
    ENTITY_LIST& list        // output list
    ) const;
```

Returns an entity list that is the output results of the modeling operation.

```
public: void SWEEP_ANNO_EDGE_TOP::remove_top_edge (
    EDGE* e                 // edge to remove
    );
```

Removes unnecessary annotations on edge.

```
public: void SWEEP_ANNO_EDGE_TOP::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.

No data — This class does not save any data.

```
public: virtual void
    SWEEP_ANNO_EDGE_TOP::set_entity_by_name (
    const char* name,       // name of entity
    ENTITY* value           // entity pointer
    );
```

Specifies the name for the entity.

```
public: void SWEEP_ANNO_EDGE_TOP::set_top_edge (
    EDGE* e                 // edge for top
    );
```

Specifies the edge to use as the top edge of sweeping.

```
public: ENTITY* SWEEP_ANNO_EDGE_TOP::top_edge ()
const;
```

Returns a pointer to the top edge from sweeping.

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

Returns the string “sweep_anno_edge_top”.

Internal Use: hook_members, lose_input_tags, lose_lists, member_lost, member_lost_internal, merge_member, save, save_common, split_member, unhook_members

Related Fncs:

is_SWEEP_ANNO_EDGE_TOP

SWEEP_ANNO_END_CAPS

Class: Feature Naming, Sweeping, SAT Save and Restore

Purpose: Defines annotation relating to capping faces of a sweep.

Derivation: SWEEP_ANNO_END_CAPS : ANNOTATION : ENTITY : ACIS_OBJECT : –

SAT Identifier: “sweep_anno_end_caps”

Filename: swp/sweep/sg_husk/sweep/swp_anno.hxx

Description: This annotation identifies the faces at the start and end of the sweep. It is unique in that only output entities are available. One should process the information immediately after the sweep, or at least before doing another sweep, so one can tell which sweep it applies to.

Inputs:

none

Outputs:

Start_face – face

End_face – face

Limitations: None

References: KERN ENTITY

Data:

```
protected ENTITY* ents[e_num_datums];
```

Array of entities.

```
protected enum;
```

Can have the values e_start_face, e_end_face, and e_num_datums.

```
protected static annotation_descriptor descriptors[];
```

Array of descriptors for the sweep annotation.

```
protected static const int num_ents;
```

Number of entities in the array.

```
protected logical unhooked_out_is_ee[e_num_datums];
```

This is a logical for internal use only indicating when the unhooked output entity ents[i] is an EE_LIST. It has no meaning when the ANNOTATION members are hooked (when the ANNOTATION logical members_are_hooked is TRUE) and when ents[i] is an input to the ANNOTATION.

Constructor:

```
public: SWEEP_ANNO_END_CAPS::SWEEP_ANNO_END_CAPS (
    FACE* start_face         // start face
        = NULL,
    FACE* end_face           // end face
        = NULL
    );
```

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

Destructor:

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

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 SWEEP_ANNO_EDGE_END_CAPS(...) then later x–>lose.)

Methods:

```
public: virtual void SWEEP_ANNO_END_CAPS::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* SWEEP_ANNO_END_CAPS::end_face () const;
```

Returns a pointer to the face representing an end face.

```
public: virtual ENTITY*& SWEEP_ANNO_END_CAPS::
    find_entity_ref_by_name(
    const char* name,       // name of entity
    logical& isInput        // flag to tell if item
                            // is input
    );
```

Implements get_entity_by_name and set_entity_by_name so they don't have to be virtual stacks.

```
public: virtual ENTITY*
    SWEEP_ANNO_END_CAPS::get_entity_by_name (
    const char* name        // name of entity
    );
```

Returns a pointer to entity specified by the given name.

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

If level is unspecified or 0, returns the type identifier SWEEP_ANNO_END_CAPS_TYPE . If level is specified, returns <class>_TYPE for that level of derivation from ENTITY. The level of this class is defined as SWEEP_ANNO_END_CAPS_LEVEL .

```
public: virtual void SWEEP_ANNO_END_CAPS::inputs (
    ENTITY_LIST& list,      // input list
    logical no_tags         // tags or not
        = TRUE
    ) const;
```

Returns a list of entities that are the inputs to the modeling operation.

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

Returns TRUE if this can be deep copied.

```
public: virtual logical SWEEP_ANNO_END_CAPS::
    is_entity_by_name (
    const char* name,        // name of entity
    ENTITY* entity           // pointer to entity
    );
```

Returns TRUE if the named entity is the current entity.

```
public: virtual const char*
    SWEEP_ANNO_END_CAPS::member_name (
    const ENTITY* entity     // entity with name
    ) const;
```

Returns the name of the member.

```
public: virtual void SWEEP_ANNO_END_CAPS::outputs (
    ENTITY_LIST& list        // output list
    ) const;
```

Returns an entity list that is the output results of the modeling operation.

```
public: void SWEEP_ANNO_END_CAPS::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.

No data	This class does not save any data.

```
public: void SWEEP_ANNO_END_CAPS::set_end_face (
    FACE* f                  // pointer to face
    );
```

Specifies the face to use as an end face.

```
public: virtual void
    SWEEP_ANNO_END_CAPS::set_entity_by_name (
    const char* name,        // name of entity
    ENTITY* value            // entity pointer
    );
```

Specifies the name for the entity.

```
public: void SWEEP_ANNO_END_CAPS::set_start_face (
    FACE* f                  // pointer to face
    );
```

Specifies the face to use as the start face.

```
public: FACE*
    SWEEP_ANNO_END_CAPS::start_face () const;
```

Returns a pointer to the face that is the start face.

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

Returns the string “sweep_anno_end_caps”.

Internal Use: hook_members, lose_input_tags, lose_lists, member_lost, member_lost_internal, merge_member, save, save_common, split_member, unhook_members

Related Fncs: is_SWEEP_ANNO_EDGE_LAT, is_SWEEP_ANNO_END_CAPS

SWEEP_ANNO_VERTEX_LAT

Class: Feature Naming, Sweeping, SAT Save and Restore

Purpose: Defines annotation relating to vertex of lateral topology of a sweep.

Derivation: SWEEP_ANNO_VERTEX_LAT : SWEEP_ANNOTATION : ANNOTATION : ENTITY : ACIS_OBJECT : –

SAT Identifier: “sweep_anno_vertex_lat”

Filename: swp/sweep/sg_husk/sweep/swp_anno.hxx

Description: This annotation identifies the lateral (side) topology created from a profile vertex during sweeping.

Inputs:

Profile – vertex
Path – edge

Outputs:

Right_lateral_face – face
Left_lateral_face – face
Right_lateral_edge – edge
Left_lateral_edge – edge
Mid_lateral_edge – edge

Limitations: None

References: KERN ENTITY

Data:

`protected ENTITY* ents[e_num_datums];`
Array of entities.

`protected enum;`
Can have the values e_right_lateral_face, e_left_lateral_face, e_right_lateral_edge, e_left_lateral_edge, e_mid_lateral_edge, and e_num_datums.

`protected static annotation_descriptor descriptors[];`
Array of descriptors for the sweep annotation.

`protected static const int num_ents;`
Number of entities in the array.

`protected logical unhooked_out_is_ee[e_num_datums];`
This is a logical for internal use only indicating when the unhooked output entity ents[i] is an EE_LIST. It has no meaning when the ANNOTATION members are hooked (when the ANNOTATION logical members_are_hooked is TRUE) and when ents[i] is an input to the ANNOTATION.

Constructor:

```
public: SWEEP_ANNO_VERTEX_LAT::
    SWEEP_ANNO_VERTEX_LAT (
    ENTITY* path            // sweep path
        = NULL,
    ENTITY* profile_vertex  // profile vertex
        = NULL,
    FACE* right_lateral_face// lateral sweep face
        = NULL,
    FACE* left_lateral_face // lateral sweep face
        = NULL,
    EDGE* right_lateral_edge// lateral edge
        = NULL,
    EDGE* left_lateral_edge // lateral edge
        = NULL,
    EDGE* mid_lateral_edge  // lateral edge
        = NULL
    );
```

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

Destructor:

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

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 SWEEP_ANNO_VERTEX_LAT(...) then later x–>lose.)

Methods:

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

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

```
public: virtual ENTITY*& SWEEP_ANNO_VERTEX_LAT::
    find_entity_ref_by_name(
    const char* name,        // name of entity
    logical& isInput         // flag to tell if item
                             // is input
    );
```

Implements get_entity_by_name and set_entity_by_name so they don't have to be virtual stacks.

```
public: virtual ENTITY*
    SWEEP_ANNO_VERTEX_LAT::get_entity_by_name (
    const char* name         // name of entity
    );
```

Returns a pointer to entity specified by the given name.

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

If level is unspecified or 0, returns the type identifier SWEEP_ANNO_VERTEX_LAT_TYPE. If level is specified, returns <class>_TYPE for that level of derivation from ENTITY. The level of this class is defined as SWEEP_ANNO_VERTEX_LAT_LEVEL.

```
public: virtual void
    SWEEP_ANNO_VERTEX_LAT::inputs (
    ENTITY_LIST& list,       // input list
    logical no_tags          // tags or not
        = TRUE
    ) const;
```

Returns a list of entities that are the inputs to the modeling operation.

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

Returns TRUE if this can be deep copied.

```
public: virtual logical SWEEP_ANNO_VERTEX_LAT::
    is_entity_by_name (
    const char* name,       // name of entity
    ENTITY* entity          // pointer to entity
    );
```

Returns TRUE if the named entity is the current entity.

```
public: ENTITY* SWEEP_ANNO_VERTEX_LAT::
    left_lateral_edge () const;
```

Returns a pointer to the left lateral edge from a sweep of a vertex.

```
public: ENTITY* SWEEP_ANNO_VERTEX_LAT::
    left_lateral_face () const;
```

Returns a pointer to the left lateral face from a sweep operation.

```
public: virtual const char*
    SWEEP_ANNO_VERTEX_LAT::member_name (
    const ENTITY* entity    // entity with name
    ) const;
```

Returns the name of the member.

```
public: ENTITY*
    SWEEP_ANNO_VERTEX_LAT::mid_lateral_edge () const;
```

Returns a pointer to the mid lateral edge from a sweep operation.

```
public: virtual void SWEEP_ANNO_VERTEX_LAT::outputs (
    ENTITY_LIST& list       // output list
    ) const;
```

Returns an entity list that is the output results of the modeling operation.

```
public: void
    SWEEP_ANNO_VERTEX_LAT::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.

No data	This class does not save any data.

```
public: ENTITY* SWEEP_ANNO_VERTEX_LAT::
    right_lateral_edge () const;
```

Returns a pointer to the right lateral edge from a sweep operation.

```
public: ENTITY* SWEEP_ANNO_VERTEX_LAT::
    right_lateral_face () const;
```

Returns a pointer to the right lateral face from a sweep operation.

```
public: virtual void
    SWEEP_ANNO_VERTEX_LAT::set_entity_by_name (
    const char* name,        // name of entity
    ENTITY* value            // entity pointer
    );
```

Specifies the name for the entity.

```
public: void
    SWEEP_ANNO_VERTEX_LAT::set_left_lateral_edge (
    EDGE* e                  // edge pointer
    );
```

Specifies the left lateral edge.

```
public: void
    SWEEP_ANNO_VERTEX_LAT::set_mid_lateral_edge (
    EDGE* e                  // edge pointer
    );
```

Specifies the mid lateral edge.

```
public: void
    SWEEP_ANNO_VERTEX_LAT::set_right_lateral_edge (
    EDGE* e                  // edge pointer
    );
```

Specifies the right lateral edge.

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

Returns the string “sweep_anno_vertex_lat”.

Internal Use: add_left_lateral_edge, add_left_lateral_face, add_right_lateral_edge, add_right_lateral_face, hook_members, lose_input_tags, lose_lists, member_lost, member_lost_internal, merge_member, save, save_common, split_member, unhook_members

Related Fncs: is_SWEEP_ANNO_VERTEX_LAT

SWEEP_ANNO_VERTEX_TOP

Class: Feature Naming, Sweeping, SAT Save and Restore

Purpose: Defines annotation relating to the vertices at the top of a sweep.

Derivation: SWEEP_ANNO_VERTEX_TOP : SWEEP_ANNOTATION : ANNOTATION : ENTITY : ACIS_OBJECT : –

SAT Identifier: “sweep_anno_vertex_top”

Filename: swp/sweep/sg_husk/sweep/swp_anno.hxx

Description: This annotation identifies the top (ending) topology created from a profile vertex during sweeping.

Inputs:

Profile – vertex

Path – edge

Outputs:

Right_top_edge – edge

Left_top_edge – edge

Right_top_vertex – vertex

Left_top_vertex – vertex

Mid_top_vertex – vertex

Limitations: None

References: KERN ENTITY

Data:

```
protected ENTITY* ents[e_num_datums];
```
Array of entities.

```
protected enum;
```
Can have the values e_right_top_edge, e_left_top_edge, e_right_top_vertex, e_left_top_vertex, e_mid_top_vertex, and e_num_datums.

```
protected static annotation_descriptor descriptors[];
```
Array of descriptors for the sweep annotation.

```
protected static const int num_ents;
```
Number of entities in the array.

```
protected logical unhooked_out_is_ee[e_num_datums];
```
This is a logical for internal use only indicating when the unhooked output entity ents[i] is an EE_LIST. It has no meaning when the ANNOTATION members are hooked (when the ANNOTATION logical members_are_hooked is TRUE) and when ents[i] is an input to the ANNOTATION.

Constructor:

```
public: SWEEP_ANNO_VERTEX_TOP::
    SWEEP_ANNO_VERTEX_TOP (
    ENTITY* path             // sweep path
        = NULL,
    ENTITY* profile_vertex   // profile vertex
        = NULL,
    EDGE* right_top_edge     // right top edge
        = NULL,
    EDGE* left_top_edge      // left top edge
        = NULL,
    VERTEX* right_top_vertex// right top vertex
        = NULL,
    VERTEX* left_top_vertex  // left top vertex
        = NULL,
    VERTEX* mid_top_vertex   // mid top vertex
        = NULL
    );
```

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

Destructor:

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

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 SWEEP_ANNO_VERTEX_TOP(...) then later x–>lose.)

Methods:

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

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

```
public: virtual ENTITY*& SWEEP_ANNO_VERTEX_TOP::
    find_entity_ref_by_name(
    const char* name,        // name of entity
    logical& isInput         // flag to tell if item
                             // is input
    );
```

Implements get_entity_by_name and set_entity_by_name so they don't have to be virtual stacks.

```
public: virtual ENTITY*
    SWEEP_ANNO_VERTEX_TOP::get_entity_by_name (
    const char* name         // name of entity
    );
```

Returns a pointer to entity specified by the given name.

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

If level is unspecified or 0, returns the type identifier SWEEP_ANNO_VERTEX_TYPE. If level is specified, returns <class>_TYPE for that level of derivation from ENTITY. The level of this class is defined as SWEEP_ANNO_VERTEX_LEVEL.

```
public: virtual void
    SWEEP_ANNO_VERTEX_TOP::inputs (
    ENTITY_LIST& list,          // input list
    logical no_tags             // tags or not
        = TRUE
    ) const;
```

Returns a list of entities that are the inputs to the modeling operation.

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

Returns TRUE if this can be deep copied.

```
public: virtual logical SWEEP_ANNO_VERTEX_TOP::
    is_entity_by_name (
    const char* name,           // name of entity
    ENTITY* entity              // pointer to entity
    );
```

Returns TRUE if the named entity is the current entity.

```
public: logical SWEEP_ANNO_VERTEX_TOP::is_top (
    const ENTITY* entity    // entity to test
    ) const;
```

Determines whether or not the specified entity is the top vertex.

```
public: ENTITY*
    SWEEP_ANNO_VERTEX_TOP::left_top_edge () const;
```

Returns a pointer to the left top edge.

```
public: ENTITY*
    SWEEP_ANNO_VERTEX_TOP::left_top_vertex () const;
```

Returns a pointer to the left top vertex.

```
public: virtual const char*
    SWEEP_ANNO_VERTEX_TOP::member_name (
    const ENTITY* entity    // entity with name
    ) const;
```

Returns the name of the member.

```
public: ENTITY*
    SWEEP_ANNO_VERTEX_TOP::mid_top_vertex () const;
```

Returns a pointer to the mid top vertex.

```
public: virtual void SWEEP_ANNO_VERTEX_TOP::outputs (
    ENTITY_LIST& list       // output list
    ) const;
```

Returns an entity list that is the output results of the modeling operation.

```
public: void
SWEEP_ANNO_VERTEX_TOP::remove_mid_top_vertex (
    VERTEX* v               // vertex to remove
    );
```

Remove unnecessary annotation on mid vertex.

```
public: void
    SWEEP_ANNO_VERTEX_TOP::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.

No data — This class does not save any data.

```
public: ENTITY*
    SWEEP_ANNO_VERTEX_TOP::right_top_edge () const;
```

Returns a pointer to the top right edge.

```
public: ENTITY*
    SWEEP_ANNO_VERTEX_TOP::right_top_vertex () const;
```

Returns a pointer to the right top vertex.

```
public: virtual void
    SWEEP_ANNO_VERTEX_TOP::set_entity_by_name (
    const char* name,        // name of entity
    ENTITY* value            // entity pointer
    );
```

Specifies the name for the entity.

```
public: void
    SWEEP_ANNO_VERTEX_TOP::set_left_top_edge (
    EDGE* v                  // pointer to edge
    );
```

Specifies the left top edge.

```
public: void
    SWEEP_ANNO_VERTEX_TOP::set_left_top_vertex (
    VERTEX* v                // pointer to vertex
    );
```

Specifies the left top vertex.

```
public: void
    SWEEP_ANNO_VERTEX_TOP::set_mid_top_vertex (
    VERTEX* v                // pointer to vertex
    );
```

Specifies the mid top vertex.

```
public: void
    SWEEP_ANNO_VERTEX_TOP::set_right_top_edge (
    EDGE* v                 // pointer to edge
    );
```

Specifies the right top edge.

```
public: void
    SWEEP_ANNO_VERTEX_TOP::set_right_top_vertex (
    VERTEX* v               // pointer to vertex
    );
```

Specifies the right top vertex.

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

Returns the string “sweep_anno_vertex_top”.

Internal Use: add_left_top_edge, add_left_top_vertex, add_right_top_edge, add_right_top_vertex, hook_members, lose_input_tags, lose_lists, member_lost, member_lost_internal, merge_member, save, save_common, split_member, unhook_members

Related Fncs: is_SWEEP_ANNO_VERTEX_TOP

sweep_options

Class: Sweeping

Purpose: Provides a data structure for sweeping operations to be used in the function api_sweep_with_options.

Derivation: sweep_options : ACIS_OBJECT : –

SAT Identifier: None

Filename: swp/sweep/sg_husk/sweep/swp_opts.hxx

Description: Provides a data structure for sweeping operations to be used in the function api_sweep_with_options.

The options are covered in the discussion topic *Options for a Law Sweep* and include:

- bool_type
- close_to_axis
- cut_end_off
- draft_angle
- <*string*>_draft_dist where <*string*> is “start” or “end”.
- draft_hole
- draft_law
- draft_repair
- gap_type
- keep_branches
- keep_law
- keep_start_face
- miter
- portion
- rail_laws
- rigid
- scale_law
- solid
- steps
- sweep_angle
- sweep_to_body
- to_face
- twist_angle
- twist_law
- two_sided
- which_side

Note that the get methods do not increment use counts or return copies of anything. Hence, what they return should be treated as read only.

The sweep_to_body option and it’s related options are mutually exclusive with the to_face option. The to_face option is used for backward compatibility. This option uses the surface of the face to locate the end of the sweep. It does not work with bool_type, keep_law or keep_branches options. These three options only work with the sweep_to_body option. The sweep_to_body option uses the entire body instead of just one surface. Combined with the related options, different parts of the resulting sweep will remain. If the to_face option is used, sweeping will unset the bool_type, keep_law and keep_branches options.

Limitations: If the rail_law array were set up using api_make_rails and it included a twist law, then rail_law would be mutually exclusive with the twist_law option. In other words, don’t use a twist law again.

References:

BASE	SPAposition
KERN	BODY, surface
LAW	law

Data:

None

Constructor:

```
public: sweep_options::sweep_options ();
```

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

```
public: sweep_options::sweep_options (
    AcisOptions*            // ACIS options
    );
```

Adopts the ACIS options.

Destructor:

```
public: sweep_options::~sweep_options ();
```

C++ destructor, deleting a sweep_options.

Methods:

```
public: sweep_options* sweep_options::copy ();
```

Makes a copy of the sweep_options, and returns a pointer to it.

```
public:sweep_bool_type
    sweep_options::get_bool_type () const ;
```

Returns the current bool_type option values.

```
public: logical
    sweep_options::get_close_to_axis () const ;
```

The sweep option close_to_axis allows a solid body to be created from a swept wire body. This method sets open wires to be swept around an axis to create a solid body. When TRUE, the ends will be extended to the axis. Care should be taken so that all of the profile is on one side of the axis, to avoid a self–intersecting body. The default is FALSE.

```
public: int sweep_options::get_cut_end_off () const;
```

Gets whether the end is cut off or not.

```
public:double
    sweep_options::get_draft_angle () const ;
```

Gets the draft angle.

```
public: draft_hole_option
    sweep_options::get_draft_hole () const;
```

Gets the draft hole option.

```
public: double
    sweep_options::get_draft_hole_angle () const;
```

Gets the draft hole angle.

```
public: law* sweep_options::get_draft_law () const;
```

Gets the draft law.

```
public:draft_repair_level
    sweep_options::get_draft_repair () const;
```

Gets the draft repair level.

```
public: double
    sweep_options::get_end_draft_dist () const;
```

Gets the end draft distance.

```
public: int sweep_options::get_gap_type () const;
```

Gets the gap type.

```
public:logical
    sweep_options::get_keep_branches () const ;
```

Gets whether or not branches are kept.

```
public: law* sweep_options::get_keep_law () const;
```

Gets the keep law.

```
public: logical
    sweep_options::get_keep_start_face () const;
```

If TRUE, the face used in the sweep is kept available after the sweep.

```
public:double
    sweep_options::get_miter_amount () const ;
```

Gets the miter amount.

```
public: miter_type
    sweep_options::get_miter_type () const;
```

Gets the miter type.

```
public: SPAposition
    sweep_options::get_portion_end () const;
```

Gets the end portion of the curve used for sweeping.

```
public:SPAposition
    sweep_options::get_portion_start () const ;
```

Gets the start portion of the curve used for sweeping.

```
public:law* sweep_options::get_rail_law (
    int which               // which law
    ) const;
```

Gets the rail law used.

```
public:int sweep_options::get_rail_num () const ;
```

Gets the rail number.

```
public: logical sweep_options::get_rigid () const;
```

Gets whether this is a rigid sweep or not.

```
public:law* sweep_options::get_scale_law () const ;
```

Gets the scale law.

```
public: logical sweep_options::get_simplify () const;
```

Gets whether or not simplification is done.

```
public: int sweep_options::get_solid () const ;
```

Gets whether or not a solid is created.

```
public: double
    sweep_options::get_start_draft_dist () const ;
```

Gets the starting draft distance.

```
public:int sweep_options::get_steps () const ;
```

Gets the number of steps.

```
public: double
    sweep_options::get_sweep_angle () const ;
```

Gets the sweep angle.

```
public:sweep_portion
    sweep_options::get_sweep_portion () const ;
```

Gets the sweep portion.

```
public:BODY*
    sweep_options::get_sweep_to_body () const ;
```

Gets the body swept to.

```
public: surface* sweep_options::get_to_face () const;
```

Gets the face swept to.

```
public: double
    sweep_options::get_twist_angle () const;
```

Gets the twist angle.

```
public: law* sweep_options::get_twist_law () const;
```

Gets the twist law.

```
public: int sweep_options::get_two_sided () const;
```

Gets whether a two–sided sheet body is created. The return will be –1 if the option has not yet been set, 0 if the option has been set to false, and 1 if the option has been set to true.

```
public:logical
    sweep_options::get_which_side () const ;
```

Gets which side of the profile should be swept.

```
public: int sweep_options::operator!= (
    sweep_options& in_sw_opt// sweep options to test
    );
```

Tests to see whether this sweep option instance is not equal to the given sweep option.

```
public: int sweep_options::operator== (
    sweep_options& in_sw_opt// sweep options to test
    );
```

Tests to see whether this sweep option instance is equal to the given sweep option.

```
public: void sweep_options::refresh ();
```

Refreshes all values to their defaults.

```
public: void
sweep_options::reset_default_copy_rail_laws ();
```

Internal function used to set the default copy rail law to NULL. Because the rail law is particular to the path used it must be set each time by the user, or the default rails will be used.

```
public: void sweep_options::set_bool_type (
    sweep_bool_type b       // UNITE, INTERSECT,
                            // SUBTRACT, KEEP_BOTH,
                            // or LIMIT
    );
```

Sets the options UNITE, INTERSECT, SUBTRACT, KEEP_BOTH, or LIMIT. The sweep option bool_type specifies what should be done with the pieces when sweeping to or through a fixed body. This is used as part of selective Booleans. The bool_type also controls the numbering on the cells of the tool body and the blank body. The number of cells can affect the results of a defined keep_law.

```
public: void sweep_options::set_close_to_axis(
    logical c               // TRUE make a solid
    );
```

The sweep option close_to_axis allows a solid body to be created from a swept wire body. This method sets open wires to be swept around an axis to create a solid body. When TRUE, the ends will be extended to the axis. Care should be taken so that all of the profile is on one side of the axis, to avoid a self–intersecting body. The default is FALSE.

The option close_to_axis may only be used when specifying the path as a position and axis of revolution.

```
public: void sweep_options::set_cut_end_off (
    int c                   // TRUE to cut
                            // perpendicular
                            // to sweep path
    );
```

The sweep option cut_end_off determines whether or not the end of the swept surface should be cut off squarely. When TRUE, the end of sweeping is planar and perpendicular to the path. When FALSE, the sweeping operation is faster, and a nonplanar profile is not cut off at the end of the sweep path. The default is FALSE.

```
public: void sweep_options::set_draft_angle (
    double                  // draft angle
    );
```

set_draft_angle represents the angle with which the swept profile is to draft out (positive) or in (negative) while sweeping.

```
public: void sweep_options::set_draft_hole (
    draft_hole_option       // draft hole
        in_draft_hole,      // option to set
    double angle            // draft hole angle
        = 0
    );
```

The sweep option draft_hole controls the offset behavior of internal loops (or holes) of a profile face when used to sweep with draft. AGAINST_PERIPHERY, ANGLE value, NO_DRAFT, or WITH_PERIPHERY may be set with this method. The default is AGAINST_PERIPHERY, in which holes are swept with the opposite draft direction as the periphery loop.

```
public: void sweep_options::set_draft_law (
    law*                    // law for drafting
    );
```

set_draft_law is a law pointer that is the offset draft distance as a function of the “pseudo–length” along the path. The pseudo–length is the parameter fraction along the path times the true length of the path.

```
public: void sweep_options::set_draft_repair_level (
    draft_repair_level l    // level of detection
    );
```

The sweep option draft_repair controls the level of detection for degeneracy and self–intersection during sweeping with draft. The default is DEGENERACY.

In ACIS, the sweep–with–draft surfaces are created by continuously offsetting the original profile along the path. If the offset amount is too large, an edge of the profile could be completely trimmed by adjacent edges. The sweep option draft_repair handles this problem. The option also handles the problem when sweeping a profile causes self–intersections.

```
public: void sweep_options::set_end_draft_dist (
    double                  // end draft distance
    );
```

set_start_draft_dist and set_end_draft_dist are generally used together. They specify an offset distance in the plane of the profile where the swept surface is to start and end.

```
public: void sweep_options::set_gap_type (
    int g                   // gap type
    );
```

set_gap_type specifies how to fill in gaps created by offsetting; valid integers are 2 for “natural”, 1 for “rounded”, and 0 for “extended”.

```
public: void sweep_options::set_keep_branches (
    logical k               // TRUE to keep all
    );
```

The keep_branches and keep_law options work together. The keep_branches option specifies whether or not graph branches should be kept. The keep_law option specifies in an ordered graph which items to keep. In this case, TRUE means that all branches are kept. The sweep_to_body option must be specified when using keep_branches.

```
public: void sweep_options::set_keep_law (
    law*                    // law to keep
    );
```

The sweep option keep_law specifies which portions of a graph are to be kept. Any law including conditional tests can be used to specify what is used for the sweep operation. When keep_law is TRUE, all graph pieces are saved. (For more information on graph theory, refer to the *Graph Theory* chapter in the *Kernel Component Manual*.)

```
public: void sweep_options::set_keep_start_face (
    logical k               // TRUE to keep start
                            // face
    );
```

The sweep option keep_start_face specifies that the face used to sweep remains available after the sweep operation. This is useful when the sweep path is a closed loop, or when selective Booleans will be used after the sweeping process.

For example, a circular profile can be swept around a circle, resulting in a torus. By default, once the torus is created, the start face is removed, along with information about where the torus surface starts and ends. When keep_start_face is turned on, the start face is left as part of the resulting model. This acts as a “handle” for the start and end location. If the swept torus were to be used in a selective boolean process, the handle would be necessary. For example, the API function api_boolean_tube_body makes use of the start face.

The sweep_to_body option should not be used with keep_start_face.

```
public: void sweep_options::set_miter (
    miter_type m            // option for miter
    );
```

Specifies how mitering is performed. The default is the enumeration value default_miter, which takes the system default. Other types include new_miter, old_miter, crimp_miter and bend_miter.

```
public: void sweep_options::set_miter_amount (
    double m                // miter radius
    );
```

Sets the minimum radius for using the miter_type bend_miter.

```
public: void sweep_options::set_portion_end (
SPAposition p               // position
    );
```

Set the end of the portion of the curve used for sweeping.

```
public: void sweep_options::set_portion_start (
SPAposition p               // position
    );
```

Set the start of the portion of the curve used for sweeping.

```
public: void sweep_options::set_rail_law (
    law*                    // law for rails
    );
```

set_rail_law specifies the orientation of the profile as it is swept. A rail law is simply a vector field along the sweep path. The field is made of unit vectors that are perpendicular to the path.

The default rail law for a non–planar path is minimum rotation, (e.g., the minor law function). For a planar path, the default rail law is a constant vector equal to the planar normal.

```
public: void sweep_options::set_rail_laws (
    law**,                  // array of laws
    int                     // size of array
    );
```

Sets an array of rail laws. A rail law is simply a vector field along the sweep path. The field is made of unit vectors that are perpendicular to the path. An array of rail laws supplied to the sweep operation dictates how the profile is to be oriented on the path. The size of the rail law array is determined by the number of segments in the path. Each path segment has its own associated rail law.

```
public: void sweep_options::set_rigid (
    logical r               // assignment
    );
```

Specifies whether or not to make the cross–sections of a sweep parallel to one another. The default is FALSE, which means that the cross–sections are perpendicular to the sweep path.

```
public: void sweep_options::set_scale_law (
    law*                    // law for scaling
    );
```

set_scale_law specifies the amount of scale in the *xyz* direction in the form of a vector law. The *x* direction is in the direction of the rail; the *z* direction is in the direction of the tangent of the path; and the *y* direction is in the direction *z* cross *x*.

```
public: void sweep_options::set_simplify (
    logical s                // simplify surface
                             // when TRUE
    );
```

When set to TRUE, swept surfaces are simplified to a geometric surface whenever possible. TRUE is the default. This method is primarily for internal use. Performance will be reduced when this option is set to FALSE.

```
public: void sweep_options::set_solid (
    int s                    // flag for solid
    );
```

set_solid is a boolean, which specifies whether the result of sweeping is intended to be a solid or a sheet body. If solid is set to FALSE then two_sided is set to TRUE, and the sweep operation creates a two–sided sheet body. If the intended result is a one–sided sheet body, then both the solid and two_sided options should be set to FALSE.

```
public: void sweep_options::set_start_draft_dist (
    double                   // start distance
    );
```

set_start_draft_dist and set_end_draft_dist are generally used together. They specify an offset distance in the plane of the profile where the swept surface is to start and end.

```
public: void sweep_options::set_steps (
    int s                    // linear segments
    );
```

Converts a circular sweep path into the specified number of linear segments. The results are polygons, and the intent is to create simpler geometry by keeping faces planar. The default is 0.

This option may only be used when specifying the path as a position and axis of revolution.

```
public: void sweep_options::set_sweep_angle(
    double s                 // angle in radians
    );
```

Specifies the angle to sweep around an axis, given in radians. The default is “2.0*M_PI”.

```
public: void sweep_options::set_sweep_portion (
    sweep_portion,                  // portion option
    SPAposition& in_st              // start position
        = * (SPAposition* ) NULL_REF,
    SPAposition& in_end             // end position
        = * (SPAposition* ) NULL_REF
    );
```

The sweep option portion permits localizing how much of the path is used for sweeping the profile. When positions are specified to localize the region for sweeping, the positions do not have to lie on the path. The closest point on the path from the specified position is used as the sweep limit.

The default is ENTIRE_PATH. Other options include BETWEEN_POINTS, FROM_PROFILE, TO_PROFILE and SWEEP_TO. AS_IS is also an option available for backward compatibility.

```
public: void sweep_options::set_sweep_to_body (
    BODY* b                 // pointer to
                            // termination body
    );
```

The sweep option sweep_to_body specifies a pointer to the body where sweeping is to finish. sweep_to_body is similar to to_face except that multiple faces on a body can be used to clip the sweep result properly. Equivalent results would be obtained if a profile were swept through a body, and if then the two bodies were Boolean united together. The default is NULL.

The sweep_to_body option is mutually exclusive with to_face. The sweep_to_body option should not be used with keep_start_face. This option must be specified when using keep_law or keep_branches.

```
public: void sweep_options::set_to_face (
    surface*                // surface for clipping
    );
```

The to_face sweep option permits clipping of the swept body at a surface. The default is NULL. The to_face option must be explicitly called. The to_face option is mutually exclusive with sweep_to_body.

```
public: void sweep_options::set_twist_angle (
    double                  // law for angle
    );
```

set_twist_angle represents how much the profile twists in total as the profile is swept along the path, regardless of the length of the path.

```
public: void sweep_options::set_twist_law (
    law*                    // law for twist
    );
```

set_twist_law should return the angle of twist in radians as a function of the length of the wire, starting at wire length equal to zero for the beginning of the wire.

```
public: void sweep_options::set_two_sided (
    logical t               // TRUE for two-sided
                            // sheet body
    );
```

The sweep option two_sided is a convenience flag. If solid is set to FALSE then two_sided is set to TRUE, and the sweep operation creates a two–sided sheet body. If the intended result is a one–sided sheet body, then both the solid and two_sided options should be set to FALSE.

```
public: void sweep_options::set_which_side (
    logical w               // flag for side.
    );
```

Establishes which side of the profile should be swept.

Internal Use: fix_up, fix_up

Related Fncs:

None