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.

cvty

Class: Model Topology, Model Geometry

Purpose: Class representing the convexity at a point or along a single edge (or something equivalent), such as "convex", or "tangent_convex" etc.

Derivation: cvty : –

SAT Identifier: None

Filename: intr/intersct/kernutil/cvty/cvxty.hxx

Description: For a precise enumeration of all the kinds of convexity value that can ever be returned see pt_cvty.hxx (convexity at a single point) and ed_cvty.hxx (convexity along an entire edge).

Limitations: None

References: by INTR ed_cvty_info, pt_cvty_info

Data:

None

Constructor:

```
public: cvty::cvty (
    unsigned int bits       // convexity
        = cvty_unset        // not set
    );
```

Default constructor.

Destructor:

None

Methods:

```
public: logical cvty::concave () const;
```

This will return TRUE for anything with the concave bit set.

```
public: logical cvty::concave_mixed () const;
```

This will return TRUE for anything with the concave_mixed bit set.

```
public: logical cvty::convex () const;
```

This will return TRUE for anything with the convex bit set (i.e. including tangent_convex, knife_convex etc.)

```
public: logical cvty::convex_mixed () const;
```

This will return TRUE for anything with the convex_mixed bit set.

```
public: void cvty::debug (
    FILE*                   // file to print to
        = debug_file_ptr    //
    ) const;
```

Debug printout. Outputs a title line and all the curves, pcurves, and surfaces details of the class for inspection to standard output or to the specified file.

```
public: logical cvty::inflect () const;
```

This will return TRUE for anything with the inflect bit set.

```
public: logical cvty::knife () const;
```

This will return TRUE for anything with the knife bit set.

```
public: logical cvty::knife_concave () const;
```

This will return TRUE for anything with the knife_concave bit set.

```
public: logical cvty::knife_concave_mixed () const;
```

This will return TRUE for anything with the knife_concave_mixed bit set.

```
public: logical cvty::knife_convex () const;
```

This will return TRUE for anything with the knife_convex bit set.

```
public: logical cvty::knife_convex_mixed () const;
```

This will return TRUE for anything with the concave bit set.

```
public: logical cvty::knife_mixed () const;
```

This will return TRUE for anything with the concave bit set.

```
public: logical cvty::mixed () const;
```

This will return TRUE for anything with the concave bit set.

```
public: logical cvty::operator== (
    cvty const& other       // other convexity
    ) const;
```

Test to see if two convexities are identical.

```
public: cvty& cvty::set_concave (
    logical                 // property setting –
        = TRUE              // default on
    );
```

Data setting functions. Pass the logical as TRUE to turn the property on, FALSE to turn it off.

```
public: cvty& cvty::set_convex (
    logical                 // property setting –
        = TRUE              // default on
    );
```

Data setting functions. Pass the logical as TRUE to turn the property on, FALSE to turn it off.

```
public: cvty& cvty::set_inflect (
    logical                 // property setting –
        = TRUE              // default on
    );
```

Data setting functions. Pass the logical as TRUE to turn the property on, FALSE to turn it off.

```
public: cvty& cvty::set_knife (
    logical                 // property setting –
        = TRUE              // default on
    );
```

Data setting functions. Pass the logical as TRUE to turn the property on, FALSE to turn it off.

```
public: cvty& cvty::set_mixed (
    logical                 // property setting –
        = TRUE              // default on
    );
```

Data setting functions. Pass the logical as TRUE to turn the property on, FALSE to turn it off.

```
public: cvty& cvty::set_tangent (
    logical                 // property setting –
        = TRUE              // default on
    );
```

Data setting functions. Pass the logical as TRUE to turn the property on, FALSE to turn it off.

```
public: cvty& cvty::set_unknown (
    logical                 // property setting –
        = TRUE              // default on
    );
```

Data setting functions. Pass the logical as TRUE to turn the property on, FALSE to turn it off.

```
public: char* cvty::string (
    char* str               // string size
    ) const;
```

The readable “string representation” of this thing, in case you want to do other things with it than send it to a file. The passed str must be big enough. Returns the given argument for convenience.

```
public: logical cvty::tangent () const;
```

Returns TRUE if this form of convexity applies.

```
public: logical cvty::tangent_concave () const;
```

Returns TRUE if this form of convexity applies.

```
public: logical cvty::tangent_concave_mixed () const;
```

Returns TRUE if this form of convexity applies.

```
public: logical cvty::tangent_convex () const;
```

Returns TRUE if this form of convexity applies.

```
public: logical cvty::tangent_convex_mixed () const;
```

Returns TRUE if this form of convexity applies.

```
public: logical cvty::tangent_inflect () const;
```

Returns TRUE if this form of convexity applies.

```
public: logical
    cvty::tangent_inflect_concave () const;
```

Returns TRUE if this form of convexity applies.

```
public: logical
    cvty::tangent_inflect_concave_mixed () const;
```

Returns TRUE if this form of convexity applies.

```
public: logical
    cvty::tangent_inflect_convex () const;
```

Returns TRUE if this form of convexity applies.

```
public: logical
    cvty::tangent_inflect_convex_mixed () const;
```

Returns TRUE if this form of convexity applies.

```
public: logical
    cvty::tangent_inflect_mixed () const;
```

Returns TRUE if this form of convexity applies.

```
public: logical cvty::tangent_mixed () const;
```

Returns TRUE if this form of convexity applies.

```
public: logical cvty::unknown () const;
```

Returns TRUE if this form of convexity applies.

```
public: logical cvty::unset () const;
```

Returns TRUE if this form of convexity applies.

Related Fncs:

None

edge_entity_rel

Class: Object Relationships

Purpose: Represents the relationship between an edge and an entity.

Derivation: edge_entity_rel : ACIS_OBJECT : –

SAT Identifier: None

Filename: intr/intersct/sg_husk/query/edentrel.hxx

Description: This class represents the relationship between an edge and an entity. The entity must be of the type BODY, FACE, EDGE, or POINT only.

Limitations: None

References: KERN EDGE, ENTITY, curve_curve_int, curve_surf_int

Data:

```
public EDGE_BODY_INT *ebrel_data;
```
The edge-body intersection data.

```
public EDGE_EDGE_INT *edrel_data;
```
The edge-edge intersection data.

```
public EDGE_FACE_INT *efrel_data;
```
The edge-face intersection data.

```
public curve_curve_int *ccrel_data;
```
The curve-curve intersection data.

```
public curve_surf_int *csrel_data;
```
The curve-surface intersection data.

```
public logical no_relation;
```
The variable that indicates that there is no possible relation between the given edge and the given entity.

```
public sg_edge_ent_rel_union rel_type;
```
The relationship type, which is determined from the union.

Constructor:

```
public: edge_entity_rel::edge_entity_rel ();
```

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

```
public: edge_entity_rel::edge_entity_rel (
    EDGE*,                      // edge
    ENTITY*,                    // entity
    edge_entity_rel* next       // next pointer
        = NULL,
    const SPAtransf& edge_trans // edge transform
        = * (SPAtransf* ) NULL_REF,
    const SPAtransf& ent_trans  // entity transform
        = * (SPAtransf* ) NULL_REF
    );
```

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

Destructor:

```
public: void edge_entity_rel::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. This is required because it has underlying entities associated with it.

Methods:

```
public: void edge_entity_rel::debug (
    FILE*,                  // file name
    const char* head        // heading text
        = NULL
    );
```

Writes debug information about edge_entity_rel to standard output or the specified file.

```
public: EDGE* edge_entity_rel::edge ();
```

Returns the edge in the edge-entity relationship.

```
public: ENTITY* edge_entity_rel::entity ();
```

Returns the entity in the edge-entity relationship.

```
public: edge_entity_rel* edge_entity_rel::next ();
```

Returns the next pointer in the edge-entity relationship.

```
public: void edge_entity_rel::set_edge (
    EDGE* edge              // edge
    );
```

Sets the edge in the edge-entity relationship.

```
public: void edge_entity_rel::set_entity (
    ENTITY* ent             // entity
    );
```

Sets the entity in the edge-entity relationship.

```
public: void edge_entity_rel::set_next (
    edge_entity_rel* next   // next pointer
    );
```

Sets the next pointer in the edge-entity relationship.

Related Fncs:

None

edge_face_int

Class: Intersectors, Object Relationships

Purpose: Records information about edge-face intersections.

Derivation: edge_face_int : ACIS_OBJECT : –

SAT Identifier: None

Filename: intr/intersct/kerndata/makeint/intsect.hxx

Description: This class records information about edge-face intersections.

Limitations: None

References:

by INTR	face_face_int
BASE	SPAparameter, SPAposition
KERN	BODY, EDGE, FACE, VERTEX, curve_surf_int

Data:

`public BODY *owning_body;`
Pointer to the body in which the edge lies.

`public EDGE *body_edge;`
Pointer to the edge on which the intersection lies.

`public EDGE *graph_edge;`
Pointer to the intersection graph edge corresponding to the next portion of the current edge, in which there is a coincidence and in which edges have been created.

`public FACE *adj_face;`
Pointer to the adjacent face to the edge, for which the auxiliary information is valid.

`public VERTEX *body_vertex;`
Pointer to the vertex at the end of an edge if the intersection is at that end; otherwise, it is NULL.

`public VERTEX *graph_vertex;`
Pointer to a vertex created for the intersection graph. It is initially set to NULL until the vertex is required. It may remain NULL if the intersection does not lie on any face lying on the surface.

`public curve_surf_int *aux_cs_int;`
The intersection between the edge curve and the auxiliary surface.

`public curve_surf_int *cs_int;`
The pointer to geometric information.

`public double int_quality;`
An indication of the reliability of this intersection, based on the sine of the angle between the face normals at the intersection. It is 1 for a right angled intersection and 0 at a tangency.

`public edge_face_int *next;`
For linking edge-face intersections.

`public edge_face_rel rel;`
Classifies whether the edge_face intersection is outside the face, inside or on the boundary of the face, or unknown.

`public logical edge_reversed;`
If the graph_edge_list is not NULL and if the graph edges have the opposite sense from the body edge it is TRUE; otherwise, it is FALSE. It is ignored if the graph_edge_list is NULL.

`public SPAparameter param;`
Parameter on the edge corresponding to int_point. It is initially set to the parameter value from the curve_surf_int.

`public SPAposition int_point;`
The position of the intersection in object space. It is initially the same as that in curve_surf_int, but may be adjusted within a neighborhood of the ideal position.

Constructor:

```
public: edge_face_int::edge_face_int (
    edge_face_int*,          // edge-face intersection
    EDGE*,                   // initialize body_edge
    curve_surf_int*          // initialize int_point &
                             // curve_surf_int
    );
```

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

Initializes the members and sets both vertex pointers and the graph edge to NULL. int_quality is initialized to 1 (perfect).

```
public: edge_face_int::edge_face_int (
    edge_face_int const&,    // edge-face intersection
    SPAposition const&,      // new int_point
    double                   // value of new parameter
    );
```

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

Makes a copy of the specified edge_face intersection for a separate point in a fuzzy region. It copies most information except the vertex or graph information associated with a particular position (which is defaulted). It also sets a NULL list pointer.

Destructor:

```
public: edge_face_int::~edge_face_int ();
```

Delete an edge_face_int.

Methods:

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

Writes the debug output for a edge-face intersection to the printer or the specified file name.

```
public: void edge_face_int::delete_list ();
```

Deletes all members of a list chained onto this one. It does not delete the first member in the chain (”this”).

Related Fncs:

debug_ef_int, print_ff_int, print_ff_int_list

ed_cvty_info

Class: Intersectors

Purpose: Returns the convexity of an edge (or equivalent).

Derivation: ed_cvty_info : –

SAT Identifier: None

Filename: intr/intersct/kernutil/cvty/ed_cvty.hxx

Description: This class represents the convexity of an edge (or equivalent), but where you don't necessarily want to commit to a particular angular tolerance. When you do want the verdict given a particular tolerance, you can "instantiate" this into an instance of the cvty class.

When instantiated it can yield the following values:

All the same ones as for points (see pt_cvty.hxx), meaning that every point on the edge has those same properties. Additionally we may get:

mixed – edge is both convex and concave in places
convex_mixed – the edge is everywhere convex or smooth (according to the first order terms)
concave_mixed – concave or smooth (to first order) everywhere
tangent_mixed – tangent everywhere, and second order terms fluctuate between convex and concave
tangent_convex_mixed – tangent everywhere, but second order terms imply convex or flat everywhere
tangent_concave_mixed – as above, but concave
tangent_inflect_mixed – tangent inflection, but varied between tangent_inflect_convex and tangent_inflect_concave
tangent_inflect_convex_mixed – everywhere tangent_inflect_convex or just tangent_inflect
tangent_inflect_concave_mixed – as above, but concave
knife_mixed – mixture of knife_convex and knife_concave
knife_convex_mixed – mixture of knife_convex and just knife
knife_concave_mixed – mixture of knife_concave and just knife

Limitations: None

References:

INTR	cvty
BASE	SPAinterval

Data:

None

Constructor:

```
public: ed_cvty_info::ed_cvty_info (
    int c                   // convexity
        = cvty_unset        //
    );
```

Default constructor to make an “unset” ed_cvty_info. Pass cvty_unknown to make an “unknown” one.

```
public: ed_cvty_info::ed_cvty_info (
    SPAinterval const& angs,// angles
    cvty tangent_cvty       // tangent convexity
    );
```

Generic constructor.

```
public: ed_cvty_info::ed_cvty_info (
    pt_cvty_info const&     // point to use
    );
```

Make an ed_cvty_info for an infinitesimal portion of edge around the given point, using that point’s pt_cvty_info.

Destructor:

None

Methods:

```
public: SPAinterval const& ed_cvty_info::angles ()
const;
```

The maximum and minimum angles between surface normals along this edge. Positive indicates convex, and negative indicates concave.

```
public: void ed_cvty_info::debug (
    FILE*                   // file to print to
        = debug_file_ptr    //
    ) const;
```

Prints out debug data on all the curves, pcurves, and surfaces to the specified file. Prints the “string representation” to the file.

```
public: cvty ed_cvty_info::instantiate (
    double tol              // angle tolerance
    ) const;
```

The actual convexity of this edge, as it appears when viewed using the given angle tolerance.

```
public: ed_cvty_info& ed_cvty_info::merge (
    ed_cvty_info const& eci // info to merge
    );
```

Merge the information currently in this object with information about a other points on the edge. Returns itself.

```
public: logical ed_cvty_info::operator== (
    ed_cvty_info const& other   // what to test
    ) const;
```

Test for equality. Just invoke "==" on all the bits.

```
public: char* ed_cvty_info::string (
    char* str               // string
    ) const;
```

The readable "string representation" of this thing, in case you want to do other things with it than send it to a file. The passed "str" must be big enough. Returns the given argument for convenience.

```
public: cvty
    ed_cvty_info::tangent_convexity () const;
```

The convexity of this edge, if the angle tolerance is such that the whole edge would be regarded as tangent.

```
public: logical ed_cvty_info::unknown () const;
```

By convention an infinite interval indicates an "unknown" ed_cvty_info. An evaluation can never result in an "unset" ed_cvty_info, but if it fails completely it can result in an "unknown" one.

```
public: logical ed_cvty_info::unset () const;
```

By convention an empty interval indicates an "unset" ed_cvty_info. An evaluation can never result in an "unset" ed_cvty_info, but if it fails completely it can result in an "unknown" one.

Related Fncs:

None

ERROR_ENTITY

Class: Intersectors, Error Handling, SAT Save and Restore

Purpose: Stores information about problem entities, to be used in an entity list of the body checker.

Derivation: ERROR_ENTITY : ENTITY : ACIS_OBJECT : –

SAT Identifier: “error_entity”

Filename: intr/intersct/sg_husk/sanity/err_ent.hxx

Description: This class is used to report problem entities. It is used in the Boolean “sanity check” code. If option check_ff_int in on (true) or if option check_level is greater than or equal to 70, extra checks are performed. These checks include:

- Adjacent faces intersect only in edges lying between the faces
- Faces that are not adjacent do not intersect
- Shell containment is correct and shells do not intersect
- No lump contains another lump, and lumps do not intersect
- When a face/face Boolean intersection fails, edge/edge intersection tests are performed each face

For each incorrect intersection or containment found, an ERROR_ENTITY is created and returned in the ENTITY_LIST of the body checker.

Each ERROR_ENTITY contains the following information:

- Pointers to each of the entities involved (for example, to each face incorrectly intersecting each other)
- An ENTITY_LIST containing edges and vertices that are in the intersection but not in the original body (this could be NULL when the problem is one of containment)
- An error ID number indicating the problem type

Limitations: None

References: KERN ENTITY, ENTITY_LIST

Data:

None

Constructor:

```
public: ERROR_ENTITY::ERROR_ENTITY ();
```

C++ allocation constructor requests memory for this object but does not populates it with the data. Applications should call this constructor only with the overloaded new operator inherited from the ENTITY class (for example, x=new ERROR_ENTITY(...)), because this reserves the memory on the heap, a requirement to support roll back and history management.

```
public: ERROR_ENTITY::ERROR_ENTITY (
    ENTITY* ent0,           // ptr to problem entity
    ENTITY* ent1,           // ptr to problem entity
    err_mess_type eid,      // error ID number
    ENTITY_LIST* i_list     // list of entities not
                            // on the body that lie
                            // on the improper
                            // intersection between
                            // ent0 and ent1
    );
```

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 inherited from the ENTITY class (for example, x=new ERROR_ENTITY(...)), because this reserves the memory on the heap, a requirement to support roll back and history management.

Destructor:

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

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

Methods:

```
public: virtual void ERROR_ENTITY::debug_ent (
    FILE*                   // output file
    ) 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: err_mess_type
    ERROR_ENTITY::get_error_id () const;
```

Returns the error number associated with entities.

```
public: ENTITY_LIST*
    ERROR_ENTITY::get_intersection_list () const;
```

Returns the intersection list between the two entities.

```
public: ENTITY* ERROR_ENTITY::get_owner (
    int i                    // owner number
    ) const;
```

Returns a pointer to specified owning entity.

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

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

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

Returns TRUE if this can be deep copied.

```
public: void ERROR_ENTITY::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 const char*
    ERROR_ENTITY::type_name () const;
```

Returns the string “error_entity”.

Internal Use: save, save_common

Related Fncs:

is_ERROR_ENTITY

ff_header

Class: Intersectors

Purpose: Enables lists of face-face intersections to be chained together.

Derivation: ff_header : ACIS_OBJECT : –

SAT Identifier: None

Filename: intr/intersct/kerndata/makeint/intsect.hxx

Description: This class enables lists of face-face intersections to be chained together.

Limitations: None

References: INTR face_face_int

Data:

```
public face_face_int *list;
```
List of face-face intersections.

```
public ff_header *next;
```
Next face-face intersection.

Constructor:

```
public: ff_header::ff_header (
    ff_header* next_head    // intersection header
        = NULL
    );
```

C++ constructor, creating a ff_header using the specified parameters. Initializes next to next head and list to NULL.

Destructor:

None

Methods:

None

Related Fncs:

debug_ef_int, print_ff_int, print_ff_int_list

hit

Class: Ray Testing

Purpose: Represents an intersection of a ray with a face, edge, or vertex.

Derivation: hit : ACIS_OBJECT : –

SAT Identifier: None

Filename: intr/intersct/kerndata/raytest/raytest.hxx

Description: This class represents an intersection of a ray with a face, edge, or vertex. Two types of hits can occur: a hit through a face, edge, or vertex where the ray coincides with the entity hit at an isolated point; and a hit along a (flat or ruled surface) face or (straight) edge, where the ray coincides with a finite region of the entity hit (when the parameter value is set to a value corresponding to a point within the coincident region of the ray).

Limitations: None

References:

BASE	SPAparameter
KERN	ENTITY

Data:

`public ENTITY* entity_hit;`
The face, edge, or vertex that is hit.

`public curve_surf_rel ray_surf_rel;`
An enumerated type that describes the relationship between the ray and the surface in one half neighborhood of the intersection.

`public hit* next;`
The next hit in a chain of hits.

`public hit_type type_of_hit;`
The type of hit, which can be either through a face edge or vertex (hit_thru) or along a face or edge (hit_along).

`public SPAparameter ray_param;`
If the type of hit is the through type, this provides the parameter value on the ray.

Constructor:

```
public: hit::hit (
    ENTITY*,                // initialize entity hit
    hit_type,               // initialize the type of
                            // hit
    double,                 // initialize the ray
                            // parameter
    curve_surf_rel          // initialize the ray
        = curve_unknown,    // surface relation
    hit*                    // initialize the next
        = NULL              // pointer
    );
```

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

Destructor:

None

Methods:

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

Outputs information about the debug to the printer or to the specified file.

Related Fncs:

debug_hit_list, enquire_hit_list, merge_hits, raytest, raytest_body, raytest_edge, raytest_face, raytest_lump, raytest_shell, raytest_vertex, raytest_wire, ray_ellipse_edge, ray_intcurve_edge, ray_straight_edge

insanity_data

Class: Debugging

Purpose: This class holds information about problems (insanities) found when checking an ACIS model.

Derivation: insanity_data : ACIS_OBJECT : –

SAT Identifier: None

Filename: intr/intersct/sg_husk/sanity/insanity_list.hxx

Description: As an ACIS model is checked, a list of problems or insanities is generated. The information about each insanity is stored an insanity_data object. The insanity_data objects are stored and accessed using the insanity_list class. The insanity_data class provides member methods for accessing information about the insanity.

Limitations: None

References:

by INTR	insanity_list
KERN	ENTITY

Data:

```
public char *aux_msg;
```
Insanity message.

```
public msg_data *aux_data;
```
Auxiliary message data.

Constructor:

```
public: insanity_data::insanity_data ();
```

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

```
public: insanity_data::insanity_data (
    ENTITY* e,              // insane entity
    int id,                 // insanity number
    insanity_type type,     // insanity type
    int(*pfunc)(ENTITY*,    // failed primary
    const SPATIAL::SPAtransf*,// function
    insanity_list*),        //
    insanity_list*(*sub_pfunc)(ENTITY*),//  failed
                            // subsequent function
    char* mesg              // insanity message
    );
```

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

```
public: insanity_data::insanity_data (
    ENTITY* e,              // insane entity
    int id,                 // insanity number
    insanity_type type,     // insanity type
    int(*pfunc)(ENTITY*,    // failed primary
    const SPATIAL::SPAtransf*,// function
    insanity_list*),        //
    insanity_list*(*sub_pfunc)(ENTITY*),//  failed
                            // subsequent function
    char* mesg,             // insanity message
    msg_data* data          // auxiliary message
                            // data
    );
```

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

```
public: insanity_data::insanity_data (
    ENTITY* e,              // insane entity
    int id,                 // insanity number
    insanity_type type,     // insanity type
    int(*pfunc)(ENTITY*,    // failed primary
    const SPATIAL::SPAtransf*,// function
    insanity_list*)         //
        = NULL,             //
    insanity_list*(*sub_pfunc)(ENTITY*)//  failed
        = NULL              // subsequent function
    );
```

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

```
public: insanity_data::insanity_data (
    ENTITY* e,              // insane entity
    int id,                 // insanity number
    insanity_type type,     // insanity type
    int(*pfunc)(ENTITY*,    // failed primary
    const SPAtransf*,       // function
    insanity_list*),        //
    insanity_list*(*sub_pfunc)(ENTITY*),//  failed
                            // subsequent function
    char* mesg              // insanity message
    );
```

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

```
public: insanity_data::insanity_data (
    ENTITY* e,              // insane entity
    int id,                 // insanity number
    insanity_type type,     // insanity type
    int(*pfunc)(ENTITY*,    // failed primary
    const SPAtransf*,       // function
    insanity_list*),        //
    insanity_list*(*sub_pfunc)(ENTITY*),//  failed
                            // subsequent function
    char* mesg,             // insanity message
    msg_data* data          // auxiliary message data
    );
```

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

```
public: insanity_data::insanity_data (
    ENTITY* e,              // insane entity
    int id,                 // insanity number
    insanity_type type,     // insanity type
    int(*pfunc)(ENTITY*,    // failed primary
    const SPAtransf*,       // function
    insanity_list*)         //
        = NULL,             //
    insanity_list*(*sub_pfunc)(ENTITY*)//  failed
        = NULL              // subsequent function
    );
```

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

Destructor:

```
public: insanity_data::~insanity_data ();
```

Methods:

```
public: ENTITY* insanity_data::get_ent ();
```

Gets ENTITY.

```
public: int insanity_data::get_insane_id ();
```

Gets insane_id.

```
public: const char* insanity_data::get_message ();
```

Gets the error or insanity message.

```
public: insanity_type insanity_data::get_type ();
```

Gets insanity type.

```
public: insanity_data::insanity_list* (
    *sub_chkfn)(ENTITY*      // Pointer to ENTITY
    );
```

Pointer to a failed subsequent function.

```
public: insanity_data::int (
    *chk_func)(ENTITY*,      // Pointer to ENTITY
    const SPATIAL::SPAtransf*,// Pointer to
                             // transformation matrix
    insanity_list*           // Pointer to insanity
                             // list
    );
```

Pointer to a failed primary function.

```
public: void insanity_data::print_message (
    FILE* fptr               // File pointer
    );
```

Prints the message into the file.

```
public: insanity_list* insanity_data::recheck ();
```

Re–checks the sanity of the entity with the failed check function.

Related Fncs:

None

insanity_list

Class: Debugging

Purpose: Implements a linked list of problems (insanities) that are found when checking a model.

Derivation: insanity_list : ACIS_OBJECT : –

SAT Identifier: None

Filename: intr/intersct/sg_husk/sanity/insanity_list.hxx

Description: This class is used to implement a linked list of problems (insanities) that are found when checking a model. Each problem is described by an insanity_data contained in the list. Also each insanity_list contains a pointer to another insanity_list which allows the formation of the list.

Limitations: None

References: None

Data:

None

Constructor:

```
public: insanity_list::insanity_list (
    insanity_data* data     //Insanity Data
        = NULL,             //
    insanity_list* next     //Pointer to next
        = NULL              //Insanity data
    );
```

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

Destructor:

```
public: insanity_list::~insanity_list ();
```

Deletes an insanity_list.

Methods:

```
public: void insanity_list::add_insanity (
    ENTITY* e,                  // Entity object
    int id,                     // Insanity Id
    insanity_type type,         // Type of Insanity
    int(*pfunc)(ENTITY*,        // Pointer to failed
    const SPAtransf*,           // primary function
    insanity_list*)             //
        = NULL,                 //
    insanity_list*(*sub_pfunc)(ENTITY*)// Pointer to
        = NULL                  //the failed subsequent
                                //function
    );
```

Makes an insanity data.

```
public: void insanity_list::add_insanity (
    insanity_data* data         //Insanity data
    );
```

Adds an insanity data to the list.

```
public: void insanity_list::add_insanity (
    insanity_list* new_list //Pointer to next
                            //Insanity data
    );
```

Creates a memory location for an insanity data.

```
public: void insanity_list::append_aux_msg (
    char*                   //Insanity message
    );
```

Appends the insanity message to the last insanity_data in the list.

```
public: int insanity_list::count (
    insanity_type           //Type of Insanity
        = ALL               //
    );
```

Count the number of instances of insanity.

```
public: insanity_data* insanity_list::data ();
```

Returns the insanity_data.

```
public: logical insanity_list::exist (
    int insanity_id         //Insanity Id
    );
```

Check if a specific insanity exists in the lists.

```
public: void insanity_list::make_entity_list (
    ENTITY_LIST* insane_ents,   //Entity list of
                                //insanities that
                                //are of same type
    insanity_type               //Type of Insanity
```

Add insane entities to the given ENTITY_LIST.

```
public: insanity_list* insanity_list::next ();
```

Returns the next list.

```
public: insanity_list* insanity_list::output ();
```

Returns a pointer to this insanity list.

```
public: void insanity_list::print_messages (
    FILE*,                  //File containing
                            //message
    insanity_type           //Type of Insanity
        = ALL               //
    );
```

Prints the insanity messages.

```
public: insanity_list* insanity_list::recheck (
    insanity_type           //Type of Insanity
        = ALL               //
    );
```

Re–check the sanity of entities only with the failed check functions.

Related Fncs:

None

mass_property

Class: Physical Properties

Purpose: Defines a class for manipulating mass properties.

Derivation: mass_property : moments : ACIS_OBJECT : –

SAT Identifier: None

Filename: intr/intersct/kernbody/massprop/massprop.hxx

Description: Define a class for manipulating mass properties.

Limitations: None

References: None

Data:

None

Constructor:

```
public: mass_property::mass_property ();
```

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

```
public: mass_property::mass_property (
    double,                 // zero moment
    SPAvector const&,       // first moment
    symtensor const&        // second moment
    );
```

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

```
public: mass_property::mass_property (
    mass_property const&    // existing mass property
    );
```

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

Destructor:

None

Methods:

```
public: mass_property const&
    mass_property::operator*= (
    double                  // double value
    );
```

Scales a mass_property by the given real.

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

Transforms the mass_property to correspond to the transformed body.

```
public: mass_property const&
    mass_property::operator+= (
    mass_property const&    // mass property
    );
```

Adds two mass properties together.

```
public: mass_property const&
    mass_property::operator-= (
    mass_property const&    // mass property
    );
```

Subtracts the given mass_property from this mass_property.

```
public: void mass_property::set_volume (
    double v                // volume
    );
```

Sets the volume of the mass_property.

```
public: double mass_property::volume () const;
```

Returns the volume of the mass_property.

Related Fncs:

```
friend: mass_property operator* (
    double,                 // double value
    mass_property const&    // mass property
);
```

Scales a mass_property by the given real.

```
friend: mass_property operator* (
     mass_property const&,  // mass property
    double                  // double value
 );
```

Scales a mass_property by the given real.

```
friend: mass_property operator* (
    mass_property const&,   // mass property
    SPAtransf const&        // transformation
);
```

Transforms the mass_property to correspond to the transformed body.

```
friend: mass_property operator+ (
    mass_property const&,   // first mass property
    mass_property const&    // second mass property
);
```

Adds two mass properties together.

```
friend: mass_property operator– (
    mass_property const&,   // first mass property
    mass_property const&    // second mass property
);
```

Subtracts the second mass_property from the first mass_property.

```
friend: mass_property operator– (
    mass_property const&    // mass property
);
```

Performs the unary minus operation.

moments

Class: Physical Properties

Purpose: Manipulates generic mass properties.

Derivation: moments : ACIS_OBJECT : –

SAT Identifier: None

Filename: intr/intersct/kernbody/massprop/massprop.hxx

Description: Manipulates generic “mass” properties. This supports volumetric, real, and linear measures.

Limitations: None

References:

BASE	SPAmatrix, SPAposition, SPAvector
KERN	symtensor

Data:

None

Constructor:

```
public: moments::moments ();
```

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

```
public: moments::moments (
    moments const&          // moment
    );
```

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

```
public: moments::moments (
    double,                 // integer
    SPAvector const&,       // vector
    symtensor const&        // symtensor
    );
```

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

Destructor:

None

Methods:

```
public: SPAposition moments::centroid ();
```

Return the centroid.

```
public: void moments::debug (
    char const*,            // character
    FILE*                   // file
        = debug_file_ptr
    ) const;
```

Outputs a title line and information about the mass property to the debug file or to the specified file.

```
public: double moments::first_error () const;
```

Returns the first error of a moment, or zero if unset.

```
public: SPAvector moments::first_moment () const;
```

Returns the first moment of the moments, in a vector, or zeroes if unset.

```
public: logical moments::first_set () const;
```

Returns TRUE if the first moments have been set.

```
public: moments const& moments::operator*= (
    double                  // factor
    );
```

Scale the mass properties by a constant factor.

```
public: moments const& moments::operator*= (
    SPAtransf const&        // double
    );
```

Transform the mass properties to correspond to the transformed body.

```
public: moments const& moments::operator+= (
    moments const&          // moment
    );
```

Add (accumulate) moments together.

```
public: moments const& moments::operator-= (
    moments const&          // moment
    );
```

Subtract moments.

```
public: SPAposition moments::origin () const;
```

Returns the origin point from which the first and second moments of the mass property are measured.

```
public: SPAmatrix moments::princ_axes ();
```

Returns the principle axes of the mass_property.

```
public: SPAvector moments::princ_inertias ();
```

Returns the principle moments of inertia.

```
public: double moments::second_error () const;
```

Returns the second error of a moment, or zero if unset.

```
public: symtensor moments::second_moment () const;
```

Returns the second moments of the moment, in a symtensor, or zeroes if unset.

```
public: logical moments::second_set () const;
```

Returns TRUE if the second moments have been set.

```
public: void moments::set_first_error (
    double e                // double
    );
```

Sets the first error of a moment.

```
public: void moments::set_first_moment (
    SPAvector const&        // vector
    );
```

Sets the first moment of the moment.

```
public: void moments::set_origin (
    SPAposition const&      // position
    );
```

Sets the origin from which the first and second moments are measured.

```
public: void moments::set_second_error (
    double e                // double
    );
```

Sets the second error of a moment.

```
public: void moments::set_second_moment (
    symtensor const&        // symtensor
    );
```

Sets the second moment of the moment.

```
public: void moments::set_zeroth_error (
    double e                // double
    );
```

Sets the zeroth error of a moment.

```
public: void moments::set_zeroth_moment (
    double                  // double
    );
```

Sets the zeroth moment of the moment.

```
public: logical moments::unset () const;
```

Returns TRUE if no moments are set.

```
public: void moments::zero ();
```

Initializes the moment to zero.

```
public: double moments::zeroth_error () const;
```

Returns the zeroth error of a moment, or zero if unset.

```
public: double moments::zeroth_moment () const;
```

Returns the zeroth moment of the moment, or zero if unset.

```
public: logical moments::zeroth_set () const;
```

Returns TRUE if the zeroth moments have been set.

Related Fncs:

```
friend: moments operator* (
    moments const&,          // moments
    double                   // factor
);
```

Scale the mass properties by a constant factor.

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

Transform the mass properties to correspond to the transformed body.

```
friend: moments operator* (
    double,                  // factor
    moments const&           // moments
);
```

Scale the mass properties by a constant factor.

```
friend: moments operator+ (
    moments const&,          // first moment
    moments const&           // second moment
);
```

Add (accumulate) mass properties.

```
friend: moments operator- (
    moments const&          // moment
);
```

Subtract mass properties.

```
friend: moments operator- (
    moments const&,         // first moment
    moments const&          // second moment
);
```

Subtract mass properties.

param_info

Class: Object Relationships

Purpose: Data representing type and in some cases parameter information about a point on an entity.

Derivation: param_info : ACIS_OBJECT : –

SAT Identifier: None

Filename: intr/intersct/kernapi/api/intrapi.hxx

Description: This class holds type and in some cases parameter data about point on an entity. It is used by some query apis to return detailed information about points they have identified on entities e.g. a point of minimum distance to another entity.

Limitations: This is a data storage class only. It does not make any attempt to maintain consistent data. It is the responsibility of the users of this class to ensure that instances are valid.

References:

BASE	SPApar_pos, SPAparameter
KERN	ENTITY

Data:

None

Constructor:

```
public: param_info::param_info ();
```

Default constructor. Entity pointer and parameter values initialised to 0.

Destructor:

None

Methods:

```
public: ENTITY* param_info::entity ();
```

Returns a pointer to the entity referred to by this instance.

```
public: param_info_type param_info::entity_type ();
```

Returns the type of the entity referred to by this instance.

```
public: param_info const& param_info::operator= (
    param_info const&  rhs  // parameter information
    );
```

Assignment operator.

```
public: void param_info::set_entity (
    ENTITY* ent             //entity
    );
```

Sets the entity referred to by this instance.

```
public: void param_info::set_entity_type (
    param_info_type ent_type//type of entity
    );
```

Sets the entity type for this instance.

```
public: void param_info::set_t (
    SPAparameter t          //parameter value
    );
```

Sets the parameter value for this instance.

```
public: void param_info::set_uv (
    SPApar_pos uv           //position co-ordinates
    );
```

Sets the (u,v) parameter position for this instance.

```
public: SPAparameter param_info::t ();
```

Returns the parameter of the point represented by this instance. This will be meaningful only if the entity referred to is an edge.

```
public: SPApar_pos param_info::uv ();
```

Returns the (u,v) parameters of the point represented by this instance. This will be meaningful only if the entity referred to is a face.

Related Fncs:

api_check_cur_smoothness, api_check_edge, api_check_entity, api_check_face, api_check_wire_self_inters, api_create_boundary_field, api_crv_self_inters, api_edent_rel, api_edfa_int, api_edge_convexity_param, api_ed_self_inters, api_entity_entity_distance, api_entity_entity_touch, api_entity_extrema, api_entity_point_distance, api_facet_curve, api_face_nu_nv_isolines, api_face_u_iso, api_face_v_iso, api_find_cls_ptto_face, api_get_ents, api_initialize_intersectors, api_intersect_curves, api_inter_ed_ed, api_point_in_body, api_ptent_rel, api_raytest_body, api_raytest_ents, api_ray_test_body, api_ray_test_ents, api_set_entity_pattern, api_silhouette_edges, api_terminate_intersectors

point_entity_rel

Class: Object Relationships

Purpose: Relates an APOINT to an ENTITY.

Derivation: point_entity_rel : ACIS_OBJECT : –

SAT Identifier: None

Filename: intr/intersct/sg_husk/query/ptentrel.hxx

Description: This class relates an APOINT to a list of entities.

Limitations: None

References: KERN APOINT, ENTITY

Data:

```
public logical no_relation;
```

Indicates that there is no possible relation between the given point and the given entity.

```
public sg_point_ent_relation rel_type;
```
Determines the relation type from the union of the point and the entity.

Constructor:

```
public: point_entity_rel::point_entity_rel (
    APOINT*,                    // point
    ENTITY*,                    // entity
    point_entity_rel* next      // next pointer
        = NULL,
    SPAtransf& ent_trans        // transform
        = * (SPAtransf* ) NULL_REF
    );
```

C++ constructor, creating a point_entity_rel using the specified parameters. The next pointer represents a relation between a point and a list of entities.

Destructor:

```
public: void point_entity_rel::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.

Methods:

```
public: void point_entity_rel::debug (
    FILE*,                  // file name
    char* head              // text heading
        = NULL
    );
```

Write the debug output for a point-entity relationship to standard output or to the specified file.

```
public: ENTITY* point_entity_rel::entity ();
```

Determine the ENTITY in the point-entity relationship.

```
public: point_entity_rel* point_entity_rel::next ();
```

Determine the next point-entity relationship.

```
public: APOINT* point_entity_rel::point ();
```

Determine the APOINT in the point-entity relationship.

```
public: void point_entity_rel::set_next (
    point_entity_rel* next  // next pointer
    );
```

Set the next point-entity relationship. The next pointer represents a relation between a point and a list of entities.

Related Fncs:

None

pt_cvty_info

Class: Object Relationships

Purpose: Returns the convexity of a single point along an edge (or equivalent).

Derivation: pt_cvty_info : –

SAT Identifier: None

Filename: intr/intersct/kernutil/cvty/pt_cvty.hxx

Description: When instantiated, the convexity values that can be produced for a single point are:

convex – edge is convex at this point
concave – edge is concave at this point
tangent_convex – edge is smooth here, but 2nd order terms imply convex (both surfaces have convex curvature here)
tangent_concave – smooth here, second order terms imply concave
tangent_inflect – smooth here,second order terms imply one surface is convex, the other concave, and curvatures are the same
tangent_inflect_convex – as above, but the convex surface is more curved than the concave one
tangent_inflect_concave – as above, but the concave surface is more curved
knife – surfaces antiparallel here, could not tell from curvatures whether the ”knife” is made of material or air
knife_convex – a ”knife” made of material
knife_concave – a ”knife” made of air

Limitations: None

References: INTR cvty

Data:

None

Constructor:

```
public: pt_cvty_info::pt_cvty_info (
    double ang,             // angle
    cvty tangent_cvty,      // convexity
    double default_tol      // tolerance
    );
```

Generic constructor. Make an “unset” or “unknown” one. The default argument makes an unset one. Pass cvty_unknown to make an unknown one.

```
public: pt_cvty_info::pt_cvty_info (
    int c                   // convexity
        = cvty_unset        //
    );
```

Default copy constructor.

Destructor:

None

Methods:

```
public: double pt_cvty_info::angle () const;
```

The convexity of this point, if using an angle tolerance such that this point is regarded as tangent.

```
public: void pt_cvty_info::debug (
    FILE*                   // file to print to
        = debug_file_ptr    //
    ) const;
```

Debug. Prints the “string representation” to the file.

```
public: cvty pt_cvty_info::instantiate (
    double tol              // angle tolerance
    ) const;
```

The actual convexity of this point, as it appears when viewed using the given angle tolerance. tol may be passed as –1, meaning to use the “default tolerance”, which is a value derived from the surface curvatures at this point (if default_tol is not set, the result will be “unknown” convexity).

```
public: logical pt_cvty_info::operator== (
    pt_cvty_info const& other   // pointer to what's
                                // tested
    ) const;
```

Test for equality. Just invoke "==" on all the bits.

```
public: char* pt_cvty_info::string (
    char* str               // string
    ) const;
```

The readable "string representation" of this thing, in case you want to do other things with it than send it to a file. The passed str must be big enough. Returns the given argument for convenience.

```
public: cvty
    pt_cvty_info::tangent_convexity () const;
```

The convexity of this point, if the angle tolerance is such that this point would be regarded as tangent.

```
public: logical pt_cvty_info::unknown () const;
```

An evaluation happened, but failed.

```
public: logical pt_cvty_info::unset () const;
```

This is not the result of any evaluation.

Related Fncs:

None

ray

Class: Ray Testing

Purpose: Represents a 3D ray.

Derivation: ray : ACIS_OBJECT : –

SAT Identifier: None

Filename: intr/intersct/kerndata/raytest/raytest.hxx

Description: This class represents a 3D ray.

Limitations: None

References: BASE SPAposition, SPAunit_vector

Data:

```
public double max_p;
```
The maximum parameter value of hits that have been found so far.

```
public double radius;
```
The ray radius.

```
public int hits_found;
```
The number of hits found.

```
public int hits_wanted;
```
The maximum number of hits wanted. If this value is 0, it means that all hits are wanted.

```
public SPAposition root_point;
```
The point on the ray.

```
public SPAunit_vector direction;
```
The direction of the ray.

Constructor:

```
public: ray::ray (
    SPAposition const&,      // point on ray
    SPAunit_vector const&,   // direction of ray
    double                   // ray radius
        = 100.0* SPAresabs,
    int                      // maximum number of hits
        = 0
    );
```

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

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

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

Destructor:

None

Methods:

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

Outputs debug information to the printer or to the specified file.

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

Transforms a ray.

Related Fncs:

debug_hit_list, enquire_hit_list, merge_hits, raytest, raytest_body, raytest_edge, raytest_face, raytest_lump, raytest_shell, raytest_vertex, raytest_wire, rayxbox, ray_ellipse_edge, ray_intcurve_edge, ray_straight_edge

Intersects a ray with a wire.

```
friend: logical operator&& (
    ray const&,             // input ray
    SPAbox const&           // input box
    );
```

Tests if the ray cuts the box.

```
friend: ray operator* (
    ray const&,             // input ray
    SPAtransf const&        // transform
    );
```

Transforms a ray.