# *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.

## ATTRIB_EYE

Class: Faceting, SAT Save and Restore

Purpose: Defines an organization attribute class.

Derivation: ATTRIB_EYE : ATTRIB : ENTITY : ACIS_OBJECT : –

SAT Identifier: ”eye”

Filename: fct/faceter/attribs/atteye3d.hxx

Description: This is an organization attribute class. Its methods are virtual and do nothing. User is supposed to implement these methods for classes derived from ATTRIB_EYE.

Limitations: None

References: None

Data:

None

Constructor:

```
public: ATTRIB_EYE::ATTRIB_EYE (
    ENTITY*                 // entity to attach to
        = 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 inherited from the ENTITY class (for example, x=new ATTRIB_EYE(...)), because this reserves the memory on the heap, a requirement to support roll back and history management.

Destructor:

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

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 ATTRIB_EYE(...) then later x–>lose.)

Methods:

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

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

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

Returns TRUE if this can be deep copied.

```
public: void ATTRIB_EYE::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*
    ATTRIB_EYE::type_name () const;
```

Returns the string “eye”.

Internal Use: save, save_common

Related Fncs: is_ATTRIB_EYE

# ATTRIB_EYE_ATTACHED_MESH

Class: Faceting, SAT Save and Restore

Purpose: Defines an attribute to attach facets to an entity as a MESH.

Derivation: ATTRIB_EYE_ATTACHED_MESH : ATTRIB_EYE : ATTRIB : ENTITY : ACIS_OBJECT : –

SAT Identifier: “fmesh”

Filename: fct/faceter/attribs/meshat.hxx

Description: The mesh manager in the faceter leaves it up to the application to determine where it will store the meshes. One possibility is to attach a mesh to its owning face. This class is intended to be a utility for this purpose. It attaches any mesh that is derived from MESH. Typically it will be used to attach the mesh (facets) of a face to the face.

Limitations: None

References: FCT MESH

Data:

None

Constructor:

```
public:
ATTRIB_EYE_ATTACHED_MESH::ATTRIB_EYE_ATTACHED_MESH  (
    ENTITY*                  // owner of this mesh
        = NULL,
    MESH* meshptr            // mesh to attach
        = NULL,
    MESH_APP_ID app_id       // application id
        = 0,
    MESH_USER_ID user_id     // user id
        = 0
    );
```

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 ATTRIB_EYE_ATTACHED_MESH(...)), because this reserves the memory on the heap, a requirement to support roll back and history management.

Initializes internal data. The application identification should be unique to the application. The user identification allows multiples meshes to be attached to the entity.

Destructor:

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

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

Methods:

```
public: void
ATTRIB_EYE_ATTACHED_MESH::change_state_id ();
```

Increment the state ID.

```
public: virtual void
    ATTRIB_EYE_ATTACHED_MESH::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: MESH_APP_ID
    ATTRIB_EYE_ATTACHED_MESH::get_app_id ();
```

Returns the application identification of this attribute.

```
public: MESH* ATTRIB_EYE_ATTACHED_MESH::get_mesh ();
```

Returns the mesh of this attribute.

```
public: MESH const* ATTRIB_EYE_ATTACHED_MESH::
    get_mesh_const() const;
```

Returns the mesh constant of this attribute.

```
public: MESH_USER_ID
    ATTRIB_EYE_ATTACHED_MESH::get_user_id ();
```

Returns the user identification of this attribute.

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

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

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

Returns TRUE if this can be deep copied.

```
public: virtual void
    ATTRIB_EYE_ATTACHED_MESH::merge_owner (
    ENTITY* mate_entity,     // given entity
    logical delete_this      // deleting owner
    );
```

Notifies the ATTRIB_EYE_ATTACHED_MESH that its owning ENTITY is about to be merged with given entity. The application has the chance to delete or otherwise modify the attribute. After the merge, this owner will be deleted if the logical deleting owner is TRUE, otherwise it will be retained and other entity will be deleted. The default action is to do nothing. This function is supplied by the application whenever it defines a new attribute, and is called when a merge occurs.

```
public: virtual logical
    ATTRIB_EYE_ATTACHED_MESH::pattern_compatible (
    ) const;
```

Returns TRUE if this is pattern compatible.

```
public: void
    ATTRIB_EYE_ATTACHED_MESH::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 ATTRIB_EYE_ATTACHED_MESH::set_app_id (
    MESH_APP_ID              // application id to set
    );
```

Sets the application identification of this attribute.

```
public: void ATTRIB_EYE_ATTACHED_MESH::set_mesh (
    MESH*                   // mesh to set
    );
```

Sets the mesh of this attribute.

```
public: void ATTRIB_EYE_ATTACHED_MESH::set_user_id (
    MESH_USER_ID            // user id to set
    );
```

Sets the user identification of this attribute.

```
public: virtual void
    ATTRIB_EYE_ATTACHED_MESH::split_owner (
    ENTITY* new_piece       // new entity
    );
```

Notifies the ATTRIB_EYE_ATTACHED_MESH that its owner is about to be split into two parts. The application has the chance to duplicate or otherwise modify the attribute. The default action is to do nothing. This function is supplied by the application whenever it defines a new attribute, and is called when a split occurs.

```
public: virtual void
    ATTRIB_EYE_ATTACHED_MESH::trans_owner (
    SPAtransf const&        // transformation
    );
```

Notifies the ATTRIB_EYE_ATTACHED_MESH that its owner is about to be transformed. The application has the chance to transform the attribute. The default action is to do nothing. This function is supplied by the application whenever it defines a new attribute, and is called when a transformation occurs.

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

Return the string “fmesh”.

```
public: virtual void
    ATTRIB_EYE_ATTACHED_MESH::warp_owner (
    law*                    // warp law
    );
```

Warp the owner of the attribute.

Internal Use: lop_change_owner, replace_owner_geometry, save, save_common

Related Fncs: `af_delete_facets`, af_query, af_update, is_ATTRIB_EYE_ATTACHED_MESH

# GLOBAL_MESH_MANAGER

Class: Faceting

Purpose: Shows an example of a mesh manager.

Derivation: GLOBAL_MESH_MANAGER : MESH_MANAGER : ACIS_OBJECT : –

SAT Identifier: None

Filename: fct/faceter/meshmgr/gmeshmg.hxx

Description: The GLOBAL_MESH_MANAGER class is an example of a mesh manager using the global indexed protocol. The mesh is written to the standard output. Because there are no graphics or permanent storage services, the entire mesh manager functionality resides compactly in a single file.

This example demonstrates the virtual member functions of MESH_MANAGER that need to be redefined for this protocol.

Limitations: None

References: None

Data: None

Constructor:

```
public: GLOBAL_MESH_MANAGER::GLOBAL_MESH_MANAGER ();
```

C++ constructor, creating a GLOBAL_MESH_MANAGER.

Destructor: None

Methods:

```
public: virtual void
    GLOBAL_MESH_MANAGER::announce_counts (
    int npoly,              // number of polygons
    int nnode,              // number of nodes
    int npolynode           // number of node
                            // references
    );
```

Announces the number of polygons, nodes, and node references by polygons, and prints them.

```
public: virtual void*
    GLOBAL_MESH_MANAGER::announce_global_node (
    int inode,              // node index
    EDGE*,                  // edge
    const SPAposition& X,   // node coordinates
    double t                // curve parameter
    );
```

Announces a node on a model EDGE with its curve parameter, and prints it.

```
public: virtual void*
    GLOBAL_MESH_MANAGER::announce_global_node (
    int inode,              // 0-based index of node
    FACE*,                  // face
    const SPAposition& X,   // node coordinates
    const SPApar_pos& uv    // surface parameter
    );
```

Announces a node on a model FACE with its surface parameters, and prints it.

```
public: virtual void*
    GLOBAL_MESH_MANAGER::announce_global_node (
    int inode,              // node index
    VERTEX*,                // vertex
    const SPAposition& X    // node coordinates
    );
```

Announces a node on a model VERTEX, and prints it.

```
public: virtual void
    GLOBAL_MESH_MANAGER::announce_indexed_polynode (
    ENTITY*,                // entity
    int,                    // integer
    int,                    // integer
    void*                   // void
    );
```

Gets a polynode of a polygon and prints the identification.

```
public: virtual void
    GLOBAL_MESH_MANAGER::announce_indexed_polynode (
    ENTITY*,                // coedge along node to
                            // next node
    int,                    // polygon index
    int,                    // local node index
    void*,                  // node identifier
    const double&,          // if the node lies on an
                            // edge then returns tpar
    const SPApar_pos&,      // parametric coordinates
    const SPAposition&,     // Cartesian coordinates
    const SPAunit_vector&   // surface normal
    );
```

Gets a polynode of a polygon and prints the identification.

```
public: virtual void
     GLOBAL_MESH_MANAGER::announce_indexed_polynode (
    int ipoly,                  // polygon index
    int i,                      // counter in polygon
    void* id                    // node ID
    );
```

Gets a polynode of a polygon and prints the identification. The polygon index and counter are zero–based. The counter increments sequentially on successive calls. The node identifier is as previously received from announce_indexed_node.

```
public: virtual void
    GLOBAL_MESH_MANAGER::end_indexed_polygon (
    int ipoly               // polygon index
    );
```

The indexed polygon is complete. Prints a new line.

```
public: virtual logical GLOBAL_MESH_MANAGER::
    need_global_indexed_polygons ();
```

Tells faceter that globally indexed polygon output is needed.

```
public: virtual logical GLOBAL_MESH_MANAGER::
    need_precount_of_global_indexed_polygons ();
```

If this function returns TRUE, the number of polygons, nodes, and node references by polygons are announced before other output. The default returns FALSE.

```
public: virtual void*
    GLOBAL_MESH_MANAGER::null_node_id ();
```

Returns the node id value that is guaranteed to be invalid. Applications using indices will typically return –1. Applications using pointers will typically return 0. The default returns –1.

```
public: virtual void
    GLOBAL_MESH_MANAGER::start_indexed_polygon (
    int ipoly,              // polygon index
    int npolynode,          // number of nodes
    int ishare              // info about which edge
        = -2                // of previous polygon is
                            // is shared with this
                            // one
    );
```

A new polygon starts and prints the polygon header.

Related Fncs:

None

# INDEXED_MESH

Class: Faceting, Viewing

Purpose: Shows an example of a mesh that stores indices to a vertex array.

Derivation: INDEXED_MESH : MESH : ACIS_OBJECT : –

SAT Identifier: None

Filename: fct/faceter/meshmgr/idx_mesh.hxx

Description: An INDEXED_MESH is a mesh format that minimizes memory places its data in contiguous arrays, presumably minimizing memory fragmentation. The indexed mesh is stored as an array of nodes and a set of polygons defined by (numerical) indices into the node set. This is a “packed” representation in which both polygons and nodes are referenced via integer that index into arrays.

The disadvantage of the INDEXED_MESH is that all of the memory is allocated up front, so you must know how many polygons and vertices will be created when you create the mesh. You also cannot go back and add new vertices and polygons.

This class represents an alternate version of SIMPLE_INDEXED_MESH.

Limitations: None

References:

| | |
|---|---|
| FCT | indexed_polygon, polygon_vertex |
| by FCT | INDEXED_MESH_MANAGER |

Data:

None

Constructor:

```
public: INDEXED_MESH::INDEXED_MESH (
    const INDEXED_MESH&     // indexed mesh
    );
```

C++ constructor, creating an INDEXED_MESH by copying one.

```
public: INDEXED_MESH::INDEXED_MESH (
    int max_vertex,         // maximum vertex
    int max_poly,           // maximum polygon
    int max_polynode        // maximum polynode
    );
```

C++ constructor, creating an INDEXED_MESH using the specified parameters.

Destructor:

```
public: INDEXED_MESH::~INDEXED_MESH ();
```

C++ destructor, deleting an INDEXED_MESH.

Methods:

```
public: int INDEXED_MESH::add_polygon (
    int ipoly,                // polygon index
    int num_vertex,           // number vertex
    VERTEX_TEMPLATE*          // pointer to class
        vertex_template       // vertex template
        = NULL,
    int ishare                // info about which edge
        = -2                  // of previous polygon is
                              // is shared with this
                              // one
    );
```

Adds a polygon.

```
public: polygon_vertex* INDEXED_MESH::add_vertex (
    const polygon_vertex&     // vertex to add
    );
```

Adds a vertex to the current polygon.

```
public: polygon_vertex* INDEXED_MESH::add_vertex (
    const SPAposition&,       // position of vertex
    const SPAunit_vector&,    // normal at vertex
    const SPApar_pos&         // parameter position
    );
```

Adds a vertex to the current polygon.

```
public: virtual SPAbox INDEXED_MESH::get_box ();
```

Returns the bounding box of the mesh.

```
public: const SPAunit_vector&
INDEXED_MESH::get_normal (
    int inode                 // index to node
    ) const;
```

Gets the normal at a node.

```
public: int INDEXED_MESH::get_num_polygon () const;
```

Returns the number of polygon on mesh.

```
public: int INDEXED_MESH::get_num_polynode () const;
```

Returns the number of polynodes on mesh.

```
public: int INDEXED_MESH::get_num_vertex () const;
```

Returns the number of vertices on mesh.

```
public: indexed_polygon* INDEXED_MESH::get_polygon (
    int poly_index          // index to polygon
    ) const;
```

Gets the pointer to a polygon.

```
public: const SPAposition& INDEXED_MESH::get_position
(
    int inode               // index to node
    ) const;
```

Gets the position of a node.

```
public: SPApar_pos INDEXED_MESH::get_uv_as_entered (
    int inode               // index to node
    );
```

Given a vertex number, get the ”as entered” *uv* value.

```
public: SPApar_pos INDEXED_MESH::get_uv_as_scaled (
    int inode               // index to node
    );
```

Given a vertex number, get a copy of the ”as scaled” (0 to 1) *uv* value.

```
public: polygon_vertex&
    INDEXED_MESH::get_vertex (
    int inode               //index to node
    ) const;
```

Gets the vertex at the node.

```
public: polygon_vertex&
    INDEXED_MESH::get_vertex_for_edit (
    int inode               // node
    );
```

Gets an editable version of the vertex. This allows one to iterate through the array of vertices and make modifications without having to use the polygon methods and visit the vertices multiple times.

```
public: int INDEXED_MESH::get_vertex_index (
    const polygon_vertex*   // polygon vertex
    ) const;
```

Gets the index of a polygon vertex.

```
public: void INDEXED_MESH::map_uv_into_01 ();
```

Maps *uv*-parameter into a range between 0 and 1.

```
public: INDEXED_MESH& INDEXED_MESH::operator|= (
    const INDEXED_MESH&     // mesh to test
    );
```

Compares this instance with the given indexed mesh.

```
public: void INDEXED_MESH::reverse ();
```

Reverses the indexed mesh.

```
public: int INDEXED_MESH::set_poly_vertex (
    int ipoly,              // polygon index
    int vertex_number,      // index of the vertex
    polygon_vertex*         // data of the vertex
    );
```

Sets the data of the polygon vertex.

```
public: virtual void INDEXED_MESH::write (
    FILE*,                  // output file stream
    const SPAtransf*        // transformation
        = NULL
    );
```

Writes the indexed mesh to a file.

```
public: virtual void INDEXED_MESH::write_raw (
    FILE*,                  // output file stream
    const SPAtransf*        // transformation
        = NULL
    );
```

Writes the mesh to a file.

Related Fncs:

None

# INDEXED_MESH_MANAGER

Class: Faceting, Viewing

Purpose: Shows an example of a mesh manager using the indexed protocol.

Derivation: INDEXED_MESH_MANAGER : MESH_MANAGER : ACIS_OBJECT : –

SAT Identifier: None

Filename: fct/faceter/meshmgr/idx_mm.hxx

Description: INDEXED_MESH_MANAGER constructs INDEXED_MESHes from facet data that the faceter announces through MESH_MANAGER.

Limitations: None

References: FCT INDEXED_MESH, VERTEX_TEMPLATE

Data:

`protected INDEXED_MESH *m_pMesh;`
Identifies the current mesh being constructed.

`protected VERTEX_TEMPLATE* vertex_template;`
Defines data attached to node.

`protected const SPAtransf* m_pTransform;`
Identifies the transformation that needs to be applied to nodes.

`protected int m_nNumPolygon;`
Identifies the total number of polygons in all meshes.

Constructor:

```
public:
    INDEXED_MESH_MANAGER::INDEXED_MESH_MANAGER ();
```

C++ constructor, creating an INDEXED_MESH_MANAGER.

Destructor:

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

C++ destructor, deleting an INDEXED_MESH_MANAGER.

Methods:

```
public: virtual void
    INDEXED_MESH_MANAGER::announce_counts (
    int npoly,              // polygon count
    int nnode,              // node count
    int nref                // node reference count
    );
```

Announces the number of polygons, nodes, and node references by polygons. Creates a mesh with the proper storage.

```
public: virtual void*
    INDEXED_MESH_MANAGER::announce_indexed_node(
    int inode,              // global node number
    const SPApar_pos& uv,   // parameter position
    const SPAposition& X,   // position
    const SPAunit_vector& N // unit vector
    );
```

Announce the indexed node.

```
public: virtual void
    INDEXED_MESH_MANAGER::announce_indexed_polynode (
    ENTITY*,                // entity
    int,                    // integer
    int,                    // integer
    void*                   // void
    );
```

Gets an indexed polynode. Stores the data into the polygon.

```
public: virtual void
    INDEXED_MESH_MANAGER::announce_indexed_polynode (
    int ipoly,              // polygon index
    int i,                  // node position
    void* pnode             // polygon node id
    );
```

Gets an indexed polynode. Stores the data into the polygon.

```
public: virtual void
    INDEXED_MESH_MANAGER::announce_indexed_polynode (
    ENTITY*,                // coedge along node to
                            // next node
    int,                    // polygon index
    int,                    // local node index
    void*,                  // node identifier
    const double&,          // if the node lies on an
                            // edge then returns tpar
    const SPApar_pos&,      // parametric coordinates
    const SPAposition&,     // Cartesian coordinates
    const SPAunit_vector&   // surface normal
    );
```

Gets an indexed polynode. Stores the data into the polygon.

```
public: virtual void
    INDEXED_MESH_MANAGER::begin_mesh_output (
    ENTITY* entity,         // faceted entity
    ENTITY* app_ref,        // applicable refinement
    ENTITY* format          // output format entity
    );
```

Announces the beginning of the output of a mesh. Cleans up old mesh if necessary.

```
public: virtual void
    INDEXED_MESH_MANAGER::end_mesh_output (
    ENTITY* entity,         // faceted entity
    ENTITY* app_ref,        // applicable refinement
    ENTITY* format          // output format entity
    );
```

The mesh is complete. Updates polygon counts and fixes up parameters.

---

```
public: INDEXED_MESH*
    INDEXED_MESH_MANAGER::mesh () const;
```

Returns the current mesh be constructed.

---

```
public: virtual logical
    INDEXED_MESH_MANAGER::need_approx_counts();
```

Provides an approximate count of polygons and nodes and number of references to nodes.

---

```
public: virtual logical INDEXED_MESH_MANAGER::
    need_duplicate_indexed_nodes_on_surface_seams ();
```

This method is called if the user wants duplicate nodes where the surface uv's differ at the same position on the surface.

---

```
public: virtual logical
    INDEXED_MESH_MANAGER::need_indexed_polygons ();
```

If this function returns TRUE, the indexed protocol is turned on. The default returns FALSE. Used with the indexed protocol to flag requested data.

---

```
public: int
    INDEXED_MESH_MANAGER::NumPolygon () const;
```

Returns the total number of polygons in all meshes.

---

```
public: virtual MESH_MANAGER_SEARCH_ORDER
    INDEXED_MESH_MANAGER::query_search_order ();
```

Returns the order of how the internal mesh of the faceter should be searched to provide output. The order specifies how successive polygons ought to share edges and can help the application to construct the desired data format. The default is to search any polygon.

```
public: virtual void
    INDEXED_MESH_MANAGER::save_mesh_output (
    ENTITY* entity,          // entity being faceted
    ENTITY* app_ref,         // current REFINEMENT
    ENTITY* format           // current output format
                             // (VERTEX_TEMPLATE)
    );
```

This function is called to announce the end of mesh output for a face.

```
protected: void
    INDEXED_MESH_MANAGER::sequence_error (
    char* s                  // error message
    );
```

Prints an error due to improper array indexing.

```
public: void INDEXED_MESH_MANAGER::SetTransform (
    const SPAtransf*         // transformation
    );
```

Sets the transformation to be applied to nodes.

```
public: virtual void
    INDEXED_MESH_MANAGER::start_indexed_polygon (
    int ipoly,               // polygon index
    int npolynode,           // number of polygon node
    int ishare               // info about which edge
        = -2                 // of previous polygon is
                             // is shared with this
                             // one
    );
```

A polygon starts. Adds it to the mesh.

Related Fncs:

None

# indexed_polygon

Class: Faceting

Purpose: Obsolete: used only in pre-1.7 Faceting.

Derivation: indexed_polygon : mesh_polygon : ACIS_OBJECT : –

SAT Identifier: None

Filename: fct/faceter/meshmgr/idx_mesh.hxx

Description: The indexed_polygon class is a derived class of mesh_polygon that stores the polygons in an INDEXED_MESH.

Limitations: None

References:

| | |
|---|---|
| FCT | VERTEX_TEMPLATE, polygon_vertex |
| by FCT | INDEXED_MESH |

Data:

```
protected VERTEX_TEMPLATE* vertex_template;
```
Vertex template defining data stored at each node of the polygon.

```
protected int m_nNumVertex;
```
Number of vertices.

```
protected int ishare;
```
Which edge from previous is shared with this polygon's first edge for proper OpenGL tristrip and other order.

```
protected polygon_vertex** m_pVertexPtrs;
```
Pointer into an array which contains pointers to the vertices for this polygon. The vertices themselves are stored in a separate array.

Constructor:

```
public: indexed_polygon::indexed_polygon ();
```

C++ constructor, creating an indexed_polygon.

These are allocated all at once in an array. There is a separate procedure to initialize the data later on.

Destructor:

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

C++ destructor, deleting an indexed_polygon.

Methods:

```
public: int indexed_polygon::get_share_info () const;
```

Gets the share information from the indexed polygon.

```
public: virtual polygon_vertex*
    indexed_polygon::get_vertex (
    int i                    // integer
    ) const;
```

Get a vertex based on its index (zero based).

```
public: VERTEX_TEMPLATE*
    indexed_polygon::get_vertex_template () const;
```

Get the vertex template for the polygon.

```
public: polygon_vertex**
    indexed_polygon::get_vertices () const;
```

Get vertices.

```
public: virtual int indexed_polygon::num_vertex ()
    const;
```

Get the number of vertices of the polygon.

```
public: virtual void
    indexed_polygon::reverse_vertices ();
```

Reverse the vertices.

```
public: virtual void indexed_polygon::set_data (
    int num_vertex,          // number vertex
    polygon_vertex** verts,  // verts
    int ishare_in            // info about which edge
        = -2                 // of previous polygon is
                             // is shared with this
                             // one
    );
```

Set the data.

```
public: void indexed_polygon::set_share_info (
    int i                    // index number
    );
```

Specifies the shared information.

```
public: int indexed_polygon::set_vertex (
    int vertex_num,          // vertex number
    polygon_vertex* vertex   // vertex
    );
```

Set a vertex pointer.

```
public: void indexed_polygon::set_vertex_template (
    VERTEX_TEMPLATE*         // pointer to class
        _vertex_template     // template to point to
    );
```

Get the vertex template for the polygon.

Related Fncs:

None

# MESH

Class: Faceting

Purpose: Defines a generic mesh class that can be attached to an entity.

Derivation: MESH : ACIS_OBJECT : –

SAT Identifier: None

Filename: fct/faceter/attribs/mesh.hxx

Description: The mesh manager in the faceter leaves it up to the application to determine where it will store the meshes. One possibility is to attach a mesh to its owning face. This class is intended to be a utility for this purpose. If an application derives its meshes from this class, it can also make use of the utility functions provided to work with the meshes.

Limitations: None

References: by FCT ATTRIB_EYE_ATTACHED_MESH

Data:

None

Constructor:

```
protected: MESH::MESH ();
```

C++ constructor, creating a MESH.

Destructor:

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

C++ destructor, deleting a MESH.

Methods:

```
public: virtual void MESH::debug (
    FILE* fp                // debug file stream
    );
```

Write debug information to a file (stub).

```
public: virtual SPAbox MESH::get_box ();
```

Return the bounding box (stub).

```
public: virtual logical MESH::transform (
    SPAtransf const&        // transformation
    );
```

Transform a MESH (stub).

Related Fncs:

None

# MESH_MANAGER

Class: Faceting

Purpose: A MESH_MANAGER class directs output of the facet data.

Derivation: MESH_MANAGER : ACIS_OBJECT : –

SAT Identifier: None

Filename: fct/faceter/meshmgr/meshmg.hxx

Description: This base class defines numerous virtual member function stubs that are grouped into protocols. Derived classes can redefine these functions to switch on the desired protocol. When the faceter has computed a set of facets, it will output them according to the protocol specified. Three protocols are provided, and the base class defines query functions so that the faceter can ask whether each protocol is required.

The *coordinate protocol* outputs the polygon vertices as explicit coordinates, without any regard for shared nodes. The *indexed protocol* first outputs the list of coordinates of all the vertices in a face mesh, then outputs the polygon vertices as indices referring to items of this list. The *global indexed protocol* is similar to the indexed protocol, but considers the entire body as a single mesh. Both the indexed protocol and the global indexed protocol maintain only one copy of each shared node, with each polygon consisting of pointers to the shared nodes.

The general flow of output is first the counts, then the nodes, then the edges, and then the polygons (for each mesh). The polygons are output node by node. The faceter's calls to the node and polynode announcements are strictly ordered — node 0 is always announced before node 1, polygon 0 before polygon 1, and so forth. The initially announced counts may exceed the actual number of nodes, polygons and polynodes to follow.

The "announcement" methods contain some redundant data. For example, the polygon counter ipoly that is passed with each polygon can also be deduced by keeping count of the number of polygons constructed to date. The design intent was to pass redundant data on to every member function that might be interested, on the assumption that it is easier to ignore extra parameters than to maintain state information.

The following terminology is used in MESH_MANAGER classes:

| | |
|---|---|
| *node* | an (*x*,*y*,*z*) position in the mesh |
| *polygon* | an ordered loop of nodes |
| *coordinate polygon* | defined by a series of direct coordinates |
| *indexed polygon* | defined by references to its nodes |
| *globally indexed polygon* | defined by references to its nodes, using global id pointers |
| *polynode* | the use of a node by a polygon |
| *polyedge* | the polygon edge immediately following a polynode, in counterclockwise order |
| *polyedge mate* | the polyedge on the opposite side of a polyedge (within the same polygon), oriented in the opposite direction |

For pre-1.7 compatibility, the faceter uses a MESH_MANAGER that attaches the facets to the face from which they were derived as POLYGON_POINT_MESH. Applications that do not use POLYGON_POINT_MESH directly should probably derive a new class from MESH_MANAGER to handle the facet data. Before faceting any entities, a pointer to an instance of the derived class must be passed to the faceter through an API. Then as the faceter calls the various virtual functions, the redefined functions of the derived class will be invoked to handle the data.

Because the mesh manager is allowed to change the refinement during faceting, it can control how each face is faceted. This can prove very useful, for example, if the application needs to tweak the refinement based on its current run-time context.

Limitations: None

References: None

Data:

None

Constructor:

```
public: MESH_MANAGER::MESH_MANAGER ();
```

C++ constructor, creating a MESH_MANAGER.

Destructor:

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

C++ destructor, deleting a MESH_MANAGER.

Methods:

```
public: virtual void
    MESH_MANAGER::announce_coordinate_polygon_node  (
    int ipoly,              // polygon index
    int i,                  // node position
    const SPApar_pos& uv,   // surface parameter
    const SPAposition& X,   // point
    const SPAunit_vector& N // normal
    );
```

Announces one node on a polygon. Used with the coordinate protocol.

```
public: virtual void MESH_MANAGER::announce_counts (
    int npoly,              // number of polygons
    int nnode,              // number of nodes
    int npolynode           // # node references
    );
```

Announces the number of polygons, nodes, and node references by polygons. Used with the coordinate, indexed and global indexed protocols.

```
public: virtual void
    MESH_MANAGER::announce_edge_indices (
    ENTITY* coedge,          // coedge of current face
    void* idV0,              // starting vertex node
    void* idE0,              // first interior node
    void* idEN,              // last interior node
    void* idV1               // ending vertex node
    );
```

Announces an edge with its coedge and node indices. The starting and ending vertex nodes define the end points of the edge. The first and last interior nodes define a sequential range of nodes in between. Used with the indexed protocol.

```
public: virtual void*
    MESH_MANAGER::announce_global_node (
    int inode,               // node index
    EDGE*,                   // edge
    const SPAposition& X,    // node coordinates
    double t                 // curve parameter
    );
```

Announces a node on a model EDGE with its curve parameter. Used with the global indexed protocol.

```
public: virtual void*
    MESH_MANAGER::announce_global_node (
    int inode,               // node index
    FACE*,                   // face
    const SPAposition& X,    // node coordinates
    const SPApar_pos& uv     // surface parameters
    );
```

Announces a node interior to a model FACE with its surface parameters. Used with the global indexed protocol.

```
public: virtual void*
    MESH_MANAGER::announce_global_node (
    int inode,               // node index
    VERTEX*,                 // vertex
    const SPAposition& X     // node coordinates
    );
```

Announces a node on a model VERTEX. Used with the global indexed protocol.

```
public: virtual void*
    MESH_MANAGER::announce_indexed_node (
    int inode,              // 0-based node index
    const SPApar_pos& uv,   // parametric coordinates
    const SPAposition& X,   // Cartesian coordinates
    const SPAunit_vector& N // surface normal
    );
```

Announces an indexed node with its surface parameters, position, and normal. Used with the indexed protocol.

```
public: virtual void*
    MESH_MANAGER::announce_indexed_polyedge (
    int ipoly,              // polygon index
    int i                   // node position
    );
```

Announces an indexed polyedge whose “mate” has not previously been visited. Used with the indexed protocol.

```
public: virtual void*
    MESH_MANAGER::announce_indexed_polyedge (
    int ipoly,              // polygon index
    int i,                  // node position
    ENTITY* ce              // the coedge this edge
                            // parallels
    );
```

Announces an indexed polyedge, including polyedge information. This version is called if need_indexed_polyedges returns TRUE. Used with the indexed protocol.

```
public: virtual void*
    MESH_MANAGER::announce_indexed_polyedge (
    int ipoly,              // polygon index
    int i,                  // node position
    void* mate              // mate id
    );
```

Announces an indexed polyedge, including polyedge mate information. This version is called if need_indexed_mate_ids returns TRUE. In the indexed mesh protocol, the application mesh manager may optionally receive additional information about the “mate” just after each polynode is sent. The mate id is the one returned by the prior announce_indexed_polyedge call. Used with the indexed protocol.

```
public: virtual void
    MESH_MANAGER::announce_indexed_polynode (
    ENTITY* E,              // coedge along the edge
                            // following the polynode
    int ipoly,              // polygon index
    int i,                  // local node index
    void* id                // node identifier
    );
```

Announces an indexed node on a polygon, including coedge pointers. This version is called if need_coedge_pointers_on_polyedges returns TRUE. The node identifier is the one previously received from announce_indexed_node. Used with the indexed protocol.

```
public: virtual void
    MESH_MANAGER::announce_indexed_polynode (
    ENTITY* E,              // coedge along the edge
                            // following the polynode
    int ipoly,              // polygon index
    int i,                  // local node index
    void* id,               // node identifier
    const double& edge_tpar,// if the node lies on an
                            // edge then returns tpar
    const SPApar_pos& uv,   // parametric coordinates
    const SPAposition& X,   // Cartesian coordinates
    const SPAunit_vector& N // surface normal
    );
```

Announces an indexed node on a polygon, including polynode coordinate data. This extended argument version is called if need_indexed_polynode_with_data returns TRUE. The node identifier is the one previously received from announce_indexed_node. Used with the indexed protocol.

```
public: virtual void
    MESH_MANAGER::announce_indexed_polynode (
    int ipoly,              // polygon index
    int i,                  // node position
    void* id                // node identifier
    );
```

Announces an indexed node on a polygon. This short version is called if both need_coedge_pointers_on_polyedges and need_indexed_polynode_with_data return FALSE, which is the default. The node identifier is the one previously received from announce_indexed_node. Used with the indexed protocol.

```
public: virtual void
    MESH_MANAGER::announce_polygon_model_face (
    ENTITY*                 // Face pointer for the
                            // group of oncoming
                            // polygons
    );
```

Announce the face pointer for oncoming polygons.

```
public: virtual void
    MESH_MANAGER::begin_global_mesh_output (
    ENTITY*                 // top level entity
    );
```

Announces the beginning of global mesh output. The top level entity is the top level entity of the entire mesh, typically a BODY, but can be a LUMP, a SHELL, or a FACE. Used with the global indexed protocol.

```
public: virtual void
    MESH_MANAGER::begin_mesh_output(
    ENTITY* faceted_entity,         // faceted entity
    ENTITY*                         // applicable
        refinement_control_entity,  // rfmnt control
    ENTITY*                         // entity
        output_format_entity        // output format
                                    // entity
    );
```

Announces the beginning of the output of a mesh on a single face on a polygon. Used with the coordinate and indexed protocols.

```
public: virtual void
    MESH_MANAGER::check_applicable_format_entity (
    ENTITY* entity,          // entity with format
    ENTITY*& format          // output format entity
    );
```

While faceting a body, lump, or shell, check_applicable_format_entity is called at each member of the hierarchy, in top down order. The application can assign a different format to it. The output format entity can be a VERTEX_TEMPLATE for pre-1.7 compatibility.

```
public: virtual void
    MESH_MANAGER::check_applicable_refinement (
    ENTITY* entity,          // entity with refinement
    AF_SURF_MODE mode,       // rfmnt surface type
    REFINEMENT*& R           // refinement
    );
```

While faceting a body, lump, or shell, check_applicable_refinement is called at each member of the hierarchy, in top down order, and for each possible surface type at each non-face member. The refinement passed in is what the faceter found for the given surface type. The application can assign a different refinement to it.

```
public: virtual AF_EDGE_DIRECTIVE
    MESH_MANAGER::check_edge_refinement (
    EDGE* E,                 // edge to check
    double& dmax,            // chordal deviation
    double& hmax,            // maximum chordal length
    double& dNmax,           // maximum angle change
                             // between chords
    int nuse,                // number of incident
                             // faces being faceted.
    int nf                   // total number of
                             // incident faces.
    );
```

The parameters that control the faceting of an edge are provided. They have been determined to be the finest in the discretization controls of the incident faces. The default behavior of this function is not to change any value, and to request the faceting to be done if the number of incidents faces equals the total number of incident faces. If first number is smaller, it requests any existing faceting to be used.

```
public: virtual void
    MESH_MANAGER::end_coordinate_polygon (
    int ipoly               // polygon index
    );
```

Announces the completion of the output of a polygon. Used with the coordinate protocol.

```
public: virtual void
    MESH_MANAGER::end_global_mesh_output (
    ENTITY*                 // Top level entity
                            // being faceted
    );
```

Terminates global mesh output. Used with the global indexed protocol.

```
public: virtual void
    MESH_MANAGER::end_indexed_polygon (
    int ipoly               // polygon index
    );
```

Announces the node ids around a polygon. Used with the indexed protocol.

```
public: virtual void MESH_MANAGER::end_mesh_output (
    ENTITY* faceted_entity,         // faceted entity
    ENTITY*                         // applicable
        refinement_control_entity,  // rfmnt control
    ENTITY*                         // entity
        output_format_entity        // output format
                                    // entity
    );
```

Announces the completion of the output of a mesh. The arguments on begin_mesh_output, end_mesh_output, and save_mesh_output are identical. This is done because different mesh managers may need the pointers at different times. This simplifies mesh manager coding because it is not necessary to make local copies of the refinement and output format. Used with the coordinate and indexed protocols.

```
public: virtual MESH_APP_ID MESH_MANAGER::
    get_app_id();
```

Returns the application identification number of the mesh.

```
public: virtual MESH_USER_ID MESH_MANAGER::
    get_user_id();
```

Returns the user identification number of the mesh.

```
public: virtual logical MESH_MANAGER::
    need_approx_counts();
```

Returns true if you are only interested in getting approximate counts of polygons, nodes, and edges. This can save time but may use more memory.

```
public: virtual logical MESH_MANAGER::
    need_coedge_pointers_on_polyedges();
```

If this function returns TRUE, each polygon node that lies on a model edge will be output together with the corresponding coedge. The default returns FALSE. Used with the indexed protocol to flag requested data.

```
public: virtual logical
    MESH_MANAGER::need_coordinate_polygons ();
```

If this function returns TRUE, the data will be announced in the coordinate protocol. The default returns FALSE. Used with the coordinate protocol to flag requested data.

```
public: virtual logical MESH_MANAGER::need_counts ();
```

If this function returns TRUE, the number of polygons, number of nodes, and number of polygon references of node are calculated and announced. If these are not needed, it should return FALSE. The default returns TRUE. Used with the coordinate and indexed protocols to flag requested data.

```
public: virtual logical
MESH_MANAGER::need_degenerate_triangles ();
```

Some applications need degenerate triangles to be topologically correct. Some applications do not need degenerate triangles and want as few triangles as possible. Default is FALSE, returning as few triangles as possible.

```
public: virtual logical
MESH_MANAGER::need_duplicate_indexed_nodes_at_singula
rities ();
```

When facets meet at a singularity, such as at the apex of a cone, each facet may have different normals. If this function returns TRUE, then duplicate nodes will be placed at singularities. If this function returns FALSE, then only one node will be placed at the singularity. Default is TRUE.

```
public: virtual logical MESH_MANAGER::
    need_duplicate_indexed_nodes_on_surface_seams ();
```

This method is called if the user wants duplicate nodes where the surface *uv*'s differ at the same position on the surface.

```
public: virtual logical
    MESH_MANAGER::need_edge_grading (
    double& mu              // relative size of
                            // faceted edge lengths
    );
```

If this function returns TRUE, there is a second pass through the model to further subdivide edges so that consecutive faceted edge lengths have controlled size relationships. The default value for mu is 2.0, which may be changed by the application or further modified for each individual face when the face is passed to need_edge_grading_on_face.

```
public: virtual logical
    MESH_MANAGER::need_edge_grading_on_face (
    FACE* F,                // face on which grading
                            // is to be done
    REFINEMENT* R,          // refinement on the face
    double& mu              // aspect ratio returned
                            // by need_edge_grading
    );
```

If this function returns TRUE, edge grading is applied one face at a time during the edge grading pass. On each pass, the faceter queries the mesh manager to determine the acceptable size ratio between the end chords of edges that share a common vertex on that face. Return FALSE to suppress grading on that face. The default implementation returns TRUE if and only if the face refinement has a grading, and it resets the aspect ratio mu to the grid_aspect_ratio setting of the refinement. If mu returned earlier by need_edge_grading is less than 1.5, it is reset to 1.5.

---

```
public: virtual logical
    MESH_MANAGER::need_edge_indices ();
```

If this function returns TRUE, then the model edges will be announced with the node indices. The default returns FALSE. Used with the indexed protocol to flag requested data.

---

```
public: virtual logical
    MESH_MANAGER::need_global_indexed_polygons ();
```

If this function returns TRUE, the facets will be output with the global indexed protocol. The default returns FALSE. Used with the global indexed protocol to flag requested data.

---

```
public: virtual logical
    MESH_MANAGER::need_indexed_polyedges ();
```

If this function returns TRUE, then the model edges will be announced with the node polyedges.

---

```
public: virtual logical
    MESH_MANAGER::need_indexed_polygons ();
```

If this function returns TRUE, the indexed protocol is turned on. The default returns FALSE. Used with the indexed protocol to flag requested data.

---

```
public: virtual logical MESH_MANAGER::
    need_indexed_polynode_with_data ();
```

If this function returns TRUE, each polygon node will be output together with the corresponding coedge, coordinates and surface normal. The default returns FALSE. Used with the indexed protocol to flag requested data.

```
public: virtual logical MESH_MANAGER::
    need_precount_of_global_indexed_polygons ();
```

If this function returns TRUE, the number of polygons, nodes, and node references by polygons are announced before other output. The default returns FALSE. Used with the global indexed protocol to flag requested data.

```
public: virtual void* MESH_MANAGER::null_node_id ();
```

Returns the node id value that is guaranteed to be invalid. Applications using indices will typically return –1. Applications using pointers will typically return 0. The default returns –1.

```
public: virtual MESH_MANAGER_SEARCH_ORDER
    MESH_MANAGER::query_search_order ();
```

Returns the order of how the internal mesh of the faceter should be searched to provide output. The order specifies how successive polygons ought to share edges and can help the application to construct the desired data format. The default is to search any polygon.

```
public: virtual void MESH_MANAGER::reannounce_counts(
    int npoly,              // Number of polygons
                            // to follow
    int nnode,              // Number of nodes
                            // to follow
    int npolynode           // Number of nodes when
                            // counted each time
                            // used by a polygon
    );;
```

This is called to reannounce the exact number of nodes and polygons that were already announced. This is useful if user used need_approx_counts() and made it return true.

```
public: virtual void
    MESH_MANAGER::save_global_mesh_output (
    ENTITY*                 // top level entity being
                            // faceted
    );
```

Signals the save of a global mesh output.

```
public: virtual void MESH_MANAGER::save_mesh_output (
    ENTITY* faceted_entity,          // entity being
                                     // faceted
    ENTITY*                          // applicable
        refinement_control_entity,   // REFINEMENT
    ENTITY* output_format_entity     // output format
                                     // entity
    );
```

Saves the mesh output.

```
public: virtual void
    MESH_MANAGER::set_app_id(
    MESH_APP_ID appid        // id to use
    );
```

Specifies the application identification number of the mesh.

```
public: virtual void
    MESH_MANAGER::set_user_id(
    MESH_USER_ID userid      // id to use
    );
```

Specifies the user identification number of the mesh.

```
public: virtual void
    MESH_MANAGER::start_coordinate_polygon (
    int ipoly,               // polygon index
    int nnode                // node count
    );
```

Announces the beginning of a new polygon output. Used with the coordinate protocol.

```
public: virtual void
    MESH_MANAGER::start_indexed_polygon (
    int ipoly,              // polygon index
    int npolynode,          // number of nodes
    int ishare              // info about which edge
        = –2                // of previous polygon is
                            // is shared with this
                            // one
    );
```

Announces the beginning of a new polygon output. Used with the indexed protocol.

```
public: virtual void
    MESH_MANAGER::start_shareable_coordinate_polygon(
    int ipoly,              // polygon index
    int nnode,              // number of nodes
    int                     // shared index
    );
```

Starts the shareable coordinate polygon.

```
public: virtual void
    MESH_MANAGER::start_shareable_indexed_polygon (
    int ipoly,              // polygon index
    int npolynode,          // number of nodes
    int ishare              // index of shared edge
    );
```

Starts the shareable indexed polygon.

Internal Use: abort_mesh_output

Related Fncs: None

# mesh_polygon

Class: Faceting

Purpose: Defines an interface to a polygon in a mesh.

Derivation: mesh_polygon : ACIS_OBJECT : –

SAT Identifier: None

Filename: fct/faceter/meshmgr/idx_mesh.hxx

Description: Defines an interface to a polygon in a mesh.

Limitations: None

References: None

Data:

None

Constructor:

```
public: mesh_polygon::mesh_polygon ();
```

C++ constructor, creating a mesh_polygon.

Destructor:

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

C++ destructor, deleting a mesh_polygon.

Methods:

```
public: virtual polygon_vertex*
    mesh_polygon::get_vertex (
    int i                    // index value
    ) const = 0;
```

Get a vertex based on its index (zero based).

```
public: virtual int
    mesh_polygon::num_vertex () const = 0;
```

Get the number of vertices of the polygon.

```
public: virtual void
    mesh_polygon::reverse_vertices() = 0;
```

Reverse the order of the vertices (this does not reverse the normal directions of the vertices).

Related Fncs:

None

# POLYGON

Class: Faceting

Purpose: Defines a POLYGON.

Derivation: POLYGON : ACIS_OBJECT : –

SAT Identifier: None

Filename: fct/faceter/meshmgr/ppm.hxx

Description: Defines a POLYGON.

Limitations: None

References:

| | |
|---|---|
| FCT | POLYGON_VERTEX, VERTEX_TEMPLATE |
| by FCT | POLYGON_POINT_MESH, POLYGON_POINT_MESH_MANAGER, POLYGON_VERTEX |

Data:

None

Constructor:

```
public: POLYGON::POLYGON (
    VERTEX_TEMPLATE*         // vertex template
        _node_template       // node template
        = 0
    );
```

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

Destructor:

```
public: POLYGON::~POLYGON ();
```

C++ destructor, deleting a POLYGON.

Methods:

```
public: void POLYGON::append (
    POLYGON_VERTEX* pv       // vertex list
    );
```

Append the POLYGON.

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

Copy the POLYGON.

```
public: int POLYGON::count ();
```

Return the count.

```
public: POLYGON_VERTEX* POLYGON::first ();
```

Return the first POLYGON.

```
public: void POLYGON::insert (
    POLYGON_VERTEX* poly,    // poly
    POLYGON_VERTEX* key,     // key
    logical check            // check
        = FALSE
    );
```

Insert the POLYGON.

```
public: POLYGON_VERTEX* POLYGON::last ();
```

Return the last POLYGON.

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

Return the next POLYGON.

```
public: void POLYGON::prepend (
    POLYGON_VERTEX* pv       // vertex list
    );
```

Prepend the POLYGON.

```
public: void POLYGON::print (
    FILE*                    // file
    );
```

Print the POLYGON.

```
public: POLYGON_VERTEX* POLYGON::remove ();
```

Remove the POLYGON.

```
public: logical POLYGON::search (
    POLYGON_VERTEX* key      // key
    );
```

Perform a search.

```
public: void POLYGON::set_vertex_template (
    VERTEX_TEMPLATE*         // vertex template
        _node_template       // node template
    );
```

Set the vertex template.

```
public: unsigned POLYGON::size ();
```

Returns the size of the polygon.

```
public: void POLYGON::transform (
    SPAtransf const& T       // transformation
    );
```

Transform the POLYGON.

```
public: VERTEX_TEMPLATE* POLYGON::vertex_template ();
```

Return the vertex template.

Related Fncs:

None

# POLYGON_POINT_MESH

Class: Faceting

Purpose: Defines a collection of polygons.

Derivation: POLYGON_POINT_MESH : MESH : ACIS_OBJECT : –

SAT Identifier: None

Filename: fct/faceter/meshmgr/ppm.hxx

Description: This class defines a mesh for polygons. The polygons in the mesh are kept in a singly linked, NULL-terminated list. The mesh maintains pointers to the first and last polygons and also a count of the number of polygons in the mesh.

Limitations: None

References:

| | |
|---|---|
| FCT | POLYGON |
| by FCT | POLYGON, POLYGON_POINT_MESH_MANAGER, POLYGON_VERTEX |

Data:

None

Constructor:

```
public: POLYGON_POINT_MESH::POLYGON_POINT_MESH ();
```

C++ constructor, creating a POLYGON_POINT_MESH.

Destructor:

```
public: POLYGON_POINT_MESH::~POLYGON_POINT_MESH ();
```

C++ destructor, deleting a POLYGON_POINT_MESH.

Methods:

```
public: void POLYGON_POINT_MESH::append (
    POLYGON* poly           // polygon pointer
    );
```

Appends a polygon.

```
public: void POLYGON_POINT_MESH::apply (
    void(*F)(POLYGON*,      // function to apply
    void*,                  // pointer to argument
    int),                   // flags
    void*arg,               // pointer to argument
    int flags               // flags
    );
```

Applies a function to all the polygons. The function is called with each polygon, the argument, and flags.

```
public: void POLYGON_POINT_MESH::concatenate (
    POLYGON_POINT_MESH*&,   // the one to concatenate
    int                     // destroy input
    );
```

Appends the polygons of the given POLYGON_POINT_MESH. If destroy is TRUE, the polygons are transferred, otherwise they are copied.

```
public: int POLYGON_POINT_MESH::count ();
```

Returns the number of polygons.

---

```
public: POLYGON* POLYGON_POINT_MESH::first ();
```

Returns a pointer to the first polygon.

---

```
public: SPAbox POLYGON_POINT_MESH::get_box ();
```

Gets the POLYGON_POINT_MESH bounding box.

---

```
public: void POLYGON_POINT_MESH::insert (
    POLYGON* poly,          // polygon to insert
    POLYGON* key,           // existing polygon
    logical check           // check existing polygon
        = FALSE
    );
```

Inserts a polygon before an existing one. If checking is requested and the *key* is not found, the polygon will not be inserted. If no checking is requested, the polygon will always be inserted.

---

```
public: POLYGON* POLYGON_POINT_MESH::last ();
```

Returns a pointer to the last polygon.

---

```
public: void POLYGON_POINT_MESH::prepend (
    POLYGON* poly           // polygon
    );
```

Adds a polygon as the first polygon in the mesh.

---

```
public: void POLYGON_POINT_MESH::print (
    FILE*                   // output file stream
    );
```

Prints a POLYGON_POINT_MESH.

---

```
public: POLYGON* POLYGON_POINT_MESH::remove ();
```

Removes all the polygons.

```
public: logical POLYGON_POINT_MESH::search (
    POLYGON* key            // polygon pointer
    );
```

Returns TRUE if the polygon given is in this mesh.

```
public: unsigned POLYGON_POINT_MESH::size ();
```

Returns the size of the polygon point mesh.

```
public: virtual logical
    POLYGON_POINT_MESH::transform (
    SPAtransf const& T      // transformation
    );
```

Transforms a POLYGON_POINT_MESH.

Related Fncs:

None

# POLYGON_POINT_MESH_MANAGER

Class: Faceting

Purpose: Specializes the MESH_MANAGER class when a mesh is maintained as a POLYGON_POINT_MESH.

Derivation: POLYGON_POINT_MESH_MANAGER : MESH_MANAGER : ACIS_OBJECT : –

SAT Identifier: None

Filename: fct/faceter/meshmgr/ppmeshmg.hxx

Description: This class demonstrates use of the *coordinate polygon* mode of output from the faceter. That is, polygons are accepted as point-by-point, with explicit coordinates (NOT pointers to possibly shared coordinates) to each node within each polygon.

At the end of the *coordinate polygon* protocol, the current_mesh pointer points to the just constructed mesh. It is assumed that this class is used as a base class for another class that takes responsibility for finding a permanent storage location for that mesh (e.g., attaches it to the ACIS model).

Limitations: None

References:

| | |
|---|---|
| FCT | POLYGON, POLYGON_POINT_MESH, POLYGON_VERTEX |
| KERN | ENTITY, FACE |

Data:

```
protected ENTITY *mesh_node_template;
```
The current VERTEX_TEMPLATE.

```
protected ENTITY *refinement;
```
The current REFINEMENT.

```
protected POLYGON *current_poly;
```
The current polygon.

```
protected POLYGON_POINT_MESH *current_mesh;
```
The current mesh.

```
protected POLYGON_VERTEX *current_poly_vertex;
```
The current polygon vertex.

Constructor:

```
public: POLYGON_POINT_MESH_MANAGER::
    POLYGON_POINT_MESH_MANAGER ();
```

C++ constructor, creating a POLYGON_POINT_MESH_MANAGER.

Destructor:

None

Methods:

```
public: virtual void POLYGON_POINT_MESH_MANAGER::
    announce_coordinate_polygon_node (
    int ipoly,              // polygon index
    int ipt,                // local node index
    const SPApar_pos& uv,   // parameter position
    const SPAposition& X,   // point
    const SPAunit_vector& N // normal
    );
```

Announces one node on a polygon. Used with the coordinate protocol.

```
public: virtual void
    POLYGON_POINT_MESH_MANAGER::begin_mesh_output  (
    ENTITY* facetted_entity,        // faceted entity
    ENTITY*                         // applicable
        refinement_control_entity,  // REFINEMENT
    ENTITY* output_format_entity    // output format
                                    // entity
    );
```

Announces the beginning of the output of a mesh. Creates a new mesh for it. The faceted entity should be a FACE and the format a VERTEX_TEMPLATE.

```
public: virtual void
    POLYGON_POINT_MESH_MANAGER::end_mesh_output  (
    ENTITY* facetted_entity,        // faceted entity
    ENTITY*                         // applicable
        refinement_control_entity,  // REFINEMENT
    ENTITY* output_format_entity    // output format
                                    // entity
    );
```

End the mesh output.

```
public: virtual logical POLYGON_POINT_MESH_MANAGER::
    need_coordinate_polygons ();
```

Tells the faceter we need coordinate polygons.

```
public: virtual void POLYGON_POINT_MESH_MANAGER::
    start_coordinate_polygon (
    int ipoly,              // polygon index
    int nnode               // number of nodes
    );
```

A polygon output starts. Creates it and add it to the current mesh.

Related Fncs:

None

# POLYGON_VERTEX

Class: Faceting

Purpose: Defines a vertex on a polygon.

Derivation: POLYGON_VERTEX : ACIS_OBJECT : –

SAT Identifier: None

Filename: fct/faceter/meshmgr/ppm.hxx

Description: This class defines a vertex on a polygon.

***Note*** *This class is not derived from* ENTITY.

Limitations: None

References:

| | |
|---|---|
| FCT | af_node_instance |
| by FCT | POLYGON, POLYGON_POINT_MESH_MANAGER |

Data:

None

Constructor:

```
public: POLYGON_VERTEX::POLYGON_VERTEX (
    POLYGON_VERTEX* pv       // vertex pointer
    );
```

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

```
public: POLYGON_VERTEX::POLYGON_VERTEX (
    VERTEX_TEMPLATE*         // vertex template
        node_template        // node template
    );
```

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

Destructor:

```
public: void POLYGON_VERTEX::destroy (
    VERTEX_TEMPLATE*         // vertex template
        node_template        // node template
    );
```

Destroys itself.

Methods:

```
public: const af_node_instance*
    POLYGON_VERTEX::data ();
```

Returns the data.

```
public: logical POLYGON_VERTEX::get_parameter_data (
    VERTEX_TEMPLATE*,        // vertex_template
    parameter_token,         // parameter token
    double*                  // node data cell type
    );
```

Gets the parameter data of the node. Returns it in the array.

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

Returns a pointer to the next vertex in a polygon.

```
public: logical POLYGON_VERTEX::normal (
    VERTEX_TEMPLATE*,        // vertex template
    SPAunit_vector&          // normal
    );
```

Returns the normal of the vertex. If the return value is FALSE, the method could not determine the normal.

```
public: logical POLYGON_VERTEX::point (
    SPAposition&             // point
    );
```

Returns the position of the vertex. If the return value is FALSE, the method could not determine the position.

```
public: void POLYGON_VERTEX::print (
    VERTEX_TEMPLATE* vt          // vertex template
        = 0,
    FILE* to                     // file to print to
        = stderr
    );
```

Print the polygon vertex.

```
public: void POLYGON_VERTEX::set_normal (
    VERTEX_TEMPLATE*,        // vertex_template
    const SPAunit_vector&    // normal
    );
```

Sets the normal of the vertex.

```
public: void POLYGON_VERTEX::set_parameter_data (
    VERTEX_TEMPLATE*,        // vertex template
    parameter_token,         // parameter token
    double*                  // node data cell type
    );
```

Sets the parameter data as specified in the array.

```
public: void POLYGON_VERTEX::set_point (
    const SPAposition&       // point
    );
```

Sets the position of the vertex.

Related Fncs:

None

# polygon_vertex

Class: Faceting

Purpose: Obsolete: used only in pre-1.7 Faceting.

Derivation: polygon_vertex : ACIS_OBJECT : –

SAT Identifier: None

Filename: fct/faceter/meshmgr/idx_mesh.hxx

Description: The polygon_vertex class is used to store the data at a polygon vertex.

Limitations: None

References:

| | |
|---|---|
| FCT | af_node_instance |
| by FCT | INDEXED_MESH, indexed_polygon |
| BASE | SPApar_pos, SPAposition, SPAunit_vector |

Data:

None

Constructor:

```
public: polygon_vertex::polygon_vertex ();
```

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

```
public: polygon_vertex::polygon_vertex (
    const polygon_vertex&   // polygon vertex
    );
```

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

```
public: polygon_vertex::polygon_vertex (
    const SPAposition&,     // position
    const SPAunit_vector&,  // unit vector
    const SPApar_pos&       // parameter position
    );
```

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

Destructor:

```
public: polygon_vertex::~polygon_vertex ();
```

C++ destructor, deleting a polygon_vertex.

Methods:

```
public: const double*
    polygon_vertex::get_color () const;
```

Return the polygon vertex color.

```
public: const SPAunit_vector&
    polygon_vertex::get_normal () const;
```

Return the normal polygon vertex.

```
public: logical polygon_vertex::get_parameter_data (
    VERTEX_TEMPLATE*,       // vertex_template
    parameter_token,        // parameter token
    double*                 // node data cell type
    ) const;
```

Gets the parameter data of the node. Returns it in the array.

```
public: const SPAposition&
polygon_vertex::get_position
    () const;
```

Return the position.

```
public: const SPApar_pos&
    polygon_vertex::get_uv () const;
```

Returns the *uv* parameter.

```
public: polygon_vertex& polygon_vertex::operator= (
    const polygon_vertex&   // for comparison
    );
```

Compares the current instances of the polygon vertex with the one supplied as the argument.

```
public: void polygon_vertex::reverse ();
```

Reverse the polygon vertex.

```
public: void polygon_vertex::set_color (
    const double* color     // color to set
    );
```

Sets the polygon vertex color.

```
public: void polygon_vertex::set_data (
    const SPAposition&,     // position
    const SPAunit_vector&,  // unit vector
    const SPApar_pos&       // parameter position
    );
```

Set the data for the polygon vertex.

```
public: void polygon_vertex::set_normal (
    const SPAunit_vector& norm  // unit vector
    );
```

Set the normal polygon vertex.

```
public: void polygon_vertex::set_parameter_data (
    VERTEX_TEMPLATE*,        // vertex template
    parameter_token,         // parameter token
    double*                  // node data cell type
    );
```

Sets the parameter data as specified in the array.

```
public: void polygon_vertex::set_position (
    const SPAposition& pos  // position
    );
```

Sets the position.

```
public: void polygon_vertex::set_uv (
    const SPApar_pos& uv    // uv parameter position
    );
```

Set the *uv* position.

```
public: void polygon_vertex::set_uv (
    double u,               // u parameter
    double v                // v parameter
    );
```

Set the *uv* parameter position.

Related Fncs:

None

# PPM_ON_FACE_MESH_MANAGER

Class: Faceting

Purpose: Shows how a MESH-derived mesh is attached to a face.

Derivation: PPM_ON_FACE_MESH_MANAGER : POLYGON_POINT_MESH_MANAGER : MESH_MANAGER : ACIS_OBJECT : –

SAT Identifier: None

Filename: fct/faceter/meshmgr/ppmface.hxx

Description: The functionality to attach the mesh is added to what have been provided by POLYGON_POINT_MESH_MANAGER.

Limitations: None

References: None

Data:

None

Constructor:

```
public: PPM_ON_FACE_MESH_MANAGER::
    PPM_ON_FACE_MESH_MANAGER ();
```

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

Destructor:

None

Methods:

```
public: virtual void PPM_ON_FACE_MESH_MANAGER::
    begin_mesh_output (
    ENTITY* face,                // entity being
                                 // faceted
    ENTITY* refinement_control, // REFINEMENT
    ENTITY* output_format        // output format
                                 // entity
    );
```

Announces the beginning of the output of a mesh. Creates the mesh and attaches it to the owning face.

Related Fncs:

None

# REFINEMENT

Class: Faceting, SAT Save and Restore

Purpose: Controls the accuracy and types of polygons generated in the faceter.

Derivation: REFINEMENT : ENTITY : ACIS_OBJECT : –

SAT Identifier: “eye_refinement”

Filename: fct/faceter/attribs/refine.hxx

Description: Each refinement is specific to a type of surface.

Available surface types are:

| | |
|---|---|
| AF_SURF_ALL | all surface types. This is the default. |
| AF_SURF_REGULAR | a surface with planar cells. |
| AF_SURF_IRREGULAR | a surface with possibly nonplanar cells. |
| AF_SURF_PLANE | a planar surface. |
| AF_SURF_CONE | a conical surface (including cylinder). |
| AF_SURF_SPHERE | a spherical surface. |
| AF_SURF_TORUS | a toroidal surface. |
| AF_SURF_SPLINE | a spline surface. |

These parameters can be used to control the faceting:

| | |
|---|---|
| adjust mode | specifies type of triangle smoothing to do. Determines if facet nodes should be adjusted for smoothing. Also determines if the grid points should be adjusted or not. The default is AF_ADJUST_NONE_GRID. |
| normal tolerance | is the maximum angle between any two normals of a facet. |
| surface tolerance | is the maximum distance between the facet and the nearest point on the surface. |
| maximum edge length | is the maximum size of an edge of a facet. |
| maximum grid lines | is the maximum number of subdivisions on a face. |
| grading | adjusts facets to get better parametric aspect ratio of a grid cell. The default is #f. |
| grid aspect ratio | is the approximate aspect ratio of each cell in grid. |
| grid handling | specifies whether a grid is used and whether the points where the grid cuts the edge is inserted to the edge. |
| triangulation control | specifies how much triangulation to do. |

Limitations: None

References: by FCT ATTRIB_EYE_REF_VT, STL_MESH_MANAGER

Data:

None

Constructor:

```
public: REFINEMENT::REFINEMENT ();
```

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

Creates a REFINEMENT with these settings:

surface type ..... = AF_SURF_ALL
normal tolerance ..... = 15.0
surface tolerance ..... = –1 (use bounding box diagonal)
maximum edge length ..... = 0.0 (ignored)
maximum grid lines ..... = 512
minimum *u* grid lines ..... = 0
minimum *v* grid lines ..... = 0
grid aspect ratio ..... = 0.0 (ignored)
grid mode ..... = AF_GRID_INTERIOR
triangulation control ..... = AF_TRIANG_FRINGE_2
triangle smoothing ..... = AF_ADJUST_NONE

```
public: REFINEMENT::REFINEMENT (
    int min,                // minimum
    int max,                // maximum
    REFINEMENT_IFLOAT ftol, // f tolerance
    REFINEMENT_IFLOAT stol, // s tolerance
    REFINEMENT_IFLOAT sdev, // s deviation
    REFINEMENT_IFLOAT ndev, // n deviation
    REFINEMENT_IFLOAT ptol, // p tolerance
    float ar,               // ar float
    int m,                  // m point
    REFINEMENT_IFLOAT edge_length   // edge length
    );
```

Obsolete: used only in pre-1.7 Faceting. Creates a refinement.

Destructor:

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

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 REFINEMENT(...) then later x–>lose.)

Methods:

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

Adds to the REFINEMENT.

```
public: REFINEMENT* REFINEMENT::copy () const;
```

This creates and returns a new refinement that is a copy of this instance of REFINEMENT.

```
public: void REFINEMENT::copy_to (
    REFINEMENT* to          // where to copy
    ) const;
```

Copies this instance of REFINEMENT to the given to refinement.

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

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

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

Indicates whether this entity is normally destroyed by lose (TRUE), or whether it is shared between multiple owners with a use count. If shared, it gets destroyed implicitly when every owner has been lost (FALSE). The default is FALSE. This is an internal method that should not be used by an application.

```
public: REFINEMENT_IFLOAT REFINEMENT::edge_length (
    ) const;
```

Returns the edge length. This is the maximum allowable length of a facet edge.

```
public: AF_ADJUST_MODE REFINEMENT::
    get_adjust_mode () const;
```

Returns the triangle smoothing mode. Refer to the set_adjust_mode method.

```
public: REFINEMENT_IFLOAT REFINEMENT::
    get_dynamic_surtol () const;
```

Returns the dynamic surface tolerance.

```
public: logical
    REFINEMENT::get_grading_mode () const;
```

Returns the grading.

```
public: REFINEMENT_IFLOAT
    REFINEMENT::get_grid_aspect_ratio () const;
```

Returns the grid aspect ratio. The grid aspect ratio is the approximate ratio of the length in the *u* direction to the length in the *v* direction of each *uv* cell in the grid.

```
public: AF_GRID_MODE
    REFINEMENT::get_grid_mode () const;
```

Returns the grid handling mode.

```
public: REFINEMENT_IFLOAT
    REFINEMENT::get_max_edge_length () const;
```

Returns the maximum allowable size of an edge of a facet.

```
public: int REFINEMENT::get_max_grid_lines () const;
```

Returns the maximum allowable number of grid lines for a refinement.

```
public: int
    REFINEMENT::get_min_u_grid_lines () const;
```

Returns the minimum number of *u* grid lines.

```
public: int
    REFINEMENT::get_min_v_grid_lines () const;
```

Returns the minimum number of *v* grid lines.

```
public: REFINEMENT_IFLOAT
    REFINEMENT::get_normal_tol () const;
```

Returns the normal tolerance, which is the maximum difference between any two normals of facet.

```
public: REFINEMENT_IFLOAT REFINEMENT::
    get_surface_tol () const;
```

Returns the surface tolerance.

```
public: AF_SURF_MODE
    REFINEMENT::get_surf_mode () const;
```

Returns the surface mode.

```
public: AF_TRIANG_MODE REFINEMENT::
    get_triang_mode () const;
```

Returns the triangulation control.

```
public: virtual int REFINEMENT::identity (
    int                         // integer
        = 0
    ) const;
```

Returns the type of this refinement.

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

Returns TRUE if this can be deep copied.

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

Returns whether this is use counted or not.

```
public: void REFINEMENT::merge (
    REFINEMENT*             // second refinement
    );
```

Merges two refinements.

```
public: void REFINEMENT::merge_owner (
    ENTITY* other,          // given entity
    logical delete_other    // deleting owner
    );
```

Notifies the REFINEMENT that its owning ENTITY is about to be merged with given entity. The application has the chance to delete or otherwise modify the attribute. After the merge, this owner will be deleted if the logical deleting owner is TRUE, otherwise it will be retained and other entity will be deleted. The default action is to do nothing. This function is supplied by the application whenever it defines a new attribute, and is called when a merge occurs.

```
public: logical REFINEMENT::need_grid () const;
```

Determines if the grid is required (TRUE) or not (FALSE).

```
public: void REFINEMENT::print (
    FILE*                   // file
        = stderr
    ) const;
```

Prints information about this refinement. MAC, NT platforms only.

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

Remove from REFINEMENT if use count reaches zero.

```
public: void REFINEMENT::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.

```
// stat = 1
for(;stat;)
    read_string                 buffer
    if(SAME_STRING(buffer,id_end_fields))
        // stat = 0
    else if(SAME_STRING(buffer,id_grid_mode))
        read_int                Set grid mode
    else if(SAME_STRING(buffer,id_triang_mode))
        read_int                Set triang mode
    else if(SAME_STRING(buffer,id_surf_mode))
        read_int                Set surf mode
    else if(SAME_STRING(buffer,id_adjust_mode))
        read_int                Set adjust mode
    else if(SAME_STRING(buffer,id_grading_mode))
        read_int                Set grading mode
    else if(SAME_STRING(buffer,id_postcheck_mode))
        read_int                Set postcheck mode
    else if(SAME_STRING(buffer,id_surface_tol))
        read_real               Set surface tolerance
    else if(SAME_STRING(buffer,id_normal_tol))
        read_real               Set normal tolerance
    else if(SAME_STRING(buffer,id_max_edge_length))
        read_real               Set maximum edge length
    else if(SAME_STRING(buffer,id_grid_aspect_ratio))
        read_real               Set grid grid aspect ratio
    else if(SAME_STRING(buffer,id_max_grid_lines))
        read_int                Set maximum grid lines
    else if(SAME_STRING(buffer,id_min_u_grid_lines))
        read_int                Set minimum u grid lines
    else if(SAME_STRING(buffer,id_min_v_grid_lines))
        read_int                Set minimum v grid lines
    else if(SAME_STRING(buffer,id_silhouette_tol))
        read_real               Set silhouette tolerance
    else if(SAME_STRING(buffer,id_flatness_tol))
        read_real               Set flatness tolerance
    else if(SAME_STRING(buffer,id_pixel_area_tol))
        read_real               Set pixel area tolerance
    else
        // stat = 0
#ifdef READ_OLD_FIELDS
    read_int                    minimum level
    read_int                    maximum level
    read_real                   flatness tolerance
```

```
    read_real                   silhouette tolerance
    read_real                   surface tolerance
    read_real                   normal tolerance
    read_real                   pixel area tolerance
    read_real                   maximum aspect ratio
    read_int                    maximum sides mode
    read_real                   maximum edge length
#endif
```

---

```
public: logical REFINEMENT::same (
    REFINEMENT*             // refinement to test
    ) const;
```

Determines if two refinements are the same.

---

```
public: void REFINEMENT::set_adjust_mode (
    AF_ADJUST_MODE          // mode to set
    );
```

The adjustment mode is used for triangle smoothing. It specifies whether triangles should be smoothed. The nongrid mode preserves the planarity of cells by avoiding points that are corners of a cell. The allowable values are:

| | |
|---|---|
| AF_ADJUST_NONE ..... | no smoothing |
| AF_ADJUST_NON_GRID ..... | applies to points and not part of a cell. |
| AF_ADJUST_ALL ..... | adjust all points connected to triangles |

---

```
public: void REFINEMENT::set_dynamic_surtol (
    REFINEMENT_IFLOAT       // value for surface tol
    );
```

Sets the maximum allowable dynamic surface tolerance.

---

```
public: void REFINEMENT::set_edge_length (
    REFINEMENT_IFLOAT tol   // length
    );
```

Sets the maximum allowable length of a facet edge.

```
public: void REFINEMENT::set_grading_mode (
    logical                 // turn on or off
    );
```

Sets the grading mode.

```
public: void REFINEMENT::set_grid_aspect_ratio (
    REFINEMENT_IFLOAT       // ratio to set
    );
```

Grid aspect ratio specifies the approximate aspect ratio of each cell in the grid. If the value is close to 1, then the cell is close to a square. This does not guarantee the aspect ratio of the facet, which may consist of only a part of a cell.

```
public: void REFINEMENT::set_grid_mode (
    AF_GRID_MODE            // mode to set
    );
```

The grid mode specifies whether a grid is used and whether the points where the grid cuts the edges should be inserted into the edge discretization. These are the allowable values:

| | |
|---|---|
| AF_GRID_NONE . . . . . | do not subdivide face with a grid |
| AF_GRID_INTERIOR . . . . . | use a grid but do not add intersection points. |
| AF_GRID_TO_EDGES . . . . . | use a grid and insert intersections points |

```
public: void REFINEMENT::set_max_edge_length (
    REFINEMENT_IFLOAT       // edge length
    );
```

The maximum edge length specifies the maximum length of a side of a cell in object space. Since a facet cannot be larger than the cell, this determines the maximum size of the facet.

```
public: void REFINEMENT::set_max_grid_lines (
    int                     // grid lines
    );
```

The maximum grid lines specifies the maximum number of grid subdivisions. This prevents the facet data of a face from getting too big. It can also be used to specify the exact number of divisions on a face by using it in conjunction with another parameter; e.g., a very small normal deviation.

```
public: void REFINEMENT::set_min_u_grid_lines (
    int                     // number of grid lines
    );
```

Sets the minimum number of *u* grid lines.

```
public: void REFINEMENT::set_min_v_grid_lines (
    int                     // number of grid lines
    );
```

Sets the minimum number of *v* grid lines.

```
public: void REFINEMENT::set_normal_tol (
    REFINEMENT_IFLOAT       // deviation
    );
```

The normal tolerancespecifies the maximum normal deviation allowed between two normals on a facet. The proper value is usually independent of the model size.

```
public: void REFINEMENT::set_surface_tol (
    REFINEMENT_IFLOAT       // refinement value
    );
```

The surface tolerance specifies the maximum distance between a facet and the true surface. The proper value is dependent on the model size.

```
public: void REFINEMENT::set_surf_mode (
    AF_SURF_MODE            // mode to set
    );
```

The surface mode specifies the type of surface to which the refinement is applicable. The allowable types are listed below. If more than one refinement is applicable, the more specific one overrides the less specific one.

| | |
|---|---|
| AF_SURF_ALL | all surface types. |
| AF_SURF_REGULAR | a surface with planar cells. |
| AF_SURF_IRREGULAR | a surface with possibly nonplanar cells. |
| AF_SURF_PLANE | a planar surface. |
| AF_SURF_CONE | a conical surface (including cylinder). |
| AF_SURF_SPHERE | a spherical surface. |
| AF_SURF_TORUS | a toroidal surface. |
| AF_SURF_SPLINE | a spline surface. |

---

```
public: void REFINEMENT::set_triang_mode (
    AF_TRIANG_MODE          // mode to set
    );
```

This sets the triangulation control, which specifies how much triangulation to perform. If AF_GRID_INTERIOR is specified, triangulation will be performed at least at the fringe cells. Allowable values are:

| | |
|---|---|
| AF_TRIANG_NONE | no triangulation. |
| AF_TRIANG_FRINGE_1 | triangulate at the fringe layer. |
| AF_TRIANG_FRINGE_2 | triangulate 2 fringe layers. |
| AF_TRIANG_FRINGE_3 | triangulate 3 fringe layers. |
| AF_TRIANG_FRINGE_4 | triangulate 4 fringe layers. |
| AF_TRIANG_ALL | triangulate all facets. |

---

```
public: virtual void REFINEMENT::set_use_count (
    int val                 // new value
    );
```

Sets the count for the number of instances of this refinement class. This is used by the lose method. Refinements are not destructed until use_count goes to zero.

---

```
public: REFINEMENT* REFINEMENT::share ();
```

Increments the use count if “this” and returns “this.”

---

```
public: void REFINEMENT::split_owner (
    ENTITY* new_entity      // new entity
    );
```

Notifies the REFINEMENT that its owner is about to be split into two parts. The application has the chance to duplicate or otherwise modify the attribute. The default action is to do nothing. This function is supplied by the application whenever it defines a new attribute, and is called when a split occurs.

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

Returns the string “eye_refinement”.

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

Counts the number of instances of this refinement class. This is used by the lose method. Refinements are not destructed until use_count goes to zero.

Internal Use: aspect_ratio, flatness, get_flatness_tol, get_pixel_area_tol, get_postcheck_mode, get_silhouette_tol, maximum_level, minimum_level, mode, normal_deviation, pixel_area, save, save_common, set_aspect_ratio, set_flatness, set_flatness_tol, set_maximum_level, set_minimum_level, set_mode, set_normal_deviation, set_pixel_area, set_pixel_area_tol, set_postcheck_mode, set_silhouette, set_silhouette_tol, set_surface_deviation, silhouette, surface_deviation, view_dependent

Related Fncs: af_adjust_mode_int, af_adjust_mode_str, af_grid_mode_int, af_grid_mode_str, af_surf_mode_int, af_surf_mode_str, af_triang_depth, af_triang_mode_int, af_triang_mode_str, is_REFINEMENT, modify_refinement

# SIMPLE_INDEXED_MESH

Class: Faceting

Purpose: A mesh format that stores indices to a vertex array.

Derivation: SIMPLE_INDEXED_MESH : MESH : ACIS_OBJECT : –

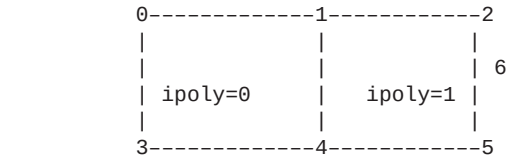
SAT Identifier: None

Filename: fct/faceter/meshmgr/idmeshmg.hxx

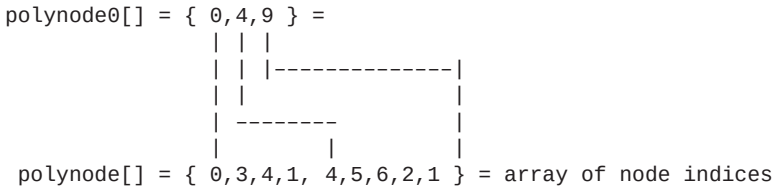
Description: A SIMPLE_INDEXED_MESH is a mesh format that minimizes memory by placing data in contiguous arrays. This should also give fast access to the data by minimizing the number of allocations required, and by minimizing fragmentation. The indexed mesh is stored as an array of nodes and a set of polygons defined by (numerical) indices into the node set. This is a packed representation in which both polygons and nodes are referenced via integers that index into arrays.

This class is an appropriate mesh storage format if reducing memory fragmentation is a priority, all mesh data is available for one-time construction, and no subsequent editing (other than moving vertices) is to be performed.

In this example, *ipoly* is a polygon number, *inode* is a global node number, *i* is a node position when numbered only within a single polygon, and all numbering is 0 based. This shows an indexed mesh of two polygons, ipoly=0 and ipoly=1. For this mesh, npoly = 2, nnode = 7, and npolynode = 9 (4 on ipoly=0, 5 on ipoly=2).




An array leading to the zero'th index per polygon is given by




Error handling conventions for this class are as follows:

a) all ipoly and inode indices are checked and adjusted to 0 if out of bounds.
b) the position and normal of node 0 are pre–initialized to (0,0,0) so that references to them can be returned even if no nodes are actually defined.

polynode0[npoly+1] points to one past the end of the polynode.

Limitations: None

References:

| | |
|---|---|
| by FCT | SIMPLE_INDEXED_MESH_MANAGER |
| BASE | SPAposition, SPAunit_vector |

Data:

None

Constructor:

```
public: SIMPLE_INDEXED_MESH::SIMPLE_INDEXED_MESH (
    int mpoly,              // maximum vertex
    int mnode,              // maximum polygon
    int mpolynode           // maximum polynode
    );
```

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

Destructor:

```
public: SIMPLE_INDEXED_MESH::~SIMPLE_INDEXED_MESH ();
```

C++ destructor, deleting a SIMPLE_INDEXED_MESH.

Methods:

```
public: SPAbox SIMPLE_INDEXED_MESH::get_box ();
```

Returns the bounding box of the mesh.

```
public: SIMPLE_INDEXED_MESH*
    SIMPLE_INDEXED_MESH::get_next ();
```

Returns the next simple indexed mesh.

```
public: int SIMPLE_INDEXED_MESH::get_nnode ();
```

Returns the node count.

```
public: const SPAunit_vector&
    SIMPLE_INDEXED_MESH::get_normal (
    int inode               // index of node
    );
```

Returns the normal *(x,y,z)* stored at node.

```
public: int SIMPLE_INDEXED_MESH::get_npoly ();
```

Returns the polygon count.

```
public: int SIMPLE_INDEXED_MESH::get_npolynode (
    int ipoly               // polygon index
    );
```

Returns the number of nodes on polygon.

```
public: int SIMPLE_INDEXED_MESH::get_polynode (
    int ipoly,              // polygon index
    int i                   // local index of node
    );
```

Returns the global node index of the $i^{th}$ node of a polygon.

```
public: void
    SIMPLE_INDEXED_MESH::get_polynode_array (
    int ipoly,              // polygon index
    const int*& id,         // array of node indices
    int& n                  // number of nodes on
                            // polygon
    );
```

Returns node indices of polygon.

```
public: const SPAposition&
    SIMPLE_INDEXED_MESH::get_position (
    int inode               // index of node
    );
```

Returns the position (*x*,*y*,*z*) of the $i^{th}$ node.

```
public: int
    SIMPLE_INDEXED_MESH::get_sum_polynodes ();
```

Returns the sum of the per-polygon node counts.

Related Fncs:

checklist, replace_in_list

# SIMPLE_INDEXED_MESH_MANAGER

Class: Faceting

Purpose: To show an example of a mesh manager using the indexed protocol.

Derivation: SIMPLE_INDEXED_MESH_MANAGER : MESH_MANAGER : ACIS_OBJECT : –

SAT Identifier: None

Filename: fct/faceter/meshmgr/idmeshmg.hxx

Description: SIMPLE_INDEXED_MESH_MANAGER uses the indexed protocol to output the the facet data. It stores the data into SIMPLE_INDEXED_MESH.

Limitations: None

References:

| | |
|---|---|
| FCT | SIMPLE_INDEXED_MESH |
| by FCT | SIMPLE_INDEXED_MESH |

Data:

```
protected SIMPLE_INDEXED_MESH *mesh;
```

Current mesh being processed.

Constructor:

```
public: SIMPLE_INDEXED_MESH_MANAGER::
    SIMPLE_INDEXED_MESH_MANAGER ();
```

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

Destructor:

None

Methods:

```
public: virtual void SIMPLE_INDEXED_MESH_MANAGER::
    announce_counts (
    int npoly,              // polygon count
    int nnode,              // node count
    int nref                // node reference count
    );
```

Announces the number of polygons, nodes, and node references by polygons. Creates a mesh with the proper storage.

```
public: virtual void* SIMPLE_INDEXED_MESH_MANAGER::
    announce_indexed_node (
    int inode,              // global node number
    const SPApar_pos& uv,   // surface parameter
    const SPAposition& X,   // position
    const SPAunit_vector& N // normal
    );
```

Gets a node. Stores it into the global node array of mesh.

```
public: virtual void SIMPLE_INDEXED_MESH_MANAGER::
    announce_indexed_polynode (
    ENTITY*,                // coedge along the edge
                            // following the polynode
    int,                    // polygon index
    int,                    // local node index
    void*,                  // node identifier
    const double&,          // if the node lies on an
                            // edge then returns tpar
    const SPApar_pos&,      // parametric coordinates
    const SPAposition&,     // Cartesian coordinates
    const SPAunit_vector&   // surface normal
    );
```

Announces an indexed node on a polygon, including coedge information.

```
public: virtual void SIMPLE_INDEXED_MESH_MANAGER::
    announce_indexed_polynode (
    int ipoly,              // polygon index
    int i,                  // node position
    void* pnode             // polygon node id
    );
```

Announces an indexed node on a polygon.

```
public: virtual void SIMPLE_INDEXED_MESH_MANAGER::
    announce_indexed_polynode (
    ENTITY*,                // entity
    int,                    // polygon index
    int,                    // local node index
    void*                   // node identifier
    );
```

Announces an indexed node on a polygon, including coedge and coordinate information.

```
public: virtual void SIMPLE_INDEXED_MESH_MANAGER::
    begin_mesh_output (
    ENTITY* entity,          // faceted entity
    ENTITY* app_ref,         // refinement entity
    ENTITY* format           // format entity
    );
```

Announces the beginning of the output of a mesh. Nothing to do—this function can be omitted.

```
public: virtual void SIMPLE_INDEXED_MESH_MANAGER::
    end_mesh_output (
    ENTITY* entity,          // faceted entity
    ENTITY* app_ref,         // refinement entity
    ENTITY* format           // format entity
    );
```

End mesh output.

```
public: virtual logical SIMPLE_INDEXED_MESH_MANAGER::
    need_indexed_polygons ();
```

If this function returns TRUE, the indexed protocol is turned on. The default returns FALSE. Used with the indexed protocol to flag requested data.

```
protected: void
    SIMPLE_INDEXED_MESH_MANAGER::sequence_error (
    char* s                  // error message
    );
```

Prints an error due to improper array indexing.

```
public: virtual void SIMPLE_INDEXED_MESH_MANAGER::
    start_indexed_polygon (
    int ipoly,               // polygon index
    int npolynode,           // # polygon nodes
    int ishare               // info about which edge
        = -2                 // of previous polygon is
                             // is shared with this
                             // one
    );
```

A polygon starts. Update the polygon data in the mesh. SIMPLE_INDEXED_MESH_MANAGER::end_indexed_polygon is not redefined (omitted) because MESH_MANAGER::end_indexed_polygon is sufficient.

Related Fncs:

checklist, replace_in_list

# STL_MESH_MANAGER

Class: Faceting

Purpose: Writes data to an stl formatted file.

Derivation: STL_MESH_MANAGER : MESH_MANAGER : ACIS_OBJECT : –

SAT Identifier: None

Filename: fct/faceter/meshmgr/stlmmg.hxx

Description: The STL_MESH_MANAGER uses the “global indexed” protocol to write data to an stl formatted file. The application must identify the output file by calling the member function set_output_file.

Limitations: None

References:

| | |
|---|---|
| FCT | REFINEMENT |
| BASE | SPAposition |

Data:

None

Constructor:

```
public: STL_MESH_MANAGER::STL_MESH_MANAGER (
    REFINEMENT* R           // refinement
        = NULL
    );
```

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

Destructor:

```
public: STL_MESH_MANAGER::~STL_MESH_MANAGER ();
```

C++ destructor, deleting a STL_MESH_MANAGER.

Methods:

```
public: virtual void
    STL_MESH_MANAGER::announce_counts (
    int npoly,              // number of polygons to
                            // follow
    int nnode,              // # nodes to follow
    int npolynode           // number of nodes when
                            // counted each time they
                            // are used by a polygon
    );
```

Announces the number of polygons, nodes, and node references by polygons.

```
public: virtual void*
    STL_MESH_MANAGER::announce_global_node (
    int inode,              // integer
    EDGE*,                  // edge
    const SPAposition& X,   // node coordinates
    double t                // curve parameter
    );
```

Announces a node on a model EDGE with its curve parameter.

```
public: virtual void*
    STL_MESH_MANAGER::announce_global_node (
    int inode,              // integer
    FACE*,                  // face
    const SPAposition& X,   // node coordinates
    const SPApar_pos& uv    // surface parameter
    );
```

Announces a node on a model FACE with its surface parameters.

```
public: virtual void*
    STL_MESH_MANAGER::announce_global_node (
    int inode,              // integer
    VERTEX*,                // vertex
    const SPAposition& X    // node coordinates
    );
```

Announces a node on a model VERTEX.

```
public: virtual void
    STL_MESH_MANAGER::announce_indexed_polynode (
    ENTITY*,                // coedge along the edge
                            // following the polynode
    int,                    // polygon index
    int,                    // local node index
    void*,                  // node identifier
    const double&,          // if the node lies on an
                            // edge then returns tpar
    const SPApar_pos&,      // parametric coordinates
    const SPAposition&,     // Cartesian coordinates
    const SPAunit_vector&   // surface normal
    );
```

Return the number of indexes.

```
public: virtual void STL_MESH_MANAGER::
    announce_indexed_polynode (
    int ipoly,              // 0-based polygon index.
                            // Same as immediately
                            // preceding call to
                            // start_indexed_polygon
    int i,                  // 0–based counter within
                            // the polygon. This
                            // increments
                            // sequentially
                            // on successive calls.
    void* pnode             // Node identifier as
                            // previously rcvd from
                            // announce_indexed_node
    );
```

Return the number of indexes.

```
public: virtual void STL_MESH_MANAGER::
    announce_indexed_polynode (
    ENTITY*,                // coedge along the edge
                            // following the polynode
    int,                    // polygon index
    int,                    // local node index
    void*                   // node identifier
    );
```

Return the number of indexes.

```
public: virtual void
    STL_MESH_MANAGER::check_applicable_refinement (
    ENTITY*,                // entity with refinement
    AF_SURF_MODE,           // refmt surface type
    REFINEMENT*&            // refinement
    );
```

Check for the applicable refinement.

```
public: virtual void
    STL_MESH_MANAGER::end_global_mesh_output (
    ENTITY*                 // top level entity
                            // being faceted
    );
```

Terminates global mesh output.

```
public: virtual void
    STL_MESH_MANAGER::end_indexed_polygon (
    int ipoly               // polygon index
    );
```

This matches the immediately preceding call to start_indexed_polygon and the (multiple) calls to announce_indexed_polynode.

```
public: virtual logical STL_MESH_MANAGER::
    need_global_indexed_polygons ();
```

Get the globally indexed polygons.

```
public: virtual logical STL_MESH_MANAGER::
    need_precount_of_global_indexed_polygons ();
```

If this function returns TRUE, the number of polygons, nodes, and node references by polygons are announced before other output. The default returns FALSE.

```
public: virtual void*
    STL_MESH_MANAGER::null_node_id ();
```

Returns the node id value that is guaranteed to be invalid. Applications using indices will typically return –1. Applications using pointers will typically return 0. The default returns –1.

```
public: void STL_MESH_MANAGER::set_output_file (
    FILE* fp                // file
        = stdout
    );
```

Set the output file.

```
public: virtual void
    STL_MESH_MANAGER::start_indexed_polygon (
    int ipoly,              // polygon index
    int npolynode,          // number of polynodes
    int ishare              // info about which edge
        = –2                // of previous polygon is
                            // is shared with this
                            // one
    );
```

Start the index.

Related Fncs:

None

# VERTEX_TEMPLATE

Class: Faceting, SAT Save and Restore

Purpose: Creates parameter templates.

Derivation: VERTEX_TEMPLATE : ENTITY : ACIS_OBJECT : –

SAT Identifier: “vertex_template”

Filename: fct/faceter/attribs/vtplate.hxx

Description: This entity enables applications to request attachment of any of the following data to the nodes of a mesh provided the MESH_MANAGER is capable of handling the request.

– coordinate data
– normal to the surface
– *uv* parameter on the surface
– partial derivatives in *u* and *v* on the surface
– magnitude of the partial derivatives in *u* and *v* on the surface
– RGB color
– transparency
– texture
– pointer to user defined data

Nonglobal indexed mesh managers and coordinate mesh managers create meshes on a face basis and so can be designed to attach this type of data to the nodes. Global indexed mesh managers share node data between faces and so face related information is not stored with this type of mesh manager.

By defining a default VERTEX_TEMPLATE or attaching one to a body, lump, shell, or face, the application program can request that certain data be attached to a nodes of the mesh corresponding to that entity. A particular MESH_MANAGER must honor this request by allocating storage and wherever possible initializing values for this data.

The currently active vertex template defines the data attached to a node of a polygon on a face and is the vertex template on the face, shell, lump, body, or default in that order. The only common data for all nodes is the coordinate. The normal, *uv* parameter, *uv* partial derivatives, and magnitudes of the *uv* partial derivatives can be calculated and stored automatically by the MESH_MANAGER if so requested. Space for the RGB color, transparency, texture, and pointer must be allocated by the MESH_MANAGER but the values must be initialized by the application program.

Limitations: None

References:

| | |
|---|---|
| FCT | af_node_mapping |
| by FCT | ATTRIB_EYE_REF_VT, INDEXED_MESH_MANAGER, POLYGON, indexed_polygon |

Data:

None

Constructor:

```
public: VERTEX_TEMPLATE::VERTEX_TEMPLATE (
    int n_tokens,           // number of tokens
    parameter_token tokens[]// tokens
    );
```

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 VERTEX_TEMPLATE(...)), because this reserves the memory on the heap, a requirement to support roll back and history management.

```
public: VERTEX_TEMPLATE::VERTEX_TEMPLATE (
    void* data              // pointer to data
        = 0
    );
```

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

After this constructor is called, the method allocate_node_mapping must be called to create a new af_node_mapping instance.

Destructor:

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

C++ destructor, deleting a VERTEX_TEMPLATE. Applications should never directly call this destructor, but should instead call the lose method.

Methods:

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

Adds to the VERTEX_TEMPLATE.

```
public: void
    VERTEX_TEMPLATE::allocate_node_mapping ();
```

Allocates an af_node_mapping. This must be called after a new VERTEX_TEMPLATE is constructed.

```
public: VERTEX_TEMPLATE*
    VERTEX_TEMPLATE::copy () const;.
```

Copies a VERTEX_TEMPLATE.

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

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

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

Returns indication of whether this vertex template is deletable.

```
public: int VERTEX_TEMPLATE::extra_cells () const;
```

Returns the number of extra cells in the VERTEX_TEMPLATE.

```
public: const af_node_mapping*
    VERTEX_TEMPLATE::get_mapping ();
```

Gets the parameter mapping for the VERTEX_TEMPLATE.

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

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

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

Returns TRUE if this can be deep copied.

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

Returns TRUE if this is use counted.

```
public: logical VERTEX_TEMPLATE::locate (
    parameter_token token,   // token ID
    int& offset,             // offset of value
    int& count               // number of cells
    ) const;
```

Gets the parameter location and site of the parameter for a given token in the VERTEX_TEMPLATE.

```
public: void VERTEX_TEMPLATE::merge (
    VERTEX_TEMPLATE*         // other VERTEX_TEMPLATE
    );
```

Merges tokens from another VERTEX_TEMPLATE into this VERTEX_TEMPLATE.

```
public: void VERTEX_TEMPLATE::merge_owner (
    ENTITY* other,           // given entity
    logical delete_other     // deleting owner
    );
```

Notifies the VERTEX_TEMPLATE that its owning ENTITY is about to be merged with given entity. The application has the chance to delete or otherwise modify the attribute. After the merge, this owner will be deleted if the logical deleting owner is TRUE, otherwise it will be retained and other entity will be deleted. The default action is to do nothing. This function is supplied by the application whenever it defines a new attribute, and is called when a merge occurs.

```
public: void VERTEX_TEMPLATE::modify (
    int n_tokens,            // number of tokens
    parameter_token tokens[]// tokens
    );
```

Changes the tokens defined in a VERTEX_TEMPLATE.

```
public: af_node_instance*
    VERTEX_TEMPLATE::new_instance ();
```

Creates a new instance of the node.

```
public: logical
    VERTEX_TEMPLATE::normal_defined () const;
```

Returns TRUE if the VERTEX_TEMPLATE contains a NORMAL_TOKEN.

```
public: VERTEX_TEMPLATE& VERTEX_TEMPLATE::operator= (
    const VERTEX_TEMPLATE&  // vertex template
    );
```

Implements an assignment operator, which makes a copy of a VERTEX_TEMPLATE.

```
public: logical VERTEX_TEMPLATE::parameter_defined (
    parameter_token id      // token ID
    ) const;
```

Determines whether the VERTEX_TEMPLATE contains the specified parameter token.

```
public: void VERTEX_TEMPLATE::print (
    FILE*                       // file pointer
        = stderr
    ) const;
```

Prints the contents of the VERTEX_TEMPLATE on the specified file.

```
public: void VERTEX_TEMPLATE::print (
    FILE*                       // file pointer
        = (&_iob[2])
    ) const;
```

Prints the contents of the VERTEX_TEMPLATE on the specified file.

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

Remove from VERTEX_TEMPLATE if the use count reaches zero.

---

```
public: void VERTEX_TEMPLATE::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.

| | |
|---|---|
| af_node_mapping::restore | Restore the information from the node mapping |

---

```
public: logical VERTEX_TEMPLATE::same (
    VERTEX_TEMPLATE*         // VERTEX_TEMPLATE
    ) const;
```

Determines if two VERTEX_TEMPLATEs are the same.

---

```
public: virtual void VERTEX_TEMPLATE::set_use_count (
    int val                  // value for count
    );
```

Sets the count for the number of instances.

---

```
public: VERTEX_TEMPLATE* VERTEX_TEMPLATE::share ();
```

Increments a VERTEX_TEMPLATE's use count.

---

```
public: void VERTEX_TEMPLATE::split_owner (
    ENTITY* new_entity       // new entity
    );
```

Notifies the VERTEX_TEMPLATE that its owner is about to be split into two parts. The application has the chance to duplicate or otherwise modify the attribute. The default action is to do nothing. This function is supplied by the application whenever it defines a new attribute, and is called when a split occurs.

---

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

Returns the string “vertex_template”.

---

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

Counts the number of instances.

---

```
public: logical VERTEX_TEMPLATE::valid () const;
```

Determines whether the VERTEX_TEMPLATE is fully defined.

Internal Use: save, save_common

Related Fncs:

---

is_VERTEX_TEMPLATE