Chapter 5.
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_COL

Class: Colors, Attributes, SAT Save and Restore

Purpose: Stores color information for an ENTITY.

Derivation: ATTRIB_COL : ATTRIB_TSL : ATTRIB : ENTITY : ACIS_OBJECT : –

SAT Identifier: "colour"

Filename: rbase/rnd_husk/attribs/col_attr.hxx

Description: This class defines the color attribute and stores the the color information for an ENTITY.

Limitations: None

References: None

Data:

None

Constructor:

```
public: ATTRIB_COL::ATTRIB_COL (
    ENTITY*                 // entity
        = NULL,
    int                     // color value
        = 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_COL(...)), because this reserves the memory on the heap, a requirement to support roll back and history management.

Destructor:

```
public: virtual void ATTRIB_COL::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_COL::~ATTRIB_COL ();
```

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_COL(...) then later x–>lose.)

Methods:

```
public: int ATTRIB_COL::colour () const;
```

Returns the rgb_color values.

```
public: virtual void ATTRIB_COL::copy_owner (
    ENTITY*                 // entity being copied
    );
```

Virtual function called when an owner entity is being copied.

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

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

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

Returns TRUE if this can be deep copied.

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

Notifies the ATTRIB_COL 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_COL::pattern_compatible () const;
```

Returns TRUE if this is pattern compatible.

```
public: void ATTRIB_COL::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.

read_int	Integer representing color

```
public: void ATTRIB_COL::set_colour (
    int                     // color value
    );
```

Sets the color value.

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

Notifies the ATTRIB_COL 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*
    ATTRIB_COL::type_name () const;
```

Returns the string “colour”.

Internal Use: save, save_common

Related Fncs: is_ATTRIB_COL

ATTRIB_RGB

Class: Colors, Attributes, SAT Save and Restore

Purpose: Stores rgb color information for an ENTITY.

Derivation: ATTRIB_RGB : ATTRIB_ST : ATTRIB : ENTITY : ACIS_OBJECT : –

SAT Identifier: “rgb_color”

Filename: rbase/rnd_husk/attribs/rgb_attr.hxx

Description: This class defines RGB color attribute and stores the RGB color information for an ENTITY. This attribute takes precedence over the ATTRIB_COL class when displaying entities.

Limitations: None

References: RBASE rgb_color

Data: None

Constructor:

```
public: ATTRIB_RGB::ATTRIB_RGB ();
```

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

```
public: ATTRIB_RGB::ATTRIB_RGB (
    ENTITY*,                // entity
    rgb_color               // rgb color value
    );
```

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

Destructor:

```
public: virtual void ATTRIB_RGB::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_RGB::~ATTRIB_RGB ();
```

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_RGB(...) then later x–>lose.)

Methods:

```
public: rgb_color ATTRIB_RGB::color () const;
```

Returns the rgb_color values.

```
public: virtual void ATTRIB_RGB::copy_owner (
    ENTITY*                 // entity being copied
    );
```

Virtual function called when an owner entity is being copied.

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

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

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

Returns TRUE if this can be deep copied.

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

Notifies the ATTRIB_RGB 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_RGB::pattern_compatible () const;
```

Returns TRUE if this is pattern compatible.

```
public: void ATTRIB_RGB::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.

read_real	Red color component
read_real	Green color component
read_real	Blue color component

```
public: void ATTRIB_RGB::set_color (
    rgb_color               // rgb color value
    );
```

Sets the rgb_color values.

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

Notifies the ATTRIB_RGB 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*
    ATTRIB_RGB::type_name () const;
```

Returns the string “rgb_color”.

Internal Use: save, save_common

Related Fncs: is_ATTRIB_RGB

NORENDER_ATTRIB

Class: Rendering Control, SAT Save and Restore

Purpose: Defines a generic attribute type that can mark a face or entity to not be rendered.

Derivation: NORENDER_ATTRIB : ATTRIB_ST : ATTRIB : ENTITY : ACIS_OBJECT : –

SAT Identifier: “norender_attribute”

Filename: rbase/rnd_husk/attribs/no_rend.hxx

Description: Refer to Purpose.

Limitations: None

References: None

Data:

None

Constructor:

```
public: NORENDER_ATTRIB::NORENDER_ATTRIB (
    ENTITY*                  // entity pointer
        = 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 NORENDER_ATTRIB(...)), because this reserves the memory on the heap, a requirement to support roll back and history management.

Declares the generic attribute type which marks a face or entity to not be rendered.

Destructor:

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

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

Methods:

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

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

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

Returns TRUE if this can be deep copied.

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

Returns TRUE if this is pattern compatible.

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

Returns the string “norender_attribute”.

Internal Use: save, save_common

Related Fncs:

find_NORENDER_ATTRIB, is_NORENDER_ATTRIB

rbase_app_callback

Class: Rendering Control, Callbacks

Purpose: Implements routines to handle various rendering application callbacks for image output and interrupts.

Derivation: rbase_app_callback : toolkit_callback : –

SAT Identifier: None

Filename: rbase/rnd_husk/intrface/rbapp_cb.hxx

Description: Callbacks handled include starting and stopping image output, output of a scanline of image data to the application, and interrupt handling.

Limitations: None

References: None

Data:

None

Constructor:

None

Destructor:

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

C++ destructor, deleting a rbase_app_callback.

Methods:

```
public: virtual int
    rbase_app_callback::check_interrupt ();
```

Handles user interrupts.

```
public: virtual void rbase_app_callback::image_end (
    int                     // frame number
    );
```

Terminates image output.

```
public: virtual void
    rbase_app_callback::image_scanline (
    int,                    // y raster line value
    void*                   // array of pixel data
    );
```

Displays a scanline of image data.

```
public: virtual void
    rbase_app_callback::image_start (
    int,                    // frame number of image
    int,                    // upper–left pixel loc
    int,                    // upper–left pixel loc
    int,                    // image width in pixels
    int,                    // image height in pixels
    float,                  // scaling factor
    int                     // clear screen before
                            // displaying
    );
```

Begins image output.

Related Fncs:

add_rbase_app_cb, remove_rbase_app_cb

Render_Arg

Class: Rendering Control, SAT Save and Restore

Purpose: Provides the mechanism for getting and setting shader parameters.

Derivation: Render_Arg : –

SAT Identifier: None

Filename: rbase/rnd_husk/include/rh_args.hxx

Description: Render_Arg objects are used for setting and getting shader parameters. Shader parameters are identified by an enumerated type, and are accessed by a name in the form of a string. Because the parameter types are not known at compile time it is necessary to be able to set the parameters at run time according to their types. Render_Arg types include:

ARG_UNDEF	Undefined argument
ARG_INT	Integer argument
ARG_REAL	Real argument
ARG_STRING	String argument
ARG_COLOR	Three real values specifying the RGB color
ARG_VECTOR	Three real values for *x*, *y*, and *z*
ARG_ON_OFF	ON or OFF
ARG_FALL_OFF	One of the values: FALL_OFF_CONSTANT, FALL_OFF_INVERSE, FALL_OFF_ISL, FALL_OFF_INVERSE_CLAMP, or FALL_OFF_ISL_CLAMP

Limitations: None

References: None

Data:

None

Constructor:

```
public: Render_Arg::Render_Arg ();
```

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

```
public: Render_Arg::Render_Arg (
    const char* val         // string
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments, creating a Render_Arg of type ARG_STRING.

```
public: Render_Arg::Render_Arg (
    const Fall_Off_Type& val    // value
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments, creating a Render_Arg of type ARG_FALL_OFF.

```
public: Render_Arg::Render_Arg (
    const int& val          // integer
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments, creating a Render_Arg of type ARG_STRING.

```
public: Render_Arg::Render_Arg (
    const On_Off_Type& val  // on or off
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments, creating a Render_Arg of type ARG_ON_OFF.

```
public: Render_Arg::Render_Arg (
    const double& val       // double
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments, creating a Render_Arg of type ARG_STRING.

```
public: Render_Arg::Render_Arg (
    const double* val       // vector
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments, creating a Render_Arg of type ARG_VECTOR.

```
public: Render_Arg::Render_Arg (
    const Render_Color& val // color
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments, creating a Render_Arg of type ARG_COLOR. This value is composed of three doubles, corresponding to red, green, and blue.

Destructor:

None

Methods:

```
public: operator const Render_Arg::char* () const;
```

Returns the Render_Arg's string value. If the value is not a string value, the result is undefined.

```
public: void Render_Arg::debug (
    FILE* fp                // debug file
    );
```

Supports debugging.

```
public: operator Render_Arg::double () const;
```

Returns the Render_Arg's real value. If the value is not a real value, the result is undefined.

```
public: operator Render_Arg::double* () const;
```

Returns the Render_Arg's vector value. If the value is not a vector value, the result is undefined.

```
public: operator Render_Arg::Fall_Off_Type () const;
```

Returns the Render_Arg's Fall_Off_Type value. If the value is not a Fall_Off_Type value, the result is undefined.

```
public: void Render_Arg::free_string ();
```

Frees the string memory pointed to by a Render_Arg of type ARG_STRING. Use this method to free the memory when completed, such as when the Render_Arg is destroyed and no other objects reference the string.

```
public: operator Render_Arg::int () const;
```

Returns the Render_Arg's integer value. If the value is not an integer value, the result is undefined.

```
public: operator Render_Arg::On_Off_Type () const;
```

Returns the Render_Arg's On_Off_Type value. If the value is not a On_Off_Type value, the result is undefined.

```
public: Render_Arg Render_Arg::operator= (
    const char* val         // string value
    );
```

Returns the Render_Arg's string value.

```
public: Render_Arg Render_Arg::operator= (
    const Fall_Off_Type& val// fall off type
    );
```

Returns the Render_Arg's Fall_Off_Type value.

```
public: Render_Arg Render_Arg::operator= (
    const int& val          // integer value
    );
```

Returns the Render_Arg's integer value.

```
public: Render_Arg Render_Arg::operator= (
    const On_Off_Type & val // on off type
    );
```

Returns the Render_Arg's Fall_Off_Type value.

```
public: Render_Arg Render_Arg::operator= (
    const double& val       // double value
    );
```

Returns the Render_Arg's double value.

```
public: Render_Arg Render_Arg::operator= (
    const double* val       // vector value
    );
```

Returns the Render_Arg's vector value.

```
public: Render_Arg Render_Arg::operator= (
    const Render_Arg& rarg  // render argument
    );
```

Returns the Render_Arg's value.

```
public: Render_Arg Render_Arg::operator= (
    const Render_Color& val // render color
    );
```

Returns the Render_Arg's Render_Color value.

```
public: operator Render_Arg::Render_Color () const;
```

Returns the Render_Arg's Render_Color value. If the value is not a Render_Color value, the result is undefined.

```
public: logical Render_Arg::restore ();
```

Restores the Render_Arg. The restore function does the actual work. It calls the base class, then reads the selector, if the save file is new enough.

```
read_int                                Argument Type
switch( arg_type )
    case ARG_INT:
        read_int                        integer value
        break
    case ARG_REAL:
        read_real                       real value
        break
    case ARG_STRING:
        rh_restore_string               string to read in
        break
    case ARG_COLOR:
        read_real                       First color value
        read_real                       Second color value
        read_real                       Third color value
        break
    case ARG_VECTOR:
        read_real                       x value
        read_real                       y value
        read_real                       z value
        break
    case ARG_ON_OFF:
        read_int                        on or off value
        break
    case ARG_FALL_OFF:
        read_int                        fall off value
        break
    case ARG_UNDEF:
        break
```

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

Saves the arg type, followed by the arg value, which can vary by type.

```
public: Arg_Type Render_Arg::type () const;
```

Returns the type of Render_Arg. If the value is not a Render_Arg type value, the result is undefined.

Related Fncs:

None

Render_Color

Class: Rendering Control, Color Patterns

Purpose: Represents an RGB color.

Derivation: Render_Color : –

SAT Identifier: None

Filename: rbase/rnd_husk/include/rh_col.hxx

Description: Colors are represented in terms of an RGB triple, where each color component ranges from 0 to 1, with 1 representing the maximum intensity. The array of three values is indexed with 0 corresponding to red, 1 to green, and 2 to blue.

Limitations: None

References: None

Data:

None

Constructor:

```
public: Render_Color::Render_Color ();
```

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

```
public: Render_Color::Render_Color (
    double r,               // red
    double g,               // green
    double b                // blue
    );
```

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

Destructor:

None

Methods:

```
public: double Render_Color::blue () const;
```

Returns the normalized blue component of the color value.

```
public: double Render_Color::green () const;
```

Returns the normalized green component of the color value.

```
public: void Render_Color::print (
    const char* sep         // separator character
        = " ",
    FILE* output            // output stream
        = stdout
    );
```

Prints the RGB color values, where each value is separated from the next by the separator character.

```
public: double Render_Color::red () const;
```

Returns the normalized red component of the color value.

```
public: void Render_Color::set_blue (
    double b                // blue
    );
```

Sets the blue color value to a double value between 0 and 1.

```
public: void Render_Color::set_green (
    double g                // green
    );
```

Sets the green color value to a double value between 0 and 1.

```
public: void Render_Color::set_red (
    double r                // red
    );
```

Sets the red color value to a double value between 0 and 1.

Related Fncs:

None

rgb_color

Class: Colors

Purpose: Defines the red, green, and blue colors for the display.

Derivation: rgb_color : –

SAT Identifier: None

Filename: rbase/rnd_husk/rgbcolor.hxx

Description: This class defines the red, green, and blue colors for the display.

Limitations: None

References: by RBASE ATTRIB_RGB

Data:

None

Constructor:

```
public: rgb_color::rgb_color ();
```

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

```
public: rgb_color::rgb_color (
    double p[ 3 ]           // array of 3 doubles
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments. Creates a rgb_color using the specified array of three doubles.

```
public: rgb_color::rgb_color (
    double r,               // red color value
    double g,               // green color value
    double b                // blue color value
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments. Creates a rgb_color using the specified three doubles.

```
public: rgb_color::rgb_color (
    int index               // integer color value
    );
```

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

Destructor:

None

Methods:

```
public: double rgb_color::blue () const;.
```

Returns the blue color value

```
public: double rgb_color::green () const;
```

Returns the green color value.

```
public: operator rgb_color::int ();
```

Extracts a color value.

```
public: RGBColor* rgb_color::mac_color (
    RGBColor* mac_rgb       // mac color
    );
```

When running in the Macintosh Environment, then this defines an additional method for converting to an RGBColor record.

```
public: int rgb_color::operator!= (
    const rgb_color& c      // rgb color value
    ) const;
```

Determines whether the color is not equal to another color.

```
public: int rgb_color::operator== (
    const rgb_color& c      // rgb color value
    ) const;
```

Determines whether a color is equal to another color.

```
public: double rgb_color::red () const;
```

Returns the red color value.

```
public: void rgb_color::set_blue (
    double b                // blue component
    );
```

Sets the blue component.

```
public: void rgb_color::set_green (
    double g                // green component
    );
```

Sets the green component.

```
public: void rgb_color::set_mac_color (
    RGBColor mac_rgb        // color for mac
    );
```

When running in the Macintosh Environment, then this defines an additional method for converting to an RGBColor record.

```
public: void rgb_color::set_red (
    double r                // red component
    );
```

Sets the red component.

```
public: void rgb_color::set_windows_color (
    unsigned long from      // Windows GDI color
    );
```

For NT platforms only. Set the color values from a Windows COLORREF.

```
public: unsigned long
    rgb_color::windows_color () const;
```

For NT platforms only. Convert to a Windows COLORREF.

Related Fncs:

None

RH_BACKGROUND

Class: Backgrounds and Foregrounds, SAT Save and Restore

Purpose: Defines a background.

Derivation: RH_BACKGROUND : RH_ENTITY : ENTITY : ACIS_OBJECT : –

SAT Identifier: “rh_background”

Filename: rbase/rnd_husk/include/rh_enty.hxx

Description: This class defines the color of a pixels at any point in the image which is not covered by an entity surface. A background can comprise a single uniform color or pattern, or can be composed of a previously–generated image or an image scanned from a photograph. Only one background can be active at any one time.

The primary constructors for RH_BACKGROUNDs accept a parameter to identify the type of background. The parameterless constructor creates a NULL entity that has no effect in terms of rendering, but it is supplied to support attribute save and restore.

Limitations: None

References: None

Data:

None

Constructor:

```
public: RH_BACKGROUND::RH_BACKGROUND ();
```

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

```
public: RH_BACKGROUND::RH_BACKGROUND (
    const char* name        // type of background
    );
```

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

Constructs a background of the type specified by the character string. This string should be the name of an implemented shader, such as “plain“ or “graduated” depending on the renderer.

```
public: RH_BACKGROUND::RH_BACKGROUND (
    RH_BACKGROUND* old      // old background
    );
```

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

Destructor:

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

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

Methods:

```
public: void RH_BACKGROUND::active (
    logical flag            // active
    );
```

Marks the background as being active (TRUE) or inactive (FALSE). Only the single active background participates in rendering operations.

```
public: void RH_BACKGROUND::debug_details (
    FILE* fp                // debug file
    ) const;
```

Supports the debug mechanism by providing details of the background.

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

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

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

Returns TRUE if this can be deep copied.

```
public: void RH_BACKGROUND::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

```
protected: void RH_BACKGROUND::restore_internal (
    int data_id             // identifier
    );
```

Internally supports the restore mechanism for backgrounds. The identifier specifies the type of component, and it can either be STANDARD_DATA or ADVANCED_DATA. This method should not be called by applications.

if (data_id == RH_STANDARD_DATA)	
read_real	First color information
read_real	First color information
read_real	First color information
read_real	Second color information
read_real	Second color information
read_real	Second color information
else if (data_id == RH_ADVANCED_DATA)	
rh_restore_string	Restore the string for advanced data.
rh_restore_pi_shader	Restore shader information based on restored string.

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

Returns the string “rh_background”.

Internal Use: save, save_common, save_internal

Related Fncs: is_RH_BACKGROUND

RH_ENTITY

Class: Rendering Control, SAT Save and Restore

Purpose: Provides common methods and data for other rendering classes.

Derivation: RH_ENTITY : ENTITY : ACIS_OBJECT : –

SAT Identifier: “rh_entity”

Filename: rbase/rnd_husk/include/rh_enty.hxx

Description: Rendering entities provide the basis for manipulating the appearance of ACIS geometric entities, image backgrounds and lighting conditions. Child classes include RH_BACKGROUND, RH_FOREGROUND, RH_ENVIRONMENT_MAP, RH_LIGHT, RH_MATERIAL, and RH_TEXTURE_SPACE.

The primary constructors for RH_LIGHT, RH_FOREGROUND, and RH_BACKGROUND require a parameter that specifies the type of shader of that class. The parameterless constructors for these types create a NULL entity that has no effect in terms of rendering, but are supplied to support attribute save and restore.

Limitations: None

References: None

Data:

```
protected int ext_id;
```
Provides a link to an external data base (or file) in which the data associated with the ENTITY can be stored or accessed.

```
protected void *li_handle;
```
Points to internal data structures created by the rendering component.

Constructor:

```
public: RH_ENTITY::RH_ENTITY ();
```

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

Destructor:

```
public: virtual void RH_ENTITY::lose ();
```

Posts a delete bulletin to the bulletin board indicating the instance is no longer used in the active model. The lose methods for attached attributes are also called.

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

This C++ destructor should never be called directly. Instead, applications should use the overloaded lose method inherited from the ENTITY class, because this supports history management. (For example, x=new RH_ENTITY(...) then later x–>lose.)

Methods:

```
public: logical RH_ENTITY::check_handle ();
```

Returns TRUE if the handle pointing to the entity's internal data is valid; otherwise, it returns FALSE if the handle is NULL.

```
public: virtual void RH_ENTITY::debug_details (
    FILE* fp                // debug file
    ) const;
```

Supports the debug mechanism by providing details of the background.

```
public: virtual void RH_ENTITY::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: void* RH_ENTITY::handle () const;
```

Returns a pointer to the internal data structure for this entity.

```
public: int RH_ENTITY::id () const;
```

Returns the ID for the RH_ENTITY list.

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

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

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

Returns TRUE if this can be deep copied.

```
public: void RH_ENTITY::restore_common ();
```

The RESTORE_DEF macro expands to the restore_common method, which is used in reading information from a SAT file. This method is never called directly. It is called by a higher hierarchical function if an item in the SAT file is determined to be of this class type. An instance of this class will already have been created through the allocation constructor. This method then populates the class instance with the appropriate data from the SAT file.

```
read_int                                data id
if ( saveres_external() && ( data_id == RH_EXTERNAL_DATA ))
    read_int                            external id
    restore_external                    save the external data
else
    restore_internal                    save the internal data
```

```
protected: void RH_ENTITY::restore_external ();
```

Application–supplied method to support the restore mechanism. Refer to the related function, rh_set_external_saveres.

```
external_restore_func                   The result of this is restored flag
```

```
protected: virtual void RH_ENTITY::restore_internal (
    int data_id                 // identifier
    );
```

Internally supports the restore mechanism for an RH_ENTITY. The identifier specifies the type of component, and it can be either STANDARD_DATA or ADVANCED_DATA. This method should not be called by applications.

```
read_int                                data ID
read_int                                external ID
if ( saveres_external() && ( data_id == RH_EXTERNAL_DATA ))
    restore_external
else
    restore_internal
```

```
protected: logical
    RH_ENTITY::saveres_external () const;
```

External save and restore is performed by two application supplied functions. These are set using the friend function rh_set_external_savres. If the two functions are not provided (one or both are NULL), then the render component will save and restore from the current ACIS sat file.

Only an external “id” number is actually saved with the ACIS file. The full set of parameters necessary to restore the RH_ENTITY must be saved in an external file/data base.

```
protected: void RH_ENTITY::save_external () const;
```

External save and restore is performed by two application supplied functions. These are set using the friend function rh_set_external_savres. If the two functions are not provided (one or both are NULL), then the render component will save and restore from the current ACIS sat file.

Only an external 'id' number is actually saved with the ACIS file. The full set of parameters necessary to restore the RH_ENTITY must be saved in an external file/data base.

```
protected: virtual void
    RH_ENTITY::save_internal () const;
```

Each Render entity has its own internal save/restore methods.

```
protected: void RH_ENTITY::set_handle (
    void* handle            // data structure
    );
```

Sets the handle that points to the internal data structure for this entity. Before performing a change, it checks if the data structure is posted on the bulletin board. If not, the method calls backup to put an entry on the bulletin board.

```
protected: void RH_ENTITY::set_id (
    int new_id              // external ID
    );
```

Sets the external ID for the entity.

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

Returns the string "rh_entity".

Internal Use: save, save_common

Related Fncs: is_RH_ENTITY, rh_set_external_savres

RH_ENVIRONMENT_MAP

Class: Environment Maps, SAT Save and Restore

Purpose: Defines an environment map.

Derivation: RH_ENVIRONMENT_MAP : RH_ENTITY : ENTITY : ACIS_OBJECT : –

SAT Identifier: “rh_env_map”

Filename: rbase/rnd_husk/include/rh_enty.hxx

Description: Environment maps simulate interobject reflections, both between bodies in a scene and between a body and the external environment. RH_ENVIRONMENT_MAP objects are used with one of the component shaders of the other rendering entities specified by an RH_MATERIAL.

Environment mapping is supported for all shading modes, except flat and simple. Interobject reflections can be simulated by computing an environment map for the currently defined entities using the function, api_rh_render_cube_environment. More commonly, a set of scanned images form the basis for a environment mapping using api_rh_create_cube_environment. The application should not construct environment maps directly; instead, it should use the appropriate API.

Limitations: None

References: None

Data:

None

Constructor:

```
public: RH_ENVIRONMENT_MAP::RH_ENVIRONMENT_MAP ();
```

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

```
public: RH_ENVIRONMENT_MAP::RH_ENVIRONMENT_MAP (
    RH_ENV_DESC* desc,      // descriptor
    logical immediate       // generate dataset
        = TRUE
    );
```

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

If immediate is set, then the internal data is generated immediately.

Destructor:

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

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

Methods:

```
public: void RH_ENVIRONMENT_MAP::debug_details (
    FILE* fp                // debug file
    ) const;
```

Supports the debug mechanism by providing details of the environment map.

```
public: virtual void RH_ENVIRONMENT_MAP::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: void RH_ENVIRONMENT_MAP::evaluate ();
```

Builds an internal environment map from the data parameters that describe the RH_ENVIRONMENT_MAP.

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

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

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

Returns TRUE if this can be deep copied.

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

Returns the string “rh_env_map”.

Internal Use: save, save_common

Related Fncs:

---

is_RH_ENVIRONMENT_MAP

RH_FOREGROUND

Class: Backgrounds and Foregrounds, SAT Save and Restore

Purpose: Defines a foreground.

Derivation: RH_FOREGROUND : RH_ENTITY : ENTITY : ACIS_OBJECT : –

SAT Identifier: “rh_foreground”

Filename: rbase/rnd_husk/include/rh_enty.hxx

Description: A foreground is the counterpart to a background. Foreground shaders provide an extra level of image processing during the shading process. It can be thought of as a filter and may be used to support atmospheric effects, such as fog or depth cueing. Only one foreground can be active at any given time.

The primary constructors for RH_FOREGROUNDs accept a parameter to identify the type of foreground. The parameterless constructor creates a NULL entity that has no effect in terms of rendering, but it is supplied to support attribute save and restore.

Limitations: None

References: None

Data:

None

Constructor:

```
public: RH_FOREGROUND::RH_FOREGROUND ();
```

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

```
public: RH_FOREGROUND::RH_FOREGROUND (
    const char* name        // type for foreground
    );
```

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

Constructs a foreground of the type specified by the character string. This string is the name of an implemented shader, such as “fog” depending on the renderer.

```
public: RH_FOREGROUND::RH_FOREGROUND (
    RH_FOREGROUND* old      // type of old foreground
    );
```

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

Destructor:

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

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

Methods:

```
public: void RH_FOREGROUND::active (
    logical flag            // active or not?
    );
```

Marks the foreground as being active (TRUE) or inactive (FALSE). Only one foreground can be active during a rendering operation.

```
public: void RH_FOREGROUND::debug_details (
    FILE* fp                // debug file
    ) const;
```

Supports the debug mechanism by providing details of the foreground.

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

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

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

Returns TRUE if this can be deep copied.

```
public: void RH_FOREGROUND::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

```
protected: void RH_FOREGROUND::restore_internal (
    int data_id             // identifier
    );
```

Internally supports the restore mechanism for foregrounds. The identifier specifies the type of component, and it can be either STANDARD_DATA or ADVANCED_DATA. This method should not be called by applications.

```
if ( data_id == RH_STANDARD_DATA )
    read_string                 name string
    read_real                   First color data
    read_real                   First color data
    read_real                   First color data
    read_real                   Second color data
    read_real                   Second color data
    read_real                   Second color data
else if ( data_id == RH_ADVANCED_DATA )
    rh_restore_string           Name string for shader
    rh_restore_pi_shader        Restore appropriate shader
```

```
protected: void
    RH_FOREGROUND::save_internal () const;
```

Each Render entity has its own internal save/restore methods.

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

Returns the string “rh_foreground”.

Internal Use: save, save_common

Related Fncs:

---

is_RH_FOREGROUND

RH_LIGHT

Class: Lights and Shadows, SAT Save and Restore

Purpose: Defines a light source.

Derivation: RH_LIGHT : RH_ENTITY : ENTITY : ACIS_OBJECT : –

SAT Identifier: “rh_light”

Filename: rbase/rnd_husk/include/rh_enty.hxx

Description: RH_LIGHTs define light sources within the Renderer. Supported light source types are “ambient,” “distant,” “eye,” “point,” and “spot.”

An enumerated type, Fall_Off_Type, is a parameter to some light types which selects how the intensity of the light varies with the distance from the light source, and has possible values of FALL_OFF_CONSTANT, FALL_OFF_INVERSE, or FALL_OFF_INVERSE_SQUARE.

Shadowing is supported for distant, point, and spot in all rendering modes except flat and simple. If an image is to be rendered with shadows, a shadow map must be computed before rendering, using api_rh_create_light_shadow for each light for which shadows are required. A shadow map is view–independent and can be reused for any number of images provided there is no change in the light source geometry or the entities it illuminates.

Limitations: None

References: None

Data:

None

Constructor:

```
public: RH_LIGHT::RH_LIGHT ();
```

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

```
public: RH_LIGHT::RH_LIGHT (
    const char* name         // type of light
    );
```

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

Constructs a light of the type specified by the character string. This string should be the name of an implemented shader, such as “ambient” or “spot” depending on the renderer.

```
public: RH_LIGHT::RH_LIGHT (
    RH_LIGHT* old            // old light
    );
```

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

Overloads the C++ new operator to allocate space on the portion of the heap controlled by ACIS. This is used in conjunction with the other constructors. The C++ sizeof function can be used to obtain the size_t of the object.

Destructor:

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

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 RH_LIGHT(...) then later x–>lose.)

Methods:

```
public: void RH_LIGHT::active (
    logical flag            // active or not?
    );
```

Marks the light as being active (TRUE) or inactive (FALSE). Multiple lights can be active at any given time.

```
public: logical RH_LIGHT::active_state ();
```

Interface for checking the active state.

```
public: void RH_LIGHT::debug_details (
    FILE* fp                // debug file
    ) const;
```

Supports the debug mechanism by providing details of the light.

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

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

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

Returns TRUE if this can be deep copied.

```
public: void RH_LIGHT::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

```
protected: void RH_LIGHT::restore_internal (
    int data_id             // identifier
    );
```

Internally supports the restore mechanism for lights. The identifier specifies the type of component, and it can be either STANDARD_DATA or ADVANCED_DATA. This method should not be called by applications.

```
if ( data_id == RH_STANDARD_DATA )
    read_string                 name string
    read_real                   it
    read_real                   First color data
    read_real                   First color data
    read_real                   First color data
    read_real                   position data
    read_real                   position data
    read_real                   position data
else if ( data_id == RH_ADVANCED_DATA )
    rh_restore_string           Name string for shader
    rh_restore_pi_shader        Restore appropriate shader
```

```
protected: void RH_LIGHT::save_internal () const;
```

Each Render entity has its own internal save/restore methods.

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

Returns the string “rh_light”.

Internal Use: save, save_common

Related Fncs: is_RH_LIGHT

RH_MATERIAL

Class: Materials, Color Patterns, Reflectance, Transparency, Displacement, SAT Save and Restore

Purpose: Defines a material consisting of color, displacement, reflectance, and transparency.

Derivation: RH_MATERIAL : RH_ENTITY : ENTITY : ACIS_OBJECT : –

SAT Identifier: “rh_material”

Filename: rbase/rnd_husk/include/rh_enty.hxx

Description: A material defines the appearance of the surface of an ACIS topological entity in terms of four components: color, reflectance, transparency, and displacement.

A color defines the color for any point on the surface of an entity to which applies and can be a simple single color or a complex pattern, such as a procedurally–defined marble effect.

The reflectance governs how the surface behaves visually in the presence of light. The reflectance defines the surface finish of an entity and models effects, such as matte, metal, or mirrored surfaces. Reflectance is not supported in the flat or gouraud rendering modes.

Transparency defines how transparent or opaque a surface is and the consequent visibility of entities that lie behind that surface. Transparency is not supported in the flat or gouraud rendering modes. The transparency component of a material can be switched on or off. If switched on, the transparency component of a material takes effect for those rendering modes where it is applicable.

A displacement simulates small surface perturbations by modifying the surface normal vector. They are an efficient method of simulating surface features such as regular indentations, which would be difficult or inefficient to model geometrically. Displacement is not supported in the flat or gouraud rendering modes. The displacement component of a material can be switched on or off. If switched on, the displacement component of a material takes effect for those rendering modes where it is applicable.

Limitations: None

References: by RBASE ATTRIB_RENDER

Data:

None

Constructor:

```
public: RH_MATERIAL::RH_MATERIAL ();
```

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

```
public: RH_MATERIAL::RH_MATERIAL (
    RH_MATERIAL* old        // old material
    );
```

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

```
public: RH_MATERIAL::RH_MATERIAL (
    void* handle            // data structure
    );
```

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

Constructs a material, initializing it using the internal data structure pointed to by handle.

Destructor:

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

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

Methods:

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

Increments the use count.

```
public: void RH_MATERIAL::debug_details (
    FILE* fp                // debug file
    ) const;
```

Supports the debug mechanism by providing details of the material.

```
public: virtual void RH_MATERIAL::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 RH_MATERIAL::
    deletable () const;
```

Usually use counted entities are marked not deletable because their lifetime is controlled by the entities that point to them.

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

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

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

Returns TRUE if this can be deep copied.

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

Returns TRUE if use counting is turned on.

```
public: virtual void RH_MATERIAL::remove (
    logical lose_if_zero    // if TRUE, call lose
    = TRUE                  // when use count becomes
                            // zero
    );
```

Decrements the use count and loses the entity if the count reaches zero and the flag is true.

```
public: void RH_MATERIAL::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

```
protected: void RH_MATERIAL::restore_internal (
    int data_id             // identifier
    );
```

Internally supports the restore mechanism for materials. The identifier specifies the type of component, and it can be either STANDARD_DATA or ADVANCED_DATA. This method should not be called by applications.

if (data_id == RH_STANDARD_DATA)	
read_real	First color data
read_real	First color data
read_real	First color data
read_real	ka data
read_real	kd data
read_real	ks data
read_real	ph data
else if (data_id == RH_ADVANCED_DATA)	
rh_restore_string	Name string for COLOR_COMP_SHADER
rh_restore_pi_shader	Restore appropriate shader
rh_restore_string	Name string for REFLECTANCE_COMP_SHADER
rh_restore_pi_shader	Restore appropriate shader
rh_restore_string	Name string for DISPLACEMENT_COMP_SHADER
rh_restore_pi_shader	Restore appropriate shader
rh_restore_string	Name string for TRANSPARENCY_COMP_SHADER
rh_restore_pi_shader	Restore appropriate shader

```
protected: void RH_MATERIAL::save_internal () const;
```

Each Render entity has its own internal save/restore methods.

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

Set the use count.

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

Returns the string “rh_material”.

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

Enables use count.

Internal Use: save, save_common

Related Fncs: is_RH_MATERIAL

RH_TEXTURE_SPACE

Class: Texture Spaces, SAT Save and Restore

Purpose: Defines a texture space.

Derivation: RH_TEXTURE_SPACE : RH_ENTITY : ENTITY : ACIS_OBJECT : –

SAT Identifier: “rh_texture_space”

Filename: rbase/rnd_husk/include/rh_enty.hxx

Description: RH_TEXTURE_SPACE entities assist in the production of a shading effect known as a wrapped texture. A wrapped texture produces the effect of a sheet of paper shrink wrapped onto the surface of a solid object. A texture space uses one of several texture space shaders to map between the coordinate system of the sheet and the coordinate system of the surface of the solid object. Texture space arguments are treated in a similar fashion to those of material components.

Limitations: None

References: by RBASE ATTRIB_RENDER

Data:

None

Constructor:

```
public: RH_TEXTURE_SPACE::RH_TEXTURE_SPACE ();
```

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

```
public: RH_TEXTURE_SPACE::RH_TEXTURE_SPACE (
    const char* name        // type of texture space
    );
```

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

Constructs a texture space of the type specified by the character string. This string should be the name of an implemented shader, such as “x” or “cylindrical” depending on the renderer.

```
public: RH_TEXTURE_SPACE::RH_TEXTURE_SPACE (
    RH_TEXTURE_SPACE* old   // old texture space
    );
```

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

Destructor:

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

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

Methods:

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

Increments the use count.

```
public: void RH_TEXTURE_SPACE::debug_details (
    FILE* fp                // debug file
    ) const;
```

Supports the debug mechanism by providing details of the texture space.

```
public: virtual void RH_TEXTURE_SPACE::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 RH_TEXTURE_SPACE::deletable ()
const;
```

Usually use counted entities are marked not deletable because their lifetime is controlled by the entities that point to them.

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

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

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

Returns TRUE if this can be deep copied.

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

Returns TRUE if use counting is turned on.

```
public:virtual void RH_TEXTURE_SPACE::remove (
    logical lose_if_zero    // toggle
    = TRUE
    );
```

Decrements the use count and loses the entity if the count reaches zero and the flag is true.

```
public: void RH_TEXTURE_SPACE::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.

rh_restore_string(name) name
rh_restore_pi_shader(th, RH_TEXTURE_SPACE_SHADER, name)

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

Set the use count.

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

Returns the string “rh_texture_space”.

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

Enables use count.

Internal Use: save, save_common

Related Fncs: is_RH_TEXTURE_SPACE