SPHIGS for ANSI-C (v1.0, March 8, 1993)

David Frederick Sklar
Christopher R. Brown

The Simple Programmer's Hierarchical Graphics Standard is composed of a library of functions, and a header file ("sphigs.h") that defines custom data types and constants, and which prototypes all SPHIGS routines. This paper is a complete but *extremely terse* description of the ANSI-C binding. You must read Chapter 7 of *Computer Graphics — Principles and Practice* (Foley, van Dam, Feiner, and Hughes, Addison-Wesley, 1990) for a full description of SPHIGS. NOTE: SPHIGS requires the SRGP library.

0) Comparison with Textbook Specification

---

1) States of the System
   1) initializing the system
   2) controlling screen regeneration

---

2) Maintenance of the Structure Database
   1) macroscopic editing
   2) intra-structure editing

---

3) Coordinate Systems
   1) geometric data types
   2) viewing transformations
   3) modeling transformations

---

4) Color and Shading

---

5) Creation of Graphics Objects
   1) polyLine
   2) polyMarker
   3) text
   4) fill area
   5) polyhedron
   6) rendering modes
   7) execution of subordinate structures

---

6) Dynamic Invisibility and Highlighting

---

7) Input
   1) control of input devices
   2) pick correlation

---

8) Control of Table Sizes

0 Comparison with Textbook Description

This section describes how SPHIGS differs from its textbook description, lists new features, features that are not currently supported but will be supported in future releases, and lists currently known bugs.

- Viewports: In the textbook figures, viewport boundaries were depicted via dashed rectangular outlines. This has been deemed too obtrusive for practical use and has been replaced by a background color — thus, SPHIGS viewports are opaque (a violation of the PHIGS standard). The application can specify a different background color for each view. The default color is white, meaning the viewport boundary will not be obvious. It is suggested that extremely light pastel-ish colors be used. (NOTE: unfortunately, there is currently no method for depicting viewport boundaries on black-and-white systems.) Overlapping viewports should be avoided.
- Rendering and Color: The GOURAUD rendering mode is not currently supported; LIT_FLAT is the current "quality" rendering mode. The textbook did not describe how colors are assigned to objects for the purposes of shaded rendering in the LIT_FLAT rendering mode, nor did it describe how simulated shading is achieved on monochrome implementations. See section 4.
- SPHIGS in a windowing environment: When run on X11 or the Macintosh, the SPHIGS "screen" appears in a window whose size is determined by the application at initialization time and may be changed by the user via a window manager action.
- **Viewing BUG!** When you call `SPH_evaluateViewMappingMatrix` with a projection type of ORTHOGRAPHIC, it is possible that the front and back clipping-planes distances you specify in that call will be lost. The problem has a simple avoidance mechanism: if you always call `SPH_setViewRepresentation` immediately after evaluating the view matrices, the problem never occurs.
- Fill areas and polyhedron faces: **MUST BE CONVEX!** No exceptions! Moreover, this version of SPHIGS does NOT handle faces that intersect.

New in this version:

- SPHIGS now builds transformation matrices with the same column orientation as used in Chapters 5 and 6 in the textbook. They are no longer the transpose of the matrices you'd expect to see. For more fun and confusion, see the NOTE at the end of section 3.
- The `FLAT` shading model is now available as a rendering mode, and multiple point light sources can be defined for the `LIT_FLAT` mode. This is described in section 4.
- Name sets for invisibility are now supported; see section 6. Highlighting is still unavailable in the current version of SPHIGS.
- `SPH_copyStructure` has finally been implemented (section 2.2).
- Pick correlation extensions!! Correlation is now supported for polylines, polymarkers and fill areas. Also, in the `WIREFRAME` rendering mode, it is no longer necessary to click exactly on polyhedra edges; primitives can be correlated anywhere on the surface of a facet.
- Rendering Ills Cured!! The rendering engine has been entirely rewritten using Binary Space Partitioning trees and will correctly draw z-ordered scenes with mixed polyline and polyfacet objects. Fill areas are now visible from either side, independent of their surface normal. See section 5.6.

1 States of the System

SPHIGS must be enabled before use, and disabled after use. At any time, SPHIGS is in exactly one of the following three operating states:

- SPHIGS disabled (initial state)
- SPHIGS enabled, no structure open
- SPHIGS enabled, structure open

1.1 initializing the system

```
void SPH_begin (int width, int height, int planecount, int shadecount);
```

This routine initializes SPHIGS and SRGP, and places SPHIGS in the enabled/no-structure-open state. The virtual-screen window is opened, with its initial dimensions (measure in pixels) determined by the first two parameters. (It can be resized by the user with a window manager, if the underlying SRGP platform supports that functionality.) The NPC "unit extent" (0.0 to 1.0 on both x and y) will map to the largest square that fits in this area.

The last two parameters allow the application to request color resources. This is explained in section 4.

```
void SPH_end (void)
```

The virtual screen window is closed, and SPHIGS enters the disabled state.

1.2 controlling screen regeneration

Whenever the application changes the structure database, SPHIGS regenerates the screen image immediately **unless** the implicit-regeneration mode has been set to `SUPPRESSED`. Its default value is `ALLOWED`. When implicit regeneration is suppressed, regeneration can be explicitly forced via the second function below. Otherwise, regeneration occurs when the implicit regeneration mode is restored to `ALLOWED`.

```
typedef enum {SUPPRESSED, ALLOWED} regenMode;
void SPH_setImplicitRegenerationMode (regenMode)
void SPH_regenerateScreen (void)
```

Double-buffering has been made available as a rendering option. If double-buffering is used, there is no flicker in any viewport since each time the scene is redrawn the picture is first generated on an off-screen canvas and is then copied to screen. Double-buffering requires a significant amount of memory overhead, but a more serious drawback is that rendering speed is lost during the expensive copy-pixel step. Rendering may become prohibitively slow with this flag on, so it is provided as an option.

```
void SPH_setDoubleBufferingFlag(int new_value)
```

Avoid making quick repeated calls to this function; this is wasteful, since SRGP allocates or frees the canvas buffer every time the flag is changed.

2 Maintenance of the Structure Database

The structure database is composed of two parts:

- Conceptually, a fixed number of structures exist in the universe, all initially empty. They are indexed by the numbers 0 through `MAX_STRUCTURE_ID`. Because a structure may invoke other structures, this set of structures actually is a set of interlocking DAGs (directed acyclic graphs) that we call **networks.**
- A *view table* having a fixed number of *views* exists; the views are indexed by the numbers 0 through `MAX_VIEW_INDEX`. Each view includes 0 or more references to networks; a single network can be referenced by more than one view. Not all networks are referenced. Those that are referenced are said to be *posted* and are therefore potentially visible.

There are two genres of editing:

- **macro-structure editing** of the structure database: posting/unposting structures, deleting structures and networks, etc.
- **intra-structure editing:** modification of the contents of a single structure

2.1 macro-structure editing

The following functions post and unpost. They may be called only when there is no open structure. The screen is regenerated immediately unless automatic regeneration is disabled.

```
typedef int structure_ID, view_ID;
void SPH_postRoot (structure_ID, view_ID);
void SPH_unpostRoot (structure_ID, view_ID);
void SPH_unpostAllRoots (view_ID);
```

2.2 intra-structure editing

One must open a structure in order to edit it. Only one structure may be opened at any time. All structures exist from time zero; one never *creates* a structure, one merely opens an empty structure and edits it. The editing operations are not considered "official" until the structure is closed, so screen updating is not performed while the structure is open.

```
void SPH_openStructure (structure_ID);
void SPH_closeStructure (void);
```

When a structure is opened, the element pointer is set to point to the very last element in the structure. Since elements are indexed by ascending integers, starting with 1, this means the element pointer is initialized to N (where N is the number of elements in the structure). If the structure being opened is empty, the element pointer is initialized to 0. The element pointer may be moved via the following functions. Movement to index 0 is useful for inserting new elements at the very beginning of a non-empty structure. Movement to an index which is meaningless (less than 0 or greater than N) causes a fatal error. Movement to a label causes a forward scan through the structure starting from (but not including) the current position of the element pointer; thus it is typically wise to set the element pointer to 0 before attempting to move the element pointer to a label. If the label is not found, a fatal error occurs.

```
typedef int elementIndex, label_ID;
void SPH_setElementPointer (elementIndex);
void SPH_offsetElementPointer (int delta);
void SPH_moveElementPointerToLabel (label_ID);
```

When a structure is open for editing, the application may inquire the index of the current element:

```
elementIndex SPH_inquireElementPointer (void);
```

The most common form of editing is simply the insertion of new elements into a structure. Whenever an element-generator function is called, the new element is placed immediately after the current element, and the element pointer is advanced to point to the new element.

Label elements have integer names which must be unique within the structure owning it. The following function generates a label element.

```
void SPH_label (label_ID);     ‡
```

NOTE: Throughout this document, each presentation of an element-generator routine is "highlighted" by the presence of this symbol: ‡

The following functions delete elements and automatically renumber the survivors. In all cases, after the deletion is made, the element pointer is moved to point to the element immediately preceding the first element deleted. The first function deletes the element indexed by the current value of the element pointer. The second function deletes the elements lying between, and including, the elements specified by the two element identifiers. The first identifier must be less than the second one; a fatal error occurs otherwise. The third function deletes the elements lying between, but not including, the label elements specified. The first label must appear in the structure before the second one, and both must appear after the current value of the element pointer; a fatal error occurs otherwise.

```
void SPH_deleteElement (void);
void SPH_deleteElementsInRange (elementIndex first, elementIndex second);
void SPH_deleteElementsBetweenLabels (label_ID first, label_ID second);
```

It is possible to copy all the elements of a given structure into the open structure (immediately after the current element) with the following library routine. The element pointer is then updated to point to the last element inserted. If the specified structure happens to be the open structure, its contents are copied into itself.

```
void SPH_copyStructure (structure_ID);
```

3 Coordinate systems

3.1 geometric data types

The following data types are defined in order to simplify the storage and specification of vectors, points, and matrices:

```
typedef double point[3], vector[3];
typedef double matrix[4][4];
```

Vectors or points may be initialized using this function:

```
double *SPH_defPoint (pt,  x, y, z)
double *pt;                          pointer to first of three consecutive doubles
double x, y, z;
```

3.2 viewing transformations

The SPHIGS viewing system involves an intermediate coordinate system called the VRC (Viewing Reference Coordinates). This first transformation simply reorients the world by specifying a new set of axes: UVN. The matrix which performs this is called the *view-orientation matrix* and can be calculated using the following function. The final argument to the function must be a pointer to a matrix (to be initialized by the function) allocated by the application.

```
void
SPH_evaluateViewOrientationMatrix
   (point view_ref_point, vector view_plane_normal, vector view_up_vector,
    matrix vo_matrix);
```

The next transformation sets up the view volume, which maps VRC to NPC (Normalized Projection Coordinates). To be more specific, it maps the contents of the view volume to a paralleliped in NPC space. (It is recommended, but not enforced, that the rectangle lie wholly inside the NPC bounded space [0,0]→[1,1].) The matrix which performs this is called the *view-mapping matrix*.

```
void
SPH_evaluateViewMappingMatrix
   (double umin, double umax,
    double vmin, double vmax,                 boundaries of window on view plane
    int proj_type,                        either PERSPECTIVE or ORTHOGRAPHIC
    point proj_ref_point,                            in uvn coordinates
    double proj_front_clip_distance,
       double proj_back_clip_distance,                 offsets from VRP on n axis
                                3D viewport: in this version, minz and maxz are ignored
    double proj_NPCviewport_minx, double proj_NPCviewport_maxx,
    double proj_NPCviewport_miny, double proj_NPCviewport_maxy,
    double proj_NPCviewport_minz, double proj_NPCviewport_maxz,
    matrix vm_matrix);
```

It is very important to understand that the projection-reference point and the window's boundaries (umin, umax, vmin, vmax) are specified NOT in WC, but as offsets from the VRP that is set by the orientation matrix. This means that in order to perform an animation which involves only simple camera motion, with no change in the "focal length" and no strange warpings, all one needs to do is change the view-orientation matrix. The prp will "follow" the vrp as the view orientation is changed.

Notice that there is no explicit way to specify focal length in this system. The distance between the VRP and the PRP can be manipulated to achieve similar results. Be careful! The z coordinate of the PRP must be positive! Also, if you specify the PRP in such a way that a line between it and the center of the view plane is not perpindicular to the view plane, shearing will occur.

The setting of an entry in the view table requires specification of both the matrices as well as a re-specification of the NPC viewport (again, with minz and maxz ignored):

```
void
SPH_setViewRepresentation
   (int view_index,
    matrix vo_matrix, matrix vm_matrix,
    double clip_NPCviewport_minx, double clip_NPCviewport_maxx,
    double clip_NPCviewport_miny, double clip_NPCviewport_maxy,
    double clip_NPCviewport_minz, double clip_NPCviewport_maxz);
```

The background color for a view's rectangular region of the screen is white by default, but you may change it (on a per-view basis):

```
void SPH_setViewBackgroundColor (int viewindex, int color);
```

A viewport can be turned on and off with the following functions (disabling a viewport erases it from view):

```
void SPH_enableView (int viewIndex);
void SPH_disableView (int viewIndex);
```

3.3 modeling transformations

Each structure has its own modeling coordinate system (MCS); primitives specified within it are placed in its MCS. When the execution of a structure (during traversal) begins, its local modeling transformation is initialized to the "identity" transformation. When a primitive element is executed, its raw coordinates are transformed by the current value of the local modeling transformation in order to place it within the MCS. When a structure-execution element is executed, the MCS of the subordinate structure is mapped into the local MCS using the local modeling transformation: thus the image of the subordinate network is transformed as a single entity to be placed within the local MCS.

The following function generates a structure element which, when executed, affects the local modeling transformation matrix, either by replacing (assigning) it or by concatenating it with a given matrix.

```
typedef enum {ASSIGN, PRECONCATENATE, POSTCONCATENATE} updateType;
void SPH_setModelingTransformation (matrix mat, updateType);    ‡
```

An application can use the following functions for creating a matrix to be stored or to be used to set a modeling transformation. Rotation, translation, scaling, and compositions thereof are supported. Rotation angles are specified in degrees, positive angles representing counter-clockwise rotation. In all functions, the last parameter specifies the application-allocated area (large enough to store a 4-by-4 array of `doubles`) in which the function is to place the calculated matrix.

```
void SPH_scale (double scaleX, double scaleY, double scaleZ,  matrix result)
void SPH_rotateX (double angleDegrees,  matrix result)
void SPH_rotateY (double angleDegrees,  matrix result)
void SPH_rotateZ (double angleDegrees,  matrix result)
void SPH_translate
     (double deltaX, double deltaY, double deltaZ,  matrix result)

void SPH_composeMatrix (matrix mat1, matrix mat2, matrix result)
```

In the textbook's Pascal code, a matrix could be calculated and made into an element in one statement, because Pascal can return arrays:

```
SPH_setModelingTransformation (SPH_scale(5.0,3.0,1.0), ASSIGN);
```

It was not possible to design an elegant method in ANSI C for the calculation routines to return a matrix in a manner that would allow the same convenience. Thus, creating a transformation element requires two C statements (unless the matrix is already available):

```
SPH_scale (5.0, 3.0, 1.0,  mat);
SPH_setModelingTransformation (mat, ASSIGN);
```

The SPHIGS header file `sphigs.h` defines a macro called `SPH_setModXform` that can come to the rescue if you abhor the two-statement sequence. The macro assumes that you have declared and are using a matrix called `temp_matrix`. Here I show the usage of the macro for the creation of a scale element:

```
SPH_setModXform (SPH_scale(5.0, 3.0, 1.0, temp_matrix), ASSIGN);
```

```
void SPH_oldComposeMatrix (matrix mat1, matrix mat2, matrix result)
```

NOTE: The original release of SPHIGS used a vector package that performed matrix-vector multiplications in the opposite commutative order as that described in the text. The matrix package now correctly uses column vectors multipilied to the left of the matrix.

The result of this confusion is that the old version of SPHIGS had to internally flip the order of multiplication in order to make composite transforms. Unfortunately, the original version of `SPH_compose`, which performed simple matrix multiplication, failed to perform this reversal. This has been fixed; the replacement function, `SPH_composeMatrix`, performs a multiply as you would expect, in the same order that you would write it out on paper.

`SPH_oldComposeMatrix` is provided for the sake of backward compatibility, to which `SPH_compose` has been aliased. This ensures that programs written with the original (beta) version of SPHIGS will compile and work as intended.

THE FINAL WORD: when introducing a new transformation, if you PRECONCATENATE (multiply on left) it with any extant transforms, the new transformation will occur after any pre-existing ones. To insert a transformation so that it happens before any others, POSTCONCATENATE it. This is consistent with the textbook description, and representative of how SPHIGS handles transformations internally.

```
Preconcatenation:    XFORM x OLD_CTM = NEW_CTM
Postconcatenation:   OLD_CTM x XFORM = NEW_CTM
```

```
void SPH_clearModelingTransformation (void)
```

The above function will restore the structure's coordinate system to that of its parent (or to the identity matrix if the structure is at the top level of hierarchy) by clearing any previous transformations. This is equivalent to `ASSIGN`ing the identity matrix.

4 Color and Shading

SPHIGS provides a device-independent method for dealing with color. An application can load the SPHIGS color table without concern for the number of colors actually available, and can assign colors to objects equally carefree. SPHIGS maps unavailable entries to white (for interiors) or black (for lines, text, markers, etc.). Moreover, when LIT_FLAT rendering is used, SPHIGS simulates shading (by using bitmap patterns) when it is not possible to use true colors for the shading.

The total number of available colors is dependent upon the number of planes requested via the third parameter to '`SPH_begin`', and on the capabilities of the underlying hardware. See the SRGP reference manual for information on color planes.

SPHIGS divides up the available colors in this manner: The first two colors are fixed at white (0) and black (1); the application may not modify these entries. The next C entries (2 through $(C+2)$) are called *custom colors* and are application-settable. The first F custom colors are called *flexicolors*. When a flexicolor is assigned to the interior of a facet or fill area, SPHIGS displays an appropriately darkened shade of that color when LIT_FLAT rendering mode is used. When a non-flexicolor is assigned to a facet or fill area, SPHIGS must simulate shading by using bitmap patterns.

The values of F and C are calculated by SPHIGS based on the total number of available SRGP colors and on the value of the 4'th parameter of '`SPH_begin`': the number of shades you wish to be created for each flexicolor. Obviously: the larger the number of shades per flexicolor, the fewer flexicolors SPHIGS will be able to provide.

After '`SPH_begin`' returns, you can inquire the values of F and C by examining these global variables: `NUM_OF_FLEXICOLORS` and `NUM_OF_APPL_SETTABLE_COLORS`.

Currently, SPHIGS only provides one routine for loading the color table:

```
void SPH_loadCommonColor (int colorindex, char *colorname);
```

For information on the second parameter, see the SRGP reference manual. When you load a color into a flexicolor entry, SPHIGS automatically generates a range of shades for that color. For this reason, you must not bypass this routine by calling SRGP directly.

SPHIGS renders a lit-flat view using a simple lighting model using a single point light source (specified in UVN coordinates, and thus attached to the camera and mobile as the camera is mobile) and Lambertian reflection. Ambient light is automatically extant as well. No light-source attenuation is considered to exist, and surfaces are not self-occluding. Because of these restrictions, best results occur when the light source is kept well away from the scene's objects themselves. The default position of the light source is above and behind the camera at a distance approximating infinity; it thus acts as a "sun" simulator.

```
void SPH_addPointLightSource (int viewIndex, int lightIndex,
    point position, boolean cameraRelative,
    double intensity, double attenuation)
void SPH_removePointLightSource (int viewIndex, int lightIndex)
void SPH_setMaxLightSourceIndex (int index)
```

These functions allow the application to control multiple light sources for any viewport. Note that this only makes a difference when the `LIT_FLAT` rendering mode has been selected. Only point lights (also called positional lights) are currently supported.

Each viewport is initialized with a single point light source, which moves relative to the camera and is positioned at an infinite distance behind and above it. More light sources can be defined by calls to `SPH_addPointLightSource`; each viewport can hold up to six point light sources (this number is arbitrary and can be changed with a call to `SPH_setMaxLightSourceIndex` before SPHIGS is initialized).

When the application adds a light source to a viewport, it sends a light index that acts as a handle to the new light. Any light that was previously defined under that index is deleted, the new light becomes active and the scene is regenerated.

A light can be positioned either in world coordinates or relative to the viewport's camera orientation (so that it moves with the camera) by setting the "cameraRelative" parameter accordingly. The application defines the light's intensity (a standard value is 1.0) and can specify an optional attenuation factor that controls intensity falloff over distance (use 0.0 for no attenuation).

Any light source can be disabled via a call to `SPH_removePointLightSource`. The viewport's initial light source is defined with an index of zero, and should be a sufficient to facilitate object recognition. Removing all the light sources from a viewport is not recommended.

```
void SPH_setViewPointLightSource (int viewIndex, double u, double v, double n)
```

Provided for backward compatibility. This sets the camera-relative position of the first light index (0) for the given viewport. The following calls produce the same effect:

```
SPH_setViewPointLightSource (viewID, u, v, n);
SPH_addPointLightSource (viewID, 0, SPH_defPoint(p, u,v,n), TRUE, 1.0, 0.0);
```

5 Creation of Graphics Objects

5.1 polyLine

```
SPH_polyLine (int vertex_count, point *vertices)   ‡
```

Attributes affecting the polyLine primitive:

```
SPH_setLineStyle (int value)   ‡                    See SRGP manual.
SPH_setLineColor (int value)   ‡
SPH_setLineWidthScaleFactor (double scale_factor)   ‡
```

5.2 polyMarker

```
SPH_polyMarker (int vertex_count, point *vertices)   ‡
```

Attributes affecting the polyMarker primitive:

```
SPH_setMarkerStyle (int value)    ‡                        See SRGP manual.
SPH_setMarkerColor (int value)    ‡
SPH_setMarkerSizeScaleFactor (double scale_factor)    ‡
```

5.3 text

Only the origin is affected by modeling and viewing transformations; the text itself is always horizontal, aligned with the virtual screen window. In this sense it is the equivalent of the ANNOTATION TEXT primitive from the PHIGS standard.

```
SPH_text (point origin, char *str)   ‡
```

The attributes affecting text are like their SRGP counterparts. In fact, you must call `SRGP_loadFontTable` to load fonts.

```
SPH_setTextFont (int font_index)    ‡
SPH_setTextColor (int color_index)   ‡
```

5.4 fill area

A fill area must be convex and planar; SPHIGS does not attempt to verify either of these conditions.

```
SPH_fillArea (int vertex_count, point *vertices)   ‡
```

Attributes that affect the appearance of both fill-area and polyhedron objects:

```
SPH_setInteriorColor (int color_index)   ‡
SPH_setEdgeColor (int color_index)   ‡
SPH_setEdgeStyle (int value)   ‡                  same values as for line style
SPH_setEdgeFlag (int value)   ‡              EDGE_VISIBLE or EDGE_INVISIBLE
SPH_setEdgeWidthScaleFactor (double scale_factor)   ‡
```

See section 4 for a description of the color-index attribute.

5.5 polyhedron

To create a polyhedron element, you present a complete list of the vertices, implicitly indexed from 0 to $(vcount - 1)$. You describe the facets by presenting a single array of numbers (SPHIGS data type `vertex_index`) that actually stores a concatenated set of facet descriptions. Each facet description is a sequence of $(V+1)$ numbers, where V is the number of vertices in the facet. The first V numbers are indices into the vertex list; the last number is (-1) and acts as a sentinel ending the facet specification. The vertex indices for a facet must be presented in counterclockwise order (as seen when looking at the facet's "outside" face). Because each facet must be a closed polygon, the first vertex in the list is implicitly the last as well.

The data type `vertex_index` must be used by the application programmer in order to guarantee future compatibility. It is of course `typedef`'ed to either a short or an int in the `sphigs.h` file.

Note: In this beta version of SPHIGS, FACETS MUST BE CONVEX!

```
void SPH_polyhedron     ‡
   (int vertex_count, int facet_count,
    point *vertices,  vertex_index *facets);
```

The appearance of a polyhedron is affected by the same attributes that affect the fill-area primitive.

5.6 rendering modes

SPHIGS currently supports only two of the rendering modes described in the textbook. The rendering mode is set by the following function:

`SPH_setRenderingMode (int value)` *WIREFRAME_RAW, WIREFRAME (default), LIT_FLAT*

The raw wireframe mode renders without performing hidden-surface elimination or clipping. Because of this, strange results (and lots of warning messages) occur if objects lie behind the camera. The default mode (WIREFRAME) performs clipping and hidden-surface elimination, but only draws edges. The most expensive mode draws surfaces as filled polygons whose colors are determined by lighting/shading algorithms. (See section 4 of this reference manual for a complete description.)

```
void SPH_setViewRenderAlgorithm (int viewIndex, int algorithm)
```

The current version of SPHIGS uses a binary-space partitioning algorithm to render three-dimensional scenes. This method is slower, but considerably more accurate, than the old version's quick-and-dirty Z-sort algorithm. If your application is interactive and uses fairly simple models, you can increase rendering speed by changing the viewport's rendering engine to `RENDER_PAINTERS`. To guarantee more accurate rendering, use `RENDER_BSP` (the default).

NOTE: this routine is provided simply as a matter of convenience. Multiple rendering algorithms will NOT be supported in future versions of SPHIGS and writing applications which depend on this particular implementation is discouraged. Also note that rendering accuracy is only guaranteed with BSP rendering; bugs in the `RENDER_PAINTERS` mode are numerous, so use it at your own risk.

5.7 execution of subordinate structures

This function inserts a structure-execution element into the currently open structure. The structure referenced by the argument need not already be defined, as explained previously, for all unknown structures are implied empty structures.

`void SPH_executeStructure (int structureIndex)` ‡

6 Dynamic Invisibility and Highlighting

During traversal, an attribute called the *name set* is maintained. This set may contain any integers that lie between 0 and `MAX_NAME`. The following elements add and delete names from the name set during traversal:

`void SPH_addToNameSet (name)` ‡ *you can only add one name at a time*
`void SPH_removeFromNameSet (name)` ‡ *you can only delete one name at a time*

Note that the scope of a name set change is local to any given structure; any call to `SPH_addToNameSet` or `SPH_removeFromNameSet` is passed along to a child (through a call to SPH_execute), but is not carried back to a parent structure.

There is a filter for each viewport that controls visibility during traversal (highlighting is not yet supported). A filter is simply a set of names: e.g., any primitives whose name set contains X will not be visible if the invisibility filter contains X. These filters are not controlled by elements in the mode; rather, application sets them whenever it wants to modify the visibility state of any part of the model. When any filter is changed, the screen is updated to reflect the new state.

```
void SPH_addToInvisFilter (name)
void SPH_removeFromInvisFilter (name)
```

7 Input

7.1 control of input devices

The locator measure is similar to that of the SRGP locator, but the position is a 3D NPC coordinate (with $z = 0$), and the index of the non-empty view (i.e., having at least one posted root) with the highest priority (i.e. index) that encloses the position is also returned:

```
typedef struct {
    point position;
    int   view_index;
    int   button_chord[3];
    int   button_of_last_transition;
} locator_measure;
```

Control of the input devices is performed a la SRGP. Any positions are 3D NPC points (z ignored) instead of 2D PDC points.

```
#define SPH_setInputMode SRGP_setInputMode
#define SPH_waitEvent SRGP_waitEvent
#define SPH_getKeyboard SRGP_getKeyboard
#define SPH_sampleKeyboard SRGP_sampleKeyboard
#define SPH_setKeyboardProcessingMode SRGP_setKeyboardProcessingMode
#define SPH_setKeyboardEchoColor SRGP_setKeyboardEchoColor
#define SPH_setKeyboardEchoFont SRGP_setKeyboardEchoFont
#define SPH_setKeyboardMeasure SRGP_setKeyboardMeasure
#define SPH_setLocatorEchoCursorShape SRGP_setLocatorEchoCursorShape
#define SPH_setLocatorButtonMask SRGP_setLocatorButtonMask

void SPH_getLocator (locator_measure*);
void SPH_sampleLocator (locator_measure*);
void SPH_setLocatorMeasure (point position);
void SPH_setKeyboardEchoOrigin (point position);
```

7.2 pick correlation

The NPC point returned from the locator can be sent to the pick-correlation utility:

```
void SPH_pickCorrelate (point npc_position, int view_index,
pickInformation *pickinfo);
```

The information is returned in a variable of type `pickInformation`:

```
typedef
   struct {
     int structureID;
     int elementIndex;
     int elementType;                    symbolic constants in elementType.h
     int pickID;
   } pickPathItem;
typedef pickPathItem pickPath[MAX_HIERARCHY_LEVEL];
typedef struct {
     int pickLevel;
     pickPath path;
   } pickInformation;
```

When no primitive is close enough to the cursor position, the value of the pickLevel field is returned as 0. When pickLevel is greater than zero, it specifies the length of the path from the root to the picked primitive. In this latter case, elements [0] through [pickLevel-1] of the path array return the identification of the structure elements involved in the path. Of course, elements [0] through [pickLevel-2] will always be structure-execution elements, whereas the [pickLevel-1] element will always be an output primitive. For each element, the information presented is: its ID, its index within its parent structure, a code presenting the element's type (e.g., ELTYP__EXECUTE_STRUCTURE, ELTYP__POLYHEDRON), and the pick ID of the element.

The pick ID is an attribute that is local to a structure (not inherited). For this reason, the application can define different pick identifiers for each level of hierarchy. It begins with the value 0, and is changed by elements created via:

```
SPH_setPickIdentifier (int pick_ID);    ‡
```

8 Control of Table Sizes

The tables that store views and structures have default sizes that in many cases are acceptable. You may, however, choose to reduce the size of a table to save memory (if you're working on a Mac Plus, for instance) or increase the size of a table if you need more entries. You must change the size of a table *before* you initialize SPHIGS!

```
void SPH_setMaxStructureID (int);
void SPH_setMaxViewIndex (int);
void SPH_setMaxNameID (int);
void SPH_setMaxLightSourceIndex (int);
```

See '`sphigs.h`' for the defaults for these sizes.