3D MODELER IMPLEMENTATION MANUAL

Jeff Butterworth (butterwo@cs.unc.edu)
Drew Davidson (davidson@cs.unc.edu)
Steve Hench (hench@cs.unc.edu)
Marc Olano (olano@cs.unc.edu)

The University of North Carolina at Chapel Hill
Spring, 1991
COMP 145

TABLE OF CONTENTS

3D Modeler Implementation Manual....1
Table of Contents....1
1. General Design Structure....1
1.1. Program Layers....1
1.2. Control Through Function Tables....1
1.3. Database....2
1.4. The Toolbox....3
1.5. Files....4
2. Specifics....5
2.1. The Interface with Vlib and Trackerlib....5
2.2. Event Handler....5
2.3. Database Manipulation....6
2.3.1. Function Organization....6
2.3.2. Groups....7
2.3.3. Geometry....7
2.3.5. The Marked Set....8
2.4. Database Implementation....10
2.4.1. Data Structures....10
2.4.2. Find Nearest Vertex....11
2.5. Undo/Redo....12
2.6. Toolbox....14
2.6.1. Icons....15
2.6.2. Transformations....15
2.6.3. The Floor and Magic Carpet....16
2.6.4. Dragging the Toolbox....16
2.6.5. Toolbox Help....17
2.6.6. Toolbox Utilities....17
2.7. Tools....17
2.7.1. Fly....17
2.7.2. Scale World Tool....18
2.7.3. Rotate World Tool....18
2.7.4. Mark Tool....18
2.7.4. Move Tool....18
2.7.5. Triangle Creation Tool....18
2.7.6. Surface of Extrusion Tool....19
2.7.7. How To Make New Tools....19
2.8. Commands....23
2.8.1. Group Command....23
2.8.2. Ungroup Command....23
2.8.3. Parent Command....23
2.8.4. Merge Command....23
2.8.5. Color Command....23
2.8.6. Default View Command....24

2.8.7. Floor On/Off Command 24
2.8.8. Magic Carpet On/Off Command 24
2.8.8. Unmark All 24
2.8.9. How To Make New Commands 24
Appendix 1 — Error Messages 27
Appendix 2 — File Format 30
2.1. Format Description 30
2.2. Sample File 31
Index 32

1. GENERAL DESIGN STRUCTURE

1.1. PROGRAM LAYERS

The structure of the program is divided into three levels (Fig 1). The center layer contains the tools and commands actually used to create the models. The top layer provides an interface between the tools and the user. This layer is implemented as an event handler, calling callback functions in the appropriate tool in response to the user's actions. The bottom layer performs the actual database manipulations. The tools communicate to this layer through a set of function calls to create, delete, or modify portions of the geometry database.

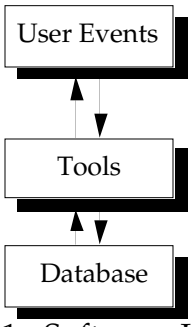

Figure 1. Software Layers

1.2. CONTROL THROUGH FUNCTION TABLES

For flexibility and speed, the user event code and toolbox are controlled by tables of functions (Fig 2). The main loop of the program is an event handler that keeps track of mouse motion and button clicks and uses that information to call functions for the current tool. The toolbox uses these tables of functions to turn tools on and off and to activate commands. Even though both tools and commands are found in the toolbox, the differences in operation of tools and commands lead to separate tables for each. Tools need functions to do set-up or clean-up work, and event functions for each possible mouse action. Commands, on the other hand, only need a single function to activate them because their work is accomplished in a single action. There is also a crossover type, the *long commands* for commands that need to receive some limited amount of input, before their task is finished. One example of a long command is the color selector. Once a color has been selected, there is no need to remain in color selection mode. The long commands are implemented as tools that turns themselves off when their work is done. The toolbox knows which tools are actually long commands, and reverts to the previous mode when the long command is done. Included with the function lists is other information for use by the event handler or toolbox, like the PPHIGS structure name for the toolbox icon, the cursor shape, and the sound identifier for the spoken name of the tool.

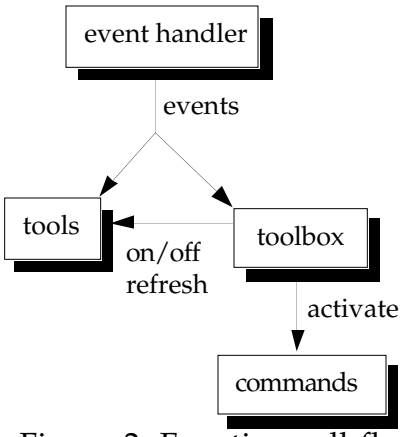

Figure 2: Function call flow

The undo and redo capability is also handled by indirect functions. Implementation of undo and redo revolve around an undo stack. Each operation that modifies the database or does any non-trivial action puts an undo record on top of the undo stack. This record contains pointers to the undo and redo functions and whatever data is needed to undo the operation. The undo command in the toolbox calls the undo function for one or more records at the top of the stack. Records are not deleted until a new operation is performed so the same data is used by the redo function to recreate the change.

1.3. DATABASE CONCEPTS

The model database is conceptually similar to the database used by PPHIGS. Like the PPHIGS database, it is a hierarchical graph made up of objects containing geometry and references to other objects. The implementation of the graph is discussed later, but for now a description of how to use the database facilities is described from the point of view of a programmer creating a tool for the 3D Modeler. An attempt to enumerate every database function or to supply exact type declarations will not be made in this document. Instead, the basic concepts and algorithms used to design the database and its associated functions will be explained.

As much as possible, details of the database implementation are hidden from the tool programmer. All manipulations and queries of the database are handled through function calls rather than by direct manipulation of low level data structures. An entity that helps with this data encapsulation is the *handle*. In this context, a handle is a pointer to a pointer to some data. The use of two levels of referencing gives the database functions the ability to move data around in memory without the knowledge of the tool programmer.

Most of the interaction with PPHIGS is taken care of by the model database functions. The reason for this is that the model database must remain synchronized with the PPHIGS display list database that is used to actually view the objects. Thus, many database functions will edit two databases at once. For instance, the Create_Triangle() function will insert a triangle into the model database, but it will also add a triangle to a PPHIGS structure.

The basic entity in the model database is the *group* (Fig 3). A group contains some geometric primitives and information about how they are interconnected. Currently, this geometry consists only of a triangle mesh. In addition, a group can contain *instances* of

other groups. An instance is essentially a copy of a group by way of a pointer to the group and a transformation describing where the copy is placed in the local coordinate system. A graphical representation of a group is shown below.

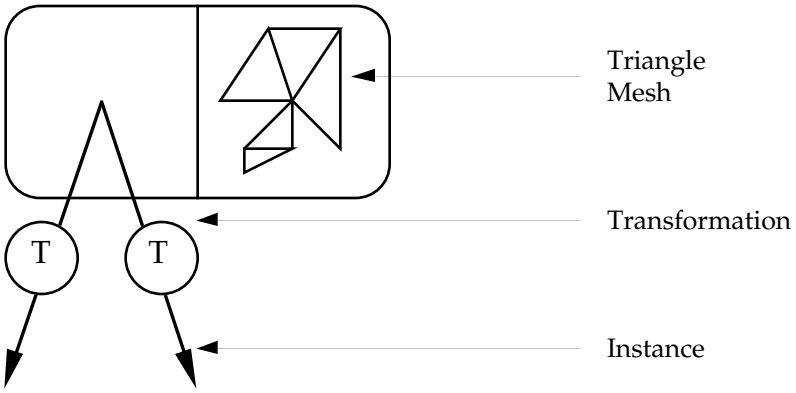

Figure 3. A Generic Group

1.4. THE TOOLBOX

The toolbox is a tray containing icons for all the actions that can be performed in the modeler. These can be divided into three classes. Tools represent some mode of the program, like marking or triangle creation, and remain active from the time they are chosen until the tool is changed. Commands have instantaneous actions, for example load or undo. The use of a command doesn't change the current tool. Long commands are similar to tools because they remain active after they have been selected, but are also similar to commands because once their job is done they revert to the current tool.

A tool must provide one or more routines that are called by the `dispatch_events()` function in response to events. The `init` event is an exception; it is sent only once at the beginning of the program. Most tools respond to the `init` event by doing any one-time-only initialization, such as creating any temporary object structures it might need. Tools generally respond to the `on` event by calling vlib to post its temporary world objects, if it has any. Also, any necessary tool data structures are initialized. Tools respond to the `off` event by removing any temporary world objects it has created and added to the world. Tools respond to the `move` and `always` eventsby doing any actions that need to be updated each frame, such as animation and rubber-banding.

Tools respond to the left and right presss, releases, clicks, and drags in variable ways; it depends on the tool. Most tools modify internal data structures in response to these events, or call database routines to add or modify the database.

Long commands provide the same `init` function and event functions that a tool would. Commands, on the other hand, don't need to know about user events, so they only need to supply an `init` function and a function to run when they are activated.

To be compatible with undo and redo, all tools and commands must store any necessary information needed to reverse their action every time an undoable action is completed. Routines must be written which accomplish the undo and redo based on the stored information. These routines are called automatically in response to an undo or redo command.

1.5. FILES

To allow the greatest usefulness for the machines at UNC, 3DM uses a file format based on the ascii PPHIGS archive files used by the Pixel-Planes machines. The format is a limited subset of the full pphigs ascii format, and is described in detail in Appendix 2. The subset of ascii archive format used by the modeler can be read by both Pixel-Planes 4 and Pixel-Planes 5, and includes support for object hierarchy not found in most object file formats. The files are parsed by a yacc-parser based on the one used by the Pixel-Planes makearch utility. To allow loading of multiple files, the lexical scanner and parser are created with the GNU flex and bison programs. These programs are upwardly compatible with the standard Unix lex and yacc, but include extensions to easily have the parser read a file without having to reroute `stdin`, and to restart the parser for a new file.

2. SPECIFICS

2.1. THE INTERFACE WITH VLIB AND TRACKERLIB

3dm uses vlib and trackerlib to handle the interface to hardware devices (that is, the displays, the trackers, and the mouse buttons). Vlib greatly aids in the handling of the world, room, tracker, head, hand, eye, and screen coordinate systems. It handles stereo for stereo head-mounted displays. Vlib also makes 3dm very device-independent.

Calls are made to initialize vlib and trackerlib when the program begins. Calls are made to trackerlib each time through the event loop to read the tracker and to vlib to update the head and hand positions, and thus the views of the virtual world. When objects move, calls are made to vlib to handle these transformations as well. See Transformations, below, for a diagram of the vlib transformation tree. The toolbox, floor, magic carpet, and model are defined relative to the objects in this tree.

2.2. EVENT HANDLER

The event handler in the `dispatch_events()` function is really the main loop of the program. It reads the mouse input, updates the display, and calls the appropriate functions for the current tool. There are some problems with using an event handler that takes complete control of execution, but we decided to go with that model anyway for two reasons. First, it takes some of the load off the other parts of the program. Second, most of the problems occur when trying to react to events for which callback functions haven't been provided; since the event handler is our own code, we can easily add these events anyway. Before calling the event functions, the event handler calls the `check_tracker()` function provided by the toolbox. If this function reports that the mouse is in the toolbox, the toolbox event functions are called instead of the current tool's. The `update()` function draws a frame based on the current mouse and head position. This function can be called during particularly long actions to keep the display from freezing.

The event handler has been designed to sometimes call several functions on one pass through the event loop. This allows the tools decide what level of event parsing it wants to see by reacting to some of these events and ignoring others. For example, a tool could work on the high level of click events (press and release of a button in the same place), the lower level of the actual button press and release actions, or the lowest level called each time through the event loop. If multiple events are called, they are called in the order they appear in the function structure. The lowest level event the event manager currently provides is `always` which is called each time through the event loop. The next level of events are the movement events, `move`, called when the mouse is moved with no buttons down, and left and right drag, called when the mouse is moved with the left or right button held down. The drag events are called every event loop from the time it is first determined that the user's action is not going to be a click (determined by the constants `NOT_MOVED_LIMIT` and `NOT_ROTATED_LIMIT`), until the button is released. Next are left and right press and release called on button transitions, then the left and right click functions called when a the press and release for a button happen in the same place. The average tool is expected to ignore most of these by using the `null_event()` function.

Element in the tool function list:

```
typedef struct {
  NameType cursor;              /* pphigs name for current mouse shape */
  char *sound;                  /* spoken name of tool */
  IconFunctPtr icon;            /* called to create icon in toolbox */
  InitFunctPtr init;            /* called program initialization */
  SwitchFunctPtr on;            /* called when tool is first activated */
  SwitchFunctPtr off;           /* called when tool is deactivated */
  RefreshFunctPtr refresh;      /* after interruption by command */
  EventFunctPtr left_press;     /* left button pressed */
  EventFunctPtr left_release;   /*             released */
  EventFunctPtr left_click;     /*             quick click */
  EventFunctPtr left_drag;      /*             move while down */
  EventFunctPtr right_press;    /* right button pressed */;
  EventFunctPtr right_release;/*              released */
  EventFunctPtr right_click;    /*             quick click */
  EventFunctPtr right_drag;     /*             moves while down */
  EventFunctPtr move;           /* move with no buttons down */
  EventFunctPtr always;         /* called every frame */
} ToolType;
```

where the function pointers are

```
void (*IconFunctPtr)(p1, p2);
PointType p1, p2;               /* bounding box of icon */

void (*InitFunctPtr)();

void (*SwitchFunctPtr)();

void (*RefreshFunctPtr)(num, which);
int num;                        /* tool or command number */
int which;                      /* TB_LCMD or TB_CMD */

void (*EventFunctPtr)(button, cursor);
v_button_type button;           /* button states */
MatrixType cursor;              /* cursor position and orientation */
```

2.3. DATABASE MANIPULATION

2.3.1. FUNCTION ORGANIZATION

Many of the database editing functions can be placed into three categories. It is important that anyone working on 3dm understand the uses of the different types of functions.

The highest level functions are the ones called by tool programmers. These functions are the ones that are *undoable*. Undoability means that a function stores a record of its database changes in the undo file and it supplies the functions to undo the change. All of this work related to undoing changes is transparent to the tool programmer. A typical example is the undoable version of triangle deletion, which is called `Delete_Triangle()`. Whenever a triangle is deleted with this function, the programmer can be sure that next time the undo command is used, the triangle will be re-

inserted into the database. In general, an undoable function calls its non-undoable counterpart to actually perform the database operation.

The next lower level of functions do not support the undo operation. These functions simply perform their operation on the database. An example of the triangle deletion function is `Delete_Triangle_NoUndo()`. This level of function may actually perform many operations on the database in order to assure consistency. For instance, the triangle deletion function not only deletes the triangle, but it also deletes any vertices and edges that may become stranded because of the deletion. To perform its work, this level of function usually calls the lowest level function that actually edits the database structures.

The lowest level of database functions actually manipulate the data structures that hold the database information. These functions perform very little consistency checks, and because of this, they should almost never be used directly by a tool. The lowest level function for triangle deletion is `Reclaim_Triangle_Space()`. This function reclaims the space in the database that the triangle was using, but it does not check to see if any vertices were left stranded.

Although three levels of database functions have been described here, the fact that the database functions were created as they were needed has caused multiple levels to be occasionally compressed into one. For instance, `Merge_Triangle()` is used throughout the program to create undoable triangles, but it encompasses the first *two* levels described above. `Merge_Triangle`() is undoable or not based on a parameter that can be set to UNDO or NO_UNDO. The low level function that actually puts the triangle in the database is `Add_Triangle()`.

2.3.2. GROUPS

The most important entity in the database is the group and many functions are supplied that manipulate it. A few examples are shown below.

A new object can be created with the function `Create_Group()`. The only parameter to this function is a string that uniquely names the group. This name is not used by the tool programmer, rather it becomes the PPHIGS name of the group and can later be used by a PPHIGS application programmer to refer to the group. The call to `Create_Group()` returns a *group handle*, which is used by the tool programmer in all future operations on the group.

A typical use of a group handle is in adding an instance of one group to another group. The function `Create_Instance()` is passed a 3D transformation and two group handles, the first being the group that will have an instance added to it, and the second being the group to be instanced. The transformation describes where the instance of the child will be placed in the local coordinate system of the parent. What is returned is an *instance handle*, which gives the programmer access to the instance so that it can be manipulated at a future time.

Other operations, such as deletion, locking, and copying can be performed on groups through various model database functions.

2.3.3. GEOMETRY

Many of the operations that the tool programmer will want to perform involve altering the geometry component of a group. Many different methods are supplied for creating geometric primitives. A few are described below.

At the lowest level, a vertex can be created with a call to the function `Add_Vertex()`. The call to `Add_Vertex()` is passed a 3D coordinate and a handle to the group that it will be part of. What is returned is a *vertex handle*. The programmer should never need to know about the internals of a vertex handle, instead this handle will be passed to other database functions in order to operate on the vertex.

An example of the use of a vertex handle is one of the methods for creating a triangle. The function `Add_Triangle()` is passed three vertex handles and the handle of the group the triangle will be part of. The result is that connections will be made in the triangle mesh of the group in order to connect the vertices into a triangle.

The most commonly used method for creating triangles is the `Merge_Triangle`() function. `Merge_Triangle`() is passed three points. It differs from `Add_Triangle`() in that it uses points from the database if they already exist. `Merge_Triangle`() also has the property that it supports the undo/redo operations, which `Add_Triangle`() does not. Unless very special work is being done, `Merge_Triangle`() should be the only function used to create triangles because of it's undo/redo capabilities.

Other operations, such as deletion, movement, and copying can be performed on the geometry of a group through various database functions.

2.3.5. THE MARKED SET

The marked set is part of the model database that has been marked in order to perform operations selectively on a subset of the database. Functions exist for marking pieces of geometry or instances of groups, for unmarking them, and for traversing the hierarchy of marked entities.

At the lowest level are functions to display marked vertices and instances via calls to PPHIGS. These functions begin with the prefixes `Set` and `Clear` and simply update the PPHIGS structures that hold all the pyramids and bounding boxes. It would be too inefficient to have separate execute structure calls for turning each mark on and off, so the primitives for the marks are all kept in two world level structures: one for vertices and one for instances.

Since PPHIGS doesn't support deletion of an element with a structure, when a vertex or instance is unmarked the corresponding primitives are cleared so that they're no longer visible. The triangles for the pyramids were initially set so that all coordinates were 0.0, but this resulted in intermittent flashes. Currently the triangles are shrunk to a centimeter and moved over 10 kilometers away; a smaller triangle at a greater distance still resulted in flashes. Moving to PPHIGS on Pixel-Planes 5 should solve this problem since the new PPHIGS is supposed to support deletion of elements.

The positions of these empty slots are stored on a stack so that when the next object is marked the cleared elements can be reused. The next level of functions controls the display of the marked set, handling the holes and deciding where the display primitives should be placed and then calling `Set` or `Clear`. These functions have the prefixes `Display` and `Hide`.

At the highest level is a set of functions that mark and unmark existing objects. These mark the objects in the database and call the `Display` and `Hide` functions. They also use connectivity to determine if objects other than those specifically being marked need to be marked as well. For example, when all three vertices of a triangle become marked the triangle should be marked as well. A counter is kept for each triangle with the number of

marked vertices. When a vertex is marked and the count becomes three, the triangle marking function is called.

When an object in the database is marked it's also important to remember which instance of the object was marked. When a triangle is marked, for example, the path along instances through the acyclic graph is stored in the group containing the triangle. Similarly the path to a marked instance is stored in the group containing the instance. It is worth noting that the path is always stored in the parent group and contains only the instances to get to the group. Thus the path to an instance does not contain the instance itself. The path to a group is stored in the group itself, and all the instances traversed to get to it. `ChangeTraversal()` is provided to compare the current path with a stored one. If the two are different it updates the path and returns 1, otherwise it returns 0. This return code can then be used in the logic to decide exactly what should be marked.

In addition to the marking and unmarking there are a number of routines to support queries and other common tasks. The most important of these is the one that traverses the marked set. `TraverseMarked()` is used by almost every other tool in order for them to perform their specific operation on only the marked set. The function takes as one parameter a set of flags describing what objects the caller is interested in. A second parameter specifies a function to be called for each such object encountered. The passed function is called a **callback** and accepts parameters specifying which object is being reported, what type it is, and how it was reached.

`TraverseMarked()` recursively traverses the database and looks for marked objects that match one of the types requested. For each such object found the callback is called and passed a pointer or handle to the object. The following call traverses the marked set and reports vertices and instances:

```
TraverseMarked(VERTEX | INSTANCE, Callback)
```

As TraverseMarked descends the database it uses a stack to keep track of what path it has followed to that point. When it finds an object of a type requested it passes this path to the callback. The path structure contains the path of instances to the vertex, as well as the composite transformations to convert between the world and each of the groups along the path. This structure is shown below. The vertex parameter is unused by `TraverseMarked()`, but is included for use by `Find_Nearest()`.

```
typedef struct
{
  int                numInstances;   /* how many instances in the path*/
  Instance_Handle    *instances;     /* instances along path */
  MatrixType         *composites;    /* composites for each instance */
  MatrixType         *inverts;       /* inverses for each instance */
  Vertex_Handle      vertex;         /* vertex used to pick the path */
} Path;
```

Each callback is declared as follows. The `which` parameter tells the callback which type of object is being reported and which of the five pointers should be examined. This example simply prints the type of each object encountered.

```
void Callback(which, vHandle, eHandle, tHandle, g, iHandle, path)
  int                   which;
  Vertex_Handle         *vHandle;
  Edge_Handle           *eHandle;
  Triangle_Handle       *tHandle;
  Group                 *g;
  Instance_Handle       *iHandle;
  Path                  *path;
{
  switch(which)
  {
    case VERTEX:        printf("Vertex\n");     break;
    case EDGE:          printf("Edge\n");       break;
    case TRIANGLE:      printf("Triangle\n");   break;
    case GROUP:         printf("Group\n");      break;
    case INSTANCE:      printf("Instance\n");   break;
  }
}
```

Several common callbacks already exist. `SetMarkCallback()` call the `Set` functions mentioned above to redisplay marks after they may have changed. `UnmarkCallback` unmarks each object reported to it, and is the heart of the Unmark command. `TransformCallback()` is used in conjunction with `TransformMarked()` to support moving and scaling marked objects.

Besides marked set traversal, there are a couple other functions that have not been mentioned. `MarkOn()` and `MarkOff()` control whether the marks are to be displayed, and `MarkMany()` marks all vertices within a specified bounding box.

2.4. DATABASE IMPLEMENTATION

2.4.1. DATA STRUCTURES

In the interest of clarity, only some of the most important data structures for the model database are described in this section rather than explaining every "typedef" in the program.

The group structure appears fairly compact because large amounts of information it can contain are hidden in the geometry and children fields. Part of the group structure is shown below.

```
struct group_struct
{
  NameType      name;           /* pphigs structure name for group */
  Geometry      geometry;       /* all primitives in this group */
  ...
  Instance      *children;      /* child groups of this group */
  ...
  int           flags;          /* includes things like GROUP_MARKED */
};
```

The instance structure contains not only the transformation and pointer to another group, but it also has pointers into the PPHIGS database for these same items. The reason

why the PPHIGS pointers are maintained here is because the database functions need to keep the model database and the PPHIGS database synchronized. Part of the object instance structure is shown below.

```
typedef struct
{
  MatrixType    transform;      /* transformation to child */
  DlistAddress  pp_xform;       /* pphigs handle to transformation */
  Group         *group;         /* child group */
  DlistAddress  pp_object;      /* pphigs handle to group */
  ...
} Instance;
```

The geometry structure contains all of the information needed to describe a triangle mesh. The fields associated with triangles will be used to illustrate how dynamic arrays are implemented in 3dm. The pointer `triangles` points to a one dimensional array of Triangle structures. There is enough memory allocated in the triangle array for triangle_space entries. `Num_triangles` is the highest possible index of a triangle in the array. If no deletions occur, then num_triangles is incremented each time a new triangle is inserted. New space is allocated for the array in chunks in order to avoid a call to realloc() each time a new triangle is added. Whenever new space is allocated, triangle_space is increased accordingly. This method of memory allocation makes most triangle creations very quick. When a deletion occurs, `num_triangle_holes` is incremented and the index of the deleted triangle is pushed onto the end of the triangle_holes array. During a deletion, `num_triangles` is **not** changed! Thus, `num_triangles` can never decrease and the real "number of triangles" is `num_triangles-num_triangle_holes`. The reason for this deletion scheme is that it is very rapid. The `_order` arrays are explained in the next section. Part of the geometry structure is shown below.

```
typedef struct
{
  Vertex   *vertices;       /* unordered array of vertices */
  int      *x_order;        /* array of vertex indices ordered on x */
  int      *y_order;        /* array of vertex indices ordered on y */
  int      *z_order;        /* array of vertex indices ordered on z */
  ...
  Edge     *edges;          /* array of edges */
  ...
  int      triangle_space; /* slots available for triangles */
  int      num_triangles;  /* slots that have been used for triangles */
  Triangle *triangles;     /* unordered array of triangles */
  int      num_triangle_holes; /* number of holes in triangle array */
  int      *triangle_holes; /* stack of holes in triangle array */
} Geometry;
```

2.4.2. FIND NEAREST VERTEX

This database function was one of the most difficult to implement because it is one of the most time critical operations. Every time the cursor is moved this function must be called in order to determine which nearby vertex to highlight. There are many algorithms

that solve the problem of finding the nearest vertex to a given point. The requirement for our algorithm was that this "find nearest" operation had to be very fast, but insertions and deletions of vertices had to be at least reasonable.

The algorithm that was implemented uses four arrays. The first array is a simple unordered list of vertices. The other three arrays contain indices into the first array and are ordered based on the X, Y, and Z components of the vertices. To insert or delete a vertex requires linear time with this data structure because large pieces of the arrays may need to be copied to make room or to fill in gaps.

The "find nearest" operation takes logarithmic time in most situations. This operation begins by finding the range of vertices that are near the test point in the X dimension. Finding this set takes logarithmic time because two binary searches are used on the x_order array to find the bounding vertices of this range. This same operation is then performed in the Y and Z arrays. Finally, these three small sets are intersected in order to find only the vertices that fall within a small box around the test point. This small set is then tested to find the nearest point to the test point. Because the "nearby" set is usually small, the dominant complexity term is from the binary searches, so the speed of this algorithm is logarithmic.

The routine that performs this search within a group is `Find_Nearest_In_Geometry()`, but it isn't used directly by the higher level tools. Rather it is used once for each group during a traversal of the database. The result of this traversal is the vertex that is closest to a probe point and the path through the database that leads to it. The function that performs this search is `Find_Nearest()`. At an even higher level is the function `TrackNearest()`, which highlights the nearest vertex in the model and returns the information it receives from `Find_Nearest()`.

2.5. UNDO/REDO

As mentioned before, any action that the user is likely to want to undo will have appropriate undo and redo functions. When the action is done, pointers to undo and redo functions and the data that they will use are placed on a stack. The appropriate undo or redo action can later be taken by reading the function pointer and data from the stack and calling the undo or redo function (Fig 4). Since we would like to have the freedom of never limiting the number of undo operations that can be done, the undo stack is stored in a file in the `/tmp` directory. The performance degradation for using a file instead of memory presents no problem because the undo stack is only accessed for single undo, or redo operations where it does not matter if they don't happen immediately, or at the completion of an action, but never in the middle of a time-critical interaction.

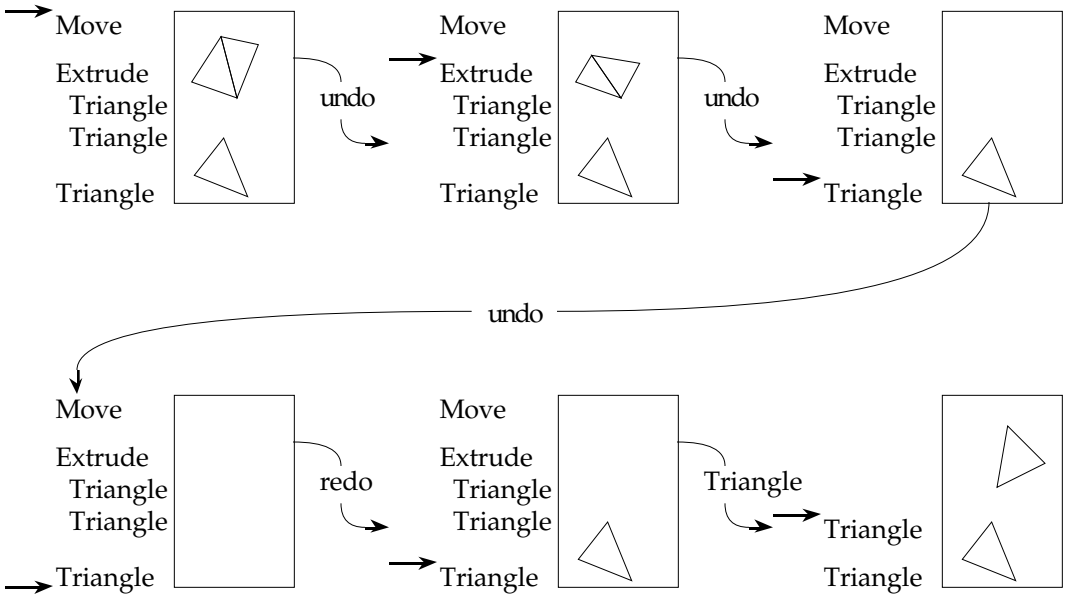

Figure 2. Operation of the Undo Stack

For the undo system to function correctly, any tool or command that changes the state of the database or marked set **must** support undo and redo. Fortunately, most of the work is taken up by the database manipulation functions themselves. It is the tool's responsibility to call `undo_start()` before any action or sets of actions that might need to be undone.

```
void undo_start(icon, code);
NameType icon;
int code;
```

The icon parameter is a copy of the toolbox icon for the tool or command. This structure is used to create the changing undo and redo icons to reflect what action will next be undone. All actions between calls to `undo_start()`, or between `undo_start()` and an actual use of undo are lumped into a single use of the undo command. The actual data used to perform the undo and redo operations is saved with calls to `undo_data()`. Along with each block of data, pointers to functions to use that data to undo and redo are given. Thus, the load command can use calls to low-level triangle and group creation utilities that each add a little extra data to the stack, and a single press of undo or redo will call the whole series of these functions in order to remove or replace the load. Figure 2 shows the changing of the undo stack through a series of undo and redo operations.

```
void undo_data(undo_func, redo_func, len, data);
UndoFunctPtr undo_func;
RedoFunctPtr redo_func;
int len;
void *data;
```

where the function pointer types are:

```
void (*UndoFunctPtr)(len, data);
void (*RedoFunctPtr)(len, data);
int len;
void *data;
```

The code parameter of `undo_start()` along with the calls `undo_check()`, and `undo_start_or_add()` are used for any tool that wants to save up tiny undo steps over several user actions to all be undone with a single use of the undo command. `undo_check()` returns the code of the currently active `undo_start()` if there is one. `undo_start_or_add()` serves the same function as `undo_add()` except if the code does not match, it also performs an `undo_start()` to make sure the actions will be grouped correctly.

```
int undo_check();

void undo_start_or_add(icon, code, undo_func, redo_func, len, data);
NameType icon;
int code;
UndoFunctPtr undo_func;
RedoFunctPtr redo_func;
int len;
void *data;
```

The file the undo data is stored in is called `/tmp/3dm.undo`. This file is not visible in the `/tmp` directory for long because it is unlinked from the directory immediately after it is opened. This does not keep the 3dm program from using the file, but forces the disk space it uses to be reclaimed as soon as the program exits. If after an operating system change, undo ceases to work, this is probably the cause. Currently this effect for unlink is documented at least under Ultrix, SunOS, and BSD Unix. This file is created, and all other undo/redo initialization is done by the `init` function for the undo toolbox command.

Undo and redo are activated by the toolbox with calls to their command `activate` functions. If an error occurs while attempting to redo, the redo request is ignored because it may still be possible to use the stored undo information. If an error occurs while trying to undo, the undo file is reset, so undo information before the current state is lost, but undo's may still be possible. If an error occurs while trying to write data to the undo file, the undo system is turned off, allowing continued use of the program but preventing repeated error messages at every undoable action.

2.6. TOOLBOX

The toolbox functions in effect as an ordinary tool, responding to events like any normal tool. Each time through the event loop, a check is made to see if the mouse is

inside the toolbox. If so, events are directed to the toolbox instead of the current tool. The toolbox tracks the mouse and mouse clicks and sends the appropriate off and on events to the appropriate tool when the user clicks in the icon of a tool other than the current tool. Also, the hand object is changed to be the current tool's hand object. The toolbox structure is modified so that the current tool icon is highlighted and the previous tool icon is not. If a user clicks in a command, the command routine, stored in a global table of command routines, is called.

2.6.1. ICONS

The icons in the toolbox are defined by routines each tool or command must supply. Routines were used instead of structure names to avoid unneeded transformations—the icons are defined in place instead of being defined separately and executed in the toolbox structure. A tool or commands `icon` routine must draw its icon between the points it is passed. When the `icon` routine is called, a PPHIGS structure is open and being defined. If the icon is to change during the course of the program, these points must be saved and space allocated for the structures to change. The structures to change must be defined in the `init` routine. Note that the `icon` routine is called before the `init` routine, but it is not an error to use the name of an undefined structure as long as it is defined before the next `pg_draw_frame()`. See the Floor On/Off tool for an example of a command that changes its icon.

2.6.2. TRANSFORMATIONS

The toolbox starts locked to the head. This is implemented by adding the toolbox structure to the head structure. The head structure starts as an empty structure created in the `init_vlib`() routine in `main.c`. Space for an execute structure call is saved in the room structure, so that we may add the toolbox to the room when it is unlocked. Adding the toolbox to the head and room structures means it will **not** be flown about, scaled, and rotated by the Fly, Scale World, and Rotate World tools. We save the `DlistAddress` of the execute structure calls, so that we can hide the toolbox later.

The toolbox structure has two xforms in it, `tb_anti_rotation` and `tb_toolbox_xform`, in that order. When the toolbox is locked, `tb_anti_rotation` is updated every frame with a call to `update_toolbox()` to prevent the toolbox from following the user's head rotationally. When the toolbox is locked, `tb_toolbox_xform` contains the `head_from_toolbox` transformation; that is, the coordinates of the toolbox in head space.

When the toolbox is unlocked, `tb_anti_rotation` is set to the identity matrix and `tb_toolbox_xform` is set to the current room_from_toolbox transformation; that is, the coordinates of the toolbox in room space. The toolbox is also removed from the head structure and added to the room structure. When the toolbox is relocked, it is removed from the room structure and added to the head structure (Fig 5).

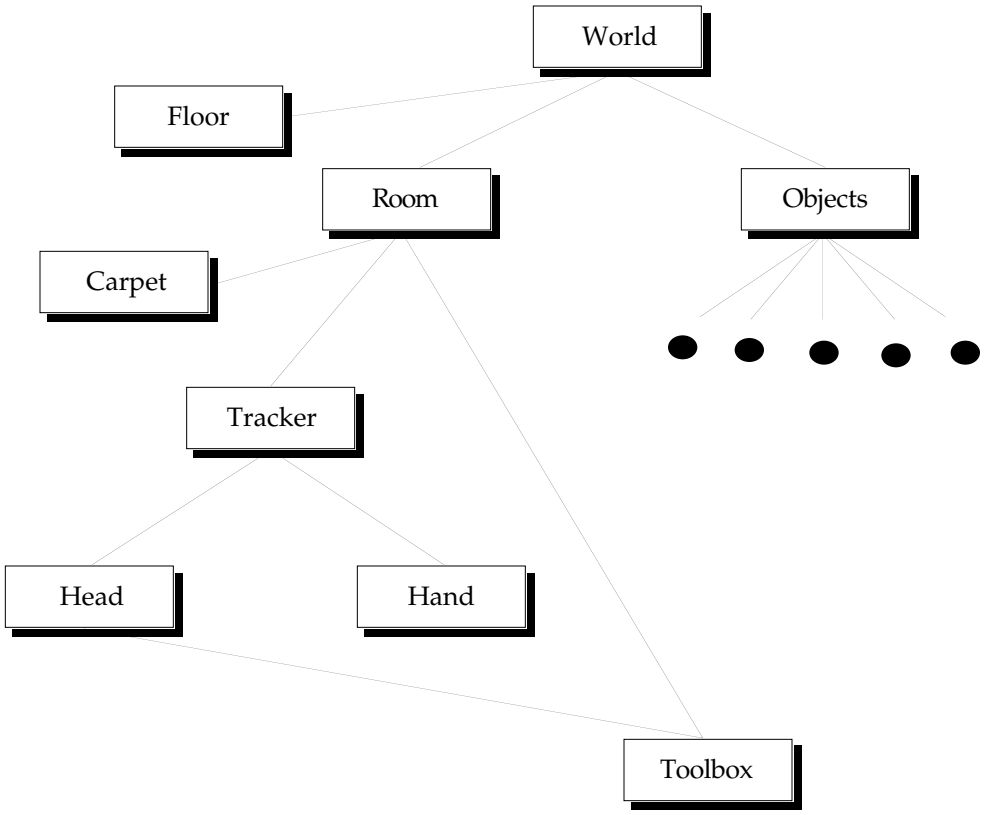

Figure 5. Locations of Toolbox, Floor, Carpet, and Objects in vlib transformation hierarchy

2.6.3. THE FLOOR AND MAGIC CARPET

The floor and magic carpet are two additional structures that make it easier to find your way around the world. They are created by the toolbox, and added

The floor is added to the world structure (Fig 5). The world structure starts as an empty structure created in the `init_vlib()` routine in `main.c`. Adding the floor to the world structure means it will be flown about, scaled, and rotated by the Fly, Scale World, and Rotate World tools respectively. We save the DlistAddress of the execute structure call, so that we can hide the floor later.

The magic carpet is added to the tracker structure (Fig 5). The tracker structure starts as an empty structure created in the `init_vlib()` routine in `main.c`. Adding the magic carpet to the tracker structure means it will **not** be flown about, scaled, and rotated by the Fly, Scale World, and Rotate World tools. We save the DlistAddress of the execute structure call, so that we can hide the magic carpet later.

2.6.4. DRAGGING THE TOOLBOX

When the left button is pressed, the hand location is remembered and several state variables are set. If the toolbox is currently locked, we unlock it. Each time we get a left drag event we update the `tb_toolbox_xform` to it's new `room_from_toolbox` position, based on the current hand position and the initial press position.

2.6.5. TOOLBOX HELP

When the right button is pressed in a tool or part of the toolbox, that tool's help sound is played. The help sound is retrieved from in the global `tool_list` variable in `event.h`.

2.6.6. TOOLBOX UTILITIES

Several useful routines are available for use in other tools.

```
void vv_hide_object(the_addr)
DlistAddress the_addr;
```

This routine is an extension to vlib. It hides the structure by replacing it with a null object

```
DlistAddress vv_reserve_execute()
```

This routine is an extension to vlib. It reserves space for an execute structure call by executing a null object and returning the dlist address. It should be called only when defining a structure.

```
void vv_replace_polygon_color(the_addr, new_color)
DlistAddress the_addr;
ColorType* new_color;
```

This routine is an extension to vlib. It replaces the color in the `pg_set_polygon_color` call pointed to by the_addr.

```
void vv_replace_line_color(the_addr, new_color)
DlistAddress the_addr;
ColorType* new_color;
```

This routine is an extension to vlib. It replaces the color in the `pg_set_line_color()` call pointed to by the_addr.

```
void change_cursor(new_cursor_name)
char* new_cursor_name;
```

This routine replaces the current cursor (hand structure) with the structure passed in `new_cursor_name`. The change takes effect immediately (i.e. at the next `pg_draw_frame`).

```
void restore_cursor()
```

This routine sets the current cursor (hand structure) to the default cursor for the current tool. The change takes effect immediately (i.e. at the next `pg_draw_frame`).

2.7. TOOLS

2.7.1. FLY TOOL

The Fly tool updates the transformation `v_tree_xforms[V_ROOM_FROM_WORLD]` each time through the loop. A vector which is computed based on the current pointing direction, the current fly speed, and the button being held down. This vector is added to the transformation, this translating the world based on the fly direction.

Several state variables are maintained to slowly increase the flying speed as the button is held down longer.

2.7.2. SCALE WORLD TOOL

The Scale World tool updates the transformation `v_tree_xforms[V_ROOM_FROM_WORLD]` each time through the loop. The current transformation scaling factor is multiplied by a new factor to scale the world up or down. The `world_scaling` and `inv_world_scaling` global variables are updated as well, so that the marks do not scale up or down.

2.7.3. ROTATE WORLD TOOL

When the left button is initially pressed, the current mouse location is saved. The Rotate World tool updates the transformation `v_tree_xforms[V_ROOM_FROM_WORLD]` each time through the loop, based on the initial press location and the current mouse location. This is the standard technique for grabbing objects. In this case, the world is being grabbed.

2.7.4. MARK TOOL

The Mark tool is basically an interface to the routines for marking and unmarking objects. If the left button is pressed near a vertex, the path to the vertex is scanned from top to bottom to find if anything is already marked or protected. Once it knows how deep it can mark, it calls `ChangeTraversal()` to update the marked path. If something new is marked the appropriate function is called to mark it, otherwise it's unmarked.

A right button press is handled much the same way as a left button press, except that a parent instance is marked rather than the lower lever object.

If the left button is pressed where there is no vertex, a box is dragged between the original press and the current mouse position. When the button is released the `MarkMany()` function is called to mark any vertices inside the box.

2.7.4. MOVE TOOL

The Move tool responds to a left press event by saving the location of the mouse press. This transformation is then used in each drag event to determine the transformation from the press location to the new hand location. `TraverseMarked()` is called each time through the loop to move the marks. When the mouse button is released, `TraverseMarkedEnd()` is called with the final transformation to actually update the database.

The Move tool responds to a right press event by saving the location of the mouse press. This transformation is then used in each drag event to determine the transformation from the press location to the new hand location. The vertical distance between the initial location and the new location is used to compute a new scaling factor. `TraverseMarked()` is called each time through the loop to scale up the marks. When the mouse button is released, `TraverseMarkedEnd()` is called with the final transformation to actually update the database.

2.7.5. TRIANGLE CREATION TOOL

The triangle tool responds to the init event by creating its world objects, which includes three lines (for rubber-banding) and two pyramids (for showing recently used vertices). These lines and pyramids can be made visible and moved with some utility functions located near the beginning of the triangle tool source file. The nearby vertex highlighting is performed by the tracking subsystem, which is not part of this tool.

A state variable is used to keep track of which vertex of the current triangle is being defined. This information is used to make decision such as when to complete a triangle and insert it into the database and when to begin a new triangle.

Every time the mouse moves, the rubber-banding lines, if they are being displayed, are updated to rubber-band to the current mouse position.

2.7.6. SURFACE OF EXTRUSION TOOL

The Surface of Extrusion tool creates a surface along the path extruded by a set of edges. Any arbitrary set of line segments can be used as the base for the extrusion, and they are not restricted to lie along a single curve, be connected, or lie on existing edges in the model. Once the extrusion base is defined, it can be dragged to another location and orientation and a surface will be created connecting corresponding segments in the original and new position of the extrusion base. The leading edge of the extruded surface is left selected as a base for another extrusion step.

When the tool is entered, it looks at all the marked edges in the database. These are used as an initial base for extrusion. Since they may be in completely different groups, the extrusion surface and extrusion base are put in model space. The base can also be created or expanded within the extrusion tool. Each click of the left mouse button creates another edge for the extrusion base with a rubber-band line showing where the edge will be created, while clicks of the right button reset the starting end-point of the rubber-band line without creating the edge. Standard vertex snapping is used during creation of the extrusion base, allowing attachment to the existing model. Temporary edges can also be created in space, and are removed when the extrusion tool is done.

The actual extrusion is done by dragging with the left button held down. Each edge marked with an extrusion flag in the model level of the database is transformed to the new position, and the corresponding segments are connected. All of the selected points in the database are turned off and the leading edge is selected for use in the next extrude. By using selection to indicate the edge for extrusion, it is possible to leave the extrusion tool, move, scale, or modify the leading edge, then reenter the extrusion tool for more work.

2.7.7. HOW TO MAKE NEW TOOLS

A tool must provide one or more routines that are called by the `dispatch_events()` function in response to events. The `init` and `icon` events are exceptions; they are sent only once at the beginning of the program. Most tools respond to the `init` event by doing any one-time-only initialization, such as reserving space in the dlist for any temporary object structures it might need. Tools generally respond to the `on` event by calling vlib to post its temporary world objects, if it has any. Also, any necessary tool data structures are initialized. Tools respond to the `off` event by removing any temporary world objects it has created and added to the world. Tools respond to the `move` and `always` events by doing any actions that need to be updated each frame, such as animation and rubber-banding. Tools respond to the left and right presses, releases, clicks, and drags in variable ways; it depends on the tool. Most tools modify internal data structures in response to these events, or call database routines to modify the database.

To be compatible with undo and redo, a tool must store any necessary information needed to reverse an action every time an undoable action is completed. A routine must be written which accomplishes the undo and redo based on the stored information. These

routines are called automatically in response to an undo or redo command being given. See Undo for more information.

The following list describes exactly how to add a new tool. After the `.c` and `.h` files are created with the appropriate event routines, event.h must be updated to actually let the toolbox and event dispatcher know about the new tool. The changes to be made are printed in **bold** text.

0) Study existing tools and read over this list several times.

1) Create a .c file with the proper event routines.

```
void new_icon(p0, p1) PointType p0, p1; { }
void new_init() { }
void new_on() { }
void new_off() { }
void  new_left_press()
v_button_type button_list; MatrixType curr_mouse; { }
void  new_left_release()
v_button_type button_list; MatrixType curr_mouse; { }
void  new_left_drag()
v_button_type button_list; MatrixType curr_mouse; { }
void  new_right_press()
v_button_type button_list; MatrixType curr_mouse; { }
void  new_right_release()
v_button_type button_list; MatrixType curr_mouse; { }
void  new_right_drag()
v_button_type button_list; MatrixType curr_mouse; { }
void  new_move()
v_button_type button_list; MatrixType curr_mouse; { }
void  new_always()
v_button_type button_list; MatrixType curr_mouse; { }
```

2) Create a .h file with the event prototypes and the icon name

```
extern  void  new_icon  PROTO((PointType,  PointType));
extern  void  new_init();
extern  void  new_on();
extern  void  new_off();
extern  void  new_left_press  PROTO((v_button_type,  MatrixType));
extern  void  new_left_release  PROTO((v_button_type,  MatrixType));
extern  void  new_left_click  PROTO((v_button_type,  MatrixType));
extern  void  new_left_drag  PROTO((v_button_type,  MatrixType));
extern  void  new_right_press  PROTO((v_button_type,  MatrixType));
extern  void  new_right_release  PROTO((v_button_type,  MatrixType));
extern  void  new_right_click  PROTO((v_button_type,  MatrixType));
extern  void  new_right_drag  PROTO((v_button_type,  MatrixType));
extern  void  new_move  PROTO((v_button_type,  MatrixType));
extern  void  new_always  PROTO((v_button_type,  MatrixType));
```

3) Add to sound.h: a `#define` for the new tool's sound, making sure there are no duplications for other sounds.

```
#define snd_UNGROUP      "ug"
#define snd_UNMARK       "um"
#define snd_TB_HANDLE    "ha"
#define snd_TB_LOCK      "lk"
#define snd_TB_DEFPOS    "dp"
#define  snd_NEWTOOL     "nt"
```

4) Record the name of the tool with MacRecorder, save it as a resource with the first two characters of its name the same as those entered into `sound.h`, and paste it into the file `3DM_SOUNDS` on the Macintosh sound server.

5) Add to `event.h`: the new tool's `.h` file

```
#include <fly.h>
/* ... */
#include <sound.h>
#include <move.h>
#include <group_cmds.h>
#include  <new_tool.h>
```

6) Add to `event.h`: a `#define` for the new tool's code and update `NUM_TOOLS`

```
/* tools */
#define TOOLBOX_TOOL      0  /* used in toolbox -- overrides current tool */
#define SELECTION_TOOL    1
#define TRIANGLE_TOOL     2
#define FLY_TOOL          3
/* ... */
#define PARENT_LCMD       10
#define MERGE_LCMD        11
#define MOVE_TOOL         12
#define  NEW_TOOL         13
#define  NUM_TOOLS        14    /* was 13 */
```

7) Add to `event.h`: a list of routines to the global `tool_list`. Remember to add a comma if the routines are being added as the last set of routines! Set the cursor if a new cursor was defined, or set it to `ARROW_CURSOR` or another existing cursor name.

```
/*-------------------------*/
/* global table variables */

/*
  toolbox tool functions
  structure order is:
  {cursor, sound,
   init, on, off,
   left_press, left_release, left_click, left_drag,
   right_press, right_release, right_click, right_drag,
   move, always}
*/
extern ToolType tool_list[NUM_TOOLS];
#ifdef DEFINE_GLOBALS
  ToolType tool_list[NUM_TOOLS] = {
    {ARROW_CURSOR, "", null_icon,                /* toolbox */
     tb_init, null_func, null_func,
     tb_left_press, tb_left_release, null_event, tb_left_drag,
     tb_right_press, null_event, null_event, null_event,
     tb_move, null_event},
    /* ... */
    {TRIANGLE_CURSOR, snd_MERGE, null_icon,      /* merge lcmd */
     MergeOn, null_func, null_func,
     MergeLeft, null_event, null_event, null_event,
     ParentRight, null_event, null_event, null_event,
     ParentMove, null_event},
```

```
    {ARROW_CURSOR, snd_MOVE, mov_icon,      /* move tool */
     mov_init, mov_on, mov_off,
     mov_left_press, mov_left_release, null_event, mov_left_drag,
     mov_right_press, mov_right_release, null_event, mov_right_drag,
      mov_move, mov_always},  /* <- note comma */
{ARROW_CURSOR, snd_NEWTOOL, new_icon,       /* new tool */
new_init, new_on, new_off,
new_left_press, new_left_release, new_left_click, new_left_drag,
new_right_press, new_right_release, new_right_click, new_right_drag,
new_move, new_always}
};
#endif DEFINE_GLOBALS
```

8) If there is an `TB_UNAVAILABLE` cell in the global `tb_cells` in `event.h`, then replace a `TB_UNAVAILABLE,TB_UNAVAILABLE` with `TB_TOOL` if this tool is a tool or `TB_LCMD` if the new tool is actually a long command, and the tool code entered earlier.

```
{ { TB_LCMD, COLOR_LCMD },
  { TB_UNAVAILABLE, TB_UNAVAILABLE },
  { TB_UNAVAILABLE, TB_UNAVAILABLE },
  { TB_UNAVAILABLE, TB_UNAVAILABLE },
   { TB_TOOL, NEW_TOOL },   /* was TB_UNAVAILABLE, TB_UNAVAILABLE */
  { TB_CMD, QUIT_CMD } }
};
#endif
```

9) If there is not an `TB_UNAVAILABLE` cell in the global `tb_cells` in `event.h`, then the toolbox will have to be expanded in size by changing the variables `TB_NUMTOOLS_H` and `TB_NUMTOOLS_V`. The initialization of the two-dimensional array `tb_cells` will have to be updated to reflect the new dimensions of the toolbox.

```
/*-------------------*/
/* toolbox #defines  */

/* toolbox is TB_NUMTOOLS_H cells across and TB_NUMTOOLS_V cells down */
#define TB_NUMTOOLS_H      7    /* was 6 */
#define TB_NUMTOOLS_V    5

/* toolbox mapping */
/* maps cell to tool or cmd  */
extern CellType tb_cells[TB_NUMTOOLS_V][TB_NUMTOOLS_H];
#ifdef DEFINE_GLOBALS
CellType tb_cells[TB_NUMTOOLS_V][TB_NUMTOOLS_H] = {
{ { TB_TOOL, SELECTION_TOOL },
  { TB_CMD, UNMARK_CMD },
  { TB_TOOL, MOVE_TOOL },
  { TB_TOOL, EXTRUDE_TOOL },
  { TB_TOOL, SURFROT_TOOL },
   { TB_TOOL, TRIANGLE_TOOL }, /* <- note comma */
  { TB_TOOL, NEW_TOOL }
},
/* etc */
```

10) If the tool changes the database, undo and redo routines must be written. See Undo implementation for more details.

2.8. COMMANDS

2.8.1. GROUP COMMAND

The Group command creates a new group at the world level, and finds a unique name for it. The current naming algorithm is to scan all the groups and look for names of the form "GROUP#", where # is any number. The largest number is remembered and the new name becomes "GROUP#", where # is the largest plus one.

Each marked triangle and instance is then deleted from its old group and inserted into the new group. Since all the hierarchical commands need to do this, it has been consolidated into a single callback called `ChangeParentCallback()`.

As it currently stands this command could be implemented as a normal command rather than as a long command. It has been left as a long command for such time when a form of text entry can be used to allow the user to choose a name for the new group.

No extra work was required to support undo and redo, since the delete and insert routines support it for us. This also holds for the other hierarchical commands.

2.8.2. UNGROUP COMMAND

The Ungroup command is almost exactly like the Group command, except that the model group is used as the new parent rather than a newly created group.

2.8.3. PARENT COMMAND

The Parent is a long command, allowing the user to specify a new parent for the marked triangles and instances. Choosing a parent is similar to marking a parent with the Mark tool, except that not only should marked and protected instances stop the search down the path, but so should encountering the previous choice for parent. The code for this procedure is based on the Mark tool, except for the added work to check for the previous selection.

When the left button is pressed the current selection is taken to be new parent for the marked objects and the `ChangeParentCallback()` is used to do the actual work. If no parent has been selected a right button press is simulated so that clicking the left button near a vertex is a shortcut for selecting the vertices parent. If no vertex is near then no parent is chosen and the command quits without making any changes.

2.8.4. MERGE COMMAND

The Merge command is a long command, and is almost the same as the Parent command. The user is allowed to select a new parent, but `MergeParentCallback()` is used instead of `ChangeParentCallback()`. For each marked instance, `MergeParentCallback()` scans its triangles and child instances and passes each to `ChangeParentCallback()`, so everything in the marked instance ends up in the new parent.

2.8.5. COLOR COMMAND

The color command, actually a long command, temporarily replaces the toolbox with its own structure with the routine `tb_replace_toolbox()`. This routine replaces the toolbox with the structure name passed in. The position and rotation xforms of the toolbox are copied and updated each loop, depending on whether or not the toolbox is locked. When the mouse is released, `tb_done()` is called to revert to the previous tool.

When we get a drag event, we compute the coordinates of the hand in object space. We use these coordinates to compute the color actually being pointed to. We update the sample color each time through the loop. When the mouse is released, we change the global color.

2.8.6. DEFAULT VIEW COMMAND

The Default View Command simply replaces the `room_from_world` transformation with its default value. The `room_from_world` transformation is updated by the Fly, Scale World, and Rotate World tools. The default value of this transformation is saved when the Default View command is initialized, and copied into the vlib tree when the command is activated.

2.8.7. FLOOR ON/OFF COMMAND

The Floor On/Off Command simply replaces the floor object, which is in world space, with a null object so that it will not be drawn. The dlist address of the floor execute structure call was saved when the toolbox was initialized. The red X in the floor icon is updated depending on the state of the toggle.

2.8.8. MAGIC CARPET ON/OFF COMMAND

The Magic Carpet On/Off Command simply replaces the magic carpet object, which is in tracker space, with a null object so that it will not be drawn. The DlistAddress of the magic carpet execute structure call was saved when the toolbox was initialized. The red X in the magic carpet icon is updated depending on the state of the toggle.

2.8.8. UNMARK ALL COMMAND

The Unmark All command simply uses `UnmarkCallback()` to unmark all vertices and instances.

2.8.9. HOW TO MAKE NEW COMMANDS

A command must provide a routine to be called when it is activated, and routines to be called to initialize it and create its icon in the toolbox. Most commands respond to the `init` event by doing any one-time-only initialization, such as reserving space in the dlist for any temporary object structures it might need.

To be compatible with undo and redo, a command must store any necessary information needed to reverse its action if it performs an undoable action. A routine must be written which accomplishes the undo and redo based on the stored information. These routines are called automatically in response to an undo or redo command being given. See Undo for more information.

The following list describes exactly how to add a new command. After the `.c` and `.h` files are created with the appropriate event routines, event.h must be updated to actually let the toolbox and event dispatcher know about the new tool. The changes to be made are printed in **bold** text.

0) Study existing commands and read over this list several times.

1) Create a .c file with the proper event routines.

```
void new_icon(p0, p1) PointType p0, p1; { }
void new_init() { }
void new_activate() { }
```

2) Create a .h file with the event prototypes and the icon name

```
extern  void  new_icon  PROTO((PointType,  PointType));
extern  void  new_init  PROTO(());
extern  void  new_activate  PROTO(());
```

3) Add to sound.h: a `#define` for the new command's sound, making sure there are no duplications for other sounds.

```
#define snd_UNGROUP       "ug"
#define snd_UNMARK        "um"
#define snd_TB_HANDLE     "ha"
#define snd_TB_LOCK       "lk"
#define snd_TB_DEFPOS     "dp"
#define  snd_NEWCMD     "nc"
```

4) Record the name of the command with MacRecorder, save it as a resource with the first two characters of its name the same as those entered into `sound.h`, and paste it into the file `3DM_SOUNDS` on the Macintosh sound server.

5) Add to `event.h`: the new command's `.h` file

```
#include <fly.h>
/* ... */
#include <sound.h>
#include <move.h>
#include <group_cmds.h>
#include  <new_cmd.h>
```

6) Add to `event.h`: a `#define` for the new command's code and update `NUM_TOOLS`

```
/* commands */
#define UNDO_CMD          0
#define REDO_CMD          1
#define SAVE_CMD          2
#define LOAD_CMD          3
/*  ...  */
#define CUT_CMD           10
#define COPY_CMD          11
#define PASTE_CMD         12
#define ERASE_CMD         13
#define  NEW_CMD        14
#define  NUM_COMMANDS  15    /* was 14 */
```

7) Add to `event.h`: a list of routines to the global `command_list`. Remember to add a comma if the routines are being added as the last set of routines!

```
/* toolbox command functions                              */
/* structure order is:                                    */
/*  {init, activate}                                 */
extern CommandType command_list[NUM_COMMANDS];
#ifdef DEFINE_GLOBALS
  CommandType command_list[NUM_COMMANDS] = {
    {snd_UNDO, undo_icon, undo_init, undo_activate},    /* undo */
    {snd_REDO, redo_icon, null_func, redo_activate},    /* redo */
    {snd_PASTE, Paste_Icon, paste_init, paste_activate},/* paste */
     {snd_DELETE, Erase_Icon, erase_init, erase_activate}, /* <- note "," */
     {snd_NEWCMD,  new_icon,  new_init,  new_activate},   /* new cmd */
};
#endif
```

8) If there is an `TB_UNAVAILABLE` cell in the global `tb_cells` in `event.h`, then replace a `TB_UNAVAILABLE,TB_UNAVAILABLE` with `TB_CMD`, and the command code entered earlier.

```
{ { TB_LCMD, COLOR_LCMD },
  { TB_UNAVAILABLE, TB_UNAVAILABLE },
  { TB_UNAVAILABLE, TB_UNAVAILABLE },
  { TB_UNAVAILABLE, TB_UNAVAILABLE },
   { TB_CMD,  NEW_CMD  },    /*  was  TB_UNAVAILABLE,  TB_UNAVAILABLE  */
  { TB_CMD, QUIT_CMD } }
};
#endif
```

9) If there is not an `TB_UNAVAILABLE` cell in the global `tb_cells` in `event.h`, then the toolbox will have to be expanded in size by changing the variables `TB_NUMTOOLS_H` and `TB_NUMTOOLS_V`. The initialization of the two-dimensional array `tb_cells` will have to be updated to reflect the new dimensions of the toolbox.

```
/*-------------------*/
/* toolbox #defines  */

/* toolbox is TB_NUMTOOLS_H cells across and TB_NUMTOOLS_V cells down */
#define  TB_NUMTOOLS_H      7     /*  was  6  */
#define TB_NUMTOOLS_V    5

/* toolbox mapping */
/* maps cell to tool or cmd  */
extern CellType tb_cells[TB_NUMTOOLS_V][TB_NUMTOOLS_H];
#ifdef DEFINE_GLOBALS
CellType tb_cells[TB_NUMTOOLS_V][TB_NUMTOOLS_H] = {
{ { TB_TOOL, SELECTION_TOOL },
  { TB_CMD, UNMARK_CMD },
  { TB_TOOL, MOVE_TOOL },
  { TB_TOOL, EXTRUDE_TOOL },
  { TB_TOOL, SURFROT_TOOL },
   { TB_TOOL,  TRIANGLE_TOOL  }, /* <- note comma */
  {  TB_CMD,  NEW_CMD  }
},
/*  etc  */
```

10) If the command changes the database, undo and redo routines must be written. See Undo implementation for more details.

APPENDIX 1 — ERROR MESSAGES

aborting load: line ***line-number*** **at "***text-from-file***"** — some internal error prevented a file load from succeeding. The line number and text give the location in the data file where the error occurred. This message will be preceded by another error message for the specific problem. Probably a memory allocation failure.

continuing without the use of undo — file error occurred so the undo system would no longer work. Rather than quit the program, undo has been turned off.

filename**: *unix error message*** — error opening or accessing file. The unix message should provide a description of the problem.

illegal vertex count: line ***line-number*** **at "***text-from-file***"** — when loading a data file, a vertex count other than three was given. PPHIGS allows polygons with more vertices, but 3dm only supports triangle. The line number and text give the location in the data file where the error occurred.

redo failed: ***unix error message*** — some file or unix system call error prevented the redo operation from succeeding.

reinitializing undo — undo information has become out of sync with the program state. It is not possible to undo any operations made prior to this message.

saving: ***unix error message*** — something went wrong trying to save the database, probably a file write error, though the unix message should describe the problem.

structure "*group-name***" undefined for execute: line** ***line-number*** **at "***text-from-file***"** — when loading a data file, an "execute" or "ex" line tried to reference an group that hasn't been defined. The line number and text give the location in the data file where the error occurred.

too few vertices: line ***line-number*** **at "***text-from-file***"** — when loading a data file, fewer than three vertices were actually given in a polygon definition. The line number and text give the location in the data file where the error occurred.

too many vertices: line ***line-number*** **at "***text-from-file***"** — when loading a data file, more than three vertices were actually given in a polygon definition. The line number and text give the location in the data file where the error occurred.

Warning: A "NULL" edge handle was passed! — Internal error in a call to one of the database functions. This message can only occur due to a program bug.

Warning: A "NULL" triangle handle was passed! — Internal error in a call to one of the database functions. This message can only occur due to a program bug.

Warning: A "NULL" vertex handle was passed! — Internal error in a call to one of the database functions. This message can only occur due to a program bug.

Warning: A deleted instance handle was passed! — Internal error in a call to one of the database functions. This message can only occur due to a program bug.

Warning: An empty instance handle was passed! — Internal error in a call to one of the database functions. This message can only occur due to a program bug.

Warning: An empty triangle handle was passed! — Internal error in a call to one of the database functions. This message can only occur due to a program bug.

Warning: Attempt to mark parent of model instance — You tried to mark the mark the parent of the model. Since the model has no parent, this is not allowed.

Warning: Delete_Triangle: Edge found totally unconnected! — Internal error in a call to one of the database functions. This message can only occur due to a program bug.

Warning: Delete_Triangle: Triangle found connected to a deleted edge! — Internal error in a call to one of the database functions. This message can only occur due to a program bug.

Warning: Delete_Triangle: Triangle found connected to a deleted vertex! — Internal error in a call to one of the database functions. This message can only occur due to a program bug.

Warning: Delete_Triangle: Vertex found totally unconnected! — Internal error in a call to one of the database functions. This message can only occur due to a program bug.

Warning: Disconnect_Edge_from_Vertex(): couldn't find edge *edge_number* — Internal error. There may be some odd behavior from the triangle and vertices that were being modified, but otherwise the program should be OK.

Warning: Disconnect_Triangle_from_Vertex: couldn't find triangle *triangle_number* — Internal error. There may be some odd behavior from the triangle and vertices that were being modified, but otherwise the program should be OK.

Warning: Edge found totally unconnected! — Internal error in a call to one of the database functions. This message can only occur due to a program bug.

Warning: Invalid instance count was found! — Internal error in a call to one of the database functions. This message can only occur due to a program bug.

Warning: Redo_Delete_Triangle: can't find group — Internal error. This message can only occur due to a program bug.

Warning: Redo_Delete_Triangle: can't find triangle — Internal error. This message can only occur due to a program bug.

Warning: Redo_Delete_Triangle: can't find vertex — Internal error. This message can only occur due to a program bug.

Warning: The group name "name" is too long. — The length of a group name is limited to about 22 characters. The creation of a group was aborted.

Warning: The model instance can't be merged with anything — You selected the Merge command when the model instance was marked. The model must always remain at the top of the hierarchy, so this is not allowed since it would result in the deletion of the model.

Warning: The parent of the model instance can't be changed — You selected the Parent command when the model instance was marked. The model must always remain at the top of the hierarchy, so this is not allowed.

Warning: Triangle found connected to a deleted edge! — Internal error in a call to one of the database functions. This message can only occur due to a program bug.

Warning: Triangle found connected to a deleted vertex! — Internal error in a call to one of the database functions. This message can only occur due to a program bug.

Warning: Unable to allocate memory for connections. — probably the only possible thing to do at this point is save the file. Restarting the program from scratch and reloading may clean up enough wasted memory to allow minor editing.

Warning: Unable to allocate memory for edge connections. — probably the only possible thing to do at this point is save the file. Restarting the program from scratch and reloading may clean up enough wasted memory to allow minor editing.

Warning: Unable to allocate memory for edges. — probably the only possible thing to do at this point is save the file. Restarting the program from scratch and reloading may clean up enough wasted memory to allow minor editing.

Warning: Unable to allocate memory for more edges. — probably the only possible thing to do at this point is save the file. Restarting the program from scratch and reloading may clean up enough wasted memory to allow minor editing.

Warning: Unable to allocate memory for more triangles. — probably the only possible thing to do at this point is save the file. Restarting the program from scratch and reloading may clean up enough wasted memory to allow minor editing.

Warning: Unable to allocate memory for more vertices. — probably the only possible thing to do at this point is save the file. Restarting the program from scratch and reloading may clean up enough wasted memory to allow minor editing.

Warning: Unable to allocate memory for the new group "name". — probably the only possible thing to do at this point is save the file. Restarting the program from scratch and reloading may clean up enough wasted memory to allow minor editing.

Warning: Unable to allocate memory for triangles. — probably the only possible thing to do at this point is save the file. Restarting the program from scratch and reloading may clean up enough wasted memory to allow minor editing.

Warning: Unable to allocate memory for vertices. — probably the only possible thing to do at this point is save the file. Restarting the program from scratch and reloading may clean up enough wasted memory to allow minor editing.

Warning: Undo_Delete_Triangle: can't find group — Internal error. This message can only occur due to a program bug.

Warning: You can't delete the single instance of the model! — The only group that must always exist is the "model". This warning occurs when you attempt to delete it. No harm is actually done to the database.

Warning: You can't put instances at the world level! — Internal error. This message can only occur due to a program bug.

APPENDIX 2 — FILE FORMAT

2.1. FORMAT DESCRIPTION

The format of files that 3dm can save and load is given below.

```
file =
   group_definition*
   Model_group_definition

group_definition =
   struct name {
      triangle*
      instance*
   };

Model_group_definition =
   struct TDM_Model post {
      triangle*
      instance*
   };

triangle =
   [poly color {color [color]};]
   poly 3 {
      vertex
      vertex
      vertex
   };

color = int int int [int]

vertex = float float float;

instance =
   mat Rep {
      float float float float
      float float float float
      float float float float
   };
   ex name;
```

Notes:
- **name** is the name of a group (not in quotes).
- **float** is any floating point number.
- **int** is an integer from 0 to 255.
- the two colors in a triangle are front and back color.
- the integers in a color are red, green, blue, and alpha.
- Groups must be defined before they are instanced.
- the mat part of an **instance** is a three by four PPHIGS style transformation matrix.

2.2. SAMPLE FILE

```
struct group1 {
   poly col {
      0 128 128 0
      0 128 128 0
   };
   poly 3 {
      0 -1 -1;
      0 -1  1;
      0  1 -1;
   };
   poly 3 {
      0 -1  1;
      0  1 -1;
      0  1  1;
   };
};
struct TDM_Model post {
   poly col {
      255 100 100 0
      255 100 100 0
   };
   poly 3 {
      -1 -1 0;
      -1  1 0;
       1 -1 0;
   };
   poly col {
      255 0 255 0
      255 0 255 0
   };
   poly 3 {
      -1  1 0;
       1 -1 0;
       1  1 0;
   };
   mat Rep {
      1 0 0 -1
      0 1 0 0
      0 0 1 0
   };
   ex group1;
   mat Rep {
      1 0 0 1
      0 1 0 0
      0 0 1 0
   };
   ex group1;
};
```

INDEX

3DM_SOUNDS 21, 25
activate event 14
Add_Triangle 7, 8
Add_Vertex 8
always event 3, 6
ARROW_CURSOR 21
bison 4
BSD 14
button 5
callback 9
change_cursor 17
ChangeParentCallback 23
ChangeTraversal 9, 18
check_tracker 5
child 23
children 10
Clear 8
click event 3, 5
color command 23
command 1, 3
command_list 25
commands 23
  color 23
  default view 24
  floor on/off 24
  group 23
  magic carpet on/off 24
  merge 23
  new commands 24
  parent 23
  ungroup 23
  unmark 10
  unmark All 24
composites 9
coordinate systems 5
Create_Group 7
Create_Instance 7
Create_Triangle 2
cursor 1, 6, 21
Database 2, 6, 10
default view command 24
delete 23
Delete_Triangle 6
Delete_Triangle_NoUndo 7
dispatch_events 3, 5, 19
display 5, 8
displays 5
DlistAddress 11, 15, 16, 24
drag event 3, 5
EDGE 10, 11
Edge_Handle 10
edges 11
event 20, 24
event handler 1, 5
EventFunctPtr 6
events 19
  activate 14
  always 3, 6
  click 3, 5
  drag 3, 5
  init 3, 6
  left_click 6
  left_drag 6
  left_press 6
  left_release 6
  move 3, 6
  null event 5
  off 3, 6, 19
  on 3, 6, 19
  press 3, 5
  refresh 6
  release 3, 5
  right_click 6
  right_drag 6
  right_press 6
  right_release 6
extrusion 19
file format 30
Files 4
Find_Nearest 9, 12
Find_Nearest_In_Geometry 12
flex 4
floor 5, 16, 24
floor on/off command 24
floor tool 15
fly tool 15, 16, 17, 24

function talbes 1
geometry 7, 10, 11
GNU 4
group 2, 10, 11
group command 23
group handle 7
GROUP_MARKED 10
group_struct 10
groups 7
handle 2, 7
head_from_toolbox 15
help 17
Hide 8
hierarchy 8, 23
icon 1, 3, 6, 13, 19, 20, 25
IconFunctPtr 6
icons 15
indirect function 2
init 19
init event 3, 6
init_vlib 15, 16
InitFunctPtr 6
insert 23
instance 2, 8, 9, 10, 11, 18, 23
instance handle 7
Instance_Handle 9, 10
instances 9
inverts 9
layers 1
left_click event 6
left_drag event 6
left_press event 6
left_release event 6
levels 1
lex 4
long command 1, 3, 23
Macintosh 21, 25
MacRecorder 21, 25
magic carpet 5, 16, 24
magic carpet on/off command 24
mark tool 18
marked set 8
MarkMany 10, 18
MarkOff 10
MarkOn 10
MatrixType 6, 9, 11
merge command 23
Merge_Triangle 7, 8
MergeParentCallback 23
mouse 5
move event 3, 6
move tool 18
NameType 6, 10, 13, 14
new commands 24
new tools 19
NO_UNDO 7
NOT_MOVED_LIMIT 5
NOT_ROTATED_LIMIT 5
null_event 5
NUM_TOOLS 25
num_triangle_holes 11
num_triangles 11
numInstances 9
off event 3, 6, 19
on event 3, 6, 19
parent 18
parent command 23
parser 4
Path 9, 10
pg_draw_frame 15
pg_set_line_color 17
Pixel-Planes 4, 8
PointType 6
pp_object 11
pp_xform 11
PPHIGS 1, 2, 4, 7, 8, 10, 15, 16
PPHIGS archive file 4
PPHIGS structure 1, 2, 8, 15, 16, 17, 23, 24
press event 3, 5
program layers 1
realloc 11
Reclaim_Triangle_Space 7
redo 2, 3, 12, 19, 22, 23, 26
redo_func 14
RedoFunctPtr 14
refresh event 6
RefreshFunctPtr 6
release event 3, 5
resource 21, 25
restore_cursor 17
right_click event 6
right_drag event 6

right_press event 6
right_release event 6
room_from_toolbox 16
room_from_world 24
rotate world tool 15, 16, 18, 24
scale world tool 15, 16, 18, 24
Set 8
SetMarkCallback 10
sound 1, 6, 20, 25
sound server 21, 25
SunOS 14
surface of extrusion tool 19
SwitchFunctPtr 6
tb_anti_rotation 15
tb_cells 22, 26
TB_CMD 26
tb_done 23
TB_LCMD 22
TB_NUMTOOLS_H 22
TB_NUMTOOLS_V 22
tb_replace_toolbox 23
TB_TOOL 22
tb_toolbox_xform 15
TB_UNAVAILABLE 22, 26
tool 1, 2, 3, 5
tool_list 17
toolbox 1, 3, 5, 14, 16, 17
tools 17
  extrusion 19
  floor 15
  fly 15, 16, 17, 24
  mark 18
  move 18
  new tools 19
  rotate eorld 18
  rotate world 15, 16, 24
  scale world 15, 16, 18, 24
  triangle creation 18
ToolType 6
trackerlib 5
trackers 5
TrackNearest 12
transform 11
transformations 15
TransformCallback 10
TransformMarked 10
TraverseMarked 9, 18
TraverseMarkedEnd 18
TRIANGLE 10, 11
triangle creation tool 18
Triangle_Handle 10
triangle_holes 11
triangle_space 11
triangles 11
Ultrix 14
undo 2, 3, 6, 7, 12, 19, 22, 23, 26
undo stack 2
undo_check 14
undo_data 14
undo_func 14
undo_start 13
undo_start_or_add 14
UndoFunctPtr 14
ungroup command 23
unlink 14
unmark 10
Unmark All 24
unmark all command 24
UnmarkCallback 10, 24
update 5
update_toolbox 15
v_button_type 6
V_ROOM_FROM_WORLD 17, 18
v_tree_xforms 17, 18
VERTEX 9, 10, 11
vertex handle 8
Vertex_Handle 9, 10
vertices 11
vlib 3, 5, 17, 19
vv_hide_object 17
vv_replace_line_color 17
vv_replace_polygon_color 17
vv_reserve_execute 17
x_order 11
y_order 11
yacc 4
z_order 11