

libDraw
a library for interactive drawing on the Pixar
version 1.3b

Flip Phillips
PIXAR
5 January 1989

Abstract

This document describes a library for interactive drawing on the Pixar Image Computer. The entire library is written in **ChapC** and should be portable to other systems which support single pixel access. **libDraw** provides arbitrary line drawing, filling, polygon operations (called contours or *edges*) and edge list operators. It was originally designed for use with **sled**, the ChapVolumes Slice Editor.

Some Introductory Remarks

libDraw provides an interactive drawing system for the Pixar Image Computer. All of the basic line and polygon (edge) tools are provided in a simple to use **ChapC** library.

libDraw is fast, however, it is not as fast as some systems. For an explanation why, we must look at the origins and intent of the Pixar Image Computer. The Pixar was originally conceived and designed to be a digital film printer for Lucasfilm. Needless to say, its design has changed substantially since then and it is now an *Image Computer*, that is, a computer for pixel processing. It is not a 'real-time' 3D machine, nor is it designed to do high speed line drawing like a PS300.

This package supplies the basic tools to allow the Pixar to behave as if it were a line drawing engine. Its speed will surprise you, considering it is moving 48 bit pixels around and is applying an arbitrary function to every pixel on every line. However, it won't perform at the speed of a PS300 (yet).

Programming libDraw

libDraw programming can be as simple as drawing lines on the Pixar, or as complicated as developing your own application functions for **libDraw** to perform. It is not entirely necessary that you know how to program the **chap**, but it is very helpful.

The things you will need to know are:

- Chad
- ChapC interface routines
- Window system programming for your machine

If you want to write more high powered utilities, a working knowledge of **ChapC** and basic Image Computer programming are necessary.

Library Design

libDraw is separated into three functional libraries: **libDraw**, **libEdge**, and **libIDraw**.

libDraw is the lowest level of these libraries. It provides all of the simple non-interactive line drawing functions and co-ordinate generation routines. For simple programs requiring no user interaction, such as a graph drawing program, **libDraw** is the library to use.

The drawing functions in **libDraw** are *destructive*, that is, they delete the image under the lines which they draw. For *nondestructive* drawing, see **libIDraw**. **libDraw** can be used by itself, it needs no routines from **libEdge** or **libIDraw**.

libEdge provides low level list management routines for **libIDraw**. It is required whenever using **libIDraw**. You can use it also to manage an edge list of your own. It does not require **libDraw** or **libIDraw** to operate.

libIDraw is the high level, interactive library. It requires both **libEdge** and **libDraw** to operate. All interactive routines are managed from this library.

Basic Concepts

libDraw uses a few notions familiar to the Pixar programmer and a few unfamiliar notions. **pixlewindows** (**CHAD_PW**) are the main form of communication within the package, just as in **Chad** and **libPixar**. This library however maintains the notion of a *current pw*. This is the **pw** last accessed or selected by a call to the library. This eliminates the **pw** field in many of the function calls to **libDraw**. You simply select a **pw** with a call to **SetPW** and draw lines with **Line(x, y)**, no need to pass around the extra baggage of a **pw** pointer.

The routines in **libIDraw** also adhere to these guidelines. A **pw** is only supplied when it is needed as a temporary storage space for the function.

The lower level functions, or *engines* in **libDraw** require that you supply a *user function*. A user function is a **ChapC** function which applies a certain operation to the currently selected pixel in the image. These functions are uniform across the package, taking the form:

```
int
func( pw, x, y, p )
    int *pw;
    int x, y;
    parallel int *p;
```

where **pw** points to the pixwin of interest, **x, y** the co-ordinates within the window, and **p**

the pixel value at this point. (Note that **p** does not contain a value in this version. It is supplied for temporary storage by the function. Note that it may go away in future versions.)

For example, If you called **BresEngine** requesting that it draw a line from (1,1) to (5,5) using function **Black**, it would apply **Black** at pixels (1,1), (2,2), (3,3), (4,4), and (5,5). The function **Black**, written by you, could look something like this:

```
int
Black(pw,x,y,p)
    int *pw;
    int x,y;
    Pixel *p;
{
    *p = 0; /* set the temporary pixel to Black */
    OnePixelWrite(pw,x,y,p); /* write it */
}
```

This function, when supplied to the call of **BresEngine**, would write the value of 0 at every pixel on the line from (1,1) to (5,5).

More complicated systems might blur the pixel located at each point or invert it.

The same paradigm applies to the **Fill** family of routines. They evaluate the user function at every point within some boundary. This can be used for histogram computation, area measurement, etc.

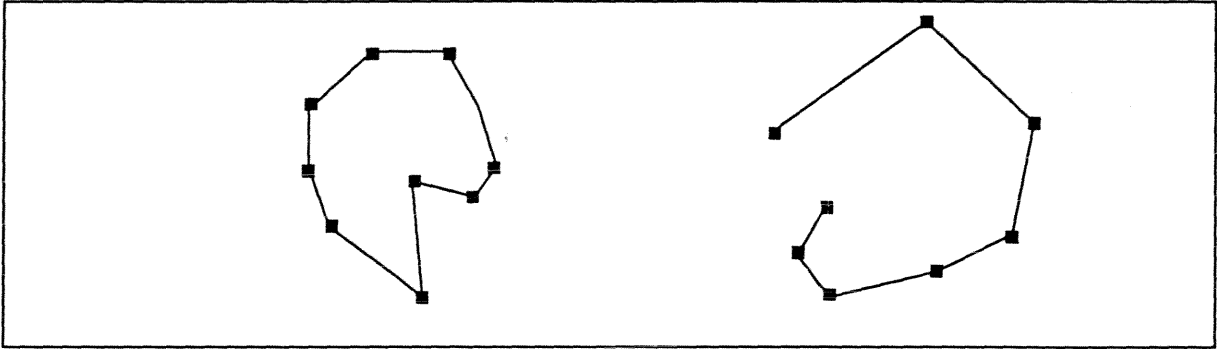
A **Fill** function which darkens a region might look something like this:

```
int
Darken(pw,x,y,p)
    int *pw;
    int x,y;
    Pixel *p;
{
    OnePixelRead(pw,x,y,p); /* read current val */
    p -= 100; /* decrement it */
    OnePixelWrite(pw,x,y,p); /* write it */
}
```

This fill function maintains the original image, only darkens it at every pixel evaluated to be within a boundary supplied to **Fill**.

As stated above, the Interactive Drawing Package supplied in **libIDraw** uses the notion of an edge to define some specified contour or polygon. These are useful for outlining areas of interest, selecting areas to be operated on, etc.

An edge can be either open or closed and can be of any shape from a straight line, simple rectangle, to a complex polygon.



Some edges

An edge description is made up of connected points or *vertices*. The vertices are marked with black dots in the above example. In **libIDraw**, vertices can be moved and deleted, more vertices can be added, the edge can be simplified to contain less vertices, and more. **libEdge** gives you a way to access these lists so you can manipulate them yourself from within a **ChapC** program.

Version 1.3 differs from past versions of **libIDraw** in that there are no limitations on edge size (number of points or vertices) other than the available memory. This needs to be set up in advance with a call to **ChCSetMallocHeap** (available in **ChapTools** version 2.2 and greater). In most of the experiments with **libIDraw** the heap has been set to 40000. You should experiment in order to customize the heap size for your application. Note that if you start the edge package with out first setting the heap size all of the memory allocation on the Pixar is dynamic, that is, it will go away after each call to a **libIDraw** function from the host.

In **libIDraw** there is the notion of the *current edge*. This , as with the current pw, is the last edge selected or operated on. The interactive routine **FindClosest** is used to select the current edge. A parameter allows you to specify how far from a mouse event an edge is allowed to be in order for it to be selected as the closest edge.

If no edge is found close enough to the mouse location, **FindClosest** returns an error and none of the edge processing operations will work until a edge is selected. Also, when an edge is created, either manually or automatically, the created edge is set as the current edge.

A note on programming style, you will probably want to call **FindClosest** on a *MouseDown* event from the window system. Then call your selected operation on the *MouseUp* or *MouseDown* events. For an example of this, see the file **sun.c** provided as a demo with this library.

libDraw library interface

Low Level Co-ordinate Generators

(*bresenham.c, fill.c*)

These functions supply the low level co-ordinate generation functionality. They are essentially *engines*, that is, given a set of conditions, generate a set of co-ordinates for the function. In **libDraw** the results from the engines are applied to an arbitrary user supplied function. These functions are uniform across the package, taking the form:

```
int
func( pw, x, y, p )
    int *pw;
    int x, y;
    parallel int *p;
```

where **pw** points to the pixwin of interest, **x,y** the co-ordinates within the window, and **p** the pixel value at this point. (Note that **p** does not contain a value in this version. It is supplied for temporary storage by the function. Note that it may go away in future versions.)

BresEngine(pw,x1,y1,x2,y2,func)

```
int *pw;
int x1,y1,x2,y2;
int (*func)();
```

This low level function evaluates **func** at every point which lies on the line from (**x1,y1**) to (**x2,y2**). **func** takes the form described above.

FillEngine(pw,x,y,crit,func)

```
int *pw;
int x,y;
int (*crit)(), (*func)();
```

This function does a seed fill, evaluating **func** at every point which **crit** declares not to be an edge pixel. Both **func** and **crit** are in the standard form described above. **crit** should return **TRUE** (1) if it has determined that it has hit an edge, **FALSE** (0) otherwise. See the code for **Fill** for a simple fill routine based on this engine.

Low Level Drawing Operators

(*fillfunc.c, line.c*)

These are the lowest level functions which should be called from any program. They provide basic line drawing functionality.

This level uses the 'current graphics state' to decide where and how something is to be drawn. Each operation moves and draws with the *pen* for that pixelwindow. Currently, only one pixelwindow is supported.

InitPort(pw)
 int *pw

Sets up the default graphics port. **pw** becomes the default graphics window, the pen location is set to (0,0), its color is set to white. **InitPort** only needs to be called once, usually at the beginning of the program. It must be called before any of the following functions.

SetPW(pw)
 int *pw;

Sets the pixelwindow which will receive the drawing commands.

int *
GetPW()

Returns the currently selected pixelwindow.

ForeColor(color)
 Pixel ●color;

Sets the pen color for all future drawing operations. Initially, it is set to the maximum value for a **Pixel**. (On a Pixar, 2047)

BackColor(color)
 Pixel ●color;

Sets the background color for all future drawing operations. Initially, it is set to the minimum value for a **Pixel**. (On a Pixar, 0)

HiliteColor(color)
 Pixel ●color;

Sets the hilight color for all future drawing operations. Initially, it is set to the average value for a **Pixel**. (On a Pixar, 1027)

MarkColor(color)
 Pixel ●color;

Sets the marking color for all future drawing operations. Initially, it is set to the average value for a **Pixel**. (On a Pixar, 1027)

MoveTo(x,y)
 int x,y;

Moves the pen to **x,y** in the current pixelwindow. No drawing operations are performed.

Move(dx,dy)
 int dx,dy;

Moves the pen in a relative manner, **x+dx** and **y+dy** from its present posi-

tion. No drawing is performed.

LineTo(*x*, *y*)
 int *x*, *y*;

Draws a line from the current pen position to *x*, *y*. The pen is left at the end of the line (*x*, *y*) upon completion.

Line(*dx*, *dy*)
 int *dx*, *dy*;

Draws a relative line to a position *x+dx* and *y+dy* from the present pen position. The pen is left at the end of the line (*x*, *y*) upon completion.

Dot(*x*, *y*)
 int *x*, *y*;

Draws a single pixel at *x*, *y*.

Stroke(*x1*, *y1*, *x2*, *y2*)
 int *x1*, *y1*, *x2*, *y2*;

Draws a line from *x1*, *y1* to *x2*, *y2*. Leaves the pen at *x2*, *y2*.

FloodFill(*mpw*, *x*, *y*)
 int **mpw*;
 int *x*, *y*;

performs a flood fill on the current pixwin from a 'border' supplied in *mpw*. This pixwin is should contain a set of non-zero pixels in its alpha channel. It uses this mask to perform a seed fill starting at *x*, *y*, painting the area with the current pen color. The *mpw* can be the same as the currently selected pixwin, as long as it has a representation of the boundary to be filled in the alpha channel .

Note that it is probably a good idea to frame the *mpw* with a non-zero alpha to avoid fill errors, should the seed point be outside some boundary or the boundary is not fully connected.

libEdge library interface

Structure Definitions

(*edge.h*)

These are the basic definitions used by **libDraw**. The file **edge.h** should be included in any **ChapC** programs using these functions.

```
typedef struct BBox {
    short xmin, xmax, ymin, ymax;
} BBox;
```

This structure contains 4 shorts to define an integer bounding box. It is used for clipping operations.

```
typedef struct point {
    short x,y;
    char flags;
} point;
```

This is a simple 2D point structure. The **flags** field is used for locking and deletion.

```
typedef struct edge {
    struct edge *next;           /* list pointer */
    bool dirty;                 /* is it modified */
    short size;                 /* # of points in loop */
    BBox bbox;                  /* bounding box */
    bool open;                  /* open or closed edge */
    point *data;                /* point data */
    double z;                   /* z height of contour */
} edge;
```

This is the basic edge structure. It contains pointers to the point data structures which describe the boundary and related support data.

Edge allocation and deallocation

(*edgefunc.c*)

These functions are the linked list support routines. They will normally not be needed in the normal programming of the drawing package. Note that these functions do not adjust several global variables in the package

```
edge *
EdgeList()
```

Returns the root edge of the edge list. Useful for alternative forms of traversal

maybe. Returns **NULL** if there are no edges allocated. Be forewarned that the first edge in the list (edge 0) contains a temporary work space for the edge tracking and simplification operations. The real edge data lives in structures after edge 0.

```
edge *  
AllocEdge( chunk )  
    short chunk;
```

Allocates and initializes an edge structure for further use. **size** is set to 0, **open** is set to **TRUE**, **dirty** is set to **FALSE**, and the **z** level is set to 0.0. The memory for the points list is allocated in **chunk** points sized blocks as they are needed. **NeedNewChunk()** is an internal routine which is called as points need to be allocated, **CompactChunks()** handles the garbage collection. You will probably never need to call these routines. Note that it does not set the globals **Selected** or **theEdge**.

```
FreeEdge( e )  
    edge *e;
```

Returns the memory and structure allocated to the free memory pool. Note that it only does the freeing, not any relinking.

```
FreeEdgeList()
```

Returns the entire edge list to the free memory pool. This is good for cleaning up after your program has completed.

```
edge *  
IndexEdge( index )  
    short index;
```

Returns the **index** edge into the edge list. Edge 0 is the first edge in the list. It will return **NULL** if the index is larger than the size of the edge list.

```
AppendEdge( e )  
    edge *e;
```

Appends **e** to the end of the edge list.

```
PrependEdge( e )  
    edge *e;
```

Places **e** at the beginning of the edge list.

```
DeleteEdge( e )  
    edge *e;
```

Removes the edge **e** from the edge list, frees its allocated memory and relinks the list. Note that it does not change the **Selected** or **theEdge** global variables.

```
InsertEdge( after, e )  
    edge *after, *e;
```

Places **e** after the edge **after** in the edge list.

```
TraverseEdgeList( func, arg )  
    int (func *) ();  
    int arg;
```

Applies **func** on every element of the edge list. **func** takes the form:

```
    int  
    func( edge, count, arg )  
        edge *edge;  
        int count, arg;
```

where **edge** is the edge structure, **count** is the index of the edge (starting at 0), and **arg** is the optional argument supplied in the parameter list. If **func** returns **TRUE** then the edge list traversal stops. Otherwise, **func** should return **FALSE**.

libIDraw library interface

Begin and End

(init.c)

These functions setup and end the interactive drawing package. They must be called before any other routines in the package are used. Note that **InitPort** from **libDraw** should be called *before* the call to **IDrawBegin**.

IDrawBegin(pw)
 int *pw;

Starts up the interactive drawing package. **pw** is a temporary pixwin used to hold the results of the interactive line routines. It should be at least the same size as the largest pixwin you will be using to draw. (That is, the largest pixwin sent to a call of **SetPW**) **IDrawBegin** returns 0 if there was an error, 1 if the begin was successful.

IDrawReset(pw)
 int *pw;

Restarts the interactive drawing package, allowing you to specify a new temporary pixwin if desired. Note that when this is called, all previously allocated edge structures are lost.

IDrawEnd()

Ends the interactive drawing package, freeing all of the Chap resources.

Selection functions

(find.c)

In **libIDraw**, there is the concept of the **currentEdge** and the **currentIndex**. The **currentEdge** is the edge last selected or operated on by some function, and the **currentIndex** is the last point selected on that edge. If the **currentEdge** does not exist (i.e. after an edge is deleted) the functions in **libIDraw** will not do anything, rather, a new edge needs to be selected. When an edge is created (with **AutoTrace** or **BeginLine**) it is set to the **currentEdge**. The state of the selection variables can be inquired with the functions **TheEdge()**, **EdgeSelected()**, and **TheIndex()**.

The central interactive selection function is **FindClosest()**, which will report the closest edge from some selected point. Non-interactive selection can be had with **SelectEdge()**. By using **IndexEdge()** and **TraverseEdgeList()** you can get at the edge information directly without changing the current state of the selection variables. Be careful when you do this! Don't do anything that would cause the selection state to contain invalid references.

```
FindClosest(x,y,eps)
    short x,y;
    short eps;
```

Finds the closest edge and point to **x,y** within **eps** pixels. This should be called from the mouse or stylus tracking procedure on the host whenever a pick is made. (i.e. whenever the mouse button is down) For example, the code snippet below shows how to poll the mouse and dispatch the functions:

```
/* main event loop */
for(;;) {
    TrackMouse(x,y,code);
    switch(code) {
        case MOUSE_DOWN: /* a selection is to be made */
            ChCCallVarargs(CHAP0,"FindClosest",
                CHC_INT, x,
                CHC_INT, y,
                CHC_INT, epsilon,
                CHC_NIX);
            break;

        case MOUSE_DRAG: /* now, lets do something */
            ChCCallVarargs(CHAP0,"Drag",
                CHC_INT, x,
                CHC_INT, y,
                CHC_NIX);
            break;

    }
}
[ ... ]
```

As you can see, the mouse is polled and then dispatched to the appropriate function via a **switch** statement, in this case, finding an appropriate point and dragging its position. The **Drag** function will be explained in a later section.

If a valid edge and edge point are found, the state variables are set appropriately and **TRUE** is returned, otherwise, **FALSE** is returned signifying no edge or point was found.

```
edge *
TheEdge()
```

Returns a pointer to the currently selected edge. **NULL** if none is selected.

```
int
TheIndex()
```

Returns which point in the edge is selected. (Points in the list start at 0).

EdgeSelected()

Returns **TRUE** if an edge is selected, **FALSE** otherwise.

Interactive Line Drawing (trace.c)

These functions are used for interactive line drawing. All of them use the notion of the *current pixwin* (see **libDraw**) for the drawing of their lines. These routines use the current **ForeColor** (see **libDraw**) as a pen color when drawing. To draw lines in a different color reset the foreground color. Vertices are marked with the current **MarkColor**, and are hilited with the current **HiliteColor**.

SaveLine(x1,y1,x2,y2)
int x1,y1,x2,y2;

Draws a line from **x1,y1** to **x2,y2** in the currently selected pixwin, saving the image underneath in the temporary pixwin sent to **IDrawBegin**.

RestoreLine(x1,y1,x2,y2)
int x1,y1,x2,y2;

Restores the image underneath the line from **x1,y1** to **x2,y2** in the currently selected pixwin. Note that for every copy to **RestoreLine** there should be a corresponding call to **SaveLine**. Otherwise, the restoration will contain unknown pixel values.

MarkVertex(x,y)
int x,y;

Marks the point located at **x,y**, saving the image in the temporary buffer. This is usually used to mark the end of a line created with **SaveLine**. The call to **RestoreLine** also restores the image under the marked vertex.

StartLine(x,y)
int x,y;

Allocates a new edge structure and begins the line drawer. It is usually called on a *MouseDown* event to indicate that a new edge is starting from this **x,y** location.

DrawLine(x,y)
int x,y;

Draws a line from the last location referenced (with either **StartLine** or **DrawLine**) to the new **x,y** location.

This function is called while the mouse is being dragged, usually a *MouseDrag* event.

```

DrawVLine(x,y,et)
    int x,y;
    double et;

```

Draws a line from the last location referenced (with either **StartLine** or **DrawLine**) to the new **x,y** location, using the **et** threshold to decide whether a vertex is needed or not to represent the curve accurately. (See **PutVertex** for a description of how this is done) This eliminates vertices which are within **et** pixels of a straight line projecting through the last two points on the edge. It helps reduce the number of points needed to represent a proper edge while maintaining the integrity of the true boundary.

This function is called while the mouse is being dragged, usually a *MouseDown* event.

```

EndLine(x,y,close)
    int x,y,close;

```

This ends the definition of an edge, closing it if the **close** flag is set to **TRUE**. Otherwise the edge is left open.

This function is called on a *MouseUp* event.

The suggested calling sequence for the interactive line drawing package is :

<pre> MouseDown StartLine(x,y) MouseDown DrawVLine(x,y,et) or DrawLine(x,y) MouseUp EndLine(x,y,close) </pre>

Automatic Edge Tracking (*auto.c, vertex.c*)

These functions facilitate an automatic edge walking system. An edge is created, based on a criteria function (which is consistent with the other user functions in the system) which evaluates a point in the image and then confirms or rejects it as an edge. The user must supply this function.

```

AutoTrace(tpw,x,y,et,func)
    int *tpw;
    int x,y;
    double et;
    int (*func)();

```

Activates the automatic edge tracking function. The temporary pixelwindow **tpw** contains iterative information for the edge search, and can be re-used af-

ter the call to **AutoTrace**. The search begins at **(x,y)** in the currently selected pixwin. The scanline is examined in the increasing x direction, evaluating **func** at every point. If **func** returns **TRUE** at any point in the scanline, the edge tracking starts from there, otherwise, **AutoTrace** returns unsuccessful. The criteria function **func** should return **TRUE** if the pixel at its **(x,y)** is part of an edge set, **FALSE** otherwise.

For example, a function which evaluates the edge based on a threshold would look like:

```
int
Threshold(pw,x,y,p)
    int *pw;
    int x,y;
    Pixel *p;
{
    extern int threshold;

    OnePixelRead(pw,x,y,p); /* read the value */
    if(p->chan[0] > threshold)
        return TRUE;
    else
        return FALSE;
}
```

The parameter **et** is a threshold supplied to **PutVertex** for keeping a minimal working set of points. See the definition for an explanation of its utility and suggested values.

```
int
PutVertex( x,y,et )
    int x,y;
    double et;
```

This function is called internally from **AutoTrace** and **DrawVLine**. It is used to minimize the amount of points necessary to represent an edge. It is very careful to maintain the integrity of the shape being described and hardly ever makes a compromise for detail to save space.

It will place a vertex into the current edge (which can be gotten from **TheEdge**) if the curvature of the line causes it to stray **et** pixels or more from a straight line.

et values of 1.0 to 2.0 provide good results. The higher the number, the less points are kept, and the less detail is maintained.

ResetWorkEdge()

This function is called internally to reset the state of the temporary workspace used by **PutVertex**. You should never need to call it.

Vertex Manipulation

(drag.c, ptadd.c)

These functions operate on a single vertex from the currently selected edge (both which are obtainable from **TheEdge** and **TheIndex**).

DragVertex(x,y)

int x,y;

Moves the currently selected vertex on the currently selected edge from its present location to **(x,y)**. This function is usually called during a **MouseDown** event from the window system. This operation is currently not undoable.

Calling Sequence:

<pre>MouseDown FindClosest MouseDown DragVertex(x,y)</pre>
--

AddVertex(x,y,which)

int x,y;

int which;

Adds vertex at (x,y) between two other selected vertices. I am not really happy with this interface and it may change in future versions. For now, **Add-EdgePoint** needs to be called three times, with the parameter **which** indicating which call it is. A value of **1** specifies that the first vertex has been selected with **FindClosest**. A value of **2** indicates the same with the second point. A value of **3** indicates that the co-ordinates in **(x,y)** are the co-ordinates of the new point. This operation is currently not undoable.

Calling Sequence:

```
MouseDown
    FindClosest
MouseUp
    AddEdgePoint (0, 0, 1)
MouseDown
    FindClosest
MouseUp
    AddEdgePoint (0, 0, 2)
MouseDown
    FindClosest
MouseUp
    AddEdgePoint (x, y, 3)
```

DeleteVertex()

Deletes the currently selected vertex from the currently selected edge description. This operation is currently not undoable

Calling sequence:

```
MouseDown
    FindClosest
MouseUp
    DeleteVertex()
```

DeleteIndexedVertex(index)

int index;

Deletes the **index**th vertex from the list of the current edge. Note that vertex indexes start at 0. This operation can be undone with a call to **RestoreIndexedVertex**.

This is an internal routine used by **DeleteSegment** and other routines. Call it at your own risk.

RestoreIndexedVertex(index)

int index;

Undeletes a vertex which was deleted with a call to **DeleteIndexedVertex**.

This is an internal routine used by **DeleteSegment** and other routines. Call it at your own risk.

DeleteSegment (which)
int which;

Deletes a set of vertices from the currently selected edge. This function is similar to **AddVertex** in its calling sequence. As with **AddVertex**, I am not too thrilled with the interface and it may change in the future.

DeleteSegment needs to be called three times, with the parameter **which** indicating which call it is. A value of **1** specifies that the first vertex has been selected with **FindClosest**. A value of **2** indicates the same with the last vertex in the set of vertices to be deleted. A value of **3** indicates that a third vertex has been selected for the deletion to go through. That is, it selects the direction of the deletion. All of the points from the first to the last *through* the third specified vertex will be deleted.

This operation can be undone with a call to **RestoreEdge**.

Calling Sequence:

MouseDown	FindClosest
MouseUp	DeleteSegment (1)
MouseDown	FindClosest
MouseUp	DeleteSegment (2)
MouseDown	FindClosest
MouseUp	DeleteSegment (3)

CleanEdge (et)
int et;

Removes all vertices which are within **et** pixels of other vertices in the vertex list of the currently selected vertex. This is useful for further reducing the number of vertices in an edge description.

This function can be undone with a call to **RevertEdge**.

RevertEdge ()

Undoes the effect of calls to **DeleteSegment**, **CleanEdge** and other undoable functions on the currently selected edge.

CompactEdge()

Takes all of the points marked for deletion with **DeleteSegment** and **CleanEdge** and permanently deletes them from the edge list. You should call this after every call to the undoable functions, before another non undoable function is called. Otherwise, the edge description may be left in an undesirable state.

Translation and Rotation

(xlate.c, rotate.c)

These functions allow you to change the position and orientation of the selected edge. They are not undoable at this time but will be in future versions.

XlateEdge(dx, dy)

int dx, dy;

This is an internal routine used by the edge translator. Basically, it translates the currently selected edge by **(dx, dy)**. You should never need to call **XlateEdge**.

StartXlate(x, y)

int x, y;

Sets up operations for the edge translation package. It is usually called on a *MouseDown* event.

Xlate(x, y)

int x, y;

Translates the currently selected edge from the origin specified in the call to **StartXlate** to the new origin at **(x, y)**. This function is called during a *MouseDown* event.

EndXlate(x, y)

int x, y;

Ends the translation operations for the current edge.

The suggested calling sequence is:

MouseDown
FindClosest(x, y, e)
StartXlate(x, y)
MouseDown
Xlate(x, y)
MouseUp
EndXlate(x, y)

RotateEdge(cx, cy, cost, sint)

int cx, cy;
double cost, sint;

An internal routine used by the rotation package which rotates the current edge around the center, specified by **(cx, cy)**, by the angle theta, whose trig values are specified in **cost, sint**. You should never need to call this routine.

StartRotate(x, y)

int x, y;

Begins the rotation operation. This version of rotate calculates the co-ordinates **x, y** as the center of rotation for the currently selected edge.

StartBBoxRotate(x, y)

int x, y;

Begins the rotation operation. This version of rotate calculates the center of gravity for the currently selected edge and uses it as the center of rotation.

Rotate(x, y)

int x, y;

Rotates the edge using a line from the center of the edge to **(x, y)** to compute the degree of rotation. The angle is computed from the x-axis and the line which runs from **(cx, cy)**, **(x, y)**, where **cx** and **cy** are the center of gravity of the currently selected edge.

EndRotate(x, y)

int x, y;

Finishes the rotation operation for the currently selected edge.

The calling sequence for the rotate operations is:

MouseDown

FindClosest(x, y, e)

StartRotate(x, y)

MouseDrag

Rotate(x, y)

MouseUp

EndRotate(x, y)

Edge Deleting
(delete.c)

EdgeDelete()

This function deletes an edge from the currently selected pixwin and from the edge list. Its memory is returned to the free memory pool and it cannot be recovered.

Calling Sequence:

<pre>MouseDown FindClosest(x,y,e) MouseUp EdgeDelete()</pre>
--

Edge Based Operations
(edge.c)

These functions are used to apply operations on an edge or the entire edge list.

DrawEdge(func)

int (*func)();

Applies **func** to every point on every line of the currently selected edge. See the **libDraw** introduction for a description of **func** format.

DrawVtx(func)

int (*func)();

Applies **func** only at the vertices of the currently selected edge. This is useful for marking the vertices.

FrameEdge()

This is the function used by the interactive drawing package to draw an edge and its vertices. In this version of the package, the lines are drawn in blue and the vertices are drawn in yellow. In future versions of this package the user will be allowed to select these colors.

FrameEdge frames the currently selected edge in the currently selected pixwin.

RestoreEdge()

This function has the effect of un-drawing the currently selected edge in the currently selected pixwin. It restores the original pixel values from the save buffer, erasing the line drawn.

FrameEdgeList()

Draws every edge in the edge list into the currently selected pixwin. This is useful for a 'redraw' function after some of the edges get damaged.

RestoreEdgeList()

Restores every edge in the edge list into the currently selected pixwin. Sometimes you will want to make a global operation on a picture without the lines in place, **RestoreEdgeList** restores the original image so the processing can proceed without operating on the lines.

For instance, you wish to blur the image. If you run your blurring function on the image with the lines on it the image will contain blur artifacts from the lines. You need to restore the image first, do the blur, and then frame the edges again. For example:

```
BlurImage()
{
    RestoreEdgeList();
    PrBlur33(thePW); /* your blur */
    FrameEdgeList();
}
```

Filling ***(edgefill.c)***

```
EdgeFill(tpw,x,y,func)
    int *tpw;
    int x,y;
    int (*func)();
```

Create the temporary boundry description needed by **FloodFill** in **tpw** and executes **FloodFill** at **(x,y)** using **func** in the currently selected pixwin. The contents of **tpw** can be reused after the function is completed.

EdgeFill applies **func** at every point within some enclosed boundry. In this case, the complete edge list is drawn into the temporary pixwin and the seed point is used as a starting point.