Overview

Introduction

PEXtk is a 3D graphics library that communicates with a PEX server. It is an Application Programmer's Interface to PEX, and thus it is intended to provide as much access to the functionality provided by the PEX server as possible. PEXtk requires a PEX server with the "immediate rendering" and "structure rendering" subsets.

PEXtk attempts to duplicate the functionality of existing graphics libraries and "common practice" to ease the porting of existing applications to PEX. To this end the library must maintain a database and state information on the host. Wherever possible, the geometry is cached in the server.

PEXtk is a *mixed-mode* library. This implies that the application can use both retained structures and immediate mode commands at the same time.

The following is a brief description on some fundamental concepts of PEXtk and provides some insight descriptions of how the various routines interact.

Initialization

PEXtk is initialized by invoking **Pinit** before any other PEXtk routine is called. The connection id to the X server that is returned from a successful call to **XOpenDisplay** is passed as the only argument for **Pinit**. This routine initializes PEXtk and sets the default state information. The *current drawable* also needs to be defined, and this is performed by invoking **Pset_drawable** as the next routine which is called. This routine instructs PEXtk to direct all graphics output to the specified X window.The id of the window returned from a call to **XCreateWindow** is passed as its only argument.

Termination

To terminate a PEXtk session, applications should call **Pexit**. This routine will close down the PEXtk session, and frees any acquired data.

Data Types

The following data types are used in PEXtk:

Fcoord

All coordinates are single precision IEEE floating point numbers.

Fcoord3D

Data structure containing x,y,z coordinates.

Fdata

All floating point data are single precision IEEE floating point numbers.

PCOLOR

PCOLOR is a structure that contains a color type and color data. The color type can be any that is supported by the system, such as Indexed, RGB, HSV, CIE, etc. The color data will be in a format that is determined by the color type. See the Colors Section for more information about colors.

When Colors are to be expressed as an *Indexed* color into a color table, a color table must be constructed using the **Pset_color_table** commands.

Angle

Angles are expressed as single precision IEEE floating point numbers in degrees.

Screencoord

Screen coordinates are integers that represent the screen space in *X window* relative coordinates. The range of these integers will depend on the size of the current *X window* (e.g. 1280x1024). The (x,y) values of (0,0) define the lower left hand corner of the screen.

Tvar

Structure containing 2 Fcoord variables for use as Texture mapping values, each member is a single precision IEEE floating point number.

Drawable

The *X window* which is to receive the rendered primitives must be specified before any PEXtk primitives are invoked. This is performed with the **Pset_drawable** routine. The drawable may be changed at any time so that multiple windows may be used. PEXtk supports up to *PEXtk_MAX_BUFFERS* number of drawables. The argument which is passed is the identifier of a window (or pixmap) created by a successful call to **XCreateWindow** (or **XCreatePixmap**).

Buffering

If the server supports double buffering, the update and display buffers can be controlled by **Pswap_buffers**, **Pset_update_buffer**, and **Pset_display_buffer**.
The type of window buffering may be set to the type required by the application by using **Pbuffer_mode**(*mode*). If it is not set, the buffering mode will default to be in single buffered mode. This is the same as invoking **Pbuffer_mode**(PEXtk_SINGLE_BUFFER). Single buffered mode displays primitives as they are rendered. This will give a ‘flashing’ appearance as the primitives are drawn. To give a smooth appearance, the buffering mode should be doubled buffered. This is performed by calling **Pbuffer_mode**(PEXtk_DOUBLE_BUFFER). Double buffering renders primitives to an off-screen area known as a back buffer. When the mode is switched to double buffered, PEXtk will create a back buffer if one has not already been created. PEXtk will attempt to use the Multi-Buffering extension if it exists. If it does not exist, then PEXtk will use a Pixmap for the back buffer. Note that the rendered primitives will not be visible until **Pswap_buffers** is called.
When in double buffer mode, any X geometry primitives (such as XFillRectangle, XDrawString, etc.) may be rendered to the back buffer by obtaining the back buffer id using the routine **Pget_backbuffer**(). Since the back buffer id changes at each **Pswap_buffers**() command, this routine would have to be called every time the buffers are swapped. An example of this is:

```
Pswap_buffers();
id = Pget_backbuffer();
```

An alternative method for double buffering is for the application to create the back buffer using either the Multi-Buffering extension or by creating a Pixmap. The buffer swapping would be performed by using **Pset_display_buffer** and **Pset_update_buffer**.

Windows

An *X window* is a window that has been successfully opened by the Xlib functions **XCreateWindow** or **XCreateSimpleWindow**.

PEXtk defines a *window* as the world coordinate space defined by a **Portho_2d_view,**

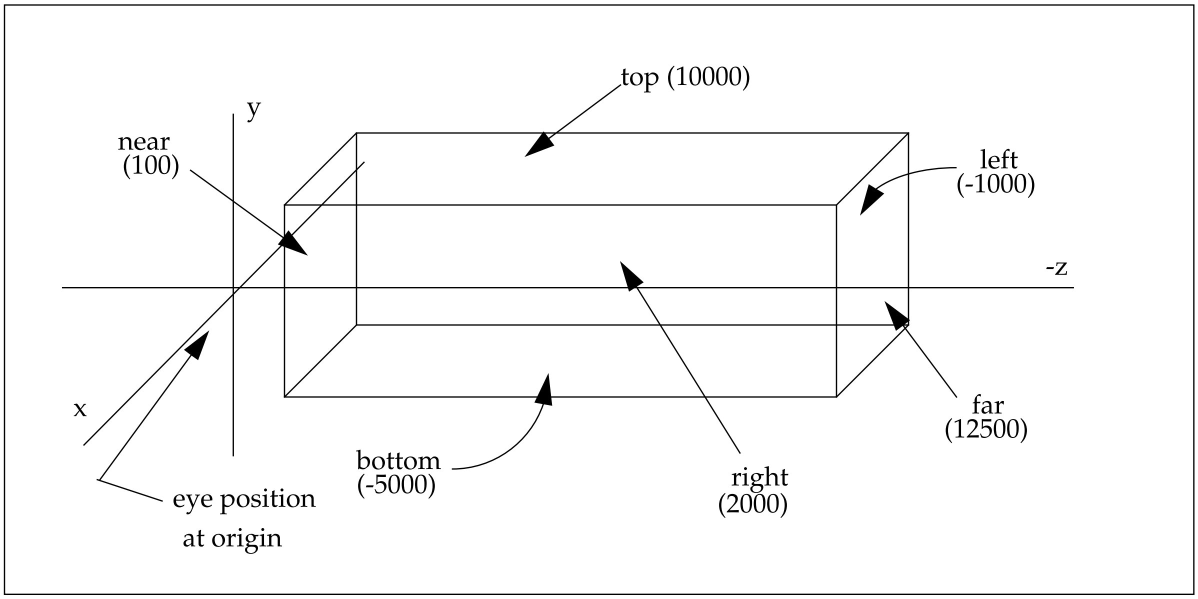

Figure 1.

Portho_view, **Ppersp_view**, or **Pwindow_view** call. A *viewport* is the area of the current *X window* that this window is mapped to using the **Pviewport** call and its coordinates are relative to the *X window*. World coordinates are clipped to the limits defined. For example, the call **Portho_view**(0, -1000.0, 2000.0, -5000.0, 10000.0, 100.0, 12500.0) will produce the clipped parallelepiped shown in Figure 1.

The **Ppersp_view**, **Portho_view**, **Portho_2d_view**, **Pwindow_view**, **Plookat_view**, or **Ppolar_view** routines can not be used inside a structure. If a structure is to contain a *window* definition, then set the *id* argument to a non zero value when defining the *window*, and use the **Pset_view_index** routine to reference that *window*. Note however, that the parameters that specify the view will essentially be "static" when stored in a structure.

Perspective projection generates a clipping area that is a frustum defined by the field of view

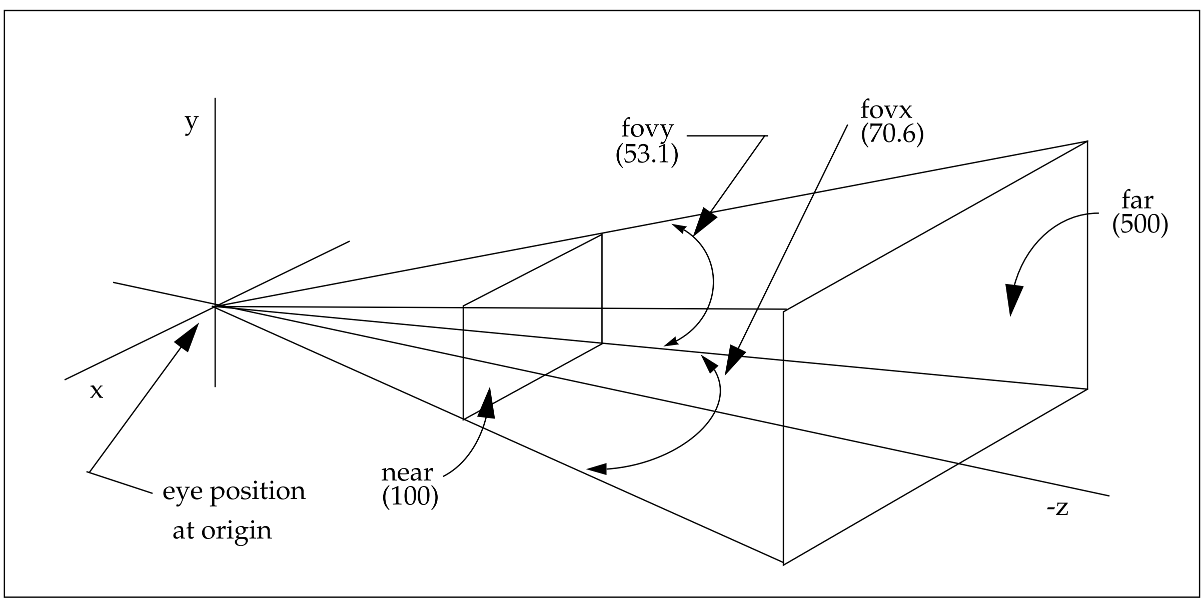

Figure 2.

and the near and far clipping planes. For example, the call **Ppersp_view**(0, 53.1, 1.33,100.0, 500.0) will produce the clipping frustum shown in Figure 2.

There are three fundamental coordinate systems used by PEXtk. The first is the *world coordinate* system. If this system is compared to PEX/PHIGS, it would be described as a combination of the *local modeling coordinate* system and the *world coordinate* system. The second is the *normalized projection coordinate* system. This system limits the x, y, z ranges to the cube $-1.0 \le x, y, z \le 1.0$. Note that for PEX/PHIGS+, this would be the unit cube $0.0 \le x, y, z \le 1.0$. The third coordinate system is the *screen coordinate* system, which is the coordinate system of a viewport. This coordinate system is the same as that used by an *X window*, and is expressed using integer values, except that the point (0,0) is at the lower left hand corner, y is positive up.

Geometric Transformations

The PEXtk rendering pipeline may be described as follows:

$$\begin{bmatrix} x' & y' & z' & w' \end{bmatrix} = \begin{bmatrix} x & y & z & w \end{bmatrix} \boldsymbol{MVPW}$$

where $\boldsymbol{M}$ is the *geometric modeling* transformation, $\boldsymbol{V}$ is the *viewing* transformation, $\boldsymbol{P}$ is the *projection* transformation, and $\boldsymbol{W}$ is the *workstation* (viewport) transformation. Note that this differs from the PEX/PHIGS+ rendering pipeline, which is:

$$\begin{bmatrix} x' & y' & z' & w' \end{bmatrix}^T = \boldsymbol{W}\boldsymbol{V}_m\boldsymbol{V}_o\boldsymbol{GL}\begin{bmatrix} x & y & z & w \end{bmatrix}^T$$

where $\boldsymbol{W}$ is the *workstation* transformation, $\boldsymbol{V}_{\mathrm{m}}$ is the *view mapping* transformation, $\boldsymbol{V}_{\mathrm{o}}$ is the *view orientation* transformation, $\boldsymbol{G}$ is the *global modeling* transformation, and $\boldsymbol{L}$ is the *local modeling* transformation.

PEXtk also provides an application with access to a client side transformation stack. There are two matrix stacks, a *modeling matrix* stack and a *viewing matrix* stack. To switch between the two, use **Pmatrix_mode**.

The method of concatenation to be performed on the stack with an input matrix may be *postconcatenation* or *preconcatenation*. If the current transformation stack has a matrix T_C, and a new matrix T_{new} is to be concatenated to it, then a *postconcatenation* is described by:

$$T_C = T_{new} \cdot T_C$$

A *preconcatenation* is described by:

$$T_C = T_C \cdot T_{new}$$

For example, to perform a scaling of an object in its own coordinate system, one would translate the object to the origin of the global coordinate system (call this T_1), perform the scaling operation (T_2), and translate it back to its original position (T_3). The logical order of the combined transformation is $T_1 \cdot T_2 \cdot T_3$, however, the order in which the matrices are to be multiplied together using the matrix stack must begin with T_3, followed by T_2, and then T_1. This is due to the fact that the matrices are concatenated together using a right to left order, and hence a transformation matrix affects everything to the left of it. Therefore, since we want matrix T_3 to affect T_2, and have matrix T_2 affect matrix T_1, then the matrices will be specified using *postconcatenation*.

Viewports

A viewport may be defined as the part of the window in which the rendered primitives will be visible. It may be the entire window, or a fraction thereof. A window may contain many viewports, and they may overlap. Note that in either immediate mode or retained mode, to view a primitive in multiple viewports requires that the application redraws (i.e., calls the primitive routine again using **Ppoly**, etc.) or re-traverses the structure or display list (i.e., by calling **Pcall_struct**, **Pexec_struct**, or **Pcall_display_list**).
In order to use the whole *X window*, a call to Xlib is required to obtain the current size of the *X window*, and then **Pviewport** should be invoked with that data. If the user is allowed to dynamically resize the *X window* (using the window manager), then a routine should be created that catches the window resize event (i.e., the XEvent) and resize the viewport accordingly.

Pclip_rectangle sets up a 2D clipping rectangle in a viewport. This rectangle clips all geometry drawn to the viewport. The **Pviewport** call will always clip to the dimensions specified, and **Pclip_rectangle** is used to specify a subregion of the viewport. For example, the following calls:

```
Pviewport(50, 350, 300, 550); /* viewport 1 */
Pviewport(100, 380, 100, 220); /* viewport 2 */
```

will generate two viewports as shown in “Window 1” in Figure 1, where viewport 1 shows the clip rectangle clipping a rectangle, and viewport 2 shows the clip rectangle clipping a triangle. The entire viewport is used, since the effective clip rectangle is (50, 350, 300, 500) for viewport 1, and (100, 380, 100, 220) for viewport 2. Note that the clip limits are at the full size of the viewport.The following calls show an example of clipping to a subregion of a viewport:

```
Pset_drawable(window2);
Pviewport(50, 330, 100, 220);
Pclip_rectangle(50, 300, 100, 190);
```

The clip rectangle in “Window 2” of Figure 1 shows this as generating a subregion of the viewport.

The **Pviewport** routine cannot be called inside a structure in PEXtk version 1.0.

Lights

PEXtk is designed to support four light types: *Ambient*, *Point*, *Infinite*, and *Spot*. The actual light types that are supported may vary among different hardware platforms. Use **Pget_support_info** to inquire about the type of lights that are supported.

An *Ambient* light is specified using **Plight_ambient**, and *Point*, *Infinite*, and *Spot* lights are specified using **Plight**. Only one *Ambient* light can be specified. An object that is illuminated only by an ambient light will appear to have no depth, since no distance information is used in the ambient light equation. For example, a ball will appear only as a flat shaded circle.

A *Point* light is specified by its color, a global x,y,z position, and an attenuation factor (see

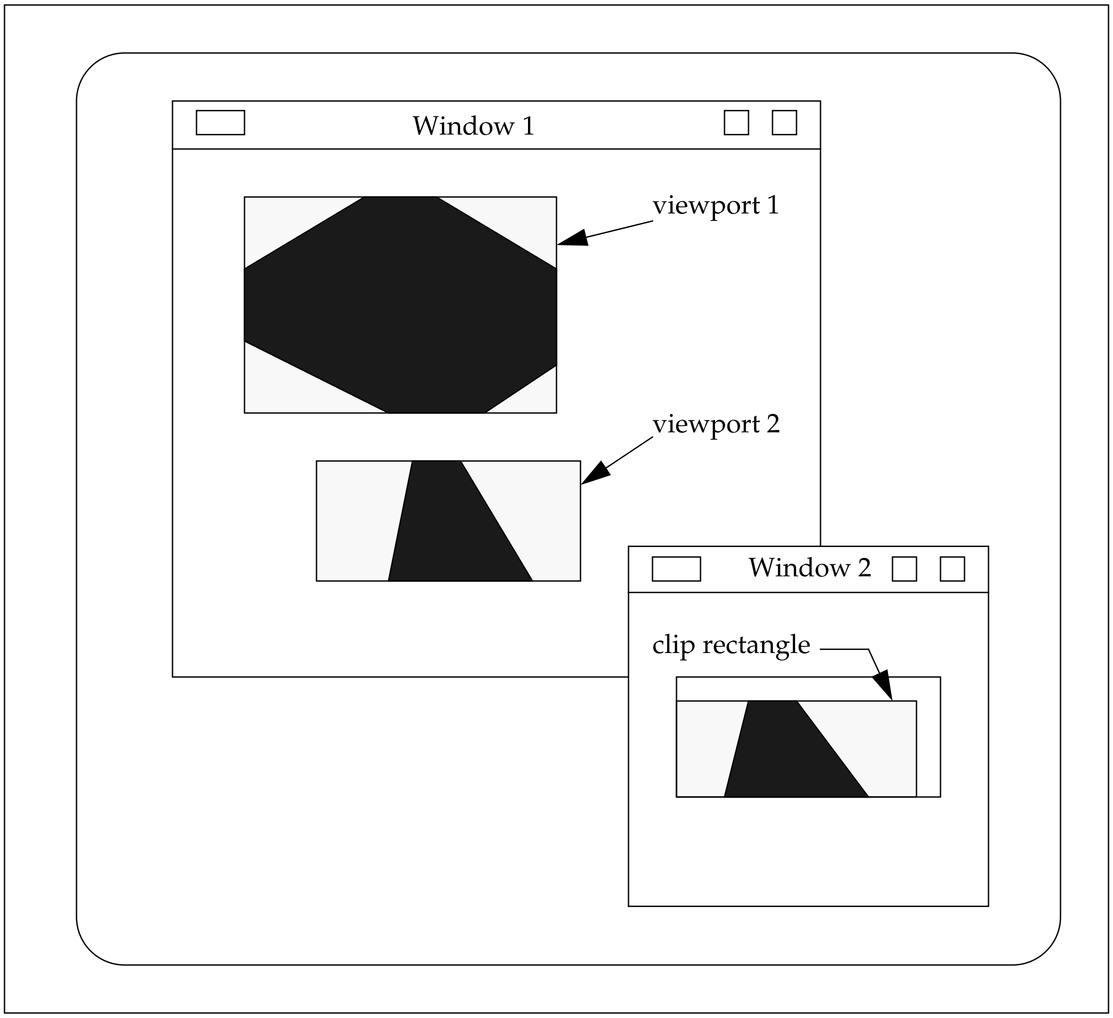

Figure 3: Viewports and Clip Rectangle

Figure 4).

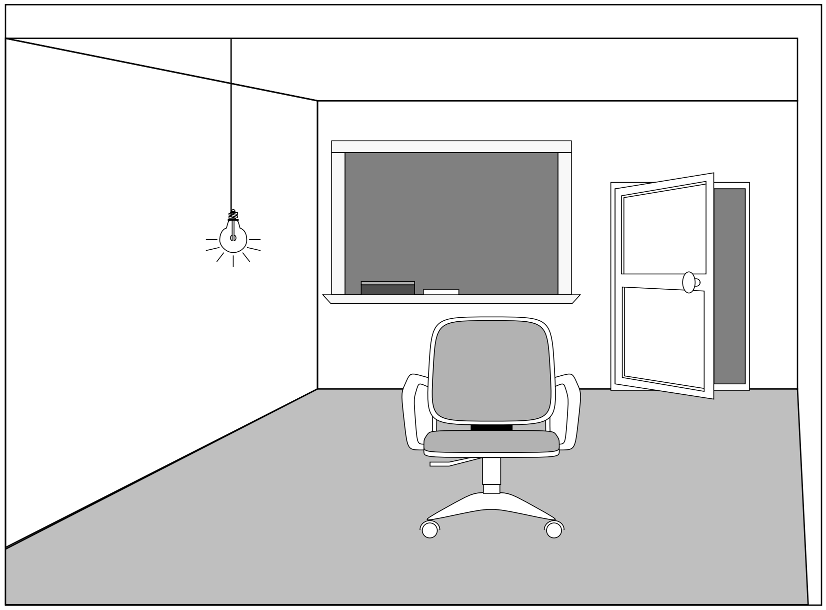

Figure 4: Light bulb as a point light

An *Infinite* light is specified by its color and a direction vector x,y,z (an example is the sun). A *Spot* light is specified by its color, a global x,y,z position, a direction vector x,y,z, an attenuation factor, and an angle of influence (see Figure 5). Note that in Figure 5 how part of the

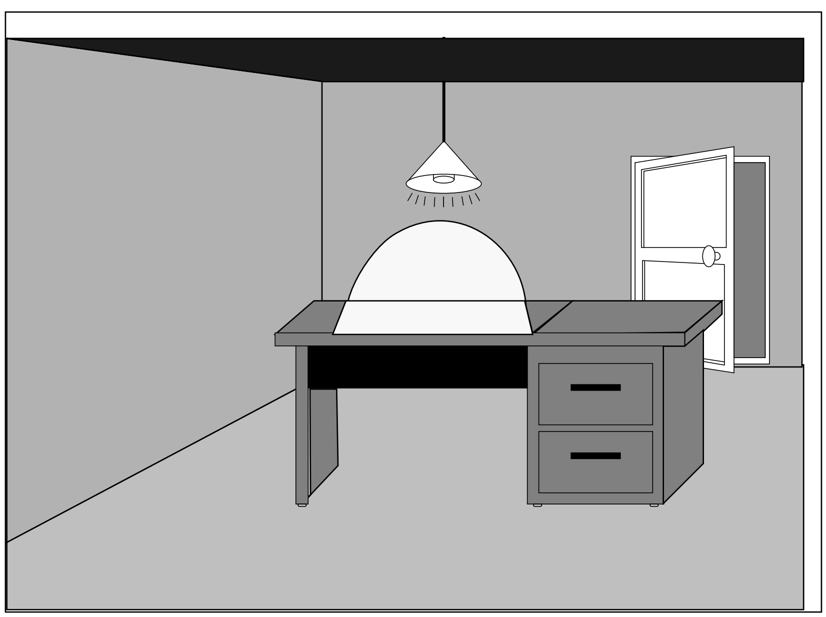

Figure 5: Light bulb as a spot light

top of the desk is illuminated directly from the light which is in the angle (or cone) of influence. The remainder of the desk and the room is illuminated by the ambient light. See Figure 6 for a description of properties for each of the supported light types.

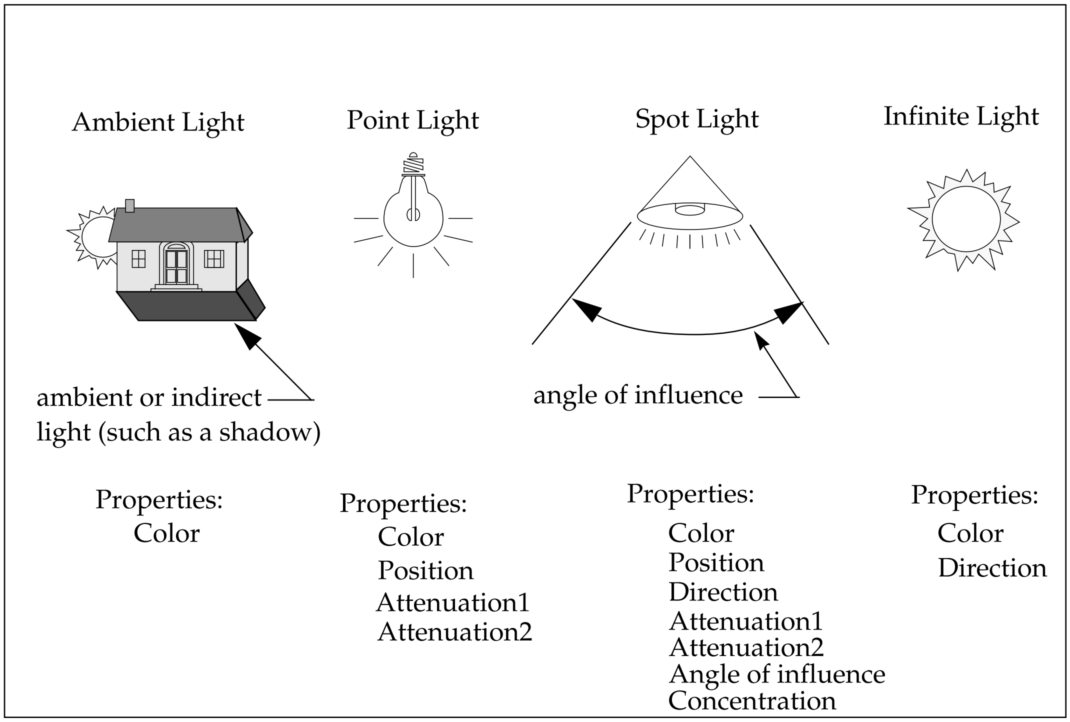

Figure 6: Supported Light Types

The effect of the light on a polygon is determined by the current shade mode, specified by the **Pshade_mode** routine, with *mode* as its argument. No lights are activated until the mode PEXtk_SHADE_LIGHTS is specified. This is true regardless of any calls to **Plight** or **Plight_ambient**. To specify a specular highlight, the specular color should be given with **Pspecular_color**, and the shading mode PEXtk_SHADE_SPECULAR should be *or*ed with *mode*. This attribute may be turned on or off as desired.

The *ambient light* intensity is calculated using:

$$I_a = K_a O_C L_C$$

where I_a is the ambient intensity, K_a is the ambient coefficient, O_C is the object's intrinsic color (from the **Pcolor** routine), and L_C is the ambient light color.

The *infinite light* intensity is calculated using $I_v = I_{vd} + I_{vs}$, where:

$$I_{vd} = K_d O_C \sum_{i=0}^{i<n} L_C (\vec{N} \cdot ((-\vec{L}_{dir})))$$

$$I_{vs} = K_s S_C \sum_{i=0}^{i<n} L_C (\vec{R} \cdot \vec{V})^{S_{conc}}$$

The diffuse component is computed using the diffuse coefficient K_d times O_C, times the sum of each enabled light color L_C multiplied by the product of the cosine of the angle between the surface normal $\vec{N}$ and the negation of the enabled light source direction vector $(\vec{N} \cdot (-\vec{L}_{dir}))$. The specular component is computed using the specular coefficient K_s time the specular color S_C, times the sum of each enabled light color L_C, which is multiplied by the product of cosine of the angle between the direction of the peak highlight from the light source $\vec{R}$ and the vector from the object to the viewing position $\vec{V}$ raised to the power of the specular coefficient S_{conc}.

The *point light* intensity is computed similarly, but has a light attenuation factor, L_{ATT}

$$I_{pd} = K_D O_C \sum_{i=0}^{i<n} L_C (\vec{N} \cdot (-\vec{L}_{dir})) L_{ATT}$$

$$I_{ps} = K_s S_C \sum_{i=0}^{i<n} L_C (\vec{R} \cdot \vec{V})^{S_{conc}} L_{ATT}$$

This equation is similar to the infinite light equation, except that $(-\vec{L}_{dir})$ is a vector in the direction of the light source, and both $\vec{L}_{dir}$ and $\vec{R}$ must be recalculated for each point to be shaded. The light attenuation factor L_{ATT} is calculated using:

$$L_{ATT} = \frac{1}{L_{A1} + L_{A2}(\| O_{pos} - L_{pos} \|)}$$

where L_{A1} and L_{A2} are light attenuation factors specified for each enabled light using **Plight**, and $\| O_{pos} - L_{pos} \|$ is the magnitude, or distance, between the object and the light. If the specified values for L_{A1} and L_{A2} are both zero, then PEXtk uses their default values.

The *spot light* intensity is computed using

$$I_{sd} = K_d O_C \sum_{i=0}^{i<n} L_C (\vec{N} \cdot (-\vec{L}_{dir})) (\vec{O}_L \cdot \vec{L}_{dir})^{L_{conc}} L_{ATT}$$

$$I_{ss} = K_s S_C \sum_{i=0}^{i<n} L_C (\vec{R} \cdot \vec{V})^{S_{conc}} (\vec{O}_L \cdot \vec{L}_{dir})^{L_{conc}} L_{ATT}$$

The term $\vec{O}_L$ is a vector from the object to the position of the spotlight L_{pos}, and $\vec{L}_{dir}$ is the direction vector that the light is pointed.

Shading Models

PEXtk supports 10 different shading modes, 4 of which are *shading models*. Some of the modes affects all polygon and polyline primitives, and the others apply only to the **Ppoly_index**, **Ppoly_fill_area** and **Ppoly_with_data** routines. PEXtk also provides primitives which give direct support of these shading models. These are listed in Table 1, in addition to the other shading models which are supported. The shading model is specified with the routine **Pshade_mode**(*mode*), where the *mode* is the only argument, given as a bitmask. The shading model PEXtk_SHADE_SPECULAR turns on specular highlights. This mode affects all polygon primitives.

PEXtk #define type (bitmask)	Description	Routines affected	Direct support routine
PEXtk_SHADE_NONE	No attributes defined	†	
PEXtk_SHADE_WIRE	Wireframe	†	**Ppoly**
PEXtk_SHADE_HIDDENLINE	Hiddenline	†	
PEXtk_SHADE_FLAT	Flat or constant	†	**Ppoly_fill**
PEXtk_SHADE_COLORS	Gouraud	†	**Pvertex_color** **Ppoly_shade**
PEXtk_SHADE_DOTPRODUCT	"Pseudo" Phong	†	**Ppoly_shade_normal**
PEXtk_SHADE_NORMALS	Phong	†	
PEXtk_SHADE_ANTIALIAS	Antialiased lines	**Ppoly_line**	
PEXtk_SHADE_LIGHTS	Enable lights	All polygon primitives	
PEXtk_SHADE_SPECULAR	Enable specular highlights	All polygon primitives	

† **Ppoly_index**, **Ppoly_fill_area** and **Ppoly_with_data**

Table 1: Supported Shading Modes

The shading models which are discussed here affect the routines **Ppoly_index**, **Ppoly_fill_area** and **Ppoly_with_data** only. The shading model PEXtk_SHADE_WIRE will render **Ppoly_index**, **Ppoly_fill_area** and **Ppoly_with_data** polygons as a wireframe (outline only) primitive. This mode is similar to using the **Ppoly_line** routine with the last vertex duplicated. The mode PEXtk_SHADE_COLORS will render the polygons as color interpolated polygons. This mode is sometimes inappropriately referred to as "Gouraud Shading". In this mode, colors are computed at the vertices of a polygon. These colors are then linearly interpolated across the interior of the polygon. For polylines, the color is computed by linear interpolation between the vertices when they are provided using **Pvertex_color_array**. The shading mode PEXtk_SHADE_DOTPRODUCT will calculate the lighting equation dot products at the vertices.

These dot products are then linearly interpolated and the light source shading computation is applied using these values to compute the color value for each pixel in the interior of the polygon. This method is intermediate in complexity between Gouraud and Phong shading, and is sometimes referred to as "Pseudo Phong". The shade mode PEXtk_SHADE_NORMALS will compute the normal at each pixel in the interior of the polygon, and then perform the light source shading computation using the computed normal.

The shading mode PEXtk_SHADE_HIDDENLINE will render polygons as a wireframe mesh, but with hidden surfaces removed. However, PEXtk_SHADE_WIRE should not be specified with PEXtk_SHADE_HIDDENLINE (since the polygons are actually solid with the filled color as the background color, wireframe mode will be turned off).
The shading mode PEXtk_SHADE_ANTIALIAS will render all polyline primitives using antialiased lines. If the mode PEXtk_SHADE_WIRE is not specified, it will be turned on.
The shading mode PEXtk_SHADE_SPECULAR will turn on specular highlights. A specular highlight appears on shiny objects. For example, if a shiny red apple is illuminated by a bright white light, a specular highlight will appear as a bright white spot, and the rest of the apple will be red (the diffuse color). The part of the apple which is not directly illuminated by the white light is illuminated by the ambient light (activated by **Plight_ambient**).

Structures

A structure is a collection of polygon and polyline primitives stored together on the server side. Not all of the PEXtk routines are available for server side storage. When PEXtk primitives are invoked between a **Pcreate_struct** and **Pclose_struct**, they are inserted into the named structure. The *name* for the structure is given as an argument to **Pcreate_struct**. The execution of the commands given between these two routines is deferred until the structure is traversed. This is called *retained structure* mode. A structure is traversed when it, or a structure that refers to it, is executed by invoking **Pexec_struct** or **Pcall_struct** in *immediate* mode or from inside a Display List (see below). An object hierarchy can be constructed by using **Pcall_struct** within a structure. To ease the construction of complicated hierarchies, a **Pcall_struct**(x) may be inserted into a structure y before x is defined. No error will occur provided that x exists before y is traversed. A property of structures is that elements can be edited without rebuilding the whole structure.
PEXtk version 1.0 currently does not support the state saving **Ppush_attributes**, **Ppush_viewport**, and **Ppush_matrix**, and the state restoring **Ppop_attributes**, **Ppop_viewport**, and **Ppop_matrix** when invoked from inside a structure. The current attribute, viewport, and matrix state are automatically saved when a **Pcall_struct** command is executed. In the current revision of PEXtk, **Pexec_struct** is the same as **Pcall_struct**. In a future revision, **Pexec_struct** will not save the current attribute, viewport, and matrix state. This would be done explicitly by calling **Ppush_attributes**, **Ppush_viewport**, and **Ppush_matrix**. The state may be restored by calling **Ppop_attributes**, **Ppop_viewport**, and **Ppop_matrix**. An example is:

```
Ppush_matrix();
Ppush_attributes();
Pexec_struct(name);
Ppop_attributes();
Ppop_matrix();
```

or

```
Pcreate_struct(n);
Ppush_attributes();
Ppush_matrix();
... /* geometry definition */
Ppop_attributes();
Ppop_matrix
Pclose_struct();
```

This implies that the transformation matrix and attributes may change farther down the hierarchy without affecting the geometry at the current level.

Editing Structures

Structures may be opened for editing with **Pedit_struct**. Elements within the structure can be replaced or elements can be added to the end of the structure or after a given label within a structure.

Display Lists

Display List Storage, or DLS, is essentially a database of commands stored on the client side for later execution. When commands are given between the **Pcreate_display_list** and **Pclose_display_list** routines, they are inserted into the named Display List. The *name* is given as an argument to **Pcreate_display_list**.
The execution of the commands between these two routines is deferred until the Display List is traversed. A Display List is traversed when it, or a Display List that refers to it, is executed by invoking **Pcall_display_list** in *immediate* mode. An object hierarchy can be constructed by using **Pcall_display_list** within a Display List. To ease the construction of complicated hierarchies, a **Pcall_display_list**(*x*) may be inserted into Display List *y* before *x* is defined. No error will occur provided that *x* exists before *y* is traversed. There are currently no editing functions available for Display Lists, but they will be provided in a follow-up revision. The interface will be similar to the structure editing routines. Since a display list is data stored on the client side, no performance gains will be achieved using them, they are for convenience only.

Geometry

PEXtk supports a variety of geometry types and geometry building primitives. In addition, it also supports two different geometric display paradigms. One is the direct specification of a primitive in regards to its shading model, such as **Ppoly_shade** for a "Gouraud shaded" polygon, or **Ppoly_fill** for a flat shaded polygon. The other paradigm is specifying the geometry and then changing its appearance before the primitive is actually rendered. This is performed in *immediate* mode by specifying the shading mode and then calling **Ppoly_index**, **Ppoly_fill_area**, or **Ppoly_with_data**, or in *retained structure* mode by building a primitive with those routines, and then changing the shading mode with **Pshade_mode** before the structure is executed. This paradigm has more meaning in *retained structure* mode, since geometry may be built once, and its appearance changed by the application for various purposes. For example, an application could switch from a "Gouraud shaded" mode to wireframe when rotating a large object to provide more interactivity and better response time.
Geometry can be built vertex by vertex using the **Ppoly_point**... routines. To begin, a **Pbegin_line**, **Pbegin_poly**, or a **Pbegin_tmesh** routine is called. Colors may be assigned to each vertex using the **Pvertex_color**... commands. For efficiency, the library will create a staging area where the geometry is built. After completion (i.e., a **Pend_**... routine is called) the geometry will be placed in the currently opened structure, or in immediate mode, it will be sent to be rendered.
Geometry may also be built as polygons by providing an array of vertices and using the **Pvertex**... commands. Colors may be assigned to each polygon vertex using the **Pvertex_color...** or **Pvertex_color_array**... commands.
For special polygons, the command **Ppoly_with_data** can be used. This allows an arbitrary number of bytes of data to be attached to each vertex.
PEXtk also supports marker primitives with the **Pmarker** routine. A marker is a two-dimensional symbol representing a location in three-dimensional space. The marker position is specified in modeling coordinates, similar to the polyline and polygon primitives. The type of the marker is specified with **Pmarker_type**, the color of the marker is specified with **Pmarker_color**, and its relative size is specified with **Pmarker_scale**.

2D Geometry

All 2D geometry is handled by the 3D geometry pipe as 3D coordinates with the z coordinate equal to 0.

X window geometry, such as XFillPolygon, XDrawLine, etc., may be used along with PEXtk primitives, by specifying the *drawable* to be the same as that given with **Pset_drawable**. For double buffered mode, however, the primitives must be placed in the back buffer, since a swap buffer call (**Pswap_buffers**) will copy over what is in the front buffer. The current back buffer id may be obtained with **Pget_backbuffer**. Note that this id will change at every **Pswap_buffers** call.

Text

The text drawing routines **Ptext** and **Ptext_annotate** both display text as stroke text (text composed of short lines). Their position and orientation are affected by the current transformation matrix. The font to be used is set by **Ptext_font**, and the text direction is specified using **Ptext_path**. The size of the text may be scaled using **Ptext_size** or **Ptext_annotate_size**.

Ptext_annotate displays text that is always parallel to the viewing plane. Attributes that affect **Ptext_annotate** are **Ptext_orientation**, **Ptext_path**, **Ptext_annotate_size**, **Ptext_font**. The routine **Ptext_orientation** takes an array of two Fcoords as its argument to specify the positive y direction to define the up position, in normalized coordinates. Figure 7 displays the coordinate system for annotation text. **Ptext_path** specifies the direction the text is to be

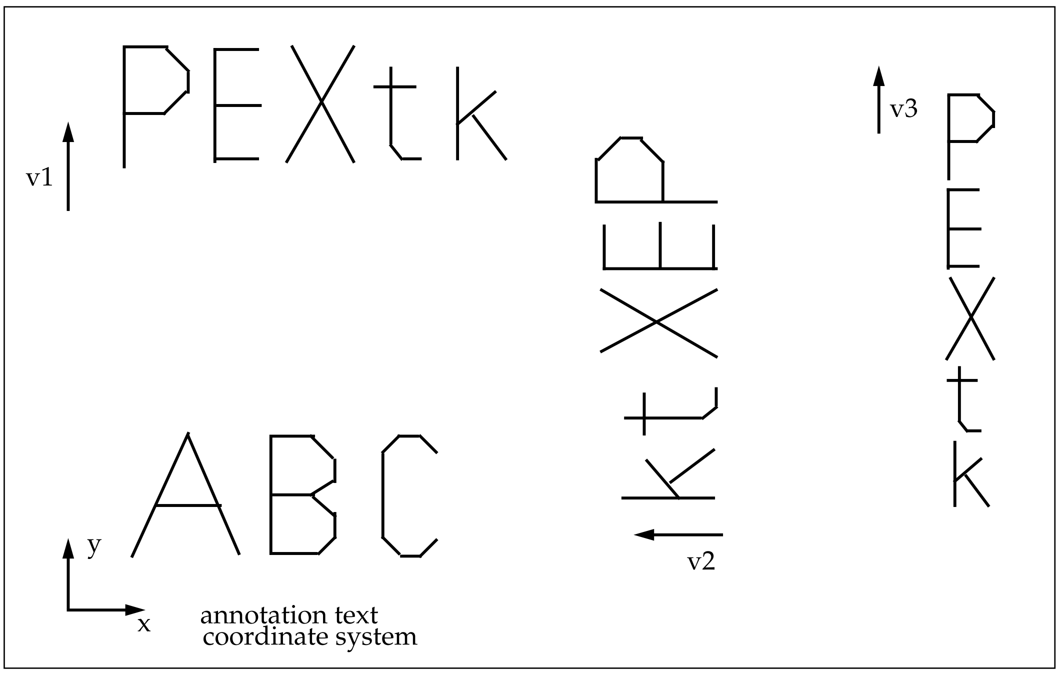

Figure 7: Text orientation

drawn, such as up, down, left or right.The vector $\vec{v}1$ shows an orientation of v1[0] = 0.0, v1[1] = 1.0, and a text path of right, which displays the text in a horizontal alignment. The vector $\vec{v}2$ shows an orientation of v2[0]=1.0, v2[1]=0.0, and a text path of left, which displays the text in a vertical alignment and backwards. The vector $\vec{v}3$ shows an orientation of v3[0]=0.0, v3[1]=1.0, and a text path of down, which displays the text in a vertical downward alignment.

Ptext displays text that is not necessarily aligned parallel to the viewing plane. Attributes that affect **Ptext** are **Ptext_orientation**, **Ptext_path**, **Ptext_size**, **Ptext_font**. The text is defined in the x-y plane, and transformed using the any of the transformation matrix routines. **Ptext_orientation** specifies the position y direction as it does for **Ptext_annotate**. Figure 8 displays some examples for **Ptext**. The ABC in Figure 8 shows an isometric view of the 3D text coordinate system. The vector $\vec{v}4$ shows a direction of v4[0] = 0.5, v4[1] = 0.0,

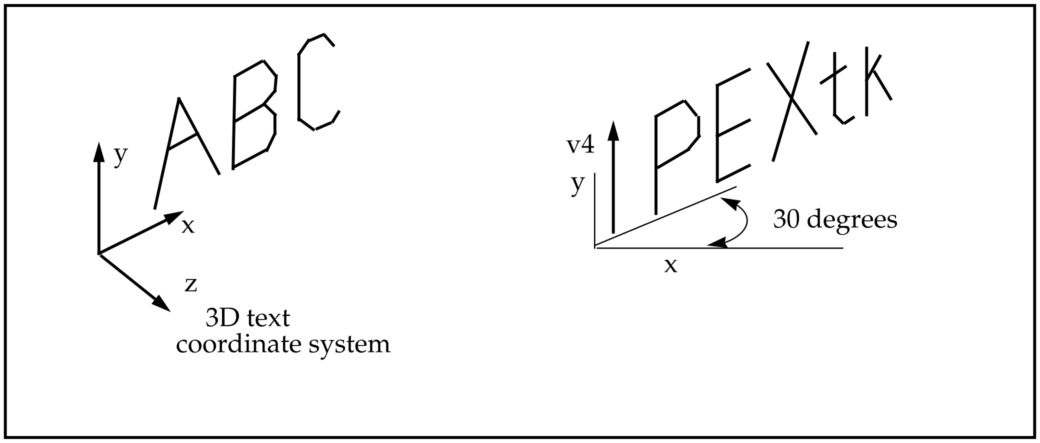

Figure 8: 3D Text

v4[2] = 0.5, and the text is rotated about y 30 degrees, and around z 40 degrees.

Colors

Colors may be specified using the PCOLOR structure, which supports *Indexed* color, *RGB* floats, *HSV* floats, *HLS* floats, and *CIE* floats. The PCOLOR structure is passed as a pointer to **Pcolor**, **Pline_color**, **Pclear**, **Plight_color**. and **Ptext_color**. The same routines appended with **_rgb** or **_hsv** will accept RGB floats or HSV floats, respectively. PEXtk currently converts all color types to RGB floats, except when in PEXtk_COLOR_INDEXED mode.If the color mode is PEXtk_COLOR_INDEXED, then and index value may be passed to the same routines appended with **_ind**.

PEXtk provides a colormap utility, **Pset_color_equation**, for use on 8-bit color systems. This utility will generate a colormap, or set the colormap if one is passed in.

Attributes

An *attribute* is an item which modifies the appearance of a geometric primitive. These items are saved and restored by **Ppush_attributes** and **Ppop_attributes**, respectively. An *attribute* is any item specified with the following routines:

Pcolor	**Pline_color**	**Plight_color**	**Pspecular_color**
Pclear	**Pborder**	**Pline_width**	**Pline_type**
Ptext_font	**Ptext_size**	**Pbackface**	**Ptext_orientation**
Ptext_color	**Ptext_path**	**Preflection_property**	
Pmarker_type	**Pmarker_scale**	**Pmarker_color**	**Ptext_annotate_size**
Ptexture_info	**Ptexture_name**		

Note that the associated routines with an **_rgb**, **_hsv**, or **_ind** are also considered attributes.

Texture Mapping

The PEXtk API provides the interface necessary to support texture mapping. However, current implementations of PEX do not provide texture mapping support. There are currently proposals in progress specifying methods on how PEX can support this feature, and these should be available in future releases of PEXtk.

Support Functions

PEXtk provides hooks for applications to specify routines to use for various functions, which are listed in Table 2. Although the hooks are primarily designed for a hardware

Support	PEXtk #define type	Description
Create back buffer	PEXtk_FUNC_CREATE_BUFF	Creates the back buffer to be used for double buffering
Delete back buffer	PEXtk_FUNC_DELETE_BUFF	Deletes (frees) the back buffer
Swap Buffers	PEXtk_FUNC_SWAP_BUFF	Swaps the back buffer to be the display buffer in double buffer mode
Color mapping	PEXtk_FUNC_COLOR_MAP	Calculates the color index from an RGB, or HSV trio of floats
Dithering	PEXtk_FUNC_DITHER	Sets the dither function
Transparency	PEXtk_FUNC_TRANSPARENCY	Sets the transparency rendering function

Table 2: Supported functions

manufacturer to specify their own hardware specific routines for these functions, an interface is provided for the application to set the support functions to their own specific routines. Most applications will not need to use this functionality, however.

High Performance Rendering

To achieve the highest performance possible, geometry should be placed in a server side structure, which is created using **Pcreate_struct**. The data will therefore be stored on the server side, and does not have to be sent 'down the wire' again when the geometry is to be displayed.

The **Ppoly_fill**, **Ppoly_shade**, **Ppoly_line** routines will provide the best possible performance. The geometry building routines, (**Pbegin_line**, **Pbegin_poly**, **Pbegin_tmesh**, and **Ppoly_point**...) should be avoided, since they must use a data cache to buffer the data before sending it to the server.

Open Issues

PEXtk currently does not completely address the following issues: Picking, Parametric Curves and Surfaces, and Error handling. The specification for these items is now in progress. A NURBS interface to the PEX server is provided with the **Pnurb_curve** and **Pnurb_surface** routines. The Picking interface will be similar to the following:

```
Pbegin_pick(x, y, xsize, ysize, buf, buf_size);
Ppick_id(10); /* set the id for geometry to be picked */
```

```
Ppoly(n, pts); /* specify geometry to be picked */
.
.
cnt = Pend_pick( buf, &more_hist);
```

A pick session will start with **Pbegin_pick**, with (*x*,*y*) specifying the center point of the pick box, and (*xsize*,*ysize*) will be the dimensions of the pick box (the actual shape of the pick box is implementation dependant). The **Pend_pick** routine will stop the picking and return the number of items picked, and a list of the pick ids in *buf*.