FINAL

IN-09

358127

User Interface Technology Transfer to NASA's Virtual Wind Tunnel System

Final Report

Grant #NCC 2-5213
April 1, 1997 - March 31, 1998

Brown University Computer Graphics Group
Department of Computer Science
PO Box 1910, Brown University
Providence, RI 02912

Principal Investigator: Andries van Dam

To: CASI

1 Project Summary

Funded by NASA grants for four years, the Brown Computer Graphics Group has developed novel 3D user interfaces for desktop and immersive scientific visualization applications. This past grant period supported the design and development of a software library, the 3D Widget Library, which supports the construction and run-time management of 3D widgets. The 3D Widget Library is a mechanism for transferring user interface technology from the Brown Graphics Group to the Virtual Wind Tunnel system at NASA Ames as well as the public domain.

2 Overview of the 3D Widget Library

The 3D Widget Library is a light-weight library built on top of OpenGL for creating and interacting with 3D widgets. The core of the library is a set of building blocks for constructing custom 3D widgets and functions that handle direct manipulation of the widgets. In addition to the basic widget building blocks, the library supports interactive shadows and gestural camera navigation controls (i.e., zooming, virtual trackball rotation, and film-plane translation) using a single mouse button.

The library can be used in many situations. For example, in scientific visualization applications, the 3D widgets can be linked to visualization techniques such as streamlines, colorplanes, isosurfaces, etc. The 3D widgets provide mechanisms for controlling parameters such as an object's 3D position and orientation, and non-spatial parameters such as the number of streamlines shown. Mouse buttons, possibly in combination with keyboard modifiers, and mouse motions drive "behaviors" on the 3D widgets. In the example included with the library, the left mouse button invokes virtual sphere rotation of objects, the middle button invokes film-plane translation of objects, and the right button performs three fundamental camera operations (film-plane translation, virtual sphere rotation, and zoom). The 3D Widget Library contains a file[1] that shows in a step-by-step manner how a rake widget[2] is created.

The enclosed programmer's manual provides further details on the components of the library and working examples.

3 Current Status and Future Plans

We are currently working with researchers to integrate the 3D Widget Library into the virtual wind-tunnel system at NASA Ames. Another group at NASA Ames (Tim Sandstrom's) and groups at other institutions (the SCIRun group at the University of Utah and the Pv3 group at MIT) are considering integrating the library with their visualization systems.

[1] See: `widgetlib/rakewidget.design.example/rakewidget.C`

[2] A "rake" is a visualization tool used in scientific visualizations. It consists of a long, thin bar from which streamlines are emitted. A rake widget is a 3D virtual object that performs the same function as its real-world counterpart allowing a scientist to explore a 3D dataset. Controls for the rake's parameters (e.g., size, position, and number of streamlines emitted) are represented by geometric objects attached to the rake widget which the user adjusts through direct manipulation.

In the future, we plan to modify the library based on user feedback. In addition, we will extend the functionality of the library, integrating aspects of our current research on interaction in 3D immersive and semi-immersive environments, thus extending the style of interaction techniques beyond conventional desktop interaction.

4 Publicity

The NSF Science and Technology Center for Computer Graphics and Scientific Visualization[3] (the STC) ran a booth at the Visualization 97 conference in October 1997. Handouts describing the 3D Widget Library were prepared (see enclosed copy) and distributed at the conference. Several inquiries and downloads of the library resulted from the conference. In addition, we have demonstrated the library and made its availability known to many visitors of our graphics lab.

5 FTP Address for the 3D Widget Library

The latest version of the 3D Widget Library is available at:

```
ftp://ftp.cs.brown.edu/u/asf/widgetlib/widgetlib.tar.gz
```

or

```
ftp.cs.brown.edu
<< anonymous login >>
cd /u/asf/widgetlib
binary
get widgetlib.tar.gz
```

See attached programmer's manual for detailed information. The 3D Widget Library contains a README file that explains how to run the demo program and integrate the library with an existing system.

6 Brown Personnel

The Brown Graphics Group, directed by Professors Andries van Dam and John F.

[3] An NSF funded five-site Science and Technology Center researching computer graphics topics such including scientific visualization, user interfaces, virtual reality, rendering techniques, and graphics hardware. Members include Brown University, the University of Utah, the California Institute of Technology, Cornell University, and the University of North Carolina at Chapel Hill.

Hughes, is a team of Ph.D., Masters, and undergraduate students and full-time staff. Professor van Dam is also currently the Director of the NSF Science and Technology Center for Computer Graphics and Scientific Visualization. He and John Hughes are co-authors of the standard computer graphics textbook, *Computer Graphics, Principles and Practice*, along with James Foley and Brown Ph.D. Steven Feiner. Van Dam is a co-founder of ACM SIGGRAPH and co-founder and first chairman of Brown University's Computer Science Department. The full-time staff of the Graphics Group includes the Director of Research (Bob Zeleznik), a Research Scientist (Timothy Miller), a User Interface Developer (Andrew Forsberg), an Educational Outreach Director (Anne Morgan Spalter), and a Software Engineer/ Researcher (Loring Holden). A number of graduate and undergraduate students complement the group by assisting on various research projects. The Media Coordinator (Mark Oribello) and three part-time students support computers, video-teleconferencing, and the group's other AV equipment. Andrew Forsberg was the principle researcher being funded by this grant.

7 Facilities and Equipment at Brown

The facilities at Brown include a variety of workstations from HP, DEC, Sun and SGI. Our Virtual Reality Lab contains a Fakespace Labs BOOM, a Virtuality Visette Pro Head-Mounted Display, a Virtual Technologies CyberGlove, and an Ascension extended-range Bird tracker. We also use a StereoGraphics VR setup (LCD-shutter glasses and two Logitech 3D mice). An Active Desk built by Input Technologies, Inc. (ITI) was recently donated to our lab by Alias/Wavefront. We have two Phantom haptic feedback devices made by Sensable Technologies. We also have a teleconference system which uses a dedicated T1 line to connect us to the four other sites of the NSF STC Center for Computer Graphics and Scientific Visualization. A full audio/video non-linear editing system is used to record footage directly from workstation screens and to edit videotapes.

We also maintain a World Wide Web site which contains general information about our group and research projects:

```
http://www.cs.brown.edu/research/graphics/
```

8 Acknowledgments

During the grant period, the Brown Graphics Group was sponsored by the following: NASA, the NSF Science and Technology Center for Computer Graphics and Scientific Visualization, Advanced Network and Services, Autodesk, Microsoft, Sun, SGI, and TACO.

The 3D Widget Library

Version 1.2
June 30, 1998

Table of Contents

1 Overview

The 3D Widget Library is a light-weight library for creating and interacting with 3D widgets. The core of the library is a set of building blocks for constructing custom 3D widgets and functions that handle direct manipulation of the widgets. In addition to the basic widget building blocks, the library supports interactive shadows and gestural camera navigation controls (i.e., zooming, virtual trackball rotation, and film-plane translation) using a single mouse button.

The library can be used in many situations. For example, in scientific visualization applications, the 3D widgets can be linked to visualization techniques such as streamlines, colorplanes, isosurfaces, etc. The 3D widgets provide mechanisms for controlling parameters such as an object's 3D position and orientation, and non-spatial parameters such as the number of streamlines shown. Mouse buttons, possibly in combination with keyboard modifiers, drive "behaviors" on the 3D widgets. In the example included with the library, the left mouse button invokes a virtual sphere rotation on objects, the middle button invokes film plane translation, and the right button performs three fundamental camera operations (film-plane translation, virtual sphere rotation, and zoom). The 3D Widget Library contains a file[1] that shows in a step-by-step manner how a rake widget[2] is created.

The library is written in C++ and is built on top of Open GL.

2 FTP Address for the 3D Widget Library

The latest version of the 3D Widget Library is available at:

```
ftp://ftp.cs.brown.edu/u/asf/widgetlib/widgetlib.tar.gz
```

or, via anonymous FTP at:

```
ftp.cs.brown.edu
cd /u/asf/widgetlib
binary
get widgetlib.tar.gz
```

3 3D Widget Library Components

The 3D Widget Library consists of two main components: widget interaction and

[1] See: `widgetlib/rakewidget.design.example/rakewidget.C`

[2] A "rake" is a visualization tool used in scientific visualizations. It consists of a long, thin bar from which streamlines are emitted. A rake widget is a 3D virtual object that performs the same function as its real-world counterpart allowing a scientist to explore a 3D dataset. Controls for the rake's parameters (e.g., size, position, and number of streamlines emitted) are represented by geometric objects attached to the rake widget which the user adjusts through direct manipulation.

management (e.g., drawing and picking) as well as camera interaction. The following sections describe each in further detail. Examples of how to use the 3D Widget Library can be found in Section 4 and in the library distribution (in the `src/` and `rakewidget.design.example/` directories).

3.1 Widget Manager

CLASS: **WidgetManager**

Purpose: To handle global operations on the widgets, such as drawing all the widgets, and passing mouse events to each widget for selecting and moving them.

METHODS: **static void add (SimpleWidget *w)**

Adds (i.e., registers) a widget to the list of widgets the WidgetManager knows about.

static void remove (SimpleWidget *w)

Removes (i.e., unregisters) a widget from the list of widgets the WidgetManager knows about.

static void draw ()

Draw all the widgets registered with the WidgetManager.

static void find2d (double x, double y)

Select a widget or part of a widget for manipulation. The programmer using the 3D Widget Library is responsible for calling this method. This is normally called while the mouse is moving freely around. (See the file `widgetlib/src/main.C` for example usage.)

static void grab2d (double x, double y, int but, int mod=0)

Move the part of a widget that was previously selected in the call to `find2d`, if any part was selected. The programmer using the 3D Widget Library is responsible for calling this method. (See the file `widgetlib/src/main.C` for example usage.)

3.2 XFobs

CLASS: **XFobs**

purpose: An observer of changes to an object's xform.

METHODS: **virtual void notify(wcomp *w, Cmat3 &t)**

Subclasses of `XFobs` register themselves with an object that maintains a list of `XFobs`'s. `notify(..)` is called when the transformation matrix of a widget component (i.e., `wcomp`) is modified. The paraemters passed in are a pointer to the widget component modified and the transformation matrix by which is was modified.

3.3 Widget3D

CLASS: **Widget3D**

DERIVES FROM: XFobs

METHODS: **virtual void draw(int flag)**

This method is called only by the WidgetManager class.

virtual double intersect(Cpt2 &, wcomp **, pt3 *)

This method is called only by the WidgetManager class.

3.4 SimpleWidget

CLASS: **SimpleWidget**

Purpose: a base class for all widgets.

DERIVES FROM: Widget3D

METHODS: **virtual int find2d(Cpt2 &, double *)**

Called only by the WidgetManager class.

virtual void grab2d(Cpt2 &, int , int)

Called only by the WidgetManager class.

virtual void highlight_picked_component(int b)

b is a boolean value. If non-zero, this widget's widget components will be highlighted when the mouse cursor is on top of them. If zero, no highlighting will occur.

virtual void notify(wcomp *w, Cmat3 &t)

notify is called when one of this widget's widget components has been transformed. It allows the programmer to update the transformations of the other widget components. For an example of how to use this method, see the RakeWidget example below.

3.5 Behavior

CLASS: **Behavior**

Holds the type of contraint placed on the motion of a component of a widget. All constraints are enforced relative to either the object's coordinate system (for the first 5), or to the camera's orientation (for the remaining 3).

TYPES: NO_TRANSFORM no motion allowed

TRANS_LINE   motion along a line

TRANS_PLANE   motion within a plane

AXIS_ROTATE   rotation about an axis

VSPHERE_ROTATE   rotation about a virtual sphere

TRANS_FILM_XY   motion parallel to the viewing direction

TRANS_FILM_XZ   motion in and out and side to side, relative to the viewer

TRANS_FILM_Z   motion in and out, relative to the viewer

METHODS:

Behavior(behavior b, wcomp *r=0)

Constructs a behavior for widget component **r**, which needs no additional information. The parameter **b** should be one of: NO_TRANSFORM, VSPHERE_ROTATE, TRANS_FILM_XY, TRANS_FILM_XZ, and TRANS_FILM_Z.

Behavior(behavior b, Cvec3 &v, wcomp *r=0)

Constructs a behavior for widget component **r**, which needs to know only a vector. The parameter **b** should be one of: TRANS_LINE and TRANS_PLANE.

Behavior(behavior b, Cpt3 &p, Cvec3 &v, wcomp *r=0)

Constructs a behavior for widget component **r**. The parameter **b** should be: AXIS_ROTATE.

3.6 Widget Component

CLASS:

wcomp

"wcomp" is short for Widget Component whose primary function is to represent a simple geometric primitive. Multiple wcomp's are grouped together (managed by a SimpleWidget object) to create interesting widgets.

DATA:

Behavior _mouse_mapping[4][3]

An array of behaviors for each button and keyboard modifier combination. The array is accessed through the `mouse_mapping(int, int)` method (see below). E.g.,

```
wcomp *w;
Behavior b;// (should be initialized to some behavior)
w->mouse_mapping(2, wcomp::MOD_SHIFT) = b;
```

will cause behavior **b** to be activated when the second mouse button is pressed *and* the shift button is pressed.

METHODS:

wcomp(Widget3D *d, int t)

Constructor for wcomp. **d** is a pointer to a SimpleWidget (which derives from Widget3D). **t** specifies the type of geometric object this widget component is and is one of the following values: `CUBE, CONE, CYL, SPHERE.`

void clear_mouse_mappings()

Sets all mouse mappings to NO_TRANSFORM (i.e., no action will occur if the user picks this widget component).

CBehavior &mouse_mapping(int but, int mod=MOD_NONE) const

Accessor to the list of mouse mappings held by a widget component (a "constant" method). Returns a behavior for a given mouse button and keyboard modifier. E.g., `mouse_mapping(2, MOD_NONE)` will return the Behavior object used when the 2nd mouse button is pressed with no modifier's depressed.

Behavior &mouse_mapping(int but, int mod=MOD_NONE)

Accessor to the list of mouse mappings held by a widget component (*not* a constant method). Used for assignment of new Behaviors to (button, modifier) conditions. See example above where the `_mouse_mapping` data member is described.

Cmat3 &xform() const

Accessor to this object's current transformation matrix.

virtual void set_xform(Cmat3 &x, int b = -1, int m = -1)

Set's this objects transformation matrix to **x**.

(The **b** and **m** parameters are used internally to propagate mouse button and modifier information. A call to `set_xform` should only pass the `Cmat3` parameter.)

virtual void mult_by (Cmat3 &x, int b = -1, int m = -1)

Multiplys this object's transformation matrix by **x**.

(The **b** and **m** parameters are used internally to propagate mouse button and modifier information. A call to `set_xform` should only pass the `Cmat3` parameter.)

void _change_xform(Cmat3 &x)

Set's this objects transformation matrix to **x**.

Although this method has similar functionality to `set_xform(..)`, the `set_xform(..)` method is used in stead of `_change_xform(..)`. There are situations where this method is used instead of set_xform(..), however, to force the transform to change. See the rakewidget example (Section 4) for further information and a situation where `_change_xform(..)` is used.

This method will not be needed in future releases.

void set_color(Cpt3 &c)

Set the color of this object to **c**.

void unset_color()

"Unset" the color for this object. I.e., it will be drawn with whatever the last GL color was set to.

void set_transp(double t)

Set the transparency of this object to **t**. The range of values for **t** is [0, 1]. A value of 0 is completely transparent and a value of 1 is no transparency.

void unset_transp()

"Unset" the transparency level for this object. I.e., it will be drawn with whatever the last GL transparency level was set to.

void highlight(int b)

b is a boolean value. If true, a bounding box is drawn around this widget component. If false, no bounding box is drawn.

void set_res (double r)

(not implemented)

void set_draw_style(int ds)

(not implemented)

void draw()

draws this object (handled by `WidgetManager` class)

RAYhit &intersect(RAYhit &r)

intersects this object (handled by `WidgetManager` class)

CGEOM *geom() const ;

Only used internally.

3.7 Camera Interaction

CLASS: **CameraInteractor**

Purpose: To provide camera interaction functionality. Our code allows camera manipulation to be controlled using only one mouse button (saving screen real-estate and mouse buttons for other uses). Three types of camera motion are supported: film-plan translation, camera zoom, and trackball rotation.

Pre-conditions:

1) GL Modelview and GL Projection matrices hold only the camera transformation and projection matrix that were used for drawing the currently displayed framebuffer.

2) Z-buffering is enabled.

METHODS: **static void mouse_down(double x, double y)**

Should be called when the mouse button is first pressed and (x, y) should specify the location of the mouse inside the graphics window. The **x** and **y** parameter units are normalized-display coordinates (i.e., they range between -1 and 1 and coordinate pairs map to the corners of the display window).
This method is used by the library to initialize camera interaction.

static void mouse_move(double x, double y, double *m)

Should be called each time the mouse is moved after the **mouse_down(x, y)** is called (and before the camera mouse button is released). A row-major matrix describing how the GL_MODELVIEW_MATRIX should be modified is calculated and returned through the third parameter **m**. (see sample code below for how to use the matrix **m**).

Again, the **x** and **y** parameter units are normalized-display coordinates.

static void mouse_up()

Must be called when the mouse is released. It is used by the library to complete camera interaction.

The CameraInteractor works by requiring that the programmer feed in mouse information whenever the mouse button assigned to camera interaction is pressed, held and dragged, or released. Each time the mouse is dragged, a new matrix is determined by which the `GL_MODELVIEW_MATRIX` should be multiplied.

The following pseudocode demonstrates the use of the `CameraInteractor` class:

```
// assume the variables 'x' and 'y' are in
// normalized-display coordinates

if (event == ButtonPress) {
      CameraInteractor::mouse_down(x, y);
} else if (event == MouseDrag) {
      double cam_delta[16];
      CameraInteractor::mouse_move(x, y, cam_delta);
      glMatrixMode(GL_MODELVIEW);
      glMultMatrixd((GLdouble *)cam_delta);
} else if (event == ButtonRelease) {
      CameraInteractor::mouse_up();
}
```

3.8 Math Library

3.8.1 Classes

CLASS: **pt3**

Represents a 3D point.

METHODS: **pt3()**

constucts a point at the origin.

pt3(double x, double y, double z)

constructs a point at the given x, y, z coordinates.

pt3(Cpt3 &p)

copy constructor.

pt3 &operator=(Cpt3 &p)

copy operator.

double operator[](int i)

accessor for elements of the point.

double &operator[](int i)

accessor for elements of the point.

int operator==(Cpt3 &p1,Cpt3 &p2)

test to see if two points are equal.

int operator!=(Cpt3 &p1,Cpt3 &p2)

test to see if two points are not equal.

pt3 operator+(Cpt3 &p, Cvec3 &v)

returns a point containing the sum of **p** and **v**.

pt3 operator+(Cvec3 &v, Cpt3 &p)

returns a point containing the sum of **p** and **v**.

pt3 operator+=(pt3 &a, Cvec3 &b)

adds **b** to **a** and returns **a**.

pt3 operator-(Cpt3 &p, Cvec3 &v)

returns a point containing the difference between **p** and **v**.

vec3 operator-(Cpt3 &a, Cpt3 &b)

returns a vector from **b** to **a**.

pt3 operator*(Cpt3 &p, double s)

returns the value of the point **p** scaled by **s**.

pt3 operator*(double s, Cpt3 &p)

returns the value of the point **p** scaled by **s**.

pt3 operator/(Cpt3 &p, double s)

returns the value of the point **p** scaled by 1/**s**.

pt3 operator%(Cpt3 &a, Cpt3 &b)

returns a point containing the sum of **a** and **b**.

ostream &operator<<(ostream &s, Cpt3 &d)

prints out the x, y, z values of the point.

CLASS: **vec3**

Represents a 3D vector.

METHODS: **vec3()**

Constructs the zero vector

vec3(double x, double y, double z)

Creates a vector with components **x**, **y**, and **z**.

vec3(Cvec3 &p)

Copy constructor.

vec3 &operator=(Cvec3 &v)

Copy operator.

double operator[](int i) const

"Constant" accessor for x, y, and z components of the vector.

double &operator[](int i)

Accessor for x, y, and z components of the vector.

double length() const

Returns the length of this vector.

int isNull() const

Returns non-zero value if the length of this vector is greater than 10^{-6} (i.e., numerically close to the zero vector).

vec3 normalize () const

Returns a copy of this vector that has been normalized.

NOTE: this vector's components are not changed.

vec3 perpend()

Returns a vector perpendicular to this vector.

double *toDoublev()

Returns a pointer to the 3-dimensional array of doubles that holds the components of this vector.

CLASS: **mat3**

Represents a 4x4 column major matrix.

METHODS: **mat3()**

Constructs a 4x4 zero-matrix.

mat3(Cmat3 &m)

Copy constructor

mat3(double m00, double m01, ..., double m33)

Constructs a 4x4 matrix from 16 doubles.

mat3(Cpt3 &p, Cvec3 &v1, Cvec3 &v2)

Constructs a 4x4 column-major transformation matrix with the first column equal to **v1**, the second column equal to **v2**, and the fourth column equal to **p**. The third column is set equal to the cross product of **v1** and **v2.**

mat3(Cpt3 &p, Cvec3 &v1, Cvec3 &v2, Cvec3 &v3)

Constructs a 4x4 column-major transformation matrix with the first column equal to **v1**, the second column equal to **v2**, the third column equal to **v3**, and the fourth column equal to **p**.

mat3(Cpt3 &p)

Constructs a 4x4 transformation with the fourth column equal to **p**.

mat3 &operator=(Cmat3 &m)

Copy operator.

double operator()(int i, int j) const

"Constant" matrix data accessor method. **i** and **j** are integers in the range [0,3].

double &operator()(int i, int j)

Matrix data accessor method. **i** and **j** are integers in the range [0,3].

mat3 transpose() const

returns a new matrix that is the transpose of this matrix. This matrix is unaltered.

mat3 invert(int rigid) const

returns the inverse of this matrix. By default, **rigid** is false and a general matrix inversion algorithm is used. If the matrix is known to be rigid, setting **rigid** to true will use a faster matrix inversion algorithm. This matrix is unaltered.

pt3 position() const

returns a **pt3** equal to the translational component (i.e., fourth column) of this matrix.

void setRMMatrix(const double *d)

sets this matrix equal to the transpose of the 4x4 row-major matrix **d**. **d** must be an array of 16 doubles.

void getRMMatrix(double *d) const

sets the row-major matrix pointed to by **d** equal to this column-major matrix.

void getCoordSystem(pt3 &pos, vec3 &x, vec3 &y, vec3 &z) const

Sets **pos**, **x**, **y**, and **z** to the position, x-axis, y-axis, and z-axis components, respectively, of this matrix. **pos** is set equal to the fourth column of this matrix. **x** is set equal to the first column of this matrix.**y** is set equal to the second column of this matrix. **z** is set equal to the third column of this matrix.

mat3 normalize_basis() const

Normalizes the upper 3x3 sub-matrix of this matrix.

static mat3 alignAndScale(Cpt3& origin,Cvec3& xx, Cvec3& yy, Cvec3& zz)

returns a matrix with first through fourth column vectors equal, respectively, to **origin**, **xx**, **yy**, and **zz**. This matrix is unaltered.

static mat3 align(Cpt3 &p1, Cvec3 &v11, Cvec3 &v21, Cpt3 &p2, Cvec3 &v12, Cvec3 &v22)

returns a matrix that maps **p1** to **p2**, **v11** to **v12**, and **v21** to **v22**. This matrix is unaltered.

static mat3 align(Cpt3 &p1, Cvec3 &v1, Cpt3 &p2, Cvec3 &v2)

returns a matrix that maps **p1** to **p2** and **v1** to **v2**. This matrix is unaltered.

OPERATORS:

int operator==(Cmat3 &m, Cmat3 &n)

returns a non-zero value if m is equal to n; otherwise, zero is returned.

int operator!=(Cmat3 &m, Cmat3 &n)

returns a non-zero value if m is not equal to n; otherwise, zero is returned.

pt3 operator*(Cmat3 &m, Cpt3 &p)

returns the result of multiplying the matrix **m** on the right by **p**.

vec3 operator*(Cmat3 &m, Cvec3 &v)

returns the result of multiplying the matrix **m** by the vector **v.**

mat3 operator*(Cmat3 &a, Cmat3 &b)

returns the result of multiplying the matrix **a** by **b**.

mat3 operator*(double s, Cmat3 &m)

returns the result of multiplying the matrix **m** by **s**.

mat3 operator*(Cmat3 &m, double s)

returns the result of multiplying the matrix **m** by **s**.

mat3 operator+(Cmat3 &a, Cmat3 &b)

returns the result of adding the matrix **b** to **a**.

mat3 operator-(Cmat3 &a, Cmat3 &b)

returns the result of subtracting the matrix **b** to **a**.

mat3 operator-(Cmat3 &m)

returns the result of negating the matrix **m**.

FUNCTIONS:

mat3 scale_mat(Cvec3 &s)

returns a matrix that scales points and vectors by the components of **s**.

mat3 scale_mat(Cpt3 &p, Cvec3 &s)

returns a matrix that scales points and vectors by the components of **s**. The center of scale is at the point **p**.

mat3 trans_mat(Cvec3 &v)

returns a matrix that translates points by the vector **v**.

mat3 rot_mat (Cvec3 &axis, double rad)

returns a matrix that rotates points and vectors about the axis **axis** by **rad** radians.

mat3 shear_mat(Cvec3 &nv,Cvec3 &sv)

returns a matrix that shears points and vectors along the vector **nv** by the components of **sv**.

mat3 stretch_mat(Cpt3 &p, Cvec3 &v)

returns a matrix that "stretches" points and vectors in the direction **v** with a center of "stretch" equal to **p**.

mat3 rotation (Cpt3 &p, Cvec3 &a, double angle)

returns a matrix that rotates points and vectors around the axis **a** by **angle** radians. The center of rotation is **p**.

mat3 trans_comp(Cmat3 &m)

returns a matrix that contains only the translational component of the composite transformation matrix **m**.

mat3 scale_comp(Cmat3 &m)

returns a matrix that contains only the scaling component of the composite transformation matrix **m**.

mat3 rot_comp(Cmat3 &m)

returns a matrix that contains only the rotational component of the composite transformation matrix **m.**

ostream &operator<<(ostream &s, Cmat3 &m)

"prints" the matrix **m** to the stream **s**.

CLASS:

pt2

Represents a 2D point.

METHODS:

Identical to those in the **pt3** class above, except for a two-dimensional point.

CLASS: **vec2**

Represents a 2D vector.

METHODS: Identical to those in the **vec3** class above, except for a two-dimensional vector.

3.8.2 Functions

pt3 plane_intersect(Cpt3 &p, Cvec3 &n, Cpt3 &a)

projects the point **a** onto the plane defined by the point and vector **p** and **n**.

return value: the projected point.

pt3 line_intersect(Cpt3 &p, Cvec3 &v, Cpt3 &a)

projects the point **a** onto the 3D line defined by the point and vector **p** and **v**.

return value: the projected point.

pt2 line_intersect(Cpt2 &p,Cvec2 &v,Cpt2 &a)

projects the point **a** onto the 2D line defined by the point and vector **p** and **v**.

return value: the projected point.

double dist_pt_to_plane(Cpt3 &p, Cvec3 &n, Cpt3 &a)

returns the distance between a point **a** and the plane defined by the point and vector **p** and **v**.

double dist_pt_to_line(Cpt3 &p, Cvec3 &v, Cpt3 &a)

returns the distance between a point **a** and the 3D line defined by the point and vector **p** and **v**.

double dist_pt_to_line(Cpt2 &p, Cvec2 &v, Cpt2 &a)

returns the distance between the point **a** and the2D line defined by the point and vector **p** and **v**.

double axis_ang(Cpt3 &p1, Cpt3 &p2, Cpt3 &axispt, Cvec3 &axis)

returns the angle (in radians) between the vector from **axispt** to **p1** and the vector from **axispt** to **v2**. Vector **axis** determines whether the return value is positive or negative.

pt3 line_intersect (Cpt3 &line1p, Cvec3 &line1vec, Cpt3 &line2p, Cvec3 &line2vec)

returns the point on one 3D line which lies closest to a second 3D line. The first line is defined by **line1p** and

line1vec, and the second by **line2p** and **line2vec.**

pt3 plane_intersect(Cpt3 &linep, Cvec3 &linev, Cpt3 &planep, Cvec3 &planev)

returns the point of intersection between a line and a plane. The line is defined by the point and vector **linep** and **linev**, and the plane is defined by the point and vector **planep** and **planev**.

NOTE: Does not currently check if there *is* an intersection—there may not be one. If there is no intersection, the return value is invalid.

pt2 intersect (Cpt2 &a, Cvec2 &b, Cpt2 &c, Cvec2 &d, int &i)

returns the intersection point between two 2D lines. The first line is defined by the point and vector **a** and **b**, and the second by **c** and **d**. It returns the value 0 in **i** if there is no intersection.

double angle_between(Cvec3 &v1, Cvec3 v2)

returns the angle between the vectors **v1** and **v2** in radians.

4 The "RakeWidget"— An Example

The following example code can be found in the 3D Widget Library distribution in the file :

`widgetlib/rakewidget.design.example/rakewidget.C`

This code is an eight step example for building a sample 3D widget. For each step, new or altered code is displayed in bold-faced, italicized text.

```
Example 3D Widget: a Rake Widget

DESCRIPTION:

A rake is a tool used in scientific visualization. It typically consists of a
long, thin bar from which streamlines are emitted. The bar can be moved in 3D
thereby allowing a scientist to explore a 3D dataset.

The Rake Widget example presented here has three components: a frame (the long
thin bar), a scaler component, and a slider component. The frame is a thin,
white cylinder that can be translated and rotated within a 3D volume. The
scaler component is a yellow ball at the end of the frame that changes the
length of the frame when it is translated. The slider is a red disc that
slides between the middle and one end of the frame component and can be used,
for example, to set the distance between streamlines along the frame.

The remainder of this file demonstrates how a Rake Widget can be constructed
```

using the Widget Library. At each stage a piece of geometry or functionality is added until the described rake widget is realized.

```
//////////////////////////////////////////////////////////////////////

Cylinder coordinate system

       ^
       | Z

   (flat top)

    |------|
    |      |
    |      |
    |      |  -> x & y
    |      |
    |      |
    |------|

//////////////////////////////////////////////////////////////////////

Use these include files:

#include <widgetlib/mlib.H>
#include <widgetlib/simplewidget.H>

//////////////////////////////////////////////////////////////////////

// 1. Create a subclass of SimpleWidget and create three widget components
// (2 cylinders and 1 sphere). Their colors are white, red, and yellow,
// respectively. Finally, they are added to the '_widget_comps' list
// which is used by the draw() and intersect() methods.
class ExampleRakeWidget : public SimpleWidget
{
 protected:
   wcomp *_frame;
   wcomp *_slider;
   wcomp *_scaler;

 public:
  ExampleRakeWidget( double length )
  {
    _frame  = new wcomp(this, wcomp::CYL);
    _slider = new wcomp(this, wcomp::CYL);
    _scaler = new wcomp(this, wcomp::SPHERE);

    _frame ->set_color(pt3(1,1,1));
    _slider->set_color(pt3(1,0,0));
    _scaler->set_color(pt3(1,1,0));

    _widget_comps += _frame;
```

```
      _widget_comps += _slider;
      _widget_comps += _scaler;
    }
  };

  ///////////////////////////////////////////////////////////////////////

  // 2. The slider and scaler mouse mappings are cleared. Both widget's
  // mouse mappings are set to translate along the z-axis in the '_frame's
  // reference frame when mouse button #2 is pressed.
  class ExampleRakeWidget : public SimpleWidget
  {
   protected:
     wcomp *_frame;
     wcomp *_slider;
     wcomp *_scaler;

   public:
    ExampleRakeWidget( double l )
    {
      _frame  = new wcomp(this, wcomp::CYL);
      _slider = new wcomp(this, wcomp::CYL);
      _scaler = new wcomp(this, wcomp::SPHERE);

      _frame ->set_color(pt3(1,1,1));
      _slider->set_color(pt3(1,0,0));
      _scaler->set_color(pt3(1,1,0));

      _slider->clear_mouse_mappings();
      _slider->mouse_mapping(2) = Behavior(Behavior::TRANS_LINE,
                             Zaxis,
                             _frame);

      _scaler->clear_mouse_mappings();
      _scaler->mouse_mapping(2) = Behavior(Behavior::TRANS_LINE,
                             Zaxis,
                             _frame);

      _widget_comps += _frame;
      _widget_comps += _slider;
      _widget_comps += _scaler;
    }

  };

  ///////////////////////////////////////////////////////////////////////

  // 3. The initial transforms for the _frame, _slider, and _scaler widget
  // are set. In addition, the 'notify(..)' method has been added (with
  // an empty body for now), and will next be used to respond to movements
  // of widget components.
  {
   protected:
     wcomp *_frame;
    wcomp *_slider;
    wcomp *_scaler;

  public:
   ExampleRakeWidget( double l )
   {
     _frame  = new wcomp(this, wcomp::CYL);
     _slider = new wcomp(this, wcomp::CYL);
     _scaler = new wcomp(this, wcomp::SPHERE);

     _frame ->set_color(pt3(1,1,1));
     _slider->set_color(pt3(1,0,0));
     _scaler->set_color(pt3(1,1,0));

     _frame ->set_xform(scale_mat(vec3(.2,.2,5)));
     _slider->set_xform(scale_mat(vec3(1,1,.2)));
     _scaler->set_xform(trans_mat(vec3(0.0, 0.0,   1)) *
                     scale_mat(vec3(1.0, 1.0, 1.0)));

     _slider->clear_mouse_mappings();
     _slider->mouse_mapping(2) = Behavior(Behavior::TRANS_LINE,
                           Zaxis,
                           _frame);

     _scaler->clear_mouse_mappings();
     _scaler->mouse_mapping(2) = Behavior(Behavior::TRANS_LINE,
                           Zaxis,
                           _frame);

     _widget_comps += _frame;
     _widget_comps += _slider;
     _widget_comps += _scaler;
   }

   void notify(wcomp *obj,const mat3 &m)
   {
   }
};

//////////////////////////////////////////////////////////////////////

// 4. The body of the 'notify(..)' method now checks if the _frame
// widget component has been moved. If it has, then both the _slider and
// _scaler are moved by the same transform.
class ExampleRakeWidget : public SimpleWidget
{
 protected:
   wcomp *_frame;
   wcomp *_slider;
   wcomp *_scaler;

 public:
  ExampleRakeWidget( double l )
  {
    _frame  = new wcomp(this, wcomp::CYL);
    _slider = new wcomp(this, wcomp::CYL);
    _scaler = new wcomp(this, wcomp::SPHERE);

    _frame ->set_color(pt3(1,1,1));
    _slider->set_color(pt3(1,0,0));
    _scaler->set_color(pt3(1,1,0));

    _frame ->set_xform(scale_mat(vec3(0.2, 0.2,   1)));
    _slider->set_xform(scale_mat(vec3(1.0, 1.0, 0.2)));
    _scaler->set_xform(trans_mat(vec3(0.0, 0.0,   1)) *
                   scale_mat(vec3(1.0, 1.0, 1.0)));

    _slider->clear_mouse_mappings();
    _slider->mouse_mapping(2) = Behavior(Behavior::TRANS_LINE,
                         Zaxis,
                         _frame);

    _scaler->clear_mouse_mappings();
    _scaler->mouse_mapping(2) = Behavior(Behavior::TRANS_LINE,
                         Zaxis,
                         _frame);

    _widget_comps += _frame;
    _widget_comps += _slider;
    _widget_comps += _scaler;
  }

  void notify(wcomp *obj, const mat3 &m)
  {
    if (obj == _frame) {
      _slider->mult_by(m);
      _scaler->mult_by(m);
    }
  }
};

////////////////////////////////////////////////////////////////////

// 5. More code has been added to the 'notify' method. If the _slider
// has been moved, then it's position is constrained such that it
// remains between the end and middle of the _frame widget component.
//
// This constraint is accomplished with the 'constrain_to_seg' function
// by specifying that the _slider widget is to be constrained to the
// positions for u between 0 and 1 given
// 'position = _frame->xform() * (Origin + u * Zaxis)'.
class ExampleRakeWidget : public SimpleWidget
{
 protected:
   wcomp *_frame;
   wcomp *_slider;
   wcomp *_scaler;
  public:
   ExampleRakeWidget( double l )
   {
     _frame  = new wcomp(this, wcomp::CYL);
     _slider = new wcomp(this, wcomp::CYL);
     _scaler = new wcomp(this, wcomp::SPHERE);

     _frame ->set_color(pt3(1,1,1));
     _slider->set_color(pt3(1,0,0));
     _scaler->set_color(pt3(1,1,0));

     _frame ->set_xform(scale_mat(vec3(0.2, 0.2,   1)));
     _slider->set_xform(scale_mat(vec3(1.0, 1.0, 0.2)));
     _scaler->set_xform(trans_mat(vec3(0.0, 0.0,   1)) *
                    scale_mat(vec3(1.0, 1.0, 1.0)));

     _slider->clear_mouse_mappings();
     _slider->mouse_mapping(2) = Behavior(Behavior::TRANS_LINE,
                          Zaxis,
                          _frame);

     _scaler->clear_mouse_mappings();
     _scaler->mouse_mapping(2) = Behavior(Behavior::TRANS_LINE,
                          Zaxis,
                          _frame);

     _widget_comps += _frame;
     _widget_comps += _slider;
     _widget_comps += _scaler;
   }

  void notify(wcomp *obj,const mat3 &m)
  {
    if (obj == _frame) {
      _slider->mult_by(m);
      _scaler->mult_by(m);
    }
    else if (obj == _slider) {
      constrain_to_seg(_slider,
                   Origin,
                   Zaxis,
                   0,1,_frame->xform());
    }
  }
};

//////////////////////////////////////////////////////////////////////

// 6. Now if _slider has been moved, first we constrain the slider to
// lie between 0.1 and 1000.0 along the Zaxis of _frame's coordinate
// system. In addition, the _frame's coordinate system is scaled
// such that it's new length passes through the center of the _scaler
// again.
class ExampleRakeWidget : public SimpleWidget
```

```
{
 protected:
   wcomp *_frame;
   wcomp *_slider;
   wcomp *_scaler;

 public:
  ExampleRakeWidget( double l )
  {
    _frame  = new wcomp(this, wcomp::CYL);
    _slider = new wcomp(this, wcomp::CYL);
    _scaler = new wcomp(this, wcomp::SPHERE);

    _frame ->set_color(pt3(1,1,1));
    _slider->set_color(pt3(1,0,0));
    _scaler->set_color(pt3(1,1,0));

    _frame ->set_xform(scale_mat(vec3(0.2, 0.2,   1)));
    _slider->set_xform(scale_mat(vec3(1.0, 1.0, 0.2)));
    _scaler->set_xform(trans_mat(vec3(0.0, 0.0,   1)) *
                       scale_mat(vec3(1.0, 1.0, 1.0)));

    _slider->clear_mouse_mappings();
    _slider->mouse_mapping(2) = Behavior(Behavior::TRANS_LINE,
                             Zaxis,
                             _frame);

    _scaler->clear_mouse_mappings();
    _scaler->mouse_mapping(2) = Behavior(Behavior::TRANS_LINE,
                             Zaxis,
                             _frame);

    _widget_comps += _frame;
    _widget_comps += _slider;
    _widget_comps += _scaler;
  }

  void notify(wcomp *obj, const mat3 &m)
  {
    if (obj == _frame) {
      _slider->mult_by(m);
      _scaler->mult_by(m);
    }
    else if (obj == _slider) {
      constrain_to_seg(_slider,
                       Origin,
                       Zaxis,
                       0,1,_frame->xform());
    }
    else if (obj == _scaler) {
      constrain_to_seg(_scaler,
                       Origin,
                       Zaxis,
                       0.1,
                   1000.0,
                   _frame->xform());

      _frame->set_xform(trans_comp(_frame->xform()) *
                  rot_comp  (_frame->xform()) *
                  scale_mat (vec3(.2, .2, (_scaler->xform().position()
                                  - _frame->xform().position()).length())));
    }
  }
};

//////////////////////////////////////////////////////////////////////

// 7. Lastly, because the scaling of the _frame propagates to the _slider,
// it is necessary to reset the scaler component of the _slider. A side
// affect of changing the _frame's scale is that the _slider's scale
// is also changed. To remedy this, we can simply reset the _slider's
// scale to it's original value. However, because the _slider's
// 'mult_by(..)' command has already been called, it is necessary to
// call the _change_xform(..) method in order to counter the scaling
// effects. (Try it out to see why this is necessary.)
class ExampleRakeWidget : public SimpleWidget
{
 protected:
   wcomp *_frame;
   wcomp *_slider;
   wcomp *_scaler;

 public:
  ExampleRakeWidget( double l )
  {
    _frame  = new wcomp(this, wcomp::CYL);
    _slider = new wcomp(this, wcomp::CYL);
    _scaler = new wcomp(this, wcomp::SPHERE);

    _frame ->set_color(pt3(1,1,1));
    _slider->set_color(pt3(1,0,0));
    _scaler->set_color(pt3(1,1,0));

    _frame ->set_xform(scale_mat(vec3(0.2, 0.2,   1)));
    _slider->set_xform(scale_mat(vec3(1.0, 1.0, 0.2)));
    _scaler->set_xform(trans_mat(vec3(0.0, 0.0,   1)) *
                  scale_mat(vec3(1.0, 1.0, 1.0)));

    _slider->clear_mouse_mappings();
    _slider->mouse_mapping(2) = Behavior(Behavior::TRANS_LINE,
                        Zaxis,
                        _frame);

    _scaler->clear_mouse_mappings();
    _scaler->mouse_mapping(2) = Behavior(Behavior::TRANS_LINE,
                        Zaxis,
                        _frame);
```

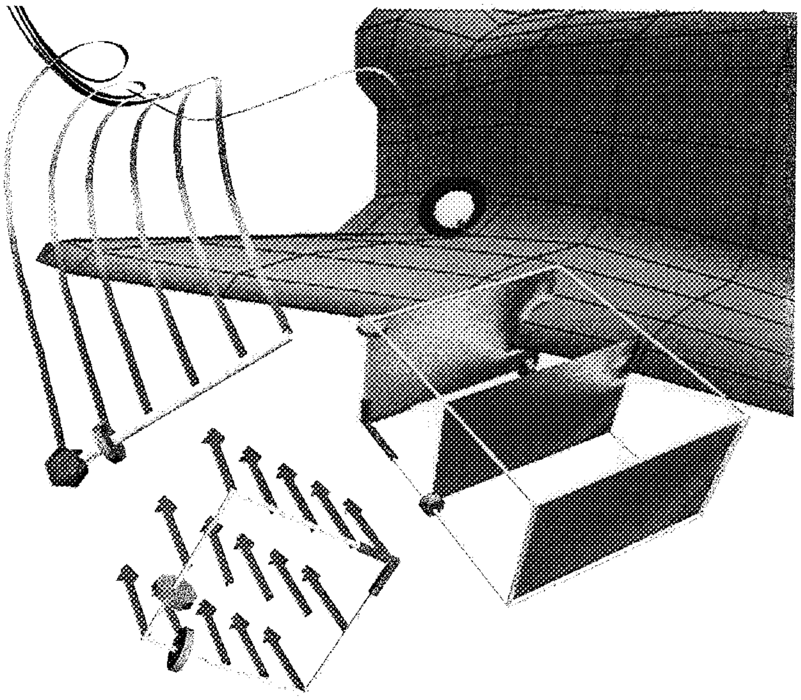
```
     _widget_comps += _frame;
     _widget_comps += _slider;
     _widget_comps += _scaler;
   }

   void notify(wcomp *obj,const mat3 &m)
   {
     if (obj == _frame) {
       _slider->mult_by(m);
       _scaler->mult_by(m);
     }
     else if (obj == _slider) {
       constrain_to_seg(_slider,
                    Origin,
                    Zaxis,
                    0,1,_frame->xform());
     }
     else if (obj == _scaler) {
       constrain_to_seg(_scaler,
                    Origin,
                    Zaxis,
                    1.0,
                    100.0,
                    _frame->xform().normalize_basis());

      _frame ->set_xform(trans_comp(_frame->xform()) *
                    rot_comp  (_frame->xform()) *
                    scale_mat (vec3(.2, .2, (_scaler->xform().position()
                                    - _frame->xform().position()).length())));

      _slider->_change_xform(trans_comp(_slider->xform()) *
                        rot_comp  (_slider->xform()) *
                        scale_mat (vec3(1,1,.2)));
    }
  }
};

///////////////////////////////////////////////////////////////////////

//
// 8. One more thing-- adding shadows for the three widget components.
//
class BrownRakeWidget : public SimpleWidget
{
 protected:
   wcomp *_frame;
   wcomp *_slider;
   wcomp *_scaler;

 public:
  BrownRakeWidget( double l )
  {
    _frame  = new wcomp(this, wcomp::CYL);
    _slider = new wcomp(this, wcomp::CYL);
  _scaler = new wcomp(this, wcomp::SPHERE);

  _frame ->set_color(pt3(1,1,1));
  _slider->set_color(pt3(1,0,0));
  _scaler->set_color(pt3(1,1,0));

  _frame ->set_xform(scale_mat(vec3(0.2, 0.2,   1)));
  _slider->set_xform(scale_mat(vec3(1.0, 1.0, 0.2)));
  _scaler->set_xform(trans_mat(vec3(0.0, 0.0,   1)) *
                  scale_mat(vec3(1.0, 1.0, 1.0)));

  _slider->clear_mouse_mappings();
  _slider->mouse_mapping(2) = Behavior(Behavior::TRANS_LINE,
                        Zaxis,
                        _frame);

  _scaler->clear_mouse_mappings();
  _scaler->mouse_mapping(2) = Behavior(Behavior::TRANS_LINE,
                        Zaxis,
                        _frame);

  _widget_comps += _frame;
  _widget_comps += _slider;
  _widget_comps += _scaler;

  pt3  p(0,0.01,0);
  vec3 v(Yaxis);
  _widget_comps += create_shadow_widget(_frame,  p, v);
  _widget_comps += create_shadow_widget(_slider, p, v);
  _widget_comps += create_shadow_widget(_scaler, p, v);
}

void notify(wcomp *obj,const mat3 &m)
{
  if (obj == _frame) {
    _slider->mult_by(m);
    _scaler->mult_by(m);
  }
  else if (obj == _slider) {
    constrain_to_seg(_slider,
                     Origin,
                     Zaxis,
                     0,1,_frame->xform());
  }
  else if (obj == _scaler) {
    constrain_to_seg(_scaler,
                     Origin,
                     Zaxis,
                     1.0,
                     100.0,
                     _frame->xform().normalize_basis());

    _frame ->set_xform(trans_comp(_frame->xform()) *
                     rot_comp  (_frame->xform()) *
```

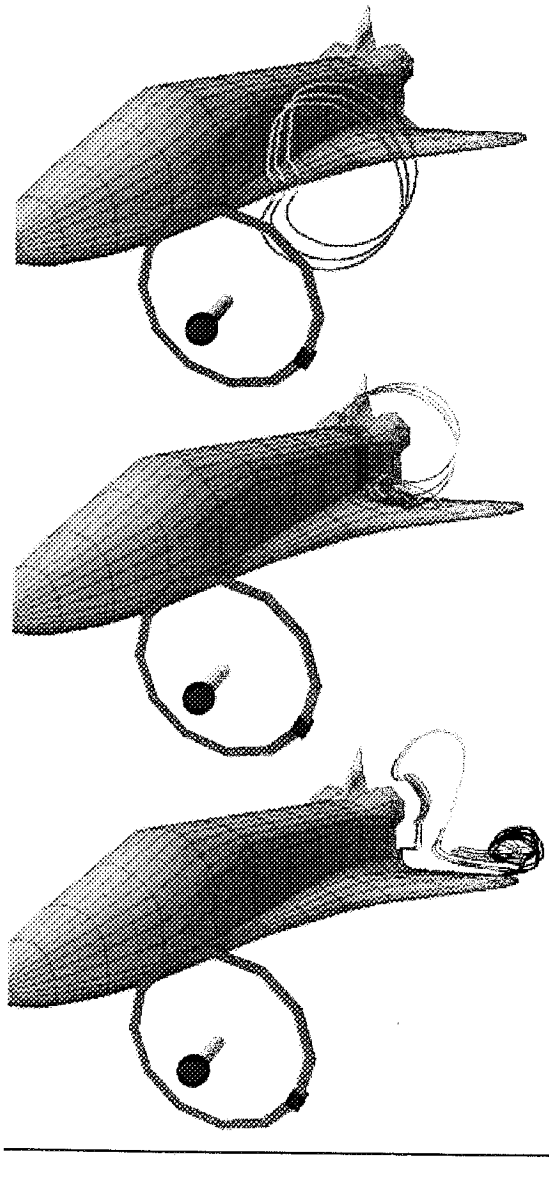
```
                    scale_mat (vec3(.2, .2, (_scaler->xform().position()
                                     - _frame->xform().position()).length())));

      _slider->_change_xform(trans_comp(_slider->xform()) *
                        rot_comp  (_slider->xform()) *
                        scale_mat (vec3(1,1,.2)));
    }
  }
};

////////////////////////////////////////////////////////////////////////
```

Brown University Computer Graphics Research Group
A site of the NSF Graphics and Visualization Center

3D Widgets Library

Funded by NASA grants for the past four years, the Brown site of the NSF Graphics and Visualization Center has developed 3D user interfaces for desktop and immersive scientific visualization applications. Our most recent user-interface project with NASA is the development of a stand-alone library for creating and interacting with 3D widgets [2][4]. The core of the library is a set of building blocks for constructing custom 3D widgets and functions that handle direct manipulation of the widgets. The library will be integrated with NASA's Virtual Windtunnel system [1] but is also available for public use.





In addition to the basic widget building blocks, the library supports interactive shadows [3] and gestural navigation controls (i.e., translation forward and backward, virtual trackball rotation, and film-plane translation) using a single mouse button. These camera controls are derived from the camera controls used in the Sketch system [5].

The library is written in C++ and is built on top of Open GL.

For more information on this and other Graphics Group projects please see http://www.cs.brown.edu/research/graphics

[1] Bryson, S. and Levit, C., The Virtual Windtunnel: An Environment for the Exploration of Three-Dimensional Unsteady Flows, NASA Ames Research Center, RNR Technical Report RNR-92-013, October 1991.

[2] Conner, D.B., Snibbe, S.S., Herndon, K.P., Robbins, D.C., Zeleznik, R.C. and van Dam, A., Three-dimensional Widgets. *Computer Graphics* (Proceedings of the 1992 Symposium on Interactive 3D Graphics), 25(2), *ACM SIGGRAPH,* March, 1992, pp. 183-188.

[3] Herndon, K.P., Zeleznik, R.C., Robbins, D.C., Conner, D.B., Snibbe, S.S. and van Dam, A., Interactive Shadows. *Proceedings of UIST '92, ACM SIGGRAPH,* November, 1992, pp. 1-6.

[4] Herndon, K.P. and Meyer, T., 3D Widgets for Exploratory Scientific Visualization, *Proceedings of UIST '94, ACM SIGGRAPH,* November, 1994, pp. 69-70.

[5] Zeleznik, R.C., Herndon, K.P., and Hughes, J.F., SKETCH: An Interface for Sketching 3D Scenes. *Computer Graphics* (Proceedings of SIGGRAPH '96), August, 1996, pp. 163-170.