# Modular Virtual Reality Visualization Tools

***Wes Bethel***
Lawrence Berkeley Laboratory
University of California
Berkeley, California 94720

Spring 1995

**Abstract**

We describe a collection of tools designed to be used in a Modular Visualization Environment (MVE) that facilitate the use of a Virtual Reality (VR) input device. VR input devices provide the means to easily and intuitively specify information which is three- or six-dimensional. Many scientific visualization tasks require this type of information. Our goal is to reduce the user interface complexity by employing a six-dimensional input device to specify six-dimensional data in scientific visualization. In the following sections, we describe relevant previous work and the functions supported by tools: device input, camera positioning, object transformations and developer's tools.

## 1.0 Introduction

While the debate continues over whether or not the term "Virtual Reality" is an oxymoron or pleonasm, those of us tasked with developing or using visualization software more often than not must eschew this largely philosophical debate and focus our attention on more practical matters: usable software and scientific progress.

VR implementations range from fully immersive to desktop systems [1]. The goal of each of these implementations is to provide a user interface in which a human can interact with a computer model in a way which is intuitive, "easy to use", interesting and engaging. We describe an approach to desktop VR geared towards the scientific user of MVE's that combine inexpensive yet practical VR input devices with a methodology for using the data generated by these devices in a variety of ways. For example, a tracker could be used in one context to position a viewpoint, but in another, to orient a slice plane.

Most VR input devices generate information which is six-dimensional: position and orientation. Along with that information, many have one or more buttons which can be used to generate boolean events. The methodology presented shows one way to manage this type of input, and capitalizes on the strengths of MVE's by allowing the user freedom of choice about the way in which the VR input device data is used in the visualization network. The benefits of MVE's which are of primary relevance are reusable software components and support for computing in a heterogeneous networked environment.

## 2.0 Relevant Previous Work

In Bryson's Virtual Windtunnel [2] used for interactive exploration of vector flow fields, a boom-mounted display tracks viewer position and orientation. The boom display is a stereo output device, presenting slightly different views to each of the left and right eyes, such that each image is computed from a slightly different viewpoint. When the user's head moves (and also the boom-mounted display), the view of the model changes to reflect the new orientation of the viewer. On the input side, a six dimensional tracker is attached to a glove device. The user is presented with an iconified representation of the input device in the virtual environment, and may position one of several flow visualization tools in the flow field.

Related work, by Meyer and Globus [3], build on the notion of "direct manipulation interfaces" (DMI's). A simple example of a DMI is a data probe. An iconified representation of a probe is positioned into a region of interest and data values at that location are reported back to the user. The user "directly" manipulates the probe into the desired position. The work in [3] describes two DMI's, one for an isosurface tool and another which controls the position and orientation of a slice plane.

The isosurface DMI in [3] consists of an icon which positioned into a three-dimensional data set. Given the location of the isosurface "widget", the surface of constant value which passes through that grid point is computed and drawn. Unlike the marching cubes algorithm [4] for isosurface computation which can compute disconnected surfaces by virtue of the fact that it examines all data points in a grid, a modified isosurface algorithm (see [3]) computes portions of the constant-valued surface passing through that grid cell. The slice plane widget permits a user to "grab" the "frame," or a rectangle which represents the edges of the slice plane, and then reposition, resize or reorient the slice plane.

In AVS, there are a number of visualization tools with DMI's, some shipped with the system and others available from the user community. Rather than modify each of these tools to process input device events directly, a more simple and elegant solution is to create a single tool which permits direct manipulation of any object under control of the AVS Geometry Viewer. Later in this paper, we will provide several examples of DMI implementation.

Our work enables desktop VR input device interfaces to all existing visualization tools in AVS which have a mouse-based interface as well as permit the user enhanced control over viewer position, via the VR input device, above and beyond the capabilities of the shipped version of AVS. The name which we have given our suite of tools is the VRModules Modules Library.

## 3.0 The VRModules Module Library

The VRModules provide support for integrating a VR input device into an AVS network, support for viewpoint manipulation, for enabling VR input device control over existing direct manipulation interfaces in AVS, and several miscellaneous tools. The source code for these modules is available via anonymous ftp from the International AVS Center (ftp.avs.ncsc.org). The tools are implemented as AVS modules.

## 3.1 Input

At our site, we have made extensive use of the Spaceball, available from SpaceTec, Inc. (Figure 1). The input module, called **spaceball,** is implemented as an AVS coroutine module. Thus, the module is freely-running, and will inject data into the visualization network whenever an event is generated by the input device.

Figure 1.

The Spaceball in action.

The **spaceball** module has a number of user-modifiable parameters. These include options for masking out certain components of the input device's event data. The user can, for example, choose to mask out all z-axis rotational data from the input data stream. Dials are provided for attenuating rotational and translational data generated by the input device. Boosting or reducing the magnitude of this data can be thought of conceptually as adjusting the sensitivity of the input device.

Often it is useful to have a mock-up, or simulated, input device. We include a **dummy device** module with VRModules. This subroutine module will generate an input device event when the user presses a button on its graphical user interface. This module will come in handy when prototyping new tools which make use of the VR input device data.

The Spaceball itself operates by measuring the displacement between two annular spheres. The translational and rotational values are not cumulative, that is, the data from this device consists of a stream of small real numbers between zero to one. These values indicate the displacement in six dimensions of the outer sphere from the inner one. These values are accumulated, or integrated, over time to produce position and orientation values (similar to a mouse). In contrast, other devices, such as magnetic-field trackers, generate absolute coordinates within a canonical space and no such temporal integration is necessary to produce coordinates. That we temporally integrate absolute displacement to produce accumulated position and orientation is an issue to keep in mind when studying or using our tools, or when implementing your own.

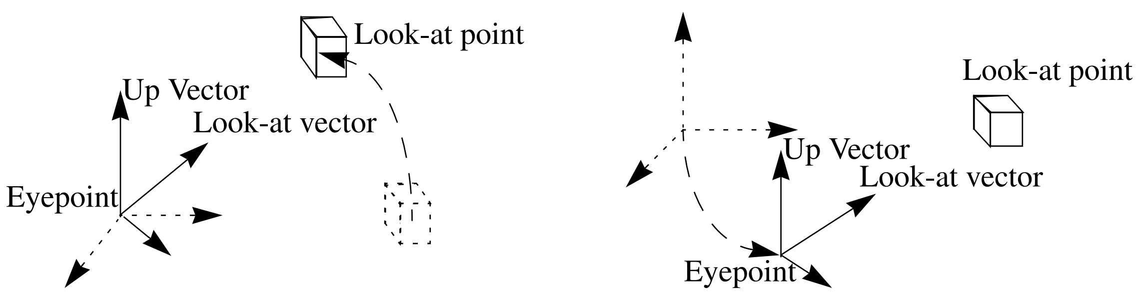
A final important point to make concerns coordinate systems. The input device generates events which are in what we refer to as "user space." Specific transformations exist between user space, object space and view space. These transformations will be described below. The term user space honors the fact that there are certain precognitive expectations which, if not met, will lead to user dissatisfaction and to comments like “this system doesn’t work.” View space and user space are orthogonal. As such, one precognitive expectation which we can trivially identify is that the user will expect, for example, a x-axis translation event generated on the input device to produce an x-axis translation in view space. This is consistent with our day-to-day experiences; we grasp something on our desk, move our arm to the left, the grasped object moves to the left also. Imagine your reaction if you moved your arm to the left, and the object moved to the right!

## 3.2 Camera Position

The items of primary importance which comprise the viewpoint in AVS are the location of the viewer (the “eyepoint”), the direction in which the viewer is looking (the “look-at” vector) and a reference vector which defines which way is up (the “up” vector). Please refer to [7] for more information concerning the specification of three dimensional viewing parameters.

There are both similarities and differences between the view transformation models in VRModules and that in AVS. Camera translations in both AVS and our model cause a change in eyepoint, without a change in either the look-at or up vectors. These vectors may be thought of as directions which are specified relative to the eyepoint. Thus, the viewer’s location changes, but the direction in which they are gazing does not.

Camera “rotations” are markedly different in our model. When a y-axis rotation event is generated by the input device, our camera will pan. In other words, the eyepoint remains fixed, but the look-at vector changes so that the view sweeps across the horizon. In the AVS model, a y-axis rotation event will “orbit” the viewer about an object (Figure 2). Informal and subjective testing, using a six-year-old subject, has shown that our model is easier to use and more intuitive.  Experience has shown that user’s often want to “fly through” data, over terrain, and so forth. This type of activity is directly facilitated by our model.




VRModules: pans the camera. AVS: rotates eyepoint about look-at point.

Figure 2.
Effects of a Y-axis Rotation Upon View Parameters

We have mapped button number one on the spaceball device to activate a “reset the camera” function, in case the user flies off into oblivion. Note that this function is not programmed into the **spaceball** module. The button push is detected by the camera transformation module which then acts upon this event by reinitializing the camera position to start-up defaults.

### 3.2.1 Implementing Camera Transformations

Manipulation of the camera position is decomposed into two components. The location of the viewer is manipulated independent of the orientation of the viewer. So, for example, we can change where the viewer is located without changing the direction of gaze. Considering only the translational events, let's assume that we are given a translation vector, $\vec{t}$. The viewer's new position, $P_{new}$, is computed as $P_{new} = P_{old} + \vec{t}$. The translational events occur in *user space,* which is orthogonal to view space. Thus, we can simply add the translational component of the device event to the old camera position to obtain the new camera position.

Rotational events are computed in matrix form. From the input device, we are given a rotation vector, $\vec{r}$. Also, let's assume that we have an existing view transformation matrix, $V_{old}$. This matrix has no translational components, since we are maintaining viewer position separately from viewer orientation. We will construct a new view transformation matrix, $V_{new}$, which is the matrix product of the new rotations, $R_{new}$ and the existing matrix $V_{old}$.

$$R_{new} = f(\vec{r})$$

$$V_{new} = V_{old} R_{new}$$

The function $f(\vec{r})$ is a process in which a three dimensional rotation matrix is constructed from the rotational components of the event produced by the VR input device. See [7] for the procedure for constructing three dimensional rotation matrices. The order in which the rotations are applied the same as the method described below in the section on object transformations.

The new values for the look-at and up vectors may be obtained directly from $V_{new}$. The middle column of $V_{new}$ is the y-axis of the rotated coordinate system [5], hence becomes the new up vector. The third column of $V_{new}$ is the z-axis of the rotated coordinate system, hence becomes the new look-at vector. The AVS Geometry Library wants an actual point rather than a vector, so we compute a three dimensional coordinate using the eyepoint, the look-at vector and a camera control parameter that we call "focal depth."

### 3.2.2 Limitations of the Camera Module

The camera transformation module works only for single camera scenes. In AVS, it is possible to have scenes in which multiple cameras, or views, are employed. Such an arrangement can be used to present orthogonal views of a given object. Our camera module does not communicate directly with the AVS kernel. Instead, all state information is maintained internally. This design was chosen primarily due to limitations in the AVS CLI interface [6] which do not permit querying of camera parameters other than a "matrix." The camera matrix is not separable into components such as eyepoint, look-at point, and so forth due to the presence of a non-uniform scaling component along each axis as well as a translational component. Multiple camera configurations can be supported by using multiple instantiations of the camera transformation module.

## 3.3 Object Manipulation

Unlike the camera module, which does not obtain any state information from the AVS kernel, the object manipulation module needs to know which object is currently selected as well as information about the current view. Input events are assimilated into a transformation matrix which is then concatenated onto the current object's transformation matrix.

### 3.3.1 Transformation matrix construction:

Let $V^{-1}$ represent the inverse of the view matrix, and $P^{-1}$ represents the inverse of the parent's transformation matrix. The transformation vector, $\vec{t}$, represents the translational component of the VR input device event.

The new transformation matrix, $T_{new}$, is computed thus:

$$\vec{t}' = \vec{t}(V^{-1}P^{-1})$$

$$T_{new} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \vec{t}'_x & \vec{t}'_y & \vec{t}'_z & 1 \end{bmatrix}$$

Unlike translations, which are commutative, some convention must be used in applying rotations, which are not commutative. For example, a rotation of A degrees about the X axis, followed by a rotation of B degrees about the Z axis, is not necessarily the same as a rotation of B degrees about the Z axis followed by a rotation of A degrees about the X axis.

The convention that we have implemented, which seems to work well, is to first consider the entire rotation vector, $\vec{r}$, as consisting of three components, ($\vec{r}_x$, $\vec{r}_y$, $\vec{r}_z$); the angular amounts by which we will rotate the given object about each axis. Next, we will conceptually apply three separate rotation operations to the object. In order, the first operation will be the rotation with the greatest angular magnitude. In other words, from ($\vec{r}_x$,$\vec{r}_y$,$\vec{r}_z$), if $\vec{r}_z$ is the greatest in magnitude, we will apply the Z-axis rotation first, followed by whichever of $\vec{r}_y$ and $\vec{r}_x$ is greater in magnitude, followed by that which is least in magnitude. The rationale motivating this convention is the observation that the human who is operating the system will generate the greatest amount of rotation about the axis which is the most important to them, and there will inherently be "noise," "user jitters" or just ordinary ambient values generated on the other axes. However, this convention does not preclude what will appear to the user to be rotations about two axes simultaneously.

To construct the rotation transformation matrix, we let $V^{-1}$ represent the view inverse matrix, and $P^{-1}$ the inverse of the parent's transformation matrix. The rotational components of the input device event are in a vector $\vec{r}$, with components $\vec{r}_x$, $\vec{r}_y$ and $\vec{r}_z$. Note that $\vec{r}_x$, $\vec{r}_y$ and $\vec{r}_z$ are rotational events generated in VR input device coordinates (user space). We want to compute the angles $\vec{r}'_x$, $\vec{r}'_y$ and $\vec{r}'_z$, in object space, which will achieve the desired rotation.

$$\vec{r}' = \vec{r}(V^{-1}P^{-1})$$

Once we have these angular values, in $\vec{r}'$, we can compute the axis about which to rotate first, second, then third and construct transformation matrices appropriately [7], and then concatenate them via matrix multiplication into a single rotation matrix. Note that a single rotational value, a z-axis rotation event generated by the input device, for example, may produce rotations in object space about any or all of the principle axes.

State information about the current object is obtained via the AVS Command Line Interface (CLI) facility, whereby a query is sent to the AVS kernel requesting the name of the current object, as well as requesting the current view matrix. We decided to go this route, rather than to use the "upstream transform" data structure, due to numerical inconsistencies (bugs) in the matrices contained in the "upstream transform" structure. The user picks the object to transform using the **geometry viewer**, rather than by entering an object name on the object transform module. This design choice keeps the user interface consistent: object picks always occur in the **geometry viewer**.

### 3.3.2 Limitations of the Object Transform Module

There are two limitations to this object transform module. First, the view matrix for "camera1" is always queried via the CLI interface. This means that this module will not necessarily compute the correct view inverse for multiple-camera views. This is really a limitation of CLI, rather than this module. A CLI command of the form "give me the view matrix for the currently selected camera" rather than "give me the matrix for camera1" would remedy this limitation.

Second, as there is no way in AVS to obtain information about an object's parent, or children, via CLI, it is impossible to accurately construct an hierarchy of transformations which is more than two levels deep. A CLI command of the form "give me the name of the parent of object <name>" would remedy this limitation, in part. Additional code would be required in the module itself to build the transformation hierarchy.

## 3.4 Miscellaneous Tools

Two modules, **VR to floats** and **VR to accumulated floats** will accept as input an AVS field constructed by one of the input modules (**spaceball** or **dummy device**), and will separate the events into scalar components. In this way, y-axis rotation events, for example, can be directly wired to any scalar floating point parameter in the system, such as isosurface threshold level.

**VR print field** will display the contents of the field in which VR input device events have been encoded. The usual module for displaying the contents of fields, **print field**, is not appropriate for printing fields containing encoded VR device events. This is due to the fact that both logical and floating point data are encoded into the these fields.

## 4.0 Construction of Direct Manipulation Interfaces

In this section, we will show how a VR input device can be used to manipulate visualization tools in AVS which have some type of DMI, as well as interfacing a VR input device to a single scalar parameter.

The first example, Figure 3, demonstrates the use of a VR input device to control the spatial location of a data probe. The bulk of this network is available to all users from the “Help Demos” in the AVS Network Editor. To the demonstration probe network, we simply add the modules **spaceball** and **VR Transform Object**. When the user selects the probe object within the geometry viewer, making it the current object, the input device then may be used to position the probe.

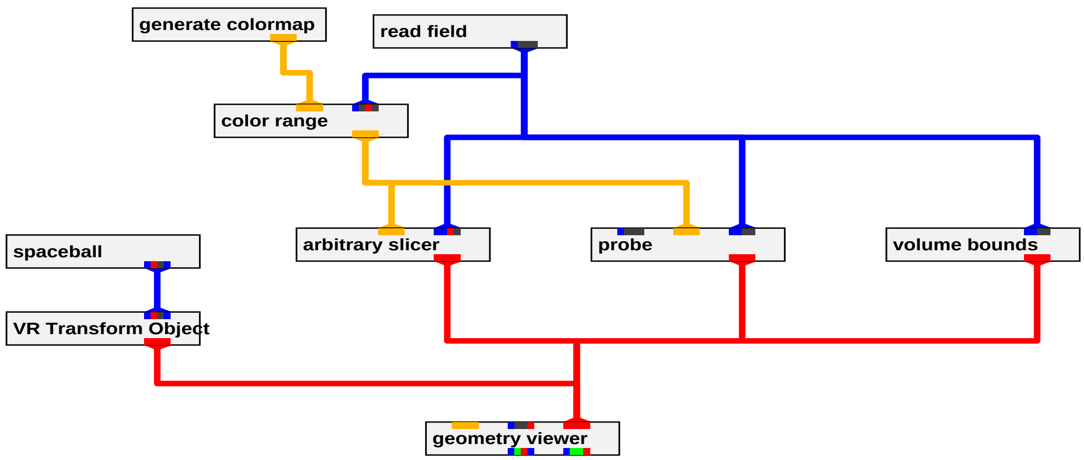



Figure 3 - VR Input Device Controls Location of Data Probe

The second example is illustrated in Figures 4 and 5. The network, shown in Figure 4, reads the five-channel “Bluntfin” CFD dataset [8] which is distributed with AVS. The data is read in using the **read field** module. The **stream lines** module is used to compute streamlines through this data,

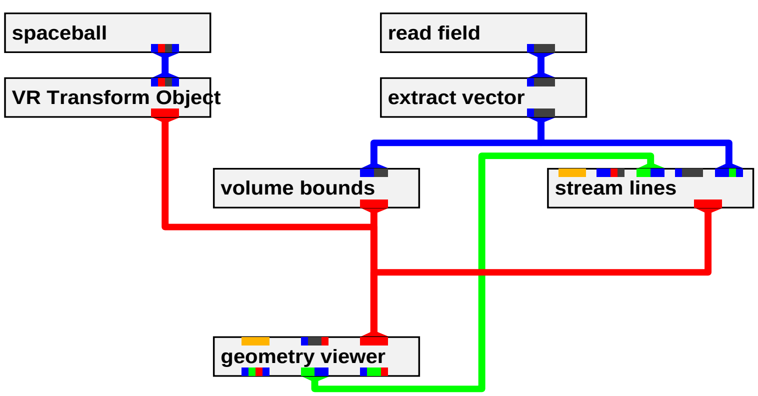



Figure 4.

VR Input Device Control of Seed Point Location for Streamlines Computation.

using seed points which lie on a plane. The **spaceball** module provides input to the **VR Xform Object** module which provides transformation events to the geometry viewer. The upstream transformation connection between **geometry viewer** and **stream lines** has been revealed for illustrative purposes. When the streamline object is made the current object in the **geometry viewer**, it may be transformed with the VR input device. The **stream lines** module will then recompute a new set of streamlines based upon the updated position of the seed point, and the resulting streamlines are then dispatched off to the **geometry viewer** for rendering and display to the user (Figure 5).

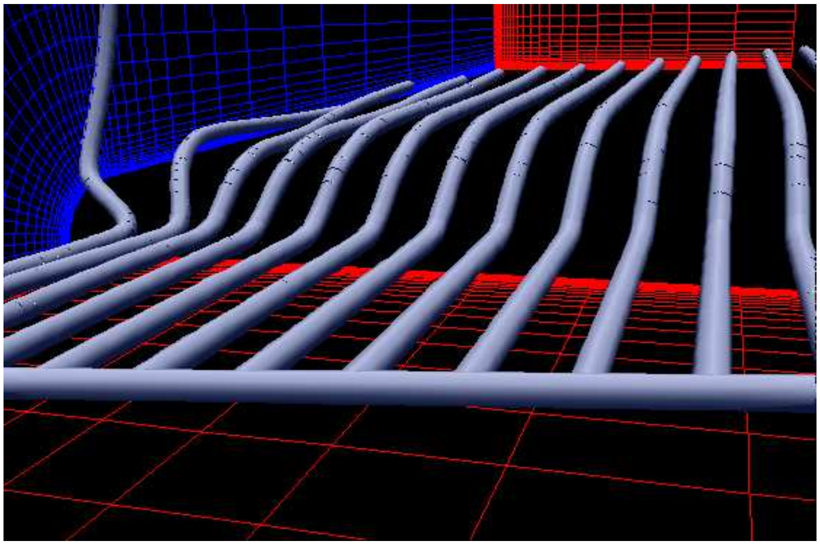


Figure 5.

Streamlines through a flow field.

The final example demonstrates how to use the VR input device to control a scalar parameter. In this case, we allow the y-axis rotations from the input device to control the threshold value on the isosurface module. This example is contrived, but demonstrates the functionality. The network is shown in Figure 6.

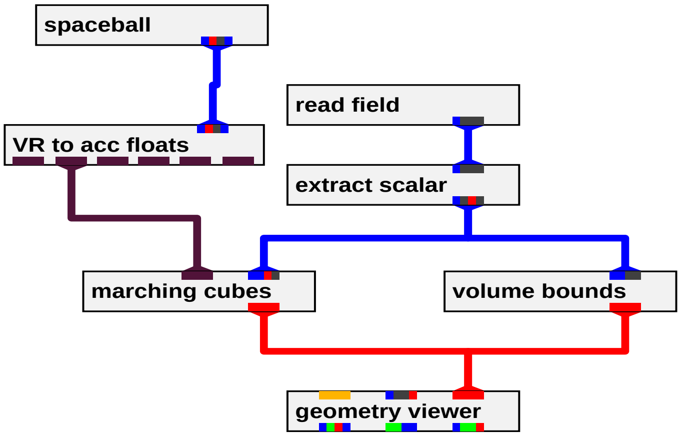



Figure 6.

VR Input Device Control of a Scalar Parameter.

## 5.0 Conclusion

Despite media bombardment that would have us believe that one must blast winged reptiles with laser bazookas in a pastel, low-resolution dreamscape displayed within the confines of a head-mounted display, we demonstrate that off-the-shelf, low-cost VR input devices have usefulness and practicality in a day-to-day scientific visualization environment.

We have built and released via anonymous ftp a set of tools which allow a scientist to use a six dimensional input device to control direct manipulation interfaces to scientific visualization tools. Built within the framework of a modular visualization environment, the tools can be used to control a wide gamut of processes, ranging from desktop viewpoint changes to controlling a compute intensive resource in a distributed computing environment.

## 6.0 Acknowledgment

This work was supported by the Director of the Scientific Computing Staff of the U. S. Department of Energy under Contract No. DE-AC03-76SF00098.

## 7.0 References

[1] Earnshaw, R., Gigante, M., Jones, H. (eds.), Virtual Reality Systems, Academic Press, 1993.

[2] Bryson, S. and Levitt, C., "The Virtual Windtunnel: An Environment for the Exploration of Three-Dimensional Unsteady Flows," Visualization 91, pp17-24, October 1991.

[3] Meyer, Tom and Globus, Al, Direct Manipulation of Isosurfaces and Cutting Planes in Virtual Environments, NAS Technical Report RNR-93-019, 1993, NAS Systems Division, NASA Ames Research Center, 1993. http://www.nas.nasa.gov/NAS/reports/nasreports.html

[4] Lorensen, William E., and Cline, H. E., "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," Computer Graphics 21:4, Proceedings of Siggraph 87, pp 163-169, July 1987.

[5] Anton, H., Elementary Linear Algebra, John Wiley & Sons, Inc., 1991.

[6] "AVS Developer's Guide," Release 4, May 1992.

[7] Newman, W., and Sproull, R., Principles of Interactive Computer Graphics, McGraw-Hill, 1973.

[8] Hung, C. M. and Buning, P. G., "Simulation of Blunt-Fin Induced Shock Wave and Turbulent Boundary Layer Separation," AIAA Paper 84-0457, AIAA Aerospace Sciences Conference, Reno NV, January 1984.