Virtual Environment Software System: Mathematics Library

1-1-2000

Bryan Kline
Jason Daly
Glenn A. Martin

Find similar works at: https://stars.library.ucf.edu/istlibrary

University of Central Florida Libraries http://library.ucf.edu

Recommended Citation

Kline, Bryan; Daly, Jason; and Martin, Glenn A., "Virtual Environment Software System: Mathematics Library" (2000). Institute for Simulation and Training. 208.
https://stars.library.ucf.edu/istlibrary/208

This Research Report is brought to you for free and open access by the Digital Collections at STARS. It has been accepted for inclusion in Institute for Simulation and Training by an authorized administrator of STARS. For more information, please contact lee.dotson@ucf.edu.
Virtual Environment Software System

Mathematics Library

Bryan Kline, Glenn Martin and Jason Daly

March 2, 2000
Virtual Environment Software System (VESS)

Mathematics Library

Bryan Kline, Glenn Martin, and Jason Daly

March 2, 2000

Institute for Simulation and Training
University of Central Florida
3280 Progress Drive
Orlando, FL 32826

IST-TR-00-02
Introduction

The Virtual Environment Software System (VESS) is a suite of libraries developed by the Visual Systems Laboratory at the University of Central Florida’s Institute for Simulation and Training (IST). It is based on lessons learned from the Virtual Environment Library (VEL) which was previously developed by IST and used to create the software for various virtual reality applications. The goal of VESS is to provide an application base that is useful and functional using today’s hardware and graphics libraries, extensible to support future hardware and graphics libraries, and easily portable to multiple platforms, graphics systems, and application programming interfaces (API’s).

VESS is designed to simplify and expedite the development of applications where virtual environments are required. It does this by providing a simple interface into the underlying graphics API while integrating support for various input devices, such as joysticks and motion tracking systems, and display devices, such as head-mounted displays and shutter glasses. Additionally, VESS provides behaviors and motion models to allow the user to manipulate his or her viewpoint as well as control and interact with objects in the virtual environment. The user's viewpoint can be independent or attached to any transformable object in the scene. Other useful routines such as collision detection and terrain following are also provided.

VESS is designed for easy portability. Its multi-layered architecture allows the developer to focus on the details of the application, without worrying about the specifics of the graphics API or hardware interfaces. Thus, applications built using the VESS libraries will be easily portable to any other supported platform. Currently, VESS runs on IRIX and Linux platforms using the SGI Performer API. Other platforms and APIs will be supported in the future.

Purpose of this document

This document describes the typical usage of the VESS Mathematics Library, as well as the internal operations of many of the routines contained within. This information is provided to allow for easy understanding of the intent and function of the library routines, for purposes of both usability and modifiability of the library. This document is intended to assist developers in using the library, and should also be useful when changes or additions need to be made to the library itself.
Library Overview

The VESS Math Library is a general-purpose library that implements commonly used mathematical data structures and functions. Currently, the math library contains three different classes: a vector class, a matrix class, and a quaternion class. These classes are structured in such a way as to be useful in the majority of graphics applications, such as those written with the rest of the VESS libraries, though they may not necessarily be appropriate for other math applications. The math library is completely self-sufficient and does not utilize any of the data structures or functions that the underlying graphics library may have available.

Euler Angles

The term Euler angles refers to the usage of a set of three angle measures commonly used in computer graphics. Each angle represents an amount of rotation around one of the three basic coordinate axes in three-dimensional space. When used with three distinct coordinate axes, these three rotations together represent all of the possible three-dimensional coordinate space rotations. Euler angle rotations of an object are typically specified as the ‘heading’ (or ‘yaw’), ‘pitch’, and ‘roll’ of that object; while the specific axes of rotation for each value tend to vary between graphical system implementations, heading or yaw generally indicates rotation left or right with respect to the direction of travel, pitch indicates rotation up or down with respect to direction of travel, and roll indicates a spin rotation around the axis of travel (think ‘barrel roll’ of an airplane). Generally, rotations follow the ‘right-hand rule’: with your right hand, curl your fingers into a fist but leave your thumb pointing out. Positive-valued rotation angles around the rotation axis, indicated by the direction your thumb is pointing, rotate points in the direction indicated by the curl of your fingers (counterclockwise).

Despite their usefulness and widespread acceptance, Euler angles are not commonly used in the VESS libraries due to some of their inherent problems. Euler angles tend to be difficult to manipulate and prone to mathematical singularities such as ‘gimbal lock’, a phenomenon sometimes encountered while using Euler rotations where two of the three coordinate axes coincide, resulting in a loss of stability and flexibility. As a result, rotation matrices or rotational quaternions are the preferred method of working with coordinate system rotations.

The VESS matrix and quaternion classes include routines for converting rotations to and from Euler angle specifications. These routines, in order to work properly, need information about which coordinate axis each separate Euler rotation is to be made around. This information is provided in the form of an axis order constant of type vsMathEulerAxisOrder. These axis order constants are enumerated in the VESS global header file vsGlobals.h++. Each constant specifies a different ordering of coordinate axes as well as the treatment of those axes during the rotations. For example, specifying the constant VS_EULERANGLES_ZYX_S in an Euler conversion function indicates that the first Euler rotation rotates points counterclockwise around the Z-axis, the second counterclockwise around the Y-axis, and the third counterclockwise around the X-axis.

Each of the axis order constants has one extra letter at the end, either ‘S’ or ‘R’. These letters stand for Static or Rotating coordinate axes, respectively. The distinction between the two has to do with the behavior of the coordinate axes themselves during the rotations. Static axes stay fixed in place during the rotation, while rotating axes move along with the points being rotated. With static axes, rotations around the same axis will always move points in the same way, regardless of any other factors. The coordinate axes stay fixed so rotations around those axes always behave the same. Conversely, rotating coordinate axes move right along with their points. Two rotations around the same coordinate axis will move points in different directions if there is an intervening rotation around a different axis. The intervening rotation has rotated the coordinate axes as well, so the new coordinate axes are different when it comes time to rotate around the same axis as the first rotation.
**Quaternions**

The math library class `vsQuat` implements a mathematical data structure known as a quaternion. A quaternion is a four-element array of floating-point values with many interesting mathematical properties. Quaternions are conceptually divided into two parts: a vector part, consisting of three elements, and a scalar part, which is the fourth element.

In the VESS libraries, quaternions are primarily used to represent coordinate system rotations, such as what would normally be the purpose of a 4x4 rotational transformation matrix. Rotational quaternions, on their simplest level, represent a coordinate system rotation using an axis of rotation (the vector part), and an amount of rotation (the scalar part). The usage of quaternions to represent rotations avoids many of the ambiguities and singularities present in traditional Euler angle rotation representations. Additionally, they are more efficient than using rotation matrices to store the same data. Rotational quaternions can be multiplied together in the same manner as rotation matrices to obtain composite rotations. Interpolation of rotational quaternions is also easier and is subject to fewer sudden changes in movement than Euler angle interpolations.

For more information on the theories and other uses of quaternions, see any one of a number of papers written by Ken Shoemake, including "Animating Rotation with Quaternion Curves", SIGGRAPH '85 paper, volume 19(3) pp. 245-254 (1985).

**Library Classes**

**vsVector**

The vsVector class implements a variable sized array of double-precision floating-point numbers. The maximum size for this array is 4 elements. The vsVector class supports many operations common to standard mathematical vectors, such as sums and differences, comparisons, and vector dot and cross products. The vsVector class enforces some size limitations on many of its operations; for example, vectors to be added together must be the same size. The vsVector class also performs bounds checks when its elements are being accessed as members of an array and can signal out-of-bounds errors based on the current size of the array.

**Implementation Notes**

Although the vsVector can vary its current size, it is always implemented internally as an array of four numbers. The elements of the array beyond the current size of the vector are simply ignored. Changing the size of the vector generally does not affect the contents of any of the vector elements. Attempting to access elements of the vector outside of its current size range, with either the `getValue()` and `setValue()` functions or the array index operator, signals an error. Additionally, many of the vsVector functions check the size of their operands before performing their operation and may return a zero vector if the input vectors' sizes are not satisfactory. Generally when performing operations with two vectors, the two vectors should be the same size. The `setSize()` function can be used to change the vector size if needed.

Several functions have the capability of changing the size of the vector. `setSize()` directly alters the vector size. Both `copy()` and `clearCopy()` make duplicates of vectors, size included. The constructors that take initial data, as well as the `set()` functions, set the size of the vector to the number of data elements passed in; i.e. a call to `set(a, b, c)` sets the vector size to 3. Under no circumstances can the vector size be set to greater than 4 or less than 1.
The `clearCopy()` function is useful for 'extending' a vector. The function makes a copy of a vector, setting all of the unused elements of the vector to zero. If the vector's size is increased afterwards, the new elements contain zero. This is helpful, for example, in using a two-dimensional coordinate as a three-dimensional point; the third (z) value ends up being zero.

The functions `isEqual()` and `isAlmostEqual()` both check for element-wise equality between two vectors. Two elements are equal if they are within a certain tolerance range of each other; the `isEqual()` function has this tolerance range pre-set, while the `isAlmostEqual()` function allows the user to specify what the tolerance value should be. This is done because exact equality between floating-point numbers is not generally very useful; due to roundoff, two computed values may be 'equal' to any person viewing the decimal equivalents, though they may not have exactly the same bit pattern in memory. `isAlmostEqual()` may be called with a tolerance of zero if exact equality is desired.

Many of the functions specified in the `vsVector` class have two versions. One version of the function copies the result of the function's computation to the matrix. The other version returns the result of the computation in a separate `vsVector` object. In all other aspects, the two functions should behave in the same manner. Additionally, all of the overloaded operator functions (with the exception of `operator[]`) behave in the same manner as the standard functions to which they relate. The majority of these overloaded operators also have two versions that only differ by the destination of the return value. For example, all of the following lines of code perform the same task:

```c++
// Adds 'b' directly to 'a'
a.add(b);
a += b;

// Adds 'a' and 'b' together, result assigned back to 'a'
a = a.getSum(b);
a = a + b;
```

**Method Descriptions**

`vsVector()`
`vsVector(int size)`
`vsVector(int size, double values[])`
`vsVector(double x, double y)`
`vsVector(double x, double y, double z)`
`vsVector(double x, double y, double z, double w)`

Constructors for the `vsVector`. The default constructor clears the vector to zero and sets the size to 4, while the second constructor clears the vector and sets the size to `size`. The rest of the constructors set the data within the vector; the third constructor sets the vector size and data to `size` and `values`, respectively, while the remaining three set the data to the given values and the size to the number of those values.

`~vsVector()`

Destructor for the `vsVector`. Does nothing.

`void set(int size, double values[])`
`void set(double x, double y)`
`void set(double x, double y, double z)`
`void set(double x, double y, double z, double w)`

Sets the data and size of the vector. The first function sets the vector size to `size`, and the vector data to the data in `values`. The other three functions set the vector data to the parameters given, and the vector size to the number of those parameters.
void copy(vsVector source)
   Makes the vector an exact copy of source.

void clear()
   Clears the vector data to zero. The size is unchanged.

void clearCopy(vsVector source)
   Makes the vector a copy of the source vector, setting all of the data values above the vector’s size to zero.

void setSize(int size)
   Sets the size of the vector.

int getSize()
   Returns the size of the vector.

void setValue(int index, int value)
   Sets one element of the vector, specified by index, to value.

double getValue(int index)
   Retrieves the element of the vector specified by index.

int isEqual(vsVector operand)
int isAlmostEqual(vsVector operand, double tolerance)
   Compares the vector to operand for element-wise equality. Two elements are equal if within a tolerance value of each other. The tolerance is predefined for isEqual(), and specified by tolerance for isAlmostEqual(). The two vectors must be the same size.

void add(vsVector addend)
vsVector getSum(vsVector addend)
   Computes the sum of this vector with addend. add() sets the vector to the sum, while getSum() returns the sum. The two vectors must be the same size.

void subtract(vsVector subtrahend)
vsVector getDifference(vsVector subtrahend)
   Computes the difference of this vector with subtrahend. subtract() sets the vector to the difference, while getDifference() returns the difference. The two vectors must be the same size.

void scale(double multiplier)
vsVector getScaled(double multiplier)
   Computes the vector with each element multiplied by multiplier. scale() sets the vector to the result, while getScaled() returns the result.

double getMagnitude()
   Returns the magnitude of the vector.

double getDotProduct(vsVector operand)
   Returns the dot product of the vector with the operand vector. The two vectors must be the same size.

void normalize()
vsVector getNormalized()
   Computes the normalized form of the vector. normalize() sets the vector to the result, while getNormalized() returns the result.
void crossProduct(vsVector operand)
vsVector getCrossProduct(vsVector operand)
    Computes the cross product of the vector with the operand vector. crossProduct() sets the vector
to the result, while getCrossProduct() returns the result. The two vectors must be of size 3.

double getAngleBetween(vsVector endVector)
    Returns the angle, in degrees, formed by this vector and endVector. The range of the return value is
between -180.0 and 180.0 degrees, inclusive.

double &operator[](int index)
    Returns a reference to one of the elements of the vector.

vsVector operator+(vsVector addend)
    Equivalent to getSum(addend).

vsVector operator-(vsVector subtrahend)
    Equivalent to getDifference(subtrahend).

vsVector operator*(double multiplier)
    Equivalent to getScaled(multiplier).

void operator+=(vsVector addend)
    Equivalent to add(addend).

void operator-=(vsVector subtrahend)
    Equivalent to subtract(subtrahend).

void operator*=(double multiplier)
    Equivalent to scale(multiplier).

int operator==(vsVector operand)
    Equivalent to isEqual(operand).

vsVector operator*(double multiplier, vsVector operand)
    Non-member function. Returns a vector with values equal to the elements of operand multiplied by
multiplier, and size equal to that of operand.

vsMatrix

The vsMatrix class implements a 4x4 matrix of double-precision floating-point numbers. While
matrices of this size are not useful for all purposes, they are very useful in computer graphics for storing
complex coordinate system transformations; this is their primary purpose within the VESS libraries.
vsMatrices can be set to contain coordinate transformations, multiplied together to produce composite
transformations, and applied to transform points or directional vectors. Though the class is capable of
performing other computations, graphics point manipulation is their main use within VESS.

Implementation Notes

The vsMatrix class contains a 4x4 matrix of values. Internally, these values are stored as a set of four
double four-element vsVector objects, each object storing data for one row of the matrix. The array index operator
returns a reference to one of these vsVectors; that operator[] function, combined with a similar function
in the vsVector class, allows reference to the values of the vsMatrix as matrix[row][column], just as
if the vsMatrix were a typical two-dimensional array.
The functions that set a vsMatrix as a rotation, translation, or scale matrix modify the entire matrix; i.e. if setTranslation() is followed by setScale(), the resulting matrix will only contain a scale transformation. If a composition of the two is desired, two matrices should be used and multiplied together.

Points or vectors to be transformed by a matrix are multiplied as column vectors on the right of the matrix. The getPointXform(), getVectorXform(), and getFullXform() functions transform the input vector by the matrix contained within. getPointXform() treats the input vector as a three-dimensional point, assuming the fourth value to be 1 so that coordinate translations in the matrix operate properly. getVectorXform() works similarly, but for three-dimensional vectors instead of points, and assumes the fourth value to be 0; this effectively disables translations, which shouldn’t affect a directional vector. getFullXform() uses the fourth value of the vector as it is; this is useful for homogeneous coordinate points, where the fourth element acts as a built-in scale value.

The functions isEqual() and isAlmostEqual() both check for element-wise equality between two matrices. Two elements are equal if they are within a certain tolerance range of each other; the isEqual() function has this tolerance range pre-set, while the isAlmostEqual() function allows the user to specify what the tolerance value should be. This is done because exact equality between floating-point numbers is not generally very useful; due to roundoff, two computed values may be ‘equal’ to any person viewing the decimal equivalents, though they may not have exactly the same bit pattern in memory. isAlmostEqual() may be called with a tolerance of zero if exact equality is desired.

Many of the functions specified in the vsMatrix class have two versions. One version of the function copies the result of the function’s computation to the matrix. The other version returns the result of the computation in a separate vsMatrix object. In all other aspects, the two functions should behave in the same manner. Additionally, all of the overloaded operator functions (with the exception of operator[]) behave in the same manner as the standard functions to which they relate. The majority of these overloaded operators also have two versions that only differ by the destination of the return value. For example, all of the following lines of code perform the same task:

```cpp
// Adds 'b' directly to 'a'
    a.add(b);
    a += b;

// Adds 'a' and 'b' together, result assigned back to 'a'
    a = a.getSum(b);
    a = a + b;
```

The vsMatrix functions that deal with Euler angles require an Euler axis order constant as one of the parameters. This axis constant specifies to the function the desired order of coordinate axes around which the Euler rotations should be made; this is the order that the following three angle degree parameters should be in. Additionally, the axis order constant specifies whether static or rotating coordinate axes are to be used. Static axes refer to the coordinate axes staying ‘fixed’ in place while points are rotated. With static axes, rotations around a particular coordinate axis will always move points in the same general way, regardless of anything else. Conversely, rotating axes rotate right along with the points when a rotation occurs. Using rotating axes, two rotations around the same axis will move points in different directions if there is an intervening rotation around a different coordinate axis.

Method Descriptions

vsMatrix()
vsMatrix(double values[4][4])

Constructors for the vsMatrix. The default constructor clears the matrix to zero. The second constructor sets the data in the matrix to the data in values.
- vsMatrix()
  Destructor for the vsMatrix. Does nothing.

void set(double values[4][4])
  Sets the matrix equal to the data in values.

void copy(vsMatrix source)
  Makes the matrix an exact copy of source.

void clear()
  Sets the matrix to zero.

double getValue(int row, int column)
  Retrieves the element of the matrix specified by row and column.

void setValue(int row, int column, double value)
  Sets one element of the matrix, specified by row and column, to value.

int isEqual(vsMatrix operand)
int isAlmostEqual(vsMatrix operand, double tolerance)
  Compares the matrix to operand for element-wise equality. Two elements are equal if within a
tolerance value of each other. The tolerance is predefined for isEqual(), and specified by
tolerance for isAlmostEqual().

void add(vsMatrix addend)
vsMatrix getSum(vsMatrix addend)
  Computes the sum of this matrix with addend. add() sets the matrix to the sum, while getSum() returns the sum.

void subtract(vsMatrix subtrahend)
vsMatrix getDifference(vsMatrix subtrahend)
  Computes the difference of this matrix with subtrahend. subtract() sets the matrix to the
difference, while getDifference() returns the difference.

void scale(double multiplier)
vsMatrix getScaled(double multiplier)
  Computes the matrix with each element multiplied by multiplier. scale() sets the matrix to the result, while getScaled() returns the result.

void transpose()
vsMatrix getTranspose()
  Computes the transpose of the matrix. transpose() sets the matrix to the result, while
getTranspose() returns the result.

void preMultiply(vsMatrix operand)
vsMatrix getPreMultiplied(vsMatrix operand)
  Multiplies this matrix by operand on the left. preMultiply() sets the matrix to the result, while
getPreMultiplied() returns the result.

void postMultiply(vsMatrix operand)
vsMatrix getPostMultiplied(vsMatrix operand)
  Multiplies this matrix by operand on the right. postMultiply() sets the matrix to the result,
while getPostMultiplied() returns the result.
vsVector getPointXform(vsVector operand)
vsVector getVectorXform(vsVector operand)
vsVector getFullXform(vsVector operand)

Multiplies the operand vector by this matrix as a column vector on the right, returning the result.

getPointXform() treats operand as a 3-dimensional point; it assumes the fourth element of the
vector to be 1, so that translations contained within the matrix function properly.

getVectorXform() treats operand as a 3-dimensional vector with the fourth element assumed to
be 0; coordinate translations shouldn't affect vectors. getFullXform() uses the entire four
elements of operand; a standard matrix/vector multiplication is performed. All of these functions
return a zero vector if the operand vector is too small. (vsVectors of at least size 3 are needed for
point or vector transforms, while size 4 is needed for a full transform.)

void setIdentity()
Sets the matrix to the identity matrix.

void setEulerRotation(vsMathEulerAxisOrder axisOrder,
                        double axis1Degrees, double axis2Degrees,
                        double axis3Degrees)
Sets the matrix to a rotation matrix equivalent to the specified Euler angle rotations. The order of axes
for the rotations are specified by the axisOrder constant; the three axis*Degrees parameters
must be in that order.

void getEulerRotation(vsMathEulerAxisOrder axisOrder,
                        double *axis1Degrees, double *axis2Degrees,
                        double *axis3Degrees)
Reconstructs a set of Euler angle rotations from the matrix. The order of axes for the rotations is
specified by the axisOrder constant; the returned rotations are given in that order. NULL pointers
may be passed in for return values that aren’t needed.

void setQuatRotation(vsQuat quat)
Sets the matrix to a rotation matrix equivalent to the rotation specified by quat.

void setTranslation(double dx, double dy, double dz)
Sets the matrix to a translation matrix that translates points by the distances given in dx, dy, and dz.

void setScale(double sx, double sy, double sz)
Sets the matrix to a (possibly non-uniform) scaling matrix that scales by the values given in sx, sy,
and sz.

vsVector &operator[](int index)
Returns a reference to one of the vsVectors containing a row of the matrix.

vsMatrix operator+(vsMatrix addend)
Equivalent to getSum(addend).

vsMatrix operator-(vsMatrix subtrahend)
Equivalent to getDifference(subtrahend).

vsMatrix operator*(vsMatrix operand)
Equivalent to getPostMultiplied(operand).

void operator+=(vsMatrix addend)
Equivalent to add(addend).
void operator-=(vsMatrix subtrahend)
   Equivalent to subtract(subtrahend).

int operator==(vsMatrix operand)
   Equivalent to isEqual(operand).

vsQuat

The vsQuat class implements a quaternion; a four-element array of double-precision floating-point values. Quaternions are useful in computer graphics for representing coordinate system rotations. They are more efficient than matrices for the same work, and have inherently fewer problems than Euler angles. vsQuats are generally only useful for quaternion operations; for other purposes requiring a four-element array, a vsVector of size 4 should be used instead.

Implementation Notes

A vsQuat contains an array of four double-precision floating-point numbers. The first three of these numbers represent the ‘vector’ part of the quaternion (x, y, z), and the fourth number is the ‘scalar’ part (w). Overall, the value of the quaternion is taken to be $q = xi + yj + zk + w$. The VESS constants VS_X, VS_Y, VS_Z, and VS_W, together with the array index operator, allow for easy manipulation of each specific value.

The rotatePoint() function transforms a point by the rotation represented by the quaternion, in the same way as a rotation matrix multiplied on the left would be. Internally, the function computes $Q * P * -Q$, where $Q$ is the quaternion, $P$ is the point (treated as a quaternion), and $-Q$ is the conjugate of the quaternion. (Technically, the equation is supposed to use the inverse of the quaternion, but for unit quaternions the inverse and the conjugate are the same.)

The slerp() function, standing for Spherical Linear intERPolation, does a linear path interpolation between the quaternion and another quaternion. However, although the path is linear, the velocity along the path varies due to the inclusion of trigonometric functions in the ratio evaluation, meriting the function’s ‘spherical’ designation. The interpolations generated by this function are more suited for camera angle interpolations that standard linear interpolation.

The isEqual() and isAlmostEqual() both check for element-wise equality between two quaternions. Two elements are equal if they are within a certain tolerance range of each other; the isEqual() function has this tolerance range pre-set, while the isAlmostEqual() function allows the user to specify what the tolerance value should be. This is done because exact equality between floating-point numbers is not generally very useful; due to roundoff, two computed values may be ‘equal’ to any person viewing the decimal equivalents, though they may not have exactly the same bit pattern in memory. isAlmostEqual() may be called with a tolerance of zero if exact equality is desired.

Many of the functions specified in the vsQuat class have two versions. One version of the function copies the result of the function’s computation to the matrix. The other version returns the result of the computation in a separate vsQuat object. In all other aspects, the two functions should behave in the same manner. Additionally, all of the overloaded operator functions (with the exception of operator[]) behave in the same manner as the standard functions to which they relate. The majority of these overloaded operators also have two versions that only differ by the destination of the return value. For example, all of the following lines of code perform the same task:
// Adds 'b' directly to 'a'
a.add(b);
a += b;

// Adds 'a' and 'b' together, result assigned back to 'a'
a = a.getSum(b);
a = a + b;

The vsQuat functions that deal with Euler angles require an Euler axis order constant as one of the parameters. This axis constant specifies to the function the desired order of coordinate axes around which the Euler rotations should be made; this is the order that the following three angle degree parameters should be in. Additionally, the axis order constant specifies whether static or rotating coordinate axes are to be used. Static axes refer to the coordinate axes staying ‘fixed’ in place while points are rotated. With static axes, rotations around a particular coordinate axis will always move points in the same general way, regardless of anything else. Conversely, rotating axes rotate right along with the points when a rotation occurs. Using rotating axes, two rotations around the same axis will move points in different directions if there is an intervening rotation around a different coordinate axis.

Method Descriptions

vsQuat()  
vsQuat(double x, double y, double z, double w)  
vsQuat(double values[])  
Constructors for the vsQuat. The default constructor clears the quaternion to zero. The second constructor sets the quaternion to that specified by (x, y, z, w). The third constructor sets the quaternion to the data in the values array; the array is assumed to contain four elements.

~vsQuat()  
Destructor for the vsQuat. Does nothing.

void set(double x, double y, double z, double w)  
void set(double values[])  
Sets the data in the quaternion. The first function sets the quaternion equal to (x, y, z, w), while the second function sets the quaternion equal to the data in the values array, which is assumed to contain four elements.

void copy(vsQuat source)  
Matrix the quaternion an exact copy of the source quaternion.

void clear()  
Sets the quaternion to zero.

void setValue(int index, double value)  
Sets one element of the quaternion, specified by index, to value.

double getValue(int index)  
Retrieves the element of the quaternion specified by index.

int isEqual(vsQuat operand)  
int isAlmostEqual(vsQuat operand, double tolerance)  
Compares the quaternion to operand for element-wise equality. Two elements are equal if within a tolerance value of each other. The tolerance is predefined for isEqual(), and specified by tolerance for isAlmostEqual().
void add(vsQuat addend)
vsQuat getSum(vsQuat addend)

Computes the sum of this quaternion with addend. add() sets the quaternion to the sum, while
getSum() returns the sum.

void subtract(vsQuat subtrahend)
vsQuat getDifference(vsQuat subtrahend)

Computes the difference of this quaternion with subtrahend. subtract() sets the quaternion to
the difference, while getDifference() returns the difference.

void scale(double multiplier)
vsQuat getScaled(double multiplier)

Computes the quaternion with each element multiplied by multiplier. scale() sets the
quaternion to the result, while getScaled() returns the result.

void multiplyQuat(vsQuat operand)
vsQuat getMultipliedQuat(vsQuat operand)

Computes the quaternion multiplied by the operand quaternion. multiplyQuat() sets the
quaternion to the result, while getMultipliedQuat() returns the result.

double getMagnitude()

Returns the magnitude of the quaternion.

void normalize()
vsQuat getNormalized()

Computes the normalized form of the quaternion. normalize() sets the quaternion to the result,
while getNormalized() returns the result.

void conjugate()

Sets the quaternion to its conjugate. The conjugate of a quaternion is the quaternion with the three
vector values negated.

void setMatrixRotation(vsMatrix theMatrix)

Sets the quaternion to a rotational quaternion equivalent to the rotation specified by theMatrix.

void setEulerRotation(vsMathEulerAxisOrder axisOrder,
double axis1Degrees, double axis2Degrees,
double axis3Degrees)

Sets the quaternion to a rotational quaternion equivalent to the specified Euler angle rotations. The
order of axes for the rotations are specified by the axisOrder constant; the three axis*Degrees
parameters must be in that order.

void getEulerRotation(vsMathEulerAxisOrder axisOrder,
double *axis1Degrees, double *axis2Degrees,
double *axis3Degrees)

Reconstructs a set of Euler angle rotations from the quaternion. The order of axes for the rotations is
specified by the axisOrder constant; the returned rotations are given in that order. NULL pointers
may be passed in for return values that aren’t needed.

void setAxisAngleRotation(double x, double y, double z,
double rotDegrees)

Sets the quaternion to a rotational quaternion. The axis of rotation is specified by the directional values
(x, y, z). rotDegrees specifies the number of degrees of counterclockwise rotation around the
given axis.
vsVector rotatePoint(vsVector targetPoint)
Transfers the targetPoint by the rotation in the quaternion. The point is transformed as if it was a column vector being multiplied by a rotation matrix on the right.

vsQuat slerp(vsQuat destination, double parameter)
Spherically interpolates between the quaternion and the destination quaternion. The parameter value is used to specify the desired fractional position between the two quaternions; a parameter of 0.0 will return a quaternion equal to this quaternion, while a parameter of 1.0 returns one equal to destination. Parameter values outside the range of [0.0 – 1.0] are not permitted. Returns the resulting quaternion as a vsQuat.

double &operator[](int index)
Returns a reference to one of the elements of the quaternion.

vsQuat operator+(vsQuat addend)
Equivalent to getSum(addend).

vsQuat operator-(vsQuat subtrahend)
Equivalent to getDifference(subtrahend).

vsQuat operator*(double multiplier)
Equivalent to getScaled(multiplier).

vsQuat operator*(vsQuat operand)
Equivalent to getMultipliedQuat(operand).

void operator+=(vsQuat addend)
Equivalent to add(addend).

void operator-=(vsQuat subtrahend)
Equivalent to subtract(subtrahend).

void operator*=(double multiplier)
Equivalent to scale(multiplier).

void operator*=(vsQuat operand)
Equivalent to multiplyQuat(operand).

int operator==(vsQuat operand)
Equivalent to isEqual(operand).

vsQuat operator*(double multiplier, vsQuat operand)
Non-member function. Returns a quaternion with values equal to the elements of operand multiplied by multiplier.