Vector Math Library Overview

PlayStation®Vita Programmer Tool Runtime Library Release 3.50

Table of Contents

1 Library Overview

Features

The Vector Math library mainly provides functionality used in 3D graphics for 3D and 4D vector operations, matrix operations, and quaternion operations. The library also provides 2D vector and matrix functionality that can be useful in user interface or human interface device programming. APIs for the C++ programming language are provided, with three formats according to the data layout:

- The scalar format, which is useful for porting to the Reference System
- The AoS (Array of Structures) SIMD format, which can easily and quickly be adapted to handle different situations
- The SoA (Structure of Arrays) SIMD format, which allows for maximum throughput

The SIMD versions of the Vector Math library are available for PlayStation®Vita, PlayStation®3 (PPU), PlayStation®3 (SPU), and Windows (SSE2) platforms.

Files

The following files are required to use the Vector Math library:

Table 1 Required Files

File Name with Relative Path	Description
`target/include_common/vectormath.h`	Header file for API (C++ language)

The following files are provided for backwards compatibility:

Table 2 Compatibility Files

File Name with Relative Path	Description
`target/include_common/vectormath/scalar_cpp/vectormath_aos.h`	Header file for API (C++ language scalar format)
`target/include_common/vectormath/cpp/vectormath_aos.h`	Header file for API (C++ language for AoS format)
`target/include_common/vectormath/cpp/vectormath_soa.h`	Header file for API (C++ language for SoA format)

Note: All functions are inlined in this library; therefore, there are no `.a` files.

The user should not directly include `mat_aos.h`, `quat_aos.h`, `vec_aos.h`, `vecidx_aos.h`, `mat_soa.h`, `quat_soa.h`, `vec_soa.h`, `boolInVec.h`, `floatInVec.h` or any of the header files in the `internal` subdirectory; they are included from the `vectormath_aos.h` and `vectormath_soa.h` header files.

2 How to Use the Library

Vector Representation Convention

In the Vector Math library, vectors are handled as column vectors (vectors in which the elements are arranged vertically). This is the same convention used in many computer graphics textbooks. According to this convention, the basis vectors and translation vector of the transformation matrix are matrix columns, and the multiplication sequence is “(matrix)(vector)”.

The row-vector convention is also often used in computer graphics. In the row-vector convention, all matrix and vector objects are transposed as compared with the column-vector convention, and thus an opposite order is used for multiplying matrices and matrices with vectors. Although there are various opinions regarding which arrangement is the best, the operations are fundamentally identical and neither is superior in terms of performance.

Scalar and SIMD Versions

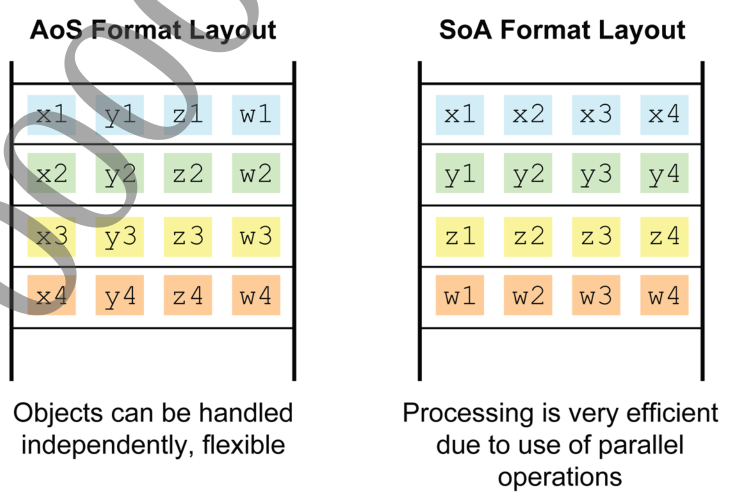
The Vector Math library contains both a Scalar and a SIMD version. On the PlayStation®Vita platform the SIMD version utilizes the Neon Advanced SIMD unit of the ARM Cortex-A9 whereas the Scalar version produces instructions to be processed by the VFP unit. Either the Scalar or SIMD version of the library can be the more optimal of the two; it depends on the characteristics of the calculation being performed. However, the user should avoid switching between the SIMD and Scalar versions of the library on a fine-grained basis.

AoS and SoA Formats

The Vector Math library uses two types of data layout in the SIMD implementation: the AoS format and the SoA format. The CPU can use these two types of data layout.

In the AoS (Array of Structures) format data layout, each object element is stored contiguously in memory. In the SoA (Structure of Arrays) format data layout, four objects are packed together, and the groups made up of the four elements are stored contiguously in memory.

Figure 1 AoS Format and SoA Format Data Layouts



The AoS format data layout's parallel operations are inefficient because:

- The data to be processed may include padding. For example, in a 3D vector addition, only the three words x, y, z (32 bits x 3) are valid, but a quadword (128 bits) operation that includes w (padding) is performed.
- For some operations, the data must be reorganized. For example, the dot product of a 4D vector must be shifted to align the x, y, z, and w fields.

However, in this case "efficiency" refers to throughput. AoS format processing sometimes has less latency than the corresponding SoA format processing; this format is also more familiar to many programmers. For applications handling data that cannot be grouped and processed uniformly, the AoS format is the better choice.

The SoA format layout is effective when processing uniform sets of data with the same code. For example, if you have an array of vertex coordinates and each vertex requires the same processing, all four vertices can be grouped into one SoA object and then processed in parallel using four-way SIMD instructions. This can maximize calculation throughput because each arithmetic instruction generally performs four valid calculations.

However, one of the restrictions of processing SoA objects is that any conditional branch that applies to each object must be transformed to a conditional move. In other words, it is possible that when simultaneously processing four objects, some may meet the conditions and some may not. Therefore, the only option is to calculate the results of both and then to select the results for each object by a mask. The Vector Math library API includes selection functions for this purpose.

The API also includes functions to place four copies of the same AoS object into an SoA object, or to convert between an SoA object and four AoS objects.

Alignment and Padding

To make the data types convenient for use with SIMD instructions, all data types defined in the Vector Math library have the alignment shown in Table 3.

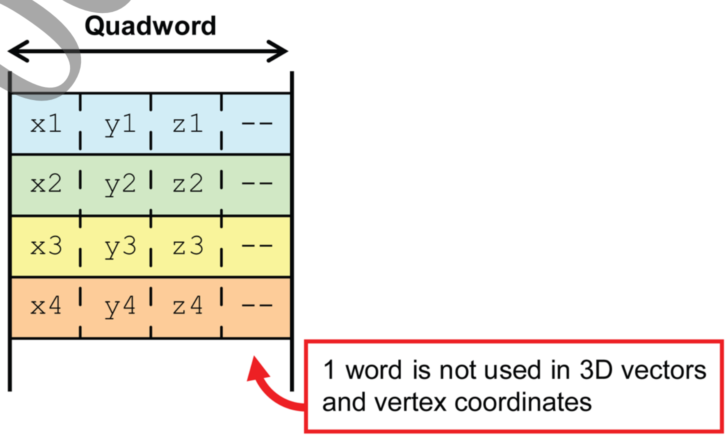
Table 3 Data Type Alignment for the Vector Math Library (by platform)

Platform	Alignment
PlayStation®Vita	dualword (64 bit)
All other platforms	quadword (128 bit)*

* The `Vector2` data type is an exception; it has singleword (32 bit) alignment.

In the AoS format data layout, because 3D vectors (`Vector3` type) and 3D Coordinates (`Point3` type) use only three words for their x, y, and z elements, one word of free space is created in each quadword. The fourth word is padding and is never explicitly set to a value or used.

Figure 2 Padding in the AoS Format



In other words, when handling this data, one third more memory and bandwidth are used than is necessary.

To avoid this problem, you could store and transfer 3D data in a more compact format and convert to and from AoS format data in quadword alignment as necessary. The APIs provide functions for this purpose; for example, the API includes `loadXYZArray()` and `storeXYZArray()` functions.

Similarly, note that the AoS format `Transform3` type and the `Matrix3` type also contain one word of padding for each column.

Floating-Point Behavior

The floating-point behavior of Vector Math library functions follows the behavior of processor intrinsics for SIMD arithmetic and standard-library functions.

SIMD Internal Representation

The SIMD version utilizes cross-platform representations of doubleword (64 bits) and quadword (128 bits) vector types for 8, 16, 32 and 64-bit signed/unsigned integers and 32 and 64-bit floating points. The definitions of these types should be considered opaque and may be altered at any time to best suit performance and compiler requirements.

Vectormath provides a suite of cross-platform wrapper functions for the underlying SIMD intrinsics, which are available with the “`sce_vectormath_*`” prefix. You can use the SIMD intrinsics if you require a particular feature; however, the functions are considered semi-internal and, compared to the Vectormath C++ classes, are more likely to be added to, changed or removed over time as required.

Namespace Structure

The Vector Math library’s API is defined by the `sce::Vectormath` namespace. This namespace contains two additional namespaces: `Simd` and `Scalar`. The `Simd` namespace contains two further namespaces, `Aos` and `Soa`, which implement the different data layouts as described in AoS and SoA Formats above. The following skeleton code illustrates the overall layout of the namespaces:

```
namespace sce
{
    namespace Vectormath
    {
        namespace Simd
        {
            namespace Aos
            {
                class Vector3;
            }

            namespace Soa
            {
                class Vector3;
            }

        }

        namespace Scalar
        {
            namespace Aos
            {
                class Vector3;
            }
```

```
                    }
                }
            }
```

Classes

The classes available within the Vector Math library are listed in Table 4.

Table 4 Available C++ Classes

Class	Description
`Vector2`	2D vector.
`Vector3`	3D vector.
`Vector4`	4D vector.
`Point3`	3D point. This 3D point has different properties from a 3D vector: – The difference between two 3D points is a vector. – Two 3D points cannot be added. – A 3D point cannot be scalar-multiplied.
`Quat`	Quaternion. When methods or functions require a unit-length quaternion, the user must clearly provide a normalized quaternion.
`Matrix2`	2x2 matrix.
`Matrix3`	3x3 matrix.
`Matrix4`	4x4 matrix.
`Transform3`	3D transformation. This is a 3x4 matrix representing a 3D affine transformation, consisting of a 3x3 matrix and 3D translation. When multiplied with a "`Vector3`", it applies the 3x3 matrix; when multiplied with a "`Point3`", it applies both the 3x3 matrix and translation.

Constructors and Type Conversion

Every class has a constructor with a single `float` argument, and this `float` is written to every element of the class. For example, `Vector3(5)` results in a 3D vector equal to `(5,5,5)`.

There are also alternate constructors for `Vector2`, `Vector3`, `Vector4`, `Point3`, and `Quat` with enough `float` arguments to set every element of the class. For example, `Vector3(1,2,3)` results in `(1,2,3)`. However, `Matrix2`, `Matrix3`, `Matrix4`, and `Transform3` do not have such constructors.

As shown in the following tables, various constructors are provided to convert between the specified types.

Table 5 Type Conversion to Vector3

Constructor	Description
`Vector3(Point3)`	Type converts from `Point3` to `Vector3`.

Table 6 Type Conversion to Vector4

Constructor	Description
`Vector4(Vector3)`	Type converts from `Vector3` to `Vector4`. Copies x, y, z elements and sets the w element to 0.
`Vector4(Point3)`	Type converts from `Point3` to `Vector4`. Copies x, y, z elements and sets the w element to 1.
`Vector4(Vector3,float)`	Type converts from the `Vector3` and `float` class to `Vector4`. Copies `Vector3` x, y, z elements and the w element from `float`.

The Vector4->setXYZ(Vector3) method can be used when type converting from Vector3 to Vector4. Matrix4->getTranslation() or Transform3->getTranslation() can be used to get the translation component of Matrix4 or Transform3 as Vector3.

Table 7 Type Conversion to Point3

Constructor	Description
Point3(Vector3)	Type converts from Vector3 to Point3.

Table 8 Type Conversion to Quat

Constructor	Description
Quat(Vector3,float)	Type converts from Vector3 and float class to Quat. Copies Vector3 x, y, z elements and the w element of float.
Quat(Vector4)	Type converts from Vector4 to Quat.
Quat(Matrix3)	Converts a 3x3 rotation matrix to unit quaternion. To get a valid result, Matrix3 must be a rotation matrix.

The Quat->setXYZ(Vector3) method can be used to type convert from Vector3 to Quat.

The static helper functions Quat::euler(Quat,RotationOrder) and Quat::rotation(Vector3, RotationOrder) can be used to convert a Quat to and from a Vector3 type containing Euler angles.

Table 9 Type Conversion to Matrix3

Constructor	Description
Matrix3(Quat)	Converts from a unit quaternion to a 3x3 rotation matrix. To get a valid result, Quat must be a unit-length quaternion.

Matrix4->getUpper3x3() or Transform3->getUpper3x3() can be used to get the Matrix4 or Transform3 upper left 3x3 matrix as Matrix3.

Table 10 Type Conversion to Matrix4

Constructor	Description
Matrix4(Transform3)	Type converts from Transform3 to Matrix4. Copies the top 3x4 elements and sets the bottom row to (0,0,0,1).
Matrix4(Matrix3,Vector3)	Converts the affine transform represented by a 3x3 matrix and translation to a matrix. Sets the bottom row to (0,0,0,1).
Matrix4(Quat,Vector3)	Converts the affine transform represented by unit quaternion and translation to a matrix. Sets the bottom row to (0,0,0,1). The Quat argument must be normalized.

Matrix4->setUpper3x3() can be used to write Matrix3 to the upper left 3x3 matrix of Matrix4. Also, Matrix4->setTranslation() can be used to write Vector3 to the translation component. In either case, the bottom row value does not change.

Table 11 Type Conversion to Transform3

Constructor	Description
Transform3(Matrix3,Vector3)	Converts from a 3x3 matrix and translation class to Transform3.
Transform3(Quat,Vector3)	Converts from a unit quaternion and translation class to Transform3. The Quat argument must be normalized.

Transform3->setUpper3x3() can be used to write Matrix3 to the upper left 3x3 matrix of Transform3. Also, Transform3->setTranslation() can be used to write Vector3 to the translation component.

Operators

The operators "*", "/", "+", and "-" are overloaded for performing vector, matrix, and quaternion operations.

The dot product and cross product operations are defined as dot() and cross() functions. The "*" operator is not overloaded for either operation.

As shown in Table 12, multiplication operators for different classes of objects are provided:

Table 12 Multiplication Operators

Operator	Description
Transform3 * Vector3	Multiplies a Vector3 by the 3x3 matrix component of a Transform3.
Transform3 * Point3	Multiplies a Point3 by the 3x3 matrix component and then adds the translation component of a Transform3.
Matrix4 * Vector3	Multiplies a Matrix4 by a Vector3 treated as if it were a Vector4 with a value of (x,y,z,0).
Matrix4 * Point3	Multiplies a Matrix4 by a Point3 treated as if it were a Vector4 with a value of (x,y,z,1).
Matrix4 * Transform3	Multiplies a Matrix4 by a Transform3 treated as if it were a Matrix4 with a bottom row of (0,0,0,1).

When using these operators, Vector3, Point3, and Transform3 require alternate homogeneous-coordinate meanings to be mathematically valid. In other words, the Vector3 class can be considered a 4D vector with a w element of 0; the Point3 class can be considered a 4D vector with a w element of 1; and the Transform3 class can be considered a 4x4 matrix with a bottom row of (0,0,0,1).

Constants

Some constant values can be accessed by using constructors and static methods. For example:

```
V3 = Vector3::zero();        // zero vector = (0,0,0)
v3 = Vector3(0.0f);          // zero vector = (0,0,0)
v3 = Vector3::xAxis();       // unit vector = (1,0,0)
v4 = Vector4::wAxis();       // unit vector = (0,0,0,1)
m3 = Matrix3::identity();    // 3x3 matrix identity
quat = Quat::identity();     // identity quaternion = (0,0,0,1)
```

Coordinate Transformations

The following static methods are provided to generate an object to perform a coordinate transformation:

- A rotation (for all matrices and for Quat)
- A scale transformation (for all matrices)
- A translation (Matrix4 and Transform3 only)

For example:

```
Quat q;
Vector3 s, t;
Transform3 m;

m = Transform3::rotation(q);    // rotation from unit quaternion
m = Transform3::scale(s);       // scale matrix from 3 scale components
m = Transform3::translation(t); // translation from vector
```

Matrix transformations can be performed by multiplying; however, it is faster to set rotation and translation components, and then use the `appendScale()` and `prependScale()` functions, which scale columns and rows, respectively. For example:

```
m = Transform3 (q,t);
m = appendScale(m,s);
```

"InVec" Types

The API provides two "`InVec`" data types:

- `floatInVec`
- `boolInVec`

To resolve SIMD/VFP mixing stalls on the Neon unit, `float` return values have been replaced with a return value of type "`floatInVec`". There are two distinct pipelines in the Neon unit:

- Advanced SIMD – performs all the vector processing
- VFP – performs scalar floating point calculations

These two pipelines cannot be active at the same time; therefore there is a cost involved in mixing instructions for the two pipelines. When using the SIMD variant of the Vector Math library there is a danger of mixing VFP and SIMD instructions – specifically when using input and outputs to functions of the library that are notionally scalar. `floatInVec` solves this problem by keeping scalar calculations on the Advanced SIMD unit.

On PlayStation®Vita, `floatInVec` is a class with dualword size that implements standard floating-point operators using Neon Advanced SIMD operations; the class is quadword size on other platforms. To make this type invisible to the user, `floatInVec` can be implicitly cast to a `float` by the compiler, so that a result of this type can be used as a `float` value in most cases.

Additionally, methods and functions that have `float` arguments have been overloaded to also accept `floatInVec` arguments. If you use a `floatInVec` result as an input to another Vector Math function, there is a much smaller chance of a VFP/Advanced SIMD pipeline stall than if a temporary `float` value is used. To avoid accidental casts to `float`, you can use:

```
#define _SCE_VECTORMATH_NO_SCALAR_CAST
```

The `floatInVec` interface includes comparison operators that return a `boolInVec`. The `boolInVec` class represents Boolean operations and comparison results, and has similar SIMD properties to `floatInVec`. The `boolInVec` data type implicitly casts to `bool` and can be used in conditional statements. Using a `boolInVec` result as an input to a "select" function avoids a move or branching (see Table 13).

Table 13 Select Functions

Function	Description
`select(boolInVec b0, boolInVec b1, boolInVec sel)`	If `sel` is true returns `b1`; else returns `b0`.
`select(floatInVec f0, floatInVec f1, boolInVec sel)`	If `sel` is true returns `f1`; else returns `f0`.
`select(Vector2 v0, Vector2 v1, boolInVec sel)`	If `sel` is true returns `v1`; else returns `v0`.
`select(Vector3 v0, Vector3 v1, boolInVec sel)`	If `sel` is true returns `v1`; else returns `v0`.
`select(Vector4 v0, Vector4 v1, boolInVec sel)`	If `sel` is true returns `v1`; else returns `v0`.
`select(Point3 p0, Point3 p1, boolInVec sel)`	If `sel` is true returns `p1`; else returns `p0`.
`select(Quat q0, Quat q1, boolInVec sel)`	If `sel` is true returns `q1`; else returns `q0`.
`select(Matrix2 m0, Matrix2 m1, boolInVec sel)`	If `sel` is true returns `m1`; else returns `m0`.
`select(Matrix3 m0, Matrix3 m1, boolInVec sel)`	If `sel` is true returns `m1`; else returns `m0`.
`select(Matrix4 m0, Matrix4 m1, boolInVec sel)`	If `sel` is true returns `m1`; else returns `m0`.
`select(Transform3 t0, Transform3 t1, boolInVec sel)`	If `sel` is true returns `t1`; else returns `t0`.