Vector Math Library
Overview

Table of Contents

Library Overview

Features

The Vector Math library mainly provides functions used in 3-D graphics for 3-D and 4-D vector operations, matrix operations, and quaternion operations. APIs for both the C and C++ programming languages are provided, along with three formats according to the data layout:

- 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 scalar format, which is useful for porting and testing

All three formats provide implementations for the PPU and SPU.

Files

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

Table 1 Required Files for PPU

File Name with Relative Path	Description
vectormath/c/vectormath_aos.h	Header file for pass-by-reference API (C language for PPU AoS format)
vectormath/c/vectormath_aos_v.h	Header file for pass-by-value API (C language for PPU AoS format)
vectormath/c/vectormath_soa.h	Header file for pass-by-reference API (C language for PPU SoA format)
vectormath/c/vectormath_soa_v.h	Header file for pass-by-value API (C language for PPU SoA format)
vectormath/cpp/vectormath_aos.h	Header file for API (C++ language for PPU AoS format)
vectormath/cpp/vectormath_soa.h	Header file for API (C++ language for PPU SoA format)

Table 2 Required Files for SPU

File Name with Relative Path	Description
vectormath/c/vectormath_aos.h	Header file for pass-by-reference API (C language for SPU AoS format)
vectormath/c/vectormath_aos_v.h	Header file for pass-by-value API (C language for SPU AoS format)
vectormath/c/vectormath_soa.h	Header file for pass-by-reference API (C language for SPU SoA format)
vectormath/c/vectormath_soa_v.h	Header file for pass-by-value API (C language for SPU SoA format)
vectormath/cpp/vectormath_aos.h	Header file for API (C++ language for SPU AoS format)
vectormath/cpp/vectormath_soa.h	Header file for API (C++ language for SPU SoA format)

Table 3 Required Files for Scalar

File Name with Relative Path	Description
common/vectormath/scalar/c/vectormath_aos.h	Header file for pass-by-reference API (C language scalar format)
common/vectormath/scalar/c/vectormath_aos_v.h	Header file for pass-by-value API (C language scalar format)
common/vectormath/scalar/cpp/vectormath_aos.h	Header file for API (C++ language scalar format)

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

In addition to the above, each directory contains the vec_aos.h, quat_aos.h, mat_aos.h, vec_soa.h, quat_soa.h, mat_soa.h, vec_aos_v.h, quat_aos_v.h, mat_aos_v.h, vec_soa_v.h, quat_soa_v.h, and mat_soa_v.h header files, but the user should not directly include these; they are included from the vectormath_aos.h, vectormath_aos_v.h, vectormath_soa.h, and vectormath_soa_v.h header files.

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)".

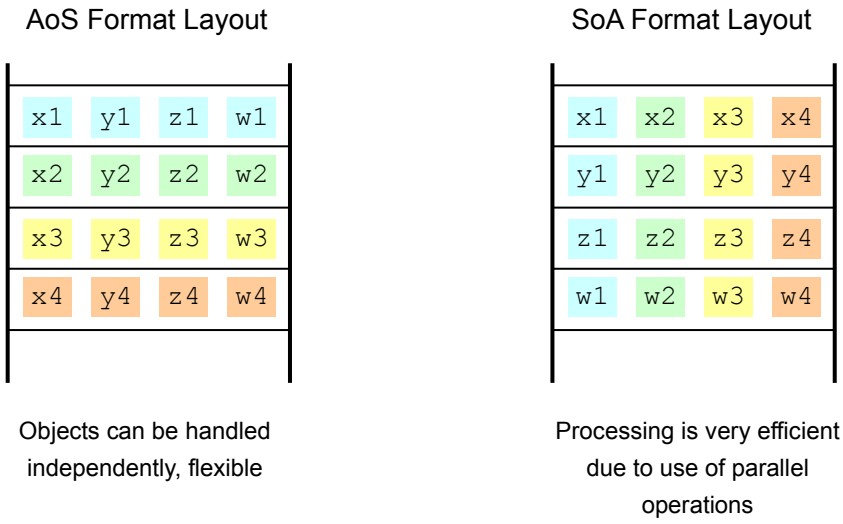
The row-vector convention is also often used in computer graphics. In the row-vector convention, all matrix and vector objects are transposed as compared with the column-vector convention, and thus an opposite order is used for multiplying matrices and matrices with vectors. Although there are various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

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. Both the PPU and the SPU 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 3-D 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 4-D vector must be shifted to align the x, y, z, and w fields.

However, in this case "efficiency" refers to throughput. AoS format processing frequently 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 using uniform sets of data to process with the same code. For example, if you have an array of vertex coordinates and if each vertex requires the same processing, all four vertices could be grouped into one SoA object and could then be processed in parallel using four-way SIMD instructions. This can maximize calculation throughput because each arithmetic instruction generally performs four valid calculations.

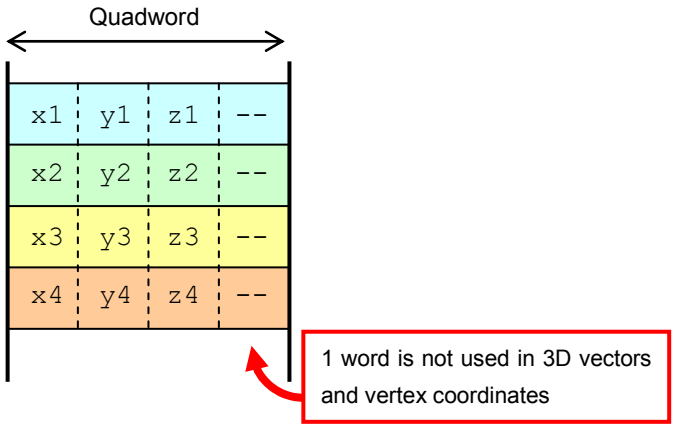
However, one of the restrictions of processing SoA objects is that any conditional branch that applies to each object must be transformed to a conditional move. In other words, it is possible that when simultaneously processing four objects, some may meet the conditions and some may not. Therefore, the only option is to calculate the results of both and then to select the results for each object by a mask. The API includes selection functions for this purpose; for example, `vmathV3Select()` and various selection functions within the `Aos` and `Soa` namespaces.

The Vector Math library API 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

In order to make them convenient for use with quadword load/store instructions and SIMD instructions, all data types defined in the Vector Math library have quadword (128 bits) alignment for both the PPU and the SPU. As a result, in the AoS format data layout, one word of free space is created in each quadword because 3-D vectors (`Vector3` type) and 3-D Coordinates (`Point3` type) use only three words for their x, y, and z elements. 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 3-D 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 C++ 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.

The scalar format implementation of AoS vectors uses the GCC alignment attribute to perform the same quadword alignment and padding. (When not compiled with GCC, `Vector3` type and `Point3` type padding does take place, but alignment is not guaranteed for any type.) Consistent size and alignment is helpful for porting prototype code.

Floating-Point Behavior

For the PPU and the SPU, the floating-point behavior of Vector Math library functions follows the behavior of processor intrinsics for SIMD arithmetic and standard-library functions. Notable examples are the C++ `divPerElem()` and `recipPerElem()` functions that behave exactly like the SIMD Math library functions `divf4()` and `recipf4()`.

C++ API

The Vector Math library's API for C++ is defined by the `Vectormath` namespace. This namespace contains two additional namespaces, `Aos` and `Soa`, which implement different data layouts as described previously.

Classes

The C++ classes that are available within the Vector Math library are listed below:

Table 4 Available C++ Classes

Class	Description
`Vector3`	3-D vector.
`Vector4`	4-D vector.
`Point3`	3-D point. This 3-D point has different properties from a 3-D vector: - The difference between two 3-D points is a vector. - Two 3-D points cannot be added. - A 3-D point cannot be scalar-multiplied.
`Quat`	Quaternion. When methods or functions require a unit-length quaternion, the user must clearly provide a normalized quaternion.
`Matrix3`	3x3 matrix.
`Matrix4`	4x4 matrix.
`Transform3`	3-D transformation. This is a 3x4 matrix representing a 3-D affine transformation, consisting of a 3x3 matrix and 3-D 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 3-D vector equal to `(5,5,5)`.

There are also alternate constructors for `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, `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`.

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 the following table, 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 both the 3x3 matrix component and 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 4-D vector with a w element of 0; the `Point3` class can be considered a 4-D 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(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 to then use the `appendScale()` and `prependScale()` functions, which scale columns and rows, respectively. For example:

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

PPU "InVec" Types

The C++ API for the PPU provides two "InVec" data types:

- `floatInVec`
- `boolInVec`

On the PPU, copying between VMX registers and floating-point registers can result in a stall. To minimize such problems, `float` return values in the C++ API have been replaced with a return value of type "floatInVec". `floatInVec` is a class with quadword size that implements standard floating-point operators using VMX operations. 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, PPU C++ 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, the data can remain in a VMX register. However, assigning the result to a temporary `float` value will not provide any benefit. To avoid accidental casts to `float`, you can use:

```
#define _VECTORMATH_NO_SCALAR_CAST
```

The `floatInVec` interface includes comparison operators that return a "boolInVec". Using this result as an input to a "select" function avoids a move. The `boolInVec` data type implicitly casts to `bool` and can be used in conditional statements.

Note: The "InVec" types are not provided in the C API. C++ allows you to take advantage of these types but to avoid explicit use of them, and to therefore maintain more portable code. These classes have also been implemented for the SPU, although the Vector Math API for the SPU does not currently use them.

C APIs

The Vector Math library APIs for C provide functionality similar to the API for C++. As described later in this section, a C API is provided for passing by reference and a second API is provided for passing by value.

Support for AoS and SoA Formats

As with C++, both the AoS format and the SoA format are supported, but in C there is no namespace. Therefore, as shown in the following table, the "Soa" prefix is included in SoA format type names and function names to differentiate them.

Table 13 Naming Conventions

Data Type	AoS Format Type Name	SoA Format Type Name
3-D Vector	`VmathVector3`	`VmathSoaVector3`
4-D Vector	`VmathVector4`	`VmathSoaVector4`

Data Type	AoS Format Type Name	SoA Format Type Name
3-D Point	VmathPoint3	VmathSoaPoint3
Quaternion	VmathQuat	VmathSoaQuat
3x3 Matrix	VmathMatrix3	VmathSoaMatrix3
4x4 Matrix	VmathMatrix4	VmathSoaMatrix4
3-D Conversion Matrix	VmathTransform3	VmathSoaTransform3

Using Functions That Have Identical Usage But Different Arguments

C does not have the concept of function overload. Therefore, when it is necessary to define functions that have identical usage but different arguments, include an argument type abbreviation in the function name.

For example:

```
void vmathV4MakeFromV3( VmathVector4 *result, const VmathVector3 *vec );
void vmathV4MakeFromP3( VmathVector4 *result, const VmathPoint3 *pnt );
void vmathV4MakeFromQ( VmathVector4 *result, const VmathQuat *quat );
```

The type name abbreviations are as follows:

Table 14 Type Name Abbreviations

Type Name	Abbreviation
VmathVector3	V3
VmathVector4	V4
VmathPoint3	P3
VmathQuat	Q
VmathMatrix3	M3
VmathMatrix4	M4
VmathTransform3	T3
VmathSoaVector3	SoaV3
VmathSoaVector4	SoaV4
VmathSoaPoint3	SoaP3
VmathSoaQuat	SoaQ
VmathSoaMatrix3	SoaM3
VmathSoaMatrix4	SoaM4
VmathSoaTransform3	SoaT3

APIs for Passing by Reference or Passing by Value

Two APIs are provided in C, a “pass-by-value” API and a “pass-by-reference” API. The former API typically passes struct arguments by value and returns a struct result by value. The latter API passes all struct arguments by pointer, and passes a pointer to a struct result as the first argument. The pass-by-value API is distinguished by function names that contain the suffix “_V”.

For example:

```
VmathVector3 vmathV3Add_V(VmathVector3 vec0, VmathVector3 vec1);
void vmathV3Add(VmathVector3 *result,
                const VmathVector3 *vec0,
                const VmathVector3 *vec1);
```

Fundamental data types are passed by value in either API.

For example:

```
float vmathV3Dot_V(VmathVector3 vec0, VmathVector3 vec1);
float vmathV3Dot(const VmathVector3 *vec0, const VmathVector3 *vec1);
```

Due to its convenience, some users may prefer the pass-by-value API. However the pass-by-reference API may improve performance. Even when the pass-by-value functions are inlined, unnecessary copying of structs may occur.

The pass-by-reference API also contains functions that can be used to copy Vector Math struct types. Code that uses these functions may run more quickly than code that uses the assignment operator.

For example:

```
void vmathSoaV3Copy(VmathSoaVector3 *result, const VmathSoaVector3 *vec);
```