Vectormath Cheat Sheet v2.7

Conventions:

`VecNV` means `ScalarV`, `Vec2V`, `Vec3V`, `Vec4V`
Abbrev. for vars: *sc* = `ScalarV`, *v2* = `Vec2V`, *bv* = `BoolV`, *vb* = `VecBoolV`, *qu* = `QuatV`, *vn* = any `VecNV`, *M33* = `Mat33V`, etc.
`Underline` may be replaced by other names – `X` could be `X`,`Y`,`Z`,`W`.
`[Bracket]` is a function variant. `Add[Fast]` → `Add` and `AddFast`
`void` return values are omitted

Functions ending in 'f' use the float pipeline (`a.Getf()`)
"Safe" means division for zero is checked and dealt with
"Fast" means lower precision (`NormalizeFast()`)

Headers:

`#include "vectormath/vecNv.h" (matMNv.h)` (classes)
`#include "vectormath/classfreefuncsv.h"` (functions)
`#include "vectormath/vectortypes.h"` (low level types, constants)
-or-
`#include "vectormath/classes.h"` grabs the whole library

Types:

Vector types: `ScalarV`, `Vec2V`, `Vec3V`, `Vec4V`
Matrix types: `Mat33V`, `Mat34V`, `Mat44V`
Additional types:
- `VecBoolV` (each component is 0xffffffff or 0x0)
- `BoolV` – a vector containing a single (splatted) bool value
- `QuatV` – quaternion.

Parameter types: `VecNV_In`, `VecNV_InOut`
Return type: `VecNV_Out`

Constructing:

Predefined constant value:
`ScalarV(V_ZERO);`
`Vec3V(V_X_AXIS_WONE);`
(See page 2 for some constant enums)

Floating point constant:
`Vec2VConstant<FLOAT_TO_INT(1.0f), FLOAT_TO_INT(2.0f)>();`
`ScalarVConstant<0x3f800000>();`

Floating point:
`Vec3V(flx, fly, flz);`
`ScalarVFromF32(fl);`
`Vec3VFromF32(fl);` (splats fl, also `…FromU32`, `…FromS32`)

Other vectors:
`Vec3V(scxyz);` `Vec2V(scx, scy);`
`Vec4V(v2xy, v2zw);` `Vec3V(scx, v2yz);`

Component Access:

`vn.GetX()` – gets a ScalarV from one component
`vn.GetXf()` – gets a float from one component
`vn.Get<Vec::X, Vec::Y, ...>()` - gets a VecNV with a permuted subset of components

Other conversions:
`v4.GetXYZ()` → v3; `v4.GetXY()`→v2; `v3.GetXY()`→v2
`vn.GetElemf(i)` → float;

Setting components:
'vn' is in a register (uses vector pipeline)
- `vn.SetX(sc)` – sets X from a scalar
- `vn.SetX(fl)` – if fl is in memory (not a reg.)

'vn' is in memory (less common, uses float pipeline):
- `vn.SetXInMemory(sc)` – sets X from a scalar
- `vn.SetXf(fl)` – sets to a float (from memory or reg.)

Other setters:
`vn.Set<Vec::X, Vec::Y, ...>()` - permuted set. Arguments specify the components to *write* to. E.g.: `v3.Set<Y, Z>(v2)` results in (v3x, v2x, v2y)

`vn.ZeroComponents()` – clears a vector

Comparison Functions:

`IsFiniteAll`		*sc*	*v2*	*v3*	*v4*	*qu*		*m34*	
`IsFiniteStable`			*v2*	*v3*	*v4*				
`SameSign`		*sc*	*v2*	*v3*	*v4*				
`SameSignAll`		*sc*	*v2*	*v3*	*v4*				
`IsFinite`		*sc*	*v2*	*v3*	*v4*	*qu*			
`IsNotNan`		*sc*	*v2*	*v3*	*v4*	*qu*			
`IsEven`		*sc*	*v2*	*v3*	*v4*				
`IsOdd`		*sc*	*v2*	*v3*	*v4*				
`IsZeroAll`		*sc*	*v2*	*v3*	*v4*	*qu*			
`IsZeroNone`			*v2*	*v3*	*v4*	*qu*			
`IsBetweenNegAndPosBounds`		*sc*	*v2*	*v3*	*v4*				
`IsEqual*`		*sc*	*v2*	*v3*	*v4*	*qu*			
`IsEqualAll*`		*sc*	*v2*	*v3*	*v4*	*qu*	*m33*	*m34*	*m44*
`IsEqualNone`		*sc*	*v2*	*v3*	*v4*	*qu*	*m33*	*m34*	*m44*
`IsEqualInt`	*vb*	*sc*	*v2*	*v3*	*v4*	*qu*			
`IsEqualIntAll`	*vb*	*sc*	*v2*	*v3*	*v4*	*qu*	*m33*	*m34*	*m44*
`IsEqualIntNone`	*vb*	*sc*	*v2*	*v3*	*v4*	*qu*	*m33*	*m34*	*m44*
`IsClose`		*sc*	*v2*	*v3*	*v4*	*qu*			
`IsCloseAll`		*sc*	*v2*	*v3*	*v4*	*qu*	*m33*	*m34*	*m44*
`IsCloseNone`			*v2*	*v3*	*v4*	*qu*	*m33*	*m34*	*m44*

* In addition to `Equal`, can also check `GreaterThan`, `GreaterThanOrEqual`, `LessThan`, `LessThanOrEqual`

Algebra:

All 'Safe' functions take an additional "safeVal" that is returned in place of a divide by 0.

vn `RoundToNearestInt`*(vn)* (round)
vn `RoundToNearestIntZero`*(vn)* (trunc)
vn `RoundToNearestIntNegInf`*(vn)* (floor)
vn `RoundToNearestIntPosInf`*(vn)* (ceil)

vn `Negate`*(vn)*
vn `Invert[Fast][Safe]`*(vn, [vn safeVal])*
vn `Abs`*(vn)*

vn `Add`*(vn, vn)* *vn* `Subtract`*(vn, vn)*
vn `Average`*(vn, vn)*

vn `Scale`*(vn, vn)* *vn* `Scale`*(vn, sc)* *vn* `Scale`*(sc, vn)*

vn `InvScale[Fast][Safe]`*(vn a, vn b, [vn safeVal])* a / b
vn `InvScale[Fast][Safe]`*(vn a, sc b, [vn safeVal])*

vn `AddScaled`*(vn a, vn b, vn s)* a + (b · s)
vn `AddScaled`*(vn a, vn b, sc s)*

vn `SubtractScaled`*(vn a, vn b, vn s)* a – (b · s)
vn `SubtractScaled`*(vn a, vn b, sc s)*

vn `Pow`*(vn a, vn b)* a^b *vn* `Expt`*(vn a)* 2^a
vn `Log2`*(vn);* *vn* `Log10`*(vn)*

vn `Max`*(vn, vn)* *vn* `Min`*(vn, vn)*
sc `MaxElement`*(vn)* *sc* `MinElement`*(vn)*

vn `Sqrt[Fast][Safe]`*(vn, [vn safeVal])*
vn `InvSqrt[Fast][Safe]`*(vn [vn safeVal])*

Linear Algebra:

sc `Dot`*(vn, vn)* for v2, v3, v4
v3 `Cross`*(v3, v3)*
v4 `Cross3`*(v4, v4)*
sc `Cross`*(v2 a, v2 b)* ax · by – ay · bx
v3 `AddCrossed`*(v3 toAdd, v3 a, v3 b)* toAdd + (a × b)
v4 `AddCrossed3`*(v4 toAdd, v4 a, v4 b)*
v3 `SubtractCrossed`*(v3 toSub, v3 a, v3 b)* toSub – (a × b)
v4 `SubtractCrossed3`*(v4 toSub, v4 a, v4 b)*
vn `Reflect`*(vn in, vn wallNormal)* for v2, v3
`OuterProduct`*(Mnn& o, vn a, vn b)* $o = a \times b^T$

Operators:

`+, -, *, /`: as expected for VecNV. All are per-component.
`==, !=, <, <=, >, >=`: For VecNV, per comp., return VecBoolV
`|, &, !, ^`: perform bitwise operations on a VecBoolV

Integer Operations:

int `vn.GetXi`*()*, `vn.SetXi`*(int)*
`vn.GetElemi`*(int)*, `vn.SetElemi`*(int, int)*
vn `AddInt`*(vn, vn)* *vn* `SubtractInt`*(vn,vn)*

Fixed point conversion:
vn `IntToFloatRaw`*<int x>(vn a)* a / 2^x (a is an int value)
vn `FloatToIntRaw`*<int x>(vn a)* a · 2^x (a is a float)

Distance and Magnitude:

Mag, Normalize and Dist functions do not work on ScalarV

sc `Mag[Fast]`*(vn)*
sc `MagSquared`*(vn)*
sc `InvMag[Fast][Safe]`*(vn, [vn safeVal])*
sc `InvMagSquared[Fast][Safe]`*(vn, [vn_safeVal])*

vn `Normalize[Fast][Safe]`*(vn, [vn_safeVal], [vn_safeMagSqThresh])*

sc `Dist[Fast]`*(vn, vn)*
sc `InvDist[Fast][Safe]`*(vn, vn, [vn safeVal])*
sc `DistSquared`*(vn, vn)*
sc `InvDistSquared[Fast][Safe]`*(vn, vn, [vn safeVal])*

sc `MagXY[Fast]`*(v3)* Projects onto XY, then gets mag.
sc `DistXY[Fast]`*(v3, v3)*

Ranges and Interpolation:

vn `Clamp`*(vn val, vn lowbound, vn highbound)*
vn `Saturate`*(vn)* clamps to [0,1]
v3 `ClampMag`*(v3 in, sc min, sc max)*

vn `Lerp`*(sc t, vn a, vn b)*
vn `Lerp`*(vn t, vn a, vn b)* Maps t = 0 → a, t = 1 → b

vn `Range[Safe][Fast]`*(vn t, vn lower, vn upper, [vn safeVal])*
Inverse of Lerp. Maps t = lower → 0, t = upper → 1
vn `RangeClamp[Fast]`*(vn t, vn lower, vn upper)*
Output clamped to [0,1]

vn `Ramp[Fast]`*(vn x, vn inA, vn inB, vn outA, vn outB)*
Maps x = inA → outA, x = inB → outB

Trig:

`SinAndCos[Fast]`*(vn& outSins, vn& outCos, vn x)*
vn `Sin[Fast]`*(vn)* *vn* `Arcsin[Fast]`*(vn)*
vn `Cos[Fast]`*(vn)* *vn* `Arccos[Fast]`*(vn)*
vn `Tan[Fast]`*(vn)* *vn* `Arctan[Fast]`*(vn)*
vn `Arctan2[Fast]`*(vn y, vn x)*
vn `CanonicalizeAngle`*(vn)* converts angle to [-π,π]

vn `SlowInOut`*(vn)*; *vn* `SlowIn`*(vn)*; *vn* `SlowOut`*(nv)*
Over the range [0,1] returns a [0,1] value on an 'ease curve' (portion of a sin curve) with the specified characteristics.

vn `BellInOut`*(vn)*
Over the range [0,1] returns a value that starts at 0, rises to 1, returns to 0 smoothly

Angular Operations:

v2 `Rotate`*(v2 in, v2 radians)*
v3 `RotateAboutXAxis`*(v3 in, sc θ)*

sc `Angle[NormInput]`*(v3 a, v3 b)* angle between a and b.
v2 or v3
sc `AngleX[NormInput]`*(v3 a, v3 b)* angle if ax = bx = 0
NormInput assumes |a| = |b| = 1

`MakeOrthonormals`*(v3 in, v3& oA, v3& oB)*
in, oA, oB will be mutually orthogonal

Permutes and Logical Operations:

vn `MergeXY`*(vn a, vn b)* → (ax, bx, ay, by)
v4 `MergeZW`*(v4 a, v4 b)* → (az, bz, aw, bw)
vn `GetFromTwo<X1,Y2,Z1,W2>`*(vn a, vn b)* → (ax, by, az, bw)
vn `MergeXY[Short|Byte]`*(vn, vn)*
v4 `MergeZW[Short|Byte]`*(v4, v4)*

vn `SelectFT`*(vb choice, vn valueIfFalse, vn valueIfTrue)*

bool `IsTrue`*(bv)*, *bool* `IsFalse`*(bv)* For single BoolV
bool `IsTrueAll`*(vb)*, *bool* `IsFalseAll`*(vb)* For VecBoolV

vn `ShiftRightAlgebraic`*(vn in, vn sh)* (inx ≫ shx, iny ≫ shy,...)
vn `And`*(vn, vn)*;
vn `Or`*(vn, vn)*;
vn `Xor`*(vn, vn)*;
vn `Andc`*(vn a, vn b)* a & !b
vn `InvertBits`*(vn)*

Constants:

(single values here get splatted across all channels)
Examples: `Vec3V(V_TWO_PI)`, `Vec4V(V_ONE_WZERO)`

`V_ZERO` ...`V_FIFTEEN`	0.0f ... 15.0f
`V_NEGONE` ...`V_NEGSIXTEEN`	-1.0f ... -16.0f
`V_INT_1` ...`V_INT_15`	0x0001 ... 0x000f
`V_ZERO_WONE`	(0.0, 0.0, 0.0, 1.0)
`V_X_AXIS_WONE`	(1.0, 0.0, 0.0, 1.0)
`V_Y_AXIS_WONE`	(0.0, 1.0, 0.0, 1.0)
`V_Z_AXIS_WONE, V_UP_AXIS_WONE`	(0.0, 0.0, 1.0, 1.0)
`V_ONE_WZERO`	(1.0, 1.0, 1.0, 0.0)
`V_X_AXIS_WZERO`	(1.0, 0.0, 0.0, 0.0)
`V_Y_AXIS_WZERO`	(0.0, 1.0, 0.0, 0.0)
`V_Z_AXIS_WZERO, V_UP_AXIS_WZERO`	(0.0, 0.0, 1.0, 0.0)
`V_FLT_MAX, V_NEG_FLT_MAX`	$\pm \sim 3.403\times10^{38}$
`V_FLT_MIN`	$\sim 1.175\times10^{-38}$
`V_FLT_EPSILON`	$\sim 1.19\times10^{-7}$
`V_FLT_SMALL_6` ... `V_FLT_SMALL_1`	10^{-6} ... 10^{-1}
`V_FLT_LARGE_8, V_FLT_SMALL_12`	10^{8}, 10^{-12}
`V_ONE_PLUS_EPSILON`	$1.0 + \sim 1.19\times10^{-7}$
`V_ONE_MINUS_FLT_SMALL_3`	$1.0 - 10^{-3}$
`V_QUARTER, V_THIRD, V_HALF, V_NEGHALF`	1/4, 1/3, ±1/2
`V_NAN, V_INF, V_NEGINF`	NaN, ±∞
`V_LOG2_TO_LOG10`	log(10)/log(2)
`V_INT_7FFFFFFF`	0x7FFFFFFF
`V_INT_80000000`	0x80000000
`V_ONE_OVER_1024`	1/1024
`V_PI, V_NEG_PI, V_TWO_PI`	±π, 2π
`V_ONE_OVER_PI, V_TWO_OVER_PI`	1/π, 2/π
`V_PI_OVER_TWO, V_NEG_PI_OVER_TWO`	±π/2
`V_TO_DEGREES, V_TO_RADIANS`	180/π, π/180

Masks: 0xff... for any named component, 0x0 otherwise. E.g.:

`V_MASKY`	0x0000	0xffffffff	0x0000	0x0000
`V_MASKXZ`	0xffffffff	0x0000	0xffffffff	0x0000
`V_MASKXYW`	0xffffffff	0xffffffff	0x0000	0xffffffff
`V_MASKXYZW`	0xffffffff	0xffffffff	0xffffffff	0xffffffff

Bools: synonyms for masks. The vectors above would be:
`V_F_T_F_F`, `V_T_F_T_F`, `V_T_T_F_T`, `V_T_T_T_T`
`V_FALSE`, `V_TRUE` Same as `V_F_F_F_F`, `V_T_T_T_T`

Vectormath Cheat Sheet v2.7

Matrix Conventions:

Vectors here represent column vectors. [a,b,c] is a matrix made of 3 column vectors

Mnn is a square matrix, (3×3 or 4×4). *Mnp* is any matrix.

· is component-wise multiplication, × is matrix multiplication

A `[3x3]` fn modifies only the upper left 3x3, normally takes Mat34Vs. Other output values are copied directly (unmodified) from an input matrix

`[4x3]` and `[3x4]` are similar, normally take Mat44Vs

Constructing Matrices:

Predefined constant value:

`Mat33V(V_ZERO)` `Mat44V(V_IDENTITY)`

Floating point:

`Mat33V(ax, ay, az, bx, by, bz, cx, cy, cz)`

sets in col. major order: [a,b,c]

`Mat33V(V_ROW_MAJOR, ax,bx,cx, ay,by,cy, az,bz,cz)`

Other vectors:

`Mat44V(v4a)` [v4a, v4a, v4a, v4a]

`Mat44V(v4a, v4b, v4c, v4d)` [v4a, v4b, v4c, v4d]

Standard transformation matrices:

Translation 't' is optional for M34, M44. Default is (0,0,0,1).

`MatNPVFromTranslation`*(Mnp& o, vn t)* M34, M44

`MatNPVFromScale`*(Mnp& o, vn s, [vn t])*

`MatNPVFromScale`*(Mnp& o, sc sx, sc sy, sc sz, [vn t])*

`MatNPVFromAxisAngle`*(Mnp& o, v3 normal, sc theta, [vn t])*

`MatNPVFromXAxisAngle`*(Mnp& o, sc theta, [vn t])*

`MatNPVFromEulersXYZ`*(Mnp& o, v3 eulers, [vn t])*

`MatNPVFromQuatV`*(Mnp& o, qu q, [vn t])*

`LookAt`*(M34& o, v3 from, v3 to, [v3 up])*

`LookDown`*(M34& o, v3 dir, v3 up)*

Matrix Component Access:

For the functions below, N,P ∈ (0,1,2,3)

vn `Mnp.GetColN[Ref|ConstRef]`*()*

float `Mnp.GetMNPf`*()* (`m.GetM20f()`) row N, col P

float `Mnp.GetElemf`*(int row, int col)*

`Mnp.SetColN`*(vn)*

`Mnp.SetCols`*(vn, vn, vn...)*

`Mnp.SetElemf`*(int row, int col, float val)*

`Mnp.SetMNP`*(fl)* (Mnp in registers) (`m.SetM31()`) row N, col P

`Mnp.SetMNPf`*(fl)* (Mnp in memory) (`m.SetM12()`) row N, col P

Matrix Algebra:

`Add[3x3|4x3]`*(Mnp& o, Mnp A, Mnp B)* o = A + B

`Subtract`*(Mnp& o, Mnp A, Mnp B)* o = A – B

`Translate`*(M34& o, M34 M, v3 t)* o = M + [0,0,0,t]

`AddScaled[3x3|4x3]`*(Mnp& o, Mnp A, Mnp B, Mnp S)* o = A + (B · S)

`AddScaled[3x3|4x3]`*(Mnp& o, Mnp A, Mnp B, vn s)* o = A + (B · [s,s,s])

`Scale`*(Mnp& o, vn v, Mnp M)* (scales each col. by v.

`Scale`*(Mnp& o, Mnp M, vn v)* i.e.: o = m · [v,v,v])

`InvScale[Fast][Safe]`*(Mnp& o, Mnp M, vn v, [vn safeVal])*

`Transpose[3x3]`*(Mnn& o, Mnn A)* $o = A^T$

`Abs`*(M33& io)* per comp., in place

`Abs[3x3]`*(Mnp& o, Mnp A)* per comp., o = |A|

sc `Determinant`*(Mnn)*

sc `Determinant3x3`*(Mnp)* M34, M44

vb `Mnp.IsOrthonormal[3x3]V`*(sc toleranceSq, [sc angToler])*

bool `Mnp.IsOrthonormal[3x3]`*(sc toleranceSq, [sc angToler])*

`ReOrthonormalize[3x3]`*(Mnp& o, Mnp A)*

`Mat34VRotateLocalX`*(M34& ioA, sc theta)*

`Mat34VRotateGlobalX`*(M34& ioA, sc theta)*

v3 `MatNPVToEulersXYZ`*(M33)* M33, M34

Matrix Multiplication

`Multiply`*(Mnn& o, Mnn A, Mnn B)* o = A × B

vn `Multiply`*(Mnp A, vp b)* A × b

vp `Multiply`*(vn a, Mnp B)* $a^T \times B$

Transform is a special version of multiply, for Mat34Vs that represent a 3x3 plus translation

`Transform`*(M34& ioA, M34 B)* ioA = ioA × B

`Transform[3x3]`*(M34& o, M34 A, M34 B)* o = A × B

v3 `Transform[3x3]`*(M34 A, v3 b)* A × b

Invert and UnTransform have Full and Ortho versions. Ortho is faster if you know your matrix is orthonormal.

`InvertFull`*(Mnn& o, Mnn A)* $o = A^{-1}$

`InvertOrtho`*(M33& o, M33 A)* $o = A^{-1} = A^T$, fast

`Invert3x3(Full|Ortho)`*(M34& o, M34 A)*

`Invert3x4(Full|Ortho)`*(M44& o, M44 A)*

`InvertTransform(Full|Ortho)`*(M34& o, M34 A)*

if Transform(A, v) = v', then Transform(o, v') = v

`UnTransform[3x3](Full|Ortho)`*(Mnp& o, Mnp A, Mnp B)*

$o = A^{-1} \times B$

vn `UnTransform[3x3](Full|Ortho)`*(Mnp A, vn b)* $A^{-1} \times b$

Quaternion conversions:

qu `QuatVFromAxisAngle`*(v3 norm, sc theta)*

qu `QuatVFromXAxisAngle`*(sc theta)*

`QuatVToAxisAngle`*(v3& oAxis, sc& oTheta, qu)*

qu `QuatVFromEulersXYZ[Fast]`*(v3 eulers)*

qu `QuatVFromEulers[Fast]`*(v3 eulers, EULER_XYZ)*

v3 `QuatVToEulersXYZ[Fast]`*(qu q)*

v3 `QuatVToEulers[Fast]`*(qu q, EULER_XYZ)*

qu `QuatVFromMat33V[Safe]`*(M33, [qu safeVal])*

qu `QuatVFromMat34V`*(M34)*

qu `QuatVFromVectors`*(v3 from, v3 to, [v3 axis])* Finds the quat. that rotates point 'from' to point 'to' (opt. rotating about axis)

Quaternion operations:

qu `Conjugate`*(qu)* (-x,-y,-z,w)

qu `Invert[Fast][Safe]`*(qu, [qu safeVal])*

qu `Normalize[Fast][Safe]`*(qu, [qu safeVal], [qu magSqThresh])*

qu `InvertNormInput`*(qu)* Fast if q is already normalized

qu `Multiply`*(qu a, qu b)* True quaternion mult.: a ∗ b

qu `QuatVScaleAngle`*(qu q, sc s)* preserves axis, scales rotation

e.g. QuatVScaleAngle(q, 3) = q∗q∗q

qu `Scale`*(qu, sc)* *qu* `Scale`*(qu, v4)*

sc `Dot`*(qu, sc)*

sc `Mag`*(qu)* *sc* `MagSquared`*(qu)*

v3 `Transform`*(qu q, v3 v)* $q * v * q^{-1}$ (rotates v by q)

v3 `UnTransformFull`*(qu q, v3 v)* $q^{-1} * v * q$ (unrotates v)

qu `PrepareSlerp`*(qu ref, qu q)*

Call before slerping – returns q or –q, based on shortest distance to ref.

qu `Slerp[Near]`*(sc t, qu a, qu b)*

qu `Slerp[Near]`*(v4 t, qu a, qu b)*

"Spherical lerp" – lerps 0 → a, 1→ b at constant velocity

qu `Nlerp`*(sc t, qu a, qu b)*

qu `Nlerp`*(v4 t, qu a, qu b)*

Normalized lerp - faster than Slerp but non-constant velocity

v3 `GetUnitDirection[Fast][Safe]`*(qu, [v3 safeVal])*

returns axis of rotation

sc `GetAngle`*(qu)*

sc `QuatVTwistAngle`*(qu q, v3 axis)* computes the amount of twist around one axis

Transforms:

A TransformV represents a position and rotation (i.e. an orthonormal vector basis)

qu `xf.GetRotation`*()* `xf.SetRotation`*(qu)*
v3 `xf.GetPosition`*()* `xf.SetPosition`*(v3)*

xf `Lerp`*(sc t, xf A, xf B)* Maps t=0 → A, t=1 → B

xf `SelectFT`*(bv test, xf A, xf B)* test ? B : A

`TransformVFromMat34V`*(xf& o, M34 a)*
`Mat34VFromTransformV`*(M34& o, xf A)*

`Transform`*(xf& o, xf A, xf B)* $o = A \times B$
`UnTransform`*(xf& o, xf A, xf B)* $o = A^{-1} \times B$
`InvertTransform`*(xf& o, xf A)*
if Transform(A, v) = v', then Transform(o, v') = v

qu `Transform`*(xf, qu)* *v3* `Transform[3x3]`*(xf, v3)*
qu `UnTransform`*(xf, qu)* *v3* `UnTransform[3x3]`*(xf, v3)*

Additional Vectormath types:

Intrinsic types:
`Vector_4` (float pipeline),
`Vector_4V` (vector pipeline)

Class types – vector pipeline:
`Vec2V`, `Vec3V`, `Vec4V`, `ScalarV`, `VecBoolV`, `BoolV`, `QuatV`, `Mat33V`, `Mat34V`, `Mat44V`
`TransformV` – a position and quaternion rotation

Class types – float pipeline:
`Vec2f`, `Vec3f`, `Vec4f`, `Quatf`
`LoadAsScalar`*(VecNf&, const VecNV&)* loads VecNV as VecNf
`StoreAsVec`*(VecNV&, const VecNf)* stores VecNf to a VecNV
`LoadAsVec`, `StoreAsScalar` (as above)

Structure of Array types:
(each type is 4x instances of the underlying types, so an SoA::Vec3V lets you operate on 4 Vec3Vs in parallel)
`SoA_ScalarV`, `SoA_Vec2V`, `SoA_Vec3V`, `SoA_Vec4V`, `SoA_VecBool1V`, `SoA_VecBool2V`, `SoA_VecBool3V`, `SoA_VecBool4V`, `SoA_QuatV`, `SoA_Mat33V`, `SoA_Mat34V`, `SoA_Mat44V`

Parameter passing:

Use the `VecNV_In`, `VecNV_Out`, and `VecNV_InOut` types.

For free or inlined functions, return vectors by value.

```
  Vec3V_Out Fn(Vec3V_In a, Vec3V_In b) {…}
```

For non-inlined member functions, or for matrices, return via a reference (a `VecNV_InOut` param)

```
 void Class::Fn(Vec3V_InOut out, Vec3V_In a) {out =…;}
 void Fn(Mat44V_InOut out, Vec3V_In a, Vec3V_In b) {…}
```

For simple accessors, use `VecNV_ConstRef` or `VecNV_Ref` – because the caller may not want to load the vector into vector registers.

```
 Vec3V_Ref GetPosition() {return m_Position;}
 Mat34V_ConstRef GetTransform() {return m_Transform;}
```

See the performance guide on the wiki for more info. Where best PC performance is necessary, pass using intrinsics and convert:

```
  Vec::V3Return128 Fn(Vec::V3Param128 a, Vec::V3Param128 b)
  {
      Vec3V va(a); Vec3V vb(b); // convert to class type
       …;
      return result.GetIntrin128();
  }
```

Legacy Conversions:

These macros are useful for converting between the old vector library and the new.

All of these functions come in RC_ and RCC_ variants (but we only list RC_ below), and they specify the type you want to convert *to*.

`RC_*` reinterprets the old type as a reference to new type. Use for lvalues.

```
RC_VEC3V(m_OldVector) = Add(v1, v2);
```

`RCC_*` reinterprets the old type as a const ref. to the new type. Use this for passing lvalues, and wherever else possible.

```
Vec3V v = Add(v1, RCC_VEC3V(m_OldVector));
```

Converting *to* vectormath:

Function:	Converts from:
`RC_SCALARV`*	`Vector3`, `Vector4`
`RC_VEC3V`	`Vector3`, `Vector4`
`RC_VEC4V`	`Vector3`, `Vector4`
`RC_QUATV`	`Quaternion`
`RC_MAT33V`**	`Matrix33`
`RC_MAT34V`**	`Matrix34`
`RC_MAT44V`**	`Matrix44`
`RC_VEC3F`	`Vector3`

Converting *to* old vector library

`RC_VECTOR3`	`Vec3f`, `Vec3V`, `Vec4V`, `ScalarV`
`RC_VECTOR4`	`Vec3V`, `Vec4V`, `ScalarV`
`RC_QUATERNION`	`QuatV`
`RC_MATRIX33`**	`Mat33V`
`RC_MATRIX34`**	`Mat34V`
`RC_MATRIX44`**	`Mat44V`

* All 4 ScalarV components need to match. Be careful when converting from Vector3 where 'w' may have been unset – convert to Vec3V and then SplatX() if necessary.

** Watch out for matrices! Multiplication order is different. See the refactoring guide on the wiki for info.

Other conversions are available. See `legacyconvert.h` for more conversion macros.