QVM

Quaternion / Vector / Matrix Library written in C++03 | Emil Dotchevski

Abstract

QVM is a generic library for working with Quaternions, Vectors and Matrices of static size. Features:

- Emphasis on 2, 3 and 4-dimensional operations needed in graphics, video games and simulation applications.
- Free function templates operate on any compatible user-defined Quaternion, Vector or Matrix type.
- Enables Quaternion, Vector and Matrix types from different libraries to be safely mixed in the same expression.
- Type-safe mapping between compatible lvalue types with no temporary objects; f.ex. transpose remaps the access to the elements, rather than transforming the matrix.
- Requires only C++03.
- Zero dependencies.

Tutorial

Quaternions, Vectors, Matrices

Out of the box QVM defines generic yet simple `quat`, `vec` and `mat` types. For example, the following snippet creates a quaternion object that rotates around the X axis:

```
quat<float> rx = rotx_quat(3.14159f);
```

Similarly, a matrix that translates by a given vector can be created as follows:

```
vec<float,3> v = {0,0,7};
mat<float,4,4> tr = translation_mat(v);
```

The usual quaternion, vector and matrix operations work on these QVM types, however the operations are decoupled from any specific type: they work on any suitable type that has been registered by specializing the `quat_traits`, `vec_traits` and `mat_traits` templates.

For example, a user-defined 3D vector type `float3` can be introduced to QVM as follows:

```
struct float3 { float a[3]; };

namespace boost { namespace qvm {

  template <>
  struct vec_traits<float3> {

    static int const dim=3;
    typedef float scalar_type;

    template <int I>
    static inline scalar_type & write_element( float3 & v ) {
      return v.a[I];
    }

    template <int I>
    static inline scalar_type read_element( float3 const & v ) {
      return v.a[I];
    }

    static inline scalar_type & write_element_idx( int i, float3 & v ) {
      return v.a[i];
    } //optional

    static inline scalar_type read_element_idx( int i, float3 const & v ) {
      return v.a[i];
    } //optional

  };

} }
```

Equivalently, using the `vec_traits_defaults` template the above can be shortened to:

```
namespace boost { namespace qvm {

  template <>
  struct vec_traits<float3>: vec_traits_defaults<float3,float,3> {

    template <int I>
    static inline scalar_type & write_element( float3 & v ) {
      return v.a[I];
    }

    static inline scalar_type & write_element_idx( int i, float3 & v ) {
      return v.a[i];
    } //optional

  };

} }
```

After a similar specialization of the `mat_traits` template for a user-defined 3x3 matrix type `float33`, the full range of vector and matrix operations defined by QVM headers becomes available automatically:

```
float3 v;
X(v) = 0;
Y(v) = 0;
Z(v) = 7;
float vmag = mag(v);
float33 m = rotx_mat<3>(3.14159f);
float3 vrot = m * v;
```

User-defined quaternion types are similarly introduced to QVM by specializing the `quat_traits` template.

C Arrays

In `boost/qvm/quat_traits_array.hpp`, `vec_traits_array.hpp` and `mat_traits_array.hpp`, QVM defines appropriate `quat_traits`, `vec_traits` and `mat_traits` specializations that allow QVM functions to operate directly on plain old C arrays:

```
float v[3] = {0,0,7};
float3 vrot = rotx_mat<3>(3.14159f) * v;
```

Naturally, operator overloads cannot kick in if all elements of an expression are of built-in types. The following is still illegal:

```
float v[3] = {0,0,7};
v *= 42;
```

The `vref` and `mref` function templates can be used to work around this issue:

```
float v[3] = {0,0,7};
vref(v) *= 42;
```

View proxies

QVM defines various function templates that provide static mapping between (possibly user-defined) quaternion, vector and matrix types. The example below multiplies column 1 (QVM indexes are always zero-based) of the matrix `m` by a scalar:

```
void multiply_column1( float33 & m, float scalar ) {
  col<1>(m) *= scalar;
}
```

The expression `col<1>(m)` is an lvalue of an unspecified 3D vector type that refers to column 1 of `m`. Note however that this does not create any temporary objects; instead `operator*=` above works directly with a reference to `m`.

Here is another example, multiplying a transposed view of a matrix by a vector of some user-defined type `float3`:

```
float3 v = {0,0,7};
float3 vrot = transposed(rotx_mat<3>(3.14159f)) * v;
```

In general, the various view proxy functions return references of unspecified, non-copyable types that refer to the original object. They can be assigned from or converted to any compatible vector or matrix type.

Swizzling

QVM allows accessing vector elements by swizzling, exposing vector views of different dimensions, and/or views with reordered elements. The example below rotates `v` around the X axis, and stores the resulting vector back in `v` but with the X and Y elements swapped:

```
float3 v = {0,0,7};
YXZ(v) = rotx_mat<3>(3.14159f) * v;
```

A special case of swizzling provides next-dimension-view of a vector object, adding either 0 or 1 as its last component. Assuming `float3` is a 3D vector type, and `float4` is a 4D vector type, the following statements are valid:

```
float3 v = {0,0,7};
float4 point = XYZ1(v); //{0,0,7,1}
float4 vector = XYZ0(v); //{0,0,7,0}
```

It is also valid for swizzling to address vector elements more than once:

```
float3 v = {0,0,7};
float4 v1 = ZZZZ(v); //{7,7,7,7}
```

QVM defines all permutations of `X`, `Y`, `Z`, `W` for 1D, 2D, 3D and 4D swizzling, plus each dimension defines variants with 0 or 1 used at any position (if 0 or 1 appear at the first position, the swizzling function name begins with underscore, e.g. `_1XY`).

The swizzling syntax can also be used to bind scalars as vectors. For example:

```
float3 v = _00X(42.0f); //{0,0,42}
```

SFINAE/enable_if

SFINAE stands for Substitution Failure Is Not An Error. This refers to a situation in C++ where an invalid substitution of template parameters (including when those parameters are deduced implicitly as a result of an unqualified call) is not in itself an error.

In absence of concepts support, SFINAE can be used to disable function template overloads that would otherwise present a signature that is too generic. More formally, this is supported by the Boost `enable_if` library.

For example, QVM defines `operator*` overload which works with any user-defined matrix and vector types. The naive approach would be to declare this overload as follows:

```
template <class Matrix,class Vector>
Vector operator*( Matrix const & m, Vector const & v );
```

Even if the function definition might contain code that would compile only for `Matrix` and `Vector` types, because the function declaration itself is valid, it will participate in overload rezolutions when multiplying objects of any two types whatsoever. This typically renders overload resolutions ambiguous and the compiler (correctly) issues an error.

Using `enable_if`, QVM declares such overloads in a way that preserves their generic signature but

only participate in overload resolutions if the passed parameters make sense depending on the semantics of the operation being defined:

```
template <class A,class B>
typename enable_if_c<
  is_mat<A>::value && is_vec<B>::value && mat_traits<A>::cols==vec_traits<B>::dim,
//Condition
  B>::type //Return type
operator*( A const & a, B const & b );
```

For brevity, function declarations throughout this documentation specify the condition which controls whether they are enabled or not without specifying exactly what `enable_if` construct is used to achieve this effect.

Interoperability

An important design goal of QVM is that it works seamlessly with 3rd-party quaternion, vector and matrix types and libraries. Even when such libraries overload the same C++ operators as QVM, it is safe to bring the entire `boost::qvm` namespace in scope by specifying:

```
using namespace boost::qvm;
```

The above using directive does not introduce ambiguities with function and operator overloads defined by a 3rd-party library because:

- Most `boost::qvm` function overloads and all operator overloads use SFINAE/`enable_if`, which makes them "disappear" unless an expression uses types that have the appropriate QVM-specific type traits defined;
- Whenever such overloads are compatible with a given expression, their signature is extremely generic, which means that any other (user-defined) compatible overload will be a better match in any overload resolution.

Bringing the entire boost::qvm namespace in scope may introduce ambiguities when accessing types (as opposed to functions) defined by 3rd-party libraries. In that case, you can safely bring namespace `boost::qvm::sfinae` in scope instead, which contains only function and operator overloads that use SFINAE/`enable_if`.

Specifying return types for binary operations

Bringing the `boost::qvm` namespace in scope lets you mix vector and matrix types that come from different APIs into a common, type-safe framework. In this case however, it should be considered what types should be returned by binary operations that return an object by value. For example, if you multiply a 3x3 matrix `m1` of type `user_matrix1` by a 3x3 matrix `m2` of type `user_matrix2`, what type should that operation return?

The answer is that by default, QVM returns some kind of compatible matrix type, so it is always safe to write:

```
auto & m = m1 * m2; // auto requires C++11
```

However, the type deduced by default converts implicitly to any compatible matrix type, so the following is also valid, at the cost of a temporary:

```
user_matrix1 m = m1 * m2;
```

While the temporary object can be optimized away by many compilers, it can be avoided altogether by specializing the `deduce_mat2` template. For example, to specify that multiplying a `user_matrix1` by a `user_matrix2` should always produce a `user_matrix1` object, you could write:

```
namespace boost { namespace qvm {

  template <>
  struct deduce_mat2<user_matrix1,user_matrix2,3,3> {
    typedef user_matrix1 type;
  };

  template <>
  struct deduce_mat2<user_matrix2,user_matrix1,3,3> {
    typedef user_matrix1 type;
  };

} }
```

Be mindful of potential ODR violation when using `deduce_quat2`, `deduce_vec2` and `deduce_mat2` in independent libraries. For example, this could happen if `lib1` defines `deduce_vec2<lib1::vec,lib2::vec>::type` as `lib1::vec` and in the same program `lib2` defines `deduce_vec2<lib1::vec,lib2::vec>::type` as `lib2::vec`.

It is best to keep such specializations out of `lib1` and `lib2`. Of course, it is always safe for `lib1` and `lib2` to use `convert_to` to convert between the `lib1::vec` and `lib2::vec` types as needed.

Specifying return types for unary operations

Perhaps surprisingly, unary operations that return an object by value have a similar, though simpler issue. That's because the argument they're called with may not be copyable, as in:

```
float m[3][3];
auto & inv = inverse(m);
```

Above, the object returned by `inverse` and captured by `inv` can not be of type `float[3][3]`, because that type isn't copyable. By default, QVM "just works", returning an object of suitable matrix type that is copyable. This deduction process can be controlled, by specializing the `deduce_mat` template.

Converting between different quaternion, vector and matrix types

Any time you need to create a matrix of a particular C++ type from any other compatible matrix type, you can use the `convert_to` function:

```
user_matrix2 m=convert_to<user_matrix2>(m1 * m2);
```

Reference

Header Files

QVM is split into multiple headers to allow different compilation units to `#include` only the components they need. Each function in this document specifies the exact header that must be `#included` in order to use it.

The tables below list commonly used components and the headers they're found in. Header names containing a number define functions that only work with objects of that dimension; e.g. `vec_operations2.hpp` contains only functions for working with 2D vectors.

The header `boost/qvm/all.hpp` is provided for convenience. It includes all other QVM headers.

In addition, Boost QVM is available in single-header format for maximum portability. See Distribution.

Table 1. Quaternion header files

Quaternion traits	`#include <boost/qvm/quat_traits.hpp>`
	`#include <boost/qvm/quat_traits_array.hpp>`
	`#include <boost/qvm/deduce_quat.hpp>`

Quaternion element access	`#include <boost/qvm/quat_access.hpp>`
Quaternion operations	`#include <boost/qvm/quat_operations.hpp>`
`quat` class template	`#include <boost/qvm/quat.hpp>`

Table 2. Vector header files

Vector traits	`#include <boost/qvm/vec_traits.hpp>`
	`#include <boost/qvm/vec_traits_array.hpp>`
	`#include <boost/qvm/deduce_vec.hpp>`

Vector element access	`#include <boost/qvm/vec_access.hpp>`

Vector swizzling	`#include <boost/qvm/swizzle.hpp>`
	`#include <boost/qvm/swizzle2.hpp>`
	`#include <boost/qvm/swizzle3.hpp>`
	`#include <boost/qvm/swizzle4.hpp>`

Vector operations	`#include <boost/qvm/vec_operations.hpp>`
	`#include <boost/qvm/vec_operations2.hpp>`
	`#include <boost/qvm/vec_operations3.hpp>`
	`#include <boost/qvm/vec_operations4.hpp>`

Quaternion-vector operations	`#include <boost/qvm/quat_vec_operations.hpp>`
Vector-matrix operations	`#include <boost/qvm/vec_mat_operations.hpp>`
Vector-matrix view proxies	`#include <boost/qvm/map_vec_mat.hpp>`
`vec` class template	`#include <boost/qvm/vec.hpp>`

Table 3. Matrix header files

Matrix traits	`#include <boost/qvm/mat_traits.hpp>`
	`#include <boost/qvm/mat_traits_array.hpp>`
	`#include <boost/qvm/deduce_mat.hpp>`

Matrix element access	`#include <boost/qvm/mat_access.hpp>`

Matrix operations	`#include <boost/qvm/mat_operations.hpp>`
	`#include <boost/qvm/mat_operations2.hpp>`
	`#include <boost/qvm/mat_operations3.hpp>`
	`#include <boost/qvm/mat_operations4.hpp>`

Matrix-matrix view proxies	`#include <boost/qvm/map_mat_mat.hpp>`
Matrix-vector view proxies	`#include <boost/qvm/map_mat_vec.hpp>`
`mat` class template	`#include <boost/qvm/mat.hpp>`

Type Traits System

QVM is designed to work with user-defined quaternion, vector and matrix types, as well as user-defined scalar types. This section formally defines the way such types can be integrated.

Scalar Requirements

A valid scalar type `S` must have accessible destructor, default constructor, copy constructor and assignment operator, and must support the following operations:

```
S operator*( S, S );
S operator/( S, S );
S operator+( S, S );
S operator-( S, S );

S & operator*=( S &, S );
S & operator/=( S &, S );
S & operator+=( S &, S );
S & operator-=( S &, S );

bool operator==( S, S );
bool operator!=( S, S );
```

In addition, the expression `S(0)` should construct a scalar of value zero, and `S(1)` should construct a scalar of value one, or else the `scalar_traits` template must be specialized appropriately.

`is_scalar`

#include <boost/qvm/scalar_traits.hpp>

```
namespace boost { namespace qvm {

  template <class T>
  struct is_scalar {
    static bool const value=false;
  };

  template <> struct is_scalar<char>           { static bool const value=true; };
  template <> struct is_scalar<signed char>    { static bool const value=true; };
  template <> struct is_scalar<unsigned char>  { static bool const value=true; };
  template <> struct is_scalar<signed short>   { static bool const value=true; };
  template <> struct is_scalar<unsigned short> { static bool const value=true; };
  template <> struct is_scalar<signed int>     { static bool const value=true; };
  template <> struct is_scalar<unsigned int>   { static bool const value=true; };
  template <> struct is_scalar<signed long>    { static bool const value=true; };
  template <> struct is_scalar<unsigned long>  { static bool const value=true; };
  template <> struct is_scalar<float>          { static bool const value=true; };
  template <> struct is_scalar<double>         { static bool const value=true; };
  template <> struct is_scalar<long double>    { static bool const value=true; };

} }
```

This template defines a compile-time boolean constant value which can be used to determine whether a type `T` is a valid scalar type. It must be specialized together with the `scalar_traits` template in order to introduce a user-defined scalar type to QVM. Such types must satisfy the scalar requirements.

---

scalar_traits

#include <boost/qvm/scalar_traits.hpp>

```
namespace boost { namespace qvm {

  template <class Scalar>
  struct scalar_traits {

    BOOST_QVM_INLINE_CRITICAL
    static Scalar value( int v ) {
      return Scalar(v);
    }

  };

} }
```

This template may be specialized for user-defined scalar types to define the appropriate conversion

from `int`; this is primarily used whenever QVM needs to deduce a zero or one value.

`deduce_scalar`

#include <boost/qvm/deduce_scalar.hpp>

```
namespace boost { namespace qvm {

  template <class A,class B>
  struct deduce_scalar
  {
    typedef typename /*exact definition unspecified*/ type;
  };

} }
```

Requires:

`A` and `B` satisfy the scalar requirements.

Returns:

If `A` and `B` are the same type, `scalar_traits<A,B>::type` is defined as that type. Otherwise for the following types:

- `signed`/`unsigned char`,
- `signed`/`unsigned short`,
- `signed`/`unsigned int`,
- `signed`/`unsigned long`,
- `float`,
- `double`,

the deduction logic is as follows:

- if either of `A` and `B` is `double`, the result is `double`;
- else, if one of `A` or `B` is an integer type and the other is `float`, the result is `float`;
- else, if one of `A` or `B` is a signed integer and the other type is unsigned integer, the signed type is changed to unsigned, and then the lesser of the two integers is promoted to the other.

For any other types `scalar_traits<A,B>::type` is defined as `void`. It can be specialized for user-defined scalar types.

This template is used by generic binary operations that return a scalar, to deduce the return type based on the (possibly different) scalars of their arguments.

scalar

#include <boost/qvm/scalar_traits.hpp>

```
namespace boost { namespace qvm {

  template <class T>
  struct scalar {
    typedef /*exact definition unspecified*/ type;
  };

} }
```

The expression `quat_traits<T>::scalar_type` evaluates to the scalar type of the quaternion type `T` (if `is_quat<T>::value` is `true`).

The expression `vec_traits<T>::scalar_type` evaluates to the scalar type of the vector type `T` (if `is_vec<T>::value` is `true`).

The expression `mat_traits<T>::scalar_type` evaluates to the scalar type of the matrix type `T` (if `is_mat<T>::value` is `true`).

The expression `scalar<T>::type` is similar, except that it automatically detects whether `T` is a vector or a matrix or a quaternion type.

---

is_quat

#include <boost/qvm/quat_traits.hpp>

```
namespace boost { namespace qvm {

  template <class T>
  struct is_quat {

    static bool const value = false;

  };

} }
```

This type template defines a compile-time boolean constant value which can be used to determine whether a type `T` is a quaternion type. For quaternion types, the `quat_traits` template can be used to access their elements generically, or to obtain their `scalar type`.

---

quat_traits

#include <boost/qvm/quat_traits.hpp>

```
namespace boost { namespace qvm {

  template <class Q>
  struct quat_traits {

    /*main template members unspecified*/

  };

  /*
  User-defined (possibly partial) specializations:

  template <>
  struct quat_traits<Q> {

    typedef <<user-defined>> scalar_type;

    template <int I>
    static inline scalar_type read_element( Quaternion const & q );

    template <int I>
    static inline scalar_type & write_element( Quaternion & q );

  };
  */

} }
```

The `quat_traits` template must be specialized for (user-defined) quaternion types in order to enable quaternion operations defined in QVM headers for objects of those types.

QVM quaternion operations do not require that quaternion types are copyable.

The main `quat_traits` template members are not specified. Valid specializations are required to define the following members:

- `scalar_type`: the expression `quat_traits<Quaternion>::scalar_type` must be a value type which satisfies the `scalar requirements`.

In addition, valid specializations of the `quat_traits` template must define at least one of the following access functions as static members, where `q` is an object of type `Quaternion`, and `I` is compile-time integer constant:

- `read_element`: the expression `quat_traits<Quaternion>::read_element<I>(q)` returns either a copy of or a `const` reference to the `I`-th element of `q`.
- `write_element`: the expression `quat_traits<Quaternion>::write_element<I>(q)` returns mutable reference to the `I`-th element of `q`.

For the quaternion `a + bi + cj + dk`, the elements are assumed to be in the following order: `a`, `b`, `c`, `d`; that is, `I=0/1/2/3` would access `a/b/c/d`.

It is illegal to call any of the above functions unless `is_quat<Quaternion>::value` is true. Even then, quaternion types are allowed to define only a subset of the access functions.

Below is an example of a user-defined quaternion type, and its corresponding specialization of the quat_traits template:

```
#include <boost/qvm/quat_traits.hpp>

struct fquat { float a[4]; };

namespace boost { namespace qvm {

  template <>
  struct quat_traits<fquat> {

    typedef float scalar_type;

    template <int I>
    static inline scalar_type & write_element( fquat & q ) {
      return q.a[I];
    }

    template <int I>
    static inline scalar_type read_element( fquat const & q ) {
      return q.a[I];
    }

  };

} }
```

Equivalently, using the `quat_traits_defaults` template the above can be shortened to:

```
namespace boost { namespace qvm {

  template <>
  struct quat_traits<fquat>: quat_traits_defaults<fquat,float> {

    template <int I>
    static inline scalar_type & write_element( fquat & q ) {
      return q.a[I];
    }

  };

} }
```

quat_traits_defaults

#include <boost/qvm/quat_traits_defaults.hpp>

```
namespace boost { namespace qvm {

  template <class QuatType,class ScalarType>
  struct quat_traits_defaults {

    typedef QuatType quat_type;

    typedef ScalarType scalar_type;

    template <int I>
    static BOOST_QVM_INLINE_CRITICAL
    scalar_type read_element( quat_type const & x ) {
      return quat_traits<quat_type>::template
        write_element<I>(const_cast<quat_type &>(x));
    }

  };

} }
```

The `quat_traits_defaults` template is designed to be used as a public base for user-defined specializations of the `quat_traits` template, to easily define the required members. If it is used, the only member that must be defined by the user in a `quat_traits` specialization is `write_element`; the `quat_traits_defaults` base will define `read_element`, as well as `scalar_type` automatically.

deduce_quat

#include <boost/qvm/deduce_quat.hpp>

```
namespace boost { namespace qvm {

  template <class Q,
    class S=typename quat_traits<Q>::scalar_type>
  struct deduce_quat {
    typedef Q type;
  };

} }
```

Requires:

- `is_quat<Q>::value` is `true`;
- `is_scalar<S>::value` is `true`;
- `is_quat<deduce_quat<Q,S>::type>::value` must be `true`;
- `quat_traits<deduce_quat<Q,S>::type>::scalar_type` must be the same type as `S`;
- `deduce_quat<Q,S>::type` must be copyable.

This template is used by QVM whenever it needs to deduce a copyable quaternion type from the quaternion type `Q`, with a scalar type `S`. Note that `Q` itself may be non-copyable.

The main template definition returns an unspecified quaternion type, except if `S` is the same type as `quat_traits<Q>::scalar_type`, in which case it returns `Q`, which is only suitable if `Q` is copyable. QVM also defines (partial) specializations for the non-copyable quaternion types it produces. Users can define other (partial) specializations for their own types.

A typical use of the `deduce_quat` template is for specifying the preferred quaternion type to be returned by the generic function template overloads in QVM depending on the type of their arguments.

---

deduce_quat2

#include <boost/qvm/deduce_quat.hpp>

```
namespace boost { namespace qvm {

  template <class A,class B,
    class S=typename deduce_scalar<
      typename scalar<A>::type,
      typename scalar<B>::type>::type>
  struct deduce_quat2 {
    typedef /*unspecified*/ type;
  };

} }
```

Requires:

- Both `scalar<A>::type` and `scalar<B>::type` are well defined;
- `is_quat<A>::value || is_quat<B>::value` is `true`;
- `is_scalar<S>::value` is `true`;
- `is_quat<deduce_quat2<A,B,S>::type>::value` must be `true`;
- `quat_traits<deduce_quat2<A,B,S>::type>::scalar_type` must be the same type as `S`;
- `deduce_quat2<A,B,S>::type` must be copyable.

This template is used by QVM whenever it needs to deduce a quaternion type from the types of two user-supplied function parameters, with scalar type `S`. The returned type must have accessible copy constructor (the `A` and `B` types themselves could be non-copyable, and either one of them may not be a quaternion type.)

The main template definition returns an unspecified quaternion type with `scalar_type` `S`, except if `A` and `B` are the same quaternion type `Q`, in which case `Q` is returned, which is only suitable for copyable types. QVM also defines (partial) specializations for the non-copyable quaternion types it produces. Users can define other (partial) specializations for their own types.

A typical use of the `deduce_quat2` template is for specifying the preferred quaternion type to be returned by the generic function template overloads in QVM depending on the type of their arguments.

`is_vec`

#include <boost/qvm/vec_traits.hpp>

```
namespace boost { namespace qvm {

  template <class T>
  struct is_vec {

    static bool const value = false;

  };

 } }
```

This type template defines a compile-time boolean constant value which can be used to determine whether a type `T` is a vector type. For vector types, the `vec_traits` template can be used to access their elements generically, or to obtain their dimension and `scalar type`.

`vec_traits`

#include <boost/qvm/vec_traits.hpp>

```
namespace boost { namespace qvm {

  template <class V>
  struct vec_traits {

    /*main template members unspecified*/

  };

  /*
  User-defined (possibly partial) specializations:

  template <>
  struct vec_traits<V> {

    static int const dim = <<user-defined>>;

    typedef <<user-defined>> scalar_type;

    template <int I>
    static inline scalar_type read_element( Vector const & v );

    template <int I>
    static inline scalar_type & write_element( Vector & v );

    static inline scalar_type read_element_idx( int i, Vector const & v );
    static inline scalar_type & write_element_idx( int i, Vector & v );

  };
  */

} }
```

The `vec_traits` template must be specialized for (user-defined) vector types in order to enable vector and matrix operations defined in QVM headers for objects of those types.

QVM vector operations do not require that vector types are copyable.

The main `vec_traits` template members are not specified. Valid specializations are required to define the following members:

- `dim`: the expression `vec_traits<Vector>::dim` must evaluate to a compile-time integer constant greater than 0 that specifies the vector size.
- `scalar_type`: the expression `vec_traits<Vector>::scalar_type` must be a value type which satisfies the `scalar requirements`.

In addition, valid specializations of the `vec_traits` template may define the following access

functions as static members, where `v` is an object of type `Vector`, `I` is a compile-time integer constant, and `i` is a variable of type `int`:

- `read_element`: the expression `vec_traits<Vector>::read_element<I>(v)` returns either a copy of or a const reference to the `I`-th element of `v`.
- `write_element`: the expression `vec_traits<Vector>::write_element<I>(v)` returns mutable reference to the `I`-th element of `v`.
- `read_element_idx`: the expression `vec_traits<Vector>::read_element_idx(i,v)` returns either a copy of or a `const` reference to the `i`-th element of `v`.
- `write_element_idx`: the expression `vec_traits<Vector>::write_element_idx(i,v)` returns mutable reference to the `i`-th element of `v`.

It is illegal to call any of the above functions unless `is_vec<Vector>::value` is true. Even then, vector types are allowed to define only a subset of the access functions. The general requirements are:

- At least one of `read_element` or `write_element` must be defined;
- If `read_element_idx` is defined, `read_element` must also be defined;
- If `write_element_idx` is defined, `write_element` must also be defined.

Below is an example of a user-defined 3D vector type, and its corresponding specialization of the `vec_traits` template:

```
#include <boost/qvm/vec_traits.hpp>

struct float3 { float a[3]; };

namespace boost { namespace qvm {

  template <>
  struct vec_traits<float3> {

    static int const dim=3;

    typedef float scalar_type;

    template <int I>
    static inline scalar_type & write_element( float3 & v ) {
      return v.a[I];
    }

    template <int I>
    static inline scalar_type read_element( float3 const & v ) {
      return v.a[I];
    }

    static inline scalar_type & write_element_idx( int i, float3 & v ) {
      return v.a[i];
    } //optional

    static inline scalar_type read_element_idx( int i, float3 const & v ) {
      return v.a[i];
    } //optional

  };

} }
```

Equivalently, using the `vec_traits_defaults` template the above can be shortened to:

```
namespace boost { namespace qvm {

  template <>
  struct vec_traits<float3>: vec_traits_defaults<float3,float,3>
  {

    template <int I>
    static inline scalar_type & write_element( float3 & v ) {
      return v.a[I];
    }

    static inline scalar_type & write_element_idx( int i, float3 & v ) {
      return v.a[i];
    } //optional

  };

} }
```

vec_traits_defaults

#include <boost/qvm/vec_traits_defaults.hpp>

```
namespace boost { namespace qvm {

  template <class VecType,class ScalarType,int Dim>
  struct vec_traits_defaults {

    typedef VecType vec_type;
    typedef ScalarType scalar_type;
    static int const dim=Dim;

    template <int I>
    static BOOST_QVM_INLINE_CRITICAL
    scalar_type write_element( vec_type const & x ) {
      return vec_traits<vec_type>::template write_element<I>(const_cast<vec_type &>(
x));
    }

    static BOOST_QVM_INLINE_CRITICAL
    scalar_type read_element_idx( int i, vec_type const & x ) {
      return vec_traits<vec_type>::write_element_idx(i,const_cast<vec_type &>(x));
    }

    protected:

    static BOOST_QVM_INLINE_TRIVIAL
    scalar_type & write_element_idx( int i, vec_type & m ) {
      /* unspecified */
    }
  };

} }
```

The `vec_traits_defaults` template is designed to be used as a public base for user-defined specializations of the `vec_traits` template, to easily define the required members. If it is used, the only member that must be defined by the user in a `vec_traits` specialization is `write_element`; the `vec_traits_defaults` base will define `read_element`, as well as `scalar_type` and `dim` automatically.

Optionally, the user may also define `write_element_idx`, in which case the `vec_traits_defaults` base will provide a suitable `read_element_idx` definition automatically. If not, `vec_traits_defaults` defines a protected implementation of `write_element_idx` which may be made publicly available by the deriving `vec_traits` specialization in case the vector type for which it is being specialized can not be indexed efficiently. This `write_element_idx` function is less efficient (using meta-programming), implemented in terms of the required user-defined `write_element`.

deduce_vec

#include <boost/qvm/deduce_vec.hpp>

```
namespace boost { namespace qvm {

  template <class V,
    int D=vec_traits<Vector>::dim,
    class S=typename vec_traits<V>::scalar_type>
  struct deduce_vec {

    typedef /*unspecified*/ type;

  };

} }
```

Requires:

- `is_vec<V>::value` is `true`;
- `is_scalar<S>::value` is `true`;
- `is_vec<deduce_vec<V,D,S>::type>::value` must be `true`;
- `deduce_vec<V,D,S>::type` must be copyable;
- `vec_traits<deduce_vec<V,D,S>::type>::dim==D`;
- `vec_traits<deduce_vec<V,D,S>::type>::scalar_type` is the same type as `S`.

This template is used by QVM whenever it needs to deduce a copyable vector type of certain dimension from a single user-supplied function parameter of vector type. The returned type must have accessible copy constructor. Note that `V` may be non-copyable.

The main template definition returns an unspecified copyable vector type of size `D` and scalar type `S`, except if `vec_traits<V>::dim==D` and `vec_traits<V>::scalar_type` is the same type as `S`, in which case it returns `V`, which is suitable only if `V` is a copyable type. QVM also defines (partial) specializations for the non-copyable vector types it produces. Users can define other (partial) specializations for their own types.

A typical use of the `deduce_vec` template is for specifying the preferred vector type to be returned by the generic function template overloads in QVM depending on the type of their arguments.

---

deduce_vec2

#include <boost/qvm/deduce_vec.hpp>

```
namespace boost { namespace qvm {

  template <class A,class B,int D,
    class S=typename deduce_scalar<
      typename scalar<A>::type,
      typename scalar<B>::type>::type>
  struct deduce_vec2 {
    typedef /*unspecified*/ type;
  };

} }
```

Requires:

- Both `scalar<A>::type` and `scalar<B>::type` are well defined;
- `is_vec<A>::value || is_vec<B>::value` is `true`;
- `is_scalar<S>::value` is `true`;
- `is_vec<deduce_vec2<A,B,D,S>::type>::value` must be `true`;
- `deduce_vec2<A,B,D,S>::type` must be copyable;
- `vec_traits<deduce_vec2<A,B,D,S>::type>::dim==D`.
- `vec_traits<deduce_vec2<A,B,D,S>::type>::scalar_type` is the same type as `S`.

This template is used by QVM whenever it needs to deduce a vector type of certain dimension from the types of two user-supplied function parameters. The returned type must have accessible copy constructor (the `A` and `B` types themselves could be non-copyable, and either one of them may be a non-vector type.)

The main template definition returns an unspecified vector type of the requested dimension with `scalar_type` `S`, except if `A` and `B` are the same vector type `V`, in which case `V` is returned, which is only suitable for copyable types. QVM also defines (partial) specializations for the non-copyable vector types it produces. Users can define other (partial) specializations for their own types.

A typical use of the `deduce_vec2` template is for specifying the preferred vector type to be returned by the generic function template overloads in QVM depending on the type of their arguments.

`is_mat`

#include <boost/qvm/mat_traits.hpp>

```
namespace boost { namespace qvm {

  template <class T>
  struct is_mat {

    static bool const value = false;

  };

} }
```

This type template defines a compile-time boolean constant value which can be used to determine whether a type `T` is a matrix type. For matrix types, the `mat_traits` template can be used to access their elements generically, or to obtain their dimensions and scalar type.

`mat_traits`

#include <boost/qvm/mat_traits.hpp>

```
namespace boost { namespace qvm {

  template <class M>
  struct mat_traits {

    /*main template members unspecified*/

  };

  /*
  User-defined (possibly partial) specializations:

  template <>
  struct mat_traits<M> {

    static int const rows = <<user-defined>>;
    static int const cols = <<user-defined>>;
    typedef <<user-defined>> scalar_type;

    template <int R,int C>
    static inline scalar_type read_element( Matrix const & m );

    template <int R,int C>
    static inline scalar_type & write_element( Matrix & m );

    static inline scalar_typeread_element_idx( int r, int c, Matrix const & m );
    static inline scalar_type & write_element_idx( int r, int c, Matrix & m );

  };
  */

} }
```

The `mat_traits` template must be specialized for (user-defined) matrix types in order to enable vector and matrix operations defined in QVM headers for objects of those types.

The matrix operations defined by QVM do not require matrix types to be copyable.

The main `mat_traits` template members are not specified. Valid specializations are required to define the following members:

- `rows`: the expression `mat_traits<Matrix>::rows` must evaluate to a compile-time integer constant greater than 0 that specifies the number of rows in a matrix.
- `cols` must evaluate to a compile-time integer constant greater than 0 that specifies the number of columns in a matrix.
- `scalar_type`: the expression `mat_traits<Matrix>::scalar_type` must be a value type which satisfies the scalar requirements.

In addition, valid specializations of the `mat_traits` template may define the following access functions as static members, where `m` is an object of type `Matrix`, `R` and `C` are compile-time integer constants, and `r` and `c` are variables of type `int`:

- `read_element`: the expression `mat_traits<Matrix>::read_element<R,C>(m)` returns either a copy of or a const reference to the element at row `R` and column `C` of `m`.
- `write_element`: the expression `mat_traits<Matrix>::write_element<R,C>(m)` returns mutable reference to the element at row `R` and column `C` of `m`.
- `read_element_idx`: the expression `mat_traits<Matrix>::read_element_idx(r,c,m)` returns either a copy of or a const reference to the element at row `r` and column `c` of `m`.
- `write_element_idx`: the expression `mat_traits<Matrix>::write_element_idx(r,c,m)` returns mutable reference to the element at row `r` and column `c` of `m`.

It is illegal to call any of the above functions unless `is_mat<Matrix>::value` is true. Even then, matrix types are allowed to define only a subset of the access functions. The general requirements are:

- At least one of `read_element` or `write_element` must be defined;
- If `read_element_idx` is defined, `read_element` must also be defined;
- If `write_element_idx` is defined, `write_element` must also be defined.

Below is an example of a user-defined 3x3 matrix type, and its corresponding specialization of the `mat_traits` template:

```
#include <boost/qvm/mat_traits.hpp>

struct float33 { float a[3][3]; };

namespace boost { namespace qvm {

  template <>
  struct mat_traits<float33> {

    static int const rows=3;
    static int const cols=3;
    typedef float scalar_type;

    template <int R,int C>
    static inline scalar_type & write_element( float33 & m ) {
      return m.a[R][C];
    }

    template <int R,int C>
    static inline scalar_type read_element( float33 const & m ) {
      return m.a[R][C];
    }

    static inline scalar_type & write_element_idx( int r, int c, float33 & m ) {
      return m.a[r][c];
    }

    static inline scalar_type read_element_idx( int r, int c, float33 const & m ) {
      return m.a[r][c];
    }

  };

} }
```

Equivalently, we could use the <<mat_traits_defaults,`mat_traits_defaults` template to shorten the above to:

```
namespace boost { namespace qvm {

  template <>
  struct mat_traits<float33>: mat_traits_defaults<float33,float,3,3> {

    template <int R,int C> static inline scalar_type & write_element( float33 & m ) {
return m.a[R][C]; }

    static inline scalar_type & write_element_idx( int r, int c, float33 & m ) {
      return m.a[r][c];
    }

  };

} }
```

mat_traits_defaults

#include <boost/qvm/mat_traits_defaults.hpp>

```
namespace boost { namespace qvm {

  template <class MatType,class ScalarType,int Rows,int Cols>
  struct mat_traits_defaults
  {
    typedef MatType mat_type;
    typedef ScalarType scalar_type;
    static int const rows=Rows;
    static int const cols=Cols;

    template <int Row,int Col>
    static BOOST_QVM_INLINE_CRITICAL
    scalar_type write_element( mat_type const & x ) {
      return mat_traits<mat_type>::template write_element<Row,Col>(const_cast<mat_type
&>(x));
    }

    static BOOST_QVM_INLINE_CRITICAL
    scalar_type read_element_idx( int r, int c, mat_type const & x ) {
      return mat_traits<mat_type>::write_element_idx(r,c,const_cast<mat_type &>(x));
    }

    protected:

    static BOOST_QVM_INLINE_TRIVIAL
    scalar_type & write_element_idx( int r, int c, mat_type & m ) {
      /* unspecified */
    }
  };

} }
```

The `mat_traits_defaults` template is designed to be used as a public base for user-defined specializations of the `mat_traits` template, to easily define the required members. If it is used, the only member that must be defined by the user in a `mat_traits` specialization is `write_element`; the `mat_traits_defaults` base will define `read_element`, as well as `scalar_type`, `rows` and `cols` automatically.

Optionally, the user may also define `write_element_idx`, in which case the `mat_traits_defaults` base will provide a suitable `read_element_idx` definition automatically. Otherwise, `mat_traits_defaults` defines a protected implementation of `write_element_idx` which may be made publicly available by the deriving `mat_traits` specialization in case the matrix type for which it is being specialized can not be indexed efficiently. This `write_element_idx` function is less efficient (using meta-programming), implemented in terms of the required user-defined `write_element`.

deduce_mat

#include <boost/qvm/deduce_mat.hpp>

```
namespace boost { namespace qvm {

  template <
    class M,
    int R=mat_traits<Matrix>::rows,
    int C=mat_traits<Matrix>::cols,
    class S=typename mat_traits<M>::scalar_type>
  struct deduce_mat {

    typedef /*unspecified*/ type;

  };

} }
```

Requires:

- `is_mat<M,R,C,S>::value` is `true`;
- `is_mat<deduce_mat<M,R,C,S>::type>::value` must be `true`;
- `deduce_mat<M,R,C,S>::type` must be copyable;
- `mat_traits<deduce_mat<M,R,C,S>::type>::rows==R`;
- `mat_traits<deduce_mat<M,R,C,S>::type>::cols==C`,
- `mat_traits<deduce_mat<M,R,C,S>::type::scalar_type` is the same type as `S`.

This template is used by QVM whenever it needs to deduce a copyable matrix type of certain dimensions from a single user-supplied function parameter of matrix type. The returned type must have accessible copy constructor. Note that M itself may be non-copyable.

The main template definition returns an unspecified copyable matrix type of size `R` x `C` and scalar type `S`, except if `mat_traits<M>::rows==R` and `mat_traits<M>::cols==Cols` and `mat_traits<M>::scalar_type` is S, in which case it returns `M`, which is suitable only if `M` is a copyable type. QVM also defines (partial) specializations for the non-copyable matrix types it produces. Users can define other (partial) specializations for their own types.

A typical use of the deduce_mat template is for specifying the preferred matrix type to be returned by the generic function template overloads in QVM depending on the type of their arguments.

---

deduce_mat2

#include <boost/qvm/deduce_mat.hpp>

```
namespace boost { namespace qvm {

  template <class A,class B,int R,int C,
    class S = typename deduce_scalar<
      typename scalar<A>::type,
      typename scalar<B>::type>::type
  struct deduce_mat2 {

    typedef /*unspecified*/ type;

  };

} }
```

Requires:

- Both `scalar<A>::type` and `scalar<B>::type` are well defined;
- `is_mat<A>::value || is_mat<B>::value` is `true`;
- `is_scalar<S>::value` is `true`;
- `is_mat<deduce_mat2<A,B>::type>::value` must be `true`;
- `deduce_mat2<A,B,R,C,S>::type` must be copyable;
- `mat_traits<deduce_mat2<A,B,R,C,S>::type>::rows==R`;
- `mat_traits<deduce_mat2<A,B,R,C,S>::type>::cols==C`;
- `mat_traits<deduce_mat2<A,B,R,C,S>::type>::scalar_type` is the same type as `S`.

This template is used by QVM whenever it needs to deduce a matrix type of certain dimensions from the types of two user-supplied function parameters. The returned type must have accessible copy constructor (the `A` and `B` types themselves could be non-copyable, and either one of them may be a non-matrix type.)

The main template definition returns an unspecified matrix type of the requested dimensions with `scalar_type` `S`, except if `A` and `B` are the same matrix type `M`, in which case `M` is returned, which is only suitable for copyable types. QVM also defines (partial) specializations for the non-copyable matrix types it produces. Users can define other (partial) specializations for their own types.

A typical use of the `deduce_mat2` template is for specifying the preferred matrix type to be returned by the generic function template overloads in QVM depending on the type of their arguments.

Built-in Quaternion, Vector and Matrix Types

QVM defines several class templates (together with appropriate specializations of `quat_traits`, `vec_traits` and `mat_traits` templates) which can be used as generic quaternion, vector and

matrix types. Using these types directly wouldn't be typical though, the main design goal of QVM is to allow users to plug in their own quaternion, vector and matrix types.

quat

#include <boost/qvm/quat.hpp>

```
namespace boost { namespace qvm {

    template <class T>
    struct quat {

      T a[4];

      template <class R>
      operator R() const {
        R r;
        assign(r,*this);
        return r;
      }

    };

    template <class Quaternion>
    struct quat_traits;

    template <class T>
    struct quat_traits< quat<T> > {

      typedef T scalar_type;

      template <int I>
      static scalar_type read_element( quat<T> const & x ) {
        return x.a[I];
      }

      template <int I>
      static scalar_type & write_element( quat<T> & x ) {
        return x.a[I];
      }

    };

} }
```

This is a simple quaternion type. It converts to any other quaternion type.

The partial specialization of the `quat_traits` template makes the `quat` template compatible with the generic operations defined by QVM.

vec

#include <boost/qvm/vec.hpp>

```
namespace boost { namespace qvm {

    template <class T,int Dim>
    struct vec {

      T a[Dim];

      template <class R>
      operator R() const {
        R r;
        assign(r,*this);
        return r;
      }

    };

    template <class Vector>
    struct vec_traits;

    template <class T,int Dim>
    struct vec_traits< vec<T,Dim> > {

      typedef T scalar_type;
      static int const dim=Dim;

      template <int I>
      static scalar_type read_element( vec<T,Dim> const & x ) {
        return x.a[I];
      }

      template <int I>
      static scalar_type & write_element( vec<T,Dim> & x ) {
        return x.a[I];
      }

      static scalar_type read_element_idx( int i, vec<T,Dim> const & x ) {
        return x.a[i];
      }

      static scalar_type & write_element_idx( int i, vec<T,Dim> & x ) {
        return x.a[i];
      }
    };

} }
```

This is a simple vector type. It converts to any other vector type of compatible size.

The partial specialization of the `vec_traits` template makes the `vec` template compatible with the generic operations defined by QVM.

`mat`

#include <boost/qvm/mat.hpp>

```
namespace boost { namespace qvm {

  template <class T,int Rows,int Cols>
  struct mat {

    T a[Rows][Cols];

    template <class R>
    operator R() const {
      R r;
      assign(r,*this);
      return r;
    }

  };

  template <class Matrix>
  struct mat_traits;

  template <class T,int Rows,int Cols>
  struct mat_traits< mat<T,Rows,Cols> > {

    typedef T scalar_type;
    static int const rows=Rows;
    static int const cols=Cols;

    template <int Row,int Col>
    static scalar_type read_element( mat<T,Rows,Cols> const & x ) {
      return x.a[Row][Col];
    }

    template <int Row,int Col>
    static scalar_type & write_element( mat<T,Rows,Cols> & x ) {
      return x.a[Row][Col];
    }

    static scalar_type read_element_idx( int row, int col, mat<T,Rows,Cols> const & x
) {
      return x.a[row][col];
    }

    static scalar_type & write_element_idx( int row, int col, mat<T,Rows,Cols> & x ) {
      return x.a[row][col];
    }

  };

} }
```

This is a simple matrix type. It converts to any other matrix type of compatible size.

The partial specialization of the `mat_traits` template makes the `mat` template compatible with the generic operations defined by QVM.

Element Access

Quaternions

#include <boost/qvm/quat_access.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_quat<Q>::value

  template <class Q> -unspecified-return-type- S( Q & q );
  template <class Q> -unspecified-return-type- V( Q & q );
  template <class Q> -unspecified-return-type- X( Q & q );
  template <class Q> -unspecified-return-type- Y( Q & q );
  template <class Q> -unspecified-return-type- Z( Q & q );

} }
```

An expression of the form `S(q)` can be used to access the scalar component of the quaternion `q`. For example,

```
S(q) *= 42;
```

multiplies the scalar component of `q` by the scalar 42.

An expression of the form `V(q)` can be used to access the vector component of the quaternion `q`. For example,

```
V(q) *= 42
```

multiplies the vector component of `q` by the scalar 42.

The `X`, `Y` and `Z` elements of the vector component can also be accessed directly using `X(q)`, `Y(q)` and `Z(q)`.

The return types are lvalues.

Vectors

#include <boost/qvm/vec_access.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_vec<V>::value

  template <int I,class V> -unspecified-return-type- A( V & v );
  template <class V> -unspecified-return-type- A0( V & v );
  template <class V> -unspecified-return-type- A1( V & v );
  ...
  template <class V> -unspecified-return-type- A9( V & v );

  template <class V> -unspecified-return-type- X( V & v );
  template <class V> -unspecified-return-type- Y( V & v );
  template <class V> -unspecified-return-type- Z( V & v );
  template <class V> -unspecified-return-type- W( V & v );

} }
```

An expression of the form of `A<I>(v)` can be used to access the `I`-th element a vector object `v`. For example, the expression:

```
A<1>(v) *= 42;
```

can be used to multiply the element at index 1 (indexing in QVM is always zero-based) of a vector `v` by 42.

For convenience, there are also non-template overloads for `I` from 0 to 9; an alternative way to write the above expression is:

```
A1(v) *= 42;
```

`X`, `Y`, `Z` and `W` act the same as `A0`/`A1`/`A2`/`A3`; yet another alternative way to write the above expression is:

```
Y(v) *= 42;
```

The return types are lvalues.

Vector Element Swizzling

#include <boost/qvm/swizzle.hpp>

```
namespace boost { namespace qvm {

  //*** Accessing vector elements by swizzling ***

  //2D view proxies, only enabled if:
  //  is_vec<V>::value
  template <class V> -unspecified-2D-vector-type- XX( V & v );
  template <class V> -unspecified-2D-vector-type- XY( V & v );
  template <class V> -unspecified-2D-vector-type- XZ( V & v );
  template <class V> -unspecified-2D-vector-type- XW( V & v );
  template <class V> -unspecified-2D-vector-type- X0( V & v );
  template <class V> -unspecified-2D-vector-type- X1( V & v );
  template <class V> -unspecified-2D-vector-type- YX( V & v );
  template <class V> -unspecified-2D-vector-type- YY( V & v );
  template <class V> -unspecified-2D-vector-type- YZ( V & v );
  template <class V> -unspecified-2D-vector-type- YW( V & v );
  template <class V> -unspecified-2D-vector-type- Y0( V & v );
  template <class V> -unspecified-2D-vector-type- Y1( V & v );
  template <class V> -unspecified-2D-vector-type- ZX( V & v );
  template <class V> -unspecified-2D-vector-type- ZY( V & v );
  template <class V> -unspecified-2D-vector-type- ZZ( V & v );
  template <class V> -unspecified-2D-vector-type- ZW( V & v );
  template <class V> -unspecified-2D-vector-type- Z0( V & v );
  template <class V> -unspecified-2D-vector-type- Z1( V & v );
  template <class V> -unspecified-2D-vector-type- WX( V & v );
  template <class V> -unspecified-2D-vector-type- WY( V & v );
  template <class V> -unspecified-2D-vector-type- WZ( V & v );
  template <class V> -unspecified-2D-vector-type- WW( V & v );
  template <class V> -unspecified-2D-vector-type- W0( V & v );
  template <class V> -unspecified-2D-vector-type- W1( V & v );
  ...
  //2D view proxies, only enabled if:
  //  is_scalar<S>::value
  template <class S> -unspecified-2D-vector-type- X0( S & s );
  template <class S> -unspecified-2D-vector-type- X1( S & s );
  template <class S> -unspecified-2D-vector-type- XX( S & s );
  ...
  -unspecified-2D-vector-type- _00();
  -unspecified-2D-vector-type- _01();
  -unspecified-2D-vector-type- _10();
  -unspecified-2D-vector-type- _11();

  //3D view proxies, only enabled if:
  //  is_vec<V>::value
  template <class V> -unspecified-3D-vector-type- XXX( V & v );
  ...
  template <class V> -unspecified-3D-vector-type- XXW( V & v );
  template <class V> -unspecified-3D-vector-type- XX0( V & v );
  template <class V> -unspecified-3D-vector-type- XX1( V & v );
  template <class V> -unspecified-3D-vector-type- XYX( V & v );
  ...
```

```
  template <class V> -unspecified-3D-vector-type- XY1( V & v );
  ...
  template <class V> -unspecified-3D-vector-type- WW1( V & v );
  ...
  //3D view proxies, only enabled if:
  //  is_scalar<S>::value
  template <class S> -unspecified-3D-vector-type- X00( S & s );
  template <class S> -unspecified-3D-vector-type- X01( S & s );
  ...
  template <class S> -unspecified-3D-vector-type- XXX( S & s );
  template <class S> -unspecified-3D-vector-type- XX0( S & s );
  ...
  -unspecified-3D-vector-type- _000();
  -unspecified-3D-vector-type- _001();
  -unspecified-3D-vector-type- _010();
  ...
  -unspecified-3D-vector-type- _111();

  //4D view proxies, only enabled if:
  //  is_vec<V>::value
  template <class V> -unspecified-4D-vector-type- XXXX( V & v );
  ...
  template <class V> -unspecified-4D-vector-type- XXXW( V & v );
  template <class V> -unspecified-4D-vector-type- XXX0( V & v );
  template <class V> -unspecified-4D-vector-type- XXX1( V & v );
  template <class V> -unspecified-4D-vector-type- XXYX( V & v );
  ...
  template <class V> -unspecified-4D-vector-type- XXY1( V & v );
  ...
  template <class V> -unspecified-4D-vector-type- WWW1( V & v );
  ...
  //4D view proxies, only enabled if:
  //  is_scalar<S>::value
  template <class S> -unspecified-4D-vector-type- X000( S & s );
  template <class S> -unspecified-4D-vector-type- X001( S & s );
  ...
  template <class S> -unspecified-4D-vector-type- XXXX( S & s );
  template <class S> -unspecified-4D-vector-type- XX00( S & s );
  ...
  -unspecified-4D-vector-type- _0000();
  -unspecified-4D-vector-type- _0001();
  -unspecified-4D-vector-type- _0010();
  ...
  -unspecified-4D-vector-type- _1111();

} }
```

Swizzling allows zero-overhead direct access to a (possibly rearranged) subset of the elements of 2D, 3D and 4D vectors. For example, if `v` is a 4D vector, the expression `YX(v) is a 2D view proxy whose `X` element refers to the `Y` element of `v`, and whose `Y` element refers to the `X` element

of `v`. Like other view proxies `YX` is an lvalue, that is, if `v2` is a 2D vector, one could write:

```
YX(v) = v2;
```

The above will leave the `Z` and `W` elements of `v` unchanged but assign the `Y` element of `v2` to the `X` element of `v` and the `X` element of `v2` to the `Y` element of `v`.

All permutations of `X`, `Y`, `Z`, `W`, `0`, `1` for 2D, 3D and 4D swizzling are available (if the first character of the swizzle identifier is `0` or `1`, it is preceded by a _, for example `_11XY`).

It is valid to use the same vector element more than once: the expression `ZZZ(v)` is a 3D vector whose `X`, `Y` and `Z` elements all refer to the `Z` element of `v`.

Finally, scalars can be "swizzled" to access them as vectors: the expression `_0X01(42.0f)` is a 4D vector with `X`=0, `Y`=42.0, `Z`=0, `W`=1.

Matrices

#include <boost/qvm/mat_access.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_quat<Q>::value

  template <int R,int C,class M> -unspecified-return-type- A( M & m );

  template <class M> -unspecified-return-type- A00( M & m );
  template <class M> -unspecified-return-type- A01( M & m );
  ...
  template <class M> -unspecified-return-type- A09( M & m );
  template <class M> -unspecified-return-type- A10( M & m );
  ...
  template <class M> -unspecified-return-type- A99( M & m );

} }
```

An expression of the form `A<R,C>(m)` can be used to access the element at row `R` and column `C` of a matrix object `m`. For example, the expression:

```
A<4,2>(m) *= 42;
```

can be used to multiply the element at row 4 and column 2 of a matrix `m` by 42.

For convenience, there are also non-template overloads for `R` from `0` to `9` and `C` from `0` to `9`; an alternative way to write the above expression is:

```
A42(m) *= 42;
```

The return types are lvalues.

Quaternion Operations

assign

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_quat<A>::value && is_quat<B>::value
  template <class A,class B>
  A & assign( A & a, B const & b );

} }
```

Effects:

Copies all elements of the quaternion `b` to the quaternion `a`.

Returns:

`a`.

convert_to

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_quat<R>::value && is_quat<A>::value
  template <class R,class A>
  R convert_to( A const & a );

  //Only enabled if:
  //  is_quat<R>::value && is_mat<A>::value &&
  //  mat_traits<A>::rows==3 && mat_traits<A>::cols==3
  template <class R,class A>
  R convert_to( A const & m );

} }
```

Requires:

`R` must be copyable.

Effects:

- The first overload is equivalent to: `R r; assign(r,a); return r;`
- The second overload assumes that `m` is an orthonormal rotation matrix and converts it to a quaternion that performs the same rotation.

`operator-=`

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_quat<A>::value && is_quat<B>::value
  template <class A,class B>
  A & operator-=( A & a, B const & b );

} }
```

Effects:

Subtracts the elements of `b` from the corresponding elements of `a`.

Returns:

`a`.

`operator-` (unary)

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

   //Only enabled if: is_quat<A>::value
  template <class A>
  typename deduce_quat<A>::type
  operator-( A const & a );

} }
```

Returns:

A quaternion of the negated elements of `a`.

The `deduce_quat` template can be specialized to deduce the desired return type from the type `A`.

`operator-` (binary)

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_quat<A>::value && is_quat<B>::value
  template <class A,class B>
  typename deduce_quat2<A,B>::type
  operator-( A const & a, B const & b );

} }
```

Returns:

A quaternion with elements equal to the elements of `b` subtracted from the corresponding elements of `a`.

The `deduce_quat2` template can be specialized to deduce the desired return type, given the types `A` and `B`.

`operator+=`

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_quat<A>::value && is_quat<B>::value
  template <class A,class B>
  A & operator+=( A & a, B const & b );

} }
```

Effects:

Adds the elements of `b` to the corresponding elements of `a`.

Returns:

`a`.

operator+

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_quat<A>::value && is_quat<B>::value &&
  template <class A,class B>
  typename deduce_quat2<A,B>::type
  operator+( A const & a, B const & b );

} }
```

Returns:

A quaternion with elements equal to the elements of `a` added to the corresponding elements of `b`.

The `deduce_quat2` template can be specialized to deduce the desired return type, given the types `A` and `B`.

---

operator/= (scalar)

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if: is_quat<A>::value && is_scalar<B>::value
  template <class A,class B>
  A & operator/=( A & a, B b );

} }
```

Effects:

This operation divides a quaternion by a scalar.

Returns:

`a`.

---

operator/ (scalar)

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if: is_quat<A>::value && is_scalar<B>::value
  template <class A,class B>
  typename deduce_quat2<A,B>>::type
  operator/( A const & a, B b );

} }
```

Returns:

A quaternion that is the result of dividing the quaternion `a` by the scalar `b`.

The `deduce_quat2` template can be specialized to deduce the desired return type from the types `A` and `B`.

`operator*=` (scalar)

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if: is_quat<A>::value && is_scalar<B>::value
  template <class A,class B>
  A & operator*=( A & a, B b );

} }
```

Effects:

This operation multiplies the quaternion `a` by the scalar `b`.

Returns:

`a`.

`operator*=`

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_quat<A>::value && is_quat<B>::value
  template <class A,class B>
  A & operator*=( A & a, B const & b );

} }
```

Effects:

As if:

```
A tmp(a);
a = tmp * b;
return a;
```

`operator*` (scalar)

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if: is_quat<A>::value && is_scalar<B>::value
  template <class A,class B>
  typename deduce_quat2<A,B>::type
  operator*( A const & a, B b );

} }
```

Returns:

A quaternion that is the result of multiplying the quaternion `a` by the scalar `b`.

The `deduce_quat2` template can be specialized to deduce the desired return type from the types `A` and `B`.

`operator*`

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_quat<A>::value && is_quat<B>::value
  template <class A,class B>
  typename deduce_quat2<A,B>::type
  operator*( A const & a, B const & b );

} }
```

Returns:

The result of multiplying the quaternions `a` and `b`.

The `deduce_quat2` template can be specialized to deduce the desired return type, given the types `A` and `B`.

---

operator==

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_quat<A>::value && is_quat<B>::value
  template <class A,class B>
  bool operator==( A const & a, B const & b );

} }
```

Returns:

`true` if each element of `a` compares equal to its corresponding element of `b`, `false` otherwise.

---

operator!=

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_quat<A>::value && is_quat<B>::value
  template <class A,class B>
  bool operator!=( A const & a, B const & b );

} }
```

Returns:

`!(a == b)`.

---

cmp

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_quat<A>::value && is_quat<B>::value
  template <class A,class B,class Cmp>
  bool cmp( A const & a, B const & b, Cmp pred );

} }
```

Returns:

Similar to `operator==`, except that it uses the binary predicate `pred` to compare the individual quaternion elements.

---

mag_sqr

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if: is_quat<A>::value
  template <class A>
  typename quat_traits<A>::scalar_type
  mag_sqr( A const & a );

} }
```

Returns:

The squared magnitude of the quaternion `a`.

mag

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if: is_quat<A>::value
  template <class A>
  typename quat_traits<A>::scalar_type
  mag( A const & a );

} }
```

Returns:

The magnitude of the quaternion `a`.

normalized

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if: is_quat<A>::value
  template <class A>
  typename deduce_quat<A>::type
  normalized( A const & a );

} }
```

Effects:

As if:

```
typename deduce_quat<A>::type tmp;
assign(tmp,a);
normalize(tmp);
return tmp;
```

The `deduce_quat` template can be specialized to deduce the desired return type from the type `A`.

normalize

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if: is_quat<A>::value
  template <class A>
  void normalize( A & a );

} }
```

Effects:

Normalizes `a`.

Ensures:

`mag(a)==scalar_traits<typename quat_traits<A>::scalar_type>::value(1).`

Throws:

If the magnitude of `a` is zero, throws `zero_magnitude_error`.

`dot`

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_quat<A>::value && is_quat<B>::value
  template <class A,class B>
  typename deduce_scalar<A,B>::type
  dot( A const & a, B const & b );

} }
```

Returns:

The dot product of the quaternions `a` and `b`.

The `deduce_scalar` template can be specialized to deduce the desired return type, given the types `A` and `B`.

`conjugate`

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if: is_quat<A>::value
  template <class A>
  typename deduce_quat<A>::type
  conjugate( A const & a );

} }
```

Returns:

Computes the conjugate of `a`.

The `deduce_quat` template can be specialized to deduce the desired return type from the type `A`.

`inverse`

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if: is_quat<A>::value
  template <class A>
  typename deduce_quat<A>::type
  inverse( A const & a );

} }
```

Returns:

Computes the multiplicative inverse of `a`, or the conjugate-to-norm ratio.

Throws:

If the magnitude of `a` is zero, throws `zero_magnitude_error`.

If `a` is known to be unit length, `conjugate` is equivalent to `inverse`, yet it is faster to compute.

The `deduce_quat` template can be specialized to deduce the desired return type from the type `A`.

---

slerp

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_quat<A>::value && is_quat<B>::value && is_scalar<C>
  template <class A,class B,class C>
  typename deduce_quat2<A,B> >::type
  slerp( A const & a, B const & b, C c );

} }
```

Preconditions:

`t>=0 && t<=1`.

Returns:

A quaternion that is the result of Spherical Linear Interpolation of the quaternions `a` and `b` and the interpolation parameter `c`. When `slerp` is applied to unit quaternions, the quaternion path maps to a path through 3D rotations in a standard way. The effect is a rotation with uniform angular velocity around a fixed rotation axis.

The `deduce_quat2` template can be specialized to deduce the desired return type, given the types `A` and `B`.

---

zero_quat

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

  template <class T>
  -unspecified-return-type- zero_quat();

} }
```

Returns:

A read-only quaternion of unspecified type with `scalar_type` `T`, with all elements equal to `scalar_traits<T>::value(0)`.

---

set_zero

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if: is_quat<A>::value
  template <class A>
  void set_zero( A & a );

} }
```

Effects:

As if:

```
assign(a,
  zero_quat<typename quat_traits<A>::scalar_type>());
```

`identity_quat`

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

  template <class S>
  -unspecified-return-type- identity_quat();

} }
```

Returns:

An identity quaternion with scalar type `S`.

`set_identity`

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if: is_quat<A>::value
  template <class A>
  void set_identity( A & a );

} }
```

Effects:

As if:

```
assign(
  a,
  identity_quat<typename quat_traits<A>::scalar_type>());
```

rot_quat

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_vec<A>::value && vec_traits<A>::dim==3
  template <class A>
  -unspecified-return-type- rot_quat( A const & axis, typename vec_traits<A
>::scalar_type angle );

} }
```

Returns:

A quaternion of unspecified type which performs a rotation around the `axis` at `angle` radians.

Throws:

In case the axis vector has zero magnitude, throws `zero_magnitude_error`.

The `rot_quat` function is not a view proxy; it returns a temp object.

set_rot

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_quat<A>::value &&
  //  is_vec<B>::value && vec_traits<B>::dim==3
  template <class A>
  void set_rot( A & a, B const & axis, typename vec_traits<B>::scalar_type angle );

} }
```

Effects:

As if:

```
assign(
  a,
  rot_quat(axis,angle));
```

rotate

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_quat<A>::value &&
  //  is_vec<B>::value && vec_traits<B>::dim==3
  template <class A,class B>
  void rotate( A & a, B const & axis, typename quat_traits<A>::scalar_type angle );

} }
```

Effects:

As if: `a *= rot_quat(axis,angle)`.

rotx_quat

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

  template <class Angle>
  -unspecified-return-type- rotx_quat( Angle const & angle );

} }
```

Returns:

A view proxy quaternion of unspecified type and scalar type `Angle`, which performs a rotation around the X axis at `angle` radians.

set_rotx

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if: is_quat<A>::value
  template <class A>
  void set_rotx( A & a, typename quat_traits<A>::scalar_type angle );

} }
```

Effects:

As if:

```
assign(
  a,
  rotx_quat(angle));
```

`rotate_x`

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if: is_quat<A>::value
  template <class A>
  void rotate_x( A & a, typename quat_traits<A>::scalar_type angle );

} }
```

Effects:

As if: `a *= rotx_quat(angle)`.

`roty_quat`

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

  template <class Angle>
  -unspecified-return-type- roty_quat( Angle const & angle );

} }
```

Returns:

A view proxy quaternion of unspecified type and scalar type `Angle`, which performs a rotation around the Y axis at `angle` radians.

`set_roty`

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if: is_quat<A>::value
  template <class A>
  void set_rotz( A & a, typename quat_traits<A>::scalar_type angle );

} }
```

Effects:

As if:

```
assign(
  a,
  roty_quat(angle));
```

`rotate_y`

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if: is_quat<A>::value
  template <class A>
  void rotate_y( A & a, typename quat_traits<A>::scalar_type angle );

} }
```

Effects:

As if: `a *= roty_quat(angle)`.

`rotz_quat`

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

    template <class Angle>
    -unspecified-return-type- rotz_quat( Angle const & angle );

} }
```

Returns:

A view proxy quaternion of unspecified type and scalar type `Angle`, which performs a rotation around the Z axis at `angle` radians.

`set_rotz`

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if: is_quat<A>::value
  template <class A>
  void set_rotz( A & a, typename quat_traits<A>::scalar_type angle );

} }
```

Effects:

As if:

```
assign(
  a,
  rotz_quat(angle));
```

`rotate_z`

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if: is_quat<A>::value
  template <class A>
  void rotate_z( A & a, typename quat_traits<A>::scalar_type angle );

} }
```

Effects:

As if: `a *= rotz_quat(angle)`.

`scalar_cast`

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if: is_quat<A>::value
  template <class Scalar,class A>
  -unspecified-return_type- scalar_cast( A const & a );

} }
```

Returns:

A read-only view proxy of `a` that looks like a quaternion of the same dimensions as `a`, but with `scalar_type` `Scalar` and elements constructed from the corresponding elements of `a`.

`qref`

#include <boost/qvm/quat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if: is_quat<A>::value
  template <class A>
  -unspecified-return-type- qref( A & a );

} }
```

Returns:

An identity view proxy of `a`; that is, it simply accesses the elements of `a`.

`qref` allows calling QVM operations when `a` is of built-in type, for example a plain old C array.

Vector Operations

`assign`

#include <boost/qvm/vec_operations.hpp>

```
namespace boost { namespace qvm {

    //Only enabled if:
    //  is_vec<A>::value && is_vec<B>::value &&
    //  vec_traits<A>::dim==vec_traits<B>::dim
    template <class A,class B>
    A & assign( A & a, B const & b );

} }
```

Effects:

Copies all elements of the vector `b` to the vector `a`.

Returns:

`a`.

`convert_to`

#include <boost/qvm/vec_operations.hpp>

```
namespace boost { namespace qvm {

    //Only enabled if:
    //  is_vec<R>::value && is_vec<A>::value &&
    //  vec_traits<R>::dim==vec_traits<A>::dim
    template <class R,class A>
    R convert_to( A const & a );

} }
```

Requires:

`R` must be copyable.

Effects:

As if: `R r; assign(r,a); return r;`

`operator-=`

#include <boost/qvm/vec_operations.hpp>

```
namespace boost { namespace qvm {

    //Only enabled if:
    //  is_vec<A>::value && is_vec<B>::value &&
    //  vec_traits<A>::dim==vec_traits<B>::dim
    template <class A,class B>
    A & operator-=( A & a, B const & b );

} }
```

Effects:

Subtracts the elements of `b` from the corresponding elements of `a`.

Returns:

`a`.

`operator-` (unary)

operator-(vec)

#include <boost/qvm/vec_operations.hpp>

```
namespace boost { namespace qvm {

    //Only enabled if: is_vec<A>::value
    template <class A>
    typename deduce_vec<A>::type
    operator-( A const & a );

} }
```

Returns:

A vector of the negated elements of `a`.

The `deduce_vec` template can be specialized to deduce the desired return type from the type `A`.

`operator-` (binary)

#include <boost/qvm/vec_operations.hpp>

```
namespace boost { namespace qvm {

    //Only enabled if:
    //  is_vec<A>::value && is_vec<B>::value &&
    //  vec_traits<A>::dim==vec_traits<B>::dim
    template <class A,class B>
    typename deduce_vec2<A,B,vec_traits<A>::dim>::type
    operator-( A const & a, B const & b );

} }
```

Returns:

A vector of the same size as `a` and `b`, with elements the elements of `b` subtracted from the corresponding elements of `a`.

The `deduce_vec2` template can be specialized to deduce the desired return type, given the types `A` and `B`.

---

operator+=

#include <boost/qvm/vec_operations.hpp>

```
namespace boost { namespace qvm {

    //Only enabled if:
    //  is_vec<A>::value && is_vec<B>::value &&
    //  vec_traits<A>::dim==vec_traits<B>::dim
    template <class A,class B>
    A & operator+=( A & a, B const & b );

} }
```

Effects:

Adds the elements of `b` to the corresponding elements of `a`.

Returns:

`a`.

---

operator+

#include <boost/qvm/vec_operations.hpp>

```
namespace boost { namespace qvm {

    //Only enabled if:
    //  is_vec<A>::value && is_vec<B>::value &&
    //  vec_traits<A>::dim==vec_traits<B>::dim
    template <class A,class B>
    typename deduce_vec2<A,B,vec_traits<A>::dim>::type
    operator+( A const & a, B const & b );

} }
```

Returns:

A vector of the same size as `a` and `b`, with elements the elements of `b` added to the corresponding elements of `a`.

The `deduce_vec2` template can be specialized to deduce the desired return type, given the types `A` and `B`.

`operator/=` (scalar)

#include <boost/qvm/vec_operations.hpp>

```
namespace boost { namespace qvm {

    //Only enabled if: is_vec<A>::value && is_scalar<B>::value
    template <class A,class B>
    A & operator/=( A & a, B b );

} }
```

Effects:

This operation divides a vector by a scalar.

Returns:

`a`.

`operator/`

#include <boost/qvm/vec_operations.hpp>

```
namespace boost { namespace qvm {

    //Only enabled if: is_vec<A>::value && is_scalar<B>::value
    template <class A,class B>
    typename deduce_vec2<A,B,vec_traits<A>::dim>::type
    operator/( A const & a, B b );

} }
```

Returns:

A vector that is the result of dividing the vector `a` by the scalar `b`.

The `deduce_vec2` template can be specialized to deduce the desired return type from the types `A` and `B`.

`operator*=`

#include <boost/qvm/vec_operations.hpp>

```
namespace boost { namespace qvm {

    //Only enabled if: is_vec<A>::value && is_scalar<B>::value
    template <class A,class B>
    A & operator*=( A & a, B b );

} }
```

Effects:

This operation multiplies the vector `a` by the scalar `b`.

Returns:

`a`.

`operator*`

#include <boost/qvm/vec_operations.hpp>

```
namespace boost { namespace qvm {

    //Only enabled if: is_vec<A>::value && is_scalar<B>::value
    template <class A,class B>
    typename deduce_vec2<A,B,vec_traits<A>::dim>::type
    operator*( A const & a, B b );

    //Only enabled if: is_scalar<B>::value && is_vec<A>::value
    template <class B,class A>
    typename deduce_vec2<A,B,vec_traits<A>::dim>::type
    operator*( B b, A const & a );

} }
```

Returns:

A vector that is the result of multiplying the vector `a` by the scalar `b`.

The `deduce_vec2` template can be specialized to deduce the desired return type from the types `A` and `B`.

`operator==`

#include <boost/qvm/vec_operations.hpp>

```
namespace boost { namespace qvm {

    //Only enabled if:
    //  is_vec<A>::value && is_vec<B>::value &&
    //  vec_traits<A>::dim==vec_traits<B>::dim
    template <class A,class B>
    bool operator==( A const & a, B const & b );

} }
```

Returns:

`true` if each element of `a` compares equal to its corresponding element of `b`, `false` otherwise.

`operator!=`

#include <boost/qvm/vec_operations.hpp>

```
namespace boost { namespace qvm {

    //Only enabled if:
    //  is_vec<A>::value && is_vec<B>::value &&
    //  vec_traits<A>::dim==vec_traits<B>::dim
    template <class A,class B>
    bool operator!=( A const & a, B const & b );

} }
```

Returns:

`!(a == b)`.

`cmp`

```
.#include <boost/qvm/mat_operations.hpp>

namespace boost
{
  namespace qvm
  {
    //Only enabled if:
    //  is_mat<A>::value && is_mat<B>::value &&
    //  mat_traits<A>::rows==mat_traits<B>::rows &&
    //  mat_traits<A>::cols==mat_traits<B>::cols
    template <class A,class B,class Cmp>
    bool cmp( A const & a, B const & b, Cmp pred );

} }
```

Returns:

Similar to `operator==`, except that the individual elements of `a` and `b` are passed to the binary predicate `pred` for comparison.

`mag_sqr`

#include <boost/qvm/vec_operations.hpp>

```
namespace boost { namespace qvm {

    //Only enabled if:
    //  is_vec<A>::value
    template <class A>
    typename vec_traits<A>::scalar_type
    mag_sqr( A const & a );

} }
```

Returns:

The squared magnitude of the vector `a`.

`mag`

#include <boost/qvm/vec_operations.hpp>

```
namespace boost { namespace qvm {

    //Only enabled if:
    //  is_vec<A>::value
    template <class A>
    typename vec_traits<A>::scalar_type
    mag( A const & a );

} }
```

Returns:

The magnitude of the vector `a`.

`normalized`

#include <boost/qvm/vec_operations.hpp>

```
namespace boost { namespace qvm {

    //Only enabled if:
    //  is_vec<A>::value
    template <class A>
    typename deduce_vec<A>::type
    normalized( A const & a );

} }
```

Effects:

As if:

```
typename deduce_vec<A>::type tmp;
assign(tmp,a);
normalize(tmp);
return tmp;
```

The `deduce_vec` template can be specialized to deduce the desired return type from the type `A`.

---

normalize

#include <boost/qvm/vec_operations.hpp>

```
namespace boost { namespace qvm {

    //Only enabled if:
    //  is_vec<A>::value
    template <class A>
    void normalize( A & a );

} }
```

Effects:

Normalizes `a`.

Ensures:

`mag(a)==scalar_traits<typename vec_traits<A>::scalar_type>::value(1)`.

Throws:

If the magnitude of `a` is zero, throws `zero_magnitude_error`.

---

dot

#include <boost/qvm/vec_operations.hpp>

```
namespace boost { namespace qvm {

    //Only enabled if:
    //  is_vec<A>::value && is_vec<B>::value &&
    //  vec_traits<A>::dim==vec_traits<B>::dim
    template <class A,class B>
    typename deduce_scalar<A,B>::type
    dot( A const & a, B const & b );

} }
```

Returns:

The dot product of the vectors `a` and `b`.

The `deduce_scalar` template can be specialized to deduce the desired return type, given the types `A` and `B`.

`cross`

#include <boost/qvm/vec_operations.hpp>

```
namespace boost { namespace qvm {

    //Only enabled if:
    //  is_vec<A>::value && is_vec<B>::value &&
    //  vec_traits<A>::dim==3 && vec_traits<B>::dim==3
    template <class A,class B>
    typename deduce_vec2<A,B,3>::type
    cross( A const & a, B const & b );

    //Only enabled if:
    //  is_vec<A>::value && is_vec<B>::value &&
    //  vec_traits<A>::dim==2 && vec_traits<B>::dim==2
    template <class A,class B>
    typename deduce_scalar<
      typename vec_traits<A>::scalar_type,
      typename vec_traits<B>::scalar_type>::type
    cross( A const & a, B const & b );

} }
```

Returns:

The cross product of the vectors `a` and `b`.

The `deduce_vec2` (and `deduce_scalar`) templates can be specialized to deduce the desired return type, given the types `A` and `B`.

zero_vec

#include <boost/qvm/vec_operations.hpp>

```
namespace boost { namespace qvm {

    template <class T,int S>
    -unspecified-return-type- zero_vec();

} }
```

Returns:

A read-only vector of unspecified type with `scalar_type` `T` and size `S`, with all elements equal to `scalar_traits<T>::value(0)`.

set_zero

#include <boost/qvm/vec_operations.hpp>

```
namespace boost { namespace qvm {

    //Only enabled if:
    //  is_vec<A>::value
    template <class A>
    void set_zero( A & a );

} }
```

Effects:

As if:

```
assign(a,
  zero_vec<
    typename vec_traits<A>::scalar_type,
    vec_traits<A>::dim>());
```

scalar_cast

#include <boost/qvm/vec_operations.hpp>

```
namespace boost { namespace qvm {

    //Only enabled if: is_vec<A>::value
    template <class Scalar,class A>
    -unspecified-return_type- scalar_cast( A const & a );

} }
```

Returns:

A read-only view proxy of `a` that looks like a vector of the same dimensions as `a`, but with `scalar_type` `Scalar` and elements constructed from the corresponding elements of `a`.

`vref`

#include <boost/qvm/vec_operations.hpp>

```
namespace boost { namespace qvm {

    //Only enabled if: is_vec<A>::value
    template <class A>
    -unspecified-return-type- vref( A & a );

} }
```

Returns:

An identity view proxy of `a`; that is, it simply accesses the elements of `a`.

`vref` allows calling QVM operations when `a` is of built-in type, for example a plain old C array.

Matrix Operations

`assign`

#include <boost/qvm/mat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_mat<A>::value && is_mat<B>::value &&
  //  mat_traits<A>::rows==mat_traits<B>::rows &&
  //  mat_traits<A>::cols==mat_traits<B>::cols
  template <class A,class B>
  A & assign( A & a, B const & b );

} }
```

Effects:

Copies all elements of the matrix `b` to the matrix `a`.

Returns:

`a`.

`convert_to`

#include <boost/qvm/mat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_mat<R>::value && is_mat<A>::value &&
  //  mat_traits<R>::rows==mat_traits<A>::rows &&
  //  mat_traits<R>::cols==mat_traits<A>::cols
  template <class R,class A>
  R convert_to( A const & a );

} }
```

Requires:

`R` must be copyable.

Effects:

As if: `R r; assign(r,a); return r;`

`operator-=`

#include <boost/qvm/mat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_mat<A>::value && is_mat<B>::value &&
  //  mat_traits<A>::rows==mat_traits<B>::rows &&
  //  mat_traits<A>::cols==mat_traits<B>::cols
  template <class A,class B>
  A & operator-=( A & a, B const & b );

} }
```

Effects:

Subtracts the elements of `b` from the corresponding elements of `a`.

Returns:

`a`.

`operator-` (unary)

#include <boost/qvm/mat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if: is_mat<A>::value
  template <class A>
  typename deduce_mat<A>::type
  operator-( A const & a );

} }
```

Returns:

A matrix of the negated elements of `a`.

The `deduce_mat` template can be specialized to deduce the desired return type from the type `A`.

`operator-`

#include <boost/qvm/mat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_mat<A>::value && is_mat<B>::value &&
  //  mat_traits<A>::rows==mat_traits<B>::rows &&
  //  mat_traits<A>::cols==mat_traits<B>::cols
  template <class A,class B>
  typename deduce_mat2<A,B,mat_traits<A>::rows,mat_traits<A>::cols>::type
  operator-( A const & a, B const & b );

} }
```

Returns:

A matrix of the same size as `a` and `b`, with elements the elements of `b` subtracted from the corresponding elements of `a`.

The `deduce_mat2` template can be specialized to deduce the desired return type, given the types `A` and `B`.

---

operator+=

#include <boost/qvm/mat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_mat<A>::value && is_mat<B>::value &&
  //  mat_traits<A>::rows==mat_traits<B>::rows &&
  //  mat_traits<A>::cols==mat_traits<B>::cols
  template <class A,class B>
  A & operator+=( A & a, B const & b );

} }
```

Effects:

Adds the elements of `b` to the corresponding elements of `a`.

Returns:

`a`.

---

operator+

#include <boost/qvm/mat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_mat<A>::value && is_mat<B>::value &&
  //  mat_traits<A>::rows==mat_traits<B>::rows &&
  //  mat_traits<A>::cols==mat_traits<B>::cols
  template <class A,class B>
  typename deduce_mat2<A,B,mat_traits<A>::rows,mat_traits<A>::cols>::type
  operator+( A const & a, B const & b );

} }
```

Returns:

A matrix of the same size as `a` and `b`, with elements the elements of `b` added to the corresponding elements of `a`.

The `deduce_mat2` template can be specialized to deduce the desired return type, given the types `A` and `B`.

`operator/=` (scalar)

#include <boost/qvm/mat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if: is_mat<A>::value && is_scalar<B>::value
  template <class A,class B>
  A & operator/=( A & a, B b );

} }
```

Effects:

This operation divides a matrix by a scalar.

Returns:

`a`.

`operator/` (scalar)

#include <boost/qvm/mat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if: is_mat<A>::value && is_scalar<B>::value
  template <class A,class B>
  typename deduce_mat2<A,B,mat_traits<A>::rows,mat_traits<A>::cols>::type
  operator/( A const & a, B b );

} }
```

Returns:

A matrix that is the result of dividing the matrix `a` by the scalar `b`.

The `deduce_mat2` template can be specialized to deduce the desired return type from the types `A` and `B`.

`operator*=`

#include <boost/qvm/mat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_mat<A>::value && is_mat<B>::value &&
  //  mat_traits<A>::rows==mat_traits<A>::cols &&
  //  mat_traits<A>::rows==mat_traits<B>::rows &&
  //  mat_traits<A>::cols==mat_traits<B>::cols
  template <class A,class B>
  A & operator*=( A & a, B const & b );

} }
```

Effects:

As if:

```
A tmp(a);
a = tmp * b;
return a;
```

`operator*=` (scalar)

#include <boost/qvm/mat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if: is_mat<A>::value && is_scalar<B>::value
  template <class A,class B>
  A & operator*=( A & a, B b );

} }
```

Effects:

This operation multiplies the matrix `a` matrix by the scalar `b`.

Returns:

`a`.

`operator*`

#include <boost/qvm/mat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_mat<A>::value && is_mat<B>::value &&
  //  mat_traits<A>::cols==mat_traits<B>::rows
  template <class A,class B>
  typename deduce_mat2<A,B,mat_traits<A>::rows,mat_traits<B>::cols>::type
  operator*( A const & a, B const & b );

} }
```

Returns:

The result of multiplying the matrices `a` and `b`.

The `deduce_mat2` template can be specialized to deduce the desired return type, given the types `A` and `B`.

`operator*` (scalar)

#include <boost/qvm/mat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if: is_mat<A>::value && is_scalar<B>::value
  template <class A,class B>
  typename deduce_mat2<A,B,mat_traits<A>::rows,mat_traits<A>::cols>::type
  operator*( A const & a, B b );

  //Only enabled if: is_scalar<B>::value && is_mat<A>::value
  template <class B,class A>
  typename deduce_mat2<A,B,mat_traits<A>::rows,mat_traits<A>::cols>::type
  operator*( B b, A const & a );

} }
```

Returns:

A matrix that is the result of multiplying the matrix `a` by the scalar `b`.

The `deduce_mat2` template can be specialized to deduce the desired return type from the types `A` and `B`.

`operator==`

#include <boost/qvm/mat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_mat<A>::value && is_mat<B>::value &&
  //  mat_traits<A>::rows==mat_traits<B>::rows &&
  //  mat_traits<A>::cols==mat_traits<B>::cols
  template <class A,class B>
  bool operator==( A const & a, B const & b );

} }
```

Returns:

`true` if each element of `a` compares equal to its corresponding element of `b`, `false` otherwise.

`operator!=`

#include <boost/qvm/mat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_mat<A>::value && is_mat<B>::value &&
  //  mat_traits<A>::rows==mat_traits<B>::rows &&
  //  mat_traits<A>::cols==mat_traits<B>::cols
  template <class A,class B>
  bool operator!=( A const & a, B const & b );

} }
```

Returns:

`!( a == b )`.

---

cmp

#include <boost/qvm/mat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_mat<A>::value && is_mat<B>::value &&
  //  mat_traits<A>::rows==mat_traits<B>::rows &&
  //  mat_traits<A>::cols==mat_traits<B>::cols
  template <class A,class B,class Cmp>
  bool cmp( A const & a, B const & b, Cmp pred );

} }
```

Returns:

Similar to `operator==`, except that the individual elements of `a` and `b` are passed to the binary predicate `pred` for comparison.

---

inverse

#include <boost/qvm/mat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_mat<A>::value && is_scalar<B>::value
  //  mat_traits<A>::rows==mat_traits<A>::cols

  template <class A,class B>
  typename deduce_mat2<A,B,mat_traits<A>::rows,mat_traits<A>::cols>::type
  inverse( A const & a, B det );

  template <class A>
  typename deduce_mat<A>::type
  inverse( A const & a );

} }
```

Preconditions:

`det!=0`

Returns:

Both overloads compute the inverse of `a`. The first overload takes the pre-computed determinant of `a`.

Throws:

The second overload computes the determinant automatically and throws `zero_determinant_error` if the computed determinant is zero.

The `deduce_mat` (and `deduce_mat2`) templates can be specialized to deduce the desired return type from the type `A` (and `B`).

`zero_mat`

#include <boost/qvm/mat_operations.hpp>

```
namespace boost { namespace qvm {

  template <class T,int D>
  -unspecified-return-type- zero_mat();

  template <class T,int R,int C>
  -unspecified-return-type- zero_mat();

} }
```

Returns:

A read-only matrix of unspecified type with `scalar_type` `T`, `R` rows and `C` columns (or `D` rows and `D` columns), with all elements equal to `scalar_traits<T>::value(0)`.

---

set_zero

#include <boost/qvm/mat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_mat<A>::value
  template <class A>
  void set_zero( A & a );

} }
```

Effects:

As if:

```
assign(a,
  zero_mat<
    typename mat_traits<A>::scalar_type,
    mat_traits<A>::rows,
    mat_traits<A>::cols>());
```

---

identity_mat

#include <boost/qvm/mat_operations.hpp>

```
namespace boost { namespace qvm {

  template <class S,int D>
  -unspecified-return-type- identity_mat();

} }
```

Returns:

An identity matrix of size `D` x `D` and scalar type `S`.

---

set_identity

#include <boost/qvm/mat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_mat<A>::value &&
  //  mat_traits<A>::cols==mat_traits<A>::rows
  template <class A>
  void set_identity( A & a );

} }
```

Effects:

As if:

```
assign(
  a,
  identity_mat<
    typename mat_traits<A>::scalar_type,
    mat_traits<A>::rows,
    mat_traits<A>::cols>());
```

`rot_mat` / Euler angles

#include <boost/qvm/mat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_vec<A>::value && vec_traits<A>::dim==3
  template <int Dim,class A,class Angle>
  -unspecified-return-type-
  rot_mat( A const & axis, Angle angle );

  template <int Dim,class Angle>
  -unspecified-return-type-
  rot_mat_xzy( Angle x1, Angle z2, Angle y3 );

  template <int Dim,class Angle>
  -unspecified-return-type-
  rot_mat_xyz( Angle x1, Angle y2, Angle z3 );

  template <int Dim,class Angle>
  -unspecified-return-type-
  rot_mat_yxz( Angle y1, Angle x2, Angle z3 );

  template <int Dim,class Angle>
```

```
  -unspecified-return-type-
  rot_mat_yzx( Angle y1, Angle z2, Angle x3 );

  template <int Dim,class Angle>
  -unspecified-return-type-
  rot_mat_zyx( Angle z1, Angle y2, Angle x3 );

  template <int Dim,class Angle>
  -unspecified-return-type-
  rot_mat_zxy( Angle z1, Angle x2, Angle y3 );

  template <int Dim,class Angle>
  -unspecified-return-type-
  rot_mat_xzx( Angle x1, Angle z2, Angle x3 );

  template <int Dim,class Angle>
  -unspecified-return-type-
  rot_mat_xyx( Angle x1, Angle y2, Angle x3 );

  template <int Dim,class Angle>
  -unspecified-return-type-
  rot_mat_yxy( Angle y1, Angle x2, Angle y3 );

  template <int Dim,class Angle>
  -unspecified-return-type-
  rot_mat_yzy( Angle y1, Angle z2, Angle y3 );

  template <int Dim,class Angle>
  -unspecified-return-type-
  rot_mat_zyz( Angle z1, Angle y2, Angle z3 );

  template <int Dim,class Angle>
  -unspecified-return-type-
  rot_mat_zxz( Angle z1, Angle y2, Angle z3 );

} }
```

Returns:

A matrix of unspecified type, of `Dim` rows and `Dim` columns parameter, which performs a rotation around the `axis` at `angle` radians, or Tait–Bryan angles (x-y-z, y-z-x, z-x-y, x-z-y, z-y-x, y-x-z), or proper Euler angles (z-x-z, x-y-x, y-z-y, z-y-z, x-z-x, y-x-y). See Euler angles.

Throws:

In case the axis vector has zero magnitude, throws `zero_magnitude_error`.

These functions are not view proxies; they return a temp object.

`set_rot` / Euler angles

#include <boost/qvm/mat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_mat<A>::value && mat_traits<A>::rows>=3 &&
  //  mat_traits<A>::rows==mat_traits<A>::cols &&
  //  is_vec<B>::value && vec_traits<B>::dim==3
  template <class A>
  void set_rot( A & a, B const & axis, typename vec_traits<B>::scalar_type angle );

  //Only enabled if:
  //  is_mat<A>::value && mat_traits<A>::rows>=3 &&
  //  mat_traits<A>::rows==mat_traits<A>::cols
  template <class A,class Angle>
  void set_rot_xzy( A & a, Angle x1, Angle z2, Angle y3 );

  //Only enabled if:
  //  is_mat<A>::value && mat_traits<A>::rows>=3 &&
  //  mat_traits<A>::rows==mat_traits<A>::cols
  template <class A,class Angle>
  void set_rot_xyz( A & a, Angle x1, Angle y2, Angle z3 );

  //Only enabled if:
  //  is_mat<A>::value && mat_traits<A>::rows>=3 &&
  //  mat_traits<A>::rows==mat_traits<A>::cols
  template <class A,class Angle>
  void set_rot_yxz( A & a, Angle y1, Angle x2, Angle z3 );

  //Only enabled if:
  //  is_mat<A>::value && mat_traits<A>::rows>=3 &&
  //  mat_traits<A>::rows==mat_traits<A>::cols
  template <class A,class Angle>
  void set_rot_yzx( A & a, Angle y1, Angle z2, Angle x3 );

  //Only enabled if:
  //  is_mat<A>::value && mat_traits<A>::rows>=3 &&
  //  mat_traits<A>::rows==mat_traits<A>::cols
  template <class A,class Angle>
  void set_rot_zyx( A & a, Angle z1, Angle y2, Angle x3 );

  //Only enabled if:
  //  is_mat<A>::value && mat_traits<A>::rows>=3 &&
  //  mat_traits<A>::rows==mat_traits<A>::cols
  template <class A,class Angle>
  void set_rot_zxy( A & a, Angle z1, Angle x2, Angle y3 );

  //Only enabled if:
  //  is_mat<A>::value && mat_traits<A>::rows>=3 &&
```

```
  //  mat_traits<A>::rows==mat_traits<A>::cols
  template <class A,class Angle>
  void set_rot_xzx( A & a, Angle x1, Angle z2, Angle x3 );

  //Only enabled if:
  //  is_mat<A>::value && mat_traits<A>::rows>=3 &&
  //  mat_traits<A>::rows==mat_traits<A>::cols
  template <class A,class Angle>
  void set_rot_xyx( A & a, Angle x1, Angle y2, Angle x3 );

  //Only enabled if:
  //  is_mat<A>::value && mat_traits<A>::rows>=3 &&
  //  mat_traits<A>::rows==mat_traits<A>::cols
  template <class A,class Angle>
  void set_rot_yxy( A & a, Angle y1, Angle x2, Angle y3 );

  //Only enabled if:
  //  is_mat<A>::value && mat_traits<A>::rows>=3 &&
  //  mat_traits<A>::rows==mat_traits<A>::cols
  template <class A,class Angle>
  void set_rot_yzy( A & a, Angle y1, Angle z2, Angle y3 );

  //Only enabled if:
  //  is_mat<A>::value && mat_traits<A>::rows>=3 &&
  //  mat_traits<A>::rows==mat_traits<A>::cols
  template <class A,class Angle>
  void set_rot_zyz( A & a, Angle z1, Angle y2, Angle z3 );

  //Only enabled if:
  //  is_mat<A>::value && mat_traits<A>::rows>=3 &&
  //  mat_traits<A>::rows==mat_traits<A>::cols
  template <class A,class Angle>
  void set_rot_zxz( A & a, Angle z1, Angle x2, Angle z3 );

  //Only enabled if:
  //  is_mat<A>::value && mat_traits<A>::rows>=3 &&
  //  mat_traits<A>::rows==mat_traits<A>::cols
  template <class A,class Angle>
  void set_rot_xzy( A & a, Angle x1, Angle z2, Angle y3 );

} }
```

Effects:

Assigns the return value of the corresponding `rot_mat` function to `a`.

`rotate` / Euler angles

#include <boost/qvm/mat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_mat<A>::value && mat_traits<A>::rows>=3 &&
  //  mat_traits<A>::rows==mat_traits<A>::cols &&
  //  is_vec<B>::value && vec_traits<B>::dim==3
  template <class A,class B>
  void rotate( A & a, B const & axis, typename mat_traits<A>::scalar_type angle );

  //Only enabled if:
  //  is_mat<A>::value && mat_traits<A>::rows>=3 &&
  //  mat_traits<A>::rows==mat_traits<A>::cols
  template <class A,class Angle>
  void rotate_xzy( A & a, Angle x1, Angle z2, Angle y3 );

  //Only enabled if:
  //  is_mat<A>::value && mat_traits<A>::rows>=3 &&
  //  mat_traits<A>::rows==mat_traits<A>::cols
  template <class A,class Angle>
  void rotate_xyz( A & a, Angle x1, Angle y2, Angle z3 );

  //Only enabled if:
  //  is_mat<A>::value && mat_traits<A>::rows>=3 &&
  //  mat_traits<A>::rows==mat_traits<A>::cols
  template <class A,class Angle>
  void rotate_yxz( A & a, Angle y1, Angle x2, Angle z3 );

  //Only enabled if:
  //  is_mat<A>::value && mat_traits<A>::rows>=3 &&
  //  mat_traits<A>::rows==mat_traits<A>::cols
  template <class A,class Angle>
  void rotate_yzx( A & a, Angle y1, Angle z2, Angle x3 );

  //Only enabled if:
  //  is_mat<A>::value && mat_traits<A>::rows>=3 &&
  //  mat_traits<A>::rows==mat_traits<A>::cols
  template <class A,class Angle>
  void rotate_zyx( A & a, Angle z1, Angle y2, Angle x3 );

  //Only enabled if:
  //  is_mat<A>::value && mat_traits<A>::rows>=3 &&
  //  mat_traits<A>::rows==mat_traits<A>::cols
  template <class A,class Angle>
  void rotate_zxy( A & a, Angle z1, Angle x2, Angle y3 );

  //Only enabled if:
  //  is_mat<A>::value && mat_traits<A>::rows>=3 &&
  //  mat_traits<A>::rows==mat_traits<A>::cols
  template <class A,class Angle>
  void rotate_xzx( A & a, Angle x1, Angle z2, Angle x3 );
```

```
  //Only enabled if:
  //  is_mat<A>::value && mat_traits<A>::rows>=3 &&
  //  mat_traits<A>::rows==mat_traits<A>::cols
  template <class A,class Angle>
  void rotate_xyx( A & a, Angle x1, Angle y2, Angle x3 );

  //Only enabled if:
  //  is_mat<A>::value && mat_traits<A>::rows>=3 &&
  //  mat_traits<A>::rows==mat_traits<A>::cols
  template <class A,class Angle>
  void rotate_yxy( A & a, Angle y1, Angle x2, Angle y3 );

  //Only enabled if:
  //  is_mat<A>::value && mat_traits<A>::rows>=3 &&
  //  mat_traits<A>::rows==mat_traits<A>::cols
  template <class A,class Angle>
  void rotate_yzy( A & a, Angle y1, Angle z2, Angle y3 );

  //Only enabled if:
  //  is_mat<A>::value && mat_traits<A>::rows>=3 &&
  //  mat_traits<A>::rows==mat_traits<A>::cols
  template <class A,class Angle>
  void rotate_zyz( A & a, Angle z1, Angle y2, Angle z3 );

  //Only enabled if:
  //  is_mat<A>::value && mat_traits<A>::rows>=3 &&
  //  mat_traits<A>::rows==mat_traits<A>::cols
  template <class A,class Angle>
  void rotate_zxz( A & a, Angle z1, Angle x2, Angle z3 );

} }
```

Effects:

Multiplies the matrix `a` in-place by the return value of the corresponding `rot_mat` function.

`rotx_mat`

#include <boost/qvm/mat_operations.hpp>

```
namespace boost { namespace qvm {

  template <int Dim,class Angle>
  -unspecified-return-type- rotx_mat( Angle const & angle );

} }
```

Returns:

A view proxy matrix of unspecified type, of `Dim` rows and `Dim` columns and scalar type `Angle`,

which performs a rotation around the `X` axis at `angle` radians.

set_rotx

#include <boost/qvm/mat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_mat<A>::value && mat_traits<A>::rows>=3 &&
  //  mat_traits<A>::rows==mat_traits<A>::cols
  template <class A>
  void set_rotx( A & a, typename mat_traits<A>::scalar_type angle );

} }
```

Effects:

As if:

```
assign(
  a,
  rotx_mat<mat_traits<A>::rows>(angle));
```

rotate_x

#include <boost/qvm/mat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_mat<A>::value && mat_traits<A>::rows>=3 &&
  //  mat_traits<A>::rows==mat_traits<A>::cols
  template <class A>
  void rotate_x( A & a, typename mat_traits<A>::scalar_type angle );

} }
```

Effects:

As if: `a *= rotx_mat<mat_traits<A>::rows>(angle)`.

roty_mat

#include <boost/qvm/mat_operations.hpp>

```
namespace boost { namespace qvm {

  template <int Dim,class Angle>
  -unspecified-return-type- roty_mat( Angle const & angle );

} }
```

Returns:

A view proxy matrix of unspecified type, of `Dim` rows and `Dim` columns and scalar type `Angle`, which performs a rotation around the `Y` axis at `angle` radians.

`set_roty`

#include <boost/qvm/mat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_mat<A>::value && mat_traits<A>::rows>=3 &&
  //  mat_traits<A>::rows==mat_traits<A>::cols
  template <class A>
  void set_roty( A & a, typename mat_traits<A>::scalar_type angle );

} }
```

Effects:

As if:

```
assign(
  a,
  roty_mat<mat_traits<A>::rows>(angle));
```

`rotate_y`

#include <boost/qvm/mat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_mat<A>::value && mat_traits<A>::rows>=3 &&
  //  mat_traits<A>::rows==mat_traits<A>::cols
  template <class A>
  void rotate_y( A & a, typename mat_traits<A>::scalar_type angle );

} }
```

Effects:

As if: `a *= roty_mat<mat_traits<A>::rows>(angle)`.

`rotz_mat`

#include <boost/qvm/mat_operations.hpp>

```
namespace boost { namespace qvm {

  template <int Dim,class Angle>
  -unspecified-return-type- rotz_mat( Angle const & angle );

} }
```

Returns:

A view proxy matrix of unspecified type, of `Dim` rows and `Dim` columns and scalar type `Angle`, which performs a rotation around the `Z` axis at `angle` radians.

`set_rotz`

#include <boost/qvm/mat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_mat<A>::value && mat_traits<A>::rows>=3 &&
  //  mat_traits<A>::rows==mat_traits<A>::cols
  template <class A>
  void set_rotz( A & a, typename mat_traits<A>::scalar_type angle );

} }
```

Effects:

As if:

```
assign(
  a,
  rotz_mat<mat_traits<A>::rows>(angle));
```

rotate_z

#include <boost/qvm/mat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_mat<A>::value && mat_traits<A>::rows>=3 &&
  //  mat_traits<A>::rows==mat_traits<A>::cols
  template <class A>
  void rotate_z( A & a, typename mat_traits<A>::scalar_type angle );

} }
```

Effects:

As if: `a *= rotz_mat<mat_traits<A>::rows>(angle)`.

determinant

#include <boost/qvm/mat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_mat<A>::value && mat_traits<A>::rows==mat_traits<A>::cols
  template <class A>
  mat_traits<A>::scalar_type
  determinant( A const & a );

} }
```

This function computes the determinant of the square matrix `a`.

perspective_lh

#include <boost/qvm/mat_operations.hpp>

```
namespace boost { namespace qvm {

  template <class T>
  -unspecified-return-type-
  perspective_lh( T fov_y, T aspect, T zn, T zf );

} }
```

Returns:

A 4x4 projection matrix of unspecified type of the following form:

xs	0	0	0
0	ys	0	0
0	0	zf/(zf-zn)	-zn*zf/(zf-zn)
0	0	1	0

where ys = cot(fov_y/2) and xs = ys/aspect.

perspective_rh

#include <boost/qvm/mat_operations.hpp>

```
namespace boost { namespace qvm {

  template <class T>
  -unspecified-return-type-
  perspective_rh( T fov_y, T aspect, T zn, T zf );

} }
```

Returns:

A 4x4 projection matrix of unspecified type of the following form:

xs	0	0	0
0	ys	0	0
0	0	zf/(zn-zf)	zn*zf/(zn-zf)
0	0	-1	0

where ys = cot(fov_y/2), and xs = ys/aspect.

scalar_cast

#include <boost/qvm/mat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if: is_mat<A>::value
  template <class Scalar,class A>
  -unspecified-return_type- scalar_cast( A const & a );

} }
```

Returns:

A read-only view proxy of `a` that looks like a matrix of the same dimensions as `a`, but with `scalar_type` `Scalar` and elements constructed from the corresponding elements of `a`.

---

mref

#include <boost/qvm/mat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if: is_mat<A>::value
  template <class A>
  -unspecified-return-type- mref( A & a );

} }
```

Returns:

An identity view proxy of `a`; that is, it simply accesses the elements of `a`.

`mref` allows calling QVM operations when `a` is of built-in type, for example a plain old C array.

Quaternion-Vector Operations

operator*

#include <boost/qvm/quat_vec_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_quat<A>::value && is_vec<B>::value &&
  //  is_vec<B>::value && vec_traits<B>::dim==3
  template <class A,class B>
  typename deduce_vec2<A,B,mat_traits<A>::rows>::type
  operator*( A const & a, B const & b );

} }
```

Returns:

The result of transforming the vector `b` by the quaternion `a`.

The deduce_vec2 template can be specialized to deduce the desired return type, given the types `A` and `B`.

Matrix-Vector Operations

`operator*`

#include <boost/qvm/vec_mat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_mat<A>::value && is_vec<B>::value &&
  //  mat_traits<A>::cols==vec_traits<B>::dim
  template <class A,class B>
  typename deduce_vec2<A,B,mat_traits<A>::rows>::type
  operator*( A const & a, B const & b );

} }
```

Returns:

The result of multiplying the matrix `a` and the vector `b`, where `b` is interpreted as a matrix-column. The resulting matrix-row is returned as a vector type.

The deduce_vec2 template can be specialized to deduce the desired return type, given the types `A` and `B`.

transform_vector

#include <boost/qvm/vec_mat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_mat<A>::value && is_vec<B>::value &&
  //  mat_traits<A>::rows==4 && mat_traits<A>::cols==4 &&
  //  vec_traits<B>::dim==3
  template <class A,class B>
  deduce_vec2<A,B,3> >::type
  transform_vector( A const & a, B const & b );

} }
```

Effects:

As if: `return a * XYZ0(b)`.

---

transform_point

#include <boost/qvm/vec_mat_operations.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_mat<A>::value && is_vec<B>::value &&
  //  mat_traits<A>::rows==4 && mat_traits<A>::cols==4 &&
  //  vec_traits<B>::dim==3
  template <class A,class B>
  deduce_vec2<A,B,3> >::type
  transform_point( A const & a, B const & b );

} }
```

Effects:

As if: `return a * XYZ1(b)`.

Matrix-to-Matrix View Proxies

del_row

#include <boost/qvm/map_mat_mat.hpp>

```
namespace boost { namespace qvm {

  template <int R>
  -unspecified-return-type- del_row();

} }
```

The expression `del_row<R>(m)` returns an lvalue view proxy that looks like the matrix `m` with row `R` deleted.

---

del_col

#include <boost/qvm/map_mat_mat.hpp>

```
namespace boost { namespace qvm {

  template <int C>
  -unspecified-return-type- del_col();

} }
```

The expression `del_col<C>(m)` returns an lvalue view proxy that looks like the matrix `m` with column `C` deleted.

---

del_row_col

#include <boost/qvm/map_mat_mat.hpp>

```
namespace boost { namespace qvm {

  template <int R,int C>
  -unspecified-return-type- del_row_col();

} }
```

The expression `del_row_col<R,C>(m)` returns an lvalue view proxy that looks like the matrix `m` with row `R` and column `C` deleted.

---

neg_row

#include <boost/qvm/map_mat_mat.hpp>

```
namespace boost { namespace qvm {

  template <int R>
  -unspecified-return-type- neg_row();

} }
```

The expression `neg_row<R>(m)` returns a read-only view proxy that looks like the matrix `m` with row `R` negated.

---

neg_col

#include <boost/qvm/map_mat_mat.hpp>

```
namespace boost { namespace qvm {

  template <int C>
  -unspecified-return-type- neg_col();

} }
```

```
The expression `neg_col<C>(m)` returns a read-only <<view_proxy,`view proxy`>> that
looks like the matrix `m` with column `C` negated.
```

---

swap_rows

#include <boost/qvm/map_mat_mat.hpp>

```
namespace boost { namespace qvm {

  template <int R1,int R2>
  -unspecified-return-type- swap_rows();

} }
```

The expression `swap_rows<R1,R2>(m)` returns an lvalue view proxy that looks like the matrix `m` with rows `R1` and `R2` swapped.

---

swap_cols

#include <boost/qvm/map_mat_mat.hpp>

```
namespace boost { namespace qvm {

  template <int C1,int C2>
  -unspecified-return-type- swap_cols();

} }
```

The expression `swap_cols<C1,C2>(m)` returns an lvalue view proxy that looks like the matrix `m` with columns `C1` and `C2` swapped.

`transposed`

#include <boost/qvm/map_mat_mat.hpp>

```
namespace boost { namespace qvm {

  -unspecified-return-type- transposed();

} }
```

The expression `transposed(m)` returns an lvalue view proxy that transposes the matrix `m`.

Vector-to-Matrix View Proxies

`col_mat`

#include <boost/qvm/map_vec_mat.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if: is_vec<A>::value
  template <iclass A>
  -unspecified-return-type- col_mat( A & a );

} }
```

The expression `col_mat(v)` returns an lvalue view proxy that accesses the vector `v` as a matrix-column.

`row_mat`

#include <boost/qvm/map_vec_mat.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if: is_vec<A>::value
  template <iclass A>
  -unspecified-return-type- row_mat( A & a );

} }
```

The expression `row_mat(v)` returns an lvalue view proxy that accesses the vector `v` as a matrix-row.

`translation_mat`

#include <boost/qvm/map_vec_mat.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if: is_vec<A>::value
  template <iclass A>
  -unspecified-return-type- translation_mat( A & a );

} }
```

The expression `translation_mat(v)` returns an lvalue view proxy that accesses the vector `v` as translation matrix of size 1 + `vec_traits<A>::dim`.

`diag_mat`

#include <boost/qvm/map_vec_mat.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if: is_vec<A>::value
  template <iclass A>
  -unspecified-return-type- diag_mat( A & a );

} }
```

The expression `diag_mat(v)` returns an lvalue view proxy that accesses the vector `v` as a square matrix of the same dimensions in which the elements of `v` appear as the main diagonal and all other elements are zero.

If `v` is a 3D vector, the expression `diag_mat(XYZ1(v))` can be used as a scaling 4D matrix.

Matrix-to-Vector View Proxies

`col`

#include <boost/qvm/map_mat_vec.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if: is_mat<A>::value
  template <int C,class A>
  -unspecified-return-type- col( A & a );

} }
```

The expression `col<C>(m)` returns an lvalue view proxy that accesses column `C` of the matrix `m` as a vector.

`row`

#include <boost/qvm/map_mat_vec.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if: is_mat<A>::value
  template <int C,class A>
  -unspecified-return-type- row( A & a );

} }
```

The expression `row<R>(m)` returns an lvalue view proxy that accesses row `R` of the matrix `m` as a vector.

`diag`

#include <boost/qvm/map_mat_vec.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if: is_mat<A>::value
  template <class A>
  -unspecified-return-type- diag( A & a );

} }
```

The expression `diag(m)` returns an lvalue view proxy that accesses the main diagonal of the matrix `m` as a vector.

translation

#include <boost/qvm/map_mat_vec.hpp>

```
namespace boost { namespace qvm {

  //Only enabled if:
  //  is_mat<A>::value &&
  //  mat_traits<A>::rows==mat_traits<A>::cols && mat_traits<A>::rows>=3
  template <class A>
  -unspecified-return-type- translation( A & a );

} }
```

The expression `translation(m)` returns an lvalue view proxy that accesses the translation component of the square matrix `m`, which is a vector of size `D`-1, where `D` is the size of `m`.

Exceptions

error

#include <boost/qvm/error.hpp>

```
namespace boost { namespace qvm {

  struct error: virtual boost::exception, virtual std::exception { };

} }
```

This is the base for all exceptions thorwn by QVM.

zero_magnitude_error

#include <boost/qvm/error.hpp>

```
namespace boost { namespace qvm {

  struct zero_magnitude_error: virtual error { };

} }
```

This exception indicates that an operation requires a vector or a quaternion with non-zero magnitude, but the computed magnitude is zero.

zero_determinant_error

#include <boost/qvm/error.hpp>

```
namespace boost { namespace qvm {

  struct zero_determinant_error: virtual error { };

} }
```

This exception indicates that an operation requires a matrix with non-zero determinant, but the computed determinant is zero.

Macros and Configuration: BOOST_QVM_

INLINE

`BOOST_QVM_INLINE`

#include <boost/qvm/inline.hpp>

```
namespace boost { namespace qvm {

  #ifndef BOOST_QVM_INLINE
  #define BOOST_QVM_INLINE inline
  #endif

} }
```

This macro is not used directly by QVM, except as the default value of other macros from `<boost/qvm/inline.hpp>`. A user-defined `BOOST_QVM_INLINE` should expand to a value that is valid substitution of the `inline` keyword in function definitions.

FORCE_INLINE

`BOOST_QVM_FORCE_INLINE`

#include <boost/qvm/inline.hpp>

```
namespace boost { namespace qvm {

  #ifndef BOOST_QVM_FORCE_INLINE
  #define BOOST_QVM_FORCE_INLINE /*platform-specific*/
  #endif

} }
```

This macro is not used directly by QVM, except as the default value of other macros from `<boost/qvm/inline.hpp>`. A user-defined `BOOST_QVM_FORCE_INLINE` should expand to a value that is valid substitution of the `inline` keyword in function definitions, to indicate that the compiler must inline the function. Of course, actual inlining may or may not occur.

INLINE_TRIVIAL

`BOOST_QVM_INLINE_TRIVIAL`

#include <boost/qvm/inline.hpp>

```
namespace boost { namespace qvm {

  #ifndef BOOST_QVM_INLINE_TRIVIAL
  #define BOOST_QVM_INLINE_TRIVIAL BOOST_QVM_FORCE_INLINE
  #endif

} }
```

QVM uses `BOOST_QVM_INLINE_TRIVIAL` in definitions of functions that are not critical for the overall performance of the library but are extremely simple (such as one-liners) and therefore should always be inlined.

INLINE_CRITICAL

`BOOST_QVM_INLINE_CRITICAL`

#include <boost/qvm/inline.hpp>

```
namespace boost { namespace qvm {

  #ifndef BOOST_QVM_INLINE_CRITICAL
  #define BOOST_QVM_INLINE_CRITICAL BOOST_QVM_FORCE_INLINE
  #endif

} }
```

QVM uses `BOOST_QVM_INLINE_CRITICAL` in definitions of functions that are critical for the overall performance of the library, such as functions that access individual vector and matrix elements.

INLINE_OPERATIONS

`BOOST_QVM_INLINE_OPERATIONS`

#include <boost/qvm/inline.hpp>

```
namespace boost { namespace qvm {

  #ifndef BOOST_QVM_INLINE_OPERATIONS
  #define BOOST_QVM_INLINE_OPERATIONS BOOST_QVM_INLINE
  #endif

} }
```

QVM uses `BOOST_QVM_INLINE_OPERATIONS` in definitions of functions that implement various high-level operations, such as matrix multiplication, computing the magnitude of a vector, etc.

INLINE_RECURSION

`BOOST_QVM_INLINE_RECURSION`

#include <boost/qvm/inline.hpp>

```
namespace boost { namespace qvm {

  #ifndef BOOST_QVM_INLINE_RECURSION
  #define BOOST_QVM_INLINE_RECURSION BOOST_QVM_INLINE_OPERATIONS
  #endif

} }
```

QVM uses `BOOST_QVM_INLINE_RECURSION` in definitions of recursive functions that are not critical for the overall performance of the library (definitions of all critical functions, including critical recursive functions, use `BOOST_QVM_INLINE_CRITICAL`).

ASSERT

`BOOST_QVM_ASSERT`

#include <boost/qvm/assert.hpp>

```
namespace boost { namespace qvm {

#ifndef BOOST_QVM_ASSERT
#include <boost/assert.hpp>
#define BOOST_QVM_ASSERT BOOST_ASSERT
#endif

} }
```

This is the macro QVM uses to assert on precondition violations and logic errors. A user-defined

`BOOST_QVM_ASSERT` should have the semantics of the standard `assert`.

STATIC_ASSERT

`BOOST_QVM_STATIC_ASSERT`

#include <boost/qvm/static_assert.hpp>

```
namespace boost { namespace qvm {

  #ifndef BOOST_QVM_STATIC_ASSERT
  #include <boost/static_assert.hpp>
  #define BOOST_QVM_STATIC_ASSERT BOOST_STATIC_ASSERT
  #endif

} }
```

All static assertions in QVM use the `BOOST_QVM_STATIC_ASSERT` macro.

THROW_EXCEPTION

`BOOST_QVM_THROW_EXCEPTION`

#include <boost/qvm/throw_exception.hpp>

```
namespace boost { namespace qvm {

  #ifndef BOOST_QVM_THROW_EXCEPTION
  #include <boost/throw_exception.hpp>
  #define BOOST_QVM_THROW_EXCEPTION BOOST_THROW_EXCEPTION
  #endif

} }
```

This macro is used whenever QVM throws an exception. Users who override the standard `BOOST_QVM_THROW_EXCEPTION` behavior must ensure that when invoked, the substituted implementation does not return control to the caller. Below is a list of all QVM functions that invoke `BOOST_QVM_THROW_EXCEPTION`:

- Quaternion operations:
  - `inverse`
  - `rot_quat`
  - `normalize`
  - `normalized`
- Vector operations:

    - `normalize`
    - `normalized`
- Matrix operations:
    - `inverse`
    - `rot_mat`

Design Rationale

C++ is ideal for 3D graphics and other domains that require 3D transformations: define vector and matrix types and then overload the appropriate operators to implement the standard algebraic operations. Because this is relatively straight-forward, there are many libraries that do this, each providing custom vector and matrix types, and then defining the same operations (e.g. matrix multiply) for these types.

Often these libraries are part of a higher level system. For example, video game programmers typically use one set of vector/matrix types with the rendering engine, and another with the physics simulation engine.

QVM proides interoperability between all these different types and APIs by decoupling the standard algebraic functions from the types they operate on — without compromising type safety. The operations work on any type for which proper traits have been specialized. Using QVM, there is no need to translate between the different quaternion, vector or matrix types; they can be mixed in the same expression safely and efficiently.

This design enables QVM to generate types and adaptors at compile time, compatible with any other QVM or user-defined type. For example, transposing a matrix needs not store the result: rather than modifying its argument or returning a new object, it simply binds the original matrix object through a generated type which remaps element access on the fly.

In addition, QVM can be helpful in selectively optimizing individual types or operations for maximum performance where that matters. For example, users can overload a specific operation for specific types, or define highly optimized, possibly platform-specific or for some reason cumbersome to use types, then mix and match them with more user-friendly types in parts of the program where performance isn’t critical.

Code Generator

While QVM defines generic functions that operate on matrix and vector types of arbitrary static dimensions, it also provides a code generator that can be used to create compatible header files that define much simpler specializations of these functions for specific dimensions. This is useful during debugging since the generated code is much easier to read than the template metaprogramming-heavy generic implementations. It is also potentially friendlier to the optimizer.

The code generator is a command-line utility program. Its source code can be found in the `boost/libs/qvm/gen` directory. It was used to generate the following headers that ship with QVM:

- 2D, 3D and 4D matrix operations:
  - `boost/qvm/gen/mat_operations2.hpp` (matrices of size 2x2, 2x1 and 1x2, included by `boost/qvm/mat_operations2.hpp`)
  - `boost/qvm/gen/mat_operations3.hpp` (matrices of size 3x3, 3x1 and 1x3, included by `boost/qvm/mat_operations3.hpp`)
  - `boost/qvm/gen/mat_operations4.hpp` (matrices of size 4x4, 4x1 and 1x4, included by `boost/qvm/mat_operations4.hpp`)
- 2D, 3D and 4D vector operations:
  - `boost/qvm/gen/v2.hpp` (included by `boost/qvm/vec_operations2.hpp`)
  - `boost/qvm/gen/v3.hpp` (included by `boost/qvm/vec_operations3.hpp`)
  - `boost/qvm/gen/v4.hpp` (included by `boost/qvm/vec_operations4.hpp`)
- 2D, 3D and 4D vector-matrix operations:
  - `boost/qvm/gen/vm2.hpp` (included by `boost/qvm/vec_mat_operations2.hpp`)
  - `boost/qvm/gen/vm3.hpp` (included by `boost/qvm/vec_mat_operations3.hpp`)
  - `boost/qvm/gen/vm4.hpp` (included by `boost/qvm/vec_mat_operations4.hpp`)
- 2D, 3D and 4D vector swizzling operations:
  - `boost/qvm/gen/sw2.hpp` (included by `boost/qvm/swizzle2.hpp`)
  - `boost/qvm/gen/sw3.hpp` (included by `boost/qvm/swizzle3.hpp`)
  - `boost/qvm/gen/sw4.hpp` (included by `boost/qvm/swizzle4.hpp`)

Any such generated headers must be included before the corresponding generic header file is included. For example, if one creates a header `boost/qvm/gen/m5.hpp`, it must be included before `boost/qvm/mat_operations.hpp` in included. However, the generic headers (`boost/qvm/mat_operations.hpp`, `boost/qvm/vec_operations.hpp`, `boost/qvm/vec_mat_operations.hpp` and `boost/qvm/swizzle.hpp`) already include the generated headers from the list above, so the generated headers don't need to be included manually.

headers under `boost/qvm/gen` are not part of the public interface of QVM. For example, `boost/qvm/gen/mat_operations2.hpp` should not be included directly; `#include <boost/qvm/mat_operations2.hpp>` instead.

Known Quirks and Issues

Capturing View Proxies with `auto`

By design, view proxies must not return temporary objects. They return reference to an argument they take by (`const`) reference, cast to reference of unspecified type that is not copyable. Because of this, the return value of a view proxy can not be captured by value with `auto`:

```
auto tr = transposed(m); //Error: the return type of transposed can not be copied.
```

The correct use of auto with view proxies is:

```
auto & tr = transposed(m);
```

Many view proxies are not read-only, that is, they're lvalues; changes made on the view proxy operate on the original object. This is another reason why they can not be captured by value with `auto`.

Binding QVM Overloads From an Unrelated Namespace

The operator overloads in namespace `boost::qvm` are designed to work with user-defined types. Typically it is sufficient to make these operators available in the namespace where the operator is used, by `using namespace boost::qvm`. A problem arises if the scope that uses the operator is not controlled by the user. For example:

```
namespace ns1 {

  struct float2 { float x, y; };

}

namespace ns2 {

  using namespace boost::qvm;

  void f() {
    ns1::float2 a, b;
    a==b; //OK
    ns1::float2 arr1[2], arr2[2];
    std::equal(arr1,arr1+2,arr2); //Error: operator== is inaccessible from namespace
std
  }

}
```

In the `std::equal` expression above, even though `boost::qvm::operator==` is made visible in namespace `ns2` by `using namespace boost::qvm`, the call originates from namespace `std`. In this case the compiler can't bind `boost::qvm::operator==` because only namespace `ns1` is visible through ADL, and it does not contain a suitable declaration. The solution is to declare `operator==` in namespace ns1, which can be done like this:

```
namespace ns1 {

  using boost::qvm::operator==;

}
```

Link Errors When Calling Math Functions with `int` Arguments

QVM does not call standard math functions (e.g. sin, cos, etc.) directly. Instead, it calls function templates declared in `boost/qvm/math.hpp` in namespace `boost::qvm`. This allows the user to specialize these templates for user-defined scalar types.

QVM itself defines specializations of the math function templates only for `float` and `double`, but it does not provide generic definitions. This is done to protect the user from unintentionally writing code that binds standard math functions that take `double` when passing arguments of lesser types, which would be suboptimal.

Because of this, a call to e.g. `rot_mat(axis,1)` will compile successfully but fail to link, since it

calls e.g. `boost::qvm::sin<int>`, which is undefined. Because rotations by integer number of radians are rarely needed, in QVM there is no protection against such errors. In such cases the solution is to use `rot_mat(axis,1.0f)` instead.

Distribution

QVM is part of Boost and is distributed under the Boost Software License, Version 1.0.

The source code is available in the QVM GitHub repository.

In addition, Boost QVM is available in single-header format for maximum portability. There are two versions of this distribution (the links below point to the actual header files for direct download):

- qvm.hpp: single header containing all QVM code, including complete swizzling overloads.
- qvm_lite.hpp: single header containing everything except for the swizzling overloads.

Portability

Boost QVM requires only C++03 but is tested on many compiler versions and C++ standards.

Feedback / Support

Please use the Boost Developers mailing list.

Q&A

1. *What is the motivation behind QVM? Why not just use uBLAS/Eigen/CML/GLM/etc?*

   The primary domain of QVM is realtime graphics and simulation applications, so it is not a complete linear algebra library. While (naturally) there is some overlap with such libraries, QVM puts the emphasis on 2, 3 and 4 dimensional zero-overhead operations (hence domain-specific features like Swizzling).

2. *How does the `qvm::vec` (or `qvm::mat`, or `qvm::quat`) template compare to vector types from other libraries?*

   The `qvm::vec` template is not in any way central to the vector operations defined by QVM. The operations are designed to work with any user-defined vector type or with 3rd-party vector types (e.g. `D3DVECTOR`), while the `qvm::vec` template is simply a default return type for expressions that use arguments of different types that would be incompatible outside of QVM. For example, if the `deduce_mat2` hasn't been specialized, calling `cross` with a user-defined type `vec3` and a user-defined type `float3` returns a `qvm::vec`.

3. *Why doesn't QVM use [] or () to access vector and matrix elements?*

   Because it's designed to work with user-defined types, and the C++ standard requires these operators to be members. Of course if a user-defined type defines `operator[]` or `operator()` they are available for use with other QVM functions, but QVM defines its own mechanisms for accessing quaternion elements, accessing vector elements (as well as swizzling), and accessing matrix elements.
