Space transformations

Geometry

In this page, we will introduce the many possibilities offered by the **geometry module** to deal with 2D and 3D rotations and projective or affine transformations.

Eigen's Geometry module provides two different kinds of geometric transformations:

- Abstract transformations, such as rotations (represented by **angle and axis** or by a **quaternion**), **translations**, **scalings**. These transformations are NOT represented as matrices, but you can nevertheless mix them with matrices and vectors in expressions, and convert them to matrices if you wish.
- Projective or affine transformation matrices: see the **Transform** class. These are really matrices.

Note

If you are working with OpenGL 4x4 matrices then Affine3f and Affine3d are what you want. Since **Eigen** defaults to column-major storage, you can directly use the **Transform::data()** method to pass your transformation matrix to OpenGL.

You can construct a **Transform** from an abstract transformation, like this:

```
Transform t(AngleAxis(angle,axis));
```

or like this:

```
Transform t;
t = AngleAxis(angle,axis);
```

But note that unfortunately, because of how C++ works, you can **not** do this:

```
Transform t = AngleAxis(angle,axis);
```

Explanation: In the C++ language, this would require **Transform** to have a non-explicit conversion constructor from **AngleAxis**, but we really don't want to allow implicit casting here.

Transformation types

Transformation type	Typical initialization code
2D rotation from an angle	`Rotation2D<float> rot2(angle_in_radian);`
3D rotation as an **angle + axis**	`AngleAxis<float> aa(angle_in_radian, Vector3f(ax,ay,az));` The axis vector must be normalized.
3D rotation as a **quaternion**	`Quaternion<float> q;  q = AngleAxis<float>(angle_in_radian, axis);`

N-D **Scaling**	`Scaling(sx, sy)`
	`Scaling(sx, sy, sz)`
	`Scaling(s)`
	`Scaling(vecN)`

N-D **Translation**	`Translation<float,2>(tx, ty)` `Translation<float,3>(tx, ty, tz)`

	`Translation<float,N>(s)` `Translation<float,N>(vecN)`
N-D **Affine transformation**	`Transform<float,N,Affine> t = concatenation_of_any_transformations;` `Transform<float,3,Affine> t = Translation3f(p) * AngleAxisf(a,axis) * Scaling(s);`

N-D Linear transformations *(pure rotations, scaling, etc.)*	`Matrix<float,N> t = concatenation_of_rotations_and_scalings;`
	`Matrix<float,2> t = Rotation2Df(a) * Scaling(s);`
	`Matrix<float,3> t = AngleAxisf(a,axis) * Scaling(s);`

Notes on rotations

To transform more than a single vector the preferred representations are rotation matrices, while for other usages **Quaternion** is the representation of choice as they are compact, fast and stable. Finally **Rotation2D** and **AngleAxis** are mainly convenient types to create other rotation objects.

Notes on Translation and Scaling

Like **AngleAxis**, these classes were designed to simplify the creation/initialization of linear (**Matrix**) and affine (**Transform**) transformations. Nevertheless, unlike **AngleAxis** which is inefficient to use, these classes might still be interesting to write generic and efficient algorithms taking as input any kind of transformations.

Any of the above transformation types can be converted to any other types of the same nature, or to a more generic type. Here are some additional examples:

```
Rotation2Df r;  r  = Matrix2f(..);       // assumes a pure rotation matrix
AngleAxisf aa;  aa = Quaternionf(..);
AngleAxisf aa;  aa = Matrix3f(..);       // assumes a pure rotation matrix
Matrix2f m;     m  = Rotation2Df(..);
Matrix3f m;     m  = Quaternionf(..);       Matrix3f m;   m = Scaling(..);
Affine3f m;     m  = AngleAxis3f(..);       Affine3f m;   m = Scaling(..);
Affine3f m;     m  = Translation3f(..);     Affine3f m;   m = Matrix3f(..);
```

top

Common API across transformation types

To some extent, **Eigen**'s **geometry module** allows you to write generic algorithms working on any kind of transformation representations:

Concatenation of two transformations	`gen1 * gen2;`
Apply the transformation to a vector	`vec2 = gen1 * vec1;`
Get the inverse of the transformation	`gen2 = gen1.inverse();`
Spherical interpolation (**Rotation2D** and **Quaternion** only)	`rot3 = rot1.slerp(alpha,rot2);`

top

Affine transformations

Generic affine transformations are represented by the **Transform** class which internaly is a (Dim+1)^2 matrix. In **Eigen** we have chosen to not distinghish between points and vectors such that all points are actually represented

by displacement vectors from the origin ($\mathbf{p} \equiv \mathbf{p} - 0$). With that in mind, real points and vector distinguish when the transformation is applied.

Apply the transformation to a **point**	`VectorNf p1, p2;` `p2 = t * p1;`
Apply the transformation to a **vector**	`VectorNf vec1, vec2;` `vec2 = t.linear() * vec1;`

Apply a *general* transformation to a **normal vector**	`VectorNf n1, n2;`
	`MatrixNf normalMatrix = t.linear().inverse().transpose();`
	`n2 = (normalMatrix * n1).normalized();`

(See subject 5.27 of this faq for the explanations)	
Apply a transformation with *pure rotation* to a **normal vector** (no scaling, no shear)	`n2 = t.linear() * n1;`
OpenGL compatibility **3D**	`glLoadMatrixf(t.data());`

OpenGL compatibility **2D**	`Affine3f aux(Affine3f::Identity());`
	`aux.linear().topLeftCorner<2,2>() = t.linear();`
	`aux.translation().start<2>() = t.translation();`
	`glLoadMatrixf(aux.data());`

Component accessors

full read-write access to the internal matrix	`t.matrix() = matN1xN1;  // N1 means N+1`
	`matN1xN1 = t.matrix();`

coefficient accessors	`t(i,j) = scalar;  <=>  t.matrix()(i,j) = scalar;` `scalar = t(i,j);  <=>  scalar = t.matrix()(i,j);`
translation part	`t.translation() = vecN;` `vecN = t.translation();`
linear part	`t.linear() = matNxN;` `matNxN = t.linear();`
extract the rotation matrix	`matNxN = t.rotation();`

Transformation creation

While transformation objects can be created and updated concatenating elementary transformations, the **Transform** class also features a procedural API:

	procedural API	equivalent natural API
Translation	`t.translate(Vector_(tx,ty,..));` `t.pretranslate(Vector_(tx,ty,..));`	`t *= Translation_(tx,ty,..);` `t = Translation_(tx,ty,..) * t;`
Rotation *In 2D and for the procedural API, any_rotation can also be an angle in radian*	`t.rotate(any_rotation);` `t.prerotate(any_rotation);`	`t *= any_rotation;` `t = any_rotation * t;`

Scaling	`t.scale(Vector_(sx,sy,..));`
	`t.scale(s);`
	`t.prescale(Vector_(sx,sy,..));`
	`t.prescale(s);`
		`t *= Scaling(sx,sy,..);`
		`t *= Scaling(s);`
		`t = Scaling(sx,sy,..) * t;`
		`t = Scaling(s) * t;`

Shear transformation (**2D only** !)	`t.shear(sx,sy);` `t.preshear(sx,sy);`	

Note that in both API, any many transformations can be concatenated in a single expression as shown in the two following equivalent examples:

```
t.pretranslate(..).rotate(..).translate(..).scale(..);
t = Translation_(..) * t * RotationType(..) * Translation_(..) * Scaling(..);
```

top

Euler angles

Euler angles might be convenient to create rotation objects. On the other hand, since there exist 24 different conventions, they are pretty confusing to use. This example shows how to create a rotation matrix according to the 2-1-2 convention.

```
Matrix3f m;
m = AngleAxisf(angle1, Vector3f::UnitZ())
  * AngleAxisf(angle2, Vector3f::UnitY())
  * AngleAxisf(angle3,
       Vector3f::UnitZ());
```