空间变换

几何学

在本页中，我们将介绍几何模块提供的许多可能性，以处理 2D 和 3D 旋转以及投影或仿射变换。

Eigen的 Geometry 模块提供了两种不同的几何变换：

- 抽象变换，例如旋转（由角度和轴或由四元数表示）、平移、缩放。这些转换不表示为矩阵，但您仍然可以将它们与表达式中的矩阵和向量混合，并根据需要将它们转换为矩阵。
- 投影或仿射变换矩阵：请参阅Transform类。这些真的是矩阵。

> 笔记
>
> 如果您使用 OpenGL 4x4 矩阵，那么 Affine3f 和 Affine3d 就是您想要的。由于Eigen默认为列优先存储，因此您可以直接使用Transform::data()方法将转换矩阵传递给 OpenGL。

您可以构建一个转换，从一个抽象的转变，就像这样：

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

或者像这样：

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

但请注意，不幸的是，由于 C++ 的工作方式，您**不能**这样做：

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

说明：在 C++ 语言中，这将要求Transform具有来自AngleAxis的非显式转换构造函数，但我们真的不想在此处允许隐式转换。

转换类型

改造类型	典型的初始化代码
从某个角度进行2D 旋转	Rotation2D<float> rot2(angle_in_radian);
3D 旋转为角度 + 轴	AngleAxis<float> aa(angle_in_radian, Vector3f(ax,ay,az)); // 轴向量必须归一化。
作为四元数的3D 旋转	Quaternion<float> q; q = AngleAxis<float>(angle_in_radian,axis);
ND 缩放	Scaling（sx，sy） Scaling（sx、sy、sz） Scaling (s) Scaling (vecN)
ND变换	Translation<float,2>(tx, ty) Translation<float,3>(tx, ty, tz) Translation <float,N>(s) Translation<float,N>(vecN)
ND仿射变换	Transform<float,N,Affine> t = concatenation_of_any_transformations; Transform<float,3,Affine> t = Translation3f(p) * AngleAxisf (a,axis) * Scaling (s);
ND 线性变换 *（纯旋转、缩放等）*	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);

旋转注意事项
要变换多个向量，首选表示是旋转矩阵，而对于其他用途，四元数是首选表示，因为它们紧凑、快速且稳定。最后Rotation2D和AngleAxis主要是创建其他旋转对象的方便类型。

关于平移和缩放的注意事项
与AngleAxis一样，这些类旨在简化线性 (Matrix) 和仿射 (Transform) 变换的创建/初始化。然而，与使用效率低下的AngleAxis不同，这些类对于编写将任何类型的转换作为输入的通用且高效的算法可能仍然很有趣。

上述任何转换类型都可以转换为任何其他具有相同性质的类型，或者转换为更通用的类型。下面是一些额外的例子：

```
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(..);
```

跨转换类型的通用 API

在某种程度上，Eigen的几何模块允许您编写适用于任何类型转换表示的通用算法：

两个转换的串联	**gen1 * gen2;**
将变换应用于向量	vec2 = gen1 * vec1;
获取变换的逆	gen2 = gen1.inverse();
球面插值 （仅限Rotation2D和Quaternion）	rot3 = rot1.slerp(alpha,rot2);

仿射变换

通用仿射变换由Transform类表示，它在内部是一个 (Dim+1)^2 矩阵。在Eigen 中，我们选择不区分点和向量，这样所有点实际上都由来自原点 ( )的位移向量表示。考虑到这一点，当应用变换时，实点和向量是不同的。p ≡ p −0

将变换应用到一个点	**VectorNf p1, p2;** **p2 = t * p1;**
将变换应用于**向量**	VectorNf vec1, vec2; vec2 = t.linear() * vec1;
对**法*向量**应用*一般*变换 （有关说明，请参阅本常见问题解答的主题 5.27 ）	向量Nf n1, n2; MatrixNf normalMatrix = t.linear().inverse().transpose(); n2 = (normalMatrix * n1).normalized();
对**法向量**应用*纯 旋转* 变换（无缩放，无剪切）	n2 = t.linear() * n1;
OpenGL 兼容性**3D**	glLoadMatrixf(t.data());
OpenGL 兼容性**2D**	Affine3f aux(Affine3f::Identity ()); aux.linear().topLeftCorner<2,2>() = t.linear(); aux.translation().start<2>() = t.translation(); glLoadMatrixf(aux.data());

组件 存取器

对内部矩阵的完全读写访问	**t.matrix() = matN1xN1; // N1 表示 N+1** **matN1xN1 = t.matrix();**
系数存取器	t(i,j) = scalar; <=> t.matrix()(i,j) = scalar; scalar = t(i,j); <=> scalar = t.matrix()(i,j);
翻译部分	t.translation() = vecN; vecN = t.translation();
线性部分	t.linear() = matNxN; matNxN = t.linear();
提取旋转矩阵	matNxN = t.rotation();

转换 创建

虽然可以通过连接基本转换来创建和更新转换对象，但Transform类还具有过程 API：

	程序API	等效的自然 API
翻译	t.translate(Vector*(tx,ty,..));* *t.pretranslate(Vector*(tx,ty,..));	t *= Translation*(tx,ty,..);* *t = Translation*(tx,ty,..) * t;
旋转 *在 2D 和程序 API 中，any_rotation 也可以是弧度的角度*	t.rotate(any_rotation); t.prerotate(any_rotation);	t *= any_rotation; t = any_rotation * t;
缩放	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;
剪切变换 （**仅限**2D**！）	t.shear(sx,sy); t.preshear(sx,sy);	

请注意，在这两个 API 中，任何多个转换都可以连接到一个表达式中，如以下两个等效示例所示：

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

欧拉角

欧拉角可能便于创建旋转对象。另一方面，由于存在 24 种不同的约定，因此使用起来非常混乱。此示例显示如何根据 2-1-2 约定创建旋转矩阵。

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