importer ' fléchette:math ' ;   
import ' package:flutter / widgets.dart ' ;   
  
typedef MatrixGestureDetectorCallback = void Function(   
Matrix4 matrix ,  
 Traduction Matrix4DeltaMatrix ,  
 Échelle Matrix4DeltaMatrix ,  
 Matrice4 rotationDeltaMatrice   
) ;   
  
*/// [ MatrixGestureDetector ] détecte les gestes de translation, d'échelle et de rotation   
/// et les combine dans un objet [Matrix4] qui peut être utilisé par le widget [Transform] /// ou par le code [ CustomPainter ] de bas niveau. Vous pouvez personnaliser les types de   
gestes rapportés /// en passant [ shouldTranslate ], [ shouldScale ] et [ shouldRotate ]   
/// parameters.///*class MatrixGestureDetector étend StatefulWidget {  
 */// [Matrix4] rappel de notification de changement   
///* dernier MatrixGestureDetectorCallback onMatrixUpdate ;  
  
 */// Le [ enfant ] contenu par ce détecteur.   
/// /// {@macro flutter.widgets.child }   
///* Widget final enfant ;  
  
 */// S'il faut détecter les gestes de traduction lors du traitement de l'événement.   
/// /// La valeur par défaut est true. ///* bool final shouldTranslate ;  
  
 */// S'il faut détecter les gestes d'échelle lors du traitement de l'événement.   
/// /// La valeur par défaut est true. ///* bool final shouldScale ;  
  
 */// S'il faut détecter les gestes de rotation lors du traitement de l'événement.   
/// /// La valeur par défaut est true. ///* bool final shouldRotate ;  
  
 */// Si le widget [ ClipRect ] doit couper le widget [ enfant ].   
/// /// La valeur par défaut est true. ///* bool final clipChild ;  
  
 */// Le comportement du test d'atteinte, transmis au GestureDetector sous-jacent .   
/// /// Par défaut , HitTestBehavior.deferToChild   
///* final HitTestBehavior comportement ;  
  
 */// Lorsqu'il est défini, il sera utilisé pour calculer un point focal "fixe"   
/// aligné par rapport à la taille de ce widget.* Alignement définitif ? alignementpointfocal ;   
  
const MatrixGestureDetector ({   
Key? key ,   
required this . onMatrixUpdate ,   
required this . child ,  
 cela . devraitTraduire = vrai,  
 cela . devraitÉchelle = vrai,  
 cela . devrait faire pivoter = vrai,  
 cela . clipEnfant = vrai,  
 cela . alignement du point focal ,  
 cela . comportement = HitTestBehavior. deferToChild ,  
 }) : super (clé : clé) ;  
  
 @passer outre  
 \_ MatrixGestureDetectorState createState () => \_ MatrixGestureDetectorState () ;  
  
 *///   
/// Composez la matrice à partir des matrices de translation, d'échelle et de rotation - vous pouvez /// passer un null pour ignorer toute matrice de la composition. /// /// Si [matrice] n'est pas nul le résultat de la composition sera concaténé /// à cette [matrice], sinon la matrice identité sera utilisée. ///* statique Matrice4 *compose* (Matrice4? Matrice , Matrice4? Matrice traduction ,  
 Matrice4 ? échelleMatrice , Matrice4 ? rotationMatrix ) {  
 //if (matrice == null ) matrice = Matrix4.identity();  
 matrice ???= Matrice4 . identité () ; // *A FAIRE MBO* if ( translationMatrix != null ) matrice = translationMatrix \* matrice ;   
if ( scaleMatrix != null ) matrice = scaleMatrix \* matrice ;   
if ( rotationMatrix != null ) matrice = rotationMatrix \* matrice ;   
matrice de retour ! ;  
 }  
  
 *///   
/// Décompose [matrice] en [ MatrixDecomposedValues. traduction ],   
/// [ MatrixDecomposedValues. échelle ] et [ MatrixDecomposedValues. rotation ] composants.  
 ///* statique MatrixDecomposedValues *decomposeToValues* (matrice Matrix4) {  
 var tableau = matrice.applyToVector3Array([ 0 , 0 , 0 , 1 , 0 , 0 ]) ;  
 Traduction de décalage = Décalage (array[ 0 ] , array[ 1 ]) ;  
 Décalage delta = Décalage (array[ 3 ] - array[ 0 ] , array[ 4 ] - array[ 1 ]) ;  
 double échelle = delta. éloignement ;  
 double rotation = delta. sens ;   
return MatrixDecomposedValues (translation , scale , rotation) ;  
 }   
}   
  
classe \_ MatrixGestureDetectorState extend State< MatrixGestureDetector > {   
Matrix4 translationDeltaMatrix = Matrice4 . identité () ;  
 Échelle Matrix4DeltaMatrix = Matrice4 . identité () ;  
 Matrice4 rotationDeltaMatrice = Matrice4 . identité () ;  
 Matrice4 matrice = Matrice4 . identité () ;  
  
 @passer outre  
 Widget build ( contexte BuildContext ) {   
Widget enfant =  
 widget . clipEnfant ? ClipRect (enfant : widget . enfant ) : widget . enfant ;   
return GestureDetector (   
behavior: widget . behavior ,  
 onScaleStart : onScaleStart ,  
 onScaleUpdate : onScaleUpdate ,  
 enfant : enfant ,  
 ) ;  
 }   
  
\_ ValueUpdater <Décalage> translationUpdater = \_ ValueUpdater (   
valeur : Décalage. *zéro* ,  
 onUpdate : ( oldVal , newVal ) => newVal - oldVal ,  
 ) ;  
 \_ ValueUpdater <double> scaleUpdater = \_ ValueUpdater (   
valeur : 1.0 ,  
 onUpdate : ( oldVal , newVal ) => newVal / oldVal ,  
 ) ;  
 \_ ValueUpdater <double> rotationUpdater = \_ ValueUpdater (   
valeur : 0.0 ,  
 onUpdate : ( oldVal , newVal ) => newVal - oldVal ,  
 ) ;   
  
void onScaleStart ( détails de ScaleStartDetails ) {  
 translationUpdater . évaluer = détails. point focal ;  
 scaleUpdater . évaluer = 1,0 ;  
 rotationUpdater . évaluer = 0,0 ;  
 }  
  
 void onScaleUpdate ( détails de ScaleUpdateDetails ) {  
 traductionDeltaMatrix = Matrice4 . identité () ;  
 échelleDeltaMatrice = Matrice4 . identité () ;  
 rotationDeltaMatrix = Matrice4 . identité () ;  
  
 // gère la traduction de la matrice  
 if ( widget . shouldTranslate ) {   
Offset translationDelta = translationUpdater .update ( details. focalPoint ) ;  
 traductionDeltaMatrix = \_translate( translationDelta ) ;  
 matrice = translationDeltaMatrix \* matrice ;  
 }  
  
 final focalPointAlignment = widget . alignementpointfocal ;   
Point focal final = Alignement du point focal == null ?  
 détails. localFocalPointlocalFocalPoint :  
 focalPointAlignment.alongSize ( context . size !) ;  
  
 // gère la mise à l'échelle de la matrice  
 si ( widget . shouldScale && détails. échelle !   
= 1.0 ) {   
double scaleDelta = scaleUpdater .update ( details.scale ) ; échelleDeltaMatrice = \_scale( scaleDelta , point focal ) ;  
 matrice = échelleDeltaMatrix \* matrice ;  
 }  
  
 // gère la rotation de la matrice  
 if ( widget . shouldRotate && détails. rotation != 0.0 ) {   
double rotationDelta = rotationUpdater .update ( details.rotation ) ;  
 rotationDeltaMatrix = \_rotate( rotationDelta , point focal ) ;  
 matrice = rotationDeltaMatrix \* matrice ;  
 }  
  
 widget . onMatrixUpdate (  
 matrice , traductionDeltaMatrix , échelleDeltaMatrix , rotationDeltaMatrix ) ;  
 }   
  
Matrix4 \_translate (Traduction décalée) {  
 var dx = traduction. dx ;   
var dy = traduction. mourir ;  
  
 // ..[0] = 1 # x échelle   
// ..[5] = 1 # échelle y // ..[10] = 1 # diagonale "un" // ..[12] = dx # x traduction // ..[13] = dy # y traduction   
// ..[15] = 1 # diagonal "un" retour Matrix4 ( 1 , 0 , 0 , 0 , 0 , 1 , 0 , 0 , 0 , 0 , 1 , 0 , dx , dy , 0 , 1 ) ;  
 }   
  
Matrix4 \_scale (double échelle , Offset focalPoint ) {  
 var dx = ( 1 - échelle) \* point focal. dx ;   
var dy = ( 1 - échelle) \* point focal. mourir ;  
  
 // ..[0] = échelle # x échelle   
// ..[5] = échelle # y échelle // ..[10] = 1 # diagonale "un" // ..[12] = dx # x traduction // ..[13] = dy # y traduction   
// ..[15] = 1 # diagonal "un" retour Matrix4 (scale , 0 , 0 , 0 , 0 , scale , 0 , 0 , 0 , 0 , 1 , 0 , dx , dy , 0 , 1 ) ;  
 }   
  
Matrix4 \_rotate (double angle , Offset focalPoint ) {  
 var c = cos(angle) ;   
var s = sin(angle) ;   
var dx = ( 1 - c) \* Point focal. dx + s \* Point focal. mourir ;   
var dy = ( 1 - c) \* Point focal. mourir - s \* Point focal. dx ;  
  
 // ..[0] = c # x échelle   
// ..[1] = s # y biais // ..[4] = -s # x oblique // ..[5] = échelle c # y // ..[10] = 1 # diagonale "un" // ..[12] = dx # x traduction // ..[13] = dy # y traduction   
// ..[15] = 1 # diagonal "un" retour Matrix4 (c , s , 0 , 0 , -s , c , 0 , 0 , 0 , 0 , 1 , 0 , dx , dy , 0 , 1 ) ;  
 }   
}   
  
typedef \_ OnUpdate <T> = T Function(T oldValue , T newValue ) ;   
  
classe \_ ValueUpdater <T> {  
 final \_ OnUpdate <T> onUpdate ;  
 valeur T ;  
  
 \_ ValueUpdater ({  
 exigé cela . valeur ,   
requis ceci . à la mise à jour ,  
 }) ;  
  
 T update (T newValue ) {   
T mis à jour = onUpdate ( value , newValue ) ;  
 valeur = nouvelleValeur ;   
retour mis à jour ;  
 }   
}   
  
classe MatrixDecomposedValues {  
 */// Translation, dans la plupart des cas utile uniquement pour les matrices qui ne sont   
/// qu'une translation (pas d'échelle et pas de rotation).* traduction offset finale ;  
  
 */// Facteur d'échelle.* double échelle finale ;  
  
 */// Rotation en radians, plage (- pi..pi ).* double rotation finale ;  
  
 MatrixDecomposedValues ( cette . traduction , cette . échelle , cette . rotation ) ;  
  
 @passer outre  
 Chaîne versChaîne () {  
 return ' MatrixDecomposedValues (translation: $ translation , scale: ${ scale .toStringAsFixed ( 3 )} , rotation: ${ rotation .toStringAsFixed ( 3 )} )' ;  
 }   
}