import 'dart:math';  
import 'package:flutter/widgets.dart';  
  
typedef MatrixGestureDetectorCallback = void Function(  
 Matrix4 matrix,  
 Matrix4 translationDeltaMatrix,  
 Matrix4 scaleDeltaMatrix,  
 Matrix4 rotationDeltaMatrix  
 );  
  
*/// [MatrixGestureDetector] detects translation, scale and rotation gestures  
/// and combines them into [Matrix4] object that can be used by [Transform] widget  
/// or by low level [CustomPainter] code. You can customize types of reported  
/// gestures by passing [shouldTranslate], [shouldScale] and [shouldRotate]  
/// parameters.  
///*class MatrixGestureDetector extends StatefulWidget {  
 */// [Matrix4] change notification callback  
 ///* final MatrixGestureDetectorCallback onMatrixUpdate;  
  
 */// The [child] contained by this detector.  
 ///  
 /// {@macro flutter.widgets.child}  
 ///* final Widget child;  
  
 */// Whether to detect translation gestures during the event processing.  
 ///  
 /// Defaults to true.  
 ///* final bool shouldTranslate;  
  
 */// Whether to detect scale gestures during the event processing.  
 ///  
 /// Defaults to true.  
 ///* final bool shouldScale;  
  
 */// Whether to detect rotation gestures during the event processing.  
 ///  
 /// Defaults to true.  
 ///* final bool shouldRotate;  
  
 */// Whether [ClipRect] widget should clip [child] widget.  
 ///  
 /// Defaults to true.  
 ///* final bool clipChild;  
  
 */// The hit test behavior, passed to the underlying GestureDetector.  
 ///  
 /// Defaults to HitTestBehavior.deferToChild  
 ///* final HitTestBehavior behavior;  
  
 */// When set, it will be used for computing a "fixed" focal point  
 /// aligned relative to the size of this widget.* final Alignment? focalPointAlignment;  
  
 const MatrixGestureDetector({  
 Key? key,  
 required this.onMatrixUpdate,  
 required this.child,  
 this.shouldTranslate = true,  
 this.shouldScale = true,  
 this.shouldRotate = true,  
 this.clipChild = true,  
 this.focalPointAlignment,  
 this.behavior = HitTestBehavior.deferToChild,  
 }) : super(key: key);  
  
 @override  
 \_MatrixGestureDetectorState createState() => \_MatrixGestureDetectorState();  
  
 *///  
 /// Compose the matrix from translation, scale and rotation matrices - you can  
 /// pass a null to skip any matrix from composition.  
 ///  
 /// If [matrix] is not null the result of the composing will be concatenated  
 /// to that [matrix], otherwise the identity matrix will be used.  
 ///* static Matrix4 *compose*(Matrix4? matrix, Matrix4? translationMatrix,  
 Matrix4? scaleMatrix, Matrix4? rotationMatrix) {  
 //if (matrix == null) matrix = Matrix4.identity();  
 matrix ??= Matrix4.identity(); //*TODO MBO* if (translationMatrix != null) matrix = translationMatrix \* matrix;  
 if (scaleMatrix != null) matrix = scaleMatrix \* matrix;  
 if (rotationMatrix != null) matrix = rotationMatrix \* matrix;  
 return matrix!;  
 }  
  
 *///  
 /// Decomposes [matrix] into [MatrixDecomposedValues.translation],  
 /// [MatrixDecomposedValues.scale] and [MatrixDecomposedValues.rotation] components.  
 ///* static MatrixDecomposedValues *decomposeToValues*(Matrix4 matrix) {  
 var array = matrix.applyToVector3Array([0, 0, 0, 1, 0, 0]);  
 Offset translation = Offset(array[0], array[1]);  
 Offset delta = Offset(array[3] - array[0], array[4] - array[1]);  
 double scale = delta.distance;  
 double rotation = delta.direction;  
 return MatrixDecomposedValues(translation, scale, rotation);  
 }  
}  
  
class \_MatrixGestureDetectorState extends State<MatrixGestureDetector> {  
 Matrix4 translationDeltaMatrix = Matrix4.identity();  
 Matrix4 scaleDeltaMatrix = Matrix4.identity();  
 Matrix4 rotationDeltaMatrix = Matrix4.identity();  
 Matrix4 matrix = Matrix4.identity();  
  
 @override  
 Widget build(BuildContext context) {  
 Widget child =  
 widget.clipChild ? ClipRect(child: widget.child) : widget.child;  
 return GestureDetector(  
 behavior: widget.behavior,  
 onScaleStart: onScaleStart,  
 onScaleUpdate: onScaleUpdate,  
 child: child,  
 );  
 }  
  
 \_ValueUpdater<Offset> translationUpdater = \_ValueUpdater(  
 value: Offset.*zero*,  
 onUpdate: (oldVal, newVal) => newVal - oldVal,  
 );  
 \_ValueUpdater<double> scaleUpdater = \_ValueUpdater(  
 value: 1.0,  
 onUpdate: (oldVal, newVal) => newVal / oldVal,  
 );  
 \_ValueUpdater<double> rotationUpdater = \_ValueUpdater(  
 value: 0.0,  
 onUpdate: (oldVal, newVal) => newVal - oldVal,  
 );  
  
 void onScaleStart(ScaleStartDetails details) {  
 translationUpdater.value = details.focalPoint;  
 scaleUpdater.value = 1.0;  
 rotationUpdater.value = 0.0;  
 }  
  
 void onScaleUpdate(ScaleUpdateDetails details) {  
 translationDeltaMatrix = Matrix4.identity();  
 scaleDeltaMatrix = Matrix4.identity();  
 rotationDeltaMatrix = Matrix4.identity();  
  
 // handle matrix translating  
 if (widget.shouldTranslate) {  
 Offset translationDelta = translationUpdater.update(details.focalPoint);  
 translationDeltaMatrix = \_translate(translationDelta);  
 matrix = translationDeltaMatrix \* matrix;  
 }  
  
 final focalPointAlignment = widget.focalPointAlignment;  
 final focalPoint = focalPointAlignment == null ?  
 details.localFocalPoint :  
 focalPointAlignment.alongSize(context.size!);  
  
 // handle matrix scaling  
 if (widget.shouldScale && details.scale != 1.0) {  
 double scaleDelta = scaleUpdater.update(details.scale);  
 scaleDeltaMatrix = \_scale(scaleDelta, focalPoint);  
 matrix = scaleDeltaMatrix \* matrix;  
 }  
  
 // handle matrix rotating  
 if (widget.shouldRotate && details.rotation != 0.0) {  
 double rotationDelta = rotationUpdater.update(details.rotation);  
 rotationDeltaMatrix = \_rotate(rotationDelta, focalPoint);  
 matrix = rotationDeltaMatrix \* matrix;  
 }  
  
 widget.onMatrixUpdate(  
 matrix, translationDeltaMatrix, scaleDeltaMatrix, rotationDeltaMatrix);  
 }  
  
 Matrix4 \_translate(Offset translation) {  
 var dx = translation.dx;  
 var dy = translation.dy;  
  
 // ..[0] = 1 # x scale  
 // ..[5] = 1 # y scale  
 // ..[10] = 1 # diagonal "one"  
 // ..[12] = dx # x translation  
 // ..[13] = dy # y translation  
 // ..[15] = 1 # diagonal "one"  
 return Matrix4(1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, dx, dy, 0, 1);  
 }  
  
 Matrix4 \_scale(double scale, Offset focalPoint) {  
 var dx = (1 - scale) \* focalPoint.dx;  
 var dy = (1 - scale) \* focalPoint.dy;  
  
 // ..[0] = scale # x scale  
 // ..[5] = scale # y scale  
 // ..[10] = 1 # diagonal "one"  
 // ..[12] = dx # x translation  
 // ..[13] = dy # y translation  
 // ..[15] = 1 # diagonal "one"  
 return 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) \* focalPoint.dx + s \* focalPoint.dy;  
 var dy = (1 - c) \* focalPoint.dy - s \* focalPoint.dx;  
  
 // ..[0] = c # x scale  
 // ..[1] = s # y skew  
 // ..[4] = -s # x skew  
 // ..[5] = c # y scale  
 // ..[10] = 1 # diagonal "one"  
 // ..[12] = dx # x translation  
 // ..[13] = dy # y translation  
 // ..[15] = 1 # diagonal "one"  
 return 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);  
  
class \_ValueUpdater<T> {  
 final \_OnUpdate<T> onUpdate;  
 T value;  
  
 \_ValueUpdater({  
 required this.value,  
 required this.onUpdate,  
 });  
  
 T update(T newValue) {  
 T updated = onUpdate(value, newValue);  
 value = newValue;  
 return updated;  
 }  
}  
  
class MatrixDecomposedValues {  
 */// Translation, in most cases useful only for matrices that are nothing but  
 /// a translation (no scale and no rotation).* final Offset translation;  
  
 */// Scaling factor.* final double scale;  
  
 */// Rotation in radians, (-pi..pi) range.* final double rotation;  
  
 MatrixDecomposedValues(this.translation, this.scale, this.rotation);  
  
 @override  
 String toString() {  
 return 'MatrixDecomposedValues(translation: $translation, scale: ${scale.toStringAsFixed(3)}, rotation: ${rotation.toStringAsFixed(3)})';  
 }  
}