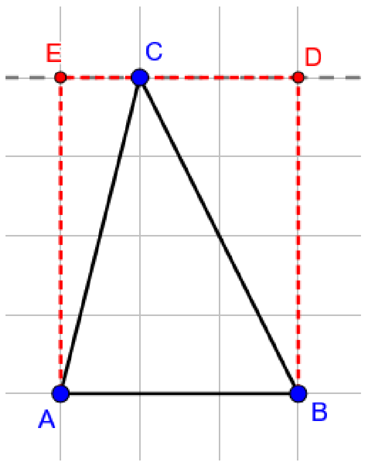
GeoGebra activity to show that the area of a triangle is half of the area of its related rectangle for the general case

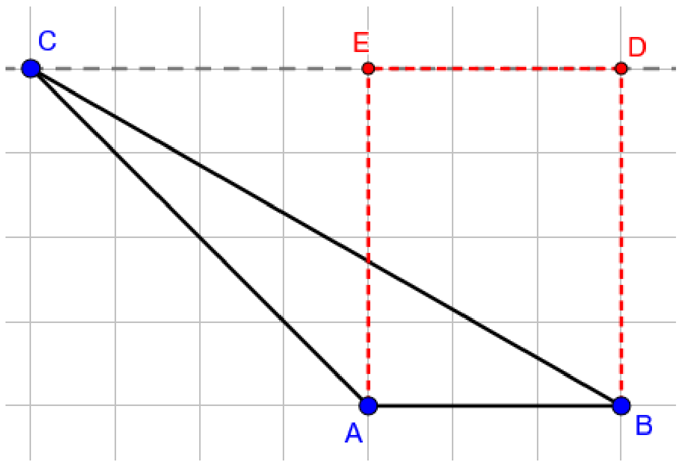
By Xuan Yuan TAN

This GeoGebra activity allows the learner to explore and discover that the area of a triangle is half of the area of its related rectangle, for any triangle.

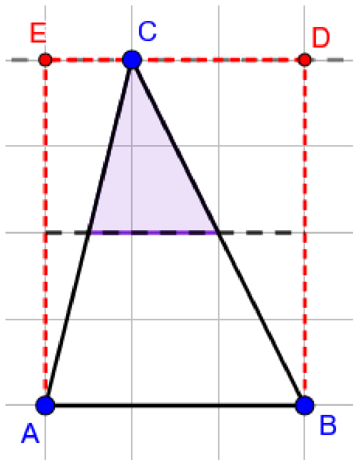
Explanation of method

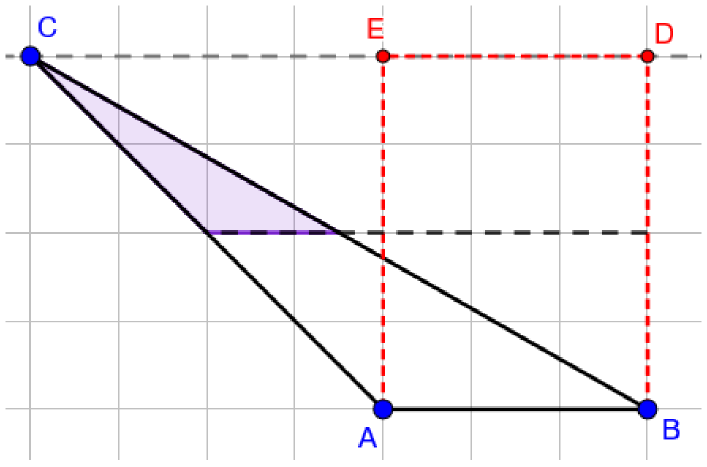
1. Given a triangle ABC, draw its related rectangle ABDE, where DE is parallel to AB and C lies on DE or DE extended. The diagrams below show the related rectangle for 2 different cases of triangle ABC.



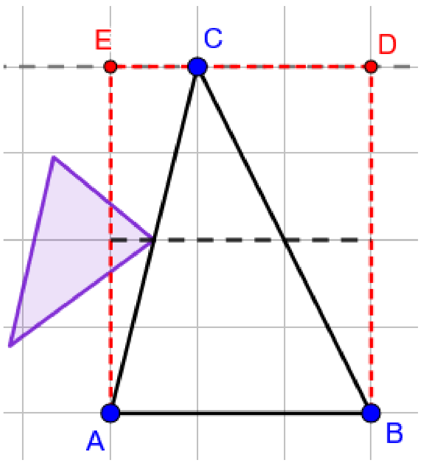


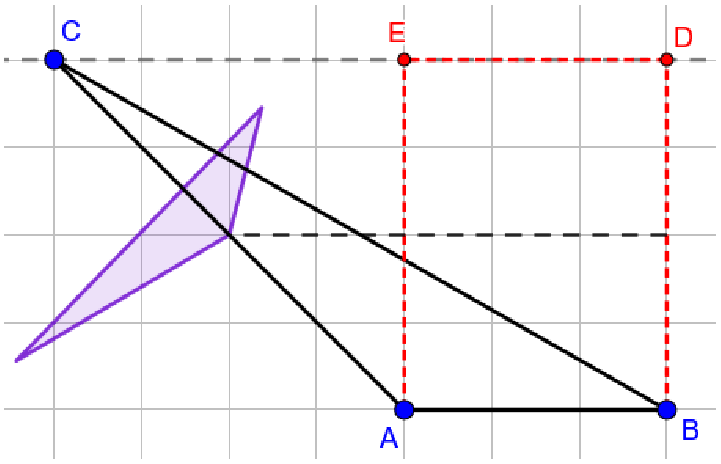
2. Divide the triangle into 2 parts, not necessarily equal, where the dividing line is parallel to AB and cuts across the middle of rectangle ABDE.

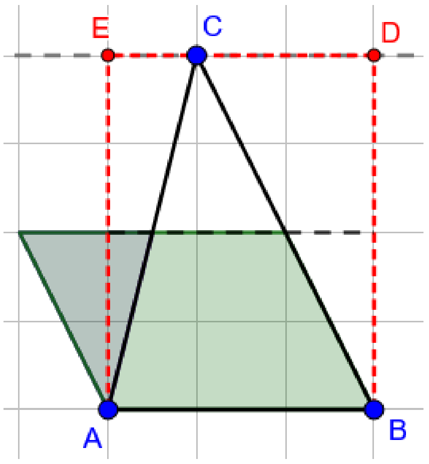


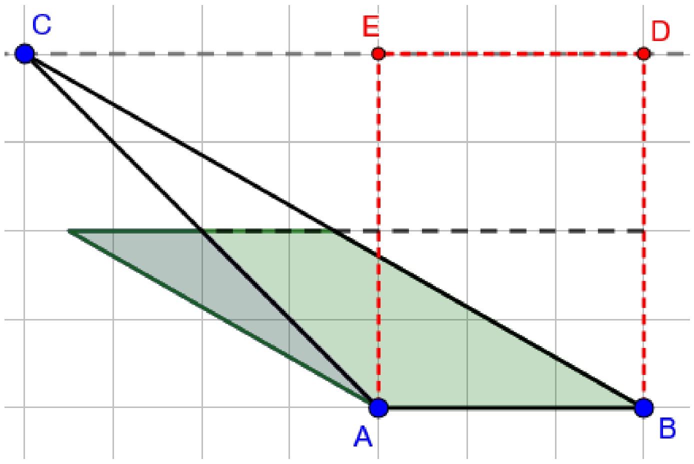


3. Rotate the top part of the triangle 180 degrees around the mid-point of one side of the triangle. In our examples here, we rotate around the mid-point of AC. After the rotation, we obtain a parallelogram.

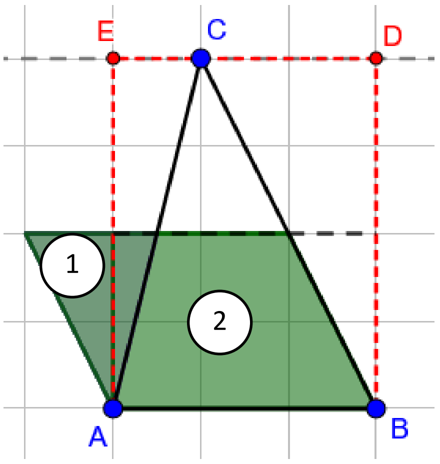


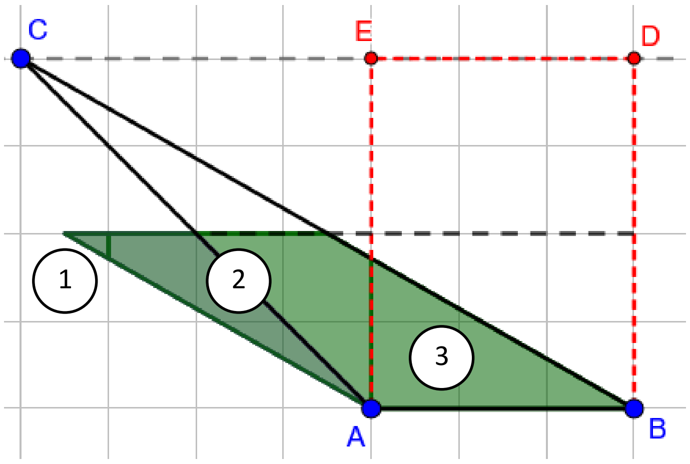




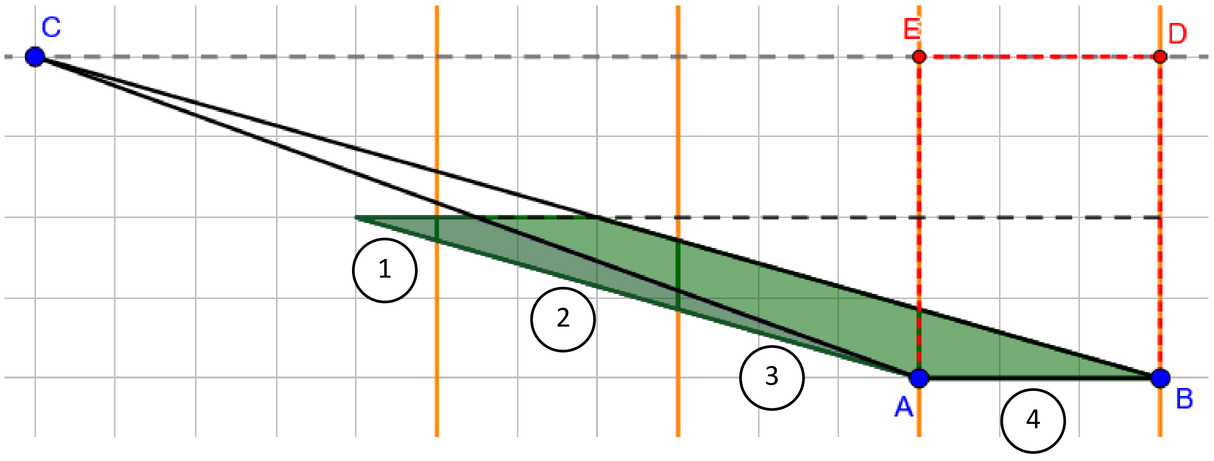


4. Split the parallelogram into smaller parts, where the width of each part is equal to the length of AB. In some cases, there may be a remaining triangular part at the corner of the parallelogram.

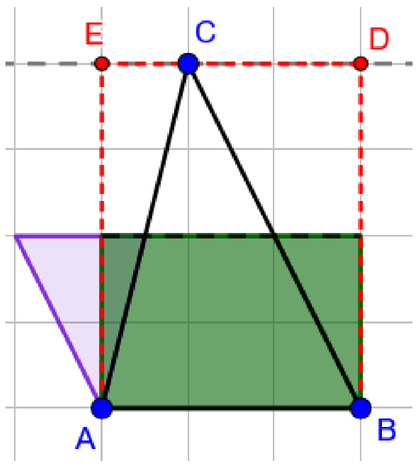


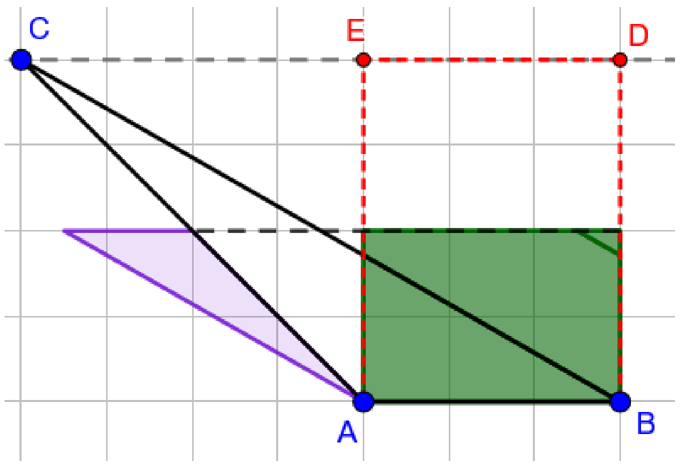


The example below illustrates the splitting more clearly with a different triangle.



5. As the sides are parallel, we can slide the parts so that they fit into the bottom half of rectangle ABDE.





6. Hence, the area of the triangle ABC is half of the area of its related rectangle ABDE. The above method works for any triangle.

7. The points A, B and C can be moved about to get a different triangle to apply the method on.

Creating the GeoGebra activity

The creation process can be split into 4 parts:

1. Creating the triangle and its related rectangle
2. Creating the things needed to rotate the top part of the triangle 180 degrees to form the parallelogram
3. Creating the things needed to split the parallelogram into smaller parts and sliding the parts into the related rectangle
4. Creating the instructions for the learner

Creating the triangle and its related rectangle

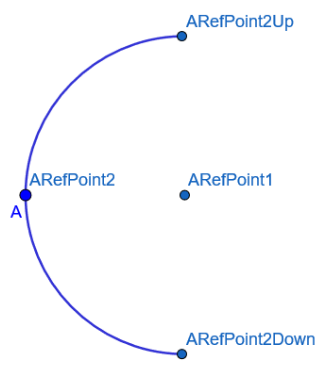
We need to constrain the vertices of the triangle in some way to prevent excessive distortion of the triangle. One way is to constrain point A to move in a circular arc and point B to move along a ray starting from A. Point C is free to move around.

1. Create the following 2 points, which will be our reference points and will be hidden:
   a. ARefPoint1 = (5, 0)
   b. ARefPoint2 = (0, 0)
2. Create a slider:

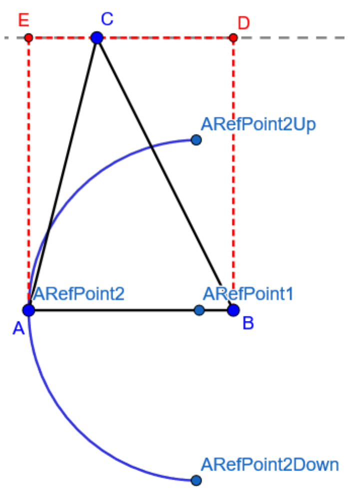
Type:	Angle		Min:	0 degrees
Name:	pointARotateConstraint		Max:	89 degrees
			Increment:	1 degree

   This slider is used to fix the angle of the circular arc. 89 degrees is 1 degree away from being perpendicular, so that we do not have to consider too many different cases of the triangle for subsequent steps.

3. We need 2 more points to construct the circular arc. Create the following 2 points:
   a. ARefPoint2Down = Rotate(ARefPoint2, pointARotateConstraint, ARefPoint1)
   b. ARefPoint2Up = Rotate(ARefPoint2, -pointARotateConstraint, ARefPoint1)
4. Now we can construct our circular arc, arcForA:
   a. arcForA = CircularArc(ARefPoint1, ARefPoint2Up, ARefPoint2Down)
5. Let A be a point on arcForA. So we have the following:



6. Construct a ray rayAArefPoint1 to put point B:
   a. rayAARefPoint1 = Ray(A, ARefPoint1)
7. Point B is a point on rayAArefPoint1. Point C is just a free point. Create segments to represent the sides of the triangle.
   a. sAB = Segment(A, B)
   b. sAC = Segment(A, C)
   c. sBC = Segment(B, C)
8. Construct the related rectangle ABDE. First, create two lines passing through A and B respectively, perpendicular to AB, and a line passing through C parallel to AB:
   a. perpLineA = PerpendicularLine(A, rayAARefPoint1)
   b. perpLineB = PerpendicularLine(B, rayAARefPoint1)
   c. lineC = Line(C, rayAARefPoint1)
9. Points D and E are the points of intersection of the lines above. Add in the line segments AE, BD and DE to complete the related rectangle ABDE.
   a. D = Intersect(lineC, perpLineB)
   b. E = Intersect(lineC, perpLineA)
   c. sAE = Segment(A, E)
   d. sBD = Segment(B, D)
   e. sDE = Segment(D, E)



10. From this point onwards, we will hide the circular arc that constrains point A.

11. Subsequent, when we cut the triangle into two and rotate the top part, whether it rotates to the left or to the right depends on whether the triangle is leaning to the left or to the right. To do this, we check whether the foot of the perpendicular from C to AB is to the left or the right of the mid-point of AB, by comparing the x-coordinates of the two points. So we create the following:

    a. perpLineC = PerpendicularLine(C, rayAARefPoint1)

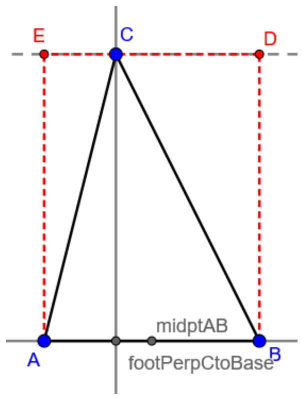
    b. lineAB = Line(A, B)

    c. footPerpCtoBase = Intersect(lineAB, perpLineC)

    d. midptAB = Midpoint(A, B)

    e. leansLeft = x(footPerpCtoBase) ⩽ x(midptAB)

    Note that we constrain point A so that here we only need to compare the x-coordinates. The result is the following. As leansLeft is a Boolean value, it is not shown.



12. We also need to check whether the triangle is upside down, that is, whether the y-coordinate of C is below the y-coordinate the foot of its perpendicular to AB.

    a. isUpsideDown = y(C) ⩽ y(footPerpCtoBase)

13. We then construct the points to cut the triangle into two. First, the mid-points:

a. F = Midpoint(A, C)

b. G = Midpoint(B, C)

c. midptAE = Midpoint(A, E)

d. midptBD = Midpoint(B, D)

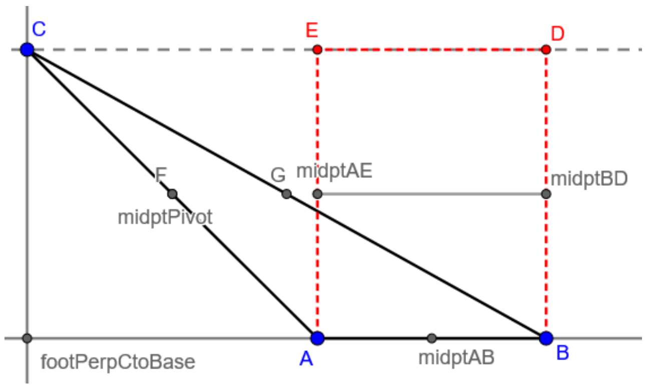
14. If leansLeft is true, point F will be used as the pivot or centre of rotation later for the top part of the triangle. Otherwise, point G will be used. We define a point based on this:

    a. midptPivot = If(leansLeft, F, G)

15. We now need to construct the line segment cutting across the middle of the related rectangle ABDE. The problem is that if the triangle is leaning too far out, we need to extend this segment. Hence, we first construct the segment cutting across the middle, but we will hide this segment:

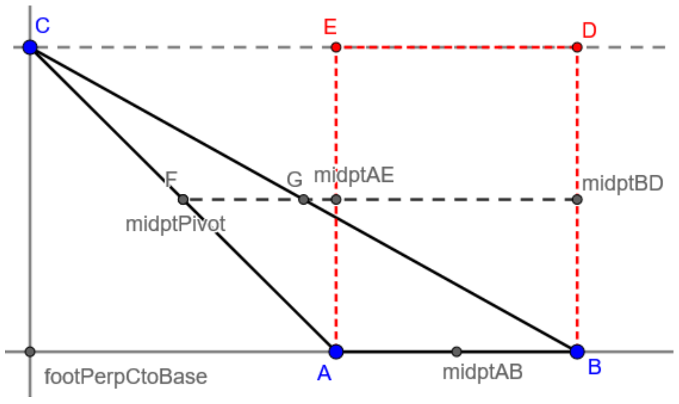
    a. sRectMidpts = Segment(midptAE, midptBD)

16. Let us illustrate the problem with an example triangle:



We need to extend the middle line all the way to midptPivot. So we need to check whether midptPivot lies on the line segment sRectMidpts to determine the line segment to draw. We can do this by checking whether the distance between midptPivot and the segment sRectMidpts is 0. If the distance is 0, then midptPivot lies on sRectMidpts, so we use sRectMidpts. Otherwise, we construct the line segment from the mid-point of the futher side of the rectangle to midptPivot. Thus, we define the following:

a. sCutHalf = If(Distance(midptPivot, sRectMidpts) == 0, sRectMidpts, Segment(If(leansLeft, midptBD, midptAE), midptPivot))



Rotating the top part of the triangle

17. The general idea is to create 3 points Cmove, Fmove and Gmove that represent the points C, F and G respectively as they are moved by a slider. These 3 points form the triangle that rotates.

18. We first create points that refer to points A and B depending on whether leansLeft is true. If leansLeft is true, point C will be rotated down to meet point A to form the corner (tip) of the parallelogram (abbreviated as pgram). As points A and B form the base of the triangle, we shall define the new points as follows:

    a. baseTipOfPgram = If(leansLeft, A, B)

    b. baseOtherTipOfPgram = If(leansLeft, B, A)

Defining Cmove

19. Point Cmove will move along a semicircle path from C to baseTipOfPgram. We are going to use the Path Parameter to adjust the position of point Cmove on the semicircle. As the path parameter takes values from 0 to 1, we create the following slider:

Type:	Number		Min:	0
Name:	CmoveParamSlider		Max:	1
			Increment:	0.01

For this increment to work, we need to set Rounding to 2 decimal places under GeoGebra Settings.

20. Construct the semicircle path. Unfortunately, we cannot just do Semicircle(C, baseTipOfPgram) because the semicircle always goes in a clockwise manner from the first point to the second point. This means that the semicircle may be constructed on the wrong side of the triangle depending on leansLeft and isUpsideDown, in which case we need to switch to using Semicircle(baseTipOfPgram, C). The table below shows the details.

leansLeft	isUpsideDown	Semicircle needed
true	false	Semicrcle(baseTipOfPgram, C)
false	false	Semicrcle(C, baseTipOfPgram)
true	true	Semicrcle(C, baseTipOfPgram)
false	true	Semicrcle(baseTipOfPgram, C)

Notice that when both leansLeft and isUpsideDown have the same value (either both true or both false), we need Semicrcle(C, baseTipOfPgram). Otherwise, we need Semicrcle(baseTipOfPgram, C). Hence, we define the path as follows:

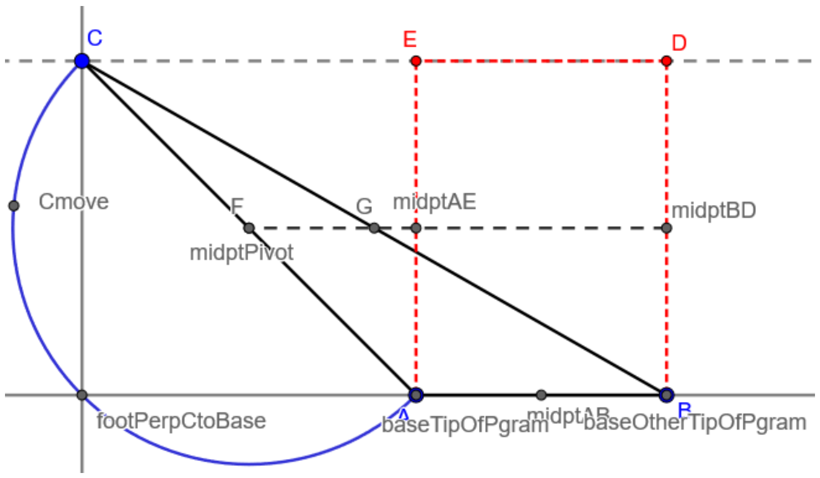
    a. CmovePath = If(leansLeft == isUpsideDown, Semicircle(C, baseTipOfPgram), Semicircle(baseTipOfPgram, C))

21. When we use Semicircle(C, baseTipOfPgram), the path parameter value is 0 at baseTipOfPgram and 1 at C. For Semicircle(baseTipOfPgram, C), the path parameter value is 0 at C and 1 at baseTipOfPgram. To ensure that the path parameter value is always 0 at C and 1 at baseTipOfPgram to correspond with the slider CmoveParamSlider, we need to define a new number to be used as the path parameter:

    a. CmovePathParam = If(leansLeft == isUpsideDown, 1 - CmoveParamSlider, CmoveParamSlider)

22. So now we can define Cmove:

    a. Cmove = Point(CmovePath, CmovePathParam)

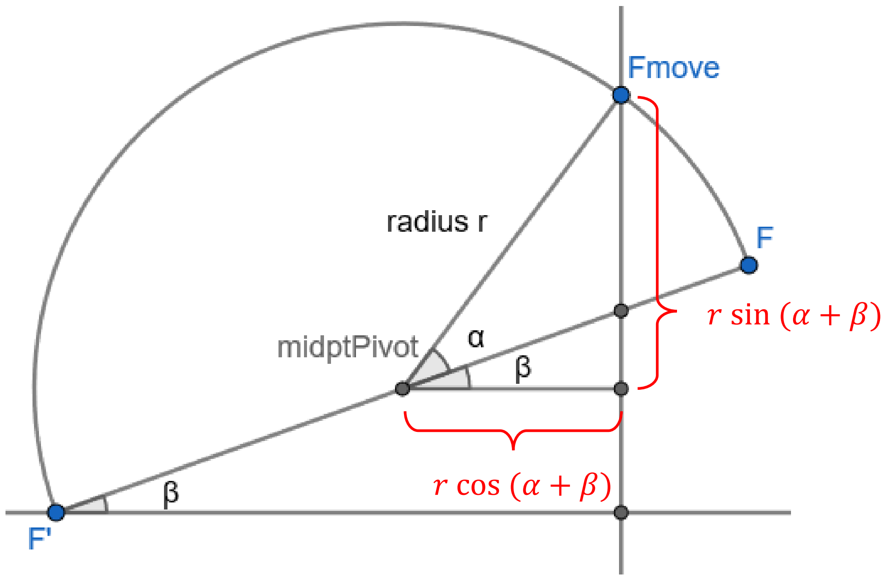


Defining Fmove and Gmove

23. Next, we move on to Fmove and Gmove. One of these points will be the pivot or centre of rotation for the rotation of the top part of the triangle. When leansLeft is true, Fmove will be the pivot point midptPivot. Otherwise, Gmove will be midptPivot. The other point will then rotate 180 degrees about midptPivot, which means that it will also move in a semicircle like Cmove. However, instead of defining a semicircle, we will use another method which manipulates the coordinates of the point directly.

    This method is easier for Fmove and Gmove because we only need to consider the line segment FG. For Cmove, if leansLeft is true, we would need to consider AC, and if leansLeft, we would need to consider BC, so it is more complicated.

24. Let us look at a simple case first. Consider the following:



Here, we have our point Fmove moving in a semicircle with radius *r* and centre midptPivot. The semicircle is tilted at an angle of β from the horizontal. When Fmove makes an angle of α with the base of the semicircle, its vertical distance from midptPivot is given by $r\sin(\alpha+\beta)$ and its horizontal distance from midptPivot is given by $r\cos(\alpha+\beta)$. Hence, the coordinates of Fmove are:

a. x-coordinate: x(midptPivot) + $r\cos(\alpha+\beta)$

b. y-coordinate: y(midptPivot) + $r\sin(\alpha+\beta)$

25. However, in the above simple case, we only added the coordinates of midptPivot to the distance of Fmove from midptPivot. In a more general case, we may need to subtract the distance from the coordinates of midptPivot instead, depending on whether the triangle is leaning to the left (leansLeft) and is upside down (isUpsideDown). Hence, we define the following numbers to help us with this:

    a. leansLeftNum = If(leansLeft, 1, -1)

    b. isUpsideDownNum = If(isUpsideDown, -1, 1)

    Notice that the sign (positive or negative) is flipped for isUpsideDownNum.

26. With the above, the coordinates of Fmove would like this:

    a. x-coordinate: x(midptPivot) + leansLeftNum $\times\, r\cos(\alpha+\beta)$

    b. y-coordinate: y(midptPivot) + isUpsideDownNum $\times\, r\sin(\alpha+\beta)$

27. Let us define the radius *r*. This is just the distance between F and G:

    a. distanceFG = Distance(F, G)

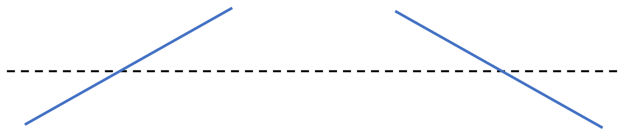
28. For the angle alpha, it should go from 0 degrees to 180 degrees. Our slider CmoveParamSlider goes from 0 to 1, so we need to transform this to get alpha:

    a. alpha = 180 deg * CmoveParamSlider

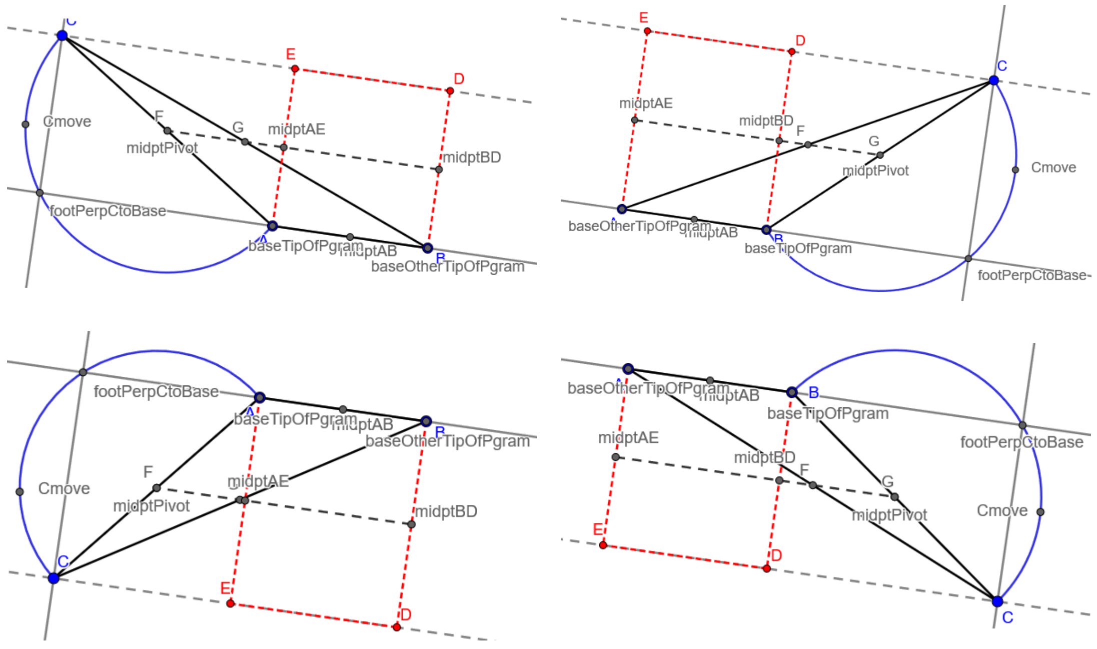
29. For the angle beta, it is the angle that the base of the semicircle makes with the x-axis. For our triangle ABC, it is the angle that the line FG makes with the x-axis. So let us define this line FG first.

    a. lineFG = Line(F, G)

30. The problem is, two different orientations of the line can give the same angle with the x-axis:



So, we need to check the sign of y(G) – y(F) to see which orientation the line is in. Also, there are 4 quadrants to consider whether to add the angle to alpha or subtract it from alpha:



This depends on the values of leansLeft and isUpsideDown. Hence, we can now define our angle beta, which we will call angleFGXsigned (signed because it considers whether it is positive or negative):

a. angleFGXsigned = Angle(xAxis, lineFG) * sgn(y(G) - y(F)) * leansLeftNum * isUpsideDownNum

31. Now we have the full x- and y-coordinates of Fmove, which we will define as two numbers FGmovingCoordX and FGmovingCoordY:

a. FGmovingCoordX = x(midptPivot) + leansLeftNum * distanceFG * cos(alpha + angleFGXsigned)

b. FGmovingCoordY = y(midptPivot) + isUpsideDownNum * distanceFG * sin(alpha + angleFGXsigned)

32. Considering that one of Fmove and Gmove will be the pivot while the other moves depending on leansLeft, we can define Fmove and Gmove as follows:

a. Fmove = If(leansLeft, F, (FGmovingCoordX, FGmovingCoordY))

b. Gmove = If(leansLeft, (FGmovingCoordX, FGmovingCoordY), G)

33. Thus, we have the top part of the triangle ABC, which is a smaller triangle t1:

a. t1 = Polygon({Cmove, Fmove, Gmove})

Notice that we enclose the 3 points with curly brackets to denote a list, because this prevents additional line segments from being generated in the Algebra pane, and we do not need to refer to the additional line segments.

Partitioning the parallelogram and sliding its parts

Getting baseMultiple and the number of parts

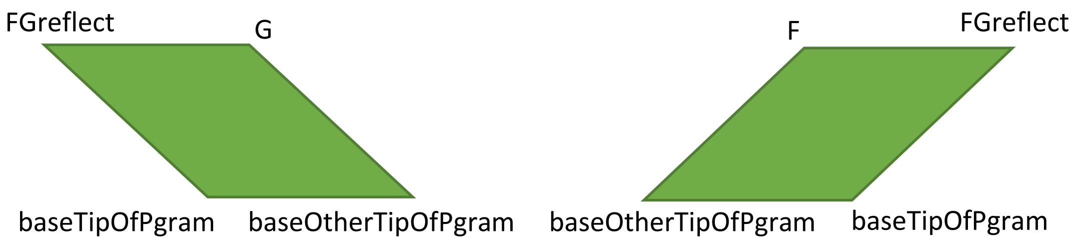
34. We need to define the end-point of the point Fmove or Gmove (depending on the value of leansLeft) after it has rotated about the point midptPivot. As the rotation is 180 degrees, it is equivalent to reflecting about the midptPivot. We call this end-point FGreflect:

    a. FGreflect = Reflect(If(leansLeft, G, F), midptPivot)

35. We define the line segment running across the top of the parallelogram:

    a. sFGreflect = Segment(If(leansLeft, G, F), FGreflect)

36. So now we have the 4 vertices of the parallelogram that is formed after the top part of triangle ABC has rotated about midptPivot. The following shows the vertices of the parallelogram in 2 different quadrants:

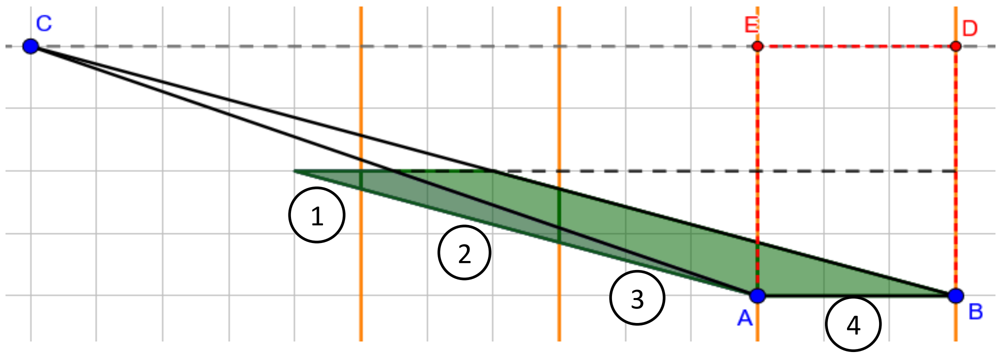


37. Let us draw this parallelogram, which will be shown once the top part of triangle ABC has finished rotating:

    a. pgramEntire = Polygon({baseOtherTipOfPgram, If(leansLeft, G, F), FGreflect, baseTipOfPgram})

38. The diagrams for the rest of this section will focus on the parallelogram and hide other objects that are not relevant.

39. We are going to split the parallelogram into parts (partitions), where each part has a width equal to the length of AB, which is the base of triangle ABC. There may be an additional remaining part, which we will call the Tail Part. Then we need to allow a slider to make the parts slide into the related rectangle ABDE. Let us use a diagram from an earlier section to show an example of the splitting:



40. As AB may be slanted, we cannot just directly adjust the x-coordinate of A to form the partitions. A method that manipulates the x- and y-coordinates directly may be possible, but we will use another method instead. We will translate (or move) point A by a vector whose

magnitude (or length) is equal to the length of the base AB. The vector will also allow us to slide the parts into the rectangle later by translating the points. Let us define this vector vBase:

   a. vBase = Vector(baseOtherTipOfPgram, baseTipOfPgram)

We define the vector this way so that it always points towards the part of the parallelogram that is protruding from the related rectangle ABDE.

41. We need to determine how many parts we are going to have. We are going to measure along lineAB to get this. So, first, drop a perpendicular line from FGreflect to lineAB and get the point at the foot of the perpendicular line:

   a. perpLineFGreflect = PerpendicularLine(FGreflect, lineAB)

   b. footPerpFGreflectToBase = Intersect(perpLineFGreflect, lineAB)

42. Then, we define the number baseMultiple that divides the distance between the foot of the perpendicular above and baseTipOfPgram by the length of vBase. This tells us how many parts there are, including the fractional part:

   a. baseMultiple = Distance(baseTipOfPgram, footPerpFGreflectToBase) / Length(vBase)

43. The fractional part indicates the Tail Part. To have whole numbers to work with, we define two numbers that refer to the floor and ceiling of baseMultiple respectively:

   a. numParts1 = floor(baseMultiple)

   b. numParts2 = ceil(baseMultiple)

The difference between the two numbers indicates the presence of the Tail Part.

Marking out the boundaries of the partitions

44. We shall define some terms to refer to the "external" and "internal" sides or edges of the parallelogram, with respect to whether they are within the related rectangle ABDE:

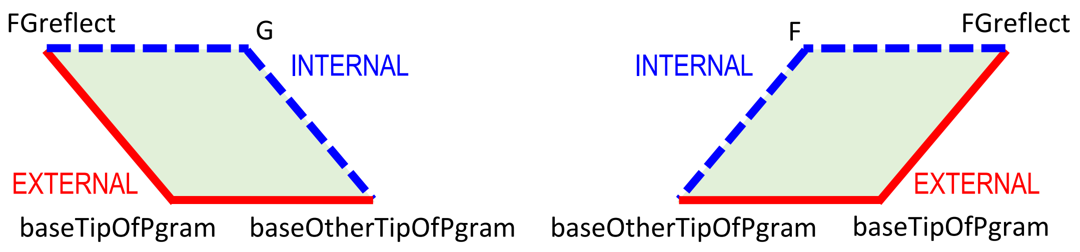
   a. pgramExtSlantEdge = Segment(baseTipOfPgram, FGreflect)

   b. pgramIntSlantEdge = If(leansLeft, Segment(B, G), Segment(A, F))

   c. pgramJoinedExtEdge = Join({{pgramExtSlantEdge}, {sAB}})

   d. pgramJoinedIntEdge = Join({{pgramIntSlantEdge}, {sFGreflect}})

The diagrams below illustrate this. The external edges are solid lines while the internal edges are dashed lines:



Creating the lists with Sequence()

45. The general idea for the partitioning is:

    a. Mark out the partitions along lineAB

    b. Drop perpendicular lines to lineC from the partition points along lineAB

    c. Mark out the points of intersection of the perpendicular lines with pgramJoinedExtEdge and pgramJoinedIntEdge

    d. Group the points of intersection according to the partitions

46. The first big sequence, seqA:

    a. Each partition point along lineAB can be obtained by translating the point baseTipOfPgram by a multiple of the vector vBase. The last partition point is translated by vBase multiplied by numParts1. So that means:

        i. For i = 0 to numParts1, do:

            1) Translate baseTipOfPgram by vector i * vBase

    b. Hence we have, in GeoGebra format:

        i. seqARaw = Sequence(Translate(baseTipOfPgram, Vector(i vBase)), i, 0, numParts1)

    c. We then add in baseOtherTipOfPgram to help us account for the first parallelogram part that is within rectangle ABDE:

        i. seqA = Join({{baseOtherTipOfPgram}, seqARaw})

47. The sequence of perpendicular lines, seqAPerpLines:

    a. Drop a perpendicular line from each partition point in seqA to lineC. So that means:

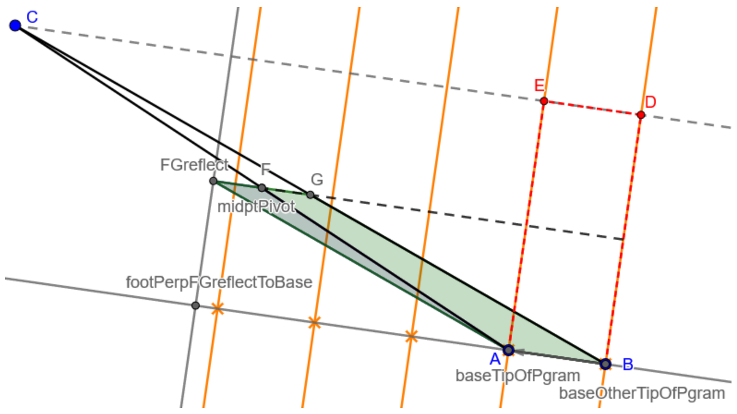
        i. For each element a in seqA, do:

            1) Construct the perpendicular line from a to lineC

    b. We can use the Zip() command in GeoGebra to do this:

        i. seqAPerpLines = Zip(PerpendicularLine(a, lineC), a, seqA)

    c. The following shows an example using a slanted triangle.



48. The second big sequence, seqB:

   a. Mark out the points of intersection of the perpendicular lines with pgramJoinedExtEdge. As pgramJoinedExtEdge contains more than one line segment, we need to get the point of intersection with each line segment separately. This requires a nested sequence, or a sequence within a sequence.

      i. For i = 1 to length of seqAPerpLines, do:

         1) For j = 1 to length of pgramJoinedExtEdge, do:

            a) Construct the point of intersection of the i-th perpendicular line with the j-th line segment

   b. In GeoGebra format:

      i. seqBRaw = Sequence(Sequence(Intersect(Element(seqAPerpLines, i), Element(pgramJoinedExtEdge, j)), j, 1, Length(pgramJoinedExtEdge)), i, 1, Length(seqAPerpLines))

   c. The above would produce a list like the following:

      i. "{{(?, ?), (3.48, 0.22)}, {(0.05, 0.71), (0.05, 0.71)}, {(-3.18, 2.6), (?, ?)}, {(-6.4, 4.49), (?, ?)}, {(-9.63, 6.39), (?, ?)}}"

   d. Notice that each element in the list is a list with two points. There are two cases:

      i. One of the points is undefined (?, ?) because the perpendicular line only intersects one of the line segments in pgramJoinedExtEdge

      ii. The two points are equal because the perpendicular line intersects the vertex of the parallelogram, which is where two line segments meet

   e. So, we need to flatten the list, remove the undefined points and remove duplicate values to obtain the final seqB, which is a simple list of the points of intersection:

      i. seqB = Unique(RemoveUndefined(Flatten(seqBRaw)))

49. The third big sequence, seqC:

   a. This is the same as seqB, just that we replace pgramJoinedExtEdge by pgramJoinedIntEdge. So that means:

      i. For i = 1 to length of seqAPerpLines, do:

         1) For j = 1 to length of pgramJoinedIntEdge, do:

            a) Construct the point of intersection of the i-th perpendicular line with the j-th line segment

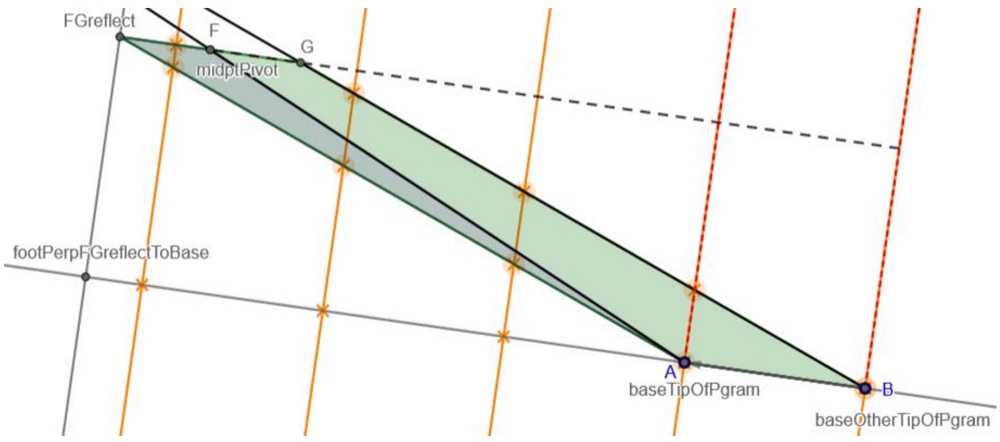
   b. In GeoGebra format:

      i. seqCRaw = Sequence(Sequence(Intersect(Element(seqAPerpLines, i), Element(pgramJoinedIntEdge, j)), j, 1, Length(pgramJoinedIntEdge)), i, 1, Length(seqAPerpLines))

   c. Then we have seqC:

i. seqC = Unique(RemoveUndefined(Flatten(seqCRaw)))

50. We now have the following, with the points in seqA, seqB and seqC marked out with orange X's:



Notice the shapes of the partitions, from left to right:

a. The Tail Part is a triangle

b. The last part before the Tail Part is a pentagon, which we will call the Pentagon Part

c. Zero or more little parallelograms

d. The first part within the related rectangle ABDE is a triangle

51. The sequence of little parallelograms, seqLittlePgrams:

a. We are going to group the points in seqB and seqC together to help us draw the little parallelograms later. Notice that the point baseOtherTipOfPgram is the first element of both seqB and seqC, so we can treat the first triangle part as a parallelogram with two vertices being the same point and include it in the sequence of little parallelograms. We stop at the second last element of seqC because the last element of seqC is the boundary between the Tail Part and the Pentagon Part.

i. For i = 1 to ((length of seqC) – 2), do:

1) Create a list that contains the 4 vertices of the little parallelogram, in the following order (anticlockwise):

a) Bottom corner: the i-th element of seqB

b) Top corner: the i-th element of seqC

c) Other top corner: the (i+1)-th element of seqC

d) Other bottom corner: the (i+1)-th element of seqB

ii. Note that i stops at (length of seqC) – 2 because this is the last little parallelogram before the Pentagon Part.

iii. We go in an anticlockwise order so that later we can feed the list directly to the Polygon() command to construct the polygon.

b. In GeoGebra format:

        i. seqLittlePgrams = Sequence({Element(seqB, i), Element(seqC, i), Element(seqC, i + 1), Element(seqB, i + 1)}, i, 1, Length(seqC) - 2)

52. The Pentagon Part:

    a. We can just define the Pentagon Part directly as a list. The vertices of the pentagon are:

        i. Second last element of seqB

        ii. Second last element of seqC

        iii. Point G if leansLeft is true, otherwise Point F

        iv. Last element of seqC

        v. Last element of seqB

    b. In GeoGebra format:

        i. pgramPentagonPoints = {Element(seqB, Length(seqB) - 1), Element(seqC, Length(seqC) - 1), If(leansLeft, G, F), Element(seqC, Length(seqC)), Element(seqB, Length(seqB))}

53. The Tail Part:

    a. The Tail Part is a triangle with the following vertices:

        i. Last element of seqB

        ii. Last element of seqC

        iii. FGreflect

    b. In GeoGebra format:

        i. pgramTailPoints = {Element(seqB, Length(seqB)), Element(seqC, Length(seqC)), FGreflect}

Sliding the parts

54. Let us create a slider to slide the parts:

Type:	Number		Min:	0
Name:	b		Max:	numParts2
			Increment:	0.01

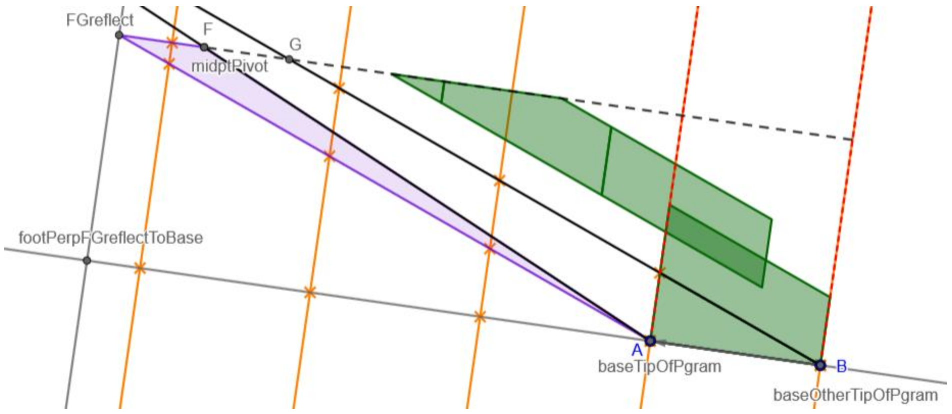
Notice that the max value of the slider is numParts2, because it depends on how many parts we need to slide into the rectangle ABDE. Each increase of 1 in the slider causes one part to finish sliding in.

55. To move each part, we translate the points of that part by a vector that is a multiple of the vector vBase. The multiple should be negative because we are sliding in the opposite direction of the vector. The multiple will also depend on the value of the slider *b*. The amount that each part slides depends on its distance from the related rectangle ABDE. We

use the Min() function to stop the sliding as the value of the slider *b* increases, so that the parts do not slide beyond the rectangle.

Part	**When should the part stop sliding as *b* increases?**	**Multiple of vBase to use for the translation vector**
Little Parallelograms	The n-th little parallelogram should stop sliding when *b* reaches n – 1	– Min(b, n – 1)
Pentagon Part	When *b* reaches numParts1	– Min(b, numParts1)
Tail Part	When *b* reaches numParts2	– Min(b, numParts2)

56. Moving the little parallelograms:
    a. For a = 1 to length of seqLittlePgrams, do:
        i. Translate the a-th element of seqLittlePgrams by the vector –Min(b, a – 1) * vBase
    b. The good thing about the Translate() command is that it can translate an object such as our little parallelogram, so that we do not need to go deeper down and translate each point in each little parallelogram individually.
    c. In GeoGebra format:
        i. seqLittlePgramsMoving = Sequence(Translate(Element(seqLittlePgrams, a), Vector(-Min(b, a - 1) vBase)), a, 1, Length(seqLittlePgrams))
57. Moving the Pentagon Part:
    a. Translate pgramPentagonPoints by the vector –Min(b, numParts1) * vBase
    b. In GeoGebra format:
        i. pgramPentagonPointsMoving = Translate(pgramPentagonPoints, Vector(-Min(b, numParts1) vBase))
58. Moving the Tail Part:
    a. Translate pgramTailPoints by the vector –Min(b, numParts2) * vBase
    b. In GeoGebra format:
        i. pgramTailPointsMoving = Translate(pgramTailPoints, Vector(-Min(b, numParts2) vBase))
59. We now construct the polygons for the points that are moving. First, combine all the moving points together into a single list:
    a. seqMoving = Join({seqLittlePgramsMoving, {pgramPentagonPointsMoving}, {pgramTailPointsMoving}})
60. Then, apply the Polygon() command on each element of seqMoving:
    a. seqPolysMoving = Zip(Polygon(a), a, seqMoving)
61. The following shows an example of the sliding in progress:



62. Finally, we construct a rectangle that we will show when all the parts have slid into the rectangle ABDE:

    a. rectHalf = Polygon({A, B, midptBD, midptAE})

Creating the instructions for the learner

63. The following describes some broad steps that are taken to create the instructions for the learner.

64. Create two checkboxes:

    a. First checkbox:

        i. Name: triCut , Caption: Cut the triangle

    b. Second checkbox:

        i. Name: pgramSplit , Caption: Split the triangle parts further

65. To help with the showing and hiding of each instruction step, we define the following Boolean values:

    a. showStep2 = triCut

    b. showStep3 = triCut && alpha == 180 deg

    c. showStep4 = triCut && alpha == 180 deg && pgramSplit

    d. showStep5 = triCut && alpha == 180 deg && pgramSplit && b == numParts2

66. Create the text boxes for the instructions for each step, as shown in the GeoGebra activity itself.

67. We also create a button with the caption "Restart from Step 1" for the learner to click on to restart from Step 1. To prevent accidental clicking of the button when laying out the objects, we define a Boolean value:

    a. buttonsEnabled = true

    We set buttonsEnabled to true in the final version, but while editing, we set it to false.

68. Clicking the button should set b to 0, uncheck pgramSplit, set CmoveParamSlider to 0 and uncheck triCut. The On Click script for the button is as follows (note that the scripting language is JavaScript):

```
if (ggbApplet.getValue("buttonsEnabled")) {

ggbApplet.setValue("b", 0);

ggbApplet.setValue("pgramSplit", 0);

ggbApplet.setValue("CmoveParamSlider", 0);

ggbApplet.setValue("triCut",0);

}
```

69. Now we just need to use showStep2, showStep3, showStep4 and showStep5 as the respective conditions to show certain objects. Some notable exceptions are as follows.

    a. pgramEntire:

        i. Condition to show object is showStep3 && b == 0

70. Turn on the grid and snapping to grid to help with positioning.

71. We are done.