Coordinate Spaces & Transformations

in InDesign CS4 – CC | Chapter 4

SCRIPTING SECRETS

Many technical details had to be discussed in the previous chapters but our efforts will be rewarded! We can now tackle the very first practical questions that arise in terms of InDesign geometry: how to specify or identify a *location* in a document. This seemingly simple problem requires, once again, a bit of meticulousness.

What is a Location?

Basically a location is nothing but a coordinate pair relative to some coordinate system, as detailed in Chapter 1. In practice, however, the question takes place in a slightly different perspective. The programmer has often in mind a certain location regardless of any coordinate (for example the top-left corner of a rectangle, the center of a page, or some `PathPoint` in a path) and what s/he actually needs is to *express* or *access* that location with respect to some coordinate system or any other convention. One may need to:

➜ Identify that location before further processing, e.g. for the purpose of analyzing the geometry of a spline item.

➜ Compare that location with another one while solving questions like *"Where is this object relative to that one?"*

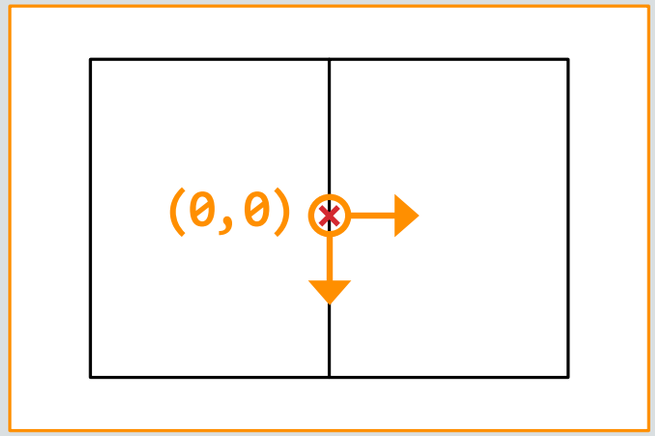
➜ Use that location as a temporary origin during a transformation, for instance when scaling or rotating objects around a point. And so on.

Thus, a location is much more than a simple pair of numbers. Better is to understand it as a determined *somewhere* in the layout, based on existing objects and processed through coordinates when it finally comes to calculate or compare numeric positions.

In that sense a same location obviously has various expressions, since InDesign provides several coordinate spaces and systems. Thus, two distinct (x, y) pairs may describe the same destination point in the device space. For example—see **Figure 25**—the origin of a balanced spread is `(0,0)` in its associated coordinate space, but the *same location* is described as well by the coordinates `(0.5,0.5)` in the inner box system[1] of that spread.

Ultimately the entire problem is to provide the Scripting DOM with an exact specification of the locations you consider and, when needed, to perform the appropriate from/to conversion between coordinates. Our first task is therefore to explore the available methods for specifying a location in InDesign.

1. Indeed, the center point of the spread area is `(0.5,0.5)` in bounding box coordinates and it usually coincides with the origin of the spread coordinate space. This statement might be wrong in non-balanced spreads though, because the facing-page mode may impact the location of the coordinate space origin.



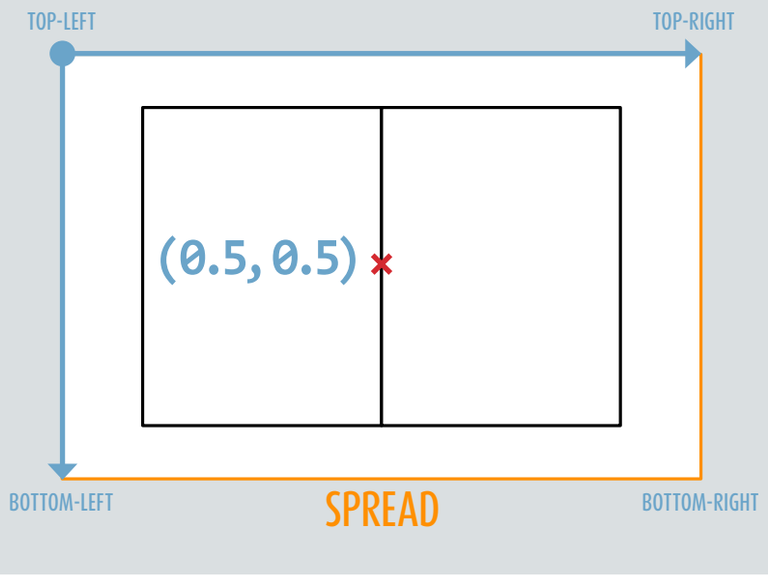


Figure 25.
A same location (red cross at the center of the spread) can be expressed by different coordinates depending on the system we consider,
`(0,0)` in the spread coordinate space *(top)*,
`(0.5,0.5)` (i.e. centerAnchor) in the spread bounding box system *(bottom)*.

4. Resolving Locations

Location Specifiers

InDesign's subsystem offers exactly three distinct ways to define a location. In the SDK[2] these are referred to as TRANSFORM-SPACE location, BOUNDS-SPACE location, and RULER-SPACE location. In this chapter we will abbreviate them respectively T-SPECIFIER, B-SPECIFIER, and R-SPECIFIER.

➔ TRANSFORM-SPACE location (T-SPECIFIER) is the easiest case. It defines a location *relative to a coordinate space*. So, given a coordinate space and a coordinate pair (x,y), this method simply allows to target the point `[x,y]` in that space, for example the point `[0,0]` in the PASTEBOARD space, or the point `[3,-2]` in the PARENT space of a page item, etc. (Details on InDesign coordinate spaces are discussed in Chapter 2.)

➔ BOUNDS-SPACE location (B-SPECIFIER) is probably the most practical case. It defines a location *relative to a bounding box coordinate system* (see Chapter 3.) This specifier is very powerful as it integrates the features attached to any bounding box, *(a)* the related coordinate space, *(b)* whether the *path* box or the *visible* box is under consideration, *(c)* the ability to supply coordinates either as numeric pairs (u,v)[3], e.g. `[0.5,1]`, or as predefined anchors, e.g. *bottom-center-anchor*. For example, given an oval (or any spline item,) a B-SPECIFIER will allow to target the top-left corner of the bounding box seen in the perspective of the parent space and including the stroke weight of the object.

➔ RULER-SPACE location (R-SPECIFIER) finally defines a location *with respect to the current rulers and preferences* in the GUI. This takes into account the active measurement units, the custom "zero point" and the option **`RulerOrigin`** (page vs. spread vs. spine origin) exposed in the **`ViewPreference`** object. As this special coordinate system hasn't been explored yet, we shall study it in more detail soon. Basically, a R-SPECIFIER involves coordinates (r_x, r_y) in ruler units and relative to the zero point—as seen in the Transform panel—but it also involves either a spread or a page specification, since rulers are spread- or page-dependent.

The table below (**Figure 26**) summarizes for each location specifier the parameters we have mentioned so far.

It is worth noting that the SDK makes no distinction between a "transform space" and a regular coordinate space, reminding us that *transformations only occur through these primary spaces*. The additional coordinate *systems* (bounding boxes and rulers) are just helpers in relation with the root mechanism.

LOCATION SPECIFIER	COORDINATE SYSTEM	COORDINATES	RELATED DOM OBJECTS	OPTIONS
T-SPECIFIER *(transform-space)*	Coordinate space.	Any $[x,y]$ pair.	The Spread, Page, or PageItem that determines the coordinate space (note: the pasteboard space can be reached from any DOM object.)	
B-SPECIFIER *(bounds-space)*	Bounding box.	Any $[u,v]$ pair or, alternately, any predefined anchor point.	The Spread, Page, or PageItem whose bounding box is considered (with respect to the options below.)	• The coordinate space which the box is framed in. • The box limits (*visible* vs. *path* bounds.)
R-SPECIFIER *(ruler-space)*	The GUI rulers.	Any $[r_x, r_y]$ pair in current ruler units (optionally in points.)	The Spread or Page which the rulers are attached to (according to the RulerOrigin preference.) In fact this parameter is implied from a child PageItem and other arguments (either a page index or an additional location specifier.)	• Ability to provide $[r_x, r_y]$ in points instead of ruler units (*consideringRulerUnits* flag.)

Figure 26.
There are three distinct ways of defining a location in InDesign. The first (T-Specifier) simply relies on regular coordinate spaces. The second (B-Specifier) makes use of bounding box coordinates. The last (R-Specifier) involves the rulers with respect to their current state and settings.

2. My main source here is the `LocationSpace` structure and the `TransformOrigin` class defined in the header file *TransformTypes.h* (InDesign SDK.) As often, source code and developers' comments are our best hints to investigate obscure topics. Adobe also released in 2007 a PDF "Working With Transformations in Javascript" (InDesign CS3 Scripting) that brought additional clues on how location specifiers are ported from the subsystem API into the Scripting DOM. (In that particular field the basic scripting reference, as well as the ESTK help, are useless.)

3. To prevent confusions between the unit length in coordinate spaces (that is, PostScript point) and the special one used in bounding box systems, we conventionally use the variables (x,y) in the former case vs. (u,v) in the latter case.

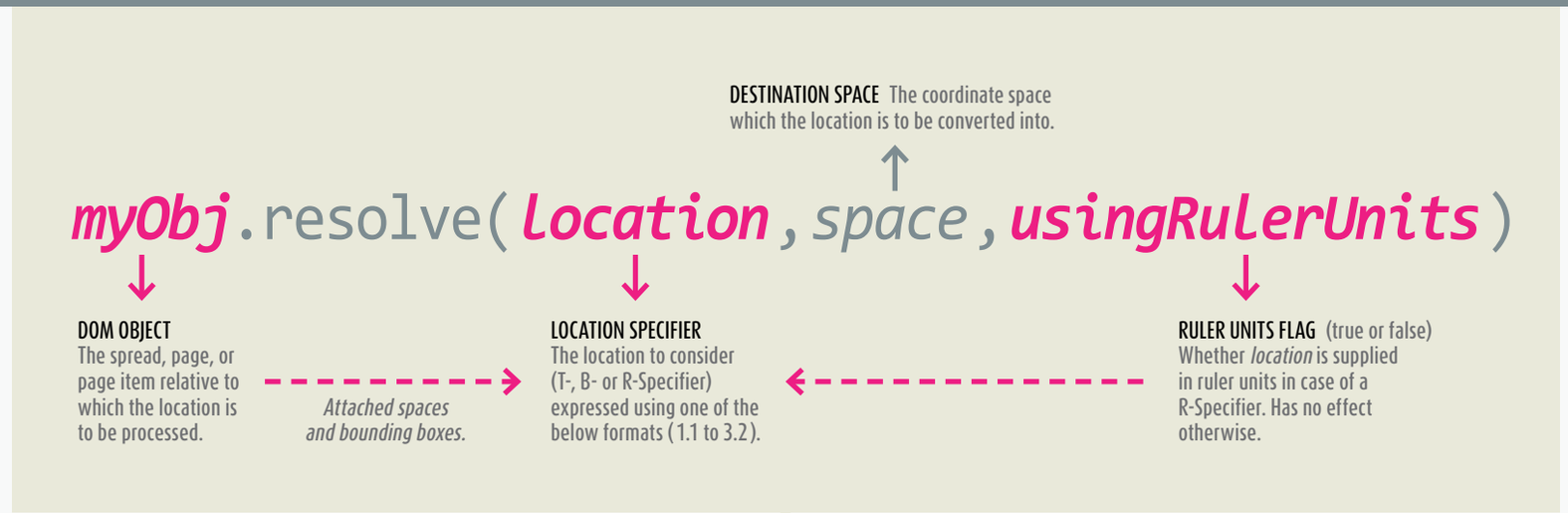


Figure 27.
All layout objects (incl. spreads and pages) expose a **resolve()** method which plays a crucial role in solving location issues. It takes a location specifier of any kind (with respect to the incoming object) and returns a singleton array having for unique element a coordinate pair [*x*, *y*] expressed in the destination space.

Syntax of a Location in ExtendScript

InDesign's scripting layer provides a few places where a LOCATION is expected as a formal argument,[4]

- ➔ *obj*.**resolve**(***location***, *space*, *usingRulerUnits*)
- ➔ *obj*.**transform**(*space*, ***originLocation***, ...)
- ➔ *obj*.**resize**(*space*, ***originLocation***, ...)

where *obj* refers to a **Spread**, a **Page**, or a **PageItem**.

The official documentation defines this parameter (*location* or *originLocation*) as follows: *"The location requested. Can accept: Array of 2 Reals, AnchorPoint enumerator or Array of Arrays of 2 Reals, CoordinateSpaces enumerators, AnchorPoint enumerators, BoundingBoxLimits enumerators or Long Integers."*

If you don't understand this gibberish, keep calm, this is normal reaction! The Scripting DOM supports all location specifiers but the expected syntax is so polymorphic that no developer can decipher it without further explanation.

Here is (finally revealed!) the complete syntax:

1. Location as a T-Specifier

1.1 **[x,y]**
Coordinates in the pasteboard space.

1.2 **[[x,y],<COORD_SPACE>]**
Coordinates in the specified coordinate space.

2. Location as a B-Specifier

2.1 **<ANCHOR_PT>**
AnchorPoint *in the visible inner box system.*

2.2a **[<ANCHOR_PT>,<BOX_LIMITS>]**
AnchorPoint *in the inner box system, considering the specified* **BoundingBoxLimits**.

2.2b **[[u,v],<BOX_LIMITS>]**
Coordinates in the inner box system, considering the specified **BoundingBoxLimits**.

2.3a **[<ANCHOR_PT>,<BOX_LIMITS>,<COORD_SPACE>]**
AnchorPoint *in the bounding box system framed in the specified coordinate space and considering the specified* **BoundingBoxLimits**.

2.3b **[[u,v],<BOX_LIMITS>,<COORD_SPACE>]**
Coordinates in the bounding box system framed in the specified coordinate space and considering the specified **BoundingBoxLimits**.

3. Location as a R-Specifier

3.1 **[[rx,ry],<PAGE_INDEX>]**
• *Coordinates in the ruler system attached to the* **Page** *specified by* PAGE_INDEX *(index in the parent spread) in case* **RulerOrigin** *is pageOrigin.*[5]
• *Otherwise, coordinates in the spread- or spine-based ruler system,* PAGE_INDEX *having no effect.*[6]

3.2 **[[rx,ry],<PAGE_LOCATION>]**
Coordinates[7] *in the ruler system attached to the* **Page** *that contains* PAGE_LOCATION *(in case* **RulerOrigin** *is pageOrigin.)* PAGE_LOCATION *is formatted as a* B-SPECIFIER *using any of the* 2.x *syntaxes, without the outer brackets.*[8]

---

4. We could also mention **geometricBounds** and **visibleBounds**, as well as **Path.entirePath** and **PathPoint**'s positioning properties (**anchor**, **leftDirection**, **rightDirection**), but those locations have a limited syntax which, unfortunately, *only supports the ruler system*.

5. The coordinates [r_x,r_y] depend on the "zero point" and are interpreted in ruler units if *usingRulerUnits*==true. Note also that if **RulerOrigin.spineOrigin** is active, the "zero point" has a fixed location which the user cannot change.

6. However, the <PAGE_INDEX> parameter is still required to comply with the 3.1 syntax! In such case you can use 0 as a fake index.

7. Here again the coordinates are interpreted in ruler units (resp. in points) if *usingRulerUnits*==true (resp. false.)

8. For example, here is a 3.2 specifier in the form [[rx,ry],2.2b], [[10,20], [0.5,1],BoundingBoxLimits.geometricPathBounds].

Understanding resolve()

The `resolve()` method (see **Figure 27**) is a good starting point for experimenting location specifiers. You can use it to study and convert locations from any kind into T-SPECIFIER coordinates.

In most cases you will call `resolve()` from a **`PageItem`**, but it is also available in **`Graphic`**, **`Spread`** and **`Page`** APIs.[9] The "calling object" is of course very important since it brings the *source* from which coordinate spaces or bounding boxes are referred to. For example

```
mySpread.resolve(AnchorPoint.topLeftAnchor,
CoordinateSpaces.parentCoordinates)[0]
```

targets the top-left location of *mySpread*'s inner box (syntax `2.1`) and returns the coordinates in *mySpread*'s parent space (that is, in the pasteboard space.)

By contrast,

```
myOval.resolve(AnchorPoint.topLeftAnchor,
CoordinateSpaces.parentCoordinates)[0]
```

targets the top-left location of *myOval*'s inner box (bounding box of the object *in its own space*) and returns the coordinates in *myOval*'s parent space—which might be either the spread space or the coordinate space of a **`PageItem`** in case *myOval* belongs to a **`Group`** or is nested into another container.

Hence the caller fully governs the meaning of the parameters, as shown in **Figure 28**.

Despite its high flexibility regarding inputs, the main limitation of *`obj`*`.resolve(...)` is that it can only output pure transform-space coordinates.

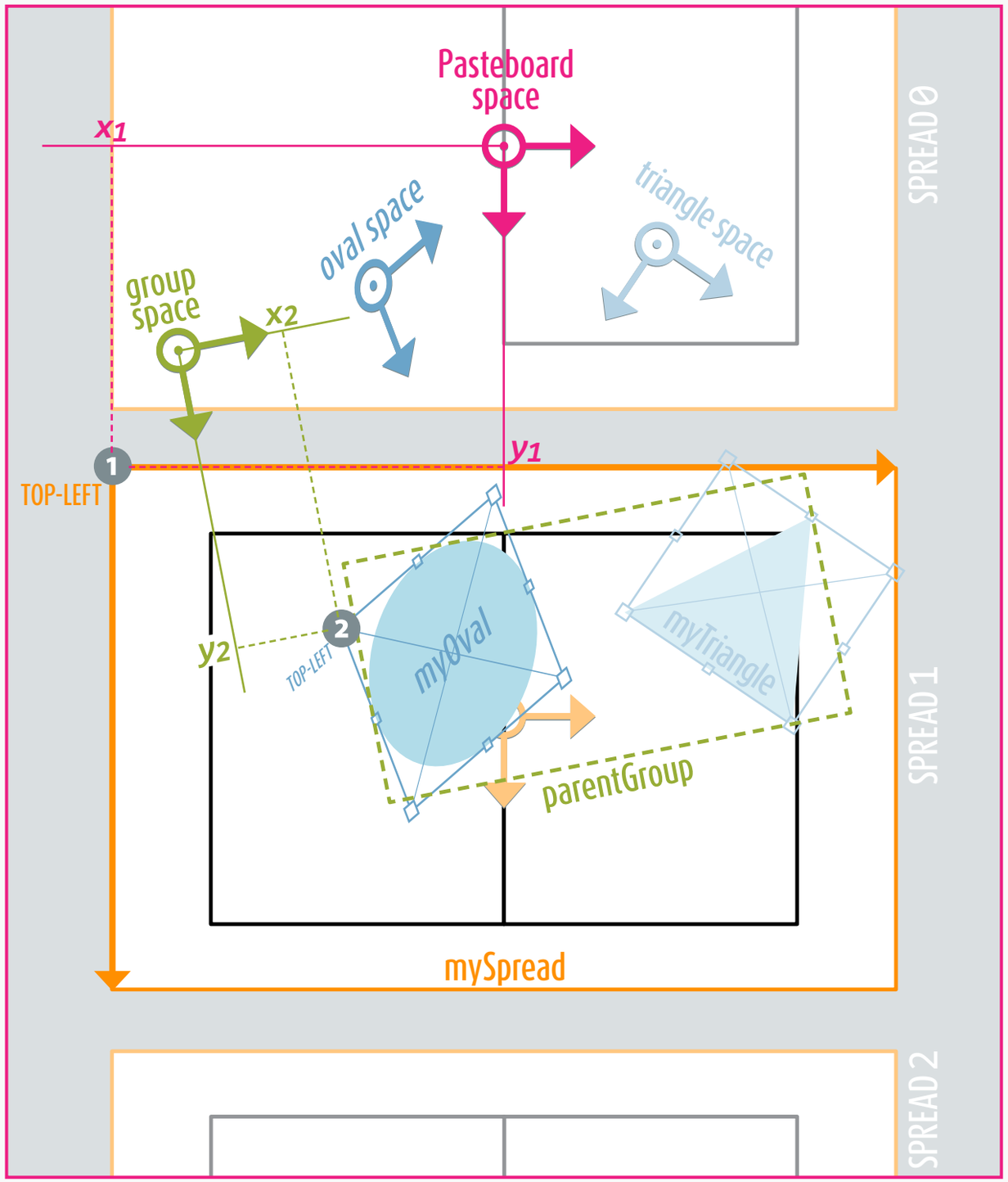


Figure 28. In this layout various transformations have been applied to the underlying coordinate spaces (oval, triangle, and parent group), and their origins have been randomly positioned to make it clear that they don't necessarily coincide.

1. The location ❶ is specified as the top-left anchor of *mySpread* inner box using the syntax `mySpread.resolve(ANCHOR_PT, ...)` This location is returned in the parent space (pasteboard) if `CoordinateSpaces.parentCoordinates` is provided as second argument. The result is [[x_1,y_1]].

2. The location ❷ is specified the same way as the top-left anchor of *myOval* inner box: `myOval.resolve(ANCHOR_PT, ...)` But since *myOval* belongs to a group (*parentGroup*) the parent space `CoordinateSpaces.parentCoordinates` here refers to the coordinate space associated to the group. The result is [[x_2,y_2]].

9. **`Page`**`.resolve()` has been added in InDesign CS4 (6.0.)

So if you need to convert say a B-SPECIFIER into a R-SPECIFIER (or into another B-SPECIFIER based on a distinct coordinate space), some extra calculations are required.

In such case the magic workaround would be:

1. Select a coordinate space as a reference, e.g. the pasteboard space or a common spread space.
2. Translate the input specifier *inLoc* into reference space coordinates (x,y) using the scheme
   `xy = obj.resolve(inLoc,refSpace...)[0]`.
3. Find the output specifier *outLoc* that would *also* translate into (x,y). That is, find *outLoc* such that
   `xy = obj.resolve(outLoc,refSpace...)[0]`.

You can then conclude that *outLoc* targets the same location than *inLoc*, which is what you were looking after.

Problem is that step 3 is not as easy as it sounds! At this stage the known variable is `xy` (array `[x,y]`) and the unknown is *outLoc*, so we want to use the `resolve()` command backwards. Let's study this problem.

Resolving Locations: The "T2B" Case[10]

Consider a T-SPECIFIER in whatever coordinate space. Its most general form[11] is `[[x,y],refSpace]`, relative to some layout object *myObj*. Our goal is to convert this location into a B-SPECIFIER, that is, into a coordinate pair `[u,v]` relative to one of the bounding boxes attached to *myObj*. (That's the T2B conversion case.)

10. T2B abbreviates "T-Specifier to B-Specifier" conversion.

11. The particular case `[x,y]` (syntax `1.1`) is just a shortcut of `[[x,y], CoordinateSpaces.pasteboardCoordinates]` (syntax `1.2`.)

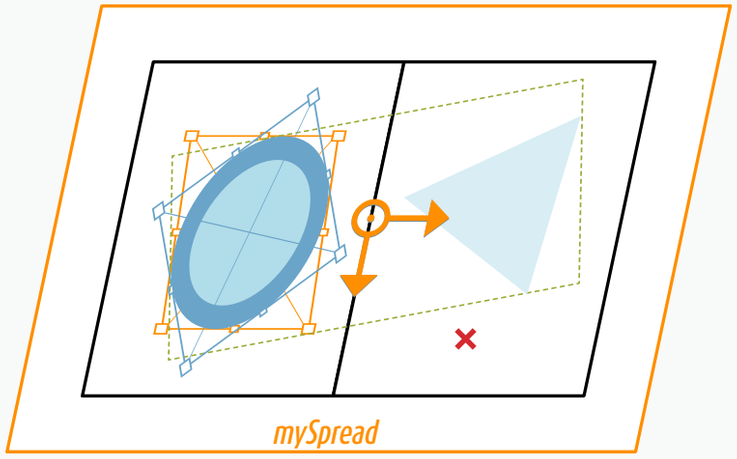


Figure 29.
The best strategy for solving location issues is to suppose all coordinate spaces are in a nontrivial transform state relative to each other along the hierarchy. This way one can discern parameters and relationships that would otherwise be coincident and unnoticeable. In this layout both the oval, the triangle and the parent group have distinct rotation or shear applied, and even the spread container is assumed skewed in the pasteboard space.

Again, the most general form of the B-SPECIFIER is `[[u,v], boxLimits,boxSpace]` (syntax `2.3b`) since any other syntax is a shortcut where either default *boxSpace* and/or default *boxLimits* are implied. Also, every predefined anchor point has a direct expression as a coordinate pair `[u,v]` in the range `[0..1,0..1]`.

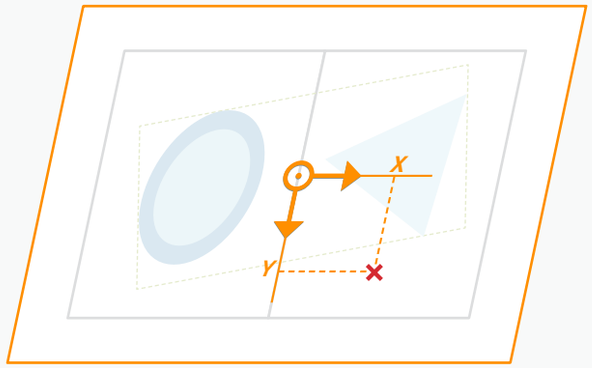
The whole question is therefore to implement a function that takes *myObj*, `[x,y]`, *refSpace*, *boxLimits*, and *boxSpace*, then outputs `[u,v]`.

For instance, focusing on the location represented by the red cross ✕ in Figure 29, we could have to handle the following input parameters,

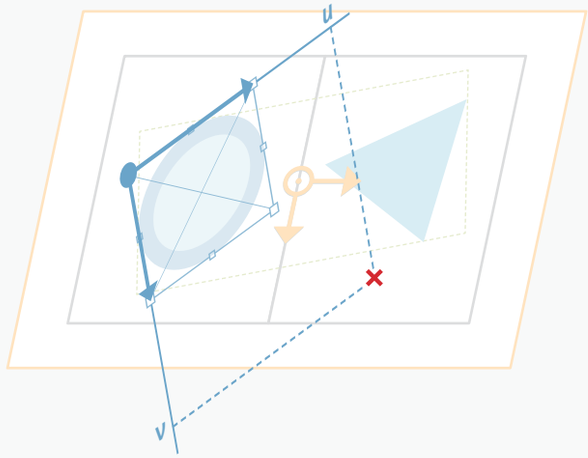
myObj	The blue oval (child of a group)
`[X,Y]`	The red cross coordinates in *refSpace*
refSpace	`CoordinateSpaces.spreadCoordinates`
boxLimits	`BoundingBoxLimits.outerStrokeBounds`
boxSpace	`CoordinateSpaces.innerCoordinates`,

then to compute and return `[u,v]` i.e. the coordinates of the red cross *relative to the visible inner box of the oval* (blue frame.)[12]

In short, here are the coordinates we already have:



and here are those we are looking for:



In terms of transformation mapping we are to determine a matrix M such that $(u,v) = (X,Y) \times M$.

Two important facts will help us. First, InDesign can easily get the matrix that maps *boxSpace* (the inner

12. Note the distinction between *refSpace*, the coordinate space in which (x,y) is provided, and *boxSpace*, the coordinate space that determines the bounding box of interest. Setting *boxSpace* as *refSpace* (spread) would lead to compute (u,v) relative to the orange frame instead. See Chapter 3 for more detail on bounding boxes.

coordinate space of the oval) to *refSpace* (the spread coordinate space, in our example.) This *boxToRefMx* matrix is given by:[13]

```
boxToRefMx=myObj.transformValuesOf(refSpace)[0];
```

The second important fact it that a bounding box system, although not being a coordinate space, always has the same *orientation* than the underlying coordinate space, meaning that neigher ROTATION nor SHEAR component is involved in the matrix that maps the box *space* to the box *system*.[14] In other words, there is a scaling matrix S and a translation matrix T such that

(1) $(u,v) = (x,y) \times S \times T$,

where (x,y) refer to coordinates in the box *space*, (u,v) being the corresponding coordinates in the box *system*.

➔ Let (t_x,t_y) be the T parameters and (s_x,s_y) be the scaling factors of S. We can then rephrase (1) as follows:

(1*a*) $u = x.s_x + t_x$,

(1*b*) $v = y.s_y + t_y$.

➔ Let TL be the top-left anchor of the box. We have

(2*a*) $u_{TL} = 0 = x_{TL}.s_x + t_x$, *according to (1a)*

(2*b*) $v_{TL} = 0 = y_{TL}.s_y + t_y$, *according to (1b)*

where (x_{TL}, y_{TL}) are easily determined using `myObj.resolve(<TOP_LEFT_LOC>,boxSpace)[0]`.

➔ Let BR be the bottom-right anchor of the box. We have

(3*a*) $u_{BR} = 1 = x_{BR}.s_x + t_x$, *according to (1a)*

(3*b*) $v_{BR} = 1 = y_{BR}.s_y + t_y$, *according to (1b)*

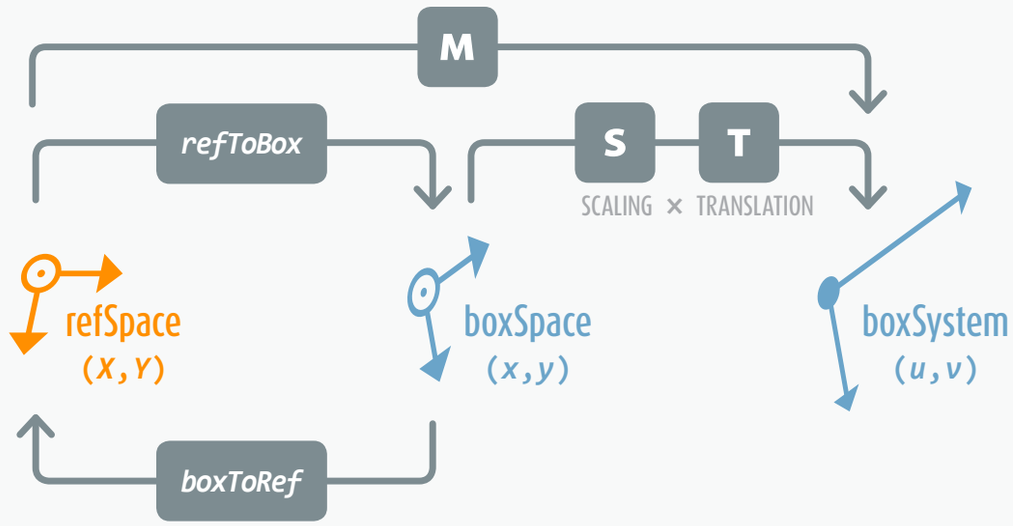


Figure 30.
What InDesign can instantly reveal us (through `transformValuesOf`) is the matrix *boxToRef* which maps an inner space (*boxSpace*) to any coordinate space (*refSpace*.) In order to find the matrix *M* that maps *refSpace* to a bounding box system (*boxSystem*) we need extra calculations. First we determine the scaling-and-translation matrix *S×T* that maps *boxSpace* to *boxSystem*. Then, by inverting *boxToRef*, we get a matrix *refToBox* that maps *refSpace* to *boxSpace*. Finally, Chasles' Relation shows that *M = refToBox × S×T*.

where (x_{BR}, y_{BR}) are easily determined using `myObj.resolve(<BOTTOM_RIGHT_LOC>,boxSpace)[0]`.

The system of equations results in

(4) $s_x = 1/(x_{BR} - x_{TL})$, $s_y = 1/(y_{BR} - y_{TL})$,

(5) $t_x = -s_x.x_{TL}$, $t_y = -s_y.y_{TL}$,

so S and T matrices are now fully determined.

Let's put together the data we have reached so far (see **Figure 30**.) The known matrix *boxToRefMx* maps the box space to the reference space. Using the notations above this translates into

(6) $(X,Y) = (x,y) \times boxToRefMx$.

The known matrix $S \times T$, i.e. $(s_x,0,0,s_y,t_x,t_y)$, maps the box space to the box system, that is,

(7) $(u,v) = (x,y) \times S \times T$.

And we are looking for a matrix M that satisfies

(8) $(u,v) = (X,Y) \times M$.

Using equalities (7) and (6) one can rewrite (8) as follows:

$$(x,y) \times S \times T = (x,y) \times boxToRefMx \times M,$$

which must remain true whatever (x,y). It follows:

(9) $S \times T = boxToRefMx \times M$.

Since every valid transformation in InDesign is invertible, we can set a matrix *refToBoxMx* as the inverse of *boxToRefMx*, using the code[15]

```
refToBoxMx=boxToRefMx.invertMatrix();
```

Now by pre-multiplying each term of the equality (9) by *refToBoxMx*, it comes

(10) $refToBoxMx \times S \times T = M$

as $refToBoxMx \times boxToRefMx$ is the IDENTITY matrix.

I detailed the whole demonstration in order to highlight what to do in terms of scripting commands, but the fact that M is equal to $refToBoxMx \times S \times T$ was quite obvious from the Chasles' Relation perspective.[16] Indeed,

refToBoxMx maps the ref-space A to the box-space B,

$S \times T$ maps the box-space B to the box-system C, and

M maps the ref-space A to the box-system C,

so the identity simply results from $M_{AB} \times M_{BC} = M_{AC}$.

---

13. Indeed, we learned in Chapter 2 that the command `myObj.transformValuesOf(anySpace)` returns a singleton array whose unique element is a matrix that maps *myObj*'s inner space to *anySpace*.

14. This fact was stated in Chapter 3. More generaly, any coordinate *system* can be seen as the transformation of a regular coordinate *space*. The proof is left as an exercise for the reader.

15. If you compare a matrix with a path that *carries* one coordinate space to another, inverting a matrix is just like reverting that path.

16. See Chapter 1 for details on matrix product; see Chapter 2 (in particular, **Figure 20**) for details on Chasles' Relation.

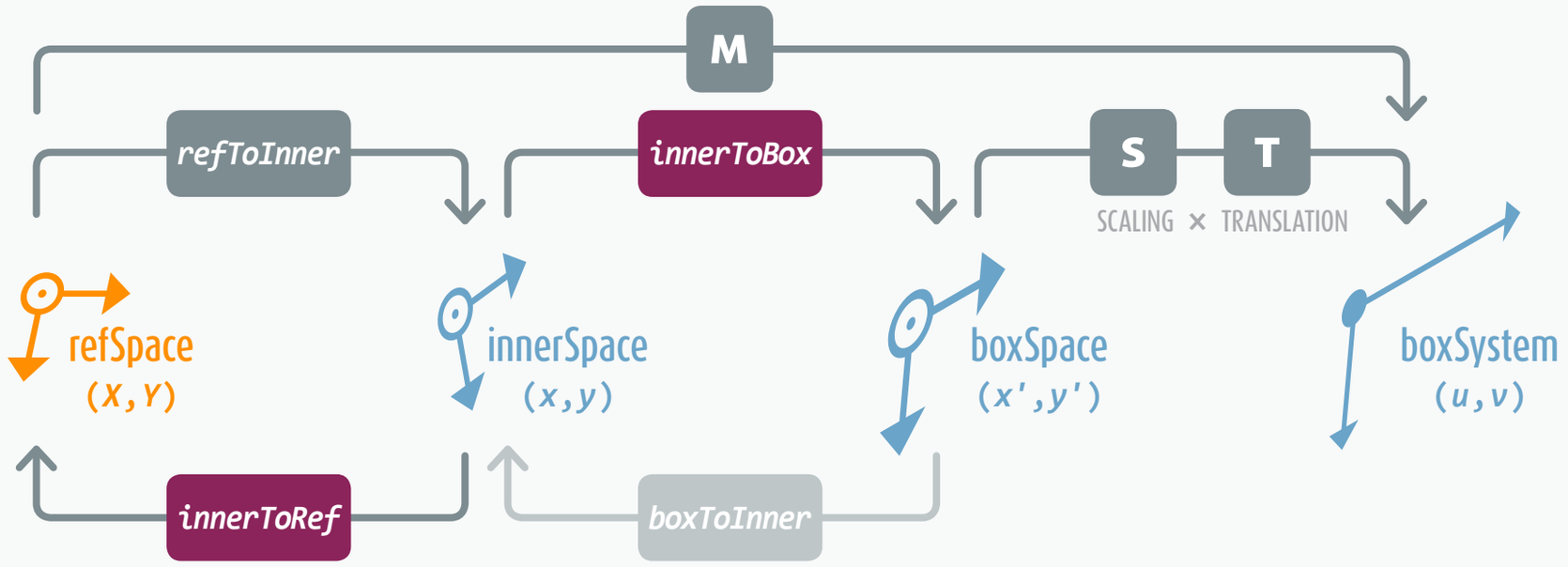


Figure 31.
Improved version of the "T2B" algorithm. It is not assumed anymore that the desired box space matches the inner space. In other words the bounding box can be observed from a different perspective, e.g. the pasteboard space. The whole transformation (`M`) now involves the following matrices: `refToInner` (the inverse of `innerToRef`), `innerToBox` (inner space to *boxSpace* mapping), and `S×T` as previously calculated (*boxSpace* to *boxSystem* mapping.) Matrices with purple background are those to which `transformValuesOf()` gives access.

Refining the "T2B" Algorithm

If you read in depth the previous Section, you may have noticed that we (deliberately!) neglected an important option. At the very beginning of the discussion we assumed that *boxSpace* (the bounding box space under consideration) was the inner space of the object. This was the case in our example, which made easy to determine the *boxToRef* matrix using `myObj.transformValuesOf(refSpace)`. The method `transformValuesOf()` is indeed designed to take the inner space, and only this one, as its input space, so it always returns an *inner-to-any* transformation matrix.

But in the most general case, we may have to convert *refSpace* coordinates into *any* of the available box systems, related to either inner, parent, page, spread, or pasteboard space. Therefore, if *boxSpace* does not refer to the inner space, an intermediate matrix is required for properly mapping the whole transformation, as shown in Figure 31.

It is easy to see that our previous *refToBox* matrix must now be decomposed *refToInner*×*innerToBox*, where *refToInner* maps *refSpace* to the inner space, while *innerToBox* maps the inner space to *boxSpace*.[17]

Implementation Notes. — The code of the function `resolveToBoxSys()` (see next page) faithfully translates into ExtendScript the algorithm we have discussed.

The required arguments are *obj*, *refSpace* and *XY* (the input object and a coordinate pair in the reference space.) The parameters *boxLimits* and *boxSpace* are made optional, default values being set respectively to `BoundingBoxLimits.outerStrokeBounds` and `CoordinateSpaces.innerCoordinates`, so the function returns $[u,v]$ relative to the *visible inner box* if other parameters are not explicitly provided.

17. In case `boxSpace==innerSpace`, the expected *innerToBox* matrix would be of course the IDENTITY matrix. Which is ensured by the fact that `myObj.transformValuesOf(CoordinateSpaces.innerCoordinates)[0]` always returns the IDENTITY `(1,0,0,1,0,0)`.

The method `resolve()` is invoked twice in order to convert the desired locations (top-left and bottom-right anchors, formatted as full B-SPECIFIERS) into *boxSpace* coordinates. This allows to determine the scaling and translation parameters s_x, s_y, t_x, t_y.

The last piece of code chains up all matrix operations to avoid the creation of temporary references. The `TransformationMatrix` API provides all we need for that purpose,

- ➔ `M.invertMatrix()` returns the inverse of M,
- ➔ `M1.catenateMatrix(M2)` returns the product $M_1 \times M_2$,
- ➔ `M.scaleMatrix(sx,sy)` returns the product $M \times S$ where S is the SCALING matrix `(sx,0,0,sy,0,0)`,
- ➔ `M.translateMatrix(tx,ty)` returns the product $M \times T$ where T is the TRANSLATION `(1,0,0,1,tx,ty)`,
- ➔ `M.changeCoordinates([x,y])` applies the matrix to (x,y)[18] and returns the final coordinate pair.

18. That is, in terms of matrix product, $[x \;\; y \;\; 0] \times M$.

```
// 05. T2B ALGORITHM
const resolveToBoxSys = function(obj, refSpace, XY, boxLimits, boxSpace)
// ---------------------------------------------------------------------
// Converts refSpace coordinates, XY, into box system coordinates [u,v].
// ---
// obj       :: a DOM object that supports resolve (PageItem,Graphic,Spread...)
// refSpace  :: a coordinate space, e.g CoordinateSpaces.spreadCoordinates,
// XY        :: coordinates in refSpace (array of two numbers), e.g [3,5],
// boxLimits :: [OPT] a BoundingBoxLimits enum, default: .outerStrokeBounds,
// boxSpace  :: [OPT] coordinate space of the box, default: .innerCoordinates.
{
    // Defaults
    // ---
    boxLimits || (boxLimits = BoundingBoxLimits.outerStrokeBounds);
    boxSpace  || (boxSpace = CoordinateSpaces.innerCoordinates);

    // Scaling and translation params (boxSpace -> boxSystem)
    // ---
    var xyTL = obj.resolve([[0,0],boxLimits,boxSpace],boxSpace)[0],
        xyBR = obj.resolve([[1,1],boxLimits,boxSpace],boxSpace)[0],
        sx = 1/(xyBR[0]-xyTL[0]),
        sy = 1/(xyBR[1]-xyTL[1]),
        tx = -sx*xyTL[0],
        ty = -sy*xyTL[1];

    // Get the result.
    // ---
    return obj.
        transformValuesOf(refSpace)[0].invertMatrix().      // REF -> INNER
        catenateMatrix(obj.transformValuesOf(boxSpace)[0]). // INNER -> BOX
        scaleMatrix(sx,sy).translateMatrix(tx,ty).          // BOX -> SYS
        changeCoordinates(XY);                              // (X,Y) => (u,v)
};
```

Figure 32. Implementation of the T2B algorithm. This function can translate any T-Specifier into the B-Specifier of your choice. Given a coordinate pair [*X, Y*] in whatever coordinate space (*refSpace*), it returns the same location expressed as a coordinate pair [*u, v*] in the bounding box system associated to *boxLimits* and *boxSpace*.

What will make the T2B algorithm an essential brick among your scripting tools is that no native DOM method returns B-SPECIFIERS while bounding boxes are probably the most natural entities for dealing with locations. Alas, the built-in `resolve()` method only takes B-SPECIFIERS as *inputs*. We now have a round trip bridge between T-SPECIFIERS and B-SPECIFIERS.

The function below (**Figure 32**) will help you answer questions like, *Where is this coordinate space location relative to that bounding box? Does this (x, y) point "belong" to the box area of that spline item?* And so on.[19]

This algorithm also brings a general pattern that one can re-use in similar problems. All is about "chaining" matrices in a consistent way from the input space to the output space (see the `return` statement.)

Note that the coordinates `XY` are processed only at the very last line. If we remove that line (and then the `XY` argument), the function will return the T2B matrix itself, which can be stored in a variable for the purpose of calling `changeCoordinates()` at different locations. Keep this trick in mind if you plan to embed the algorithm as a module in a wider project.

InDesign's Ruler System

Before we go further in processing R-SPECIFIERS we need additional hints on how rulers work in InDesign. Unlike the coordinate spaces and the bounding box systems—which are context-independent and therefore very secure from a scripting standpoint—the ruler

19. Also, *(u, v)* coordinates have a clear "meaning." We know `(0.5,0.5)` is the center point of the box and we can easily visualize locations like `(0.25,0.5)` or `(1/4,2/3)` even if they don't match the predefined set of anchor points.

system depends on preferences and user choices. In a perfect world script developers would prefer to guard against user whims. Unfortunately the Scripting DOM is deeply stuck to the rulers. Most basic properties and methods—such as `PageItem.geometricBounds`, `PageItem.move()`, `PathPoint.anchor`, and many others—involve the ruler system. Also, the Transform panel and related components display ruler-related coordinates and dimensions.

As long as you control document settings, measurement units, view preferences, and provided that no special transformation occurs in the layout, ruler coordinates remain reliable and easy to use.[20] But if you are automating tasks related to complex geometry, nested splines, IDML processing or similar stuff, a key rule is to address coordinates and transformations in the most *agnostic* way. Always assume the user is working in a rotated spread view, uses custom units and plays with skewed objects throughout non-uniformly resized pages, as in **Figure 33**!

Let's enumerate the parameters that make the ruler system so special:

➔ Unlike coordinate spaces it supports custom units, namely `ViewPreference.horizontalMeasurementUnits` and `ViewPreference.verticalMeasurementUnits`.[21]

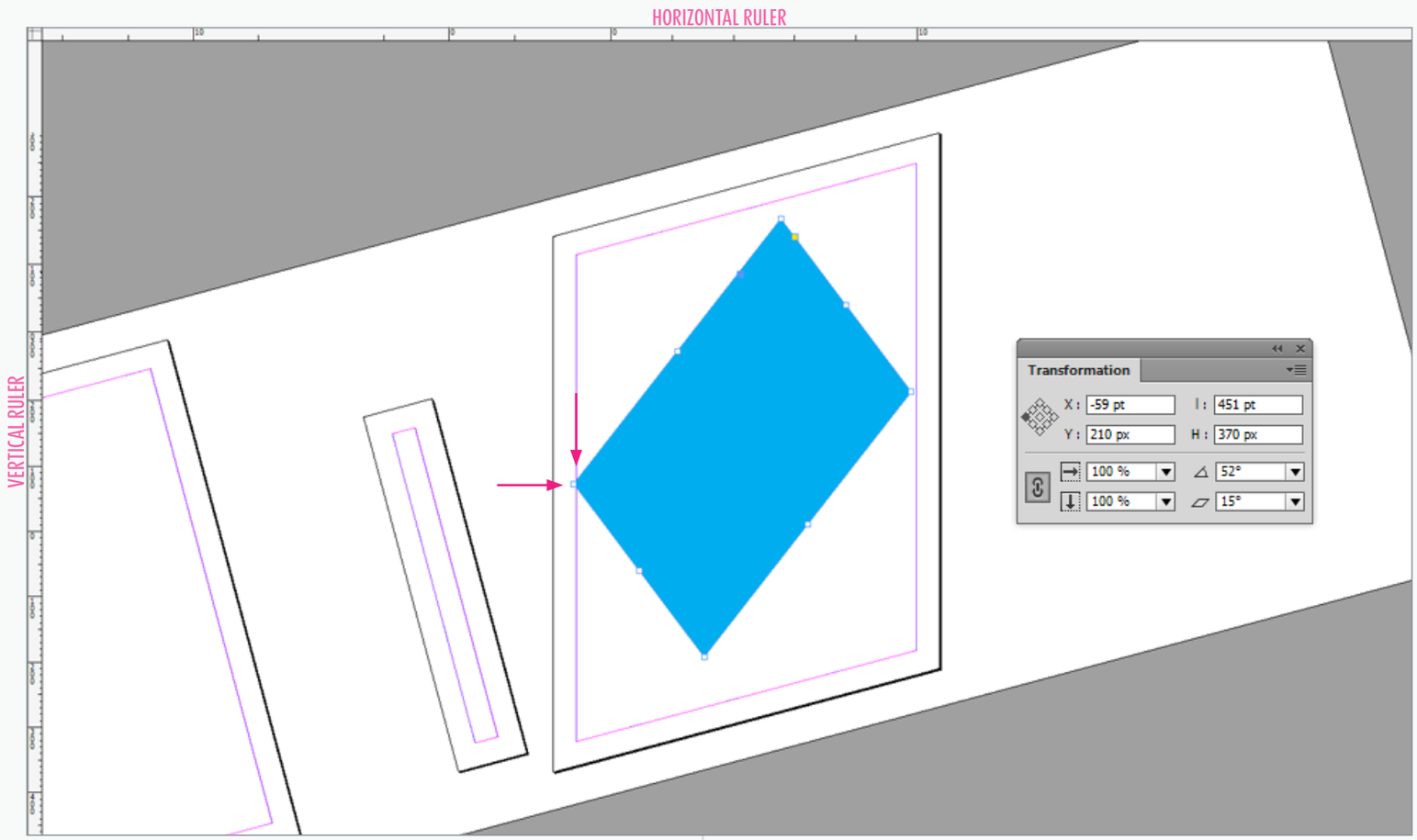


Figure 33. Screenshot of InDesign's viewport under odd settings (custom spread rotation, skewed page, custom units, randomly positioned Zero Point...) Problem now is to properly use ruler coordinates for parsing and processing locations, bounds, width, height of the blue rectangle!

➔ As a consequence, the rulers involve horizontal and vertical directions regardless of the transform state of the spread under consideration. For example, if a spread is `90°` CW rotated, the horizontal ruler (which carries *X* coordinates in the corresponding units) will in fact match the orientation of the Y-axis in the spread coordinate space! So, *in terms of orientation*, the ruler system seems *rigidly attached to the pasteboard space*. But even this rule may become wrong, as we shall see.

➔ InDesign rulers support a user defined origin *"specified as page coordinates in the format [x, y]"* via the property `Document.zeroPoint`. Adobe's documentation lacks exactness and accuracy on what the term *"page coordinates"* is supposed to refer to, since there is no apparent relationship between rulers' orientation and the transform state of the pages (see again **Figure 33**.)

➔ In fact, the *default origin location* of the ruler system depends on `ViewPreference.rulerOrigin`,

20. The overwhelming majority of available InDesign scripts relie on the ruler system. Hence, they properly work under some implicit assumptions about InDesign settings that could be easily broken in odd environments. Understanding this issue is the key for strengthening your scripts and making them useable at a larger scale.

21. In InDesign CS5 and later, the object `ScriptPreference` provides a property `measurementUnit` that allows to bypass GUI units and use those specified. (`ViewPreference` also offers useful additional properties: `strokeMeasurementUnits`, `typographicMeasurementUnits`, `textSizeMeasurementUnits`, etc.)

which opens three options:[22]

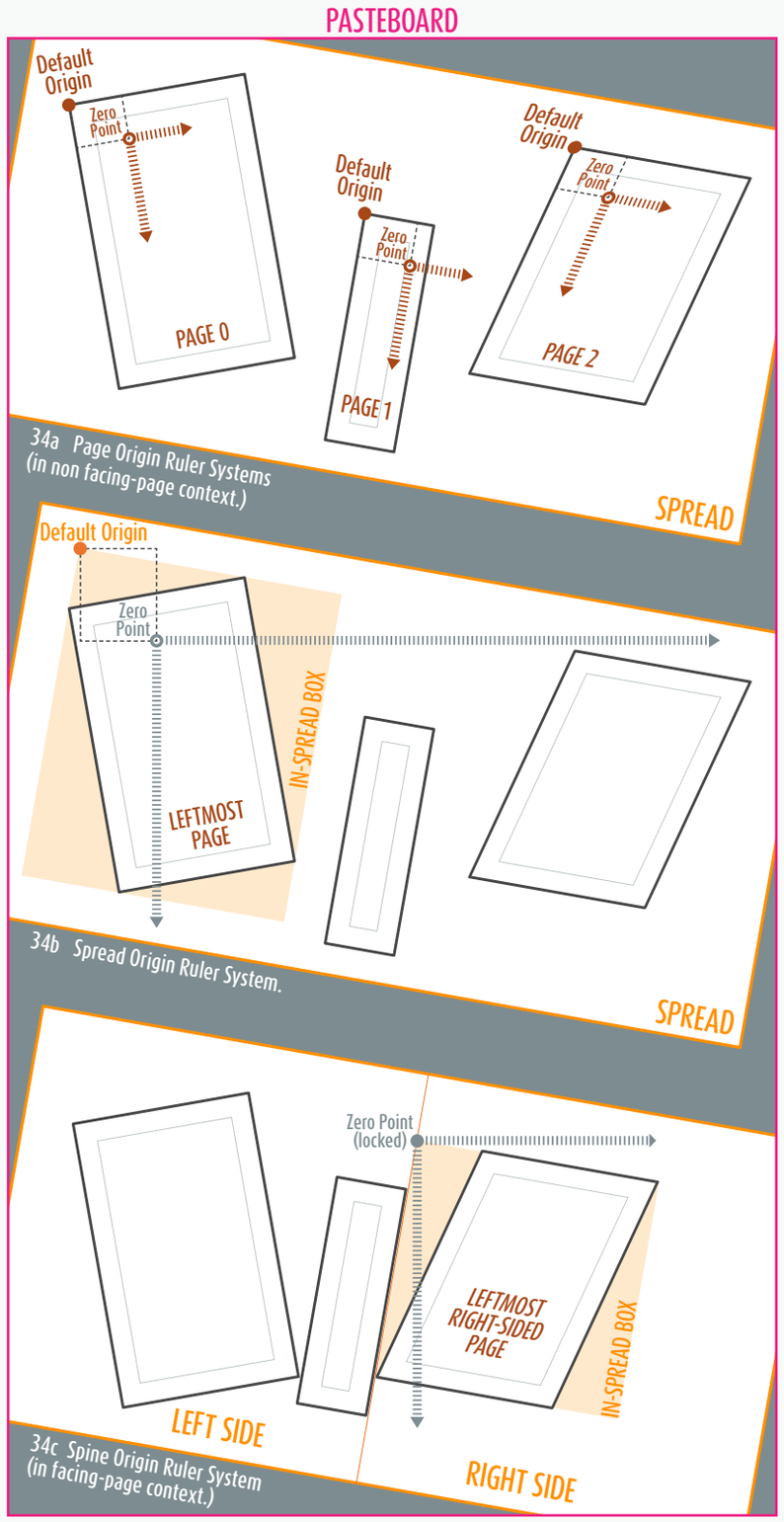
RulerOrigin	Default Origin Location	Base
pageOrigin	• *In non facing-page layout,* top-left corner of (the inner box of) each page. • *Otherwise,* top-left corner of the in-spread box of each page.	PAGE
spreadOrigin	Top-left corner of the in-spread box of the leftmost page.	SPREAD
spineOrigin *(Locked)*	• *In facing-page layout,* top-left corner of the in-spread box of the leftmost right-sided page. • *Otherwise,* top-left corner of the in-spread box of the leftmost page.	SPREAD

Note that whatever the **RulerOrigin** option, the default location of the origin (the default *zero point*) coincides with the top-left corner of a certain page bounding box. Should this page undergo some unusual transformation, that location would remain fully determined.

The **pageOrigin** case is very counterintuitive when **DocumentPreference.facingPages** is turned off. Here the *inner* bounding box of the page is considered, and it *determines the actual horizontal and vertical axes of the system*—even though the GUI tells you another story!

22. For the record, here is how the scripting reference describes these options. **RulerOrigin.pageOrigin**: *"the top-left corner of each page is a new zero point on the horizontal ruler."* **RulerOrigin.spineOrigin**, *"the zero point is at the top-left corner of the leftmost page and at the top of the binding spine. The horizontal ruler measures from the left-most page to the binding edge, and from the binding spine through the right edge of the right-most page. Also locks the zero point and prevents manual overrides."* **RulerOrigin.spreadOrigin**, *"the zero point is at the top-left corner of the spread and the ruler increments continuously across all pages of the spread."*

Figure 34. The ruler system in different modes.
a. Page Origin (in non-facing page layout),
b. Spread Origin,
c. Spine Origin (in facing-page layout.)



In all other cases, the *in-spread* bounding box of the page[23] is considered, and *axes are oriented as the pasteboard space*. Figure 34 shows these distinct systems.

➔ Finally, the [*x*,*y*] coordinates of the **Document.zeroPoint** property allows to reset[24] the origin relative to the *default* zero point, with respect to both the custom units and the horizontal and vertical orientations of the rulers. This results in what we call a RULER SYSTEM.[25]

Details About Page-Based Ruler Systems

The pageOrigin mode (Figure 34a) is undoubtedly the most complex configuration. In that mode, *each* page has a dedicated ruler system (while the whole spread has a single ruler system in spineOrigin or spreadOrigin mode.) Also, in non facing-page layouts, the actual orientation of the page rulers fits the inner space of the pages, although InDesign still displays

23. You may assume that there is no interesting distinction between the *inner box* and the *in-spread* box a a page. Most of the time, they just coincide. But they differ if the page undergoes e.g. a rotation or a skew relative to the spread. Then the top-left corner of the page (inner box) does not coincide with the top-left corner of the rectangle that encloses the page in the spread perspective (in-spread box.)

24. In *spine origin* mode, changing **Document.zeroPoint** has no effect on the current ruler origin. But the property is actually modified.

25. As discussed in Chapter 1, a coordinate system is entirely specified by a location (origin), two axes, and a unit length along each axis.

"horizontal" and "vertical" rulers aligned with the document window! Under special conditions the ruler coordinates (visible in the Transform panel) may therefore become quite obscure.

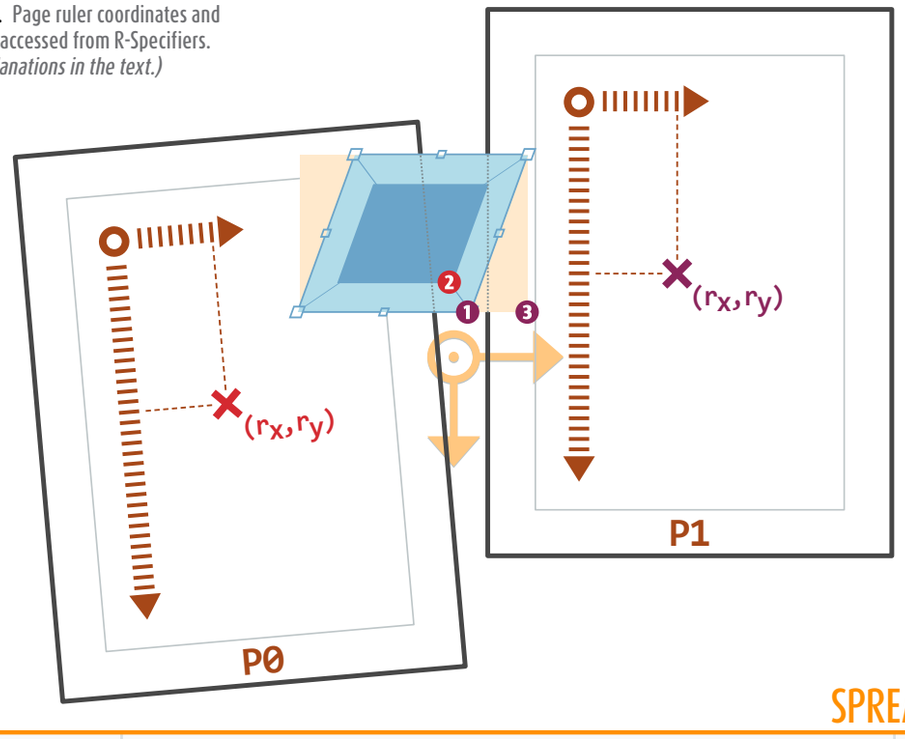
In addition, a single location in the spread can be expressed by different ruler coordinates: one for each page! A script could access a point on the *first* page using the ruler system of the *third* page. Conversely, when you retrieve `PathPoint` coordinates from a spline that overlaps multiple pages, the specific page which rulers are currently based on must be known.[26]

Any R-SPECIFIER (ruler-space location) expects either a determined `Page`, or "at least" a determined `Spread`. In the latter case there is no ambiguity about the spread under consideration, since every DOM method is triggered from an object which has a known, implied parent spread.

On the contrary, page-based ruler systems require *an identification of the page under consideration* in the implied spread. This explains the weird thoroughness of R-SPECIFIER's formats we have already mentioned:

3.1 `[[rx,ry],<PAGE_INDEX>]`,
3.2 `[[rx,ry],<PAGE_LOCATION>]`.

Figure 35. Page ruler coordinates and locations accessed from R-Specifiers. *(See explanations in the text.)*



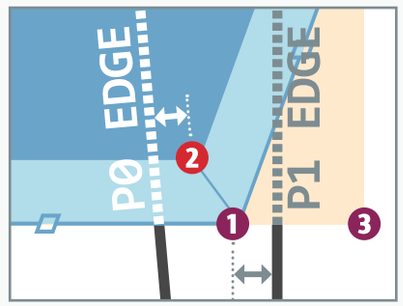
The first syntax (`3.1`) speaks for itself and will do the trick in almost every case. If a spread-based ruler system is active, `<PAGE_INDEX>` is nothing but a *formal placeholder*, so any index number (say `0`) can be passed in. Otherwise, `<PAGE_INDEX>` is of course the index of the page in the spread.

But the syntax `3.2` is much more sophisticated. Here InDesign expects a parameter, `<PAGE_LOCATION>`, formatted as a B-SPECIFIER without outer brackets, e.g. `AnchorPoint.bottomLeftAnchor` or `[0.75,0.5]`, `BoundingBoxLimits.geometricPathBounds`. Relative to the source object, this B-SPECIFIER points out to a location, which in turn determines a page. Which page? The nearest from the given location! And finally, the `[rx,ry]` coordinates are interpreted in the specific ruler system *of that page*.

To make this more concrete, consider the skewed rectangle in **Figure 35** and study the following bounding box locations: ❶ bottom-right corner of the *visible inner* box, ❷ bottom-right corner of the *geometric inner* box, ❸ bottom-right corner of the *visible inspread* box.[27]

Then, still assuming that `pageOrigin` is the active ruler mode, let's choose a coordinate pair (r_x, r_y) in the current horizontal and vertical units. Which location(s) will the R-SPECIFIERS `[[rx,ry],❶]`, `[[rx,ry],❷]`, and `[[rx,ry],❸]` address?

An obvious fact is that the same coordinate numbers are involved. But the final location is the *red cross* (page `P0`) in case ❷, while it is the *purple cross* (page `P1`) in cases ❶ and ❸. Indeed, locations ❶ and ❸ implicitly refer to page `P1` since it is the nearest (❸ belongs to it, and ❶ is closer to `P1`'s left edge than to `P0`'s right edge.) By contrast, ❷ refers to `P0`, as the location is now a bit closer to `P0`'s right edge *(see below.)*



26. This problem becomes critical when a script needs to supply ruler coordinates (as mostly expected by DOM entities and methods) while `pageOrigin` is selected. In absolute, a (r_x, r_y) pair has no meaning as long as the target page is unknown!

27. These B-SPECIFIERS can be expressed as follows,
❶ `AP.bottomRightAnchor`,
❷ `AP.bottomRightAnchor,BL.geometricPathBounds`
❸ `AP.bottomRightAnchor,BL.outerStrokeBounds,CS.spreadCoordinates`
using the abbreviations `AP=AnchorPoints`, `BL=BoundingBoxLimits`, and `CS=CoordinateSpaces`.