Coordinate Spaces & Transformations

in InDesign CS4 – CC | Oct. 2021 (3.2)

SCRIPTING SECRETS

Dealing with coordinate spaces and transformation matrices is one of the most obscure and underappreciated exercises in InDesign scripting and programming. The fault mainly lies with Adobe documentation, especially the Scripting DOM reference, which does not clearly explain the topic and some of its essential keys. This document attempts to shed some light on the beast.

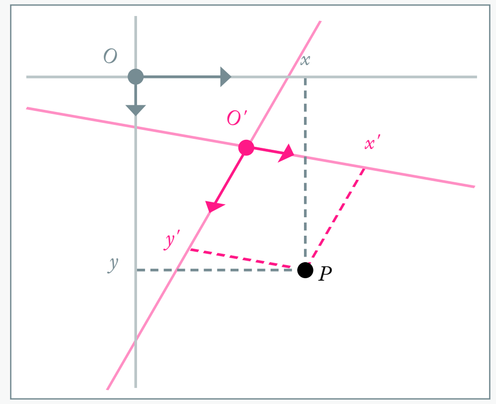


Figure 2. The same location **P** can be expressed by different coordinate pairs – (x, y) vs. (x′, y′) – depending on the coordinate system we consider.

2D Coordinate Systems

Within InDesign, the geometric location of a point is defined in terms of coordinates within a two-dimensional space. A coordinate is a pair of numbers (usually denoted *x* and *y*) that locate a point

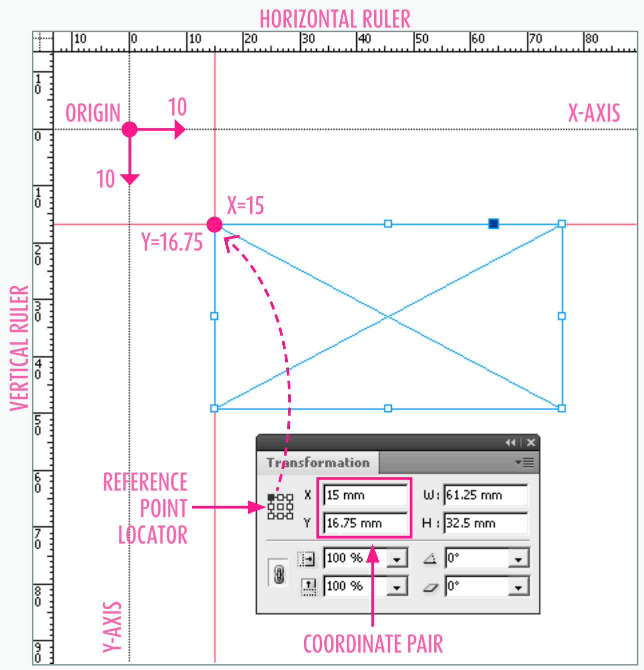


Figure 1. The ruler coordinate system makes it easy to check locations and measurements from the application interface.

relative to a given *origin*, according to the orientation of two given *axes* and with respect to the length of some *units* along each axis. These three parameters form a 2D COORDINATE SYSTEM.

InDesign handles multiple coordinate systems. A given location in the layout can be expressed by different coordinate pairs depending on the system. Users can easily experience how coordinates and measurements vary when playing with on-screen rulers, changing measurement units, moving the origin or the Reference Point (cf. **Figure 1**). The actual position and size of layout elements do not change, but both the Control panel, the Info panel and the Transform panel accordingly update the coordinates of the objects and other related values such as width and height of page items. Special display settings (e.g. *Show Content Offset* and *Dimensions Include Stroke Weight*) also affect how measurements display in the application interface.

Affine Maps

Every coordinate system is somewhat *arbitrary*. Whatever the origin, the units and the orientations of the axes, we can point out to the same geometric point, or path, by simply adjusting the coordinates to the desired coordinate system (cf. **Figure 2**). In other words, we can *convert* any coordinate pair from one system into another.

Fortunately such *conversion* is easy to describe in mathematical terms (no matter what coordinate systems or points we are considering). The functional relationship between two coordinate systems is known as an AFFINE MAP. An important property of any affine map is that it can always be entirely defined by an array of six real numbers. (We'll talk more about these a bit later.)

Relative Locations & Inner Space

When rendering graphics, paths and frames, InDesign needs to address their final locations according to various parameters. Some are *extrinsic* (e.g. screen resolution, parent window size, zoom factor, scrolling state), other are *intrinsic* in that they specify the inner geometry of the layout items and their relationships within the publication.

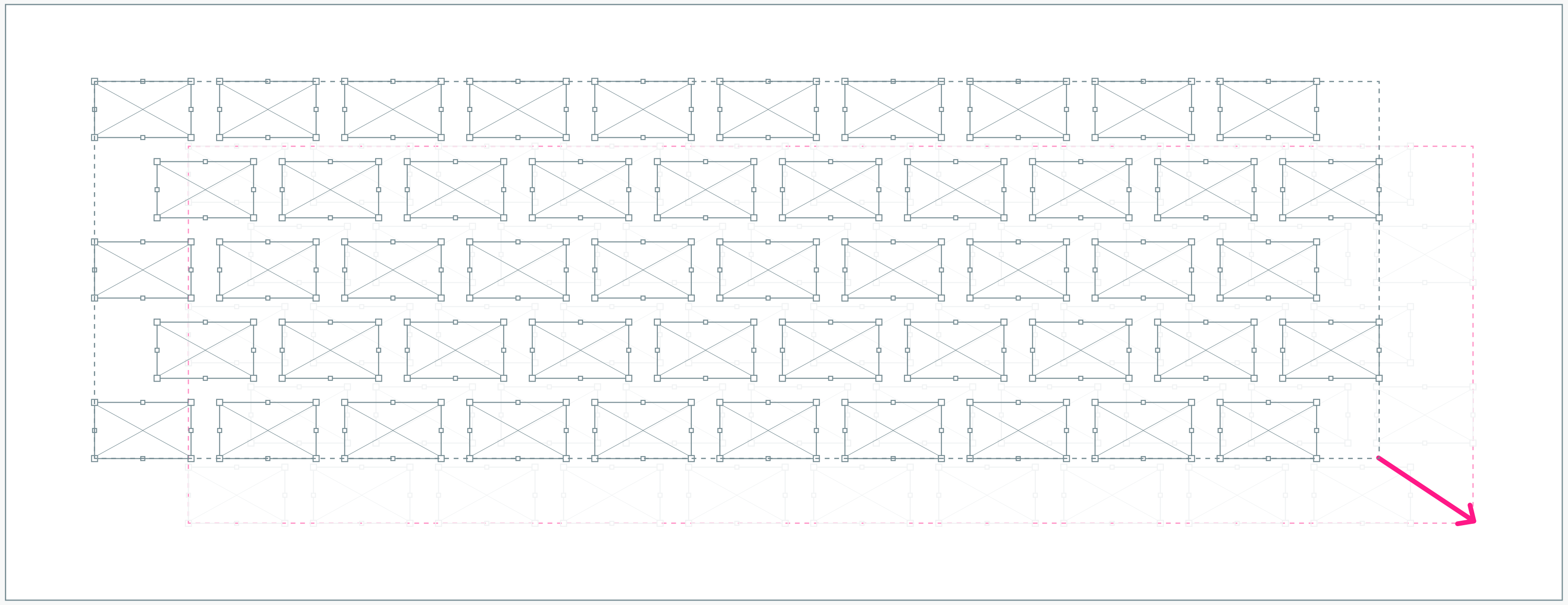


Figure 3. When a group of 50 rectangles is moving, InDesign doesn't need to update the location and the inner path points of every child item. Instead the group tells its parent—typically, a spread container—that its location has changed. Technically, this is done by simply adjusting the affine map attributes that connect the group's inner space to its parent space. This way the child objects are not modified at all (as their respective position relative to the group remains unchanged).

As the whole document relies on a hierarchical structure that involves a lot of dependencies, geometric constraints and nested elements, any change made at any level is likely to affect the location of *every* child object. Consider what is happening when the user moves a group formed by 50 rectangles (cf. **Figure 3**). Does this mean that every individual rectangle is in some way rewritten so that its inner path fits the new location? Of course not!

To accurately manage such operations, InDesign stores locations and geometric data through a hierarchical model that exactly reflects how layout objects are nested or linked. In this model, each component (including groups, pages, spreads, and even on-screen views) has a virtual coordinate system (usually referred to as its INNER COORDINATE SPACE) which is associated to an array of six numbers that specify *how to convert any coordinate pair from that inner system into the parent's coordinate system* (see **Affine Maps** above).

That's it! Now when the user is moving a group of page items, InDesign only has to change the map attributes that connect the group to its parent spread (in terms of coordinate systems). So there is no need to update children' locations.

Transformations Only Re-Map Coordinates

From the user's perspective, all goes as if page items themselves were *transformed*. We can make them smaller or wider, we can rotate them, shear them, etc. Anyway, the most important rule to learn regarding transformations is that *a transformation never alters the actual geometry of graphics objects*.

In other words, whatever the transformations we apply, every path point that underlies the target object will keep its intrinsic location *in the inner coordinate space* of that object.

In InDesign a transformation *only* affects the relationship between two coordinate systems. "Transforming an object" should be understood as changing in some way the affine map that *translates* the inner coordinate space of this object into its parent's coordinate space.

This definition may seem quite abstract, so let me take an example. Say you want to integrate a heart shape vector in your layout. At some point a page item is created storing *only* the geometry of this object within its inner space (cf. **Figure 4**). Note that the object

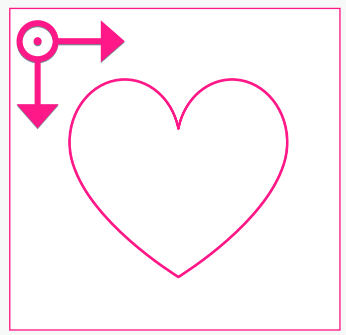


Figure 4. Inner space of a basic page item. The heart shape vector represents the object's geometry, made up by a set of path points. Here the location of each point is expressed relative to the inner coordinate system.

is not *visible* yet, as we didn't specify how and where it is supposed to take place in the publication. To render the page item, InDesign needs to target a DEVICE SPACE, that is, an imageable area where layout contents ultimately appear, such as an on-screen window or a printed page.

Let's not go into details and just assume that a device space is in turn a coordinate system which has the ability to draw graphics. Now, suppose that the heart shape is a direct child of the device along the object hierarchy. The child then can *convert* any coordinate pair from its inner coordinate system to the device coordinate system—since this is the purpose of the affine map associated to the child.

Figure 5 shows how the device and the page item interact via the affine map (M). As you can see, the device can draw the entire shape in its own coordinate space without altering the inner geometry of the page item: any required coordinate pair is just *re-mapped through* M.

Now remember that M is somehow encoded as a sequence of six numeric attributes. It is easy to understand that changing these attributes will cause the device to redraw the heart shape as if taking place in a different coordinate system. **Figure 6** shows the result of such "transformation"—making the shape look smaller in the device space. This is just an example of *scaling* the page item.

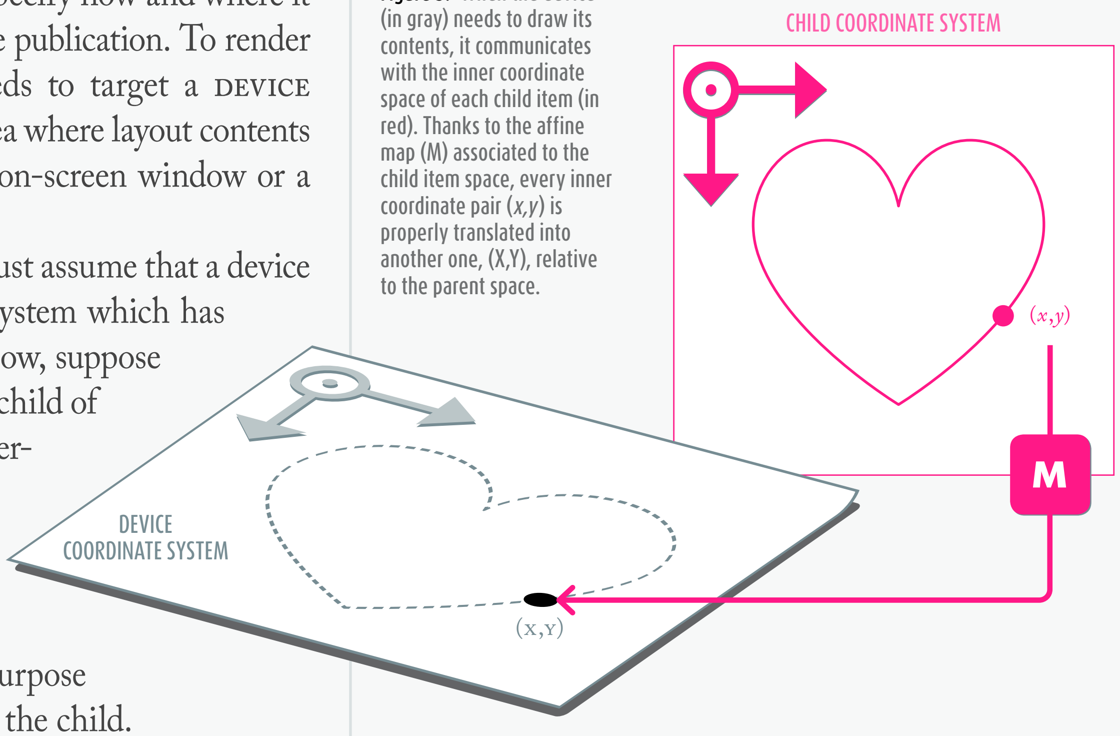


Figure 5. When the device (in gray) needs to draw its contents, it communicates with the inner coordinate space of each child item (in red). Thanks to the affine map (M) associated to the child item space, every inner coordinate pair (x,y) is properly translated into another one, (X,Y), relative to the parent space.

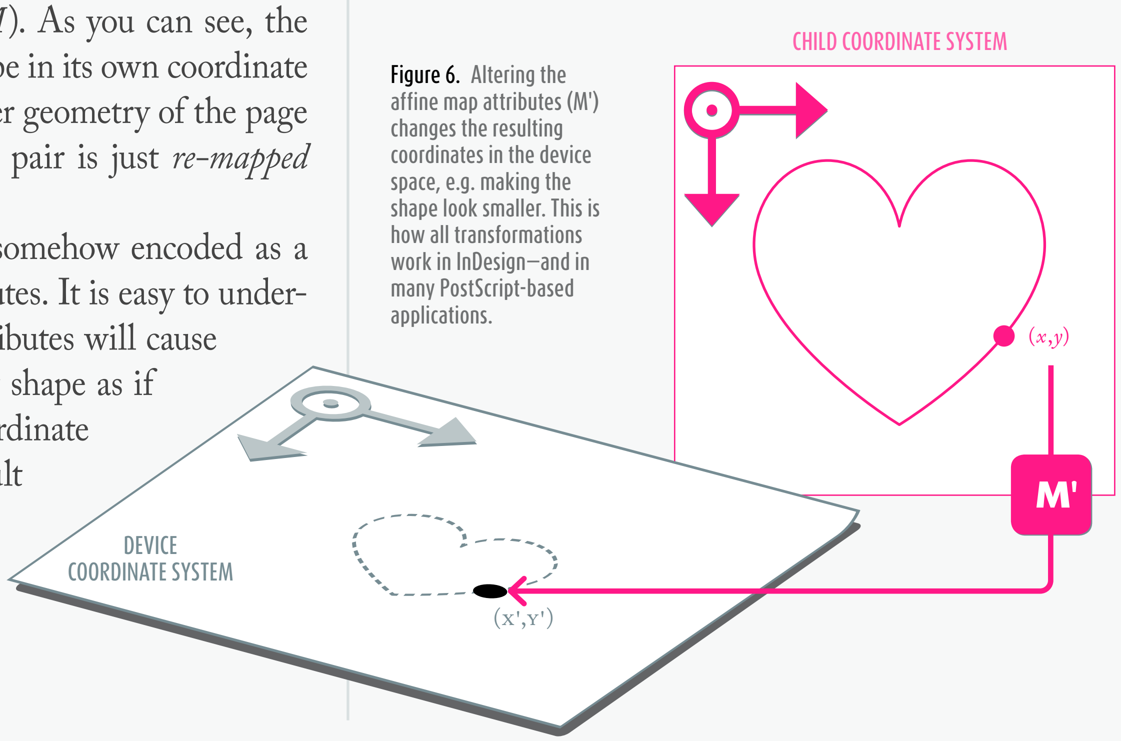


Figure 6. Altering the affine map attributes (M') changes the resulting coordinates in the device space, e.g. making the shape look smaller. This is how all transformations work in InDesign—and in many PostScript-based applications.

Maps, Transformations and Matrices

Before we go any further, there is an important point to highlight: the only *internal* difference between **Figure 5** and **Figure 6** above, is the change from M to M'. Not only the inner geometry of the heart shape but also the respective coordinate systems are preserved[1]. This means that all about transformations regards affine maps, and only affine maps … until the output device space is reached.

As said earlier, an affine map M is based on a sequence of six numbers. Although it is not vital to understand how these attributes operate behind the scenes, an essential key is that, in InDesign, any transformation T is encoded through a sequence of six numbers too. To put it differently: transformations and affine maps are substantially encoded the same way and can play the same role. In mathematical terms, applying T to M amounts to calculate a kind of *product*:

$$M' = M \times T$$

where M' refers to the resulting map (once the transformation is done). Of course the above terms are not real numbers. Each in fact is a 3-by-3 MATRIX that encapsulates the corresponding sequence of attributes.

---

1. Some authors take a different approach and consider that affine transformations actually affect coordinate systems; other argue that graphics objects themselves undergo transformations (rather than coordinate systems). Either point of view may be self-consistent, depending on how the underlying concepts are defined and developed. Anyway, my personal approach is that a transformation does only change an affine map, and that any affine map connects two coordinate systems.

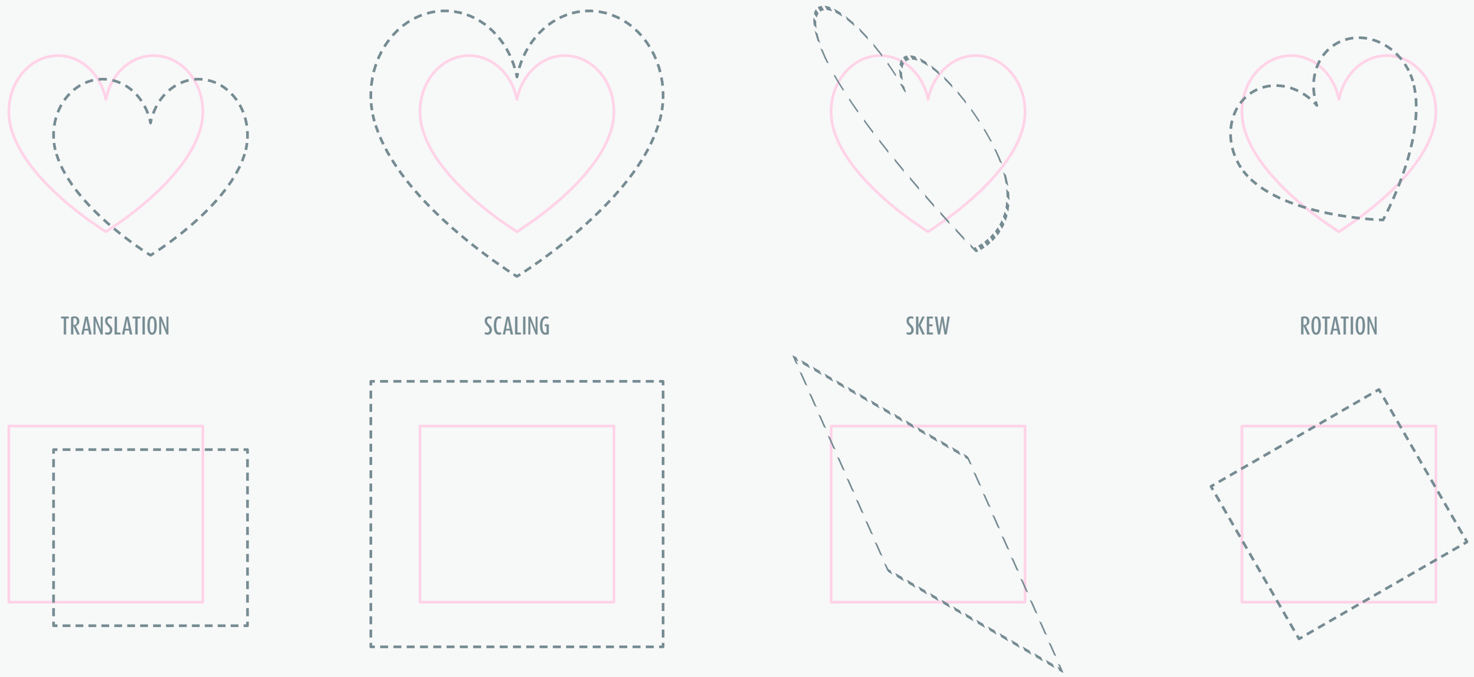


Figure 7. Examples of basic transformations applied to an heart shape (top) and to a rectangle (bottom). The initial geometry is shown in light red; the resulting shapes are shown dotted. (If you mentally replace rectangles with coordinate system bases, you get the same picture in terms of affine mapping.)

The reason why transformations can be encoded as affine maps, and vice-versa, is that InDesign only supports *affine transformations* of the plane, a group of geometric transformations that both preserve collinearity, ratios of distance *and* parallel lines[2]. These are: TRANSLATION, SCALING, ROTATION, REFLECTION, SHEAR, and any combination thereof. **Figure 7** shows basic examples.

In the InDesign SDK, scripting DOM and IDML terminologies, affine maps are often referred to as page item transform(ation) states:

— `ITransform` is the *"transformation matrix that maps from the inner coordinate space to the parent coordinate space."* (SDK)

— `PageItem.transformValuesOf`: *"After an object is transformed, you can get the transformation matrix that was applied to it, using the transformValuesOf() method."* (Scripting DOM)

— `ItemTransform`: *"The relationship of the inner coordinates of the child element to the coordinate system of the <Spread> element (or other parent element) is defined by the ItemTransform attribute of the child element."* (IDML specification)

These definitions all refer to the *current* affine map from a coordinate space to its parent's space, which at a given time is stored as a property of the component. Transformations themselves are *temporary* operands, used in methods that cause an affine map to change. In all cases, however, attributes or arguments are implemented, stored and/or processed as matrix stuctures or similar. For this reason, the term TRANSFORMATION MATRIX in Adobe documentation may refer to either an affine map (the *state*) or a transformation (the *action*).

2. An affine transformation always takes a parallelogram to a parallelogram; and, given two parallelograms P and P', there is always an affine transformation that takes P to P'. This strictly equates to the concept of affine mapping. By contrast, perspective projections are not affine transformations.

Matrix Patterns

By convention every transformation matrix is written:

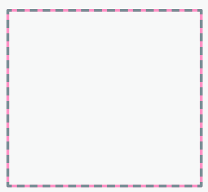
$$\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix}$$

where a, b, c, d, e, f are the numeric attributes of the affine transformation, or map. To *apply* the matrix to a given (x, y) coordinate pair, we calculate the following product:

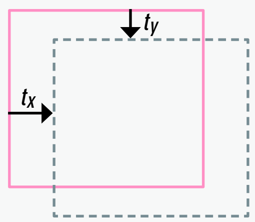
$$\begin{bmatrix} x & y & 1 \end{bmatrix} \times \begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix}$$

which leads to

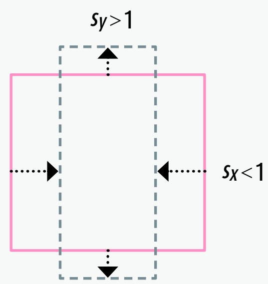
$$\begin{bmatrix} xa+yc+e & xb+yd+f & 1 \end{bmatrix}.$$



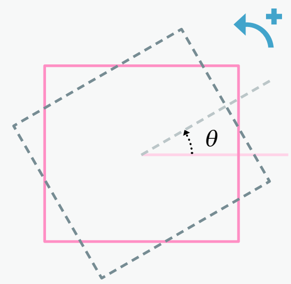
IDENTITY. Strictly identical mapping from the source space to the destination space.



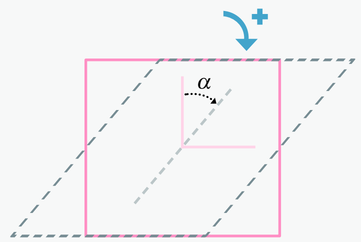
TRANSLATION. Takes any (x, y) pair to $(x+t_x, y+t_y)$. Allows to 'reposition' the object in the destination space.



SCALING. Takes any (x, y) pair to $(x\times s_x, y\times s_y)$. If $s_x=s_y$, this results in a uniform (i.e. homothetic) scaling.



ROTATION. Takes any (x, y) pair to: $(x\times\cos\theta + y\times\sin\theta,\ -x\times\sin\theta + y\times\cos\theta)$ where θ is the counter-clockwise rotation angle.



SHEAR (along the *x*-axis). Takes any (x, y) pair to: $(x - y\times\tan\alpha,\ y)$ where α is the clockwise shear angle relative to the y-axis.

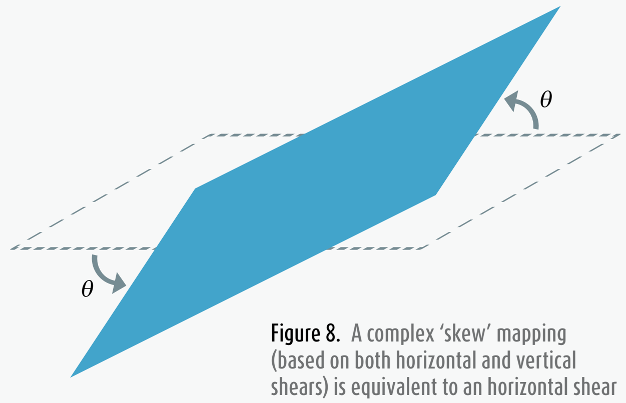


Figure 8. A complex 'skew' mapping (based on both horizontal and vertical shears) is equivalent to an horizontal shear followed by the appropriate rotation.

Ignoring the third dimension, the resulting coordinate pair is finally defined by:

$$(x', y') = (xa + yc + e,\ xb + yd + f).$$

But the above presentation is somewhat artificial. We can see that the 3D matrix only allows to compound a 2D *linear* transformation, based on the attributes a, b, c, d, with a 2D translation, based on the $[e\ f]$ vector. In two-dimensional terms, we would have as well:

$$[x'\ y'] = \underbrace{[x\ y] \times \begin{bmatrix} a & b \\ c & d \end{bmatrix}}_{\text{LINEAR TRANSFORMATION}} + \underbrace{[e\ f]}_{\text{TRANSLATION}}$$

Again, this evidences that any transformation matrix is made up to perform an affine mapping, i. e. a linear transformation *modulo* a translation.

The linear component—a, b, c, d—typically allows to apply scaling, flipping, rotation, or skew, around the origin of the input coordinate system, while the translation component—e, f—allows to reposition the result within the destination space. For this reason, the e and f attributes are often written t_x and t_y instead.

Here are the most common matrix patterns:

IDENTITY: $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

TRANSLATION: $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{bmatrix}$

SCALING: $\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$

ROTATION: $\begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$

SHEAR: $\begin{bmatrix} 1 & 0 & 0 \\ -\tan\alpha & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

Note that the above *rotation* and *shear* formulas fit the default orientation of the y-axis in InDesign (i. e. values increasing from up to bottom) and with respect to the sign of either the 'rotation angle' or the 'shear angle' as shown in the GUI as well as in transformation settings. These may differ from academic patterns.

In particular, for skew mapping parallel to the y-axis, InDesign will use something like:

$$\begin{bmatrix} 0 & -1 & 0 \\ 1 & \tan\beta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

where β denotes the shear angle relative to the x-axis. It is not difficult to see that this matrix results from applying a 90° rotation after the SHEAR pattern.[3] Indeed, InDesign treats any skew mapping as a combination of a *shear-along-x* and a *rotation* (see **Figure 8**).

3. Viz: $\begin{bmatrix} 1 & 0 \\ -\tan\beta & 1 \end{bmatrix} \times \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$ (having $\cos 90° = 0$ and $\sin 90° = 1$).

Matrix Product

Let M and M' be two matrices. No matter their respective attributes, the product $M \times M'$ always results in a new matrix. Technically:

$$\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix} \times \begin{bmatrix} a' & b' & 0 \\ c' & d' & 0 \\ e' & f' & 1 \end{bmatrix} = \begin{bmatrix} aa'+bc' & ab'+bd' & 0 \\ ca'+dc' & cb'+dd' & 0 \\ ea'+fc'+e' & eb'+fd'+f' & 1 \end{bmatrix}$$

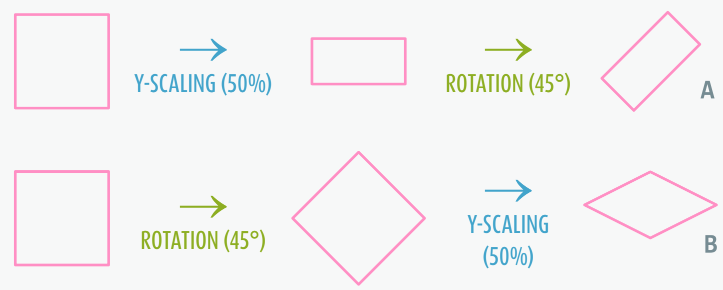
where a, b, c, d, e, f (resp. a', b', c', d', e', f') are the numeric attributes of M (resp. M'). These complex calculations are of little interest though. What is important is to get the *meaning* of the product: $M \times M'$ reflects the combination of the two transformations, that is, the M-transformation *followed* by the M'-transformation.

For example, suppose that M represents some SCALING and M' represents some ROTATION. Then, $M \times M'$ is the matrix that encodes the global transformation (SCALING, *then* ROTATION).

At any level, whenever InDesign *applies* a transformation, it simply computes the product of an existing matrix (map or transformation) by an incoming matrix. This way, successive transformations applied to an object—in fact, to its affine map—have not to be stored by themselves. The map is simply updated as the result of a matrix product, and its new attributes represent the whole effect of all transformations it has undergone from its creation.

No matter how, and how much, you multiply transformations on an object, the result is always as simple as a unique transformation, entirely described by six numbers, which finally encodes the resulting affine map.

However, during the computation, the order of transformations does matter, as $M \times M'$ is ordinary not equivalent to $M' \times M$. It is easy to check visually that scaling first, then rotating, is not the same as performing rotation before scaling:



In other words:

SCALING × ROTATION ≠ ROTATION × SCALING

This inequality can be generalized to most matrix products.

InDesign's Canonical Transformation Order (S×H×R×T)

But, as an experienced user, you may have noticed that the shape **A** (above) is easier to obtain than the shape **B**. Indeed, atomic transformations (TRANSLATION, SCALING, ROTATION, and SHEAR) cannot be applied in an arbitrary order to an object *via* the GUI—although this could be done through scripting, as we shall see later.

We must highlight here that every transformation matrix M can be decomposed as a product of four atomic matrices, in the following order:

$$M = S \times H \times R \times T,$$

where S is a SCALING matrix, H a SHEAR matrix, R a ROTATION matrix and T a TRANSLATION matrix (*see the previous page for the related patterns*). This canonical decomposition is unique, and InDesign uses it as an internal mechanism to link any transformation matrix to a set of user-friendly attributes in the interface. Developers will also access those attributes from the `TranformationMatrix` object; namely: horizontal and vertical scale factor (sx, sy), clockwise shear angle (α), counterclockwise rotation angle (θ), horizontal and vertical translation (tx, ty).

While this fact is generally overshadowed in the literature, it is of the utmost importance to understand the underlying principle before you deal with transformations. Given the matrix values (a, b, c, d, e, f), InDesign automatically resolves and maintains the correlated S×H×R×T scheme so that one *always* has the relation:

$$\underset{M}{\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix}} = \underset{\text{SCALING}}{\begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix}} \times \underset{\text{SHEAR}}{\begin{bmatrix} 1 & 0 & 0 \\ -\tan\alpha & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}} \times \underset{\text{ROTATION}}{\begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}} \times \underset{\text{TRANSL.}}{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ tx & ty & 1 \end{bmatrix}}$$

This decomposition has many advantages. First, since the translation is the final term of the product, the (tx, ty) components are not involved with previous calculations and remains independent, so $(e, f) = (tx, ty)$. The remaining equation has a pure 2D-linear form:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} sx & 0 \\ 0 & sy \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ -\tan\alpha & 1 \end{bmatrix} \times \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

Here we can see that the *determinant* of both the shear and the rotation matrices is 1 (which reflects the fact that these transformations preserves the area). So the determinant of the entire matrix is simply $sx \times sy$ $(= a \times d - b \times c)$. This signed value represents the area scale factor, noting that a negative number indicates whether the shape is mirrored.

Also, we can distinguish the neutral parameters, viz. the IDENTITY matrix for each term: $(sx, sy) = (1,1)$ [no scaling]; $\alpha = \emptyset$ [no shear]; $\theta = \emptyset$ [no rotation].

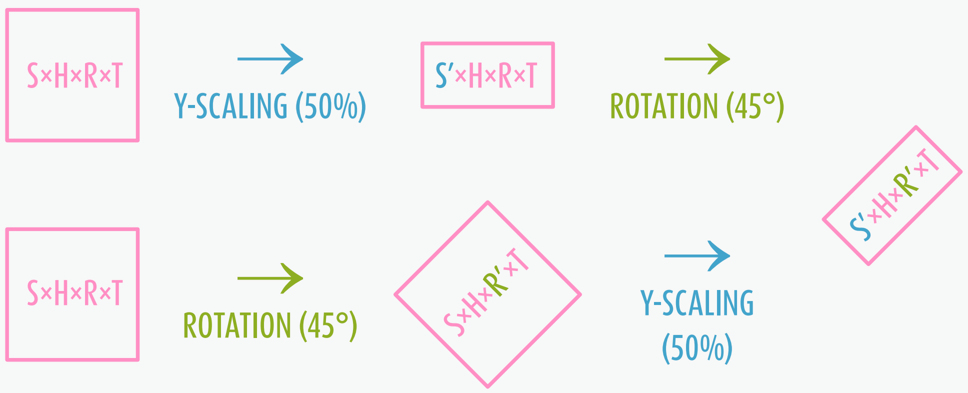


Figure 9.
Object transformations specified from the GUI are order-insensitive, because at each step InDesign only has to update a single matrix component without altering the canonical decomposition order (S×H×R×T).

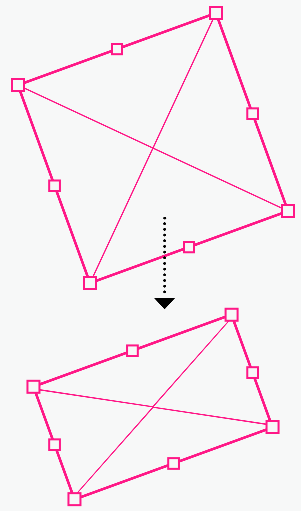


Figure 10a.
Changing the Y-scale of the selected object simply results in changing the SCALING component of the affine map. This is a TRANSFORMATION.

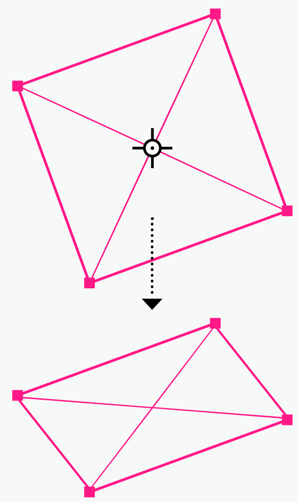


Figure 10b.
Manually selecting all path points with the Direct Selection tool then changing the Y-scale results in actually moving the points "along the transformation." This is a DEFORMATION.

A matrix is *invertible* iff its determinant is not zero, i.e. $s_x \times s_y \neq 0$—here you see why InDesign does not allow you to scale anything at 0%! In InDesign, any (valid) transformation matrix is invertible.

Now, let's consider a document page item. Its affine map (M), in its current state, can be expressed as $M = S \times H \times R \times T$ (keeping the above notation). Suppose that the user changes the shear angle from the Transform panel. What does this mean in terms of re-mapping?

You could think that some matrix is created, say U, *encoding the shear transformation specified by the user*, and that M is changed to $M \times U$.

That would make perfect sense, but that's not what happens. InDesign does *not* change M to $M \times U$, because it does *not* apply any shear matrix to the existing map! Instead, it only updates the existing shear component (H) so that it now reflects the desired shear angle. In other words, H just becomes H', and M therefore becomes $S \times H' \times R \times T$.[4]

4. Of course the six matrix values (a, b, c, d, tx, ty) are recalculated accordingly.

The same thing happens when you change the scaling of a page item on which a rotation is already applied. The application doesn't compute $M \times U_{scaling}$, it just changes the map to $S' \times H \times R \times T$, where S' denotes the new scaling component. This is the reason why either the order *scaling-then-rotation*, or *rotation-then-scaling*, leads to the same result in the GUI (see **Figure 9**).

Transformation vs. Deformation

Since a transformation in itself only affects an affine map and does not impact the actual geometry of the shape, it is relevant for developers to draw a formal distinction between TRANSFORMING and DEFORMING an object. The former only deals with re-mapping coordinates *via* transformation matrices, the latter regards actual moves of path points relative to the inner space.

InDesign provides various ways to *deform* a path, that is, the geometry of a spline item. In this regard, an interesting operation is to select a set of path points using the Direct Selection tool (A) and to "apply a transformation" of whatever kind. What does happen under the hood? InDesign *does not transform* the object in the sense of updating its affine map. Instead, it moves the points along the transformation, as shown in **Figure 10b**. In this very specific case, although the system internally performs a transformation on the set of selected path points, no trace of this operation is stored in the transformation matrix.

Hierarchical Mapping

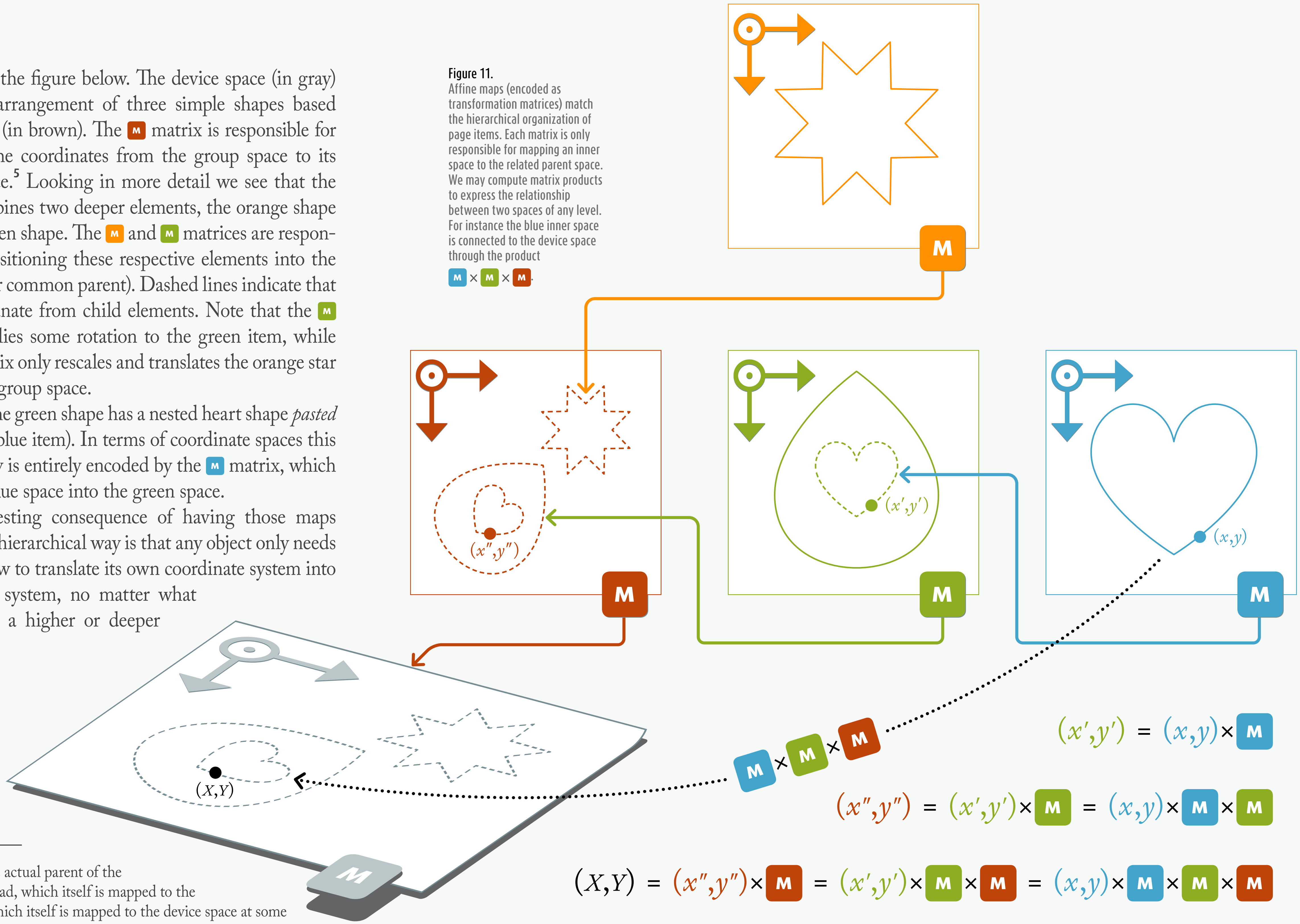
As said earlier, every new object along the document hierarchy—including groups, pages and spreads—has its own coordinate space bound to the coordinate space of the parent object *via* an affine map. Technically, each of those affine maps is encoded in a single transformation matrix. This is the way all graphic objects are positioned relative to each other.

Consider the figure below. The device space (in gray) shows an arrangement of three simple shapes based on a group (in brown). The M matrix is responsible for mapping the coordinates from the group space to its parent space.[5] Looking in more detail we see that the group combines two deeper elements, the orange shape and the green shape. The M and M matrices are responsible for positioning these respective elements into the group (their common parent). Dashed lines indicate that shapes emanate from child elements. Note that the M matrix applies some rotation to the green item, while the M matrix only rescales and translates the orange star within the group space.

Finally, the green shape has a nested heart shape *pasted into* it (the blue item). In terms of coordinate spaces this dependency is entirely encoded by the M matrix, which maps the blue space into the green space.

An interesting consequence of having those maps linked in a hierarchical way is that any object only needs to *know* how to translate its own coordinate system into the parent system, no matter what happens at a higher or deeper level.

Figure 11.
Affine maps (encoded as transformation matrices) match the hierarchical organization of page items. Each matrix is only responsible for mapping an inner space to the related parent space. We may compute matrix products to express the relationship between two spaces of any level. For instance the blue inner space is connected to the device space through the product M × M × M.



5. In fact, the actual parent of the group is a spread, which itself is mapped to the pasteboard, which itself is mapped to the device space at some point. We will discuss later these specific coordinate systems.

SUMMARY

A 2D coordinate is a pair of numbers (x, y) that locate a point relative to a given system of axes, origin, and units—referred to as a COORDINATE SYSTEM.

In InDesign every layout component (including pages and spreads) is bound to its own coordinate system, also known as its INNER COORDINATE SPACE.

The functional relationship between two coordinate systems is called an AFFINE MAP. Any affine map is determined by a set of six real numbers, conventionally arranged in a matrix—a TRANSFORMATION MATRIX.

The way we operate on such matrices might be purely described in terms of geometrical transformations. They address *"any linear mapping of two-dimensional coordinates, including translation, scaling, rotation, reflection, and skewing."* (InDesign SDK)[1]

InDesign internally reduces any transformation matrix to a combination of four atomic transformations: SCALING, SHEAR, ROTATION, and TRANSLATION, in that order. This "canonical decomposition" S×H×R×T is unique and allows to treat separately the underlying parameters (scaling factors, shear angle, etc.).

1. Regarding transformation matrices, InDesign complies with the rules of the PDF Specification: *"A transformation matrix specifies the relationship between two coordinate spaces. By modifying a transformation matrix, objects can be scaled, rotated, translated, or transformed in other ways."* (PDF 32000-1:2008, p. 117.)

When one "applies" a TRANSFORMATION onto an object—say a rotation—InDesign does not really modify the geometry of the underlying shape. Instead, the application updates the affine map that links the coordinate space of that object to the coordinate space of its parent. Therefore, what is said a transformed object is nothing but the *same* object seen from a different perspective and/or location.

However, under specific circumstances InDesign may allow to use transformation tools in a way that actually impacts the inner geometry of the target object, rather than its affine map. This case will be referred to as a DEFORMATION.[2]

Affine maps are chained according to the layout hierarchy. Thanks to this mechanism transformations that occur at any level may be described in the perspective of any other coordinate space.

EXERCISES

001. Let `Obj` be a **`PageItem`** and `(1,2,-1,0,3,1)` the matrix values of its affine map. Express in `Obj`'s parent space the coordinate pair of its inner space origin.

002. Explain why a shear angle cannot amount to 90°.

003. Let `G` be a **`Group`** having two rectangles `R1` and `R2` as direct children. (None of those page items has been scaled, skewed, or rotated yet.) Assume that the user then selects `G` and applies a `45°` rotation to it. How do the transformation matrices of `G`, `R1`, and `R2` now look like?

004. Suppose that the area of some shape, measured in its inner space, is `8pt²`. Let `(2,3,3,6,-7,5)` be the matrix values of its affine map. What is the shape area measured in the parent space?

005. Let `S = (sx,0,0,sy,0,0)` be a valid SCALING matrix. What is the *inverse* of `S`? [The inverse of a matrix `M` is a matrix `M'` such that `M × M' = M'× M =` IDENTITY.]

006. Can an InDesign user change the location of an object relative to its parent **`Spread`** without changing at all its affine map?

2. During a deformation, the transformation parameters have only a temporary existence.

Object locations and transformations cannot be understood without a clear comprehension of InDesign-specific coordinate spaces. This section presents those fundamental frames to programmers before they fiddle with geometry.

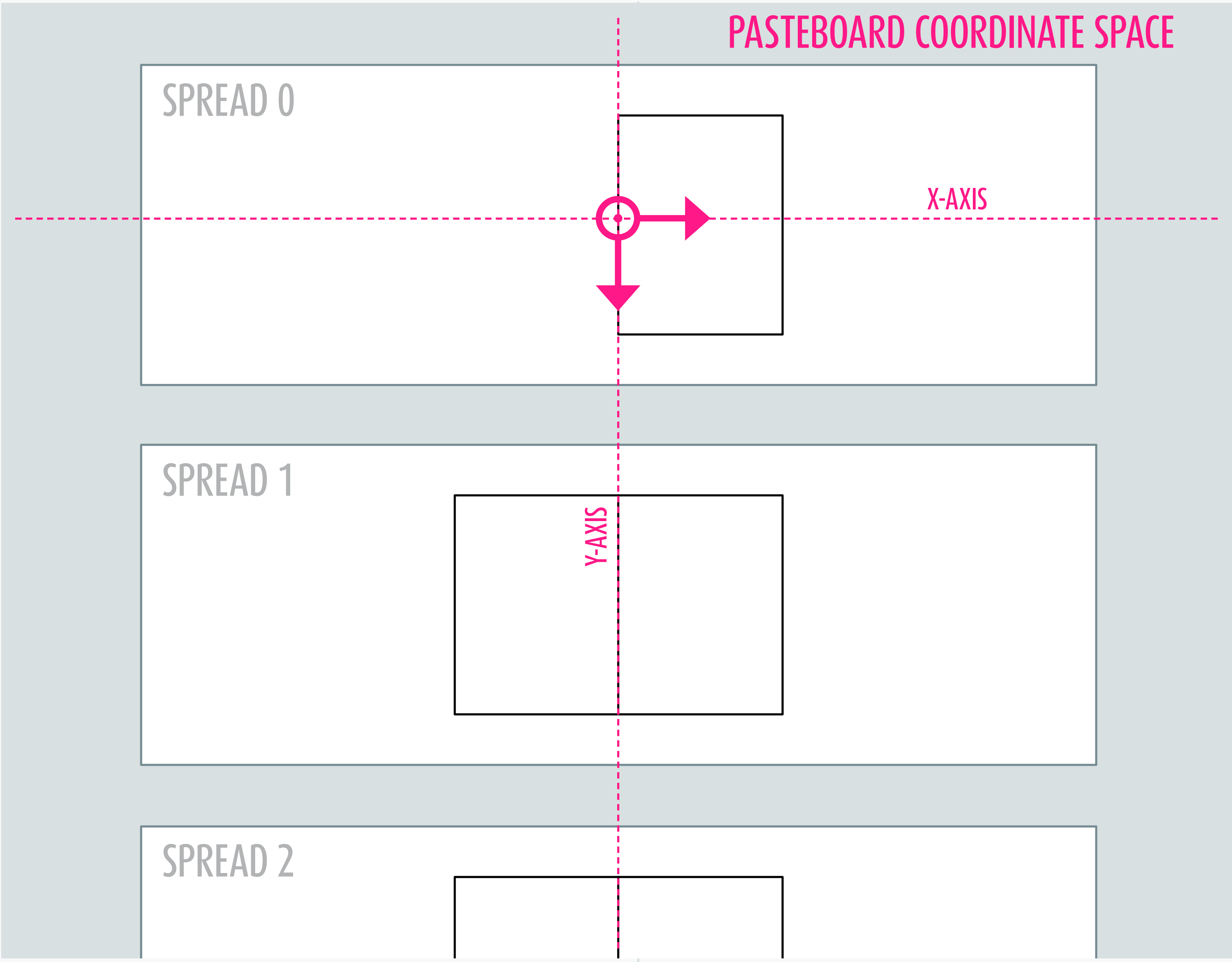


Figure 12. The Pasteboard coordinate space.

Pasteboard Coordinate Space

At the very top level is the PASTEBOARD COORDINATE SPACE, a kind of Galilean reference frame. It is the global, absolute coordinate system that surrounds the whole document.

The pasteboard[1] space encompasses the entire workspace, including interstitial areas where no object can be laid out at all. This root entity might be seen as the virtual parent of every document spread. We will consider it a *device space* for the on-screen layout. InDesign uses in fact higher coordinate systems that reflect how the layout is shown in different *views* (based on windows, scrolling, magnification…) but these do not regard the imageable document.

Any location can be easily and univocally expressed in the pasteboard coordinate space, whose origin is the center of the first spread of the document. The *x*-axis is horizontal with values increasing from left to right; the *y*-axis is vertical with values increasing from up to bottom (see **Figure 12**).

1. Note that the pasteboard is not represented as an object in the InDesign Scripting DOM, which may be confusing. The word 'pasteboard' commonly denotes the outer region of a page (white background) which still contains or may contain layout items. In fact, this extra region would be rather described in terms of spread margins, because anything that lives there is in the scope of the corresponding spread and actually belongs to it. Due to this confusion some settings that regard spreads are referred to as *pasteboard* things in the DOM. For example, the `PasteboardPreference` object (available under `Document` and `Application`) exposes a `pasteboardMargins` property (array of two measurement units) which controls the width and the height of the spread margins. In InDesign CS4, only the height of the spread margin was addressable, *via* the `minimumSpaceAboveAndBelow` property.

The length along each axis is measured in points and there is no way to change this. InDesign documents and all basic coordinate spaces handle measurements in PostScript points, although both Scripting DOM and ExtendScript's core provide tools to convert measurements into other units, as this is done in the application GUI.

Since the pasteboard is not an actual DOM object, it has no parent and therefore no transformation matrix bound to it—at least, nothing that we could reach through scripting.[2]

Given a document, you can refer to the pasteboard coordinate space using the enumerated value `CoordinateSpaces.pasteboardCoordinates` in any method that handles coordinates. We will see later various uses for this key.

Spread Coordinate Space

"Each spread has its own coordinate space, also known as the inner coordinate space for a spread. The origin of the spread coordinate space is the center of the spread. The parent coordinate space is pasteboard coordinate space." (InDesign SDK, see **Figure 13**.)

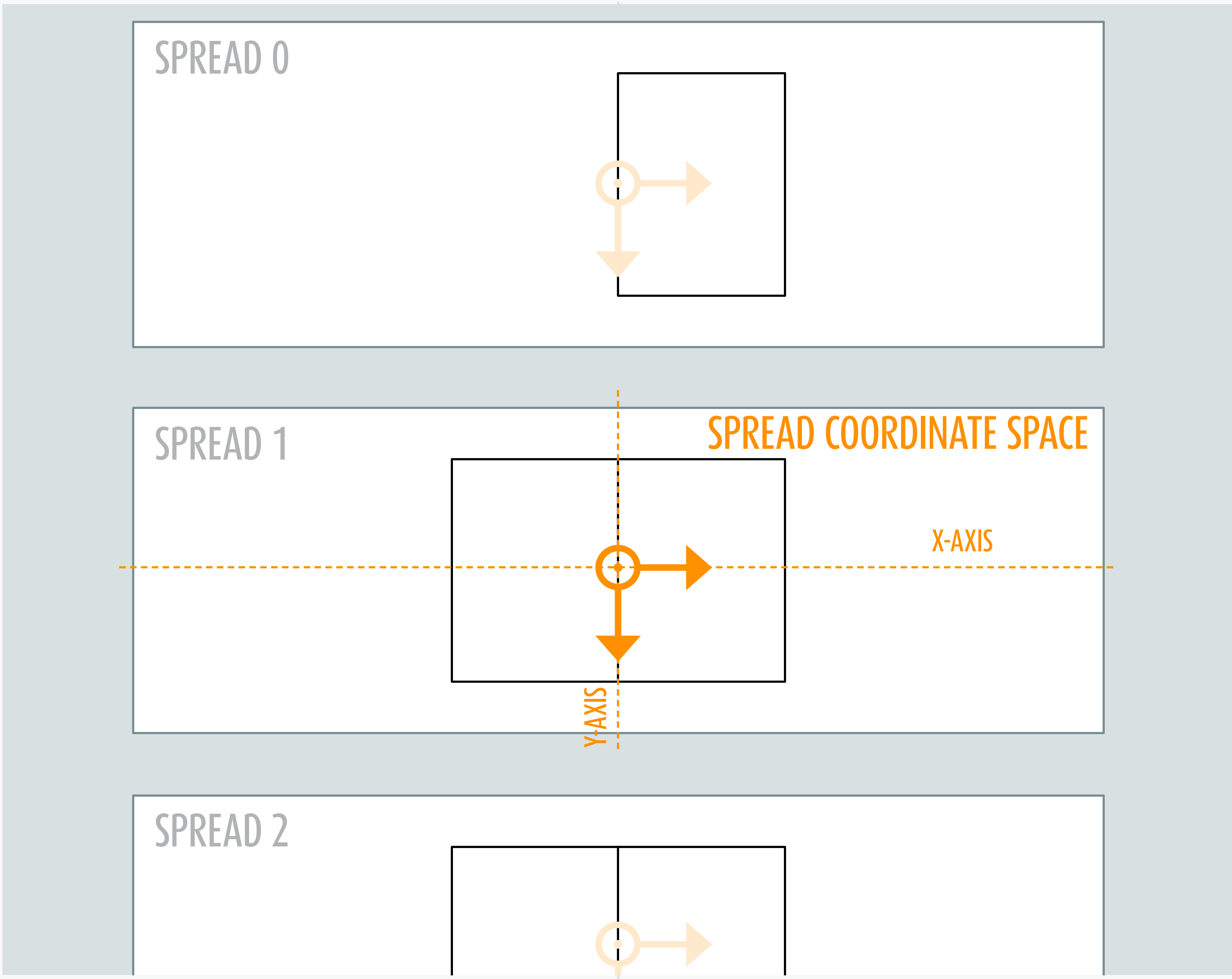
A `Spread` object being known, you can directly refer to its specific coordinate space using `CoordinateSpaces.spreadCoordinates` in every method that handles coordinates. Be aware that the origin of a spread coordinate space does not coincide with the default zero point in Ruler Per Spread mode. Also, unlike the rulers coordinate system (which we will study later), spread coordinate spaces only support measurements in points.

As a general rule, the transformation matrix of a spread—read: the affine map that connects this spread space to the pasteboard space—will specify a TRANSLATION in the form:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & t_y & 1 \end{bmatrix}$$

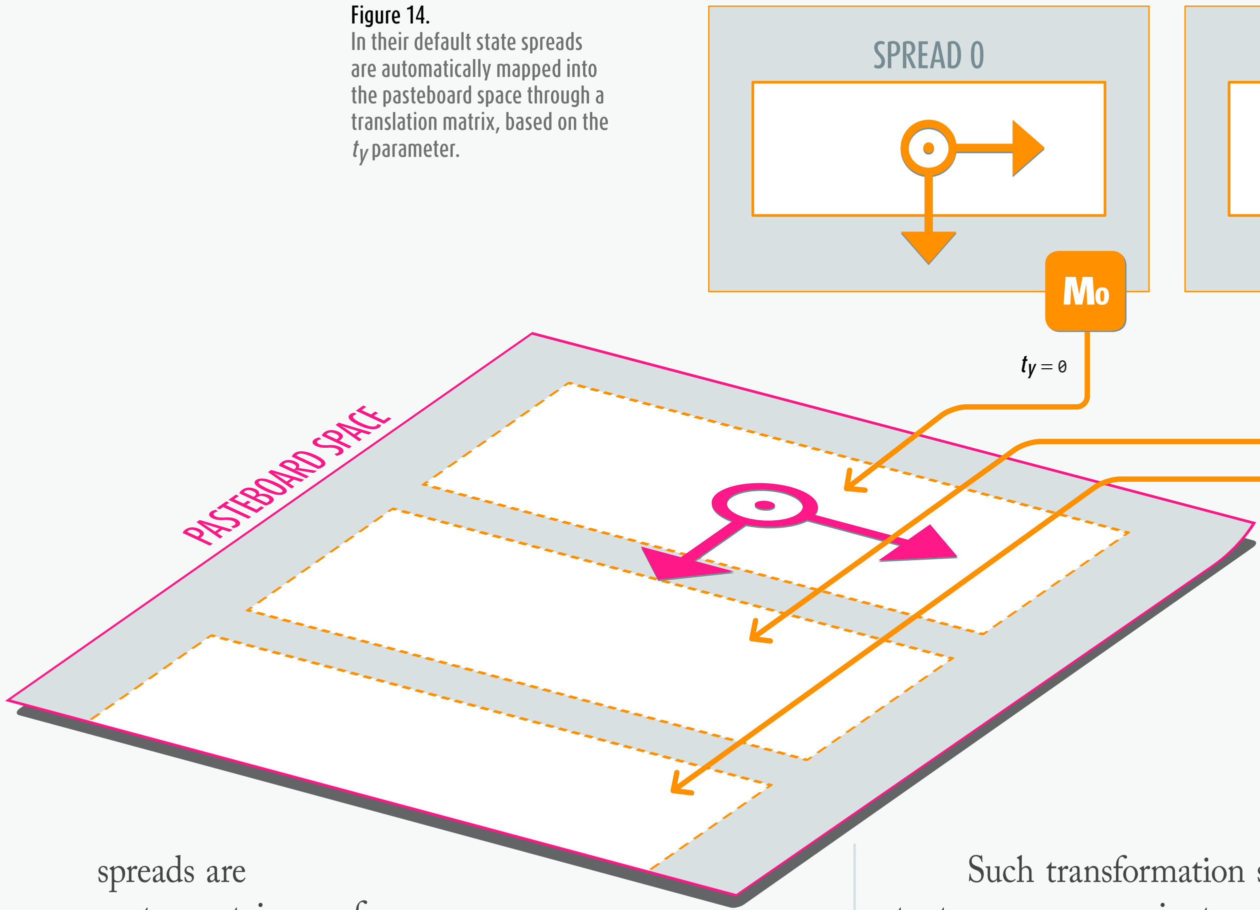
where t_y represents the offset along the y-axis relative to the pasteboard space origin. Indeed, although

Figure 13. A typical spread coordinate space.



2. It is worth noting, however, that the pasteboard space origin directly depends on the size and state of the *first* document spread, meaning it may *move* in some circumstances, such as changing the page size or transforming a spread. Also, facing-pages vs. non-facing-pages documents follow distinct rules on positioning spreads.

Figure 14.
In their default state spreads are automatically mapped into the pasteboard space through a translation matrix, based on the t_y parameter.



spreads are root containers for pages and page items, they can usually be treated from the pasteboard perspective as simple rectangular regions ordered along the vertical axis (see **Figure 14**).

But in InDesign CS4 and later spreads support transformations, with a few restrictions though. That is, the affine map of a spread may contain non-default SCALING, SHEARING and/or ROTATION parameters.[3] As a basic example let's apply a 90° Rotated Spread View on a spread. The pasteboard perspective will then look like in **Figure 15** below.

3. The only component that you cannot control is the TRANSLATION part. The (t_x, t_y) parameters of a spread matrix entirely depend on automatic positioning, which in turn depends on respective spread areas and orientation, facing-pages option, etc.

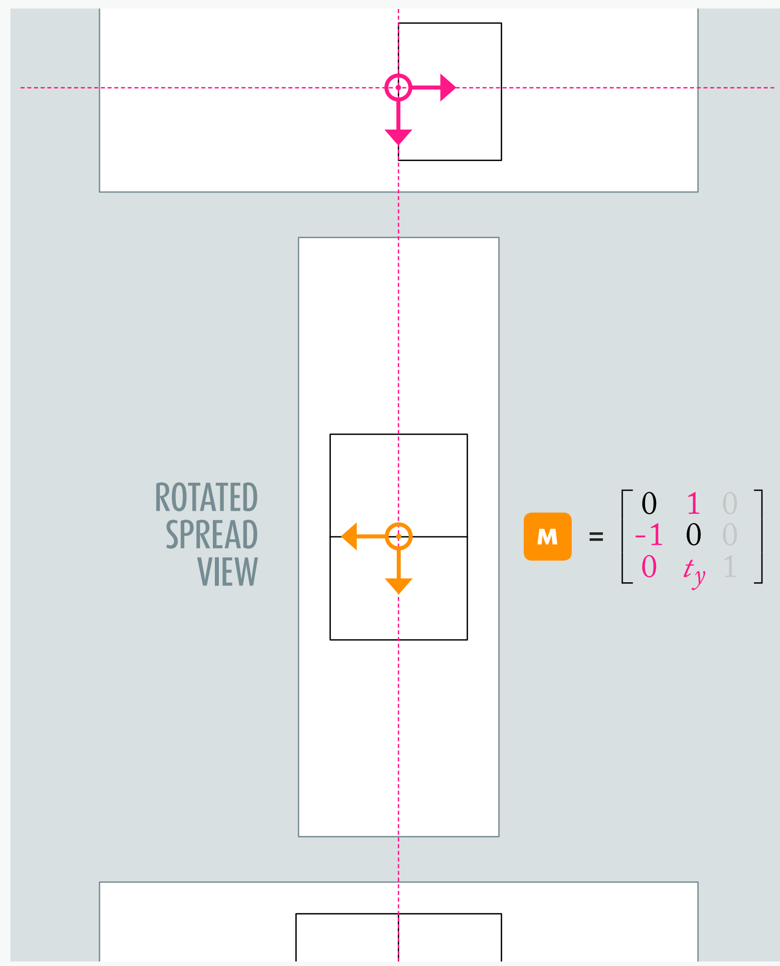
Such transformation state has of course important consequences in terms of object locations and metrics.

How to get details on rotated views from your scripts? You need first to retrieve the affine map of the spread, as follows:

```
// 01. GET THE AFFINE MAP OF A SPREAD
const CS = CoordinateSpaces,
      CS_PARENT = CS.parentCoordinates;
var mx = mySpread.transformValuesOf(CS_PARENT)[0];
alert(mx.matrixValues);
```

The method `transformValuesOf(anySpace)` returns a singleton array whose unique element is a matrix that maps the caller inner space to *anySpace*. Therefore, the following code:

Figure 15.
When a "Rotated Spread View" is applied (from the Pages panel), the transformation matrix of the underlying spread space is accordingly changed to reflect the rotation.



```
anyObj.transformValuesOf(CoordinateSpaces.
parentCoordinates)[0]
```

always returns the affine map attached to *anyObj*, as the enum value **CoordinateSpaces.parentCoordinates** points out to the parent coordinate space associated to *anyObj* along the hierarchy.[4]

We store the result, a **TransformationMatrix**, in the variable **mx** so that we can study the underlying components. The property **mx.matrixValues** reveals the matrix parameters in the form [a, b, c, d, tx, ty] (array of six numbers, keeping the notations used in the previous chapter.)

From then it's easy to get the *meaning* of these data:

Matrix Values	Spread Rotation State
[1, 0, 0, 1, 0, ty]	Default. (No rotation applied.)
[0, 1,-1, 0, 0, ty]	90° clockwise (CW).
[0,-1, 1, 0, 0, ty]	90° counterclockwise (CCW).
[-1, 0, 0,-1, 0, ty]	180°.

Alternately the **TransformationMatrix** object exposes a property, **counterclockwiseRotationAngle**, which indicates the ccw rotation angle in degrees.

```
// 02. DISPLAY THE ROTATION ANGLE OF A SPREAD
const CS = CoordinateSpaces,
      CS_PARENT = CS.parentCoordinates;
var mx = mySpread.transformValuesOf(CS_PARENT)[0];
alert(mx.counterclockwiseRotationAngle);
```

4. As we are considering a **Spread** object, **CoordinateSpaces.parentCoordinates** is, in fact, equivalent to **CoordinateSpaces.pasteboardCoordinates**. The former syntax is just more generic.

Page Coordinate Space

"Each page has its own coordinate space, also known as the inner coordinate space for a page. The parent coordinate space for page coordinate space is spread coordinate space. The origin of page coordinate space is the top-left corner of the page." (InDesign SDK, see **Figure 16**.) A **Page** being known, you can refer to its specific space using **CoordinateSpaces.pageCoordinates** in CS6 and later.

Here again, note that the origin of a page coordinate space is in no way determined by the zero point in Ruler Per Page mode—users can move the zero point anywhere in the page area. Also, unlike rulers' coordinate system, a page coordinate space only handles measurements in points.

As a general rule, the transformation matrix of a page—read: the affine map that connects this page space to the parent spread space—will specify a TRANSLATION in the form:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & T_y & 1 \end{bmatrix}$$

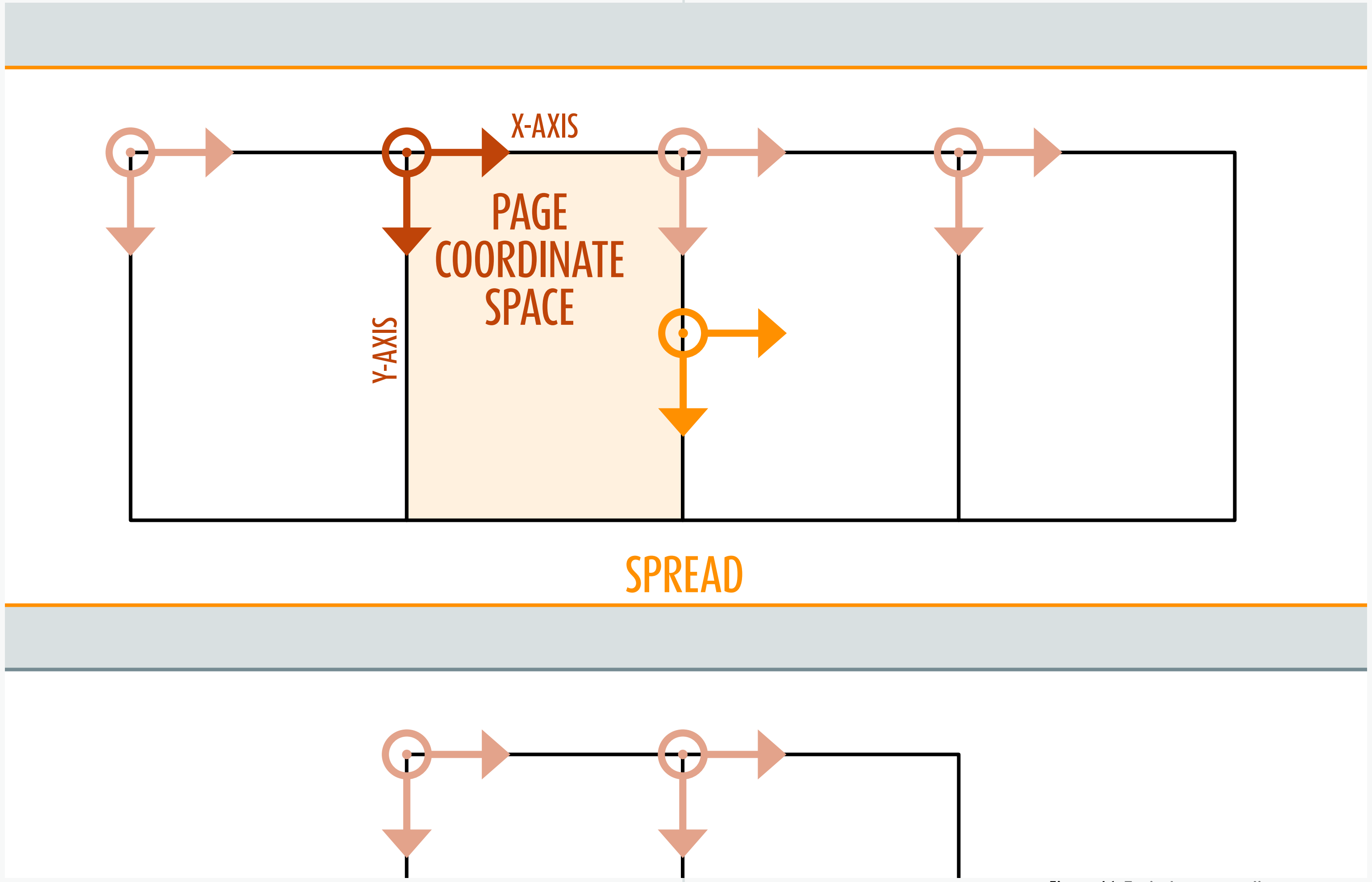


Figure 16. Typical page coordinate space.

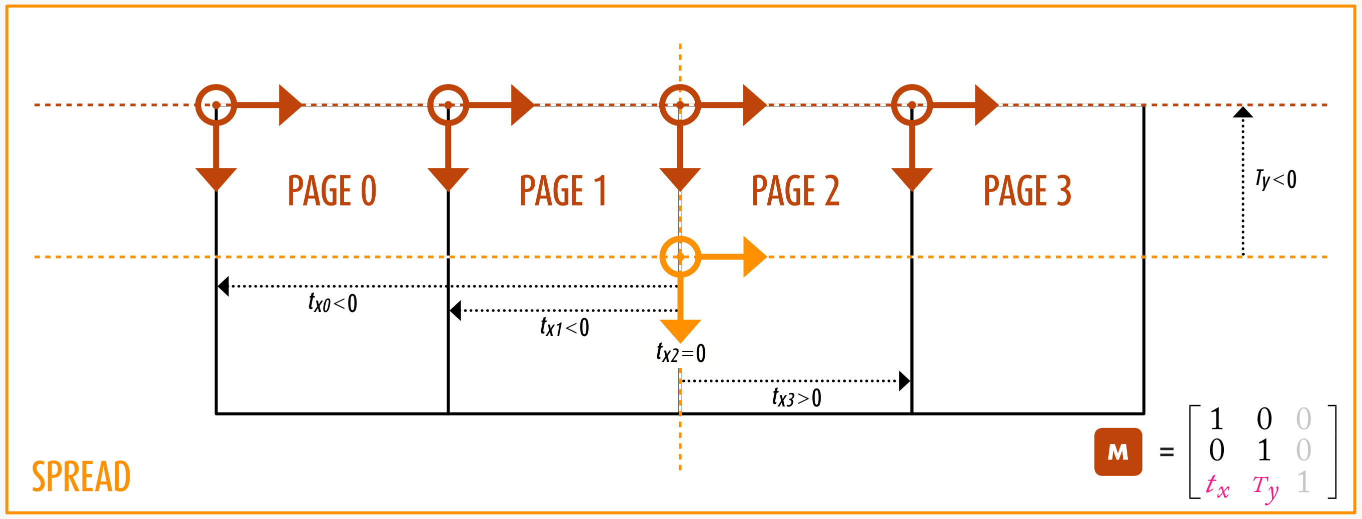


Figure 17. A typical four-page spread in its default state. As long as pages remain untransformed, their affine map boils down to a simple (tx,Ty)-translation. Note that rotating the spread view (that is, changing the affine map of the spread itself) wouldn't have any effect on those page-to-spread matrices (M).

where t_x represents an offset along the x-axis relative to the spread coordinate space, while T_y stands for some constant y-offset, as illustrated in Figure 17.

Let's pause for a moment and try to clarify why T_y is a negative offset. A primary reflex is to think that the TRANSLATION encoded in **M** should *move* the page origin *to* the spread origin, which would lead to $T_y > 0$. That's a misrepresentation of what the TRANSLATION is. As said earlier the purpose of the map **M** is to convert page-relative coordinates into spread-relative coordinates. In particular, applying the map to `(0,0)`—i.e., the origin of the page in its own coordinate space—must result in a coordinate pair (x_o, y_o) which *positions* that origin in the spread coordinate space. Considering PAGE 2 in the figure we clearly expect $x_o = 0$ and $y_o < 0$. Let's apply the affine map:

$$[\,x_o, y_o, 1\,] = [\,0, 0, 1\,] \times \mathbf{M} = [\,t_x, T_y, 1\,].$$

It comes $t_x = x_o (= 0)$, and $T_y = y_o (< 0)$, so we can express the rule as follows: *The affine map of a page is usually a simple TRANSLATION whose (t_x, t_y) parameters reflect the location of the page space origin relative to the parent spread origin (i.e. the center point of the spread).*

Let's reveal these translation parameters using `Page.transformValuesOf()`:

```
// 03.  DISPLAY THE TRANSLATION VALUES OF
//      ALL PAGES HOSTED BY spreads[spdIndex]
const CS = CoordinateSpaces,
      CS_PARENT = CS.parentCoordinates;
var spd = app.activeDocument.spreads[spdIndex],
    pgs = spd.pages.everyItem(),
    a = [].concat
        (pgs.transformValuesOf(CS_PARENT))[0],
    i = a.length;
while( i-- ) (a[i]=a[i].matrixValues).splice(0,4);
alert( a.join('\r') );


//  Typical result for a four-page spread
//  in facing-pages mode
//  ---
//      -1200,-425
//      -600,-425
//      0,-425
//      600,-425
```

Page Size and Location Issues

Prior to InDesign CS5 pages couldn't be *transformed* at all (that is, page affine maps couldn't be programmatically changed). Now **Page** objects support the **transform()** method, meaning that we can alter the underlying matrix so that a specific page appears transformed within its parent spread.

Page transformation is Pandora's box. It leads to both great possibilities and unexpected troubles regarding page coordinate spaces. First above all, you cannot assume that page sizes are uniform anymore. Document settings only specifies a *default page size*. Using the Page Tool, the user can change the dimensions of a specific page and/or its default location. Such effects depend on document facing-pages options, shuffling behavior between spreads, layout rules involving master page inheritance mechanism, and so on.[5]

---

5. In InDesign CS5 and later, you cannot even be sure that the origin of a page coordinate space will match the top-left corner of the corresponding page! It's easy to break rules playing with the Page Tool or applying custom transformations to the master-to-page matrix (**Page.masterPageTransform**). See next page for an example.

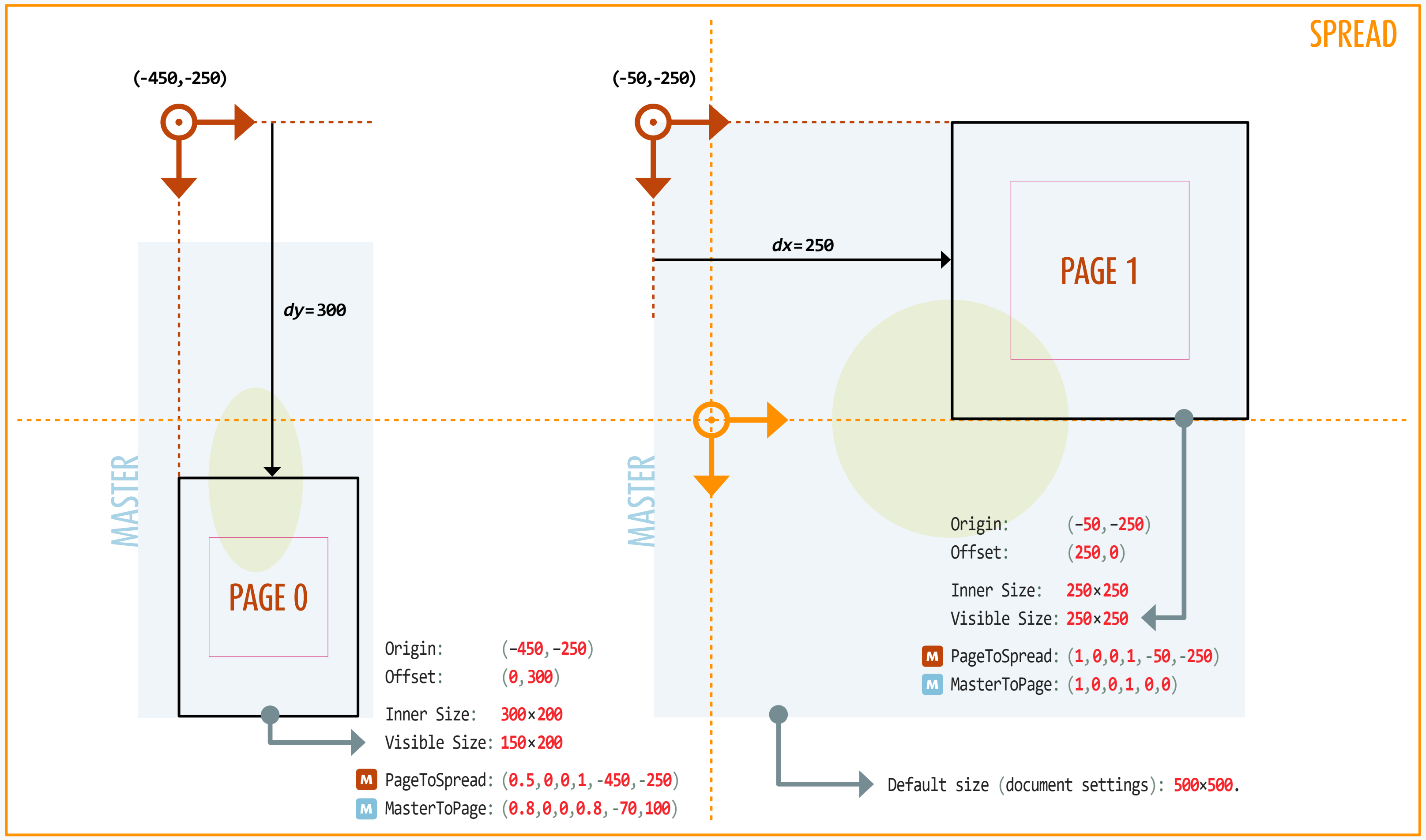


Figure 18.

Sample spread demoing various issues regarding page location and size.

The actual dimensions of PAGE 0 is 300×200 (inner size) but since its affine map *PageToSpread* specifies a 50% scaling along the *x*-axis, it appears reduced to 150×200 in the parent spread. In addition, a custom transformation *MasterToPage* (80% scaling + translation) is applied to its master page relative to the page space* before it undergoes the page-to-spread mapping. The result becomes pretty difficult to predict!

By contrast PAGE 1 has no scaling applied (relative to the spread) and its masterPageTransform matrix is transparent (identity). However it still has a custom size (250×250) relative to the document settings (500×500).

In both cases we can observe that the origin of each page coordinate space—as revealed by the translation values of the respective affine maps—does not coincide with page's top-left corner. A vertical offset (*dy*=300) appears for PAGE 0 and an horizontal offset (*dx*=250) appears for PAGE 1. So, you cannot blindly trust the "page coordinate space" system as defined in Adobe's documentation.

* From *IDML File Format Specification* (version 8.0, page 157): "Because the master page applied to each page can be of a different size than the page, InDesign provides a way to position the contents of the master page as they appear on the page. In IDML, this transformation appears as the MasterPageTransform attribute on the `<Page>` element. While this is a complete transformation matrix, **only translations are supported**." The last statement is wrong! `Page.masterPageTransform` is available in InDesign's DOM from CS5 and it behaves as a fully customizable matrix.

When **DocumentPreference.facingPages** is turned off, in particular, pages within a spread can be freely *repositioned* along both the *x*- and the *y*-axis. Depending on how this is done the user may shift the top-left corner of the page relative to the actual origin of its inner space (see **Figure 18**).

Another issue should be mentioned. The global preference **app.transformPreferences.whenScaling** has a critical impact on how scaling is performed on graphic components, including pages:

➔ **WhenScalingOptions.applyToContent** prevents any scaling operation from being registered as a transformation. In other words, scaling is treated as a *deformation*, meaning that the inner geometry of the target object is actually *resized*. In this context, applying some scaling transformation to a page does not update the scaling values of its affine map. Instead, the actual (inner) size of the page will change.[6]

➔ On the contrary, the option **WhenScalingOptions.adjustScalingPercentage** forces InDesign to manage scaling through transformation matrices, so that the inner geometry of the target is not resized. In other words, scaling a page will not change its actual inner size. What you see from the spread perspective ("visible size") is not what you get on printing or exporting that page—unless you output the spread itself.

For all these reasons, it is worth considering pages as just rectangular items—which they actually are, under the hood—and to compute coordinates for pages as well as for page items, that is, relative to the parent spread space or the pasteboard space (depending on your needs).

As for determining the real size of a page considered as a *device space*, best is to use the bounding box coordinate system, as we shall see.

6. By contrast, transforming a spread never results in a *deformation*. The printable size of a spread is determined by internal rules, disregarding whether that spread is transformed (e.g. *scaled*) and how it is rendered in the perspective of the pasteboard coordinate space.

Finally, remember that the actual parent of any top-level item is a **`Spread`**.[7] Thus, there is no rigid connection between a page and the objects which happen to stand on it.

Inner Coordinate Space of a Page Item

Adobe's documentation does not tell much about basic **`PageItem`**'s inner coordinate space: *"Each page item has its own coordinate space, known as its inner coordinate space. Each page item has an associated parent coordinate space (…)"* (InDesign SDK.)

Although we already know that any coordinate space handles measurements in points and that the associated transformation matrix describes the affine map that connects the inner coordinate space to the parent space, a question remains unanswered:

Where exactly is located the origin of a PageItem inner space relative to its own geometry?

One might intuitively assume that, given a **`PageItem`**, its inner space coincide with either the top-left corner, or maybe the center point, of some intrinsic bounding box. Unfortunately this is usually[8] not the case!

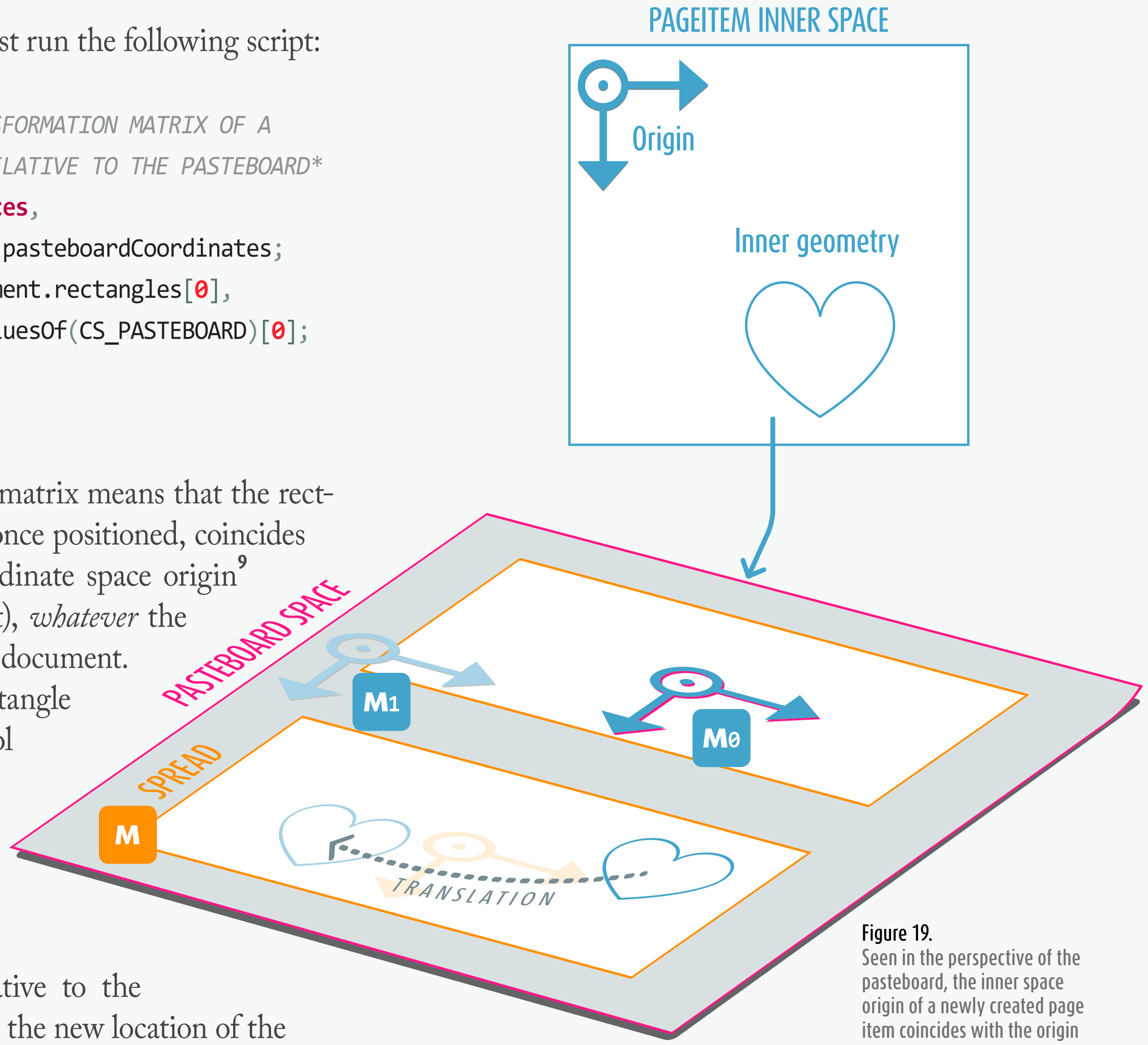
Let's set up a new InDesign document having several empty pages, then draw a basic rectangle on the last page, using the Rectangle tool. Do not move or transform the object, just run the following script:

```
// 04.  DISPLAY THE TRANSFORMATION MATRIX OF A
//      NEW RECTANGLE *RELATIVE TO THE PASTEBOARD*
const CS = CoordinateSpaces,
      CS_PASTEBOARD = CS.pasteboardCoordinates;
var rec = app.activeDocument.rectangles[0],
    mx = rec.transformValuesOf(CS_PASTEBOARD)[0];
alert( mx.matrixValues );
//  Result: 1,0,0,1,0,0
```

The resulting IDENTITY matrix means that the rectangle inner space origin, once positioned, coincides with the pasteboard coordinate space origin[9] (first-spread's center point), *whatever* the location of the object in the document.

Now if you move the rectangle using the Selection tool and re-run the script, you get another result in the form `(1,0,0,1,` tx, ty`)`, where (tx, ty) are the TRANSLATION values relative to the pasteboard space—that is, the new location of the inner space origin within the pasteboard space.

Figure 19 illustrates how moving a shape affects matrix mapping. Let M0 be the affine map of the rectangle in its original state, M the affine map of its parent spread. The product M0 × M (that is, *ItemToSpread* × *SpreadToPasteboard*) results in the *ItemToPasteboard*



M0 × M = `(1,0,0,1,0,0)`

M1 × M = `(1,0,0,1,` t_x, t_y`)`

Figure 19. Seen in the perspective of the pasteboard, the inner space origin of a newly created page item coincides with the origin of the pasteboard coordinate space, whatever the parent spread and the location of the object. Then, when some transformation is applied (e.g. a TRANSLATION) the affine map is updated accordingly. Anyway, note that the "inner geometry" does not change at all. Page item's path points keep the same location relative to the inner coordinate space.

7. This statement is partially wrong in CS4, where a **`PageItem`** could still *belong* to a **`Page`**. However, even in CS4 the *parent coordinate space* of a page-child item is the related spread coordinate space.

8. Exceptionally, **`TextFrame`**'s inner space has its origin *centered in the frame*. But the other **`PageItem`** objects do not follow this special rule.

9. Indeed the (0,0) location in the inner space is *mapped* onto the (0,0) location in the pasteboard space.

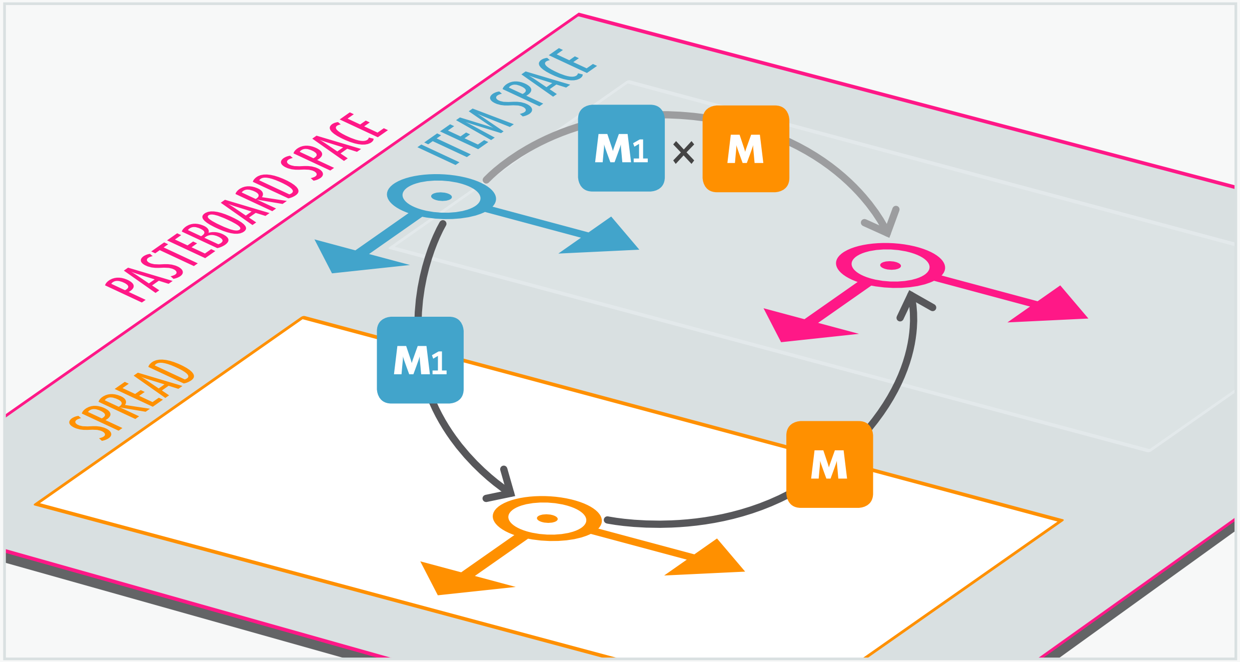


Figure 20.
Chasles' Relation between the *ItemToSpread* matrix M1, the *SpreadToPasteboard* matrix M, and the resulting *ItemToPasteboard* matrix M1 × M.
It is assumed here that the page item is a direct child of the spread. (See **figure 11** for a detailed visualization of matrix products.)

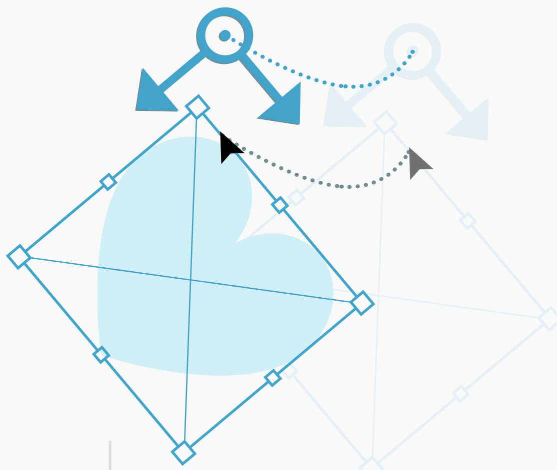


Figure 21a.
Moving a page item, the regular way (TRANSLATION).

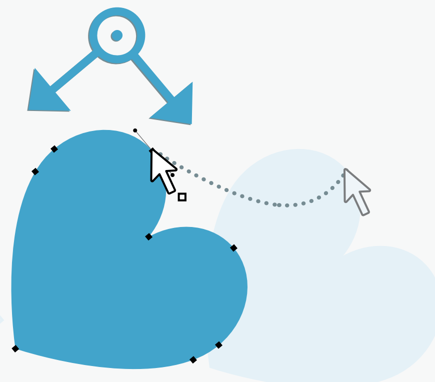


Figure 21b.
Displacing a path, relative to the inner origin.

matrix, which as we have just observed is the IDENTITY mapping. In short, M0 × M = IDENTITY. You can check by yourself that this equality remains true regardless of the transformation state of the spread at the time you create the item. From this we can derive an interesting property: *The affine map of a newly created page item is nothing but the inverse matrix of its parent spread affine map.*

As for M1 × M, this product just reflects the transformation which the page item space globally undergoes *relative to the pasteboard space*, i.e. the TRANSLATION $(1, 0, 0, 1, t_x, t_y)$ in our example. More generally we have something of a Chasles' Relation between transformation matrices. Let

M_{AB} be the matrix that maps space A to space B,
M_{BC} be the matrix that maps space B to space C,
M_{AC} be the matrix that maps space A to space C,

then we have $M_{AB} \times M_{BC} = M_{AC}$.

Figure 20 helps you visualize this rule.

Moving vs. Displacing

When we *move* a page item using the Selection tool, or even when we cut-and-paste this object onto another page, the location of its path points do not change within the inner coordinate space. That the reason why we claim that "moving an object" actually means applying a TRANSLATION to its affine map—or, to put it equivalently, altering the TRANSLATION values of its affine map.

Therefore, most of the time *a move is a transformation*. However, as already observed, some InDesign tools allow the user to perform a *deformation* instead of a *transformation*.[10] In such cases we will rather talk about a DISPLACEMENT to avoid any confusion with regular moves processed through TRANSLATION. See **Figure 21a** vs. **21b**.

10. See Chapter 1, Key Concepts: "Transformation vs. Deformation."

	TRANSFORMATION	DEFORMATION
NAME	Translation (*regular move*)	Displacement (*fake move*)
HOW-TO	Usual means, e. g. select and drag the bounding box of the page item.	"Direct-select" path points and drag them all to another location.
INNER GEOMETRY	Unchanged.	Changed (relative to the origin).
LOCATION OF THE ORIGIN	Changed (relative to the parent space).	Unchanged.

2. InDesign Coordinate Spaces

SUMMARY

InDesign defines several coordinate spaces relative to which object coordinates are interpreted and transformations are processed. All “InDesign coordinate spaces” rely on a 2D ORTHOGONAL basis listed in CLOCKWISE order and using POSTSCRIPT POINT as canonical unit.[1]

➔ The PASTEBOARD COORDINATE SPACE surrounds the whole document. Its origin is the center point of the first spread. It has no affine map (as it represents the top of the hierarchy). In the Scripting DOM it is referred to as `CoordinateSpaces.pasteboardCoordinates`.

➔ A SPREAD COORDINATE SPACE reflects the inner space for a specific `Spread` object. Its affine map, usually a TRANSLATION, links it to the pasteboard space. Its origin is the center point of the underlying spread. In the Scripting DOM it is referred to as `CoordinateSpaces.spreadCoordinates`.

➔ A PAGE COORDINATE SPACE reflects the inner space for a specific `Page` object. Its affine map links it to the parent spread space. Its origin is, *in most cases*, the top-left corner of the page. In the Scripting DOM it is referred to as `CoordinateSpaces.pageCoordinates` in InDesign CS6 and later.[2]

➔ Any `PageItem` has an associated inner coordinate space, whose affine map is connected to the parent coordinate space—which can be either a `Spread` space, or another `PageItem` space depending on the document hierarchy.[3] The inner space origin of a `PageItem` usually coincides with the pasteboard origin while the object is created, then subsequent moves are reflected in TRANSLATION components. The `TextFrame` object is an exception: its inner origin is originally located *at the center of the bounding box.*

➔ The Scripting DOM provides a special keyword, `CoordinateSpaces.parentCoordinates`, that refers to the parent coordinate space of any *transformable* object. The code `myObject.transformValuesOf(CoordinateSpaces.parentCoordinates)[0]` returns a `TransformationMatrix` that describes the affine map applied to `myObject`’s coordinate space.

Depending on InDesign versions, `Spread` and/or `Page` coordinate spaces can be transformed in unexpected ways using ROTATION, SCALING, and SHEAR. Tranformed pages may reveal obscure artifacts: shift of the default origin location, difference between “inner size” and “visible size,” etc.

CHASLES’ RELATION applies to matrix product. Given three coordinate spaces A, B, and C, if the matrix M_{AB} maps A to B while the matrix M_{BC} maps B to C, then $M_{AB} \times M_{BC}$ maps A to C.

EXERCISES

001. The Scripting DOM exposes an enum we haven’t discussed yet: `CoordinateSpaces.innerCoordinates`. Try to predict its usage and the result of

```
anyObj.transformValuesOf(CoordinateSpaces.
innerCoordinates)[0].matrixValues
```

where *anyObj* may refer as well to a `Spread`, a `Page`, or any `PageItem` in any transformation state.

002. Let’s set up a new `Document` having a single `Page` (non-facing mode). In case *(a)* we draw an `Oval` at the center of the page, then we rotate the spread view 90° clockwise. In case *(b)* we rotate the spread view 90° clockwise, then we draw an `Oval` at the center of the page. Are there differences between *(a)* and *(b)* in terms of transformation matrices? Provide a script that corroborates your answer.

003. Using the Selection tool, select the left edge of a `SplineItem`’s bounding box and drag it to the left so that the width of the underlying shape is increased. Does this change the associated affine map? Provide a script that corroborates your answer.

004. A document has a unique `Page` whose affine map is `(0.5,-0.25,0.25,0.5,-125,-125)`. The containing `Spread` has a 180° “rotated view” applied. What is the transformation matrix of the page coordinate space relative to the pasteboard coordinate space?

---

1. Bounding box and rulers coordinate systems are not pure coordinate spaces. Those systems will be discussed in the next chapters.

2. Prior to CS6 a page coordinate space has no dedicated enum value but it already makes sense, as `CoordinateSpaces.innerCoordinates`, in the scope of a `Page` object.

3. Fortunately (?) the parent coordinate space of a `PageItem` space cannot be a page coordinate space, despite the fact that *in CS4* `PageItem.parent` may refer to a `Page`. (See also page 17, note 7.)

A bounding box is *the smallest rectangle that encloses a geometric page item.** This definition intuitively corresponds to what we perceive as a selection frame in the interface. When the user is in drawing a vector shape and switches to the Selection tool s/he immediately sees a rectangle bordering the entire path. This sounds dead easy at first sight, but now it's time to open that black box.

A Bounding Box Depends on a Coordinate Space

You might think that a given page item has a single, uniquely determined BOUNDING BOX. But does it really make sense to talk about *the* smallest rectangle that encloses a shape? **Figure 22** shows that we can construct as many enclosing rectangles as desired, depending on a chosen orientation.

Here is a cleaner definition of a bounding box in InDesign's workspace: *Given a* COORDINATE SPACE *and given a geometric object, the associated* BOUNDING BOX *is the smallest rectangle that encloses the shape with respect to the specific axes and orientation of that coordinate space.*

Each coordinate space governs how to frame an object. Unlike InDesign's GUI, which always displays bounding boxes relative to the inner space,[1] a script can select any other reference frame (parent space, spread space, pasteboard space).

From then we can discern at least four distinct bounding boxes for the same page item:

➔ The INNER SPACE RELATIVE BOUNDING BOX (which we will abbreviate to "inner box" from now on). It is the enclosing rectangle aligned with the axes of the *inner* coordinate space.

➔ The PARENT SPACE RELATIVE BOUNDING BOX (in short, the "in-parent box"). It is the enclosing rectangle aligned with the axes of the *parent* coordinate space.

➔ The SPREAD RELATIVE BOUNDING BOX (in short, the "in-spread box"). It is the enclosing rectangle aligned with the axes of the *spread* coordinate space. Of course the in-spread box *is* the in-parent box if the page item has no intermediate owner along the hierarchy. Also, if the object we consider is the spread itself, its in-spread box is de facto its inner box.

➔ The PASTEBOARD RELATIVE BOUNDING BOX (in short, the "in-board box"). It is the enclosing rectangle aligned with the axes of the *pasteboard* coordinate space, that is, the horizontal and vertical axes of your screen. As long as the containing spread is untransformed (no rotated view applied, etc.), the in-board box of a page item coincides with its in-spread box.

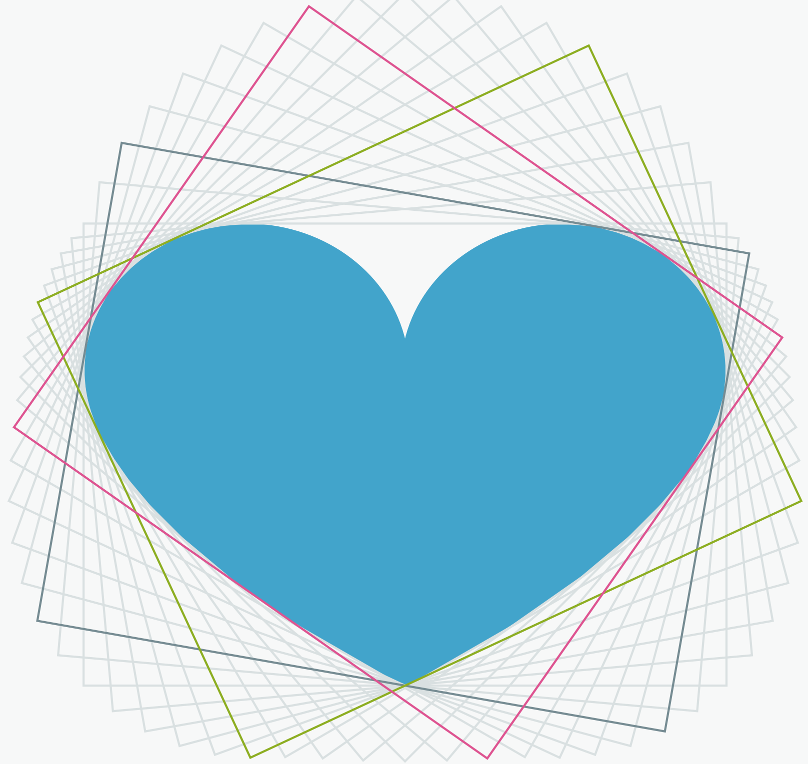


Figure 22.
In the absence of further instructions there are infinite ways of enclosing a shape in a rectangular region. In InDesign geometry, several bounding boxes can be defined for the same object depending on the coordinate space we consider.

---

* *InDesign SDK*, "Bounding box and IGeometry."

1. Except when multiple items are selected; in such case InDesign shows the bounding box aligned with the pasteboard space.

In CS6 and later you could also consider the “in-page box” of an object, that is, the bounding box aligned with the associated page coordinate space, provided that the object actually meets a page. In practice it is discouraged to rely on in-page boxes because of the many inconsistencies already mentioned about page coordinate spaces.[2]

A Bounding Box Defines a Coordinate System

A bounding box must be understood as an oriented rectangle in that it defines a system of anchor points in clockwise order, TOP-LEFT, TOP-RIGHT, BOTTOM-RIGHT, BOTTOM-LEFT—with respect to the orientation of the associated coordinate space.

This set of anchor points makes bounding boxes similar to coordinate spaces although they do not use the same origin, nor the same units. To avoid any confusion we shall talk about *bounding box coordinate system*, or “box system.”[3]

Box systems use the top-left anchor as the origin. The distance from the top-left anchor to the top-right anchor provides the *horizontal unit*, while the distance from the top-left anchor to the bottom-left anchor provides the *vertical unit*. It follows in particular that

2. The best you can do is to handle the inner box of a given page (that is, its own bounding box relative to its inner space) then to resolve associated anchor points into spread or pasteboard coordinates. Considering page bounding box is helpful when you have to fix page size and/or location issues (see previous chapter, page 15).

3. Adobe’s documentation may refer as well to “bounds space,” keeping in mind that such coordinate system is not strictly a coordinate space. (Coordinate spaces are discussed in Chapter 2.)

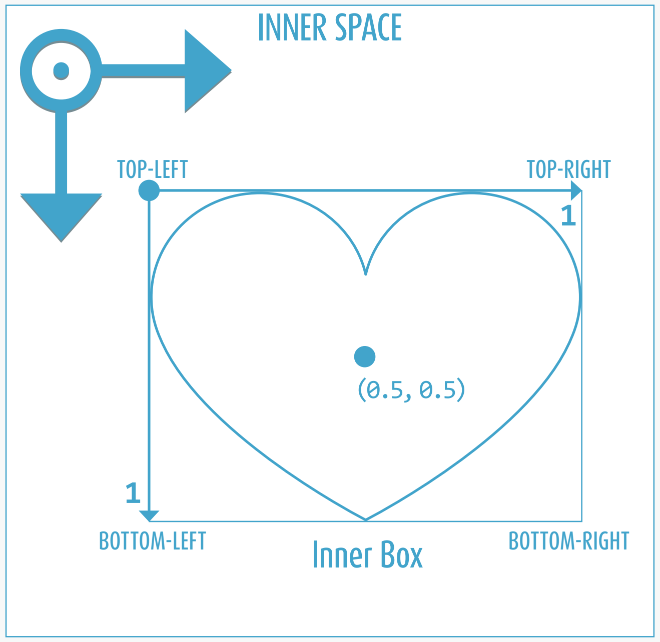
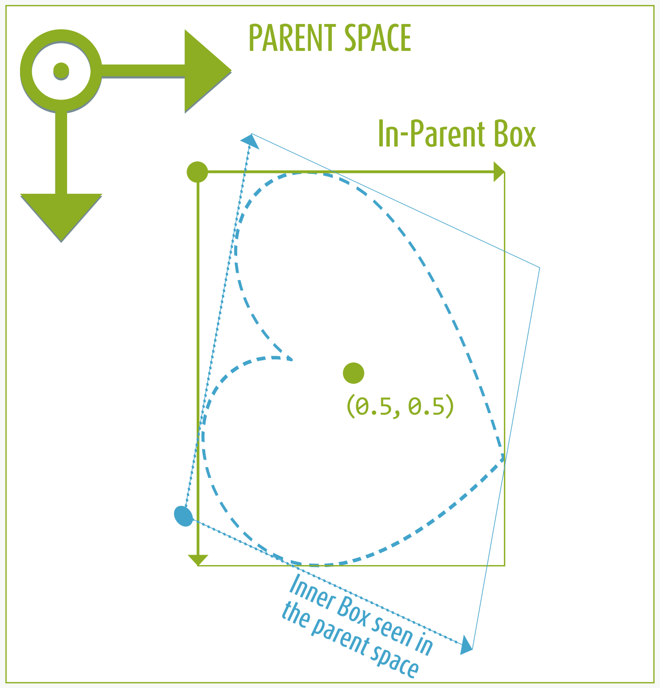


Figure 23.
Inner box (blue) and in-parent box (green) of a page item. Seen in its associated coordinate space a bounding box is always rectangular and oriented as the space basis. However these properties may be lost if the box is observed in the perspective of another space (as shown below).



the center anchor point of the bounding box has the coordinates `(0.5,0.5)` whatever the actual dimensions of the enclosing rectangle (see **Figure 23**). Using this system—or dedicated enumerators—a script can easily specify any anchor point location. It can also access to other relative locations, such as `(0.25,0.75)`, `(-1,0)`, `(2,3)` etc. Note that box systems allow to define locations outside the bounding box.

While a given shape has different bounding boxes (one for each connected space) and then different bounds coordinate systems, an interesting property is that the location of the center anchor point remains invariant.

Path Bounds vs. Visible Bounds

A bounding box may or may not fit the path stroke or other border effects that affect the *visible bounds* of a page item, such as rounded corners.

InDesign’s GUI always renders visible-bounds-dependent bounding boxes, which the Scripting DOM refers to as **`outerStrokeBounds`**[4] or **`visibleBounds`**, in contrast with the inherent path bounds, referred to as **`geometricPathBounds`**[4] or **`geometricBounds`**.

At the scripting level one can always refer to either the “path box” or the “visible box,” meaning that *two* box systems are actually available for any spline item we consider. Selecting the right system is of primary importance when moving or resizing objects within a specific region of your layout.

4. Those two values are exposed by the **`BoundingBoxLimits`** enumeration, which plays an important role in the **`resolve`** method, as we shall see later.

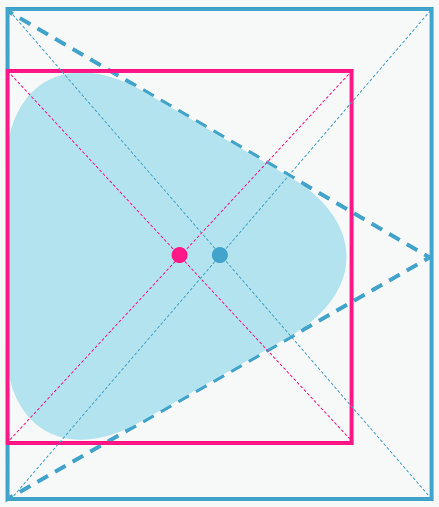
The figure below (see **Figure 24**) shows that a visible box can be larger, or narrower, than its corresponding path box. As an immediate consequence the associated coordinate systems do not match. For instance, the path box system origin (blue point) is not at the same location as the visible box system origin (magenta point.)[5]

However, relative to some coordinate space, the path box and the visible box of an object always have the same orientation. Therefore we could describe the relationship between these two systems by a transformation matrix that only involves, in the worst case, SCALING and TRANSLATION components.

Also, since **Spread** and **Page** objects never undergo stroke effects, their inner path[6] and visible boxes are always identical.

---

5. We can even show that the visible box and the path box do not necessarily share the same center point `(0.5,0.5)`:



6. In fact, talking about "path" boxes is something of a misnomer for spreads and pages, as those objects are not spline items at all.

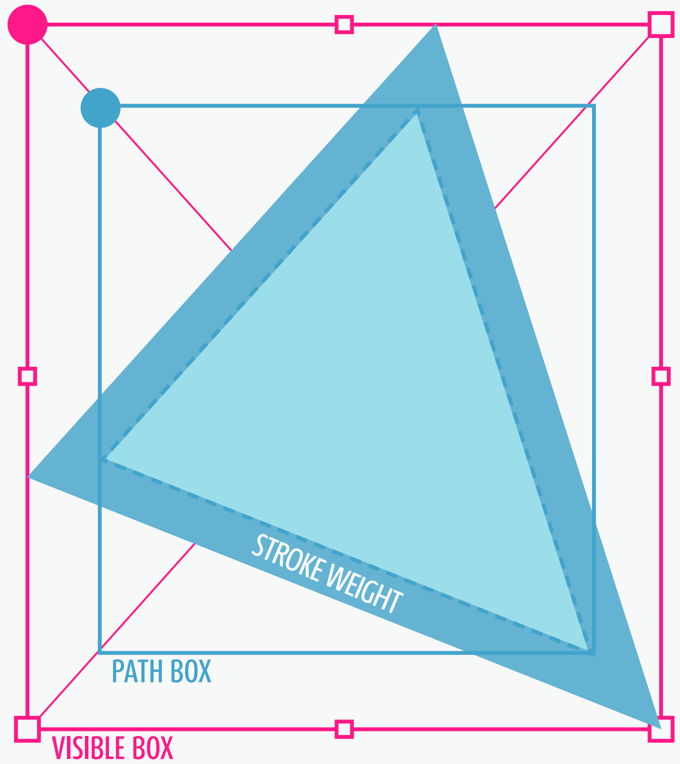
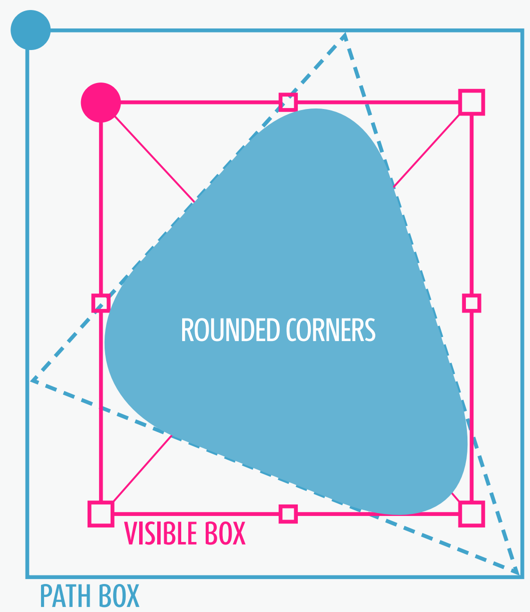


Figure 24.
Depending on applied stroke effetcs the visible box (magenta) can be larger or smaller than the path box (blue.) In the top figure the triangle has a solid border (outer stroke) that increases its visible box. In the bottom figure rounded corners have been applied to the triangle, so that the visible box is smaller than the path box.



SUMMARY

Given a coordinate space S and an object O being either a page item, a page, or a spread, the *visible bounding box* of O *relative to* S is the smallest rectangle in S that encloses O with respect to S's basis.

In addition, the *path bounding box* of O *relative to* S is the smallest rectangle in S that encloses the path of O (disregarding stroke effects) with respect to S's basis.

Each bounding box determines a coordinate system (not a coordinate space) having its top-left anchor point as the origin, the basis being defined by the top-left to to top-right vector (x-axis) and the top-left to bottom-left vector (y-axis.)

EXERCISES

001. Let T be an equilateral triangle having some side aligned with the bottom line of the associated path box. Express the location of the geometric center of T in the inner path box system.

002. Suppose that InDesign's GUI displays the bounding box of a page item as a non-rectangular frame. Explain why this frame is anyway a parallelogram. Show that the transformation matrix which maps the inner space of that page item to the pasteboard space necessarily contains some SHEAR component.

003. Can the *path* box and the *inner* box of a spline item coincide—in any given space—if this object has a nonzero outer stroke weight?

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.

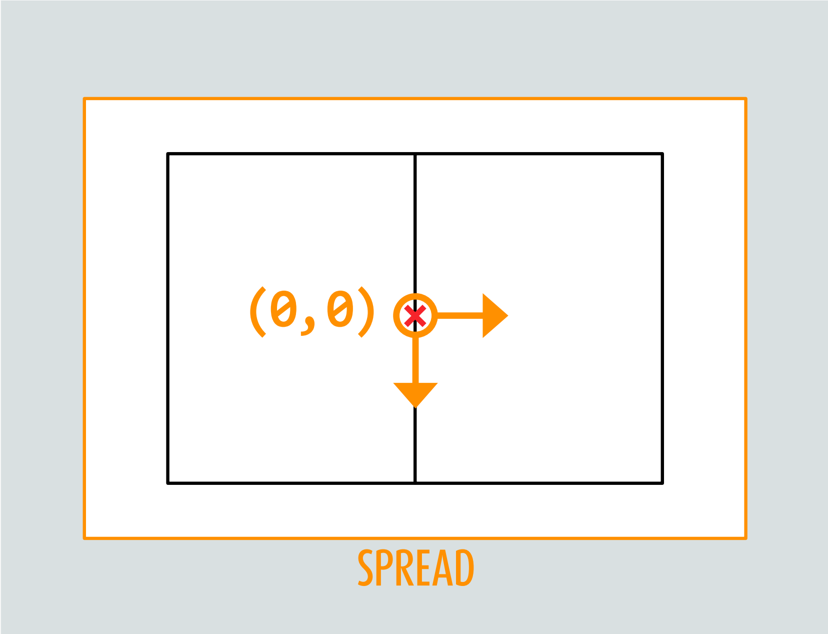
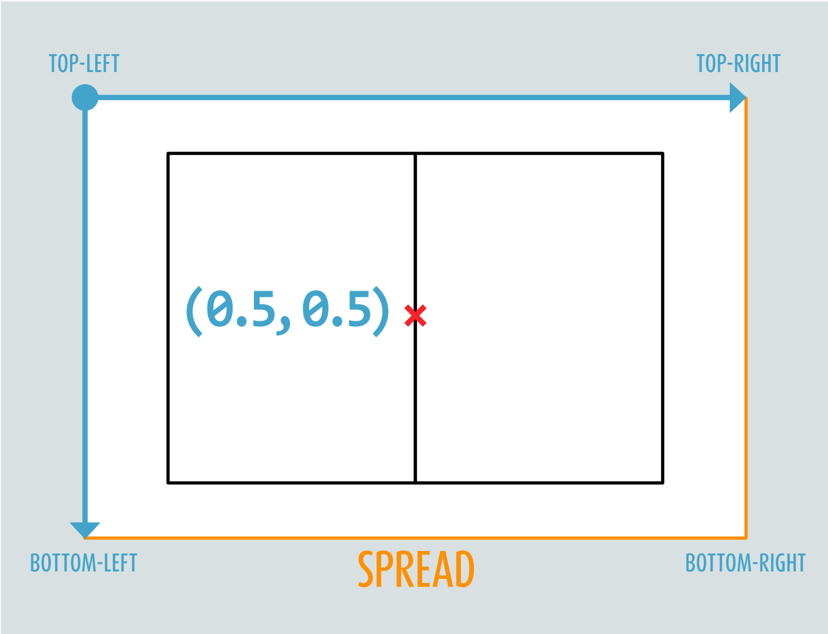


Figure 25.
A same location (red cross at the center of the spread) can be expressed by different coordinates depending on the system we consider,
`(0,0)` in the spread coordinate space *(top)*,
`(0.5,0.5)` (i.e. centerAnchor) in the spread bounding box system *(bottom)*.



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, alternatively, 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.

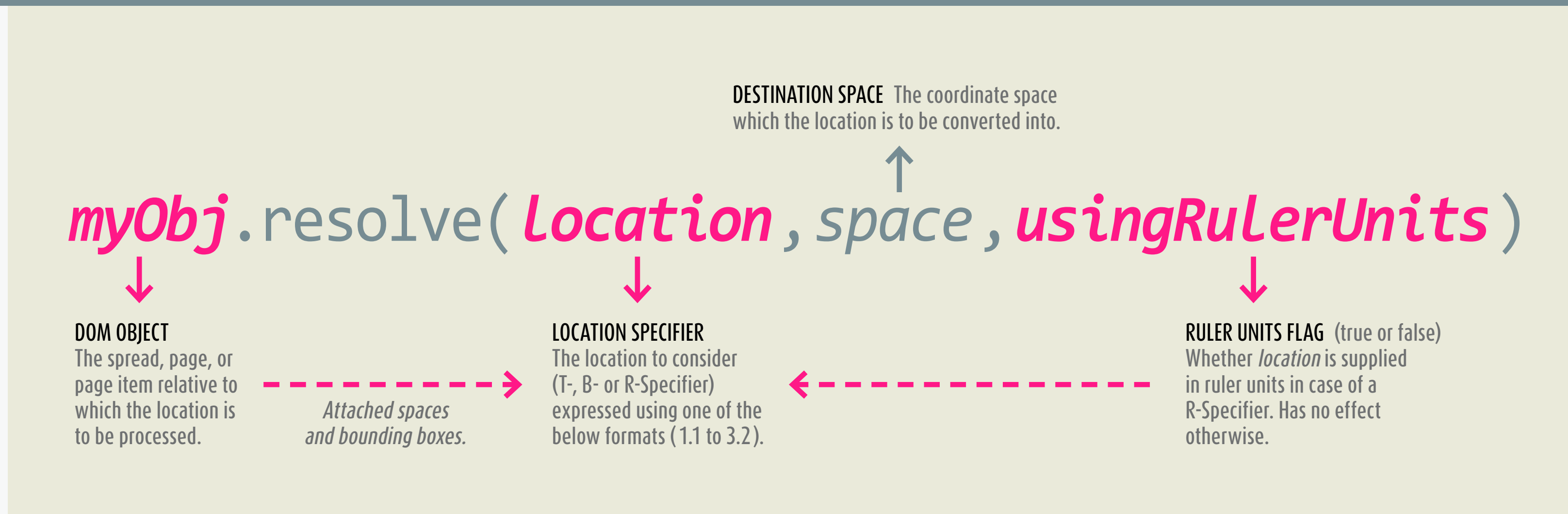


Figure 27. All layout objects (incl. spreads and pages) expose a **resolve()** method which plays a crucial role in solving location issues. It takes a location specifier of any kind (with respect to the incoming object) and returns a singleton array having for unique element a coordinate pair [*x*,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

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*.

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]

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.

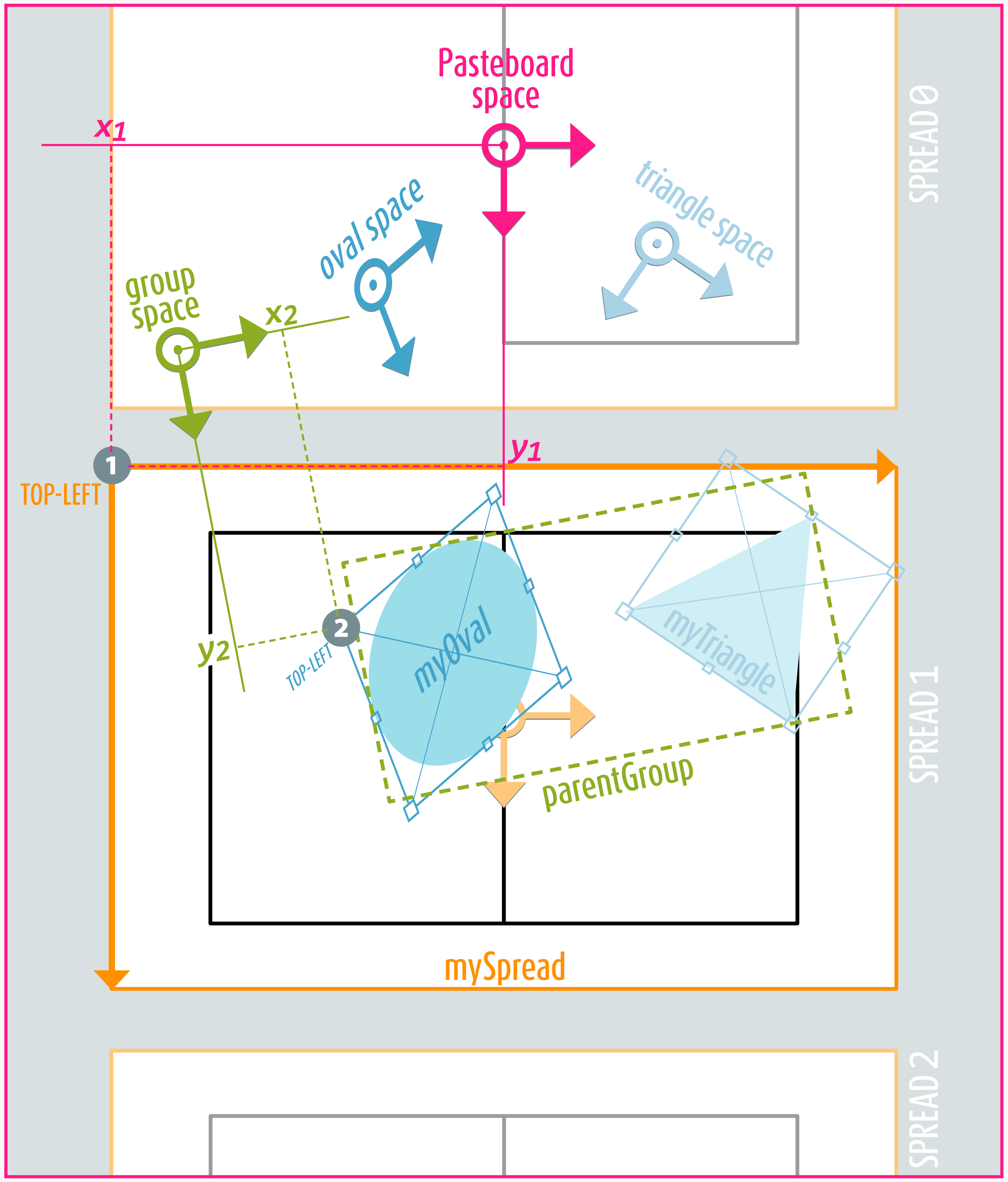


Figure 28. In this layout various transformations have been applied to the underlying coordinate spaces (oval, triangle, and parent group), and their origins have been randomly positioned to make it clear that they don’t necessarily coincide.

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`.)

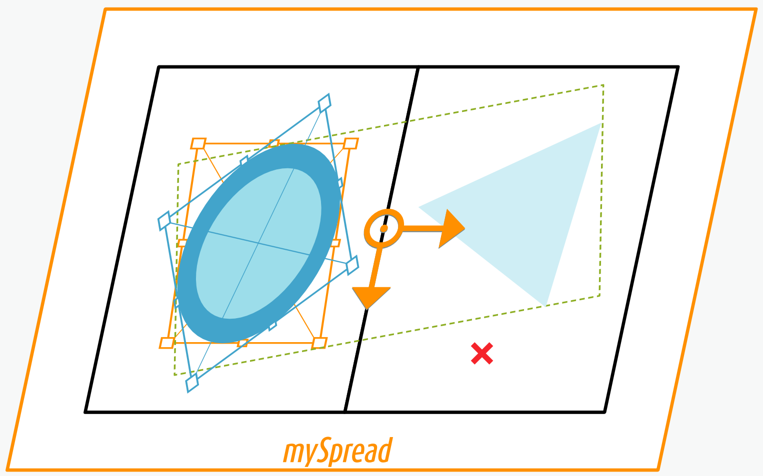


Figure 29.
The best strategy for solving location issues is to suppose all coordinate spaces are in a nontrivial transform state relative to each other along the hierarchy. This way one can discern parameters and relationships that would otherwise be coincident and unnoticeable. In this layout both the 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]`.

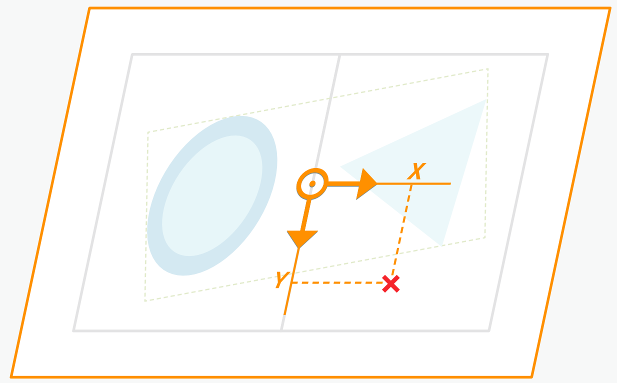
The whole question is therefore to implement a function that takes `myObj`, `[x,y]`, `refSpace`, `boxLimits`, and `boxSpace`, then outputs `[u,v]`.

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

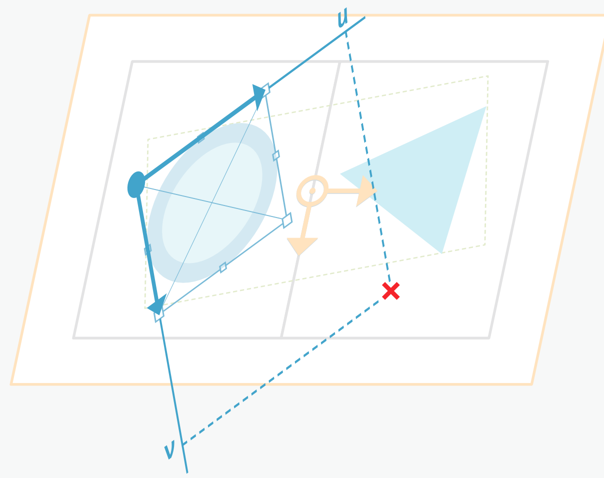
`myObj`	The blue oval (child of a group)
`[X,Y]`	The red cross coordinates in `refSpace`
`refSpace`	`CoordinateSpaces.spreadCoordinates`
`boxLimits`	`BoundingBoxLimits.outerStrokeBounds`
`boxSpace`	`CoordinateSpaces.innerCoordinates`,

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:

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

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

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

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

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

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

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

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.

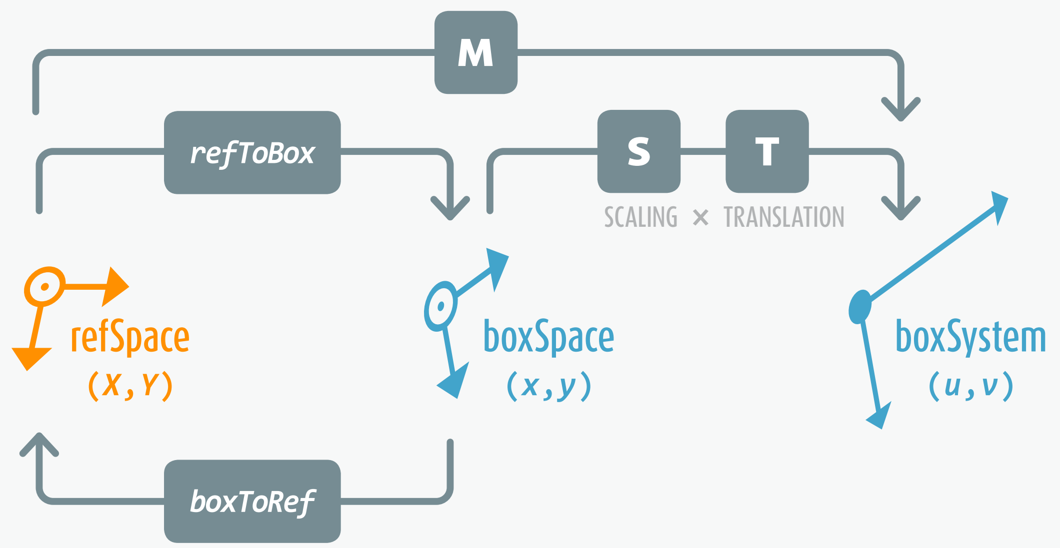


Figure 30. What InDesign can instantly reveal us (through `transformValuesOf`) is the matrix ***boxToRef*** which maps an inner space (*boxSpace*) to any coordinate space (*refSpace*.) In order to find the matrix ***M*** that maps *refSpace* to a bounding box system (*boxSystem*) we need extra calculations. 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 $M_{AB} \times M_{BC} = M_{AC}$.

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.

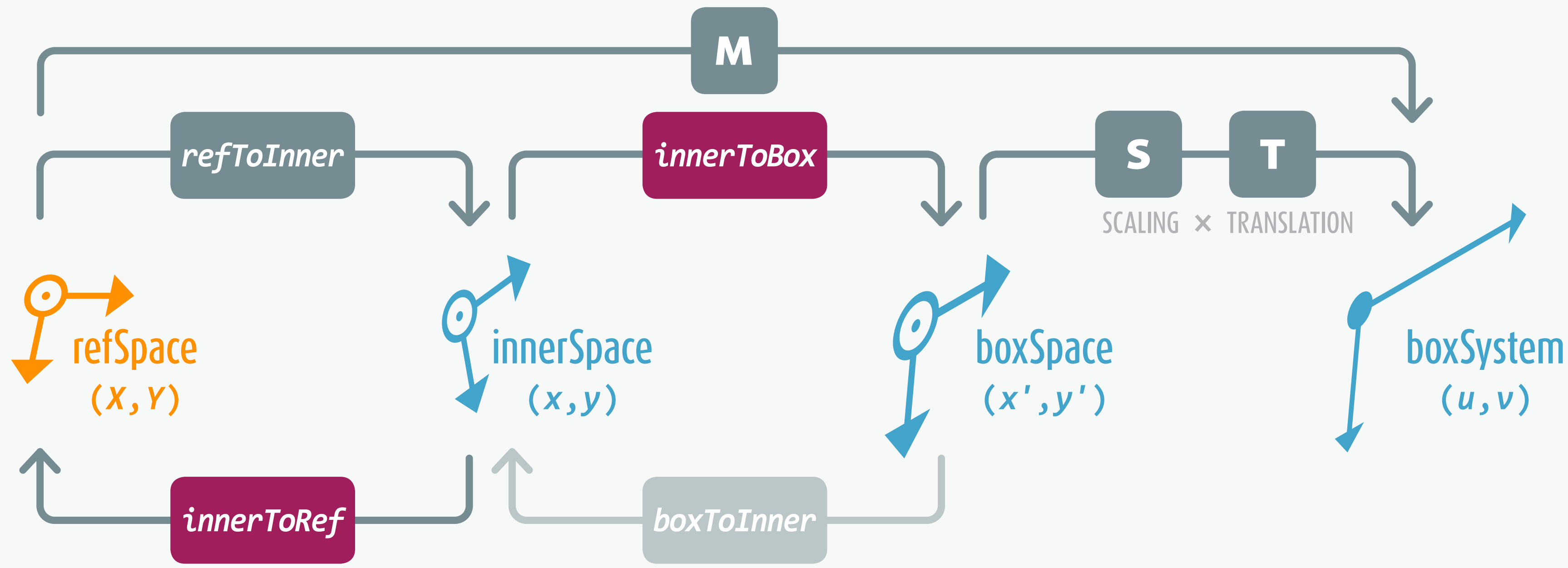


Figure 31.
Improved version of the "T2B" algorithm. It is not assumed anymore that the desired box space matches the inner space. In other words the bounding box can be observed from a different perspective, e.g. the pasteboard space. The whole transformation (*M*) now involves the following matrices: ***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]

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.)

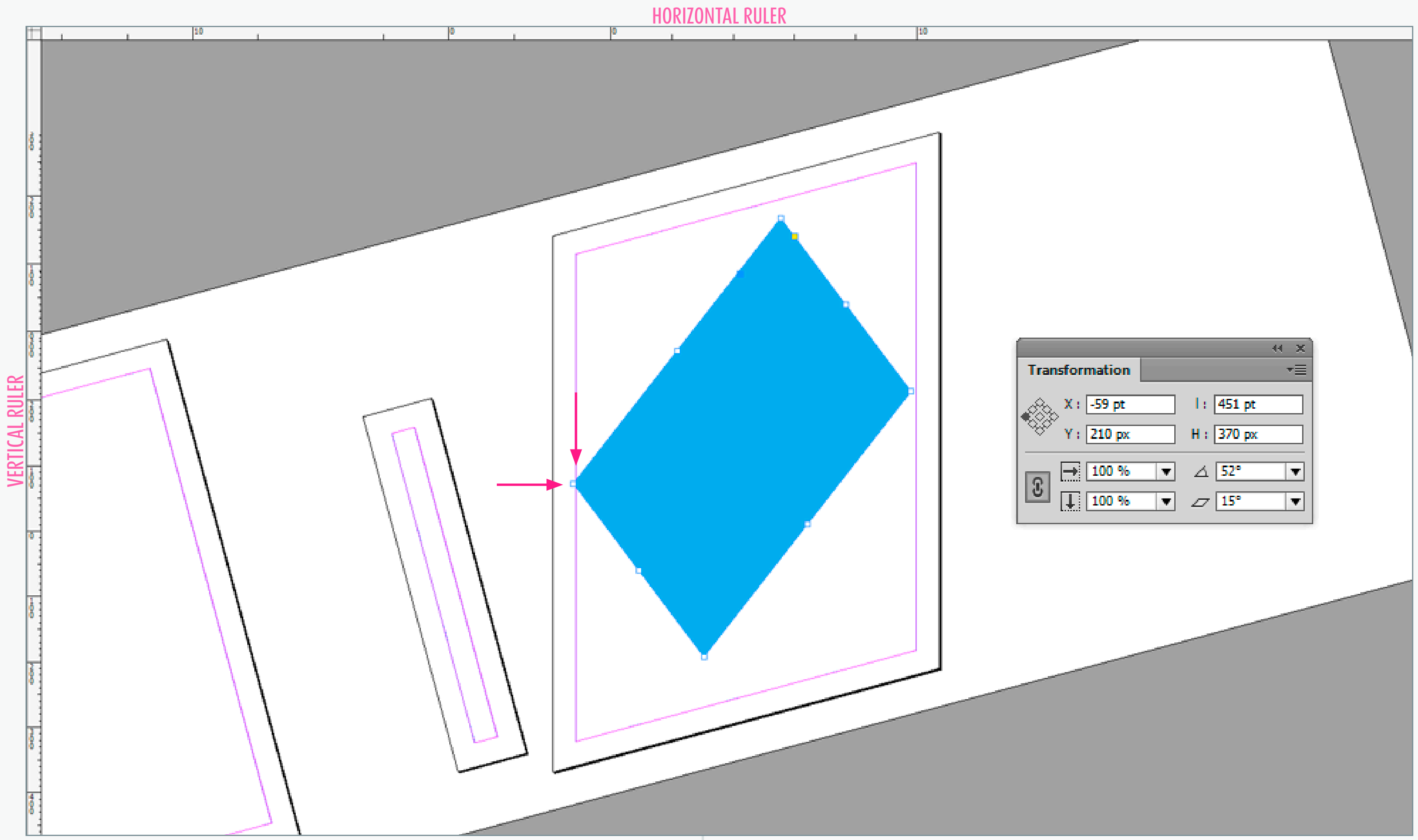


Figure 33. Screenshot of InDesign's viewport under wild settings (custom spread rotation, skewed page, custom units, randomly positioned Zero Point…) Problem now is to properly use ruler coordinates for parsing and processing locations, bounds, width, height of the blue rectangle!

➔ 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`, 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 area of all pages (Fig. 34b.)	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 bounding box. Should the pages undergo unusual transformations, that location remains fully determined.

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

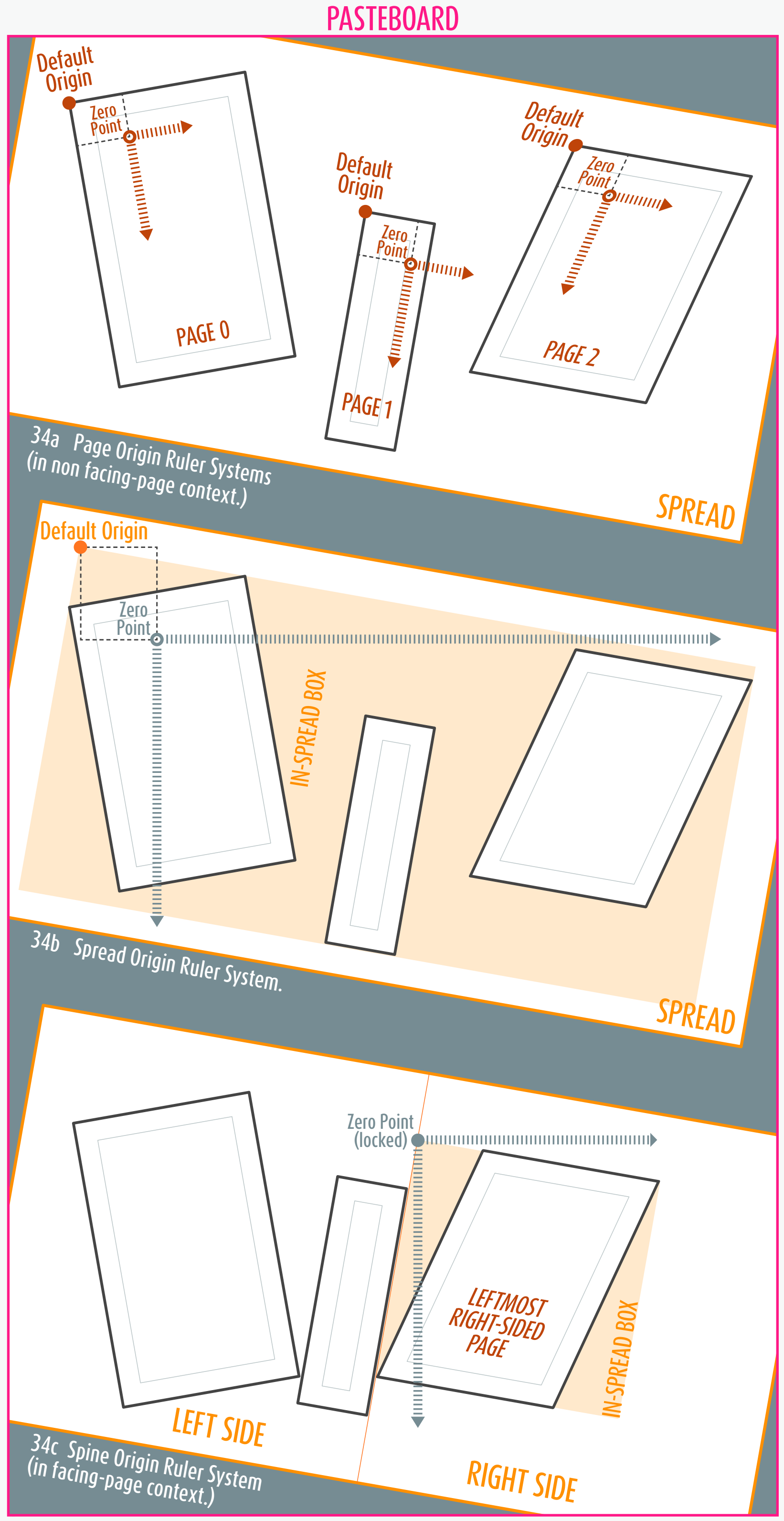
In all other cases, the *in-spread* bounding box of the page[23] is considered, and *axes are oriented as the pasteboard space*. **Figure 34** shows these distinct systems.

➔ 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 may call a RULER SYSTEM.[25]

Details About Page-Based Ruler Systems

The **pageOrigin** mode (**Figure 34a**) is undoubtedly the most complex. *Each* page then has a dedicated ruler system (while single ruler system is assigned to the whole spread in **spineOrigin** and **spreadOrigin** modes.) Also, in non facing-page layouts, the actual orientation of the page rulers fits the inner space of the pages, although InDesign still displays "horizontal"

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.)



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."*

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 **spineOrigin** 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.

and "vertical" rulers aligned with the document window! Thus the coordinates visible in the Transform panel may become quite obscure under various conditions.

In addition, a single location in the spread can be expressed by different ruler coordinates: one for each page! A script can identify 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 the spread under consideration is clearly known, since every DOM method is triggered from an object which has a definite, implied parent spread.

On the contrary, page-based ruler systems require *the page under consideration* to be identified within the implied spread. This explains the weird thoroughness of R-SPECIFIER's formats, as already detailed:

3.1 `[[rx,ry],<PAGE_INDEX>]`,

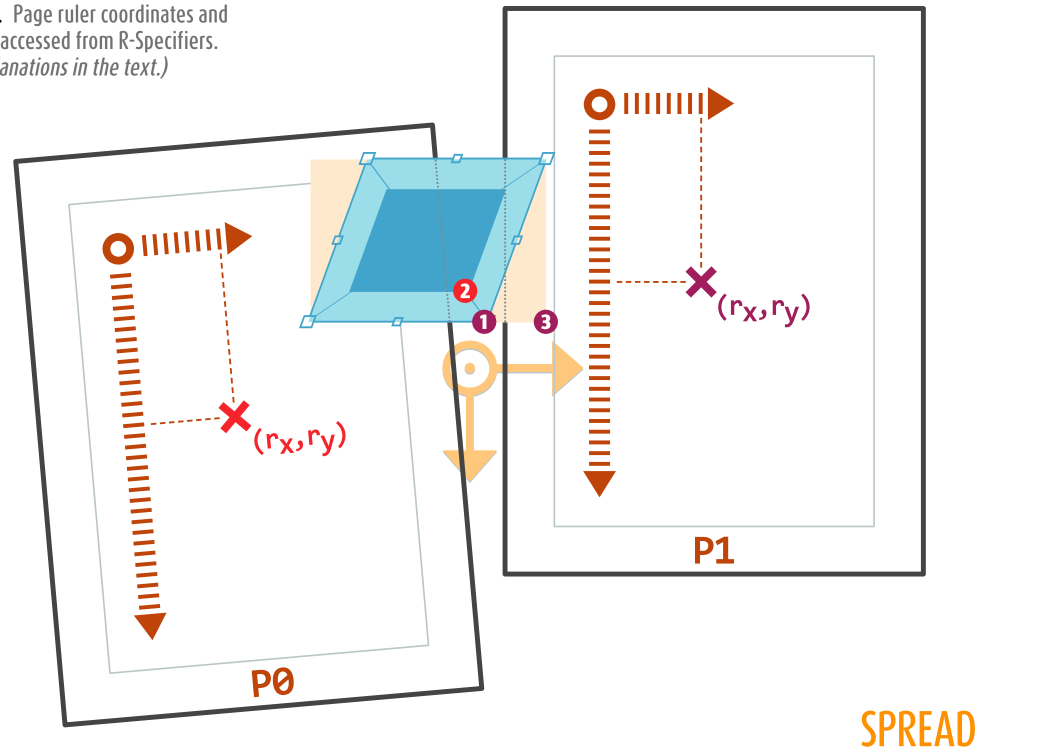
3.2 `[[rx,ry],<PAGE_LOCATION>]`.

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.

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 in-spread* box.[27]

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



Then, still assuming that the active ruler mode is `pageOrigin`, let's choose a coordinate pair (r_x, r_y). So far we don't know whether (r_x, r_y) should refer to the *red* cross (page `P0`) or to the *purple* cross (page `P1`.)

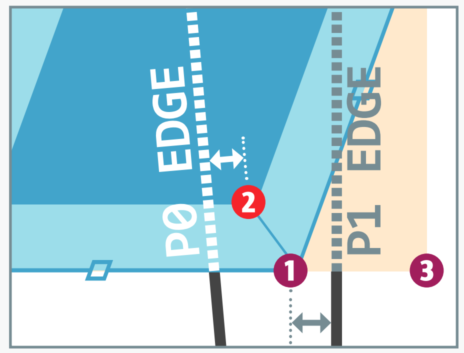
Consider the following R-SPECIFIERS:

– `[[rx,ry],` ❶`]`,
– `[[rx,ry],` ❷`]`,
– `[[rx,ry],` ❸`]`.

Although the exact same numbers are involved in terms of ruler coordinates, the resulting locations are respectively:

– the *purple cross* in cases ❶ and ❸,
– the *red cross* in case ❷.

Indeed, ❶ and ❸ implicitly refer to `P1` as it is the nearest page: ❸ belongs to it, ❶ is closer to `P1`'s left edge than to `P0`'s right edge. By contrast ❷ refers to `P0` since the corner is now a bit closer to `P0`'s right edge, as shown 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**.

SUMMARY

InDesign's scripting DOM provides three distinct ways of specifying *locations* in transform-wise methods.

A TRANSFORM-SPACE location is relative to a *regular coordinate space*. Its complete specifier looks like `[[x,y],<SPACE>]`. For example, `[[0,0],CoordinateSpaces.spreadCoordinates]` refers to the origin of the spread space. A shorter form, `[x,y]` alone, implicitly relates to the pasteboard space.

A BOUNDS-SPACE location determines a position *relative to a bounding box system*. The complete specifier looks like `[[u,v],<BOX_LIMITS>,<SPACE>]` where `[u,v]=[0,0]` represents the top-left anchor and `[u,v]=[1,1]` the bottom-right anchor. Usual `AnchorPoint` enumerators can be used rather than (u,v) coordinates. `<BOX_LIMITS>` stands for a `BoundingBoxLimits` enumerator. If specified, `<SPACE>` indicates the coordinate space of the box, otherwise the *inner box* system is considered. Shorter forms are available. In particular, a simple `AnchorPoint` enumerator abbreviates `[<ANCHOR_PT>,BoundingBoxLimits.outerStrokeBounds,CoordinateSpaces.innerCoordinates]`.

A RULER-SPACE location is relative to the current GUI rulers and user preferences (namely the custom `zeroPoint` and the `rulerOrigin` option.) The usual specifier looks like `[[rx,ry],<PAGE_INDEX>]`, where the ruler coordinates (r_x, r_y) may be interpreted either in points (default), or in custom ruler units if the parameter *`usingRulerUnits`* is set to true in the invoked method.

The `resolve()` method allows to convert any specified location into coordinates within a regular coordinate space. `<OBJ>.resolve(<LOCATION>,<SPACE>)` returns a *singleton array*[28] whose unique element is the desired coordinate pair (array of two numbers) expressed in the `<SPACE>` system. `<OBJ>` can be any DOM object that supports transformations: `Spread`, `Page`, `Group`, `Graphic`, and of course any `SplineItem`.

There is no direct way to resolve a location *into* BOUNDS-SPACE or RULER-SPACE coordinates. We provided an algorithm for converting TRANSFORM-SPACE coordinates into bounding box coordinates *(see page 30.)*

EXERCISES

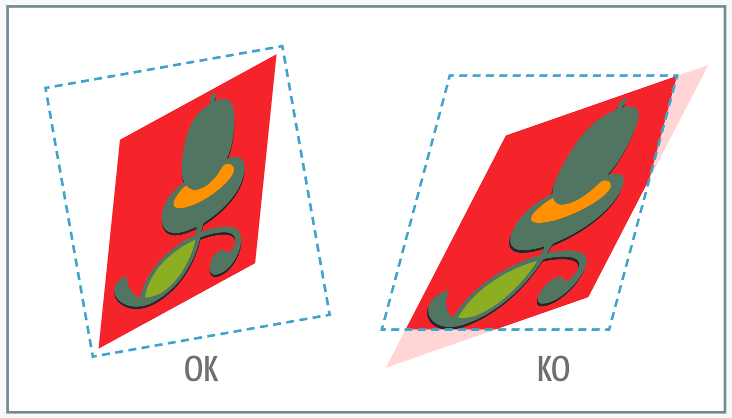
001. Let G be a `Group` formed of three circles (`Oval` objects.) Keeping in mind that G might undergo some rotation as well as other transformations, write a script that checks whether the centers of the circles are aligned. *Constraint*: use bounds-space locations.

002. Given a multi-spread document, how would you calculate the vertical distance—in the pasteboard— between two given pages?

28. `resolve()` does not exactly work as we would expect on plural elements accessed through `everyItem()`. For example, *`myGroup`*`.pageItems.everyItem().resolve(...)` returns a singleton array whose unique element is an array of n coordinates, n being the number of page items. Fine! These coordinates are correct as long as they rely on a location explicitly attached *to the inner space*. But if the parent space is involved, *all coordinates will be identical*, as emanating from the group itself. A typical example is `AnchorPoint.centerAnchor`, which (wrongly?) relates to the center point of the group. A workaround is to use the full specifier syntax (in the inner space.)

003. Suppose a facing-page document contains five pages within a single spread. What are the (x,y) coordinates, expressed in `spreadCoordinates` space, of the upper left corner of each `Page`? What are the (u,v) coordinates, in the spread box system, of the exact center point of each `Page`?

004. Let a bitmap `Image` X belong to a `Rectangle` R (assumed without stroke weight or rounded corner.) Both X and R may undergo independent transformations of any kind, including rotation and/or shear. Provide a code that checks whether X's area is *entirely enclosed*[29] in R.



005. Compute the perimeter of any `Polygon` from its path points, whatever its transform state. *(Your script must return the length, in points, as perceived in the pasteboard, without altering user preferences, measurement units or other ruler settings.)*

29. For an advanced discussion on this topic, see http://indiscripts.com/post/2016/12/indesign-scripting-forum-roundup-10#hd2sb1

Chapter 1 told you all needed about affine maps, transformations and matrix product. Then we studied coordinate spaces and locations, the (many) bounding boxes available and those tricky ruler coordinates. Yet we still don't know how the Scripting DOM actually performs a transformation. It's time to act!

The "Transform Space" Enigma

Really, it took me years to clear up the **`transform(…)`** method. Not because the process of *transforming* is obscure (after all, we only need to compute matrices), but because of the very first argument of the function: *"the coordinate space to use."* Seriously, why should I specify a coordinate space since I already know which object map needs to be processed? Isn't it obvious that calling ***`myObj`*`.transform(…)`** just means, *"Hey, take a transformation matrix and blend it with the affine map of myObj"*?

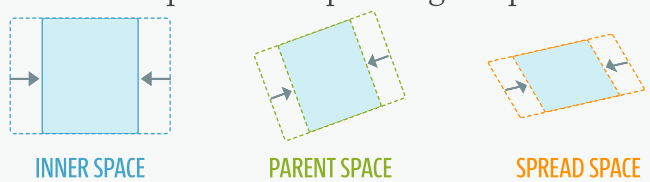
Think about it for a few seconds. Say you have a **`Rectangle`** somewhere in a document. Maybe it belongs to a complex group, maybe it contains children itself, and maybe all this stuff already undergoes a slew of nested transformations (rotations, scaling, at your pleasure!) Anyway if our goal is to apply a new transformation—say, SCALING—to *that* rectangle, this specifically concerns its affine map, that is, the relationship between its inner coordinate space and its parent space. This ultimately amounts to changing the existing attributes by asking InDesign to calculate a matrix product. So, again, why may we supply *another* coordinate space?

The answer is more exciting than the question. While the scenario just limned is almost correct, InDesign gives you more power than you suspected.

Figure 36 represents the inital state of our rectangular source object, in blue. Parent and spread spaces are pictured as well, assuming the parent object (a green oval) belongs to a spread. As usual we note M the affine map of the source object (i.e, the INNER-TO-PARENT matrix), P the PARENT-TO-SPREAD matrix, and S the SPREAD-TO-PASTEBOARD matrix.

As you can see M contains ROTATION attributes and P adds a bit of SCALING. As a result our rectangle looks slightly skewed in the perspective of the spread space. Indeed $M \times P$ (the INNER-TO-SPREAD matrix) puts end-to-end ROTATION and SCALING components, which typically introduces a SHEAR angle.

Now let's consider the transformation T we want to apply, e.g a 60% horizontal scaling specified by `[0.6,0,0,1,0,0]`. Before any calculation it is easy to imagine how changing the source object *from its inner space* would impact the shape in higher spaces:



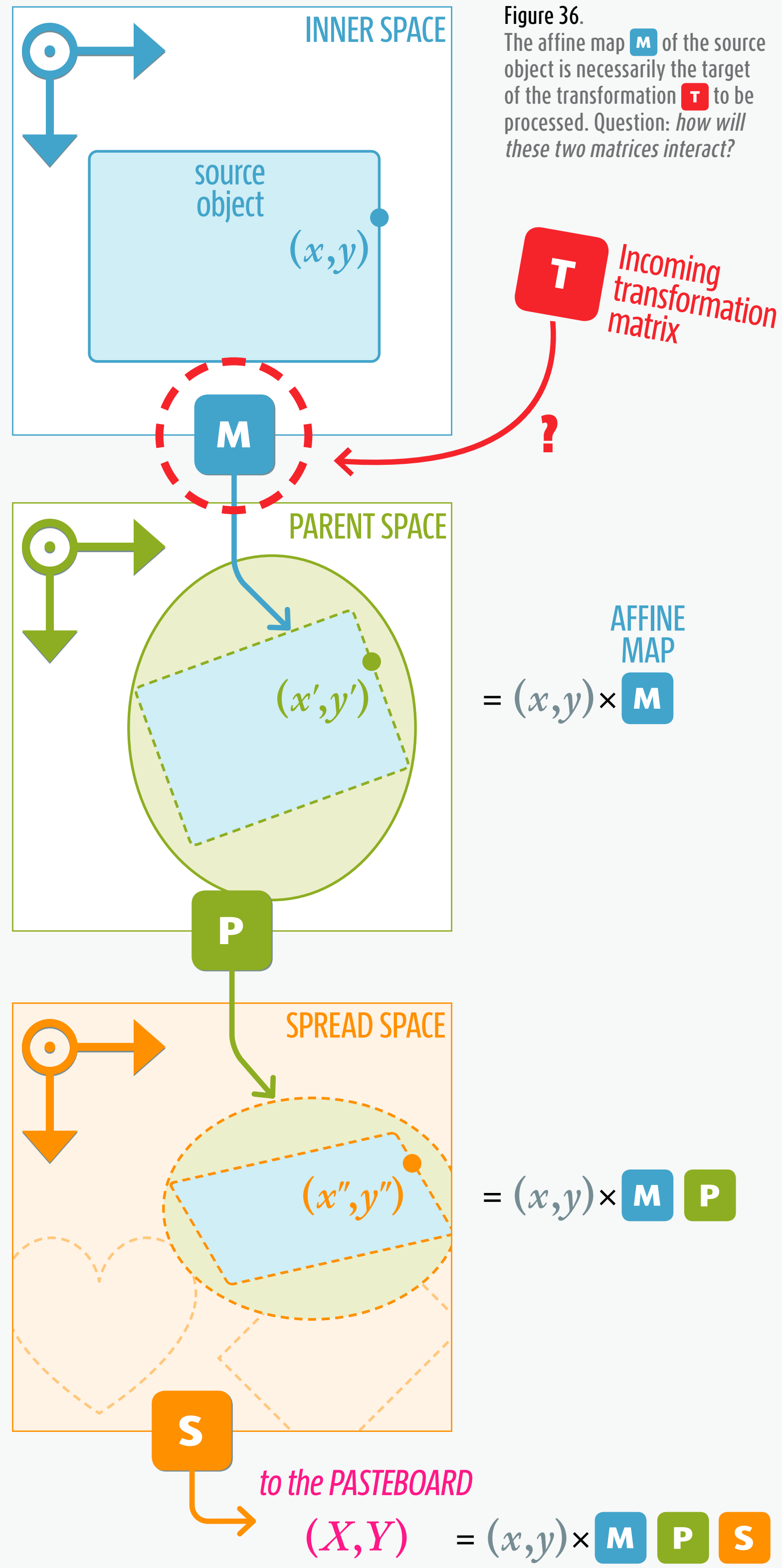


Figure 36. The affine map M of the source object is necessarily the target of the transformation T to be processed. Question: *how will these two matrices interact?*

This looks familiar to us because InDesign's GUI exactly reacts as just pictured when the user rescales or resizes an object. However, we know it technically wrong to represent the transformation *in* the inner space, since the inner space never transforms itself. So we rather use the expression "*from* the inner space," evoking that the matrix T has to be somehow sandwiched between the inner-to-inner matrix (the IDENTITY I) and the affine map M. Every simple transform stage works this way. Applying T *from the inner coordinate space* results in turning the affine map M into

$$M' = I \times T \times M,$$

which of course reduces to $T \times M$ since I has no effect. Hence, transforming any source *from its inner space* amounts to inserting T *before* M in terms of matrix product (see **Figure 37**), and we finally conclude that T undergoes[1] the transformation M. This paradoxical sounding result is better understood if we break down the process in two stages: T applies to I first, then the result goes through M to get the affine map updated.

Now you might prefer to transform the very same source object *from its parent space*, which then leads to:

$$M' = M \times T.$$

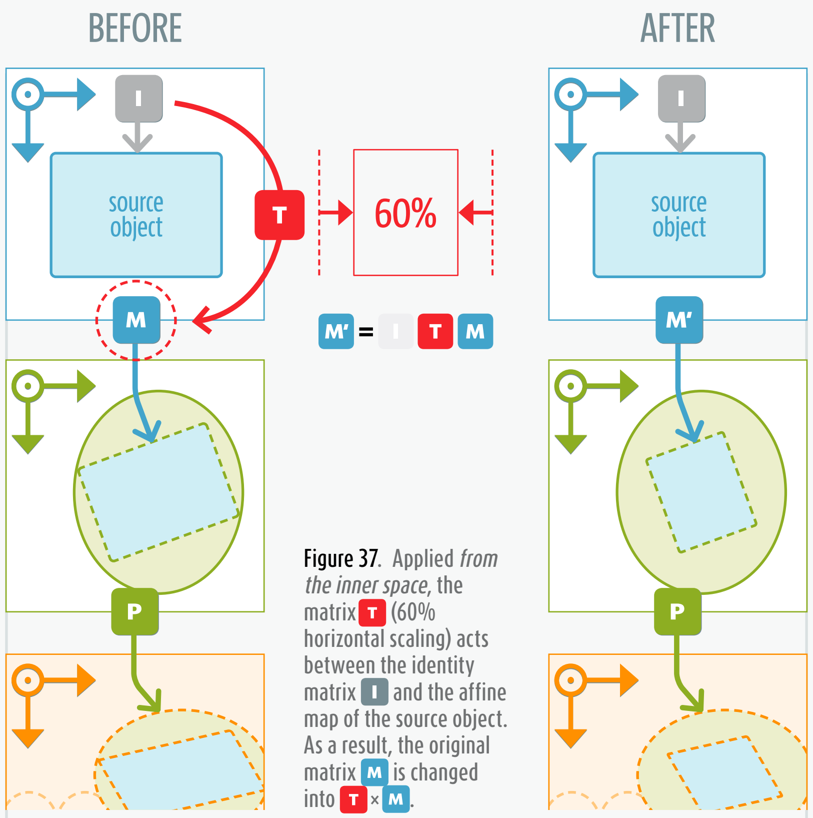


Figure 37. Applied *from the inner space*, the matrix T (60% horizontal scaling) acts between the identity matrix I and the affine map of the source object. As a result, the original matrix M is changed into T × M.

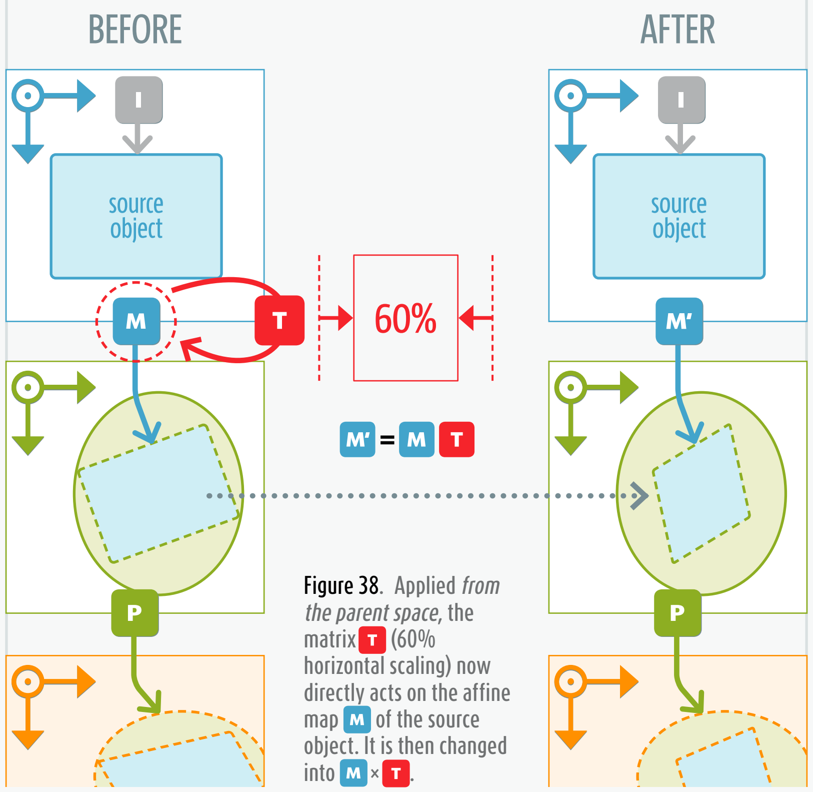
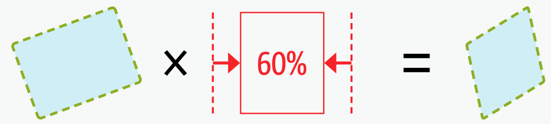


Figure 38. Applied *from the parent space*, the matrix T (60% horizontal scaling) now directly acts on the affine map M of the source object. It is then changed into M × T.

As shown in **Figure 38** this makes a huge difference. Indeed, the 60% horizontal scaling now seems to alter the shape *from the perspective of its parent space*. We could represent this as follows:



Note that the whole process leads anyway to changing the affine map M into $M \times T$, while the affine map of the parent (P) remains unchanged. So, transforming A from A's parent space is not the same as transforming the parent of A (say B) from B's inner space. Compare **Figure 39a** vs. **39b** to see the difference.

The rule is, whatever the parameters you send to `myObj.transform(...)`, the target matrix which actually changes is always the affine map M attached to `myObj`, the source object.

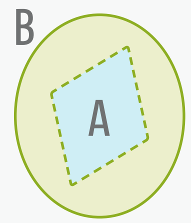


Figure 39a. Object A (the child) has undergone the transformation "from its parent space."

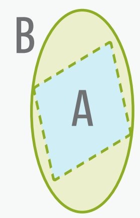


Figure 39b. Object B (the parent) has undergone the transformation "from its (own) inner space."

So far we have detailed two transform schemes:

➔ *From the inner space* result is $T \times M$.

➔ *From the parent space* result is $M \times T$.

But what is the general scheme? For example, what does it mean to apply T to `myObj` *from the spread space*? First we need to find the matrix that maps the inner geometry up to the spread space, that is, $M \times P$ in our example. T should operate at this point (*on* P), but we know that $M \times P \times T$ is not a valid result in terms of affine map, since it's not an INNER-TO-PARENT matrix. So we have to go back from the spread to the parent

1. As already discussed in Chapter 1 (page 4, note 1), the meaning of "applying A to B" is a matter of pure convention, provided that the author maintains a consistent paradigm throughout his presentation. In this document, "A applies to B" (or "B undergoes A") corresponds to the product $B \times A$ (the *applied* matrix being the right operand.) Since, in general, $B \times A \neq A \times B$, it is important to clearly distinguish "applying A to B" from "applying B to A."

space using the inverse matrix of P, noted P^{-1}. Finally,

$$M' = M \times P \times T \times P^{-1}.$$

The trick is, write the affine map in a form that brings up the target space, then apply T at this point:

- ➔ *Inner scheme:* $M = \underline{I} \times M$ -> $M' = \underline{I \times T} \times M = T \times M$
- ➔ *Parent scheme:* $M = \underline{M}$ -> $M' = \underline{M \times T}$
- ➔ *Spread scheme*[2]: $M = M \times \underline{P} \times P^{-1}$ -> $M' = M \times \underline{P \times T} \times P^{-1}$

etc.

This machinery, although a bit technical, provides a universal way to handle in mathematical terms the process behind the **`transform`** method. Now if you are not comfortable with matrix algebra, don't panic! You still have the option to get the picture from a more intuitive approach: just visualize the source object in the target space and imagine the transformation T as taking place *in that frame*. This allows you to easily *see* the result—although this does not detail *how* the affine map is actually changing.[3]

In conclusion, the meaning and the purpose of `myObj.transform(space,...,T)` is to apply T in the perspective of *space*, keeping in mind that the end result of this operation is updating `myObj`'s affine map accordingly. The unique object that `myObj.tranform(...)` truly alters is the INNER-TO-PARENT matrix.

2. Remembering that $P \times P^{-1} = I$ (the IDENTITY matrix.)

3. Luckily, we developers are rarely confronted with the numerical side of transformations. InDesign does the job for us and we can at any time retrieve the transform state of any source object relative to any coordinate space, using `myObj.transformValuesOf(space)[0]`.

Transformation Origin

According to Adobe's documentation the **`transform`** method expects three mandatory arguments (plus two optional arguments we shall investigate soon.)

➔ First, the *"coordinate space to use."* It would be better said, as just discussed, the *perspective* space of the transformation.

➔ Secondly, the *"temporary origin during the transformation,"* which happens to be a LOCATION specifier in the terms of the previous chapter.

➔ Then, the *"transform matrix"* itself (T), supplied as either a pure **`TransformationMatrix`** instance, or a simple **`Array`** of six numbers reflecting the matrix attributes.[4]

Here again, let's be candid about the *temporary origin.* Why are we supposed to provide such argument? *Can't any matrix deal with the origin of the perspective space?*

Technically, the answer is a disappointing "no." In itself *a transformation matrix has no origin*, it just encapsulates numbers that act on coordinates. The origin relative to which these coordinates make sense is not governed by the matrix, it intrinsically belongs to the coordinate space under consideration when the transformation is performed.

We are then facing an apparent limitation. What if I want to apply a 60% horizontal SCALING `[0.6,0,0,1,0,0]`, or some ROTATION, with respect to an arbitrary center Ω? Playing with the translation attributes (i.e, the two last numbers of the matrix) will not help, because TRANSLATION comes into action at the very end of the process.

4. Review Chapter 1 to resfresh your memory on this topic.

So, given a target space S, a transformation T and a location Ω which is not the origin of S, our goal is to make as if Ω were temporarily the origin of S while applying T. To solve this problem, let's pretend that T occurs in a virtual space S' defined as a pure copy of S centered in Ω. The coordinates of Ω in S, (x_Ω, y_Ω), become `(0,0)` in S'. In other words, the matrix that maps S to S' is $T_{-\Omega}$=`[1,0,0,1,`$-x_\Omega$`,`$-y_\Omega$`]` and, reciprocally, the matrix that maps S' back to S is $T_{+\Omega}$=`[1,0,0,1,`x_Ω`,`y_Ω`]`.

Now we can lucidly grasp the concept of *temporary origin*, and how it works (see **Fig. 40**.) Instead of just applying T from the perspective space S, InDesign

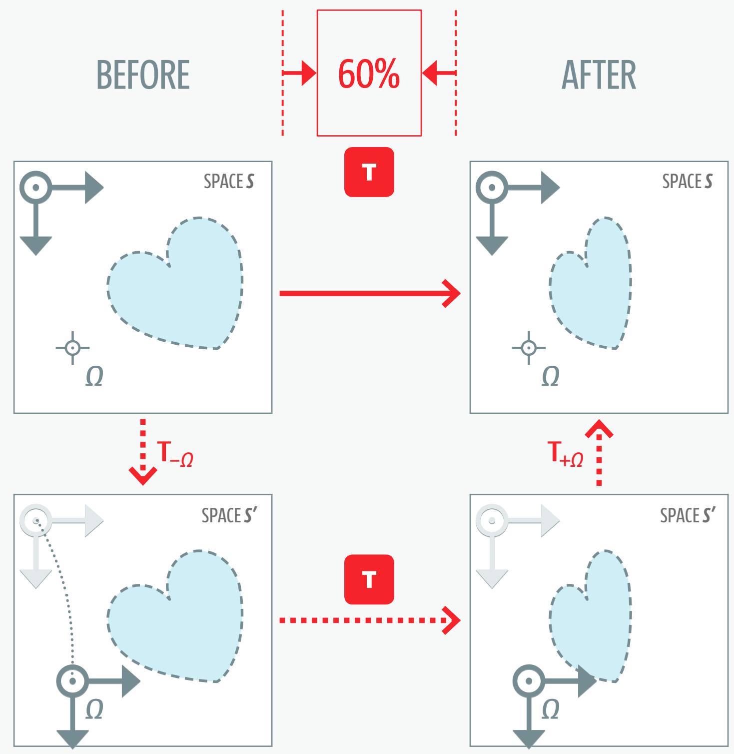


Figure 40. Applying a transformation **T** in the perspective of the space S and using Ω as temporary origin is equivalent to processing **T** in a virtual space S' centered on Ω. This involves two reciprocal translations $T_{-\Omega}$ and $T_{+\Omega}$. Finally, in the perspective of the space S, the actual matrix in use, although implied, is $T_{-\Omega} \times$ **T** $\times T_{+\Omega}$.

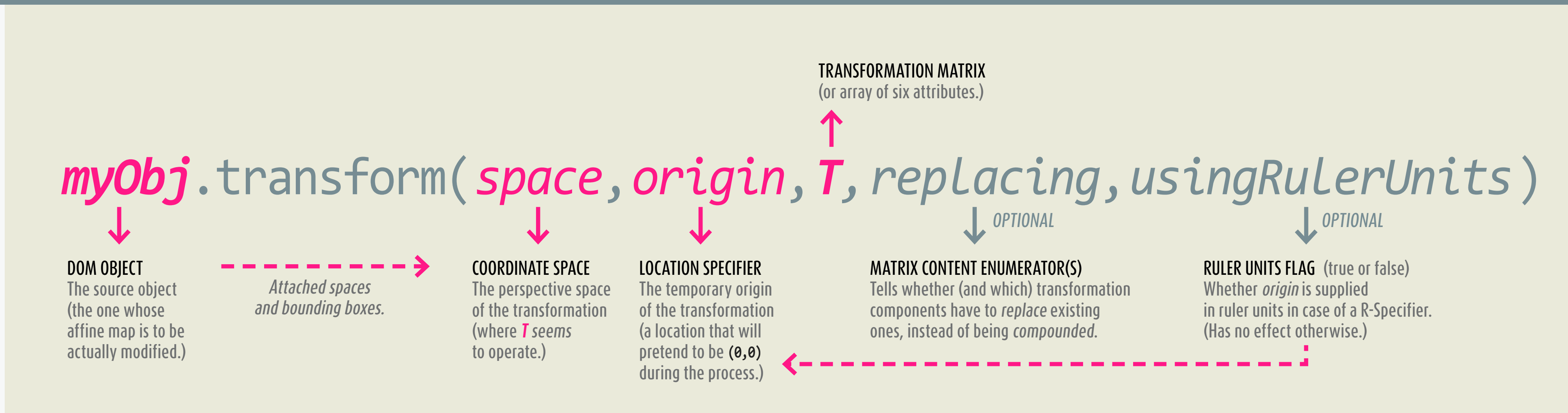


Figure 41. Summary of the parameters involved in the `transform` method.

applies in fact the transformation $\mathrm{T}_{-\Omega}\times T\times\mathrm{T}_{+\Omega}$ which makes Ω the apparent origin of the coordinate space *during* the transformation.

And this unstated back and forth translation is performed behind the scenes, thanks to the second argument of the **`transform`** method. It is both very convenient and very powerful. On one hand, it avoids explicitly computing and supplying the transitory matrices. Also, it offers the full syntax of any LOCATION specifier, so Ω can be expressed in bounds-space coordinates or in the ruler system, as well as in a regular coordinate space.

This also makes clear the last optional argument, ***`usingRulerUnits`***. It obviously refers to a *"ruler based origin"* (and has effect in such case only,) giving the option to interpret coordinates *"using ruler units rather than points."* (See Chapter 4 for further detail on ruler based locations.)

Matrix Content Flags

To avoid confusing the reader we have until now supported a basic assumption, that the transformation T is necessarily applied in terms of matrix product, meaning that the outgoing affine map, M', should always result from compounding existing matrices with T. The most general form we found to sum up the process is

$$M' = \underbrace{\text{INNER-TO-SPACE} \times T}_{\text{TRANSFORMATION}} \times \underbrace{\text{SPACE-TO-PARENT}}_{\text{REMAPPING}}$$

where *SPACE* refers to either the inner space itself (giving $M' = T\times M$), the parent space (giving $M' = M\times T$), or any available space in the hierarchy.[5]

Let's write T in its canonical form $S\times H\times R\times T$ where the submatrices represent, respectively, the SCALING, SHEAR, ROTATION, and TRANSLATION components of T.[6] In the same way the INNER-TO-*SPACE* matrix, which is the target of T, can be decomposed $\dot{S}\times\dot{H}\times\dot{R}\times\dot{T}$. So far we assumed that the transformation calculus was invariably consisting of processing:

$$\dot{S}\times\dot{H}\times\dot{R}\times\dot{T} \times S\times H\times R\times T.$$

5. We now know that these formulas are exact *modulo* the round-trip translations taking into account the temporary origin. But this point does not alter the formalization of the subject.

6. On the *canonical transformation order* in InDesign—S×H×R×T—see Chapter 1, page 7.

This amounts to *fully mixing* the components of the existing matrix with those of the transformation matrix. But the scripting DOM is more permissive than we thought! You can decide to simply *replace* some components—say $\dot{S}$ and $\dot{R}$—by those specified in the transformation matrix. Then we get

$$S\times\dot{H}\times R\times\dot{T},$$

$\dot{S}$ and $\dot{R}$ being purely abandoned, H and T being purely ignored. Regarding the transformation itself (before remapping the result to the parent space) SCALING and ROTATION components are now forcibly set to those specified in T, while SHEAR and TRANSLATION components do not undergo any impact from T.

The fourth parameter of the **`transform`** method (denoted *`replacing`* in **Fig. 41**) controls this special use.

➔ If *`replacing`* is missing, undefined, or an empty **`Array`**, the function behaves in default, full mix mode ($\dot{S}\times\dot{H}\times\dot{R}\times\dot{T}\times S\times H\times R\times T$.)

➔ If *`replacing`* has one or several **`MatrixContent`** enumerator(s)[7], it determines which component(s) from $S\times H\times R\times T$ will *replace* those in $\dot{S}\times\dot{H}\times\dot{R}\times\dot{T}$.

7. Namely, `MatrixContent.scaleValues` for S, `.shearValue` for H, `.rotationValue` for R, and `.translationValues` for T.

If used, the *`replacing`* argument can be either a single **`MatrixContent`** enum, or an **`Array`** of **`MatrixContent`** enums (in no particular order.) InDesign even gives you the option to redefine *all* matrix components by passing in the array

```
[
MatrixContent.scaleValues,
MatrixContent.shearValue,
MatrixContent.rotationValue,
MatrixContent.translationValues
].
```

All transformation attributes of the source object (in the perspective space) will then be reset to those supplied in *T*.[8]

As a concrete example, here is an explicit implementation of the **`clearTransformations`** method (which resets $\dot{S}$, $\dot{H}$, $\dot{R}$ attributes relative to the pasteboard space):

```
// 06. CLEAR TRANSFORMS (IN PASTEBOARD PERSP.)
var CS_PB = CoordinateSpaces.pasteboardCoordinates;
var ORIGIN = AnchorPoint.centerAnchor;
var T = [ 1, 0, 0, 1, 0, 0 ];
var MC = MatrixContent;
var MC_SHR = [ MC.scaleValues, MC.shearValue,
    MC.rotationValue ];
myObj.transform(CS_PB, ORIGIN, T, MC_SHR);
```

8. Note, however, that clearing the TRANSLATION component is tricky and rarely desired, since the (t_x, t_y) attributes remain somewhat arbitrary in InDesign spaces. Two objects may be at the same location in the layout while being translated differently in terms of affine map. It suffices that the positioning of internal path points (the inner geometry) compensates the translation effect. Thus, redefining (t_x, t_y) is generally a bad idea, and **`MatrixContent`**`.translationValues` is of little use in practice.

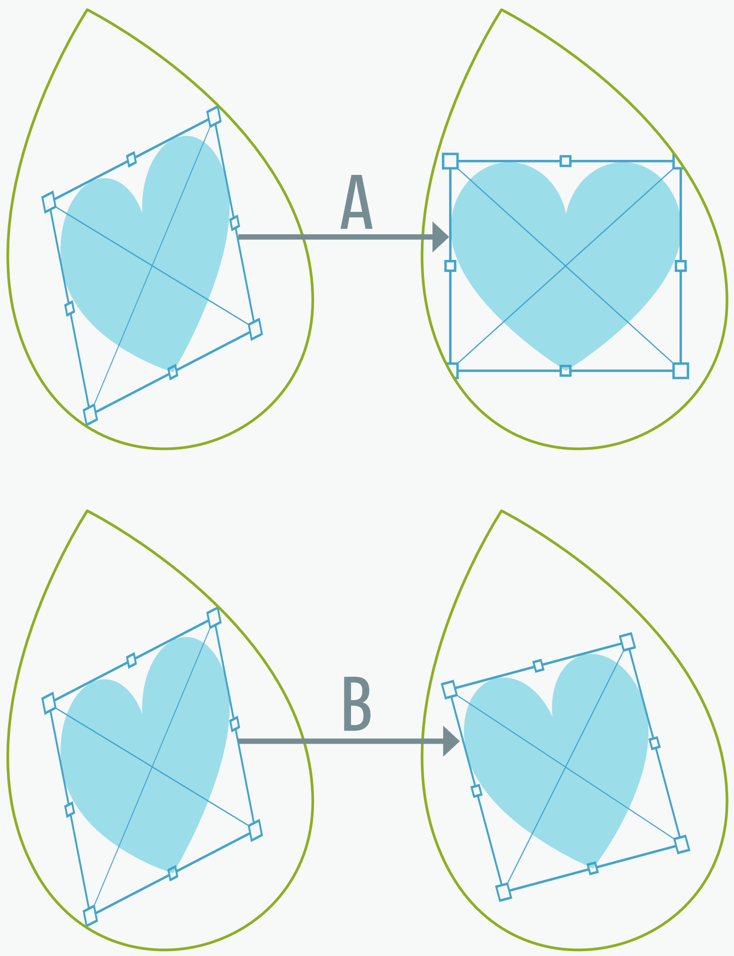


Figure 42.
A. Clearing the transformations of the child object (blue frame) relative to the pasteboard resets all matrix components—except translation—so that the shape looks 'untransformed' in the pasteboard space.

B. By contrast, clearing transformations relative to the parent (green shape) only resets matrix components in the perspective of the parent space. The child still undergoes the effects (scaling, rotation) that specifically affect the parent.

The above code is—obviously!—more complex than *`myObj`*`.clearTransformations()`, but it opens options not available in the built-in method.

For example, we could now clear transformations relative to the parent (not the pasteboard) space: just specify the perspective **`CoordinateSpaces`**`.parentCoordinates` rather than **`CoordinateSpaces`**`.spreadCoordinates`. The process will then readjust the INNER-TO-PARENT relationship while leaving higher level transformations untouched (PARENT-TO-PASTEBOARD.) See, in **Figure 42**, case A vs. case B.

Transform Preferences

The property **`app.transformPreferences`** controls a **`TransformPreference`** object of great importance when you are playing with transformations.

First and foremost, note that this preference set is **`Application`** scoped, which makes it persistent across InDesign sessions until the user, or a script, changes it. You cannot safely assign custom transform preferences to a specific **`Document`**: you need to check the state of affairs at the application level whenever you call **`transform()`** or similar methods.

By good fortune most **`TransformPreference`** members are harmless, for they only affect display behaviors. The boolean properties **`dimensionsIncludeStrokeWeight`**, **`showContentOffset`**, and **`transformationsAreTotals`** are of that kind. They just tell how Transformation and Control panels *expose* metric information (width and height, ruler coordinates, transformation attributes.)

Here is a short summary of the **TransformPreference** properties as documented by Adobe:

Property (Type)	Description
dimensionsIncludeStrokeWeight (Boolean)	If true, *"includes the stroke weight when displaying object dimensions."* If false, *"measures objects from the path or frame."*
showContentOffset (Boolean)	If true, *"measures the x and y values of the object relative to the containing frame."* If false, *"measures the x and y values relative to the rulers."*
transformationsAreTotals (Boolean)	If true, *"transform values are relative to the parent object."* If false, *"transform values are absolute values."*
whenScaling (**WhenScalingOptions**)	*"The method used to scale a page item."*
-> applyToContent	*"Apply scaling to the item's content."*
-> adjustScalingPercentage	*"Adjust the scaling percentage of the item's transform."*
adjustStrokeWeightWhenScaling (Boolean)	*"If true, strokes are scaled when objects are scaled."*
adjustEffectsWhenScaling (Boolean)	*(Available from InDesign CC.)* *"If true, transparency effects are scaled when objects are scaled."*

Thanks to the General Preferences dialog (**Figure 43**) we know that the boolean properties `adjustStrokeWeightWhenScaling` and `adjustEffectsWhenScaling`[9] only make sense if **whenScaling** is set to **WhenScalingOptions.applyToContent**.

9. Effect adjustment was not available before InDesign CC.

As you might guess, this special setting ("Apply to Content") has a deep impact on transformation matrix processing. In short, it tells InDesign to turn any SCALING *transformation* into a *deformation*.[10] That is, while the **transform** method operates, the source object is not scaled in transform matrix terms, it is purely *resized* (in terms of its inner geometry.)

For example—still assuming "Apply to Content" active—the code

```
myRectangle.transform(space,origin,
  [2,0,0,1,0,0])
```

will no longer apply a 200% horizontal scaling to the related matrix in the perspective space, it will actually change the underlying path points of *myRectangle* to get, visually, the same result. So, all happens as if a 200% scaling were applied, but the existing affine map remains unaltered.

In these circumstances, it may be desirable to adjust stroke weight and/or transparency effects accordingly, or to keep their original magnitude as it is.[11] This explains why the checkboxes "Include Stroke Weight" and "Include Effects" specifically regard the option "Apply to Content."

There is one exception to the rule: no matter the value of **TransformPreference.whenScaling**, **Spread** transformations never come out into deformations. Any scaled spread is and remains a scaled spread (that's a crucial difference between **Spread** and **Page** objects.) The following snippet proves our affirmation:

10. On the distinction between *transformation* and *deformation*, see Chapter 1, page 8.

11. Such options would be pointless in the "Adjust Scaling Percentage" case, because an *actual* transformation necessarily affects strokes and effects, as it does with everything in the scope of the coordinate space.

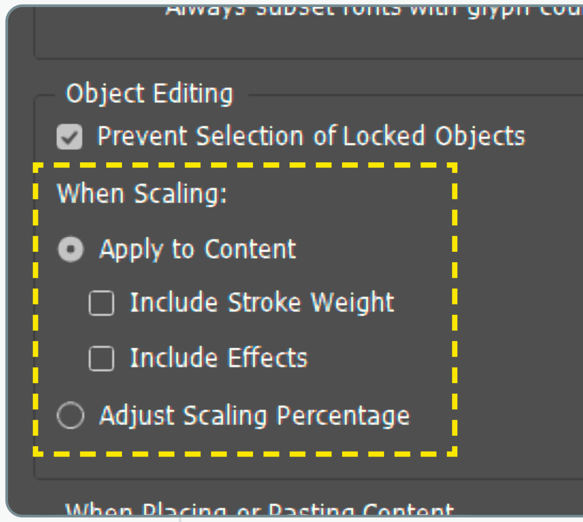


Figure 43.
InDesign users go into Preferences > General... > Object Editing to take control of the most important option regarding scaling: "Apply to Content" vs. "Adjust Scaling Percentage." The Scripting DOM reflects this option in `TransformPreference.whenScaling`.

```
// 07. SPREAD SCALING TEST
app.transformPreferences.whenScaling =
    WhenScalingOptions.applyToContent;
var mySpread = app.activeDocument.spreads[0];
// Rescale mySpread by (200%,50%)
mySpread.transform(CoordinateSpaces.
    pasteboardCoordinates, [0,0], [2,0,0,.5,0,0]);
alert(mySpread.transformValuesOf(CoordinateSpaces.
    pasteboardCoordinates)[0].matrixValues);
    // => 2, 0, 0, 0.5, 0, 0
```

SCALING factors are still visible in the final matrix despite the value assigned to **whenScaling**. Now if you run the same test on a **Page** object, the end matrix looks like `[1,0,0,1,tx,ty]`, meaning that during the deformation of the page bounds, the affine map did not change.

SUMMARY

The `transform()` method is a powerful tool for processing a transformation of any kind onto a source object. It expects three mandatory arguments:

➔ A COORDINATE SPACE, which specifies the *perspective* of the transformation, that is, the frame where it *appears* to occur. The actual, underlying process is anyway about changing the affine map of the source object (with respect to the perspective space.)

➔ A LOCATION specifier that defines the temporary origin of the transformation. Two reciprocal translations matrices are in fact involved (because a transformation matrix as such cannot specify a custom origin.)

➔ A `TransformationMatrix` object (or a set of six equivalent numbers) that defines all components of the transformation to be processed.

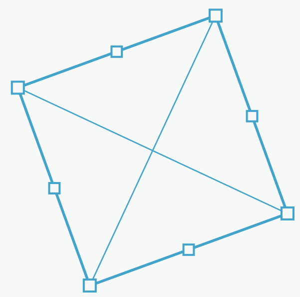
In addition, `transform()` supports a fourth, optional argument, based on `MatrixContent` enumerator(s). It allows to forcibly reset matrix data instead of compounding existing components with new ones.

A crucial preference, `app.transformPreferences.whenScaling`, may radically change the way SCALING operations are executed. If `WhenScalingOptions.applyToContent` is active, then any scaling transformation is turned into a *deformation* (resizing) unless the source object is a `Spread`.

EXERCISES

001. Noting that an affine map M can be rewritten $M \times P \times S \times S^{-1} \times P^{-1}$ (P denoting the PARENT-TO-SPREAD matrix, S denoting the SPREAD-TO-PASTEBOARD matrix), express the final affine map M' once a transformation T has been applied, *from the pasteboard perspective*, to the source object.

002. Let Q be an already rotated, `100%` scaled `Rectangle` living in a `Spread` (as in the figure below,) and T any ROTATION matrix.[12]



Consider the following code template:

```
Q.transform(<space>,AnchorPoint.centerAnchor,T)
```

Why does it produce the same result from whatever perspective *<space>* (either `innerCoordinates` or `parentCoordinates`)? Would you have observed the same outcome with a X-SCALING matrix? Why?

003. Many `Graphic` objects of a `Document` have been mistakenly skewed (by various angles) relative to their containers. All those shear effects should affect the parent frames instead! Write a script that fixes the problem, *with respect to other existing transform states.*

12. For example, `T=[cos20° -sin20° sin20° cos20° 0 0]`, but the rotation angle does not matter here.

004. A colleague asks you to evaluate a script that contains the following line:

```
myFrame.transform(
    CoordinateSpaces.pageCoordinates,
    [[100,20], AnchorPoint.centerAnchor],
    [1,0,-2,1, 0,0], undefined, true);
```

Explain the meaning and impact of each argument. Why is there good reason to suspect that *`myFrame`* will move?

005. Using `transform()` in both stages, divide by `2` the height of a `Page` (*deformation*), then apply a `200%` scaling factor along its vertical axis (*transformation*) so it finally looks exactly as it was at the beginning (in the GUI.) Show, however, that the final state is not identical to the original state.[13]

13. Interesting experiments can be done through PDF export.