Scalable Vector Graphics (SVG) Compositing Specification

W3C

This version:
Latest version:
Previous version:
Editors:
Anthony Grasso, Canon Information Systems Research Australia <anthony.grasso@cisra.canon.com.au>
Authors:
The authors of this specification are the participants of the W3C SVG Working Group.

Abstract

SVG is a language for describing vector graphics, but it is typically rendered to a display or some form of print medium. The SVG Compositing module adds support for the full range of Porter and Duff operators [PorterDuff] and blending modes. The module allows for raster and vector objects to be combined to produce eye catching effects.

This document defines the markup used by SVG Compositing for display and printing environments. It explains the technical background and gives guidelines on how to use the SVG Compositing specification with the SVG 1.1 Full and SVG 1.2 Tiny specifications and other SVG modules.

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. The latest status of this document series is maintained at the W3C.

This document is the of the SVG Compositing specification. It defines features of SVG specific to display and printing. It is a draft in progress; some descriptions in this document may be incomplete. This document shows the current thoughts of the SVG Working Group on the use of SVG for display and printing and should not yet be considered stable. This specification defines conformance criteria, new and reintroduced language features for SVG Compositing, and lists the ways SVG Compositing may be used for displaying and compositing.

This document has been produced by the W3C SVG Working Group as part of the W3C Graphics Activity within the Interaction Domain. The Working Group expects to advance this Editor's Draft to Recommendation Status.

We explicitly invite comments on this specification. Please send them to public-svg-compositing@w3.org (archives). For comments on the core SVG language, use www-svg@w3.org: the public email list for issues related to vector graphics on the Web(archives). Acceptance of the archiving policy is requested automatically upon first post to either list. To subscribe to these lists send an email to public-svg-compositing-request@w3.org or www-svg-request@w3.org with the word subscribe in the subject line.

Publication as a Editor's Draft does not imply endorsement by the W3C Membership. This is a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to cite this document as other than work in progress.

This document was produced by a group operating under the 5 February 2004 W3C Patent Policy. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

How to read this document and give feedback

The main purpose of this document is to encourage public feedback. The best way to give feedback is by sending an email to public-svg-compositing@w3.org. Please identify in the subject line of your message the part of the specificationto which your comment refers (e.g Compositing blend modes). If you have comments on multiple areas of this document, then it is preferable to send several separate comments.

The public are welcome to comment on any aspect in this document, but there are a few areas in which the SVG Working Group are explicitly requesting feedback. These areas are noted in place within this document with a class attribute value of 'feedback', which look **like this.**

Table of Contents

Introduction

Conformance statements in this document are marked as normative, and all equations in this document are normative. All other content is informative. This document contains explicit conformance criteria that overlap with some RNG definitions in requirements. If there is any conflict between the two, the explicit conformance criteria are the definitive reference.

By definition compositing is the process by which the colors of objects (and their surrounding regions) are combined together. In addition to its regular color values (such as red, green, and blue), an object may contain a channel to represent the transparency of the color. This channel is commonly known as an alpha channel and is often said to represent the 'opacity' of an object. In effect the opacity of an object controls how much of the object's color is used when compositing.

Compositing involving the alpha channel is referred to alpha compositing. By default SVG Full 1.1 [SVG11] and SVG Tiny 1.2 [SVGT12] both use Simple Alpha Compositing that gives a resultant effect of overlaying the object on to the background. If the overlayed object contains transparency, the color of the background may show through the overlayed object.

The SVG Compositing Module attempts to addresses compositing requirement for graphical features outlined in SVG 1.1/1.2/2.0 Requirements document [SVGReqs]. To achieve this requirement the SVG Compositing Module extends the Simple Alpha Compositing model in SVG Full 1.1 [SVG11] and SVG Tiny 1.2 [SVGT12]. This SVG module supports the following clipping/masking features:

- advanced alpha compositing, which may be used each time a new element is placed on the canvas. The operation specified determines the combination of the source color and alpha, and the destination color and alpha.
- clipping paths, which use any combination of 'path', 'text' and basic shapes to serve as the outline of a 1-bit mask (in the absence of anti-aliasing), where everything on the "inside" of the outline is allowed to show through but everything on the outside is masked out
- masks, which are container elements which can contain graphics elements or other container elements which define a set of graphics that is to be used as a semi-transparent mask for compositing foreground objects into the current background.

Masking with an element containing only color components with full luminance (e.g. r=g=b=1) will produce the equivalent result to compositing using the src-in or dst-in operators.

Terms Used in This Specification

This section in normative.

group alpha
: The group alpha is a single channel offscreen buffer that is typically created in-memory when a container element with nested graphical elements is encountered. The group alpha buffer is used to track percentage of the background in the group image buffer. When the group image is composited on to the canvas, the group alpha is used to ensure that the correct amount of the group image is present in the final result.

group image
: The group image is an offscreen buffer that is typically created in-memory when a container element with nested graphical elements is encountered. Initialisation of a group image buffer is controlled by the 'enable-background' property. Elements nested within the container element are rendered into the group image buffer. The group is then composited on to the canvas.

painted region
: The host language is reponsible for defining the painted region represented by each element.

  For SVG shapes and text, the painted region is the union of fill and stroke regions producing a resultant outline that represents the canvas area painted by the object regardless of any opacity values applied to the object. When calculating the painted region of an object the user agent must use all fill and stroke properties to determine the final painted region. The fill and stroke values of elements that make up the markers placed on shape must contribute to the painted region of an object.

  Examples:

  - If an object contains computed 'fill' value **#FFF**, then all painted pixels of the object contribute to the object's painted region even if it has a 'fill-opacity' of **0**.
  - If an object contains computed 'fill' value of **none** and a dashed stroke defined by 'stroke-dasharray', then its painted region will only be the pixels touched by the dashes in the computed stroke.
  - If a path contains a computed 'fill' and 'stroke' value of **none** but contains markers along it, then the painted region will only be the pixels touched by the computed 'fill' and 'stroke' values of the elements that make up the markers along the path.

  For SVG images and videos, if the computed value of the reference points to a valid resource, the painted region is the bounds of the object. Otherwise, the object has no painted region.

  For SVG filters, the painted region is the object's painted region that references the filter.

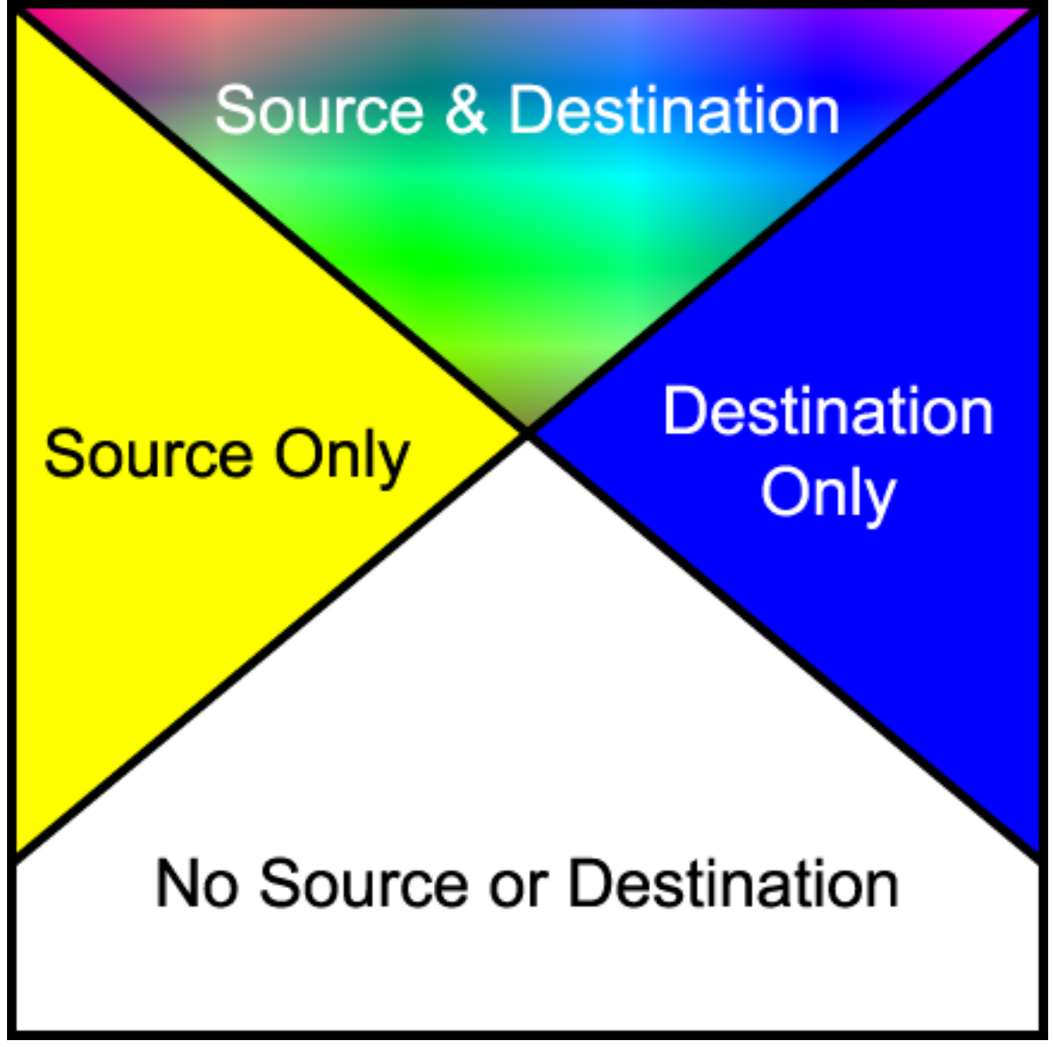
Alpha Compositing

As out lined in their paper Compositing Digital Images, Thomas Porter and Tom Duff defined algebra expression for compositing. These expressions resulted in what is known today as the twelve "Porter Duff" operators. The "Porter Duff" operators define the results of mixing the four sub-pixel regions formed by the overlapping of graphical objects that have an alpha channel value.

Graphics elements are composited onto the elements already rendered on the canvas based on an extended Porter-Duff compositing model, in which the resulting color and opacity at any given pixel on the canvas depend on the 'comp-op' specified. The base set of 12 Porter-Duff operations shown below always result in a value between zero and one, and as such, no clamping of output values is required.

In addition to the base set of 12 Porter-Duff operations, a number of blending operations are supported. These blending operations are extensions of the base Porter-Duff set and provide enhanced compositing behavior. The extended operations may result in color and opacity values outside the range zero to one. The opacity value should be clamped between zero and one inclusive, and the premultiplied color value should be clamped between zero and the opacity value inclusive.

The following diagram shows the four different regions of a single pixel that are considered when compositing.



Depending on the compositing operation the resultant pixel includes input from one or more of the regions in the above diagram. For the regions where only source or destination are present, a choice of including or not including the input is available. For the region where both are present, various options are available for the combination of input data.

For groups containing compositing operators, the operation used to composite the group onto the canvas is the 'comp-op' property of the container element itself. Other properties on container elements, such as 'opacity', specify operations that are performed after the children have been combined and before the group is composited onto the background. The 'enable-background' and 'knock-out' properties specify the state the group buffer is initialized to prior to use, any modification to the compositing of the group's children, and in some cases a post rendering step to be performed after rendering the children and prior to any other post rendering steps.

Implementation note: Various container elements calculate their bounds prior to rendering. For example, rendering a group generally requires an off-screen buffer, and the size of the buffer is determined by calculating the bounds of the objects contained within the group. SVG 1.0 implementations generally calculated the bounds of the group by calculating the union of the bounds of each of the objects within the group. Depending on the compositing operations used to combine objects within a group, the bounds of the group may be reduced, and so, reduce the memory requirements. For example, if a group contains two objects - object A 'in' object B - then the bounds of the group would be the intersection of the bounds of objects A and B as opposed to the union of their bounds.

While container elements are defined as requiring a buffer to be generated, it is often the case that a user agent using various optimizations can choose not to generate this buffer. For example, a group containing a single object could be directly rendered onto the background rather than into a buffer first.

The following variables are used to describe the components of the background, group and extra opacity channel buffers. This definition list is normative.

Sc
: Non-premultiplied source color component

Sca
: Premultiplied source color component

Sra Sga Sba
: Premultiplied source color component

Sa
: Source opacity component

Dc
: Non-premultiplied destination color component

Dca
: Premultiplied destination color component

Dra Dga Dba
: Premultiplied destination color component

Da
: Destination opacity component

Da(d)
: Group alpha buffer containing the percentage of the background channel in the group buffer.

D<n>
: Destination buffer <n> where the background is 0, groups in the top level 'svg' element 1, nested groups 2 and so forth

D'
: The results of the destination post a compositing step

The operation used to place objects onto the background is as follows:

```
Dca' = f(Sc, Dc) × Sa × Da  + Y × Sca × (1-Da)  + Z × Dca × (1-Sa)
Da'  =         X × Sa × Da  + Y × Sa × (1-Da)   + Z × Da × (1-Sa)
```

Depending on the compositing operation, the above equation is resolved into an equation in terms of premultiplied values prior to rendering. The following are specified for each compositing operation:

```
X, Y, Z, f(Sc, Dc)
```

defined as:

f(Sc,Dc)
: The intersection of the opacity of the source and destination multiplied by some function of the color. (Used for color)

X
: The intersection of the opacity of the source and destination. (Used for opacity)

Y
: The intersection of the source and the inverse of the destination.

Z
: The intersection of the inverse of the source and the destination.

Depending on the compositing operation, each of the above values may or may not be used in the generation of the destination pixel value.

Alpha Compositing Syntax

This section in normative.

When compositing using Porter-Duff extended blending operations color and opacity values may fall outside the range zero to one.

> A User Agent MUST clamp color and opacity values between zero and one inclusive.

> A User Agent MUST clamp premultiplied color values between zero and one inclusive.

Container Element Background Control

The 'clip-to-self' property

This section in normative.

The 'clip-to-self' property provides compatibility with Java2D by determining if the object effects pixels not covered by the object.

'clip-to-self'

Value:	object | canvas | inherit
Initial:	canvas
Applies to:	All elements that render. The host language is responsible for stating which elements render. For SVG: container elements and graphics elements.
Inherited:	no
Percentages:	N/A
Media:	visual
Animatable:	yes

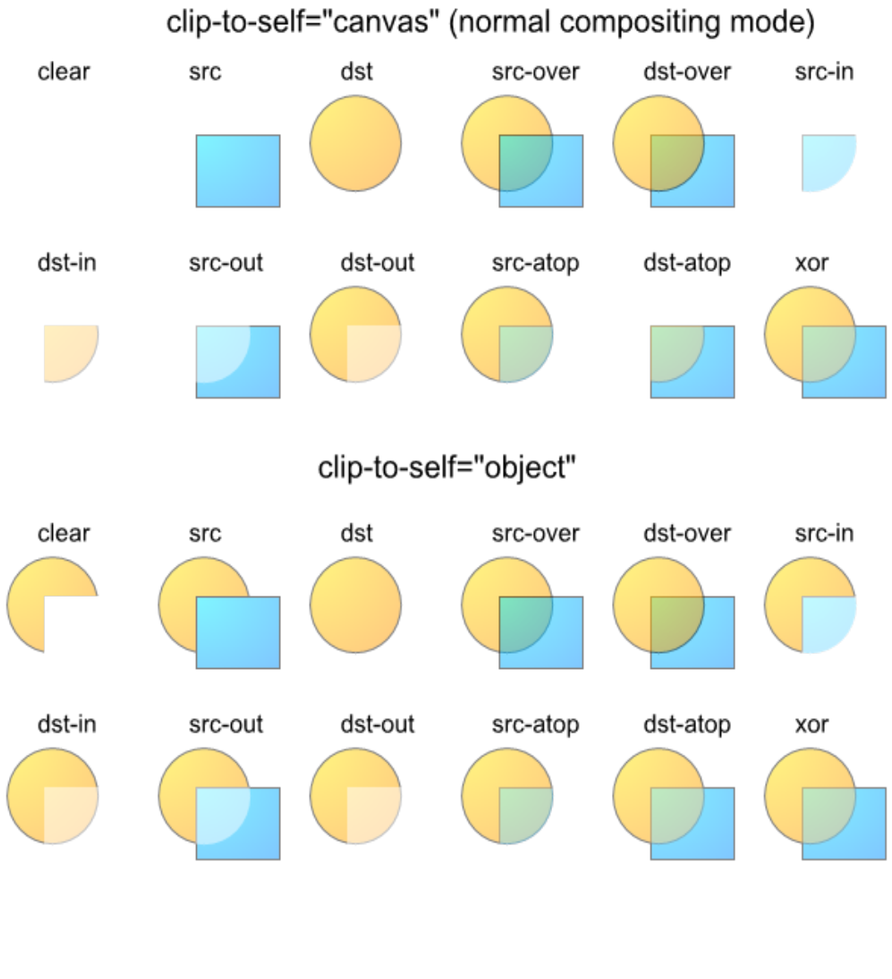
canvas
: Compositing an object effects all pixels on the canvas by compositing completely transparent source onto the destination for areas not covered by the object. This is the lacuna value.

object
: Compositing an object only effects the pixels covered by the object as by the object's painted region.

Graphics elements where the 'clip-to-self' property is set to **object** only effect the pixels within the extent of the element's painted region. A clipping path can be created from an element's painted region when performing a compositing operation where 'clip-to-self' is set to **object**. When the element is composited onto the canvas, it is composited through the generated clipping path and thus pixels outside of the extent of the element remain unmodified.

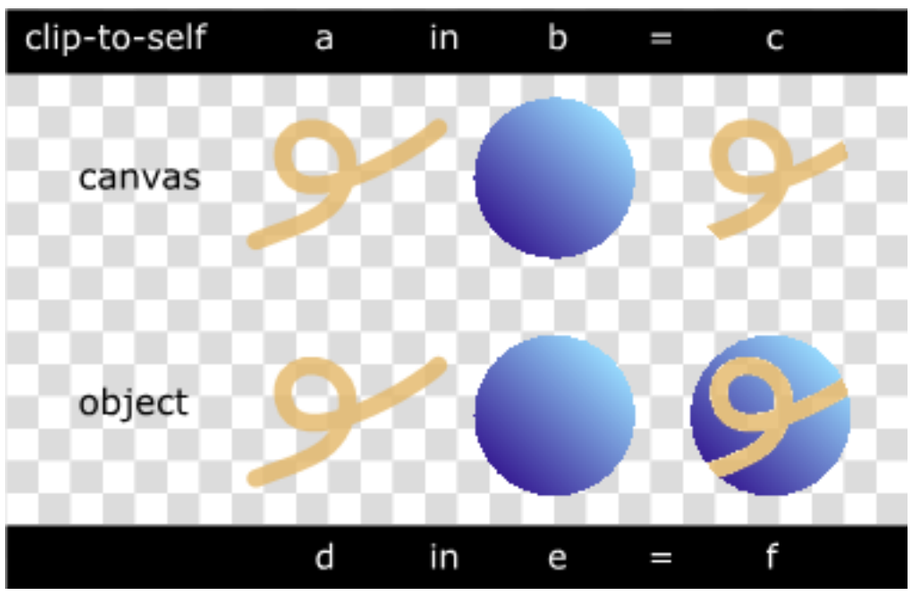
Container elements where the 'clip-to-self' property is set to **object** only effect the pixels within the extend of the container element. For example, if a container element contains two circles, and the container element has the 'clip-to-self' property is set to **object**, then region outside the circles is unaffected. To perform this operation, the user agent needs to keep track of the extent of each of the elements within the container element and ensure that only the elements are modified. This can be produced by creating a clipping path from each object's painted region and unioning the clipping paths together to produce a resultant clipping path that defines the extent of the pixels covered by all the elements within the container element. Where a container element contains nested container elements, the operation is performed within the sub-container elements to produce the resultant clipping path. When the container element is composited onto the canvas, it is composited through the resultant clipping path and thus pixels outside of the extent of the elements within the container remain unmodified.

> A User Agent MUST affect the pixel region as specified by the 'clip-to-self' property.



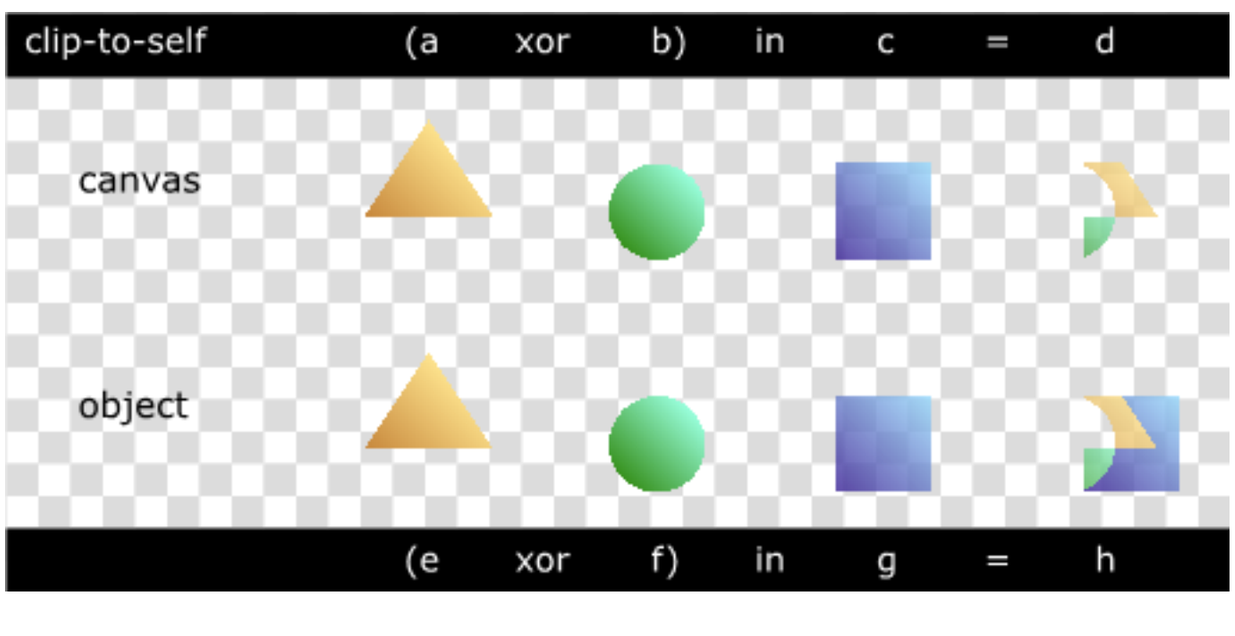
[View this image as SVG (SVG Compositing enabled browsers only)]

Most compositing operations do not remove the destination and as such for these operations, the 'clip-to-self' property has no effect. The compositing operations that remove the destination are **clear**, **src**, **src-in**, **dst-in**, **src-out** and **dst-atop**. These operations are illustrated in the compositing operation diagrams and are the operations that remove the right-hand blue region in diagram. For all other operations the 'clip-to-self' property has no effect.



[View this image as SVG (SVG Compositing enabled browsers only)]

For some container elements where the 'clip-to-self' property is set to **canvas**, the container element might effect the background outside bounds of the container element.



[View this image as SVG (SVG Compositing enabled browsers only)]

The 'enable-background' property

This section in normative.

The 'enable-background' property controls how the group image canvas for group compositing is initialized and managed.

'enable-background'

Value:	accumulate | new [<x> <y> <width> <height>] | inherit
Initial:	accumulate
Applies to:	Elements that contain elements that render. The host language is responsible for stating which elements contain elements that render. For SVG: container elements.
Inherited:	no
Percentages:	N/A
Media:	visual
Animatable:	no

new

A group image buffer is established which is initialized to transparent black. All children of the current element shall be rendered into the new group image.

The optional **<x>**, **<y>**, **<width>** and **<height>** parameters indicate in user space, the subregion of the container element where objects are composited onto. These parameters act as a clipping rectangle on the group image canvas enabling the SVG usage agent to potentially allocate a smaller temporary image buffer than the effective bounds of the container element. If not all the **<x>**, **<y>**, **<width>** and **<height>** values are specified, or if either **<width>** or **<height>** are specified as a value less than 1, then the objects are composited as if the 'enable-background' property was set to **accumulate**.

The optional values for the new property is under consideration

accumulate

A group image buffer is established which is initialized with corresponding area of the current canvas copied into it. Additionally, a group alpha buffer is established which is initialized to be opaque. The group alpha is used to store the percentage of background in the group image. All children of the current element shall be rendered into the group image. This is the lacuna value.

A User Agent MUST initialize the buffer of the container element as specified by 'enable-background' property.

A User Agent MUST render children with in a container element as if the 'enable-background' property was set to **accumulate** if not all the optional **new** parameters are specified when optional parameters are provided.

A User Agent MUST render children with in a container element as if the 'enable-background' property was set to **accumulate** if the **new** optional <width> or <height> parameters are less than 1.

A User Agent MUST apply the reduction to the additional background buffer caused by compositing an object in the group when 'enable-background' is set to accumulate.

For a container element with 'enable-background' set to **new**, the container element's group image buffer is initially cleared to transparent. This group image is treated as the canvas for the containers's children. When the complete contents of the container element are rendered onto the group image, the buffer is composited onto the canvas using the container element's specified compositing operation.

For a container element with 'enable-background' set to **accumulate**, the corresponding area of the canvas is copied into the container element's group image buffer. A group alpha buffer which has only an opacity channel is also created. This buffer Da(d) stores the percentage of the background in the group image and is initially opaque. The group image is treated as the canvas for the children of the group as usual. Additionally, as objects are placed into the group image, they are also placed into the Da(d) group alpha buffer using one of the operations listed below. When all the children of the container element have been composited in to the group image the following steps are performed to merge the group image with the canvas.

1. The canvas color is removed from the group image color.
2. The group alpha buffer is inverted to represent the amount of background that needs to be removed from the canvas.
3. Any post rendering effects such as group 'opacity' are applied to the group image and group alpha buffers respectively. If no post rendering effects are specified then this step can be ignored.
4. The group image is composited with the canvas background using the **src-over** operator and the group alpha data. The group alpha is used to control the the **src-over** operation such that the correct amount of alpha is removed from the canvas.

For container elements with an 'enable-background' set to **accumulate**, the compositing operation used to place the group image onto the background (canvas) is modified. The operation will apply any reduction to the background caused by the objects.

When drawing elements within a container element with 'enable-background' set to **accumulate**, the standard equations as listed below are used to draw the object into the group image. Depending on the compositing operation, one of two operations listed below are used to draw the object into the extra group alpha buffer Da(d).

For the operations **clear**, **src**, **src-in**, **dst-in**, **src-out** and **dst-atop**:

```
Da(d)' = 0
```

For all other compositing operations:

```
Da(d)' = Da(d) × (1 – Sa)
```

Once the contents of a container element are rendered into the container element's group image buffer and before operations such as 'opacity' or 'filter' effects are applied to the buffer, the remaining background (canvas) is removed from the container element's buffer using the following operations:

```
Dca1'   = Dca1 – Dca0 × Da1(d)
Da1'    = Da1 – Da0 × Da1(d)
```

At this point Da1(d) should be inverted. The inverted Da1(d) represents the amount of data to be removed from the background when placing the container element onto the background.

```
Da1(d)' = 1 – Da1(d)
```

The next operation to perform is the application of 'opacity' or 'filter' effects to the container element's buffer. During this step, the operation(s) performed on Da1 should also be performed on Da1(d).

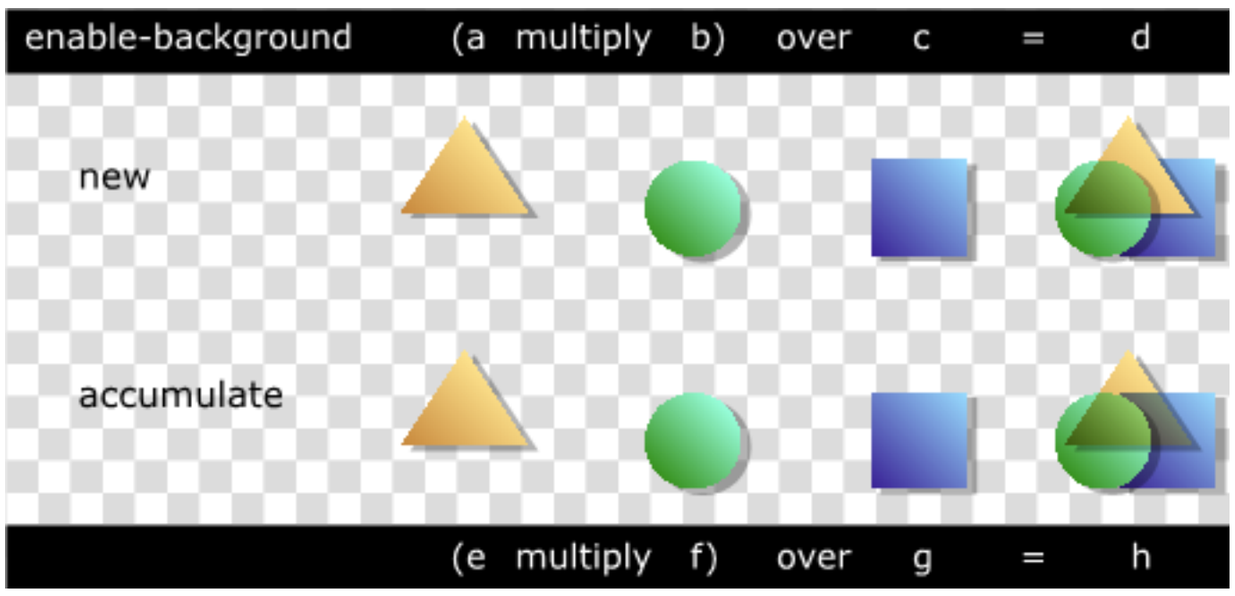
When compositing the container element's group image buffer onto the background, rather than the standard compositing operation listed above, the following operations should be used:

```
Dca0' = f(Dc1,Dc0) × Da1 × Da0 + Y × Dca1 × (1-Da0) + Z × Dca0 × (1-Da1(d))
Da0'  =          X × Da1 × Da0 + Y × Da1 × (1-Da0)  + Z × Da0 × (1-Da1(d))
```

The last term in the above equations uses the Da(d) buffer rather than Da.

Filters have access to the nearest ancestor group's buffer through the **BackgroundImage** and **BackgroundAlpha** images. The buffer created for the ancestor group element of the element referencing the filter, is passed to the filter. Where no ancestors of the element referencing the filter containing an 'enable-background' property value of **new**, transparent black is passed as input to the filter.

While container elements are defined as requiring a buffer to be generated, it is often the case that a user agent using various optimizations can choose not to generate this buffer. For example, a group containing a single object could be directly rendered onto the background rather than into a buffer first.



View this image as SVG (SVG Compositing enabled browsers only)

The 'knock-out' property

This section in normative.

The 'knock-out' property determines if the color and opacity of an object replaces the color and opacity of objects it overlaps in the container.

'knock-out'

Value:	replace I preserve I inherit
Initial:	preserve
Applies to:	Elements that contain elements that render. The host language is responsible for stating which elements contain elements that render. For SVG: container elements.
Inherited:	no
Percentages:	N/A
Media:	visual
Animatable:	no

replace
: The object color and opacity replaces that of other objects within the container element.

preserve
: The object color and opacity is overlayed normally as per the container compositing operation. This is the lacuna value.

For a complex group where the 'knock-out' property is set to **replace**, the buffer is created. The initial contents of the buffer and whether a secondary opacity channel is created depends on the value of the 'enable-background' property.

A User Agent MUST effect the color and opacity of the objects within the container element as specified by the 'knock-out' property.

For each object within the container element, the object color and opacity replaces that of other objects, rather than overlaying it. In effect, the destination input to the compositing operations for the complex group's children is the original contents of the buffer, rather than the current buffer for the complex group.

For **knock-out: preserve**:

```
Dca1' = f(Sca, Sa, Dca1, Da1)
Da1' = f(Sa, Da1)
```

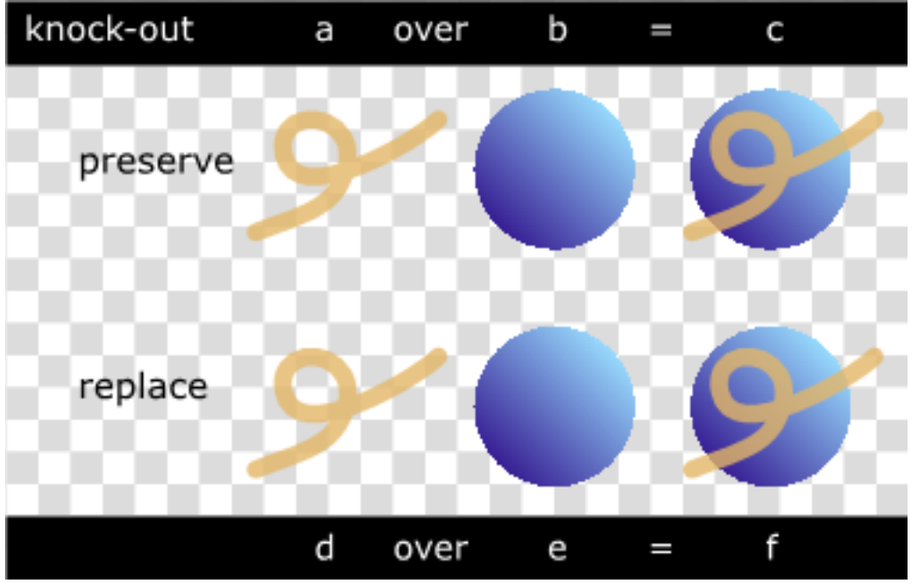
For **knock-out: replace** and **enable-background: new**:

```
Dca1' = f(Sca, Sa, 0, 0)
Da1' = f(Sa, 0)
```

For **knock-out: replace** and **enable-background: accumulate**:

```
Dca1' = f(Sca, Sa, Dca0, Da0)
Da1' = f(Sa, Da0)
```

An element in a knockout group that does not have the 'clip-to-self' property set, in effect clears all prior elements in the group.



View this image as SVG (SVG Compositing enabled browsers only)

Container Element Compositing Operators

The 'comp-op' property

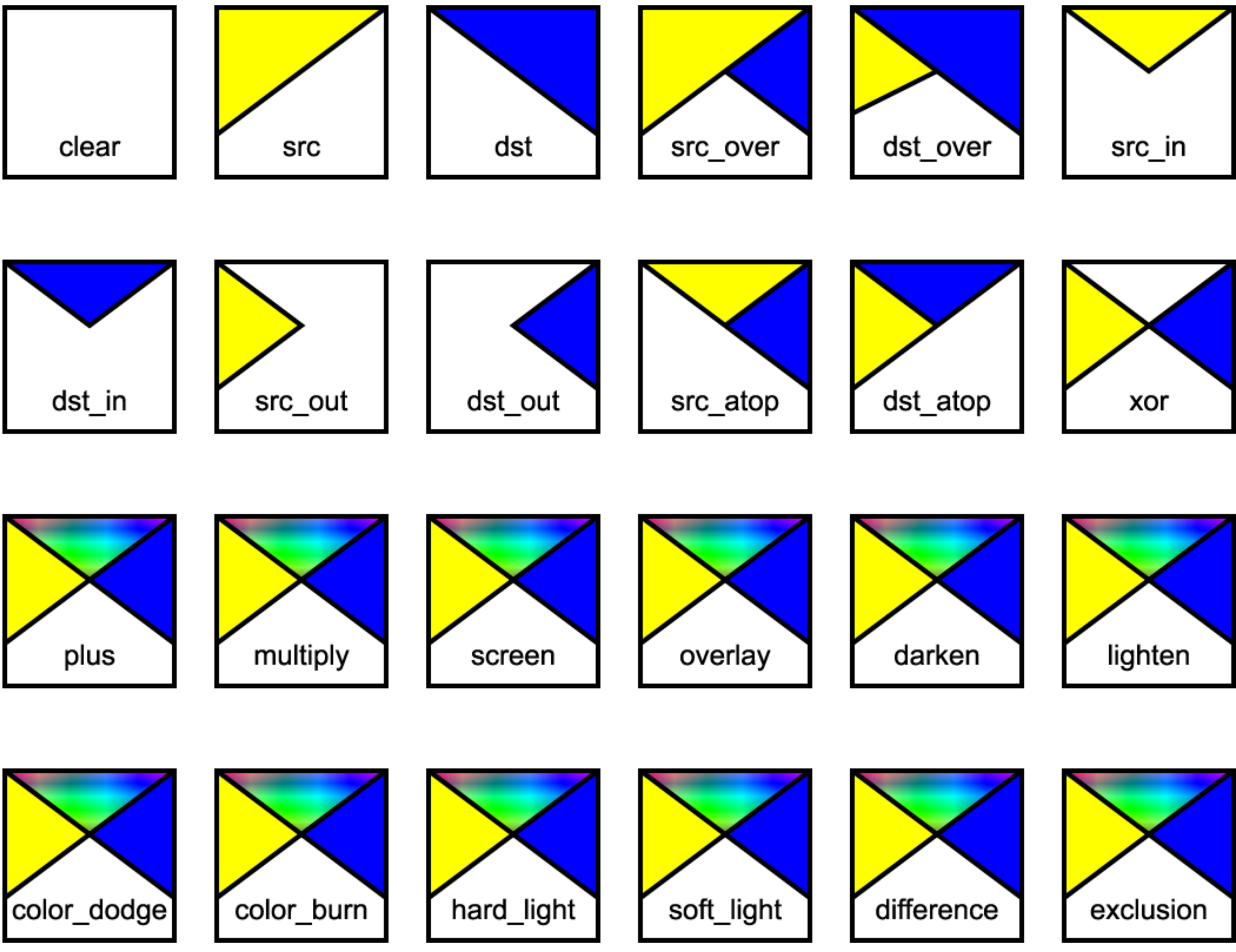
This section in normative.

The 'comp-op' property determines the compositing operation used when placing elements onto the canvas.

'comp-op'

Value:	clear | src | dst | src-over | dst-over | src-in | dst-in | src-out | dst-out | src-atop | dst-atop | xor | plus | multiply | screen | overlay | darken | lighten | color-dodge | color-burn | hard-light | soft-light | difference | exclusion | inherit
Initial:	src-over
Applies to:	All elements that render. The host language is responsible for stating which elements render. For SVG: container elements and graphics elements
Inherited:	no
Percentages:	N/A
Media:	visual
Animatable:	yes

The diagram below shows the sub-pixel regions output by each of the compositing operations.

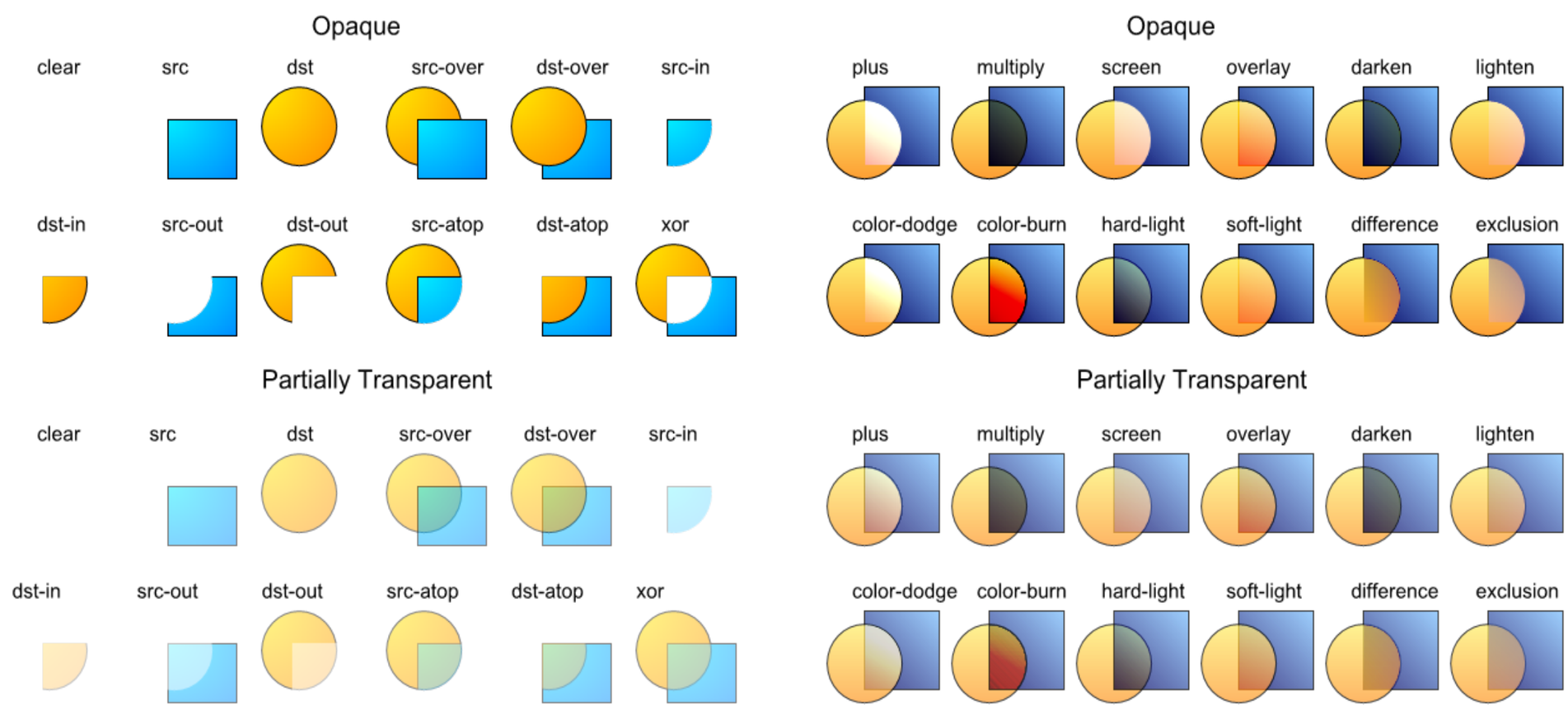


As discussed in the previous section, the bounds of the parent container element can be optimized to save in memory usage and hence, pixel writing requirements. Once the bounds of the parent container element have been determined, each element can only affect the pixels within those bounds.

A User Agent MUST effect the color and opacity of the objects within the container element as specified by the 'comp-op' property.

The following operators change pixels where the source is transparent: **clear**, **src**, **src-in**, **dst-in**, **src-out** and **dst-atop**.

The user agent may be required to create a backing store in which to generate a container element. The size of the backing store for a container element using the default compositing operator **src-over** is simply the union of the bounds of the sub-elements of the container element. When other compositing operators are used, the bounds of the container element are determined using the compositing operator diagram above. Starting with an empty bounds, the compositing operator specifies that the bounds of each successive object within the container element either replaces the result or is unioned with the result or is intersected with the result. For most compositing operators the bounds are unioned with the result. For the **clear** composite the current result is set to empty. For **src**, **src-out** and **dst-atop**, the bounds are set to the source bounds. For **dst**, **dst-out** and **src-atop**, the bounds are left unchanged. For **src-in** and **dst-in** the bounds are intersected with the result.



[View this image as SVG]

[View this image as SVG]

All color components listed below refer to color component information premultiplied by the corresponding alpha value. The following identifiers have the attached meaning in the equations following on from the identifiers.

```
Sc  - The source element color value.
Sa  - The source element alpha value.
Dc  - The canvas color value prior to compositing.
Da  - The canvas alpha value prior to compositing.
Dc' - The canvas color value post compositing.
Da' - The canvas alpha value post compositing.
```

The canvas contains color components and an optional alpha component. When placing new elements onto the canvas, the resulting pixel values on the canvas are calculated using the following equations.

clear

Both the color and the alpha of the destination are cleared. Neither the source nor the destination are used as input.

```
f(Sc,Dc) = 0
X        = 0
Y        = 0
Z        = 0

Dca' = 0
Da'  = 0
```

src

The source is copied to the destination. The destination is not used as input.

```
f(Sc,Dc) = Sc
X        = 1
Y        = 1
Z        = 0

Dca' = Sca × Da + Sca × (1 – Da)
     = Sca
Da'  = Sa × Da + Sa × (1 – Da)
     = Sa
```

dst

The destination is left untouched.

```
f(Sc,Dc) = Dc
X        = 1
```

```
Y         = 0
Z         = 1

Dca' = Dca × Sa + Dca × (1 - Sa)
     = Dca
Da'  = Da × Sa + Da × (1 - Sa)
     = Da
```

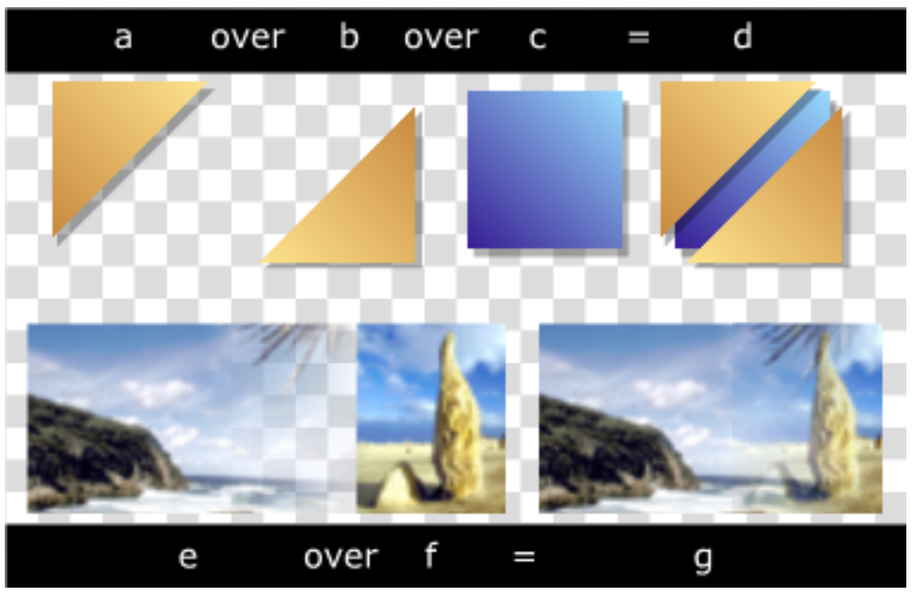
src-over

The source is composited over the destination. This is the lacuna value.

```
f(Sc,Dc) = Sc
X        = 1
Y        = 1
Z        = 1

Dca' = Sca × Da + Sca × (1 - Da) + Dca × (1 - Sa)
     = Sca + Dca × (1 - Sa)
Da'  = Sa × Da + Sa × (1 - Da) + Da × (1 - Sa)
     = Sa + Da - Sa × Da
```

The following diagram shows src-over compositing:



View this image as SVG (SVG Compositing enabled browsers only)

dst-over

The destination is composited over the source and the result replaces the destination.

```
f(Sc,Dc) = Dc
X        = 1
Y        = 1
Z        = 1

Dca' = Dca × Sa + Sca × (1 - Da) + Dca × (1 - Sa)
     = Dca + Sca × (1 - Da)
Da'  = Da × Sa + Sa × (1 - Da) + Da × (1 - Sa)
     = Sa + Da - Sa × Da
```

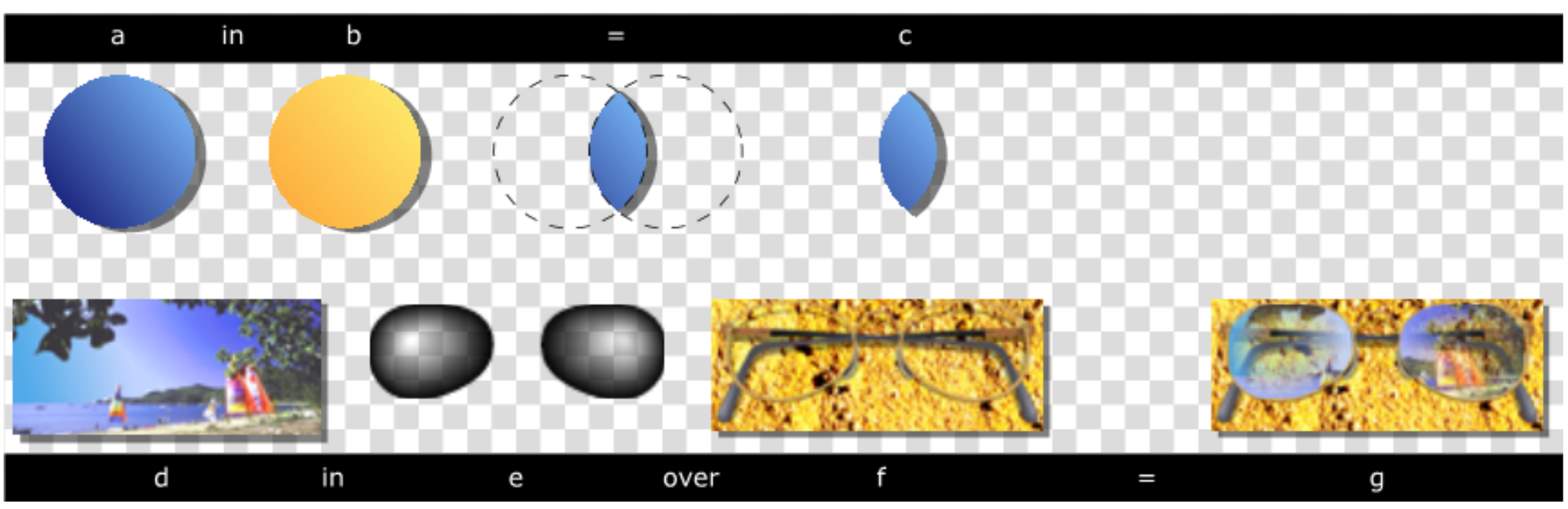
src-in

The part of the source lying inside of the destination replaces the destination.

```
f(Sc,Dc) = Sc
X        = 1
Y        = 0
Z        = 0

Dca' = Sca × Da
Da'  = Sa × Da
```

The following diagram shows src-in compositing:



View this image as SVG (SVG Compositing enabled browsers only)

dst-in

The part of the destination lying inside of the source replaces the destination.

```
f(Sc,Dc) = Dc
```

```
X         = 1
Y         = 0
Z         = 0

Dca' = Dca × Sa
Da'  = Sa × Da
```

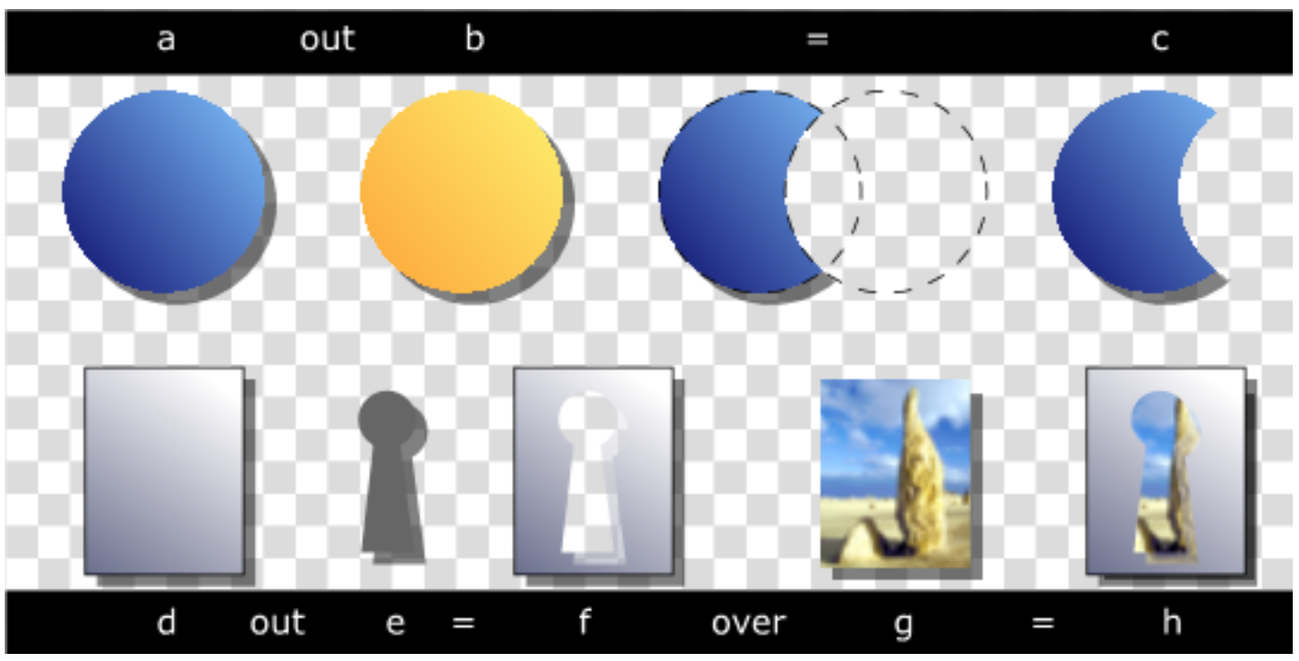
src-out

The part of the source lying outside of the destination replaces the destination.

```
f(Sc,Dc) = 0
X        = 0
Y        = 1
Z        = 0

Dca' = Sca × (1 - Da)
Da'  = Sa × (1 - Da)
```

The following diagram shows src-out compositing:



View this image as SVG (SVG Compositing enabled browsers only)

dst-out

The part of the destination lying outside of the source replaces the destination.

```
f(Sc,Dc) = 0
X        = 0
Y        = 0
Z        = 1

Dca' = Dca × (1 - Sa)
Da'  = Da × (1 - Sa)
```

src-atop

The part of the source lying inside of the destination is composited onto the destination.

```
f(Sc,Dc) = Sc
X        = 1
Y        = 0
Z        = 1

Dca' = Sca × Da + Dca × (1 - Sa)
Da'  = Sa × Da + Da × (1 - Sa)
     = Da
```

The following diagram shows src-atop compositing:

View this image as SVG (SVG Compositing enabled browsers only)

dst-atop

The part of the destination lying inside of the source is composited over the source and replaces the destination.

```
f(Sc,Dc) = Dc
X        = 1
Y        = 1
Z        = 0

Dca' = Dca × Sa + Sca × (1 - Da)
Da'  = Da × Sa + Sa × (1 - Da)
     = Sa
```

xor

The part of the source that lies outside of the destination is combined with the part of the destination that lies outside of the source.

```
f(Sc,Dc) = 0
X        = 0
Y        = 1
Z        = 1

Dca' = Sca × (1 – Da) + Dca × (1 – Sa)
Da'  = Sa × (1 – Da) + Da × (1 – Sa)
     = Sa + Da – 2 × Sa × Da
```

The following compositing operators add blending of source and destination colors beyond the base 12 Porter-Duff operations. The behavior of these operators necessitates clamping of the output values after compositing.

plus

The source is added to the destination and replaces the destination. This operator is useful for animating a dissolve between two images.

```
f(Sc,Dc) = Sc + Dc
X        = 1
Y        = 1
Z        = 1

Dca' = Sca × Da + Dca × Sa + Sca × (1 – Da) + Dca × (1 – Sa)
     = Sca + Dca
Da'  = Sa × Da + Da × Sa + Sa × (1 – Da) + Da × (1 – Sa)
     = Sa + Da
```

multiply

The source color is multiplied by the destination color and replaces the destination. The resultant color is always at least as dark as either the source or destination color. Multiplying any color with black results in black. Multiplying any color with white preserves the original color.

```
f(Sc,Dc) = Sc × Dc
X        = 1
Y        = 1
Z        = 1

Dca' = Sca × Dca + Sca × (1 – Da) + Dca × (1 – Sa)
Da'  = Sa × Da + Sa × (1 – Da) + Da × (1 – Sa)
     = Sa + Da – Sa × Da
```

The following diagram shows multiply compositing:

View this image as SVG (SVG Compositing enabled browsers only)

screen

The source and destination colors are complemented, multiplied and the resultant color replaces the destination. The resultant color is always at least as light as either the source or destination colour. Screening any color with white results in white. Screening any color with black preserves the original color.

```
f(Sc,Dc) = Sc + Dc – (Sc × Dc)

X        = 1
Y        = 1
Z        = 1

Dca' = (Sca × Da + Dca × Sa – Sca × Dca) + Sca × (1 – Da) + Dca × (1 – Sa)
     = Sca + Dca – Sca × Dca
Da'  = Sa + Da – Sa × Da
```

The following diagram shows screen compositing:

View this image as SVG (SVG Compositing enabled browsers only)

overlay

The destination color is used to determine if the resultant is either a multiplication or screening of the colors. Source colors overlay the destination whilst preserving its highlights and shadows. The destination color is mixed with the source color to reflect the destination lightness or darkness.

```
if 2 × Dc <= 1
  f(Sc,Dc) = 2 × Sc × Dc
otherwise
  f(Sc,Dc) = 1 - 2 × (1 - Dc) × (1 - Sc)
X        = 1
Y        = 1
Z        = 1

if 2 × Dca <= Da
  Dca' = 2 × Sca × Dca + Sca × (1 - Da) + Dca × (1 - Sa)
otherwise
  Dca' = Sa × Da - 2 × (Da - Dca) × (Sa - Sca) + Sca × (1 - Da) + Dca × (1 - Sa)
       = Sca × (1 + Da) + Dca × (1 + Sa) - 2 × Dca × Sca - Da × Sa

Da' = Sa + Da - Sa × Da
```

The following diagram shows overlay compositing:

View this image as SVG (SVG Compositing enabled browsers only)

darken

The resultant color is the darker of source or destination colors. If the source is darker, it replaces the destination. Otherwise, the destination is preserved.

```
f(Sc,Dc) = min(Sc,Dc)
X        = 1
Y        = 1
Z        = 1

Dca' = min(Sca × Da, Dca × Sa) + Sca × (1 - Da) + Dca × (1 - Sa)
Da'  = Sa + Da - Sa × Da

or

if Sca × Da < Dca × Sa
  src-over()
otherwise
  dst-over()
```

The following diagram shows darken compositing:

View this image as SVG (SVG Compositing enabled browsers only)

lighten

The resultant color is the lighter of source or destination colors. If the source is lighter, it replaces the destination. Otherwise, the destination is preserved.

```
f(Sc,Dc) = max(Sc,Dc)
X        = 1
Y        = 1
Z        = 1

Dca' = max(Sca × Da, Dca × Sa) + Sca × (1 - Da) + Dca × (1 - Sa)
Da'  = Sa + Da - Sa × Da

or

if Sca × Da > Dca × Sa
  src-over()
otherwise
  dst-over()
```

The following diagram shows lighten compositing:

View this image as SVG (SVG Compositing enabled browsers only)

color-dodge

The destination color is brightened to reflect the source color. Painting with black preserves the original color.

```
if Sc == 1
  f(Sc,Dc) = 1
otherwise
  f(Sc,Dc) = min(1, Dc/(1 - Sc))
X        = 1
Y        = 1
Z        = 1

if Sca == Sa and Dca == 0
  Dca' = Sca × (1 - Da) + Dca × (1 - Sa)
       = Sca × (1 - Da)
otherwise if Sca == Sa
  Dca' = Sa × Da + Sca × (1 - Da) + Dca × (1 - Sa)
otherwise if Sca < Sa
  Dca' = Sa × Da × min(1, Dca/Da × Sa/(Sa - Sca)) + Sca × (1 - Da) + Dca × (1 - Sa)

Da'  = Sa + Da - Sa × Da
```

The following diagram shows color-dodge compositing:

View this image as SVG (SVG Compositing enabled browsers only)

color-burn

The destination color is darkened to reflect the source color. Painting with white preserves the original color.

```
if Sc == 0
  f(Sc,Dc) = 0
otherwise
  f(Sc,Dc) = 1 - min(1, (1 - Dc)/Sc)
X        = 1
Y        = 1
Z        = 1

if Sca == 0 and Dca == Da
  Dca' = Sa × Da + Sca × (1 - Da) + Dca × (1 - Sa)
       = Sa × Da + Dca × (1 - Sa)
otherwise if Sca == 0
  Dca' = Sca × (1 - Da) + Dca × (1 - Sa)
       = Dca × (1 - Sa)
otherwise if Sca > 0
  Dca' = Sa × Da - Sa × Da × min(1, (1 - Dca/Da) × Sa/Sca) + Sca × (1 - Da) + Dca × (1 - Sa)
       = Sa × Da × (1 - min(1, (1 - Dca/Da) × Sa/Sca)) + Sca × (1 - Da) + Dca × (1 - Sa)

Da'  = Sa + Da - Sa × Da
```

The following diagram shows color-burn compositing:

View this image as SVG (SVG Compositing enabled browsers only)

hard-light

The source color is used to determine if the resultant is either a multiplication or screening of the colors. If the source color is lighter than 0.5, the destination is lightened as if it were screened. If the source color is darker than 0.5, the destination is darkened, as if it were multiplied. The degree of lightening or darkening is proportional to the difference between the source color and 0.5. If it is equal to 0.5 the destination is unchanged. Painting with pure black or white produces black or white.

```
if 2 × Sc <= 1
  f(Sc,Dc) = 2 × Sc × Dc
otherwise
  f(Sc,Dc) = 1 - 2 × (1 - Dc) × (1 - Sc)
X        = 1
Y        = 1
Z        = 1


if 2 × Sca <= Sa
  Dca' = 2 × Sca × Dca + Sca × (1 - Da) + Dca × (1 - Sa)
otherwise
  Dca' = Sa × Da - 2 × (Da - Dca) × (Sa - Sca) + Sca × (1 - Da) + Dca × (1 - Sa)
       = Sca × (1 + Da) + Dca × (1 + Sa) - Sa × Da - 2 × Sca × Dca

Da'  = Sa + Da - Sa × Da
```

The following diagram shows hard-light compositing:

View this image as SVG (SVG Compositing enabled browsers only)

soft-light

The source colour is used to determine if the resultant color is darkened or lightened. If the source color is lighter than 0.5, the destination is lightened. If the source color is darker than 0.5, the destination is darkened, as if it were burned in. The degree of darkening or lightening is proportional to the difference between the source color and 0.5. If it is equal to 0.5, the destination is unchanged. Painting with pure black or white produces a distinctly darker or lighter area, but does not result in pure black or white.

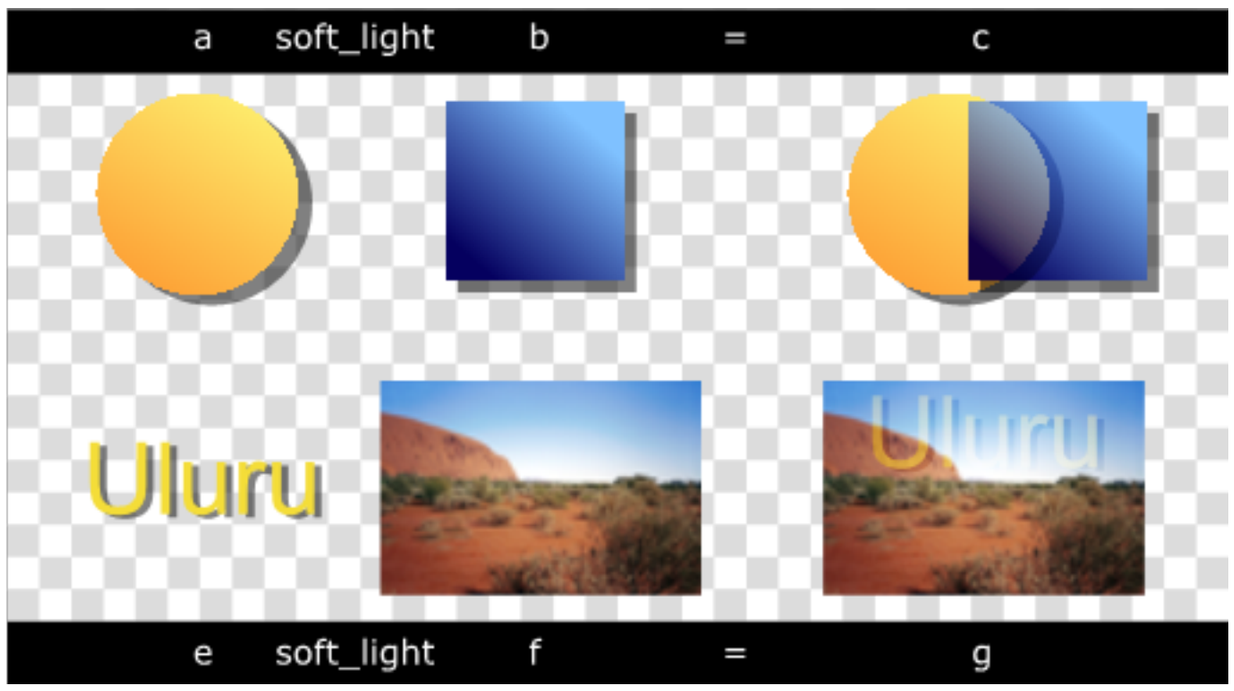
```
if 2 × Sc <= 1
  f(Sc,Dc) = Dc - (1 - 2 × Sc) × Dc × (1 - Dc)
otherwise if 2 × Sc > 1 and 4 × Dc <= 1
  f(Sc,Dc) = Dc + (2 × Sc - 1) × (4 × Dc × (4 × Dc + 1) × (Dc - 1) + 7 × Dc)
otherwise if 2 × Sc > 1 and 4 × Dc > 1
  f(Sc,Dc) = Dc + (2 × Sc - 1) × ((Dc)^0.5 - Dc)
X        = 1
Y        = 1
Z        = 1

if 2 × Sca <= Sa
  Dca' = Dca × (Sa + (2 × Sca - Sa) × (1 - m)) + Sca × (1 - Da) + Dca × (1 - Sa)
otherwise if 2 × Sca > Sa and 4 × Dca <= Da
  Dca' = Dca × Sa + Da × (2 × Sca - Sa) × (4 × m × (4 × m + 1) × (m - 1) + 7 × m) + Sca × (1 - Da) + Dca × (1 - Sa)
       = Da × (2 × Sca - Sa) × (16 × m^3 - 12 × m^2 - 3 × m) + Sca - Sca × Da + Dca
otherwise if 2 × Sca > Sa and 4 × Dca > Da
  Dca' = Dca × Sa + Da × (2 × Sca - Sa) × (m^0.5 - m) + Sca × (1 - Da) + Dca × (1 - Sa)
       = Da × (2 × Sca - Sa) × (m^0.5 - m) + Sca - Sca × Da + Dca

Da'  = Sa + Da - Sa × Da

Where:
  m = Dca/Da
```

The following diagram shows soft-light compositing:



View this image as SVG (SVG Compositing enabled browsers only)

difference

The resultant color is the absolute difference between the source and destination colors. The destination color is inverted when white is used. The destination color is preserved when black is used.

```
f(Sc,Dc) = abs(Dc - Sc)
X        = 1
Y        = 1
Z        = 1

Dca' = abs(Dca × Sa - Sca × Da) + Sca × (1 - Da) + Dca × (1 - Sa)
     = Sca + Dca - 2 × min(Sca × Da, Dca × Sa)
Da'  = Sa + Da - Sa × Da
```

The following diagram shows difference compositing:

View this image as SVG (SVG Compositing enabled browsers only)

exclusion

The resultant color is similar to that of the **difference** operation. However, the **exclusion** resultant color appears as a lower contrast than that of the **difference** resultant color. The destination color is inverted when white is used. The destination color is preserved when black is used.

```
f(Sc,Dc) = Sc + Dc - 2 × Sc × Dc
X        = 1
Y        = 1
Z        = 1

Dca' = (Sca × Da + Dca × Sa - 2 × Sca × Dca) + Sca × (1 - Da) + Dca × (1 - Sa)
Da'  = Sa + Da - Sa × Da
```

These equations are approximations which are under review. Final equations may differ from those presented here.

The following diagram shows exclusion compositing:

View this image as SVG (SVG Compositing enabled browsers only)

For many of the operators listed above, the destination is modified in regions of the image where the source is completely transparent. Pixels that the source does not touch are considered transparent, and as such may be modified, depending on the compositing operation.

References

Normative References

[SVG11]
Scalable Vector Graphics (SVG) 1.1 (Second Edition) Specification, Erik Dahlström, Jon Ferraiolo, 藤沢 淳 (FUJISAWA Jun), Anthony Grasso, Dean Jackson, Chris Lilley, Cameron McCormack, Doug Schepers, Jonathan Watt, Patrick Dengler editors, W3C, 22 June 2010 (Working Draft). See http://www.w3.org/TR/2010/WD-SVG11-20100622/

[SVGT12]
Scalable Vector Graphics (SVG) Tiny 1.2 Specification, Ola Andersson, Robin Berjon, Erik Dahlström, Andrew Emmons, Jon Ferraiolo, Anthony Grasso, Vincent Hardy, Scott Hayman, Dean Jackson, Chris Lilley, Cameron McCormack, Andreas Neumann, Craig Northway, Antoine Quint, Nandini Ramani, Doug Schepers, Andrew Shellshear editors, W3C, 22 December 2008 (Recommendation). See http://www.w3.org/TR/2008/REC-SVGTiny12-20081222/

Informative References

[PorterDuff]
Compositing Digitial Images, Thomas Porter and Tom Duff, Computer Graphics Volume 18, Number 3, July 1984.

[SVGReqs]
SVG 1.1/1.2/2.0 Requirements, Dean Jackson editor, W3C, 22 April 2002 (Working Draft). See http://www.w3.org/TR/2002/WD-SVG2Reqs-20020422/

Author List

The authors of this specification are the participants of the W3C SVG Working Group.

- Anthony Grasso, Canon Information Systems Research Australia
- Doug Schepers, W3C