# 2. Overview of Blob Behavior

The principal use of the Blob Manager is to represent and process situations (or *scenarios*) involving associative transactions between sets of objects. As an example, suppose a simple interrogative scenario is to be presented, such as whether a user knows the capital cities of a set of states. The names of the states and of the cities are easily displayed, but how should associations between them be represented? The user could be made to type in names — but that's contrary to the general spirit of Macintosh usage. Lines might be drawn between states and cities — but that can easily result in a messy display, especially if the elements of the sets are too numerous to be displayed in single rows or columns.

Alternatively, the user might click on the city name and drag it onto the state name, so that the association would be represented by proximity. It would not be desirable for the state name to be covered up by the city name, though, since that results in a loss of display information. Instead, a region close to the state name could be used as an area onto which to drag the city name.

The click-and-drag approach is the one implemented by the Blob Manager. It enjoys the following characteristics:

- **Highly graphic.**
- **Consistent with Macintosh usage. Clicking and dragging are part of the canon of standard Macintosh operations.**
- **Clean display. City names can be dragged to the appropriate states and be displayed in conjunction with them in such a way that the associations are clear, and such that the state names are not obscured.**
- **The integrity of the display is not compromised by extraneous objects (such as extra lines between objects).**
- **The user is freed from the keyboard: all actions are entirely mouse-based.**

The Blob Manager allows the user to change his mind: he can drag the city name from one state to another, or clear it from a state altogether. City names can be swapped by dragging a city that's associated with one state onto another state associated with a city. The Blob Manager can provide feedback by assessing which associations are correct and indicating that visually to the user.

This simple scenario illustrates one way of characterizing the Blob Manager, i.e., it provides a graphic representation of interrogative situations — questions and answers — together with mechanisms for association formation. The method it provides is intuitive, easy to understand, and consistent with a number of Macintosh applications that involve object movement in some form or another.

## 2.1 Blob Anatomy

Scenarios such at the one described above (and others) are implemented by creation and manipulation of special objects known as blobs. Blobs come in sets: "no blob is an island, sufficient unto itself." The city-state scenario uses two sets, one for state name blobs and another for city name blobs. In this scenario, state blobs function as *receptor* blobs and city blobs function as *donor* blobs.

The overall area that a blob occupies is its *general region*. This region is divided into subregions. One part is the *drag region*, which is the part of the blob that can be dragged onto other blobs (if it's a donor blob) or which can have other blobs dragged onto it (if it's a receptor blob). The part of the blob that is not in the drag region is denoted as the *static region*. Thus the static and drag regions normally are mutually exclusive and form a partition of the general blob region. The blob illustrated in Figure 2–1 consists of a rectangulat drag region and a static region surrounding the drag region.

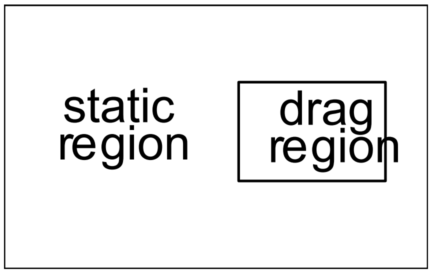

**Figure 2–1. Blob Regions**

Some possible relationships between the regions of a blob include the following:

- **The general blob region and its drag region are equal. In this case the static region is empty. This relationship is used when no additional information is needed except that contained in the drag region, which is very common for donor blobs.**

- **The general blob region and its static region are equal. In this case the drag region is empty (not particularly useful or interesting).**

- **The drag region does not need to be contained entirely or even partially within the static region.**

The static region serves as a display-only region that remains unaffected by user actions. It provides a means for display of information associated with a blob that is not obscured when a donor is dragged onto it.

In contrast, the drag region is under user control. When the mouse is clicked in the drag region of a donor blob and moved around, the Blob Manager responds by tracking it: an outline of the region follows the mouse until the button is released. For a receptor blob, the drag region is that region onto which a donor blob's drag region may be dragged — a kind of landing pad.

The appearance of a blob is determined by what is drawn within the general region — the *general image*. Like the general region, the general image is composed of two parts, the *drag image* and the *static image*. In most respects, a region and its corresponding image may be referred to synomymously.

Blob regions can be defined with the full generality of QuickDraw regions.

## 2.2 Blob Transactions

The Blob Manager handles several types of *transactions*, which the host performs by calling the built-in transaction handling routine `BlobClick()`. Permission for most transaction types may be turned on or off by the host.

Two transaction types take place between donors and receptors. A donor blob may be dragged onto a receptor to become *glued* to it. The donor is then known as the receptor's *glue blob* (or *glob* for short). A glob may also be *unglued* from the receptor with which it is associated by double-clicking it, or by dragging it back to the donor it came from. The donor/receptor association is represented graphically by displaying a copy of the donor blob's drag image in the receptor blob's drag region. The donor drag image is scaled to fit the receptor drag region.

Glob gluing and ungluing may also be done by the host program explicitly, with or without visual feedback to indicate what's going on.

**Note**
A copy of a donor's drag image is displayed with the associated receptor. "Dragging a donor onto a receptor" does not mean that the donor itself is dragged onto the receptor blob. The reason for this is that some scenarios require donors to be glued onto multiple receptors, which would not be possible were the donors themselves actually moved.

Three other transactions take place between receptors, although they all involve donors. Each is performed by dragging a glob from one receptor to another. Depending on the requirements of the host application, one of three things happens when this is done:

- **The glob may be *transferred* from the first receptor to the other.**
- **The glob may be *duplicated*, so that the donor glued to the first receptor also becomes glued to the second.**
- **If the second receptor also has a glob, the globs glued to the two receptors may be *swapped*.**

Each of these three types of transactions is allowed or disallowed under host control. Since each is performed by the same user action, precedence is enforced in the case where more than one of these transaction types is allowed. Swaps have the highest precedence, transfers the lowest.

Since a receptor may only have one glob, any glob a receptor has must be unglued in order for another glob to be glued to the receptor. This implies another kind of transaction — *replacement* — which occurs implicitly when a glob is transferred or duplicated onto a receptor already having a glob, and during swaps. Replaces may also be allowed or disallowed, which modifies the availability of the three explicit inter-receptor transactions. Note that if replacement permission is turned off, swaps are disallowed entirely, regardless of swap permission.

The host application usually need not do much to engage in any of these transactions, beyond passing mouse clicks to the transaction handler. The transaction permissions are modifiable under host control. Frequently, the transaction types allowed by a scenario can be handled entirely by `BlobClick()` simply by setting the permissions appropriately.

## 2.3 Display Attributes: Hidden, Dimmed and Frozen Blobs

The discussion in the previous section assumes that all blobs are available for transactions. That need not be true. For one thing, a blob must be visible in a window. Hidden blobs are not considered by the transaction handler `BlobClick()`. The purpose of this section is to discuss various ways of controlling visibility of blobs and their availability to transactions. ("Availability" here is taken with respect to user actions performed with the mouse, as handled by `BlobClick()`. The host program can perform any transaction it likes, whether it makes sense or not.)

The most basic kind of control that may be exerted over a blob is whether it shows on the screen or not. `ShowBlob()` enables a blob and draws it. `HideBlob()` disables and erases it.

**Note**
It is usually the case that an enabled blob is also visible, but a blob may be enabled or disabled without drawing or erasing it, with `EnableBlob()` and `DisableBlob()`. This should be done carefully.

Blobs or parts of blobs may be drawn *normally* or *dimmed* under application control with `HiliteBlob()`. This routine takes a blob handle, a part code indicating the part of the blob to affect, and a mode in which to draw the indicated part.

```
typedef enum
{
                                        inStatBlob = 1,      /* static
                                        inDragBlob = 2,      /* drag r
                                        inFullBlob = 3       /* entire
};

typedef enum
{
                                        normalDraw = 1,      /* draw b
                                        dimDraw = 2
};
```

No redrawing is done if the blob is not enabled, or if the current drawing mode of the affected part does not change.

`HiliteBlobSet()` may be used to highlight entire sets of blobs.

Dimming the drag region of a blob makes it unavailable to transactions regardless of the general transaction permissions. This is a fundamental Blob Manager concept. The mouse cannot be used to drag a dimmed donor to a receptor, and dimmed receptors cannot receive donors, have globs unglued from them, or engage in inter-receptor transactions.

Donor blobs may be set to go dim automatically when they are simultaneously glued to a specified number of receptors. The participation of a visible donor blob in transactions is a function of its *maxiumum glue count* and its *actual glue count*. The maximum count is the upper limit on the number of receptors that the blob may be glued to simultaneously. This value is set at blob creation time and may be changed any time thereafter. The actual glue count is equal to the number of blobs the blob is glued to at a given moment. A donor goes dim when it has been glued to as many blobs as its maximum glue count allows, and becomes unavailable for further glue or duplication transactions. Its drag region does dim to signal that fact. If a donor is unglued from one or more of the receptors it is glued to, it becomes available for dragging again; its drag region is then undimmed as well.

**Note**
Transfers and swaps of donors from one receptor to another do not change the number of receptors a blob is glued to, and so are legal operations even for dimmed donors. It is also legal to drag a glob back to the donor it came from, even if the donor is dimmed.

Since single-use blobs can only be glued to one receptor, duplication is not allowed for them, even if duplicate permission is turned on.

The special constant `infiniteGlue` may be used to specify that a blob may be glued to an unlimited number of receptors.

```
# define                                infiniteGlue         0x8000
```

For scenarios where donors and receptors stand in one-to-one relationship, the donors are likely candidates for a maximum glue count of one (i.e., to be considered single-use blobs). For example, since states and capital cities are one-to-one, the cities could be made single-use. Each city blob would go dim if glued to a state, and undimmed if unglued from a state.

Single-use donors are not always appropriate. A tic-tac-toe game could be implemented with a 9-element set of receptor blobs for the 3x3 board and a 2-element set of donor blobs (an X and an O — one for each player). Since each player makes more than one move, single-use donor blobs wouldn’t work: they must be made multiple-use.

Use of single-use blobs may be likened to sampling without replacement, while use of infinitely-gluable blobs may be likened to sampling with replacement.

> **Note**
> A blob being dimmed is generally, but not always, synomymous with its being glued to the number of receptors specified in its `glueMax` field. Although the Blob Manager automatically dims blobs when it performs transactions, blobs may be dimmed “manually” with `HiliteBlob()`.

Sometimes it is desirable to make blobs unavailable for transactions without hiding or dimming them. For example, certain board positions that are not legal moves ought to be removed from play without changing the way they look. This is accomplished by *freezing* the blobs, which does not change their display state, but makes them unavailable for hit-testing or transaction operations. A frozen blob may be *thawed* to restore it to its state prior to being frozen. This includes restoring its display state, if that was changed after the blob was frozen.

For instance, the implementation of tic-tac-toe must allow users to drag the X-blob and the O-blob onto the board, but only one symbol is legal for each player. While each player takes his turn, the other player’s symbol could be frozen to make it unavailable.

A blob that is enabled and not frozen is said to be *active*. It may be dimmed or undimmed.

## 2.4 User-Initiated Transactions: The `BlobClick()` Routine

The Blob Manager procedure `BlobClick()` may be called whenever there is a mouse click in a window containing blobs. `BlobClick()` automatically performs Blob Manager transactions, and because it is so fundamental to most programs using blobs, may be configured to allow or disallow various types of transactions. This provides some degree of generality, which can be used to support a number of different kinds of scenarios.

`BlobClick()` takes four parameters: the location and time of a mouse click, a handle to the set of donors and a handle to the set of receptors. If there is no donor set (this often makes sense — see the “Donorless Scenarios” section of the Miscellaneous Topics), `nil` may be passed instead.

The following discussion describes first the default behavior of `BlobClick()` first, then those aspects of its behavior that may be modified.

If the mouse is pressed in the drag region of any donor blob, `BlobClick()` drags an outline of its drag region around until the mouse button is released. If the mouse is released in a receptor, the donor is glued to the receptor. If the receptor blob already has a glue blob, the two globs are swapped.

If the mouse is pressed in the drag region of any receptor that has a glob, `BlobClick()` drags an outline of the glob’s drag region around until the mouse button is released. If the button is released inside of a second receptor, the glue blob is transferred to the second receptor.

If the button in released inside of the donor that the glob came from, the glob is unglued.

If the mouse is double-clicked in the drag region of any receptor blob that has a glob, the glob is unglued. An outline of the glob is zoomed back to the donor blob.

The appropriate redrawing and/or dimming/undimming of the blobs involved is performed.

`BlobClick()` also indicates “bad drags,” which are defined as drags of blobs that don’t end up on something that the dragged blob can be glued to. Such bad drags are indicated by zooming the outline back to the starting point of the drag.

> **Note**
> `BlobClick()` only considers active blobs with undimmed drag regions, except when a glob is dragged off a receptor back to the donor it came from. In that case it doesn't matter if the donor is dimmed or not.

The above discussion indicates the behavior of `BlobClick()` according to the default transaction permissions, zooming flags, and dragging specifications. The default transaction permissions are unglue, replace, transfer and swap allowed, duplicate disallowed. Zoomback occurs for bad drags and for unglue transactions done by double- clicking. The drag specifications are: no constraint on the dragging axis, limit rectangle is the size of a standard Macintosh screen, slop rectangle is a wide-open rectangle. See the section on hit-testing, tracking and dragging for more information.

Any of these may be changed to suit the requirements of a particular scenario. The drag rectangles in particular normally should be changed to something more appropriate for the scenario. The default drag specifications may be obtained with `GetBDragRects()` and `GetBDragAxis()`, and changed with `SetBDragRects()` and `SetBDragAxis()`. The zoomback flags are obtained and set with `GetBCZoomFlags()` and `SetBCZoomFlags()`. Transaction permissions are obtained and set with `GetBCPermissions()` and `SetBCPermissions()`.

## 2.5 Program-Initiated Transactions

The transactions discussed so far are those initiated by the user with the mouse. The host often performs transactions independently of the user, for instance to clear a game board, show answers, or give hints.

Transactions performed by the host are often indicated by zooming so that it's clear to the user what's going on. Donors can be zoomed, for example, to the receptors that they should be associated with, to show a set of answers.

To perform non-user-initiated transactions, the host accesses lower level transaction routines: `GlueGlob()`, `UnglueGlob()`, `TransferGlob()`, `SwapGlob()`, `DupGlob()`. Most of these come in versions that do zooming: `ZGlueGlob()`, `ZUnglueGlob()`, `ZDupGlob()`. There are also routines (`UnglueGlobSet()`, `ZUnglueGlobSet()`) that unglue entire sets of receptors, which is useful for resetting the state of a scenario.

## 2.6 Quiet Blobs and Blob Matching

That receptor blobs may be considered as questions to be asked implies a mechanism for assessing correctness of answers. This mechanism involves the concepts of blob *matching* and whether blobs are *quiet* or *noisy*.

A blob may be designated either to require an explicit match or not. To be explicitly matched, a blob must have a glob that is a member of an explicitly specified set of matching blobs (blobs that are considered valid answers). Such a blob is *quiet* if it is matched, *noisy* if not. (Note that this implies that if the match set is empty, the blob must have *no* glob in order to be matched; this is occasionally a useful property.)

A blob that does not require an explicit match is the same, except it is also quiet if it has no glob, or if it has a glob but an empty match set. In the latter instance, an empty match set is taken to mean "any glob will do."

Whether a blob is quiet or noisy is nearly, but not exactly, synonymous with whether a blob is matched or not. Think of it this way: Blobs requiring explicit matches yell unless they have a glob that's in their match set. Blobs not requiring explicit matches don't yell unless they are given a glob that's not in their match set.

It may not be obvious how blobs not requiring explicit matches are useful. Consider the following example. Suppose arithmetic problems are to be presented for students to answer. This can be done by displaying the problem to be solved and two sets of blobs. The donor blobs represent the digits 0 through 9. The receptor blobs correspond to the digit positions of the answer as well as positions into which carry digits may be placed. Overall, the goal is to solve the problem. The question asked (implicitly) by each individual receptor is “what digit goes here?” Sometimes the answer may be “nothing” — for instance, in the carry digits.

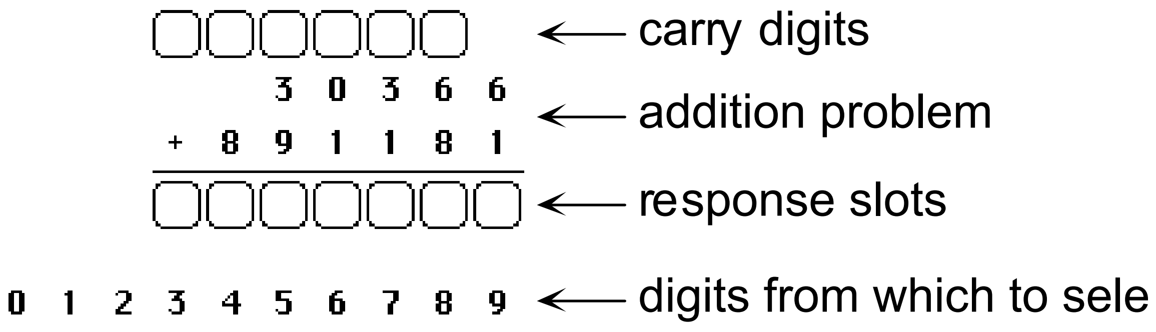

**Figure 2–2. Arithmetic problem scenario**

Carry receptors shouldn’t be noisy if they are left empty (e.g., if the user can carry in his head). If all the digits of the answer to the arithmetic problem were matched, but no carry receptors were filled in, the answer would still be correct. (This could not be done if all receptors had to be matched explicitly. Non-explicit receptors accept “no glob” as a match.) On the other hand, the carry digits should become noisy if the user carries incorrectly, because that indicates a failure of arithmetic ability. (This too is a characteristic of non-explicitly matched receptors.)

The distinction between blobs that require explicit matches and those that don’t is recognized by Blob Manager routines that test blobs or blob sets to see if they are quiet or not. `BlobQuiet()` returns true if the blob passed to it is quiet according to the rules given above. `BlobSetQuiet()` returns true if every blob in a set is quiet. With proper setup it is often possible to assess whether a user has answered all questions correctly, whether a game is over, etc., with a single call to `BlobSetQuiet()`.

**Note**
The Blob Manager provides no mechanisms for assessment of matches that depend on contingencies. Testing of conditions such as “blob A is matched by blob X only if blob B has blob Y glued to it” must be done by the host program.

The Blob Manager does provide a way to override the default matching criteria, by allowing a procedure to be installed which is called to test whether a blob is quiet or not.

## 2.7 Visual Feedback

It is sometimes desirable to provide feedback during problem solving activities. For example, a user may wish to know which of a set of answers are correct and which are not. The `BlobFeedback()` routine graphically indicates which in a set of blobs are noisy (incorrect) and which are quiet (correct). Typically, quiet receptor blobs are dimmed while noisy ones are displayed normally, or vice versa. If you want to change the dimming pattern and mode to something else, you can do so. For example, setting the pattern and mode to `black` and `patXor` results in dimmed blobs being drawn inverted.

Since normal blob display is overridden by this operation (receptors are not usually dimmed), `BlobFeedback()` freezes blob sets it operates on. For this reason, a call to `BlobFeedback()` is undone by a call to `ThawBlobSet()`.

If you prefer a different method of providing feedback (perhaps by drawing an outline around noisy blobs), you can freeze blobs yourself, but you should remember to thaw them out again.

> **Note**
> In some scenarios, giving feedback effectively induces a display mode, since the receptors are frozen. If this is so, you should provide a clear indication that blobs are inactive (perhaps an alert, or a Resume button) to avoid confusing the user.

When calling `BlobFeedback()`, your program may wish to freeze donor blobs also, because even if you allow them to be dragged onto frozen receptors, the Blob Manager won't attach them.

## 2.8 Distinction Between Donor and Receptor Blobs

The above discussion is phrased in terms of donor and receptor blobs for conceptual purposes only. Structurally, the two types of blobs are identical. The distinction is maintained by the host program, not by the Blob Manager. Under some circumstances, a single set may function both as the donor set and the receptor set.

Phrasing the discussion mostly in terms of interrogative scenarios is also simply for conceptual purposes. Blobs may be used to represent non-interrogative scenarios as well. Examples are the Tower of Hanoi, tic-tac-toe, symposium agenda construction with a set of event blobs and a set of time slot blobs, card games, or anagrams.