# Overview of The Macintosh Blob Manager

Paul DuBois
*dubois@primate.wisc.edu*

Document Revision: 1.02
Revision Date: 30 December 1993

The Macintosh Blob Manager is a collection of routines and data types designed to allow objects of arbitrary appearance to be created, then controlled in highly specific ways. The principal use of the Blob Manager is to represent and process situations (or *scenarios*) involving associative transactions between sets of objects. These include interrogative or object management tasks such as puzzles, games, quizzes (verbal and pictorial), simple animation. Blobs may contain text, shapes, patterns, and so forth — anything you can draw on the Macintosh screen. Blobs may be enabled, disabled, frozen, thawed, dragged around or resized. A blob may be dragged on top of another blob to become associated with it, or cleared from the blob it is currently associated with. Blobs may be marked so that they are considered matched if certain other blobs are associated with them. Matched and unmatched blobs may be displayed in distinct ways to provide visual feedback.

This article is primarily intended for programmers and is best read with a running copy of the Blob Manager Demo application in front of you. It presents an absurdly simplified overview and serves essentially as a brief introduction. More information may be found in “The Macintosh Blob Manager: A Programmer’s Guide.”

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. This 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 lines), since all necessary information is on the screen.

- 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 consistent with a number of Macintosh applications that involve object movement in some form or another. Also, the goal of being intuitive and easy to understand is apparently satisfied: I am told that young children find it very easy to manipulate blob scenarios.

## Blob Anatomy

Blobs are organized into *blob sets:* "no blob is an island, sufficient unto itself." Every blob in the set is represented internally by a *blob record* containing all the pertinent information about the blob. Although sets as mathematical entities are unordered, blob sets in particular have an ordering induced on them by the implementation (sets are treated as lists). When blobs are created, they are added to the end of their set, thus the *i*-th blob in a set — a concept with no mathematical meaning — is easily obtained. This allows treatment of blob sets as single-dimensional arrays. More complex data structures are easily induced on a blob set by making use of this fact.

The overall area that a blob occupies is its *general region*. This region is divided into two 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. Blob regions can be defined in the full generality of QuickDraw regions.

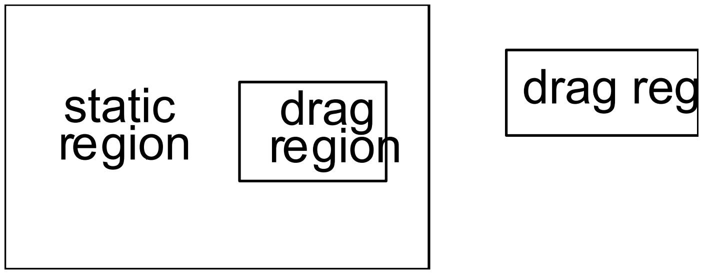

**Figure 1. Blob Regions**

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 synonymously.

The static region serves as a display-only region which is unaffected by user actions. It provides a means for display of information in a blob which remains unobscured 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.

## Blob Imaging

There are two methods for defining blob images. The first is to issue a set of QuickDraw calls as a picture definition, and install the picture into the blob record. Blobs imaged this way are *picture blobs*. Whenever the Blob Manager draws a picture blob, it replays the picture at the appropriate location, clipped to the region of the blob that is to be drawn. The second method allows blobs to arrange to have themselves passed to a procedure that knows how to draw them. Blobs imaged this way are *procedure blobs*. Picture and procedure blobs may be mixed within a scenario.

The Blob Manager can be told to draw either or both the static and drag regions of a blob. The image drawn in the static region is always the blob's own image, clipped to the static region. The image drawn in the drag region is either its own or the drag region of any blob that was dragged onto it, in either case clipped to its own drag region.

Each region is drawn either *normally* or *dimmed*. Each region has its own drawing mode, so each region can be drawn normally or dimmed independently of the other. By default, a region is dimmed by drawing it normally, then filling the region with gray in patBic mode. (The same method used to dim controls and menu items.) The reason for dimming blobs is explained later.

## Blob Transactions

The kinds of operations the user may perform with the mouse are known as *transactions*, which the host effects by calling the transaction handling routine `BlobClick()` when mouse clicks occur in a blob window. Because `BlobClick()` is so fundamental to most programs using blobs, it may be configured to allow or disallow various types of transactions. This generality can be exploited to support a wide variety of scenarios.

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.

Three other transactions types 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: (i) the glob may be *transferred* from the first receptor to the other; (ii) the glob may be *duplicated*, so that the donor glued to the first receptor also becomes glued to the second; (iii) 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 dragging a glob, precedence is enforced in the case where more than one of these transaction types is allowed. Swaps have the highest precedence, transfers the lowest.

A receptor may only have one glob, so any glob a receptor has must be unglued in order for another glob to be glued to it. This implies another kind of transaction — *replacement* — that occurs implicitly when a glob is transferred or duplicated onto a receptor already having a glob, and during swaps. Replaces may 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. Since the transaction permissions are modifiable under host control, the particular requirements of a scenario can often be handled entirely by `BlobClick()` simply by setting the permissions appropriately.

Besides performing transactions, `BlobClick()` also visually 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 blob outline back to the starting point of the drag.

The transactions discussed so far are those initiated by the user with the mouse. The host may perform 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.

## Hidden, Dimmed, and Frozen Blobs

This section discusses ways that visibility of blobs and their availability to transactions may be controlled by modifying display attributes.

*Visible/invisible blobs.* The most basic kind of control that may be exerted over a blob is whether it shows on the screen or not. (Hidden blobs are not considered by the transaction handler `BlobClick()`.) `ShowBlob()` enables a blob and draws it. `HideBlob()` disables and erases it.

*Dimmed blobs.* Blobs or parts of blobs may be drawn *normally* or *dimmed* under application control. 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.

*Blob glue counts.* 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 *maximum 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. (Blobs may be given an infinite maximum glue count.) 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.

Appropriate redrawing and/or dimming/undimming of blobs involved in user-initiated transactions is automatic.

*Frozen/thawed blobs.* Sometimes it is desirable to make blobs unavailable for transactions without hiding or dimming them, for example if certain board positions that are not legal moves are 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.

## Blob Matching, Visual Feedback

That receptor blobs may be considered as questions to be asked implies a mechanism for assessing correctness of answers. This is done by specifying, on a per-blob basis, which other blobs are considered a match for it. The Blob Manager has mechanisms for testing whether individual blobs or blob sets are matched or not. The test is usually made after calling `BlobClick()`. If `BlobClick()` performed a transaction resulting in a termination state (all answers correct, game over, etc.), this is often detectable with a single Blob Manager call.

It is sometimes desirable to provide feedback during problem solving activities, and the Blob Manager can visually indicate the state of blob matches. For example, a user may wish to know which answers are correct. The `BlobFeedback()` routine provides a mechanism for graphically indicating which in a set of blobs are matched and which are not. Typically, matched blobs are dimmed while unmatched ones are displayed normally, or vice versa.

## Blob Hit-testing, Tracking and Dragging

For many scenarios, the `BlobClick()` routine is entirely sufficient to handle all transactions. However, should the host itself wish to determine which blobs mouse clicks fall in, or do tracking or dragging operations itself, it can do so.

`FindBlob()` may be used to determine which, if any, of a set of blobs a mouse point lies in. `TestBlob()` determines whether a mouse point is in a given blob or not and returns a *part code* to indicate whether the point is in the static region or the drag region, or zero to indicate that the point is not in the blob.

When the mouse button is pressed in a blob, the Blob Manager may be told to drag around a gray outline of the blob or part of the blob. The blob may be dragged to the point where the mouse is released, or the Blob Manager may be told to simply return the result of the blob drag.

The tracking routines track the blob by making an outline of it follow the movements of the mouse. In contrast, `BTrackMouse()` is used to track whether the mouse remains within a stationary blob when the mouse moves. This is analogous to `TrackControl()`. It takes a blob, a starting point and a part code. The region specified by the part code is highlighted (by inverting it) whenever the mouse is inside it. If the mouse is released inside the region, `BTrackMouse()` returns true. Conceptually, this operation is identical to the way push buttons are tracked when clicked with the mouse. Since the appearance of blobs is less constrained, one can easily implement “controls” of highly arbitrary appearance. (Several Blob Manager Demo scenarios have vertical buttons — they’re actually blobs.)

## More on Transactions

Often illegal moves can be ruled out by freezing certain blobs or by setting the transaction permissions appropriately. For instance, in tic-tac-toe, all illegal moves can be prevented before the user clicks the mouse simply by freezing every square on the board that is already occupied. Not every situation is so simple, however, and in such cases it is desirable to know something about transactions while they are actually occurring.

For example, in a game of checkers, illegal moves cannot be prevented by freezing certain board positions, because the set of legal moves is dependent on the piece the user wishes to move. If it were known which piece the mouse was clicked in and which square of the board it was dragged to, the legality of the move could be assessed and the move either allowed or disallowed before the piece was actually moved. In a game of solitaire, only certain cards may legally be laid on any given card, but the set of cards that may be placed on that card varies. Knowing which cards the user planned to move, and where, is information that could be used to disallow transactions which might be legal according to the general transaction permissions, but illegal according to the particular semantics of the game.

To this end, `BlobClick()` allows the host to specify an advisory filter function to cope with scenarios that may be too complex to handle with the normal mechanism. `BlobClick()` calls this filter whenever the mouse is clicked in a blob and when a blob is dragged onto something to which it can be glued. The advisory function examines its arguments, takes whatever action it deems appropriate, then returns a boolean value indicating to `BlobClick()` indicating whether to continue processing or not. Installation of an advisory function allows the host access to information about transactions in progress, so that it can, in effect, “call the shots” on transactions without having to perform the mechanics of hit-testing, dragging, etc.

The advisory function does not extend the scope of allowable transactions or override the general transaction permissions. It is not even called unless `BlobClick()` considers the action done by the user to be legal according to those permissions, so the advisory can only further restrict the user’s actions. Thus if a donor is dragged to a receptor already having a glob, but replace permission is turned off, the advisory is called only when the donor is clicked, not when the donor is dragged to the receptor (`BlobClick()` wouldn’t consider replacing the glob that is already there, anyway).

The Blob Manager is in the public domain. It is provided in source form, with documentation. The Blob Manager was written for THINK C™. The primary example of Blob Manager programming is the Blob Manager Demo (included in the standard distribution). The Demo runs on top of the TransSkel transportable application skeleton (public domain, same author). Contact one of the addresses below for more information.

Internet: dubois@primate.wisc.edu

U.S. Mail: Paul DuBois
1220 Capitol Court
Madison, WI 53715–1299
USA