Libraries

COLLABORATORS			
	TITLE : Libraries		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		July 23, 2024	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Libraries

1.1 Amiga® RKM Libraries: 28 Graphics Sprites, Bobs and Animation

This chapter describes how to use the functions provided by the graphics library to manipulate and animate Graphic Elements (also called GELs). It is divided into six sections:

* An overview of the GELs animation system, including fundamental terms and structures
* Explanation of simple (hardware) Sprites and an example showing their usage
* Explanation of VSprites and an example showing their usage
* Explanation of Bobs and an example showing their usage
* Discussion of topics that apply to all GELs such as collision detection and data structure extensions.
* Discussion of animation, using AnimComps and AnimObs and an example showing their usage

1.2 28 Graphics Sprites, Bobs and Animation / About the GELs System

Before going into details, a quick glossary is in order. A playfield forms the background that GELs operate in. It encompasses the View, ViewPort, and RastPort data structures. (VSprites appear over, and Bobs

appear in the playfield.) Playfields can be created and controlled at several levels. Refer to the "Graphics Primitives" and "Layers Library" chapters for details on lower-level playfield control. The chapter "Intuition Screens" explains how to get higher-level access to playfields.

GELs, or graphic elements, are special graphic objects that appear in the foreground and can be moved easily around the display. They are software constructs based on the Amiga's sprite and blitter hardware. The GELs system is compatible with all playfield modes, including dual-playfield. All the various types of GELs are defined by data structures found in <graphics/gels.h>.

Types of GELs The GELs System

1.3 28 / About the GELs System / Types of GELs

The GEL types are (in order of increasing complexity):

VSprites — for Virtual Sprites. These are represented by the VSprite data structure and implemented with sprite hardware.

Bobs — Blitter Objects. These are represented by the VSprite and Bob data structures and implemented with blitter hardware.

AnimComps — Animation Components. These are represented by the VSprite, Bob and AnimComp data structures and implemented with blitter hardware.

AnimObs — Animation Objects. These are used to group AnimComps. They are not strictly GELs, but are described here.

Simple Sprites VSprites Bobs and AnimComps AnimObs VSprites vs. Bobs

1.4 28 / / Types Of GELs / Simple Sprites

Simple Sprites (also known as hardware sprites) are not really part of the GELs system but are the basis for VSprites. Simple Sprites are graphic objects implemented in hardware that are easy to define and easy to animate. The Amiga hardware has the ability to handle up to eight such sprite objects. Each Simple Sprite is produced by one of the Amiga's eight sprite DMA channels. They are 16-bits wide and arbitrarily tall.

The Amiga system software offers a choice of how to use these hardware sprites. After a sprite DMA channel has displayed the last line of a Simple Sprite, the system can reuse the channel for a different sprite lower on the screen. This is how VSprites are implemented--as a software construct based on the sprite hardware.

Hence, Simple Sprites are not really part of the animation system (they are not GELs). In fact, if Simple Sprites and GELs are used in the same display, the GELs system must be told specifically which Simple Sprites to

avoid. Simple Sprites are described in this chapter because they are alternatives to VSprites.

1.5 28 / / Types Of GELs / VSprites

The VSprite, or virtual sprite, is the simplest type of GEL. The VSprite data structure contains just a bit more information than is needed to define a hardware sprite. VSprites take advantage of the system's ability to reuse sprite DMA channels--each VSprite can be temporarily assigned to a hardware sprite, as needed. This makes it appear to an application program that it has a virtually unlimited supply of VSprites.

Since VSprites are based on hardware sprites, rules that apply to hardware sprites apply to VSprites too. VSprites are not rendered into the underlying BitMap of the playfield and so do not affect any bits in the BitMap. Because they are hardware based, they are positioned at absolute display coordinates and are not affected by the movement of screens. The starting position of a sprite must not occur before scanline 20, because of certain hardware DMA time constraints. VSprites have the same size limitations as hardware sprites, they are 16-bits wide and arbitrarily tall.

The VSprite data structure also serves as the root structure of more complex GEL types--Bobs and AnimComps.

1.6 28 / / Types Of GELs / Bobs and AnimComps

Like VSprites, Bobs and AnimComps are graphics objects that make animation easier. They are rendered using the blitter. The blitter is a special Amiga hardware component used to move data quickly and efficiently, optionally performing logical operations as it does. It can be used to move any kind of data but is especially well suited to moving rectangular blocks of display data.

It is important to keep in mind that Bobs and AnimComps are based on the blitter hardware while VSprites use the sprite hardware. However all three GEL types use the VSprite structure as their root data structure. The system uses pointers to link the VSprite, Bob and AnimComp structures, "extending" the VSprite structure to include all GEL types.

Since Bobs and AnimComps are rendered with the blitter they actually change the underlying playfield BitMap. The BitMap area where the GEL is rendered can be saved. By moving the GEL to new locations in small increments while also saving and restoring the Bitmap as you proceed, you can create an animation effect. Bobs and AnimComps use the same coordinates as the playfield and can be any size.

1.7 28 / / Types Of GELs / AnimObs

The AnimOb (Animation Object) is a data structure that is used to group one or more AnimComps for convenient movement. For example, an AnimOb could be created that consists of two AnimComps, one that looks like a planet and another containing a sequence that describes orbiting moons. By moving just the AnimOb the image of the planet can be moved across the display and the moons will travel along with it, orbiting the planet the entire time. The system automatically manages the movement of all the AnimComps associated with the AnimOb.

1.8 28 / / Types Of GELs / VSprites vs. Bobs

If you are going to manage the movement and sequencing of GELS yourself, you need to decide if sprite animation (VSprites) or blitter animation (Bobs, AnimComps and AnimObs) best suit your needs. If you've got simple requirements or lots of coding time, you may even opt to use only Simple Sprites, and control them yourself. On the other hand if you want the system to manage your animations, AnimComps must be used and they are Bobs at heart.

Some fundamental differences between VSprites and Bobs are:

* VSprite images and coordinates currently low-resolution pixels, even on a high resolution display. Bob images and coordinates have the same resolution as the playfield they are rendered into.

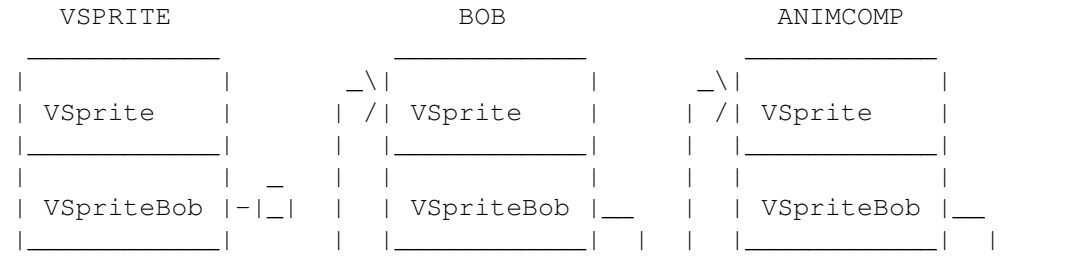
* VSprites have a maximum width of 16 (low resolution) pixels. Bobs can be any width (although large Bobs tend to slow down the system). The height of either VSprites or Bobs can be as tall as the display.

* VSprites have a maximum of three colors (Simple Sprites can have fifteen if they're attached). Because the system uses the Copper to control VSprite colors on the fly, the colors are not necessarily the same as those in the background playfield. Bobs can use any or all of the colors in the background playfield. Limiting factors include playfield resolution and display time. Bobs with more colors take longer to display.

* VSprites are positioned using absolute display coordinates, and don't move with screens. Bobs follow screen movement.

In general, VSprites offer speed, while Bobs offer flexibility.

The following figure shows how the various GEL data structures, VSprites, Bobs and AnimComps are linked together.



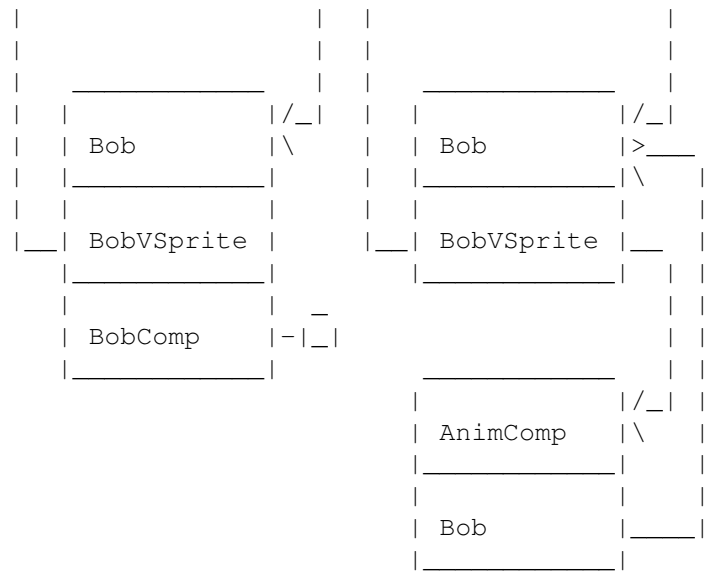


Figure 28-1: GEL Structure Layout

1.9 28 / About the GELs System / The GELs System

Before you can use the GELs system, you must set up a playfield. The GELs system requires access to a View, ViewPort, and RastPort structure. These structures may be set up through the grapics library or Intuition. For most examples in this chapter, the Intuition library is used for this purpose.

All GELs have a VSprite structure at their core. The system keeps track of all the GELs that it will display (the active GELs) by using a standard Exec list structure to link the VSprites. This list is accessed via the GelsInfo data structure, which in turn is associated with the RastPort. The GelsInfo structure is defined in the file <graphics/rastport.h>. When a new GEL is introduced to the system, the new GEL is added immediately ahead of the first existing GEL whose x, y value is greater than or equal to that of the new GEL, always trying to keep the list sorted.

As GELs are moved about the screen, their x, y values are constantly changing. SortGList() re-sorts this list by the x, y values. (Although this is a list of VSprite structures, bear in mind that some or all may really be Bobs or AnimComps.)

The basic set up of the GelsInfo structure requires three important fields: sprRsrvd, gelHead and gelTail. The sprRsrvd field tells the system which hardware sprites not to use when managing true VSprites. For instance, Intuition uses sprite 0 for the mouse pointer so this hardware sprite is not available for assignment to a VSprite. The gelHead and gelTail are VSprite structures that are used to manage the list of GELs. They are never displayed. To activate or deactivate a GEL, a system call is made to add it to or delete it from this list.

Other fields must be set up to provide for collision detection, color optimization, and other features. A complete example for setting up the

GelsInfo structure is shown in the animtools.c lisitng at the end of this chapter.

Initializing the GEL System

1.10 28 / / The GELs System / Initializing the GEL System

To initialize the animation system, call the system function InitGels(). It takes the form:

```
struct VSprite  *vsHead;
struct VSprite  *vsTail;
struct GelsInfo *gInfo;

InitGels(vsHead, vsTail, gInfo);
```

The vsHead argument is a pointer to the VSprite structure to be used as the GEL list head. (You must allocate an actual VSprite structure for vsHead to point to.) The vsTail argument is a pointer to the VSprite structure to be used as the GEL list tail. (You must allocate an actual VSprite structure for vsTail to point to.) The gInfo argument is a pointer to the GelsInfo structure to be initialized.

InitGels() forms these structures into a linked list of GELs that is empty except for these two dummy elements (the head and tail). It gives the head VSprite the maximum negative x and y positions and the tail VSprite the maximum positive x and y positions. This is to aid the system in keeping the list sorted by x, y values, so GELs that are closer to the top and left of the display are nearer the head of the list. The memory space that the VSprites and GelsInfo structures take up must already have been allocated. This can be done either by declaring them statically or explicitly allocating memory for them.

Once the GelsInfo structure has been allocated and initialized, GELs can be added to the system. Refer to the setupGelSys() and cleanupGelSys() functions in the animtools.c lisitng at the end of the chapter for examples of allocating, initializing and freeing a GelsInfo structure.

1.11 28 Sprites, Bobs and Animation / Using Simple (Hardware) Sprites

Simple Sprites can be used to create animations, so even though are not really part of the GELs system, they are desribed here as a possible alternative to using VSprites. For more information on the sprite hardware, including information on attached sprites, see the Amiga Hardware Reference Manual.

The SimpleSprite structure is found in <graphics/sprite.h>. It has fields that indicate the height and position of the Simple Sprite and a number that indicates which of the 8 hardware sprites to use.

Simple Sprites are always 16 bits wide, which is why there is no width member in the SimpleSprite structure. Currently, sprites always appear as

low-resolution pixels, and their position is specified in the same way. If the sprite is being moved across a high-resolution display in single pixel increments, it will appear to move two high-resolution pixels for each increment. In low-resolution mode, a single Lores pixel movement will be seen. Similarly, in an interlaced display, the y direction motions are in two-line increments. The same sprite image data is placed into both even and odd fields of the interlaced display, so the sprite will appear to be the same size in any display mode.

The upper left corner of the ViewPort area has coordinates (0,0). Unlike VSprites, the motion of a Simple Sprite can be relative to this position. (That is, if you create a Simple Sprite relative to your ViewPort and move the ViewPort around, the Simple Sprite can move as well, but its movement is relative to the origin of the ViewPort.)

The Sprite pairs 0/1, 2/3, 4/5, and 6/7 share color registers. See "VSprite Advanced Topics" later in this chapter, for precautions to take if Simple Sprites and VSprites are used at the same time.

The following figure shows which color registers are used by Simple Sprites.

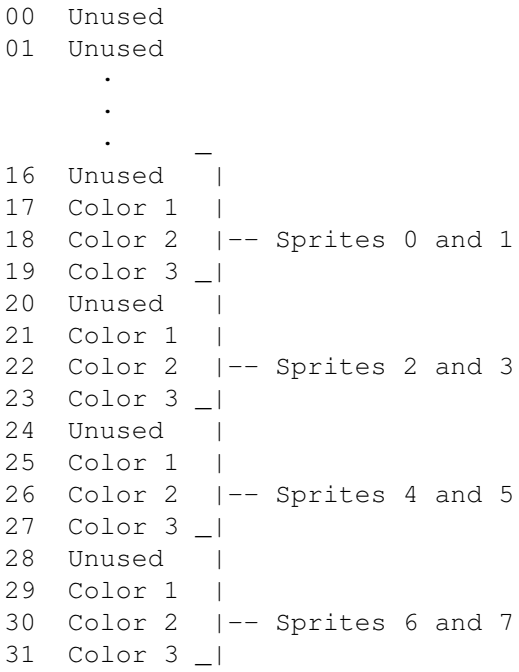


Figure 28-2: Sprite Color Registers

Sprites do not have exclusive use of the color registers. If the ViewPort is 5 bitplanes deep, all 32 of the system color registers will still be used by the playfield display hardware.

Note:

Color zero for all Sprites is always a "transparent" color, and the colors in registers 16, 20, 24, and 28 are not used by sprites. These colors will be seen only if they are rendered into a playfield. For

```
    further information, see the Amiga Hardware Reference Manual.

If there are two ViewPorts with different color sets on the same display,
a sprite will switch colors when it is moved across their boundary.  For
example, Sprite 0 and 1 will appear in colors 17-19 of whatever ViewPort
they happen to be over.  This is because the system jams all the
ViewPort's colors into the display hardware at the top of each ViewPort.

 Simple Sprite Functions
```

1.12 28 / Using Simple (Hardware) Sprites / Simple Sprite Functions

```
There are four basic functions that you use to to control Simple Sprites:

       GetSprite()  Attempts to allocates a sprite for exclusive use
    ChangeSprite()  Modifies a Simple Sprite's image data
      MoveSprite()  Changes a Simple Sprite's position
      FreeSprite()  Relinquishes a sprite so it can be used by others

To use these Simple Sprite functions (or the VSprite functions) the SPRITE
flag must have been set in the NewScreen structure for OpenScreen(). If
Intuition is not being used, this flag must be specified in the View and
ViewPort data structures before MakeVPort() is called.

 Accessing A Hardware Sprite
 Changing The Appearance Of A Simple Sprite
 Moving A Simple Sprite
 Relinquishing A Simple Sprite
 Controlling Sprite DMA
 Complete Simple Sprite Example
```

1.13 28 / / Simple Sprite Functions / Accessing A Hardware Sprite

```
GetSprite() is used to gain exclusive access to one of the eight hardware
sprites.  Once you have gained control of a hardware sprite, it can no
longer be allocated by the GELs system for use as a VSprite.  The call is
made like this:

    struct SimpleSprite *sprite;
    SHORT               number;

    if (-1 == (sprite_num = GetSprite(sprite, number)))
        return_code = RETURN_WARN;  /* did not get the sprite */

The inputs to the GetSprite() function are a pointer to a SimpleSprite
structure and the number (0-7) of the hardware sprite to be accessed, or
-1 to get the first available sprite.

A value of 0-7 is returned if the request was granted, specifying which
sprite was allocated.  A returned value of -1 means the requested sprite
was not available.  If the call succeeds, the SimpleSprite data structure
will have its sprite number field filled in with the appropriate number.
```

1.14 28 / / Sprite Functions / Changing The Appearance Of A Simple Sprite

The ChangeSprite() function can be used to alter the appearance of a Simple Sprite. ChangeSprite() substitutes new image data for the data currently used to display a Simple Sprite. It is called by the following sequence:

```
    struct ViewPort     *vp;
    struct SimpleSprite *sprite;
    APTR                newdata;

    ChangeSprite(vp, sprite, newdata);
```

The vp input to this function is a pointer to the ViewPort for this Sprite or 0 if this Sprite is relative only to the current View. The sprite argument is a pointer to a SimpleSprite data structure. (You must allocate an actual SimpleSprite structure for sprite to point to.) Set newdata to the address of an image data structure containing the new image. The data must reside in Chip (MEMF_CHIP) memory.

The structure for the new sprite image data is shown below. It is not a system structure, so it will not be found in the system includes, but it is described in the documentation for the ChangeSprite() call.

```
    struct spriteimage
        {
        UWORD posctl[2];  /* position and control data for this Sprite */

        /* Two words per line of Sprite height, first of the two words
         *  contains the MSB for color selection, second word contains
         * LSB (colors 0,1,2,3 from allowable color register selection
         *  set). Color '0' for any Sprite pixel makes it transparent.
         */
        UWORD data[height][2];  /* actual Sprite image */

        UWORD reserved[2];  /* reserved, initialize to 0, 0 */
        };
```

1.15 28 / / Simple Sprite Functions / Moving A Simple Sprite

MoveSprite() repositions a Simple Sprite. After this function is called, the Simple Sprite is moved to a new position relative to the upper left corner of the ViewPort. It is called as follows:

```
    struct ViewPort     *vp;
    struct SimpleSprite *sprite;
    SHORT               x, y;

    MoveSprite(vp, sprite, x, y);
```

There are three inputs to MoveSprite(). Set the vp argument to the address of the ViewPort with which this Simple Sprite interacts or 0 if this Simple Sprite's position is relative only to the current View. Set sprite to the address of your SimpleSprite data structure. The x and y

arguments sepcify a pixel position to which the Simple Sprite is to be moved.

1.16 28 / / Simple Sprite Functions / Relinquishing A Simple Sprite

The FreeSprite() function returns a hardware sprite allocated with GetSprite() to the system so that GELs or other tasks can use it. After you call FreeSprite(), the GELs system can use it to allocate VSprites. The syntax of this function is:

```
WORD sprite_number;

FreeSprite(sprite_number);
```

The sprite_number argument is the number (0-7) of the sprite to be returned to the system.

1.17 28 / / Simple Sprite Functions / Controlling Sprite DMA

Two additional functions used with Simple Sprites are the graphics library macros ON_SPRITE and OFF_SPRITE. These macros can be used to control sprite DMA. OFF_SPRITE prevents the system from displaying any sprites, whether Simple Sprites or VSprites. ON_SPRITE restores the sprite display.

Be Careful With OFF_SPRITE.

The Intuition mouse pointer is a sprite. Thus, if OFF_SPRITE is used, Intuition's pointer will disappear too. Use care when calling OFF_SPRITE. The macro turns off sprite fetch DMA, so that no new sprite data is fetched. Whatever sprite data was last being displayed at this point will continue to be displayed for every line on the screen. This may lead to a vertical color bar if a sprite is being displayed when OFF_SPRITE is called.

1.18 28 Graphics Sprites, Bobs and Animation / Using Virtual Sprites

This section describes how to set up the VSprite structure so that it represents a true VSprite. True VSprites are managed by the GELs system which converts them to Simple Sprites and displays them. (Later sections describe how a VSprite structure can be set up for Bobs and AnimComps.)

Before the system is told of a VSprite's existence, space for the VSprite data structure must be allocated and initialized to correctly represent a VSprite. Since the system does no validity checking on the VSprite structure, the result of using a bogus structure is usually a fireworks display, followed by a system failure.

The system software provides a way to detect collisions between VSprites and other on-screen objects. There is also a method of extending the VSprite structure to incorporate user defined variables. These subjects

are applicable to all GELs and are explained later in the section on
"Collisions and GEL Structure Extensions".

1.19 28 / Using Virtual Sprites / Specification of VSprite Structure

The VSprite structure is defined in the include file <graphics/gels.h> as
follows:

```
    /* VSprite structure definition */
    struct VSprite {
        struct VSprite *NextVSprite;
        struct VSprite *PrevVSprite;
        struct VSprite *DrawPath;
        struct VSprite *ClearPath;
        WORD            OldY, OldX;
        WORD            Flags;
        WORD            Y, X;
        WORD            Height;
        WORD            Width;
        WORD            Depth;
        WORD            MeMask;
        WORD            HitMask;
        WORD           *ImageData;
        WORD           *BorderLine;
        WORD           *CollMask;
        WORD           *SprColors;
        struct Bob     *VSBob;
        BYTE            PlanePick;
        BYTE            PlaneOnOff;
        VUserStuff      VUserExt;
        };
```

There are two primary ways to allocate and fill in space for VSprite data.
They can be statically declared, or a memory allocation function can be
called and they can be filled in programmatically. The declaration to
statically set up a VSprite structure is listed below.

```
    /* VSprite static data definition.
    ** must set the following for TRUE VSprites:
    **    VSPRITE flag.
    **    Width to 1.
    **    Depth to 2.
```

```
    **    VSBob to NULL.
    */
    struct VSprite myVSprite =
        {
        NULL, NULL, NULL, NULL, 0, 0, VSPRITE, 0, 0, 5, 1, 2, 0, 0,
        &myImage, 0, 0, &mySpriteColors, NULL, 0x3, 0, 0
        };
```

This static allocation gives the required VSprite structure, but does not allocate or set up collision masks for the VSprite. Note that the VSprite structure itself does not need to reside in Chip memory.

Refer to the makeVSprite() and freeVSprite() functions in the animtools.c listing at the end of the chapter for an example of dynamically allocating, initializing and freeing a VSprite structure.

1.20 28 / Using Virtual Sprites / Reserved VSprite Members

These VSprite structure members are reserved for system use (do not write to them):

```
    NextVSprite and PrevVSprite -- These are used as links in the
                                   GelsInfo list.

         DrawPath and ClearPath -- These are used for Bobs, not true
                                   VSprites.

                  OldY and OldX -- Previous position holder, the system
                                   uses these for double buffered Bobs,
                                   but application programs can read
                                   them too.
```

The values can be set like this:

```
    myVSprite.NextVSprite = NULL;
    myVSprite.PrevVSprite = NULL;
    myVSprite.DrawPath  = NULL;
    myVSprite.ClearPath = NULL;
    myVSprite.OldY = 0;
    myVSprite.OldX = 0;
```

1.21 28 / Using Virtual Sprites / Using VSprite Flags

The Flags member of the VSprite structure is both read and written by the system. Some bits are used by the application to inform the system; others are used by the system to indicate things to the application.

The only Flags bits that are used by true VSprites are:

VSPRITE
    This may be set to indicate to the system that it should treat the structure as a true VSprite, not part of a Bob. This affects the

interpretation of the data layout and the use of various system variables.

VSOVERFLOW
The system sets this bit in the true VSprites that it is unable to display. This happens when there are too many in the same scan line, and the system has run out of Simple Sprites to assign. It indicates that this VSprite has not been displayed. If no sprites are reserved, this means that more than eight sprites touch one scan line. This bit will not be set for Bobs and should not be changed by the application.

GELGONE
If the system has set GELGONE bit in the Flags member, then the GEL associated with this VSprite is not on the display at all, it is entirely outside the GEL boundaries. This area is defined by the GelsInfo members topmost, bottommost, leftmost and rightmost (see <graphics/rastport.h>). On the basis of that information, the application may decide that the object need no longer be part of the GEL list and may decide to remove it to speed up the consideration of other objects. Use RemVSprite() (or RemBob(), if it's a Bob) to do this. This bit should not be changed by the application.

The VSprite.Flags value should be initialized like this for a VSprite GEL:

```
myVSprite.Flags = VSPRITE;
```

1.22 28 / Using Virtual Sprites / VSprite Position

To control the position of a VSprite, the x and y variables in the VSprite structure are used. These specify where the upper left corner of the VSprite will be, relative to the upper left corner of the playfield area it appears over. So if VSprites are used under Intuition and within a screen, they will be positioned relative to the upper left-hand corner of the screen.

In a 320 by 200 screen, a y value of 0 puts the VSprite at the top of that display, a y value of (200 - VSprite height) puts the VSprite at the bottom. And an x value of 0 puts the VSprite at the left edge of that display, while an x value of (320 - VSprite width) puts the VSprite at the far right. Values of less than (0,0) or greater than (320, 200) may be used to move the VSprite partially or entirely off the screen, if desired.

See the "Graphics Primitives" chapter for more information on display coordinates and display size. See the Amiga Hardware Reference Manual for more information on hardware sprites.

Position VSprites Properly.

It is important that the starting position of true VSprites is not less than -20 in the y direction, which is the start of the active display area for sprites. Also, if they are moved too far to the left, true VSprites may not have enough DMA time to be displayed.

The x, y values may be set like this to put the VSprite in the upper-left:

```
    myVSprite.Y = 0;
    myVSprite.X = 0;
```

1.23 28 / Using Virtual Sprites / VSprite Image Size

A true VSprite is always one word (16 pixels) wide and may be any number of lines high. It can be made to appear thinner by making some pixels transparent. Like Simple Sprites, VSprite pixels are always the size of a pixel in low-resolution mode (320x200); regardless of the resolution the display is set to. To specify how many lines make up the VSprite image, the VSprite structure member, Height, is used. VSprites always have a Depth of two, allowing for three colors. The values may be set like this:

```
    myVSprite.Width  = 1;      /* ALWAYS 1 for true VSprites. */
    myVSprite.Height = 5;      /* The example height. */
    myVSprite.Depth  = 2;      /* ALWAYS 2 for true VSprites. */
```

1.24 28 / Using Virtual Sprites / VSprites and Collision Detection

Some members of the VSprite data structure are used for special purposes such as collision detection, user extensions or for system extensions (such as Bobs and AnimComps). For most applications these fields are set to zero:

```
    myVSprite.HitMask    = 0;  /* These are all used for collision */
    myVSprite.MeMask     = 0;  /* detection                        */
    myVSprite.BorderLine = 0;
    myVSprite.CollMask   = 0;

    myVSprite.VUserExt = 0;    /* Only use this for user extensions to */
                               /* VSprite                              */

    myVSprite.VSBob = NULL;    /* Only Bobs and AnimComps need this */
```

The special uses of these fields are explained further in the sections that follow.

1.25 28 / Using Virtual Sprites / VSprite Image Data

The ImageData pointer of the VSprite structure must be initialized with the address of the first word of the image data array. The image data array must be in Chip memory. It takes two sequential 16-bit words to define each line of a VSprite. This means that the data area containing the VSprite image is always Height x 2 (10 in the example case) words long.

A VSprite image is defined just like a real hardware sprite. The combination of bits in corresponding locations in the two data words that define each line select the color for that pixel. The first of the pair

of words supplies the low-order bit of the color selector for that pixel; the second word supplies the high-order bit.

These binary values select colors as follows:

```
00 - selects "transparent"
01 - selects the first of three VSprite colors
10 - selects the second VSprite color
11 - selects the third VSprite color
```

In those areas where the combination of bits yields a value of 0, the VSprite is transparent. This means that the playfield, and all Bobs and AnimComps, and any VSprite whose priority is lower than this VSprite will all show through in transparent sections. For example:

```
(&VSprite->ImageData)          1010 0000 0000 0000
(&VSprite->ImageData + 1)      0110 0000 0000 0000
```

Reading from top to bottom, left to right, the combinations of these two sequential data words form the binary values of 01, 10, 11, and then all 00s. This VSprite's first pixel will be color 1, the next color 2, the third color 3. The rest will be transparent, making this VSprite appear to be three pixels wide. Thus, a three-color image, with some transparent areas, can be formed from a data set like the following sample:

Address	Binary Data	VSprite Image Data
mem	1111 1111 1111 1111	Defines top line
mem + 1	1111 1111 1111 1111	3333 3333 3333 3333
mem + 2	0011 1100 0011 1100	Defines second line
mem + 3	0011 0000 0000 1100	0033 1100 0011 3300
mem + 4	0000 1100 0011 0000	Defines third line
mem + 5	0000 1111 1111 0000	0000 3322 2233 0000
mem + 6	0000 0010 0100 0000	Defines fourth line
mem + 7	0000 0011 1100 0000	0000 0032 2300 0000
mem + 8	0000 0001 1000 0000	Defines fifth line
mem + 9	0000 0001 1000 0000	0000 0003 3000 0000

The VSprite.Height for this sample image is 5.

1.26 28 / Using Virtual Sprites / Specifying the Colors of a VSprite

The system software provides a great deal of versatility in the choice of colors for Virtual Sprites. Each VSprite has its own set of three colors, pointed to by SprColors, which the system jams into the display's Copper list as needed.

SprColors points to the first of three 16-bit values. The first value represents the color used for the VSprite bits that select color 1, the

second value is color 2, and the third value is color 3. When the system assigns a hardware sprite to carry the VSprite's image, it jams these color values into the Copper list (the intermediate Copper list, not the color table), so that the View's colors will be correct for this VSprite at the time the VSprite is displayed. It doesn't jam the original palette's colors back after the VSprite is done. If there is another VSprite later, that VSprite's colors will get jammed; if there is not another VSprite, the colors will remain the same until the next ViewPort's colors get loaded.

If the SprColors pointer is set to NULL, that VSprite does not generate a color-change instruction stream for the Copper. Instead, the VSprite appears drawn in whatever color set that the hardware sprite happens to have in it already.

Since the registers are initially loaded with the colors from the ViewPort's ColorMap, if all VSprites have NULL SprColors, they will appear in the ViewPort's colors.

To continue our example, a set of colors can be declared and the VSprite colors set with the following statements:

```
    WORD mySpriteColors[] = { 0x0000,
                              0x00f0,
                              0x0f00 };  /* Declare colors statically  */

    myVSprite.SprColors = mySpriteColors;  /* Assign colors to VSprite */
```

1.27 28 / Using Virtual Sprites / Adding and Removing VSprites

Once a true VSprite has been set up and initialized, the obvious next step is to give it to the system by adding it to the GEL list. The VSprite may then be manipulated as needed. Before the program ends, the VSprite should be removed from the GELs list by calling RemVSprite(). A typical calling sequence could be performed like so:

```
    struct VSprite  myVSprite = {0};
    struct RastPort myRastPort = {0};

    AddVSprite(&myVSprite, &myRastPort);

    /* Manipulate the VSprite as needed here */

    RemVSprite(&myVSprite);
```

The &myVSprite argument is a fully initialized VSprite structure and &myRastPort is the RastPort with which this VSprite is to be associated. Note that you will probably not like the results if you try to RemVSprite() a VSprite that has not been added to the system with AddVSprite(). See the Amiga ROM Kernel Reference Manual: Includes and Autodocs for additional information on these functions.

1.28 28 / Using Virtual Sprites / Changing VSprites

Once the VSprite has been added to the GELs list and is in the display, some of its characteristics can be changed dynamically by:

* Changing y, x to a new VSprite position

* Changing ImageData to point to a new VSprite image

* Changing SprColors to point to a new VSprite color set

Study the next two sections to find out how to reserve hardware Sprites for use outside the VSprite system and how to assign the VSprites.

1.29 28 / Using Virtual Sprites / Getting the VSprite List In Order

When the system has displayed the last line of a VSprite, it is able to reassign the hardware sprite to another VSprite located at a lower position on the screen. The system allocates hardware sprites in the order in which it encounters the VSprites in the list. Therefore, the list of VSprites must be sorted before the system can assign the use of the hardware Sprites correctly.

The function SortGList() must be used to get the GELs in the correct order before the system is asked to display them. This sorting step is essential! It should be done before calling DrawGList(), whenever a GEL has changed position. This function is called as follows:

```
struct RastPort myRastPort = {0};

SortGList(&myRastPort);
```

The only argument is a pointer to the RastPort structure containing the GelsInfo.

1.30 28 / Using Virtual Sprites / Displaying the VSprites

The next few sections explain how to display the VSprites. The following system functions are used:

```
DrawGList()  Draws the VSprites into the current RastPort.
   MrgCop()  Installs the VSprites into the display.
 LoadView()  Asks the system to display the new View.
  WaitTOF()  Synchronizes the functions with the display.
```

```
Drawing the Graphics Elements    Loading the New View
Merging VSprite Instructions     Synchronizing with the Display
```

1.31 28 / / Displaying the VSprites / Drawing the Graphics Elements

The system function called DrawGList() looks through the list of GELS and prepares the necessary Copper instructions and memory areas to display the data. This function is called as follows:

```
struct RastPort myRastPort = {0};
struct ViewPort myViewPort = {0};

DrawGList(&myRastPort, &myViewPort);
```

The myRastPort argument specifies the RastPort containing the GelsInfo list with the VSprites that you want to display. The &myViewPort argument is a pointer to the ViewPort for which the VSprites will be created.

1.32 28 / / Displaying the VSprites / Merging VSprite Instructions

Once DrawGList() has prepared the necessary instructions and memory areas to display the data, the VSprites are installed into the display with MrgCop(). (DrawGList() does not actually draw the VSprites, it only prepares the Copper instructions.)

```
struct View *view;

MrgCop(view);
```

The view is a pointer to the View structure whose Copper instructions are to be merged.

1.33 28 / / Displaying the VSprites / Loading the New View

Now that the display instructions include the definition of the VSprites, the system can display this newly configured View with the LoadView() function:

```
struct View *view;

LoadView(view);
```

Again, view is a pointer to the View that contains the the new Copper instruction list (if you are using GELs in an Intuition Screen, do not call LoadView().)

The Copper instruction lists are double-buffered, so this instruction does not actually take effect until the next display field occurs. This avoids the possibility of some function trying to update the Copper instruction list while the Copper is trying to use it to create the display.

1.34 28 / / Displaying the VSprites / Synchronizing with the Display

To synchronize application functions with the display, call the system function WaitTOF(). WaitTOF() holds your task until the vertical-blanking interval (blank area at the top of the screen) has begun. At that time, the system has retrieved the current Copper instruction list and is ready to allow generation of a new list.

```
WaitTOF();
```

WaitTOF() takes no arguments and returns no values. It simply suspends your task until the video beam is at the top of field.

1.35 28 Graphics: Sprites, Bobs and Animation / VSprite Advanced Topics

This section describes advanced topics pertaining to VSprites. It contains details about reserving hardware sprites for use outside of the GELs VSprite system, information about how VSprites are assigned, and more information about VSprite colors.

1.36 28 / VSprite Advanced Topics / Reserving Hardware Sprites

To prevent the VSprite system from using specific hardware sprites, set the sprRsrvd member of the GelsInfo structure. The pointer to the GelsInfo structure is contained in the RastPort structure. If all of the bits of this 8-bit value are ones (0xFF), then all of the hardware sprites may be used by the VSprite system. If any of the bits is a 0, the sprite corresponding to that bit will not be utilized by VSprites.

```
Reserving Can Cause Problems.
-----------------------------
Reserving sprites increases the likelihood of the system not being
able to display a VSprite (VSOVERFLOW). See the next section,
"How VSprites are Assigned," for further details on this topic.
```

You reserve a sprite by setting its corresponding bit in sprRsrvd. For instance, to reserve sprite zero only, set sprRsrvd to 0x01. To reserve sprite three only, set sprRsrvd to 0x08.

If a hardware sprite is reserved, the system will not consider it when it makes VSprite assignments. Remember, hardware sprite pairs share color register sets. If a hardware sprite is reserved, its mate should probably be reserved too, otherwise the reserved sprite's colors will change as the unreserved mate is assigned different VSprites. For example, it is common practice to reserve Sprites 0 and 1, so that the Intuition pointer (Sprite 0) is left alone. This could be accomplished with the following statements:

```
struct RastPort myRastPort = {0};  /* the View structure is defined */
```

```
    myRastPort.GelsInfo->sprRsrvd = 0x03;  /* reserve 0 and 1 */
```

The GfxBase structure may be examined to find which sprites are already in use. This may, at your option, impact what sprites you reserve. If Intuition is running, sprite 0 will already be in use as its pointer.

The reserved sprite status is accessible as

```
    currentreserved = GfxBase->SpriteReserved;
```

The next section presents a few trouble-shooting techniques for VSprite assignment.

1.37 28 / VSprite Advanced Topics / How VSprites Are Assigned

Although VSprites are managed for you by the GELs system there are some underlying limitations which could cause the system to run out of VSprites.

As the system goes through the GEL list during DrawGList(), whenever it finds a true VSprite, it goes through the following procedure. If there is a Simple Sprite available (after the reserved sprites and preceding VSprites are accounted for), Copper instructions are added that will load the sprite hardware with this VSprite's data at the right point on the screen. It may need to add a Copper instruction sequence to load the display's colors associated with the sprite as well.

There are only 8 real sprite DMA channels. The system will run out of hardware sprites if it is asked to display more than eight VSprites on one scan line. This limit goes down to four when the VSprites have different SprColor pointers. During the time that there is a conflict, the VSprites that could not be put into Simple Sprites will disappear. They will reappear when (as the VSprites are moved about the screen) circumstances permit.

These problems can be alleviated by taking some precautions:

* Minimize the number of VSprites to appear on a single horizontal line.

* If colors for some Virtual Sprites are the same, make sure that the pointer for each of the VSprite structures for these Virtual Sprites points to the same memory location, rather than to a duplicate set of colors elsewhere in memory. The system will know to map these into Sprite pairs.

If a VSprite's SprColors are set to NULL, the VSprite will appear in the ViewPort's ColorMap colors. The system will display the VSprite in any one of a set of four different possible color groupings as indicated in the Simple Sprite section above.

If SprColors points to a color set, the system will jam SprColors into the display hardware (via the Copper list), effectively overriding those ColorMap registers. The values in the ColorMap are not overwritten, but anything in the background display that used to appear in the ColorMap

colors will appear in SprColors colors.

1.38 28 / Advanced Topics / How VSprite and Playfield Colors Interact

At the start of each display, the system loads the colors from the ViewPort's color table into the display's hardware registers, so whatever is rendered into the BitMap is displayed correctly. But if the VSprite system is used, and the colors are specified (via SprColors) for each VSprite, the SprColors will be loaded by the system into the display hardware, as needed. The system does this by generating Copper instructions that will jam the colors into the hardware at specific moments in the display cycle. Any BitMap rendering, including Bobs, which share colors with VSprites, may change colors constantly as the video display beam progresses down the screen.

This color changing can be avoided by taking one of the following precautions:

* Use a four bitplane playfield, which only allows the lower 16 colors to be rendered into the BitMap (and allows Hires display mode).

* If a 32-color playfield display is being used, avoid rendering in colors 17-19, 21-23, 25-27, and 29-32, which are the colors affected by the VSprite system.

* Specify the VSprite SprColors pointer as a value of NULL to avoid changing the contents of any of the hardware sprite color registers. This may cause the VSprites to change colors depending on their positions relative to each other, as described in the previous section.

1.39 28 Graphics Sprites, Bobs and Animation / Using Bobs

The following section describes how to define a Bob (blitter object). Like VSprites, a Bob is a software construct designed to make animation easier. The main advantage of a Bob over a VSprite is that it allows more colors and a width greater than 16 pixels to be defined.

To create a Bob, you need both a Bob structure and a VSprite structure. The components common to all GELs -- height, collision-handling information, position in the drawing area and pointers to the image definition -- are part of the VSprite structure. The added features -- such as drawing sequence, data about saving and restoring the background, and other features not applicable to VSprites -- are further specified in the Bob structure.

1.40 28 / Using Bobs / The VSprite Structure and Bobs

The root VSprite structure is set up as described earlier for true VSprites, with the following exceptions:

Y, X	Bob position is always in pixels that are the same resolution as the display.
Flags	For Bobs, the VSPRITE flag must be cleared. SAVEBACK or OVERLAY can also be used.
Height, Width	Bob pixels are the size of the background pixels. The Width of Bobs may be greater than one word.
Depth	The Depth of a Bob may be up to as deep as the playfield, provided that enough image data is provided.
ImageData	This is still a pointer to the image, but the data there is organized differently.
SprColors	This pointer should be set to NULL for Bobs.
VSBob	This is a pointer to the Bob structure set up as described below.

1.41 28 / Using Bobs / VSprite Flags and Bobs

The bits in the VSprite.Flags field that apply to a Bob are the VSPRITE flag, the SAVEBACK flag and the OVERLAY flag. When a VSprite structure is used to define a Bob, the VSPRITE flag in the VSprite.Flags field must be set to zero. This tells the system that this GEL is a Bob type.

To have the GEL routines save the background before the Bob is drawn and restore the background after the Bob is removed, specify the SAVEBACK flag (stands for "save the background") in the VSprite structure Flags field. If this flag is set, the SaveBuffer must have been allocated, which is where the system puts this saved background area. The buffer must be large enough to save all the background bitplanes, regardless of how many planes the Bob has. The size in words can be calculated as follows:

```
/* Note that Bob.Width is in units of words. */
size = Bob.Width * Bob.Height * RastPort.BitMap.Depth;
```

To allocate this space, the graphics function AllocRaster() can be used. AllocRaster() takes the width in bits, so it is a convenient way to allocate the space needed. The makeBob() routine below shows another way to correctly allocate this buffer. For example:

```
/* space for 16 bits times 5 lines times 5 bitplanes */
```

```
    myBob.SaveBuffer = AllocRaster( (UWORD) 16, (UWORD) (5 * 5) );


    Warning:
    --------
    The SaveBuffer must be allocated from Chip memory and contain an
    even number of word-aligned bytes.  The AllocRaster() function
    does this for you.  The AllocRaster() function rounds the width
    value up to the next integer multiple of 16 bits which is greater
    than or equal to the current value an it obtains memory from the Chip
    memory pool.
```

OVERLAY is the other VSprite.Flags item that applies to Bobs. If this flag is set, it means that the background's original pixels show through in any area where there are 0 bits in the Bob's shadow mask (ImageShadow, explained later). The space for the ImageShadow shadow mask must have been allocated and initialized. The ImageShadow mask must be allocated from Chip memory.

If the OVERLAY bit is cleared, the system uses the entire rectangle of words that define the Bob image to replace the playfield area at the specified x,y coordinates. See the paragraphs below called "ImageShadow.

1.42 28 / Using Bobs / The Bob Structure

The Bob structure is defined in the include file <graphics/gels.h> as follows:

```
struct Bob
  {
  WORD               Flags;     /* general purpose flags                 */
  WORD              *SaveBuffer;/* buffer for background save            */
  WORD              *ImageShadow; /* shadow mask of image                */
  struct Bob        *Before;   /* draw this Bob before Bobs on this list */
  struct Bob        *After;     /* draw this Bob after Bobs on this list */
  struct VSprite    *BobVSprite;/* this Bob's VSprite definition         */
  struct AnimComp   *BobComp;   /* pointer to this Bob's AnimComp def    */
  struct DBufPacket *DBuffer;   /* pointer to this Bob's dBuf packet     */
  BUserStuff         BUserExt;  /* Bob user extension                    */
  };
```

The Bob structure itself does not need to be in Chip memory. The (global) static declaration of a Bob structure could be done like so:

```
    struct Bob myBob =
        {
        0, NULL, NULL, NULL, NULL, NULL, NULL, NULL, 0
        };
```

However, since most of the Bob structure members are pointers, it is more common to allocate and set the Bob up dynamically. Refer to the makeBob() and freeBob() functions in the "animtools.c" example at the end of the chapter for an example of allocating, initializing and freeing a Bob structure.

Linking Bob and VSprite Structures

1.43 28 / / The Bob Structure / Linking Bob and VSprite Structures

The VSprite and Bob structures must point to one another, so that the system can find the entire GEL. The structures are linked with statements like this:

```
myBob.BobVSprite = &myVSprite;
myVSprite.VSBob  = &myBob;
```

Now the system (and the application program) can go back and forth between the two structures to obtain the various Bob variables.

1.44 28 / Using Bobs / Using Bob Flags

The following paragraphs describe how to set the Flags field in the Bob structure (note that these flags do not apply to the Flags field of the VSprite structure).

SAVEBOB
: To tell the system not to erase the old image of the Bob when the Bob is moved, specify the SAVEBOB flag in the Bob structure Flags field. This makes the Bob behave like a paintbrush. It has the opposite effect of SAVEBACK.

  It's Faster To Draw A New Bob.

  It takes longer to preserve and restore the raster image than simply to draw a new Bob image wherever required.

BOBISCOMP
: If this Bob is part of an AnimComp, set the BOBISCOMP flag in the Bob structure to 1. If the flag is a 1, the pointer named BobComp must have been initialized. Otherwise, the system ignores the pointer, and it may be left alone (though it's good practice to initialize it to NULL). See "Animation Data Structures" for a discussion of AnimComps.

BWAITING
: This flag is used solely by the system, and should be left alone. When a Bob is waiting to be drawn, the system sets the BWAITING flag in the Bob structure to 1. This occurs only if the system has found a Before pointer in this Bob's structure that points to another Bob. Thus, the system flag BWAITING provides current draw-status to the system. Currently, the system clears this flag on return from each call to DrawGList().

BDRAWN
: This is a system status flag that indicates to the system whether or not this Bob has already been drawn. Therefore, in the process of examining the various Before and After flags, the drawing routines

can determine the drawing sequence. The system clears this flag on return from each call to DrawGList().

BOBSAWAY
To initiate the removal of a Bob during the next call to DrawGList(), set BOBSAWAY to 1. Either the application or the system may set this Bob structure system flag. The system restores the background where it has last drawn the Bob. The system will unlink the Bob from the system GEL list the next time DrawGList() is called, unless the application is using double-buffering. In that case, the Bob will not be unlinked and completely removed until two calls to DrawGList() have occurred and the Bob has been removed from both buffers. The RemBob() macro sets the BOBSAWAY flag.

BOBNIX
When a Bob has been completely removed, the system sets the BOBNIX flag to 1 on return from DrawGList(). In other words, when the background area has been fully restored and the Bob has been removed from the GEL list, this flag in is set to a 1. BOBNIX is especially significant when double-buffering because when an application asks for a Bob to be removed, the system must remove it from both the drawing buffer and from the display buffer. Once BOBNIX has been set, it means the Bob has been removed from both buffers and the application is free to reuse or deallocate the Bob.

SAVEPRESERVE
The SAVEPRESERVE flag is a double-buffer version of the SAVEBACK flag. If using double-buffering and wishing to save and restore the background, set SAVEBACK to 1. SAVEPRESERVE is used by the system to indicate whether the Bob in the "other" buffer has been restored; it is for system use only.

1.45 28 / Using Bobs / Specifying the Size of a Bob

Bobs do not have the 16-pixel width limit that applies to VSprites. To specify the overall size of a Bob, use the Height and Width members of the root VSprite structure. Specify the Width as the number of 16-bit words it takes to fully contain the object. The number of lines is still specified with the Height member in the VSprite data structure.

As an example, suppose the Bob is 24 pixels wide and 20 lines tall. Use statements like the following to specify the size:

```
myVSprite.Height = 20;  /* 20 lines tall. */
myVSprite.Width  = 2;   /* 24 bits fit into two words. */
```

Because Bobs are drawn into the background playfield, the pixels of the Bob are the same size as the background pixels, and share the color palette of the ViewPort.

1.46 28 / Using Bobs / Specifying the Shape of a Bob

The layout of the data of a Bob's image is different from that of a VSprite because of the way the system retrieves data to draw Bobs. VSprite images are organized in a way convenient to the Sprite hardware; Bob images are set up for easy blitter manipulation. The ImageData pointer is still initialized to point to the first word of the image definition.

Note:

As with all image data, a Bob's ImageData must be in Chip memory for access by the blitter.

The sample image below shows the same image defined as a VSprite in the "Using Virtual Sprites" section above. The data here, however, is laid out for a Bob. The shape is 2 planes deep and is triangular:

```
            <first bitplane data>

  mem        1111 1111 1111 1111    Least significant bit of sprite line 1
  mem + 1    0011 1100 0011 1100    Least significant bit of sprite line 2
  mem + 2    0000 1100 0011 0000    Least significant bit of sprite line 3
  mem + 3    0000 0010 0100 0000    Least significant bit of sprite line 4
  mem + 4    0000 0001 1000 0000    Least significant bit of sprite line 5

            <second bitplane data>

  mem + 5    1111 1111 1111 1111    Most significant bit of sprite line 1
  mem + 6    0011 0000 0000 1100    Most significant bit of sprite line 2
  mem + 7    0000 1111 1111 0000    Most significant bit of sprite line 3
  mem + 8    0000 0011 1100 0000    Most significant bit of sprite line 4
  mem + 9    0000 0001 1000 0000    Most significant bit of sprite line 5

            <more bitplanes of data if Bob is deeper>
```

1.47 28 / Using Bobs / Specifying the Colors of a Bob

Typically a five-bitplane, low-resolution mode display allows playfield pixels (and therefore, Bob pixels) to be selected from any of 32 active colors out of a system palette of 4,096 different color choices. Bob colors are limited to the colors used in the background playfield.

The system ignores the sprColors member of the VSprite structure when the VSprite structure is the root of a Bob. Instead, the Bob's colors are determined by the combination of the Depth of the Bob image and its PlanePick, PlaneOnOff and ImageShadow members.

Use the Depth member in the VSprite structure to indicate how many planes of image data is provided to define the Bob. This also defines how many colors the Bob will have. The combination of bits in corresponding Y,X positions in each bitplane determines the color of the pixel at that position.

For example, if a Depth of one plane is specified, then the bits of that

image allow only two colors to be selected: one color for each bit that is a 0, a second color for each bit that is a 1. Likewise, if there are 5 planes of image data, all 32 colors can be used in the Bob. The Bob Depth must not exceed the background depth. Specify Depth using a statement such as the following:

```
myVSprite.Depth = 5;   /* Allow a 32 color, 5-bitplane image. */
```

1.48 28 / Using Bobs / Other Items Influencing Bob Colors

The three other members in the VSprite structure that affect the color of Bob pixels are ImageShadow, PlanePick, and PlaneOnOff.

ImageShadow PlanePick PlaneOnOff

1.49 28 / / Other Items Influencing Bob Colors / ImageShadow

The ImageShadow member is a pointer to the shadow mask of a Bob. A shadow mask is the logical or of all bitplanes of a Bob image. The system uses the shadow mask in conjunction with PlaneOnOff, discussed below, for color selection. It also uses the shadow mask to "cookie cut" the bits that will be overwritten by this Bob, to save and later restore the background.

The following figure shows the shadow mask of the image described above.

```
mem + 0    1111 1111 1111 1111    Shadow mask for line 1
mem + 1    0011 1100 0011 1100    Shadow mask for line 2
mem + 2    0000 1111 1111 0000    Shadow mask for line 3
mem + 3    0000 0011 1100 0000    Shadow mask for line 4
mem + 4    0000 0001 1000 0000    Shadow mask for line 5
```

Space for the ImageShadow must be provided and this pointer initialized to point to it. The amount of memory needed is equivalent to one plane of the image:

```
shadow_size = myBob->BobVSprite->Height * myBob->BobVSprite->Width;
```

The example image is 5 high and 1 word wide, so, 5 words must be made available.

```
Note:
-----
The ImageShadow memory must be allocated from Chip memory
(MEMF_CHIP).
```

1.50 28 / / Other Items Influencing Bob Colors / PlanePick

Because the Depth of the Bob can be less than the background, the PlanePick member is provided so that the application can indicate which background bitplanes are to have image data put into them. The system starts with the least significant plane of the Bob, and scans PlanePick starting at the least significant bit, looking for a plane of the RastPort to put it in.

For example, if PlanePick has a binary value of: 0 0 0 0 0 0 1 1 (0x03) then the system draws the first plane of the Bob's image into background plane 0 and the second plane into background plane 1.

Alternatively, a PlanePick value of: 0 0 0 1 0 0 1 0 (0x12) directs the system to put the first Bob plane into plane 1, and the second Bob plane into plane 4.

1.51 28 / / Other Items Influencing Bob Colors / PlaneOnOff

What happens to the background planes that aren't picked? The shadow mask is used to either set or clear the bits in those planes in the exact shape of the Bob if OVERLAY is set, otherwise the entire rectangle containing the Bob is used. The PlaneOnOff member tells the system whether to put down the shadow mask as zeros or ones for each plane. The relationship between bit positions in PlaneOnOff and background plane numbers is identical to PlanePick: the least significant bit position indicates the lowest-numbered bitplane. A zero bit clears the shadow mask shape in the corresponding plane, while a one bit sets the shadow mask shape. The planes Picked by PlanePick have image data - not shadow mask - blitted in.

This provides a great deal of color versatility. One image definition can be used for many Bobs. By having different PlanePick / PlaneOnOff combinations, each Bob can use a different subset of the background color set.

There is a member in the VSprite structure called CollMask (the collision mask, covered under "Detecting GEL Collisions") for which the application may also reserve some memory space. The ImageShadow and CollMask pointers usually, but not necessarily, point to the same data, which must be located in Chip memory. If they point to the same location, obviously, the memory only need be allocated once.

An example of the kinds of statements that accomplish these actions (see the makeVSprite() and makeBob() examples for more details):

```
#define BOBW 1
#define BOBH 5
#define BOBD 2

/* Data definition from example layout */
WORD chip BobData[]=
    {
    0xFFFF, 0x300C, 0x0FF0, 0x03C0, 0x0180,
    0xFFFF, 0x3E7C, 0x0C30, 0x03C0, 0x0180
    };
```

```
/* Reserve space for the collision mask for this Bob */
WORD chip BobCollision[BOBW * BOBH];

myVSprite.Width  = BOBW;     /* Image is 16 pixels wide (1 word) */
myVSprite.Height = BOBH;    /* 5 lines for each plane of the Bob */
myVSprite.Depth  = BOBD;     /* 2 Planes are in ImageData */

/* Show the system where it can find the data image of the Bob */
myVSprite.ImageData = BobData;

/* binary 0101, render image data into bitplanes 0 and 2 */
myVSprite.PlanePick = 0x05;

/* binary 0000, means colors 1, 4, and 5 will be used.
*  binary 0010 would mean colors 3, 6, and 7.
*    "    1000   "    "     "    9, C, and D.
*    "    1010   "    "     "    B, E, and F.
*/
myVSprite.PlaneOnOff = 0x00;

/* Where to put collision mask */
myVSprite.CollMask = BobCollision;

/* Tell the system where it can assemble a GEL shadow */
/* Point to same area as CollMask */
myBob.ImageShadow = BobCollision;

/* Create the Sprite collision mask in the VSprite structure */
InitMasks(&myVSprite);
```

1.52 28 / Using Bobs / Bob Priorities

This subsection describes the choices for inter-Bob priorities. The inter-Bob priorities tell the system what order to render the Bobs. Bobs rendered earlier will appear to be behind later Bobs. A Bob drawn earlier is said to have the lower priority and a Bob drawn later is said to have the higher priority. Thus, the highest priority Bob will be drawn last and will never be obstructed by another Bob.

1.53 28 / / Bob Priorities / Letting the System Decide Priorities

The priority issue can be ignored and the system will render the Bobs as it finds them in the GelsInfo list. To do this, set the Bob's Before and After pointers to NULL. Since the GelsInfo list is sorted by GEL x, y values, Bobs that are higher on the display will appear behind the lower ones, and Bobs that are more to the left on the display will appear behind Bobs on the right.

As Bobs are moved about the display, their priorities will change.

1.54 28 / / Bob Priorities / Specifying the Drawing Order

To specify the priorities of the Bobs, use the Before and After pointers. Before points to the Bob that this Bob should be drawn before, and After points to the Bob that this Bob should be drawn after. By following these pointers, from Bob to Bob, the system can determine the order in which the Bobs should be drawn. (Take care to avoid circular dependencies in this list!)

```
Note:
-----
This terminology is often confusing, but, due to historical reasons,
cannot be changed.  The system does not draw the Bobs on the Before
list first, it draws the Bobs on the After list first. Next, it draws
the current Bob, and, finally, the Bobs on the Before list.
```

For example, to assure that myBob1 always appears in front of myBob2, The Before and After pointers must be initialized so that the system will always draw myBob1 after myBob2.

```
myBob2.Before = &myBob1;     /* draw Bob2 before drawing Bob1 */
myBob2.After  = NULL;        /* draw Bob2 after  no other Bob */
myBob1.After  = &myBob2;     /* draw Bob1 after  drawing Bob2 */
myBob1.Before = NULL;        /* draw Bob1 before no other Bob */
```

As the system goes through the GelsInfo list, it checks the Bob's After pointer. If this is not NULL, it follows the After pointer until it hits a NULL. Then it starts rendering the Bobs, going back up the Before pointers until it hits a NULL. Then it continues through the GelsInfo list. So, it is important that all Before and After pointers of a group properly point to each other.

```
Note:
-----
In a screen with a number of complex GELs, you may want to specify
the Before and After order for Bobs that are not in the same
AnimOb.  This will keep large objects together.  If you do not do
this, you may have an object drawn with half of its Bobs in front of
another object!  Also, in sequences you only set the Before and
After pointers for the active AnimComp in the sequence.
```

1.55 28 / Using Bobs / Adding a Bob

To add a Bob to the system GEL list, use the AddBob() routine. The Bob and VSprite structures must be correct and cohesive when this call is made. See the makeBob() and makeVSprite() routines in the animtools.c file listed at the end of this chapter for a detailed example of setting up Bobs and VSprites. See the setupGelSys() function for a more complete example of the initialization of the GELs system.

For example:

```
    struct GelsInfo myGelsInfo = {0};
    struct VSprite dummySpriteA = {0}, dummySpriteB = {0};
    struct Bob myBob = {0};
    struct RastPort rastport = {0};

    /* Done ONCE, for this GelsInfo.  See setupGelSys() at the end of this
    ** chapter for a more complete initialization of the Gel system
    */
    InitGels(&dummySpriteA, &dummySpriteB, &myGelsInfo);

    /* Initialize the Bob members here, then AddBob() */
    AddBob(&myBob, &rastport);
```

1.56 28 / Using Bobs / Removing a Bob

Two methods may be used to remove a Bob. The first method uses the RemBob() macro. RemBob() causes the system to remove the Bob during the next call to DrawGList() (or two calls to DrawGList() if the system is double-buffered). RemBob() asks the system to remove the Bob at the next convenient time. See the description of the BOBSAWAY and BOBNIX flags above. It is called as follows:

```
    struct Bob myBob = {0};

    RemBob(&myBob);
```

The second method uses the RemIBob() routine. RemIBob() tells the system to remove this Bob immediately. For example:

```
    struct Bob      myBob = {0};
    struct RastPort rastport = {0};
    struct ViewPort viewport = {0};

    RemIBob(&myBob, &rastport, &viewport);
```

This causes the system to erase the Bob from the drawing area and causes the immediate erasure of any other Bob that had been drawn subsequent to (and on top of) this one. The system then unlinks the Bob from the system GEL list. To redraw the Bobs that were drawn on top of the one just removed, another call to DrawGList() must be made.

1.57 28 / Using Bobs / Sorting and Displaying Bobs

As with VSprites, the GelsInfo list must be sorted before any Bobs can be displayed. This is accomplished with the SortGList() function. For Bobs, the system uses the position information to decide inter-Bob priorities, if not explicitly set by using the Bob.Before and Bob.After pointers.

Once the GelsInfo list has been sorted, the Bobs in the list can be displayed by calling DrawGList(). This call should then be followed by a

call to WaitTOF() if the application wants to be sure that the Bobs are rendered before proceeding. Call these functions as follows:

```
struct RastPort myRastPort = {0};  /* Of course, these have to be */
struct ViewPort myViewPort = {0};  /* initialized...              */

SortGList(&myRastPort);
DrawGList(&myRastPort, &myViewPort);/* Draw the elements (Bobs only) */
WaitTOF();
```

Warning:

If your GelsInfo list contains VSprites in addition to Bobs, you must also call MrgCop() and LoadView() to make all the GELs visible. Or, under Intuition, RethinkDisplay() must be called to make all the GELs visible.

1.58 28 / Using Bobs / Changing Bobs

The following characteristics of Bobs can be changed dynamically between calls to DrawGList():

* To change the location of the Bob in the RastPort drawing area, adjust the X and Y values in the VSprite structure associated with this Bob.

* To change a Bob's appearance, the pointer to the ImageData in the associated VSprite structure may be changed. Note that a change in the ImageData also requires a change or recalculation of the ImageShadow, using InitMasks().

* To change a Bob's colors modify the PlanePick, PlaneOnOff or Depth parameters in the VSprite structure associated with this Bob.

* To change a Bob's display priorities, alter the Before and After pointers in the Bob structure.

* To change the Bob into a paintbrush, specify the SAVEBOB flag in the Bob.Flags field.

Changes Are Not Immediately Seen.

Neither these nor other changes are evident until SortGList() and then DrawGList() are called.

1.59 28 / Using Bobs / Double-Buffering

Double-buffering is the technique of supplying two different memory areas in which the drawing routines may create images. The system displays one memory space while drawing into the other area. This eliminates the "flickering" that is visible when a single display is being rendered into

at the same time that it is being displayed.

```
    Double-buffering For One Means Double-buffering For All.
    --------------------------------------------------------
    If any of the Bobs is double-buffered, then all of them must be
    double-buffered.
```

To find whether a Bob is to be double-buffered, the system examines the pointer named DBuffer in the Bob structure. If this pointer has a value of NULL, the system does not use double-buffering for this Bob. For example:

```
    myBob.DBuffer = NULL; /* do this if this Bob is NOT double-buffered */
```

DBufPacket and Double-Buffering

1.60 28 / / Double-Buffering / DBufPacket and Double-Buffering

For double-buffering, a place must be provided for the system to store the extra information it needs. The system maintains these data, and does not expect the application to change them. The DBufPacket structure consists of the following members:

BufY, BufX — Lets the system keep track of where the object was located in the last frame (as compared to the Bob structure members called OldY and OldX that tell where the object was two frames ago). BufY and BufX provide for correct restoration of the background within the currently active drawing buffer.

BufPath — Assures that the system restores the backgrounds in the correct sequence; it relates to the VSprite members DrawPath and ClearPath.

BufBuffer — This field must be set to point to a buffer the same size as the Bob's SaveBuffer. This buffer is used to store the background for later restoration when the system moves the object. This buffer must be allocated from Chip memory.

To create a double-buffered Bob, execute a code sequence similar to the following:

```
    struct Bob        myBob = {0};
    struct DBufPacket myDBufPacket = {0};

    /* Allocate a DBufPacket for myBob same size as previous example */
    if (NULL != (myDBufPacket.BufBuffer = AllocRaster(48, 20 * 5)))
        {
        /* tell Bob about its double buff status */
        myBob.DBuffer = myDBufPacket;
        }
```

The example routines makeBob() and freeBob() in the animtools.c listing at the end of this chapter show how to correctly allocate and free a

double-buffered Bob.

1.61 28 / / Collisions and GEL Structure Extensions

This section covers two topics that are applicable to all GELs: how to extend GEL data structures for your own purposes and how to detect collisions between GELs and other graphics objects.

1.62 28 / Collisions and GEL Structure Extensions / Detecting Gel Collisions

All GELs, including VSprites, can participate in the software collision detection features of the graphics library. Simple Sprites must use hardware collision detection. See the Amiga Hardware Reference Manual for information about hardware collision detection.

Two kinds of collisions are handled by the system routines: GEL-to-boundary hits and GEL-to-GEL hits. You can set up as many as 16 different routines to handle different collision combinations; one routine to handle the boundary hits, and up to fifteen more to handle different inter-GEL hits.

You supply the actual collision handling routines, and provide their addresses to the system so that it can call them as needed (when the hits are detected). These addresses are kept in a collision handler table pointed to by the CollHandler field of the GelsInfo list. Which routine is called depends on the 16-bit MeMask and HitMask members of the VSprite structures involved in the collision.

When you call DoCollision(), the system goes through the GelsInfo list which, is constantly kept sorted by x, y position. If a GEL intersects the display boundaries and the GELs HitMask indicates it is appropriate, the boundary collision routine is called. When DoCollision() finds that two GELs overlap, it compares the MeMask of one with the HitMask of the other. If corresponding bits are set in both, it calls the appropriate inter-GEL collision routine at the table position corresponding to the bits in the HitMask and MeMask, as outlined below.

1.63 28 / / Detecting Gel Collisions / Preparing for Collision Detection

Before you can use the system to detect collisions between GELS, you must allocate and initialize a table of collision-detection routines and place the address of the table in the GelsInfo.CollHandler field. This table is

an array of pointers to the actual routines that you have provided for your collision types. You must also prepare some members of the VSprite structure: CollMask, BorderLine, HitMask, and MeMask.

1.64 28 / / Detecting Collisions / Building a Table of Collision Routines

The collision handler table is a structure, CollTable, defined in <graphics/gels.h>. It is accessed as the CollHandler member of the GelsInfo structure. The table only needs to be as large as the number of bits for which you wish to provide collision processing. It is safest, though, to allocate space for all 16 entries, considering the small amount of space required.

Call the routine SetCollision() to initialize the table entries that correspond to the HitMask and MeMask bits that you plan to use. Do not set any of the table entries directly, instead give the address to SetCollision() routine and let it handle the set up of the GelsInfo.CollTable field.

For example, SetCollision() could be called as follows:

```
    ULONG            num;
    VOID           (*routine)();
    struct GelsInfo *GInfo;

    VOID myCollisionRoutine(GELA, GELB)   /* sample collision routine */
    struct VSprite *GELA;
    struct VSprite *GELB;
    {
        /* process gels here - GELA and GELB point to the base VSprites */
        /* of the gels, you can use the user extensions to identify what */
        /* hit (if you need the info). */
    }

    /* GelsInfo must be allocated and initialized */

    routine = myCollisionRoutine;

    SetCollision(num, routine, GInfo)
```

The num argument is the collision table vector number (0-15). The (*routine)() argument is a pointer to your collision routine. And the GInfo argument is a pointer to the GelsInfo structure.

1.65 28 / / Detecting Gel Collisions / VSprite Collision Mask

The CollMask member of the VSprite is a pointer to a memory area allocated for holding the collision mask of that GEL. This area must be in Chip memory and its size is the equivalent of one bitplane of the GEL's image. The collision mask is usually the same as the shadow mask of the GEL, formed from a logical-OR combination of all planes of the image. The following figure shows an example collision mask.

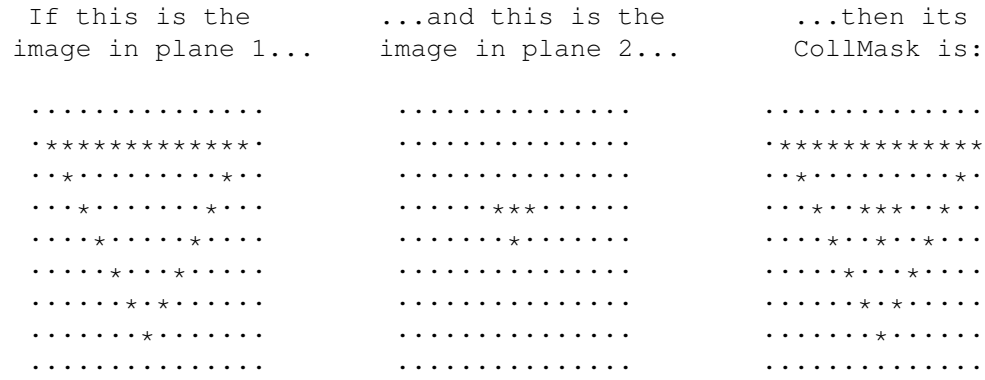


Figure 28-3: A Collision Mask

Alternatively, you may have a collision mask that is not derived from the image. In this case, the actual image isn't relevant. The system will not register collisions unless the other objects touch the collision mask. If the collision mask is smaller than the image, other objects will pass through the edges without a collision.

1.66 28 / / Detecting Gel Collisions / VSprite BorderLine

For faster collision detection, the system uses the BorderLine member of the VSprite structure. BorderLine specifies the location of the horizontal logical-OR combination of all of the bits of the object. It may be compared to taking the whole object's shadow/collision mask and squishing it down into a single horizontal line. You provide the system with a place to store this line. The size of the data area you allocate must be at least as large as the image width.

In other words, if it takes three 16-bit words to hold one line of a GEL, then you must reserve three words for the BorderLine. In the VSprite examples, the routine makeVSprite() correctly allocates and initializes the collision mask and borderline. For example:

```
    WORD myBorderLineData[3];  /* reserve space for BorderLine */
                               /* for this Bob */

    myVSprite.BorderLine = myBorderLineData;  /* tell the system */
                                              /* where it is */
```

Here is a sample of an object and its BorderLine image:

```
            011000001100    Object
            001100011000
            001100011000
            000110110000
            000010100000

            011110111100    BorderLine image
```

Using this squished image, the system can quickly determine if the image is touching the left or rightmost boundary of the drawing area.

To establish default BorderLine and CollMask data, call the InitMasks() function.

1.67 28 / / Detecting Gel Collisions / VSprite HitMask and MeMask

Software collision detection is independently enabled and disabled for each GEL. Further, you can specify which of 16 possible collision routines you wish to have automatically executed. DoCollision(), in addition to sensing an overlap between objects, uses these masks to determine which routine (if any) the system will call when a collision occurs.

When the system determines a collision, it performs a logical-AND of the HitMask of the upper-leftmost object in the colliding pair with the MeMask of the lower-rightmost object of the pair. The bits that are 1s after the logical-AND operation choose which one of the 16 possible collision routines to perform.

* If the collision is with the boundary, bit 0 is always a 1 and the system calls the collision handling routine number 0. Always assign the routine that handles boundary collisions to vector 0 in the collision handling table. The system uses the flag called BORDERHIT to indicate that an object has landed on or moved beyond the outermost bounds of the drawing area (the edge of the clipping region). The VSprite example earlier in this chapter uses collision detection to check for border hits.

* If any one of the other bits (1 to 15) is set, then the system calls your collision handling routine corresponding to the bit set.

* If more than one bit is set in both masks, the system calls the vector corresponding to the rightmost (the least significant) bit only.

1.68 28 / / Detecting Gel Collisions / Using HitMask and MeMask

This section illustrates the use of the HitMask and MeMask to define one type of collision.

Suppose there are two classes of objects that you wish to control on the display: ENEMYTANK and MYMISSILE. Objects of class ENEMYTANK should be able to pass across one another without registering any collisions. Objects of class MYMISSILE should also be able to pass across one another without collisions. However, when MYMISSILE and ENEMYTANK collide, the system should generate a call to a collision routine.

Choose a pair of collision detect bits not yet assigned within MeMask, one to represent ENEMYTANK, the other to represent MYMISSILE. You will use the same two bits in the corresponding HitMask. In this example, bit 1 represents ENEMYTANK objects and bit 2 represents MYMISSLE objects.

```
                    MeMask   HitMask
                    ------   -------
Bit Number           2  1     2  1
--- ------           -  -     -  -
ENEMYTANK 1          0  1     1  0
ENEMYTANK 2          0  1     1  0
MYMISSILE            1  0     0  1
```

In the MeMask, bit 1 is set to indicate that this object is an ENEMYTANK. Bit 2 is clear (zero) indicating this object is not a MYMISSILE object. In the HitMask for ENEMYTANK objects, bit 1 is clear (zero) which means, "I will not register collisions with other ENEMYTANK objects." However, bit 2 is set (one) which means, "I will register collisions with MYMISSILE objects."

Thus when a call to DoCollision() occurs, for any objects that appear to be colliding, the system ANDs the MeMask of one object with the HitMask of the other object. If there are non-zero bits present, the system will call one of your collision routines.

In this example, suppose that the system senses a collision between ENEMYTANK 1 and ENEMYTANK 2. Suppose also that ENEMYTANK 1 is the top/leftmost object of the pair. Here is the way that the collision testing routine performs the test to see if the system will call any collision-handling routines:

```
Bit Number              2  1
--- ------              -  -
ENEMYTANK 1  MeMask     0  1
ENEMYTANK 2  HitMask    1  0

Result of logical-AND   0  0
```

Therefore, the system does not call a collision routine. But suppose that DoCollision() finds an overlap between ENEMYTANK 1 and MYMISSILE, where MYMISSILE is the top/leftmost of the pair:

```
Bit Number              2  1
--- ------              -  -
MYMISSILE MeMask        1  0
ENEMYTANK 2 HitMask     1  0

Result of logical-AND   1  0
```

Therefore, the system calls the collision routine at position 2 in the table of collision-handling routines.

1.69 28 / / Setting Up For Boundary Collisions

To specify the region in the playfield that the system will use to define the outermost limits of the GEL boundaries, you use these GelsInfo members: topmost, bottommost, leftmost, and rightmost. The DoCollision() routine tests these boundaries when determining boundary collisions within this RastPort. They have nothing whatsoever to do with graphical

clipping. Graphical clipping makes use of the RastPort's clipping rectangle.

Here is a typical program segment that assigns the members correctly (for boundaries 50, 100, 80, 240). It assumes that you already have a RastPort structure pointer named myRastPort.

```
    myRastPort->GelsInfo->topmost    = 50;
    myRastPort->GelsInfo->bottommost = 100;
    myRastPort->GelsInfo->leftmost   = 80;
    myRastPort->GelsInfo->rightmost  = 240;
```

Parameters To Your Boundary Collision Routine
Parameters To Your Inter-GEL Collision Routines
Handling Multiple Collisions

1.70 28 / / / Parameters To Your Boundary Collision Routine

During the operation of the DoCollision() routine, if you have enabled boundary collisions for a GEL (by setting the least significant bit of its HitMask) and it has crossed a boundary, the system calls the boundary routine you have defined. The system will call the routine once for every GEL that has hit, or gone outside of the boundary. The system will call your routine with the following two arguments:

* A pointer to the VSprite structure of the GEL that hit the boundary

* A flag word containing one to four bits set, representing top, bottom, left and right boundaries, telling you which of the boundaries it has hit or exceeded. To test these bits, compare to the constants TOPHIT, BOTTOMHIT, LEFTHIT, and RIGHTHIT.

See the VSprite example given earlier for an example of using boundary collision.

1.71 28 / / / Parameters To Your Inter-GEL Collision Routines

If, instead of a GEL-to-boundary collision, DoCollision() senses a GEL-to-GEL collision, the system calls your collision routine with the following two arguments:

* Address of the VSprite that is the uppermost (or leftmost if y coordinates are identical) GEL of a colliding pair.

* Address of the VSprite that is the lowermost (or rightmost if y coordinates are identical) GEL of the pair.

1.72 28 / / Set Up For Boundary Collisions / Handling Multiple Collisions

When multiple elements collide within the same display field, the following set of sequential calls to the collision routines occurs:

* The system issues each call in a sorted order for GELs starting at the upper left-hand corner of the screen and proceeding to the right and down the screen.

* For any colliding GEL pair, the system issues only one call, to the collision routine for the object that is the topmost and leftmost of the pair.

1.73 28 / / Adding User Extensions To Gel Data Structures

This section describes how to expand the size and scope of the VSprite, Bob and AnimOb data structures. In the definition for these structures, there is an item called UserExt at the end of each. If you want to expand these structures (to hold your own special data), you define the UserExt member before the <graphics/gels.h> file is included. If this member has already been defined when the <graphics/gels.h> file is parsed, the compiler preprocessor will extend the structure definition automatically. If these members have not been defined, the system will make them SHORTs, and you may still consider these as being reserved for your private use.

To show the kind of use you can make of this feature, the example below adds speed and acceleration figures to each GEL by extending the VSprite structure. When your collision routine is called, it could use these values to transfer energy between the two colliding objects (say, billiard balls). You would have to set up additional routines, executed between calls to DoCollision(), that would add the values to the GELs position appropriately. You could do this with code similar o the following:

```
struct myInfo
    {
    short xvelocity;
    short yvelocity;
    short xaccel;
    short yaccel;
    };
```

These members are for example only. You may use any definition for your user extensions. You would then also provide the following line, to extend the VSprites structure, use:

```
/* Redefine VUserStuff for my own use. */
#define VUserStuff struct myInfo
```

To extend the Bobs structure, use:

```
#define BUserStuff struct myInfo
```

To extend the AnimObs structure, use:

```
#define AUserStuff struct myInfo
```

When the system is compiling the <graphics/gels.h> file with your program, the compiler preprocessor substitutes "struct myInfo" everywhere that UserExt is used in the header. The structure is thereby customized to include the items you wish to associate with it.

```
Typedef Cannot Be Used.
-----------------------
```
You cannot use the C-language construct typedef for the above statements. If you want to substitute your own data type for one of the UserStuff variables, you must use a #define.

1.74 28 Graphics Sprites, Bobs and Animation / Animation with GELs

An animation sequence is composed of a series of drawings. Each drawing differs from the preceding one so that when they are arranged in a stack and viewed sequentially, the images appear to flow naturally.

In classic film animation, image drawing is done in two stages. The background for each scene is painted just once. Then, the cartoon characters and any other foreground objects are painted on transparent sheets of celluloid called cells which are placed over the background. With cells, animation can be achieved by redrawing only the parts of the scene that move while the background stays the same. Animation on the Amiga works similarly. The background is formed by the playfield while the objects that move can be conveniently handled with the GELs system.

1.75 28 / Animation with GELs / Animation Data Structures

There are two main data structures involved in Amiga animation: AnimComp and AnimOb.

The AnimComp (Animation Component), is an extension of the Bob structure discussed in the previous section. An AnimComp provides a convenient way to link together a series of images so that they can be sequenced automatically, and so multiple sequences can be grouped together. An AnimComp is analogous to one sheet of celluloid representing a single image to be placed over the background.

```
struct AnimComp
{
    WORD Flags;                 /* AnimComp flags for system & user */
    WORD Timer;
    WORD TimeSet;
    struct AnimComp  *NextComp;
    struct AnimComp  *PrevComp;
```

```
        struct AnimComp  *NextSeq;
        struct AnimComp  *PrevSeq;
        WORD (*AnimCRoutine)();
        WORD YTrans;
        WORD XTrans;
        struct AnimOb    *HeadOb;
                           /* Pointer back to the controlling AnimOb */
        struct Bob       *AnimBob;
                           /* Underlying Bob structure for this AnimComp */
    };
```

The AnimComp structure contains pointers, PrevSeq and NextSeq, that lets you group these cells into stacks that will be viewed sequentially. The AnimComp structure also has PrevComp and NextComp pointers that let you group stacks into complex objects containing multiple independently moving parts.

The second animation data structure is the AnimOb which provides the variables needed for overall control over a group of AnimComps. The AnimOb itself contains no imagery; it simply provides a common reference point for the sequenced images and specifies how the system should move that point.

```
    struct AnimOb
    {
        struct AnimOb    *NextOb, *PrevOb;
        LONG Clock;
        WORD AnOldY, AnOldX;            /* old y,x coordinates          */
        WORD AnY, AnX;                  /* y,x coordinates of the AnimOb*/
        WORD YVel, XVel;                /* velocities of this object    */
        WORD YAccel, XAccel;            /* accelerations of this object */
        WORD RingYTrans, RingXTrans;    /* ring translation values      */
        WORD (*AnimORoutine)();         /* address of user procedure    */
        struct AnimComp  *HeadComp;     /* pointer to first component   */
        AUserStuff AUserExt;            /* AnimOb user extension        */
    };
```

These structures can be used in various ways. A simple animation of a rotating ball could be created with three or four AnimComps linked together in a circular list by their NextSeq and PrevSeq fields. The system displays the initial AnimComp (the "top of the stack"), then switches to the AnimComp pointed to by NextSeq, and then switches to its NextSeq and so on until it reaches the end of the sequence. The sequence starts over again automatically if the last AnimComp.NextSeq points back to the first AnimComp in the stack.

For a more complex animation of a walking human, you could use five stacks, i.e., five circular lists of AnimComps; four stacks for the arms and legs and a single stack for the head and torso. To group these stacks into one cohesive unit showing a human figure, you use the PrevComp and NextComp pointers in the AnimComp structure. All the stacks would also share a common AnimOb, so that the combined sequences can be moved as a single object.

Figure 28-4: Linking AnimComps For a Multiple Component AnimOb

1.76 28 / Animation with GELs / Animation Types

The GELs system provides several ways of setting up automatic animation, loosely based on some categories of movement in real life. Some things (like balls or arrows) can move independently of the background, and look even more realistic if they tumble or rotate as they move; other things (like worms, wheels, and people) must be anchored to the background, or they will appear to slide unnaturally.

The system software allows these types of animation through simple motion control, motion control with sequenced drawing, and sequenced drawing using Ring Motion Control.

Simple Motion Control Sequenced Drawing Ring Motion Control

1.77 28 / / Animation Types / Simple Motion Control

To produce motion of a simple object, such as a ball, the object is simply moved relative to a background display, a little at a time. This is simple motion control, and can be accomplished with one AnimComp and one AnimOb, by simply changing the AnimOb's position every N video frames. The apparent speed of the object is a combination of how often it is moved (every frame, every other frame, etc.) and how far it is moved (how much the AnimOb's AnX and AnY are changed).

1.78 28 / / Animation Types / Sequenced Drawing

To make the ball appear to rotate is a little more complex. To produce apparent movement within the image, sequencing is used. This is done by having a stack of AnimComp's that are laid down one after the other, a frame at a time. The stack can be arranged in a circular list for repeating movement. So, when you combine a sequence of drawings using AnimComps with simple motion control using an AnimOb, you can perform more complex animations such as having a rotating ball bounce around.

1.79 28 / / Animation Types / Ring Motion Control

Making a worm appear to crawl is similar to the rotating ball. There is still a stack of AnimComps that are sequenced automatically, and one controlling AnimOb. But each AnimComp image is drawn so that it appears to move relative to an internal point that remains stationary throughout the stack. So instead of the AnimOb's common reference point moving in each frame, you tell the system how far to move only at the end of each AnimComp sequence.

Figure 28-5: Ring Motion Control

As illustrated in the figure at left, the sequence of events for Ring Motion Control look like this:

```
    Draw AnimComp1, Draw AnimComp2, Draw AnimComp3, Move AnimOb,
    Draw AnimComp1, Draw AnimComp2, Draw AnimComp3, Move AnimOb,
    Draw AnimComp1...
```

1.80 28 / Animation with GELs / Specifying Animation Components

For each AnimComp, you initially specify:

* A pointer to the AnimComp's controlling AnimOb.

* Initial and alternate views, their timing and order.

* The initial inter-component drawing priorities (for multiple AnimComp sequences, this specifies which sequence to display frontmost).

* A pointer to a special animation routine related to this component (optional).

* Your own extensions to this structure (optional).

Sequencing AnimComps
Position of an AnimComp
Specifying Time for Each Image
Linking Multiple AnimComp Sequences
Component Ordering

1.81 28 / / Specifying Animation Components / Sequencing AnimComps

To specify the sequencing of AnimComp images, the pointers called PrevSeq and NextSeq are used to build a doubly-linked list. The sequence can be made circular (and usually is) by linking the first and last AnimComps in the sequence: the NextSeq of the last AnimComp must point back to the first AnimComp, and the PrevSeq of the first AnimComp must point to the last AnimComp. If the list is a loop, then the system will continue to cycle through the list until it is stopped. If the list is not a loop, then the program must act to restart the sequence after the last item is displayed. The AnimCRoutine field of the last AnimComp can be used to do this.

1.82 28 / / Specifying Animation Components / Position of an AnimComp

To specify the placement of each AnimComp relative to its controlling AnimOb, you set the AnimComp members XTrans and YTrans. These values can be positive or negative.

The system is designed so that only one of the AnimComps in any given sequence is "active" (being displayed) at a given point in time. It is the only image in the sequence that is (or is about to be) linked into the GelsInfo list. The Timer determines how long each Component in the sequence remains active, as described below.

1.83 28 / / Animation Components / Specifying Time for Each Image

The AnimComp members Timer and TimeSet are used to specify how long the system should keep each sequential image on the screen.

When the system makes an animation component active, it copies the value you have put in the TimeSet member into the Timer member. As the animation proceeds, the system decrements Timer; as long as it is greater than zero, then that AnimComp remains active. When the Timer value reaches zero, the system makes the next AnimComp in the sequence active, and the process repeats.

If you initialize the value in TimeSet to zero, the system will not sequence this component at all (and Timer will remain zero).

1.84 28 / / Animation Components / Linking Multiple AnimComp Sequences

When an AnimOb is built from multiple AnimComp sequences, the sequences are linked together by the the PrevComp and NextComp fields of the AnimComps. These pointers must be initialized only in the initial AnimComp of each sequence. The other components that are not initially active should have their PrevComp and NextComp pointers set to NULL.

    Do Not Use Empty Fields.

    You cannot store data in the empty PrevComp and NextComp fields. As the system cycles through the AnimComps, the NextComp and PrevComp fields are set to NULL when an old AnimComps is replaced by a new AnimComp. The new AnimComp is then linked in to the list of sequences in place of the old one.

1.85 28 / / Specifying Animation Components / Component Ordering

The PrevSeq, NextSeq, PrevComp and NextComp linkages have no bearing on the order in which AnimComps in any given video frame are drawn. To specify the inter-component priorities (so that the closest objects appear frontmost) the Before and After pointers in the initially active AnimComp's underlying Bob structure are linked in to the rest of the system, as described previously in the discussion of Bobs.

This setup needs to be done once, for the initially active AnimComps of the AnimOb only.

The animation system adjusts the Before and After pointers of all the underlying Bob structures to constantly maintain the inter-component drawing sequence, even though different components are being made active as sequencing occurs.

These pointers also assure that one complete object always has priority over another object. The Bob Before and After pointers are used to link together the last AnimComp's Bob of one AnimOb to the first AnimComp's Bob of the next AnimOb.

1.86 28 / Animation with GELs / Specifying the Animation Object

For each AnimOb, you initially specify:

* The starting position of this object

* Its velocity and acceleration (optional).

* A pointer to the first of its AnimComps.

* A pointer to a special animation routine related to this object (optional).

* Your own extensions to this structure (optional).

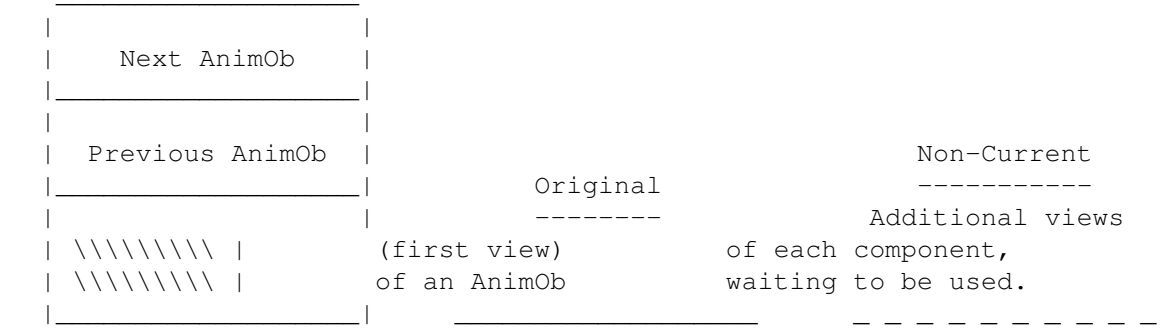
1.87 28 / / / Linking the AnimComp Sequences to the AnimOb

Within each AnimOb there may be one or more AnimComp sequences. The HeadComp of the AnimOb points to the first AnimComp in the list of sequences.

Each sequence is identified by its "active" AnimComp. There can only be one active AnimComp in each sequence. The sequences are linked together by their active AnimComps; for each of these the NextComp and PrevComp fields link the sequences together to create a list. The first sequence in the list (HeadComp of the AnimOb), has its PrevComp set to NULL. The last sequence in the list has its NextComp set to NULL. None of the inactive AnimComps should have NextComp or PrevComp fields set.

To find the active AnimComp at run time, you can look in the AnimOb's HeadComp field. To find the active AnimComp from any another AnimComp, use the HeadOb field to find the controlling AnimOb first and then look in its HeadComp field to find the active AnimComp.

The figure below shows all the linkages in data structures needed to create the animation GELs.



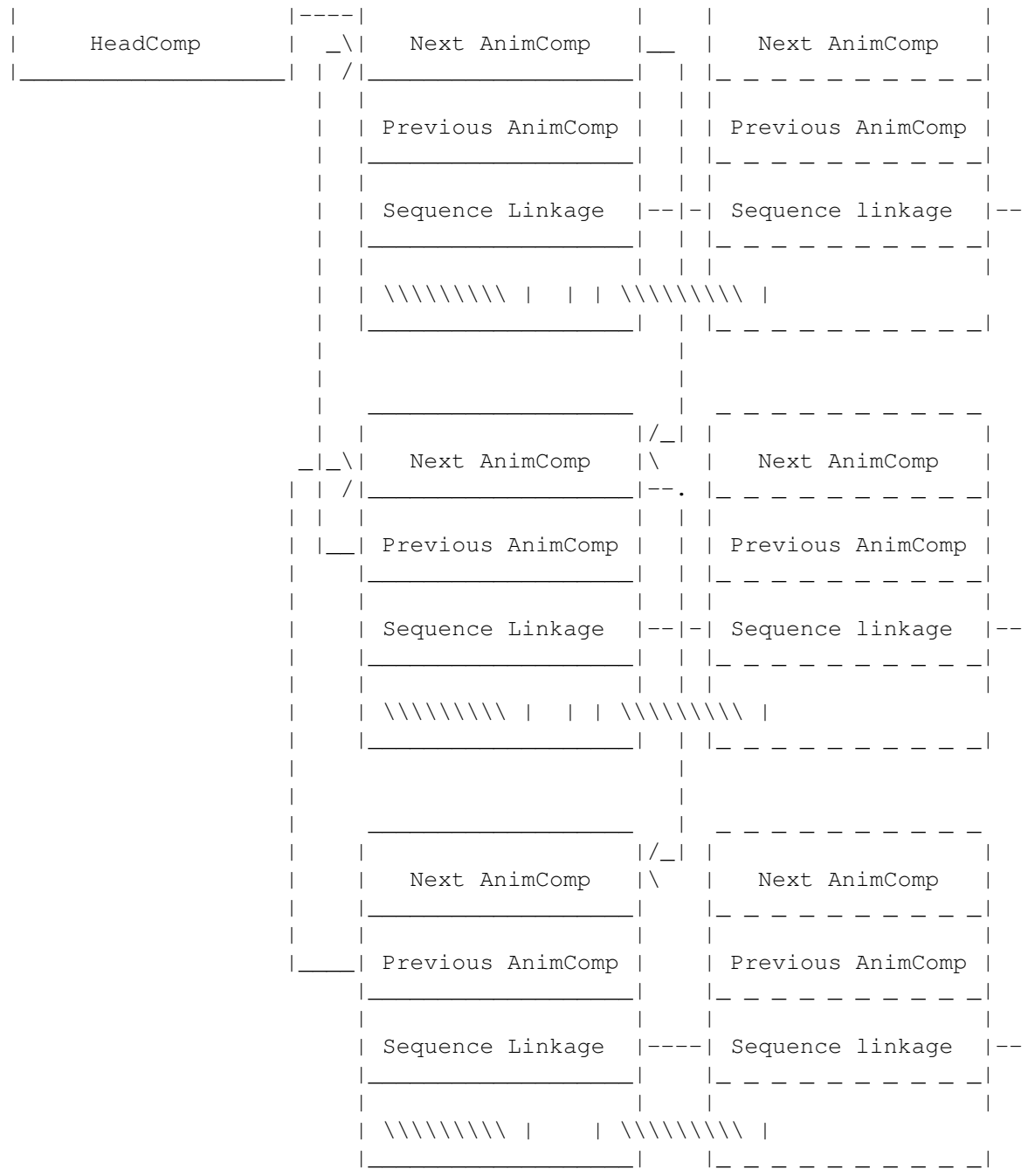


Figure 28-6: Linking of an AnimOb

1.88 28 / / Specifying the Animation Object / Position of an AnimOb

To position the object and its component parts, use the AnimOb structure members AnX and AnY. The following figure illustrates how each component has its own offset from the AnimOb's common reference point.

Figure 28-7: Specifying an AnimOb Position

When you change the animation object's AnX and AnY, all of the component

parts will be redrawn relative to it the next time DrawGList() is called.

1.89 28 / / Specifying Animation Object / Setting Up Simple Motion Control

In this form of animation, you can specify objects that have independently controllable velocities and accelerations in the X and Y directions. Components can still sequence.

The variables that control this motion are located in the AnimOb structure and are called:

* YVel, XVel--the velocities in the y and x directions. These values are added to the position values on each call to Animate() (see below).

* YAccel, XAccel--the accelerations in the xa nd y directions. These values are added to the velocity values on each call to Animate() (see below). The velocity values are updated before the position values.

1.90 28 / / Specifying Animation Object / Setting Up Ring Motion Control

To make a given component trigger a move of the AnimOb you set the RINGTRIGGER bit of that AnimComp's Flags field. When the system software encounters this flag, it adds the values of RingXTrans and RingYTrans to the AnX and AnY values of the controlling AnimOb. The next time you execute DrawGList(), the drawing sequence will use the new position.

You usually set RINGTRIGGER in only one of the animation components in a sequence (the last one); however, you can use this flag and the translation variables any way you wish.

1.91 28 / / / Using Sequenced Drawing and Motion Control

If you are using Ring Motion Control, you will probably set the velocity and acceleration variables to zero. For instance, consider the example of a person walking. With Ring Motion Control, as each foot falls it is positioned on the ground exactly where originally drawn. If you included a velocity value, the person's foot would not be stationary with respect to the ground, and the person would appear to "skate" rather than walk. If you set the velocity and acceleration variables at zero, you avoid this problem.

When the system activates a new AnimComp, it checks the Flags field to see if the RINGTRIGGER bit is set. If so, the system adds RingYTrans and RingXTrans to AnY and AnX respectively.

1.92 28 / Animation with GELs / The AnimKey

The system uses one pointer, known as the AnimKey, to keep track of all the AnimObs via their PrevOb and NextOb linkage fields. The AnimKey acts as the anchor for the list of AnimObs you are using and is initialized with code such as the following:

```
struct AnimOb *animKey;

InitAnimate(&animKey);
    /* Only do this once to initialize the AnimOb list */
```

As each new object is added (via AddAnimOb()), it is linked in at the beginning of the list, so AnimKey will always point to the object most recently added to the list. To search forward through the list, start with the AnimKey and move forward on the NextOb link. Continue to move forward until the NextOb is NULL, indicating the end of the list. The PrevOb link will allow you to move back to a previous object.

Set Up PrevOb and NextOb Correctly.

It is important that the NextOb link of the last object is NULL, and that the PrevOb of the first object is NULL. In fact, the system expects the animation object lists to be exactly the way that they are described above. If they are not, the system will produce unexpected results.

1.93 28 / Animation with GELs / Adding Animation Objects

Use the routine AddAnimOb() to add animation objects to the controlled object list. This routine will link the PrevOb and NextOb pointers to chain all the AnimObs that the system is controlling.

```
struct RastPort myRPort;
struct AnimOb myAnimOb;
struct AnimOb *animKey;  /* Must be initialized with InitAnimate() */

AddAnimOb(&myAnimOb, &animKey, &myRPort);
```

1.94 28 / Animation with GELs / Moving the Objects

When you have defined all of the structures and have established all of the links, you can call the Animate() routine to move the objects. Animate() adjusts the positions of the objects as described above, and calls the various subroutines (AnimCRoutines and AnimORoutines) that you have specified.

After the system has completed the Animate() routine, some GELs may have been moved, so the GelsInfo list order may possibly be incorrect. Therefore, the list must be re-sorted with SortGList() before passing it to a system routine.

If you are using collision detection, you then perform DoCollision(). Your collision routines may also have an effect on the relative position of the GELs. Therefore, you should again call SortGList() to assure that the system correctly orders the objects before you call DrawGList(). When you call DrawGList(), the system renders all the GELs it finds in the GelsInfo list and any changes caused by the previous call to Animate() can then be seen.

This is illustrated in the following typical call sequence:

```
struct AnimOb **myAnimKey;
struct RastPort *rp;
struct ViewPort *vp;

/* ... setup of graphics elements and objects */

Animate(myAnimKey, rp);       /* "move" objects per instructions */
SortGList(rp);                /*  put them in order */
DoCollision(rp);              /*  software collision detect/action */
SortGList(rp);                /*  put them back into right order */
DrawGList(vp, rp);            /*  draw into current RastPort */
```

1.95 28 / Animation with GELs / Your Own Animation Routine Calls

The AnimOb and AnimComp structures can include pointers for your own routines that you want the system to call. These pointers are stored in the AnimOb's AnimORoutine field and in the AnimComp's AnimCRoutine field, respectively.

When Animate() is called, the system performs the following steps for every AnimOb in the AnimKey list:

* Updates the AnimOb's location and velocities.

* Calls the AnimOb.AnimORoutine routine if one is supplied.

* The for each AnimComp of the AnimOb:

    · If this sequence times out, switches to the new AnimComp.

    · Calls the AnimComp.AnimCRoutine if one is supplied.

    · Sets the underlying VSprite's x,y coordinates.

If you want a routine to be called, you put the address of the routine in either AnimComp.AnimCRoutine or AnimOb.AnimORoutine member as needed. If no routine is to be called, you must set these fields to NULL. Your routines will be passed one parameter, a pointer to the AnimOb or AnimComp it was related to. You can use the user structure extensions discussed earlier to hold the variables you need for your own routines.

For example, if you provide a routine such as this:

```
VOID MyOCode(struct AnimOb *anOb)
{
```

```
    /* whatever needs to be done */
    }
```

Then, if you put the address of the routine in an AnimOb structure:

```
    myAnimOb.AnimORoutine = MyOCode;
```

MyOCode() will be called with the address of this AnimOb when Animate() processes this AnimOb.

1.96 28 / Animation with GELs / Standard Gel Rules Still Apply

Before you use the animation system, you must have called the routine InitGels(). The section called "Bob Priorities" describes how the system maintains the list of GELs to draw on the screen according to their various data fields. The animation system selectively adds GELs to and removes GELs from this list of screen objects during the Animate() routine. On the next call to DrawGList(), the system will draw the GELs in the list into the selected RastPort.

1.97 28 / Animation with GELs / Animations Special Numbering System

Velocities and accelerations can be either positive or negative. The system treats the velocity, acceleration and Ring values as fixed-point binary fractions, with the decimal point at position 6 in the word. That is: vvvvvvvvvv.ffffff where v stands for actual values that you add to the x or y (AnX, AnY) positions of the object for each call to Animate(), and f stands for the fractional part. By using a fractional part, you can specify the speed of an object in increments as precise as 1/64th of an interval.

If you set the value of XVel at 0x0001, it will take 64 calls to the Animate() routine before the system will modify the object's x coordinate position by a step of one. The system constant ANFRACSIZE can be used to shift values correctly. So if you set the value to (1 << ANFRACSIZE), it will be set to 0x0040, the value required to move the object one step per call to Animate(). The system constant ANIMHALF can be used if you want the object to move every other call to Animate().

Each call you make to Animate() simply adds the value of XAccel to the current value of XVel, and YAccel to the current value of YVel, modifying these values accordingly.

1.98 28 / Animation with GELs / Animtools.h and Animtools.c

Here is the listing of the animtools.h header file and the animtools.c link file used by the examples in this chapter. The makeSeq() and makeComp() subroutines here demonstrate how to use the GELs animation system.

```
     animtools.h          animtools.c
```

1.99 28 Graphics Sprites, Bobs and Animation / Function Reference

The following are brief descriptions of the Amiga's graphics animation functions. See the Amiga ROM Kernel Reference Manual: Includes and Autodocs for details on each function call.

Table 28-1: Graphics Animation Functions

Animation Function	Description
AddAnimOb()	Add an AnimOb to the linked list of AnimObs.
AddBob()	Add a Bob to the current gel list.
AddVSprite()	Add a VSprite to the current gel list.
Animate()	Process every AnimOb in the current animation list.
ChangeSprite()	Change the sprite image pointer.
DoCollision()	Test every gel in gel list for collisions.
DrawGList()	Process the gel list, queueing VSprites, drawing Bobs.
FreeGBuffers()	Deallocate memory obtained by GetGBuffers().
FreeSprite()	Return sprite for use by others and virtual sprite machine.
GetGBuffers()	Attempt to allocate all buffers of an entire AnimOb.
GetSprite()	Attempt to get a sprite for the simple sprite manager.
InitGels()	Initialize a gel list; must be called before using gels.
InitGMasks()	Initialize all of the masks of an AnimOb.
InitMasks()	Initialize the BorderLine and CollMask masks of a VSprite.
MoveSprite()	Move sprite to a point relative to top of ViewPort.
RemBob()	Remove a Bob from the gel list.
RemIBob()	Immediately remove a Bob from the gel list and the RastPort.
RemVSprite()	Remove a VSprite from the current gel list.
SetCollision()	Set a pointer to a user collision routine.
SortGList()	Sort the current gel list, ordering its y,x coordinates.