Sprite Animation Toolkit

by Ingemar Ragnemalm

A programmer's library for making sprite-based animation (especially games).
For Think Pascal or Think C on the Macintosh.

Version 2. (color version)

Contents

1. Introduction

Foreword

This is Sprite Animation Toolkit, hereafter refered to as SAT. SAT is intended for novice to intermediary level programmers who want to make animations on the Macintosh, especially arcade games with animation over a background. Since the Mac does not have any hardware sprites, the creation of such games is not a trivial task. This package is intended to relieve the non-expert of the burden of re-inventing all the tricks that have to be used, and to provide a library that makes development easy.

The package has evolved out of my own needs when making games, so it is made from a game makers point of view. It has so far resulted in six *released* games: **Slime Invaders**, **Bachman**, **HeartQuest** and **Ingemar's Skiing Game** by myself, **Missions Of The Reliant** by Mike Rubin, **Asterax** by Michael Hanson and **Space Invaders** by Bettini Simone, and there are several more coming.

The strongest points with SAT, compared to other packages, are:

– Several demos with different complexity ranging from trivial, very easily understood examples to a complete arcade game, in both Pascal and C. (HeartQuest and Collision /// are only in Pascal so far.) Considering that Think Pascal still has the best debugging system for the Mac, this makes it considerably more attractive for beginners than a C package.

– Direct-to-screen (fast) drawing in b/w, 16 and 256 colors.

– An easy-to-use sound module which switches to Sound Driver if Sound Manager is not available, and that includes workarounds for the bugs in Sound Manager (before version 3).

– A simple programming interface, which makes simple games as simple as they should be, with advanced calls to switch to when the defaults are not what you want. Uses 'cicn' resources by default - which is generally the simplest source for the programmer - but you can use any way you like to draw them.

– Simplifies support for b/w as well as color.

All in all, a complete toolkit for game making – and it is free of charge for non-professional use.

This document describes version 2, the color version of SAT, which is a major revision of the black-and-white SAT from spring 1992. This new version has a simpler interface than the old version, using fewer calls to accomplish more, and adds a bunch of utility functions and options.

Legal terms

This package (the SAT package) consists of this manual, the SAT library itself (Pascal and C versions) plus resource files, project files and source code to the example programs *SATminimal*, *Collision, Collision][, Collision ///, MyPlatform, Zkrolly, SAT Invaders* and *HeartQuest* (all in Pascal versions, most of them in C versions).

This package is **free of charge** when used for freely distributable products (public domain, freeware or shareware).

With the exception of compiled shareware programs using SAT, no part of the SAT package may be sold for profit without my written permission. Commercial shareware distributors should ask for permission before including it in their distribution.

If you use SAT to produce a game or program that is distributed as Public Domain, freeware, shareware, or similar conditions you should send me a free copy and mention SAT in the doc and/or "About" box.

If you use SAT to produce a commercial program, you must have my written permission.
My internet address is ingemar@lysator.liu.se, and my real address is:

Ingemar Ragnemalm
Plöjaregatan 73
S-58330 Linköping
SWEDEN

Standard disclaimer: The package is delivered "as is". I don't take any responsibility for damage, loss of data etc that may occur from using it.

Background

I have always liked to make computer games. It has been one of my hobbies since the late 70's.When I started using Macs, of course I wanted to make some games for it too. Among the games I liked were games like Glider, Zero Gravity and Cairo Shootout: shareware games with nice, smooth animation over a detailed background. After occasional hacking over many weekends, using code fragments from comp.sys.mac.programmer, I got a horse race game working (never released), and later a Space Invaders-style game, which has been released as freeware as *Slime Invaders*, then a downhill skiing game (*ISG*, which was not until recently polished enough to be complete) and more recently a Pacman game (*Bachman*) and a game for my wife (*HeartQuest*). Now, I think the routines I'm using have become good enough to distribute, to help others make nice games.

After all, they say that there are too few good games for the Mac, and this way I might help some programmers get started. Perhaps this can help you save time that you can spend on making your games *interesting* rather than just make all the animation and sound code work.

What you need

To use SAT, you need **a Mac** – any Mac with enough memory to run your development system, though a color Mac is preferrable – and a development system, which can be **Think Pascal** version 4, **Think C** version 5 or **Symantec C++** version 6. (I'll make a CodeWarrior version when I get the time.)

You also need a bit of curiosity, creativity and patience. Even with the services SAT provides, making a good, complete game is far from trivial, and the final touches take surprisingly much time. Hopefully, the features in SAT combined with the game-related stuff in the demos (esp. HeartQuest, since that is a complete game) will help you on the way there.

Features and limitations

The features of SAT have evolved from needs in my own game making projects (Slime Invaders, Bachman, Ingemars Skiing Game, HeartQuest…). The ambition has consequently been to relieve the game/animation programmer from as many troublesome issues as possible, hiding compatibitity issues and complicated drawing sequences, thereby making it easier to make games that are both fast and compatible. (SAT works under both system 6 and 7, with or without Color QuickDraw, and has been tested on most Mac models.) The programmers interface was made primarily to be simple and easy to use. Many SAT programs can manage with only a few basic calls. More flexible calls are also provided.

SAT produces flicker-free **animation with sprites** over a background. As implied above, the goal was to produce animation of the quality we find in games like Glider (by John Calhoun) and Cairo Shootout (by the late Duane Blehm). The main problem SAT solves is drawing, the sprite animation, but it also has a bunch of other features like **asynchronous sound** (), other drawing facilities and some miscellaneous utilities. In the demos, you can also find **other game-related functions** like high score list management.
The drawing routines give you the option to draw directly to the screen (fastest) or with QuickDraw (safest). With the faster routines, the game can animate a decent number of sprites (let's say a dozen or so) even on the oldest Macs.
SAT supports b/w and color graphics, using QuickDraw in any depth or **direct-to-screen graphics** in 1, 4 and 8 bits (that is 2, 16 and 256 colors), wp in translating the demos to C, and to Nathaniel Woods, for suggesting several good enhancements and for finding a rather serious bug.

n't. (If you don't know what it is, don't worry about it.)

Library source code is available separately, directly from me (only!). You only need it if you want to make changes, or port it to, say, PowerPC? You get the sources – for personal use only – for $20 (or equivalent). Note that this is a service for those of you who must have the source code for whatever reason, and I don't guarantee anything about the usefulness or style of the code. If you only use SAT as a library (again, for making freeware or shareware) – which I recommend – you don't have to pay me anything.
Updates to SAT will occasionally be distributed on USENET. I can snail-mail updates for $10. I prefer if you can handle that over the net, though.
Example programs
Seven example programs are included in SAT, namely "SATminimal", "Collision", "Collision][", "Collision ///", "MyPlatform", "SAT Invaders" and "HeartQuest".
"**SATminimal**" is extremely simple, making a trivial animation until the user clicks the mouse. "**Collision**" adds the simplest collision handling. "**Collision][**" demonstrates a more flexible collision handling plus simple background manipulation.

"**Collision ///**" is very different. It demonstrates some not too well known options: creates sprite faces on-line, using QuickDraw calls, uses a moveable window without any borders, does collision detection using the mask regions of the sprite faces, and more. Also, I intentionally break some of my own conventions (all code in one file, sprites are set up in the code that creates them instead of in separate setup procedures…), just to show that you may break them if you feel like it. As a game, the demo is poor, and needs improvement, but I think it demonstrates what it's supposed to anyway.
"**MyPlatform**" demonstrates one fairly simple way to make platform games with SAT. The player sprite isn't perfect - most of all, it jumps too high. I might improve it in a later version.
"**SAT Invaders**" is somewhat more elaborate, a stripped down Space Invaders. It uses the TransSkel library to get menus and event handling.
"**HeartQuest**" is the biggest example, a complete arcade game with scores, various settings and high score list. You can find lots of useful stuff in it, like preference file handing, on-line generation of dithered backgrounds and more.
Most of the demos are pretty ugly, quick, unpolished hacks, graphic-wise, but the *artwork* is not the problem SAT is designed to solve for you. Generally, I have avoided beautiful backdrops only to keep the size of the package down to a reasonable level.
I believe that examples should be simple enough, so it should be possible to understand all of the code with reasonable effort. Start with SATminimal and Collision to get the hang of it (and complain to me if you don't understand).
In the list below, C files are in parenthesis together with corresponding Pascal files.

<u>SATminimal source files:</u>

SATminimal.proj, SATminimal.π.rsrc (SATminimal.π)
Project file and resource file.

sMySprite.p (sMySprite.c)
A sprite unit.

SATminimal.p (SATminimal.c)
The main program, which initializes SAT and the sMySprite sprite, and runs the animation.

<u>Collision source files:</u>

Collision.proj, Collision.π.rsrc (Collision.π)
Project file and resource file.

sMrEgghead.p, sApple.p
Two sprite units, one defining "Mr Egghead" and the other the apples.

Collision.p
Main program.

<u>Collision][source files:</u>

Collision][.proj, Collision][.π.rsrc,
sMrEgghead][.p, sApple][.p, Collision][.p
(C files: Collision][.π, sMrEgghead][.c, sApple][.c, Collision][.h)
See Collision. Collision][is slightly bigger, adding some features and icons.

Collision/// source files:

Collision///.π, Collision///.rsrc
Project file and resource file.

Collision///.p
The entire source in one fairly big source file. (This breaks my convention to encapsulate sprites as far as possible in their own units, but was done just to show you that you have the freedom.)

MyPlatform source files:

myPlatform.π.rsrc, myPlatform.proj (myPlatform.π)
Resource and project files.

myPlatform.p (myPlatform.c, myPlatform.h)
Main program (and C header file).

sPlatform.p (sPlatform.c)
Sprite unit, defining static platforms as "invisible sprites".

sMovPlatform.p, sHMovPlatform.p (sMovPlatform.c, sHMovPlatform.c)
Two sprite units defining platforms moving vertically and horizontally.

sPlayerSprite.p (sPlayerSprite.c)
The sprite unit defining the player. (This one could be improved a lot!)

SAT Invaders source files:

SAT Invaders.π, SAT Invaders.π.rsrc
Project file and resource file.

gameGlobals.p
Global variables and resource numbers.

soundConst.p
Sound resource numbers and their handles.

sMissile.p, sEnemy.p, sShot.p, sPlayer.p
Four sprite units.

main.p
Main program. Game window handling, game driver, initializations...

(Also in C.)

HeartQuest source files:

HeartQuest.π, HeartQuest.π.rsrc
Project file and resource file.

gameGlobals.p
Global variables and resource numbers.

soundConst.p
Sound resource numbers and their handles.

scores.p
Score and high score list handling.

sPoints.p, sHeart.p, sBonus.p, sFlypaper.p, sPlayer.p
Five sprite units.

gameWindow.p
Handlers for the game window. Most of the game driver is here. RunSAT is called from this file.

main.p
The main program, mostly initializations.

(We are working on the C port.)
Other files
ICN# -> cicn
ICN# -> cicn is a small utility that converts "ICN#" resources to "cicn" resources. It is included for those of you who do some or all development on a Mac with 68000, on which the "cicn" editor in ResEdit won't run.
(Warning: Use ICN#->cicn with some caution. The current version is a quick hack to solve the problem, nothing else. Avoid saving on an existing file.)

The following file must also be in the HeartQuest and SAT Invaders projects, but it is not made by me:

transSkel.p (TransSkel.c, TransSkel.h)
The public domain subroutine package by Paul DuBois and Owen Hartnett. It takes care of all tedious window and menu handling, all the trivial parts that makes many programs so unnecessarily large. Sadly, it is still not MultiFinder aware, but MultiFinder and System 7 features can be implemented on top of it.
The Pascal version included here has some modifications made by me, to allow hierarchical menus, some fixes for dialogs etc. It works well for me, but don't blame Paul and Owen if I've made mistakes. The C version is also modified, by Bob Schumaker. I havn't used it a lot.
You may want to consider version 3 of the package (in C, useable from Pascal - thanks Paul!), which is MultiFinder aware. It is available by "anonymous ftp" from *ftp.primate.wisc.edu*.

3. Usage
General principles
SAT manages a list of *sprites*, describing position, current icon (preloaded to a face structure, see below) and action procedures called each frame and in collisions, if desired. You seldom have to access the list yourself, but if you do, you can get the first item with the global variable *gSAT.sRoot* and follow the pointers through the list from there.
SAT uses two offscreen bitmaps/pixmaps, named *gSAT.offScreen* and *gSAT.backScreen*. backScreen holds the background image, the backdrop. You can, when needed, SetPort(gSAT.backScreen) to perform drawing. (See the "Modifying the background" section.) The other bitmap/pixmap, offScreen, is a copy of the screen.

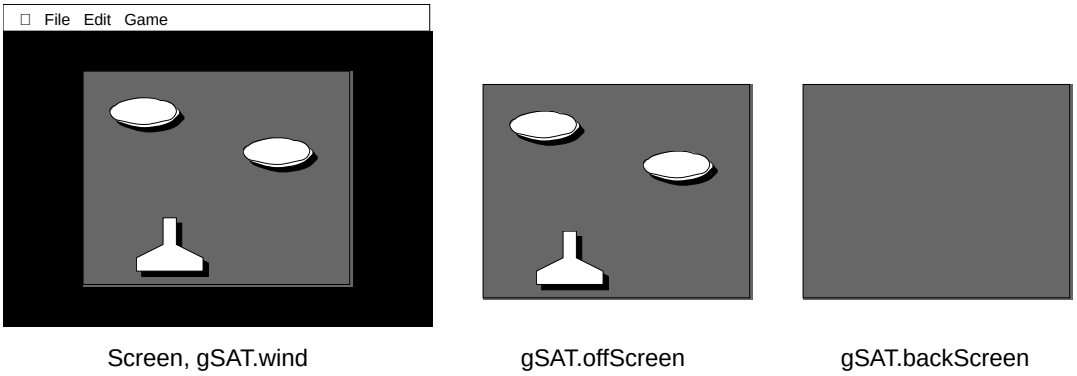

Figure 1. The screen, gSAT.offScreen and gSAT.backScreen, respectively.

SAT can draw the background automatically for you, if you pass the resource numbers for two PICT resources to it. These numbers are stored in the global variables *gSAT.pict* (for use in color) and *gSAT.bwpict* (for use in b/w). The background is drawn by SAT when SAT is initialized and when the screen depth is changed.
SAT by default uses the main screen. (There is a way to select another screen, but this feature is not tested much and still has a few problems.) Using the default setup, SAT fills the screen, excluding the menu bar, with a window. If the screen is bigger than the desired drawing area, SAT fills the rest of the window with a black border. All these features are configurable.

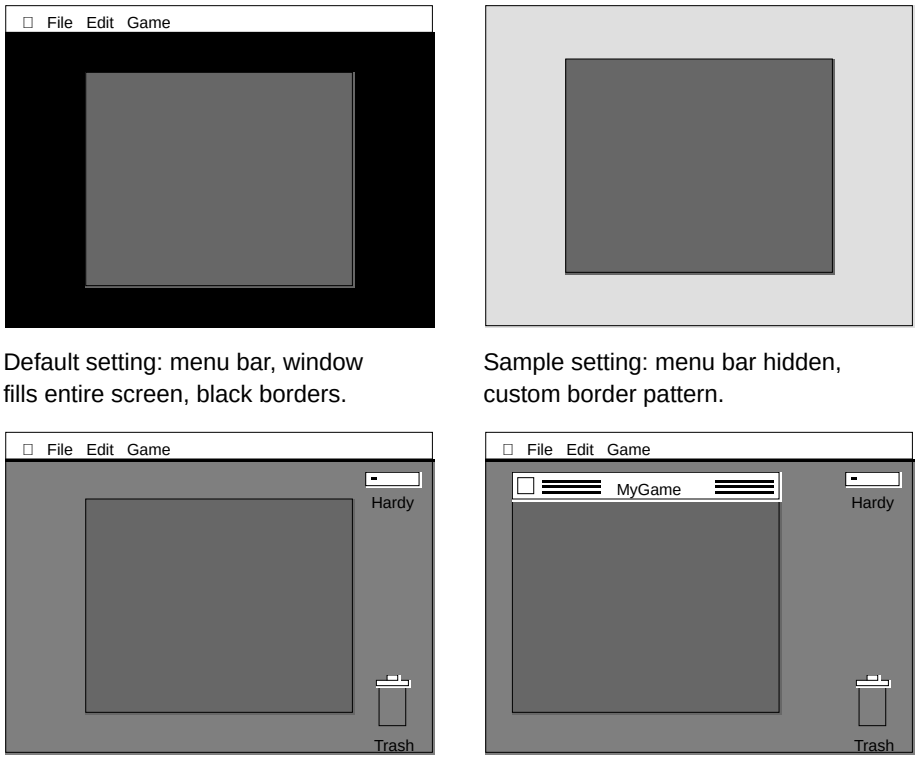

Figure 2. The default setting and a few of the alternatives.

SAT can also produce asynchronous sound. This is in one channel onlyYou may use channels for othe SAT uses Sound Manager if available, otherwise it switches to Sound Driver. When using Sound Manager, SAT keeps its channel open for extended periods. Thus, you must call SATSoundShutup before quitting, to dispose of the sound channel.

Using SAT

When you want to use SAT for a program of your own for the first time, it is easiest to start from one of the examples. The simplest one is SATminimal. When you want to see how SAT is used in a more complete program, with menus and event handling, check out SAT Invaders.

A (real) application using SAT typically include the following things:

Initialization. Initialize SAT (with InitSAT) and all your sprite units. In SAT Invaders, see the main program and GameWindInit in main.p. In SATminimal, this is just a call to InitSAT plus a call to initialize the sprite unit.

Routines for setting up a new game, new levels etc. In SAT Invaders, this is done in the StartGame and SetupLevel routines, respectively. All sprites are created in the SetupLevel routine, but you will often want to create sprites at other times, especially in the Setup* or Hit* routines. (See the next section.) In SATminimal, this is only a few calls to NewSprite.

A main loop for the game. In SAT Invaders, this is the MoveIt procedure, where RunSAT is called repeatedly until the game over condition is fulfilled. It is possible to call this as a background

task, from the normal event loop (where all normal window and menu handling is performed), but my experience is that action games will not run smoothly enough this way.
In SATminimal, the main loop is a very small loop, calling RunSAT until the user clicks the mouse, and calling TickCount to limit speed to the system clock.
When a game is not in progress, the program should be as most other Mac applications, with an event loop with menu and window handling etc. If you call *SATDepthChangeTest* often, either on update events (recommended) or before starting a new game, SAT will be able to change screen depth as needed.
Several sprite units. SAT Invaders has four such units: mPlayer, mEnemy, mShot and mMissile. SATminimal has only a single sprite unit.
The following figure shows an outline of the typical SAT-using application. "Application" and "Sprite Units" are the parts that have to be rewritten for every new game, while "SAT" and "SATSnd" are in the SAT library. Procedure names refer to the ones used in SAT Invaders. (1) and (2) are procedure calls that wouldn't fit in the drawing.

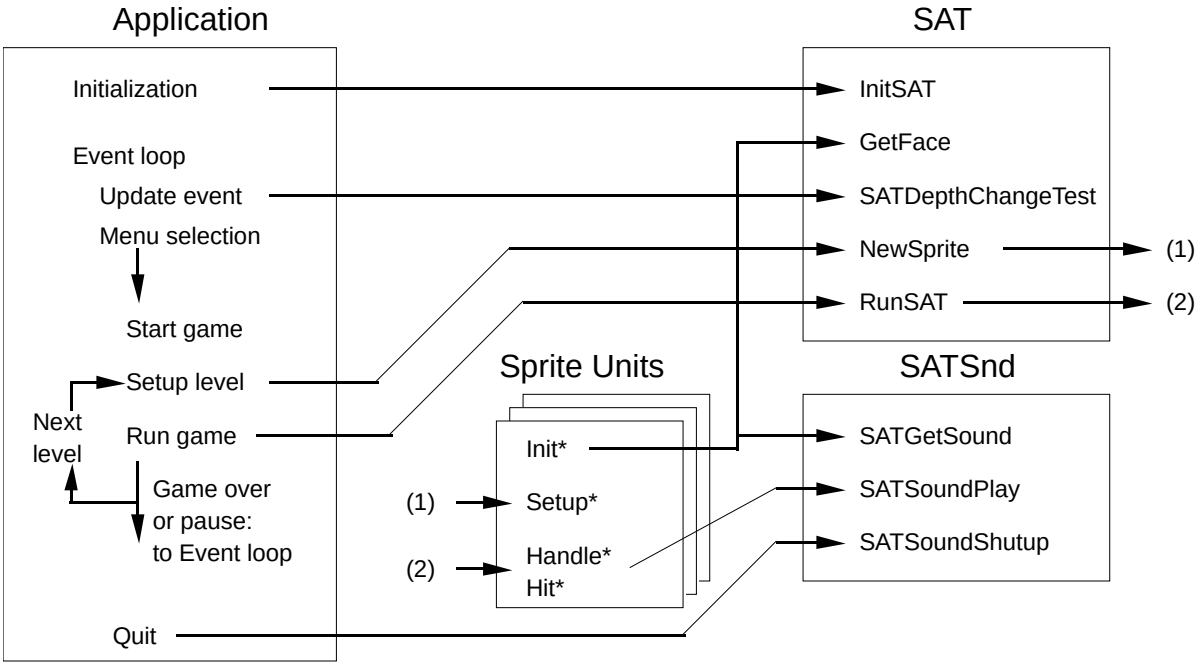

Figure 3. Outline of a typical program.

Many less important and/or more advanced procedures and functions are not shown in the Figure.
Initialization
Many, even most, of the routines in SAT demand that SAT has been initialized, either with InitSAT or CustomInitSAT. For example, GetFace preloads the chosen icon in the current screen depth, which demands that SAT knows what device it should use.
Thus, initialize SAT before calling other SAT routines, except ConfigureSAT, SATSetStrings and SATSetSpriteRecSize.

How to write a new sprite unit
In the following, I use the term *sprite unit* for the file defining a sprite type. For OOP people, this is something similar to a *class*. With *sprite*, I refer to a specific object on the screen, an instance of the sprite unit, created with the NewSprite (or NewSpriteAfter) call.
When you need a new kind of sprite, a moving object, you should make a separate unit of it, a sprite unit. The unit may contain any private procedures needed, but four procedures are standard:

```
procedure Init*;
procedure Setup*(me: SpritePtr);
procedure Handle*(me: SpritePtr);
procedure Hit*(me, him: SpritePtr);
```

The Init* procedure has whatever parameters you like.
The Setup* and Handling* procedures pass a pointer to the sprite itself.
The Hit* procedure pass pointers to the sprite and the sprite it has collided with.

These procedures must have unique names for every sprite unit. The suggested convention is use the names above, replacing * with the sprite uni

NewSprite:

function NewSprite (kind, hpos, vpos: integer; setup: ProcPtr): SpritePtr;

The call might lookwhere the sprite should appear.
kind is an integer that isI recommend that the sprites set their fields themselves, even the kind field, and the initial value of the *kind* field is used as variant selector for sprites with variable behaviour.
setup is a ProcPtr to a procedure that initializes the sprite, usually by setting the *task*, *hitTask*, *face* and *hotRect*.
hitTa should set its hotRect like this: `SetRect(sp^.hotRect, 0,0,32,32);` (where sp is a pointer to the sprite). Many sprites will, of course, be much smaller. The hotRect will often be smaller than the sprite itself.
– Collisions can be detected by inspecting the *kind* field and see if it has changed, or be resolved in the Hit* *(hittask)* procedure.
– (KindCollision only.) To make a sprite harmless (not react on collisons), set its *kind* field to zero (0).
– To move a sprite, change its *position* field. (Coordinates of upper left corner.) Check against borders using offSizeH and offSizeV.
– To change the appearance of a sprite, change its *face* field. If you want to cycle through a sequence of faces, make an array of *FacePtr*'s. See the examples.
– When you want to change the behavior of a sprite, change the Handle* procedure by setting the *task* field to the address of another procedure.
– When you make your own game, use lots of icons to get animation. This means lots of fiddling with icons (and/or good skill with a raytracer), but it is worth it. (My "Bachman" Pacman-clone game uses almost 200 icons, and Michael Hanson's "Asterax" space game uses almost 400 – as many as 36 for a single object!)
Sorting
The sprite list is incrementally sorted during the animation. One step of "BubbleSort" is performed for each frame. As the default, the sprites are sorted in order of their *position.v* (*kVPositionSort*), but you can change that to be according to the *layer* field (*kLayerSort*) or turn it off completely (*kNoSort*).
If you use the default sorting, a sprite located above another will be drawn before the other, so if they overlap, the higher one will appear to be farther away from the viewer. This sorting also allows the collision detection to work efficiently by only searching as long as it finds sprites within a certain distance (32 by default, but this can be changed using ConfigureSAT).

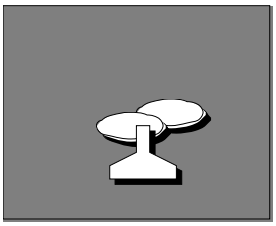

Figure 5. With standard sorting, closer to the bottom edge means foreground, closer to the top means background - an ordering that is often useable, but that can be changed as needed.

The sorting is a good reason for using NewSpriteAfter. If one sprite is created by another one (for example, a shot), it will be placed in the right part of the sprite list if NewSpriteAfter is used. If NewSprite is used, it may take a few frames before the sprite is in the right part of the list.
Collision detection and handling
Collision detection is extremely application dependant. Still, I have tried to put in a reasonably flexible system to help you with it. You have the option to turn it off and do it yourself in case it doesn't fit your needs. *If you find it confusing, check out the Collision and Collision][demos!*

SAT does collision detection by checking if the hotRects (displaced by position) of sprites overlap. Then, SAT provides two ways to report collisions. In both cases, the effects of the collision is expected to be resolved by the sprite units, not by the main program.
The following two methods are supported, one simplified and one general:

1) SAT changes the *kind* fields of the colliding sprites. (kKindCollision only!)

To use this kind of collision detection, use *kKindCollision* in ConfigureSAT. Then, only sprites will different signs on their *kind* fields can collide (positive vs negative). This mode is intended as a simplified mode for the rather commoyou don't care what kind of enemy that hit you, When colliding, sprites with kind>0 are assigned a kind of **10**. Sprites with kind<0 are assigned a kind of **-10**. (Sprites with kind=0 are neutral and don't collide at all, e.g. explosions.) The sprites can check for changes in the kind field in their Handle* procedures. This tells them only that they have collided, not with *what*.

2) SAT calls callback routines for each sprite.

A sprite may have a callback routine, the *hitTask* in the NewSprite call. When a sprite collides, the hittask is called. As mentioned above, the routine in question should be declared as

```
procedure Hit*(me, him: SpritePtr);
```

where *, by convention, is the name of the sprite unit. The SpritePtr *me* points to the sprite itself (the one that has the hittask being called) and *him* is the sprite with which it has collided. How to determine what kind of sprite the sprite collided with is up to you. (You can use some variable for identification, or the address of the *task* field.)
To use hit tasks, use any kind of collision detection except *kNoCollision*, but note that when using kKindCollision, only sprites with different signs on the kind field can collide.

Typical things to do when a collision occurs include:
• Removing the sprite. To do that, set its *task* to nil (me^.task := **nil**).

• Start another sprite (e.g. an explosion), using NewSprite or NewSpriteAfter.
• Play sounds (using SATSoundPlay and sound handles preloaded with SATGetSound).
• Modifying variables in the sprite, changing its position or application-defined variables (e.g. its speed, behaviour, look). Note that it is possible to the *task* and *hittask* to other procedures if desired, as long as these procedures take the same arguments as the ones they replace.

Note: The collision detection is
All-to-all collision detection: If kNoSort is chosen, the entire sprite list is searched for every sprite (for ForwardCollision and BackwardCollision, from the sprite to the end of the list). This can be fairly time-consuming if you use a lot of sprites, but is no problem if you only use a few.
Collision detection limited to "close" sprites: If kVPositionSort or kLayerSort is chosen, the search for hits is only performed for sprites within a certain distance (set by ConfigureSAT) from the *position.v* or *layer* of the sprite.
In simple cases, kVPositionSort and kKindCollision, the defaults, will be what you need - fast and easy to handle.
Additional notes on collision detection:
Non-rectangular sprites:
Even if your sprites aren't rectangular, you should use the hotRects to get possible collisions. If you use hitTasks, you can do additional checking once you know that the hotRects overlap. It is much faster to check rectangles than general shapes!
For example, consider that you want to check if two balls, circular shapes, collide. You set their hotRects to rectangles that the shapes fit in. When a collision is detected, you check if the distance between the two sprites is small enough for it to be a collision. (I.e. you check the squared distance (me^.position.h* me^.position.h+me^.position.v* me^.position.v) - (him^.position.h*him^.position.h+him^.position.v*him^.position.v) against the squared diameter of a ball. Got that?)
The demo Collision/// demonstrates one more advanced collision detection scheme, where the mask regions of the faces are used in order to determine if the masks overlap. This allows arbitrary shapes!

Custom collision detection (Advanced programmers):
If you need some other collision detection scheme, you can write it yourself. You can, for example, let each sprite search the sprite list in the Handle* routine. This is not a recommended method (why re-invent the wheel?), but might be of interest in special cases.
[I could make an example of this, but I think I'll leave it to the hackers who want it.]
Faceless sprites
It is legal for a sprite to have no face at all, that is, to assign the face to **nil**. In such a case, the sprite will be invisible, but it can still collide with other sprites. This can be used for collision detection with static objects, drawn in the background. By making such objects faceless, they are not drawn over and over again, which will save time.
(Note: I have made a test program using this, where faceless sprites are used for making platforms in a "platform game". This program, myPlatform, is part of the current distribution but is likely to be removed or at least heavily revised. It shows one out of many ways to implements platforms using faceless sprites. Unfortunately, it has rather poor controls so far.)

Note that this is not the only way to make moving sprites interact with static objects. You can also make your own structures. A typical approach is to use a 2-dime

or using patterns, the Pattern U

many format that SAT can use for direct-to-screen drawing) and draws it where you like. If you pass **nil** as GrafPtr, SAT assumes that you want it drawn in backScreen and calls SATBackChanged for you. You can use it for other tasks, drawing on the screen, in gSAT.offScreen or other ports, but that should be done with caution. The port to draw in must be a CGrafPort on color Macs, and must be a GrafPort otherwise (i.e. 68000-equipped Macs). SATPlotFace has a cousin, SATPlotFaceToScreen, that you should use for plotting sprites directly on the screen.

Making a face from other sources than cicn resources

The cicn resource format is a wonderful format for making sprites. In one single resource, you get both a color icon, a b/w icon and a mask. It is fairly comfortable to edit in ResEdit (though you may copy it to a program like PhotoShop for some advanced tasks) and it allows sizes up to 64*64. That's pretty good.

However, there are several cases where you still need to get your faces from another source.

• Many games put all their graphics in huge PICTs. This is a lot harder to edit, but is faster to load and takes less space on disk. If you are willing to spend the time putting together the PICTs and getting it all out right, it is preferrable over cicns.

• You might need sprites that are bigger than 64*64 (for boss monsters or whatever). In that case, you can store the icons for that sprite in a couple of PICT resources.

• You might want to generate your sprite faces on-line, putting text in them, reusing one icon in several faces with minor variations, etc.

• If you want your game to scale itself to the current screen size, you might want to rescale the icons to a size that is appropriate.

In order to make this possible, the routines *NewFace*, *SetPortFace*, *SetPortMask* and *ChangedFace* are provided. You create a blank face with *NewFace*. Then you set the port to it with *SetPortFace* and draw the face, set the port to its mask with *SetPortMask* and draw the mask. Finally, you tell SAT that you have modified the face by calling ChangedFace. (BUG NOTE: For now, you must always call *ChangedFace* some time before using a face created by *NewFace*. If you don't, you might get a system error.)

A face created by *NewFace* rather than *GetFace* will not be redrawn automatically when a depth change occurs. Instead, you must redraw it in the new bit depth. Obviously, you are also responsible for keeping compatibility with Macs without Color QuickDraw.

For a working demo, see **Collision ///**. It creates sprite faces from code. For another example, see the file FaceFromPICT.p or FaceFromPICT.c. They show how to load a face from PICT resources, where one PICT holds only one icon.

Scrolling backgrounds

Many games use scrolling backgrounds, e.g many Nintendo games. On dedicated game platforms, this is usually a rather simple task, since the scrolling can be done in hardware. On the Mac (like many other general-purpose computers), there's no general way that we can use for hardware scrolling, so we must do it in software. This means copying the entire visible area to the screen for every frame. Even with the fastest possible blitting routines (which aren't too much faster than QuickDraw when copying large areas) this is bound to be slow.

SAT can be used for making animations over a scrolling background. You do this by disabling the normal drawing (using a routine installed by *SATInstallSync*) and copying a part of the offScreen to the screen yourself. You might also want to set beSmart to false in CustomInitSAT, if you want to use a big offscreen area.

There is a demo program for this. Currently, it is called **Zkrolly**, and is very simple. (This might change.) The current demo is intended to show you the problems rather than impressing you:

• It uses a small, 200x200 window.

• It has a raster in the background, which causes an irritating flicker in some cases.

• It does no attempts to synch to the screen refresh, so there is some jumps in the graphics sometimes.

The first problem can only be solved by faster computers, lower frame rates or fewer colors. The othe

```
do all your drawing here…
SATSetPortScreen;
```

For those interested: this is equivalent to:

```
SetPort(gSAT.backScreen);
if colorFlag then
 if gSAT.backScreenGD <> nil then
   SetGDevice(gSAT.backScreenGD);
…do all your drawing here…
SetPort(gSAT.wind);
if colorFlag then
 if SATDevice <> nil then
   SetGDevice(gSAT.device);
```

Dying with dignity

When SAT encounters a fatal error, usually running out of memory, it will display an error message and quit. The error handling was desiged to let you forget about most error checking, and just run until it breaks. This way, SAT will find most errors for you, and exit without crashing.

However, this will sometimes not be enough. You may want another error message than the built-in one (though that can be changed with SATsetStrings), and you may want to take other action before quitting, like **saving the game**. SAT provides a hook for this. You may install a (pascal-declared) procedure with SATInstallEmergency. The procedure should take no parameters. It will be called after the error handling routines have disposed of the offscreen buffers, so you will usually have plenty of memory.
Note, however, that SAT doesn't (currently) let you abort the quitting. It can not continue from where the error was encountered, so quitting is the only way out.
Sound
You game needs sounds. Many sounds. In my experience, it's better to make many short sounds than few large ones.
SAT has routines for asynchronous sound. They are very easy to use, lets you forget all about channels, callbacks, and all the headaches that Apple gave us with the Sound Manager. All the typical application has to use is *SATGet[Named]Sound* to load sounds, *SATSoundPlay* to play them, and *SATSoundShutup* whenever you want silence or have to return the channels to the system (e.g. when quitting or task switch).
By default, SAT uses only **one** channel, but you can ask it to use more with *SATSoundInitChannels*. Ichannel for special use.
Why only one channel as default? Because playing sounds puts more load on the processor, so we shouldn't use more channels than necessary.
The sound playing routines (e.g. SATSoundPlay) take priority parameters, that tells SAT what sounds have preference over others. A high priority sound can, if necessary, stop a low priority one in order to be played immediately, while a low priority sound will be discarded or delayed (if you allow delay) if all channels are busy playing sounds with higher priority. As a convention, I use priorities 1 to 20, but you can use just about any value.
Finally, what is demanded for using the sound routines? Anything. If your program is used on a Mac with a very old system (System 5 or older) that has no Sound Manager, SAT will use Sound Driver (though it will only sound good with 11 kHz sounds that are uncompressed). The sound routines also have some workarounds for the bugs in Sound Manager up to version 2, making them stable on all Macs except a few accelerators. Sound Manager 3 is recommended, especially if you want many channels.
Some questions that I expect might become frequent
– I've made my first SAT program, but it crashes. Why?
• There are many possible reasons, but here are a few. Have you initialized SAT before trying to load faces? If you use Think C, have you initialized the appropriate managers? Is your resource file open in ResEdit? (Both Think C and Think Pascal fails to detect this problem.) Did your development system find your resource file? (Think C is rather brain-damaged on this point.)
Of course, the reason can be a bug of my part. Are you doing something radically different from the demo programs?

– Can I remove the black borders?
• Yes, you can. The black borders are drawn by PeekOffscreen, which you can replace by CopyBits'ing from offScreen to screen:

```
SetRect(r, 0, 0, gSAT.offSizeH, gSAT.offSizeV);
CopyBits(gSAT.offScreen^.portbits, gSAT.wind^.portbits, r, r, srcCopy, nil);
```

Except for the black border, these two lines are **all** that PeekOffScreen does, so you are losing nothing by replacing it.

You can also use a

s are disposed, though.
A code snippet follows, where the present environment is disposed, but all faces kept. gSAT.wind is diposed and recreated, but we could of course reuse it if we like. See also Collision ///.

```
KillSAT;                                {Nuke the old environment.}
DisposeWindow(gSAT.wind);                       {Nuke the old window}
SetRect(theArea, 70, 20, 220, 380);     {The new area}
CustomInitSAT(128, 129, theArea, nil, nil, true, true, false, true, true);
{Make new sprites here!}
ShowWindow(gSAT.wind);                  {Show the window.}
SelectWindow(gSAT.wind);
PeekOffScreen;                          {Redraw.}
```

– I'm running out of memory, despite setting X ridiculously high (where X is some memory assignment number)!
• SAT uses at least two offscreen buffers, and quite a bit of other data. Make sure you give your program enough memory, and that you do that in the appropriate place.
Running from inside Think Pascal: both the memory allocation for TP (Get Info) and the project zone size (in "Run options") must be big enough.
Running from inside Think C: the "partition" in "Set project type", plus that you need enough free memory outside Think C.
Running stand-alone: Your "SIZE" resource (i.e. Get Info) must ask for enough memory.
All demos should, as delivered, have enough memory to run in 8 bits a , but may need more memory if run in bigger screen depths or n bigger .

– I'm trying to play a sound with SATSoundPlay, but nothing happens or the sound isn't played until much later.
• This might happend if you play sounds during times when you don't call RunSAT repeatedly. RunSAT calls SATSoundEvents, which picks sounds from SAT's internal sound queue. (If you are curious, this is a workaround for a bug in Apple's Sound Manager before version 3.) If you want to use the sound routines when you don't animate with RunSAT, you must call SATSoundEvents yourself. Try **calling SATSoundEvents once** immediately after SATSoundPlay.

– Sometimes it works, sometimes not, and I just can't find any reason for the crashes. Sometimes my Mac crashes when I quit my program or Think Pascal/C.
• You might be bitten by one of the weaknesses that I havn't managed to solve in a really good way yet: having the wrong device chosen when the program quits.
First thing to try: Call SATSetPortScreen upon exit. That will set the device to the main screen (granted that you use the main screen and not another one). Most SAT calls preserve the port and device, but some – i.e. the SATSetPort*** calls – change it, so if you use them, you must restore port and (most importantly) the device yourself.

– When I play the first sound, there's a big delay before it starts.

• This is the Mac OS fiddling around with the memory. Compacting memory before starting may be a good idea to avoid this. You can also try SATPreloadChannels.

– If I call GetFace or SATGetSound several times with the same resource number, will I get the same FacePtr/Handle, or will it load several times?
• It will load several times.

– I thought this was supposed to be easy. Must I learn all those calls?
• No, I recommend that you start with the basics, and learn more when you need it. Browse SATminimal: it uses a very small part of SAT, a suitable start for a beginner. The following calls are rather fundamental:

```
procedure InitSAT (pictID, bwpictID, Xsize, Ysize: integer);
function NewSprite (kind, hpos, vpos: integer; callback, setup, hittask: ProcPtr): SpritePtr;
function GetFace (resNum: integer): FacePtr;
procedure RunSAT(fast:Boolean);
procedure PeekOffScreen;
function SATGetNamedSound (name: Str255): handle;
procedure SATSoundPlay (theSound: handle; priority: integer; canWait: boolean);
procedure SATSoundShutup;
```

– Wouldn't it be good if I could pick one out of a few standard handling procedures? And how about looping a set of faces automatically?
• Most sprites move and change faces in different ways. Only primitive games have only one appearance on all its sprites, looping a fixed sequence. We can only cover a few special cases, and that is pointless. If you make a game where all sprites move the same way, you can write your own standard behaviour, and call that from the handling procedures. For example, for making a sprite bounce around:

```
procedure SATBounce (me: SpritePt
```

```
sprite is     hitT
  destructTask: ProcPtr; { Callback when the sprite is disposed. Usually nil. }
```

and All other fields are set to zero (nil). You should always set *task* to point to a handling procedure (even if it is an empty one), and you should usually set *face* and *hotRect* to something appropriate too.

The field *kind* is used in kKindCollision mode, and is otherwse used for whatever you like.
position is the position of the sprite, of course.
hotRect is a rectangle that is used in collision detection. Given a face, a good default value is face^.iconMask.bounds. *hotRect2* is hotRect displaced by position (done by SAT).
face is a pointer to the face that the sprite should have.
task is a pointer to the handling procedure (called every frame).
hitTask is a pointer to a procedure to call when a collision is detected.
destructTask is a pointer to a procedure to call when the sprite is disposed of.
oldpos, *r* and *oldr* are internal variables used for updating the screen correctly. If you change them, there is a risk that the sprite will leave garbage.
next and *prev* are pointers to the next and previous sprite in the sprite list. Use them if you want to make your own collision detection scheme. Don't change them from inside RunSAT (i.e. a handling or hit procedure. If you make your own sorting mechanism, run it outside RunSAT.
layer is used in kLayerSort, and is otherwise used as you please.
speed, *mode*, *appPtr* and *appLong* are for free use. You can rename them, change type, and even add more variables, but if you do that, if the size of the sprite record changes, you **must** call SATSetSpriteSize(sizeof(Sprite)) before any sprites are allocated!

to switch to b/w drawing when approprite,
The following globals are **not part of gSAT** (since they are not directly associated with the current SAT setup, and thus shouldn't be changed if we switch between two SAT environments):

faceRoot: FacePtr;

faceRoot (not part of gSAT) is a pointer to the first face in the face list.

gCopyBitsAddr: ProcPtr;

gCopyBitsAddr is a pointer to the CopyBits procedure. You have no reason to change this unless you want to use it to plug in your own CopyBits replacement. (There are already custom blitters for b/w, 4-bit color and 8-bit color in the library, so you have little reasons to hack SAT this way – but you can.)

gSizeofSprite: Longint;

gSizeOfSprite is the size of the sprite record. SATSetSpriteSie sets this global.

e) on which SAT should run its animation. Pass nil to get the main device. [Bug note: There is a bug that causes incorrect colors if your main screen and the screen on which SAT is drawing are in different depths.] If ChosenScreen is not the main device, useMenuBar is ignored. (Only the main device has a menu bar.)
The flag useMenuBar tells SAT that it should use the menu bar space if needed, since we intend to hide the menu bar while animation is in progress. (See also HideMBar and ShowMBar.) If you intend to hide the menu bar, pass true. If you pass false, the drawing area is clipped in order not to touch the menu bar.
If centerDrawingArea is true, SATdrawingArea is centered on the main screen.
If fillScreen is true, the created window fills the whole screen. Otherwise, the window is set to the SATdrawingArea rectangle. If preloadedWind is not nil, FillScreen is ignored.
If dither4bit is true, SAT dithers all sprites when running in 4-bit color. This will usually look a lot better if the icons are drawn in 256 colors. If your icons are drawn in 16 colors, you may want to turn this off.
If beSmart is true, SAT will limit the animation area as mentioned above (under SATdrawingArea). If it is false, you get what you ask for. You should usually pass true. Turning this "smartness" off is interesting in two cases that I can think of right away: 1) If you intend to make all drawing with QuickDraw and want to turn off the clipping to coordinated divisible by 8, or 2) if you make a scrolling game, in which case you want the offscreens to be bigger than the window and perhaps even the screen.
Calling CustomInitSAT(pictID, bwpictID, r, nil, nil, false, true, true, true, true, true), where r is a rectangle with appropriate width and height, is equivalent to a call to InitSAT.

procedure ConfigureSAT (PICTfit: boolean; newSorting: SortType; newCollision: CollisionType; searchWidth: integer); [For advanced users.]

ConfigureSAT lets you set certain parameters that affect SAT's behaviour. It usually should be called before InitSAT or CustomInitSAT, during program startup, but can also be called later. If you don't call it at all, SAT defaults to false, VpositionSort, KindCollision and 32.
If PICTfit is true, any background PICTs (pictID and bwpictID above or the globals SATpict and SATbwpict) are scaled to fit the drawing area. The NewSorting parameter tells SAT how it should sort the objects. You have the following options:

kVPositionSort: Sort after the position.v field. Makes low sprites appear to be in the front.
kLayerSort: Sort after the layer field (thus defined by the application).

kNoSort: Don't sort at all. (Use this if you want to make your own sorting scheme or if none is needed, i.e. you create all sprites at the proper places and they aren't supposed to change order.)

The NewCollision parameter tells SAT how it should detect collisions. You have the following options:

kKindCollision: Collisions are detected using the HotRect's and the kind field. Objects with kind=0 never collide, and others collide only if they have different signs on their kinds. Useful when the game has a distinct good and evil side, where collisions between friends are not important.
kForwardCollision: Search forward in the sprite list, and report collisions with the HitTask procedure.
forward in the sprite list, and report collisions with the HitTask procedurekBackwardCollision: Search backwards in the sprite list. Essentially the same as ForwardCollision.
kNoCollision: No collision detection. Use this if you don't need collision detection or if you perform it yourself.

Note that all collision detection routines depend on what sorting is performed. If the sprite list is sorted after position.v (kVPositionSort), only sprites within SearchWidth pixels are checked. If it is sorted after layer (kLayerSort), sprites with a layer value within SearchWidth from the sprite is checked. In other cases, all sprites are checked. You may consider using kNoCollision and perform the detection yourself.
Sprite management:
function GetFace (resNum: integer): FacePtr;

GetFace (formerly called LoadIcon) loads the 'cicn' resource with n

ld take no parameters but return a boolean, i.e. be declared: function MySynch:Boolean;
This procedure is called once per frame, immediately before any drawing takes place on the screen. This function is intended for two things:
• synchronizing the animation to the screen vertical retrace, which may be needed in some programs.
• disabling drawing altogether, which is intended for scrolling games. If the function returns false, SAT draws as usual, but if it returns true, no drawing is done to the screen at all.
Most games have no need for synchronization to the vertical retrace. Consider it if your animation feels shaky, flickering and not smooth enough. (This is typically games where sprites move in constant speed over many frames, or scrolling games.)
SAT has, at present, no built-in synching, but the option to install a procedure this way makes it possible for you to add it later. My experience so far is that it's very hard to synch animation of this kind to be totally smooth. Fortunately, most programs don't need it.
Making scrolling games on the Macs is rather hard. Forget about scrolling the entire screen if you want decent speed. Try a smaller area. Also, for keeping speed up, you may choose to turn sprites invisible (setting the face to nil) when they are outside the currently visibe area.
For scrolling games, you are responsible for copying the appropriate parts of offScreen to the screen. You may choose to do that in the synch procedure. Use CopyBits (safest) or SATCopyBitsToScreen (fastest, esp for rather small areas).

procedure SATInstallEmergency (theEmergencyProc: ProcPtr);

(Advanced initialization.) SATInstallEmergency installs a procedure theEmergencyProc, to be called when a fatal error occurs, before SAT exits. The emergency proc should take no parameters and leave no return value. The most common fatal error is out of memory. Typical actions to take in theEmergencyProc include:
- Save the game or document (if your game supports that).
- Record the current score in the high score list.

function NewFace (faceBounds: Rect): FacePtr;

(Advanced face management.) Creates an empty face with the specified size, to be drawn in with SetPortFace and SetPortMask. If a screen depth change occurs, you are responsible for redrawing the face.

procedure SetPortFace (theFace: FacePtr);
procedure SetPortMask (theFace: FacePtr);

(Advanced face management.) Sets the current port to a face or the mak of a face, so you can use QuickDraw calls to draw in it. This can be used for resizing sprites or for generating them from the program rather than from resources. When done drawing, you must call ChangedFace.
Warning: You can not trust the port set by these functions to stay valid over extended periods. Calls to RunSAT, GetF and ChangedFaceace will invalidate it.
The ports used by these routines are found in gSAT:
SetPortFace uses gSAT.iconPort and gSAT.iconPortGD.
SetPortFace2 uses gSAT.iconPort2 + gSAT.iconPort2GD.
SetPortMask uses gSAT.bwIconPort.

procedure ChangedFace (theFace: FacePtr);

(Advanced face management.) Preshifts the graphics in theFace for 1-bit and 4-bit graphics. You should always call this after drawing in a face with SetPortFace and SetPortMask.

procedure SATSetStrings (ok, yes, no, quit, memerr, noscreen, nopict, nowind: Str255);

(International utility) With this call, you can set all strings that SAT uses (error messages and button names) to the strings of your choice. This is intended for making programs in other languages. The following string can be set:
ok: The "OK" button in ReportStr.
yes, no: The "Yes" and "No" buttons in QuestionStr.
quit: The "Quit" button in the fatal error alert box.
memerr: Out of memory error message.
noscreen: A rather unlikely error, where no screen device is found.
nopict: Error message: The background PICT demanded by InitSAT or CustomInitSAT could not be found, or we went out of memory when trying to load it.
nowind: A rather unlikely error, where we have no window after initialization. (Probably out of memory.)
You must set all strings when calling SATSetStrings. Don't be too clever with replacing them with humorous messages: users might not appriciate it. They want to know what the problem is and how to fix it, nothing else. (Humor is better used in other places.)
The recommended usage is to load strings from resources, preferrably a STR# resource, and pass the appropriate ones to SATSetStrings and use others for your own strings constants.

Consider doing this once you have a working program. HeartQuest does this, and thus is fully translateable without recompilation.

procedure SkipSAT;

SkipSAT does the same things as RunSAT except drawing. Collision detection, sound playing and sprite handling routines are performed. It should typically be used instead of RunSAT in order to get reasonably high speed on Macs that are too slow to keep up with the speed you want when calling RunSAT for all frames. You should avoid skipping more than one frame at a time, since the animation will get jerky.
Most applications will run better without SkipSAT, even if they run slightly slower than intended on slow Macs. Consider SkipSAT if you use so many sprites that the program gets unreasonably slow on the slowest Macs it may be used on (typically MacPlus).

procedure KillSAT;

KillSAT disposes of SAT's entire environment except gSAT.wind, sounds, and faces, so you may re-initialize SAT to another state (e.g. another screen). Demo in Collision ///.

procedure SATMakeOffscreen (var portP: GrafPtr; rectP: Rect; var retGDevice: GDHandle); {Make offscreen buffer in current screen depth and CLUT.}
procedure SATDisposeOffScreen (var portP: GrafPtr; theGDevice: GDHandle); {Get rid of offscreen}

If you need an extra offscreen buffer, SATMakeOffscreen and SATDisposeOffScreen are usually what you need. SATMakeOffscreen creates an offscreen buffer of the same size, depth and color table as the other offscreens. SATDisposeOffScreen disposes of the created structures. Both call the functions below when used on color Macs.

function CreateOffScreen (bounds: Rect; depth: Integer; colors: CTabHandle; var retPort: CGrafPtr; var retGDevice: GDHandle): OSErr; {From Principia Offscreen}
procedure DisposeOffScreen (doomedPort: CGrafPtr; doomedGDevice: GDHandle);{From Principia Offscreen}

CreateOffScreen and DisposeOffScreen are taken directly from Apples technote Principia Off-Screen Graphics Environments. They will only work on a color Mac, as opposed to the above routines. Use them if you have special needs, like offscreen buffers with another color table.
Sound routines:
The sound with priority handling, managing the bugs in Apple's Sound Manager as well as possible. configure it to use more If Sound Manager is not available, the Sound Driver is used instead. MACE-compressed sounds may be used when Sound Manager is available.

procedure SATSoundInit;

Initializes the sound package. This is called from InitSAT so you hardly have to use it directly.

procedure SATSoundOn;
procedure SATSoundOff;

These routines turns SATSnd on and off. After InitSATSnd is called, SATSnd is on. SATSoundOff does not stop sounds being played. These sounds will play until they are finished. To turn off sound immediately (i.e. if the user issues a "sound off" command), you can call SATSoundOff and SATSoundShutUp in sequence.

procedure SATSoundPlay (theSound: handle; priority: integer; canWait: boolean);

Play a sound. The handle should have been created by calling MakeSoundHandle (see below). The priority should be 0-9 for less important sounds and >10 for the more important sounds (like extra life sound, dying sound...) CanWait tells whether the sound should be queued until

channel is free, in case sound, or if it should be discarded in such a case.

procedure SATSoundEvents;

SATSoundEvents is usually not needed in your programs, since it is called from RunSAT. If you use the sound routines when no animation is running (that is, when you don't call RunSAT repeatedly) you need to call SATSoundEvents a few times per second or so, or after each call to SATSoundPlay. The latter is only recommended if you never try to play several sounds in very short time.

procedure SATSoundShutup;

Stop any sound in progress. Must be called before the program terminates or the sound channels may be left open! It does not turn off SATSound, merely stops ongoing sounds.

function SATGetSound (sndId: integer): handle;
function SATGetNamedSound (name: Str255): handle;
procedure SATDisposeSound (theSnd: handle);

A call to SATGetSound or SATGetNamedSound preloads a sound. This should be done for all sounds at startup. (Don't do it while animating - it will take too much time.) Use either of the two calls. If you are done with a sound and don't need it more, you can dispose it with SATDisposeSound. Don't dispose a sound that might still be playing.
Multi-channel sound:
function SATSoundInitChannels (num: integer): integer;

SATSoundInitChannels is generally the only routine you have to use for multi-channel sound. It sets the number of channels that you want to use, if they are available. This is done in three different ways, depending on the parameter num:
num > 0: num specifies the number of channels that should be allocated.
num < 0: num specifies how many channels that should not be allocated. In that case, SATSoundInitChannels allocates all channels that are available, and then frees up -num channels.
num = 0: SATSoundInitChannels uses half of all available channels.
Why use less that all channels available? Because more sounds playing means less time for animation. Also, you may want to have some other sound-making package running that needs a few channels (a music playing package, for example).
The return value is the number of channels that SAT will use. It may be what you ask for or less. On some old Macs you will always get a single channel.
After setting the number of channels this way, SAT will direct sounds to appropriate channels for you, so you don't have to bother about what channel a sound is played in. If you need to control a channel yourself, you can use the routines SATSoundReserveChannel , SATSoundPlayChannel and SATSoundShutupChannel, below.
The rest of the routines in this section are advanced routines that you should ignore until you really need them!

function SATSoundDoneChannel (chanNum: integer): Boolean;

SATSoundDoneChannel inspects the specified channel and tells whether it is busy or not. It

returns true if the channel is free.

procedure SATSoundPlayChannel (theSound: Handle; chanNum: integer);

SATSoundPlayChannel plays a sound on the specified channel This stops any sound in progress in the channel regardless of priority, and bypasses the queues that SATSoundPlay uses. If

the channel is not reserved (see SATSoundReserveChannel), the sound is treated as a priority 10 sound with respect to sounds played with SATSoundPlay.
NOTE: Due to bugs in Apple's Sound Manager prior to SM version 3, the Mac can crash if you access a sound channel too quickly, and SATSoundPlayChannel has no protection against that.

procedure SATSoundReserveChannel (chanNum: integer; reserve: Boolean);

SATSoundReserveChannel sets the reserve bit for the specified channel. A channel with its reserve bit set will not be used by SATSoundPlay, only by SATSoundPlayChannel.

procedure SATSoundShutupChannel (chanNum: integer);

SATSoundShutupChannel silences and disposes of the specified channel. It will be re-allocated whenever a new sound is played on the channel.

procedure SATPreloadChannels;

SATPreloadChannels allocates all channels (up to the number that was returned by the SATSoundInitChannels call), in order to avoid unnecessary Memory Manager calls after the animation has started. [This call should be considered experimental for now.]
Pattern utility routines:
The following routines are added in order to simplify pattern handling. They allow you to define a 'PAT ' resource and a 'ppat' resource with the same ID, and the appropriate one will be picked automatically. You only need the 'ppat' resource, even for Macs without Color QD.
The point with these utility routines is that they provide a "glue" to make your program work on b/w Macs as well as color ones, and to use the b/w patterns built into 'ppat' resources without any extra checks for screen depth.
Example: In order to fill the background with a pattern, get the pattern with SATGetPat, set the pen to it with SATPenPat, and fill with PaintRect.

procedure SATPenPat (SATpat: SATPatHandle);
procedure SATBackPat (SATpat: SATPatHandle);

SATPenPat and SATBackPat sets the pen and background pattern, respectively, to SATpat. (Replaces PenPat/PenPixPat and BackPat/BackPixPat.) If the Mac runs in b/w, the b/w (old-style) pattern is used.

function SATGetPat (patID: integer): SATPatHandle;

SATGetPat replaces GetPattern and GetPixPat. It gets the 'ppat' with ID patID if the resource exists. If not, it tries to get the 'PAT ' with the same ID.

procedure SATDisposePat (SATpat: SATPatHandle);

SATDisposePat releases the pattern resource and diposes of the record.
Utility routines:
Most of these should be rather self-expanatory.

```
procedure DrawInt (i: integer);
procedure DrawLong (l: longint);
function Rand (n: integer): integer;
function Rand10: integer;
function Rand100: integer;
procedure ReportStr (str: str255);
function QuestionStr (str: str255): boolean;
function SATFakeAlert (s1, s2, s3, s4: Str255; nButtons, defButton, cancelButton: integer; t1,
t2, t3: Str255): integer;
function TrapAvailable (theTrap: Integer): Boolean;
procedure SetMouse (where: point);
function SATGetCicn (cicnId: integer): CIconHandle;
procedure SATPlotCicn (theCicn: CIconHandle; dest: GrafPtr; destGD: GDHandle; r: Rect);
procedure SATDisposeCicn (theCicn: CIconHandle);
```

DrawInt and DrawLong converts the argument to a string and draws it with DrawString.

Rand(n) routines produce a random number in the range [0..n-1]. Rand10 is equivalent to Rand(10), and Rand100 to Rand(100).

ReportStr makes an alert with an "OK" button and the string str. QuestionStr gives a similar alert, but with two buttons, "yes" and "no", and returns true if "yes" is pressed.

SATFakeAlert is a more general alert function, allowing up to three buttons. s1 to s4 are strings forming the message. t1 to t3 are the button names. nButtons is the number of buttons. defButton i the default button, which is framed. cancelButton is the button selected by command-period.

TrapAvailable is a function I found in Inside Mac 6. It tells if a certain trap is implemented. You can use it to see if Gestalt is available, but I find it rather useful for checking for many kinds of functionality without using Gestalt at all. For example, to see if 32-bit QD is around, call TrapAvailable($ab1d).

SetMouse is the routine most likely to break in the future, since it depends on low-memory globals. It may be wise to avoid it if you don't really need it.

SATGetCicn, SATPlotCicn and SATDisposeCicn are non-color compatible replacements for GetCIcon, PlotCIcon and DisposeCIcon. With color QuickDraw, the only difference is that SATPlotCicn takes a port and a GDevice as parameters. (You may pass nil for those, in which case the current port is used.) Without color QuickDraw, SATGetCicn loads the cicn, locks it, and set its pointers. SATPlotCicn uses CopyMask, and does special processing in the case of a 1 byte wide BitMap (which old QuickDraw can't handle).
They require colorFlag to be correct, but does otherwise not depend on SAT being initialized.

Final words

The present SAT is still a beta version, but now (at 2.0b9) it's getting really close to final 2.0. I still want comments, ideas for improvements.

– Is the manual informative? Does it help you in writing new programs? Any grammatical errors? (After all, english isn't my native language.)
– Is the interface to SAT good? What can be improved?
– Any missing features? Ideas for improvements? Limitations that should be fixed? Any ideas about how to make the collision detection system better?
– Are the example programs informative? Should they be changed, expanded, shortened, polished, or perhaps totally different?

– What topics could be added to the manual? Some I have in mind:
•How to do your own collision detection (by searching through the sprite list)?
•How to do your own sorting routine (by modifying the sprite list)?
•A list of suggestions for games to make? Ideas?

Quick reference

Initialization:
procedure InitSAT (pictID, bwpictID, Xsize, Ysize: integer);

Customized initialization:
procedure ConfigureSAT (PICTfit: boolean; newSorting: SortType; newCollision: CollisionType; searchWidth: integer);
procedure CustomInitSAT (pictID, bwpictID: integer; SATdrawingArea: Rect; preloadedWind: WindowPtr; chosenScreen: GDHandle; useMenuBar, centerDrawingArea, fillScreen, dither4bit, beSmart: Boolean);

Sprite and face routines:
function NewSprite (kind, hpos, vpos: integer; setup: ProcPtr): SpritePtr;
function NewSpriteAfter (afterthis: SpritePtr; kind, hpos, vpos: integer; setup: ProcPtr): SpritePtr;
procedure KillSprite (who: SpritePtr);
function GetFace (resNum: integer): FacePtr;
procedure DisposeFace (theFace: FacePtr);

Running the animation:
procedure RunSAT(fast:Boolean);

Drawing:
procedure SATPlotFace (theFace: FacePtr; theGrafPtr: GrafPtr; where: Point; fast: boolean);
procedure SATPlotFaceToScreen (theFace: FacePtr; where: Point; fast: boolean);
procedure SATCopyBits (src, dest: GrafPtr; destGD: GDHandle; srcRect, destRect: Rect; fast: Boolean);
procedure SATCopyBitsToScreen (src: GrafPtr; srcRect, destRect: Rect; fast: Boolean);
procedure SATBackChanged (r: Rect);

SetPort replacements:
procedure SATGetPort (var port: GrafPtr; var device: GDHandle);
procedure SATSetPort (port: GrafPtr; device: GDHandle);
procedure SATSetPortOffScreen;
procedure SATSetPortBackScreen;
procedure SATSetPortScreen;

Maintainance:
function SATDepthChangeTest: boolean;
procedure SATDrawPICTs (pictID, bwpictID: integer);
procedure PeekOffScreen;

Menu bar:
procedure ShowMBar;

```
procedure HideMBar(wind: WindowPtr);
```

Advanced calls:

```
procedure SATInstallSynch (theSynchProc: ProcPtr);
procedure SATInstallEmergency (theEmergencyProc: ProcPtr);
procedure SATSetSpriteRecSize(theSize: longint);
procedure SetPortMask (theFace: FacePtr);
```

procedure SetPortFace (theFace: FacePtr);
function NewFace (faceBounds: Rect): FacePtr;
procedure ChangedFace (theFace: FacePtr);
procedure SATSetStrings (ok, yes, no, quit, memerr, noscreen, nopict, nowind: Str255);
procedure SkipSAT;
procedure KillSAT;
procedure SATMakeOffscreen (var portP: GrafPtr; rectP: Rect; var retGDevice: GDHandle);
procedure SATDisposeOffScreen (var portP: GrafPtr; theGDevice: GDHandle);
function CreateOffScreen (bounds: Rect; depth: Integer; colors: CTabHandle; var retPort: CGrafPtr; var retGDevice: GDHandle): OSErr;
procedure DisposeOffScreen (doomedPort: CGrafPtr; doomedGDevice: GDHandle);{

Sound:
procedure SATSoundInit; {Usually not used by applications.}
procedure SATSoundOn;
procedure SATSoundOff;
procedure SATSoundPlay (theSound: handle; priority: integer; canWait: boolean);
procedure SATSoundEvents; {Usually not used by applications.}
procedure SATSoundShutUp;
function SATGetSound (SndId: integer): handle;
function SATGetNamedSound (name: Str255): handle;
procedure SATDisposeSound(theSnd: handle);
function SATSoundInitChannels (num: integer): integer;
function SATSoundDoneChannel (chanNum: integer): Boolean;
procedure SATSoundPlayChannel (theSound: Handle; chanNum: integer);
procedure SATSoundReserveChannel (chanNum: integer; reserve: Boolean);
procedure SATSoundShutupChannel (chanNum: integer);
procedure SATPreloadChannels;

Pattern utilities:
procedure SATPenPat (SATpat: SATPatHandle);
procedure SATBackPat (SATpat: SATPatHandle);
function SATGetPat (patID: integer): SATPatHandle;
procedure SATDisposePat (SATpat: SATPatHandle);

Misc:
procedure DrawInt (i: integer);
procedure DrawLong (l: longint);
function Rand (n: integer): integer;
function Rand10: integer;
function Rand100: integer;
procedure ReportStr (str: str255);
function QuestionStr (str: str255): boolean;
function SATFakeAlert (s1, s2, s3, s4: Str255; nButtons, defButton, cancelButton: integer; t1, t2, t3: Str255): integer;

```
function TrapAvailable (theTrap: Integer): Boolean;
procedure SetMouse (where: Point);
function SATGetCicn (cicnId: integer): CIconHandle;
procedure SATPlotCicn (theCicn: CIconHandle; dest: GrafPtr; destGD: GDHandle; r: Rect);
procedure SATDisposeCicn (theCicn: CIconHandle);
```