Sprite Animation Toolkit

by Ingemar Ragnemalm

A programmer's library for making sprite-based animation (especially games).

For Think Pascal ,Think C or CodeWarrior on the Macintosh.

Copyright © 1992-1994 by Ingemar Ragnemalm. All rights reserved.

Version 2.3

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 at least a dozen *released* games: **Slime Invaders**, **Bachman**, **HeartQuest** and **Ingemar's Skiing Game** by myself, **Missions Of The Reliant** by Mike Rubin, **Asterax** by Michael Hanson, **Invaders** by Bettini Simone and **CyberNation** by Roy Dictus, and… I'm almost losing track of them… **Bert**, **LetterLand**, **Bedlam**, **NeXus**, **Spacewar**, **Warbirds**… 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, with the option to plug in bliters for other depths (for advanced programmers).

– 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, old Macs like Plus, SE, Classic included.

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) under the conditions below.

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 SAT 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 and my name in the documentation and the "About" box.

If you want to use SAT to produce a commercial program, contact me.
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, **Symantec C++** version 6 or 7 or **CodeWarrior** Pascal or C version 4.5.

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** (one channel or multi-channel, depending on how much time you want to have left for the animation), 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), with support for switching between bit depths even after initialization. The sprites can have any size that you can use in a "cicn" resource, that is, up to 64·64. With a little more effort, you can use other sources, like PICTs, in which case any size is possible. It can use a scrolling background, though that isn't generally recommended. (It's slow.)

SAT is written in a "pseudo-object-oriented" fashion that I find rather comfortable for the problem, where sprites provide callback routines for SAT to call as appropriate.

Disclaimer

Though I believe the package to be of good quality and useable for most of my game- and animation-making needs, I do not guarantee that it will suit your specific needs, nor that it will work on all Macs or system versions. I take no responsibility for damage, loss of data etc that may be caused by using SAT.

Acknowledgements

Special thanks to Juri Munkki for help with the color drawing routines, to Michael A. Kelly for sharing the code from which I based my direct-to-screen code on, to Tony Myles for advice (and good competition), and to Frank A. Lonigro, who long ago posted a small code sample on the net with which everything started.

Thanks to Paul DuBois and Owen Hartnett for the TransSkel package. I use it all the time, and find snippets from its demos everywhere in SAT.

Many thanks to all the beta testers, especially Mike A. Balfour, Alex Ivrii and Mike Rubin. I hope we will see your great games on the net soon! (Mike Rubin's game,

Missions of the Reliant, is released now - and it's already a hit!)

Thanks to Mike Zimmerman (MyPlatform) and Ken Long (Collision, SATInvaders, Zkrolly, HeartQuest) for the help in translating the demos to C, and to Nathaniel Woods, for suggesting several good enhancements, for finding a rather serious bug and for helping me making SAT possible to use from C++. Nathaniel's C++ interface is coming soon.

And thanks to all people who are using SAT for their programming projects! You are making it worthwhile!

Related packages

Other programmers have, of course, had the same idea as I, to let others use the code they have developed. Some have simply offered to sell source code to their programs (both Duane Blehm and John Calhoun). Personally, I find it hard to take big examples and do something useful, so I'm trying the library approach instead.

Juri Munkki's *Vector Animation Tookit* (VAkit) deserves mentioning. It is part of the source code to the game Arashi (a.k.a. Storm), available from various ftp archives. It produces color vector graphics in high speed. It requires 256 colors. Consider it for making games like Star Wars or Vectrex-style games.

On the subject of sprite animation packages, there are a few demos with source code (Tony Small's *Cellusoft Graphics* routines, the *Cheesetoast* sources and my own recent contribution *Offscreen Toys*). Those are (hopefully) of educational value for people who want to roll their own sprite-using programs.

A few take the approach of SAT, aiming for a library with lots of reuseable stuff. The oldest one I know is *Sprite Manager* by Eli Bishop, from 1989. B/W only. I would consider it out-of-date today.

Another one is Ricardo Batista's *Sprite Manager* (same name but a different package), which was distributed on one developer CD in 1993 (?). As far as I can tell, that project wasn't ever really completed, since it is poorly documented (at least in the package I have) and the demo often crashes. Color only.

Then we have *SpriteWorld* by Tony Myles. Well-working beta version available from various archives. Supports old QuickDraw as well as color, best used from C.

The latest appearing is *Graphics Elements* by Al "Captn Magneto" Evans. It has a somewhat different scope, including functions more related to controls (as in Control Manager) than games. In C, color only (?).

2. Packing list

Files in the toolkit

The following files are the files in the SAT toolkit itself.

Pascal library:

SAT.p
Interface file for the library.

SAT.lib
The Sprite Animation Toolkit compiled to a library.

You should include both files in your project. All units using SAT should have *SAT* in their **uses** clause.

C library:

SAT.h
Include file for using the C library.

SATC.π
The Sprite Animation Toolkit compiled to a Think C library. The extra "c" in "SATC.π" reflect that it is not exactly the same as SAT.lib. To be precise, it includes µRuntime.lib, which SAT2.lib doesn't. (If you don't know what it is, don't worry about it.)

CodeWarrior library:

SAT.p, SAT.h
Interface files.

SAT.lib
SAT as CodeWarrior library, useable from both Pascal and C.

Resources:

SAT blitters.rsrc
The direct-to-screen blitter resources. If you don't include them, your program will only use QuickDraw.

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. (CodeWarrior project files are not mentioned below. They generally are files with suffix .µp or .µc.)

<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.

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

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

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

<u>HeartQuest source files:</u>

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. SATRun 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.

The Pascal version included here has some modifications made by me, to allow hierarchical menus, some fixes for dialogs, handling Apple Events and MultiFinder events etc. It works well for me, but don't blame Paul and Owen if I've made mistakes. You can find docs and demos for the original Pascal version on the ftp site below. The C version is also modified, by Bob Schumaker. I havn't used it a lot myself.

You may want to consider version 3 of the package (in C, useable from Pascal - thanks Paul!). It is available by "anonymous ftp" from *ftp.primate.wisc.edu*. At that archive, you will also find the complete Pascal TransSkel version 2, with docs and demos, to complement the TransSkel.p file included here.

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.

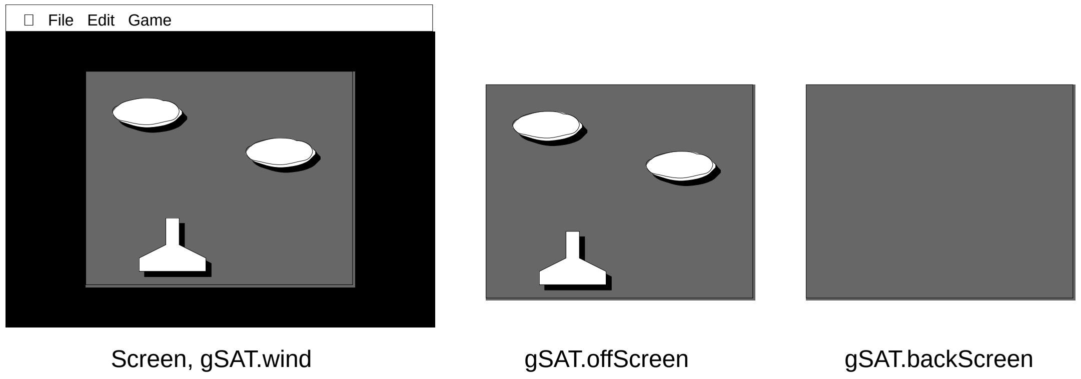
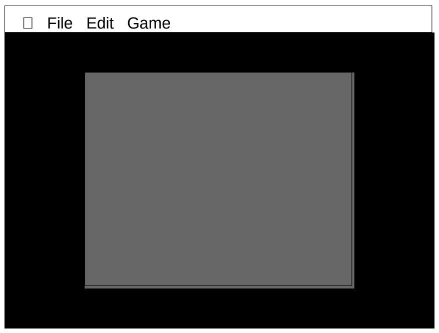


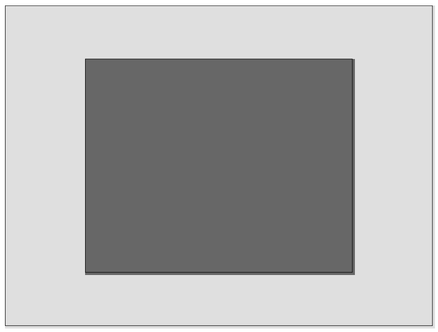
Figure 1. The screen, gSAT.offScreen and gSAT.backScreen, respectively.

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.

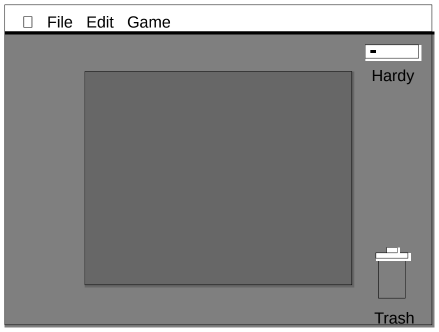
SAT by default uses the main screen. (There is a way to select another screen, but this feature is not tested much and still has a few problems.) Using the default setup, SAT fills the screen, excluding the menu bar, with a window. If the screen is bigger than the desired drawing area, SAT fills the rest of the window with a black border. All these features are configurable.



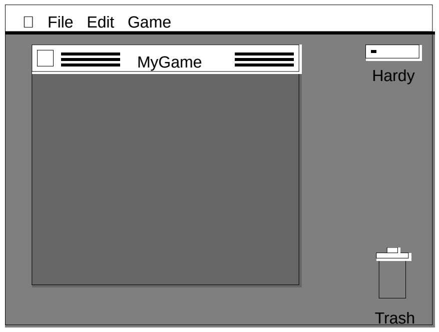
Default setting: menu bar, window fills entire screen, black borders.



Sample setting: menu bar hidden, custom border pattern.



Sample setting: no borders.



Sample setting: no borders, not centered, window can be dragged.

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

SAT can also produce asynchronous sound. This is by default in one channel only, but can be configured to use more. You may also use remaining channels for other tasks with your own sound routines. SAT uses Sound Manager if available, otherwise it switches to Sound Driver. When using Sound Manager, SAT keeps its channels 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 SATInit or SATCustomInit) 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 SATInit 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 SATNewSprite.

A main loop for the game. In SAT Invaders, this is the MoveIt procedure, where SATRun 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 SATRun 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.

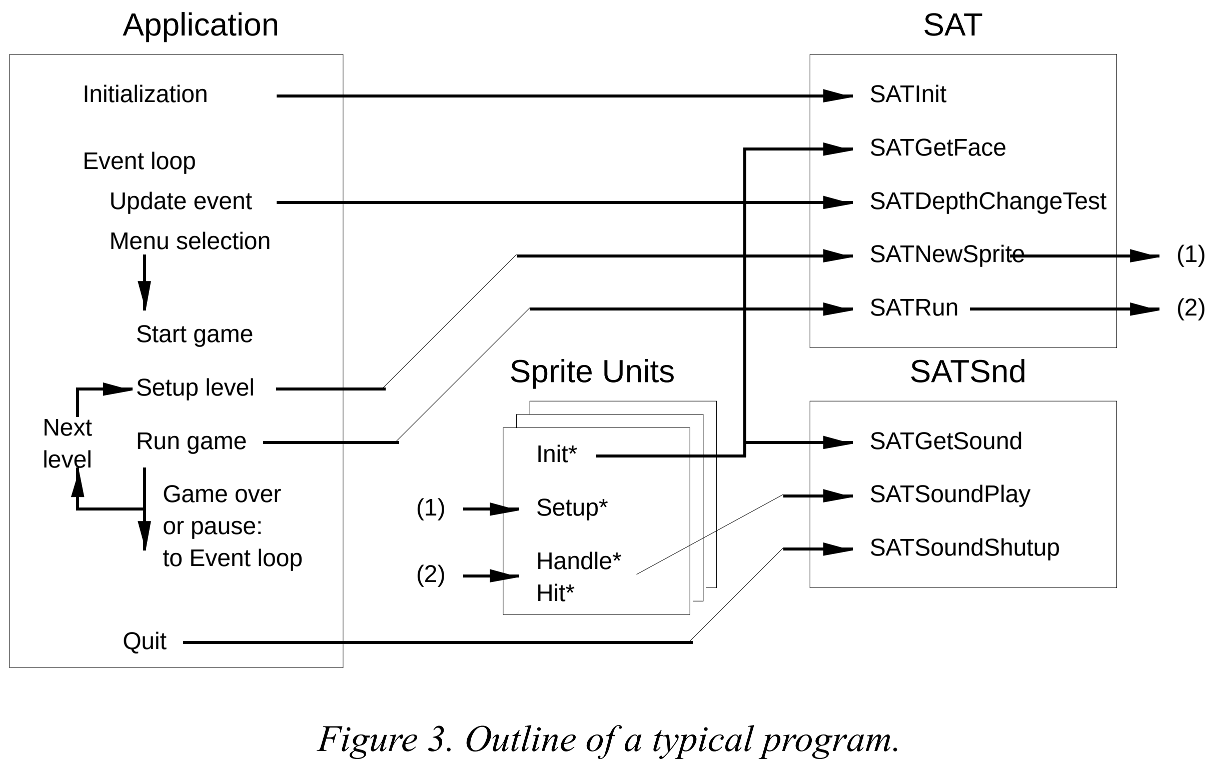


Figure 3. Outline of a typical program.

Many less important and/or more advanced procedures and functions are not shown in the Figure.

Initialization

Many of the routines in SAT demand that SAT has been initialized, either with SATInit or SATCustomInit. From version 2.2, a lot of routines (including SATGetFace and SATNewSprite) do not require that, but there is no point in calling SATRun without initializing first.

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 SATNewSprite (or SATNewSpriteAfter) 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 unit name for the cases shown, and naming the unit s* and the file s*.p. This is the convention used in the example programs.

*Init** should be called once when the program starts up. It is generally used to preload icons (faces) and sounds for the sprite. If a sprite only uses icons and sounds available from other units, you can omit it.

*Setup** is called from SATNewSprite when a new sprite is created. If no setup is needed, you can omit it and pass **nil** for it in SATNewSprite. Most sprites will need some setup. As a minimum, you should use it to assign the *task* field of the sprite, that is, a pointer to the Handle* routine! If you want a Hit* procedure, a pointer to it should be assigned to the *hitTask* field.

*Handle** is called once per frame for each sprite. It must always exist, even if it points to an empty procedure. This is because it is used to signal when a sprite is to be removed. Its pointer is stored in the *task* field of the sprite record. When *task* is set to nil (as is done in HandlePoints below), the sprite will be removed from the list and disposed.

Some sprite units have two or more Handle* procedures, for easy switching between different modes.

*Hit** is called when a sprite collides with another sprite. These procedures should manipulate the variables in the record pointed to by the SpritePtr to tell SAT where the sprite should be (the *position* field), how it should look (the *Face* field) and what to do in case of a collision.

Let us look at a simple example, the 'sPoints.p' sprite from HeartQuest, a stationary object flashing the number '50', used when the player takes bonus objects (as in Figure 4).

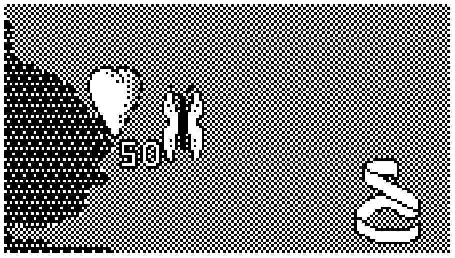


Figure 4. A snapshot from HeartQuest (old version), where an sPoints sprite has just been created when the butterfly took a bonus (sBonus) sprite.

Here follows the source for the sprite unit:

```
unit sPoints;
interface

uses
 SAT;

procedure InitPoints;
procedure SetupPoints (mp: SpritePtr);
procedure HandlePoints (me: SpritePtr);

implementation

var
 pointsFace, fPointsFace: FacePtr;

procedure InitPoints;
 var
  h: Handle;
begin
 fPointsFace := SATGetFace(132);
 pointsFace := SATGetFace(131);
end;

procedure SetupPoints (mp: SpritePtr);
begin
 mp^.face := pointsFace;
 mp^.mode := 0;
 SetRect(mp^.hotRect, 3, 4,29,32); {Not needed}
 mp^.task := @HandlePoints;
 {mp^.hitTask not assigned - not used.}
end;

procedure HandlePoints (me: SpritePtr);
begin
 me^.mode := me^.mode + 1;
 if (me^.mode > 32) or (band(me^.mode, 8) = 0) then
  me^.face := pointsFace
 else
  me^.face := fPointsFace;

 if me^.mode > 60 then
  me^.task := nil;
end;

end.
```

Now, let's have a look at what the standard routines are doing in this case.

InitPoints initializes the two icons (faces) that the sprite unit uses, loading them from 'cicn' resources with the *SATGetFace* procedure.
SetupPoints sets up the variables for a new sprite (the fields of the record to which the SpritePtr pointers refers). In this case, only *task* really needs to be set. In most cases you will also want to set the *hotRect* to something appropriate.

HandlePoints is called once for every frame. This is where the sprite is moved, animated, etc. In this case, we increment a counter (mode) to see when to remove the sprite instance, and change the *face* to make it flash.

To create a new *sPoints* sprite, you should use SATNewSprite:

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

The call might look like this:

```
   sp := SATNewSprite(0, hpos, vpos, @SetupPoints);
```

hpos and *vpos* are the coordinates where the sprite should appear.

kind is an integer that is put in the sprite's *kind*. I 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*.

The SATNewSprite procedure returns a pointer to the sprite, in case you need it. (Usually you don't.)

Here are some rules and tips on how to write a sprite unit:

– Always assign the *task* field in the Setup* routine. If you don't, the sprite will self-destruct.

– *task* is a pointer to the Handling* procedure.

– *hitTask* is a pointer to the Hit* procedure.

– To remove a sprite, set its *task* field to **nil**.

– The size of a sprite with respect to collisions is determined by its *hotRect* field. A filled 32·32 icon 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).

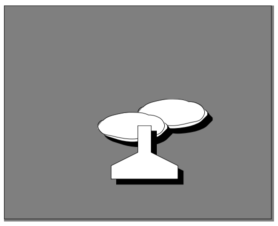


Figure 5. With standard sorting, closer to the bottom edge means foreground, closer to the top means background - an ordering that is often useable, but that can be changed as needed.

The sorting is a good reason for using SATNewSpriteAfter. 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 SATNewSpriteAfter is used. If SATNewSprite 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 common case where objects are either good or evil, you don't care what kind of enemy that hit you, and you don't care if objects pass over each other. For other cases, either use hit tasks, described below, or search through the sprite list yourself.
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 SATNewSprite 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.

Generally, the hittask will be called for both the sprites, but if you use *kForwardOneCollision*, only *one* of the two sprites is called, not both, even if they both have a hittask. This mode is intended for cases where all sprites can collide, and all sprites handle collisions in a similar way.

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 SATNewSprite or SATNewSpriteAfter.
• 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 change 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 dependant on the sorting chosen:
<u>All-to-all collision detection</u>: 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.
<u>Collision detection limited to "close" sprites</u>: 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:

<u>Non-rectangular sprites:</u>
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!

<u>Custom collision detection (Advanced programmers)</u>:
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-dimensional array describing a maze etc, and let the sprites check that array to see where they are allowed to go. (This is what I do in Bachman.) Use what seems best for your problem.

4. Bits and pieces

Notes and advice of various kinds.

Responding to update events

Most programs using SAT will not respond to events the normal way (using WaitNextEvent) while the animation is running, since it will make it too jerky of low-end Macs, but when the animation is not running, your program should respond to update events like any other Mac application. (Failing to handle update events is one of the most common beginner errors.)

When you get an update event to gSAT.wind, the simplest response is:

```
ignore := SATDepthChangeTest;
SATRedraw;
```

This will allow the user to change the depth of the screen while your program is loaded (note, however, that it may run out of memory if the user picks a big screen depth, in which case SAT will emergency exit), and update the window by copying from offScreen.

If you want your program to handle this in a more elegant way, you can check if the current screen depth is equal to gSAT.initDepth before calling it, change the cursor to a watch cursor before calling SATDepthChange, and do InitCursor afterwards. However, in my experience it is fully acceptable to set the cursor to a wait cursor in every update event, since it is reset to an arrow so quickly that you will never see the watch cursor if the depth wasn't changed.

Miscellaneous functions

SAT includes a number of utility functions that you may or may not need. Some are of a general nature, such as *DrawInt* and *DrawLong* (which simply does NumToString followed by DrawString), or *Rand* (generating random integers in a given range). Ignore them and use your own if they aren't what you need.

Sometimes you will have to move the cursor. For that, *SetMouse* is provided. Note that SetMouse depends on undocumented global variables, and may fail on some systems. I believe it already fails under A/UX. Use it when you must, but if possible, include a way to disable its use.

SAT does not by default hide the menu bar, but provides functions for doing it, namely *SATShowMBar* and *SATHideMBar*. I recommend that you don't hide the menu bar until rather late in your development, when most bugs are already fixed.

For using patterns, the Pattern Utilities are provided. The point with these is that they allow you to use a single "ppat" resource, and use that on Macs with or without QuickDraw. A nice pattern is often as good as a huge picture, and takes much less disk space, so using patterns should be encouraged. Bug note: *SATDisposePat* is not guaranteed to succeed in disposing the pattern (as documented in IM5). Unnecessary allocation and disposing of SATPatterns may therefore cause a memory leak.

ReportStr and *QuestionStr* are two functions that display a message in an alert box. *QuestionStr* gives two buttons, yes and no, and returns true if "yes" was pushed. They both use the more general function *SATFakeAlert* (which is my own version of a function I found in TransSkel).

SATSetStrings is a function that sets the error messages and button names used by SAT. Use it if you make a program in another language (swedish, french, japanese…). HeartQuest uses it in order to store all text in resources, so it can easily be translated without recompiling.

Modifying the background

Many games can be done over a static, never changing background. However, many games will demand the background to change, not just by replacing the backdrop with another, but making local changes. For example, if we want to add some barriers in SATinvaders, we could do that by drawing them in the background and erasing parts of them (drawing the background color on them) when they are hit by shots.

It is possible to make such changes by doing a SetPort to gSAT.backScreen, draw what you like, and then CopyBits it to the screen. However, that will often interfere with sprites, causing flicker. With the function *SATBackChanged*, you can send a Rect to SAT, which will then update the screen when the time comes.

A related function is *SATPlotFace*. It takes an icon in the form of a FacePtr (that is, the icon is preloaded to a 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 *SATNewFace*, *SetPortFace*, *SetPortMask* and *SATChangedFace* are provided. You create a blank face with *SATNewFace*. 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 SATChangedFace. (BUG NOTE: For now, you must always call *SATChangedFace* some time before using a face created by *SATNewFace*. If you don't, you might get a system error.)

A face created by *SATNewFace* rather than *SATGetFace* 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.

A related (preminimary) new demo if SATMinimalX, which demonstrated how you can load a whole set of faces from a PICT.

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 SATCustomInit, 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 other two are problems that can be solved by proper design and VBL synching.

Note that scrolling games must use a fairly small visible area to be fast enough. You might want to demand or recommend that your game is run in 4-bit or even 1-bit color on not-so-fast Macs. However, running in these bit depths means that you must use CopyBits to get arbitrary scrolling. With SATCopyBits, you have limitations that you must take into account. In 4 bits, you can't scroll to odd horiontal positions, and in 1-bit, you can only scroll to horizontal positions divisible by 8. (This means that vertical scrolling is easier to use than horizontal.)

An interesting alternative is step-scrolling. This is what, among many others, the game Oxyd does. In this case, we don't update the entire game window for every frame, since we only scroll when the "hero" sprite (or whatever) gets too close to the edge. This can be done with SAT by changing the origin and the globals gSAT.ox and gSAT.oy (two globals telling the offset from the topleft corner of the screen to the topleft corner of the animation area - usually of absolutely no interest to you). This is demonstrated in the demo **StepZkrolly** (which resembles Zkrolly). Note: When step-scrolling, you should either use "safe" mode (SATRun(false)) or fill the screen completely in the direction it can scroll. SAT will clip sprite to the screen borders, but it will not clip to the window borders when the offscreens are bigger.

You can also avoid the true scrolling and "fake" it by using a smooth background with a bunch of sprites over, so it appears to be scrolling. This is somewhat a waste with SATs capabilities (wasting time restoring a background that doesn't exist), but might be ok for some games, and can be done with the standard way to use SAT.

Mac-friendly programming

When SAT takes care of some (most?) of the animation issues for you, what more is there to think about other than to design a spectacular game? I'll give a few ideas of how I think a Mac game should be.

Behave like a normal Mac application when the game/animation isn't running. Use standard menus (possibly hidden when the game is running), allow background processing (by calling GetNextEvent/WaitNextEvent often), allow switching to other applications. See the more advanced sample programs, SAT Invaders and HeartQuest.

Don't get in the way of the user! gSAT.wind usually covers the entire screen, which is nice when the game is running. However, if the user switches to Finder, gSAT.wind must be hidden or resized in order to allow access to the desktop (at least under sys 6, where you can't hide it from the Finder). It must also call SATSoundShutUp (SAT's sound termination procedure) at least when switching out (but usually immediately after pausing or ending the game). See the sample programs.

Design for all Macs, not just your own. The **timing** should be done after the system clock, as in the examples. No silly for-loop for delaying, please – TickCount is pretty good and easy enough.

Don't rely on features in a specific system version (i.e. Sys 7) if you don't have to. If you do, at least **check the system version -** it's much nicer than crashing or quitting unexpectedly. Even better, if you rely on specific features like QuickTime, use Gestalt to make sure they are around.

Design it either to work on **all screen sizes** or on the **9"** (Classic-sized) **screen**. That way, all Mac users can use it. Use the global variables gSAT.offSizeH and gSAT.offSizeV for checking game area bounds rather than hard-coded numbers. You may want to rescale your sprite faces depending on the screen size used.

The GDevice business

As long as you use the "fast mode", this section is of no interest to you. It is only a question of getting good performance in "safe mode", that is, when using QuickDraw instead off the custom blitters in SAT.

Each of our two offscreen grafports (gSAT.offScreen and gSAT.backScreen) have a sidekick, its GDHandle. QuickDraw, and CopyBits in particular, works much better when the right GDevice is set. I have tried to hide this as far as possible in SAT, giving you backwards compatibility without bothering with colorFlag or other dependencies.

Some functions must have the GDHandle together with the GrafPtr. When running on old Macs, the GDHandle will be ignored. If a nil handle is passed, the drawing might be slower, but it should still work. (E.g. SATPlotFace and SATCopyBits.)

The rule is: *if you call a SAT function where you pass a GrafPtr, and that function wants a GDHandle, send the GDHandle that corresponds to the destination port..*

When using QuickDraw directly, though, you must set the port and device before drawing. The typical case where your program does much drawing using QuickDraw is when setting up, drawing the background. To let you forget about GDhandles and whether you run in color or not, the routines *SATSetPortOffScreen*, *SATSetPortBackScreen* and *SATSetPortScreen* are provided. They are simply SetPort to the port in question plus a SetGDevice if applicable. The typical use is:

```
{Set up a level, draw objects in backScreen}
SATSetPortBackScreen;
…do all your drawing here…
SATSetPortScreen;
```

For those interested: this is equivalent to:

```
SetPort(gSAT.backScreen);
if gSAT.colorFlag then
 if gSAT.backScreenGD <> nil then
   SetGDevice(gSAT.backScreenGD);
…do all your drawing here…
SetPort(gSAT.wind);
if gSAT.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*. It is also possible to reserve channels for special use (which means that SAT won't direct sounds to that channel except when you explicitly ask for it).

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. (Older versions of SAT: Have you *initialized SAT* before trying to load faces?) 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.) Think Pascal: Do your callback routines have the *appropriate parameters*?

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 SATRedraw, 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 SATRedraw does, so you are losing nothing by replacing it.

You can also use a smaller window in the first place, which holds only the drawing area and no borders at all.

– After initializing SAT to a part of my window, all my own drawing goes to the wrong place!
• SAT changes the origin of your window. My best advice is to use the new coordinate system. The new origin is in the upper left corner of the area used by SAT, which is not necessarily the same as the upper left corner of the window.

– Can SAT be used in a draggable window?
• For now, avoid it if you want fast animation. It can be done if you update the variables ox and oy appropriately – but SAT has, for speed, limited checking against screen borders. It is best avoided altogether. It works just fine when using "safe" animation (i.e. SATRun(false)).

– How do I change the background? If I draw a new image in gSAT.backScreen, SAT sometimes overwrites it with the first PICT I gave it.
• If you change the background (drawing another PICT resource in backScreen and copying it to OffScreen), you should also change the globals *SATpict* and *SATbwpict*, since they tell SAT where to go for the new PICT if it has to redraw them (i.e. when the screen depth changes). Set them to zero (0) if you take care of the drawing yourself.
Drawings not based on PICTs are your own responsibility to make and update. SetPort to backScreen and draw what you need (possibly using SATPlotFace and SATBackChanged).

– Can I resize my sprites?
• Yes, by using the new calls SATNewFace, SATSetPortFace, SATSetPortMask and SATChangedFace, you can create faces with other size than the icon you want to draw in it. This is for advanced programmers! See the Collision /// demo.

– I have an older version of Think Pascal/C. Can I use SAT with it?
• With Pascal, it should work, though you will have to figure out what files to put in the projects and in what order, so I don't recommend it. For C, no - the C library is incompatible with versions prior to v 5, but versions 5 to 7 work. CodeWarrior: I don't think you can use SAT.lib with any version prior to 4.5, or at least 4, but who would need it?

– I have a 68000-based Mac. ResEdit won't edit the "cicn" resources I must use!
• Use the utility program "ICN#->cicn", which is part of SAT. It converts ICN# resources to b/w cicns. See the list of files above. (Note: Even though you will get something on screen on a 68000-based Mac even if you don't have any mask, you shouldn't leave the masks blank, since they are necessary for color Macs.)
You can also choose to use ICN# resources instead, but then you must write a procedure for loading faces from them. See FaceFromPICT.p. The drawback with this approach is that your program can then not be colorized later.

– Can I do animation over a scrolling background with SAT?
• Yes. See the appropriate section above, and the demos Zkrolly and StepZkrolly.

– But scrolling like in Zkrolly is darn slow for large areas! Can I step-scroll like Oxyd does?
• Sure - this is what StepZkrolly does. It has a setup similar to Zkrolly, but scrolls only when needed. To make SAT redraw correctly, you must change its origin. When running in "safe" mode (i.e. SATRun(false)), you can change the origin of gSAT.wind with SetOrigin. When running in "fast" mode (i.e. SATRun(true)) you must also set the gSAT.ox and gSAT.oy globals appropriately. I only recommend the "fast" mode in a case like this as long as you use as much of the screen as possible.

– One of my sprites draws some garbage where it shoudn't!
• SAT expects your 'cicn's to be "clean", with no drawings where the mask is zero. (This causes problems only when running in b/w.) If you use sprites with odd sizes, ResEdit may leave some garbage in parts that you don't see when editing it. You may have to clean the cicn in the hex editor. [This problem should be fixed in 2.0b6.]
SAT also expects the cursor to be hidden, or you can get so called "mouse droppings" when the cursor is moved over a sprite - intentional problem in SATminimal! This is avoided by using HideCursor or ShieldCursor.
[Bug note: Up to version 2.0b5, SAT draws sprites with certain widths incorrectly when drawing directly to screen. Only sprites with widths divisible by 8 always worked correctly. This problem should be fixed from 2.0b6.]

– Will my program work on all Macs?

• SAT by itself supports as many Mac models and systems as possible, and tries to help you to do so too. When you call SAT, SAT does its best to switch to routines that will work on the Mac it runs on. The most likely case where you must do some checks yourself is when using QuickDraw to draw backgrounds etc. Test the globals colorFlag and gSAT.initDepth to determine what routines to use.

The only Macs I know where SAT fails, is Macs with SoundManager older than version 3, equipped with certain accelerator boards. On those Macs, my workaround for the bugs in the old Sound Manager are not enough, so they run a risk of crashing when playing sounds. Not much we can do about it, really, except keeping silent.

– I want to dispose of everything to set up SAT differently.

• You may use SATKill for this. Note, however, that faces and sounds are not disposed by that call, but must be disposed of separately. Sprites 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 ///.

```
SATKill;                                    {Nuke the old environment.}
DisposeWindow(gSAT.wind);                           {Nuke the old window}
SetRect(theArea, 70, 20, 220, 380); {The new area}
SATCustomInit(128, 129, theArea, nil, nil, true, true, false, true, true);
{Make new sprites here!}
ShowWindow(gSAT.wind);                      {Show the window.}
SelectWindow(gSAT.wind);
SATRedraw;                                          {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 in a Classic-sized area, but may need more memory if run in bigger screen depths or in bigger areas (e.g. full 14").

– 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 SATRun repeatedly. SATRun 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 SATRun, 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.

• If this isn't just some common bug like writing outside an array or following a nil pointer, 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 SATGetFace 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 only once in both cases. (In older versions, it loads several times in both cases.)

– 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 SATInit (pictID, bwpictID, Xsize, Ysize: integer);
function SATNewSprite (kind, hpos, vpos: integer; callback, setup, hittask: ProcPtr): SpritePtr;
function SATGetFace (resNum: integer): FacePtr;
procedure SATRun(fast:Boolean);
procedure SATRedraw;
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: SpritePtr);
begin
 me^.position.h := me^.position.h + me^.speed.h;
 me^.position.v := me^.position.v + me^.speed.v;
 if me^.position.h < 0 then
  me^.speed.h := abs(me^.speed.h);
 if me^.position.h > gSAT.offSizeH - me^.hotRect.right then
  me^.speed.h := -abs(me^.speed.h);
 if me^.position.v < 0 then
  me^.speed.v := abs(me^.speed.v);
 if me^.position.v > gSAT.offSizeV - me^.hotRect.bottom then
  me^.speed.v := -abs(me^.speed.v);
end;
```

– I want the source, for learning how you do it.
• So you want to do it all over again, re-inventing the wheel? Believe me, the sources to SAT isn't the right place to start. Do you want to wade through several hundred kilobytes of code, just to end up doing what SAT already does? That's a bad idea. If you want to learn, you are better off with a small demo. I've made one just for you: it's called **Offscreen Toys**, and demonstrates sprite animation in only 25k of source code, and is well commented. (Note: OffScreen Toys SAT is a SAT-using look-alike. Please do not confuse Offscreen Toys with Offscreen Toys SAT!)

– I want the source to make some changes/port to another platform.
• OK, this is a valid reason, which is why I make the full source code available – separately, for personal use only (not for free distribution), for the modest fee of $20. Be warned, though: hacking in changes might be harder than you think. The code is not intended as a tutorial, so it is structured for my needs, and commented for my needs. However, if you make modifications, especially if you plug in better direct-to-screen routines (which is quite possible), please share it with the rest of us. How about sending it to me so there can be a single "official" version?

Don't bother making a C port of the library source. The library can be used from C, and Pascal is just as fast anyway. It's a waste of time.

Oh, BTW… if all you want is to plug in some custom blitter of your own, that is possible without the full source code. You can, for example, change SATs pointer to CopyBits (This is a ProcPtr named gCopyBitsAddr, which must point to a routine which takes the same parameters as CopyBits. This ProcPtr is called when SAT runs in in "safe" mode.)

– I use Think C version 6 / Symantec C++ version 6, and I get link errors.

• It's all Symantec' fault. See below, "The C interface". However, I believe I have found a way around it, so the problem should be gone now (from SAT 2.0b8).

The C interface

So far, I have assumed that you are using SAT from Think Pascal. There is, however, a C interface, in the form of two libraries (SAT1.π and SAT2c.π) and a header file (SAT.h). Most demos are included in C.

You can use the SATC libraries just like any C library. Put the "SAT C Lib *f*" with the library and the header file at the appropriate place, i.e. in your Think C folder. Include the library in your project and #include SAT.h in appropriate source files. I will not waste more space on these trivial things. SAT.h describes the programmers interface, and using the rest of this manual shouldn't be much harder than for a Pascal programmer.

A few notes, though:

• All callback functions, functions that SAT calls using procedure pointers you provide, **must** be declared "pascal" (e.g. pascal void HandleSprite()). With a copy of SAT.h that is new enough, and pointer type checking on, this should be no problem.

• Think Pascal does all standard initializations automatically. Think C does not.

In older versions you may have had problems using SAT from Think C v 6 (Symantec C++ v 6), e.g. a link error telling that MaxApplZone is undefined (which is nonsense - it is a standard toolbox function that has been around since 1984). This is due to a bug in the compiler (or perhaps linker), which fails to find MaxApplZone when used from the initialization procedure in µRuntime.lib, a Think Pascal library I must include (which is also due to poor design by Symantec).

The fix was almost embarrasingly simple. I opened SAT2c.π with Think C v5 and removed the segment %_TOOLBOX. This isn't exactly what I'd call a convincing fix, but it seems like that's what we need until Symantec gets its act together and makes products that are compatible with each other.

So the message to you SC++ users is: it **works**. (If it doesn't – if you get a link error telling that MaxApplZone is undefined – I might have rebuilt the libraries without remembering to take out that segment. Tell me and I'll fix it.)

How about Pascal vs C with CodeWarrior? Fortunately, MetroWerk seems to have designed their compiler with more communication between the Pascal and C teams, so we can use the same library for both.

The C++ interface

Can we use SAT from C++? Well, at the time of writing, I don't think so, but we (that means myself and Nathaniel Woods) are working on it - Nathaniel by telling me what is needed for making SAT useable from C++, and I by trying to provide such features. The following calls and features are intended for making this C++ interface possible:

Routines (in C syntax, since C users are those who are likely to need them):

```
/* New procedures, EXPERIMENTAL, intended for a C++ interface */
		pascal SpritePtr SATNewSpritePP(SpritePtr, Ptr, short, short, short, TaskPtr);
		pascal void SATCopySprite(SpritePtr, SpritePtr);
		pascal FacePtr SATNewFacePP(Rect*, Ptr);
		pascal FacePtr SATGetFacePP (short, Ptr);
		pascal void SATCopyFace(FacePtr, FacePtr);
		pascal void SATDisposeFacePP (FacePtr);
```

SATNewSpritePP, SATGetFacePP and SATNewFacePP are variants of SATNewSprite, SATGetFace and SATNewFace where you can provide a pre-allocated storage. NOTE: With SATGetFacePP, a new face is always created if you pass a pre-allocted storage, while SATGetFace will check the face list to see if the face was already loaded.

SATDisposeFacePP is a variant of SATDisposeFace where the Face record is not disposed. (SATKillSprite can be told not to dispose by using a destructTask.) SATCopySprite and SATCopyFace copies a sprite or face to a new structure, which is linked into the lists as a new object. The storage must be pre-allocated but must **not** be an existing sprite or face.

Another change that was made partially for this was the new auto-initialization feature, that makes SAT initialize itself if you call certain routines (e.g. SATNewFace, SATNewSprite, SATPlotFace…) before SAT is initialized.

I don't expect this C++ "friendliness" to be useful before Nathaniel's interface is released, but hopefully, that can happen soon.

Writing your own blitters

The new system with the blitters in resources (introduced in SAT 2.3) gives you some new possibilities:

• You can remove any blitters that aren't needed to save space.

• You can plug in replacements for the existing blitters.

• You can make new blitters for depths that are not supported by the existing ones. SAT will automatically recognize blitter resources for 1, 2, 4, 8, 16 and 32 bits (b/w, 4 , 16 , 256, thousands and millions of colors, respectively).

A blitter resource is a resource of type 'RBlt' (rectangular blitter) or 'MBlt' (masked blitter). The resource ID corresponds to the depth for which it is intended. ID #0 is used for blitters for non-color-QuickDraw Macs (Plus, SE, Classic, PB100…).

I will document the programming interface for SAT blitters in a future version. For the moment, I am not sure whether the interface I have is perfect, so it might change.

5. The programming interface

SAT Data types

```
type

{Information about a "face", i.e. a color icon. You hardly have to bother about this data type.}
 FacePtr = ^Face;
 Face = record
   colorData: Ptr;
   resNum: integer;
   iconMask: BitMap;
   rowBytes: integer;
   next: FacePtr;
   maskRgn: RgnHandle;
  end;
```

The field *colorData* holds the image. This is for internal use, since it has different formats depending on depth.
The integer *resNum* tells the resource id of the cicn the face was loaded from.
The BitMap *iconMask* is usually a straight copy of the iconMask in the cicn. It is a valid BitMap that you can access as appropriate.
The integer *rowBytes* tells the rowBytes of the data in colorData.
The next face in the face list is found in *next*.
The mask region *maskRgn* is a region created from the bitmap iconMask. It is only used by SAT when SAT runs in "safe" mode, but can be useful for collision handling (as in Collision///).

```
{Information about a sprite, that is one object on the screen.}
 SpritePtr = ^Sprite;
 Sprite = record
{ Variables that you should change as appropriate }
   kind: Integer; { Used for identification when using KindCollision. >0: friend. <0 foe }
   position: Point;
   hotRect, hotRect2: Rect; { Tells how large the sprite is }
{hotrect is set by you. hotrect2 is set by SAT - forget about it}
   face: FacePtr; { Pointer to the Face (appearance) to be used. }
   task: ProcPtr; { Callback-routine, called once per frame. If task=nil, the sprite is removed. }
   hitTask: ProcPtr; { Callback in collisions. }
   destructTask: ProcPtr; { Callback when the sprite is disposed. Usually nil. }
   clip: RgnHandle; { Clipping region for this sprite - usually nil. }
{ SAT variables that you shouldn't change: }
   oldpos: point; { The 'task' routine is not allowed to change this! }
   next, prev: SpritePtr;
   r, oldr: Rect;
{ Variables for internal use by the sprites. Use as you please. }
   layer: integer; {For free use, or for sorting.}
   speed: Point; { Can be used for speed, but not necessarily. }
   mode: integer; { Usually used for different modes and/or tells what face to use next.}
   appPtr: Ptr; {Pointer for use by the application - i.e. pointer to extra data}
   appLong: Longint; {Longint for free use by the application.}
  end;
```

When a sprite is created, the fields *kind* and *position* are set according to parameters to SATNewSprite or SATNewSpriteAfter. 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. See SATKillSprite.
clip is a new field, introduced with SAT 2.3. It lets you specify a region with which to clip the sprite. This is extremely useful when you want a sprite to move behind large, static objects. However, when clip is not nil, the sprite will be drawn with QuickDraw, so it will be a little slower.
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 SATRun (i.e. a handling or hit procedure. If you make your own sorting mechanism, run it outside SATRun.
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!

Global variables

Up to 2.0b5, SAT had a lot of global variables, most of them carelessly named. Nowadays, most of SAT's environment is stored in a global record, gSAT. This structure holds some fields that are of big interest to the programmer, but also quite a few of no or minor interest (that were previously hidden from the programmer).

The main point with gSAT is that it collects most global variables in one place, both eliminating the risk for name collisions and making the code easier to understand. Another point with it is that it is a preparation for making it possible to have SAT operating in two or more different environments (call it "worlds" if you like) simultaneously, or switching between them.

Below, the global variables of interest to you are described. There are several others, but don't worry about them.

```
gSAT.offScreen, gSAT.backScreen: GrafPtr;
gSAT.offScreenGD, gSAT.backScreenGD: GDHandle;
```

offScreen and *backScreen* point to the two offscreen GrafPorts used by SAT. They are initialized with *SATInit* (see below). *offScreen* is a copy of the screen, while *backScreen* is the background image, over which the sprites are animated. You can SetPort to them (see also SATSetPortOffScren and SATSetPortBackScreen below) and draw the desired images. Every time you want to change the background image, make the change in backScreen, and notify SAT with SATBackChanged.
When running on color capable Macs, *offScreenGD* and *backScreenGD* are the offscreen graphic devices for each of our two offscreen GrafPorts. Certain calls (SATPlotFace, SATCopyBits) require both the GrafPtr and the GDHandle.

```
gSAT.offSizeH, gSAT.offSizeV: integer;
```

offSizeH and *offSizeV* contain the size of the drawing area. Use them when calculating drawing positions etc, but you should avoid changing them when SAT is already initialized.

gSAT.pict, gSAT.bwpict: integer;

These two integers point to two PICT resources that should be used as the background, in color and b/w, respectively. They are set by the SATInit, SATCustomInit and SATDrawPICTs calls. The value zero means no PICT.

gSAT.wind: WindowPtr;

gSat.wind is a pointer to the window SAT uses. SAT expects this window to be frontmost when drawing direct-to-screen.

gSAT.sRoot: SpritePtr;

sRoot is a pointer to the first element in the sprite list. You rarely have to access it directly. If you do, do it with caution. Be care ful when removing sprites from the list yourself (both by modifying the linked list and using SATKill). If you do, they will not be erased properly.

gSAT.anyMonsters: Boolean;

anyMonsters is a flag set by SAT, that you may optionally use to detect when a level is completed and similar tasks. It is false when there are *no sprites with kind* < *-1* in the list. It is <u>only</u> working when KindCollision is being used.

gSAT.initDepth: integer;

gSAT.initDepth gives the screen depth with which SAT is loaded. This is usually the same as the screen depth. You can inspect it if you want to warn the user about using a bit depth that is not supported, to switch to b/w drawing when approprite, or use it when making additional offscreen pixmaps, if needed.

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 (from 2.2 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. SATSetSpriteSize sets this global.

colorFlag: Boolean;

colorFlag is a flag (from 2.2 part of gSAT) that simply tells whether color QuickDraw is available or not. It is set by a call to SysEnvirons. I find this flag to be nice to have around (to say the least - SAT has lots of glue functions that rely on it).

gSATSoundErrorProc: ProcPtr;

gSATSoundErrorProc is a procedure pointer (not part of the gSAT record) pointing to a function that you want to have called if SAT's sound routines encounter an error. The procedure should take an OSErr as parameter.

SAT procedures

<u>Easy initialization:</u>

function SATInit (pictID, bwpictID, Xsize, Ysize: integer): WindowPtr;

SATInit does all of the initializations needed for SAT. It initalizes the internal lists and the sound package, creates the SAT window (returning a pointer to it) and, if you pass pictID and bwpictID other than 0, draws the appropriate PICT in the offscreen buffer. The window (the return value, also in the global pointer *gSAT.wind*) fills the main screen, and use a drawing area that is Xsize*Ysize pixels. If Xsize*Ysize can't fit on the screen, the screen size is used. (Classic size, excl. menu bar, is 512*322 pixels.)
After initializing, SATInit will also show gSAT.wind and update it.

<u>Customized initialization:</u>

procedure SATCustomInit (pictID, bwpictID: integer; SATdrawingArea: Rect; preloadedWind: WindowPtr;chosenScreen: GDHandle; useMenuBar, centerDrawingArea, fillScreen, dither4bit, beSmart: Boolean);

SATCustomInit is a more powerful version of SATInit, for programmers with other needs than the default. Use it if you need any of the following:

• A drawing that isn't centered.
• A window that doesn't cover the entire screen.
• Attach SAT to an existing window.
• Hide the menu bar while animating.
• Run the animation on a screen other than the main device.
• Disable SATs habit of cutting down the offscreens to what it thinks you should have (making it fit on the screen and in your window, and have horizontal borders on coordinates divisible by 8).

The integers *pictID* and *bwpictID* work as in SATInit.

SATdrawingArea is a rectangle that specifies where the drawing area should be. If preloadedWind is nil, this is in global coordinates, otherwise in coordinates local to preloadedWind. The rectangle is the *maximum* area you can get. If beSmart is true, it will be clipped down to fit the screen and PreloadedWind (if any). The left and right coordinates will also be adjusted to coordinates divisible by 8.

The WindowPtr *preloadedWind* points to a window to use rather than creating a new one. Note that you are responsible for this window to be a color window on color Macs and an old-style window on old Macs. Pass **nil** for preloadedWind if you want SAT to create it.

The handle *chosenScreen* specifies a screen (device) 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 SATHideMBar and SATShowMBar.) 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.

Unlike SATInit, SATCustomInit will not show gSAT.wind. Once you switch from SATInit to SATCustomInit, you must do that yourself. A call to SATInit is equivalent to the following snippet:

```
procedure SATInit (pictID, bwpictID, Xsize, Ysize: integer);
 var
  frameRect: Rect;
begin
 SetRect(frameRect, 0, 0, Xsize, Ysize);
 SATCustomInit(pictID, bwpictID, frameRect, nil, nil, false, true, true, true, true);
 ShowWindow(gSAT.wind);
 SelectWindow(gSAT.wind);
 SATRedraw;
end;
```

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

SATConfigure lets you set certain parameters that affect SAT's behaviour. It usually should be called before *SATInit* or *SATCustomInit*, 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.
kForwardOneCollision: Seach forward in the sprite list, and report collisions with the HitTask procedure – but only to *one* of the two colliding sprites!
kBackwardCollision: 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 SATGetFace (resNum: integer): FacePtr;
```

SATGetFace (formerly called *LoadIcon)* loads the 'cicn' resource with number *resNum*, and returns a pointer to the resulting FacePtr. This pointer can be used for the *face* field in the sprite records. This routine is generally used from the setup procedure in all sprite units.
NOTE: This routine was recently changed to avoid loading faces several times (which may happen sometimes when several units use the same icons). This feature can be disabled by using SATGetFacePP with a pre-allocated storage, which forces a new face to be created. (See the C++ interface.)

```
procedure SATDisposeFace (theFace: FacePtr);
```

SATDisposeFace removes the Face from the list of Faces and frees up the memory used by it. Use it to free up memory when you no longer need a Face. (Most games don't need it.) [WARNING: This routine is not thoroughly tested.]

```
function SATNewSprite (kind, hpos, vpos: integer; setup: ProcPtr): SpritePtr;
function SATNewSpriteAfter (afterthis: SpritePtr; kind, hpos, vpos: integer; setup: ProcPtr): SpritePtr;
procedure SATKillSprite (who: SpritePtr);
```

The two routines *SATNewSprite* and *SATNewSpriteAfter* add new sprites to the list of animated sprites. Choose *SATNewSpriteAfter* if you want it in a special place in the sprite list. The parameter *setup* points to a routine that will be called right after the sprite has been allocated. That routine should, at a minimum, assign the *task* field of the sprite.
SATKillSprite removes a sprite, but does not guarantee that it is erased properly from the screen. Use it only when cleaning up the sprite list between levels and similar situations. (In other cases, set its task to **nil** to tell SAT to remove it.)
SATKillSprite takes out the sprite from the sprite list. After that is done, it will *either* call the destructTask, if one is provided, *or* dispose of the SpritePtr. Note that this implies that the destructTask is responsible for disposing if it exists!
A little hint: to clear the entire sprite list (which is often desired when starting new games or new levels), you can do:

```
 while sRoot <> nil do SATKillSprite(sRoot);
```

followed by some kind of update to remove the "dead" images on the screen and gSAT.offScreen.

Running the animation:

procedure SATRun(fast:Boolean);

SATRun processes one frame of animation. Pass *true* or *false* depending on whether you need high speed (writing directly to screen memory) or code that works on as many Macs as possible (drawing with ordinary Toolbox calls).

Drawing:

The following routines are often useful for drawing things in other ways than SATRun does. This may include modifying the background during animation, but also to draw game layouts etc between "levels". For simple sprite animation, SATRun does all drawing!

procedure SATPlotFace (theFace: FacePtr; theGrafPtr: GrafPtr; theGDevice: GDHandle; where: Point; Fast: boolean);
procedure SATPlotFaceToScreen (theFace: FacePtr; where: Point; Fast: boolean);

SATPlotFace draws the icon stored in a *Face* structure in the GrafPort *theGrafPtr*. The GrafPort in question must be at least as big as the drawing area, and must be a CGrafPort if the Mac is color capable.
The normal use for SATPlotFace is to draw Faces on backScreen, in order to modify the background. If you pass **nil** for *theGrafPtr*, SATPlotFace assumes that you want the drawing in backScreen, plus calls SATBackChanged for you. (See below.)
theGDevice is the device returned by SATMakeOffScreen, or in the case of the standard buffers, offScreenGD or backScreenGD.
SATPlotFaceToScreen is a variant for drawing to the screen.

procedure SATCopyBits (src, dest: GrafPtr; destGD: GDHandle; srcRect, destRect: Rect; fast: Boolean); [OBSOLETE]
procedure SATCopyBitsToScreen (src: GrafPtr; srcRect, destRect: Rect; fast: Boolean); [OBSOLETE]

SATCopyBits and *SATCopyBitsToScreen* are not significantly faster than CopyBits, esp. not for large areas. In order to simplify the blitter interface, I'm taking those calls out. SAT will, for some time, still support them, but only by calling CopyBits.
Will anyone REALLY miss them? If you do, I might put them back in.
~~*SATCopyBits*~~ and *SATCopyBitsToScreen* are two replacements for CopyBits, possibly faster than the normal CopyBits (especially for rather small areas). With fast set to false, the normal CopyBits is used.
With fast set to true, the area being copied is expanded to full bytes, which means that, in b/w, if the chosen area does not have horizontal boundaries divisible by 8, a few extra pixels can be included per row. In 4-bit, one extra pixel is copied per row when boundaries are on odd coordinates. (This is usually no problem.)
Also note that SATCopyBits won't do arbitrary shifting! This means that you can't copy with arbitrary sideways shift if you run in less than 256 colors. In 16 colors, you are limited to every other position (i.e. you can't shift an odd number of pixels), in b/w, you can only shift a multiple of 8, and on Macs with 68000, you can only shift a multiple of 16!
The GrafPtrs src and dest are generally offScreen, backScreen or some additional offscreen you have made yourself. The rectangles srcRect and destRect specify the source and destination area. If these areas are not the same size, the width and height of destRect is used.
[WARNING: Currently, you are responsible for making sure the copied area fits within the destination!]

procedure SATBackChanged (r: Rect); {Tell SAT about changes in backScreen}

Use *SATBackChanged* when you have modified the background (backScreen) to tell SAT to update that part. SAT wil then update it in the proper time, during SATRun.

procedure SATGetPort (var port: GrafPtr; var device: GDHandle);
procedure SATSetPort (port: GrafPtr; device: GDHandle);

SATGetPort and SATSetPort are intended for saving and restoring the port and device. They do a GetPort/SetPort, plus a GetGDevice/SetGDevice if we are running on a color capable Mac. Three special cases of SATSetPort follows:

procedure SATSetPortOffScreen;
procedure SATSetPortBackScreen;
procedure SATSetPortScreen;

All these three calls do a SetPort, plus a SetGDevice if we are running in color. Use *SATSetPortOffScreen* or *SATSetPortBackScreen* instead of SetPort if you want fastest possible speed when drawing in any of the offscreen buffers using normal QuickDraw calls (especially if you use CopyBits). Use SATSetPortScreen to restore (or better, save the port and device and restore to what they were). However, simple SetPort calls will work if you are not in a hurry.

Maintainance:

function SATDepthChangeTest: boolean;

SATDepthChangeTest should be called either repeatedly or before each game starts. It checks if the screen depth has changed, and if it has, it re-initialize the offscreen buffers and the face list. This should only happen after a pass through an ordinary event loop. If SATDepthChangeTest returns true, the depth has changed. In such a case, the game window needs to be updated (e.g. with SATRedraw) and any drawing you do yourself offscreen must be redrawn.
A very good time to call SATDepthChangeTest is when you get an update event.

procedure SATRedraw;

SATRedraw copies the gSAT.offScreen buffer to gSAT.wind and paints any borders outside the active area black. If you prefer to draw the borders yourself (i.e. if you want something more than black there) you can CopyBits the appropriate area and draw the borders yourself. See the "Some questions…" section above.

procedure SATDrawPICTs (pictID, bwpictID: integer);

SATDrawPICTs draws the PICT with ID pictID or bwpictID in the background, just like SATInit does. The IDs are stored and used by SATDepthChangeTest if needed. Use this if you need to either redraw (to get rid of modifications) or if you want to change background to another PICT.

Menu bar hiding:

procedure SATShowMBar;

procedure SATHideMBar (wind: WindowPtr);

SATHideMBar hides the menu bar, and SATShowMBar shows it again. These calls use low-memory globals that make them potentially dangerous. To get the best future compatibility, either avoid hiding the menu bar or use a "compatibility option" that allows the user to disable menu bar hiding.

SATHideMBar takes a window as parameter. This is the window that you want to cover the menu bar with. If you pass **nil**, gSAT.wind is used.

The usual way to use these functions is to hide the menu bar when a game is started, and show it again when the game ends or is paused. After hiding the menu bar, you should immediately update the menu bar part of the screen. The simplest way to do this is to call SATRedraw immediately after SATHideMBar.

Special functions, advanced calls:

procedure SATSetSpriteRecSize (theSize: longint);

(Advanced initialization.) *SATSetSpriteRecSize* is a special function for prorammers who need a bigger Sprite record than the default. Most programmers should never need this. If you must have more data for each sprite than the default, modify SAT.p appropriately and call SATSetSpriteRecSize(sizeof(Sprite)) **after** SATInit/SATCustomInit (but before any sprites are allocated).

procedure SATInstallSynch (theSynchProc: ProcPtr);

(Advanced initialization.) *SATInstallSynch* installs a procedure theSynchProc, which should 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 SATNewFace (faceBounds: Rect): FacePtr;

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

procedure SATSetPortFace (theFace: FacePtr);
procedure SATSetPortFace2 (theFace: FacePtr);
procedure SATSetPortMask (theFace: FacePtr);

(Advanced face management.) Sets the current port to a face or the mask 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 SATChangedFace.

Warning: You can not trust the port set by these functions to stay valid over extended periods. Calls to SATRun, SATGetFace and SATChangedFace, and later calls to SetPortFace will invalidate it. If you must have two faces in a valid port each at the same time, call SetPortFace for the first and SATSetPortFace2 for the second. (Except for this case, SetPortFace and SetPortFace2 are identical.)

The ports used by these routines are found in gSAT:
SATSetPortFace uses gSAT.iconPort and gSAT.iconPortGD.
SATSetPortFace2 uses gSAT.iconPort2 + gSAT.iconPort2GD.
SATSetPortMask uses gSAT.bwIconPort.

procedure SATChangedFace (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 SATSetPortFace and SATSetPortMask.

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 SATInit or SATCustomInit 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 SATSkip;

SATSkip does the same things as SATRun *except drawing*. Collision detection, sound playing and sprite handling routines are performed. It should typically be used instead of SATRun in order to get reasonably high speed on Macs that are too slow to keep up with the speed you want when calling SATRun 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 SATSkip, even if they run slightly slower than intended on slow Macs. Consider SATSkip if you use so many sprites that the program gets unreasonably slow on the slowest Macs it may be used on (typically MacPlus).

procedure SATKill;

SATKill 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. (You could use GWorlds for that just as well, except that demands 32-bit QD.)

Sound routines:

The sound routines produces sound with priority handling, managing the bugs in Apple's Sound Manager as well as possible. By default, it uses a single sound channel, but you can configure it to use more if more channels are available. 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 SATInit so you hardly have to use it directly.

procedure SATSoundOn;
procedure SATSoundOff;

These routines turns SATSnd on and off. After SATSoundInit 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 a channel is free, in case all channels are busy playing sounds with higher priority, 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 SATRun. If you use the sound routines when no animation is running (that is, when you don't call SATRun 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.

procedure SATSetSoundInitParams (params: Longint);

SATSetSoundInitParams sets the parameters for initializing channels. Use this only if you are not happy with the default setting (mono, no interpolation).

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.]

function SATGetNumChannels: integer;

SATGetNumChannels returns the number of channels that SAT uses.

function SATGetChannel (chanNum: integer): Ptr;

SATGetChannel returns a pointer to the SndChannelPtr for the specified channel. It does not return the SndChannelPtr itself (which might be nil). You may use this call to initialize a channel the way you like (though in such a case, you might just as well use a private channel that SAT doesn't know about).

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 SATTrapAvailable (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.

SATTrapAvailable 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 SATTrapAvailable($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 gSAT.colorFlag to be correct, but does otherwise not depend on SAT being initialized.

Final words

I believe SAT is pretty stable and useful now. There may still lurke some bugs (and probably does), but the bugs that have turned up recently have been non-critical, in rarely used functions. The core feels sturdy. The standard questions remain, though:

– 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 SATInit (pictID, bwpictID, Xsize, Ysize: integer);

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

Sprite and face routines:
function SATNewSprite (kind, hpos, vpos: integer; setup: ProcPtr): SpritePtr;
function SATNewSpriteAfter (afterthis: SpritePtr; kind, hpos, vpos: integer; setup: ProcPtr): SpritePtr;
procedure SATKillSprite (who: SpritePtr);
function SATGetFace (resNum: integer): FacePtr;
procedure SATDisposeFace (theFace: FacePtr);

Running the animation:
procedure SATRun(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); [OBSOLETE]
procedure SATCopyBitsToScreen (src: GrafPtr; srcRect, destRect: Rect; fast: Boolean); [OBSOLETE]
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 SATRedraw;

Menu bar:
procedure SATShowMBar;
procedure SATHideMBar(wind: WindowPtr);

Advanced calls:
procedure SATInstallSynch (theSynchProc: ProcPtr);
procedure SATInstallEmergency (theEmergencyProc: ProcPtr);
procedure SATSetSpriteRecSize(theSize: longint);
procedure SATSetPortMask (theFace: FacePtr);

procedure SATSetPortFace (theFace: FacePtr);
procedure SATSetPortFace2 (theFace: FacePtr);
function SATNewFace (faceBounds: Rect): FacePtr;
procedure SATChangedFace (theFace: FacePtr);
procedure SATSetStrings (ok, yes, no, quit, memerr, noscreen, nopict, nowind: Str255);
procedure SATSkip;
procedure SATKill;
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;
function SATGetNumChannels: integer;
function SATGetChannel (chanNum: integer): Ptr;
procedure SATSetSoundInitParams (params: Longint);

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 SATTrapAvailable (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);