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.

Version 2.5.0

Table of Contents

Page

SECTION 1 - INTRODUCTION 1-1
FORWARD 1-1
LEGAL TERMS 1-2
BACKGROUND 1-3
WHAT YOU NEED 1-3
FEATURES AND LIMITATIONS 1-3
DISCLAIMER 1-4
ACKNOWLEDGEMENTS 1-4
RELATED PACKAGES 1-5

SECTION 2 - PACKAGE CONTENT LIST 2-1
FILES IN THE TOOLKIT 2-1
Think Pascal Library 2-1
Think C Library 2-1
Codewarrior Library 2-1
Resources 2-2
Library Source Code 2-2
EXAMPLE PROGRAMS 2-2
Collision Source Files 2-5
Collision][Source Files 2-5
Collision/// Source Files 2-5
Stepplatform Source Files 2-5
Resource And Project Files 2-6
Offscreen Toys SAT Source Files 2-6
Bricks Source Files 2-6
Fastload Demo Source Files 2-6
StrictGrid Demo 2-6
TileScroll Demo 2-6
RealTimeScaling Demo 2-7
SAT Invaders Source Files 2-7
HeartQuest Source Files 2-7
Add-ons 2-7
Misc. Files 2-8
Compatibility (Under SAT Think Libs) 2-8

SECTION 3 - USAGE 3-1
GENERAL PRINCIPLES 3-1
Using SAT 3-2
Initialization 3-3
How To Write A New Sprite Unit 3-3
Sprite Unit Source 3-4
Sorting 3-7
Collision Detection And Handling 3-8
The Collision Detection Rectangle, The hotRect 3-8
Collision Notification 3-9
Additional Notes On Collision Detection 3-10
Custom Collision Detection 3-10
Faceless Sprites 3-11

SECTION 4 - BITS AND PIECES 4-1
Responding To Update Events 4-1
Miscellaneous Functions 4-1

Table of Contents (continued)

Page

Modifying The Background ... 4-2
Making A Face From Other Sources Than Cicn Resources ... 4-2
Scrolling Backgrounds ... 4-3
Continuous Scrolling ... 4-3
Step-scrolling ... 4-4
Fake Scrolling ... 4-5
Continuous Scrolling With TileScroll.p ... 4-5
Starfields ... 4-5
Mac Friendly Programming ... 4-6
The SATPort ... 4-6
Dying With Dignity ... 4-7
Sound ... 4-7
Frequently Asked Questions ... 4-8
The C Interface ... 4-13
The C++ Interface ... 4-14
Routines (In C Syntax, Since C Users Are Those Who Are Likely To Need Them) ... 4-14
Writing Your Own Blitters ... 4-14

SECTION 5 - THE ADD-ONS ... 5-1

INTRODUCTION ... 5-1
Sprite Behavior ... 5-2
SATToolbox ... 5-2
What's Not There Yet ... 5-3

SECTION 6 - THE PROGRAMMING INTERFACE ... 6-1

THE INTERFACE ... 6-1
SAT Data types ... 6-1
Global Variables ... 6-3
SAT PROCEDURES ... 6-5
Easy Initialization: ... 6-5
Customized Initialization: ... 6-5
Sprite Management ... 6-8
Drawing ... 6-10
Maintainance ... 6-11
Menu Bar Hiding ... 6-12
Sound Routines ... 6-16
Multi Channel Sound ... 6-18
Pattern Utility Routines ... 6-20
Scrolling ... 6-22
Utility Routines ... 6-22
Final Words ... 6-24
QUICK REFERENCE ... 6-24
Initialization ... 6-24
Customized initialization ... 6-24
Sprite And Face Routines ... 6-24
Drawing ... 6-25
SetPort Replacements ... 6-25
Maintainance ... 6-25
Menu Bar ... 6-25
PICT Resource Utilities ... 6-26
Advanced Calls ... 6-26
Sound ... 6-26
Pattern Utilities: ... 6-27

Table of Contents (continued)

Page

Pixel arrays 6-27
Scrolling 6-28
Misc 6-28

List of Figures

No.	Title	Page
Figure 3-1.	The Screen, gSAT.offScreen And gSAT.backScreen, Respectively	3-1
Figure 3-2.	The Default Setting And A Few Of The Alternatives	3-2
Figure 3-3.	Outline Of A Typical Program	3-3
Figure 3-4.	A Snapshot From HeartQuest (Old Version), Where An sPoints Sprite Has Just Been Created When The Butterfly Took A Bonus (sBonus) Sprite	3-4
Figure 3-5.	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	3-7
Figure 3-6.	A Sprite Face And Three Different Suggestions For A hotRect For It	3-8

SECTION 1 - INTRODUCTION

FORWARD

This is the 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 a whole bunch released games:

- Slime Invaders
- Bachman
- HeartQuest
- Ingemar's Skiing Game
- Bert
- Solitaire House (soon)
- Smack a Skunk
- Missions Of The Reliant by Mike Rubin
- Asterax by Michael Hanson
- Invaders by Bettini Simone
- CyberNation by Roy Dictus
- LetterLand
- Bedlam
- NeXus
- Spacewar
- Warbirds
- Slick Willie
- Catch The Buzz
- Tetris Plus
- Smart Move
- FARM Patrol
- Star Chaos
- Zap'T'Balls
- Escape Velocity
- Foobar Versus the DEA
- Boom
- P'tong
- Space Debris
- Centaurian
- Snood

I know I'm forgetting several (please remind me!), and more appear all the time.

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

- Several demos with different complexity ranging from trivial, very easily understood examples (including a tutorial) to a complete arcade game, in both Pascal

and C. Some demos are only in Pascal so far. (Think Pascal still has - still, in 1998! - the best source-level debugging system for the Mac.)

- Direct-to-screen (fast) drawing in b/w, 16, 256 and thousands of colors, with the option to plug in blitters 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 older versions of 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 supporting both b/w and color, old Macs like Plus, SE, Classic included.

All in all, a complete toolkit for game making - and it is free of charge. I demand only credits and a free copy of released products.

This document describes version 2, the color version of SAT, which was released in 1993 as a major revision of the black-and-white SAT from spring 1992. From 2.0 up to 2.5.0, the interface has been further refined, many features have been added, and lots of new material has been added, not least as add-ons.

LEGAL TERMS

This package (the SAT package) consists of this manual, the SAT library itself (Pascal and C versions), the add-on source-code and libraries, and resource files, project files and source code to the tutorial and the example programs *SATminimal*, *Collision*, *Collision][*, *Collision ///*, *MyPlatform*, *Zkrolly*, *Off-ScreenToysSAT*, *StepPlatform*, *Bricks*, *SATCluts*, *RealTimeScaling Demo*, *StrictGrid Demo*, *TileScrollDemo*, *FastLoad Demo*, *SAT Invaders* and *HeartQuest* (all in Pascal versions, most of them in C versions).

This package is free of charge when used for any Macintosh based software product (public domain, freeware, shareware or even commercial) 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 or sold in any way (as public domain, freeware, shareware, or commercial) you should send me a free copy (making me a registered user of the program, if that applies) and mention SAT and my name in the documentation and the About box. (Commercial users should strongly consider buying the full source code.)

My internet address is **ingemar@lysator.liu.se**, and my real address is:

Ingemar Ragnemalm
Plöjaregatan 73
S-58333 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 polished enough to be complete until much later) followed by a *Pacman* game (*Bachman*) and a game for my wife (*HeartQuest*). Then, I thought the routines I'm using had 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 believe I help new game programmers to 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* (any version, but all or most project files are for CW Pro 2, so with an older version you must remake the project files). *CodeWarrior* is mandatory for using the PPC library.

Other PowerMac compilers need to use the shared SAT library for making native code using SAT. I have tried this with *Symantec C++* version 8. However, the shared library is no longer part of the standard distribution since few use it. It was included up to SAT 2.3.8.

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. 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, and 8 too as far as I know, with or without Color QuickDraw, and has been tested on most Mac models.) The programming 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 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, 8 and 16 bits (that is 2, 16, 256 and thousands of 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 64x64. 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 is only recommended if you only aim for fairly fast Macs, like 68LC040 and faster.

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.

And thanks to… sorry, I forgot your name (I will put it in when I remember, promise!)… the guy who explained the scrolling technique I used for Tile-

Scroll. (It is the same person who maintains SpriteWorld these days, for which he wrote a similar engine.)

Many thanks to all the beta testers, especially Mike A. Balfour, Alex Ivrii and Mike Rubin. (Mike Rubin's game, *Missions of the Reliant*, was the first released SAT-based game other than my own - and was a hit and is probably classic today!) Many later testers and bug reporters would deserve mention.

Thanks to Mike Zimmerman (*MyPlatform*) Ken Long (*Collision*, *SATInvaders*, *Zkrolly*, *HeartQuest*), Richard Bannister (*Bricks*, tutorial, plus testing 2.5.0 and more), Charles Brunet (*Offscreen Toys*) and Peter Amberg (*Collision ///*, plus testing 2.5.0 and more) for the help in translating the demos to C, to Miguel Frias for grammatical corrections/proofreading, 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++. A preliminary version of Nathaniel's C++ interface is available separately. Thanks the Christopher Ross (Baker Miller & Ross) for the manual layout & design).

And more special thanks to Bo Lindbergh, who wrote the PPC mask-blitter!

Finally, 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 contributions *Offscreen Toys* and *MicroAnimation*). 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.

Finally, there is also *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 (?).

SECTION 2 - PACKAGE CONTENT LIST

FILES IN THE TOOLKIT

The following files are in the SAT toolkit itself.

Think Pascal Library

SAT.p Interface file for the library.

SAT.lib The *Sprite Animation Toolkit* compiled to a library. This is actually an MPW .o file, but I have never tried using it with MPW, only with *Think Pascal*. You probably have to rename it to SAT.o to use it from MPW.

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

Think C Library

SAT.h Include file for using the C library.

SAT.p The *Sprite Animation Toolkit* compiled to a *Think C* library. The extra c in SATC.p 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.)

> **NOTE:** SAT.p is a library, not a project. You should not open it from Project Manager, and must not remove objects on it. If you do that, you render it unusable, and get link errors!

ThinkCstuff.p This is a library that *Think C* users must add to their projects.

ThinkCstuffA.p Same as above, but without BitMapToRegionGlue. Use this if you get a link error telling you that BitMapToRegionGlue is multiply defined.

SAT(PPC).shlb Shared library, primarily intended for the PPC side of *Symantec C++*.

> **NOTE:** You must use 2-byte struct alignment when you use this lib. (No longer included but can be provided on demand.)
>
> Since Think C is really, really stupid about finding files that have been moved (it just doesn't - you have to remove the files and add them again), you should at the very least put SAT.p and SAT.h in a folder in the development system.

Codewarrior Library

SAT.p, SAT.h Interface files.

SAT(68k).lib, SAT(PPC).lib SAT as *CodeWarrior* libraries, useable from both Pascal and C.

> **NOTE:** You must use 68k struct alignment when you use the PPC lib.

Resources

SAT blitters.rsrc The direct-to-screen blitter resources. If you don't include them, your program will only use QuickDraw. Some demos include them in their resource files already - but not all of them.

Library Source Code

Library source code is available separately, directly from me (only!). You only need it if you want to make changes, port it to some other platform etc. You get the sources for personal use only for $20 (or equivalent).

For commercial use, the charge is $100 (site license). 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 - which I recommend - you don't have to pay me anything.

Updates to SAT will occasionally be distributed on Internet. I can snail-mail updates for $10. I prefer if you can handle that over the net, though.

EXAMPLE PROGRAMS

Fifteen example programs of varying complexity are included in SAT:

- *SATminimal*
- *Collision*"
- *Collision][*
- *Collision ///*
- *Zkrolly*
- *StepPlatform*
- *Offscreen Toys SAT*
- *Bricks*"
- *FastLoad Demo*
- *SATCluts*
- *StrictGrid Demo*
- *TileScroll Demo*
- *RealTimeScaling Demo*
- *SAT Invaders*
- *HeartQuest*

There is also a separate tutorial with a special tutorial demo.

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.

Zkrolly & StepPlatform Demonstrate scrolling games, the former continuous scrolling (which s slow) and the latter step-scrolling, which is usually preferrable. StepPlatform also demonstrated a way to make platform games. It is a merge between the old demos MyPlatform and StepZkrolly. It was made by Nissan Zamfir, and resulted in a demo with much more game feeling, so I decided to let it replace its predecessors.

Offscreen Toys SAT Is the SAT version of my small sprite demo *Offscreen Toys*. Do not confuse it with the stand-alone demo. Event and menu handling (without TransSkel).

Bricks Is a demo that demonstrates the use of resting sprites, using SATRun2 (preliminary name). It uses a very large number of sprites that are still most of the time. Event and menu handling (without TransSkel).

FastLoad Demo Demonstrates the use of the new (as of SAT 2.3.8) add-on FastLoad. It loads 100 faces from a pair of PICT resources in a time much lower than it would take to do it from *cicn* resources. The demo itself is pretty boring: a single sprite controlled by the mouse that picks one of the 100 faces depending on position. Check this out if you use many faces and find the loading time high!

SATCluts Demonstrates how to use different color tables with SAT, switching instantly for effects like palette animation.

StrictGrid Demo Demonstrates how to use SATStrictGridToolbox after the revisions for 2.5.0.

TileScroll Demo Demonstrated the new scrolling engine.

RealTimeScaling Demo Demonstrates the real-time scaling add-on (part of SAT from version 2.5.0).

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, rudimentary Apple Event and Multifinder event handling 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).

If you want even more demos, check out my ftp archive (*ftp.lysator.liu.se/pub/mac/sat*). Space limitations may force me to remove some demos, but additional demos may include the following:

SATminimal Turbo A modified SATminimal whith a dialog that lets you choose between many different setups. Here you can see how to get that particular setup you want.

Kopter — A pretty good game draft by B.J. Köbben. You fly a helocopter, can shoot at things, and the screen scrolls continuously. If you build on this demo, please dont forget the credits to Mr Köbben!

myPlatform Demo — My old platform demo that was superseded by Nissan Zamfirs StepPlatform.

Tint Demo — Shows a way to colorize sprites on the fly.

BloatBlit — Shows how to use your own blitters, and compares a few blitter types (none of them of any real practical use since the default ones are better).

Text Demo — Shows how to make sprite faces directly in QuickDraw. In this demo, all faces are generated by DrawString each timethey are drawn.

Bricks + FastLoad — The Bricks demo, but with all the faces loaded with the FastLoad add-on.

SATPlayMovie — Plays a QuickTime movie in a sprite face.

SmoothMove — Moves a sprite using the routines in the SATToolbox add-on, resulting in nice and smooth movement in 64 directions.

SATminimal / blend Shadow — The SATminimal demo with a shadow aded on the faces. The shadow is not painted over the background, but blended into it, just darkening it.

WrapZkrolly — The Zkrolly demo hacked to wrap-around, horizontally.

There are even a few more interesting demos on my disk that I may release in the future. No specific promises - they need to mature a bit.

You can find even more demos out on the net. Peter Amberg has made a number of SAT demos that you can find on his SAT page (*http://www.vis.inf.ethz.ch/students/pamberg/sat.htm*). Another SAT demo, related to HeartQuest, is Rasta Chomp by Brian Jones (*www.gazsoftware.com*).

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.

SATminimal source files

SATminimal.proj, SATminimal.p.rsrc (SATminimal.p) — 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.

The folder SC++PPC contains a copy of the demo with project file for *Symantec C++ 8* for PowerPC. This demo worked for me when I could try it, but please note that I can't verify this for every update of SAT, since I have to use SC++ on a friend's Mac, since I don't own it myself. So, no guarantees - I provide the shared lib, that's all I can do.

Collision Source Files

Collision.proj, Collision.p.rsrc (Collision.p)	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.

Collision][Source Files

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

Collision/// Source Files

Collision///.p 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.)

Stepplatform Source Files

StepPlatform_Demo.p.rsrc StepPlatform.p (StepPlatform_Demo.p)	Resource and project files.
StepPlatform.p (StepPlatform.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!)
PlatformGlobals.p, InformUser.p (InformUser.c, VBLSync.c)	Miscellaneous sources. Sorry, I havn't made the VBLSync in Pascal yet.

Zkrolly source files	Zkrolly.p.rsrc, Zkrolly.proj (Zkrolly.p)

Resource And Project Files

Zkrolly.p (Zkrolly.c)	Main program.
sZprite.p (sZprite.c)	Z-shaped sprite, the sprite that the view follows.
sXprite.p (sXprite.c)	X-shaped sprite, moves around independently.

Offscreen Toys SAT Source Files

OffscreenToys SAT.rsrc **OffscreenToys SAT.p**	Resource and project files.
OffscreenToysSAT.p	Complete source code in one file.

Bricks Source Files

Bricks.rsrc **Bricks.p**	Resource and project files.
Bricks.p	Source file with main program, event and menu handling, code for the brick sprite. (Also in C.)

Fastload Demo Source Files

Fast load.rsrc **Fast Load.p**	Resource and project files.
FastLoadDemo.p	Source file with main program. (Also in C.)

StrictGrid Demo

StrictGrid demo.p **StrictGrid demo.rsrc** **(StrictGrid demo.c)**	Source and resource files.
StrictGrid demo.p **StrictGrid Demo.μ** **StrictGrid Demo.cp**	Project files.

TileScroll Demo

sPlayer.p **TileScrollDemo.p** **TileScrollDemo.rsrc**	Source and resource files.
TileScrollDemo.p **TileScrollDemo(PPC).μ**	Project files (*Think Pascal* and *CodeWarrior*).

RealTimeScaling Demo

Source files like SAT minimal, on which it is based. Project files for *Think Pascal* and *Think C*.

SAT Invaders Source Files

SAT Invaders.p
SAT Invaders.p.rsrc
Project file and resource file.

gameGlobals.p
Global variables and resource numbers.

soundConst.p
Sound resource numbers and their handles.

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

main.p
Main program. Game window handling, game driver, initializations... (Also in C.)

HeartQuest Source Files

HeartQuest.p
HeartQuest.p.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. (There might be a C port some day. Anyone?)

Add-ons

The add-ons are a number of units, useful when making games with SAT. None of them are mandatory, but they can simplify many tasks for you.

They are divided in four groups:

Load faces
which loads faces from other sources than *cicn* resources.

Sprite behavior
with units that simplifies sprite programming.

Storage with units that handles scores and settings.

Graphic effects for add-ons that perform certain graphic effects of interest, like starfields and screen fades. Libraries with most of the units are included, primarily intended for C users.

These files are documented in some more detail in Section 5 - The Add-ons.

Misc. Files

The Misc folder holds files that may be important for you, but that don't directly fit in any other catagory.

ICN# -> cicn 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 originally 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 Multi-Finder 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, or as part of *TransSkel Pascal 2.5*, which I have uploaded to the major ftp sites (*Sumex, UMich*).

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 2.0, with docs and demos, to complement the TransSkel.p file included here.

Compatibility (Under SAT Think Libs)

InterfacesUI.p *Think Pascal* is not delivered with Universal Interfaces. All SAT demos use the new names in the Univeral Interfaces. InterfacesUI.p define the most important new names for use with *Think Pascal*. Another option is to get the recently (autumn 95) released Universal Interfaces for *Think Pascal*!

For Think P v3 Users of the old *Think Pascal* version 3 need the BitMapToRegionGlue routine and the Gestalt trap. You can find those here.

Tutorial For getting a really easy start, there is a tutorial folder, with some really fundamental exercises, with solutions. I consider it preliminary, but I hope it can help some of you. See further the files in that folder.

SECTION 3 - USAGE

GENERAL PRINCIPLES

SAT manages a list of sprites, describing position, current appearance (an icon or PICT 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.port) to perform drawing. See the *Modifying The Background* section. The other bitmap/pixmap, offScreen, is a copy of the screen.

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, passing it as chosen device to SATCustomInit.) 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 through the parameters to SATCustomInit.

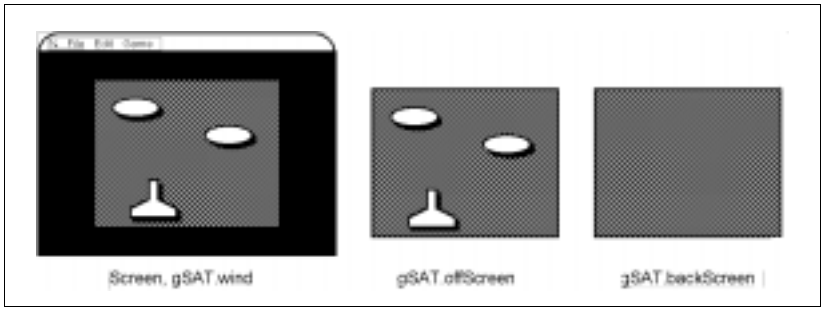

Figure 3-1. The Screen, gSAT.offScreen And gSAT.backScreen, Respectively

SAT can also produce asynchronous sound. This is by default in one channel only, but can be configured to use more (through SATSoundInitChannels). 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 (a rare event these days). 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.

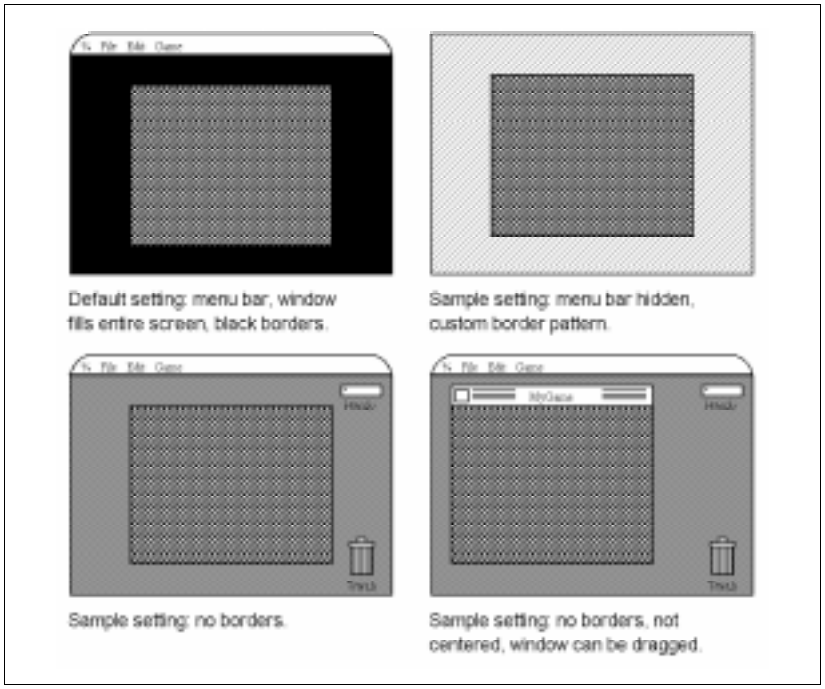

Figure 3-2. The Default Setting And A Few Of The Alternatives

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: sPlayer, sEnemy, sShot and sMissile. 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.

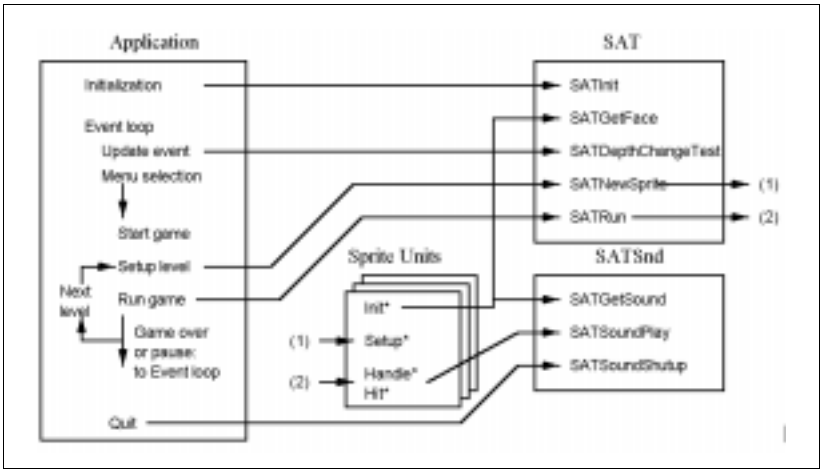

Figure 3-3. Outline Of A Typical Program

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. You will usually want 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 3-4. A Snapshot From HeartQuest (Old Version), Where An sPoints Sprite Has Just Been Created When The Butterfly Took A Bonus (sBonus) Sprite

Sprite Unit Source

```
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 set the task field in the Setup* routine, if you have one. If you don't, the sprite will self-destruct. (Concerning sprites that don't need a Handle* procedure, see below.)
- task is a pointer to the Handle* 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 gSAT.offSizeH and gSAT.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.
- The Handle* or Hit* procedures of a sprite should avoid re-organizing the sprite list, change the order, unlink sprites or dispose sprites with SATKillSprite! Changes in the sprite list should either be done from your main loop, or done by SAT according to your configuration options. Sprites should usually be disposed by SAT, indicated by seting the task field to nil. Adding sprites from Handle* or Hit* is OK, though.
- 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 game uses almost 200 icons, and Michael Hanson's *Asterax* space game uses almost 400 - as many as 36 for a single object! If you have that many, check out the FastLoad add-on! It will save loading time as well as disk space!

- If you want to make sprites that have no callback procedures at all, sprites that will be immobile and inactive, or controlled by other sprites or the main program, pass nil for the setup procedure when calling SATNewSprite. That will create a sprite that has the task procedure set to an empty procedure.

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 higher than another (lower position.v value) will be drawn before the other, so if they overlap, the higher one will appear to be farther away from the viewer, giving a pseudo-3D effect. 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 SATConfigure).

Figure 3-5. 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.

If the BubbleSort is not sufficient for your problem, you can sort the sprite list yourself. (As an example of a program that would benefit from a better sorting algorithm, look at *Bricks* during the first seconds it runs.) If you do make your own sorting, you should not do it from inside a sprite handler, since those routines are called by going through the list, and SAT expects them to stay in order while that happens. However, if you sort from your main program, from where you call SATRun, there is no problem.

The add-on SortingUtils may be of help if you need a better sorting algorithm. The Bucket Sort in that add-on is extremely fast.

You may ask why sort at all? Perhaps your sprites never overlap? If so, why not sort? If the sprite list is sorted, the collision detection will be faster, since each

sprite will only be tested up to a certain distance in each direction (a distance you set with SATConfigure)..

> **NOTE:** If you use SATRun2, you must set the dirty flag on overlapping sprites that change order, or they may be incorrectly drawn. The internal sorting handles this automatically.

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!

The Collision Detection Rectangle, The hotRect

Collision detection is generally done with rectangles, for speed. SAT does collision detection by checking if rectangles attached to each sprite overlap. These rectangles are the hotRect fields of the sprites. SAT makes copies of the hotRects and displaces them by the position.

When a sprite is created, the hotRect is set to (0,0,0,0), an empty rectangle. If you don't change it, it will never collide with anything.

So, what should you set it to? You can check for collisions using the bounding boxes of the faces of each sprite. If that's what you want, try the following:

```
theSprite^.hotRect := theSprite^.face^.iconMask.bounds;
```

However, this is not necessarily perfect for all situations.

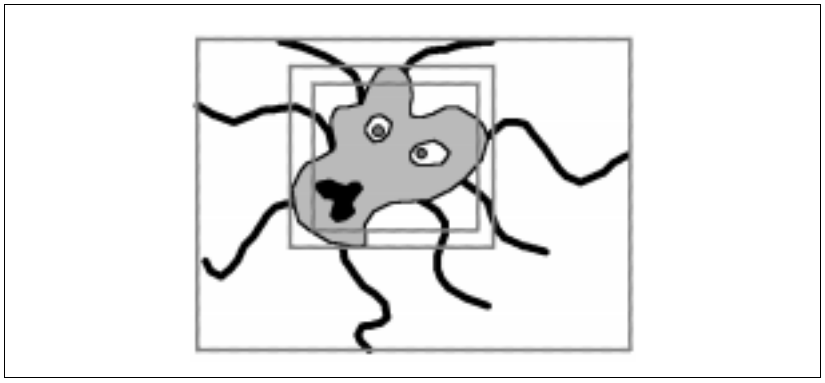

Figure 3-6. A Sprite Face And Three Different Suggestions For A hotRect For It

The sprite face in figure 6 shows something related to an octopus. It has a body that is much smaller than the face, since the face must also show the legs. If something shoots at it, it shouldn't be killed by a hit in the outermost rectangle, right? Even the middle one may be a bit too sensitive.

Do you find it insufficient to detect collisions with rectangles? It is possible, even fairly easy, to check if the masks of the face of two sprites overlap. That can be done once you know that their hotRects overlap. The demo *Collision ///* shows how to do it. See also below.

Collision Notification

Once a collision is detected, 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:

All-to-all collision detection

If kNoSort is chosen, the entire sprite list is searched for every sprite (for ForwardCollision and BackwardCollision, from the sprite to the end of the list). This can be fairly time-consuming if you use a lot of sprites, but is no problem if you only use a few.

Collision detection limited to close sprites

If kVPositionSort or kLayerSort is chosen, the search for hits is only performed for sprites within a certain distance (set by ConfigureSAT) from the position.v or layer of the sprite.

In simple cases, kVPositionSort and kKindCollision, the defaults, will be what you need - fast and easy to handle.

Additional Notes On Collision Detection

There are some variants you can consider when doing collision detection. These notes are primarily for advanced programmers.

Non-rectangular sprites

Even if your sprites aren't rectangular, you should use the hotRects to get possible collisions. If you use hitTasks, you can do additional checking once you know that the hotRects overlap. It is much faster to check rectangles than general shapes!

For Example

Consider that you want to check if two balls, circular shapes, collide. You set their hotRects to rectangles that the shapes fit in. When a collision is detected, you check if the distance between the two sprites is small enough for it to be a collision. (I.e. you check the squared distance (me^.position.h* me^.position.h+me^.position.v* me^.position.v) - (him^.position.h*him^.position.h+him^.position.v*him^.position.v) against the squared diameter of a ball. Got that?)

The demo Collision/// demonstrates one more advanced collision detection scheme, where the mask regions of the faces are used in order to determine if the masks overlap. This allows arbitrary shapes!

Custom Collision Detection

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.

SECTION 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 SATDepthChangeTest, 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 SATDrawInt and SATDrawLong (which simply does NumToString followed by DrawString), or SATRand (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, SATSetMouse is provided. Note that SATSetMouse 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. Note that the gSAT.wind.port window must cover the menu bar or SATHideMBar won't produce the expected result.

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.

SATReportStr and SATQuestionStr are two functions that display a message in an alert box. SATQuestionStr 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.port, 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 the destination SATPortPtr, 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 SATPorts.

SATPlotFace has a cousin, SATPlotFaceToScreen, that you should use for plotting faces 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 64x64 (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.

> **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 (prelininary) new demo if SATMinimalX, which demonstrated how you can load a whole set of faces from a single 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 can do this in one out of two ways: continuous scrolling or step-scrolling.

In both cases, you should initialize SAT with SATCustomInit, with the flag beSmart set to false. That will allow you to create an offscreen that is larger than the screen.

Continuous Scrolling

Continuous scrolling is what you get if you redraw the entire animation window for every frame. 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.

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.

CopyBit is what you should use to copy the offscreen to the screen. To get maximum speed, you should make sure that you are using CopyBits correctly. Copy at least byte-aligned when running in 4 bits/pixel or less, preferrably long-word-aligned. (That means that your offset should alway be divisible by 4, 8 if you want decent speed even in B/W.) Always keep the size of the source and detination equal. The foreground color should be black and background white.

Step-scrolling

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 a sprite (or whatever) gets too close to the edge. This can be done with SAT by changing the origin of the animation window. Step-scrolling is easiest done by calling SATStepScroll, with the player sprite's position as the viewPoint parameter.

When using SATStepScroll, you need to know a little about its assumptions and limitations. SATStepScroll inspects certain variables to find out how to operate, and is not extremely general. This is what it demands:

- The animation window is only used for the scrolling area. Keep any graphics that should not be scrolled (i.e. scores) in another window. SATStepScroll should be allowed to use the window all the way to the edges.
- The window should not be bigger than the screen. SATStepScroll checks the window size, not the screen size! Note that SATCustomInit, with beSmart set to false (necessary to create big off-screens) will create a window just as big. Thus, you should either create the window yourself and pass it to SATCustomInit, or resize the window after init.

Step-scrolling is demonstrated in the demo StepPlatform (and in the older, obsolete demo *StepZkrolly*, which resembles *Zkrolly*).

Fake Scrolling

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. A SAT-based game using fake scrolling is *Ingemar's Skiing Game*. 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.

There is a demo that does fake scrolling, but it is not included in the standard distribution. It is called *RoadTest*, and animates a road and a few cars. Look for it on my ftp archive.

A special case of fake scrolling is to use a smooth background with starfields animating over it. See the next section.

Continuous Scrolling With TileScroll.p

The most recent addition (SAT 2.5.0) to the scrolling methods is the one implemented by the TileScroll add-on. This is a more complicated way to get scrolling, but it gives you a huge world for a low memory cost. Only the visible part of the world is actually drawn off-screen, and parts that are scrolled in are drawn on the fly.

The present TileScroll code assumes, as the name implies, that the world is built from tiles, in a grid. Also, the built-in to-screen blitter only works in 8-bit color and only in 32-bit addressing mode. These limitations may be changed in the future.

This is an extremely useful add-on that I wholeheardtedly recommend. It is somewhat slower than the other methods, but on a 68040 or better, it performs well. For games that must run well on old Macs, step-scrolling is still the best option.

See the demo *TileScroll Demo*.

Starfields

A frequently demanded feature in SAT is the ability to make starfields, many single-pixel stars moving over the screen to give an illusion of motion. This, and other operations where you want to plot many single pixels, is inefficient to do with normal sprites. Instead, there are routines and types designed to handle arrays of pixels, plotting them all in one operation. The routines are SATDrawPixels and SATCopyPixels, and their safer relatives SATDrawPixelsSafe and SATCopyPixelsSafe. If you create a pixel array (which should be done by allocating it with NewPtr), you can plot the pixels to screen or an offscreen port with SATDrawPixels, and copy the pixels from one port to another with SATCopyPixels.

Note that the non-safe routines make no border checks whatsoever! It is up to you to make sure that every pixels stays within the bounds of the port.

So, where are you supposed to draw and erase? Well, that, I basically leave up to you, but try drawing directy to the screen (gSAT.wind), and erasing by copying from gSAT.offScreen.

> **NOTE:** I first intended to make the starfields part of the SAT library, but due to limitations in Think C (which can't handle libraries over a certain size) I had to move it out to the Add-ons. That is where you will find it now.

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 SATPort

The structure SATPort is somewhat related to a GWorld. It holds a port (GrafPtr/CGrafPtr), a GDHandle (which is usually of no interest for you) and a table that is used internally. Most SAT routines work only on SATPorts, since they require all these three components.

When you want to save the current port and device in order to restore it later (often a good idea!), using SATGetPort and SATSetPort will result in briefer code as well as backwards compatibility.

Dying With Dignity

When SAT encounters a fatal error, typically 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 is 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.

Frequently Asked Questions

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/CodeWarrior*: 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? PowerMac users: Is your project set to 68k struct alignment?

Have you updated your program from an older version of SAT, and it doesn't work any more? If the older version had blitter resources, did you replace the old ones with the latest ones?

Are you re-organizing the sprite list from a sprite handler call-back? Perhaps starting a new level by calling it from a handling or hit routine? You just may be pulling away the sprite list from under SAT's feet, disposing or unlinking the sprite SAT is calling!

Of course, the reason can be a bug of my part. Are you doing something radically different from the demo programs? Try to pin-point the problem as far as possible and then report it to me.

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.port^.portbits, gSAT.wind.port^.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.

The animation works, but I get no collisions!

Have you set an appropriate hotRect for each sprite? A sprite with (0,0,0,0) won't ever collide with anything.

What kind of collision handling do you use? Collision handling type, sorting and search width (all set by SATConfigure) all affect collision handling. Compare what you do to what I do in the demos.

The animation works, but the clip region has no effect!

Are you using FastLoad? If you do, you need to let it create mask regions for the faces used by clipped sprites.

Are you using custom drawProcs? If that one doesn't support clip regions, you get no clipping. The clipping support is in SATSafeMaskBlit.

Can SAT be used in a draggable window?

Yes. See *OffScreenToysSAT* and *Collision ///*. It works just fine when using safe animation (i.e. SATRun(false)). If you run fast animation, you should swicth to "safe" when the animation window is not frontmost. Also, consider using ShieldCursor for avoiding mouse droppings.

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 and no. It is quite doable to create a set of faces from one face, in different scales, 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. However, SAT does not rescale on the fly with the default blitters. (It is quite possible though. My game Smack a Skunk uses real-time scaling with SAT, but the scaling routines are not yet in the distribution.)

I have an older version of Think Pascal/C. Can I use SAT with it?

With *Think Pascal,* version 3 works, but only if you use the files in the folder Misc: For *Think P v3*. For *Think C*, version 5 is supported but not older than that. The C library is incompatible with versions prior to v 5, but versions 5 to 7 work. *CodeWarrior*: I only support the latest version, *CW11* at the time of writing this, but the libs should work a few versions back.

Can't you make a MPW version?

There is one: The *Think Pascal* library SAT.lib is an MPW .o file. Rename it to SAT.o. I don't have MPW, so I make no promises, but it has the right format.

Can I use SAT from FutureBasic (or any other development system not mentioned here)?

Only if your development system can import MPW .o files (or any of the other library formats included).

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, using the add-on FaceFromICN.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 *StepPlatform* (the latter replacing the older *StepZkrolly*), and, separately available, *Kopter*. I may release an even better scrolling engine fairly soon, let's say during 1997.

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.

NEWS in 2.3.5: Step-scrolling is now built-in, by the routine SATStepScroll.

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.

If you use SATRun2, you can't use the provided 1-bit and 4-bit blitters, or you will get incorrect results. Also, the older 8-bit PPC mask blitter had some problems with SATRun2, which should be fixed with the new one (SAT 2.4.0).

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.

Just re-initialize! Note, however, that faces and sounds are not disposed by that call, but must be disposed of separately. Sprites are disposed, though.

NOTE: In older versions of SAT, the call SATKill was used, and you were recommended to dispose the window yourself. Now, SAT will assume that if it created the window gSAT.wind.port itself, it is also free to do whatever it likes with it, move, dispose and re-initialize, anything, while if it was supplied by the host program, it is left for you to dispose.

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").

If you run out of memory after a while rather than during startup, perhaps you have a memory leak? Do you allocate memory that isn't disposed?

I'm trying to play a sound with SATSoundPlay, but nothing happens or the sound isn't played until much later.

This can happen 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 haven'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.)

When I call SATInit or SATCustomInit, it gives me an animation area that is slightly smaller than what I ask for.

Pass false to the beSmart field. By passing true to beSmart, you tell SAT that it is OK to cut down the width a little to some value SAT thinks is better, to make sure all blitters get an easy job. However, if you pass false to beSmart, don't complain if you get problems with blitters drawing slightly outside the animation area.

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. Check out the Tutorial, or 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;
```

Using these calls, and using a few fields in the sprite reconds, most importantly position, face and hotRect, and possibly face^.iconMask.bounds to get the size of faces, you can write entire SAT-based games.

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;
```

For some routines that does some standard tasks, check out the SATToolbox file in the Add-ons folder. It gives you routines for standard movement and colliion handling.

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 a few such demos just for you:

MicroAnimation is an extremely stripped-down demo, doing sprite animation in about a single page of code plus comments. Pascal, *Think* or *CodeWarrior*.

Offscreen Toys demonstrates sprite animation on an intermediary level, in about 25k of source code, and is well commented. *OffScreen Toys SAT* is a SAT-using look-alike. Please do not confuse *Offscreen Toys* with *Offscreen Toys SAT*! Pascal, *Think* or *CodeWarrior*.

Both *MicroAnimation* and *Offscreen Toys* can be downloaded from *<ftp://ftp.lysator.liu.se/pub/mac/source/>*.

SpriteEngine is a simpler sprite engine, which comes with full source code in both C and Pascal. You can find it on the CD that comes with *Tricks Of The Mac Game Programming Gurus*, a game programming book I wrote a substantial part of.

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. For commercial use, the fee is $100, and the sources may be used freely within the purchasing company - essentially a site license.

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. The blitters are code resources. The interface for them is not quite fixed yet, which is why I don't document it in this manual, but if you want to write a new one (i.e. for 16-bit color or perhaps compiled sprites) just ask me.

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

It's all *Symantec's* 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* or *Metrowerks Pascal*. There is, however, a C interface, in the form of two libraries (SAT.p and ThinkCstuff.p) and a header file (SAT.h). Most demos are included in C.

You can use the SAT C libraries just like any C library. Put a folder 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:

- 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 ThinkCstuff.p 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, *MetroWerks* seems to have designed their compiler with more communication between the Pascal and

C teams, so we can use the same library for both. You can even mix C and Pascal in the same project!

The C++ Interface

Can we use SAT from C++? Well, at the time of writing, there are no demos in C++ and no complete C++ interface included, 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.

Version 0.1 of *Nathaniel Woods*' C++ interface is available from my ftp site (connect by WWW or ftp to *ftp://ftp.lysator.liu.se/pub/mac/sat/*), but it is not actively supported.

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).

- You can write special-purpose drawing procedures and install them in appropriate faces as their drawProcs.

Note that a blitter does not have to be a resource. The drawProc field in ythe faces allow you to install a blitter that is in the main program code. This is particularly useful for debugging new blitters effectively and for special-purpose drawing procedures.

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…). The PowerMac version uses blitters with resource types *PRBl* and *PMBl*.

Below follows the programming interface for SAT blitters. You should consider this preliminary. The C version use *main* for the function names, since that's what my C compiler demands as entry point for code resources. Whatever the function is named, it should be the main entry point to the code resource, if it is a code resource. You can also use a function in your program, and put a pointer to it in the appropriate place in gSAT.

The rect blitters are called as follows:

In Pascal

```
procedure RectBlit (var srcBits, destbits: SATPort; var r: rect);
```

In C

```
pascal void main(SATPort *srcPort, SATPort *dstPort, Rect *r)
```

The mask blitters are called as follows:

In Pascal

```
procedure MaskBlit (face: FacePtr; theSprite: SpritePtr; var destBits: SATPort;

    srcPt, dstPt: Point; width, height: Integer);
```

In C

```
pascal void main (FacePtr face, SpritePtr theSprite, SATPort *dstPort, Point
srcPt,Point dstPt,short width,short height)
```

At my ftp site, you may find a demo called BloatBlit. That demo includes several custom blitters, none of them worth using over the standard ones, but they do show how it is done. The demos *Tint Demo* and *Text Demo* show two ways to use drawProcs for special effects.

The two QuickDraw-based blitters are built into the SAT library. They are used as default when no appropriate blitter resource is available. They are declared as follows:

```
procedure SATSafeRectBlit (var srcBits, destbits: SATPort; var r: rect);

procedure SATSafeMaskBlit (face: FacePtr; theSprite: SpritePtr; var destBits: SAT-
Port;
```

You never need to call these directly, but you may want to force certain faces to use them, by installing them in the drawProc field.

> **NOTE:** SAT uses the row lists in the SATPorts in its own blitters. Also note that the parameter list has changed since SAT 2.3.9: the clip region handle is replaced by the SpritePtr, which makes it more general and allows more advanced faces.

SECTION 5 - THE ADD-ONS

INTRODUCTION

Surprise, surprise: I can't put absolutely everything you'll ever need in your games in the SAT library - not even everything you'll ever need in 2D sprite-based games! The library is already so big that Think C chokes if I add more. (For some reason, Think C is the least space-tolerant one of the development systems I support. The others can cope with bigger libs.)

There are some features that I find a bit beside the point, or that are not reusable enough to go into the lib. There are lots of subjective descisions there, but I have to put limit somewhere.

Some of the useful features that don't necesarily belong in an animation library are file handling like preference files. In earlier versions, I put that in as part of the demos. HeartQuest included both screen fades (through CLUT fading), preference file, scores etc. Some of the files were reusable (fading and Preferences) and some were very specific to the game and had to be totally reworked to fit another (like Scores).

However, there still are plenty of things that could be standardized, that we would benefit from having some default implementation of, that we can work from. Wht you find in the demos can be used, but as long as I've made no attempts to make it reuable, it can be hard. The Add-on folder was aded to fix this. In that folder, you will find lot of reusable code. To be precise, this is 300k of game-related reusable source-code!

The following information is (at the time of writing this) also on the doc file in the Add-ons folder.

Load faces

Even though SATGetFace is the standard way to load a face, it is definitely not the only one. In the "Load faces" folder you will find three units for creating them in other ways. Use them as they are or study them to create your own solution.

FaceFromPict

This is an old file that has been part of SAT for a very long time. It creates a face from *PICT* resources. It comes in both Pascal and C versions.

FaceFromText

This routine will create a face from a string, in the appropriate size, style and color. It comes in both Pascal and C versions.

FaceFromIcn

This routine loads a face from a *ICN#* resource. What did you say? Unnecessary? Obsolete? Oh no it isn't. I wrote it because I had a real need for it. However, most SAT users certainly don't need it.

FastLoad This unit holds routines for loading many faces from one single *PICT*. This is very useful for speeding up the loading process when using large amounts of faces.

> **NOTE:** The faces in your picts must have a spacing that is divisible by 8, and prefers a spacing divisible by 16 or even 32.

FaceFromICN This unit loads a face from an *#ICN* resource. Yes, that means 32x32 pixels and B/W only.

Sprite Behavior

SAT as it is gives you total freedom, and thereby little help, with the way your sprites move around. In the sprite behavior folder, you will find lots of routines that will simplify your sprites a great deal, granted that they apply to your problems. For example, you will find basic routines to do the standard border check, routines to make sprites bounce off each other, and more advanced ones, helping you with grid-bound sprite movement etc.

SATToolbox

This is a large unit with very useful routines that I've long wanted in the library itself, but it will be in this add-on to begin with. The routines handle sprite movement, border checks, collisions, plus look-up table based trig functions and square root.

SATGridToolbox This unit handles sprite movement in a grid (somewhat like Oxyd). SATToolbox is required.

SATStrictGridToolbox This unit handles sprite movement in a strict grid. A struct grid means that the sprites can only move from grid position to grid position. It is useful for making games like *PacMan*.

SortingUtils The routine BucketSort is the interesting part. It sorts the entire sprite list completely in very short time. Beware of changing the order of overlapping sprites when using SATRun2 though!

TileScroll A scrolling engine of the Power Pete kind, fairly fast and memory conservative. Currently, it is best used in 8-bit color, where it provides a special screen blitter to reduce the tearing effect that is quite visible in other depths.

Warning: I have still not used this engine myself in any complete game, so I don't consider it fully tested yet. Bugs or limitations may surface under heavy use. A variant of it was used in *Harry The Handsome*, so you definitely can use it, though you may want to madify it to suit your needs.

Storage These units deal with file management of various kinds. If you use either scores or settings, you should also use preferences to have a preferences file to pass to the others.

Preferences This is almost the same file that used to be in the *HeartQuest* project. It is slightly improved, has no longer any globals. It opens and creates preference files (in the preferences folder).

Scores This unit handles score and high score display. With 2.5.0, the stubs file ScoresStubs is no longer needed! The high score list can be drawn in any window you like.

Settings This unit simplifies settings. You should use preferences for putting the settings in a preferences file. Settings initializes and saves them, and includes a routine for asking for key configuration.

> **NOTE:** You really don't have to create neither pref file or settings structures until the user changes a setting or gets a high score. However, it is quite a bit more trouble, so I usually create them right away.

Graphic effects Graphics-related features that are not related to sprites.

GammaFade Routines for fading the screen with the gamma table. This works on most color/bw Macs. This gamma fading unit will allow the user to abort the fading effect with a mouse click. (Long fades will often get tedious otherwise.)

ProgressBar An adaptive progress bar, very easy to use. You don't have to worry about how far it should draw; just call it often enough and it will work and look pretty ok. Check this out - making a progress bar has never been easier!

Pixels: Routines for drawing and copying sets of single pixels. This is much faster than using lots of small sprites for making things like starfields or pixel-baed explosions. See *SAT Invaders* for a demo.

AlphaSAT Routines for manipulating the alpha channel. This is of interest when making animations to mix with live video. This is only meaningful in 32-bit color and with hardware that supports the alpha channel. Since this is a rather special-purpose unit, it is not included in the add-on library, but only included as a source-code unit.

MySlotVBL A unit for synchronizing the graphics with the screen update. The unit is not SAT-dependant, but can be used in any program. It uses SlotVBL if available (i.e. all modern Macs) but falls back on old-style VBL if it isn't.

SATSetDepth A unit for setting the screen depth. Updating faces that have already been loaded, if any, (for cicn-based faces, by calling SATDepthChangeTest) is your responsability. C users can use the file depth-switcher.c, by Richard Bannister.

MiscGraphics This unit holds a bunch of useful graphics-related routines. It has better glue code for patterns than the old ones in SAT.lib, and similar routines for cursors. There are transitions to use for scenery switches, and some simple utilities just for simplifying your code a bit.

SATScaling This unit lets you rescale sprites in real-time, fast enough for gaming purposes. It is limited to 8-bit (256) colors only. This unit was used in the game *Smack A Skunk*.

What's Not There Yet

There are some modules that some SAT programmers have asked for that you will not find here yet. In some cases, I have not added those modules despite

having them, since I am not the author, have added little, and feel I shouldn't put it in for that reason. Two such cases are the following:

- Music. Try Frank Seide's Sound-Trecker or Antoine Rosset's Player Pro drivers. The C interfaces are available from the major ftp archives. Pascal users can find Pascal interfaces at *ftp://ftp.lysator.liu.se/pub/mac/source*. Another option is to use QuickTime's music playing abilities. For that, you can find the demo *Piggy* in my ftp archive.

- Fades and wipes other than the ones in GammaFade and MiscGraphics. MSG Demo includes lots of fine routines for CopyBits-based fades and wipes. For Pascal users, I've built a Pascal library of the most interesting calls, available from my ftp archive. *ftp://ftp.lysator.liu.se/pub/mac/sat* or *ftp://ftp.lysator.liu.se/pub/mac/source*

SECTION 6 - THE PROGRAMMING INTERFACE

THE INTERFACE

SAT Data types

```
type
{SAT's record for port, device and row list}
   SATPort = record
   port: GrafPtr;
   device: GDHandle;
   rows: Ptr;
   end;
SATPortPtr = ^SATPort;
```

In the SATPort, you generally only need to use the port, for SetPort, ShowWindow, CopyBits etc, or pass the entire SATPort to SAT procedures.

{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;
   rows, maskRows: Ptr;
   redrawProc: ProcPtr;
   drawProc: ProcPtr;
end;
```

The field colorData holds the image. This is for internal use, since it has different formats depending on depth. Do not expect this to always be a pointer to the image data!

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///).

The pointers rows and maskRows are used internally.

The redrawProc is a pointer to a callback procedure to be called on screen depth changes. Most faces will have nil here, but if you create the face with SATNewFace rather than SATGetFace, you need to redraw it, which may then be done in this callback procedure . The procedure should be declared as follows:

```
procedure MyRedrawProc(myFace: FacePtr; theDepth: Integer);
```

The FacePtr myFace is a pointer to the face. The integer theDepth tells what depth it has. (gSAT.initDepth may not have changed yet!) Before the redrawProc is called, the port is set to the face.

The drawProc is a pointer to a callback procedure that can be used for a face-specific draw procedure. It should usually be nil. The procedure should be declared as the mask blitter in "Writing your own blitters". This feature is extremely useful for advanced programmers, and is in frequent use in my own programs!

```
{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;
  oldFace: FacePtr;{ Used by SATRun2}
  dirty: Boolean;{ Used by SATRun2}
{ Variables for internal use by the sprites. Use as you please - and change 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 otherwise 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 lets you specify a region with which to clip the sprite. This is useful when you want a sprite to move behind large, static objects. However, the blitter resources do not support this field, only the safe blitters, using QuickDraw. Thus, SAT automatically calls SATSafeMaskBlit if the clip field is not nil.

Also note that the clip field has no effects on sprites where the face has no mask region! This is usually no problem, but the FastLoad has an option to skip the mask regions in order to speed up loading time.

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.

oldFace and dirty are used by SATRun2. If you don't call SATRun2, use them freely.

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

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.

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: SATPort;

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 SATSetPortOffScreen 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.

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: SATPort;

gSat.wind is a pointer to the window SAT uses. SAT expects this window to be frontmost when drawing direct-to-screen. The field gSAT.wind.port is the actual WindowPtr and can be used as such.

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 careful when removing sprites from the list yourself (both by modifying the linked list and using SATKillSprite). 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 only 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.

gSAT.faceRoot: FacePtr;

faceRoot is a pointer to the first face in the face list.

gSizeofSprite: Longint;

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

gSAT.colorFlag: Boolean;

colorFlag is a flag 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).

```
iconPort: SATPort;{Internal}
iconPort2: SATPort;{Internal}
bwIconPort: GrafPtr; {Internal}
```

These three ports are the ports you use when using SATSetPortFace, SATSetPortFace2 and SATSetPortMask, respecively. If you want to CopyBits to or from a face, you need the GrafPtr to each port. You can then use the port field in iconPort or iconPort2, or bwIconPort. When copying between two faces, you should SATSetPortFace to one, and SATSetPortFace2 to the other, to make both ports valid at the same time.

curRectBlit, curMaskBlit: ProcPtr;{Currently selected fast blitter}

These procedure pointers give you the current blitters. They are of interest if you write a blitter/custom drawProc of your own that has some need for calling the ordinary blitter.

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

Easy Initialization:

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

pictID	resource ID of a color picture
bwpictID	resource ID of a B/W picture
Xsize, Ysize	(maximum) dimensions of the animation area

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 uses 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. The 14" screen is 640*480 pixels - 640*460 without menu bar.)

After initializing, SATInit will also show gSAT.wind and update it.

If you disagree on any part of the setup SATInit does for you, try SATCustomInit instead! It will let you change practically everything.

Customized Initialization:

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

pictID	resource ID of a color picture
bwpictID	resource ID of a B/W picture
SATdrawingArea	(maximum) dimensions and position of the drawing area:
preloadedWind	a window in which the animation should take place, use nil for a new one
chosenScreen	the screen on which animation should take place, use nil fo main screen
useMenuBar	true if the animation area may overlap the menu bar
centerDrawingArea	true if the drawing area should be centered on the screen
fillScreen	true if SAT-created window should fill the screen
dither4bit	true if SAT should use dithering when running in 16 colors
beSmart	true if SAT should limit animation areas to screen bounds etc

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.port);
  SelectWindow(gSAT.wind.port);
  SATRedraw;
end;
```

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

PICTfittrue if SAT should resize pictures to fit the animation area

newSortingselected sorting method

newCollisionselected collision handling

searchWidthaffects collision detection

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, kVpositionSort, kKindCollision 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 Search 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.

kBothCollision (Added in SAT 2.4.0.) Search both forward and backward in the sprite list, and report collisions with the hitTask procedure. Functionally equivalent to kForwardCollision and kBackwardCollision, but is faster, especially for cases where only some of the sprites have a hitTask. (This mode makes kForwardCollision and kBackwardCollision obsolete, but they will stay for some time, until kBothCollision is thoroughly tested and also for backwards compatibility.)

The parameter searchWidth tells how far from a sprite the collision detection should search. This reduces the time spent searching for collisions, but can also be a source for errors if you set it improperly.

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;

resNumresource number for a 'cicn' resource

SATGetFace 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.) Calling SATGetFacePP with the fStorage field set to nil is equivalent to SATGetFace.

procedure SATDisposeFace (theFace: FacePtr);

theFacepointer to the face to dispose

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.)

You should not use this call to dispose faces that are allocated in unusual ways, i.e. with the FastLoad add-on.

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

kinda value for the kind field (application-defined usage)

hpos, vposposition of top-left corner

setuppointer to a procedure responsible for additional setup

function SATNewSpriteAfter (afterthis: SpritePtr; kind, hpos, vpos: Integer; setup: ProcPtr): SpritePtr;

afterthisa pointer to another sprite, defining desired place in the sprite list

kind, hpos, vpos, setupsee above

procedure SATKillSprite (who: SpritePtr);

whopointer to sprite to be disposed

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 the SpritePtr if a destructTask exists!

SATKillSprite is called by SATRun when it encounters a sprite with the task = nil. That is the most common way for it to be called.

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);
fasttrue if direct-to-screen blitters should be used (if available)

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).

procedure SATRun2(fast:Boolean);
fasttrue if direct-to-screen blitters should be used (if available)

SATRun2 works like SATRun, except that it checks for non-changing sprites in order to avoid redrawing them. If your program needs a fair amount of non-moving sprites (say, a Centipede game where sprites are used for the mushrooms), then you should use SATRun2 rather than SATRun. Run the Bricks demo to check out the difference!

NOTE: The current 1-bit and 4-bit rect blitters are not compatible with SATRun2. Projects using SATRun2 should not include those blitters. The 8-bit and 16-bit blitters, however, work.

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; dest: SATPortPtr; where: Point; fast: boolean);

procedure SATPlotFaceToScreen (theFace: FacePtr; where: Point; fast: boolean);

theFacePointer to face to be drawn

destDestination port

whereTop-left corner of destination

fastIf true, custom blitters are used (if available)

SATPlotFace draws the icon stored in a Face structure in the SATPort dest. The port in question must be at least as big as the drawing area.

The normal use for SATPlotFace is to draw Faces on backScreen, in order to modify the background. If you pass nil for dest.port, SATPlotFace assumes that you want the drawing in backScreen, plus calls SATBackChanged for you. (See below.) (This is not as elegant as it used to be. It might change.)

SATPlotFaceToScreen is a variant for drawing to the screen.

NOTE: In Pascal, you make a SATPortPtr from a SATPort myPort by typing @myPort.

procedure SATCopyBits (src, dest: SATPortPtr; srcRect, destRect: Rect; fast: Boolean); [OBSOLETE]

procedure SATCopyBitsToScreen (src: SATPortPtr; 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.

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

rRectangle enclosing the area that needs to be updated.

Use SATBackChanged when you have modified the background (i.e. after drawing to gSAT.backScreen) to tell SAT to update that part. SAT will then update it in the proper time, during SATRun.

procedure SATGetPort (var port: SATPort);

procedure SATSetPort (port: SATPort);

portThe port to get or set.

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

unction 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-initializes 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 SATRedrawOffscreen;{Draw all sprites off-screen}
```

SATRedrawOffscreen makes a full update of gSAT.offScreen by copying gSAT.backScreen and drawing all sprites. It does not copy to screen!

You may wish to call SATRedrawOffscreen after setting up sprites for a new situation (e.g. a new level), followed by copying gSAT.offScreen to the screen (e.g. SATRedraw, WipeIn, WipeOut or CopyScreen, the latter three in the Misc-Graphics add-on).

```
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 (wind: WindowPtr);

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.port is used. If you pass gSAT.wind (or nil), SAT also updates the part of the window that was under the menu bar by copying from gSAT.offScreen.

SATShowMBar also takes a window as parameter. Pass the same window that you passed when hiding the menu bar.

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.

PICT resource utilities:

```
procedure SATGetandDrawPICTRes (id: integer);

procedure SATGetandDrawPICTResInRect (id: integer; frame: Rect);

procedure SATGetandCenterPICTResInRect (id: integer; frame: Rect);
```

The functions above draw a *PICT* resource, either in its own rectangle, resized to fit a specified rectangle, or centered in/around a rectangle. They are very space-conservative, so your project will need less memory allocated than if you

use GetPicture and DrawPicture. SAT uses them internally for drawing the backdrop *PICT*.

You only need these functions if your program must draw very large *PICT* resources.

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 .

SATSetPortFace2 uses gSAT.iconPort2.

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 appreciate 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). This is called automatically if you call SATInit/SATCustomInit again, so you have little need of calling it yourself.

```
procedure SATMakeOffscreen (var portP: SATPort; rectP: Rect); {Make offscreen buffer in current screen depth and CLUT.}

function SATMakeOffscreen2 (var portP: SATPort; var rectP: Rect): OSErr;

procedure SATDisposeOffScreen (var portP: SATPort); {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 depth and color table as the other offscreens. SATMakeOffscreen2 is the same, but returns an error code on failure rather than using SAT's emergency exits. 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.)

```
procedure SATWindMoved;
```

If the SAT window (gSAT.wind.port) is moved, you should call SATWindMoved to tell SAT to recalculate its variables. This is only important if you use fast animation (e.g. SATRun(true)) and have blitter resources installed.

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 SAT's sound on and off. After SATSoundInit is called, the sound is on.

In older versions of SAT, SATSoundOff did not stop sounds being played. From 2.5.0, SATSoundOff will silence all channels immediately.

```
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 SATSoundPlay2 (theSound: Handle; priority: integer; canWait: Boolean);
```

The SATSoundPlay call has one drawback: the two ranges of priorities. The sounds over 10 are really treated differently from the ones below. Thus, a new call was added: SATSoundPlay2, which has the new flag skipIfSame. If skipIfSame is true, the sound will be skipped if it is the same priority as the one already playing in the assigned channel.

This is of somewhat limited use, since it is played on the least busy channel, but if you know that you are doing something more important in all channels but one, it is useful for certain frequent sounds that you rather want played to the end than cut off over and over.

SATSoundPlay is equivalent to:

```
SATSoundPlay2(theSound, priority, canWait, priority >= 5)
procedure SATSoundPlayEasy (theSound: handle; canWait: Boolean);
```

Both SATSoundPlay and SATSoundPlay2 have more parameters than you usually need. SATSoundPlayEasy is nice when you just want to fire off a sound and don't want to think about how. All that remains is the canWait flag.

SATSoundPlayEasy(theSound, canWait); is equivalent to:

```
SATSoundPlay(theSound, 1, canWait);
```

Nowadays (2.5.0), all these calls call SATSoundPlayVolume with full volume (256, 256). See below.

```
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.

```
function SATSoundDone: Boolean; {Any sound going on ?}
```

SATSoundDone returns true if any channel is busy. SATSoundDone, like SATSoundDoneChannel, inspects a list of flags that is updated when SATSoundEvents is called. Hence, if you want to wait until all sounds have completed or a certain channel is free, you must call SATSoundEvents between each check. To wait for all sounds to complete, do:

```
while not SATSoundDone do
  SATSoundEvents;
  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.

NOTE: The handle returned is a normal resource handle. If you load sounds from an external file, you must DetachResource or the sounds will go away when you close the file.

```
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 than 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!

NOTE: Channels are numbered from 1 and up!

```
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. SATSoundDoneChannel, like SATSoundDone, inspects a list of flags that is updated when SATSoundEvents is called. Hence, if you want to wait until all sounds have completed or a certain channel is free, you must call SATSoundEvents between each check. To wait for a certain channel, do:

```
while not SATSoundDoneChannel(myChanNum) do
SATSoundEvents;
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.

```
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).

```
procedure SATSoundPlayVolume (theSound: handle; priority: integer; canWait,
skipIfSame: Boolean; volume: Point);
```

SATSoundPlayVolume is similar to SATSoundPlay and SATSoundPlay2. It plays a sound with volume 0-256, separate for each channel. This is useful for stereo placement of sounds.

```
procedure SATSoundFadeChannel (chanNum: integer; volume: Point);
```

SATSoundFadeChannel sets the volume of a channel. This is particularily use-for for fading looped sounds. The volume only lasts until next time a sound is played on the channel.

```
procedure SATSoundLoop (firstSound, loopSound: Handle; chanNum: integer);
```

SATSoundLoop plays a sound continuously, in a loop. The call reserves the channel, and if desired, you must call SATSoundReserveChannel yourself you release it.

NOTE: The reason why SAT hasn't provided looped sound support until now is that I don't like unnecessarily large games. In 1992, modems were slow and disks were small, so a few music samples made games very tedious to download. Now, in 1997, modems are faster and disks larger, so I can stand some larger sound effects, but please, stay within reasonable limits! Music is usually best made with QuickTime MIDI or a MOD playing library.

Pattern Utility Routines

The pattern utilities are obsolete. The new add-on MiscGraphics.p are replacing them. The old pattern utilities will remain for some time for compatibility reasons. They work as well as the new ones, but they make unnecessary memory allocations, a marginal disadvantage tothe new ones.

Note that no matter what method you use to handle patters - SAT's old or new ones or QuickDraw directly - you will probably notice a loss of memory that looks very much like a memory leak. It isn't as bad as it seems. If you use the same patterns again, no further loss occurs. This is not my fault but a part of Color QuickDraw's design. (But by all means, prove me wrong! You have the source to MiscGraphics, so you know what I do!)

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.

Pixel array utilities

NOTE: Not part of SAT.lib! You find it in the add-ons.

A feature that was often requested in older versions was to draw starfields and other effects that works with many single pixels. Ordinary sprites are not well suited for this, since they have overhead for the bigger sizes and masks.

From SAT 2.3.5, the following types and routines are supported. A pixel is described by the record Pixel, which is a Point and four bytes of data for your use. The type Pixels is defined to be an array of Pixel. It should be allocated with NewPtr or NewPtrClear, so that GetPtrSize can be use to determine its size.

All routines support screen depths of 1, 4, 8, 16 and 32 bits.

```
type
   Pixel = record
   position: Point;
   data1, data2, data3, data4: SignedByte;
end;


Pixels = array[0..32000] of Pixel;
PixelPtr = ^Pixels;
procedure SATDrawPixels (pix: PixelPtr; var port: SATPort; value: Longint);
```

SATDrawPixels draws all the pixels in the array pix with the value value. NOTE: No border checks are done, and port is assumed to have a row start table (i.e. the row field must point to a valid table).

One call to SATDrawPixels will draw all the pixels in pix in the same color. If you want several colors, you need to use several arrays.

```
procedure SATCopyPixels (pix: PixelPtr; var src, dest: SATPort);
```

SATCopyPixels copies all the pixels in the array pix from the port src to the port dest.

NOTE: No border checks are done, and both ports are assumed to have row start tables.

```
procedure SATDrawPixelsSafe (pix: PixelPtr; var port: SATPort; value: Longint);

procedure SATCopyPixelsSafe (pix: PixelPtr; var src, dest: SATPort);
```

These two routines are the same routine as the ones above, except that they make border checks, so that you can't accidentally write outside the screen. This implies that they may be a bit slower. NOTE: These two are not even in the add-on lib any more, due to the space limitations in Think C. You need to edit the Pixels.p file and recompile the library if you need them.

Scrolling

The following call can help you with one of the scrolling methods, step-scrolling. Note that this is just one of several ways to do scrolling.

```
function SATStepScroll (viewPoint: Point; marginH, marginV, scrollSpeed: Integer): Boolean;
```

SATStepScroll checks if viewPoint is closer to the window border than the margins marginH and marginV. If it is, it scrolls the screen so that the viewPoint is centered.

SATStepScroll assumes that the animation window (gSAT.wind.port) either holds the entire animation area (in which case no scrolling is ever needed) or that the animation area is allowed to fill the entire window. In other words, if you need some graphics that must not be scrolled (e.g. scores or scroll bars), put it in another window! If you don't use blitters to screen (i.e. SATRun(false)) you can also set the clip region of the window to the animation area when SAT is drawing, and restore to draw scrores, scrollbars etc.

There are a few pitfalls when making a scrolling SAT-using game. See the scrolling section above for details.

Utility Routines

Most of these should be rather self-expanatory.

```
procedure SATDrawInt (i: integer);

procedure SATDrawLong (l: longint);

function SATRand (n: integer): integer;

function SATRand10: integer;

function SATRand100: integer;

procedure SATReportStr (str: str255);

function SATQuestionStr (str: str255): boolean;

function SATFakeAlert (s1, s2, s3, s4: Str255; nButtons, defButton, canceButton: 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);

procedure SATInitToolbox;

procedure SATGetVersion (var versionString: Str255);
```

SATDrawInt and SATDrawLong converts the argument to a string and draws it with DrawString.

SATRand(n) routines produce a random number in the range [0..n-1]. SATRand10 is equivalent to SATRand(10), and SATRand100 to SATRand(100). Note that SAT doesn't seed the random number generator for you. Call GetDateTime(randSeed) (Think P), GetDateTime(qd.randSeed) (CW Pascal) or GetDateTime(&qd.randSeed) (CW C) to do that.

SATReportStr makes an alert with an "OK" button and the string str. SATQuestionStr 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. This is an improved version of FakeAlert from the TransSkel package.

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).

SATSetMouse 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.

SATInitToolbox initializes the Mac toolbox. You can use it to get a one-line initialization in any environment except Think Pascal, which does it automatically. It performs the following initializations:

```
InitGraf(@thePort);

InitFonts;

InitWindows;

InitMenus;
```

```
TEInit;

InitDialogs(nil);

InitCursor;

MaxApplZone;
```

SATGetVersion returns a Pascal string with the version number of the library.

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; pre-
loadedWind: WindowPtr; chosenScreen: GDHandle; useMenuBar, centerDrawin-
gArea, 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);

procedure SATRun2(fast:Boolean);

Drawing

procedure SATPlotFace (theFace: FacePtr; dest: SATPortPtr; where: Point; fast: boolean);

procedure SATPlotFaceToScreen (theFace: FacePtr; where: Point; fast: boolean);

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

procedure SATCopyBitsToScreen (src: SATPort; srcRect, destRect: Rect; fast: Boolean); [OBSOLETE]

procedure SATBackChanged (r: Rect);

SetPort Replacements

procedure SATGetPort (var port: SATPort);

procedure SATSetPort (port: SATPort);

procedure SATSetPortOffScreen;

procedure SATSetPortBackScreen;

procedure SATSetPortScreen;

Maintainance

function SATDepthChangeTest: boolean;

procedure SATDrawPICTs (pictID, bwpictID: integer);

procedure SATRedraw;

procedure SATRedrawOffscreen;

Menu Bar

procedure SATShowMBar(wind: WindowPtr);

procedure SATHideMBar(wind: WindowPtr);

PICT Resource Utilities

procedure SATGetandDrawPICTRes (id: integer);

procedure SATGetandDrawPICTResInRect (id: integer; frame: Rect);

procedure SATGetandCenterPICTResInRect (id: integer; frame: Rect);

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 SATWindMoved;

procedure SATMakeOffscreen (var portP: SATPort; rectP: Rect);

function SATMakeOffscreen2 (var portP: SATPort; var rectP: Rect): OSErr;

procedure SATDisposeOffScreen (var portP: SATPort);

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.}

function SATSoundDone: Boolean; {Any sound going on ?}

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:

(obsolete - use MiscGraphics.p instead)

```
procedure SATPenPat (SATpat: SATPatHandle);

procedure SATBackPat (SATpat: SATPatHandle);

function SATGetPat (patID: integer): SATPatHandle;

procedure SATDisposePat (SATpat: SATPatHandle);
```

Pixel arrays

NOTE: Not part of SAT.lib! You find it in the add-ons.

```
procedure SATDrawPixels (pix: PixelPtr; var port: SATPort; value: Longint);

procedure SATCopyPixels (pix: PixelPtr; var src, dest: SATPort);

procedure SATDrawPixelsSafe (pix: PixelPtr; var port: SATPort; value: Longint);

procedure SATCopyPixelsSafe (pix: PixelPtr; var src, dest: SATPort);
```

Scrolling

```
function SATStepScroll (viewPoint: Point; marginH, marginV, scrollSpeed: Integer): Boolean;
```

Misc

```
procedure SATDrawInt (i: integer);

procedure SATDrawLong (l: longint);

function SATRand (n: integer): integer;

function SATRand10: integer;

function SATRand100: integer;

procedure SATReportStr (str: str255);

function SATQuestionStr (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 SATSetMouse (where: Point);

function SATGetCicn (cicnId: integer): CIconHandle;

procedure SATPlotCicn (theCicn: CIconHandle; dest: GrafPtr; destGD: GDHandle; r: Rect);

procedure SATDisposeCicn (theCicn: CIconHandle);

procedure SATInitToolbox;

procedure SATGetVersion (var versionString: Str255);
```