

# **Sprite Animation Toolkit**

**by Ingemar Ragnemalm**

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

For Think Pascal or Think C on the Macintosh.

Copyright © 1992, 1993 by Ingemar Ragnemalm. All rights reserved.

Version 2.0b5 (color version, 5th beta)

## Contents

1. Introduction	1
Foreword	1
Legal terms	1
Background	1
Features and limitations	2
Disclaimer	2
Acknowledgements	3
Related packages	3
2. Packing list	4
Files in the toolkit	4
Example programs	4
Other files	6
3. Usage	7
General principles	7
Using SAT	7
Initialization	8
How to write a new sprite unit	9
Sorting	11
Collision detection and handling	12
Additional notes on collision detection:	13
Faceless sprites	13
4. Bits and pieces	14
Responding to update events	14
Miscellaneous functions	14
Modifying the background	14
Mac-friendly programming	15
The GDevice business	15
Dying with dignity	16
Some questions that I expect might become frequent	16
The C interface	17
5. The programming interface	19
SAT Data types	19
Global variables	20
SAT procedures	21
Easy initialization:	21
Customized initialization:	21
Sprite management:	22

Running the animation:	23
Drawing:	23
Maintenance:	24
Menu bar hiding:	24
Special functions, advanced calls:	25
Sound routines:	26
Pattern utility routines:	27
Utility routines:	27
Final words	29
Quick reference	30

# 1. Introduction

## Foreword

This is Sprite Animation Toolkit, hereafter referred 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.

This document describes version 2, the first *color* version of SAT, which is a major revision of the older B/W version from spring 1992, which never got beyond the beta version due to lack of beta tester feedback. This new version has a simpler interface than the old version, using fewer calls to accomplish more, and adds a bunch of utility functions and options.

## Legal terms

This package consists of this manual, the SAT library itself (Pascal and C versions) plus resource files, project files and source code to the example programs *SATminimal*, *SATcollision*, *SATcollisionII*, *SAT Invaders* and *HeartQuest*, plus *SATminimalC* in C.

This package is free of charge, under the following conditions:

- No part of the SAT package may be sold for profit without my written permission. The only exceptions are compiled programs made with SAT by other authors. For these programs, the terms below apply:
  - If you use SAT to produce a game or program that is distributed, as Public Domain, freeware, shareware, or similar conditions you shall send me a free copy and mention SAT in the doc and/or "About" box.
  - If you use SAT to produce a commercial program, you must have my written permission. Please contact me for release conditions. (I want in! ;-)
- My internet address is [ingemar@isy.liu.se](mailto:ingemar@isy.liu.se), and my real address is:

Ingemar Ragnemalm  
Plöjaregatan 73  
S-58330 Linköping

## SWEDEN

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

### **Background**

I have always liked to make computer games. It has been one of my hobbies since the late 70's. When I started using Macs, of course I wanted to make some games for it too. Among the games I liked were games like Glider, Zero Gravity and Cairo Shootout: shareware games with nice, smooth animation over a detailed background. After occasional hacking over many weekends,

using code fragments from comp.sys.mac.programmer, I got a horse race game working, and later a Space Invaders-style game, which has been released as freeware as Slime Invaders, and more recently a Pacman game (Bachman) and a game for my wife (HeartQuest). Now, I think the routines I'm using have become good enough to distribute, to help others make nice games.

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

## Features and limitations

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

SAT produces flicker-free animation with sprites over a background. As implied above, the goal was to produce animation of the quality we find in games like Glider (by John Calhoun) and Cairo Shootout (by the late Duane Blehm). The main problem SAT solves is drawing, the sprite animation, but it also has a bunch of other features like asynchronous sound (one channel - the rest is up to you), 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 with fast graphics in 1, 4 and 8 bits (that is 2, 16 and 256 colors), with support for switching between bit depths even after initialization. The sprites can have any size that you can use in a "cicn" resource, that is, up to 64·64. [BUG NOTE: The fast routines can't use "any" size right now, due to a bug in the sprite plotting routines. Widths divisible by 8 are ok.] It can not use a scrolling background yet. (We need even faster routines to do that.)

SAT is written in a "pseudo-object-oriented" fashion. Perhaps Object Pascal or TCL

would have been better, but I am not experienced enough with them. Besides, the experience I have with OOP tells me that it might slow things down, and this is supposed to be fast.

Possible future improvements:

- Faster 1-bit graphics.
- Sprite faces from other resources than "cicn".
- Some kind of support for scrolling games.

## **Disclaimer**

This is a hack I've made over the weekends (lots of them). Don't expect it to be a masterpiece; I made it just for fun. I have tried to tidy it up so it should be useable for others, but I know there are many things that could be improved. Many variable names are badly chosen. Especially, the terminology does not follow normal OOP terminology very well. I hope you will understand what I mean, and I appreciate corrections.

## Acknowledgements

Special thanks to Juri Munkki for help with the color drawing routines, to Michael A. Kelly for sharing the code from which I made my direct-to-screen code, to Tony Myles for helpful advice, 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 Harnett for the TransSkel package. I use it all the time, and find snippets from its demos everywhere in SAT.

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

## 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 Toolkit* (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.

As so often before, competitors who are doing the same thing show up right when you are getting somewhere. (When I had Bikaka working, Hextris was released, then when I released Hexmines, MacMines and Saddam's Revenge were released, and when I released Bachman, Tsuji's Pacman and Macman Classic were released... That's life, I guess.) Recently (this is late spring -93) two other sprite handling packages showed up:

One is Ricardo Batista's *Sprite Manager*, which was distributed on one developer CD. I don't know if it is still being worked on it or if we should consider it dead.

The other is *SpriteWorld* by Tony Myles. Well-working beta version available from various archives.



## 2. Packing list

### Files in the toolkit

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

#### Pascal library:

##### **SAT.p**

Interface file for the library.

##### **SAT.lib**

The Sprite Animation Toolkit compiled to a Think Pascal library.

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

#### C library:

##### **SAT.h**

Include file for using the C library.

##### **SATC. $\pi$**

The Sprite Animation Toolkit compiled to a Think C library.

### Example programs

Six example programs are included in SAT, namely "SATminimal", "SATcollision", "myPlatform", "SAT Invaders" and "HeartQuest". SATminimal is extremely simple, making a trivial animation until the user clicks the mouse. SATcollision adds the simplest collision handling. SATcollision demonstrates a more flexible collision handling plus simple background manipulation. MyPlatform demonstrates one fairly simple way to make platform games with SAT. The player sprite isn't perfect - most of all, it jumps too high. I might improve it in a later version.

SAT Invaders is somewhat more elaborate, a stripped down Space Invaders. HeartQuest is the biggest example, a complete arcade game with scores, various settings and high score list. All the demos are pretty ugly, quick, unpolished hacks, graphic-wise, but the artwork is not the problem SAT is designed to solve for you.

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 SATcollision to get the hang of it (and complain to me if you don't understand).

In the list below, C files are in parenthesis together with corresponding Pascal files.

SATminimal source files:

**SATminimal.proj, SATminimal.π.rsrc (SATminimal.π)**

Project file and resource file.

**sMySprite.p (sMySprite.c)**

A sprite unit.

**SATminimal.p (SATminimal.c)**

The main program, which initializes SAT and the sMySprite sprite, and runs the animation.

SATcollision source files:

**SATcollision.π, SATcollision.rsrc**

Project file and resource file.

**sMrEgghead.p, sApple.p**

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

**SATcollision.p**

Main program.

SATcollision][ source files:

**SATcollision][.proj, SATcollision][.π.rsrc,****sMrEgghead][.p, sApple][.p, SATcollision][.p**

(C files: SATcollision][.π, sMrEgghead][.c, sApple][.c, SATcollision][.h)

See SATcollision. SATcollision][ is slightly bigger, adding some features and icons.

MyPlatform source files:

**myPlatform.π.rsrc, myPlatform.proj (myPlatform.π)**

Resource and project files.

**myPlatform.p (myPlatform.c, myPlatform.h)**

Main program (and C header file).

**sPlatform.p (sPlatform.c)**

Sprite unit, defining static platforms as "invisible sprites".

**sMovPlatform.p, sHmovPlatform.p (sMovPlatform.c, sHmovPlatform.c)**

Two sprite units defining platforms moving vertically and horizontally.

**sPlayerSprite.p (sPlayerSprite.c)**

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



SAT Invaders source files:

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

**gameGlobals.p**  
Global variables and resource numbers.

**soundConst.p**

Sound resource numbers and their handles.

**sMissile.p, sEnemy.p, sShot.p, sPlayer.p**

Four sprite units.

**main.p**

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



HeartQuest

HeartQuest source files:

**HeartQuest.π, HeartQuest.π.rsrc**

Project file and resource file.

**gameGlobals.p**

Global variables and resource numbers.

**soundConst.p**

Sound resource numbers and their handles.

**scores.p**

Score and high score list handling.

**sPoints.p, sHeart.p, sBonus.p, sFlypaper.p, sPlayer.p**

Five sprite units.

**gameWindow.p**

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

**main.p**

The main program, mostly initializations.

**Other files****ICN# -> cicc**

ICN# -> cicc is a small utility that converts "ICN#" resources to "cicc" resources. It is included for those of you who do some or all development on a Mac with 68000, on which the "cicc" editor in ResEdit won't run.

(Warning: Use ICN#->cicc with some caution. The current version is a quick hack to solve the problem, nothing else. Avoid saving on an existing file.)

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

**transSkel.p**

The public domain subroutine package by Paul duBois and Owen Harnett. It takes care of all tedious window and menu handling, all the trivial parts that makes many programs so unnecessarily large. Sadly, it is still not MultiFinder aware, but MultiFinder and System 7 features can be implemented on top of it. There are C versions available as well.

Finally, a file that is needed for C users:

**μRuntime.π**

This is nothing but μRuntime.lib converted to a C library (project). As such, it's not a part of SAT, but I have included it so you won't have to bother about converting it yourself. (In earlier versions, it was included in SATC.π, but SAT has grown too big since then.) I guess I should say: portions © Symantec corp, because that's what it is. I hope they will not mind me including it, since it is necessary for using the C library.

## 3. Usage

### General principles

SAT manages a list of sprites, describing position, current icon (preloaded to a face structure, see below) and action procedures called each frame and in collisions, if desired. You seldom have to access the list yourself, but if you do, you can get the first item with the global variable `sRoot` and follow the pointers through the list from there.

SAT uses two offscreen bitmaps/pixmaps, named *offScreen* and *backScreen*. *backScreen* holds the background image, the backdrop. You can, when needed, `SetPort(backScreen)` 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 *SATpict* (for use in color) and *SATbwict* (for use in b/w). The background is drawn by SAT when SAT is initialized and when the screen depth is changed.

SAT by default uses the main screen. (There is a way to select another screen, but this feature is not yet tested and most likely does not work yet!) 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.

SAT can also produce asynchronous sound. This is done in one channel only. (You may use other channels for other tasks, if there are any left.) SAT uses Sound Manager if available, otherwise it switches to Sound Driver. When using Sound Manager, SAT keeps its channel open for extended periods. Thus, you must call `SATSoundShutup` before quitting, to dispose of the sound channel.

### Using SAT

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

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



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

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

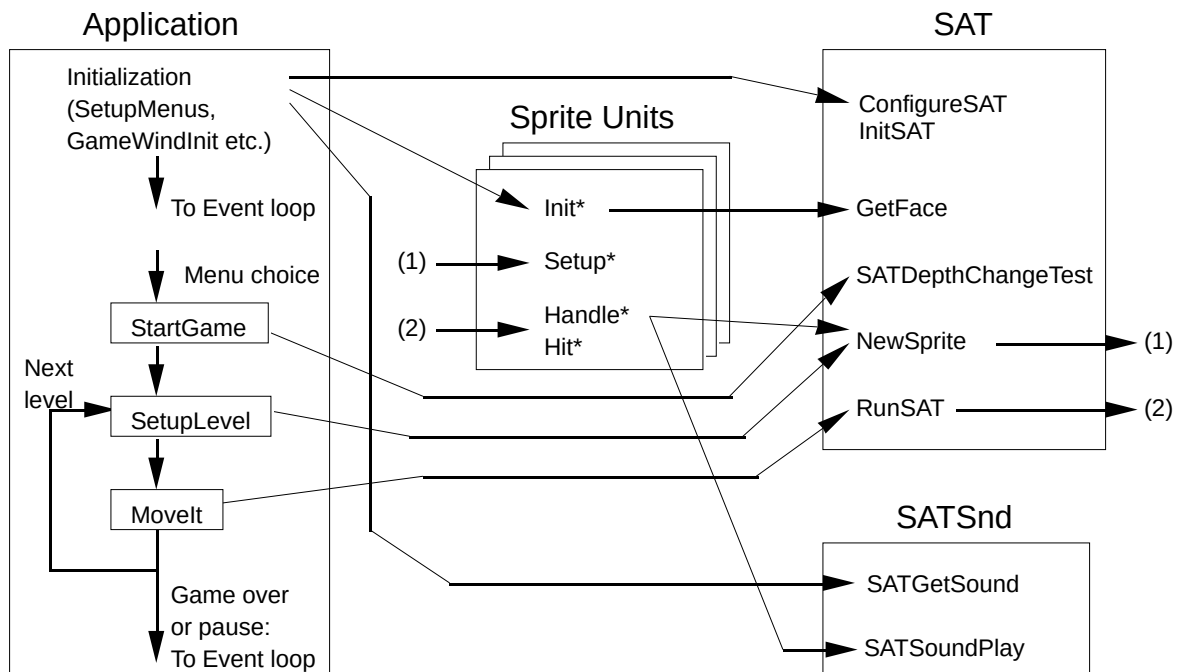
A main loop for the game. In SAT Invaders, this is the MoveIt procedure, where RunSAT is called repeatedly until the game over condition is fulfilled. It is possible to call this as a background task, from the normal event loop (where all normal window and menu handling is performed), but my experience is that action games will not run smoothly enough this way.

In SATminimal, the main loop is a very small loop, calling RunSAT until the user clicks the mouse, and calling TickCount to limit speed to the system clock.

When a game is not in progress, the program should be as most other Mac applications, with an event loop with menu and window handling etc. If you call *SATDepthChangeTest* often, either on update events (recommended) or before starting a new game, SAT will be able to change screen depth as needed.

Several sprite units. SAT Invaders has four such units: mPlayer, mEnemy, mShot and mMissile. SATminimal has only a single sprite unit.

The following figure shows an outline of the typical SAT-using application. "Application" and "Sprite Units" are the parts that have to be rewritten for every new game, while "SAT" and "SATsnd" are in the SAT library. Procedure names refer to the ones used in SAT Invaders. (1) and (2) are procedure calls that wouldn't fit in the drawing.



*Figure 1. Outline of a typical program.*

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

## Initialization

Many, even most, of the routines in SAT demand that SAT has been initialized, either with InitSAT or CustomInitSAT. For example, GetFace preloads the chosen icon in the current screen depth, which demands that SAT knows what device it should use.

Thus, initialize SAT before calling other SAT routines, except ConfigureSAT and SATSetSpriteRecSize.

## How to write a new sprite unit

In the following, I use the term *sprite unit* for the file defining a sprite type. For OOP people, this is something similar to a *class*. With *sprite*, I refer to a specific object on the screen, an instance of the sprite unit, created with the NewSprite (or NewSpriteAfter) call.

When you need a new kind of sprite, a moving object, you should make a separate unit of it, a sprite unit. The unit may contain any private procedures needed, but four procedures are standard:

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

The Init\* procedure has no parameters.

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 NewSprite when a new sprite is created. If no setup is needed, you can omit it and pass **nil** for it in NewSprite. Most sprites will need some setup.

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 2).



Figure 2. A snapshot from HeartQuest (with its old background), where an sPoints sprite has just been created when the butterfly took a bonus (sBonus) sprite.

Here follows the source for the sprite unit:

```
unit sPoints;  
interface
```

```

uses
  SAT;

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

implementation

var
  pointsFace, fPointsFace: FacePtr;

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

procedure SetupPoints (mp: SpritePtr);
begin
  mp^.face := pointsFace;
  mp^.mode := 0;
  mp^.kind := 0;
  SetRect(mp^.hotRect, 3, 4,29,32); {Not needed}
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 *GetFace* 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 *face* 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 a call like this:

```
sp := NewSprite(0, hpos, vpos, @HandlePoints, @SetupPoints, nil);
```

*hpos* and *vpos* are the coordinates where the sprite should appear. The 0 is the value to put in the sprite's *kind*. (0 means that the sprite is neutral, never collides with anything.) The *NewSprite* 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:

- 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.
- *task* is a pointer to the Handling\* procedure.
- *hittask* is a pointer to the Hit\* procedure.
- To remove a sprite, set its *task* field to **nil**.
- Collisions are detected by inspecting the *kind* field and see if it has changed, or they are resolved in the Hit\* (*hittask*) procedure.
- To make a sprite harmless (not react on collisions), set its *kind* field to zero (0). (KindCollision only.)
- To move a sprite, change its *position* field. (Coordinates of upper left corner.) Check against borders using *offSizeH* and *offSizeV*.
- To change the appearance of a sprite, change its *face* field. The suggested convention for doing this is to make an array of *FacePtr*'s, and use the *mode* field to index the array. For example, if you use five icons, increment *mode* each frame until it reaches 5, and then reset it. See the examples.
- When you want to change the behavior of a sprite, it is perfectly legal to change the Handle\* procedure by setting the *task* field to the address of another procedure.
- When you make your own game, use lots of icons to get animation. This means lots of fiddling with icons (and/or good skill with a raytracer), but it is worth it. (My "Bachman" Pacman-clone game uses almost 200 icons, and Andrew Welch's Maelstrom uses over 500 – often as many as 40 for a single object!)

## Sorting

The sprite list is incrementally sorted during the animation. One step of "BubbleSort" is performed for each frame. As the default, the sprites are sorted in order of their position.v (*VPositionSort*), but you can change that to be according to the layer field (*LayerSort*) or turn it off completely (*NoSort*).

If you use the default sorting, a sprite located above another will be drawn before the other, so if they overlap, the higher one will appear to be farther away from the viewer. This sorting also allows the collision detection to work efficiently by only searching as long as it finds sprites within a certain distance (32 by default, but this can be changed using *ConfigureSAT*).

The sorting is a good reason for using *NewSpriteAfter*. If one sprite is created by another one (for example, a shot), it will be placed in the right part of the sprite list if *NewSpriteAfter* is used. If *NewSprite* is used, it may take a few frames before the sprite is in the right part of the list.

## **Collision detection and handling**

Collision detection is a dirty chapter, since it is so 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 SAT Collision and SAT Collision JJ demos!*

SAT does collision detection by checking if the hotRects (displaced by position) of sprites overlap. Then, SAT provides two ways to report collisions. In both cases, the effects of the collision is expected to be resolved by the sprite units, not by the main program.

The following two methods are supported:

### 1) SAT changes the *kind* fields of the colliding sprites.

To use this kind of collision detection, use *KindCollision* in *ConfigureSAT*. Then, only sprites with different signs on their *kind* fields can collide (positive vs negative).

When colliding, sprites with *kind*>0 are assigned a kind of 10. Sprites with *kind*<0 are assigned a kind of -10. (Sprites with *kind*=0 are neutral and don't collide at all, e.g. explosions.) The sprites can check for changes in the *kind* field in their *Handle\** procedures. This tells them only that they have collided, not with *what*.

### 2) SAT calls callback routines for each sprite.

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

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

where \*, by convention, is the name of the sprite. 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 *hittasks*, use any kind of collision detection except *NoCollision*, but note that when using *KindCollision*, only sprites with different signs on the *kind* field can collide.

Typical things to do when a collision occurs include:

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

Note: The collision detection is dependant on the sorting chosen.

All-to-all collision detection: If *NoSort* 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 *VPositionSort* or *LayerSort* 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 most cases, *VPositionSort* and *KindCollision*, the defaults, will be what you need - fast and easy to handle.



**Additional notes on collision detection:**

Non-rectangular sprites:

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

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

#### Custom collision detection (Advanced programmers):

If you need some other collision detection scheme, you can write it yourself. You can, for example, let each sprite search the sprite list in the Handle\* routine. This is not a recommended method (why re-invent the wheel?), but might be of interest in special cases.

[I could make an example of this, but I think I'll leave it to the hackers who want it.]

## **Faceless sprites**

It is legal for a sprite to have no face at all, that is, to assign the face to **nil**. In such a case, the sprite will be invisible, but it can still collide with other sprites. This can be used for collision detection with static objects, drawn in the background. By making such objects faceless, they are not drawn over and over again, which will save time.

(Note: I have made a test program using this, where faceless sprites are used for making platforms in a "platform game". This program, myPlatform, is part of the current distribution but is likely to be removed or at least heavily revised. It shows one out of many ways to implements platforms using faceless sprites. Unfortunately, it has rather poor controls so far.)

Note that this is not the only way to make moving sprites interact with static objects. You can also make your own structures. A typical approach is to use a 2-dimensional array describing a maze etc, and let the sprites check that array to see where they are allowed to go. (This is what I do in Bachman.) Use what seems best for your problem.

## 4. Bits and pieces

Notes and advice of various kinds.

### Responding to update events

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

When you get an update event to `SATwind`, the simplest response is:

```
ignore := SATDepthChangeTest;  
PeekOffScreen;
```

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`.

### Miscellaneous functions

SAT includes a number of utility functions that you may or may not need. Some are of a general nature, such as *DrawInt* and *DrawLong* (which simply does `NumToString` followed by `DrawString`), or *Rand* (generating random integers in a given range). I hope that these utilities are more or less intuitive.

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

### Modifying the background

Many games can be done over a static, never changing background. However, some 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 `backScreen`, draw what you

like, and then CopyBits it to the screen. However, that will often interfere with sprites, causing flicker. With the function *SATBackChanged*, you can send a Rect to SAT, which will then update the screen when the time comes.

A related function is *SATPlotFace*. It takes an icon in the form of a FacePtr (that is, the icon is preloaded to a format that SAT can use for direct-to-screen drawing) and draws it where you like. If you pass **nil** as GrafPtr, SAT assumes that you want it drawn in backScreen and calls SATBackChanged for you. You can use it for other tasks, drawing on the screen, in offScreen or

other ports, but that should be done with caution. The port to draw in must be a CGrafPort on color Macs, and must be a GrafPort otherwise (i.e. 68000-equipped Macs).

SATPlotFace has a cousin, SATPlotFaceToScreen, that you should use for plotting sprites directly on the screen.

## Mac-friendly programming

When SAT takes care of some of the animation issues for you, what more is there to think about 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, 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!** SATwind usually covers the entire screen, which is nice when the game is running. However, if the user switches to Finder, SATwind 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 off 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, **check the system version** - it's much nicer than crashing or quitting unexpectedly.

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 offSizeH and offSizeV for checking game area bounds rather than hard-coded numbers.

## The GDevice business

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

Each of our two offscreen grafports (offScreen and backScreen) have a sidekick, its GDHandle. QuickDraw, and CopyBits in particular, works much better when the right

GDevice is set. I have tried to hide this as far as possible in SAT, giving you backwards compatibility without bothering with ColorFlag or other dependencies.

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

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

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

```
{Set up a level, draw objects in backScreen}  
SATSetPortBackScreen;
```

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

For those interested: this is equivalent to:

```
SetPort(backScreen);
if colorFlag then
  if backScreenGD <> nil then
    SetGDevice(backScreenGD);
  ...do all your drawing here...
SetPort(SATwind);
if colorFlag then
  if ourDevice <> nil then
    SetGDevice(ourDevice);
```

## Dying with dignity

When SAT encounters a fatal error, usually running out of memory, it will display an error message and quit. The error handling was designed 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 (e.g. in another language), 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.

## Some questions that I expect might become frequent

– Can I remove the black borders?

• Yes, you can. The black borders are drawn by `PeekOffscreen`, which you can replace by `CopyBits`'ing from `offScreen` to `screen`:

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

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

– Can SAT be used in a draggable window?

• For now, avoid it if you want fast animation. It can be done if you update the variables `ox` and `oy` appropriately – but SAT has, for speed, no checking against screen borders. It is best avoided altogether. (It should work when using "safe" animation.)

– How do I change the background? If I draw a new image in `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 *SATbw pict*, since they tell SAT



where to go for the new PICT if it has to redraw them (i.e. when the screen depth changes). 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?

• No - not with any help from SAT. You are totally on your own here. You can do it by building your own Face structures and link them into the face list - if you must.

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

• With Pascal, it should work, though you will have to figure out what files to put in the projects and in what order, so I don't recommend it. For C, no - the C library is incompatible with versions prior to v 5. It should be possible to use it with Think C version 6, but I can't verify that myself.

– 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.

– Can I do animation over a scrolling background with SAT?

• No, not yet. It's quite possible to do, though it must be done in a fairly small area with a system as general as SAT, to be fast enough.

– One of my sprites draws some garbage where it shouldn't!

• SAT expects your 'cicn's to be "clean", with no drawings where the mask is zero. 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.

SAT also expects the cursor to be hidden, or you can get so called "mouse droppings" when the cursor is moved over a sprite. This is avoided by using HideCursor or ShieldCursor.

[Bug note: The current SAT draws sprites with certain widths incorrectly when drawing directly to screen. For now, use sprites with widths divisible by 8.]

– 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 ourDepth to determine what routines to use.

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

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

– I'm running out of memory, despite setting X ridiculously high!

• 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 on a 14" monitor, but may need more memory if run in bigger screen depths or on bigger screens (if you scale them up to your screen).

## The C interface

So far, I have assumed that you are using SAT from Think Pascal. There is, however, a C interface, in the form of a library (SATC. $\pi$ ) and a header file (SAT.h). Three of the demos, SATminimal, SAT Collision ][ and myPlatform, are included in C.

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

One note, though: All callback functions, functions that SAT calls using procedure pointers you provide, must be declared "pascal" (e.g. `pascal void HandleSprite()`).

## 5. The programming interface

### SAT Data types

type

{Information about a "face", i.e. a color icon. You hardly have to bother about this data type.}

```
FacePtr = ^Face;
Face = record
  colorData: Ptr;
  resNum: integer;
  iconMask: BitMap;
  rowBytes: integer;
  next: FacePtr;
  maskRgn: RgnHandle;
end;
```

{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 centered around origo }
{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. }
{ SAT variables that you shouldn't change: }
  oldpos: point; { The 'task' routine is not allowed to change this! }
  next, prev: SpritePtr;
  r, oldr: Rect;
{ Variables for internal use by the sprites. Use as you please. }
  layer: integer; {For free use, or for sorting.}
  speed: Point; { Can be used for speed, but not necessarily. }
  mode: integer; { Usually used for different modes and/or tells what face to use next.}
  appPtr: Ptr; {Pointer for use by the application - i.e. pointer to extra data}
  appLong: Longint; {Longint for free use by the application.}
end;
```

```
BMPtr = ^BitMap;
```

When a sprite is created, the fields are initialized as follows:

*kind*, *position*, *task*, *hittask* are set according to parameters to NewSprite or NewSpriteAfter.

*speed* is set to (0,0). *mode* is set to 0. *Face* is set to **nil**.

Note that this implies that if you customize the size of the sprite record, variables replacing *speed* and *mode* are zeroed, and if you shorten the record, *mode* must still fit in it.

## Global variables

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

*offScreen* and *backScreen* point to the two offscreen GrafPorts used by SAT. They are initialized with *InitSAT* (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.

When running in color, *offScreenGD* and *backScreenGD* are the offscreen graphic devices for each of our two offscreen GrafPorts. Certain calls (SATPlotFace, SATCopyBits) require both the GrafPtr and the GDHandle.

`offSizeH, 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.

`SATpict, SATbwPict: 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 InitSAT, CustomInitSAT and SATDrawPICTs calls. The value zero means no PICT.

`SATWind: WindowPtr;`

*SatWind* is a pointer to the window SAT uses. SAT expects this window to be frontmost when drawing direct-to-screen. (This WindowPtr is also returned by InitSAT and CustomInitSAT.)

`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. Don't remove sprites from the list yourself. If you do, they will not be erased properly.

`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.

`ourDepth: integer;`

*ourDepth* gives the current screen depth. You can inspect it if you want to warn the user about using a bit depth that is not supported, or use it when making additional offscreen pixmaps, if needed.

## SAT procedures

### Easy initialization:

function InitSAT (pictID, bwPictID, Xsize, Ysize: integer): WindowPtr;

*InitSAT* does all of the initializations needed for SAT. It initializes 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 *SATwind*) fills the main screen, and use a drawing area that is *Xsize\*Ysize* pixels. If *Xsize\*Ysize* can't fit on the screen, the screen size is used. (Classic size, excl. menu bar: 512\*322.)

### Customized initialization:

function CustomInitSAT (pictID, bwPictID: integer; SATdrawingArea: Rect; preloadedWind: WindowPtr; chosenScreen: GDHandle; useMenuBar, centerDrawingArea, fillScreen, dither4bit: Boolean): WindowPtr;

*CustomInitSAT* is a more powerful version of *InitSAT*, for programmers with specific needs. 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.

The integers *pictID* and *bwPictID* work as in *InitSAT*.

*SATdrawingArea* is a rectangle that specifies where the drawing area should be, in global coordinates. The rectangle is the *maximum* area you can get. 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.

*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.

*ChosenScreen* specifies a screen (device) on which SAT should run its animation. Pass **nil** to get the main device. [**WARNING:** I don't usually have access to any Mac with more than one screen, so this feature is not tested much!] If *ChosenScreen* is not the main device, *UseMenuBar* is ignored. (Only the main device has a menu bar.)

*UseMenuBar* tells SAT that it should use the menu bar space if needed, since we intend to hide the menu bar while animation is in progress. (See also *HideMBar* and *ShowMBar*.) If you intend to hide the menu bar, pass **true**. If you pass **false**, the drawing area is clipped in order not to touch the menu bar.

If *CenterDrawingArea* is true, *SATdrawingArea* is centered on the main screen.

If *FillScreen* is true, the created window fills the whole screen. Otherwise, the window is set to the *SATdrawingArea* rectangle. If *preloadedWind* is not nil, *FillScreen* is ignored.

If *Dither4bit* is true, SAT dithers all sprites when running in 4-bit color. This will usually look a lot better if the icons are drawn in 256 colors. If your icons are drawn in 16 colors, you may want to turn this off.

Calling CustomInitSAT(pictID, bw pictID, r, **nil**, **nil**, false, true, true, true, true), where r is a rectangle with appropriate width and height, is equivalent to a call to InitSAT.

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

*ConfigureSAT* lets you set certain parameters that affect SAT's behaviour. It usually should be called before *InitSAT* or *CustomInitSAT*, during program startup, but can also be called later. If you don't call it at all, SAT defaults to *false*, *VpositionSort*, *KindCollision* and 32.

If *PICTfit* is true, any background PICTs (*pictID* and *bw pictID* above or the globals *SATpict* and *SATbw pict*) are scaled to fit the drawing area. The *NewSorting* parameter tells SAT how it should sort the objects. You have the following options:

VPositionSort: Sort after the position.v field. Makes low sprites appear to be in the front.

LayerSort: Sort after the layer field (thus defined by the application).

NoSort: 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:

KindCollision: 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.

ForwardCollision: Search forward in the sprite list, and report collisions with the HitTask procedure.

BackwardCollision: Search backwards in the sprite list. Essentially the same as ForwardCollision.

NoCollision: No collision detection. Use this if you don't need collision detection or if you perform it yourself.

Note that all collision detection routines depend on what sorting is performed. If the sprite list is sorted after position.v (*VPositionSort*), only sprites within *SearchWidth* pixels are checked. If it is sorted after layer (*LayerSort*), sprites with a layer value within *SearchWidth* from the sprite is checked. In other cases, all sprites are checked. You may consider using *NoCollision* and perform the detection yourself.

### **Sprite management:**

function GetFace (resNum: integer): FacePtr;

*GetFace* (formerly called *LoadIcon*) loads the 'cicn' resource with number *ResNum*, and returns a pointer to the resulting FacePtr. This pointer can be used for the *Face* field in the sprite records. This routine is generally used from the setup procedure in all sprite units.

procedure DisposeFace (theFace: FacePtr);

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

```
function NewSprite (kind, hpos, vpos: integer; callback, setup, hittask: ProcPtr): SpritePtr;
function NewSpriteafter (afterthis: SpritePtr; kind, hpos, vpos: integer; callback, setup, hittask: ProcPtr): SpritePtr;
procedure KillSprite (who: SpritePtr);
```

The two routines *NewSprite* and *NewSpriteafter* add new sprites to the list of animated sprites. Choose *NewSpriteafter* if you want it in a special place in the sprite list. *KillSprite* 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.)

### **Running the animation:**

```
procedure RunSAT(fast:Boolean);
```

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

### **Drawing:**

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

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

*SATPlotFace* draws the icon stored in a *Face* structure in the GrafPort *theGrafPtr*. The GrafPort in question must be at least as big as the drawing area, and must be a CGrafPort if the Mac is color capable.

The normal use for *SATPlotFace* is to draw Faces on backScreen, in order to modify the background. If you pass **nil** for *theGrafPtr*, *SATPlotFace* assumes that you want the drawing in backScreen, plus calls SATBackChanged for you. (See below.)

*theGDevice* is the device returned by *SATMakeOffScreen*, or in the case of the standard buffers, *offScreenGD* or *backScreenGD*.

*SATPlotFaceToScreen* is a variant for drawing to the screen.

```
procedure SATCopyBits (src, dest: GrafPtr; destGD: GDHandle; srcPt, destPt: Point; width, height: integer; fast: Boolean);
procedure SATCopyBitsToScreen (src: GrafPtr; srcPt, destPt: Point; width, height: integer; fast: Boolean);
```

*SATCopyBits* and *SATCopyBitsToScreen* are two replacements for *CopyBits*, possibly faster than the normal *CopyBits*. With *fast* set to *false*, the normal *CopyBits* is used.

With *fast* set to *true*, the area being copied is expanded to full bytes, which means that, in b/w, if the chosen area does not have horizontal boundaries divisible by 8, a few extra pixels can be included per row. In 4-bit, one extra pixel is copied per row when boundaries are on odd coordinates. (This is usually no problem.)

[WARNING: Currently, you are responsible for making sure the copied area fits within the destination!]

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

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

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. However, simple SetPort calls will work if you are not in a hurry.

### **Maintainance:**

function SATDepthChangeTest: boolean;

SATDepthChangeTest should be called either repeatedly or before each game starts. It checks if the screen depth has changed, and if it has, it re-initialize the offscreen buffers and the face list. This should only happen after a pass through an ordinary event loop. If SATDepthChangeTest returns true, the depth has changed. In such a case, the game window needs to be updated (e.g. with PeekOffScreen) 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 PeekOffScreen;

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

procedure SATDrawPICTs (pictID, bw pictID: integer);

*SATDrawPICTs* draws the PICT with ID pictID or bw pictID in the background, just like InitSAT 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 ShowMBar;  
procedure HideMBar (wind: WindowPtr);

HideMBar hides the menu bar, and ShowMBar 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.



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

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

### **Special functions, advanced calls:**

procedure SATSetSpriteRecSize (theSize: longint);

(Advanced initialization.) *SATSetSpriteRecSize* is a special function for prorammmers 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 InitSAT/CustomInitSAT (but before any sprites are allocated).

procedure SATInstallSynch (theSynchProc: ProcPtr);

(Advanced initialization.) *SATInstallSynch* installs a procedure theSynchProc, which should take no parameters. That procedure is called once per frame, immediately before any drawing takes place on the screen. This function is intended for synchronizing the animation to the screen, which may be needed in some programs. Many games have no need for this. 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.)

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.

procedure SATInstallEmergency (theEmergencyProc: ProcPtr);

(Advanced initialization.) *SATInstallEmergency* installs a procedure theEmergencyProc, to be called when a fatal error occurs, before SAT exits. 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.

procedure SkipSAT;

*SkipSAT* does the same things as RunSAT *except drawing*. Collision detection, sound playing and sprite handling routines are performed. It should typically be used instead of RunSAT in order to get reasonably high speed on Macs that are too slow to keep up with the speed you want when calling RunSAT for all frames. You should avoid skipping more than one frame at a time, since the animation will get jerky.

Most applications will run better without SkipSAT, even if they run slightly slower than intended on slow Macs. Consider SkipSAT if you use so many sprites that the program gets unreasonably slow on the slowest Macs it may be used on (typically MacPlus).

procedure KillSAT;

*KillSAT* disposes of all internal structures that you can't dispose in other ways (that is, not sounds, sprites or faces), so you may re-initialize SAT to another state (e.g. another screen).

```

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

```

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

```

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

```

*CreateOffScreen* and *DisposeOffScreen* are taken directly from Apples technote *Principia Off-Screen Graphics Environments*. They will only work on a color Mac, as opposed to the above routines. Use them if you have special needs, like offscreen buffers with another color table.

### **Sound routines:**

The sound routines produces sound in one channel with priority handling, and managing the bugs in Apple's Sound Manager as well as possible. (Support for more channels may be added in the future.) 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; { Was InitSATSnd }

```

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

```

procedure SATSoundOn;
procedure SATSoundOff;

```

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

```

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

```

Play a sound. The handle should have been created by calling MakeSoundHandle (see below). The priority should be 0-9 for less important sounds and >10 for the more important sounds (like extra life sound, dying sound...) *CanWait* tells whether the sound should be queued until the channel is free, in case a sound with higher priority is playing, or if it should be discarded in such a case.

```

procedure SATSoundEvents;

```

*SATSoundEvents* is usually not needed in your programs, since it is called from RunSAT. If you use the sound routines when no animation is running (that is, when you don't call RunSAT repeatedly) you need to call SATSoundEvents a few times per second or so.

```
procedure SATSoundShutup; { Was ShutUp }
```

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.

### **Pattern utility routines:**

The following routines are added in order to simplify pattern handling. They allow you to define a 'PAT' resource and a 'ppat' resource with the same ID, and the appropriate one will be picked automatically. You only need the 'ppat' resource, even for Macs without Color QD.

The point with these utility routines is that they provide a "glue" to make your program work on b/w Macs as well as color ones, and to use the b/w patterns built into 'ppat' resources without any extra checks for screen depth.

Example: In order to fill the background with a pattern, get the pattern with SATGetPat, set the pen to it with SATPenPat, and fill with PaintRect.

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

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

```
function SATGetPat (patID: integer): SATPatHandle;
```

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

```
procedure SATDisposePat (SATpat: SATPatHandle);
```

*SATDisposePat* releases the pattern resource and disposes of the record.

### **Utility routines:**

These should be rather self-explanatory.

```
procedure DrawInt (i: integer);  
procedure DrawLong (l: longint);  
function Rand (n: integer): integer;  
function Rand10: integer;  
function Rand100: integer;  
procedure ReportStr (str: str255);
```

```
function QuestionStr (str: str255): boolean;  
procedure SetMouse (where: point);
```

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

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

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

## Final words

The present SAT is still a beta version. I still want comments, ideas for improvements.

- Is the manual informative? Does it help you in writing new programs? Any grammatical errors? (After all, english isn't my native language.)
- Is the interface to SAT good? What can be improved?
- Any missing features? Ideas for improvements? Limitations that should be fixed?
- Is the SAT code good? Ugly tricks in the code that should be corrected? Bugs? Incompatibilities?
- 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 make changes in the backdrop (by drawing to backScreen and CopyBits'ing it over to offscreen and SATwind)? [And, nowadays, by using SATPlotFace and SATBackChanged.]
  - 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:

function InitSAT (pictID, bw pictID, Xsize, Ysize: integer; PICTfit: boolean): WindowPtr;

#### Customized initialization:

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

#### Sprite and face routines:

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

#### Running the animation:

procedure RunSAT(fast:Boolean);

#### Drawing:

procedure SATPlotFace (theFace: FacePtr; theGrafPtr: GrafPtr; where: Point; fast: boolean);  
procedure SATPlotFaceToScreen (theFace: FacePtr; where: Point; fast: boolean);  
procedure SATCopyBits (src, dest: GrafPtr; srcPt, destPt: Point; width, height: integer; fast: Boolean);  
procedure SATCopyBitsToScreen (src: GrafPtr; srcPt, destPt: Point; width, height: integer; fast: Boolean);  
procedure SATBackChanged (r: Rect);

#### Maintainance:

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

#### Menu bar:

procedure ShowMBar;  
procedure HideMBar(wind: WindowPtr);

#### Advanced calls:

procedure SATInstallSynch (theSynchProc: ProcPtr);  
procedure SATInstallEmergency (theEmergencyProc: ProcPtr);  
procedure SATSetSpriteRecSize(theSize: longint);  
procedure SkipSAT;  
procedure KillSAT;

#### Sound:

procedure SATSoundInit; {Usually not used by applications.}  
procedure SATSoundOn;

```
procedure SATSoundOff;  
procedure SATSoundPlay (theSound: handle; priority: integer; canWait: boolean);  
procedure SATSoundEvents; {Usually not used by applications.}  
procedure SATSoundShutUp;  
function SATGetSound (SndId: integer): handle;  
function SATGetNamedSound (name: Str255): handle;  
procedure SATDisposeSound(theSnd: handle);
```

Pattern utilities:

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

Misc:

```
procedure DrawInt (i: integer);  
procedure DrawLong (l: longint);  
function Rand (n: integer): integer;  
function Rand10: integer;  
function Rand100: integer;  
procedure ReportStr (str: str255);  
function QuestionStr (str: str255): boolean;  
procedure SetMouse (where: Point);
```