SAT Tutorial

Introductory course to Sprite Animation Toolkit, by Ingemar Ragnemalm 1994.

DRAFT! If I get no comments/corrections, I will assume it is perfect.

The solutions are only in Pascal today. If you make the tutorial in C, perhaps we should add your code as C solutions?

***

Introduction

This document introduces SAT at a very basic level. If you find it too hard to learn SAT from its demos, you can start here to get a short guided tour through the basics. The tutorial is divided into a few sections where some essential functions are explained, with a programming assignment after each.

This tutorial will not explain each and every call in SAT. The goal with the tutorial is that you should understand the most fundamental

goals, so you can explore the more advanced features with confidence, building your own SAT-using application in the process. Only some calls are defined here. See the manual for the ones used but not described.

There are solutions for a number of assignments, and a resource file that all solutions share. You can use that resource file when making the assignments if you like, but you should get used to making graphics and putting together resource files as soon as possible.

1. Initialization

SAT must be initialized, or it will not know where to do its drawing. There are two calls to do this: SATInit and SATCustomInit. SATInit sets up with a set of defaults that will work with many games. If you don't like it, consider SATCustomInit.

Definition:

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

pictID is the resource number of a PICT that should be used as backdrop when running in color.
bwpictID is the resource number of a backdrop PICT for b/w.
Xsize is the desired height of the drawing area.
Ysize is the desired width of the drawing area.

Note: Unless you use Think Pascal, you must have initialized the Mac toolbox before initializing SAT. There is a call in

SAT named SATInitToolbox for doing this in one line.

Assignment 1:

Create a new project. Add the SAT library and (for Pascal) the interface file SAT.p. Write a program file that initializes SAT with SATInit, then waits for a mouse click, and quits. Add that file to the project. Use a resource file with PICT resources for the backdrop.

Possible problems:

- Compile errors. Check the parameters used. Have you done "uses SAT"/"#include <SAT.h>?
- Link errors. Have you included the SAT library in the project file? Pascal: Have you included SAT.p?
- C: Can't find SAT.h. Make sure SAT.h is in the compiler's search path. For Think C, I recommend that you put both library and header file in a subfolder to where Think C itself is located.
- Out of memory. Check the memory assignment.
- The window is all black. Are the PICT numbers correct? Are the numbers for the drawing area correct? Is the PICT made with a big offset? Try moving it to the topleft corner in your drawing program. (This is only a problem with older versions of SAT.)
- Think C, CodeWarrior Pascal/C: The Toolbox initialization is not complete or not correct. Check out the init in various demo sources, e.g. SATminimal.

When it works:

The resulting program should fire up a window and fill it with your picture. If the area you have asked for is smaller than the entire screen, it will have a black border around it. It then waits until you click the mouse button.

Solution:

```
program Assignment1;
uses SAT;
begin
SATInit(128, 129, 478, 302);
while not Button do
 ;
end.
```

2. Making a sprite

For making a sprite, we must do the following things:
• Load the face(s) (icons, the apparance of the sprite) with SATGetFace.
• Write a routine that describes the behaviour of the sprite.
• Write a routine that sets up the sprite.
• Create the sprite with SATNewSprite.
• Call SATRun repeatedly to run the animation.

A sprite pointer points to a sprite record, which holds a large number of fields. For the moment, let's only bother with two of them: face and task. face points to the face that should be used for drawing the sprite.

Definitions:

```
function SATGetFace (resNum: integer): FacePtr;
```

SATGetFace loads a "cicn" resource to a face structure, to be used as appearance for a sprite. This should usually be done during initializations.

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

Use SATNewSprite to create a new sprite. The parameters hpos and vpos specify where it should appear. The setup parameter points to a setup procedure where you should do additional initialization. There, you should always assign the task field of the sprite, usually the face field. When collision detection is used, the hotRect field must also be assigned something appropriate. (We'll return to that later.) The kind parameter is for your own use. It is put in the kind field of the sprite.

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

SATRun runs one frame of animation, calls all sprite tasks and performs collision detection. All SAT-using programs call this (or its cousin SATRun2) repeatedly during animation.

Assignment 2:

To the previous program, add two procedures SetupSprite and HandleSprite, and create a non-moving sprite with one face. Since the sprite doesn't move or change face, HandleSprite can be an empty proceure. Call RunSAT repeatedly until the mouse button is pressed.

Possible problems:

- No sprite appears. Did you assign the face of the sprite? Was the face loaded successfully? Was the task of the sprite assigned to @HandleSprite? Is the position of the sprite within the bounds of the drawing area?
- Crash or bus error. Was some pointer, for example the face or the task of the sprite, assigned to an invalid pointer?

When it works:

You should get the same result as before, but with the sprite in the position you asked for in the NewSprite call.
Note: Using a sprite to draw a stationary object is overkill, except in the case when sprites should be able to pass behind it. However, we will soon make it move.

Solution:

```
program Assignment2;
uses SAT;

procedure HandleSprite(me: SpritePtr);
begin
end; {HandleSprite}

procedure SetupSprite(me: SpritePtr);
begin
me^.task := @HandleSprite;
me^.face := SATGetFace(128);
end; {SetupSprite}

begin
SATInit(128, 129, 478, 302);
ignore := SATNewSprite(0,200,200,@SetupSprite);
while not Button do SATRun(true);
end.
```

3. Sprite handling

Now we have a sprite, doing nothing. Let's get some action. To move a sprite, you change the position field in the sprite record. That is a point, so it has two components, h and v. The sprite is called for each frame of animation. At that time, you can change the position, change the face, etc.

Assignment 3:

Add code in the HandleSprite routine that moves the sprite horizontally, back and forth between the coordinates 0 and 200. Use a global variable to tell whether it should move to the right or left in any given moment.

Possible problems:

- The sprite disappears, or isn't visible at all. Did you check the end points of the movement correctly?
- The sprite moves ridiculously fast. Don't worry about it. We'll soon fix that.

Solution:

```
var
 direction: Integer;

procedure HandleSprite (me: SpritePtr);
begin
 me^.position.h := me^.position.h + direction;
 if me^.position.h <= 0 then
  direction := 1;
 if me^.position.h >= 200 then
  direction := -1;
end; {HandleSprite}

procedure SetupSprite (me: SpritePtr);
begin
 me^.task := @HandleSprite;
 me^.face := SATGetFace(128);
 direction := 1;
end; {SetupSprite}
```

Note: If we want to limit movement to the bounds of the animation area, we can use the variables gSAT.offSizeH and gSAT.offSizeV.

4. Speed limit

If you have a fast Mac, the sprite in assignment 3 probably moves unreasonably fast. Regardless of whether it has the right speed on your Mac or not, you should put in a speed limit, or it will grow old quicker than necessary.

The easiest way to put in a speed limit is to rely on the TickCount routine. It returns the number of ticks (60th of a second) since the Mac started up last time.

Other was that you can find include:
• Putting in a delay loop. This is a bad option. Even if you allow a choice between a few different speeds, there will come a day, probably sooner than you think, when even the lowest speed is too fast. Avoid!
• Using Time Manager.
• Using a VBL task. This is far more troublesome than usig TickCount, but gives you the option to synch it with VBL.
• Using WaitNextEvent with a sleep time. This is unreliable.
• Setting sprites to positions and faces determined by the time and not by frame number. This is ok in many cases, and has the advantage that all objects will move in the right speed even on slow Macs, but it may also lead to unpredictable behavior in collision detection.

Assignment 4:

Modify assignment 3 so that you get the same maximum speed on any Mac. I suggest that you use TickCount.

Possible problems:

- It doesn't work. Debug it, or check the solution.

Solution:

```
const kTicksPerFrame = 2;
```

```
var t: Longint;

begin
SATInit(128, 129, 478, 302);
ignore := SATNewSprite(0,200,200,@SetupSprite);
while not Button do begin
 t := TickCount;
 SATRun(true);
 while TickCount < t + kTicksPerFrame do ;
end;
end.
```

5. Collisions

Before we can start making a game out of our program, we usually need to learn collision detection as well. SAT provides two ways to handle that: one that reports the collision in one of the variables in the sprite record (the "kind" field) and one that uses a callback procedure (the "hittask" pointer).

You may choose any that you like, the one you feel most comfortable with. If you plan to use the hittask, you probably should call SATConfigure at some time to tell SAT not to rely on the kind field. If you do not do this, then you will not be notified if two sprites with the same sign on the "kind" field collides.

You choose the collision type with SATCollision, using one of the constants kNoCollision, kKindCollision, kForwardCollision, kBackwardCollision and kForwardOneCollision.

When using collision detection, the hotRect of each sprite must be set to enclose as much of the sprite as needed. Start with the size of the face used (for example, use myFace^.iconMask.bounds), and trim it down a bit if needed.

When a collision is detected, you generally want to take some action. It may be to move a sprite (pushing the sprites apart), change a sprite to another look and/or behavior, or removing it altogether. Removing a sprite is done by setting its "task" field to nil. That is a signal to SAT to remove it as soon as it is properly erased.

[Add sound lesson here or later?]

Assignment 5:

Write a program (or modify the old one) that has two sprite units. The first sprite (player) should follow the mouse pointer, by using GetMouse. The second unit (target) is the same as in assignment 2: it moves back and forth.

One player sprite and at least one target should be created when the program starts. When a collision is detected between the player and a target, the other target should be removed, and a new one should be created in another position. Play a sound when this happens.

The cursor should be hidden before the animation starts, and showed again after it has ended.

Possible problems:

- No collisions are detected. Did you set the hotRect rectangle to something appropriate? If you use hitTasks, have you set the hitTask field of the sprites appropriately? Was SAT configured for the kind of collision detection you wanted?
- Collisions seem not to always be detected when they should. Do you use sprites bigger than 32 pixels high? If so, you might have to use SATConfigure to increate the search width.
- No sound. Is the sound loaded? Is the name/ID correct?

Solution:

```
program Assignment5;
uses
 SAT;
```

```
const
 kSpeed = 5;
var
 ignore: SpritePtr;
 direction: Integer;
 theSound: Handle;

procedure HandleSprite (me: SpritePtr);
begin
 GetMouse(me^.position);
end; {HandleSprite}

procedure SetupSprite (me: SpritePtr);
begin
 me^.task := @HandleSprite;
 me^.face := SATGetFace(128);
 SetRect(me^.hotRect, 0, 0, 32, 32);
end; {SetupSprite}

procedure SetupTarget (me: SpritePtr);
forward;

procedure HandleTarget (me: SpritePtr);
begin
 me^.position.h := me^.position.h + direction;
 if me^.position.h <= 0 then
  direction := kSpeed;
 if me^.position.h >= 200 then
  direction := -kSpeed;

 if me^.kind <> -1 then {Hit me!}
  begin
   me^.task := nil;
   ignore := SATNewSprite(-1, 0, Rand(gSAT.offSizeV), @SetupTarget);
{We could also re-use the old sprite for a new one, if we like.}
   SATSoundPlay(theSound, 1, true);
  end;
end; {HandleTarget}

procedure SetupTarget (me: SpritePtr);
begin
 me^.task := @HandleTarget;
 me^.face := SATGetFace(129);
 SetRect(me^.hotRect, 0, 0, 32, 32);
 direction := kSpeed;
end; {SetupTarget}

const
 kTicksPerFrame = 2;
var
 t: Longint;

begin
SATInit(128, 129, 478, 302);
ignore := SATNewSprite(1, 200, 200, @SetupSprite);
ignore := SATNewSprite(-1, 0, Rand(gSAT.offSizeV), @SetupTarget);
theSound := SATGetNamedSound('TestSound');
HideCursor;
while not Button do
 begin
  t := TickCount;
  SATRun(true);
  while TickCount < t + kTicksPerFrame do
   ;
 end;
ShowCursor;
end.
```

Note: There is another version that use callback procedures, the hitTask field, instead.

6. Making a game of it

Let's go a little closer to a real game. Now, we will need to know a little more about what happens behind the screen. SAT has a large set of global varables, of which we only need to know a few. They are kept in the gSAT record.

• gSAT.offSizeH and gSAT.offSizeV tells the size of the animation area.
• gSAT.offScreen is a copy of the screen. You should rarely need to touch it.
• gSAT.backScreen holds the background image. You may want to draw to it yourself, using QuickDraw calls. See below.
• gSAT.initDepth tells the screen depth. 1=B/W, 4=16 colors/grays, 8=256 colors/grays etc. You should use it for deciding whether to draw color or b/w patterns.
• gSAT.colorFlag tells whether Color QuickDraw is available or not. You should use it for deciding whether to use NewWindow or NewCWindow etc.

When drawing in backScreen, you will often have use for the routine SATBackChanged:

Definition:

```
procedure SATBackChanged (r: Rect);
```

SATBackChanged should be called to notify SAT that you have made changes in backScreen that you want to be displayed on screen. When you have done some drawing, call SATBackChanged, passing it a rectangle that encloses the changed area.
To draw in gSAT.backScreen, you must SetPort to it, or better, call SATSetPortBackScreen. Beware, however, that you should always restore the port and device after changing them (and SATSetPortBackScreen changes both). SATGetPort and SATSetPort are useful for that. You typically do:

```
var
 savePort: SATPort;

SATGetPort(savePort);
SATSetPortBackScreen;
{Your drawing here…}
SATSetPort(savePort);
```

Assignment 6:

Modify assignment 5 to be closer to a full game:
• Add 1 to a score variable each time an object is caught.
• Display the score on the screen. Draw with MoveTo, EraseRect and DrawString, and notify SAT about it with SATBackChanged.
• Time limit?
• Control the player sprite another way, e.g. keyboard control or velocity instead of position.
• Keep the player sprite visible all the time.

Possible problems:

Lots of them. Sorry, but it is no longer easy to pinpoint the most likely errors you can make. Fire up your debugger and start stepping through the code.

Solution:

(Two selected routines with most of the added stuff. Based on the hitTask-using solution to assignment 5.)

```
procedure HandleSprite (me: SpritePtr);
 var
  event: EventRecord;
begin
{Now hold on to the hat! I'm using GetOSEvent to get key down events.}
{the keydowns then affect the speed variable in the sprite. That speed}
{is added to the position. Finally, I check the position against the screen}
{borders, and modify the speed in case I reach a border.}

 if GetOSEvent(keyDownMask, event) then
  if event.what = keyDown then
   case char(BitAnd(event.message, charCodeMask)) of
    'a':
     me^.speed.h := me^.speed.h - 1;
    's':
     me^.speed.h := me^.speed.h + 1;
    'w':
     me^.speed.v := me^.speed.v - 1;
    'z':
     me^.speed.v := me^.speed.v + 1;
   end;

 me^.position.h := me^.position.h + me^.speed.h;
 me^.position.v := me^.position.v + me^.speed.v;
 if me^.position.h < 0 then
  begin
   me^.speed.h := abs(me^.speed.h);
   me^.position.h := 0;
  end;
 if me^.position.h > gSAT.offSizeH - 32 then
  begin
   me^.speed.h := -abs(me^.speed.h);
   me^.position.h := gSAT.offSizeH - 32;
  end;
 if me^.position.v < 0 then
  begin
   me^.speed.v := abs(me^.speed.v);
   me^.position.v := 0;
  end;
 if me^.position.v > gSAT.offSizeV - 32 then
  begin
   me^.speed.v := -abs(me^.speed.v);
   me^.position.v := gSAT.offSizeV - 32;
  end;
end; {HandleSprite}

procedure HitTarget (me, him: SpritePtr);
 var
  savePort: GrafPtr;
  r: Rect;
begin
 if him^.task = @HandleSprite then {Chack what we hit!}
  begin
   me^.task := nil;
   ignore := SATNewSprite(-1, 0, Rand(gSAT.offSizeV - 32), @SetupTarget);
{We could also re-use the old sprite for a new one, if we like.}
   SATSoundPlay(theSound, 1, true);

{Add to the score}
   score := score + 1;

{Draw the score on the screen.}
   SATGetPort(savePort); {Save port and device!}
   SATSetPortBackScreen;
   SetRect(r, 100, 0, 100 + stringWidth(stringof('Score: ', score)), 15);
   EraseRect(r);
```

```
    MoveTo(r.left, r.bottom - 3);
    DrawString(stringof('Score: ', score));
    SATBackChanged(r);
    SATSetPort(savePort); {Always restore!}
   end;
  end; {HitTarget}
```

7. And then…

The tutorial really ends here, but I'll end with suggesting a few more assignments to work on while you decide on what your first game should be like. Look in the manual and the demos for help and related code.

Extra assignment 1:

Take apart your solution of tutorial 6(?) so you get one file for each sprite. You have to export the routines that have to be accessible from outside the file.

Extra assignment 2:

Make a system with levels for your program, so a different set of sprites is started for each level.

Extra assignment 3:

Put in an event loop in your program, so you can start the game by a menu selection, and switch it out. Make it possible to pause. Make a menu selection for turning on or off sound.

Extra assignment 4:

Make a high score list. Make routines for updating it and displaying it. Save it in a resource, perhaps in a separate file?

If someone is really ambitious, perhaps we can provide solutions to those as well. (It isn't too hard - it's all in my demos.)

***

Don't forget to add the resources in SATblitters.rsrc to your resource file if you need top speed! That can give a considerable speedup, especially on low-end Macs.