Paper 012-30

SAS/AF® Considerations

Richard A. DeVenezia, Independent Consultant

ABSTRACT

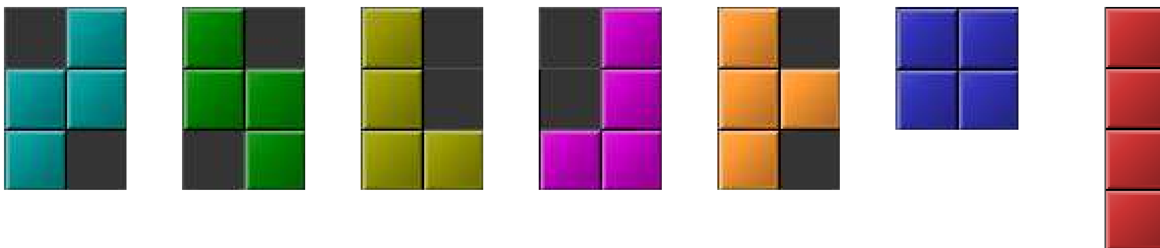
Game development has a long and proud history. You may have found the SAS menu Solutions/ Accessories/Games an interesting diversion. Can you guess who wrote "Dr. G's Blackjack"? I thought it was time to breathe some new life into this venerable category. This paper discusses design and implementation considerations encountered while building the popular game TETRIS in SAS/AF.

INTRODUCTION

TETRIS really needs no introduction. It is one of the most popular of all puzzle style video games and has many active fans. Have you considered that making your own version of the game in SAS/AF is almost as much fun as playing it? There are several questions that need to be answered. What are the pieces? How are they organized? How do they move? Can I make sounds? How do I keep score?

GAME PIECES

There are seven distinct shapes that can be made from four connected blocks. These are the game pieces.



GAME PLAY

The pieces fall one at a time from the top. The player moves and rotates the piece until it hits bottom or another piece. When a row of blocks is made, three things happen. The score is updated, the row is removed and the blocks above the row drop down. The game field is 10 blocks across and 20 blocks high.

MOVEMENT

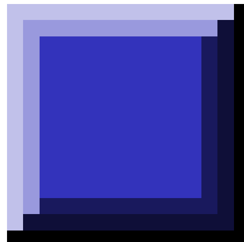
Pieces can be moved left and right, or dropped straight down. When a piece is dropped the player has a small amount of time during which he can tweak it left or right (if there is space). The keys for moving are based on a T pattern in the keypad keys.

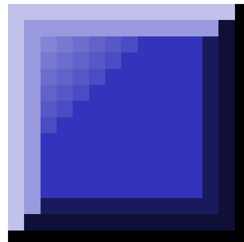
- 4 – move left
- 5 – rotate clockwise
- 6 – move right
- 2 – drop
- D also drops, Q is for quitters.

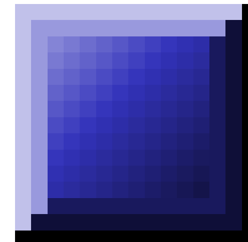
GAME PIECES - BLOCKS

The pieces look cool because the blocks they are made from have a border and a gradient fill. The upper left part of the border is a lighter shade of the block color, and the lower right part is a darker shade. The gradient is made by drawing 45 degree lines across the face. The color of the lines vary linearly from a light to dark shade of the block color.

The fill progression becomes clear when using exaggerated shading factors and zooming in on a face having a small number of pixels.


Blank face


Partially filled


Gradient filled

The program `CreateBlocks.source` uses DSGI to create seven gradient-filled block images, saved to files named A.GIF to G.GIF. Each image has a color corresponding to a game piece. The images are 30 by 30 pixels and are the basis of each piece.

The program also creates a grid image for the game field. The image is 301 pixels wide and 601 pixels high. There is an extra pixel so that when the blocks of a piece are positioned, the left and top edges are adjacent to a grid line.

Another program LoadBocks.scl reads the gif files and saves them as IMAGE catalog entries. The Image Data Model class is used to perform this task. Methods _readFilePath and _writeCatalog do all the necessary conversions.

```
init:
  in_path = pathname ('WORK');
  sn = screenname();
  out_cat = scan(sn,1,'.')
  || '.' || scan(sn,2,'.');

  declare sashelp.fsp.imgdat img
  = _new_ sashelp.fsp.imgdat ();

  do i = 0 to 6;
   name = byte(rank('a')+i)||'.gif';
   link import;
  end;

  name = 'grid.gif'; link import;

  img._term();
return;
```

```
import:
  file = in_path || '\' || name;
  ent  = out_cat || '.' || name;

  img._readFilepath (file);
  img._writeCatalog (ent);

  declare list attr = {};

  img._getAttributes (attr);

  w = getNitemN (attr,'width');
  h = getNitemN (attr,'height');

  put name= w= h=;

  attr = dellist (attr);
return;
```

GAME LAYOUT

A SAS/AF Frame is constructed to contain the objects that comprise the game: a game field grid, controls for reporting the score, the level and the number of blocks remaining, a “game over” notice, text labels for simple instructions and an all important Text Entry Control for capturing keypresses. The game pieces are dynamically instantiated and manipulated at runtime.

TIMERED SIMULATION

Hold it, timeout, wait a second! On first glance, SAS/AF does not appear to have a Timer class, nor any functions to perform time sliced or interrupt driven programming. How will time driven events occur?

One might be tempted to consider the crudest form of waiting, the spin loop

```
later = datetime() + 20/1000; * wait 20 milliseconds;
do until (datetime() > later);
end;
```

Spin loops are severe no-no's. They peg CPU usage at 100%, and are unyielding, which means the program cannot process a click or keypress while the loop is running. However, there are two very important functions upon which the entire game is based.

SLEEP (N, 0.001)

“The CALL SLEEP routine suspends the execution of a program that invokes this call routine for a period of time that you specify”[i] Sleep is an operating system friendly function. While the SAS session is suspended it yields to the operating system; no CPU usage will occur, nor will any response to mouse clicks and keystrokes.

EVENT ()

“Reports whether a pending event has occurred...An event can be a mouse button press or a keyboard key press”[ii] Event returns 1 when an event needs to be processed by the SAS session manager. In a code block that is looping, 1 is a signal that the loop should end and let the AF event handling system proceed.

```
Animate: method return=num;
  * return -1 if game over, 1 if piece stopped, 0 if event occurred;
  declare num dt=datetime() oldy ;

  do until ( event() );  * wait for user action;

    do while (dt<timeout and not event()); * wait for timered action;
      sleep (10,0.001);  * time-slice;
      dt = datetime();
    end;

    * -1 indicated timered action resulted in game over state?
    if field.gameover then return -1;

    if (dt >= timeout) then do;
      * timeout occurred, drop the piece a little;
      oldy = y; down (gravity);
```

```
        declare Wave click = _self_.field.clicks[row]; click.play(); * bleep;

        if oldy = y then return 1;  * piece did not go down since last timeout;

        timeout = computeNextTimeOut (dt, field.level);
      end;
    end;

    return 0;
  endmethod;
```

Sleep and Event are used in combination to make a piece drop and playing a sound while waiting for a keypress. This is not interrupt driven programming, but rather a form of time sliced animiation. 10 milliseconds (ms) was chosen as the largest acceptable duration in which the SAS session would be totally unresponsive. Every 10ms a check is made for a pending event and if so, the Animate routine exits and lets AF event handling proceed normally. However, if after 10ms no event has occurred, the loop sleeps again... until the current time is at or beyond the cutoff timeout. When a timeout occurs, then the animation actions happen; the piece is moved down, a sound is played and the next timeout is computed.

Note: `Animate` is a method for a piece. If a piece does not go down when so instructed, it means the piece has hit bottom or is on top of another piece; a new piece should start at the top.

WAIT

So what is calling `Animate`? This is the trickiest part of the application. The Text Entry Control on the frame is named `kbd`. Kbd is customized during frame `INIT` as follows:

```
sn = screenname();
cat = scan(sn,1,'.') || '.' || scan(sn,2,'.') || '.';
kbd.keyFeedback = 'Yes';
kbd._setInstanceMethod ('_onKey', cat||'Playmethods', 'onkey');
```

The SCL method `onkey` in the `Playmethods` catalog entry will run whenever a key is pressed while kbd has focus. The method has a section to dispatch behavior actions in accordance to the rules of play.

```
onkey:
method; * kbd keypress handler;
  ...
  select (upcase(_self_.text));  * handle action request;
    when ('4') piece.left();
    when ('5') piece.rotate();
    when ('6') piece.right();
    when ('2') piece.drop();
    when ('D') piece.drop();
    when ('Q') field.gameover = 1; otherwise ;
  end;

  * reset text so after next keypress text will have only one char;
  _self_.text = '';
```

```
wait: * top of wait loop;
  rc = piece.animate();
  if rc then do;
    if rc > 0 then piece.lock();
    if field.gameover then do; ... return; end;

    piece._term();

    a2g = byte(41x+ranuni(0)*7);
    piece = _new_ Piece(field,a2g);
    ...
    goto wait;
  end;
endmethod;
```

A wait loop is entered after responding to a keypress with an appropriate action. The statement `rc=piece.animate()` causes the application to enter the timered simulation state. The piece will drop a little and bleep until animate returns:

- -1 when the game is over,
- 1 when a new piece is needed, or
- 0 when an event is pending.

If rc is 0, the wait loop ends, and the onKey method returns. Now, here's the trick. The only event pending that impacts game play is a keypress in the **kbd** Text Entry, which causes onKey to reenter (via the SAS/AF executor event handling system).

WAVE TO THE SPEAKER

No, no, don't wave to me. Send the bit sample of a wave form to the platform's audio output system. This is only valid on Windows platforms. A moduleN function is used to invoke the Windows API routine **PlaySound**

WAVE CLASS

The Wave class accepts three parameters for computing the wave form data of a sound effect: duration, frequency and volume. A character variable is used as a data buffer. The wave data conforms to the RIFF media format and is computed at instantiation

```
  wave:
  public method dur:num freq:num vol:num;
    ...
    buffer = ... * 44 byte RIFF content header * ...

    theta = 0;
    ix = 45;

    do i = 1 to samplecount;
      level = floor (maxlevel * sin(theta));
      theta + thetaStep;
```

```
      thetaStep + 0.0003;

      substr (buffer,ix,2) = put(level,ib2.);

      ix + 2;
    end;
  endmethod;
```

The small rise in theta step gives the sound a little texture. Play() uses the Windows API function **PlaySound** to output the sound. Play() is called from inside the piece Wait() loop.

```
Play:method;
rc = modulen ('PlaySoundA', buffer, 0, &SND_ASYNC + &SND_MEMORY + &SND_NOSTOP);
```

The SASCBTBL fileref which describes how SAS interfaces with DLL routines, is prepared in the frame INIT.

PIECE CLASS

The Piece Class is another core of the game; the piece arranges blocks into the appropriate shape and implements behaviors such as down, left, right, rotate, drop, animate and lock. Additionally, collision detection is performed in the class whenever a motion-related behavior is attempted.

BLOCK ARRANGEMENT

Arrays are used to specify the block layout of each piece in each of its rotations. The piece class adjusts itself to be any of the seven shapes. Note: The intialValue of piece A's layout shown here has been pivoted to reduce space.

```
private num A [ 4, 4, 4 ] / ( initialValue =
  { ., ., 1, .     , ., ., ., .     , ., ., 4, .     , ., ., ., .
  , ., ., 2, .     , ., ., ., .     , ., ., 3, .     , ., ., ., .
  , ., ., 3, .     , 4, 3, 2, 1     , ., ., 2, .     , 1, 2, 3, 4
  , ., ., 4, .     , ., ., ., .     , ., ., 1, .     , ., ., ., .  };
```

Dimension one is for the four states of rotation, dimension two and three define the two dimensional layout of the blocks in the piece.

block = Layout [state, row, col]

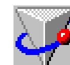
When a piece is instantiated, an inverse mapping is determined to simplify location lookup.

Location [state, block, 1] = row
Location [state, block, 2] = col

Four Image Viewer Controls, *block1 - block4,* are created and populated with the correct colored block image, and positioned according to the location array. The visual effect is a piece.

SURPRISE BLOCKS

Games are more exciting when they have secrets and surprises. The SAS icon image is randomly used as a block's image. A scoring bonus occurs when the player eliminates this block.



POSITIONING

When the X pixel position of a piece is changed, new horizontal positions of the Image Viewers are computed in the attributes `setcam` method. Note the use of the `location[]` array.

```
/*
 * The pieces horizontal position is to be set
 * errorMessage is an attribute of all Objects, and will be
 * non-blank if an attempt is made to place and imageviewer (block)
 * outside allowed bounds.
 */

private num incamX / ( initialValue = 0 );  * semaphore to prevent reentry;
setcamX:
protected method x:num return=num;
  if errorMessage ne '' then return 1;
  if incamX then return 0;
  incamX = 1;

  declare num blockno;
  do blockno = 1 to 4;
    block[blockno].horizontalPosition = x + location[state,blockno,2] * side;
  end;

  col = floor ( ( x-1 - field.horizontalPosition ) / side ) + 1;

  incamX = 0;
  return 0;
endmethod;
```

A similar method is coded for the Y pixel position. A piece can also be positioned according to row and column. This coordinate system is based on block size.

```
x = field.horizontalPosition + (col-1) * side + 1;
```

COLLISION DETECTION

The collision detection logic is as follows; if, after the piece is moved, would any of the corners of any of the blocks of the piece reside in a cell that is occupied by a block previously locked? Note the use of field.blocks, an array that tracks which blocks have been previously locked.

```
down:
method dy:input:num;
  declare num i trow brow lcol rcol bblocked blocked ;

  do i = 1 to 4;
    trow = blockRowOf ( block[i].verticalPosition + dy );
    brow = blockRowOf ( block[i].verticalPosition + dy + side-1 );
    lcol = blockColOf ( block[i].horizontalPosition );
```

```
    rcol = blockColOf ( block[i].horizontalPosition + side-1 );

    bblocked = (brow > field.getNumberOfRows());

    if not bblocked then do;
      blocked = 0;
      if brow > 0 then do;
        blocked = blocked or _self_.field.blocks[brow,lcol];
        blocked = blocked or _self_.field.blocks[brow,rcol];
      end;
      if trow > 0 then do;
        blocked = blocked or _self_.field.blocks[trow,lcol];
        blocked = blocked or _self_.field.blocks[trow,rcol];
      end;
    end;
    else
      blocked = 1;

    if blocked then do;
      * requested dy is too much, what is minimum that would work?;
      dy = round(topOfRow (brow-1) - block[i].verticalPosition,1);
      if dy < 0 then do;
        field.gameover = 1;
        return;
      end;
    end;
  end;

  y + dy;
endmethod;
```

Similar methods are coded for left, right and rotate.

LOCKING BLOCKS

When a piece cannot move, the blocks (the Image Viewers) that compose the piece are locked into the `field` for scoring considerations and future collision detection.

```
lock:
public method;
  declare num i r c rr cc rowsCleared;
  declare Object b;

  do i = 1 to 4; * determine if game over should occur;
    r = row + location[state,i,1] ;
    c = col + location[state,i,2] ;
    * can not lock a block above the top game field row, game over;
    if r < 1 then do; field.gameover = 1; return; end;
    * can not lock a block where one already locked, game over;
```

```
    if _self_.field.blocks[r,c] then do; field.gameover = 1; return; end;
  end;

  do i = 1 to 4; * lock the blocks of the piece;
    r = row + location[state,i,1] ;
    c = col + location[state,i,2] ;
    _self_.field.blocks[r,c] = block[i];  * save Image Viewer reference;
  end;
```

CLEARING ROWS

After the blocks are locked, each row of the game field needs to be examined. If a row has a block in each column, then the row has to be eliminated.

```
  declare num nr = field.getNumberOfRows();
  declare num nc = field.getNumberOfColumns ();
  declare num advancer = 0;

  rowsCleared = 0; * for scoring;

  do r = nr to 1 by -1;
    do c = 1 to nc ; if not _self_.field.blocks[r,c] then leave; end;
    if c <= nc then leave;  * row is not filled;

    * row is filled, hide the blocks of the filled row, count bonus blocks;
    do c = 1 to nc;
      _self_.field.blocks[r,c].visible = 'No';
      * getting a bonus block makes the player score as if
      * on a  higher level;
      if index (_self_.field.blocks[r,c].image, 'SASHELP') then advancer+1;
    end; refresh; sleep (175,0.001);

    * free the Image Viewers and clear the references to them;
    do c = 1 to nc;
      b = _self_.field.blocks[r,c]; b._term(); _self_.field.blocks[r,c] = 0;
    end;

    * move the blocks above the row down one row;
    do rr = r-1 to 1 by -1;  do cc = 1 to nc;
      if _self_.field.blocks[rr,cc] then
      _self_.field.blocks[rr,cc].verticalPosition + side;
      b = _self_.field.blocks[rr,cc] ;
      _self_.field.blocks[rr+1,cc] = b;
    end;  end;

    * clear the topmost row;
    do cc = 1 to nc; _self_.field.blocks[1,cc] = 0; end;

    rowsCleared + 1; r + 1;
```

```
    end;
```

SCORING

The points scored depends on how many rows were cleared by locking a piece. The more rows cleared the higher the multiplier factor.

```
* per http://folk.uio.no/perjp/java/;
declare num factor [0:4] = (0,40,100,300,1200);

field.score + factor[rowsCleared] * (field.level+1+advancer);
```

CONCLUSION

Programming a game requires a range of skills and the ability to blend a variety of objects. SAS/AF is capable of producing enjoyable game play that looks and sounds nice. Creative use of sleep and event allow an SCL programmer to implement self-running simulations.

ABOUT THE AUTHOR

Richard A. DeVenezia is an independent consultant and has worked extensively with SAS products for over ten years. His specialties include developing SAS/AF applications to support ad hoc data exploration and consolidation. He has presented at previous SUGI, NESUG and SESUG conferences. Richard has an active interest in learning and applying new technologies. He is an active contributor on SAS-L, where he looks for new techniques and offers some of his own.

SAMPLE APPLICATION

A SAS transport file containing the Tetris application, with source code, can be found at the authors website. Visit http://www.devenezia.com and follow the link to Papers.

Tetris can also be installed by visiting http://www.devenezia.com/downloads/sas/af?topic=27 The game will play with only Base SAS installed, you do not need SAS/AF installed.

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

i SAS 9.1.3 Help and Documentation – SAS Institute Inc., Cary, NC, USA
ii SAS 9.1.3 Help and Documentation – SAS Institute Inc., Cary, NC, USA