AGDS - Adventure Game Development System

OVERVIEW

AGDS was developed as a fast means to implement adventure games using a machine portable language. It outwardly resembles the programming language "C".

AGDS is a specialized set of commands which help manipulate objects on a 3-D plane. Before looking at the commands, it is important to understand some basic terminology.

ROOM -- we divide our games into "rooms"--locations based upon a map. For instance, if a game is inside a house, room 1 might be the living room, room#2 the kitchen, etc. When the adventure moves outdoors, room #11 might be the left side of a lake and room #12, the right side. The amount of landscape in a room is controlled by the graphics of the room's picture.

PICTURE -- Usually, each ROOM has an associated picture. In some cases a room has multiple pictures. In King's Quest 2, Dracula's castle had numerous rooms with two pictures; one with the room lit, and another mainly black picture (indicating the lights were out). Donald uses three pictures within the banner page room, a logo picture, a title picture and the credits screen.

OBJECT -- OBJECTs may be either ANIMATED OBJECTS or INVENTORY OBJECTS, the difference being whether thye appear graphically on screen or exist only to be carried. Example: in Quest2 there were two objects, the mallet and the bracelet. King Graham sees a tree in the woods that contains a mallet. He can say "GET MALLET" and the mallet goes into his inventory. With the mallet in inventory, he can say "SHOW MALLET" and a window opens, displaying the mallet. The bracelet is seen lying on the beach. When he says "GET BRACELET" it disappears off the screen. Once in inventory, it can be "look"ed at just like the mallet. Because the bracelet appears on screen during the game, it is an "animated object." Since the mallet exists only to be taken into inventory, it is called an "inventory object." "Animated" objects can be "DRAWN" on the screen and animated. "Inventory" OBJECTs can only be held in inventory. Both may be "SHOWn" in a pop-up screen window, once they are in inventory.

CELLS -- AGDS works like traditional cartoon animation. If Little Red Riding Hood is seen stooping to pick flowers, she must be drawn as a series of CELLS showing her going from standing up to bending over. At NORMAL speed, AGDS cells are cycled at the rate of 9 per second. If Riding Hood is 20 pixels by 20 pixels, each CELL uses about 200 bytes of memory.

Obviously, all CELLS to be displayed must be in memory when the animation begins. A collection of CELLS that form an animation sequence is called a LOOP. Each cell contains the specific bit-maps AGDS displays on-screen.

LOOPS -- Each loop contains from 1 to 10 CELLS. Little Red Riding Hood picking a flower is a LOOP, Red skipping is another LOOP, Red walking without skipping is yet another.

VIEWS -- Views consist of from 1 to 4 LOOPS. An animated object may have a view consisting of 4 LOOPS: right, left, front, and back. You can only load VIEWS, not just a LOOP. Often, memory constraints will force a character to have only 2 loops, a right and a left. AGDS determines which LOOP is displayed, based on the direction of an object. Only the four directions [r, l, f, b] are supported.

LOGICS -- Each room has LOGIC associated with it. This LOGIC is the procedural set of instructions that gets executed describing game play. AGDS always automatically loads, and executes ROOM 0. From there, the logic programmer must handle everything.

ROOM #0 contains the LOGICS that are common to the entire game, low level functions like "SAVE GAME," "HELP," etc. Instead of coding instructions to handle saving the game into every room, put it in room 0.

DYNAMIC LOGICS are used for the same reason. They are rooms that have no picture, but process commands that occur in multiple ROOMs. Every ROOM that needs swimming can load a LGC.SWIM which deals with commands like SWIM, and standing ego up when he is no longer in water. It would be a nuisance and a waste of disk space to code this redundantly for each ROOM with water in it. If it were in ROOM #0 it would occupy ram even in ROOMs without water. Memory is valuable; anything that cuts down memory requirements is good!

LAST LOGICS are simply the last LOGICS executed. Here you handle things like the player giving a command that the other logics don't recognize. For instance, if the player asks the game to swim when there is no water in a ROOM, no dynamic logic is loaded to deal with water. Somehow the player needs to be told we don't understand what he said. LAST LOGICs handles all code trying to provide intelligent responses to input not dealt with by one of the earlier-processed logics.

SOUNDS -- sound effects and music work just like PICTURES and OBJECTS, load them, then call them.

PRIORITIES

AGDS pictures are composed of a picture screen, and a priority screen. The picture is arranged as 160 by 168 pixels, numbered from 0-159, and 0-167. Each pixel is one of 16 colors. Associated with each pixel is a "priority", from 0 to 15. Priorities 4 to 15 determine the 3-D plane, 0 to 3 are "control" priorities, with special meaning.

THREE-D PRIORITIES

Priority 4 is the lowest 3-D priority; any pixel on this priority will always be in the background, and ego and all other objects appear "in front" of it. All objects appear behind priority 15 pixels, unless the object is also a priority 15.

Objects can only be one priority at a time. An object's priority is usually determined by the Y coordinate of its baseline, according to the following table:

PRIORITY TABLE

Y-coord	Priority
0-47	4
48-59	5
60-71	6
72-83	7
84-95	8
96-107	9
108-119	10
120-131	11
132-143	12
144-155	13
156-167	14

Example: if EGO is at 50/75, he is at priority 7. He will show behind pixels at priority 8 to 15 and in front of priorities 4 to 7.

If an object with priority 8 is drawn on a picture that also has priority 8, the object appears in front. If it appears in front of another object with the same priority, the object with the lower Y coordinate appears in front.

CONTROL PRIORITIES

These priorities apply only to pictures. Object may not be set to priorities

0 thru 3.

PRIORITY 0 -- called "Priority Zero." No pixel of an object's baseline may land on a picture pixel of priority 0. It is used at the base of trees, rocks, etc, to stop objects from passing thru them. However, objects set to priority 15 appear to "float" over the entire picture, and thus do ignore Priority 0.

PRIORITY 1 -- called "block." Nearly the same as priority 0, except objects can pass over it, or not. The action "ignore.blocks()" allows objects to cross priority 1. There is also an action "block(x,y,x,y)" that can be established through the logics; it works exactly the same way, but can be defined, redefined, and discarded on the fly.

PRIORITY 2 -- called "special." When any pixel of EGO's baseline coincides with a pixel of priority 2, the "hit.special" global flag is set. This priority is used when you need to know if EGO has touched a location, and a position block won't work. Used to know if ego has walked into a chasm.

PRIORITY 3 -- called "water." When all pixels of EGO's baseline are on priority 3, the "on.water" global flag is set. This priority is often used for water, so EGO can swim, or to keep a character within a boundary.

COMMENTS ON PRIORITIES:
Control pixels still have a visual priority from 4 to 15. To accomplish this, AGDS scans directly down the control priority until it finds some "non-control" priority.

Some objects step by more than 1. This means that their baseline when moving along the Y coordinate can step over priority 0. To get around this, in Quest 2 we put vertical bars of priority 0 inside trees and rocks etc.

Priority 15 objects ignore all control priorities in the picture. For instance, they pass over priority 0 as if it weren't there.

See the file AGI.DOC for a complete discussion on each individual function.

WORKING WITH LOGIC FILES

CG -- the game compiler.
This is the most used. Assuming you use BRIEF to edit the source code files, call the compile macro (often assigned to the F9 key). It saves your file, calls the CG game compiler, routes CG's output to your LGC subdirectory, and if errors are found, displays them in the current window.

In addition, 'cg' can now be run properly from BRIEF.

* It generates output in the filename format which the interpreter and utilities want. Thus:
  ```
  cg rm0
  ```
  puts its output in 'rm.0'.

* You can reroute output with the '-o' switch:
  ```
  cg rm0 -o..\lgc
  ```
  puts its output in ..\lgc\rm.0.

* It takes wildcards in filenames. You can compile the world with:
  ```
  cg *.cg
  ```

* You can pass 'cg' its parameters from a file:
  ```
  cg @file
  ```
  gets its command line from the file 'file'. This is used in the 'make' and 'makeuse' batch files.

MAKE
'make' uses the program 'cgupd' to create a list of all files for which the source code date/time is more recent than the object code date/time. It feeds this list to 'cg' for compilation. Thus, you can make modifications to lots of files, then type 'make' to recompile all of them.

MAKEUSE
'Makeuse' is similar to make, but uses 'find' to generate a list of all files containing a given string. If, for example, you change the number corresponding to a given flag, you can update everything by typing:
```
makeuse flag_name
```

MACROS
The BRIEF macros "allfiles.m" and "modify.m" allow you to modify all of your source files at once. To use them, edit the file "\brief\macro\modify.m" to do whatever you want done to your files. Compile it (use F9) until it is

correct, and automatically loaded. Then press F10 (BRIEF's "command"), and enter "allfiles."

"allfiles" asks for a "modification criterion," which will probably be "rm*.cg". It must be a regular BRIEF search string. Allfiles then calls "ls" to read the current directory, searches for files matching the "modification criterion," reads each file into BRIEF, calls the macro "modify," then saves the file. After all files are done, exit BRIEF and do a "cg" with the same "modification criterion." There should be no need to modify "allfiles"; it is just a general purpose driver that calls "modify."

For example, to modify the action names, set up "modify" with the appropriate "replace/translate" commands, then invoke "allfiles" with a criterion of {gamedefs}|{*.cg}.

THE OBJECT EDITOR

```
Lower-Case command keys:
          l ..... set screen block upper-Left corner position.
          r ..... set screen block lower-Right corner position.
          e ..... Edit cel (prompts for cel #).
          s ..... Save cel (prompts for cel #).
          g ..... Get loop file from DOS (prompts for filename).
          p ..... Put loop file to DOS (prompts for filename).
          m ..... Mirror the current cel.
          x ..... set X-step for current loop (prompts for value).
          y ..... set Y-step for current loop (prompts for value).
          t ..... set step-Time for current loop (prompts for value).
          c ..... set current Color (prompts for color code).
          n ..... display Next cel at upper-left corner of current block.
          h ..... Hack current loop's cel count (described below).

Upper-Case command keys:
                I ..... set Insert mode (described below).
                D ..... set Delete mode (described below).
                N ..... set Normal mode (described below).
                S ..... set Slow speed mode (described below).
                M ..... set Med. speed mode (described below).
                F ..... set Fast speed mode (described below).
                Z ..... set Zip. speed mode (described below).
                C ..... Clear screen & re-display colorbar, etc.
                Q ..... Quit edit session & return to DOS.
                B ..... Block fill (described below).

Cursor control keys:
                     up arrow ....... Up.
                     right arrow .... Right.
                     down arrow ..... Down.
                     left arrow ..... Left.
                     Pg Up .......... Up-Right.
                     Pg Dn .......... Down-Right.
                     End ............ Down-Left.
                     Home ........... Up-Left.
```

Insert, Delete, & Normal modes:

In Insert or Delete modes, a color dot is written at the current cursor position (current color for insert, black for delete) each time the cursor is moved, at each position the cursor crosses. The dot is written at the current cursor position BEFORE moving to the next, NOT at the new position. Normal mode does not alter the screen on cursor movement.

ins	Insert current color dot at current cursor position.
del	Delete dot at current cursor position.

Speed modes:

Each cursor movement key will move the cursor in the direction indicated for the number of positions indicated by the current speed mode selected. Slow speed cuases the cursor to move one dot for each cursor movement key. Medium, Fast and Zip move the cursor 2, 3, and 4 dots per cursor movement key, respectively.

NOTE: In Insert or Delete modes, the screen alteration functions will alter all positions passed over by the cursor in its movement, not just the destination position.

Block fill command:

Fills the area of the screen described by the last Set upper-Left, Set lower-Right commands with the current color.

Hack cel count command:

Will delete the specified cel and all higher-numbered cels from the current loop (prompts for cel number).

MAKING VIEWS

"makeview" is a utility that creates views out of loops. It reads
a text file called VIEWS in the \loops subdirectory for its instructions.
A driver batch file, "mv.bat" is used to automatically call makeview and
update the \view subdirectory.

The VIEWS file must contain an entry for each view, including some (or all)
of the following information, in the following format:

```
     view 3, skip 3: 3r 3l 3f 3b

     skip 3    means this object was drawn with a background color of 3
               (default, black 0).
     r,l,f,b   gives the number & description of the loop
```

To make, or fix, a view:

1. Edit the file "VIEWS," in the loops subdiretory.

2. type:

```
     mv ## [##-##]  (where ## is the view number to be made; viewnums can be
                    single, or ranges separated by a "-")
```

THE PICTURE EDITOR

CURSOR MOVEMENT:

1 pixel	arrow keys
diagonally 1 pixel	Fn-arrow keys
10 pixels	ctrl-PgUp,PgDn,left,right

PICTURE EDITING:

Space	forward 1 instruction
Backspace	back up 1 instruction
Tab	forward 1 picture code
Shift-tab	backward 1 picture code
Del	delete previous step
B	goto beginning of picture
E	goto end of picture

H	start horizontal line
V	start vertical line
D	start diagonal line
C	start curved line

F	start color fill
Fn1	set color
Fn2	clear color
Fn3	set priority
Fn4	clear priority
Fn5	toggle rubberband mode
Fn6	toggle status display
Fn7	toggle RGB/composite mode
Fn9	show graphics screen
Fn10	show priority screen
ESC	to drawing screen from Fn9/Fn10
Shift-Fn1	save screen image as bitmap

G	goto Nth instruction (tab N times)
L	get picture file from disk
S	save picture file from disk
W	wipe out present picture (clear)
<	delete from beginning to current location
>	delete to end from current location
A	appends a picture to end of present picture
Q	quit (confirm?)

PRIORITY & COLOR NUMBERS:

0	black
1	blue
2	green
3	cyan
4	red
5	magenta
6	brown
7	white
8	dark grey
9	bright blue
10	bright green
11	bright cyan
12	bright red
13	bright magenta
14	yellow
15	bright white

ON DRAWING SCREEN:

purple lines	priority lines
white lines	priority & color lines
green lines	color only

THE SOUND EDITOR

The four vertical bands represent voice1, voice2, voice3 and the noise channel respectively. Each horizontal bar is one frame of the interpreter (or 1/9-second) in length. The beginning of the sound is at the top of the screen, the end is at the brown line. Voice3 can be linked to the noise channel, or not.

All keys below are non-caps sensitive. Use the number keypad for best results: the most commonly used keys are on it. Editor must be booted directly from the floppy with DOS 2.10.

DISK ACCESS

G	get a file from disk
P	put a file on disk
Esc	cancel current request

CURSOR KEYS

-	back a frame
+	forward a frame
Home	back a voice
PgUp	forward a voice
Del/.	delete a frame
Ins	insert a frame (of silence)
Enter	copy current frame & insert

PITCH CONTROL

Up/Down	adjust volume up/down
Left/Right	adjust pitch lower/higher
L	Lengthen current note
S	Shorten current note

MISC. FUNCTIONS

End	play to the END of the sound
PgDn	toggle between: all voices, current voice, no voices
R	Repeat present sound continously
C	Couple voice3 to noise generator
Q	Quit

MUSIC ROUTINES

BIG PICTURE:
Name the source files:

```
          snd##.mt
```

(The extension refers to the compiling program, maketune.) After entering the source code, use the "MT" batch file to compile it. Use the form:

```
          mt ##
```

where ## is the number of the sound that matches the ## in the snd##.mt filename. The batch file calls MASM to assemble the code, LINK to link it, EXE2BIN to convert it, then MAKETUNE to convert it to an adventure game file named snd.##. It may then be tested with the PLAY program:

```
          PLAY snd.##
```

When the file is correct, enter:

```
          UPDATEM
```

which will clean the various junk files out of the music directory, and transfer the snd.## files from the \music directory to the ..\snd directory.

SOURCE CODE FORMAT

INCLUDES
Files must start with:
```
     include   pitches
     include   tempo##
```

Files must end with:
```
     include   musicend
```

Somewhere in the file must be the labels:
```
     voice1
     voice2
     voice3
```

whether the other voices are used or not.

PITCHES
Notes are specified by name and octave number, followed by a rhythm, and an optional integer articulation. A middle B-flat quarter-note, separated from the following note by a two-cycle space would be:
```
     bf3  q,2
```

Pitch range is from C0 to B5. Accidentals are specified by either "f" or "s" before the octave number.

RHYTHMS
The "tempo" files contain the range of rhythms, generally from s for sixteenth to w for whole notes. See them for details.

RESTS
Rests are specified by "rest" and a duration:
```
     rest q
```

VOLUME
Volume is specified by:
```
     atten ##
```

where ## is an integer from 0 to 14. Maximum volume = 0;   minimum is 14. Volume remains set until a new atten is specified. Do NOT follow an atten by a rest!

TO MAKE A DIRECT I/O GAME DISK

1. edit the file "diskinfo" to describe every disk in the game. See the "makedisk" subdirectory for example. This file allocates disk space for whatever you will later write to disk.

2. Create a file called "vol0.drv" (and/or "vol1.drv" etc.) that describes volume 0. It must contain, in order:

```
     volume 0            the name of the volume
     for rm, pic, etc    the pathname to each subdirectory
     files list          the name of every file to be included in the volume
```

3. Run "makedisk" batch file, with a formatted disk in drive a:.

TO UPDATE AN OLD DISK:

1. Don't run "makedisk" as it does everything all over, and perhaps you only need to change one small part.

2. Call "mkvol," giving it as a parameter, the number of the volume to be made. Example, to construct volume 0, type:

```
          MKVOL 0
```

3. After all volumes are made, call "mkdisk" with the following parameters:

```
          MKDISK all          re-do everything
                 p            change interpreters only
                 b            change boot files only
                 ao           change aniobj and object tables only
                 wt           change words.tok only
                 v01          change directories only
                 0v           change volume 0 only
```

ERROR MESSAGES

Error messages are now contained in a separate dynamic logic, which must be on every disk. They appear in the format:

```
error # : n
```

1	can't find "where"
2	object #n too large in set.view
3	view not loaded for set.view
4	object #n too large in set.loop
5	loop #n too large in set.loop
6	view not set
7	object #n too large in set.cel
8	cel #n too large in set.cel
9	message #n not defined in print
10	too many logics in load.logics
11	too many pictures in load.pic
12	too many animated objects
13	object #n too large in animate.obj
14	object #n too large in get.obj
15	object #n too large in drop.obj
16	invalid action
17	out of memory by #n bytes
18	picture not loaded in draw.pic
19	object #n too large in draw
20	no view of object #n in draw
21	object #n too large in erase
22	cell size > save size for view #n
23	picture not loaded in discard
24	view not loaded in discard
25	too many views in add.to.pic
26	stack overflow
27	attempt to set memory mark when one is already set
28	start/stop update when obj not drawn