Arcade Game Designer

With thanks to Kevin Thacker and Mauricio Muñoz Lucero for their help during development

Instructions for version 1.4

Arcade Game Designer is a tool for writing your own simple arcade games. While it won't help you create snazzy scrolling shoot-em-ups, it should be capable of producing a variety of basic arcade affairs, puzzlers and platformers so long as you're patient and prepared to stick at it. That is, of course, provided that AGD does not crash horribly or something. Games produced using the utility are stand-alone and can be distributed freely.

The Spectrum version of AGD has already been put to good use by various authors, and is responsible for games like Trooper: point 5, Apulija-13, Donkey Kong Reloaded, Hedgehogs and Toofy in Fan Land. All are thoroughly recommended, and can be found in the World of Spectrum archives if you really want to see what AGD can do. The CPC port should be capable of similar things.

Now for the bad news. AGD has its own built-in language, a rather simplistic affair with simple functions and a few built-in variables which you'll have to pick up. In truth the lexicon is quite small, it's mostly a question of understanding how to use each particular command or function. While the program will tell you if it finds a word it doesn't understand (although it won't say where), syntax checking is non-existent. Unlike PGD and SEUD, AGD is not quite so user-unfriendly.

Because I wanted to produce a games designer for everyone to use, the program is free to download from my page at itch.io. Copyright, however, remains with the author. Donations, while happily accepted, are not necessary.

AGD comes as you find it, probably containing bugs I haven't yet found. Please do not expect email support; I will be happy to answer questions and hear requests for enhancements and bug fixes on the AGD forums instead. I get around to updates from time to time, other projects and life generally permitting. Until then save your work as often as possible, and preferably as a snapshot if you are using an emulator.

In 2018 I released MPAGD, or Multi-Platform Arcade Game Designer. This is a tool for Windows 10 and allows the user to create games for a number of different formats, including the CPC. Direct conversions of old Spectrum AGD games are now possible, it will even generate placeholder graphics for you.

Most importantly of all, have fun. In the meantime, I'm off to Ripley to get hideously drunk.

Jonathan Cauldwell, 20th February 2013.

Website:

http://www.spanglefish.com/egghead/

AGD T-shirts:

https://www.redbubble.com/people/RetroGameDev/shop?asc=u

AGD Downloads:

https://jonathan-cauldwell.itch.io/

Constructing a game

AGD requires a little swapping of tapes for saving and loading, so if you are using an emulator it is a good idea to switch off the option to automatically load tape files as soon as they are inserted. Failure to do so could result in the loss of any work you have done on a game should you attempt to change the tape.

Basically, games are a combination of different elements:

* character blocks are combined together to form screens
* sprites (including the player) are positioned in their starting positions on each screen, and assigned a type characteristic
* logic is then written for each type of sprite - player, enemies, objects etc.

After that, it's a question of adding anything else you want.

Main Menu

Enables you to select a function of the game you wish to design. The amount of memory remaining available to your application is displayed in the top right corner of the this screen.

Window area

You can vary the play area's size and position in the Window/scrolling option. The cursor keys allow you to position the window anywhere on screen.

Changing the window size and/or scrolling direction can alter the amount of data required for each column or row of blocks in each screen, and thus invalidate the present levels. If this is the case, the program will ask if you wish to destroy the map data before allowing you to proceed. It is a good idea to select the size of the play area first, and stick with it.

1 - narrow window
2 - widen window
Q - shorten window
A - lengthen window
ESC/ENTER = Return to main menu

Character Block Design

Blocks are used to construct the screens. Each block is 8 x 8 pixels in size, and has a different type assigned to it. There are a number of different types, each with its own different set of attributes. These are:

SPACE - player moves freely through free space blocks.
PLATFORM - player moves freely past platforms from left, right or below.
WALL - a block which is impassable from any direction.
LADDER - player moves freely through ladder blocks in any direction.
FODDER - as solid wall, can be removed with certain functions.
DEADLY - sprites move through this, DEADLY function tests for contact
CUSTOM - sprites move through this, CUSTOM function tests for contact

Move cursor around the character with the cursor keys. Use SPACE or 0 to set a pixel. ENTER or ESCAPE returns to the main menu. You can set the current pen colour with P.

Q = Move left through list of block properties
W = Move right through list of block properties
L = Last block
N = Next block
P = Paper colour
B = Change brightness and flash bits
X = Create a new character block
D = Delete current character block
ESC/ENTER = Return to main menu

Screen Layout

Move cursor around the screen with cursor keys. Use 1 and 2 to select the character block you wish to place at the current cursor position. SPACE or 0 places the desired block on the current screen at this position. Alternatively, you may wish to use fast draw mode, by pressing F. This automatically places the selected block at the current cursor position every time you move the cursor. To exit fast draw mode and return to manual placing of blocks, press the F key again. You can copy a screen. Press M to mark the screen you wish to copy, then go to the screen you wish to copy over and press K to paste it in. ENTER returns to the main menu.

1 = Move left through block table
2 = Move right through block table
F = Toggle fast draw mode
M = Mark current screen
K = Paste from marked screen
N = Next screen
P = Previous screen
X = Create a new screen
D = Delete current screen
ESC/ENTER = Return to main menu

Sprite Images

Sprites are 9x16 pixel images which make up moving parts such as the player, player bullets, enemy craft, etc. Sprites may have any number of frames.

Move around the grid with the cursor, fixing and deleting pixels with SPACE or 0. Because AGD's collision detection is coordinate based, it is a good idea to fill out as much of the 9x16 area as you can, particularly for player and enemy sprites. Collectables don't matter as much, players seldom notice or care if they pick up a collectable from a couple of pixels away.

X - insert sprite
D - delete sprite
M - copy sprite or tile to clipboard
K - copy sprite or tile from clipboard
N - next sprite
L - previous sprite
P – pen colour
I - insert frame
R - remove frame
F - next frame
ESC/ENTER = Return to main menu

Sprite Positions

This allows you to position sprites in their start positions for each level using the cursor keys. 8 sprite types are available, and sprites can be set up as any of these, allowing you to position bonuses, enemy sprites, the player sprite's starting position or any other type of sprite you have set up. Sprite images are separate from sprite types. Images are just the images you draw in the sprite editor, whereas a sprite's behaviour is defined by its type. Type 0 is usually used for the player's sprite (or sprites), although you could change this. You will need to allocate a type for each different moving item, such as aliens, bullets or collectables, and determine their movements using a simple scripting language. See the section on events for information about moving sprites.

N - next screen
P - previous screen
Q - move next sprite
I - change sprite image
T - change sprite type
D - delete current sprite from screen
X - add new sprite to screen
ESC/ENTER = Return to main menu

Objects

Distinct from sprites, objects are static items which a sprite might pick up, or possibly drop. Similar to sprites, they consist of 8x16 pixel images, although only have one frame. These can be edited in much the same way as sprite images.

For each object you will need to define its position at the beginning of the game. Unlike sprites, objects do not respawn every time the player enters a room, so they can be picked up in one room then dropped in another, making them ideal for arcade adventure games. The starting screen is changed with Q and W keys. A starting screen of 255 indicates that the object starts the game in the player's inventory. Objects which are missing at the start of play, ie don't appear on any screen or in the player's inventory, should be assigned a starting screen of 254. Press P to place an object in its designated position on its starting screen.

X - insert object
D - delete object
I – pen ink colour
M - copy image or tile to clipboard
K - copy image or tile from clipboard
N - next object
L - last object
Q - change starting screen
W - change starting screen
P - position object on starting screen
ESC/ENTER = Return to main menu

Objects may be handled in the events code with GOT, GET, PUT and DETECTOBJ. Additionally, the OBJ variable stores the result of the last DETECTOBJ command, although you can use it for other things as well, if required. IF GOT n will be true if the player has object n in his inventory. GET n will put object n in the player's inventory (regardless of wherever the object currently is), PUT n will drop an object at the current sprite position, and DETECTOBJ will check to see if the current sprite is touching an object, placing the result in the OBJ variable. If no object is detected at the sprite position the value 255 will be returned.

For example, the following code will automatically pick up any object over which a sprite passes:

```
DETECTOBJ
IF OBJ <> 255
  GET OBJ
ENDIF
```

Because GET and PUT make no check to see if the object is already in the player's inventory, they can be used to place objects on screen or remove them more or less at will. You may wish to have enemy sprites dropping bonuses, then remove them after a few seconds. Just because you remove an object from the screen doesn't mean you have to award the points or bonus that the player would have gained by collecting it himself.

Map Layout

AGD will allow sequential levels, but also has the ability to create explorer games where the player can explore a map.

The "map" of your game is arranged as a grid of 10 x 8 locations, all of which start off empty. Empty locations appear as two hyphens "--", rooms appear as the screen numbers, eg "01" or "12". As each room is designed in the screen designer it can be placed in this grid at a chosen location. To move around the map simply use SCREENUP, SCREENDOWN, SCREENLEFT, SCREENRIGHT commands. During the game it will not be possible for the player to move into an empty grid space. commands such as NEXTLEVEL or LET SCREEN=9 will not alter the current map position, so are best used in games with sequential levels, unless you are confident of what you are doing.

The red cursor can be moved around the grid using the cursor keys, to change the room at a particular grid location move use keys '1' and '2'. Rooms are displayed in the bottom section of the screen as they are selected to make matters easier.

Your map will follow the rules of Euclidean geometry, but you can bend the rules to create a warped playfield should you desire. Moving left from a room situated at the left edge of the map will cause the player to re-appear in the room placed at the right edge of the next row up, if one is placed there. Similarly, moving right from a room at the right edge will take the player to the room at the extreme left edge of the room one row below. If you don't want this to happen you should construct walls on the relevant screens to form a physical barrier. It is also possible to re-use a room, that is to make it appear more than once in your map.

Moving from one screen to another will not alter the player sprite's coordinates, so these should be set manually at the same time as the SCREENLEFT, SCREENRIGHT etc. is performed. Any player sprites set up for a screen will only be used for the very first screen, or to spawn a new player sprite should he die on that screen. Should the player die on a screen where he has no default position he will not be respawned.

Press 'X' to declare a grid location as the point at which the player is to begin the game.

1 = Select previous room from list
2 = Select next room from list
X = Select room where player first starts the game
ESC/ENTER = Return to main menu

Jump Table

If your game makes use of gravity, you may want to edit the jump table. This allows you to edit the steepness of jumps and/or falls, allowing you to determine how high sprites can jump, or how quickly they descend when they fall through gaps in the floor.

The jump table is separated into a series of individual steps, and you can change the distance between these steps with the cursor keys. The red column represents the step you are currently editing. When you are happy with your jump table, press ENTER to return to the main menu.

Sound

H - Hear present sound
N - Next sound
P - Previous sound
X - Create new sound
D - Delete sound

Move around the values with the cursor keys, increasing and decreasing them with 1 and 2. SPACE or 0 will toggle noise or tone off.

To play a sound in your game, use the SOUND command in the relevant event.

Save Game

This displays the memory locations used by your game and prompts for you to press a key. Insert a blank disc and press a key. The file game.bin will be written to the disc. To reload your saved game from the disc, just type run"game

Test Game

Allows you to test your creation. Press ESCAPE at any point to return to the editor.

Messages

This is where you can define the text messages your game will use. Press ESCAPE or ENTER at any point to return to the main menu.

N = Next message
P = Previous message
SPACE or 0 = Edit current message
X = Create new message
D = Delete message

Collision Distance

The collision distance is the sprite collision detection adjuster. Standard sprites are 16x16 pixels, so normally they will hit each other if their coordinates are less than 16 pixels apart. For a chunky sprite such as Monty Mole or Egghead, this is ideal. However, you may wish to draw a player sprite which is somewhat slimmer, leaving gaps around both sides. You won't want to count the gap as part of the sprite collision detection, so you can reduce this distance to a lower pixel count. Bear in mind that this setting only affects horizontal distances, and applies to all collision detection between sprites.

Events

This is the part where you get to have a say about the game logic, and can change the way it works in a variety of different ways. While the editor and compiler are not going to rival a proper language like BASIC, AGD does provide a limited number of statements, functions and variables which should enable a variety of different arcade games or arcade adventures to be created. Think of it as an arcade version of Incentive Software's GAC - you may need to be inventive about how you implement the features you want, but then that is half

the fun.

There are several events for which you can write the logic. Aside from those which occur at certain times in the game, there are events associated with 8 sprite types. These are the events which control the movement and logic of each type of sprite. Sprite type 0 is usually reserved for the player's sprite, so this code should test for keys and move the sprite around accordingly. The rest are all yours to do with as you wish. For example, you could choose to make sprite types 1 and 2 different alien nasties with different movement patterns, and perhaps use sprite type 3 for bonus sprites which the player picks up.

```
Player (type 0) control - player movement, reading keys, collision detection
Sprite type 1           - behaviour of sprites with type 1
Sprite type 2           - behaviour of sprites with type 2
Sprite type 3           - behaviour of sprites with type 3
Sprite type 4           - behaviour of sprites with type 4
Sprite type 5           - behaviour of sprites with type 5
Sprite type 6           - behaviour of sprites with type 6
Sprite type 7           - behaviour of sprites with type 7
Game initialisation     - at start of game, set up variables, (intro page?)
Restart screen          - what happens when player restarts a screen
Initialise sprite       - whenever a sprite is initialised
Main loop 1             - every game loop
Main loop 2             - every game loop
Completed game          - when game completed successfully, eg congratulations
Kill player             - happens when player loses a life with a KILL command
```

To modify the events, use cursor up/down to select the event, then press space or 0. Once you have selected an event, you can edit its code. ESCAPE returns to the event selection screen. The code editor has keys to cut and paste individual lines should you wish to move sections around. When finished, the editor will quickly run over your code, and report back if it does not understand any of it. Your code is then compiled directly to lightning-fast machine code.

Functions can only be used after an IF.

IF
Test. If the following condition is true the code up to the next ENDIF statement is executed. IF can be used with a function, or to test variables or sprite parameters against each other, or against specific numeric values.

ENDIF
Marks the end of the conditional code.

LET
As in BASIC, this allows you to assign a value to a variable or sprite parameter. The value assigned can be a number, or another variable or sprite parameter.

KEY
Function. Expects a single numeric argument and condition is true if the key is pressed.

CANGOUP, CANGODOWN, CANGOLEFT, CANGORIGHT
Functions. Condition is true if the current sprite can move up/down/left/right.

LADDERABOVE, LADDERBELOW
Functions. Condition is true if the current sprite can go up/down a ladder.

X, Y
Sprite parameters. These are the coordinates of the current sprite.

A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P
Global variables. These hold 8-bit values, so values of 0-255 are possible.

SCREEN, LIVES
Global variables. These contain the current screen number and the lives remaining.

TYPE
Sprite parameter. This is the type of sprite being processed. Best used in conjunction with the IMAGE parameter, setting this parameter will completely change the sprite's behaviour - handy for turning a nasty into a bonus, or making a sprite stop and explode before killing it. There is no reason why you couldn't change a sprite to type zero and put it under the player's control, or change the player's sprite type to something else with a slightly different set of controls. So long as the new sprite type has appropriate code set up in the relevant event, there's no limit to what you could do. If you haven't set up any code for the sprite type your sprite will just sit there - which may be okay if that's what you want.

IMAGE, FRAME
Sprite parameters. These are the sprite and frame numbers shown in the sprite editor. You can change the sprite according to whichever direction the player is facing, or perhaps you might want to give the player a choice of vehicles to control. Setting a frame number beyond the limit of the sprite will result in a different sprite image being displayed, so use FRAME with caution. When setting the image, it is a good idea to set the frame to 0 at the same time, unless you have a very good reason for not doing so.

DIRECTION, PARAMA, PARAMB
Sprite parameters. You can use these as you see fit, perhaps to indicate the direction in which a particular sprite is moving, or the particular phase it is going through. Invaluable for any form of enemy AI.

ANIMATE
Command. Animates the present sprite, automatically cycling through the frames in ascending order.

ANIMBACK
Command. As ANIMATE, but cycles through frames in descending order.

NEXTLEVEL, RESTART
Commands. Move to next level, and restart current level respectively.

SPRITEUP, SPRITEDOWN, SPRITELEFT, SPRITERIGHT
Commands. Move the current sprite accordingly. No check is made for blocks in the way, or out-of-screen conditions, so you will have to do that yourself with functions such as CANGOLEFT or LADDERABOVE.

SPAWN
Command. Expects 2 parameters for sprite type and image. This spawns a new sprite with the specified type and image at the current sprites's position. The new sprite is created with FRAME, DIRECTION, PARAMA and PARAMB all set to zero. The current sprite is unaffected.

SPAWNED
Command. Should only be used after a SPAWN instruction. This command selects the newly spawned sprite. Any code written after SPAWNED will refer to the new sprite. Use ORIGINAL to switch back to the original primary sprite. Alternatively, you may prefer to place your code in the Sprite Initialisation event, it's up to you.

REMOVE
Command. Removes the present sprite from the table. Useful for destroying enemies or picking up objects.

DIGUP, DIGDOWN, DIGLEFT, DIGRIGHT
Commands. These remove any fodder blocks above, below, or to the left or right of the current sprite. Any other blocks are unaffected.

The DIG commands are useful for opening doors. You could set up your door as fodder blocks, which are normally impenetrable, then use conditional DIG commands in your player movement code when the player has performed a certain action - maybe a variable has been set to a certain value, or a particular object has been collected.

COLLISION
Function. Requires one numeric argument to specify the sprite type. Condition is true if the current sprite is in collision with another sprite of the type specified.

OTHER
Command. Should only be used after a successful COLLISION check. This command selects the other sprite, ie the secondary one with which the original sprite has just collided. Any code written after OTHER will refer to the secondary sprite. Use ORIGINAL to switch back to the original primary sprite when you have finished the code for the secondary sprite.

ORIGINAL
Command. Used after OTHER and SPAWNED commands, this reverts to the original sprite.

ENDGAME
Command. Ends the game in victory. This performs the Completed Game event. Completing the last screen in a game with sequential levels will do the same. The only other way in which a game can finish is if the player loses all his lives, but that does not perform the Completed Game event.

SHOWSCORE
Command. Shows the score at the current cursor position. Should be immediately preceded by instructions setting up the line and column position for the cursor.

SCORE
Command. Expects to be followed by a number to add to the score. SCORE 100 will add 100 points to the player's total. Values 0 to 255 are valid.

SOUND
Command. Starts a sound effect with the AY sound chip. Expects a single parameter for the sound number to play.

CLS
Command. Clears the screen.

BORDER
Command. Expects an argument from 0 to 7 inclusive. Sets the border colour.

COLOUR
Command. Expects an argument. Sets the ink colour.

DELAY
Command. Expects an argument. Pauses for the duration specified.

MESSAGE
Command. Expects an argument for the number of the message to display.

KILL
Command. Initiates the kill player event and decrements the life counter. You should set up the lives counter in the initialisation event using something like LET LIVES = 3.

LINE, COLUMN
Variables. These determine the position at which the score or message will be displayed. They are temporary and change every time a sprite is displayed, so always use them immediately before a MESSAGE or SHOWSCORE command.

GETRANDOM
Function. Generates a random number between zero and the argument, and places it in the RND variable. GETRANDOM 100 will generate a number from 0 to 99, GETRANDOM 2 will generate a zero or 1.

RND
Variable. The last random number generated by GETRANDOM.

ADD, SUBTRACT
Commands. Add or subtract to or from a sprite parameter or variable, eg. ADD 1 TO A or SUBTRACT 5 FROM B.

DISPLAY
Command. Displays the numeric value in the variable or parameter that follows. For example, DISPLAY LIVES.

SCREENUP, SCREENDOWN, SCREENRIGHT, SCREENLEFT
Commands. Move up, down, left or right one screen if possible.

DEADLY
Function. Condition is true if the current sprite is in contact with a deadly block.

CUSTOM
Function. Similar to DEADLY, true if the current sprite is in contact with a CUSTOM block.

WAITKEY
Command. Waits for a keypress.

JUMP
Command. The current sprite will jump, provided it is not already in mid-air and there are no walls in the way. As you would expect, any sprite can be made to jump, not just those under the direct control of the player.

FALL
Command. Provided the sprite is not already falling or jumping, this checks to see if the sprite is standing on top of solid ground. If not, it will start to fall, using the descending half of the jump table (your game will automatically find this point, even if you have modified the jump table). FALL is useful if a sprite type is subject to gravity, eg platform games.

GOT
Function. Used in IF statements and expects a single argument. The expression is true if the specified object is owned by the player. The argument can be a variable if required, so IF GOT 1 is valid, as is IF GOT A. You could even use a sprite parameter as an argument if you wish.

GET
Command. Expects a parameter specifying the object to get. Places the specified object in the player's inventory, regardless of where it is. The object could be on the current screen, another screen, or just missing. It is up to you to decide when the player can get a particular object. You may want to use DETECTOBJECT to determine when a sprite is touching an object first.

PUT
Command. Requires an argument specifying the object number. The object is dropped onto the screen at the current sprite's position, unless it is already on the current screen. Argument can be numeric, a variable or sprite parameter.

DETECTOBJ
Command. Detects objects touched by the current sprite, and places the result in the OBJ variable. If the sprite is touching more than one object, only the object with the lowest number is detected. If no object is detected, OBJ will be set to 255.

OBJ
Variable. The last object number detected by DETECTOBJ.

EXIT
Command. All processing of the current event is terminated.

REPEAT
Command. Requires a parameter to determine the number of repeats. The code up to the next ENDREPEAT will be repeated the given number of times. REPEAT cannot be nested.

ENDREPEAT
Command. This marks the end of the block of code to be repeated. Code placed in a REPEAT..ENDREPEAT loop is automatically indented by the editor to improve readability.

MULTIPLY
Command. Multiply a sprite parameter or variable, eg. MULTIPLY C BY 7.

Example 1

If we select sprite type 3 as our static collectable sprite type, we may write something along these lines to detect player collisions and pick them up:

```
IF COLLISION 0
  REMOVE
  SCORE 100
ENDIF
```

We could add a sound effect:

```
IF COLLISION 0
  SOUND 1
  REMOVE
  SCORE 100
ENDIF
```

Going further, we can insert the following lines at the start we can change the sprite's behaviour:

```
IF CANGOUP
  SPRITEUP
ELSE
  REMOVE
ENDIF
```

So our event now looks like this:

```
IF CANGOUP
  SPRITEUP
ELSE
  REMOVE
ENDIF
IF COLLISION 0
  SOUND 1
  REMOVE
  SCORE 100
ENDIF
```

We now have a collectable sprite which moves up until it hits something, and then disappears, like a bubble. Now we need some code to spawn our new sprites at the bottom of the screen. We can do this in the main loop, so select the Main loop 1 event, and insert these lines:

```
GETRANDOM 127
IF RND <= 6
  SPAWN 3 0
  SPAWNED
  IF TYPE = 3
    LET X = 176
    GETRANDOM 80
    LET Y = RND
    ADD 4 TO Y
  ENDIF
  ORIGINAL
ENDIF
```

This code first generates a random number from 0 to 127, and if that number is less than or equal to 6, it generates a sprite of type 3 (the previous event we edited) and a sprite image of 0. If the sprite has spawned properly, it then sets the x position to the bottom of the play area (assuming the play area ends 8 pixels from the bottom of the screen), and chooses a random y position between 4 and 84.

Example 2

Assume we have a play window 22 characters high with a single character gap at the top and bottom of the screen. Say we want to draw a box around the window. We can do this by displaying messages in the Game initialisation event. We could set up message 0 as the top line to display above the window, with message 1 as the bottom line. To draw the lines down the sides we could set them up in message 2, and then use a REPEAT loop to draw them. Something like this might do the trick for us:

```
LET LINE = 0
MESSAGE 0
REPEAT 22
  MESSAGE 2
ENDREPEAT
MESSAGE 1
```

Technical information

Games created with the utility are stand-alone, and should work independently of the editor itself. AGD does not change the interrupt mode, so you are free to set up your own interrupts, to play music, for example. However, you should not disable the interrupts. If you wish to set up your own interrupts using IM2, make sure your service routine increments the 4-byte clock, or your game will hang. AGD does not change the value of the IY register, although IX is used throughout, mostly as a sprite pointer.

The Save game option shows the memory occupied by a game, the first address being the start address. Below this, the code for the editor is located. RAM at address 41728 upwards is used as a dummy collision map area to distinguish between different types of blocks - walls, ladders, empty space and so on. No buffer is required for screen data decompression, since these are expanded on-the-fly by the routine that draws the screen.

AGD 1.4 for the CPC has been designed with the same functionality as AGD 3.5 for the ZX Spectrum, and it should be possible to convert the Spectrum games to the CPC, and vice versa. Future releases may diverge as bespoke functionality is added for each platform. However, should you wish to produce a game for both platforms, CPC v1.4 and Spectrum v3.5 are almost identical. You will have to cope wih the different screen sizes in your own way.

Games are controlled via joystick and it is possible to select which one your game will use by poking address 13580. A value of 0 tells your game to use joystick 0, poking any other value will use joystick 1.

Other address you may find handy to interrogate at the end of each game are the score (a six-digit string) at 14935, the lives remaining at address 13642 and the flag which determines whether or not the player completed the game successfully at 13666.

Further Reading

Paul Jenkinson has produced a series of excellent AGD tutorials on YouTube, and while they are for the Spectrum, the CPC version operates in exactly the same way.

The official AGD forum is the place to find news of the latest releases, tips on getting the best out of the utility, and details of games which have already been written with AGD or are currently under development. You can find the forum here:

http://arcadegamedesigner.proboards.com/

Upgrading to MPAGD

Multi-Platform Arcade Game Designer is designed to make conversions between CPC, ZX Spectrum and other machines straightforward.

MPAGD is a far more powerful tool than AGD. It has READ and DATA local to each event or sprite, WHILE loops, a particle engine for starfields, vapour trails, explosions and the like, scrolling ticker messages, functionality to put redefinable keys in your game, collectable blocks, a full set of operators > < = >= <= <>, more global and sprite variables and you can finally comment your code at last!

You can start from scratch and design your own game for the CPC, then convert it to other machines later should you desire. Alternatively, take one of over a hundred (and that number is rapidly growing) Spectrum games written with AGD 4.0 to 4.7 and convert it to the CPC.