GameDev

Sam Jordan

COLLABORATORS

	TITLE : GameDev	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Sam Jordan	July 19, 2024	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

GameDev

1.1 System-compliant games development on the PPC

System-compliant games development on the PPC
1997/98 by Sam Jordan

Tips, thought and suggestions regarding the development of games for the ↩
PowerAMIGA

1.2 Preface

The first floppy disk any newly bought AMIGA swallowed was most likely a games-disk in most of the cases. It was in my case, at least, and I still remember the game very well: Silkworm. That was around the end of 1991.

Games have always been one of the supporting legs of the AMIGA - together with the operating system and the famous custom-chips. It must be for this reason that the AMIGA got stuck with the image of a games-machine - and this prejudice is still uttered by uninformed users of different models. But his prejudice contains a grain of truth for sure.

To me, the prime time of AMIGA games development was at the beginning of the nineties. The AMIGA was still doing good and games were high in demand. With

the problems and the final demise of Commodore the games began their downward spiral as well. Quantity as well as quality started to diminish. The short interlude with AMIGA Technologies under Escom wasn't enough to bring new momentum to the games business. Nevertheless, the games scene was still alive although it had of course a lot of its importance.

During all this time the games developers on competing systems did all but sleep. In the past years gaming technology has experienced a literal boom which was mostly caused by the ever-improving hardware. In the shadow of this huge technology jump, the AMIGA fell by the wayside. Only in the areas where it had been generations ahead of the competition it could now, at best, keep competing.

The demands of gamers have fundamentally changed. Today graphics, music and speed are what matters while a few years back values such as playability and atmosphere were stressed a lot more. This isn't even meant to imply that today's games don not have these qualities anymore. It rather means that the priorities have changed. And in order to be successful in the highly fought-over games market, one HAS to pay attention to what the player actually wants.

The AMIGA lacked one thing: speed. Exactly this missing speed is having a highly negative impact on the games business. If you wanted speed, you went out and bought a competing system that boasts 3-digit CPU clocks and got your kick out of the speed frenzy.

So let me get to the point of this document. The AMIGA is now getting what it used to lack: speed!

By employing the PowerPC-processors the AMIGA can leap-frog ahead and be at equal terms with the competition again. The AMIGA is given another chance at being at the very top. The same holds true for the games.

A PowerAMIGA makes games possible that achieve the same quality as those on the competing machines. The PowerPC-processor is state-of-the-art and can even outpeform for example the Intel-processors.

However, in order to get the maximum performance out of games for the PowerAMIGA, a lot of know-how is necessary. The AMIGA is not only a processor, it consists of many single components that closely work together. As important as the processor is the graphics hardware. If access to the video memory completely bogs down a game even the fastest processor on this planet is of no use. The key is to spot existing bottlenecks and to circumvent them. This document in front of you contains some very valuable tips for games-programming. The biggest bottlenecks are described and solutions are presented.

The WarpOS-archive contains two demo-programs that have a high significance in this respect. 'Cybermand' and, first and foremost, 'voxelspace'. Both programs show how you can and should develop a game for the PPC. And both programs impressively demonstrate what amazing speeds can be achieved in a 100% system-compliant fashion if you have enough experience.

This document not only covers programming-techniques but also something even more important: game-design. A game only gets bought if is capable of convincing customers of its quality. An important matter in this context is the creation of suitable demo-version. The very biggest part of all demo

version I have seen to this date were not worth a thing. It really is a pity that such mistakes can destroy the large amount of work invested in a game.

Why do I take the trouble of writing such a document?
Simple: we at HAAGE&PARTNER like good games. And I want the AMIGA to step forward into a better future - and I want the games to take that step with it.

Sam Jordan

1.3 Introduction

The purpose of this document is to provide new impulses for games development on the AMIGA. It describes what must be considered when developing games especially for the PowerPC-processor and what the strengths and weaknesses of the PPC are as well as how to use these to your advantage.

At first, technical matters are discussed and the biggest problems that may occur are described. After that the aspects of game-design are covered - I will discuss what has to be considered during the development process in order to create a game of high playability and quality.

What exactly can be achieved with the PPC is demonstrated by the two demo-programs 'cybermand' and, above all, 'voxelspace'. These two programs demonstrate how a PPC-game can be structured and impressively prove what high speed can be achieved in a completely system-compliant way.

What is necessary for super-fast games: they must be able to run under WarpOS. WarpOS is currently the only chance to run fast games/demos on the dual-processor-board. WarpOS was also specially optimized for games and offers features which can be very helpful especially when developing games.

1.4 Philosophy

Right at the beginning I want to state which philosophy I think should be applied when developing games in the future:

Future games should be completely compliant with the respective operating system they run on, but they should also offer the user the option of switching on certain non-compliant options.

AMIGA games used to be fast, even very fast. The secret of this high speed is easily uncovered: the majority of all games switched off the operating system and took over the hardware. Result: A game that ran on every AMIGA without problems was rarely seen. Especially AMIGAs that were not yet built at the time a game was written made a lot of problems as the programmers didn't (and couldn't) know anything about those new machines.

As a counter-argument to my above statement, one could say:
System-compliant games are too slow.

What anybody who utters this argument should first do is to take a look at the provided 'cybermand' and 'voxelspace' demos. After that, the argument looses

any credibility. The 'cybermand' and 'voxelspace' programs run 100% system-compliant and are very fast nevertheless. It IS possible!

'Voxelspace' also demonstrates the second part of my philosophy: it offers the user really 'evil' hacks as additional parameters. The user must explicitly activate them and can then enjoy even higher speed if the program still runs at all. This is exactly where the problem is: the hacks will possibly not work on every machine anymore. And that is the reason why it is absolutely mandatory that the game runs completely system-compliant before all other things. Because if it is system compliant it will always run.

There are even more arguments in favour of the 'system-compliant + optional hacks' philosophy. First of all: Hardware is getting ever faster. If a program runs system-compliant today, it will still run with later hardware but at increased performance. If the game relies on hacks chances are quite good it will not run at all on later hardware.

Developing system-compliant games is a lot more efficient than developing games that keep re-inventing the wheel all the time and are only marginally faster. At the time we are living one has to work efficiently to satisfy the quality expectations of the users and still release the game in time. Very often games aren't released until they are already outdated...

It is important that AMIGA-games receive a new image. The potential customer must not be afraid that the game might not run on his machine at home. This fear often leads to the game not getting bought at all. I know this from personal experience. As a HighEnd-user I very often considered it too risky to buy a game which was quite likely to be incompatible with some kind of hardware in my system.

1.5 Choice of programming language

If you wanted fast games in the past, there was no question on what language to use for that. Assembly language. Anything else was too slow.

Nowadays this has changed fundamentally. Games are getting ever more complex and bigger and bigger at the same time. As things stand today, the part that is really responsible for the speed is only a small fraction of the entire project.

It is absolutely obvious that it is pure madness to create projects of these sizes in 100% assembly language. This is simply too much work and results in too little advantage.

On the other hand, assembler is not an obsolete language. If a game was developed entirely in a high-level language it usually is a lot slower than if select subroutines have been written in assembly language. In my opinion, that is what comprises the ideal mixture.

For a high-level programming language one need not look any further than C, as it is a very portable language. For this reason a project should first be done entirely in C. If the program is done, it gets tested. This is where a so-called profiler can be a valuable aid: it measures the load of the single functions and then generates a statistic that shows which functions are time-critical and which are not. On the basis of such a profiler-statistic it

is decided which functions should be implemented in assembly language. The key is to find a reasonable limit here - not too many functions should be adapted but neither too few.

After choosing the appropriate functions to be adapted you should first take a look at the assembler-output of the compiler. You should do this because very often you need not rewrite the entire function but rather only have to edit the assembler-output by hand. After adapting a function the game gets tested again and the effect is analysed. This analysis is possibly very useful to help with the decision which further functions should be re-written in assembly language.

Let me use this opportunity to make another, quite unconventional, suggestion: an interesting idea would be to provide such core routines as external modules or as very small programs. If these mini-programs are well documented, assembler-specialists could re-write these routines and then simply exchange the module. This would also be of advantaged to the creator of the game. This idea was used a few times when it came to ChunkyToPlanar-conversion algorithms which could simply be exchanged as modules.

1.6 Structure of a PPC-game

When developing software for the dual-processor-system there is always one decision to be made: which parts of the program will be compiled for the 68K and which ones for the PPC?

A first approach is of course to run all computing-intensive program parts on the PPC while all parts that call a lot of system-function and have little or no influence on the speed are compiled for the 68K. If you are equipped with a good developer's environment such as 'StormC', you can recompile any source code for the respective other CPU without having to change the source code at all.

However, this approach must be viewed in a wider context. The dual-processor-solution is most likely to be only a first step on the way to a pure PPC-AMIGA. It can be expected that the AMIGA-OS (or just parts of it) will be ported to the PPC in the future and 68K-software is then executed through an emulator.

If you then also consider that developing a game takes a significant amount of time, it makes a lot more sense to compile most if not all parts of the software for the PPC. The performance-losses on the dual-processor board that are caused by the many CPU-switches when calling system functions are reduced to a bare minimum due to the high-speed communication-interface of WarpOS.

If it should show that the performance loss is too big in certain areas of the program, you can still compile that part for the 68K. When there is a completely native AMIGA-OS in the future, the game will still run and the games developer can still release an update that has the appropriate parts compiled for the PPC.

When designing the inner loops that are most time-critical in the case of most games, special care must be taken. You have to consider very carefully how to design such a main loop and for which CPU to compile it. Your options are the following approaches:

1. The entire main loop is compiled for the PPC. This results in a CPU-switch becoming necessary for every system call (e.g. for the message-handling). If a game only needs few system-calls, this approach is the most ideal one.

2. The main loop is compiled for the 68K and the computing-intensive functions are done on the PowerPC. When taking this approach you should be able to make do with very few CPU-switches. This is most likely to be the ideal approach very often. This is also the approach taken for the two demo-programs 'cybermand' and 'voxelspace', both of which make do with one or two PPC-calls.

3. The main loop and the computing-intensive functions run in two different tasks on two different CPUs. By the use of signals the two tasks can be synchronized. This approach is very delicate as multiprocessing very often has counter-productive effects.

You should always choose the most ideal approach for the inner loops in order to ensure maximum performance. If this approach is not ideal for a pure PPC-system, you can still create an update that achieves maximum performance on the target system.

As a general rule: system-calls are a large performance-factor if they cause a CPU-switch. If only few calls are necessary, this has very little effect - if a lot of CPU-switches occur, however, this has a detrimental effect on performance. For this reason it is vital to create the structure of the game-core in a way that minimizes the amount of CPU-switches.

1.7 Graphics programming

In this section we will cover the technical aspects of games programming, especially graphics programming. If you want to proceed to the technical part immediately, please go to the Main Graphics Menu.

Nowadays the main weakness of the AMIGA is graphics access - exactly that which use to be one of its strengths in the past. The famous custom-chips of the AMIGA were mainly designed for scrolling and sprite handling - areas in which the AMIGA is still able to shine today. In the age of 3D-games, the custom chips are unfortunately useless.

On top of that, the custom-chips can only operate in Chip-RAM. That means that if the processor were to provide the custom-chips with data, it would have to write all data into the unbelievably slow Chip-RAM. The access to the Chip-Memory was ideally suited to the 68000 ages ago. Needless to say that this kind of access speed is in-acceptable for a processor that is more than a hundred times faster.

Where is the way out of this problem? If a high performance is desired, a graphics board is necessary. And exactly that is currently the main weakness of the AMIGA:

1. The so-called 'standard' AMIGA does not have a graphics board as of today.

2. AMIGA graphics-boards are incredibly expensive compared to similar boards on competing systems and offer a worse performance at the same time. The

high price seems to be related to the small market volume as graphics-board owners still are a minority.

3. The normal AMIGA graphics-system is based on bitplanes. However, many games (especially 3D-games) rely on the chunky-format used by most graphics boards for the internal representation of graphics. This results in further performance loss of the software because the graphics data has to be converted to bitplane-format first. In this case the performance increase caused by the PPC-processor becomes minimal because the algorithm is heavily memory-dependent and of course especially because it has to write into the slow Chip-RAM all of the time.

4. The largest part of games does not support graphics boards. As a direct result, games run hopelessly slower even on fast 68K-systems equipped with a 68040 or 68060 than on competing systems that don't even have a much faster processor. A PowerPC-processor doesn't even make sense for any of those games.

When using a PowerPC-processor, a graphics board becomes an absolute must. For this reason games written for the PPC must support graphics boards in the most optimal way possible. In the following part, progamming graphics boards is described as well the system-compliant programming of ECS/AGA-graphics.

At this point I want to place a remark to a software which already supports a lot of the techniques explained below and which offers appropriate functions to the programmer: RTGMaster from Steffen Haeuser (found on Aminet under gfx/board). Game programmers should really consider to do the graphics programming with RTGMaster instead of coding everything new. RTGMaster is still developed further intensively and supports in future new graphics hardware.

The following explanations are of interest for these programmers who can't use RTGMaster due to certain reasons.

ECS/AGA-programming
Graphics-boards programming
CyberGFX+
TurboGFX

1.8 ECS/AGA-Programming

The major part of all AMIGAs still has no graphics card. When developing a game you should always analyse your target market - if a large part of this market does not own a graphics board, ECS and AGA must be supported. It is important that a game not only supports the LowEnd-AMIGAs but also the HighEnd-AMIGAs. So a modern game must be able to run on different graphics systems.

In the following part, no distinction will be made between AGA and ECS as this difference is negligible when programming in a system-compliant way. This kind of programming is referred to as PAL-programming as because games programmed in the way described will run in PAL-mode.

Most of the games for ECS/AGA used to be programmed in a non-system-compliant way because the AMIGA-OS didn't offer very good support for games programming. Below I will show how to develop system-compliant games with the operating system still running.

The following examples are shown in assembly language. Sense and purpose of these examples can of course be transferred to any high-level language as well. The assembley-parts are direct excerpts from the source code of the voxelspace-demo.

Generally speaking, a system-compliant ECS/AGA-game with double-buffering works in the same way as a game that was programmed close to the hardware. Therefore the examples below always operate on two bitplanes. The order of the following menu items corresponds to the order of programming.

The following statements take it as granted that the game uses all of the available screen area and that there is only one window present. Games such as strategy games or industry simulations can of course use several windows. In these cases the display-speed doesn't play a vital role anyways. That is why this case won't be covered any further in this place. The following statements are mostly of interest for 3D-games and similar ones.

Allocating the bitplanes
Opening a screen
Assigning the colors
Opening a window
Clearing the mouse pointer
Generating the graphics
Switching the image buffers
Closing the window
Closing the screen
Freeing the bitplanes

1.9 Allocating the bitplanes

First of all you have to allocate two image buffers that in turn consist of several bitplanes. The number of bitplanes depends on the number of colors to be displayed. In the case of AGA, eight bitplanes are commonly used.

I will now discuss a method that also works with earlier version of the operating system. There are also alternative ways that require a higher version number.

At first you have to create space for two bitmap-structures. This can be done statically as well as dynamically. The organisation of the bitmap-structures is described in the 'graphics/gfx.i' include-file.

After that both of the bitmap structures are initialized by using 'graphics/InitBitMap'. This function is passed these arguments: address of a bitmap-structure; height, width and depth of the bitmap.

Finally the 'AllocRaster' function is called for every bitmap to be allocated. 'AllocRaster' allocates the necessary memory for the bitplanes. After that, the return values are entered into the bitmap-structure as PlanePtrs. You can also allocate several bitmaps at the same time if you proceed as described below.

From now on we will name the two bitmap-structures as follows:

ActualBitmap : The bitmap that is currently active. The bitplanes that are part of this bitmap-structure are currently displayed.
HiddenBitmap : The bitmap that is currently inactive. The bitplanes that are part of this bitmap structure are available for creating the graphics.

Below you find an excerpt from the voxelspace source-code that allocates the bitplanes:

```
*******************************************************************************
*
*       d0 = SetupBitmaps
*
*       prepares two bitmaps for double buffering
*
*       Out:
*       d0 = error code
*
*       error codes:    -1 = success
*                        4 = not enough memory
*******************************************************************************
SetupBitmaps
                movem.l d1/d5-a2,-(sp)
                moveq   #2-1,d7                 ;two bitmap-structures
                lea     bitmap1,a0              ;a0 -> first bitmap-structure
                move.l  a0,ActualBitmap         ;bitmap1 is ActualBitmap
.loop
                move.l  a0,a2
                lea     bm_Planes(a2),a2        ;a2 -> bitplane-pointer-array
                moveq   #8,d0                   ;color depth (256 colors)
                move.l  #320,d1                 ;width of the bitplanes
                move.l  #256,d2                 ;height of the bitplanes
                CALLGRAF        InitBitMap      ;initialize bitmap-structure
                move.l  #320,d0                 ;Width of a bitplane
                move.l  #256*8,d1               ;Height of a bitplane * 8
                CALLGRAF        AllocRaster     ;allocate a 8 bitplanes
                tst.l   d0                      ;enough memory available?
                beq.b   .error                  ;no -> error message
                move.l  d0,a0
                move.l  #(256*8*320/8/4)-1,d5
.clear
                clr.l   (a0)+                   ;an image-buffer is cleared
                subq.l  #1,d5
                bne.b   .clear
                moveq   #8-1,d6                 ;all 8 bitplane-Ptr are entered
.loop2                                          ;into the bitmap-structure
                move.l  d0,(a2)+
                add.l   #320/8*256,d0           ;jump to next bitplane
                dbra    d6,.loop2
                lea     bitmap2,a0              ;now repeat everything for
                move.l  a0,HiddenBitmap         ;the 2nd bitmap (HiddenBitmap)
                dbra    d7,.loop
                moveq   #-1,d0                  ;function successful
                bra.b   .end
.error
                moveq   #4,d0                   ;return error code
.end
                movem.l (sp)+,d1/d5-a2
                rts
```

1.10 Opening a screen

After allocating the bitmaps, a screen is opened. Best suited to this purpose is the 'OpenScreenTagList' function of intuition.library. This function requires a NewScreen-structure or a taglist as parameters. In our example we will only use a taglist.

If the taglist is implemented in assembley language, it can be implemented as follows (example from the voxelspace source-code):

```
ScreenTags
                dc.l    SA_Left                 ;left screen border
                dc.l    0
                dc.l    SA_Top                  ;upper screen border
                dc.l    0
                dc.l    SA_Width                ;height of screen (here: 320)
                dc.l    320
                dc.l    SA_Height               ;width of screen (here: 256)
                dc.l    256
                dc.l    SA_Depth                ;depth of screen (here: 256
                dc.l    8                       ;colors)
                dc.l    SA_BitMap               ;pointer to a bitmap-structure
                dc.l    bitmap1
                dc.l    SA_Quiet                ;Keeps intuition from messing
                dc.l    TRUE                    ;with the screen
                dc.l    SA_Type                 ;ScreenType
                dc.l    CUSTOMSCREEN
                dc.l    TAG_DONE
```

It is important that the longword after SA_Bitmap contains a pointer to an initialized bitmap-structure. If the bitmap was allocated dynamically, the pointer still remains to be entered.

'OpenScreenTagList' can now be called. This function returns a pointer to a screen-structure which will be used later. The screen-structure can be used to determine a pointer to the ViewPort that will also be needed later. The following code demonstrates the opening of a screen:

```
                sub.l   a0,a0                   ;not a NewScreen-structure
                lea     ScreenTags,a1           ;pointer to above taglist
                CALLINT OpenScreenTagList       ;open screen
                moveq   #5,d1                   ;provide error code
                move.l  d0,_Screen              ;save screen-pointer
                beq.w   .error                  ;Is it Null? -> error
                move.l  d0,a0
                move.l  d0,ScreenAddress        ;enter in window-taglist
                lea     sc_ViewPort(a0),a0      ;get address of viewports
                move.l  a0,_VPort               ;save viewport-address
```

The 'move.l d0,ScreenAddress' line will be discussed later in the 'Opening a window' chapter.

1.11 Assigning the colors

This section is valid for ECS/AGA as well as for graphics-board programming.

After the screen has been opened, the address of the ViewPort is also available. This enables you to assign the desired colors to the screen. There are two cases that have to be distinguished:

1. ECS/OCS

If the game has to support ECS/OCS, the following path must be taken:

The assigning of the colors is done using the 'graphics/LoadRGB4' system function. This functions requires a pointer to the ViewPort, a table of all colors and the amount of colors as parameters. The color table is a simple array of USHORT (2 bytes) the elements of which signify the RGB-value of the color. An example of a color table with 4 colors:

```
ColorTable
                        dc.w    0               ;black
                        dc.w    $f00            ;red
                        dc.w    $00f            ;blue
                        dc.w    $fff            ;white
```

The code for assigning these colors then looks as follows (example for 32 colors):

```
                move.l  _VPort,a0               ;a0 -> ViewPort
                lea     ColorTable,a1           ;a1 -> color table
                moveq   #32,d0                  ;d0 = number of colors
                CALLGRAF        LoadRGB4
```

2. AGA / CyberGFX

If AGA/CyberGFX are to be supported, the matter has to be handled differently:

The assigning of the colors is done by using the 'graphics/LoadRGB32' system function. This function requires a pointer to the ViewPort and a pointer to the color table as parameters. However, this table has a different organisation than that for the ECS/OCS-variant:

The first value of the table signifies the amount of colors to be loaded. The second word is the first color number to be loaded (usually zero). After that follows the actual table. Each color consists of 3 longwords with the first longword signifying the red-component, the second longword the green-component and the third longword the blue-component. Another important peculiarity: the color values must be provided LEFT-ALIGNED! In the case of 8-bit color-values the actual color value must be left-shifted by 24 bits. The end of the table is marked by a null-longword.

Example for a color table with 4 colors:

```
ColorTable
        dc.w    4                               ;4 colors
        dc.w    0                               ;first color is color number 0
        dc.l    $ff000000,$ff000000,0           ;yellow
        dc.l    0,0,0                           ;black
        dc.l    $7f000000,$7f000000,$7f000000   ;grey
        dc.l    0,0,$40000000                   ;dark blue
```

```
        dc.l    0                               ;end of table
```

The code for assigning the colors then looks as follows:

```
                move.l  _VPort,a0               ;a0 -> ViewPort
                lea     ColorTable,a1           ;a1 -> color table
                CALLGRAF        LoadRGB32
```

1.12 Opening a window

This section is valid for ECS/AGA as well as for graphics-board programming.

Now the window can be opened. Best suited to this purpose is the 'OpenWindowTagList' function of intuition.library. This function requires a NewWindow-structure or a taglist as arguments. In our example we will only use a taglist.

If the taglist is implemented in assembley language, it can look as follows (example from the voxelsspace source-code):

```
WindowTags
                dc.l    WA_Left                 ;left window border
                dc.l    0
                dc.l    WA_Top                  ;upper window border
                dc.l    0
                dc.l    WA_Width                ;window width (PAL: 320)
WinWidth
                dc.l    320                     ;CyberGFX : dynamically
                dc.l    WA_Height               ;height of window (PAL: 256)
WinHeight
                dc.l    256                     ;CyberGFX : dynamically
                dc.l    WA_Activate             ;window is to be activated
                dc.l    TRUE                    ;immediately
                dc.l    WA_Borderless           ;the window has no border
                dc.l    TRUE
                dc.l    WA_RMBTrap              ;right mouse button will be
                dc.l    TRUE                    ;trapped
                dc.l    WA_ReportMouse          ;mouse movement will be
                dc.l    TRUE                    ;reported
                dc.l    WA_IDCMP                ;IDCMP-flags to be supported
                dc.l    IDCMP_MOUSEBUTTONS!IDCMP_RAWKEY!IDCMP_MOUSEMOVE! ↩
                   IDCMP_DELTAMOVE!IDCMP_ACTIVEWINDOW!IDCMP_INACTIVEWINDOW
                dc.l    WA_CustomScreen         ;address of the parent screen
                dc.l    0
                dc.l    TAG_DONE
```

The longword after WA_CustomScreen must be filled with the address of the screen as described in the 'Opening a screen' chapter.

This window-structure is of course only an example. Depending on the application it will look differently. It depends on e.g. which interaction by the user should be handled. If mouse control is not provided for it does not make sense to specify the WA_ReportMouse and the corresponding IDCMP-flags.

Here we will cover the IDCMP-flags a little further. The following flags might

be of interest for games:

```
IDCMP_MOUSEBUTTONS      : The user pressed a mouse button
IDCMP_MOUSEMOVE         : The User moved the mouse. IDCMP_MOUSEMOVE should
                          always be used together with IDCMP_DELTAMOVE.
IDCMP_RAWKEY            : The user has pressed a key. In this mode the keys
                          are passed unmodified. This can be used to test
                          special keys such as Ctrl, Alt, Shift, etc.
IDCMP_ACTIVEWINDOW      : Might be used if a game was put into pause-mode by
                          switching screens. By activating the window with the
                          mouse the game can be resumed.
IDCMP_INACTIVEWINDOW    : Can be used to put the game into pause-mode if the
                          user switches screens an de-activates the window.
IDCMP_DELTAMOVE         : Should always be used when IDCMP_MOUSEMOVE is used.
IDCMP_VANILLAKEY        : The user has pressed a key. The data has already
                          been processed in this mode. Therefore it can not
                          be used to watch all keys.
```

Now 'OpenWindowTagList' can be called. This function returns a pointer to a window-structure which will be needed later. The window-structure can be used to determine the pointer to the RastPort that will also be used later. The following code demonstrates the opening of a window:

```
sub.l   a0,a0                   ;Not a NewWindow-structure
lea     WindowTags,a1           ;Pointer to above taglist
CALLINT OpenWindowTagList       ;Open windows
moveq   #6,d1                   ;provide error codes
move.l  d0,_Window              ;save window-pointer
beq.b   .error                  ;Is it zero? -> error
move.l  d0,a0
move.l  wd_RPort(a0),_RPort     ;Save RastPort address
```

1.13 Clearing the mouse pointer

This section will show you how to make the mouse pointer disappear. The reason to do this is that an ever-persistent mouse pointer can be quite annoying under certain circumstances.

First of all, a small memory area of Chip-RAM is allocated (e.g. 16 bytes). This area is then used as the mouse pointer which becomes completely transparent and thus invisible.

After that the 'intuition/SetPointer' function is called. This can be done as seen below:

```
move.l  #16,d0                  ;allocate 16 bytes
move.l  #MEMF_CHIP!MEMF_CLEAR,d1
CALLEXEC        AllocVec        ;allocate Chip-RAM
move.l  d0,NullPointer          ;store address
move.l  d0,a1                   ;to a1 for SetPointer
move.l  _Window,a0              ;window address to a0
moveq   #1,d0
```

```
                moveq   #1,d1
                moveq   #0,d2
                moveq   #0,d3
                CALLINT SetPointer              ;clear mouse pointer
```

If you want the mouse pointer to re-appear, this can be done by calling 'intuition/ClearPointer'.

If the window is closed, the mouse pointer automatically re-appears.

1.14 Creating the graphics

After all preparations have been made, the game enters the main loop where, iteration for iteration, the next frame is computed and displayed.

In modern games (such as 3D-games, for example) the graphics are first created in Fast-RAM using the chunky-format (i.e. one byte per pixel). After that the picture is converted using a conversion algorithm. These so-called C2P or ChunkyToPlanar-converters are available by the score. Special attention has to be paid to selecting an algorithm that fits the task to be accomplished. Very often such algorithms only work for a certain color depth.

Any such C2P-function should be implemented for the PPC. The algorithm is actually not at all suited to demonstrate the power of the PPC because it is very memory-intensive and writes to Chip-RAM. On the other hand, it is possible to achieve small performance-improvements over a 68K-processor. This could possibly have visible impact on the overall result.

After generating the actual background-graphics, any status bars and texts are displayed. These can be written into the invisible image buffer using conventional graphics.library functions (e.g. Move, Text, ...) before the buffers are swapped and the changes become visible. An example for this can be seen in the 'voxelspace' demo in which, after computing the landscape, the status bar is generated using 'Move' and 'Text'.

Important: When using standard-functions of graphics.library, the RastPort-structure and the the RasInfo-structure within the ViewPort must be adapted before invoking the functions. The reason for this is that these functions operate directly on the current image buffer. Details can be found in the 'Switching the image buffers' chapter.

1.15 Switching the image buffers

A typical main loop could be structured similar to this:

```
                - compute image and put it into Fast-RAM
                - copy image to invisible image buffer/convert
                - swap visible and invisible image buffers
```

In this way you can create smooth and flowing animations (double buffering). Now we have to find a way to swap the two image buffers.

As mentioned in the previous chapter, 'Creating the graphics', some preparations must be made before calling standard-functions of graphics.library. These function always affect the current image buffer but we want their output to be made into the invisible buffer in order for the animation to proceed cleanly.

Switching is done as follows:

1. adapt RastPort and RasInfo-structures
2. use graphics.library functions
3. make above changes visible using 'ScrollVPort'
4. swap bitmap-pointers

First of all, care must be taken that the graphics.library functions actually affect the invisible buffer. This is done by making the invisible bitmap visible for these function without actually switching the bitplanes. This is done as described below:

```
move.l  _RPort,a0               ;get RastPort address
move.l  HiddenBitmap,a1         ;put address of the hidden bitmap ↩
   into a1
move.l  a1,rp_BitMap(a0)        ;enter into RastPort
move.l  _VPort,a0               ;get ViewPort address
move.l  vp_RasInfo(a0),a0       ;get RasInfo address
move.l  a1,ri_BitMap(a0)        ;enter into RasInfo
```

After that, functions such as 'Move' and 'Text' can be applied. These functions therefore still affect the invisible buffer.

As the third step, the changes to the RastPort and RasInfo are made visible. By doing so, the two image buffers are effectively switched:

```
move.l  _VPort,a0               ;get ViewPort address
CALLGRAF        ScrollVPort     ;swap image buffers
```

Finally the pointers to the actual and hidden bitmaps are exchanged with each other so that the game can create the next image in the other image buffer that has now become invisible:

```
move.l  ActualBitmap,d0
move.l  HiddenBitmap,ActualBitmap
move.l  d0,HiddenBitmap
```

1.16 Closing the window

After the game was finished a clean exit should been made. This means that e.g. all memory should be freed again. The window should of course be closed, too. This is done by using the 'intuition/CloseWindow' function:

```
move.l  _Window,d0              ;get window-structure address
```

```
                beq.b   .nowindow               ;was the window opened?
                move.l  d0,a0                   ;address to a0
                CALLINT CloseWindow             ;close window
.nowindow
```

1.17 Closing the Screen

It is needless to say that the screen should be closed as well. This is done through 'intuition/CloseScreen':

```
                move.l  _Screen,d0              ;get screen-structure address
                beq.b   .noscreen               ;was the screen opened?
                move.l  d0,a0                   ;address to a0
                CALLINT CloseScreen             ;close screen
.noscreen
```

1.18 Freeing the bitplanes

The bitplanes that were allocated should be freed. First of all, the 'FreeRaster' function is called for every bitplane. 'FreeRaster' is the counterpart to 'AllocRaster'.

If the bitmap-structures were allocated dynamically, these should be freed as well.

Below you find an excerpt from the voxelspace-demo. It is the part that frees the bitplanes:

```
*****************************************************************************
*
*       FreeBitmaps
*
*       frees all the memory allocated by 'AllocRaster'
*
*****************************************************************************
FreeBitmaps
                movem.l d0/d1/d6-a2,-(sp)
                moveq   #2-1,d7                 ;two bitmap-structures
                lea     bitmap1,a0              ;a0 -> 1st Bitmap
.loop
                move.l  a0,a2
                lea     bm_Planes(a2),a2        ;a2 -> Array of bitplane-pointer
                moveq   #8-1,d6                 ;8 bitplanes to be freed
.loop2
                move.l  (a2)+,d0                ;read bitplane-pointer
                beq.b   .next                   ;Null? -> nothing is freed
                move.l  d0,a0
                move.l  #320,d0                 ;width of bitplane
                move.l  #256,d1                 ;height of bitplane
                CALLGRAF        FreeRaster      ;free bitplane
.next
                dbra    d6,.loop2
```

```
lea     bitmap2,a0              ;repeat for 2nd bitmap
dbra    d7,.loop
movem.l (sp)+,d0/d1/d6-a2
rts
```

1.19 Graphics-board programming

System-compliant graphics-board programming is slightly different from ECS/AGA-programming. One significant difference: no double-buffering is done. Usually a game/demo that does not use double-buffering starts to flicker, but if CyberGFX is programmed in the right way these effects almost never occur.

The programming process is similar to that for ECS/AGA:

Choosing the screen mode
Opening a screen
Assigning the colors
Opening a window
Clearing the mouse pointer
Creating the temp. RastPort
Creating the graphics
Closing the window
Closing the screen
Freeing the temp. RastPort

1.20 Choosing the screen mode

Graphics boards usually allow choosing the screen mode quite freely. Therefore AMIGA-users that own a graphics-board will most likely use different screen modes. It is important especially for games to work with any sensible screen mode.

The choice of screen mode has also direct influence on the game performance. The smaller the resolution, the faster the game will run. In this way it is even possible to play existing 68K-games super-smoothly (e.g. by choosing a 192*128 screen mode). The graphics will look quite blocky in this case, but due to the smooth animation this will very often go unnoticed.

A game should offer the user the option of choosing his preferred screen mode. CyberGFX offers a function for this purpose which displays a screen mode-requester and returns the mode chosen by the user. In order to use this function, cybergraphics.library must be opened successfully.

The 'CModeRequestTagList' function requires a taglist as parameter and returns the DisplayID of the selected screen mode which is used later. The taglist can be used to filter the screen modes to be displayed in the requester. If the game does not work with certain screen dimensions, these should be filtered out of the list. Further information on this can be found in the autodocs for cybergraphics.library.

Most of the time the screen mode-entries will be filtered by their color depth. Games will only work with 16- or 24-bit-modes in the rarest of cases.

The taglist will then look as follows (the include-file for cybergraphics.library must also have been loaded):

```
CyberModeTags   dc.l    CYBRMREQ_CModelArray
                dc.l    ColorModel
                dc.l    TAG_DONE
ColorModel
                dc.w    PIXFMT_LUT8
                dc.w    -1
```

Now the screen mode-requester can be displayed:

```
                sub.l   a0,a0                   ;must be NULL
                lea     CyberModeTags,a1        ;pointer to taglist
                CALLCYBERGFX    CModeRequestTagList     ;show requester
                moveq   #8,d1                   ;provide error code
                tst.l   d0                      ;function successful?
                beq.w   .error                  ;no -> error
                moveq   #0,d1                   ;provide error code
                cmp.l   #-1,d0                  ;did the user choose 'Cancel'?
                beq.w   .error                  ;then -> quit game
                move.l  d0,DispID               ;store DisplayID
```

1.21 Opening a screen

Opening a screen works almost as described above, by calling 'OpenScreenTagList'. The only addition is that the screen mode can now be chosen by the user, therefore data such as height, width and DisplayID is dynamic. These values must first be determined and then placed in the taglist for 'OpenScreenTagList'.

The DisplayID is returned by 'CModeRequestTagList' (see the earlier chapter, 'Choosing the screen mode'). Width and height must now be determined. This can be done with another CyberGFX-function: 'GetCyberIDAttr'. It requires a DisplayID and a mode as parameters. The mode specifies which information is desired. We will call this function twice: At first with the 'CYBRIDATTR_HEIGHT' parameter to get the height, then with the 'CYBRIDATTR_WIDTH' parameter to determine the width. All these values are placed in the screen-taglist after that.

After opening the screen, the width and height should be read from the screen-structure and placed in the window-taglist. Additionally, the address of the ViewPort is determined from the screen-structure and then stored.

The screen-taglist can be organized as follows:

```
ScreenTags_C
                dc.l    SA_Quiet                ;keeps intuition from
                dc.l    TRUE                    ;messing with the screen
                dc.l    SA_Width                ;screen width
ScreenWidth
                dc.l    0
                dc.l    SA_Height               ;screen height
ScreenHeight
                dc.l    0
```

```
                dc.l    SA_Depth                ;screen depth
                dc.l    8
                dc.l    SA_DisplayID            ;screen-DisplayID
DispID
                dc.l    0
                dc.l    TAG_DONE
```

The code then looks as follows:

```
                move.l  DispID,d1               ;get DisplayID
                move.l  #CYBRIDATTR_HEIGHT,d0   ;height will be determined
                CALLCYBERGFX    GetCyberIDAttr  ;get height
                move.l  d0,ScreenHeight         ;and place in taglist
                move    d0,AreaHeight           ;store height
                move.l  DispID,d1               ;get DisplayID
                move.l  #CYBRIDATTR_WIDTH,d0    ;width will be determined
                CALLCYBERGFX    GetCyberIDAttr  ;get width
                move.l  d0,ScreenWidth          ;and place in taglist
                sub.l   a0,a0                   ;not a NewScreen-structure
                lea     ScreenTags_C,a1         ;pointer to taglist
                CALLINT OpenScreenTagList       ;open screen
                moveq   #5,d1                   ;provide error code
                move.l  d0,_Screen              ;store pointer to screen
                beq.w   .error                  ;pointer is NULL? -> error
                move.l  d0,a0
                move    sc_Width(a0),AreaWidth  ;store width
                move    sc_Width(a0),WinWidth+2 ;width -> in Window-Taglist
                move    sc_Height(a0),WinHeight+2 ;height -> in Window-Taglist
                move.l  d0,ScreenAddress        ;pointer to screen in Window-TL
                lea     sc_ViewPort(a0),a0      ;get ViewPort address
                move.l  a0,_VPort               ;store address
```

1.22 Creating a temp. RastPort

The actual copying of the image data into the graphics-memory is done by the 'graphics/WritePixelArray8' function. Further details on this will be given at a later point. What now is important is that this function requires a temporary RastPort.

First of all, memory must be provided for the RastPort-structure (the structure is defined in the 'graphics/rastport.i' include-file). This can be done either statically or dynamically. In the following example this RastPort will be named 'tempRP'.

The RastPort is now initialized using the 'graphics/InitRastPort' function. After that, a bitmap-structure and the appropriate bitplanes are allocated using 'graphics/AllocBitMap'. This function requires AMIGA-OS V3.0. If you want to support graphics boards, you can safely assume that OS V3.0 is present.

The code then looks as follows:

```
                lea     tmpRP,a1                ;a1 -> space for temp. RastPort
```

```
CALLGRAF        InitRastPort    ;initialise RastPort
moveq   #0,d0
move    AreaWidth,d0            ;determine screen width
moveq   #1,d1                   ;Height = 1 row
moveq   #8,d2                   ;depth = 256 colors
move.l  #BMF_MINPLANES,d3       ;special flag
move.l  _RPort,a0               ;get window-RastPort
move.l  rp_BitMap(a0),a0        ;pass Bitmap as 'Friend'
CALLGRAF        AllocBitMap     ;allocate Bitmap + Bitplanes
lea     tmpRP,a0                ;address of temp. RastPort -> a0
move.l  d0,rp_BitMap(a0)        ;enter new bitmap
```

1.23 Creating the graphics

Within the main loop the image is (as before) first created in FAST-RAM. It now must be copied into the graphics-RAM in some way so it can be displayed.

There are two functions that can take care of this copying:

1. cybergraphics/WritePixelArray
2. graphics/WritePixelArray8

The first function has the advantage that it is also able to copy just parts of the screen. The disadvantage: This function is INFINITELY slow.

The second function is as fast as possible. Unfortunately it has the disadvantage of offering only very few parameters. This function is commonly used to copy the entire image into the graphics memory.

The very biggest problem when programming graphics-boards in this way: as no double-buffering is possible, it is not possible anymore to add further graphical elements using the standard graphics.library functions without the game starting to flicker heavily. All additional graphical elements must already be created in the 'ChunkyBuffer' (picture in FAST-RAM) which can be quite tedious as the standard graphics.library functions can not be used to do this.

The code for copying the entire picture from the 'ChunkyBuffer' (Fast-RAM) into the graphics-RAM could look as follows:

```
move.l  _RPort,a0               ;get RastPort address
moveq   #0,d0                   ;xstart = 0
moveq   #0,d1                   ;ystart = 0
move    AreaWidth,d2            ;xstop = width - 1
subq    #1,d2
move    AreaHeight,d3           ;ystop = height - 1
subq    #1,d3
move.l  ChunkyBuffer,a2         ;a2 -> image-data in Fast-RAM
lea     tmpRP,a1                ;a1 -> temp. RastPort
CALLGRAF        WritePixelArray8 ;copy image
```

1.24 Freeing the temp. RastPort

The temporary RastPort that was created for the 'WritePixelArray8' function should be freed again. This can easily be done using the 'FreeBitMap' function.

```
lea     tmpRP,a0                ;RastPort address -> a0
move.l  rp_BitMap(a0),a0        ;bitmap address -> a0
CALLGRAF        FreeBitMap      ;free bitmap
```

Finally, the temp. RastPort itself should be freed if it was allocated dynamically.

1.25 CyberGFX+

A very big disadvantage of the graphics-board programming described above is the lack of double- or multi-buffering support. This also results in the situation that no additional elements can be added to the already computed graphics using the standard-graphics-functions. This poses of course a severe limitation.

Below I will describe an extension to system-compliant graphics programming that is able to do real multi-buffering by employing a few tricks. Adding additional graphics elements isn't a problem anymore, either.

This technique is system-compliant although switching the buffers using 'ScrollVPort' is not guaranteed anywhere. However, this method has always worked for me. In order to avoid any problems, games should always support graphics-board access without multi-buffering whenever possible.

The programming itself is done very similar to the conventional graphics-board programming described earlier:

Choosing the screen mode
Triple Buffering
Opening a screen
Assigning the colors
Opening a window
Clearing the mouse pointer
Creating the temp. RastPort
Creating the graphics
Closing the window
Closing the screen
Freeing the temp. RastPort
Problems

1.26 Opening a screen

Opening a screen works almost as described above, by calling 'OpenScreenTagList'. The only addition is that the screen mode can now be chosen by the user, therefore data such as height, width and DisplayID is dynamic. These values must first be determined and then placed in the taglist

for 'OpenScreenTagList'.

As described in the 'Triple Buffering' section, the screen is opened at three times the height that was originally specified.

The DisplayID is returned by 'CModeRequestTagList' (see the earlier chapter, 'Choosing the screen mode'). Width and height must now be determined. This can be done with another CyberGFX-function: 'GetCyberIDAttr'. It requires a DisplayID and a mode as parameters. The mode specifies which information is desired. The function is then called twice. At first with the 'CYBRIDATTR_HEIGHT' parameter to get the height, then with the 'CYBRIDATTR_WIDTH' parameter to determine the width. All these values are placed in the screen-taglist after that.

After opening the screen the width and height should be read from the screen-structure and placed in the window-taglist. Additionally, the address of the ViewPort is determined from the screen-structure and then stored.

In addition to the conventional graphics-card programming, the vertical positions of the three buffers are now determined. The position of the first buffer is always zero, the position of the second buffer is equal to the buffer height and the position of the third buffer is equal to twice the buffer height.

The screen-taglist can be organized as follows:

```
ScreenTags_C
                dc.l    SA_Quiet                ;keeps intuition from
                dc.l    TRUE                    ;messing with the screen
                dc.l    SA_Width                ;screen width
ScreenWidth
                dc.l    0
                dc.l    SA_Height               ;screen height
ScreenHeight
                dc.l    0
                dc.l    SA_Depth                ;screen depth
                dc.l    8
                dc.l    SA_DisplayID            ;screen DisplayID
DispID
                dc.l    0
                dc.l    TAG_DONE
```

The code then looks as follows:

```
                move.l  DispID,d1               ;get DisplayID
                move.l  #CYBRIDATTR_HEIGHT,d0   ;height will be determined
                CALLCYBERGFX    GetCyberIDAttr  ;get height
                move    d0,AreaHeight           ;store height
                move.l  d0,d1                   ;multiply height by three
                add.l   d0,d0
                add.l   d1,d0
                move.l  d0,ScreenHeight         ;and place in Taglist
                move.l  DispID,d1               ;get DisplayID
                move.l  #CYBRIDATTR_WIDTH,d0    ;width will be determined
                CALLCYBERGFX    GetCyberIDAttr  ;get width
                move.l  d0,ScreenWidth          ;and place in Taglist
```

```
sub.l   a0,a0                   ;not a NewScreen-structure
lea     ScreenTags_C,a1         ;pointer to taglist
CALLINT OpenScreenTagList       ;open screen
moveq   #5,d1                   ;provide error code
move.l  d0,_Screen              ;store pointer to screen
beq.w   .error                  ;Is it NULL? -> error
move.l  d0,a0
move    sc_Width(a0),AreaWidth  ;store width
move    sc_Width(a0),WinWidth+2 ;width -> in window-Taglist
move    sc_Height(a0),WinHeight+2 ;height -> in window-Taglist
move.l  d0,ScreenAddress        ;pointer to screen in window-TL
lea     sc_ViewPort(a0),a0      ;get ViewPort address
move.l  a0,_VPort               ;store address
move    AreaHeight,d1           ;read buffer height
clr     ActualOffset            ;y-Pos. buffer 1 = 0
move    d1,HiddenOffset         ;y-Pos. buffer 2 = d1
add     d1,d1
move    d1,ThirdOffset          ;y-Pos. buffer 3 = d1*2
```

1.27 Creating the Graphics

Within the main loop the image is (as before) first created in FAST-RAM. It now must be copied into the graphics-RAM in some way so it can be displayed.

There are two functions that can take care of this copying:

1. cybergraphics/WritePixelArray
2. graphics/WritePixelArray8

The first function has the advantage that it is also able to copy just parts of the screen. The disadvantage: This function is INFINITELY slow.

The second function is as fast as possible. Unfortunately it has the disadvantage of offering only very few parameters. This function is commonly used to copy the entire image into the graphics memory.

After copying the graphics-data, the standard functions of graphics.library can be used to add additional graphical elements to the invisible buffer.

Now the buffers are switched. This is done through the 'graphics/ScrollVPort' function. After switching, the position values (ActualOffset, HiddenOffset, and ThirdOffset) are rotated so that the correct buffer can be chosen in the next iteration.

The code for copying the entire image from the 'ChunkyBuffer' (Fast-RAM) into the graphics-RAM could look as follows:

```
move.l  _RPort,a0               ;get RastPort address
moveq   #0,d0                   ;xstart = 0
moveq   #0,d1                   ;ystart = 0
move    AreaWidth,d2            ;xstop = width - 1
subq    #1,d2
move    AreaHeight,d3           ;ystop = height - 1
subq    #1,d3
add     HiddenOffset,d2         ;select invisible buffer
```

```
add     HiddenOffset,d3         ;select invisible buffer
move.l  ChunkyBuffer,a2         ;a2 -> image-data in Fast-RAM
lea     tmpRP,a1                ;a1 -> temp. RastPort
CALLGRAF        WritePixelArray8 ;copy image
```

At this point the additional graphical objects can be created. After that, the buffers are switched and the position values rotated:

```
move.l  _VPort,a0               ;get ViewPort address
move    HiddenOffset,d0         ;determine and negate position
neg     d0                      ;of the invisible buffer
move    d0,vp_DyOffset(a0)      ;enter into ViewPort
CALLGRAF        ScrollVPort     ;switch buffers
move    ActualOffset,d0         ;rotate position values of
move    HiddenOffset,ActualOffset ;the buffers
move    ThirdOffset,HiddenOffset
move    d0,ThirdOffset
```

1.28 TurboGFX

It is quite likely that some people already asked themselves why the image has to be created in FAST-RAM first and then copied into the graphics memory when programming graphics-boards. It should be faster to create the image directly within the graphics memory. That should make games even faster.

This section will now show that this is indeed possible and how to program and implement it.

Hinweis: Diese hier vorgestellte Technik ist im Prinzip systemkonform. Der direkte Zugriff auf das Grafik-RAM bzw. auf die Bitmap des Screens wird durch Locking-Mechanismen geschuetzt, wie es auch von CyberGFX verlangt wird. Diese Technik ist allerdings als LowLevel zu betrachten und es ist ueberhaupt keine gute Idee, sich völlig darauf abzustuetzen. Zudem kann es durchaus sein, dass diese Technik auf anderen Systemen mit anderer Gfx-Software nicht funktioniert. Deswegen sollten Spiele und Demos diese Technik als Ergänzung zu den anderen Techniken anbieten.

Note: the techniques presented here are basically system-compliant. The direct access to the graphics RAM resp. to the bitmap of the screen is enclosed by locking mechanisms, as required by CyberGFX. This technique has to be considered lowlevel and it's not a good idea only to support this technique without supporting other ones. And it's possible that this technique doesn't work with other gfx-interface-software. Therefore games and demos should only use this technique as supplement to the other ones.

The term TURBOGFX is derived from a CLI-parameter of the voxelspace-demo. This demo was my prototype for this new technique. The voxelspace-demo supports direct writing into graphics memory with both the 68K as well as the PowerPC.

Wenn ein Spiel mit TurboGFX läuft, dann ist es wichtig, dass die Screens nicht umgeschaltet werden, weil sonst Grafik-Fehler auf der Workbench erscheinen können. Bei richtiger Programmierung von TURBOGFX sollte das theoretisch nicht passieren, beim Voxelspace-Demo ist es aber schon

vorgekommen.

If a game is using TurboGFX it is important that no screens are swapped during the game, otherwise graphics failures can occur on the workbench. When programming TURBOGFX correctly this problem shouldn't occur, but it didn't occur sometimes with the voxelspace demo.

The following topics will be discussed:

Triple Buffering
The address of the image
Modulo-Problems
Implementation
Optimizing
Problems

1.29 Triple Buffering

A graphics-frame of a game is usually not built from the top left to the bottom right in a linear way. For this reason it is an absolute necessity to use several buffers when using TurboGFX. In order to completely avoid any flicker-effects, Triple-Buffering is used, i.e. three buffers that are rotated after each iteration.

Unfortunately there is no immediate way to program this Triple-Buffering (e.g. through a library function). Instead, a few tricks must be used in order for this to work:

When opening the screen its height is tripled. Now this over-height screen is vertically broken down into three parts. Each of these parts will be treated as an image buffer in its own right from now on. Furthermore, the fact that these buffers are located directly above each other can be used to our advantage.

One of these three image buffers will always be displayed and the next image will be created in one of the other invisible buffers. Switching is done using graphics/ScrollVPort (very similar to ECS/AGA).

1.30 The address of the image

One important question must still be answered: where is the left upper corner of the image located in the graphics-memory?

Now we have a problem: To this date I have not found a 100 percent reliable method to determine this pointer. Below I will discuss several methods and want to encourage the programmer to implement as many of these methods as possible and let the user choose among them with appropriate switches.

Now following are all methods that are known to me:

1. Determine the pointer to the graphics-RAM by using cybergraphics/ GetCyberMapAttr (CYBRMATTR_DISPADR parameter)

This method is used in the voxelspace-demo, if the option "MODE2" is enabled. It has worked almost every time for me.

However, I have the suspicion that the pointer to the beginning of the gfx-RAM need not be identical to the pointer to the beginning of the image in gfx-RAM. I have also heard about displacements of the image on screen - a problem that might be directly related to this.

2. Determine the pointer using cybergraphics/LockBitmapTagList

The mentioned function is called with the LBMI_BASEADDRESS parameter, followed by a call to cybergraphics/UnLockBitmap.

This method is the one authorised by CyberGFX and should be used as default method, just like in the voxelspace demo.

Judged by its definition this method seems to be a good one. It is used by the voxelspace-demo if the 'MODE2' CLI-parameter is activated - in case the first method should cause problems.

However, in certain older versions of CyberGFX the necessary function was broken. Therefore you should not rely 100 percent on this method.

3. cybergraphics/DoCDrawMethodTagList

This method was never tested by me. It should also be usable to determine the pointer. This function was also completely broken in earlier version of CyberGfx.

4. Use a library-function that returns the pointer

Such a function is rumoured to exist already. If such a library is well supported and maintained, this is a method that is recommended over all the others. If it suddenly stops working, only the library needs to be replaced.

1.31 Modulo-Problems

A further problem remains to be solved. The second row of an image is not necessarily located directly after the first row. This means an image can have a horizontal modulo. If this is not taken into account, the graphics-display might be come completely distorted.

There are two ways to find out how large the distance (in bytes) between two rows is. To do this, the 'cybergraphics/GetCyberMapAttr' function is called with the CYBRMATTR_XMOD argument. This function returns the width of the bitmap used - this is identical to the desired difference between to rows.

The algorithm for doing the actual image-calculations must therefore always assume that such a modulo-value exists.

1.32 Implementation

Below you find a few code fragments that illustrate programming in TurboGFX-mode.

First of all, the screen height must be multiplied by three when opening the screen.

After that you find the code to determine the address of the image as well as the modulo-value. In the example below, two methods are shown how to get the address of the bitmap.

Explanation of the variables:

```
BitmapWidth     : Bitmap width (distance between two rows)
ActualOffset    : number of rows between the beginning of the image and the
                  1st image buffer
HiddenOffset    : number of rows between the beginning of the image and the
                  2nd image buffer
ThirdOffset     : number of rows between the beginning of the image and the
                  3rd image buffer
ActualBitmap    : Address of the 1st image buffer
HiddenBitmap    : Address of the 2nd image buffer
ThirdBitmap     : Address of the 3rd image buffer
ChunkyBuffer    : ImageBuffer that is used for creating the graphics
AreaHeight      : Height of one image-buffer
Mode2           : Is 0, if the bitmap's address should be evaluated using
                  'LockBitMapTagList' and -1, if 'GetCyberMapAttr' should be
                  used.
```

```
                move.l  _RPort,a0               ;evalute rastport address
                move.l  rp_BitMap(a0),a0        ;address of the bitmap structure
                tst.b   Mode2                   ;which mode is enabled?
                bne.b   .nomode2                ;GetCyberMapAttr -> jump
                lea     CyberLBTLTags,a1        ;pointer to taglist for LBTL
                CALLCYBERGFX    LockBitmapTagList ;lock bitmap
                move.l  LBMI_Addr,d3            ;evaluate address of bitmap
                move.l  d3,BitMapAddr
                move.l  d0,a0
                CALLCYBERGFX    UnLockBitmap    ;unlock bitmap
                bra.b   .mode2
.nomode2
                move.l  #CYBRMATTR_DISPADR,d0
                CALLCYBERGFX    GetCyberMapAttr ;evaluate address of bitmap
                move.l  d0,d3
.mode2
                move.l  _RPort,a0               ;get RastPort address
                move.l  rp_BitMap(a0),a0        ;Bitmap-structure address
                move.l  #CYBRMATTR_XMOD,d0
                CALLCYBERGFX    GetCyberMapAttr ;find out Bitmap width
                move    d0,BitmapWidth          ;and store it
                move    AreaHeight,d1           ;find out height of image buffer
                clr     ActualOffset            ;vert. position of 1st buffer
                move    d1,HiddenOffset         ;vert. position of 2nd buffer
                move    d1,d2
                add     d1,d1
                move    d1,ThirdOffset          ;vert. position of 3rd buffer
                mulu    d0,d2
```

```
                move.l  d3,ActualBitmap         ;address of 1st buffer
                add.l   d2,d3
                move.l  d3,HiddenBitmap         ;address of 2nd buffer
                move.l  d3,ChunkyBuffer         ;2nd buffer used first
                add.l   d2,d3
                move.l  d3,ThirdBitmap          ;address of 3rd buffer
```

The taglist, which is passed to the function 'LockBitMapTagList', can look like this:

```
CyberLBTLTags   dc.l    LBMI_BASEADDRESS
                dc.l    LBMI_Addr               ;-> free space for bitmap address
                dc.l    TAG_DONE
LBMI_Addr       dc.l    0                       ;this is the place for the addr.
```

The voxelspace demo assumes that the new screen is in the foreground when 'LockBitMapTagList' is called, so the bitmap address points to the graphics RAM, not to a backup buffer. This should always be the case, because the screen was just opened a short time ago.

During the main loop the program should test after each 'locking' if the returned bitmap address is identical to the one evaluated at startup. If, for example, the screens are switched, the bitmap is copied to the FAST-RAM. A game/demo should then usually enter waiting state until the screen has been switched to the foreground again.

The main loop then roughly follows this schematic:

1. Lock bitmap using 'LockBitmapTagList'
2. Compute image in the buffer that 'ChunkyBuffer' points to
3. Create additional elements in the invisible buffer using graphics.library functions.
4. Unlock bitmap using 'UnLockBitmap'
5. Rotate image buffer using 'graphics/ScrollVPort'
6. Rotate necessary pointers and offsets as well

1. First the bitmap should be locked using 'LockBitMapTagList'. If this isn't done, the game/demo will still work but it isn't legal anymore from the view of CyberGFX. A game/demo should maybe offer the possibility not to lock/unlock the bitmap as an option, in the case that the locking mechanisms produce problems.

   In the following example it is also tested, if the bitmap address returned is identical to the one evaluated at startup. In this case the voxelspace demo enters a waiting mode and tests periodically if the screen is at the foreground again.

   Note: The voxeldemo hangs up itself in the following function if the locking mechanims don't work. A game/demo should offer a possiblity to leave the program with an error message instead.

```
                tst.b   Mode2
                bne.b   .noLBTL
.retry
```

```
                move.l  _RPort,a0               ;evaluate rastport address
                move.l  rp_BitMap(a0),a0        ;address of bitmap structure
                lea     CyberLBTLTags,a1        ;pointer to the taglist for LBTL
                CALLCYBERGFX    LockBitmapTagList ;lock bitmap
                move.l  d0,d2
                beq.b   .delay                  ;if error, then jump
                move.l  LBMI_Addr,d0            ;evaluate bitmap address
                move.l  BitMapAddr,d1           ;get original bitmap address
                cmp.l   d1,d0                   ;compare addresses
                beq.b   .noLBTL                 ;if equal than proceed
                move.l  d2,a0
                CALLCYBERGFX    UnLockBitmap    ;unlock bitmap
.delay
                moveq   #5,d1
                CALLDOS Delay                   ;wait a bit
                bra.b   .retry                  ;restart the procedure
.noLBTL
```

2. This item of course depends on the game being developed. Take care that the modulo-value is taken into account.

3. The TurboGFX-mode allows using standard-functions of graphics.library again. You must pay attention to adapting the vertical position in a way that ensures that the correct image buffer is selected. The vertical position 0 is always the first row of the first image buffer.
   Any such vertical shift is easiest achieved by adding the 'HiddenOffset' value.

4. After all accesses to the bitmap are completed, the bitmap has to be unlocked if it was really locked previously . The following code assumes that the handle returned by 'LockBitmapTagList' is situated in d2.

```
                tst.b   Mode2
                bne.b   .noLBTL2
                move.l  d2,a0
                CALLCYBERGFX    UnLockBitmap
.noLBTL2
```

5. The switching of the image buffers is done as follows:

```
                move.l  _VPort,a0               ;get ViewPort address
                move    HiddenOffset,d0         ;negate vert. Offset of
                neg     d0                      ;the 2nd image buffer
                move    d0,vp_DyOffset(a0)      ;and enter it into the Viewport
                CALLGRAF        ScrollVPort     ;switch image buffers
```

6. Only the necessary pointers and offsets remain to be rotated so that the next image will be created in the correct buffer:

```
move.l  ActualBitmap,d0
move.l  HiddenBitmap,ActualBitmap
move.l  ThirdBitmap,HiddenBitmap
move.l  d0,ThirdBitmap
move.l  HiddenBitmap,ChunkyBuffer
move    ActualOffset,d0
move    HiddenOffset,ActualOffset
move    ThirdOffset,HiddenOffset
move    d0,ThirdOffset
```

1.33 Optimizing

If the TurobGFX-technique is used, a couple of additional things must be considered in order to reach optimum performance. The access to the RAM of the graphics board will usually be done through the Zorro3-bus (Zorro2 in earlier AMIGA-models). Exactly this access of the processor via the Z3-bus to the graphics board is significantly slower than the access of the processor to the Fast-RAM which is often located on the processor-board itself.

In addition, the Fast-RAM is usually accessed in copyback-mode while the graphics-board RAM is always accessed 'noncachable' - that is with the cache turned off.

This leads to the conclusion that it is very important whether the graphics are written into the RAM in byte- or in longword-increments. In the former case the performance may drop by a significant amount.

As a more 'hands-on' example let me mention the voxelspace-algorithm here. This algorithm projects the landscape-data onto the screen in strips. The code writes vertical columns into the graphics-RAM from left to right. If these columns are 4 pixels wide, this results in longword-accesses which are optimal. In case of 2- or even only 1-pixel columns these accesses are not optimal at all anymore.

This access can still be optimized, though. In the voxelspace-demo in case of 1-pixel columns groups of 4 strips are formed and created in a Fast-RAM-buffer. After that this buffer is copied into graphics-RAM in longword-units and then the next 4 strips are created. As this buffer is relatively small, these accesses still profit from the processors data cache. The byte-accesses to Fast-RAM are optimized by the copyback-mode of the cache.

When using the TurboGFX-technique you should always make sure to access the graphics-RAM in longword-units. 'Detours' through Fast-RAM can often lead to tremendous performance improvements.

1.34 Problems

At this point I want to point out a well-known problem with CyberGfx+/TurboGFX.

It may happen that pressing mouse buttons can lead to the machine hanging or even crashing. In this case it is recommended to de-activate all commodity-programs running in the system (if you know which program causes the problem you can of course simply deactivate that one). An example for these kind of programs are screen-blankers.

1.35 Interaction

The essence of every game is the interaction with the player. The player uses the keyboard, mouse or joystick to direct the game in the way he desires.

The following section covers how to evaluate user-input in a system-compliant way. Joystick-programming is not covered here as I have never used it myself. In case joystick-input is desired, you should resort to the documentation of the 'gameport.device' which is used for that purpose.

Generally the user input occurs in the active window. The operating system notes this input and sends messages to the program that allow it to evaluate what kind of input occurred. The game must decide which kinds of input to evaluate when 'Opening the window', this is done by specifying the appropriate IDCMP-flags in the window-taglist.

The main loop of the game will then look similar to this:

1. Get window-message
2. Evaluate window-message and decide which actions to take (if any)
3. Answer window-message
4. Execute actions
5. Compute and display image

1. Get window-message

A message is read using the 'exec/GetMsg' system function. It expects a message-port as parameter. The UserPort of the window is passed. This then looks as follows:

```
move.l  _Window,a0              ;get window-address
move.l  wd_UserPort(a0),a0      ;find out UserPort-address
CALLEXEC        GetMsg          ;get message
tst.l   d0                      ;is there a message?
beq.w   .loop                   ;no -> no evaluation
move.l  d0,d4                   ;save message for ReplyMsg
move.l  d0,a0                   ;put message into a0 for  ↩
    evaluation
```

2. Evaluate window-message

A window-message has a defined structure (which can be found in the 'IntuiMessage' structure in the 'intuition/intuition.i' include-file). Some elements of this structure can only be used for evaluation. First of all the 'im_Class' field which classifies the type of input is evaluated.

After getting the message all interesting elements are read:

```
                move.l  im_Class(a0),d0         ;get message-class
                move    im_Code(a0),d1          ;get message-subclass
                move    im_MouseX(a0),d2        ;get mouse-delta-position
                move    im_MouseY(a0),d3        ;and for Y-direction
```

Now the message-class is checked (the possible values correspond to the IDCMP-flags that were specified when opening the window). Example:

```
                cmp.l   #IDCMP_MOUSEBUTTONS,d0  ;mouse buttons pressed?
                beq.b   .checkmouse
                cmp.l   #IDCMP_RAWKEY,d0        ;key pressed?
                beq.w   .checkrawkey
                cmp.l   #IDCMP_MOUSEMOVE,d0     ;mouse was moved?
                beq.w   .checkdeltamove
                cmp.l   #IDCMP_ACTIVEWINDOW,d0  ;window activated?
                beq.b   .activewindow
                cmp.l   #IDCMP_INACTIVEWINDOW,d0 ;window deactivated?
                beq.b   .inactivewindow
                bra.w   .reply                  ;else reply to message
```

Depending on the message-class further information is now extracted. This is most often done in the 'im_Code' filed that now is located in d1:

IDCMP-Flag MOUSEBUTTONS:

```
                cmp     #IECODE_LBUTTON,d1      ;left mouse button pressed?
                beq.b   .leftdown
                cmp     #IECODE_LBUTTON+IECODE_UP_PREFIX,d1 ;released?
                beq.b   .leftup
                cmp     #IECODE_RBUTTON,d1      ;right mouse button pressed?
                beq.b   .rightdown
                cmp     #IECODE_RBUTTON+IECODE_UP_PREFIX,d1 ;released?
                beq.b   .rightup
                bra.w   .reply
```

IDCMP-Flag RAWKEY:

The 'im_Code' field contains the key-code of the key that was pressed. This code is NOT identical to the ASCII-code (the IDCMP_VANILLAKEY IDCMP-flag must be used to do this). The codes must be taken from a table or be determined by trial&error.

Bit 7 of the key-code specifies whether the key was pressed or released. This can be tested using the IECODEF_UP_PREFIX tag.

Example:

```
                cmp     #$45,d1                 ;ESC pressed?
                beq.w   .esc
                btst    #IECODEB_UP_PREFIX,d1
                bne.w   .keyup
                cmp.b   #$50,d1                 ;F1 pressed?
                beq.w   .F1pressed
                cmp.b   #$51,d1                 ;F2 pressed?
```

```
                beq.w   .F2pressed
                ...
                bra.w   .reply
.keyup
                bclr    #IECODEB_UP_PREFIX,d1
                cmp.b   #$55,d1                 ;F6 released?
                beq.w   .F6released
                cmp.b   #$56,d1                 ;F7 released?
                beq.w   .F7released
                ...
                bra.w   .reply
```

IDCMP-Flag MOUSEMOVE:

This IDCMP-flag should always be used together with IDCMP_DELTAMOVE. The 'im_MouseX' and 'im_MouseY' fields then contain the number of units the mouse was moved by. The scale of these values must be determined simply by trying out. Very often these values are scaled for further use in the game.

ACTIVEWINDOW and INACTIVEWINDOW IDCMP_Flags:

No further information can be gained for these flags.

3. Reply to window-message

After evaluating the message it must be replied to. This is done through the 'ReplyMsg' exec-function:

```
                move.l  d4,a1                   ;put message into a1
                CALLEXEC        ReplyMsg        ;reply to message
```

The items 4 and 5 are program-specific and do not belong to the interaction process as such.

1.36 RAM is slow

In the past years CPU power has risen almost exponentially. On the contrary, RAM-chips of the kind used in conventional computers have only improved a little bit. The access to RAM has become more and more of a bottleneck.

This bottleneck is successfully being fought using ever-increasing cache memories. The caches have become an important performance factor for conventional applications.

But games are no conventional applications. Games are very memory-intensive as they have to cope with larger and larger amounts of data. Caches are often even counter-productive in certain areas. For this reason it is important to pay attention to minimizing memory access during the development process.

In this context some programming philosophies as they are know from older processors must be turned upside down. When programming the 68000, games were optimized by doing as many calculations as possible in advance and storing the results in tables from where they were read in the realtime-part of the

program.

As a lot of games were developed using this philosophy, many of them did not experience the expected performance increase when running on a faster processor. Even the very fastest processor accesses RAM only a little faster than a conventional 68000.

Modern processors can execute 50 to 100 or even more commands in the same time they need to do a memory access. This leads to the conclusion that it is often faster to compute data in realtime than to read it from a table. On top of that, large tables are very cache-unfriendly. They lead to a large efficiency-loss of the cache and therefore decrease performance a lot.

The voxelspace-demo puts this new philosophy into action. Many voxelspace-algorithms described in scientific journals were optimized for older processors by creating many structures in advance. The voxelspace-demo does nearly all of these calculations in realtime. The main loop contains only one memory access: the reading of the height/color data of the landscape. Of course all accesses for creating the graphics come on top of that. These kinds of algorithms that almost entirely consists of calculation commands also further the pipelining and thus increase the throughput of the commands.

It is of decisive importance for the programming of PPC-games to optimize the algorithms for a minimum of memory accesses in order to utilize the full power of these processors.

1.37 MMU and Cache

This section contains information on how the cache and the MMU can be used to achieve significant performance improvements.

The caches are most efficient if the same memory areas are accessed very often. In that case time-consuming memory-accesses can be avoided. The best example for this is the processor-stack.

When it comes to game programsb it is often the case that huge amounts of data must be accessed but only rarely the same memory area is accessed several times. In these cases the caches actually become counter-productive and slow the game down. Caches are managed in segments of 32 bytes internally. As soon as a data element is accessed one such cacheline is loaded into the cache as a whole. This memory access takes as long as 8 conventional memory accesses (this applies to processors with a 32-bit bus). This extra time is usually compensated for because all following accesses to this cacheline only have to be done on the cache.

For games that only use such data elements once this leads to memory access becoming slower than if the data cache is turned off.

Now we have a problem: if the data cache is switched off globally, you have taken care of the above problem but now you are losing performance because access to those areas that could take advantage of the cache is now slowed down. The only solution is the creation of an optimized MMU-setup that assigns the separate memory areas different cache-modes. Unfortunately AMIGA-OS offers no MMU-support at all. The only solution is to use ’evil hacks’, which means direct access to the hardware - something that actually shouldn’t be done

anymore.

When employing the PowerPC and the WarpOS operating systems this is different. WarpOS offers applications the option of allocating memory areas with a certain cache-mode. This offer a system-compliant way to use the MMU and the cache in a most optimal way.

The 'AllocVecPPC' function of the powerpc.library supports additional memory attributes that can be used to specify the desired cache-mode. Games can now simply allocate a memory area and mark it as 'noncachable' (using the MEMF_NOCACHE attribute). For a detailed description of this function I recommend the 'WarpOS.guide' and 'powerpc.doc' documents.

Here a futher note to this problem: The local disable of the data cache can accelerate a program or it can even brake it down, dependant on the processor. The greater the data cache and the faster the memory access the greater the probability that a program is braked down. So it should be always tested on several machines and eventually there should be several versions for different processors. The voxelspace demo enables the local disable of the data cache for the 603E, but enables it on 604E (with CyberStormPPC) because the 604E was indeed braked down.

I now want to point out a further problem. The MMU has the task of doing address translations. It looks up in a table which logical address matches which physical address and then performs the translation. On every 68K-processor that has an MMU this is done automatically in hardware. This process is called the tablesearch or tablewalk.

Not all of the PowerPC-processors support the hardware-tablesearch. The PPC603 and the PPC603E do not know any hardware tablesearch. On these processors the tablesearch is done in software. Such a software-tablesearch is of course extremely slow. However, this is not relevant most of the time as a 'MMU-cache' takes care that these tablesearches occur seldom enough to not pose a performance-problem.

Again, this can be completely different for games. Let's take a look at 'voxelspace', for example: The voxelspace-algorithm is based on a geographic map that is 2 MByte in size. The algorithm now uses a certain method to read the map data. In this process it traverses an extremely large address space. This results (similar to the effect on the cache) in the efficiency of the MMU-cache dropping close to zero. As a result a tablesearch has to be done for almost every memory access which can lead to the game stalling.

This effect can even be observed on systems that have hardware-supported table search, but the effects are not very severe in that case.

Again, there is no system-compliant solution to this problem on 68K-systems. The voxelspace-demo offers an additional option for 68K-systems which must be considered a 'hack' and is therefore not entirely system-compliant. This hack sets up the MMU using the transparent translation registers and in this way keeps tablesearches from occurring. This can result in as much as 40-50 percent performance increase.

On the PPC, WarpOS offers a system-compliant way to solve this problem. WarpOS supports the so-called BAT registers that work similar to the transparent translation registers. When running WarpOS, each PPC-task has the option of filling the 4 BAT-registers with a memory-area of your choice. There are 4

BAT-registers available, one of which will most often be used by the operating system to cover the graphics-RAM. By employing the BAT-registers, tablesearches for the specified memory areas can be avoided.

The 'AllocVecPPC' function of the powerpc.library knows the MEMF_BAT memory attribute. If this is specified the allocated memory area will be controlled by a BAT-register.

The voxelspace uses this additional feature if the TURBOPPC option is switched on. It allocates the memory for the map as well the memory for the sky using the MEMF_NOCACHE and MEMF_BAT attributes.

In order to demonstrate the power of this feature, I want to point out the following:

The voxelspace-demo runs about twice as fast on a PPC603E/150 than on a 68060/50 if the standard options are used and both processors are driven in the most optimal way. In this case the 68060 does not run system-compliant.

If the 'MMU-hack' for the 68060 is switched off, the PowerPC runs as much as three times as fast. This means that games that are programmed in a system-compliant way can squeeze even more power out of the PowerPC.

1.38 Multiprocessing

To put first things first: this section does not explain how to use multiprocessing in order to increase performance but rather explains why multiprocessing can not be used to increase performance.

A dual-processor-board might give reason to hope to be able to run both processors at the same time in order to achieve a higher performance. The problem is: both processors use the same bus. During every memory access by one processor the bus is blocked for the other one. Algorithms that rely on a lot of memory accesses then result in the overall performance of both processors decreasing.

Now let's assume that both processors are executing algorithms that are not dependant on memory access - in this case a performance increase could actually be achieved. But especially in this case the PPC outperforms the 68K by such a large margin that it doesn't pay off anymore to run both CPUs in parallel. On top of that, pure computing-intensive algorithms are very rare. The mandelbrot-algorithm is an example for that. And even in the case of 'cybermand' parallel-processing does not lead to a noticeable performance increase.

Conclusion: when designing a game you must take care that it runs sequentially on both processors. You should also not use several tasks as this can result in similar problems.

1.39 Scheduling / Optimizing

Scheduling is the art of arranging commands in a way that optimally exploits the internal execution units of the processor. Scheduling has an increased importance for modern processors. Scheduling should not be overrated, however. Only applying the scheduling-rules systematically (as described in the PPC user-manuals, for example), can lead to a performance-increase. And that is a task for high-level programming language compilers.

Nevertheless, I want to point out some possibilities to optimally arrange commands. The voxelspace-demo contains such a case. The innermost main loop contains a floating-point division as well as a memory access. Both command have a long execution time, which means they are very slow. The performance can now be optimized by placing the floating-point command directly before the memory access (that is the FP-command followed by the memory access). Now the PPC can execute both commands in parallel and quite some time can be saved.

Avoiding dependencies is the most efficient scheduling method which can be implemented best. The powerpc works most efficiently if its pipelines are filled as good as possible. But if there are commands one after the other which need the result of the predecessor, such a command has to wait until the result is available. Clever placement of commands can result in much better performance in many cases.

As a general rule it is advisable to split algorithms among several execution units. Integer- and FPU-commands should be placed alternately so that they can be executed in parallel. This makes sense if the algorithm has at least a certain size. In the case of small algorithms you will face the problem that the floating-point<->integer conversion will eat up all the time gained by scheduling the commands.

When programming in assembly language you should therefore always try to place integer-, floating-point- and memory-access-commands alternately.

If a programmer has to choose if a program should be based on integer or FPU algorithms, the FPU version is mostly the more optimal one. The FPU of the PowerPC is extremely powerful and offers many powerful commands, such as the combined Multiply-Add/Sub commands.

But it has also to be considered that the PPC604E has several integer units. In this case, an integer algorithms is maybe faster than the corresponding FPU algorithm.

Here some hints on optimizing: thanks to the many registers of the PPC, memory accesses can often be avoided. Before executing the actual function, all constants and variable can be loaded into registers and all calculations in the main loop can be done on the registers. This of course requires a clean documentation as one still has to be able to determine which register contains what value/variable/constant.

1.40 Configurability

Now we leave the technical area and turn our attention towards game-design. First of all we will discuss the configurability of games.

Games often gain a lot of attractiveness if they have a lot of options and

switches that allow playing the game from a different perspective or under different conditions. This includes the screen resolution which should be freely selectable by the player (if a graphics card is present in the system).

The control should also be as flexible as possible. Several types of control should be supported as well as parameters that define speed, inertia, etc.

The keyboard map should be as freely selectable as possible as every player has different preferences in this respect. Too many games got a bad review because of the simple reason that they had an absolutely unusable keyboard layout.

Very often technical parameters are among the favourite toys of the player. These include screen resolution, screen detail, window size, approximations, landscape parameters, displaying the framerate, to just name a few. The more of the these toys are available, the more interesting it gets to play the game several times.

As a general rule you should use your imagination to the fullest in order to enable the player to play the game in as many different ways as possible.

1.41 Control

Just too many promising games failed due to a miserable control. In most cases the control is hard-coded into the game and the player has no influence on it.

Generally speaking, your game should support as many control-modes as possible, e.g. joystick, mouse and keyboard. Every gamer has a preference for one these control modes.

The control-speed also takes a central role. Here some negative examples from well-known games:

A popular car-racing game used a system of time-limits, i.e. a certain distance must be covered within a certain time. The time expired absolutely, which is exactly as a real clock. If the game was played on a slower machine, the hardware was only able to display fewer frames in the same amount of time. Result: the car covered less distance in the same time and at the same speed. This is of course very bad as the user of slower machines are at an disadvantage and it is also very unfair in itself.

A popular 3D-shooting-game uses a well though-out mechanism for rotating the player if he presses the left and right cursor keys. The actual rotation speed was supposed to be constant, however - i.e. the player was supposed to be able to turn the same angle in the same time, regardless of the game speed. Result: in order to cover the same angle in the same amount of frames on slower machines, the angle hat to be increased dramatically. As a direct result, a short press on the cursor key sufficed to rotate the player by a half turn. This of course made any precise control virtually impossible and the game was not playable anymore on most systems (except when you artificially increased the game speed by choosing a smaller screen size/resolution).

These examples show how important it is that the control is adapted to the respective hardware. Positive examples exist as well, certain games allow setting parameters such as walking- and rotation speed and even the movement

inertia.

Another important thing is to make the control as precise as possible. Many games became un-playable simply because certain tasks that had to be accomplished could only be mastered through sheer luck due to the un-precise control of the game.

A game should also take into account the limited movement accuracy of the player. If e.g. collisions are to be detected, any such checks should not react immediately if only one pixel of an object touches one pixel of another object. A more generous check is needed here in order to avoid detecting collisions if actually none have occurred (if two cars touch this is quite different from when they crash head-on).

1.42 Difficulty

Gamers can be broken down into several categories: those who play a game every now and then, those who play more or less regularly and the professionals. At best, a game should be attractive for all of these potential customers and in order to accomplish this it needs several levels of difficulty.

A lot of games have difficulty levels - and almost none of them actually employ this option in a sensible way. If a popular action-game offers four levels of difficulty and calls the hardest one 'Maniac', then it is simply not right that a professional action gamer walks through half of the game on the first try and then quits the game himself in order to be able see something new the next time he plays it.

Absolute beginners are neglected in the same way as often. The most simple level of difficulty is very often still too hard for those that have never seen a game of the genre. Rule:

The programmer should choose the most simple level of difficulty so that he thinks even a toddler could manage it - after that the difficulty is halved.

For the most difficult level you should proceed analogously. The higher the difference, the better. Even top-profis will be offered an additional challenge.

'Simple' and 'difficult' are often interpreted in a completely wrong way. A game is often made more difficult my simply increasing the amount of luck needed to get past a certain point. Very often the strength of the player is simply reduced and this is then sold as a separate level of difficulty.

An easy level of difficulty should interpreted in that way that mistakes made by the player have less dire consequences. There is no sense in extremely reducing the number of enemies if an unskilled player keeps ramming the landscape and looses a life in the process. It would be more advisable to lessen the consequences of such collisions.

Different levels of difficulty should not only change the game parameters. They should also introduce new elements into the game in order for advanced gamers to be required to really think and change their strategy. If you use a lot of creativity on that area, you can greatly increase the persistant

attractivity of a game.

You should also check whether the owners of system with different speeds are at an advantage or disadvantage. This should never be the case and must be compensated for by employing appropriate measures.

1.43 The necessary change

More and more often you can read things such as 'over 33 levels', 'more than 100 tasks', etc. in advertisements or on game boxes.

It is actually very good if a game is large enough to keep the player entertained for a long time. Unfortunately it is most often the case that games developers shoot all of their powder in the first 10 percent of the game. During this time the player enjoys the game only to become completely disappointed once nothing new turns up and no change is provided anymore.

It is important to spread all elements evenly throughout the game. You should not introduce all elements right at the beginning of the game as very often the player will simply be overtaxed by this. It is better to confront the player with the aspects of the game step by step so that he has time to adapt to each new feature.

A game should also keep up the motivation of the player to continue playing. Special events (e.g. super-enemies) that really impress the player should occur, or bonus sequences - always different ones if possible. Also, it is often interesting to put a real "barrier" into the game that truly taxes the player (a task often accomplished by super-enemies). Defeating a strong enemy is an impressive event and poses a high motivating force as often several tries are needed until the player finally makes it. It also requires the player to concentrate during all parts of the game as he has to save resources for defeating the super-enemy.

Most of the super-enemies I have seen to this date were very unimaginative and easy to see through. Most of the time they can be defeated using a very primitive strategy. I would be best if in addition to innovative attack-strategies (through which the player will still see after three or four times), dexterity would also be required so that the player must muster the same concentration every time he challenges this particular enemy. By the way - it is always advisable to keep the challenging the players' dexterity as dexterity isn't something you'll be able to 'learn by heart'. Most of the games get boring as time passes because the player has 'seen everything'.

1.44 Playability / Fairness

Playability must be the one thing which many games lack most. A game is playable if it is actually fun to play. For this reason playability should be of high importance during games development.

One classic example of un-playable elements are the famous backside-attacks. These tactics are often used to artificially increase the level of difficulty. But especially these kinds of attacks have no effect in the long run except

making the game more boring as the player will soon know when to expect those unfair attacks.

Many games boast about the speed at which the objects move. All too often this results in a cool effect in the short run but this effect is not all so cool anymore after having seen it two or three times. What remains is most often a scene that has nothing to offer in terms of actual game action.

Another common error: a game is programmed in a way that makes it behave virtually the same every time it is run. Most often the player is forced to always take one path, to always fight the same enemies which always attack in the same way and which can, of course, always be defeated in the same way. In the long run this is nothing but highly uninteresting.

Modern games should give the player as much freedom of action as possible and also employ random elements to spice up things a little. Special care should be taken regarding random elements, however, because otherwise they will backfire and result in unfair situations occuring.

Games should keep offering dexterity-elements on a regular basis in order to force the player to concentrate. The goal of a game should be to capture the player and draw his undivided attention in order to create a thrilling atmosphere. As soon as the player is able to lean back and easily defeat the game 'from a distance', any motivation to play this game will soon be lost.

1.45 Demo-Versions

90 percent of all demo-versions I have seen and tested to this day were complete and utter trash.

It is hard to believe how much energy is lost simply because the presentation of a game was done with an un-satisfying demo version. Demo-versions should offer a first step into the game and also demonstrate the advantages of the game. Demo-versions are also often used as a measure whether a game will be bought or not. In times when well-know Amiga-magazines rate almost every game as 'good', this is likely to be the only chance to get an impression of the game.

These kinds of demos receive only a very short residence-permit on my hard drive:

- Demos in which I get killed after 10 seconds without ever knowing what just happened.
- Demos that within the first 2 seconds throw tons of objects at me that want to finish me off.
- Demos that want to present all elements of the full game at one time and completely overtax me in the process.
- Demos that have a bad, un-precise control
- Demos that have no (reasonable) documentation
- Demos that don't run or run unstable

This at the same times says a lot about how demos should look. Demos are often confused with prototypes. A prototype may violate all of the above items. But it is not meant for the public. The potential customers deserve better.

Demos should fulfil a tutorial-function. They should slowly introduce the

player to the game an present some of its elements as well as confront the player with tasks he can handle.

Demos should offer enough playing time for the player to be able to get an idea of the the game.

It is important to invest a sufficient amount of time in creating demo-versions. This time will pay for itself as a good demo-version motivates the potential customer into buying the game.

1.46 Thoughts on 3D

I want to take the opportunity to utter my thoughts on the 3D-topic. 3D-games have been experiencing a boom for quite some time already since the hardware of competing systems could reach the necessary speed. For a very long time, not a single game of this kind could be found on the AMIGA.

Nowadays the technology of 3D-games on competing systems has already advanced very far. And only now old 3D-games are slowly being copied on the AMIGA.

Most of the 3D-games for the AMIGA still use the antiquated floor-wall-technique which severely hampers the players' freedom of movement. On competing systems complete freedom of movement has been the standard for quite some time now.

At his place I want to summon everybody familiar with 3D-techniques to not simply keep the delay on the competing systems by simply pulling old source codes over to the AMIGA without any creativity but rather use the newest techniques so that this delay simply won't exist for much longer.

The PowerPC-processor offers us the opportunity to achieve the same speeds as competing systems, therefore we should also be using the same new techniques. For this reason all 3D-specialists must join forces in order to define new 3D-techniques the target of which must me to top the competition. Today the games of tomorrow should be created - not those of yesterday!

1.47 Address of the author

I have invested a lot of time to put all thoughts and all know-how into this document. By doing so I want to show that it is about that time to make the technical know-how available to everyone in order for the AMIGA to be able to catch up with the competition again in a joint effort.

On the AMIGA-sector there is absolutely no sense in hiding technical know-how from the 'evil' competition. I want to remind everybody again: the technology is only a tool. A game is measured by its quality, not by the technology used to create it. Therefore it is my goal that the games developers allow full access to all technical matters and then again concentrate on the actual gameplay again - an area where they can still keep honing their business secrets.

I would be very happy to get in contact with people who aim at developing

unique and innovative games for the AMIGA. Through discussion and exchange of technical refinements the lead of competing systems could be diminished and the quality of games greatly enhanced as not so much time would have to be spent on technical details.

Anybody who wants to contact me can do so by one of the following ways:

regular mail:

HAAGE&PARTNER GmbH
Sam Jordan
Mainzer Straße 10A
D-61191 Rosbach
Germany

Tel: ++49/(0)6007/930050
Fax: ++49/(0)6007/7543

eMail:

s.jordan@haage-partner.com