3D GameStudio

Workshop
2D Animation

for A5 Engine 5.10
by Alain Brégeon September 2001

The latest news, demos, updates and tools, as well as the Users' Magazine, the Users' Forum and the annual Contest are available at the GameStudio main page http://www.3dgamestudio.com.

Contents

Foreword

Dear Reader,

I created this workshop in order to help you to find an answer to the question "How to make a **2D Animation** with 3D GameStudio?" The features used in this workshop are available with the version **4.25** or later.

This workshop, like others before is aimed at users who have some previous experience with 3D GameStudio. I assume that you have worked through the tutorials and know, how to use the tools (WED,MED and WDL).

This text improves the documentation which comes with 3DGameStudio, and cannot replace it. If something in this workshop is unclear to you, please read through the manuals enclosed. I apologize in advance for any unclear wording, faulty code, errors or omissions.

I hope you find the workshops informative and enjoyable.

Alain Brégeon
mailto:alainbregeon@hotmail.com

Get the latest version

Before you begin, please make sure to have the latest version of 3DGameStudio (**4.25** or later).

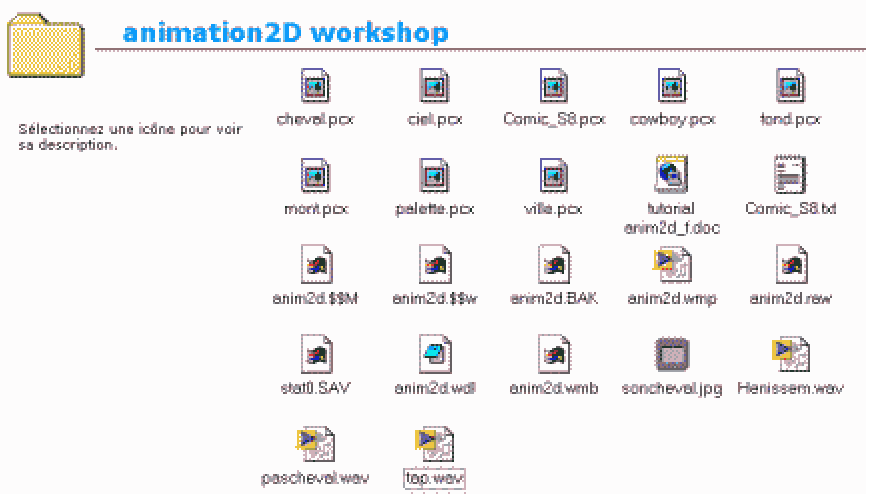
Prepare your workspace

Create a folder called "animation2d Workshop" in your GSTUDIO folder. This is the directory where you will store all the game elements.

The first thing that we are going to do, is to add the entities which we need for our game into the folder. If you do not have them yet, visit conitec's download site ((http://www.conitec.net / a4update.htm) and get them. Unzip the contents into your folder.

Your folder should now contain the following files:

```
cheval.pcx
ciel.pcx
comic_s8.pcx
fond.pcx
mont.pcx
palette.pcx
ville.pcx
tutorial.doc
comic_s8.txt
henissem.wav
pascheval.wav
tap.wav
```



animation2D workshop

Sélectionnez une icône pour voir sa description.

Folder Workshop 2d Animation

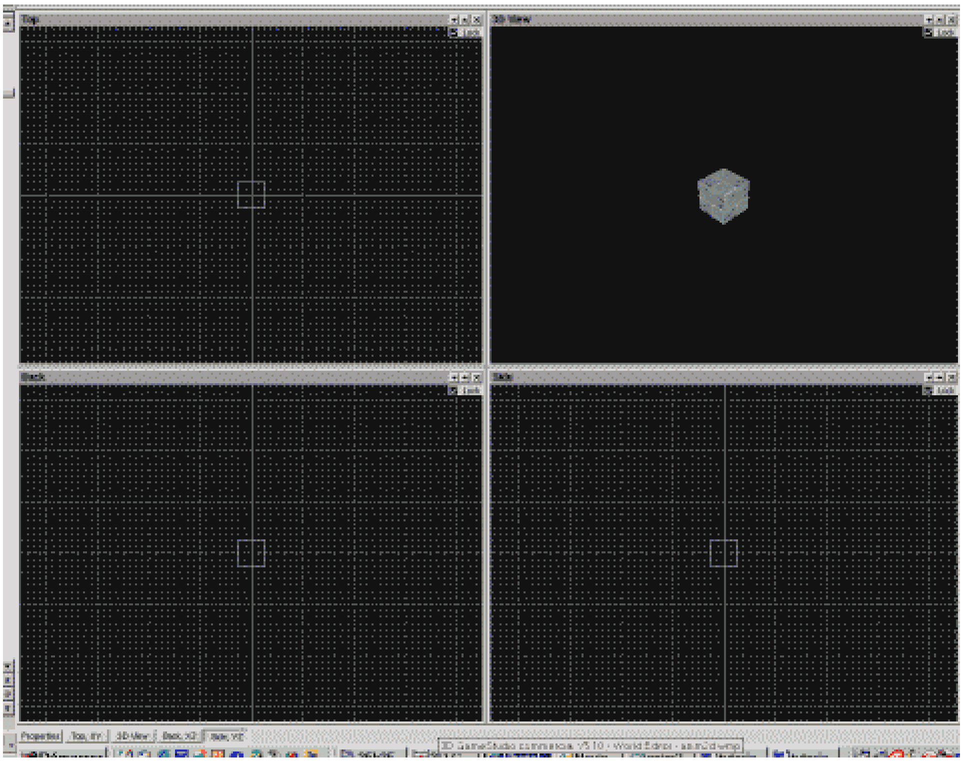
Create Your Level

Our level will be very simple: just empty space. But to be able to execute the engine at all, we need at least one dummy object. So let us create the following map:

Open **WED** and choose **File-> New**

Choose **Add Primitive - > Cube(small)**.

We apply the texture by default, we save the level and call it **`anim2d`**. Let us **build** and run it: nothing happens, that is normal.



The Level

Create your script

Create a script for your level. Just open your property window (**File > MapProperties**) and click the button **new**. The button next to it should change from `ndef` into `anim2d.wdl.`

Open your level folder, select and open (double-click) the file `anim2d.wld`. If Windows asks which application to use to open the file, choose "notepad" (or any other plain text editor).

You will notice that the standard game "Templates" have been created for you. This is fine for most projects but we want to do something more advanced this time. Go ahead and select everything (in Microsoft Notepad use **Edit->Select All**) and hit the delete key. Now you're starting from ground zero!

Adding Paths

Lets start by defining the paths our program is going to use. Paths are needed to tell the engine where it can locate the files used in our project (images, sounds, other scripts, etc.). The folder we are in ("Anim2d Workshop") is already included so, in our case, we only need to add the `template` folders.

Type in the following line:

```
path  "..\\template";    // Path to WDL templates subdirectory
```

Note that all paths are relative to our level folder. The line above says the following, "Go 'up' one directory ("...") and go into the template folder (`\\template`)".

Adding includes

After we set up our paths we should add our include files. Type in the following below **`path`**:

```
include <movement.wdl>;  // libraries of WDL functions
include <messages.wdl>;
include <menu.wdl>;      // menu must be included BEFORE doors and weapons
include <particle.wdl>; // remove when you need no particles
```

The **`include`** command tells the engine to replace this line with the contents of the file between the angular brackets (<...>). It is as if you went to the file in question and copied all the code from it and pasted it into you script.

This is a powerful tool because it allows us to reuse code from other projects and take advantage of updates in the included files without having to rewrite your code. For example, the 4.19 update included code to allow the player to swim in water. So now any project that includes **`movement.wdl`** can use this new swimming code.

Just like paths, the order of the **`include`** lines is important. Since some scripts use values that are defined in other scripts. For example: **`actors.wdl`** uses the variable **`force`** which is defined in the **`movement.wdl`** If we put **`actors.wdl`** before **`movement.wdl`** we would get errors.

It's okay to **`include`** scripts even if you don't use features from them in you code. Most of the files in the template are interdependent on each other so if you plan to use one of them, you should **`include`** all of them just to be safe. The exception to this rule is the **`venture.wdll`** script, which is not used by any of the other scripts, but uses many of them.

Starting Engine Values

Now we are going to set some important values that will help to determine how the simulator will be displayed. These values effect the resolution, color depth, frame rate, and lighting. Add the following lines beneath the **`include`** lines:

```
 // Starting engine values
ifdef lores;
var video_mode = 4; // 320x240
ifelse;
var video_mode = 6; // 640x480
endif;
var video_depth = 16; // D3D, 16 bit resolution
var fps_max = 50; // 50 fps max
```

Our variables of games

It is here that we shall enter our variables of games according to our needs

```
//our skills ****************************************************
```

Display the A4/A5 logo

We apply the display of the logo, which is located in the templates directory:

```
////////////////////////////////////////////////////////////////
// define a splash screen with the required A4/A5 logo
bmap splashmap = <logodark.bmp>; // the default A5 logo in templates
panel splashscreen { bmap = splashmap; flags = refresh,d3d; }
```

The Main function

In any project you need a **`main`** function. This is the very first function to be called when the program starts. In most cases the **`main`** function is very simple, our **`main`** is no exception. Please enter the following lines (below your last line):

```
function main()
{
   fps_max = 50;
   warn_level = 2;    // announce bad texture sizes and bad wdl code

// center the splash screen for non-640x480 resolutions
   splashscreen.pos_x = (screen_size.x - bmap_width(splashmap))/2;
   splashscreen.pos_y = (screen_size.y - bmap_height(splashmap))/2;
// set it visible
   splashscreen.visible = on;
// wait 3 frames (for triple buffering) until it is renderight and flipped to the foreground
   wait(3);

// now load the level
   load_level (<anim2d.wmb>);
// wait the requiright second, then switch the splashscreen off.
   waitt(16);
   splashscreen.visible = off;
   bmap_purge(splashmap);   // remove logo bitmap from video memory

// load some global variables, like sound volume
   load_status();

   game();
}
function game()
{
}
```

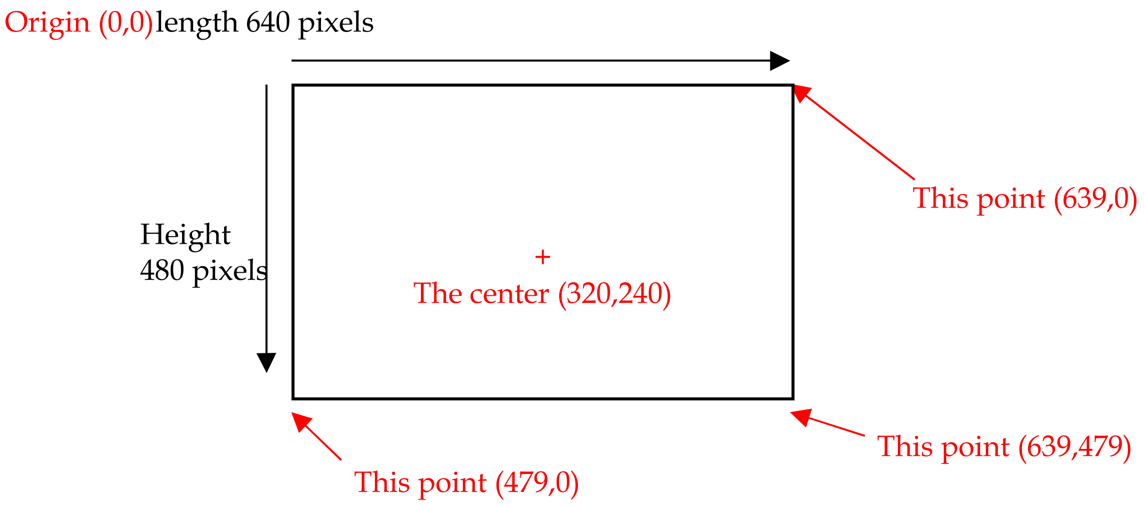
Each of these lines is commented, so there is no need of further explications.

The Basics of 2D

To take advantage of 2D, it is important to know some certain things which we are going to describe now:

Coordinates

The screen is represented by its dimensions in pixels, length and height, for example 640 x 480. The first charactetristic to be known is the position of the origin which is not at the bottom left, as one could expect but in the upper left corner. Every point can be described by (x,y) :



The Palette Adaption:

The explanation of this concept seems to be a little barbaric. First we open a wellknown picture, the loge of A5:

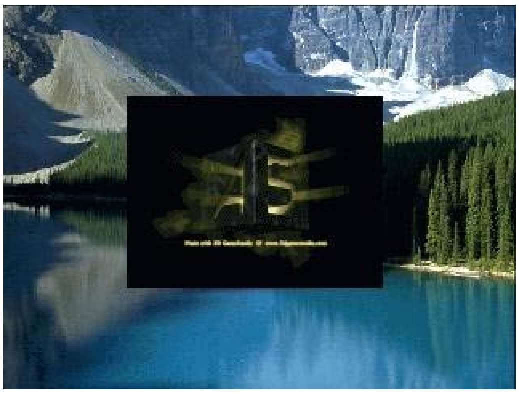
The A5 Logo

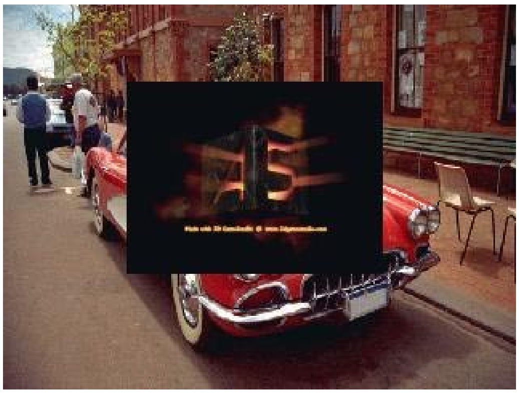
Then we take two beautiful pictures:

Idyllic Landscape

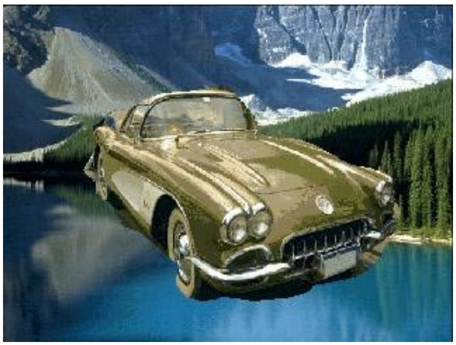
Nice Car, Isn't It?

And we copy / Paste our logo in each of them, that is what we get:


????????


...what In The Hell Does That Mean?

The logo does no longer look as splendid as it did before. Well, if you do not see a difference, just put the car into the picture with the idyllic lake and you will understand even better:


Looks Like Illegal Waste Disposal

Explications:
First you need to know, that this effect only turns up in 8 bit mode. But what in the hell is 8 bit mode? If you've got that, everything else becomes more clear.

In the 8 bit mode every point (pixel) of the picture is represented by a color number from a scale of values ranged in between 0 and 255 (in binary code 00000000 in 11111111, so it is 8 bits). Each of these values is equivalent to an index in a table which consists of the 3 values a color is made up of (Red, Green and Blue). Well, this table is the so called **palette**.

Let us take a little example by using the very first pixel point of our picture (for an easier understanding, we use the first 7 values of our palette):

Position	0	1	2	3	4	5	6	…	255
Color	▣	🟩	🟦	🟨	🟫	🟥	🟫		□

Position	0	1	2	3	4	5	6	…	255
Color	▣	🟥	🟨	🟩	🟦	🟪	🟦		□

My first point, the one at the upper left side of my picture, has the value 4, so it ist the color found at the 4th position of the palette.

First palette :

Our first pixel point has the color on position 4 of **this palette** and so it looks like this: 🟦

Second palette :

Our first pixel point has the color on position 4 of **that palette** and so it looks like this: 🟫

My picture has not changed, the value of it's very first pixel point still is 4. But what actually has changed is the graphical display of the values: in the first case the palette tells according to position 4, the color ist 🟦, while the second palette finds another color on this place and logical displays that one: 🟫 .

Note:

Working with this engine, the first color (**0**) of each palette should always be black (values R = 0, G = 0, B = 0) and the last one (**255**) ought to be white (values R =255, G = 255, B = 255).

Of course we work as much as possible with pictures of 16 or 32 bits but we want our game to be compatible to all environments and we know that on certain elder machines it is recommended to turn into the 8 bit mode.

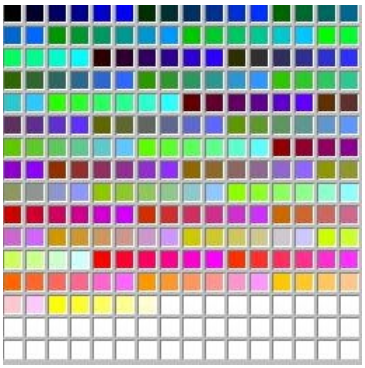
One understands at once that the car looses it's color by sticking it onto the picture with the lake, because there is no red at all within the lake palette. Here you are the examples of the two palettes for the car and the lake:

Palette Of The Lake

Palette Of The Car

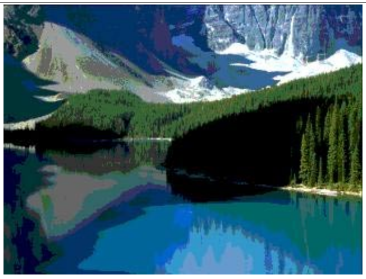
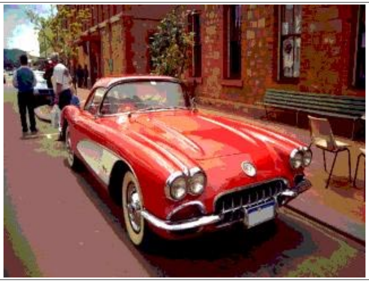
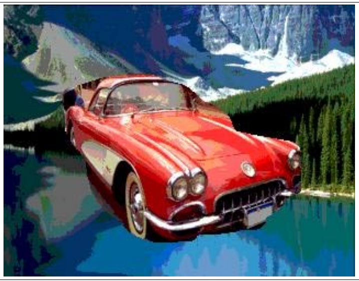
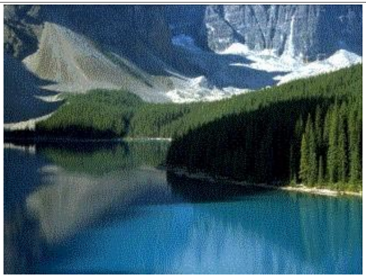
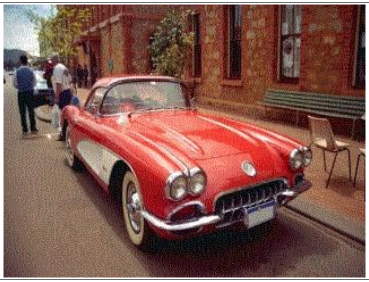
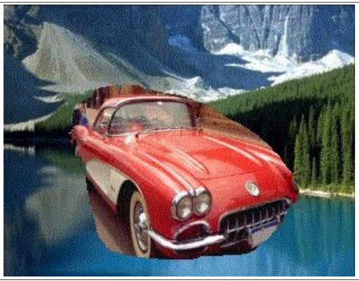
Then how to reconcile the incompatible?

The nearest solution is to work with the standard palettes of Windows, but what turns out is rarely brilliant. This is what you get using the so called palette 666:



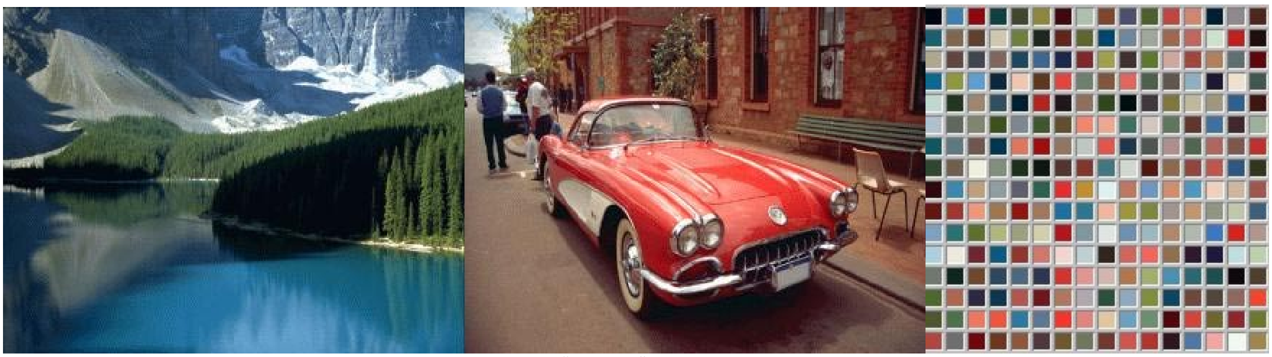
A Little Bit Of Everything

My paint program allows me to choose weather I want the “tramage” mode or not, here are the two results:

No tramage			
With tramage			

What In The Hell Ist Tramage???

Another solution is to unite the two pictures in 16 bit mode into one and then to convert the received picture into optimized 8 bit mode. This procedure checks all used colors and generates the optimum palette as a compromise. Here is the result:



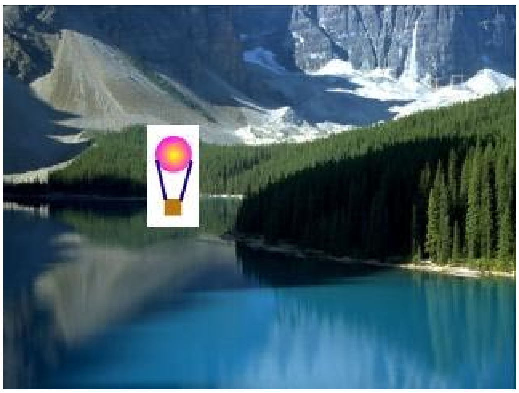
No “Tramage” No Worry…

The overlay:

Overlays are another very important term to know about in realizing 2 D games. The WDL manual explains as follows: “If this flag is set, the color 0 (not necessarily black) of an image will not be drawn...”

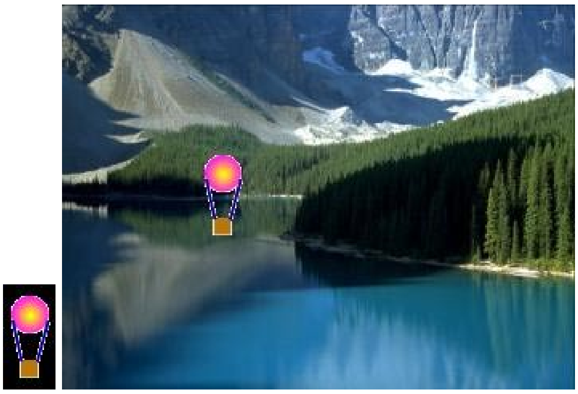
Actually I have a picture which I want to appear over my basic image. How about a hot-air balloon gliding across my lake?

Pasting this hot-air balloon, this is what I get:



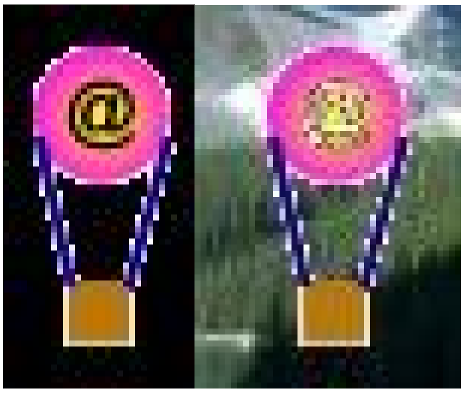
A Strange Rectangle Diturbing

This ist not what we intended, so we are going to use the overlay funktion. For that we need to replace surfaces we want to be hidden (here the white background) by the color number 0 (black in most of the cases) to be not displayed. Well this is the result:



Guess Why It Is Empty

It is much better.
But that's over as soon as there is something black on the balloon itself which we want to keep. Let us suppose an advertisment on it, it is easy to guess what comes out:

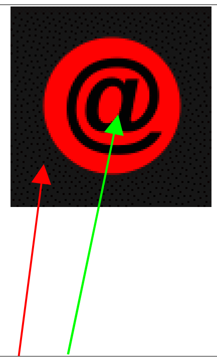
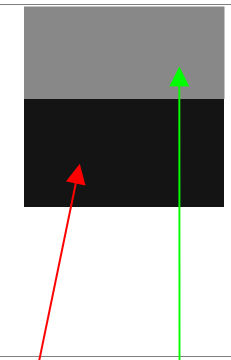
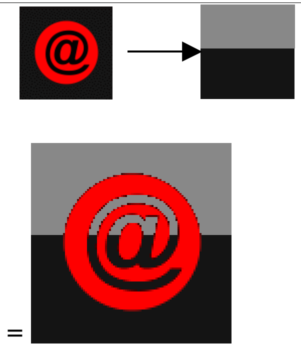
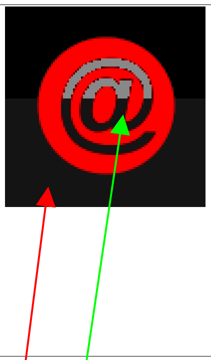


Where Have All The @'s Gone?

Well, the @ which was originally in black became transparent and dissapeared; isn't that terrible?

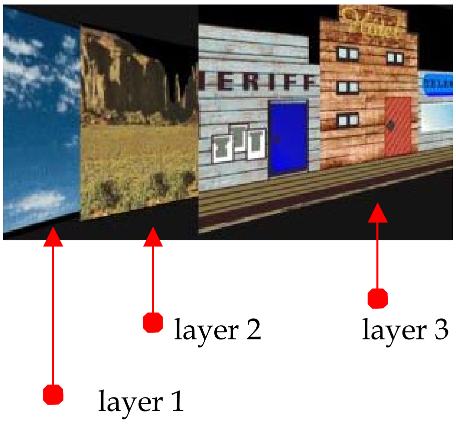
A little hint from a programming veteran:

In my paint program I put the picture of the hot-air balloon onto a background of dark grey. I have set the black color to transparent, so all black pixels become grey. Then I fill the outside with black, making it transparent again this way as you see in the pictures. I've made the dark grey brighter for better visibility.

			
The black is deep, dark, total black (r = 0, g = 0 et b = 0)	All the bottom is normally dark grey. That is that the black whom you see is not deep black but dark gray (r = 20, g = 20 et b = 20)	I ask for the transparency for my black, I see so my grey bottom (turned darker) by transparency. I merge both.	We put back of the black (0,0,0) Outside, the internal black being dark gray (20,20,20) But who sees the difference?

Layer

Layers determine the order of panels. Within this tutorial we are going to create a villige with a mountain behind it and a sky at the back. So there are 3 images to be stacked and each will have a layer number.



Well, enough of theory, let's pass on to practice.

Our goal ist to design an old fashioned western city and within we want a walking cowboy. Even though we're dealing with 2 D, we will try to achieve some 3D like effect for the whole thing.

For this we are going to use a trick they usual do in movies. We move our three layers in different speeds, thus giving an impression of depht.

Let us begin by positioning our sky:

First we open our file **`anim2d.wdl`** and define the basic panel. This panel is a bitmap of nothing but black and 640 x 480 pixels in size. We called it **`fond.pcx.`**

We define it as follows (to be insert into our variables):

```
bmap fond_map,<fond.pcx>;

panel fond_pan
{
   pos_x = 0; pos_y = 0;
   layer = 0;
   bmap = fond_map;
}
```

And into our function **`game`** we type :

```
fond_pan.visible = on;
```

If we run the level now, we should get a black screen after displaying the logo.

So use a little color to create the sky. It is a bitmap of 640 x 120 pixels in size and it is named **`ciel.pcx`**. The definition is as it follows (to be insert within the variables):

```
bmap ciel_map,<ciel.pcx>;

panel ciel_pan
{
   pos_x = 0; pos_y = 0;
   layer = 1;
   bmap = ciel_map;
   flags = refresh;
}
```

And at the end of our function **`game`** we type:

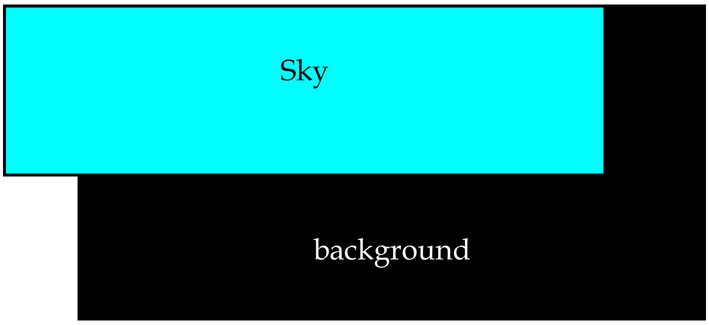
```
ciel_pan.visible = on;
```

Now, running the level, we should have a beautiful blue sky. It would be perfect to animate this sky, don't you think so? To do so, we are going to act by the following method:

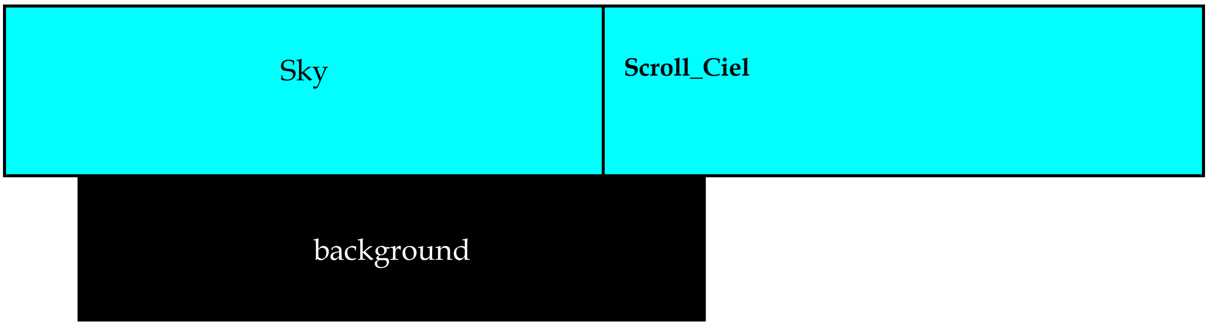
1 – we define a **`scroll_ciel_pan`** identical with **`ciel_pan:`**

```
panel scroll_ciel_pan
{
   pos_x = 0; pos_y = 0;
   layer = 1;
   bmap = ciel_map;
   flags = refresh;
}
```

2 - We show our sky at the position that corresponds to the value of the movement we wish to do.



3 – We show our **`scroll_ciel`** at (640 - the position) that corresponds to the value of the movement which we wish to do.



What results in:

```
var ciel_pos =0;//
```

... at the beginning of our variables.

Then at the end of the function **`game`** we type:

```
ciel_pan.visible = on;
scroll_ciel_pan.visible = on;
while(1)
{
     ciel_pan.pos_x = - ciel_pos;
     scroll_ciel_pan.pos_x = 640 - ciel_pos;
     avance_ciel();
     waitt (1);
}
```

And we type our function **`avance_ciel`**:

```
function avance_ciel()
{
      ciel_pos +=1;
      if (ciel_pos >= 640){ciel_pos = 0;}
}
```

Now let's turn to the mountain. It is made up of a bitmap of 640 x 200 pixels in size, named **`mont.pcx`**.

We define as follows (to be insert into our variables):

```
bmap mont_map,<mont.pcx>;

panel mont_pan
{
   pos_x = 0;pos_y = 35; // y = 35 c'est à dire que la montagne est plus basse que le ciel
      layer = 2;
   bmap mont_map;
   flags = overlay,refresh; //overlay pour rendre le noir transparent
}
```

And in our function **`game`**, before **`while(1)`** we type:

```
mont_pan.visible = on;
```

If we run the level now, we should have a mountain scenery underneath a blue sky. Not bad, how do you like it?

Let us prepare the scrolling of the mountain, as we did for the sky by adding the following instructions:

Write at the beginning of variables:

```
var mont_pos = 0;
```

Then below the definition of the panel **`mont_pan:`**

```
panel scroll_mont_pan
{
   pos_x = 0;pos_y = 35; // y = 35 c'est à dire que la montagne est plus basse que le ciel
      layer = 2;
   bmap mont_map;
   flags = overlay,refresh; //overlay pour rendre le noir transparent
}
```

and in **`game():`**

```
scroll_ mont_pan.visible = on;
```

Then within our loop **`while(1)`** we add (what is in red):

```
   while(1)
   {

      ciel_pan.pos_x = - ciel_pos;
      scroll_ciel_pan.pos_x = 640 - ciel_pos;
      mont_pan.pos_x = - mont_pos;
      scroll_mont_pan.pos_x = 640 - mont_pos;
      avance_ciel();
      deplace();
      wait (1);
   }
```

Then we write our new **`deplace`** function:

```
function deplace()
{
   if (key_cur == 1) //droite
   {
      ciel_pos += 1;
      if (ciel_pos >= 640){ciel_pos = 0;}
      mont_pos  += 3;
      if (mont_pos >= 640){mont_pos = 0;}

   }
   if (KEY_CUL == 1) //gauche
   {
      ciel_pos -= 1;
      if (ciel_pos <= 0){ciel_pos = 640;}
      mont_pos  -= 3;
      if (mont_pos <= 0){mont_pos = 640;}

   }
}
```

Now save and run the level. Please move around using the arrow keys. Rather nice, isn't it?

Let us turn to the remaining part, the city itself. Therefore we take the bitmap called **`ville.pcx`**. The rest should now be rouitine for you:

```
bmap ville_map = <ville.pcx>;
var ville_pos = 0;

panel ville_pan
{
   pos_x = 0;pos_y = 85; // y = 85 c'est à dire que la ville est encore plus basse
   layer = 3;
   bmap = ville_map;
   flags = overlay,refresh; //overlay pour rendre le noir transparent
}
```

And in our function **`game`**, before **`while(1)`** we type :

```
ville_pan.visible = on;
```

Then in our loop **`while(1)`** we add:

```
      ville_pan.pos_x = - ville_pos;
```

For the movement of our city we do not have to make a circular scrolling because we should be able to stop at every edge. So a single panel is just fine. By contrast we must stop the scrolling of the sky and mountain as soon as we are at each of the edges.

We add the following instuctions into our function **`deplace`** (red lines):

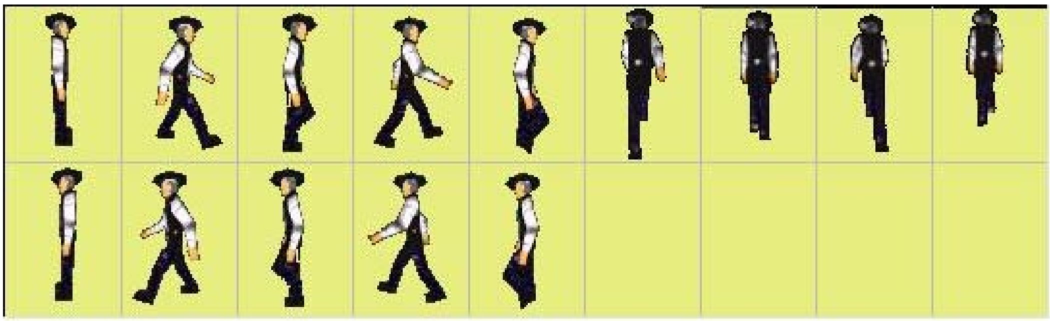
```
   if ((key_cur == 1) && (ville_pos < 1600)) //droite
   {
      ville_pos += 6;
- -  --  - - - -  -

   if ((key_cul == 1) && (ville_pos >134)) //gauche
   {
      ville_pos -= 6;
```

We can run our level and go for a walk through the citiy in every single direction we want to.

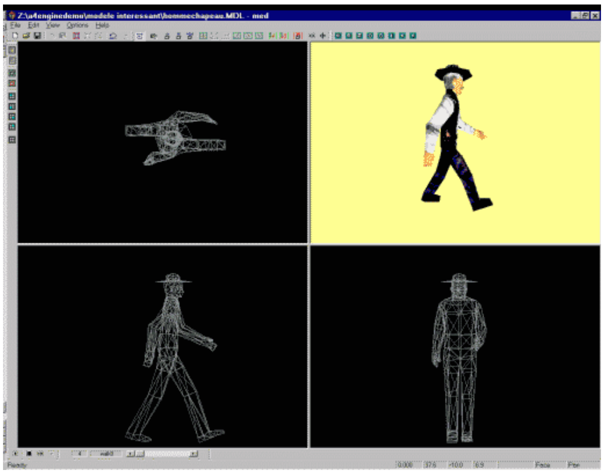
Displaying the cowboy

Let us think back to our theory. The display of the cowboy requires some explication. Actually the cowboy is an animation bitmap. Here you are the image that we are going to use:



For a better visibility, I took a yellow background here. But keep in mind, that it needs to be the color of the index “0” on your palette to be used as an overlay. (not necessarily, but in most cases black). If you don't use a palettetized image, but a true color picture, the background must be black (0,0,0).

The first question you'd probably ask ist how to animate a person in 3D? The answer is very simple: by using MED and making screen shots. Here is for instance the first walking stage:



Animate The Cowboy

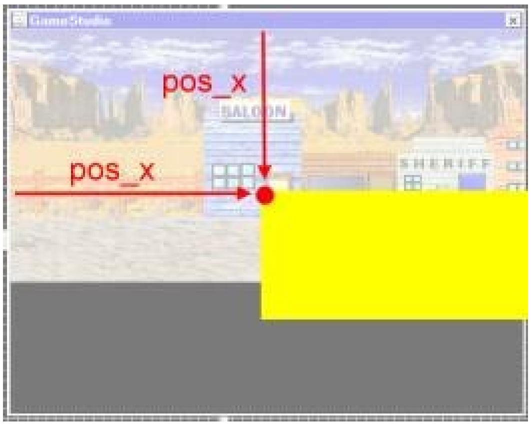
After a skilled cut and setting the scale, we obtain the final image.

We use the window element of the panel.

The instruction is written like:

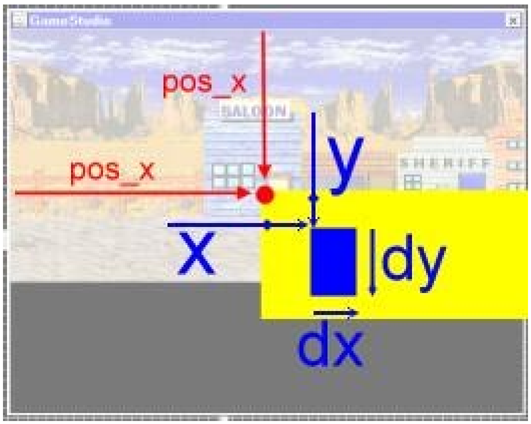
```
window = x,y,dx,dy,bmap,varx,vary;
```

At first we need to set our panel (the yellow here) like this:



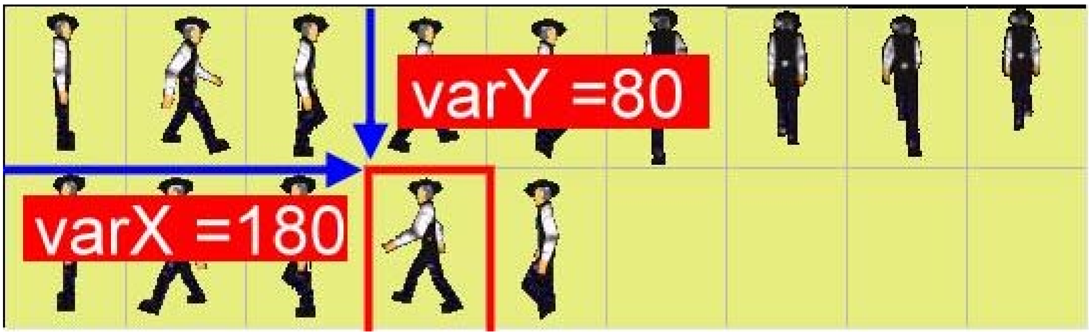
Where Is Our Cowboy Panel?

Then we place a window (blue here) at the position xy relative to the origin of our yellow panel (**`pos_x / pos_y`**) at width of dx and height of dy like this:



The Blue Peephole

Then we define our bitmap image and the coordinates of a point which allows us to cut an image of the size of our blue window (the example below shows our 4-th cowboy):



Cowboy Marches On

And this is how we write it:

within the variables...

```
var cowboy_frame_pos[2]=0,0;
```

The first value is our **`varX`** and takes the values 0, 60, 120, 180 and 240 for the walking on the side.

The second value is our **`varY`** and has the value 0 if one walks towards the right side and 80 if one walks to the left.

So we define our panel as follows:

```
bmap cowboy _map,<cowboy.pcx>;

panel cowboy _pan
{
   pos_x = 310;pos_y = 200;
   layer = 7;
   window = 0,0,60,80,cowboy_map,cowboy_frame_pos.x,cowboy_frame_pos.y;
   flags = d3d,overlay,refresh;
}
```

Of course, we find again our **pos_x** and **pos_y** values. These values correspond to the cowboy's starting position in front of the saloon door.

Why layer 7 while we only defined 4 (0 - 3)? Simply because we are farsighted like every good programmer is. It might be, that later on there are some more things in-between the houses and the cowboy.

Next we define our window (0, 0, 60, 80) which we want to be in the upper left corner of the panel where it's on and has 60 pixels in width and 80 pixels of height in size. **Cowboy_map**at least is the name of the picture from which we are going to pick up our frames.

Phew! This needed of an explanation.

There is nothing left to do, than to display the panel. This will be done by writing the following instruction at the beginning of **game**, before the **while(1):**

```
   cowboy_pan.visible = on;
```

And we manage the motions of the cowboy by the following lines (in red):

```
   if ((key_cur == 1) && (ville_pos < 1570)) //droite
   {
      cowboy_frame_pos.x +=60;cowboy_frame_pos.y = 0;
      if (cowboy_frame_pos.x >= 300){cowboy_frame_pos.x = 60;}
   - - - -  --  - - - - -

   if ((key_cul == 1) && (ville_pos >128)) //gauche
   {
      cowboy_frame_pos.x +=60;cowboy_frame_pos.y = 80;
      if (cowboy_frame_pos.x >= 300){cowboy_frame_pos.x = 60;}
```

Now we want to run the program and have a pleasant walk through our western city.

Unfortunately the pleasure soon will pass because of all these little flaws that catch our eyes.

Troubleshouting

First defect: if you stop the player while he is walking, he keeps his leg in the air and that does'nt look so perfect.

Second improvement to be made: it really would be much better if you needed to hit the arrow key only once to make the player goe to the regarding door. (You will have noticed already the goal is to enter buildings).

That is what we need to fix these problems: a variable for the directon, a step counter and a variable for the movements. So go ahead and type within the variables:

```
var waiting = 0;
var go_right = 1;
var go_left = 2;
var state = 0;

var look_right = 1;
var look_left = 2;
var look_at = 1;

var compteur_pas = 0;
```

And we modify our `deplace` routine in a noticeable way.

Here it is in it's entirety:

```
function deplace()
{
   if ((key_cul == 1) && (state == waiting))
   {
      state = go_left;
      compteur_pas = 20;
      if (key_ctrl == 1){ compteur_pas = 40;}
   }

   if ((key_cur == 1) && (state == waiting))
   {
      state = go_right;
      compteur_pas = 20;
      if (key_ctrl == 1){ compteur_pas = 40;}
}


   if ((state == go_right) && (look_at == look_left))
   {
      state = waiting;
      look_at = look_right;
      cowboy_frame_pos.x = 0;cowboy_frame_pos.y = 0;
   }

   if ((state == go_right) && (compteur_pas > 0))
   {
      look_at = look_right;
      if (ville_pos < 1570) //droite
      {
         cowboy_frame_pos.x +=60;cowboy_frame_pos.y = 0;
         if (cowboy_frame_pos.x >= 300){cowboy_frame_pos.x = 60;}
         ville_pos += 6;
         ciel_pos += 1;
         if (ciel_pos >= 640){ciel_pos = 0;}
         mont_pos  += 3;
         if (mont_pos >= 640){mont_pos = 0;}

         compteur_pas -=1;
         if (compteur_pas ==0)
         {
            state = waiting;
            cowboy_frame_pos.x = 0;cowboy_frame_pos.y = 0;
         }
      }
      else
      {
         compteur_pas = 0;
         state = waiting;
         cowboy_frame_pos.x = 0;cowboy_frame_pos.y = 0;
      }

   }

   if ((state == go_left) && (look_at == look_right))
   {
      state = waiting;
      look_at = look_left;
      cowboy_frame_pos.x = 0;cowboy_frame_pos.y = 80;
   }

   if ((state == go_left) && (compteur_pas > 0))
   {
      look_at = look_left;
      if (ville_pos >128) //gauche
      {
         cowboy_frame_pos.x +=60;cowboy_frame_pos.y = 80;
         if (cowboy_frame_pos.x >= 300){cowboy_frame_pos.x = 60;}
         ville_pos -= 6;
         ciel_pos -= 1;
         if (ciel_pos <= 0){ciel_pos = 640;}
         mont_pos  -= 3;
         if (mont_pos <= 0){mont_pos = 640;}

         compteur_pas -=1;
         if (compteur_pas ==0)
         {
            state = waiting;
            cowboy_frame_pos.x = 0;cowboy_frame_pos.y = 80;
         }
      }
      else
      {
         compteur_pas = 0;
         state = waiting;
         cowboy_frame_pos.x = 0;cowboy_frame_pos.y = 80;
      }
   }
}
```

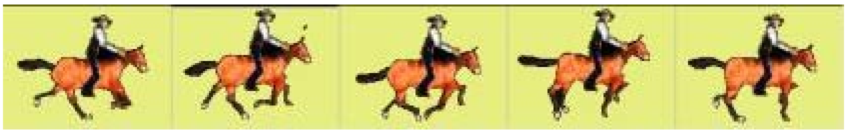
If you now run the level, you will notice that:

- A change of direction while stopping now simply changes the direction without moving forward.
- Assigning a direction by a key, our cowboy moves towards the next door in the given direction.

Not bad, but it would be goodto put some more life to whole thing, don't you think so?

The Horseman

We are going to make a rider appear who will cross the city at random. Here is the bitmap image (the background is usualy black and there are no separation lines):



To the usual question: "how to liven up a horse"? The answer is always the same: "Just use MED". But the difference was that I could not find any fully animated horse, so I had to create one by myself with all it's animation details. (Since this is not subject of this workshop, I refrain from explaining all the details here. Maybe you will find an explanation in a further mini tutorial or, what is more likely, in the AUM user topics).

For the animation of the horse we are going to use the same window element like we did with the cowboy. The main difference to the cowboy is, that the horse moves a lot more across the screen, which makes us to move the panel as well. How to do that?

We will do it in two stepts. First, let us animate the horse in the middle of the screen:

We create a variable:

```
var cheval_frame_pos = 0;
```

Then we define our panel:

```
bmap cheval_map,<cheval.pcx>;
panel cheval_pan
{
   pos_x = 400;pos_y = 200;
      layer = 8;
   window = 0,0,140,102,cheval_map,cheval_frame_pos.x,0;
         flags = d3d,overlay,refresh;
}
```

`Cheval_frame_pos.x` will have the values 0, 140, 280, 420 and 560. At the value of 700 it will be reset to 0.

All we have left to do now, is to display the panel. That is done by entering this intstruction at the beginning of **`game`**, before the **`while(1):`**

```
   cheval_pan.visible = on;
```

And the animation of the horse is managed by the following (red) lines:

```
function deplace()
{
         cheval_frame_pos.x +=140;
         if (cheval_frame_pos.x >= 700){cheval_frame_pos.x = 0;}

   if ((key_cul == 1) && (state == waiting))
```

Run the level and have a look. Please , do not forget to applaud.

Now we just have to move our horse and make sure that it turns up at random.

For testing purposes, we will assign the random function to a key (f.i. space):

```
on_space anim_cheval;

function anim_cheval
{
   cheval = 1;
}
```

and at the beginning of our variables we write:

```
var cheval = 0;
```

The difficulty is (a small one, I promise), that we do not move the horse relative to to the screen, but relative to our city which, because of the scrolling, is moved itself.

In in our variables we add:

```
var cheval_pos = 0;
```

And in our function **`game`** just before the **`call`** of the function **`deplace`** we type:

```
      cheval_pan.pos_x = cheval_pos - ville_pos;
      deplace();
```

And as we are impatient, we have a look right away to find out what happens. (Nothing happens as long as you don't hit the space key). Oh, isn't tha beautiful...

But what a frustration: what does really happen before and after the display? Does the horse still move on? Are we sure that everything works as we want? And what if you want to activate the random function (hit the space key) and there is no horse to come? Is this because of a bug in the random function - does it not return the value you're expecting? Then it is no more just frustration, it is a kind of metaphysical fear: “Am I good or am I not?”

Of course, we are equipped with debug functions but that is not the right thing to dissipate our anxieties. Wouldn't it be perfect if we could display the variables we wish to know about on our screen during the complete process? Sure, we could edit the function **`D`** and make it schow our variables but it is better to do it the other way (we are here to learn, aren't we?) And so we prepare the progress of the game, there are still things to write.

Displaying text requires a font. What a font is? It is the file which gives us the image of the character we want to print. We want to show an 'A' for examle and it appears on the screen by it's graphical representation:

A or A or E or A or A or **A** or A or A or A or A or A or A or ✌ or ✌ or B (why not)…

Since the code of characters is standardized (fortunately, otherwise we'd have anarchy), 'A' for example always has the value 64 in the ASCII standard (the one we use). The space, which is the first printable character, has the value 32. The graphical representation of the values 0, 48 etc. is up to you. If you want to, you can say 'if I ask to display character 64, draw me a little car'.

But before starting, let's take a little time to think and then ask the good questions.

- What do I need for the display? If it's only numbers, it will do to make a font which includes just 11 characters (space + 10 numbers). If I use the 7 bits ASCII table, I am going to make a font of 128 characters and using the full (8 bits) ASCII table, my font will have 256 characters.

- What is the desired size of the characters?

- Am I going to use an existing font or will I draw my very own characters?

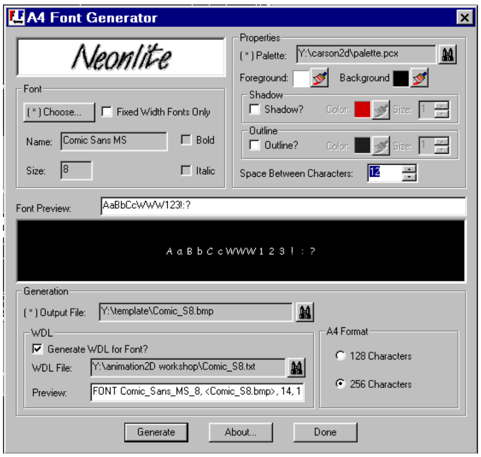
In our case, where we want to be international and therefore need to deal with the specifics of different languages, we should use the full ASCII, which is a font of 256 characters.

Please note that there is, even with identical fonts, a difference in appearance under Windows and with 3D GameStudio. Windows uses the proportional spacing (meaning that the actual used space varies according to the character, the style etc. so an 'i' takes less space than a 'w' f.i.) A4 / A5 however manages characters as components of fixed spacing.

To proof this, here you are the same line written in various fonts:

Font: arial corps 11	Abcdefghijkl WAIW	Proportional spacing
Font: agency corps 11	Abcdefghijkl WAIW	Proportional spacing
Font courrier corps 11	`Abcdefghijkl WAIW`	Fixed spacing

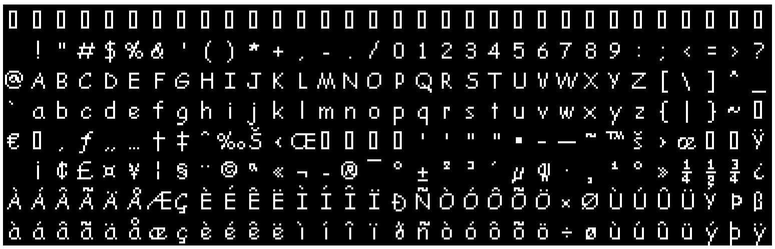
It is said, that the 3D GameStudio community is a great one. On the link page you will find an utility that allows you to generate your font from a Windows font. This is how it looks like:



(*) Palette: Y:\carson2d\palette.pcx
Foreground: Background:
(*) Choose... ☐ Fixed Width Fonts Only
Name: Comic Sans MS ☐ Bold
Size: 8 ☐ Italic
☐ Shadow? Color: Size: 1
☐ Outline? Color: Size: 1
Space Between Characters: 12
Font Preview: AaBbCcWWW123!:?
A a B b C c W W W 1 2 3 ! : ?
(*) Output File: Y:\template\Comic_S8.bmp
☑ Generate WDL for Font?
WDL File: Y:\animation2D workshop\Comic_S8.txt
Preview: FONT Comic_Sans_MS_8, <Comic_S8.bmp>, 14, 1
○ 128 Characters
● 256 Characters

Font Generator (don't Mix It Up With Babelfish)

And here is the result:

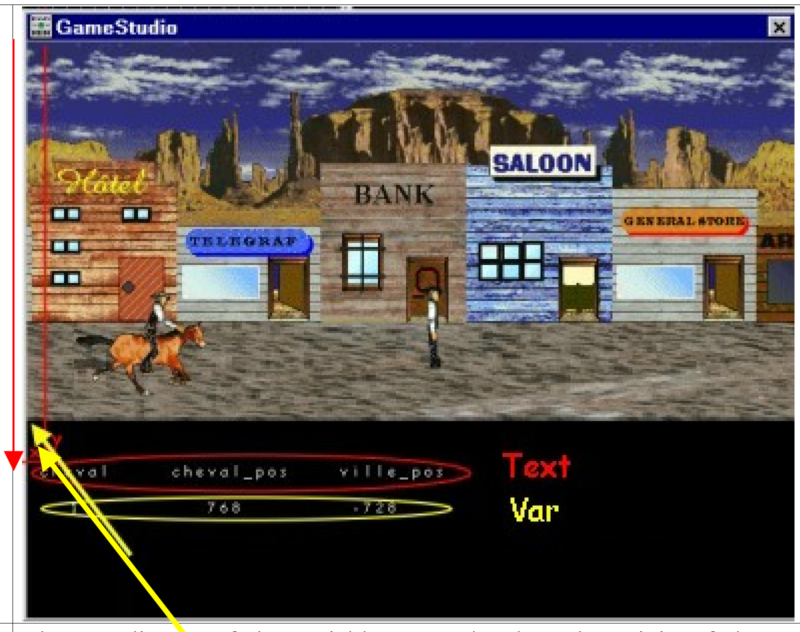


Completed Font (but How To Bring It Into An Intelligible Order?)

The last thing to do about it, is to define the font within our skript (you can copy the TXT file generated by A4 font generator). Please type at the beginning of our variables:

```
font comic, <comic_s8.bmp>,10,15;
```

This is the desired result:

The text coordinates are related to the window's origin (upper left corner).	 GameStudio Hôtel BANK SALOON TELEGRAF GENERAL STORE cheval cheval_pos ville_pos 1 768 -728 Text Var
	The coordinates of the variables are related to the origin of the panel they are displayed in.

To define our text, we write the following lines:

```
string mouchard = "cheval     cheval_pos    ville_pos";
text mouchard_txt
{
   pos_x = 10;pos_y = 350; // distance par rapport au 0,0 de l'écran
   layer = 10;
   font = comic;
   string = mouchard;
}
```

Then we provide the display of our variables:

```
panel affiche_var_pan
{
   pos_x = 0;pos_y = 380; // position de notre panneau par rapport au 0,0 de l'écran
      layer = 11;
   bmap fond_map; //on reprend notre fond noir d'origine
   digits = 35,0,1,comic,1,cheval;
   digits = 140,0,4,comic,1,cheval_pos;
   digits = 270,0,4,comic,1,ville_pan.pos_x;
         flags = overlay,refresh;
}
```

How is that to be read? Let's take the blue line as an example:

The instruction is: **`digits = x, y, len, font, factor, var;`**

Digits = 35, // place at a distance of 35 pixel away from the left edge of the panel **`affiche_var`**
➔ X (horizontal direction)

0, // place at a distance of 0 pixel from the upper edge of the panel **`affiche_var`**
➔ Y (vertical direction)

1, // number of characters to be displayed (our variable can take the values 0 or 1 , so

one character is enough ➔ **lengh**
comic, //use the font named `comic` ➔ **font**
1, // multiply the result by 1 ➔ **factor**
cheval ;// the contents of the variable called `cheval` ➔ **var**

The factor is very helpful, because `digits` displays the entire variable. If your variable currently is at 0,123, `digits` displays 0. Now you multiply the factor = 1000 and `digits` displays 123.

A hint: if you draw for example a little red circle instead of the 0 and a little green one instead of the 1 into your bitmap image, the value of the variable `cheval`won't be displayed as 0 or 1 any longer, but as a red or green circle.

There is nothing left to do than to release the display by hitting the key **[F12]** for example (so insert at the end of the skript):

```
on_f12 espion;

function espion()
{
   if (mouchard_txt.visible == off)
   {
         affiche_var_pan.visible = on;
         mouchard_txt.visible = on;
        }
        else
   {
         affiche_var_pan.visible = off;
         mouchard_txt.visible = off;
        }
}
```

We run the game and pressing **[F12]** the display appears. If you hit the **[space]**-key you notice, that the varible `horse` changes from `0` to `1` and horses position incremets (moving across the screen). If we start our cowboy to move, the city's position modifies as well. Watching this, we can note that everything is well-done.

Great, and now we want the horse not to start by pressing the **[space]**-key but at random.

We modify the beginning of our `deplace` function as follows (line in red):

```
   if (cheval == 1)
   {
      cheval_frame_pos.x +=140;
      if (cheval_frame_pos.x >= 700){cheval_frame_pos.x = 0;}
      cheval_pos += 12;
      if (cheval_pos > 2185)
      {
         cheval_pos = 0;
         cheval = 0;
      }
   }
   else {cheval = int(random(100));}
```

We delete the following lines (you can keep them at the moment, but don't forget to remove them before you finish the game!):

~~on_space anim_cheval;~~

```
~~function anim_cheval~~
~~{~~
~~   cheval = 1;~~
~~}~~
```

And we modify the following line (the blue is replaced by the red):

Before:

```
   digits = 35,0,1,comic,1,cheval;
```

After

```
   digits = 35,0,2,comic,1,cheval;
```

Actually we want 2 characters to be displayed to see the result of **`int`** (random (100)).

Sound

But what would our game be like without any sound effects, the world of silence?

We are going to begin with the easiest and most evident: the steps of the walking cowboy.

We create 2 variables for the sound which we insert into our variables:

```
sound tap, <tap.wav>;
var taphandle = 0;
```

Then we play the sound while the cowboy moves and we stop it at the same time the cowboy stops.

To copy into our **`movement`** routines(lines in red only):

```
   if ((state == go_right) && (compteur_pas > 0))
   {
      if (taphandle ==0) //le son n'est pas joué
      {
         play_loop (tap,20);
         taphandle = result;
      }

      -   - - - - - - - - - - - - -

      else
      {
         compteur_pas = 0;
         state = waiting;
         cowboy_frame_pos.x = 0;cowboy_frame_pos.y = 0;
      }
      if (state == waiting)
      {
         stop_sound (taphandle);
         taphandle = 0;
      }

      -   - - - - - - - - - - - - -

   if ((state == go_left) && (compteur_pas > 0))
```

```
  {
     if (taphandle ==0) //le son n'est pas joué
     {
        play_loop (tap,20);
        taphandle = result;
     }
      -   - - - - - - - - - - - - -

     else
     {
        compteur_pas = 0;
        state = waiting;
        cowboy_frame_pos.x = 0;cowboy_frame_pos.y = 80;
     }
     if (state == waiting)
     {
        stop_sound (taphandle);
        taphandle = 0;
     }
      -   - - - - - - - - - - - - -
```

That was not the hardest, even if it is not ingeniuous. I leave it up to you to look for a proper step sound that you like.

The interesting point is when it comes to the horsesteps. It is obvious that depending on the distance between the horse and the cowboy, the sound volume has to vary.

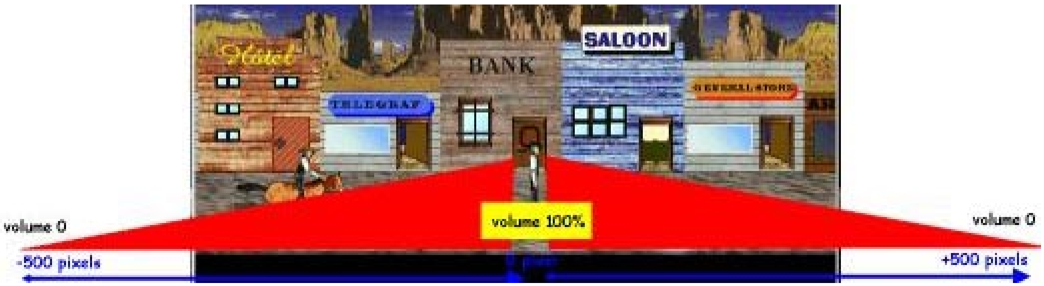
If you pay good attention to the instructions regarding sounds, the only one providing sound tuning is:

```
tune_sound(handle,varvol,varfreq);
```

Of course, we will play with the variable **`varvol`**.

What we want: the more the horse approaches, the louder should become the sound, the more it goes away, the more the sound should turn down.

I set up a completely arbitrary rule to determine that there is no sound to be heard beyound a distance of 500 pixels between the horse and cowboy. And within 500 and 0 pixels the sound volume increases. Which the following picture may illustrate:



Sound Pyramide

It goes without saying that it is not the position of the horse in relation to us we are interested in, but the distance of the horse regarding the city relative to us. Although we are always in the middle of the screen, our positon regarding the city is not constant. This position is calculated already since it serves us to display the panel which contains the horse and it's name is: **`cheval_pan.pos_x`**. We substract it from our position in relation to the screen (the medium = 320). The result gives us a value ranging in-between 500 and 0 (more ore less). Thus we divide our

result by 5 to receive a value that ranges between 0 and 100 which are precisely the limits of our volume tuner. Of course we take the absolute value to disregard the sign.
The little problem that remains is that the result we get returned is at 0 in case that the horse is close to us and a volume of 100 as soon the horse is at it's farest distance. We deal with that by deducting the given result from 100 and re-enter the whole thing into the instuction.

In computer algebra it's written:

```
volume = 100-(abs(320-cheval_pan.pos_x)/5);
```

Please trust me, I'm telling you it works.

All we've got to do now is tu put all that stuff into our skript. We create the following variables:

```
sound pascheval, <pascheval.wav>;
var paschevalhandle = 0;
var volume = 0;
```

Then into our **`movement`** routine we add (red lines):

```
function deplace()
{
   if (cheval == 1)
   {
      if (paschevalhandle ==0) //le son n'est pas joué
      {
         play_loop (pascheval,5);
         paschevalhandle = result;
      }
      else
      {
         volume = 100-(abs(320-cheval_pan.pos_x)/5);
         tune_sound (paschevalhandle,volume,0);
      }

      cheval_frame_pos.x +=140;
      if (cheval_frame_pos.x >= 700)
      {
         cheval_frame_pos.x = 0;
      }
      cheval_pos += 12;
      if (cheval_pos > 2185)
      {
         cheval_pos = 0;
         cheval = 0;
      }
   }
   else
   {
      cheval = int(random(100));
      stop_sound(paschevalhandle);
      paschevalhandle = 0;
   }
```

Run the level and you will have a nice little pleasure.

It is not bad but nevertheless I feel a little disappointed. I have the helmet on my ears, my sound card supports stereo effects, it would be much better if the sound was to the left when the horse is on the left side and to the right when the horse is overthere. How about that?

I reread the parameters of the instruction, But in vain. Nothing that allows us such a differenciation. Perhaps I did not take the good instruction? Let us continue to read the WDL manual.

Wouldn't be the instruction **`play_entsound`** (my, sound,Var), which plays a 3D sound more appropriate? Yes but the explanations talk about such as **`entity`** and **`my.`** But as long as we are not on drugs, we have none of that as we are in a 2D game.

Well, I will open your eyes: the A4/A5 engine is so fantastic, that even in 2D we have a camera and we also can create entities. Isn't that amazing?

So we are going to create an entity as a peg which we will attach to our horse and we will position the camera at our player's site.

Go ahead and type these lines within our variables:

```
synonym bidon {type entity;}
var bidon_pos[3] =0,200,0;
```

And in our function **`main`** please write the lines in red:

```
   load_status();
   create <bidon.pcx>,bidon_pos,f_bidon;
   game();
}

function f_bidon
{
   bidon = me;
   while (bidon == null){wait 1;}
}
```

In our while loop of **`game`** we add lines in red:

```
   cheval_pan.visible = on;
   camera.x = 0;
   camera.y = 0;
   camera.z = 0;

   while(1)
   {

      ciel_pan.pos_x = - ciel_pos;
      scroll_ciel_pan.pos_x = 640 - ciel_pos;
      mont_pan.pos_x = - mont_pos;
      scroll_mont_pan.pos_x = 640 - mont_pos;
      ville_pan.pos_x = - ville_pos;
      //avance_ciel();
      bidon.x = cheval_pan.pos_x+60;
```

And in our **`deplace`** function replace the blue lines by the ones in red:

```
function deplace()
{
   if (cheval == 1)
   {
      if (paschevalhandle ==0) //le son n'est pas joué
      {
         play_loop (pascheval,5);
         paschevalhandle = result;
      }
      else
      {
         volume = 100-(abs(320-cheval_pan.pos_x)/5);
         tune_sound (paschevalhandle,volume,0);
      }


   if (cheval == 1)
   {
      if (paschevalhandle ==0) //le son n'est pas joué
      {
         play_entsound (bidon,pascheval,200);
         paschevalhandle = result;
      }
      else {if (snd_playing(paschevalhandle)==0){paschevalhandle =0;}}
```

Run the level and have a walk. Straightaway it is more realistic, don't you think so?

Now that you understood everything well, it will be done quickly to make the horse neigh at random when it is close to the player.

We type the red lines:

```
sound pascheval, <pascheval.wav>;
var paschevalhandle = 0;
sound henni, <henissem.wav>;
var hennihandle = 0;
- - - - - - - - - - -

   if (cheval == 1)
   {
      //henni si moins de 100 pixels du joeur et 1 fois sur 3
      if ((abs(320-cheval_pan.pos_x) < 100) && (random(3) <1))
      {
         if (hennihandle ==0) //le son n'est pas joué
         {
            play_entsound (bidon,henni,100);
            hennihandle = result;
         }
         else {if (snd_playing(hennihandle)==0){hennihandle =0;}}

      }

      if (paschevalhandle ==0) //le son n'est pas joué
```

Conclusion

And yes, it has finished already. This learning software serves it's purpose when it gives you first basics to create your own 2D animations.

Footnotes of history: Carson City is a game that first was developed on Spectrum sinclair and then on Amstrad assembler. Then I started again to work on the project with C++ and Directdraw but then 3D GameStudio was released and made me change my plans.

This fact made me thinking about making a 3D version. But then, maybe because the 3D GameStudio is anyway simpler than C++, the idea was born to do even the 2D game wirh 3D GameStudio.

At the moment I hope you had fun to walk along with me on one little road within that fascinating universe of game programming and that I succeded to share some of my knowledge and much of my passion.

Visit my site: http://alainbrgeon.free.fr

PS: you will find the complete files of scenario under the name of **`anim2d.al1`** .