Camera Tutorial Ver. 2

Or: How do I get those views to work?

By Gnometech

Preface to Version 2

I would like to thank all of you who gave me positive feedback regarding this tutorial. However, there are still blank spots on the map regarding cameras and views. One of the frequently asked questions in the 3DGS forum during the last few weeks was "How do I create a Resident Evil style camera that faces the player all the time, but stays fixed in one place?" And since this is really not very hard to achieve, another question followed almost anytime: "How can I switch between several of those cameras?" Now, the last one is a little more interesting and I will try to cover it in the newly added chapters (beginning with Chapter 5). I will also take the chance to explain about pointers and handles since there seem to be many people who still do not know what a pointer exactly is, what it does and why we need those things.
As before, I hope you enjoy this tutorial. March 2002

Chapter 1: Creating the map and preparing the session

Greetings!

My name is Gnometech and this is my camera tutorial. I won't make any promises like „You will definitely be able to create your own camera views after working through this", but chances are that things are a little clearer after reading and working through this one.

Start by creating a small map or take one of your sample maps. I am assuming you know how to do this and won't tell you how to do it here. Place the player model somewhere in the map and assign it your favourite movement action, either the predefined player_move or one you yourself have written.

For the cameras to work it is essential that the entity synonym „player" is known and defined and that it is... well, the player. movement.wdl assigns it, if you have written your own movement script, make sure you do, too.

Since you want to call your own camera code, be sure to change the line where you call the predefined „move_view" to the new one, which we will call „update_views". Again, if you don't use predefines scripts, it doesn't matter at all, where you call this routine, just make sure that it is called once every frame cycle.

Now create a new WDL file called „cam.wdl" and include it in your main script. Ready to go!

Chapter 2: The 1st Person View

This one is actually quite simple. Put the following code lines into the WDL script:

```
view 1st_person
{
        layer = 1;
        pos_x = 0;
        pos_y = 0;
}

function init_cameras()
{
        camera.visible = off;
        1st_person.size_x = screen_size.x;
        1st_person.size_y = screen_size.y;
        1st_person.genius = player;
        1st_person.visible = on;
}
```

```
function update_views()
{
	1st_person.x = player.x;
	1st_person.y = player.y;
	1st_person.z = player.z;
	1st_person.pan = player.pan;
	1st_person.roll = player.roll;
	1st_person.tilt = player.tilt;
}
```

Now, include the call "init_cameras" somewhere in your main function, after the level load.
Save your script and run your level. If you have done everything correctly, the camera should follow the player's movements.
What have we done here? Well, we have defined a new view. Imagine a view to be your connection to the game world. It is a "window", which leads there. You can define the position and size of the window on the screen by setting pos_x, pos_y, size_x and size_y. As you can see, the upper left corner of our view is equal to the upper left corner of the screen and the size is exactly the screen size.
The function update_views is updated every frame cycle and sets new values for the x, y and z coordinates of the camera (the view) in the world. Here, we just set them to the player's coordinates, thus making the camera always be "inside" the player.
The script isn't perfect, yet. For example, the camera is positioned at the center of the player's model and not at his head, causing the view to be a little too near to the floor. And we can't look up or down, yet.

The first thing is fixed fast. We just define a variable at the beginning of the new script:

```
var eye_height = 20;
```

And then, instead of

```
1st_person.z = player.z;
```

we write

```
1st_person.z = player.z + eye_height;
```

thus setting the camera higher by 20 quants. If the value doesn't fit your needs, adjust it, if needed even during gameplay.

Now, we want the player to be able to look up or down. For this purpose, we need other variables:

```
var tilt_1st = 0;
var cam_turnspeed = 2;
var max_tilt_1st = 40;
```

Now change the line reading:

```
1st_person.tilt = player.tilt;
```

to

```
1st_person.tilt = player.tilt + tilt_1st;
```

Then, add the functions:

```
function look_up()
{
	if (tilt_1st < max_tilt_1st) { tilt_1st += cam_turnspeed; }
}

function look_down()
{
	if (tilt_1st > -max_tilt_1st) { tilt_1st -= cam_turnspeed; }
```

```
}
```

to your script. These functions just have to be called during gameplay. For example, you could define:

```
on_pgup = look_up;
on_pgdn = look_down;
```

Save your work and run the level. Hmmm... not bad, but not the desired effect. If you press one of the keys and hold it, the view changes a little and won't move any further. You have to press it repeatedly for the view to change and that slowly.

How do we fix this? Change the on_pgup and on_pgdn definitions to:

```
on_pgup = handle_pageup;
on_pgdn = handle_pagedown;
```

Then define two more functions:

```
function handle_pageup()
{
	while (key_pgup)
	{
		look_up();
		wait(1);
	}
}

function handle_pagedown()
{
	while (key_pgdn)
	{
		look_down();
		wait(1);
	}
}
```

This way our functions are called each framecycle as long as the keys are pressed. If the turning of the camera appears to be to fast or to slow for you, adjust the cam_turnspeed value. Also, if you feel the field of view is too limited and that the player should be able to change his view angle more, set the max_tilt_1st value differently. A value of 90 means here: he is able to look directly up and directly down. A value above 90 even lets him perform a somersault... ;)

Enough for this chapter. The 1st Person view is working now, on for something different: a 3rd person camera.

Chapter 3: Freely rotatable 3rd Person View

Now we want to create a rotatable 3rd Person View. There are two possibilites how we can achieve this: the first would be to create camera that stays outside the player all the time and does NOT turn, even when he does, always facing him from one certain direction. The other would be a camera that turns when the player turns, thus always staying behind him (or in front of him, or to the side, or...)

I will make this camera freely turnable and zoomable. If you need just a top view for your game or a side view... no problem at all, just choose the correct values (you will see which) and don't include the functions to change them. ;-)

Let's start. I will do the first attempt, creating a 3rd Person View that faces the player always from the same direction and then tell you what need to be changed when you want the view to turn with the player.

Define a second view now:

```
view 3rd_person
```

```
{
	layer = 1;
	pos_x = 0;
	pos_y = 0;
}
```

Add the following lines somewhere within the "init_cameras" function:

```
...
	3rd_person.size_x = screen_size.x;
	3rd_person.size_y = screen_size.y;
...
```

We won't make it visible, yet. Instead, we will be able to toggle both cameras during gameplay. Let us define the switching routine now:

```
var cam_mode = 0; // 0 meaning 1st Person, 1 meaning 3rd Person

function toggle_cams()
{
	if (cam_mode == 0)
	{ // Change to 3rd Person
		1st_person.visible = off;
		3rd_person.visible = on;
		cam_mode = 1;
	}
	else
	{ // Change to 1st Person
		3rd_person.visible = off;
		1st_person.visible = on;
		cam_mode = 0;
	}
}

on_f8 = toggle_cams;
```

By hitting F8 now, we are able to change between the views. However, that isn't of much use, yet, since the 3rd persons x, y and z aren't defined anywhere. We will do this now by changing the "update_views" function. (Remember, that is the one called each frame cycle.)

But first, we have to think... how can we define a 3rd person camera, that is freely rotatable around the player? It should (for now at least) stay in the same plane, so the z value will be the same. But how can we calculate the fitting x and y values? We need a little maths for that.

Imagine the player seen from above. Best thing would be to take a piece of paper and put a dot on it, somewhere, to indicate the player's position seen from above. Draw a circle around this dot. This should be the circle on which the camera can be moved around the player. This circle has a certain radius. If you set a certain point of the circle as "0" (for example the "top", but which one doesn't matter), you can describe every other point P on the circle just by the arc between the lines drawn from "0" to the centre and from P to the centre.

Thus, by defining the distance from the player (the circle's radius) and the angle in the plane, the camera's position is determined. I will give you the formula, you can easily get to it yourself with a little maths:

```
var dist_planar =  300; // distance from the player
var cam_angle = 0;

function update_views()
{
	if (cam_mode == 0)
	{
		1st_person.x = player.x;
		1st_person.y = player.y;
```

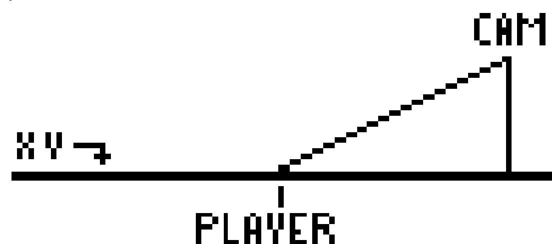
```
                1st_person.z = player.z + eye_height;
                1st_person.pan = player.pan;
                1st_person.roll = player.roll;
                1st_person.tilt = player.tilt + tilt_1st;
        }
        else
        {
                3rd_person.x = player.x - cos (cam_angle) * dist_planar;
                3rd_person.y = player.y - sin (cam_angle) * dist_planar;
                3rd_person.z = player.z;
                3rd_person.pan = cam_angle;
                3rd_person.roll = 0;
                3rd_person.tilt = 0;
        }
}
```

Save the script and run the level. Hit F8 to toggle the views. You should now view your player from outside. If you move through your map, the camera should always keep the same distance to the player and won't even change position, when the player turns.
If you change the values of dist_planar during gameplay (via TAB for example or if you define a function to change them by hitting a key... you should be able to create such a function yourself now), the camera zooms in or out. If you change the cam_angle, it turns around the player, always facing him. (3rd_person.pan = cam_angle makes sure of that)

There is however a problem with walls. The camera will go straight through walls and more often than not, you will have obstacles blocking the line of sight to the player. We will engage this problem in the next chapter, first we will change the camera code a little... we now want to be able to move the camera upwards, while always facing the player. It should also keep the same distance, thus moving not on a circle around him, but on a sphere.

How can we achieve this? Well, imagine your player now being seen from the side. If the camera is above the XY-plane of the player and you draw a line from it to the player, you get an angle between the XY plane and the line. This angle defines the position of the camera exaclty.

Of course, if you want to keep the total distance the same, the distance regarding the XY plane changes! A picture to show this:



Now for the code adjustments:

```
var dist_total = 300; // Change THIS value to zoom in or out
var tilt_3rd = 0;

function update_views()
{
```

```
...
        else
        {
                dist_planar = cos (tilt_3rd) * dist_total;
                3rd_person.x = player.x - cos (cam_angle) * dist_planar;
                3rd_person.y = player.y - sin (cam_angle) * dist_planar;
                3rd_person.z = player.z + sin (tilt_3rd) * dist_total;
                3rd_person.pan = cam_angle;
                3rd_person.roll = 0;
                3rd_person.tilt = - tilt_3rd;
        }
```

By changing the tilt_3rd, cam_angle and dist_total values we can now move the camera freely around the player. It moves on a sphere where the radius can be changed by changing dist_total.
It is up to you to write functions for changing these values during gameplay and assign keys to it. Also, you will have to keep them limited, in order to prevent the player to zoom in or out to much or to tilt the camera too much.
In the next chapter, we will deal with the wall problem.

Chapter 4: How to keep the camera from going through walls

One thing first: there are several solutions for this and I don't claim mine to be the best. However, it worked fine for me so far, so give it a look. It isn't hard to understand, either.

The idea is to send a trace from the player to the new calculated camera position and see, if an obstacle is being hit on the way. If so, the distance between the player and the camera is altered... by the amount of space between the player and the obastacle, thus allowing a "smooth" handling of the cam. No jerky movement. :)

Let's see how it is done. After your calculations regarding the 3rd_person values (see last chapter), include the following line:

```
function update_views()
{
...
                3rd_person.roll = 0;
                3rd_person.tilt = - tilt_3rd;
                validate_view();
        }
...
}
```

This function will be defined now:

```
var dist_traced;

function validate_view()
{
        my = player;
        trace_mode = ignore_me + ignore_passable;
        dist_traced = trace (player.x, 3rd_person.x);
        if (dist_traced == 0) { return; } // No obstacles hit... fine
        if (dist_traced < dist_total)
        {
                dist_traced -= 5; // Move it out of the wall
                dist_planar = cos (tilt_3rd) * dist_traced;
                3rd_person.x = player.x - cos (cam_angle) * dist_planar;
                3rd_person.y = player.y - sin (cam_angle) * dist_planar;
                3rd_person.z = player.z + sin (tilt_3rd) * dist_traced;
        }
}
```

The function to determine the camera's position is exaclty the same, it just uses a new distance from the player... dist_traced, the one obtained by trace. It is modified a little, to prevent the camera from being directly inside the wall.

Now the camera should avoid walls and other obstacles in 3rd person mode smoothly.

One thing is left to do, before I leave you alone... letting the camera stay behind the player all the time. It is very simple to add this, just modify the lines:

```
3rd_person.x = player.x - cos (cam_angle) * dist_planar;
3rd_person.y = player.y - sin (cam_angle) * dist_planar;
```

to

```
3rd_person.x = player.x - cos (cam_angle + player.pan) * dist_planar;
3rd_person.y = player.y - sin (cam_angle + player.pan) * dist_planar;
```

So, all you have to do is add the player's pan to cam_angle in the calculation. Be sure to do the same in the validate_view or you will get strange errors! You also have to change the line

```
3rd_person.pan = cam_angle;
```

from the update_views to

```
3rd_person.pan = cam_angle + player.pan;
```

to get the camera's pan correct. (I am sorry that I forgot to mention this in the first version of this tut. My fault)

If the camera isn't behind the player now at gamestart, but faces him, let's say, adjust the cam_angle, until it suits your needs... and then just don't let the player change this value anymore. Then the camera will stay behind the player all the time.

Chapter 5: Pointers and handles

Ok, the next thing we will create are "Resident Evil" style cameras. For those who are not familiar with the game, those are cameras that stay fixed at a certain position (for example in the corner of the room), but follow the player's movement. As the player moves away, the game switches to another of those cameras and so forth.
In order to handle more than one camera position, it will come in very handy to work with pointers and handles. But, what exactly are these? Let me give you a brief introduction to these structures. Of course, those things are used all throughout the 3DGS Scripting world and their uses are not just limited to cameras and views!

I will start with a pointer. A pointer is, in principle, a variable. But this one does not hold numbers (well, since everything in the computer is a number, it does indeed hold "numbers", but they are not interpreted as such). The pointer can store information that allows you to access a certain entity. One familiar example comes from the templates and I also worked with it in the above example: player.
The pointer player is a pointer to the player entity. And this also explains what the pointer does: it allows access to the specific data of the player entity. Each entity has ist own data space, where ist position (x, y, z), ist orientation (pan, tilt, roll) and other things are stored, like 48 skills and 8 flags. Since there are normally so many entities in a game, we can't just write "x" in the code and expect the compiler to understand which entity we mean. Instead, we tell him with a pointer:

```
player.x += 50;
```

This will increase the x value of the special entity the pointer "player" points at by 50. Two very special pointers are my / me and you / your. (The names are equivalent, you can use either at any time you want to access one of those pointers)
The my pointer is set automatically when you are within an entities action (assigned via WED) and it refers to the entity to which you assigned the action. It is quite simple: an action like:

```
action turn
{
```

```
	while(1)
	{
		my.pan += 3;
		wait(1);
	}
}
```

will change the pan of the entity to which the action is assigned. If you have more than one entity with this action in your level, no problem, they all will turn, since my always refers to the correct entity.
"You" is another case. This pointer is set by some events and functions by the engine. Refer to the manual for more detail, here just some examples: whenever you do a trace or a scan or if your entity collides with another one, then "you" is set to the other entity.
Now, how to declare a pointer? Similar to a var, you just write:

```
entity* this_is_a_pointer;
```

The star behind the name just means that this is a pointer to the object "entity" (C syntax)
Please note that just like a variable that is declared that way, the pointer is always global. This can lead to problems. It is perfectly Ok to have a pointer called player, since in most cases there will be only one player (multiplayer games also have to deal differently with such things). However, a pointer called "enemy" would be useless in most cases, since it can point only to one enemy and not to all at once. But what do we do if we want to have an array of pointers? For this, we have two very useful functions: handle and ptr_for_handle.

The first one receives a pointer and converts it into a number. The second one takes a number (which was obtained this way) and converts it back to the original pointer. So, with these functions we can store pointers anywhere where we can store numbers: in arrays, in skills...

One last thing: if a pointer is defined, it points to nothing. This means, it has the predefined value "null". Whenevr you try to access a skill or a var through a null-pointer, you will get an error message (empty pointer), so be careful about it.

Since we now know this, we can approach our Resident Evil like cameras.

Chapter 6: Fixed Cameras

I assume that if you have fixed cameras, you don't want any of the other type, so I will just start over. If you want to combine these types, it isn't very hard, just look at the example above how I toggled between the cameras.

Now, open a new script and type the lines:

```
entity* fixed_cam;

view fixed
{
	layer = 1;
	pos_x = 0;
	pos_y = 0;
}

function init_cameras()
{
	camera.visible = off;
	fixed.size_x = screen_size.x;
	fixed.size_y = screen_size.y;
	fixed.visible = on;
}

function update_views()
{
	if (fixed_cam == null) { return; }
```

```
        vec_set (fixed.x, fixed_cam.x);
        vec_set (temp, player.x);
        vec_sub(temp, fixed.x);
        vec_to_angle(fixed.pan, temp);
}
```

Include the script into your main.wdl. Call init_cameras from your main and update_views from within your player's action loop. Now, what does it do? In update_views, the position of the view is set to the position of the entity "fixed_cam". Then, the view is turned so that it faces the player.
It can't yet work this way, since fixed_cam isn't set anywhere. To change this, write the following action:

```
action set_fixed_pos
{
        fixed_cam = me;
        my.invisible = on;
        my.passable = on;
}
```

Now place an entity (anything will do) in your level, whereever you want the camera to be and assign this action to it. Then run the map.

The camera should be at the place where you put the entity (which in turn should be invisible) and it should turn to face the player's movement all the time. Wow, this was easy.
However, in most cases a level consists of more than one room and in this case, we might want to change the camera to a new one. This will be the next thing we do.

Now, put more of those dummy entites in your map and assign all of them the action set_fixed_pos. In order to distinguish them from one another, give each a different "skill1", starting with 0 for the first one. Those will be your cameras from now on.

Then modify the action:

```
var cam_array[200]; // a maximum of 200 cameras

action set_fixed_pos
{
        if (my.skill1 >= 200) { return; }
        cam_array[my.skill1] = handle(me);
        my.invisible = on;
        my.passable = on;
}
```

Now, this is different from the approach above. When we had one camera, we could take a global pointer and set it to it. Now we have more. And here we use handles to store the pointer values in an array.

What is still missing is a function that sets the view to a certain camera:

```
function set_view_to_cam(cam_number)
{
        if (cam_array[number] == 0) { return; }
        fixed_cam = ptr_for_handle (cam_array[number]);
}
```

The fixed_cam pointer is the one that matters for update_views. And that one is modified by this function. Note that this function takes a parameter. The usage is clear: whenever you want to change the view to a certain camera (say, camera number 4), just write:

```
        set_view_to_cam (4);
```

You can scatter small entities throughout your map, with trigger events that will change the view when the player gets near... you can let the player scan every so often to check the newest camera and set the view to this

one... you can determine the newest with vector arithmetics and do a trace to check if the player can indeed be seen... there are many possibilities and I will let your imagination take control now.

Chapter 7: Camera follows path

For those of you who are interested, I recently wrote a small "Camera follows path" function. It has nothing to do with the rest of this tutorial and I won't explain it in detail, I will just include it here for those of you who are interested. Here is the code:

```
DEFINE _speed, skill1;
DEFINE _turnspeed, skill2;
DEFINE _crit_dist, skill3;

DEFINE _turnto1, flag1;

entity* cam_ent;
var to_angle[3];
var pan_diff;
var tilt_diff;

function turn_towards_point()
{
	vec_set(temp, my.skill21);
	vec_sub (temp, my.x);
	vec_to_angle (to_angle, temp);
	my.pan = ang(my.pan);
	my.tilt = ang(my.tilt);
	to_angle.pan = ang(to_angle.pan);
	to_angle.tilt = ang(to_angle.tilt);
	pan_diff = abs(my.pan - to_angle.pan);
	if (pan_diff > 180) { pan_diff = 360 - pan_diff; }
	tilt_diff = abs(my.tilt - to_angle.tilt);
	if (tilt_diff > 180) { tilt_diff = 360 - tilt_diff; }
	if (pan_diff > 6 * time * my._turnspeed) { my.skill48 = time * my._turnspeed; }
	else {my.skill48 = pan_diff / 6; }
	if (((my.pan > to_angle.pan) && (abs(my.pan - to_angle.pan) < 180)) || ((my.pan < to_angle.pan) &&
(abs(my.pan - to_angle.pan) >= 180)))
	{ my.pan -= my.skill48; } else { my.pan += my.skill48; }
	if (tilt_diff > 6 * time * my._turnspeed) { my.skill48 = time * my._turnspeed; }
	else {my.skill48 = tilt_diff / 6; }
	if (((my.tilt > to_angle.tilt) && (abs(my.tilt - to_angle.tilt) < 180)) || ((my.tilt < to_angle.tilt) &&
(abs(my.tilt - to_angle.tilt) >= 180)))
	{ my.tilt -= my.skill48; } else { my.tilt += my.skill48; }
}

// uses _speed, _turnspeed, _crit_dist, _turnto1
action mobile_cam
{
	my.invisible = on;
	my.passable = on;
	cam_ent = me;
	if (my._speed == 0) { my._speed = 5; }
	if (my._turnspeed == 0) { my._turnspeed = 6; }
	if (ent_path("cam_path") == 0) { return; }
	my.skill10 = ent_waypoint (my.skill21, 0);
	if (my._turnto1 == on)
	{
		vec_set (target, my.skill21);
		vec_sub (target, my.x);
		vec_to_angle (my.pan, target);
	}
```

```
		while(1)
		{
			turn_towards_point();
			my.skill24 = my._speed * time;
			ent_move (my.skill24, nullvector);
			if (vec_dist(my.x, my.skill21) <= max(time * my._speed, my._crit_dist))
			{
				ent_waypoint(my.skill21, my.skill9);
				my.skill9 += 1;
				if (my.skill9 >= my.skill10) { my.skill9 = 0; }
			}
			vec_set (camera.x, my.x);
			vec_set (camera.pan, my.pan);
			wait(1);
		}
}
```

The usage: include the code, put a path into your map with the name "cam_path", put an entity in with the action "mobile_cam" and set the skills:

skill1 = speed of the camera
skill2 = turning speed of the camera
skill3 = distance from the next path point when it starts to turn towards the next
flag1 = if it is set, the camera is turned towards the first path point at game start, if not it turns while moving

Note that the default camera has to be active for this to work (or simply change "camera" to "1st_person" or whatever your view is called)

That's it for now. If you have anymore questions regarding this tutorial, ask in the 3DGS Forum or eMail me at:

gnometech@gmx.de

Enjoy.

Gnometech