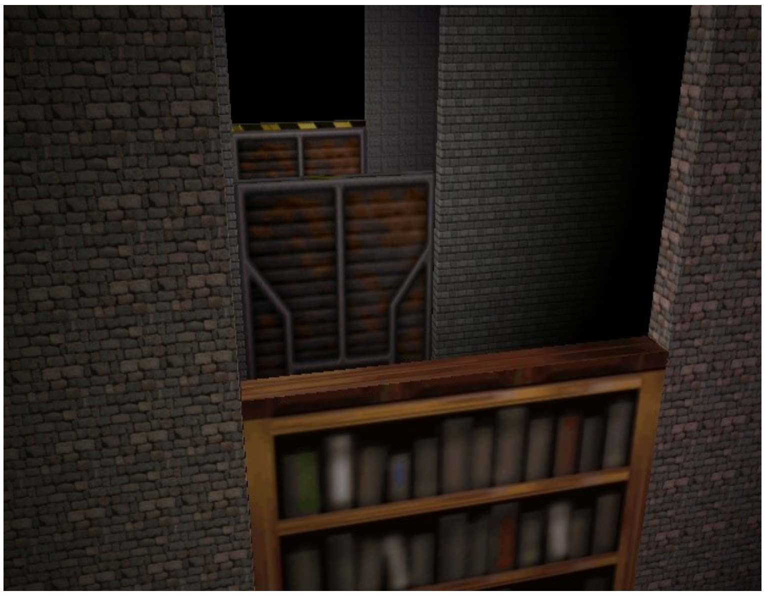
Lite-C and A8

Part 2: Doors



Introduction

Hello to the second part of these tutorials. In this tutorial you will hopefully learn how to code different kind of doors and especially be able to understand the ideas behind it.

Actions

In the previous tutorial, the duck was created in your code, which can be usefull for many things, but is very uncomfortable if it comes to creating a whole level. Right, we used the level_load instruction, which implies that it can be used to load a level with content and not just an empty one. This level, as you hopefully already found out, can be created with the help of WED, Gamestudios WorldEDitor. So far so good, BUT when creating the duck, it was possible to assign a function to it, so how to do the same when creating a level with WED?

WTF is an action?

The solution is, what gamestudio calls action. An action is nothing else, but a function assigned to an entity, which usually doesn´t return anything. This means, that the ducks function already was a first action. In WED, each object has a property panel and one of its tabs when marking an entity is called behaviour.
In this behaviour panel, there is a text field titled action. Now, you just have to write your functions name into that text field and you are done. The disadvantage is, that it is very uncomfortable to always write down the name of your function. Because of that, there is a button next to the action field which opens a new panel with the list of your functions. At least basicly. The problem is, that a list of all your functions would be very messy, thus, you have to mark the functions you want to be displayed there, within your code, by not using void or var or whatever as return type of your function, but the one called action. Action is just a synonym for void, but will be parsed by WED for entity functions.

Preparation

To get started, I already prepared an ugly level and two empty script files.
The first thing we want to do, is to load the level and to add some empty actions to our code.

Folders

To keep things clean, you usually want to structure your files within folders. I already created such folders and placed the files accordingly in the example project. If you now want to access those files, the compiler or at least the engine will tell you that it can´t find the files. The reason is, that it only searches them within the projects root directory. What we have to do, is to let the compiler and the engine know all the folders it is meant to search for files in.
For this, gamestudio uses the precompiler command `#define PRAGMA_PATH "path"`, where path is the relative or absolute path to the folder we want to add to our project.
In case of the example project, there are the folders *Codes*, *Entities* and *Levels*. While the files from the last two are loaded at runtime, it doesn´t matter where we define them, but because the Scripts are already needed at compile time, their folder has to be defined before a script from within the folder is included.
Because of this, I usually just put all the path definitions on top of the main script like this:

```
#define PRAGMA_PATH "Codes"
#define PRAGMA_PATH "Entities"
#define PRAGMA_PATH "Levels"
```

As we want to use A8 and the default camera, we have to incude the two coresponding files. There is also a Doors.c which is meant to hold the door actions. This results in three more lines:

```
#include <acknex.h>
#include <default.c>

#include "Doors.c"
```

Loading the level

Loading a level created with WED is very easy. We just have to call `level_load("level.wmb")` where level.wmb is the compiled level, which you get when pressing the build button in WED. The example level is called *doors.wmb*. Create a `main` function and load that level.
When you testrun it, you should be able to move through the level after pressing 0.

Setting up the doors

Open *Doors.c*, which you will find within the *Codes* folder and declare three actions. They are all meant to contain an empty `return;` and might be called `DoorSlideUp`, `DoorSlide` and `DoorRotate`.

Copy the file *doors.wmp,* located within the *Levels* folder into the same folder as *Main.c* (just to make sure that it handles our relative paths correctly) and open it with WED. If WED tells you, that it can´t find *standart.wad*, remove it and add it again from your gamestudio folder. Now we have to tell WED about our paths and actions. To do so, Open File->Map Properties, click at the "Choose Script" button and choose our *Main.c*.

Select the first door (door1_wmb_000 in the levels tree structure on the left side in WED), open its behaviour panel (right click -> Properties -> Behaviour, or directly right click -> behaviour), click the "Choose Action" button and assign the `DoorSlideUp` action to it. Do the same with the other two doors, where the second one is meant to get `DoorSlide` assigned and the third one `DoorRotate`.

Save the changes in WED and build the level.

Door sliding up

There are of course many different kind of doors. I will explain how to code two basic ones, starting with a simple one, which just slides up, creating a more flexible version of it and finishing with an a rotating one.

Idea for a first door

You often see doors opening by moving a bit up. As the orientation of the door doesn´t really matter, the only thing we have to do, is to check if the camera is close and move the door up a certain amount in case it is. We can access the camera position through the global camera pointer, which points to the default view and has x, y and z components just like entities. There is happily the already defined function `vec_dist(VECTOR *v1, VECTOR *v2)`, which returns the distance of the two given vectors.
So we just have to check the distance between the camera and the my entity every frame in a while loop and open the door if the camera is close.
As you see, vec_dist wants pointers to VECTORs and as a VECTOR are just three vars placed behind each other in memory, which is the same as the entities and cameras x, y and z koordinates are placed, we can just pass the pointer to the cameras and entities x component to `vec_dist` and similar functions:

```
	var distance = vec_dist(&camera->x, &my->x);
```

if-else

Now we need to check if the camera is close and as such an if-situation is very common, there is the much more fundamental expression than a loop:

```
	if(condition)
	{
		do something...
	}else
	{
		do something else...
	}
```

Which does something, if condition is true, or something else if it is not. The else part can also be skipped if we don´t want our program to do something else.
True is any value but zero, zero is false.
The condition is usually a logic expression like smaller, bigger or equal. You will most of the time want to check if two values are equal (notice the two equal signs, as otherwize we would assign the value):

```
	if(val1 == val2)
	{
		do something...
	}
```

bigger:

```
	if(val1 > val2)
	{
		do something...
	}
```

smaller:

```
	if(val1 < val2)
	{
		do something...
	}
```

smaller or equal:

```
if(val1 <= val2)
{
	do something...
}
```

or bigger or equal:

```
if(val1 >= val2)
{
	do something...
}
```

It is also possible to connect several conditions with and (`&&`) or or (`||`):

```
if((val1 > val2 && val3 <= val1) || val2 == val3)
{
	do something...
}
```

which will do something, if `val1` is bigger than `val2` AND `val3` is smaller or equal to `val1`.
But it will also do something if `val2` equals `val3`.

Coding a door

This is what the doors action could look like to play a beep sound if the camera gets close:

```
action DoorSlideUp()
{
	var distance;

	while(1)
	{
		distance = vec_dist(&camera->x, &my->x);
		if(distance < 200)
		{
			beep();
		}

		wait(1);
	}
}
```

Now we just have to move it along the z-axis until it reaches a given height and moves back to its initial height as soon as the camera is again further away.
A first movement could look like this:

```
action DoorSlideUp()
{
	var distance;

	while(1)
	{
		distance = vec_dist(&camera->x, &my->x);
		if(distance < 200)
		{
			my->z += 10*time_step;
		}else
		{
			my->z -= 10*time_step;
```

```
                }

                wait(1);
            }
        }
```

But it of course doesn´t stop its movement yet, so we need another if to check if it already reached the height we want it to have:

```
        action DoorSlideUp()
        {
            var distance;

            while(1)
            {
                distance = vec_dist(&camera->x, &my->x);
                if(distance < 200)
                {
                    if(my->z < 250)
                    {
                        my->z += 10*time_step;
                    }else
                    {
                        my->z = 250;
                    }
                }else
                {
                    if(my->z > 0)
                    {
                        my->z -= 10*time_step;
                    }else
                    {
                        my->z = 0;
                    }
                }

                wait(1);
            }
        }
```

This already works quite nice, but what if we would want to place the door at a higher floor? It would still move back to this height which will be wrong then.
The solution is, to simply store the starting height and add it to the height values:

```
        action DoorSlideUp()
        {
            var startheight;
            var distance;

            startheight = my->z;

            while(1)
            {
                distance = vec_dist(&camera->x, &my->x);
                if(distance < 200)
                {
                    if(my->z < startheight+250)
                {
                        my->z += 10*time_step;
                }else
                {
                        my->z = startheight+250;
                }
        }else
        {
                if(my->z > startheight)
                {
                        my->z -= 10*time_step;
                }else
                {
                        my->z = startheight;
                }
        }

        wait(1);
    }
}
```

This will already work independant on where the door is placed, but there is still another problem. The origin of the door, we compare the camera position with, moves away with the door and it is just a point which in case of the example is even placed at the bottom of the door. To keep things simple, we will just make sure that the position we compare the camera position with, is in the center of the door at a height specified by us and that it won´t move with the door. For this, we have to get the size of the door using the `min_xyz` and `max_xyz` parameters each entity has:

```
action DoorSlideUp()
{
        var startheight;
        var distance;

        //Update the collision hull
        c_setminmax(my);

        //The "sensor" position
        VECTOR center;

        /*Calculated from the bounding boxes min and max values
        as the half of their sum is the offset from the origin
        to the objects center.
        This offset is rotated with the object and the objects
        position is added. The height is user defined, relative
        to the objects lowest point.*/
        center.x = (my->min_x+my->max_x)*0.5;
        center.y = (my->min_y+my->max_y)*0.5;
        center.z = my->min_z+200;
        vec_rotate(&center, &my->pan);
        vec_add(&center, &my->x);

        //Store the starting height
        startheight = my->z;

        while(1)
        {
                //Get the distance between camera and door
                distance = vec_dist(&camera->x, &center);

                //Open the door if the camera is close
                if(distance < 200)
                {
                     if(my->z < 250+startheight)
                     {
                          my->z += 10*time_step;
                     }else
                     {
                          my->z = startheight+250;
                     }
                }else//close it otherwize
                {
                     if(my->z > startheight)
                     {
                          my->z -= 10*time_step;
                     }else
                     {
                          my->z = startheight;
                     }
                }

                wait(1);
           }
     }
```

That already is a fully working first door. As you can see, I made comments in my code above. Everything between `/*` and `*/` is ignored by the compiler as well as everything behind `//` until the end of the line.

Customizing

You can probably imagine, that this code is problematic, if we have a level with a lot of those doors, when they should stop after moving different distances, or if thei should be triggered at different camera distances and so on, as the way it is, we would have to write a different action each time. The solution are skills which are 100 free to use parameters, which are part of each entity. This means, that if we would create our doors at runtime, we could just set some of those skills to the needed values. But we don´t want to that and because we are the programmers and not good at adjusting such things, there is the possibility to have the level designer enter those values to the first 20 skills from within WED, which you may noticed when assigning the action. To keep it clean, it is even possible to change the name of those skills they are displayed with in WED.

This can be archieved by adding some special comments above an action. The command has to look like this:

```
     //skillN: Name Value
```

Where N is the skill number, Name is the name we want it to be displayed with and Value is the Value automatically assigned to it in WED. There are many more comment tags like this. You can find a list of them including their description in the manual: http://www.conitec.net/beta/customscript.htm.

Applied to our current code, it could look like this:

```
     //skill1: TriggerDist 200
     //skill2: TriggerHeight 200
     //skill3: MoveHeight 250
```

```
//skill4: MoveSpeed 10
action DoorSlideUp()
{
      ...
}
```

So just put those lines on top of your action and open the level in WED again, if you closed it (make sure that it is still in the same folder as *Main.c*!).
You can now just select our first door and reasign its action and you will see that our names are now displayed instead of skill1-3 and that they have got the default value assigned. If you rightclick the door and directly choose "Behaviour", you will see that only our renamed skills are displayed. And as it is possible to add headlines and descriptions and so on, this is what you may want to design your scripts for.
Save and build the level.

Now open *Doors.c* again, as we of course still have to use those skills. They can be accessed just like the position components:

```
my->skill1 = my->skill100;
```

There is also the sometimes nicer way to access them as an array like this:

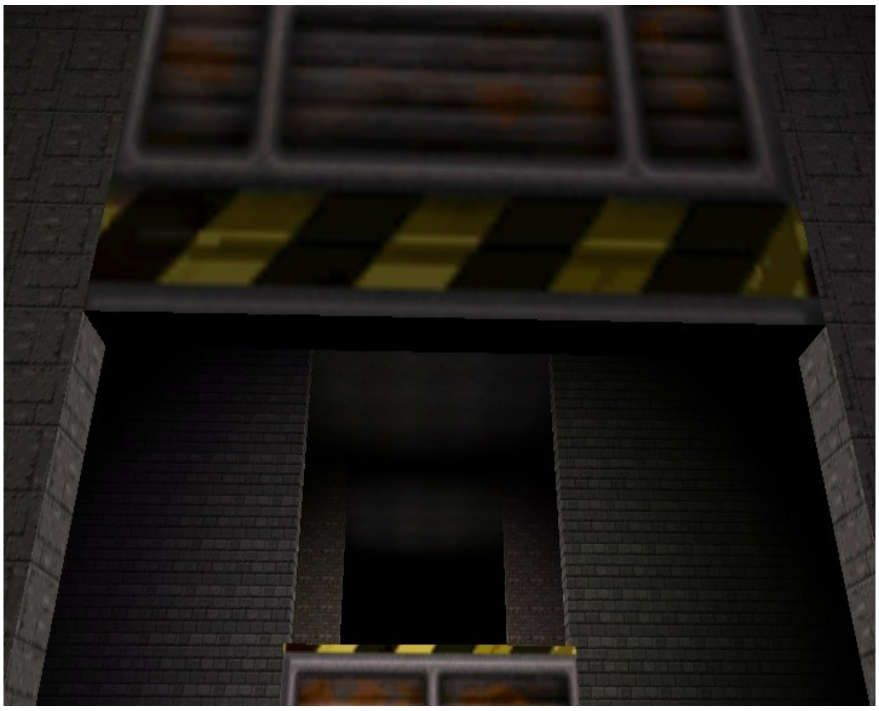
```
my->skill[0] = my->skill[99];
```

So we just have to replace our hardcoded values with those skills and our first door is ready:

```
//skill1: TriggerDist 200
//skill2: TriggerHeight 200
//skill3: MoveHeight 250
//skill4: MoveSpeed 10
action DoorSlideUp()
{
      var startheight;
      var distance;

      //Update the collision hull
      c_setminmax(my);

      //The "sensor" position
      VECTOR center;

      /*Calculated from the bounding boxes min and max values
      as the half of their sum is the offset from the origin
      to the objects center.
      This offset is rotated with the object and the objects
      position is added. The height is user defined, relative
      to the objects lowest point.*/
      center.x = (my->min_x+my->max_x)*0.5;
      center.y = (my->min_y+my->max_y)*0.5;
      center.z = my->min_z+my->skill2;
      vec_rotate(&center, &my->pan);
      vec_add(&center, &my->x);

      //Store the starting height
      startheight = my->z;

      while(1)
      {
```

```
                //Get the distance between camera and door
                distance = vec_dist(&camera->x, &center);

                //Open the door if the camera is close
                if(distance < my->skill1)
                {
                        if(my->z < my->skill3+startheight)
                        {
                                my->z += my->skill4*time_step;
                        }else
                        {
                                my->z = startheight+my->skill3;
                        }
                }else//close it otherwize
                {
                        if(my->z > startheight)
                        {
                                my->z -= my->skill4*time_step;
                        }else
                        {
                                my->z = startheight;
                        }
                }

                wait(1);
        }
}
```

There are of course still ways to improve it. One could for example make it always move to the maximum height, even if the camera is again far away. Something else could be some smooth movement, by weather accelerating the movement speed or for example using the sine function.
But I leave those things to you if you want them ;).



Door sliding to the side

This next door meant to move in any direction we want. This is very similar to the previous one, but involves some more maths, as we of course don´t want to place all doors according to the coordinate axes and just move them along those.

Getting started

Just copy and paste the previous code to the DoorSlide action and remove the movement parts:

```
	action DoorSlide()
	{
		var startheight;
		var distance;
		
		//Update the collision hull
		c_setminmax(my);
		
		//The "sensor" position
		VECTOR center;
		center.x = (my->min_x+my->max_x)*0.5;
		center.y = (my->min_y+my->max_y)*0.5;
		center.z = my->min_z+my->skill2;
		vec_rotate(&center, &my->pan);
		vec_add(&center, &my->x);
		
		//Store the starting height
		startheight = my.z;
		
		while(1)
		{
			//Get the distance between camera and door
			distance = vec_dist(&camera->x, &center);
			
			//Open the door if the camera is close
			if(distance < my->skill1)
			{
			
			}else//close it otherwize
			{
			
			}
			
			wait(1);
		}
	}
```

As we already use some skills here, we should already have them displayed with a name in WED.
We of course still need the camera distance to trigger the door with and the height of the center. We should also again have parameter on how far the door is meant to move and we have to set a movement speed. Nothing new, but this time we want to have it move in any direction, so we need a direction. We could of course add three more values each for the direction along one of the axes. But as We often want to move such a door along one of its local axes, I prefer the way to already combine the direction with the speed. This means that we get rid of the speed parameter and have three velocity values instead:

```
	//skill1: TriggerDist 200
	//skill2: TriggerHeight 200
```

```
//skill3: MoveRange 200
//skill4: VelocityX 0
//skill5: VelocityY 10
//skill6: VelocityZ 0
```

This will move the door to the side by default if we would set `VelocityY` to -10 it would move to the other side. `VelocityX` would move it forward and backward and `VelocityZ` would move it up and down.

Save, reassign the action in WED and build the level.

Let it slide (again)

As we of course need the starting position again, wich can be any this time, just storing the height isn´t enough. Because of this, we have to replace this line:

```
//Store the starting height
startheight = my->z;
```

with something like this:

```
//Store the starting position
VECTOR startpos;
vec_set(&startpos, &my->x);
```

`vec_set` just copies the values of the second `VECTOR` to the first `VECTOR`. Next, we have to transform the local velocity to world space, the coordinate system of the level. As we don´t want it to scale or move with our model (it is just the velocity after all) we just have to rotate it with our model. Happily gamestudio already provides the amazing (I am really tired of writing that on my own in other game engines) `vec_rotate` function. It does nothing else but rotating the vector around the origin, by the given euler angles. The euler angles are in our case the pan, tilt and roll values of the door:

```
//Rotate the velocity vector
vec_rotate(&my->skill4, &my->pan);
```

Now we just have to scale that vector with time_step and add it to the doors current position, until the distance between the current position and the starting position is bigger than the door is allowed to move. To close the door again, we just have to substract instead of adding. To make sure that it perfectly reaches its destination positions, we will directly put it there if it is close, as we already did before:
To scale the velocity, we have to declare an other vector.

```
//skill1: TriggerDist 200
//skill2: TriggerHeight 200
//skill3: MoveRange 200
//skill4: VelocityX 0
//skill5: VelocityY 10
//skill6: VelocityZ 0
action DoorSlide()
{
	var startheight;
	var distance;

	//Update the collision hull
	c_setminmax(my);

	//The "sensor" position
	VECTOR center;
	center.x = (my->min_x+my->max_x)*0.5;
```

```
          center.y = (my->min_y+my->max_y)*0.5;
          center.z = my->min_z+my->skill2;
          vec_rotate(&center, &my->pan);
          vec_add(&center, &my->x);

          //Store the starting position
          VECTOR startpos;
          vec_set(&startpos, &my->x);

          //Rotate the velocity vector
          vec_rotate(&my->skill4, &my->pan);

          VECTOR scaledvel;

          while(1)
          {
               //Get the distance between camera and door
               distance = vec_dist(&camera->x, &center);

               //Open the door if the camera is close
               if(distance < my->skill1)
               {
                    if(vec_dist(&startpos, &my->x) < my->skill3)
                    {
                         vec_set(&scaledvel, &my->skill4);
                         vec_scale(&scaledvel, time_step);

                         vec_add(&my->x, &scaledvel);
                    }else
                    {
                         vec_set(&scaledvel, &my->skill4);
                         vec_normalize(&scaledvel, my->skill3);
                         vec_add(&scaledvel, &startpos);
                         vec_set(&my->x, &scaledvel);
                    }
               }else//close it otherwize
               {
                    if(vec_dist(&startpos, &my->x) > vec_length(&my-
>skill4)*time_step)
                    {
                         vec_set(&scaledvel, &my->skill4);
                         vec_scale(&scaledvel, time_step);

                         vec_sub(&my->x, &scaledvel);
                    }else
                    {
                         vec_set(&my->x, &startpos);
                    }
               }

               wait(1);
          }
     }
```

The best way to understand those `vec_` functions, is in my opinion to just use them and see what they do.

But you should also look them up in the holy manual, for a brief description.

A rotating door

This last door is meant to rotate around its origin into a user defined direction. Just like the previous one, but with a rotation.

Coding it

You can hopefully imagine that this is very similar to the previous door, but with a rotation instead of a translation. Because of that we can use the previous code to start with and change all the parts setting a position to the angles instead:

```
//skill1: TriggerDist 200
//skill2: TriggerHeight 200
//skill3: MoveRange 120
//skill4: VelocityPan 10
//skill5: VelocityTilt 0
//skill6: VelocityRoll 0
action DoorRotate()
{
	var startheight;
	var distance;

	//Update the collision hull
	c_setminmax(my);

	//The "sensor" position
	VECTOR center;
	center.x = (my->min_x+my->max_x)*0.5;
	center.y = (my->min_y+my->max_y)*0.5;
	center.z = my->min_z+my->skill2;
	vec_rotate(&center, &my->pan);
	vec_add(&center, &my->x);
```

```
        //Store the starting rotation
        VECTOR startrot;
        vec_set(&startrot, &my->pan);

        //Rotate the velocity vector
        vec_rotate(&my->skill4, &my->pan);

        VECTOR scaledvel;

        while(1)
        {
            //Get the distance between camera and door
            distance = vec_dist(&camera->x, &center);

            //Open the door if the camera is close
            if(distance < my->skill1)
            {
                if(vec_dist(&startrot, &my->pan) < my->skill3)
                {
                    vec_set(&scaledvel, &my->skill4);
                    vec_scale(&scaledvel, time_step);

                    vec_add(&my->pan, &scaledvel);
                }else
                {
                    vec_set(&scaledvel, &my->skill4);
                    vec_normalize(&scaledvel, my->skill3);
                    vec_add(&scaledvel, &startrot);
                    vec_set(&my->pan, &scaledvel);
                }
            }else//close it otherwize
            {
                if(vec_dist(&startrot, &my->pan) > vec_length(&my-
>skill4)*time_step)
                {
                    vec_set(&scaledvel, &my->skill4);
                    vec_scale(&scaledvel, time_step);

                    vec_sub(&my->pan, &scaledvel);
                }else
                {
                    vec_set(&my->pan, &startrot);
                }
            }

            wait(1);
        }
    }
```

Now reassign the third doors action, rebuild and run it.
I am actually not sure, if this works correctly in every situation as the previous one, but at least it does it job in the test level, so that you hopefully get the idea of how it works.

The end

Now, clean up your folders by putting the wmp and wmb file back into the Levels folder and delete all the others but *Main.c*.

Homework

You could try to melt the last two codes to one über-door script, which allows all kind of doors one could think of. So just one action, which combines the sliding and the rotating door.

Review and Preview

This tutorial actually took a lot of time to write and didn´t turn out as I wanted it to, as my idea to keep things as exact and flexible as possible resulted in code which isn´t as easy to understand as it should be for the beginning. So just don´t be too scared, if you didn´t understand every line of my example codes. Important is just, that you understand the idea behind actions and how to solve simple problems like doors. If and else is also a very important aspect of course.

In the next tutorial, I want to explain, how to write a basic playermovement and a first and third person camera.

I hope that even though I am not as happy with this tutorial as with the previous one, that it is helpfull,
Nils Daumann.