Lite-C and A8

Part 3: Player and Camera

Introduction

Hello to the third part of these tutorials. In this tutorial I will show you how to move a player including gravity and jumping and how to create a simple first person camera. I actually decided to skip the third person camera.
The provided ressources are the result of the previous tutorial. But I added a player model, added it to the level and assigned the aPlayer action to it, wich is defined in the newly added Player.c, which is already included in main.c. The players skills are also already set to the values we will use in this tutorial, so that we can completely focus on the code.
Credits for the barbarian model go to Ryan Gregory, who was so kind to contribute this great model for free

Input

You should already know how to move and rotate an object by changing its position and rotation parameters, but how to do this related to the users input? Gamestudio happily provides several extremely easy ways for this.

Getting keyinput

There is a function called `inchar()`, which stops the application until a key is pressed and then writes the pressed character into the given char. As this stops the application, it isn´t really suited for movements, but can do a good job, if you want to have the user enter texts. But as it always only provide one character, there is an easier solution for this as well, provided by the `inkey()` function, which writes characters into a string until the uses presses enter.
What we want now instead is the function `key_pressed()`. It works for all standard buttons on your keyboard, your mouse buttons and even for joystick buttons and just tells you for a specific number, representing a one of those buttons, if it is pressed or not by returning 1 or 0. This is what you usually want to use for a real game with customizable key mapping. You can find all buttons "scancodes" in the manuals

key mapping chapter. The alternative is to hardcode the keys, which is what we do for this tutorial to keep things a bit easier to understand. For this, Gamestudio provides a variable for each key, which tells us if it is pressed or not. This variable is called key_name, where name is the name of the key. The "key" specifies that it is a keyboard button in this case, it is called different for mouse buttons. In this tutorial, we will use `key_w`, `key_a`, `key_s` and `key_d` for the movement and `key_space` for jumping.

Getting mouse movement

We can just directly access the mouse movement speed through the `mouse_force` struct which has just an `x` and a `y` component.
`key_force.x` does thus for example tell you how fast the mouse is currently moving horizontally.

Basic movement

Open Main.c and Player.c as we now want to get started with some first movement along the coordinate axes based on our input. All the following code is meant to be put into the aPlayer action.

First movement!

As we want to check the pressed keys every frame again, as well as doing the movement, we start with an endless `while` loop with a `wait(1)` in it and declare a vector, which is meant to store our movement direction:

```
VECTOR movedir;
while(1)
{
	wait(1);
}
```

We could now just check our key_ variables with `if`s, but these variables provide a much nicer solution:

```
movedir.x = key_w-key_s;
movedir.y = key_a-key_d;
movedir.z = 0;
```

We aren´t going to use the z component.
As we now have our wished direction based on the user input, we just have to multiply it with our speed as well as with time_step to make it framerate independant and add the vector to our players position:

```
VECTOR movedir;
while(1)
{
	movedir.x = key_w-key_s;
	movedir.y = key_a-key_d;
	movedir.z = 0;

	vec_scale(&movedir, my->skill1*time_step);
	vec_add(&my->x, &movedir);

	wait(1);
}
```

Test this code by running the Main.c now. You should already be able to move your player around with WASD!

Rotation

Now we want to use our mouse to rotate the player.
This is even easier than the translation, as we just have to multiply `mouse_force.x` with our speed and with time_step and substract (because `mouse_force.x` is flipped, adding would be the wrong way around) it from the players pan value:

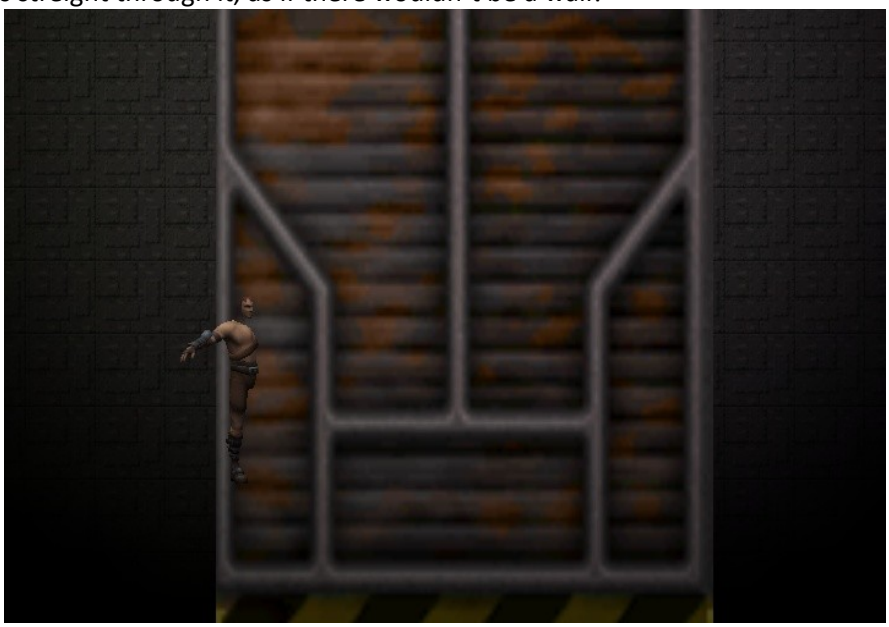
```
VECTOR movedir;
while(1)
{
	movedir.x = key_w-key_s;
	movedir.y = key_a-key_d;
	movedir.z = 0;

	my->pan -= mouse_force.x*my->skill2*time_step;

	vec_scale(&movedir, my->skill1*time_step);
	vec_add(&my->x, &movedir);

	wait(1);
}
```

Believe it or not, but that is already everything. You should test the code again now.
You will notice that the rotation works fine, but that he doesn´t go into the direction he looks.
The reason is, that we are just adding our movement to our models position in the global coordinate system, but actually want to change the position relative to our model.
To change this, we have to apply the players current rotation to the direction vector:

```
VECTOR movedir;
while(1)
{
	movedir.x = key_w-key_s;
	movedir.y = key_a-key_d;
	movedir.z = 0;

	my->pan -= mouse_force.x*my->skill2*time_step;

	vec_rotate(&movedir, &my->pan);
```

```
        vec_scale(&movedir, my->skill1*time_step);
        vec_add(&my->x, &movedir);

        wait(1);
    }
```

If you test the code again, you will notice that this really fixed the problem.
But if you try to move against a wall you will notice another problem.
The player moves streight through it, as if there wouldn´t be a wall!



Collision detection

That is of course what we are still missing and gamestudio happily provides us a couple of very handy functions for this.
For translation, there is a function called `c_move(ENTITY* entity,VECTOR* reldist,VECTOR* absdist,var mode)`, which wants our entity pointer as first parameter, followed by a vector defining the relative direction we want to move.
The `absdist` parameter is another movement direction, but this time applied within the absolute coordinate system. The `mode` parameter wants some flags to specify some special behaviours regarding the collision detection, like for example ignoring some special objects.
First we concentrate on the `reldist` and `absdist` vectors now. Instead of adding our direction to the players position, we can now just call `c_move()` with our direction as `absdist` vector:

```
    VECTOR movedir;
    while(1)
    {
        movedir.x = key_w-key_s;
        movedir.y = key_a-key_d;
        movedir.z = 0;

        my->pan -= mouse_force.x*my->skill2*time_step;

        vec_rotate(&movedir, &my->pan);
        vec_scale(&movedir, my->skill1*time_step);
        c_move(my, nullvector, &movedir, GLIDE);
```

```
        wait(1);
    }
```

Try it if you want, but what we want to do now instead is to use the `reldist` vector instead. This takes us the need to rotate our vector, as `c_move()` will do so for us:

```
    VECTOR movedir;
    while(1)
    {
        movedir.x = key_w-key_s;
        movedir.y = key_a-key_d;
        movedir.z = 0;

        my->pan -= mouse_force.x*my->skill2*time_step;

        vec_scale(&movedir, my->skill1*time_step);
        c_move(my, &movedir, nullvector, GLIDE);

        wait(1);
    }
```

Now run this code.
It should actually do a great job, but will still allow you to rotate your player into the wall, which can cause troubles in some circumstances.

The solution is the `c_rotate()` function, which just takes our entity pointer, our angle and a flag:

```
    VECTOR movedir;
    while(1)
    {
        movedir.x = key_w-key_s;
        movedir.y = key_a-key_d;
        movedir.z = 0;

        c_rotate(my, vector(-mouse_force.x*my->skill2*time_step, 0,
0), GLIDE);

        vec_scale(&movedir, my->skill1*time_step);
        c_move(my, &movedir, nullvector, GLIDE);

        wait(1);
    }
```

Flags

They are actually a bit tricky to understand but very easy to use. If you for example want a `c_move()` to make your player ignore passable objects and glide along walls, you can easyly combine the two needed flags with the `|` sign like this:

```
    c_move(.., GLIDE|IGNORE_PASSABLE);
```

You can combine as many flags as you want this way.
Many gamestudio objects like entities have a `flags` variable, which you best change using the `set()` and reset() macros. Which don´t do anything else than `ent->flags |= yourflags;` or `ent->flags &= ~yourflags;`
What you actually do with those strange signs is, that you change single bits of the flag variable. The smalest datatype is a char, which consists of one byte or 8 bits. Each bit can be on or off, 1 or 0, true or false and the more bits the datatype consists of, the more values it can represent. As a char has 8 bit, it can store

2^8 = 256 values.
The representation of 0 is 00000000, while the representation of 1 is 10000000 and the representation of 2 is 01000000. If you now do something like char flag = 1|2; the result will be 11000000 which represents 3 in this case. This way you can set single bits. Which saves some space, if you need many values just meant to represent 1 or 0.
Just to give you a basic idea.

Gravity and jumping

This is the part where most mistakes happen and I hope to show you a correct and easy way to do it.
The jumping is going to be easy, but we need the gravity to work first, which is a bit more tricky.

Free fall

A falling object accelerates with 9.81m/s² on the earth. This means that the falling speed is growing in a linear way and can then be added to the position which causes a quadratic position change when time is going on.
In our code this means that as long as the players feet don´t touch the ground, we have to accelerate the movement speed towards the ground and reset the speed to zero as soon as we reached the ground:

```
var gravity = 0;

gravity -= my->skill4*time_step;
c_move(my, nullvector, vector(0, 0, gravity*time_step), GLIDE);
```

While this already describes our needed movement very well, we still have to check the distance to the ground. We can do so using `c_trace(VECTOR* from, VECTOR* to, var mode);` which should already be selfexplaining. This function just sends a ray from the `from` position to the `to` position and returns the distance from the `from` position to a obstacle, if there is one, 0 otherwize. It also sets some predefined variables, which can be very handy in some cases. But in our case, the returned distance between our player and and a position far below it is enough:

```
var gravity = 0;
var grounddist = 0;

grounddist = c_trace(&my->x, vector(my->x, my->y, my->z-100000),
USE_BOX|IGNORE_ME);
if(grounddist > 0)
{
	gravity -= my->skill4*time_step;
}else
{
	gravity = 0;
}
c_move(my, nullvector, vector(0, 0, gravity*time_step), GLIDE);
```

But this does still cause troubles, if the ground wants us to move up a bit.

The bounding box

You hopefully noticed, that the `c_trace()` above uses `USE_BOX` as mode. This causes our trace to not be an endlessly thin line, but a bigger volume instead, based on our objects bounding box.
This bounding box is usually set automatically by gamestudio and used for collision detection when using `c_move()` or `c_rotate()` as this is faster and more stable than polygonal collision. While the bounding box actually isn´t a box anymore, we can still adjust it as if it was a box. What we want to do now is, to

adjust it to our models size and to get rid of it at the leggs. This is going to make our trace start from our models origin instead of from the feet, will nearly always return a distance to the ground and even tells us the height of an obstacle below our player. This speeds up the `c_move()` as it doesn´t always has to do the gliding above the ground and will make it easy for us to move the player up, if there is an obstacle as we know the height.
To set the bounding box to our models current size, we just have to call `c_setminmax()`.
To adjust the bounding box, we can directly our entities `min_x`, `min_y`, `min_z`, `max_x`, `max_y` and `max_z` variables. In our case we just want to know the height of the origin above the ground, which is `-min_z` and to set `min_z` to 0 afterwards:

```
	c_setminmax(my);
	var feetheight = -my->min_z;
	my->min_z = 0;
```

This makes our player code look like this:

```
	VECTOR movedir;
	var gravity = 0;
	var grounddist = 0;
	var feetheight;

	c_setminmax(my);
	feetheight = -my->min_z;
	my->min_z = 0;

	while(1)
	{
		movedir.x = key_w-key_s;
		movedir.y = key_a-key_d;
		movedir.z = 0;

		c_rotate(my, vector(-mouse_force.x*my->skill2*time_step, 0,
0), GLIDE);

		vec_scale(&movedir, my->skill1*time_step);
		c_move(my, &movedir, nullvector, GLIDE);

		grounddist = c_trace(&my->x, vector(my->x, my->y, my->z-
100000), USE_BOX|IGNORE_ME)-feetheight;
		if(grounddist > 0)
		{
			gravity -= my->skill4*time_step;
		}else
		{
			gravity = grounddist;
		}
		c_move(my, nullvector, vector(0, 0, gravity*time_step),
GLIDE);

		wait(1);
	}
```

This is already very close to what we want to archieve, but isn´t going to work yet correctly, as you will notice if you run it.

Fixing the gravity and cleaning up

The first problem is, that our player is meant to move further down than it actually can and our bounding box doesn´t fit the model anymore, as we changed it. To solve this, we are now going to restrict our applied gravity to the possible maximum using `minv()`. This also removes the need for the c_move() to apply the gravity:

```
	grounddist = c_trace(&my->x, vector(my->x, my->y, my->z-
100000), USE_BOX|IGNORE_ME)-feetheight;
	if(grounddist > 0)
	{
		gravity += my->skill4*time_step;
	}else
	{
		gravity = 0;
	}
	my->z -= minv(gravity*time_step, grounddist);
```

Test run it and everything should work.

The other promises

Jumping

All we have to do is to set the gravity to our jump strength if our player is on the ground. Everything else is handled by the gravity:

```
	if(grounddist > 0)
	{
		gravity += my->skill4*time_step;
	}else
	{
		gravity = 0;

		if(key_space)
		{
```

```
                gravity = my->skill5;
            }
        }
```

The only problem of this is, that it is going crazy as soon as the player hits an obstacle placed above it. But you should be able to solve this yourself now with `c_trace()` or `c_move()`.

Running

If shift is pressed, we just multiply our movement speed by the run factor:

```
if(key_shift)
    vec_scale(&movedir, my->skill3);
```

Camera

Believe it or not, but a basic first person camera also isn´t hard to code. All we need to do is to place the camera at our players position with a little height offset and set it to the same rotation:

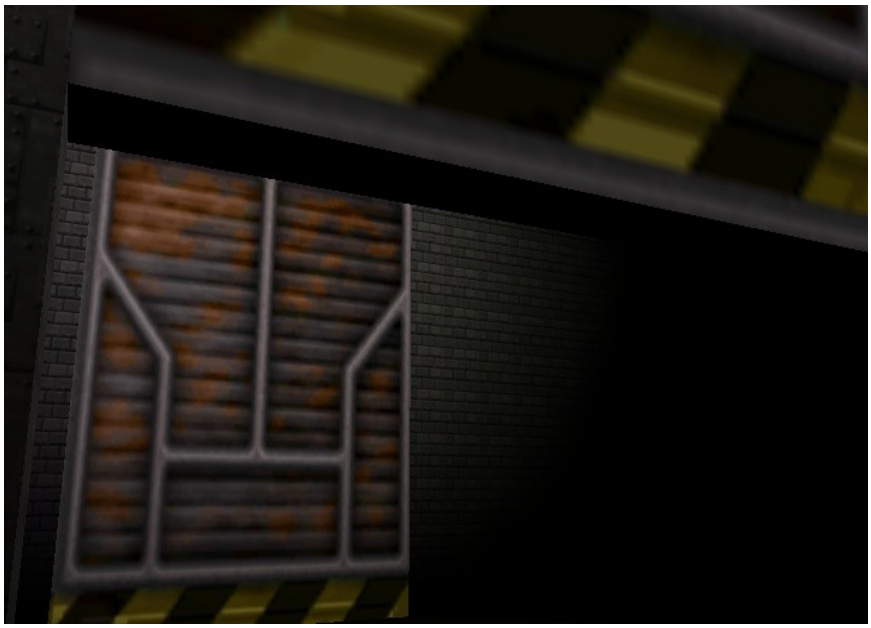
```
vec_set(camera->x, vector(my->x, my->y, my->z+my->skill6));
vec_set(camera->pan, my->pan);
```

But because this does not allow us to look up and down, we only set the cameras pan to the players and set the tilt independant of the player but depending on the mouse movement. We should also restrict the tilt to 90 degrees or something similar:

```
vec_set(camera->x, vector(my->x, my->y, my->z+my->skill6));
camera->pan = my->pan;
camera->tilt += mouse_force.y*my->skill2*time_step;
camera->tilt = clamp(camera->tilt, -90, 90);
```

As we don´t want to see the body, we also set `camera->genius` to the player.

```
vec_set(camera->x, vector(my->x, my->y, my->z+my->skill6));
camera->pan = my->pan;
camera->tilt += mouse_force.y*my->skill2*time_step;
camera->tilt = clamp(camera->tilt, -90, 90);
camera->genius = my;
```



The end

This is what my final player action looks like:

```
	//skill1: WalkSpeed 16
	//skill2: TurnSpeed 16
	//skill3: RunFac 2
	//skill4: Gravity 10
	//skill5: JumpStrength 15
	//skill6: CamOffset 15
	action aPlayer()
	{
		VECTOR movedir;
		var gravity = 0;
		var tgravity = 0;
		var grounddist = 0;
		var feetheight;
		var jump = 0;

		c_setminmax(my);
		feetheight = -my->min_z;
		my->min_z = 0;

		while(1)
		{
			movedir.x = key_w-key_s;
			movedir.y = key_a-key_d;
			movedir.z = 0;

			if(key_shift)
				vec_scale(&movedir, my->skill3);

			c_rotate(my, vector(-mouse_force.x*my->skill2*time_step,
0, 0), GLIDE);

			vec_scale(&movedir, my->skill1*time_step);
			c_move(my, &movedir, nullvector, GLIDE);

			grounddist = c_trace(&my->x, vector(my->x, my->y, my->z-
100000), USE_BOX|IGNORE_ME)-feetheight;
			if(grounddist > 0)
			{
				gravity += my->skill4*time_step;
			}else
			{
				gravity = 0;

				if(key_space)
				{
					gravity = my->skill5;
				}
			}
			my->z -= minv(gravity*time_step, grounddist);

			vec_set(camera->x, vector(my->x, my->y, my->z+my-
>skill6));
```

```
            camera->pan = my->pan;
            camera->tilt += mouse_force.y*my->skill2*time_step;
            camera->tilt = clamp(camera->tilt, -90, 90);
            camera->genius = my;

            wait(1);
        }
    }
```

Homework

You could make the camera shake a little while moving. You could also add stamina for running. Be creative ;).

Review and Preview

This tutorial hopefully explained the basic concepts behind a basic player code.
I am not sure if I am gong to continue these tutorials and in case I do, if I should explain sounds, animations and panels or if it would be better to write something more detailed about some important mathematic topics.

I really hope that this tutorial is helpfull, as I have the feeling that it lacks some explanations...
Nils Daumann.