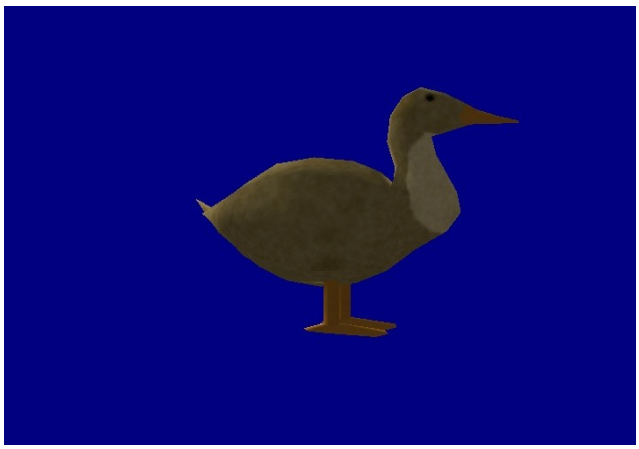
Lite-C and A8

Part 1: Dry Introduction



Introduction

The target of this first part is to make you understand some basics about including files, functions, variables, pointers and some very basics of A8.

Start by downloading the project I prepared (http://www.slindev.com/files/tutorials/LiteCTutPart1.zip). Unzip it and open Main.c with SED. If you don´t own GameStudio, you can download a free version at www.3dgamestudio.com.

Functions

In Lite-C, programs consist of functions which are executed from command to command and can call each other. As each program needs to start somewhere, there is a special function called main, which is executed before everything else and can call other functions to get executed.

"Hello World"

A first program could look like this (just give it a try to see that it really works, by copy and pasting the lines into the opened file in SED and click at the "run" button (a black arrow)):

```
void main()
{
	printf("Hello World");
	sys_exit("");
}
```

Lets look at the first line:

```
void main()
```

More generalized, this could look like this:

```
returntype name(type parameter1, ...)
```

Where `returntype` specifies the type of what the function returns, name is the name of the function,

type is the type of the first parameter and `parameter1` is the name of the first parameter.
The `returntype` in this case is `void`, which means that nothing is returned. The name is `main` and there are no parameters. The braces `{}` mark the body of the function, which defines what the function does. In the example above, two functions are called, where the first one opens a dialog box with the text "Hello World" and the second one shuts down the program.

A first "real" function

Now, we want to write a program which defines and uses a function that adds two numbers.
I am calling that new function `add` and as it is meant to return a number, we use A8s standart type for numbers called `var` as returntype. We will need two parameters, which I will call `a` and `b`, both of the type `var`.
The result looks like this:

```
var add(var a, var b)
{
}
```

Now we of course want the function to return the sum of the two parameters. To return a value or variable, there is the command `return value`. To add two values or variables, there is the `+` operator:

```
var add(var a, var b)
{
	return a+b;
}
```

As you may noticed, each operation has to be closed by a semicolon. If we now want to use this new function within the main function, we just have to use the function´s name, put the value we want to assign to the parameters between the brackets and close it with a semicolon:

```
void main()
{
	add(1, 2);
}
```

It is of course also possible to use the result of our addition in other operations. For example, we can multiply it by two and show the result:

```
void main()
{
	var res = add(1, 2)*2;
	printf("Result: %.0f", (double)res);
	sys_exit("");
}
```

I am not getting into detail about `printf(...)` and `sys_exit(...)`, as especially `printf` is a bit special. I am just using them to illustrate that our function actually works.

Variables

In this last function, we have a first variable called res that looks and works very similar to the parameters a and b from above. A variable declaration in general looks like this:

```
type name;
```

`type` specifies what kind of variable it is. For numbers this is `var`, as seen above. `Void` cannot be used as the type of a variable. It is possible to use a variable of the type `var` in the same way as plain numbers:

```
void main()
```

```
{
	var x = 2;
	var y = 3;
	var z = 4;
	var res = add(x, y)*z;
	printf("Result: %.0f", (double)res);
	sys_exit("");
}
```

And the value of a variable can be changed anytime:

```
void main()
{
	var x = 2;
	var y = 3;
	var z = 4;
	var res = add(x, y)*z;
	res = 5;
	printf("Result: %.0f", (double)res);
	sys_exit("");
}
```

How is this code processed?

To understand this last programm a bit better, you have to imagine, how it gets processed (which isn´t perfectly right, but works very well for understanding the basics):

At first, the compiler opens the file and reads it from left to right and from the top to the bottom. The compiler translates the code to into a format that the computer "understands".
It starts with our first line, `var add(var a, var b)`, notices that it is a function and translates what it does into machine code. Next it finds `void main()`, identifies it as a function, in this case a very special one, one that the computer starts the actual program with, and then goes on to process its body. There it eventually reaches the line `var res = add(x, y)*z;` and identifies that there is a call of the function `add(...)`. It looks up the already translated functions, finds add, and tells the computer to change to this function at this point, process its body, and then return to main to continue with the next instruction as soon as it is done with processing `add( )`. It goes on in this manner until it reaches the end of the file. This also means that if you would put the `add( )` function under function `main()`, the compiler wouldn´t know it within `main()` and thus, would give you an error message.

If it was compiled successfully, you can run it. The computer looks for function `main()` and starts processing it from top to bottom. If it reaches the end of `main()`, the programm is done.

It starts with the first line `var x = 2;`, takes a block of the systems memory and sets the bytes to represent the value 2. This block can now be accessed through the variable name x. The same is done for the next two lines. In the fourth line `var res = add(x, y)*z;`, it also takes a part of the memory and three for `var add(var a, var b)`. `a` and `b` are set to `x` and `y` and the other part is set to the returned value of the function.
This part is now multiplied with z and copied to res, so that res finally contains our result. After that, res is set to 5, the value of res is displayed (which is of course 5) and the engine is shut down.

Pointers

If you pass a variable to a function through parameters, the value is copied. This means that a change of the parameters value doesn´t change the variable passed to the function.
If you understand this, the next, very important thing are pointers. As written above, the computer reserves

a memory block for each variable. This block of memory has a fixed address, which allows us to read and change the value of the variable, similar to using the name. This address can be stored in another variable, which is then called a pointer, as it points to the block of memory.

If we don´t want to use a return value for the add function above, we can change it using a pointer to the following:

```
void add(var *res, var a, var b)
{
	*res = a+b;
}
```

As you hopefully see, the declaration of a pointer always looks like this:

```
type *name;
```

Where `type` is the type of the variable the pointer points to, `*` marks it as a pointer and and `name` is the name of the pointer that will be used to refer to it later.

In the functions body, we have `*res`. Here, `*` tells the computer to access the memory block the pointer points to. Without the asterisk, the pointer itself would get a new value and thus point to a different memory block. A main function for this program could look like this:

```
void main()
{
	var res;
	add(&res, 4, 6);
	printf("Result: %.0f", (double)res);
	sys_exit("");
}
```

It declares a variable of the type `var` with the name `res` and then calls the add function. The second and third parameters are the same as before, but the first one is new. `&` is the oppisite of `*` above. It gives us the address of the variable that follows. This means, that the pointer in `add( )` gets passed the address of the variable `res`, and the function then sets a new value for `res`, which is then output in `main`.

Pointers are one of the most important things to understand in my opinion, so please try to understand what I wrote :). Read it over again if you are unsure.

Arrays

If you want to store many values in variables, it is a lot of work to declare a new variable for each of them, especially for example, if you want to read the values from a file and they can have a variable count. For something like this it is important to understand the concept of arrays.

An array is a connected block of memory which contains memory blocks for several variables. The declaration of an array looks like this:

```
type name[size];
```

Where `type` is again the type of the values you want to store, `size` is the name of the array and `number` is the number of elements this array is meant to have. This number cannot be a variable.

The different values stored in this array can be accessed like this:

```
name[element]
```

Where `name` is the name of an already declared array and `element` is the number of the element in the

array we want to access, ranging from 0 to `size`-1. `element` can also be outside of this range, but if you don´t exactly know what you are doing, this will cause random crashs.

The array name is nothing but a pointer to the first element of the array, and `[number]` does nothing else but changing the address and giving us the memory block the new address points at. This is important if you want to have arrays of a variable size, but I won´t get into details about this yet.

Strings

We of course don´t want to just store numbers in variables, but also for example text. A bunch of text is nothing more than an array of characters, making up what we call a string. Each character is represented as a number ranging from 0 to 255. This type of variables is called a `char`. This means that the declaration of a string basically just looks like this:

```
char name[100];
```

As this is very uncomfortable and string are used very often, there is a special predefined type called `STRING`. As these strings are meant to be handled by A8, we will usually just work with `STRING`-pointers, which are automatically set by the engine. These pointers are also, what we will usually pass as parameter if a function wants a string. Here is a simple example programm:

```
void main()
{
	STRING* str;
	str = str_create("My first string :)");
	printf(_chr(str));
	sys_exit("");
}
```

After the declaration, `str` has a random value and points to a random position in system memory.
`str_create` then "creates" a string in memory and returns the address.
`str_create` expects a char array as a parameter. A constant, hardcoded char array is marked by quotation marks, in our case it is "My first string".

As noted, `printf` is a bit special. It doesn´t want a `STRING` pointer as parameter, but a pointer to a char array. `_chr` is a function that returns the char array of a `STRING` the given pointer points to.

Working with A8

Now open the file Ents.c in SED, next to Main.c. Define a new function called aRotate(), which isn´t meant to return anything and doesn´t have a parameter. But because gamestudio can´t handle empty functions, put an empty `return` into it:

```
return;
```

Now delete everything from Main.c and add the following lines:

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

The first line lets the compiler know about default functions defined by A8, and the second line implements some other useful and commonly used functions. The third line of course tells the compiler, to also compile our new Ents.c.

The order that these lines are in is very important. A function defined in another file can only be called after the file containing it is included. The # marks this as a special kind of instruction which influences

compiling. The <> make sure that the files are searched for in your gamestudios include folder, while the files with "" are searched for within the current project folder. The first two includes were done automatically before, as we didn´t include a file ourselves.

Using the manual

Now, we actually want to create an empty level with a duck inside, which we will rotate. For this, we first have to define a `main` function, which loads an empty level. The function the engine provides for this is `level_load`.

Look up in the manual on how to use the `level_load` instruction by pressing F1 in SED, selecting the Index tab in the newly opened manual and typing in the keyword: `level_load`. Choose the `level_load` entry from the list. The now displayed description looks like this:

> **level_load (STRING* filename);**
> *Level change; creates an empty level or loads a new level based on a level map (WMB), terrain (HMP) or model (MDL).*
>
> **Parameters:**
> *filename - STRING* or char*; name of the map, model, or terrain file, or **NULL for an empty level**.*

The first line shows the syntax of the function, which in this case means that it needs one parameter of the type STRING*. The parameters section explains what this parameter is expected to be, in our case, NULL. The resulting main function should look like this:

```
void main()
{
	level_load(NULL);
}
```

We still have to load the model. To do so, there is the function `ent_create`. If you look it up in the manual, you will find the following:

> **ent_create(STRING* filename, VECTOR* position, function);**
>
> **Parameters:**
> *filename - name of the entity file to be created, in wmb, hmp, mdl, tga, pcx, bmp, or dds format; or NULL for creating a dummy entity.*
> *position - Inital position of the entity.*
> *function - Action of the entity, or NULL for no action.*

Filename is meant to be a pointer to a string and it has to be the name of the file we want to create our object from:

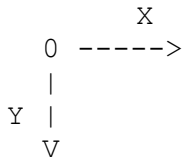
```
void main()
{
	level_load(NULL);

	STRING* filename = str_create("Ente.mdl");
	ent_create(filename, ...);
}
```

Coordinates

The second parameter is meant to be the position within the level that our object will be displayed at. For this to be accurate and repeatable, we need to use some sort of a coordinates system. For 2D, think of the

coordinates system as two rulers. One is aligned to the upper horizontal border of your screen with growing numbers from left to right. 0 is in the top left corner. We will call this the **x-axis**.
The other ruler is aligned to the left vertical screen border, with growing numbers from top to bottom. 0 is again in the top left corner, and we will call this the **y-axis**.



Using this system, it is possible to specify any point on the screen by its unique numerical position. Moving your finger along to the y-axis and x-axis "rulers", until it reaches the point where they meet and write down the number of the rulers at that position will provide a specific target Y-X coordinate.

In order to do the same thing in 3D space, we need to place another ruler orthogonal to the layer stretched by the x and y axis, which has again 0 at the same position as the other rulers. A bit inconsistent, but the way it is actually done in math, the axis looking up (higher position gets a higher value, positions at the other side of the origin, which is the point all axis's meet, get negative values) is called z-axis, the one looking right is called y-axis and the one looking in front of you is called x-axis.

Vectors

This means that our objects position is set by three values, one for each axis. The position has to be of the type `VECTOR`, which is is a structure, which is a bit similar to an array. The difference is, that a stucture can hold variables of different types and can itself be used as a type. For know, it is enough to know how to work with the predefined ones. `STRING` is a structure aswell.

Our position vector can be declared like this:

```
VECTOR pos;
```

And we can set values for each of its elements by using a point followed by the name of the element. The `VECTOR` structure has three elements, called `x`, `y` and `z`:

```
pos.x = 300;
pos.y = 0;
pos.z = 0;
```

This defines a position 300 units away from the camera. As everything is relative, a unit doesn´t have a fixed size, which means that we could for example say that one unit is 10cm or something like that, and scale and place all objects accordingly. As `ent_create` needs a pointer to this vector, we use `&` to pass it.
The third parameter `ent_create` needs, is a pointer to a function, which will be called as soon as the entity is created, or NULL if we don´t want a function to be called. The pointer to a function is just the name, without brackets and parameters, in our case `aRotate`:

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

void main()
{
	level_load(NULL);

	STRING* filename = str_create("Ente.mdl");
	VECTOR pos;
	pos.x = 300;
```

```
        pos.y = 0;
        pos.z = 0;
        ent_create(filename, &pos, aRotate);
}
```

If you compile and run this now, you should see my ugly duck.

Entities

Now look into the empty body of your `aRotate` function. All information about our new object are stored in a structure called `ENTITY`. If a function is called by `ent_create`, it is possible to access the pointer to this structure as its adress will be assigned to a globally accessible pointer called `my`.

Using this pointer, we can get and change the position, rotation and scale of the entity. The position can again be accessed through `x`, `y` and `z`, the scale by `scale_x`, `scale_y` and `scale_z` and the rotation by `pan`, `tilt` and `roll`. As `my` is a pointer, we first have to get the memory block it points at by using `*` and then we can change the `pan` element using `.pan`:

```
(*my).pan = 90;
```

While this works well, it takes many keystrokes to type it, and as it is used very often in code, there is a handy shortcut:

```
my->pan = 90;
```

It does exactly the same as the line above, but is faster to write and looks nicer. As Lite-C is as the name says a light version of C, there are several things making it easier than C. One of those things is the fact that it basicly makes sure that you don´t have to care about the difference between "normal" variables and pointers. This means that you can also write:

```
my.pan = 90;
```

This is what you will find in 99% of the code out there. As I consider it important to know the difference, I will stick to `my->pan` while writing this tutorial to keep things clear.

Endless loops

Now we of course don´t just want the model to start with a different angle but to rotate while we are looking at it. In order to achieve this, we have to add a small value again and again to the models rotation. To repeat something in code, there are loops. There are different kind of loops, but for now I will just explain the very common `while` loop. It looks like this:

```
while(condition)
{
        do something
}
```

This will `do something` until the `condition` is false. As we want to repeat it endlessly, or at least until the programm is shut down, our condition will be 1, which is always true:

```
void aRotate()
{
        while(1)
        {
                my->pan += 1;
        }
}
```

New here is also the += operator, there is also -=, *= and so on. It is just a nice shortcut for:

```
my->pan = my->pan+1;
```

This will definatly rotate the duck, but wait, the programm gets stucked and doesn´t show anything.

The problem is, that only one function at a time can be processed, which means that everything else, including the rendering, is blocked, until this function is done, which never happends thanks to the loop. The solution is the instruction `wait(frames);`. This pauses the running function for the given amount of program cycles, which equals the number of the rendered images, process everything else and goes on in the function at the position it stopped at, in this case, repeating the loop again:

```
void aRotate()
{
	while(1)
	{
		my->pan += 1;
		wait(1);
	}
}
```

Framerate dependence

This works very well, but there is still a problem: The speed of the rotation will be different depending on the speed of the system, as one program cycle on a slow system takes longer than on a fast one. It may also be possible that once in a while a very slow function has to be processed, which could result in changing rotation speeds. The solution is to assume, that the current rendering cycle takes a very similar time to the previous one, so that we can take the time and multiply our rotation speed with it. In A8, this time is even smoothed between several frames, so that it should give us a quite good factor, to get a constant rotation speed. The variable holding the time is called time_step**:**

```
void aRotate()
{
	while(1)
	{
		my->pan += 16*time_step;
		wait(1);
	}
}
```

Compile and run this and you will see a rotating duck! You can press the 0 key and move around freely using the mouse(while holding the right mousebutton down) and WASD on your keyboard. F6 takes a screenshot and F11 opens the debug panel. Quit it by pressing Esc. All this functionality was added by including default.c at the beginning of Main.c.

Cleaning up

Now look again at your Main.c. You will see that it needs many lines of code to just create an object. This can of course be shortened. We could leave out `filename` and put the `str_create` directly into `ent_create`, but that also doesn´t look very nice and as Lite-C handles hardcoded char arrays the same way as a `STRING*`, if a `STRING*` is needed, we can just write `ent_create("Ende.mdl", &pos, aRotate);`. Now the big "problem" still is the position. Happily there is a function which takes thre parameters and returns a pointer to a vector using this values as x, y and z. The function is called `vector`:

```
ent_create("Ente.mdl", vector(300, 0, 0), aRotate);.
```

The file Main.c should now look like this:

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

```
#include "Ents.c"

void main()
{
	level_load(NULL);
	ent_create("Ente.mdl", vector(300, 0, 0), aRotate);
}
```

Ents.c should look like this:

```
void aRotate()
{
	while(1)
	{
		my->pan += 16*time_step;
		wait(1);
	}
}
```

Compile and run it again and you will see that it works the same as before.

Some more homework (assuming that you are anyways doing this stuff at home) :)

As a first learning task, try to rotate the model around all its angles while moving it along the x-axis, away from the camera, using `my` plus `pan`, `tilt`, `roll` and `x`. If you are able to do this, you should be ready for my next tutorial, which explains how to create a door which opens if the camera comes close.

I hope that this was understandable, helpfull and at least a little bit fun to read through,
Nils Daumann.