asrealize-2 Documentation

By anotherSimpleCoder

LICENSE

What is asrealize-2?

asrealize-2 is a compiler for the asrScript language, which is being used to describe/render 2D graphics. It's written in 2 programming languages: Rust and C++ and is based on the 2D rendering library asr.

How does asrealize-2 work?

On the surface it works the following way: You have an .asr file contains asrScript code. That file is putted into asrealize2 and it either shows you the render on a windows or compiles a PNG image. However (since that would be too short for a documentation and I wanna look like some cool dude lol) I am going to explain it here a bit more in detail.
asrealize can be splitted into two components: the compiler and the executer.
The compiler takes the human readable asrScript code and turns it into binary/machine code, which is understandable by the executer. So let's first of all start with the….

Chapter 1: Compiler

The compiler again can be splitted into the following modules:

- the file-reader
- the lexer
- the parser
- the binary-filer

File reader

The file reader as the name says is responsible for reading files. In this context it's getting the path to an asr file. Then it opens that file and copies the entire file content onto the memory. Let's use an example to explain the entire process:

```
#meta
-height:300
-width:300
-title:"foo"

#body
x : 13
(3,3) : (1, 2, x, 1)
```

Then that data is being handed over to the……

Lexer

The lexer receives the file content of the file reader ,scans through it and stores so called „tokens". A token is being used to indicate a specific type of symbol in text (for example a „:" is being stored as a token called COLON). This helps the computer to understand what exactly is being written on the file since the asr file is just readable for humans. While this process is going on, the lexer already checks for symbols, which aren't part of the asrScript syntax. If it detects an unknown symbol it returns an error message and lets the user know that an invalid symbol is being used in the code. While lexing those tokens are being stored in a LinkedList. In our example it would look like this:

```
[HASH; LT(m); LT(e); LT(t); LT(a)]
[DASH; LT(h); LT(e); LT(i); LT(g); LT(h) ;LT(t); COL ;DIG(3); DIG(0); DIG(0)]
[DASH; LT(w);LT(i) ;LT(d); LT(T); LT(h); COL; DIG(3); DIG(0); DIG(0)]
[DASH; LT(t); LT(i); LT(t); LT(l); LT(e); COL; QT; LT(f); LT(o) ;LT(o); QT]
```

```
[HASH; LT(b); LT(o); LT(d); LT(y)]
[LT(x); COL; DIG(1); DIG(3)]
[OB; DIG(3); COM; DIG(3); CB; COL; OB; DIG(1); COM; DIG(2); LT(x); COM; DIG(1); CB]
```

The tokens are then being processed by the…..

Parser

The task of the parser is to detect structures in code with the tokens and create so called „entries", which are being used to represent a line of code and also tells the executer later on if the entry is a section, attribute, assignment or a command. Also it's representing the structure of a command and the entire code. In our example it would like this:

```
Entry(SECTION, „meta");
Entry(ATTRIBUTE, „height", „300");
Entry(ATTRIBUTE, „width", „300");
Entry(ATTRIBUTE, „title", „foo");
Entry(SECTION, „body");
Entry(ASSIGNMENT, „x", „3");
Entry(COMMAND, „3", „3", „1", „2", „x", „1");
```

These entries will be handed over to the…

Binfiler

The binfiler is responsible for converting the received entries into binary code. It's being done the following way:

There are binary codes for asrScript instructions. Here is an overview:

Sections	
SEC_META	#0xD001
SEC_BODY	#0xD002
Attributes	
ATTR_HEIGHT	#0xAB01
ATTR_WIDTH	#0xAB02

ATTR_TITLE	#0xAB03
Indicators	
ASSIGN	#0xA0FF
COMM	#0xC0FF
ALPHABETIC	#0xABCD
NUL	#0xFFFF

The binfiler goes through the entry-register containing all the entries created by the parser and converts it to binary code. In our example the binary code would look like this (this is just representation and the actual file doesn't look like that):

```
0xD001         SEC_META            //meta section
0xAB01         ATTR_HEIGHT         //height attribute
0x012C         300
0xAB02         ATTR_WIDTH          //width attribute
0x012C         300
0xAB03         ATTR_TITLE          //title attribute
0x0066         102                 //ascii „f‟
0x006F         111                 //ascii „o‟
0x006F         111                 //ascii „o‟

0xD002         SEC_BODY            //body section

0xA0FF         ASSIGN              //assign indicator
0xABCD         ALPHABETIC          //alphabetic indicator
0x0058         88                  //ascii „x‟
0XABCD         ALPHABETIC          //alphabetic indicator
0x0003         3
0xA0FF         ASSIGN              //assign indicator

0xC0FF         COMMAND             //command indicator
0x0003         3
0x0003         3
0x0001         1
0x0002         2
0XABCD         ALPHABETIC          //alphabetic indicator
0x0058         88
0xABCD         ALPHABETIC          //alphabetic indicator
0x0001         1
0xC0FF         COMMAND             //command indicator
```

Then the binary code is being loaded into a buffer and gets stored into the binary file. However as you can see the binary instructions are 16-bit numbers. However a typical binary file is just able to write 8bits. So every 16 bit instruction is being split into 2 8-bit instructions, which are later on being merged back in the executer (which will be the next chapter). So at the end the binfiler will then compile and output an asb (another Simple Binary) file and that’s how the compiler module works.

Chapter 2: Executer

The executer can be split into the following modules:

- the binary file reader
- the execution module

Binary file reader

The binary file reader takes the path to a .asb file and processes the values to be later on stored to a component called the „binary register“ (which is just an array for unsigned 16-bit integers). First the file reader takes all the binary content and loads it onto the memory of the computer. However as you know from the last chapter the values on the files are 8-bit values now, not 16. So now we have to merge the values in the file together to get back 16- bit values. So in order to do that we have to understand on how the 16-bit values are being split. It's easier to show this by example. Let's take the following value:

```
0xD001          SEC_META             //meta section
```

This value indicates to the executer that there is a meta section in the file. Let's convert the hex value to binary:

```
11010000 00000001
```

The binfiler from the compiler split the 16-bit instruction to two separate 8-bit instructions. That makes it look like this:

```
00000001
11010000
```

Since 8-bits are 1 byte I'm gonna call it like that now. So as you can see the 16-bit instruction is split in 2 separate 1 byte instructions. However the 2nd byte instruction comes first and the first byte instruction comes later. Now in order to merge these two 8-bit values back to one 16-bit value we have to go through following calculation algorithm (shown in pseudo code):

```
function byteFusion(u_int8[] byteInstructions) -> u_int16{
    u_int16 byteBuffer
    byteBuffer = byteBuffer + byteInstructions[0];
    byteBuffer = byteBuffer + (byteInstructions[1] * power(2,8));

    return byteBuffer;
}
```

So what is being done here is the following:

The 2nd byte is being multiplied by `power(2,8)`, which will change the 2nd byte to the following binary value:

```
11010000 00000000
```

So we have now 8 more bits again to store the 2nd byte instruction. This is being done by adding the 2nd byte to the 1st byte, which then brings the split value to:

```
11010000 00000001
```

After this calculation is being done the resulting 16-bit value is then being stored onto the „binary register“ (which is a vector for 16-bit unsigned integers). After that the register is being handed over to the….

Execution module

What the execution module now does is that it iterates through the vector until the end and executes each instruction. Now let’s see how different instructions are being executed through our example from the compiler. This is now all the values in the binary register (I have used hex values so it’s easier to comprehend and read):

```
0xD001
0xAB01
0x012C
0xAB02
0x012C
0xAB03
0x0066
0x006F
0x006F
0xD002
0xA0FF
0xABCD
0x0058
0XABCD
0x0003
0xA0FF
0xC0FF
0x0003
0x0003
0x0001
0x0002
0XABCD
0x0058
0xABCD
0x0001
0xC0FF
```