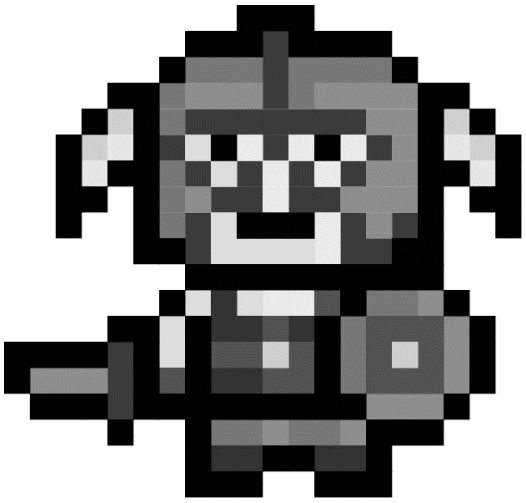
80’s Skyrim Report

Corneel Soete Aaron Munsters



2018-2019

1. Contents:

2. Introduction

Welcome to the paper on the assembly version of Skyrim made by Aäron Munsters and Corneel Soete. For the assignment we had the task to make a small program with graphical input/output, so that is exactly what we did. We started by thinking of what games we liked and which one could use a remake. After some brainstorming we came to the conclusion that both our favorite video game is one and the same *and* that it hasn't been remade for at least half a year. So naturally we took it to ourselves to do Bethesda a favor and make the next game in the series of Skyrim remakes.

So with the premise of making Skyrim we had to think very deep to what we could actually keep from the original game, of course we couldn't keep the 3d graphic because this would be way too complex to achieve within the short timespan we had at hand. So instead we took graphical inspiration from the graphics of the original Legend of Zelda. In the same fashion we want to be able to look from top perspective. In the Legend of Zelda the screen also doesn't move with the player, but the player moves in a part of the world, and when he crosses the edge of a field, he goes to the next field. This is also a mechanic which will be used in our game. The more we actually think about it, it is in fact the legend of Zelda with a Skyrim reskin we are making.

Skyrim and Zelda both have in common that they are RPG's (role-play games), so we clearly wanted to do the same. But to make a small RPG-game we had to think about some of the game-mechanics. For instance, how will we implement this field behavior? Will there be an inventory, if so, how would we manage its contents and how would it affect the game? We had a difficult task at hand, but we nevertheless succeeded in achieving what we wanted. In the following paper our thought process will be explained. We will also discuss some of the specifications in how to use our code and some of the obstacles we (tried) had to overcome while implementing the game. This paper is be divided in two major sections being the game logic and the graphics. We chose to split into these sections mainly because this is the way we spread our workload.

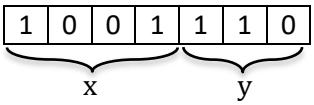
3. Game logic

There are two main elements which determine how the core of the game functions: the data structures and the program flow. The data structures are continuously read and treated by the program, but the order in which and reasoning for why this happens will be decided by the program flow.

3.1. Data structures

3.1.1. Getters and setters

The game logic operates on a lot of data, because we write on such a low program level, we will exploit the benefits of programming with bits. Since the game is based on running around in a static field, we chose to work in a field with a height of 8 and a width of 16, because both of these are powers of two. This allows us to store a position by only using 7 bits. The next figure illustrates this:



This position can read as an x-y-coordinate $(9, 6)$. Further exploiting this will result in the following data structures for entities:

I	I	M	T	T	W	W	S	A	H	L	L	D	D	X	X	X	X	Y	Y	Y	A

$I(ntelligence), M(ode\ of\ AI), T(ype\ of\ enemy), W(eapon), S(hield), A(rmour), L(ife), D(irection), X(-position), Y(-position), A(ctive)$

To read and manipulate bits we mainly make use of two procedures, getDataVal and giveDataNewVal. An example of how to call these functions would be:

call getDataVal, entity_data, shift_4, bits_4

Which would shift the data 4 to the left and return the 4 rightmost bytes in the register *eax*, which would be the x-value of the entity. Thus getDataVal needs to know how many bits (counting from the least significant bit) it should shift and how many bits it should take. The procedure giveDataNewVal on the other hand would require the same information together with a new value, thus for example to store a new y-value the caller should write:

call giveDataNewVal, entity_data, new_data, shift_1, bits_3

and this will return the updated entity-value in the register *eax*.

Note:

Due to extensive usage of getDataVal and giveDataNewVal, a lot of macros are defined to discard the hassle of remembering the size and location of bits in the data. This makes writing and reading the code easier.

3.1.2. Collections of data in memory

Another data structure will be the memory in which we save these entities. Take for example a field full of enemies, during the game we would like to update these enemies (imagine every entity moving forward in its desired direction). One way we could keep all these enemies would be to store them in a couple of global variables, but this is not flexible. Thus we chose to reserve some memory (the amount of entities times the size needed to store a single entity) and store every entity in this "array". If we would want to update the enemies and draw them this would require us to apply a procedure to every entity. Inspired by the course "Structure of computer programs 1" we wrote two procedures, `mapMem` and `forEachMem`, which take a pointer to an array, the size of the array and a procedure and applies the procedure to every piece of data inside this array.

Note:
Because we use these four data-procedures very often, we mainly make use of double words which makes data control a lot easier. The downside however is that a lot of bits are wasted in memory and move-operations are more costly.

3.1.3. Some data conventions

In the game there are only four directions, thus this can be stored in two bits. By choosing the following directions:

Up = 0 0 Right = 01 Down = 10 Left = 11

We can easily turn around an entity by adding 2 to its direction, the procedure `giveDataNewVal` will only keep the two rightmost bits.

Another choice was to store a cell as four item-bits and four floor-bits, if an item is on the floor it would be stored in the four most significant bits and the floor-type will be stored in the four least significant bits. A deliberate choice was to ensure the least significant bit for cells would determine whether the cell is walkable or not, all this together leaves the game with 15 possible items and 16 possible floor-types half of which walkable.

3.2. Program flow

While we've gotten comfortable with how our data will be handled, we are still unaware of why and when it will be handled. When should and enemy move? Why should it move? How will the player be affected by the environment? These questions will be answered by how the game logic operates. One could say this is the core of the gameplay. As with most games, the core is a loop which continuously runs. The following comes directly from the code:

```
PROC programLoop

	@@programLoop:
	cmp [ProgramLoop], IN_GAME
	jne @@inMenu

	;@@inGame
	call checkKeyPresses		;-> process user key-press
	call updateField		;-> update environment
	call checkPlayerStatus		;-> check if player is still alive
	;; graphically
	call awaitVBIStart	;-> waits for VBI
	call drawEverything	;-> refreshes
	;; repeat
	jmp @@programLoop

	@@inMenu:
	call checkMenuKeyPresses	;-> process user key-press
	;; graphically
	call awaitVBIStart	;-> waits for VBI
	call drawEverythingMenu		;-> refreshes
	;; repeat
	jmp @@programLoop

	ret
ENDP programLoop
```

As is clear by reading the code, there are two main loops, the game loop and the menu loop. As most are clear about what they do by their name (and their comments), only unique procedures will be further explained.

3.2.1. Abstraction

A lot of functions in the first part of skyrim.asm are some definitions like walkable? and alive? , these make the code a lot more manageable. Some of these functions are only called once, when this is the case it will be because future game-updates would certainly require these functions. Another benefit with these procedures is that the code becomes a lot more readable.

3.2.2. Two-dimensional fields

There are three core two-dimensional arrays in the program; the main ones are:

- All cells in a field
- All fields in a world
- All items in an inventory

Because of these we work a lot with x- and y-dimensions and their boundaries. To avoid writing the same code for the same concept we opt to implement a procedure that will be used individually by these fields, being getRelPosition which returns a relative position to the given coordinate.

3.2.3. Moving entities

To give the player the impression he or she is wandering in a vivid world, we have to ensure there are other entities. Not only should the player move, but his allies and enemies as well. We created a general procedure moveEntity which takes an entity and returns the entity moved in its direction. This procedure also takes an argument (a Boolean) which tells the procedure if the field-focus is linked to this entity. If this is the case and the entity crosses an edge of the field, the world will shift along to the new field and the player will be placed on the opposite side of the field, and so creating the scroll-effect.

Next to ensuring only the player can move around in the world, this allows for simple animations that move another character through the world (for example the main antagonist who runs over to the mountains).

3.2.4. The combat system

To turn a program into a game requires the program to give the user a target to achieve. To turn a game into a challenging game is to ensure the target can only be achieved by overcoming obstacles. A challenge we deemed necessary is providing a combat system, which the player should get familiar with to survive in the game. The main idea is the following: If a certain other entity is hostile and notices the player, it will try to get rid of the player by damaging it. The way other entities behave will be discussed in section 3.2.7, we will now focus on how damage will be dealt.

If an entity wishes to damage its damage will depend on which type of weapon it is carrying. Take for example that a skeleton tries to attack our player with its bare hands. The player is carrying a sword, a shield, a chest plate and have 2 lives left. If the skeleton hits the player, the shield will be broken. If it hits the player again the chest plate will break. If the skeleton attacks us once more we will start losing lives. This idea is clearly worked out by the procedure doDamage which returns the damaged entity provided by the caller. How equipment will be removed from the player's inventory will be discussed in section 3.2.6.

3.2.5. The right mobs in the right field

As the player wanders through the world, it is necessary to keep the right mobs in the right fields. When updating mobs the pointer to the array of mobs is passed to mapMem which is called with the procedure updateEntity.
To keep the right pointer to the mobs, we update this pointer every time the field is shifted. This is what the procedure moveCurrFieldMobs takes care of.

3.2.6. The inventory

When the player hovers over an item it will automatically be picked up. The item is passed to the procedure insertInInventory which takes an item and puts the item in the first free inventory spot.
When damage is dealt to the main character and he is carrying an item which was defending him, due to the damage this item will break and be removed from the inventory. To make this possible, we make use of the procedure removeItem which takes an item and removes it from the inventory (only once!).

3.2.7. How enemies think

The artificial intelligence of the mobs (the way they move and act) can be described by the following pseudo-code:

```
... game update code ...
forAll(mobs, updateMob);
... game update code ...

proc updateMob(currMob){
        aiMode = getAiMode(currMob);
        switch(aiMode){
                case aggressive :
                        return actAggressive(currMob);
                case nonAggressive :
                        return actNonAggressive(currMob);
        }
}
```

Where actAgressive and actNonAggressive are two procedures which loop between four states. For aggressive behavior these four are a cycle of approaching the enemy, attacking the enemy, backing off and waiting. For nonaggressive behavior these four are looking around, waiting to look around, walking and waiting to walk around.

In case a mob sees the player, it becomes aggressive and speeds up the rate at which mobs are being updated, this gives the player the impression that one mob alerts all mobs about danger. In case the player leaves a field this rate will be set to the normal speed.

3.2.8. The inventory

When the player is in the menu, there are a few options to be selected. Start and stop navigate back to the game or out of the program. When navigating to the left, the player can scroll through the items and press a or b whenever desired to. In case the item is some form gear, it will be assigned to the right equipment-slot. In other cases it will be assigned to either the a- or b-slot and thus the character will be holding it in his hand. All of this is handled by the procedure clickInventory.

4. Graphics

Before describing everything that has to do with the graphics we first have to define which aspects of the code belong to the graphics department. Every bit of code that has to do with displaying content on screen has been encapsulated by the term graphics. In practice this means the complete Graphics.ASM file, the sprites.ASM file and a few procedures in the skyrim.ASM file. The next sections will explain the features of the graphics, the procedures that make this work and lastly the difficulties we had to overcome during the build process. We continue by clarifying the structure of the files and why some procedures are written in the Skyrim file and some in the graphics file.

4.1. File structure

All graphic related code can be found in three files (with exception to the macros):

- ❏ graphics.ASM
- ❏ skryim.ASM
- ❏ logicf.ASM

The file graphics.ASM contains drawing procedures that are generic and can be used to make other games as well. In this file there is no game logic, so this code is independent of the game structure. These are all procedures that make it possible for the programmer of the game logic not to be bothered on how the drawing works but only has to call a function “display_…” to display something on the screen. The sprites.ASM file is a file filled with arrays of hexadecimal color values. These arrays store all the sprites that are being used. How these sprites work will be explained in section 4.3.1. Last but not least are the graphics procedures in the skyrim.ASM file. These make use of the procedures in graphics.ASM but first determine with the game representation what values should be passed on to the graphical procedures. For example the procedure drawDovah will calculate where Dovah has to be drawn, in which direction Dovah is headed, which items Dovah is carrying and will finally call display_dovah. These procedures should and have no clue about how the actual drawing of the sprites works.

4.2. Features

This section will explain which procedures the Game logic programmer is able to use from the graphics.ASM file. We can get a clear overview on all these procedures in the graphics.inc file and don't need a unique explanation. There are a lot of procedures that expect a bit string of four bits to determine which item or terrain has to be displayed. These are macros that correspond directly to the place in the array of sprites so this should be a convention between the logic programmer and the graphics programmer. For example: the macro of a wall (a type of terrain) is 0000b and in the array of terrains we see that wall is the first sprite in this array. Thus it is very important that both departments respect this convention. All other procedures are pretty self-explanatory and extra information can be found in the file graphics.inc where all the global procedures are listed with a short explanation on what they do and what arguments they expect.

4.3. Special procedures

Some procedures require a detailed explanation. We start with three small procedures which abstract the window-size coordinates. These procedures turn the abstract coordinates into coordinates used by the screen. In the code below only the procedure for calculating field coordinates is shown, the same applies for calculating inventory coordinates and the minimap coordinates.

```
PROC real_coord       ;procedure puts the real x-value in eax
        ARG @@abs_x : dword
        USES edx
        mov  edx, CELLSIZE    ;width/height of a cell
        mov  eax, [@@abs_x]
        mul  edx

        ret
ENDP real_coord
```

4.3.1. The draw procedure

This procedure gets its own section because it is the core to make almost all other procedures work. As shown in the code below the procedure draw takes a lot of arguments, this is because it is made to be as flexible as possible to draw all kinds of sprites in all kinds of places. Its gets an x- and y-value where the sprite has to be drawn, these are exact window-coordinates and not our abstract ones so they will have to be converted with the procedure discussed in section 4.3. The next argument is an index to where the sprite might me be stored in the list (which is given as next argument). And lastly we have to give a 1 or a 0 to say if the sprite has to be drawn in the field or not (when it has to be drawn in the field it should be centered in a cell and not displayed in a corner).

```
PROC real_coord
       ARG @@x0 : word, @@y0:word, @@idx:dword, @@list:dword, @@infield?:dword
```

The other part of the code states how to find the starting point in the video memory address where the sprite has to be drawn and where the sprite is stored (find the sprite in the list making use of the index) all of this is rather self-explanatory in the code so there are no further details required. A more important aspect is how the sprites are being drawn and how they are actually stored in memory.

As told before, the sprites are stored in a different file called sprites.ASM. Here all sprites are stored in arrays with hexadecimal values. In the code below there is an example on how this might look like. The value 28H is highlighted so it becomes clear how different values determine the sprite. Each value corresponds to a color value in the color palette. The first 2 values of the array correspond to the width and the height of the sprite and are needed to find the correct sprite with the index with the following formula: $index * width * height =$ offset in array.

```
UITEMS  DB 8, 8

;heart alive sprite
DB 0FFH,  28H,  28H, 0FFH, 0FFH,  28H,  28H, 0FFH
DB  28H,  28H,  28H,  28H,  28H,  28H,  28H,  28H
DB  28H,  28H,  0FH,  28H,  28H,  28H,  28H,  28H
DB  28H,  28H,  28H,  28H,  28H,  28H,  28H,  28H
DB 0FFH,  28H,  28H,  28H,  28H,  28H,  28H, 0FFH
DB 0FFH,  28H,  28H,  28H,  28H,  28H,  28H, 0FFH
DB 0FFH, 0FFH,  28H,  28H,  28H,  28H, 0FFH, 0FFH
DB 0FFH, 0FFH, 0FFH,  28H,  28H, 0FFH, 0FFH, 0FFH
```

Concerning the procedure draw and how to actually draw these sprites. In the code below loop goes through the sprite to draw it. Before this loop is started there is already some data in the registers: *esi*

```
;--- loop through the sprite to draw it ---
@@outerloop:
        @@innerloop:
                mov al, [esi + ebx]
                cmp al, 0FFH
                je @@nodraw
                mov [edi + ecx], al
                @@nodraw:
                inc cl
                inc ebx
                cmp cl, [esi]
                jne @@innerloop

        add edi, SCRWIDTH
        xor ecx, ecx
        inc dl
        cmp dl, [esi + 1]
        jne @@outerloop
```

holds the array in which the sprite is stored, *ebx* holds the offset to the start of the sprite in the array, *edi* holds the memory address and the other registers are all set to 0. This is all in all a loop through the sprite and writing the value to the memory address, something to clear out is that when the hexadecimal value is 0FFH, we skip this pixel and don't draw anything. This makes it possible to draw transparent sprites. This value should draw black but since there are other similar black values we chose this one.

4.3.2. The display_map procedure

This procedure also requires some additional explanation. The procedure only takes 2 arguments, a pointer to an array which holds the complete world but only the terrain value of each cell. So beforehand we have to apply a map-procedure on the real world array to get the new minimap array. The second argument is an integer which tells us what field we are currently in, this will draw a white rectangle around the field in the minimap where the player is currently located.

```
PROC display_map
       ARG @@map:dword, @@selectedField:dword
```

This procedure takes the value of the terrain in the minimap array and looks in another array called TERRAIN_COLORS what color corresponds with this value e.g.: we take the upper left most terrain in the first field, which is a rock. The terrain value of a rock is 0010b (2 in decimal) so we will look in the TERRAIN_COLORS array and find the value 0F7H (this corresponds to dark grey). This will eventually result in a minimap where the top left corner has a dark grey color pixel.

```
TERRAIN_COLORS DB 73H, 2EH, 0F7H, 0FH, 06H, 1BH, 31H, 2BH, 02H, 2CH, 37H, 37H
```

We chose to dynamically draw the minimap, this has the advantage for us to redesign the world and the minimap would show the new world. It allows us to e.g. explode a building during gameplay and the minimap will show this.

5. Difficulties

A major difficulty we came across and couldn't get rid of was that the screen would flicker during screen refreshes. After we get some help from the assistant we were able to overhaul the flickering by instead of writing to the video memory directly, first writing to a framebuffer. After this write, copying from the framebuffer to the video memory. Another solution could've been to only update visually the cell which where updated in game. This seemed like a challenge too great to accomplish in the short timespan we had left.

Another difficulty was efficiently generating sprites. The sprites file consists out of approximately 160.000 characters, writing these all by hand (copying them from the color-palette) would've been a time of loss. To reduce this time loss we developed a small paint-like program in Scheme where you can draw a sprite and export the result in hexadecimal values.

6. References

Cornelis J. G., & Blinder D. (2018). *Assembly Programming Compendium*, Brussels.

Hyde, R. (2010). *The Art of Assembly Language ; 2nd Edition*. No Starch Press.

For the sprites:
Main character: www.flickr.com/photos/chaosrules01/8343033400
Other sprites we made ourselves.