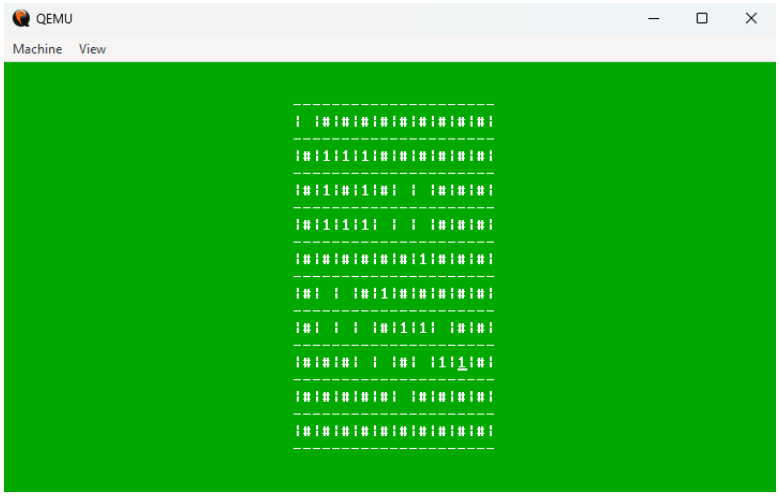
Saper mieszczący się na sektorze rozruchowym - Sprawozdanie

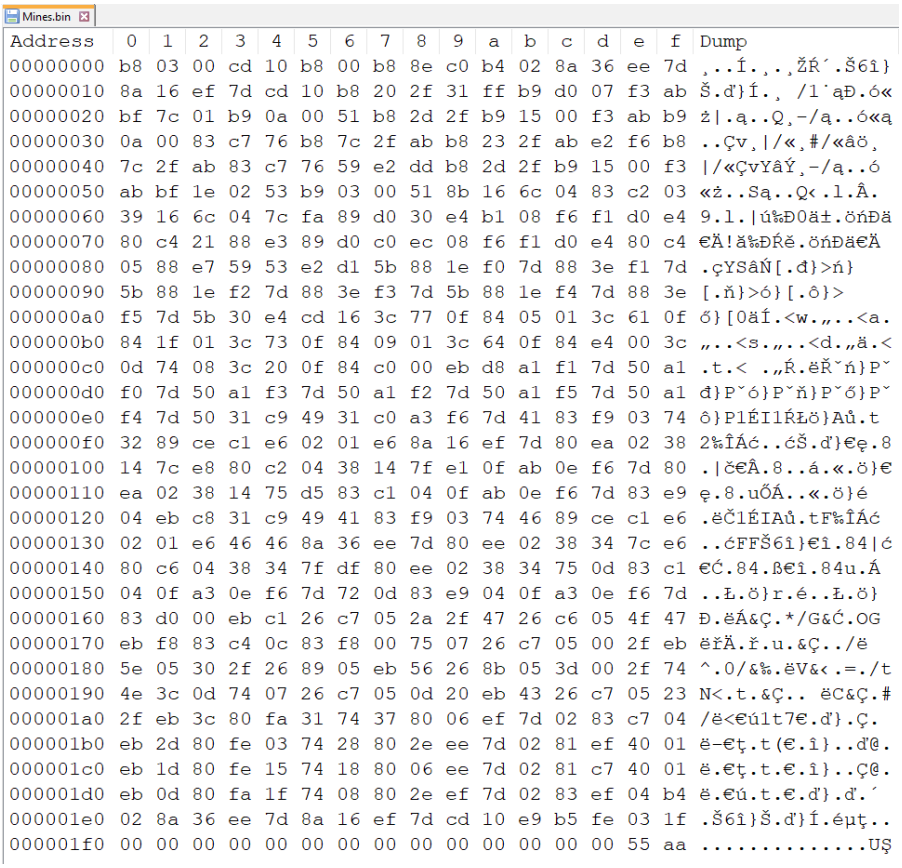
Autor: Piotr Noga



Ilustracja 1: Zrzut ekranu z rozgrywki z ostatecznej wersji gry

Wstęp

W niniejszym dokumencie przedstawiony zostanie proces tworzenia uproszczonej wersji gry komputerowej “Saper”, która w całości mieści się na sektorze rozruchowym. Fakt ten, iż program może być umieszczony na takim sektorze, sprawia, że program może być wczytany do pamięci urządzenia niemal od razu po jego uruchomieniu. By móc tego dokonać, należy przede wszystkim rygorystycznie przestrzegać odpowiedniego rozmiaru całego kodu, odpowiedzialnego za działanie gry. W jednym z rozdziałów zostaną udokumentowane ustępstwa, jakie zostały podjęte w trakcie pisania kodu programu, aby on mógł być możliwie jak najmniejszy. Na początku każdego rozdziału, w którym prezentowane będą zmiany w kodzie, przedstawiony będzie jego fragment wraz z oryginalnymi komentarzami, którego się tyczy rozdział. Gdy dany fragment kodu zostanie dodany do istniejącej już etykiety, bądź będzie on zmieniony, zostanie to odpowiednio odnotowane za pomocą nawiasów ostrokątnych. Sprawozdanie to jest pisane w oparciu o napisany już kod, który został też odpowiednio zoptymalizowany pod kątem jego rozmiaru w skompilowanej formie. Prezentowane jego fragmenty zatem mogą początkowo powodować kłopoty w kompleksowym zrozumieniu kodu. Jak wspomniano wcześniej, zastosowane optymalizacje zostaną odpowiednio wytłumaczone w jednym z rozdziałów. Mimo faktu, że dokument ten jest pisany w odniesieniu do gotowego kodu, zachowane zostaną kroki, w których powstawał cały program. Przedstawiony zostanie przebieg wytwarzania kodu, poparty zrzutami ekranu, które pokazują efekty pracy w danym momencie.


Mines.bin

Address	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	Dump
00000000	b8	03	00	cd	10	b8	00	b8	8e	c0	b4	02	8a	36	ee	7d	¸..Í.¸.¸ŽŔ´.Š6î}
00000010	8a	16	ef	7d	cd	10	b8	20	2f	31	ff	b9	d0	07	f3	ab	Š.ď}Í.¸ /1˙ąĐ.ó«
00000020	bf	7c	01	b9	0a	00	51	b8	2d	2f	b9	15	00	f3	ab	b9	ż|.ą..Q¸-/ą..ó«ą
00000030	0a	00	83	c7	76	b8	7c	2f	ab	b8	23	2f	ab	e2	f6	b8	..Çv¸|/«¸#/«âö¸
00000040	7c	2f	ab	83	c7	76	59	e2	dd	b8	2d	2f	b9	15	00	f3	|/«ÇvYâÝ¸-/ą..ó
00000050	ab	bf	1e	02	53	b9	03	00	51	8b	16	6c	04	83	c2	03	«ż..Są..Q‹.l.Â.
00000060	39	16	6c	04	7c	fa	89	d0	30	e4	b1	08	f6	f1	d0	e4	9.l.|ú‰Đ0ä±.öńĐä
00000070	80	c4	21	88	e3	89	d0	c0	ec	08	f6	f1	d0	e4	80	c4	€Ä!ă‰ĐŔě.öńĐä€Ä
00000080	05	88	e7	59	53	e2	d1	5b	88	1e	f0	7d	88	3e	f1	7d	.çYSâŃ[.đ}>ń}
00000090	5b	88	1e	f2	7d	88	3e	f3	7d	5b	88	1e	f4	7d	88	3e	[.ň}>ó}[.ô}>
000000a0	f5	7d	5b	30	e4	cd	16	3c	77	0f	84	05	01	3c	61	0f	ő}[0äÍ.<w.„..<a.
000000b0	84	1f	01	3c	73	0f	84	09	01	3c	64	0f	84	e4	00	3c	„..<s.„..<d.„ä.<
000000c0	0d	74	08	3c	20	0f	84	c0	00	eb	d8	a1	f1	7d	50	a1	.t.< .„Ŕ.ëŘ˝ń}P˝
000000d0	f0	7d	50	a1	f3	7d	50	a1	f2	7d	50	a1	f5	7d	50	a1	đ}P˝ó}P˝ň}P˝ő}P˝
000000e0	f4	7d	50	31	c9	49	31	c0	a3	f6	7d	41	83	f9	03	74	ô}P1ÉI1ŔŁö}Aů.t
000000f0	32	89	ce	c1	e6	02	01	e6	8a	16	ef	7d	80	ea	02	38	2‰ÎÁć..ćŠ.đ}€ę.8
00000100	14	7c	e8	80	c2	04	38	14	7f	e1	0f	ab	0e	f6	7d	80	.|č€Â.8..á.«.ö}€
00000110	ea	02	38	14	75	d5	83	c1	04	0f	ab	0e	f6	7d	83	e9	ę.8.uŐÁ..«.ö}é
00000120	04	eb	c8	31	c9	49	41	83	f9	03	74	46	89	ce	c1	e6	.ëČ1ÉIAů.tF‰ÎÁć
00000130	02	01	e6	46	46	8a	36	ee	7d	80	ee	02	38	34	7c	e6	..ćFFŠ6î}€î.84|ć
00000140	80	c6	04	38	34	7f	df	80	ee	02	38	34	75	0d	83	c1	€Ć.84.ß€î.84u.Á
00000150	04	0f	a3	0e	f6	7d	72	0d	83	e9	04	0f	a3	0e	f6	7d	..Ł.ö}r.é..Ł.ö}
00000160	83	d0	00	eb	c1	26	c7	05	2a	2f	47	26	c6	05	4f	47	Đ.ëÁ&Ç.*/G&Ć.OG
00000170	eb	f8	83	c4	0c	83	f8	00	75	07	26	c7	05	00	2f	eb	ëřÄ.ř.u.&Ç../ë
00000180	5e	05	30	2f	26	89	05	eb	56	26	8b	05	3d	00	2f	74	^.0/&‰.ëV&‹.=./t
00000190	4e	3c	0d	74	07	26	c7	05	0d	20	eb	43	26	c7	05	23	N<.t.&Ç.. ëC&Ç.#
000001a0	2f	eb	3c	80	fa	31	74	37	80	06	ef	7d	02	83	c7	04	/ë<€ú1t7€.đ}.Ç.
000001b0	eb	2d	80	fe	03	74	28	80	2e	ee	7d	02	81	ef	40	01	ë-€ţ.t(€.î}..đ@.
000001c0	eb	1d	80	fe	15	74	18	80	06	ee	7d	02	81	c7	40	01	ë.€ţ.t.€.î}..Ç@.
000001d0	eb	0d	80	fa	1f	74	08	80	2e	ef	7d	02	83	ef	04	b4	ë.€ú.t.€.đ}.đ.´
000001e0	02	8a	36	ee	7d	8a	16	ef	7d	cd	10	e9	b5	fe	03	1f	.Š6î}Š.đ}Í.éµţ..
000001f0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	55	aa	..............UŞ

Ilustracja 2: Kod programu w formie binarnej

Konfiguracja środowiska

Zanim będzie można przystąpić do tworzenia kodu programu, należy najpierw skonfigurować środowisko, w którym będzie można sprawnie doświadczyć efekty pracy. W tym celu zainstalowane zostały emulator QEMU oraz kompilator NASM. QEMU w prosty sposób umożliwia uruchamianie przygotowanego kodu w taki sposób, jakby on faktycznie znajdował się w sektorze rozruchowym na nośniku danych. Kompilator NASM natomiast za pomocą jednego polecenia kompiluje kod asemblera do formy binarnej, która jest obsługiwana przez QEMU.

Pierwsze kroki

W oparciu o zebrane notatki a w szczególności o wideoporadniki dotyczące tworzenia innych gier na podobnej zasadzie, podjęty został plan działania przy tworzeniu kodu. Kolejne rozdziały tego dokumentu stanowią rozwinięcie ogólnikowych działań, które były zaplanowane w ów planie. Wyglądał on następująco:

1. Zapewnienie graficznej prezentacji gry
2. Wdrożenie sterowania
3. Napisanie logiki gry.

Najbardziej widocznym efektem pracy jest wizualne przedstawienie gry, zatem od tego został rozpoczęty proces twórczy.

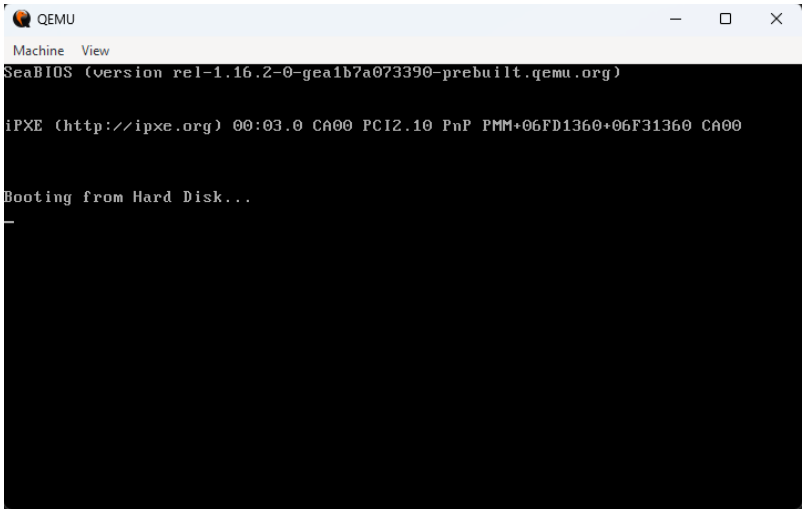
Ustawienia kodu

Fragment omawianego kodu:

use16 ; 16-bitowy kod
org 7C00H ; ustawienie bootsectora w pamieci na pozycji 7C00H

times 510-($-$$) db 0 ; zerowanie niewykorzystanego miejsca
dw 0AA55H ; zakonczenie pliku sygnatura

Jeszcze przed rozpoczęciem pisania faktycznego kodu, należy pokrótce zawrzeć w kodzie kilka instrukcji. Na samym początku programu jest zawarta dyrektywa *use16*, która informuje kompilator NASM o tym, że kod asemblerowy będzie działał w trybie 16-bitowym. *org 7C00H* umożliwia uruchomienie programu w taki sposób, jakby znalazł się on na sektorze rozruchowym. Przedostatnia linia kodu jest odpowiedzialna za wypełnienie niewykorzystanego miejsca bajtami zerowymi, natomiast *dw 0AA55H* sprawia, że na końcu kodu w formie binarnej będzie zapisana sygnatura 55 AA. Odwrotna kolejność w kodzie programu wynika z zapisu *little endian*.



Ilustracja 3: Emulator QEMU pomyślnie uruchomił pusty program

Rysowanie po ekranie - zmiana tła

Fragment omawianego kodu:

setup:

mov ax, 0003H ; ustawienie trybu graficznego. AH = 00 i AL = 03 to 16-kolorowy tryb VGA, gdzie mozna umiescic 80x25 znakow

int 10h ; wywolanie przerwania

mov ax, 0B800H ; nie mozna bezposrednio przypisac stalej liczbowej do es, wiec trzeba najpierw do dostepnego rejestru

mov es, ax ; ES:DI = B800:0000, gdzie DI bedzie wyzerowany w cyklu

```
; glowna petla rysujaca tablice
screen:
        mov ax, 2F20h ; AH = 20 - zielone tlo i czarny tekst AL = 0
        xor di, di ; zerowanie di
        mov cx, 80*25 ; wpisanie do rejestru licznika lacznej liczby znakow ile mozna przypisac w wybranym trybie graficznym
        rep stosw
```

Pierwszym elementem programu nad jakim została podjęta faktyczna praca, była zmiana koloru tła. By to zrobić, najpierw trzeba ustalić, w jakim trybie graficznym w ogóle będzie prezentowana gra. Wybór padł na 16-kolorowy tryb tekstowy, w którym można zapisać 80 znaków w 25 poziomych liniach. Cztery pierwsze linie powyższego fragmentu kodu umożliwiają ustawienie wspomnianego trybu graficznego. Objaśnienia poszczególnych linijek kodu zawarte są w oryginalnych komentarzach zawartych w kodzie.

Ilustracja 4: Uruchomiony tekstowy tryb graficzny

Na powyższym zrzucie ekranu zauważyć można, że jest czarny ekran, a wspomniana została zmiana koloru tła. W tym momencie wykorzystana zostanie pozostała sekcja kodu z fragmentu kodu. Objaśnienia poszczególnych linii kodu również są zawarte w komentarzach, natomiast wyjaśniając podjęte działanie w skrócie:

- wybrany został do rejestru ax pusty znak, który nie jest widoczny
- wraz ze znakiem, ustawiony został kolor tła
- następnie został wyzerowany rejestr di, który umożliwi zmianę tła ekranu
- do rejestru cx została wpisana łączna liczba znaków, która może znaleźć się na ekranie
- na koniec *rep stosw* powoduje powtarzanie tyle razy „*mov [es:di], ax oraz inc di dwa razy*”, ile wynosi łączna liczba znaków (licznik w rejestrze cx)

Ilustracja 5: Widodczna zmiana koloru tła

Rysowanie po ekranie - wyświetlanie planszy

Fragment omawianego kodu:

```
table:
        mov di, 80*2*2 + 30*2 ; ustawienie pisania od drugiego wiersza z przesunieciem w prawo o 30 znakow. Trzeba mnozyc oba skladniki razy dwa, bo pozycja znaku z tego co wywnioskowalem, znajduje sie na parzystych numerach w pamieci


        mov cx, 10 ; ustawienie liczby powtorzen rysowania wierszy
        drawing:
                push cx ; wrzucenie na stos licznika wierszy
                ; rysowanie kreski oddzielajacej kolejny wiersz
                mov ax, 2F2DH ; bialy znak na zielonym tle (2F), znak '-' (2D)
                mov cx, 21 ; 21 znakow
                rep stosw ; zoptymalizowane uzycie instrukcji: mov [es:di], ax   oraz   inc di dwa razy


                ; rysowanie wiersza tablicy
                mov cx, 10
                add di, (80-21)*2 ; przejscie na poczatek kolejnej linii rysowanej tablicy ((szerokosc ekranu - 20 znakow) * 2)
                row:
                        mov ax, 2F7CH ; bialy znak na zielonym tle (2F), znak '|' (2D)
                        stosw
                        mov ax, 2F23H ; bialy znak na zielonym tle (2F), znak '#' (23)
                        stosw
                        loop row
                mov ax, 2F7CH ; bialy znak na zielonym tle (2F), znak '|' (2D)
                stosw
```

; przejscie do kolejnego wiersza

*add di, (80-21)*2 ; przejscie na poczatek kolejnej linii rysowanej tablicy ((szerokosc ekranu - 20 znakow - 1 znak, by wyrownac) * 2)*

pop cx ; pobranie ze stosu licznika wierszy

loop drawing

;rysowanie ostatniego wiersza

mov ax, 2F2DH ; bialy znak na zielonym tle (2F), znak '-' (2D)

mov cx, 21 ; 21 znakow

rep stosw

*mov di, 80*3*2 + 31*2 ; ustawienie di na poczatek tablicy, by zmienianie znakow w tablicy odbywalo sie w tym samym miejscu, w ktorym jest kursor*

Wymiary planszy

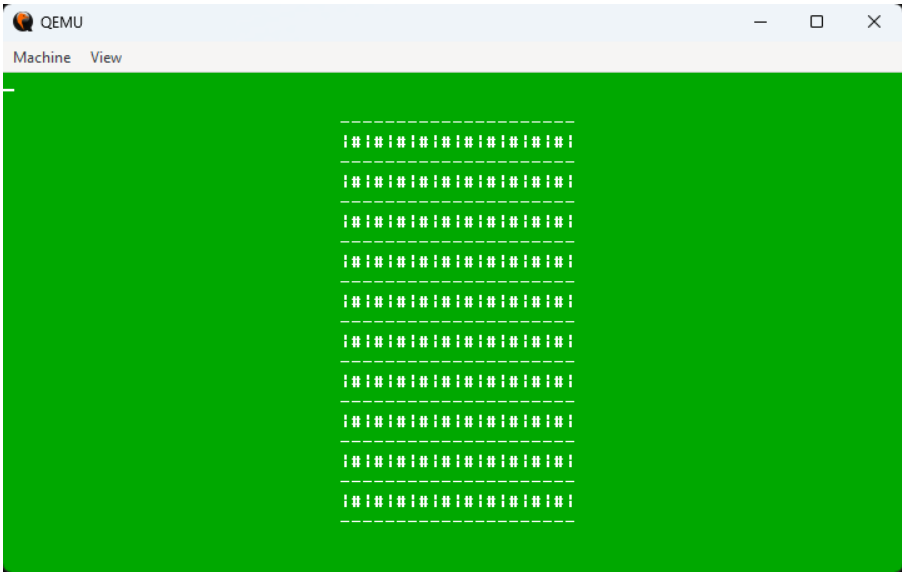
Gdy już kolor tła został zmieniony, można przystąpić do stworzenia planszy, na której będzie rozgrywać się gra. Plansza w “Saperze” domyślnie jest kwadratowa, tak więc taka też powinna być w zaimplementowanym kodzie. Żeby móc reprezentować pojedyncze pole planszy w trybie tekstowym, należy użyć trzy znaki w osi pionowej i trzy znaki w osi poziomej, przy dwóch polach potrzeba pięć znaków w poziomie i w pionie itd.. Łatwo można wywnioskować, że liczba potrzebnych znaków na reprezentowanie X pól w danej osi wynosi 2*X+1. Zważywszy na fakt, że w ustalonym wcześniej trybie tekstowym może być wyświetlanych maksymalnie 25 poziomych linii i 80 pionowych linii, X może wynieść maksymalnie 12. Stąd też została podjęta decyzja, że plansza w grze będzie o wymiarach 10x10, żeby móc jeszcze zachować odstępy od krawędzi ekranu, dla lepszej estetyki programu.

Implementacja planszy

Implementacja planszy w krokach wygląda następująco:

1. ustalenie początkowej pozycji, od której będzie rysowana plansza *mov di 80*2*2 + 30*2*
2. ustawienie licznika wierszy w rejestrze cx *mov cx, 10*
3. **rysowanie poziomego obramowania planszy:**
    - wrzucenie na stos powyższego licznika, gdyż później będzie nadpisany przez licznik kolumn *push cx*
    - wpisanie do rejestru ax białego znaku “-” na zielonym tle, który służy jako poziome obramowanie planszy *mov ax, 2F2DH*
    - wpisanie do licznika cx liczby kolumn (10 kolumn, czyli 2*10+1 znaków) *mov cx, 21*
    - wpisanie do pamięci ekranu zawartości rejestru ax oraz przesunięcie do pozycji kolejnego znaku na ekranie *rep stosw*

4. **rysowanie wiersza z polami:**
    - wpisanie do rejestru cx liczby kolumn *mov cx, 10*
    - przejście do kolejnego wiersza na ekranie. przesunięcie do nowego wiersza w tej samej kolumnie to przesunięcie o 80 znaków, natomiast trzeba cofnąć się o tyle znaków, żeby móc być w pierwszej kolumnie planszy, czyli o 21 znaków do tyłu. pozycja na ekranie jest zapisywana na parzystych bajtach, stąd też trzeba pomnożyć liczbę razy 2. *add di, (80-21)*2*
    - rysowanie na przemian na ekranie znaków “|” oraz “#” poprzez wpisywanie odpowiednik znaków i kolorów do rejestru ax i wykorzystaniu stosw, tak jak w poprzednim punkcie *mov ax, 2F7CH stosw mov ax, 2F23H stosw*
    - wywoływanie powyższego podpunktu w pętli *loop row*
    - wpisanie do pamięci ekranu ostatniego znaku “|” poza pętlą *mov ax, 2F7CH stosw*
5. powtarzanie kroków 3 i 4 poprzez przesunięcie pozycji rysowania na ekranie do początku kolejnego wiersza planszy i przywracania ze stosu licznika wierszy *add di, (80-21)*2 pop cx loop drawing*
6. podobnie jak pod koniec kroku 4, ostatni wiersz należy narysować poza pętlą *mov ax, 2F2DH mov cx, 21 rep stosw*
7. ustawienie pozycji rysowania na ekranie na pierwsze pole planszy, stanowiące planowaną pozycję początkową gracza, które znajduje się na ekranie w trzecim wierszu i trzydziestej pierwszej kolumnie *mov di, 80*3*2 + 31*2*



Ilustracja 6: Widoczna plansza

Ustawienie pozycji kursora

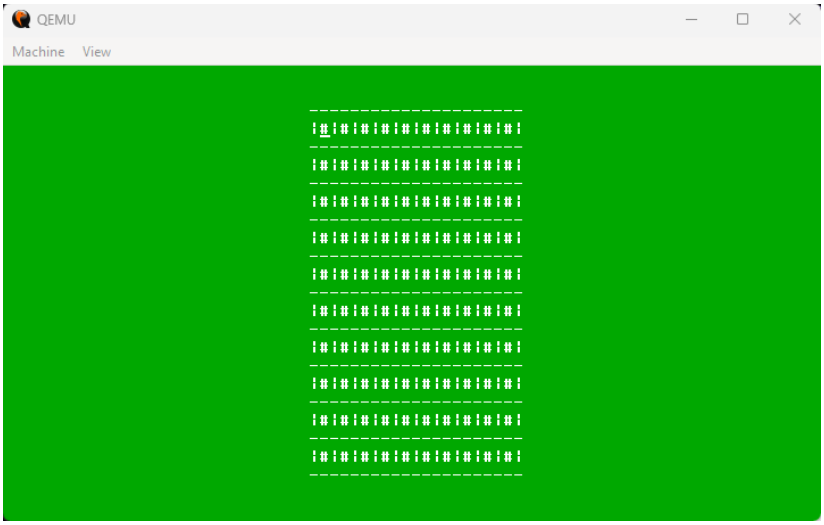
Fragment omawianego kodu:

```
; ustawienie kursora na pozycji startowej
mov ah, 0x2 ; tryb ustawienia kursora
mov dh, [currentRow] ; przypisanie do dh liczby obecnego wiersza
```

mov dl, [currentColumn] ; przypisanie do dl liczby obecnej kolumny
int 10h ; wywolanie odpowiedniego przerwania

currentRow db 3
currentColumn db 31

Kolejną rzeczą, jaka została zaplanowana, była zmiana pozycji kursora. Jak można zauważyć w poprzednim rozdziale, kursor znajduje się w lewym górnym rogu ekranu. Domyślnie powinien wskazywać na pierwsze pole planszy. By to zrobić, wystarczy włączyć w trybie graficznym możliwość ustawienia kursora poprzez odpowiednie przerwanie. Wpisanie do rejestru ax liczby 2 umożliwia wywołanie trybu ustawienia kursora. Rejestr dx przechowuje dwie zmienne, w młodszej części rejestru (rejestr dl) przechowywana jest kolumna kursora, natomiast w starszej części (rejestr dh) wiersz. Z tak ustawionymi parametrami w rejestrach wystarczy wywołać przerwanie. Efekt jest od razu widoczny, co przedstawiono na poniższym zrzucie ekranu.



Ilustracja 7: Widoczna zmiana pozycji kursora na pozycję początkową

Poruszanie kursorem na ekranie

Fragment omawianego kodu:

game:
    move:

        xor ah,ah; zerowanie ah
        int 16h ; pobranie klawisza

        cursor:
            direction:
                cmp al, 'w' ; 'w' - up
                je up

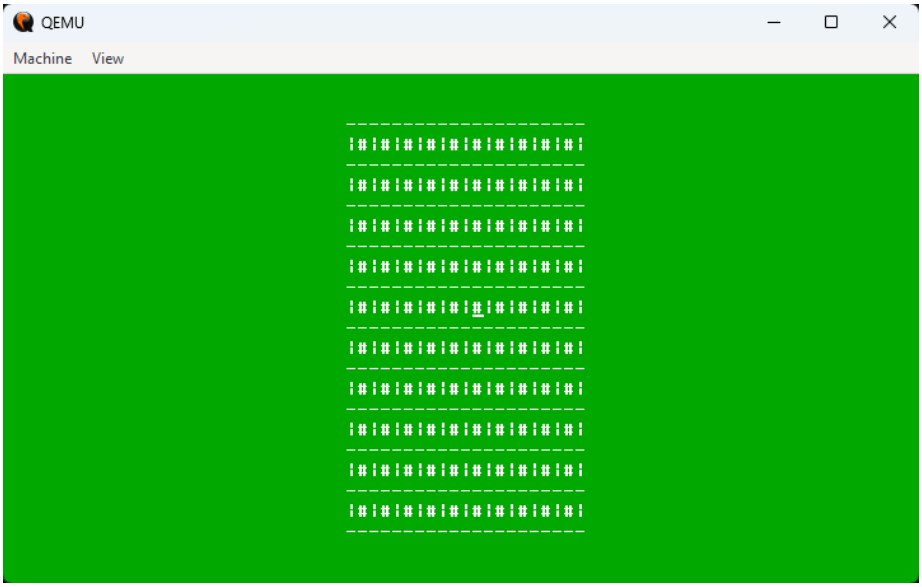
```
                cmp al, 'a' ; 'a' - left
                je left
                cmp al, 's' ; 's' - down
                je down
                cmp al, 'd' ; 'd' - right
                je right
                jmp game

                ; pojscie w prawo
                right:
                        cmp dl, 49 ; sprawdzanie, czy kursor nie wykracza poza tabele z
prawej strony
                        je moveCursor
                        add byte [currentColumn], 2 ; zwiekszenie obecnej kolumny o dwa -
przesuniecie w prawo o dwa pola
                        add di, 4 ; przesuniecie rysowania o dwa pola w prawo
                        jmp moveCursor
                ; pojscie w gore
                up:
                        cmp dh, 3 ; sprawdzanie, czy kursor nie wykracza poza tabele z
gornej strony
                        je moveCursor
                        sub byte [currentRow], 2 ; zmniejszenie obecnego wiersza o dwa -
przesuniecie w gore o dwa pola
                        sub di, 320; przesuniecie rysowania w gore o dwa pola
                        jmp moveCursor

                ; pojscie w dol
                down:
                        cmp dh, 21 ; sprawdzanie, czy kursor nie wykracza poza tabele z
dolnej strony
                        je moveCursor
                        add byte [currentRow], 2 ; zwiekszenie obecnego wiersza o dwa -
przesuniecie w dol o dwa pola
                        add di, 320 ; przesuniecie rysowania w dol o dwa pola
                        jmp moveCursor
                ; pojscie w lewo
                left:
                        cmp dl, 31 ; sprawdzanie, czy kursor nie wykracza poza tabele z
lewej strony
                        je moveCursor
```

```
                    sub byte [currentColumn], 2 ;zmniejszenie obecnej kolumny o dwa -
przesuniecie w lewo o dwa pola
                    sub di, 4 ; przesuniecie rysowania o dwa pola w lewo

            moveCursor:
                    mov ah, 0x2 ; tryb ustawienia kursora
                    mov dh, [currentRow] ; przypisanie do dh liczby obecnego wiersza
                    mov dl, [currentColumn] ; przypisanie do dl liczby obecnej kolumny
                    int 10h ; wywolanie odpowiedniego przerwania

        jmp game
```

Po ustawieniu kursora w odpowiedniej pozycji startowej, pora, aby można było zmieniać jego pozycję. W tym celu należy zaimplementować sterowanie. Aby to zrobić, należy odpowiednio wczytywać wciśnięty klawisz z klawiatury, a następnie wykonać adekwatne do klawisza instrukcje. Pobranie wciśniętego klawisza odbywa się za pomocą wywoływania w pętli odpowiedniego przerwania. Żeby w nim wskazać, że powinno ono zapisać w rejestrze al, jaki klawisz został wciśnięty, należy przed jego wywołaniem wyzerować rejestr ax. Po wywołaniu przerwania wystarczy sprawdzić za pomocą mnemonika *cmp*, czy poszczególne klawisze zostały wciśnięte w danym momencie. Jeśli przykładowo został wciśnięty klawisz ‘d’, to powinno wykonać instrukcje przypisane do tego klawisza. W przypadku gdy wykryto wciśnięcie innego klawisza, niż tego, który został zaimplementowany w kodzie, zostanie wykonana ponownie pętla ze sterowaniem.
Poruszanie w niezależnie którym kierunku składa się z pięciu poleceń (z wyjątkiem poruszania w lewo, gdzie jest jedno polecenie mniej w celu optymalizacji kodu). W każdym z nich wygląda to na podobnej zasadzie, dlatego ten aspekt zostanie omówiony ogólnikowo w krokach:

- na początku sprawdzany jest skrajny warunek, tj. czy po wykonaniu danego ruchu kursor wyszedłby poza wyznaczone ramy planszy.
- Jeśli kursor wyszedłby poza planszę, to nie jest wykonywana de facto żadna zmiana i następuje przejście do końcowej pętli przesunięcia kursora *moveCursor*.
- Gdy kursor może zmienić swą pozycję, następuje przesunięcie pozycji kursora poprzez odpowiednią zmianę zmiennej *currentRow*, bądź *currentColumn.*
- Po zmianie pozycji kursora następuje również zmiana pozycji rysowania na ekranie poprzez odpowiednie dodanie, bądź odjęcie odpowiedniej wartości rejestru di. Gdy kursor przesuwa się w pionie, wartość w rejestrze di zmienia się o **320** (jeden wiersz ma **80** znaków, zapisanie jednego znaku w pamięci wymaga **2** bajtów, natomiast pozycja kursora zmienia się tak naprawdę o **2** znaki, gdyż między polami jest fragment obramowania, zatem **80*2*2=320**), natomiast gdy w poziomie, to zmienia się o **4** (analogicznie jak w pionie, przesunięcie kursora o **2** znaki i zapis na **2** bajtach czyli **2*2=4**).
- Wykonywany jest skok do etykiety *moveCursor* (przy ruchu w lewo nie ma potrzeby wykonania tego skoku, gdyż jego fragment kodu po tej etykiecie znajduje się bezpośrednio po ów ruchu).

- W *moveCursor* wykonywany jest dokładnie ten sam kod, który został przedstawiony w rozdziale Ustawienie pozycji kursora.



Ilustracja 8: Widoczna zmiana pozycji kursora

Odkrywanie pola oraz stawianie flag

Fragment omawianego kodu:

<w etykiecie *direction:* dodano następujący fragment kodu>

```
cmp al, 0DH ; enter
je enter
cmp al, 20H ; space
je space
```

</>

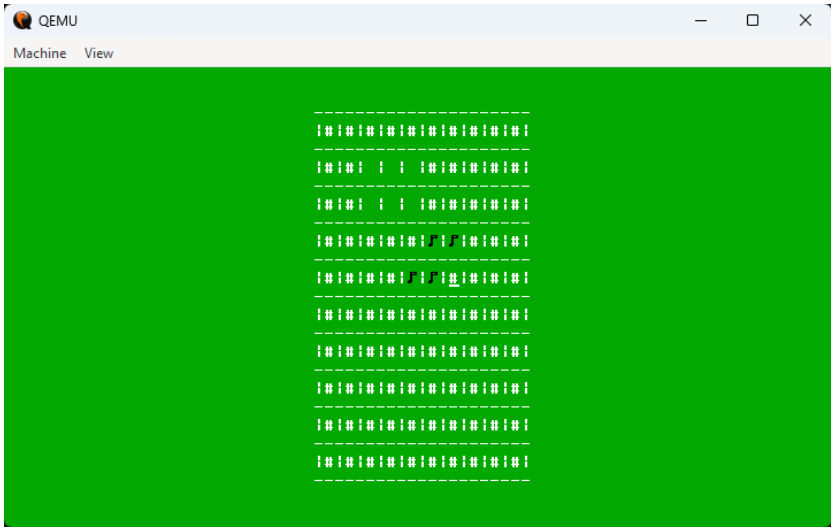
<w etykiecie *cursor:* dodano następujący fragment kodu>

```
; enter odslania dane pole
enter:
        mov [es:di], word 2f00H ; bialy znak na zielonym tle (2F), znak ' ' (00)
        jmp moveCursor

; spacja ustawia flage na danym polu
space:
        mov ax, [es:di] ; pobierz znak z danego pola
        cmp ax, 2F00H ; jesli jest to puste pole, to nic nie rob
        je moveCursor
        cmp al, 0DH ; jesli to pole zawiera flage, to ja zdejmij
        je takeFlag
        ; ustawienie danego pola jako oznaczonego flaga
```

```
mov [es:di], word 200DH ; czarny znak na zielonym tle (2F), znak nuty (0D)
jmp moveCursor
; ponowne ustawienie "niewiadomego" pola
takeFlag:
    mov [es:di], word 2F23H ; bialy znak na zielonym tle (2F), znak '#' (23)
    jmp moveCursor
```

</>

Po udanym zaimplementowaniu sterowania kursorem, pora na główną funkcję w sterowaniu, odkrywanie pola oraz opcjonalnie stawianie flag. Na tym etapie kodu wystarczy sprawdzić, czy odsłanianie pola za pomocą klawisza entera oraz stawianie flag za pomocą klawisza spacji będą prawidłowo działać. W sekcji wykrywania wciśnięcia klawiszy została dodana obsługa entra oraz spacji. W obu przypadkach wykorzystywany jest fakt, że odpowiednia pozycja do rysowania na ekranie została już wcześniej odpowiednio ustawiona przy poruszaniu się kursorem, zatem wystarczy, że będzie się wpisywało odpowiednie znaki do pamięci ekranu. W przypadku wciśnięcia entera, wpisywany jest pusty znak na zielonym tle. Stawianie flagi jest trochę bardziej zawiłe, mianowicie najpierw do rejestru ax pobierany jest znak z aktualnie wskazywanego pola. Jeśli jest to nieodkryte pole, to będzie można postawić flagę. Jeśli na danym polu znajduje się już flaga, to będzie trzeba tę flagę zdjąć i ponownie zostawić nieodkryte pole. Trzecim warunkiem jest sprawdzenie, czy pole nie jest już odkryte. W każdym z warunków zmiana znaku odbywa się na tej samej zasadzie: do pamięci ekranu wpisywany jest odpowiedni znak, a następnie wykonywany jest skok do aktualizacji kursora.



Ilustracja 9: Odsłanianie kilku pól oraz stawianie flag

Generowanie koordynatów min

Fragment omawianego kodu:

<w etykiecie *setup:* dodano następujący fragment kodu>

```
mines:
    push bx
    mov cx, 3 ; ustawianie licznika petli na 3, beda generowane trzy miny
    generate:
```

```
push cx ; zapisz na stosie stan licznika, bo cx bedzie uzyty przy generowaniu wspolrzednych
mov dx, [046CH] ; pobranie wartosci tick od momentu uruchomienia programu
add dx, 3 ; zwiekszanie tickow trzy razy
delay:
    cmp [046CH], dx ; sprawdzanie czy minelo odpowiednio duzo zasu na wpuszczenie do kolejnego etapu generowania
    jl delay

mov ax, dx ; przeniesienie mlodszej czesci liczby z cx:dx
; jako ze ax to 16 bitow, to od razu mozna uzyskac z niej wspolrzedna x i y, ktore mozna spokojnie zapisac na 8 bitach. mozna sie pokusic o zapisywanie obu wspolrzednych w jednej liczbie 8-bitowej, bo liczba 10 miesci sie na czterech bitach
xor ah, ah ; zerowanie starszej czesci ax
mov cl, 8 ; ustawienie dzielnika na 8 (plansza min jest 8x8, objasnienie na samym dole w sekcji "Optymalizacja"), by moc uzyskac liczbe modulo 8 w kolejnych krokach
div cl ; dzielenie al przez 8, w ah bedzie liczba modulo 8
shl ah, 1 ; mnozenie razy dwa wspolrzednej x
add ah, 33 ; dopasowanie do tabelki poprzez dodanie drugiej mozliwej kolumny
mov bl, ah ; wrzucanie do rejestru bl wspolrzednej x
mov ax, dx ; ponowne wpisanie liczby tickow zegara z dx do ax
shr ah, 8 ; zerowanie ah, tym razem w taki sposob, aby jej czesc znalazla sie w al
div cl
shl ah, 1 ; mnozenie razy dwa wspolrzednej y
add ah, 5 ; dopasowanie wspolrzednej do tabelki poprzez dodanie drugiego wiersza tablicy
mov bh, ah ; wspolrzedna y do rejsetru bh
pop cx ; przywracanie licznika petli
push bx ; wpisanie wspolrzednych danej miny na stoie
loop generate
pop bx ; pobierz pierwsza mine ze stosu. W bx sa obie wspolrzedne danej miny - w bl jest wspolrzedna x, w bh jest wspolrzedna y
mov [mine1X], bl ; przypisanie wspolrzednej x
mov [mine1Y], bh ; przypisanie wspolrzednej y
pop bx ; druga mina ze stosu
mov [mine2X], bl
mov [mine2Y], bh
pop bx ; trzecia mina ze stosu
mov [mine3X], bl
mov [mine3Y], bh
pop bx
</>
```

<w etykiecie *direction:* dodano następujący fragment kodu>

```
cmp al, 'x'
je debugX
cmp al, 'c'
je debugC
cmp al, 'v'
je debugV
```

</>

<w etykiecie *cursor:* dodano następujący fragment kodu>

```
; komendy ktore ustawiaja kursor na pozycje poszczegolnych min
debugX:
        mov ah, 0x2
        mov dh, [mine1Y]
        mov dl, [mine1X]
        int 10h
        jmp game
debugC:
        mov ah, 0x2
        mov dh, [mine2Y]
        mov dl, [mine2X]
        int 10h
        jmp game
debugV:
        mov ah, 0x2
        mov dh, [mine3Y]
        mov dl, [mine3X]
        int 10h
        jmp game
```

</>

```
mine1X db 0
mine1Y db 0
mine2X db 0
mine2Y db 0
mine3X db 0
mine3Y db 0
```

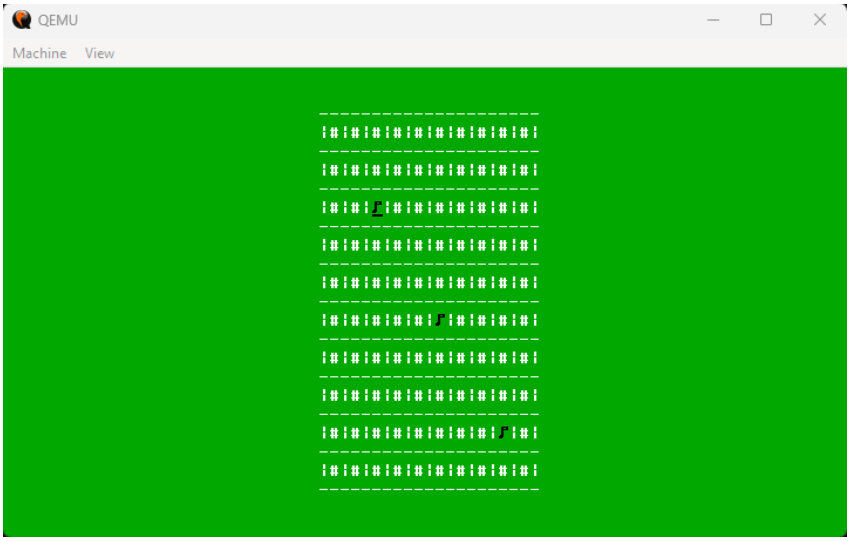
Jednym z kluczowych aspektów gry “Saper” jest jej losowość. Gracz nie powinien zatem wiedzieć już od samego początku, gdzie dokładnie znajdują się rozsiane po planszy miny. Należy więc wprowadzić losowe generowanie współrzędnych min. Ze względu na rygorystyczne

ograniczenia dostępnego miejsca na kod, w tej grze generowane są jedynie trzy miny. Z tych samych względów losowanie koordynatów min stanowi jedynie namiastkę losowości. Mimo tych niedogodności została podjęta próba sprostania temu wyzwaniu.

W krokach zostanie przedstawiony proces generowania współrzędnych min:

- na początku działania programu zostały zaalokowane w pamięci zmienne określające współrzędne poszczególnych min w osiach X i Y.
- Dla bezpieczeństwa zawartość rejestru bx jest zapisywana na stosie. Ów rejestr będzie wykorzystany do generowania koordynatów.
- Rejestr cx służy jako licznik, ile min powinno być wygenerowanych.
- Od tego momentu zaczyna się faktyczny proces generowania współrzędnych miny.
    - Na stos zapisywany jest rejestr cx.
    - Do rejestru dx zapisywana jest wartość aktualnego tick zegara, a następnie jej wartość jest zwiększana o trzy, aby móc opóźnić działanie programu, by zapewnić pseudolosowość.
    - Pobrana wartość umożliwia generowanie dwóch współrzędnych naraz, współrzędną X i współrzędną Y.
    - Na wstępie generowana jest współrzędna x poprzez odpowiednie operowanie wyrażeniami arytmetycznymi:
        - obszar, na którym mogą być miny, został ustawiony na rozmiar 8x8, stąd też ustawiany jest dzielnik równy 8, który umożliwi działanie modulo 8 na wartości tick.
        - Jako że koordynaty dotyczą pozycji znaku na ekranie, uzyskana liczba jest pomnożona razy dwa, gdyż pola, na których może być mina, są od siebie oddalone co dwa znaki.
        - Do tej liczby dodawana jest wartość pierwszej możliwej kolumny, na której może znajdować się mina.
    - Ponownie pobierana jest wartość tick do rejestru ax, żeby móc wygenerować współrzędną Y
    - Generowanie w tym momencie działa na podobnej co poprzednio, z tą różnicą, że wykorzystana jest starsza część wartości tick, gdyż obie współrzędne zapisane są na 8 bitach każdy, a wartość tick na 16 bitach.
    - Obie współrzędne zapisywane są do rejestru bx
    - Zawartość rejestru bx, czyli koordynaty miny, jest zapisywana na stos
- Ze stosu pobierane są po kolei współrzędne min
- Zapisywane są one do poszczególnych zmiennych min

Żeby móc zobaczyć pozycje, w których znajdują się miny, tymczasowo zaimplementowane zostały opcje ustawienia kursora na nie pod poszczególne klawisze. W sekcji sterowania zaimplementowane zostało wykrywanie klawiszy ‘X’, ‘C’ oraz ‘V’. Każdy z klawiszy ustawia kursor na tej samej zasadzie, co w rozdziale Ustawienie pozycji kursora, z tą różnicą, że zamiast *currentRow* i *currentColumn* wpisywane są mineKX i mineKY, gdzie K to numer miny.



Ilustracja 10: Zaznaczone flagami pozycje min

Sprawdzanie kolizji z miną i warunek końcowy

Fragment omawianego kodu:

<w etykiecie *enter:* usunięto poprzedni kod i dodano następujący>

```
; enter odslania dane pole
enter:
        ; wrzucanie na stos poszczegolnych wspolrzednych min

        ; mina pierwsza
        mov ax, [mine1Y]
        push ax
        mov ax, [mine1X]
        push ax

        ; mina druga
        mov ax, [mine2Y]
        push ax
        mov ax, [mine2X]
        push ax
        ; mina trzecia
        mov ax, [mine3Y]
        push ax
        mov ax, [mine3X]
        push ax
        ; zerowanie cx, ktory bedzie licznikiem petli
        xor cx, cx
        ; zmniejszanie cx do -1, by obieg petli mozna bylo zaczac od 0
```

```
dec cx

singleFieldLookUp:
    xor ax, ax ; zerowanie reejstru ax, ktory posluzy za licznik wystapien min wokol danego pola
    mov [mineBool], ax
    horizontal:
    inc cx ; nalezy juz za wczasu zwiekszyc licznik petli, gdyz jesli w trakcie sprawdzania warunkow wyjdzie, ze wspolrzedna x nie miesci sie w danym przedziale, to nie zwiekszyloby inaczej indeksu
    cmp cx, 3 ; jesli petla wykonuje sie poraz 4, to znaczy, ze wspolrzedna x jakiejkolwiek miny nie miescila sie w przedziale <-1;1> wzgledem przeszukiwanego pola
    je verticalStart
    ; w tym miejscu powinno wystapic adresowanie indeksowane ([sp+4cx]) lecz nie jest to mozliwe w 16 bitach, dlatego wystepuje tutaj obejscie
    mov si, cx ; do czystego si dodac cx
    shl si, 2 ; przesuniecie w lewo trzy razy powoduje pomnozenie 4 razy
    add si, sp ; dodanie sp, dzieki czemu jest sp+4cx

    mov dl, [currentColumn] ; przypisanie danej kolumny

    sub dl, 2 ; zmniejszanie kolumny o jedna kolumne w tablicy - sprawdzanie poczatku przedzialu

    cmp [si], dl ; [si] to [sp+4cx]
    jl horizontal
    ; wspolrzedna x miny jest wieksza rowna (wspolrzednej x szukanego pola - 1)
    add dl, 4 ; przesuniecie o dwa pola w tablicy w prawo
    cmp [si], dl

    jg horizontal
    bts [mineBool], cx
    sub dl, 2
    cmp [si], dl
    jne horizontal
    add cx, 4
    bts [mineBool], cx
    sub cx, 4
    jmp horizontal

    ; przejscie do szukania w poprzek
    verticalStart:
        xor cx, cx ; zerowanie cx
```

```
        dec cx ; cx do -1, by petla zaczynala sie od 0
    vertical:
        inc cx
        cmp cx, 3
        je endFind ; zakonczenie petli szukajacej liczby min blisko danego pola

        mov si, cx
        shl si, 2
        add si, sp
        inc si
        inc si

        mov dh, [currentRow]
        sub dh, 2
        cmp [si], dh ; [bx] = [sp+4cx+2]
        jl vertical
        ; wspolrzedna y miny jest wieksza rowna (wspolrzednej y szukanego pola - 1)
        add dh, 4
        cmp [si], dh

        jg vertical
        sub dh, 2 ; powrot do pola startowego
        cmp [si], dh ; sprawdzanie, czy na szukanym polu jest wspolrzedna y miny
        jne bool ; jesli nie, to tylko zaznacz niebezpieczenstwo

        ; wspolrzedna y miny jest na szukanym polu
        add cx, 4 ; dodaj 4 do licznika, by moc sprawdzic, czy wspolrzedna x miny tez sie
zgadza z obencym x
        bt [mineBool], cx ; sprawdzanie wspolrzednej x, czy tez jest taka sama
        jc explodeMine ; jesli tak, koniec gry
        sub cx, 4 ; nie zgadza sie, powrot do petli
; wspolrzedna y sie zgadza, pora zobaczyc, czy wspolrzedna x danej miny rowniez sie zgadza
        bool:
            bt [mineBool], cx ; jesli i wspolrzedna x, i wspolrzedna y mieszcza sie w
danym przedziale, to mina jest blisko danego pola i w carry flag bedzie 1
            adc ax, 0 ; zwiekszenie licznika min o jeden, jesli mina jest blisko
            jmp vertical ; sprawdzanie kolejnych min, jesli jeszcze sa
    explodeMine:
        mov [es:di], word 2F2AH ;  bialy znak na zielonym tle (24), znak '*' (2A)
        ; osobna petla, by nie nadpisywac znakow
        explodeLoop:
```

```
                inc di
                mov [es:di], byte 4FH ; ustawienie czerwonego tla i bialych znakow na
ekranie
                inc di
                jmp explodeLoop

        ; zakoczono przeszukiwanie liczby min
        endFind:
                add sp, 12 ; zwalnianie stosu ze wczesniej wrzuconych na niego
wspolrzednych min (2*6 = 24)
                cmp ax, 0 ; czy zliczono w danym polu jakiekolwiek miny w poblizu
                jne numberField

                ; nie znaleziono min, w takim razie puste pole
                mov [es:di], word 2F00H ; bialy znak na zielonym tle (2F), znak ' ' (00)
                jmp moveCursor

        ; jest chociaz jedna mina w poblizu
        numberField:
                add ax, 2F30H ; bialy znak na zielonym tle (2F), znak '0' (30)
                mov [es:di], ax
                jmp moveCursor

mineBool db 0
```

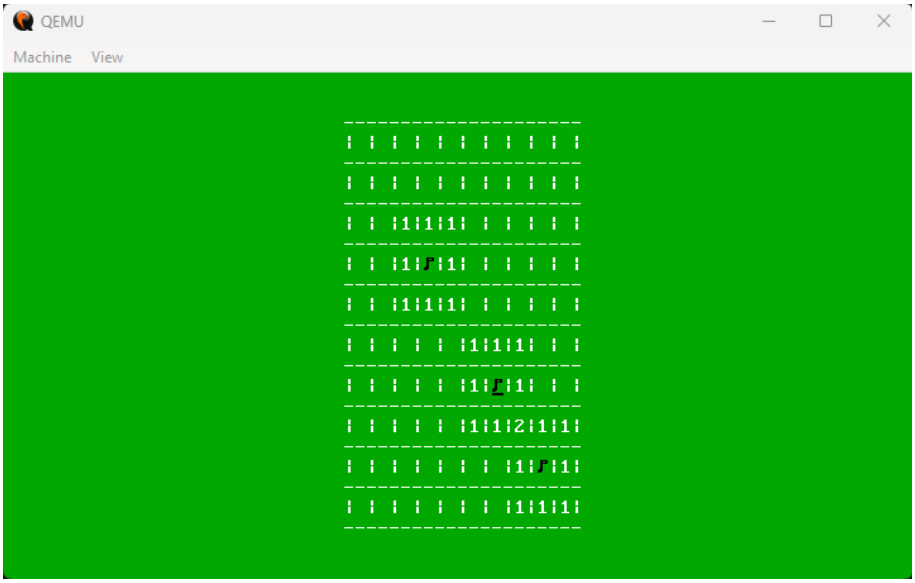
Ostatnim elementem zanim gra będzie w pełni funkcjonalna, jest sprawdzanie kolizji z minami oraz związany z tym (jedyny) warunek końcowy gry. Mimo obszernego fragmentu kodu, jego działanie nie wymaga równie obszernego wyjaśnienia. W tym momencie kod bez optymalizacji rozmiaru programu w formie binarnej uniemożliwiał skompilowanie pełnego kodu, zatem doszło do nieplanowanych ustępstw. Początkowo w planach było odkrywanie pola 3x3 oraz kilka warunków końcowych gry. Niestety w ostatecznej formie programu możliwe jest odkrywanie wyłącznie pojedynczego pola i tylko jeden warunek końcowy - nadepnięcie na minę.

Sprawdzanie kolizji z minami

W krokach zostanie przedstawione jego działanie:

- na początku wrzucane są na stos współrzędne min.
- Wyzerowanie rejestru cx, a następnie dekrementowanie go do -1.
- Następuje pętla sprawdzająca pojedynczego pola:
    - wyzerowanie rejestru ax, by móc w nim zliczać liczbę wystąpień min wokół

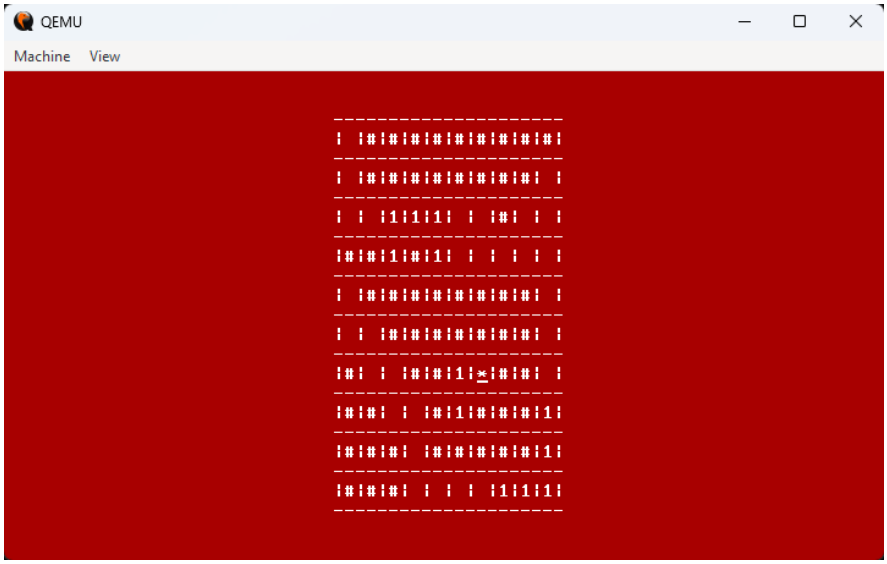
rozpatrywanego pola.
    - Wpisanie zera do zmiennej przechowującej liczbę min wokół danego pola
    - Inkrementacja wartości rejestru cx, a następnie sprawdzenie, czy wszystkie miny były rozpatrzone.
    - Następuje sprawdzanie, ile min znajduje się w przedziale <-1:1> od danego pola w osi poziomej. Przykładowo, jeśli mine1X = 3, mine2X = 4, mine3X = 6, a współrzędna X kursora wynosi 4, to funkcja w tym momencie zapisuje do zmiennej mineBool, że mina numer 1 i mina numer 2 są w pobliżu w pobliżu pola w osi poziomej, za pomocą ustawienia odpowiednich bitów na 1. mineBool w tym momencie wynosi: 0000 0011b
    - Dodatkowo ów funkcja sprawdza, czy współrzędna X min, które są w pobliżu, zgadza się dokładnie ze współrzędną X kursora. Jeśli występuje taka sytuacja, to w zmiennej mineBool również ustawiany jest na odpowiedniej pozycji bit w starszej części bajtu. We wspomnianym w poprzednim punkcie przykładzie, występuje taka sytuacja, zatem mineBool wynosi: 0010 0011b.
    - Przeszukiwanie przedziału w osi pionowej działa na dokładnie tej samej zasadzie, co w poziomie, z tym że zamiast używania zmiennej mineBool w przypadku, gdy współrzędna Y miny znajduje się w przedziale <-1:1>, inkrementuje się wartość rejestru ax.
    - Różnica w przeszukiwaniu jednak występuje w przypadku szukania dokładnie tej samej współrzędnej Y min, co kursora. Gdy współrzędna Y miny zgadza się z pozycją kursora, sprawdzany jest odpowiedni bit w mineBool. Kontynuując przykład z przeszukiwania osi poziomej: zakładając, że mina numer 2 ma również tę samą współrzędną Y, co kursor, następuje sprawdzenie bitu 2-1+4 (gdzie 2 to numer miny, -1 to uregulowanie pozycji w bajcie, gdyż pozycje liczone są od 0, a 4 to przesunięcie do starszej części bajtu).
    - Jeśli bit na tej pozycji jest równy 1, następuje koniec gry (poprzez przejście do etykiety *explodeMine*), gdyż koordynaty miny zgadzają się z pozycją kursora. Gdy bit jest równy 0, w tym momencie nic się nie dzieje, gdyż liczba min w pobliżu pola już została zwiększona wcześniej.
- Po zakończeniu przeszukiwania min w pobliżu pola, następuje przejście do etykiety *endFind*.
- W niej następuje zwolnienie z pamięci stosu współrzędnych min oraz wpisanie do pamięci ekranu liczby min w pobliżu przeszukanego pola na dokładnie tej samej zasadzie, co przy rysowaniu planszy w rozdziale Implementacja planszy.



Ilustracja 11: Wyczyszczona plansza z minami

Warunek końcowy gry

W poprzednim podrozdziale wspomniane zostało, że w przypadku nadepnięcia na minę, następuje koniec gry. Jest to jedyny warunek końcowy. Zawarty jest on w fragmencie kodu związanym z etykietą *explodeMine*. Najpierw wstawiany jest znak oznaczający nadepniętą minę. Następnie następuje nieskończona pętla, w której tło ekranu jest zmieniane na kolor czerwony, oznaczający porażkę. Od tego momentu gracz nie ma możliwości poruszania się, zatem jest to ostateczne zakończenie rozgrywki. Żeby móc ponownie zagrać w grę, należy uruchomić ponownie komputer, na którym znajduje się program.



Ilustracja 12: Koniec gry

Optymalizacja

Jednym z motywów przyświecających przy pisaniu programu była optymalizacja rozmiaru kodu. Z powodu rygorystycznego ograniczenia przestrzeni, na której mógł znajdować się kod - 512 bajtów, każdy zaoszczędzony bajt był na wagę złota. W programie musiało nastąpić wiele uproszczeń pewnych rozwiązań. Niekiedy należało również znaleźć sposób na uzyskanie tego

samego efektu poprzez wykorzystanie poleceń niekoniecznie przychodzących od razu na myśl.

Jedną z pierwszych rzeczy, dzięki którym można było oszczędzić kilka bajtów, było zamienianie pewnych mnemoników na inne. Przykładowo zamiast napisać *mov ax, 0*, lepiej jest użyć *xor ax, ax*, gdyż pierwsze polecenie zajmuje w programie bajt więcej. W ten sam sposób można oszczędzić bajt, gdy zamiast *add ax, 1*, użyje się *inc ax*, jeśli autor kodu chce zwiększyć wartość w rejestrze o jeden. Z w miarę powszechnie używanych, lecz już mniej oczywistych optymalizacji, można jeszcze wyróżnić wykorzystane w programie *shl si, 1*, dzięki któremu jednym rozkazem można pomnożyć zawartość rejestru si razy 2, zamiast pisać *mov ax, 2 mul si mov si, ax*, które zajmuje zdecydowanie więcej bajtów.

Można też zmniejszyć rozmiar zajmowanego miejsca przez program, gdy będzie się wykonywało operacje na liczbach znajdujących się już w samej pamięci. W niektórych przypadkach nie ma potrzeby, aby liczby te najpierw znalazły się w którymś z rejestrów, wykonano w rejestrze daną operację i na koniec z powrotem wpisywało je do pamięci. Takim przykładem z kodu może być *add byte [currentColumn], 2*, gdzie do zmiennej przechowywanej w pamięci można od razu dodać liczbę 2. Nie ma potrzeby by pobierać zmienną *currentColumn* do rejestru, dodania w rejestrze 2 i wpisania tego z powrotem do tej zmiennej. Jest to co prawda mniej eleganckie rozwiązanie, natomiast pozwala ono zaoszczędzić kilka bajtów.

Istnieją również mnemoniki w asemblerze x86, które zostały stworzone po to, by za pomocą jednego polecenia móc wykonywać dwie lub więcej czynności na raz, oszczędzając przy tym mnóstwo bajtów. Jednym z nich jest stosowany w programie *stosw* w połączeniu z mnemonikiem *rep*. *stosw* to tak naprawdę użycie *mov [es:di], ax add di, 2*, natomiast mnemonik *rep* powtarza dany mnemonik tyle razy, ile zostało to wpisane w rejestrze cx. Analogiczny kod złożony z nierozdzielnych mnemoników, który zajmuje zdecydowanie więcej bajtów, wyglądałby następująco:

```
petla:                  ;       |rep
    mov [es:di], ax     ;|stosw |
    add di, 2           ;|      |
    loop petla          ;       |
```

W powyższym kodzie jako komentarz zaznaczono zakres działania danych mnemoników, które zastępują ten kod.

Inną strategią na optymalizację kodu, poza używaniem odpowiednich mnemoników, jest pisanie kodu w odpowiedni sposób. Początkowo program był pisany w taki sposób, że każda pojedyncza rzecz w programie była uwzględniana z osobna. Dobrym przykładem obrazującym takie pisanie było sprawdzanie kolizji z minami. Początkowo w programie nie było żadnych pętli, które sprawdzały miny. Każda z min była rozpatrywana osobno w jednym ciągu. Po pierwsze było to niepotrzebne powtarzanie niemal tego samego kodu z drobnym zmienianiem zmiennych, a po drugie zajmowało to za dużo miejsca w bajtach. Rozwiązaniem problemu z przykładu było wykorzystanie pamięci stosu i traktowanie min jako obiektów danej klasy. Wystarczy napisać pętle sprawdzającą współrzędne min, które będą przechowywane w pamięci stosu. Niemal w każdym aspekcie jest to lepszy sposób, gdyż oszczędza on zdecydowanie dużo miejsca, jest bardziej czytelny i pozwala na w miarę szybką zmianę w kodzie, gdyby zmieniła się liczba min.

Niekorzystnym, lecz koniecznym sposobem na zmniejszenie rozmiaru programu, jest obcięcie jego zawartości. W ten sposób nie zostało zaimplementowane odkrywanie pól 3x3 czy też lepszy sposób generowania koordynatów min, gdyż zajęłyby one zdecydowanie za dużo miejsca.

Zakończenie

Projekt ten z pewnością należał do jednych z najciekawszych w toku studiowania. Pozwolił on na wybór tematu, z którym czułem się odpowiednio. Sposób prowadzenia tego projektu sprawił również, że zmieniło się trochę moje podejście do zarządzania projektem. Pierwszy raz spotkałem się z czymś takim, by to przed rozpoczęciem tworzenia programu zbierać obszerne notatki, a nie w trakcie. W poprzednich projektach od razu podejmowałem się pisania kodu, przez co sam ten proces zajmował zdecydowanie więcej czasu, gdyż potrzebowałem sprawdzać po kolei prace innych programistów.

Projekt nauczył również zachować umiar w optymalizowaniu kodu. W trakcie pisania tego dokumentu, odkryłem sposób na zmniejszenie rozmiaru programu w jednym miejscu o jednego bajta. Wiązało się to jednak z niekorzystną zmianą we wzorze, w jaki układają się generowane miny w grze. Mimo tego, że powinien być ten sam efekt, co przed optymalizacją, z niewiadomego powodu zmienia się sposób, w jaki układają się miny. Z tego powodu ów optymalizacja i tak nie trafiłaby do ostatecznej formy programu.

Addendum

Jest to sprawozdanie, które zostało napisane na potrzeby studiów. W tej wersji, którą właśnie czytasz, zostało ono delikatnie zmodyfikowane, by nie odnosiło się bezpośrednio do spraw uczelnianych itp.. Świadomy tego, że kod w tym dokumencie może nie być czytelny, polecam przegląd kodu źródłowego. Zawarte są w nim również dodatkowe komentarze, które nie zostały przedstawione w tym sprawozdaniu.

Dziękuję za zainteresowanie się moim projektem jak również za poświęcony czas, który przeznaczyłeś by przeczytać ten dokument!