VEGOVA | ELEKTROTEHNIŠKO-RAČUNALNIŠKA STROKOVNA ŠOLA IN GIMNAZIJA LJUBLJANA

DOOM na mikrokrmilniku ESP32

Raziskovalna naloga

Računalništvo in informatika

Avtor: Luka Leskovšek, G 4. B

Mentor: Aleš Volčini

Ljubljana, 2023

Kazalo

Povzetek

V računalniški sferi pogosto zasledimo razne presenetljive zanimivosti. Ena od njih je, na katero platformo je programerjem uspelo spraviti in igrati igro Doom. V tej raziskovalni nalogi sem opisal, kako sem na novo in v svojem grafičnem pogonu sprogramiral klasično tridimenzionalno igro Doom, pri čemer sem uporabil le najosnovnejše knjižnice. Igra teče na mikrokrmilniku ESP32-S3 N8R8, ki generira tudi sliko v VGA-načinu, za vhodno napravo pa uporablja kar tipkovnico s PS/2-priključkom. V nalogi sem se poglobil v delovanje grafičnih pogonov, velik poudarek pa sem dal optimizaciji delovanja in postopku risanja sveta.

Abstract

In the world of computer science we often find various points of intrigue. One of them is the variety of platforms to which programmers have managed to port Doom to. In this research paper I describe, how I programed the classic 3D game Doom anew and with my very own graphical engine, for which I only used the most basic of libraries. The game is intended to be ran on an ESP32-S3 N8R8 microprocessor, which simultaneously generates a VGA image. A PS/2 keyboard is used for data collection. In this paper I delve into the inner workings of graphical engines with a large emphasis on optimisation and the process of drawing the world.

1 Uvod

Ta projekt se je začel predvsem kot šala. Na internetu je namreč znana šala o tem, da na napravah, ki sploh niso namenjene temu, da bi na njih tekla igra Doom, teče igra Doom. To je brezčasna igra proizvajalcev ID software, ki je izšla leta 1993 in postavila standard za to, kako naj bo videti igra v žanru "streljačina" ("first person shooter") [1]. A za namene te raziskovalne naloge ima le vlogo dobrega primerka zgodnje računalniške grafike v treh dimenzijah. Sprva sem menil, da je kljub vsem tehnološkim napredkom v zadnjih 30 letih igra Doom še vedno prezahtevna za skromen mikrokrmilnik, a več kot sem o tem premišljeval, bolj sem bil prepričan, da je možno. Najprej bi moral doseči, da se izhod mikrokrmilnika prikaže na ekranu. Odločil sem se uporabiti grafični protokol VGA, saj je zaradi starosti zelo enostaven in primeren za nižje ločljivosti. Za vnos podatkov sem izbral tipkovnico s protokolom PS/2 predvsem zato, ker sem jo pač imel pri roki, a kasneje se je izkazalo, da je bila to zelo dobra izbira. Za mikrokrmilnik sem izbral ESP32-S3 N8R8, kratko rečeno zato, ker je zelo zmogljiv. Podrobneje so razlogi predstavljeni v 4. poglavju. Za programski jezik sem izbral C zaradi hitrosti, zanesljivosti in osebne preference. Dodatno sem si postavil še en izziv: uporabil ne bom nobenih knjižnic, ki niso namenjene poganjanju ESP-ju vgrajenih strojnih naprav. Dopustil sem si le uporabo knjižnic math.h, string.h in stdio.h.

Hipotezi naloge sta sledeči:

- Mikrokrmilnik ESP32-S3 N8R8 je sposoben primerno hitro (hitrost osveževanja 30 slik na sekundo) poganjati program, ki je prvi približek igre Doom.
- Mikrokrmilnik je ob poganjanju Doom-a sposoben tudi generirati VGA sliko.

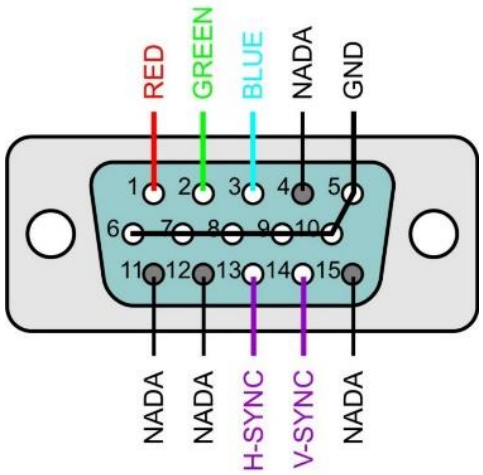
2 Protokol VGA

VGA je protokol, ki je namenjen prenašanju vizualnih informacij z računalnika na zaslon.

2.1 Sestava

Protokol uporablja sistem šestih žic [2]:

- Tri žice za prenos vizualnih podatkov. Vsaka izmed njih prenaša eno izmed treh temeljnih barv: rdeča, zelena in modra.
- Dve žici za sinhronizacijska signala: vertikalna sinhronizacija ali "V-sync" in horizontalna sinhronizacija ali "H-sync."
- Ena žica za električno ozemljitev.

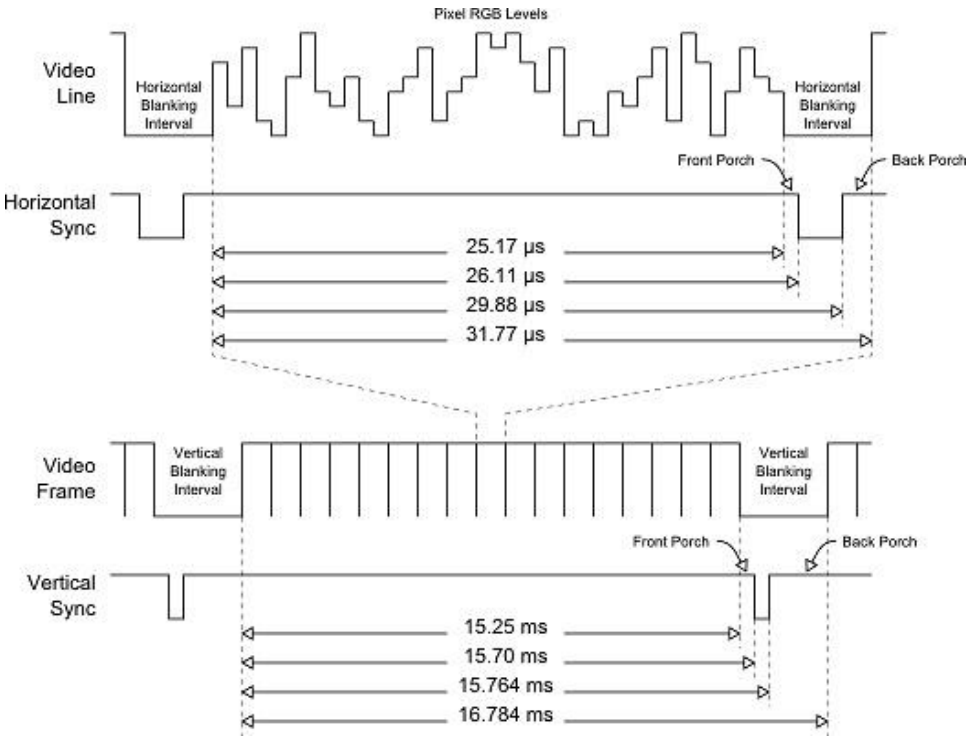


Slika 1: VGA priključek

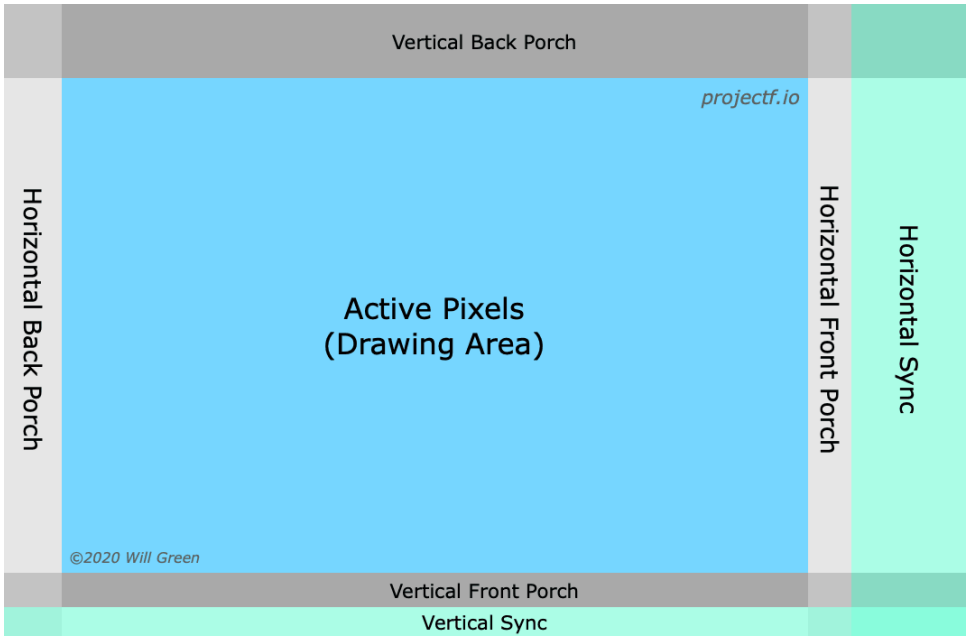
Za prenos slike računalnik spreminja napetost na podatkovnih žicah s točno določeno modulacijo, ki je odvisna od ločljivosti in hitrosti osveževanja zaslona. V pomoč uporablja V-sync in H-sync signala.

2.2 Delovanje

Vsakič, ko računalnik pride do konca vrstice, nastavi H-sync signal na 0 V (v prvotnem stanju je na 5 ali 3,3 V). To sporoči zaslonu, naj prestavi kraj risanja na začetek naslednje vrstice, ne glede na to, kje trenutno riše. Preden se lahko to zgodi, mora računalnik najprej za nekaj časa pred in po tem signalu nastaviti vse podatkovne žice na nizko napetost. Ta časovna odseka se imenujeta "front porch" in "back porch." Skupaj s časom, kjer je sam H-sync ugasnjen, se to imenuje horizontalni "blanking" oz. zatemnitveni interval. Podobno velja za delovanje navpičnega sinhronizacijskega signala V-sync, razen da ta prestavi kraj risanja nazaj na zgornji del zaslona in se aktivira ob koncu risanja ene slike na zaslonu, zato je vertikalni zatemnitveni interval mnogo daljši od horizontalnega (pogosto traja toliko kot več vrstic skupaj).



Slika 2: Prikaz časovnega poteka protokola VGA



Slika 3: Shematični prikaz protokola VGA

Frekvenca osveževanja zaslona je enaka frekvenci V-sync signala, vertikalno ločljivost pa določa H-sync signal. Vertikalna ločljivost je odvisna od tega, kolikokrat se sproži H-sync signal v enem intervalu V-sync signala (to izračunamo tako, da delimo frekvenco V-sync signala s frekvenco H-sync signala). Horizontalna ločljivost je odvisna izključno od t. i. piksel clock-a. Ta določa koliko časa traja, da signal opiše eno točko na ekranu.

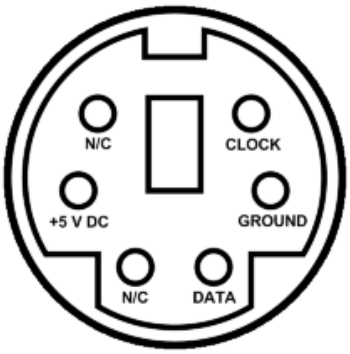
3 Protokol PS/2

PS/2 je protokol, ki je bil prvotno namenjen prenosu podatkov med računalnikom in vnosnimi napravami, kot sta npr. tipkovnica in miška [4].

3.1 Sestava

Kljub temu, da ima standardni PS/2 priključek 6 pinov, so uporabljeni le 4. Ti so:

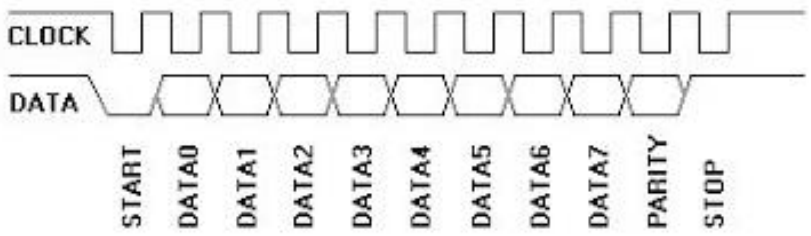
- +5 V za napajanje naprave vnosne naprave.
- Električna ozemljitev (GND).
- Prenos podatkov.
- Sinhronizacija.



Slika 4: PS/2 priključek

3.2 Delovanje

Za prenos podatkov je potrebno najprej podatkovni signal spustiti na 0 V, da se zaznamuje začetek sporočila. Nato lahko pošljemo 8 bitov, ki predstavljajo podatek, ki ga želimo poslati. Na koncu sta še paritetni bit, ki je namenjen zaznavanju morebitnih napak v prenosu, in dvig signala nazaj na 5 V, da se zaznamuje konec sporočila. V času prenosa signal za sinhronizacijo pomaga pri časovno pravilnem branju podatkov. Časovni diagram tega postopka je razviden na sliki 5.



Slika 5: Prikaz časovnega poteka protokola PS/2

3.3 Implementacija

Signal s tipkovnice, če ga uporabljamo samo v smer proti računalniku, izgleda pravzaprav kot standarden UART protokol z dodano sinhronizacijo, le da je pri UART frekvenca znana že vnaprej. Torej lahko za prejem podatkov uporabimo strojno napravo za prejemanje UART signalov, ki jo premore vsak spodoben mikrokrmilnik. Posledica je, da signala za sinhronizacijo sploh ni treba priključiti, kadar poznamo sinhronizacijsko frekvenco tipkovnice. Med tipkovnico in mikrokrmilnikom ESP32 moramo vključiti tudi delilnik napetosti, saj ESP-jevi GPIO-ji niso sposobni prenašati 5 V.

Po prejemu podatkov jih je potrebno dekodirati. PS/2 namreč uporablja drugačne oznake za določene črke kot ASCII [5], ki sem ga uporabil za notranje obdelovanje podatkov. Ker ASCII nima znakov za nekatere tipke na tipkovnici (npr. puščica gor, shift, F5), sem te tipke pripisal neuporabljenim kontrolnim znakom (0-31). Protokol tudi zaznamuje, kdaj je tipka spuščena, tako da pred identifikacijsko oznako tipke pošlje še šestnajstiško število 0xf0. Tako je možno vedno vedeti natanko, katere tipke so pritisnjene in katere ne.

4 ESP32-S3 N8R8

Preden lahko začnemo s pisanjem kode, moramo najprej izbrati mikrokrmilnik.

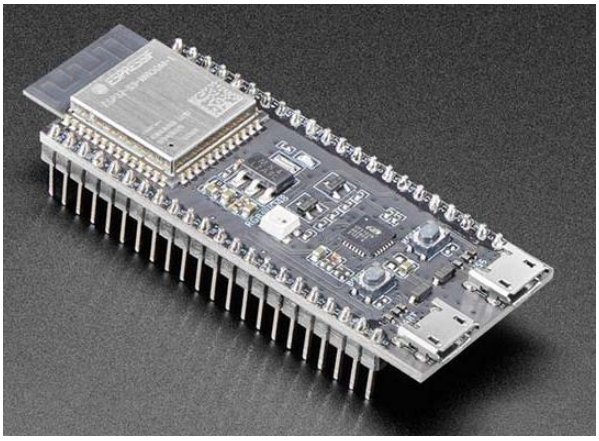
4.1 Uvod

Glavni omejevalni faktor pri tem projektu je hitrost. Protokol VGA in igra Doom sta namreč kljub starosti še vedno zelo zahtevna za mikrokrmilnik. Zato želimo izbrati najzmogljivejšo napravo, ki jo lahko najdemo. Poleg tega mora izpolnjevati še nekaj dodatnih pogojev: vsaj nekaj sto KB notranjega spomina, vsaj nekaj MB programskega spomina, vsaj 20 splošno namenskih (GPIO) pinov in po možnosti tudi kaj, kar bi pomagalo pri poganjanju VGA-ja.

4.2 Zahteve

V ta namen je pametna izbira ESP32-S3 N8R8, ki izpolni vse navedene potrebe [6]:

- Zelo hitra centralna procesna enota ali CPE (2 jedri x 240 MHz)
- Strojna naprava, ki omogoča generiranje signalov kot pri VGA
- Dovolj veliko število GPIO-jev (36)
- Dovolj notranjega spomina (512 KB)
- Dovolj programskega spomina (8 MB)



Slika 6: Mikrokrmilnik ESP32-S3 N8R8

5 Implementacija signala VGA

Mikrokrmilnik moramo pripraviti do tega, da generira VGA signal.

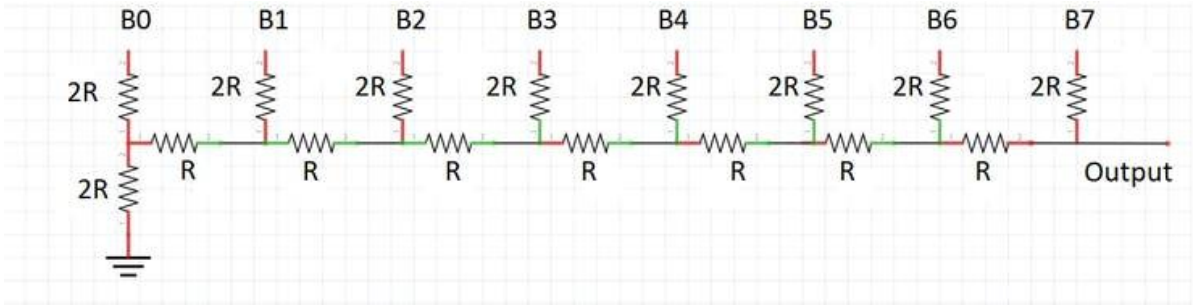
5.1 Strojna podpora generiranju signalov LCD_CAM

Za poganjanje VGA-ja se uporablja strojna napravo LCD_CAM, ki je namenjena poganjanju modernejših zaslonov, a ima vse potrebno, kar zahteva tudi VGA protokol [7]. Grobo rečeno, ko je pravilno nastavljena in ima primerno podporo zunanje strojne opreme, je sposobna sama generirati želen signal. Podatke vzame iz medpomnilnika, ki si ga sama v spominu rezervira (velikost medpomnilnika je sicer odvisna od podane ločljivosti, to bo kasneje povzročalo probleme). Uporabnik lahko tedaj v medpomnilnik zapiše, kar želi, da se prikaže na ekranu.

A za to je treba napravo najprej usposobiti. Glavni problem je sledeč: VGA je analogni signal, ESP sam po sebi pa ni sposoben generirati analognih vrednosti. Zato vezje LCD_CAM deluje tako, da podane podatke enostavno prestavi na pine GPIO. Uporabniku je prepuščeno, da iz tega naredi kaj uporabnega. Te digitalne signale smo spremenili v analogne, v ta namen smo uporabili t. i. R2R DAC (ang. digital to analog converter), katerega diagram je razviden na sliki 7.

5.2 R2R DAC

R2R DAC vzame za vhod želeno število digitalnih signalov in jih skupaj združi v analogno vrednost, ki se jo izračuna tako, da se zaporedje vhodov prebere kot binarno število in nato pomnoži s faktorjem $\frac{V_{in}}{2^{št.\ vhodov}}$ [8]. Tako se lahko za želeno napetost na register, ki določa trenutno stanje pinov GPIO, vpiše vrednost, ki se jo izračuna tako, da napetost pomnožimo s faktorjem $\frac{2^{št.\ vhodov}}{V_{in}}$. ESP ima prvotno napetost na GPIO-jih 3,3 V, torej če bi imeli zgrajen R2R DAC z 8 vhodi in bi želeli generirati napetost 2 V, bi v register zapisali 2^8×2V/3,3V~=155, kar bi dalo najboljši približek 1,998 V. Ločljivost te vrste DAC-a je odvisna od števila bitov, torej če bi želeli vrednost, ki je bližja 2 V, bi morali povečati št. vhodov.



Slika 7: Shema R2R DAC-a

5.3 Omejitve LCD_CAM

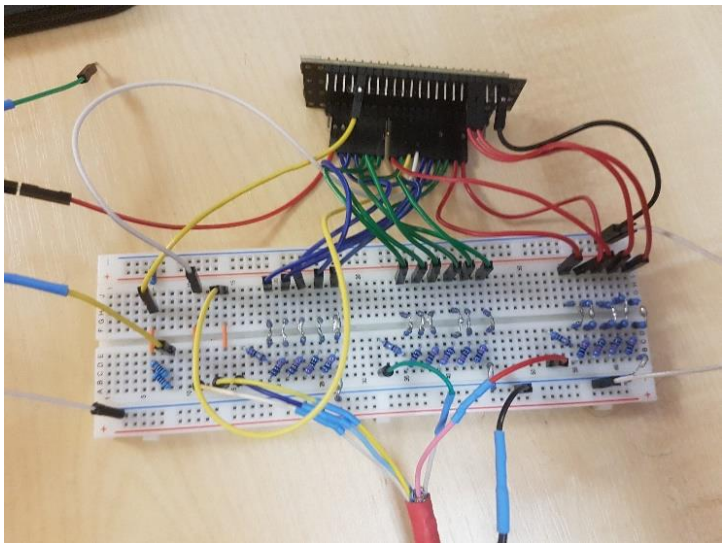
Zaradi omejitve strojne opreme imamo lahko največ 16 podatkovnih izhodov, ki jih moramo razporediti med 3 barve: rdeča, zelena in modra. Te porazdelimo v razmerju 5:6:5, saj je človeško oko najobčutljivejše na zeleno svetlobo. Na koncu vezja moramo dodati 82 ohmski upor, saj VGA signal zahteva napetost med 0 V in 0,7 V. Sestavljeni DAC-i so razvidni na sliki 9. LCD_CAM torej samo še nastavimo, da ustreza našim potrebam in že lahko generiramo VGA signal. Primer rezultata je razviden na sliki 8.

Slika 8: Primer VGA slike

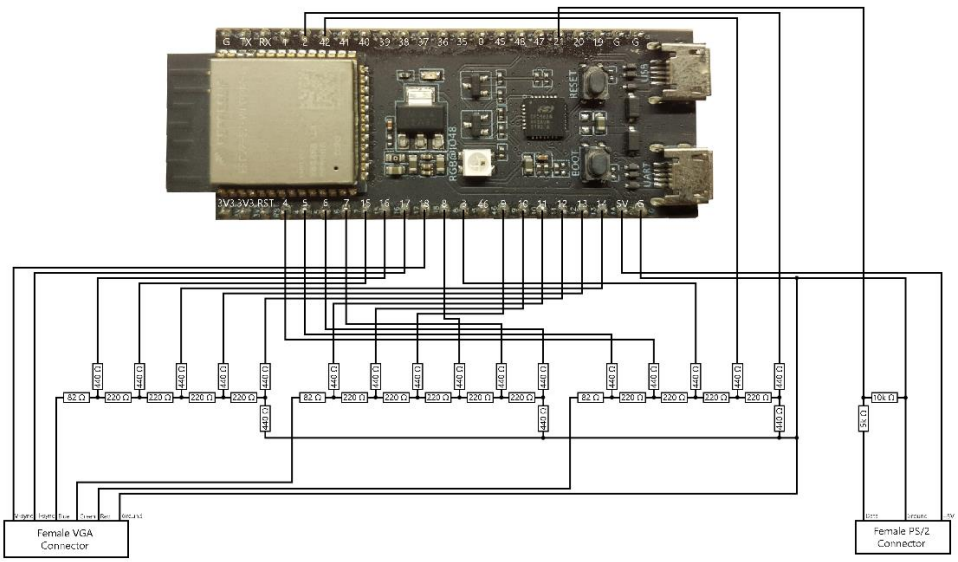
A le generiranje VGA signala ne zadostuje, namreč LCD_CAM v privzetem načinu potrebuje medpomnilnik, katerega velikost je odvisna od podane ločljivosti. Želimo ločljivost 640x400, a zaradi časovnih in spominskih omejitev ne bomo dejansko risali 640x400 slike, ampak le 240x150, kar bomo nato raztegnili na 640x400 [3]. A če napravi podamo ločljivost 640x400, bomo potrebovali 640×400×2=512 KB bajtov, kar znese skoraj ves notranji pomnilnik. To nam seveda ne ustreza, saj ga bomo potrebovali še za drugi zaslonski medpomnilnik, na katerega bomo risali novo sliko, med tem ko se prejšnja pošilja ekranu, in t. i. globinski medpomnilnik, katerega bomo spoznali kasneje. Količina spomina, ki ga dejansko potrebujemo, znaša dva zaslonska medpomnilnika (2×240×150×2) in globinski medpomnilnik (240×150×4), kar je 288 KB. Najti moramo torej način, kako sproti v živo raztezati sliko.

To lahko naredimo z drugim delovnim načinom LCD_CAM-a, ki se imenuje ang. "bounce buffer only." V privzetem načinu se pričakuje, da se celoten zaslonski medpomnilnik naenkrat napolni s podatki, a v tem načinu se ga polni le po potrebi. Naprava za delovanje potrebuje dva majhna medpomnilnika, katerih velikost lahko določimo sami (izbral sem 640, saj je to velikost ene vrstice). Ko pride do konca enega medpomnilnika, sproži prekinitev, ki nam pove, da je zmanjkalo podatkov in da moramo pripraviti nove. Takrat lahko tudi raztegnemo sliko. Zaslonska medpomnilnika si lahko torej sami rezerviramo pri želeni velikosti.

Sedaj imamo enostaven način prikaza podatkov na ekranu, a zaradi rahle netočnosti osnove za ESP timer PLL_F160M, je slika nekoliko vijugasta. To se najbrž dá popraviti z izbiro in nastavitvijo druge osnove, a to je izven obsega te raziskovalne naloge. Na sliki 9 je razviden končni izdelek, na sliki 10 pa njegova shema.



Slika 9: Končni izdelek



Slika 10: Shema končanega izdelka

6 Grafični pogon

En od izzivov te naloge je izdelava lastnega grafičnega pogona.

6.1 Uvod

Preden lahko začnemo s pisanjem igre, je treba imeti način prikazovanja sveta, v katerem se igra odvija. Vsi bodoči elementi igre so v osnovi zgrajeni iz mnogokotnikov, torej potrebujemo način, kako te mnogokotnike prikazati na ekranu. Prikazovali jih bomo preko kasneje opisanega večstopenjskega procesa.

6.1.1 Zahteve

Končne zahteve so sledeče:

- Prikaz glede na trenutni položaj igralca
- Perspektivna projekcija
- Pravilni sistem zakrivanja
- Teksture normalne
- Paralelizacija in splošno hitro delovanje

6.1.2 Medpomnilniki

Za pomoč pri risanju bomo uporabili tudi tri medpomnilnike:

- Zaslonski (72 KB) – vsebuje sliko, ki se jo trenutno prikazuje na zaslonu
- Delovni (72 KB) – nanj se riše sliko, ki bo naslednja prikazana na zaslonu
- Globinski (144 KB) – v pomoč pri risanju slike

6.1.3 Mnogokotniki

Ker bo cel svet sestavljen iz njih, moramo najprej dobro definirati vse podatke, ki jih bo vseboval mnogokotnik. Ti bodo:

- Množica točk, ki ga sestavljajo (poleg položaja v prostoru tudi dve številki, ki ju bomo potrebovali pri risanju tekstur)
- Tekstura
- Dodatni podatki (npr. kakšni vrsti elementa igre pripada)

Za vsak mnogokotnik, ki ga želimo narisati, moramo izvesti postopek, ki je opisan v nadaljevanju.

6.2 Zamik igralca

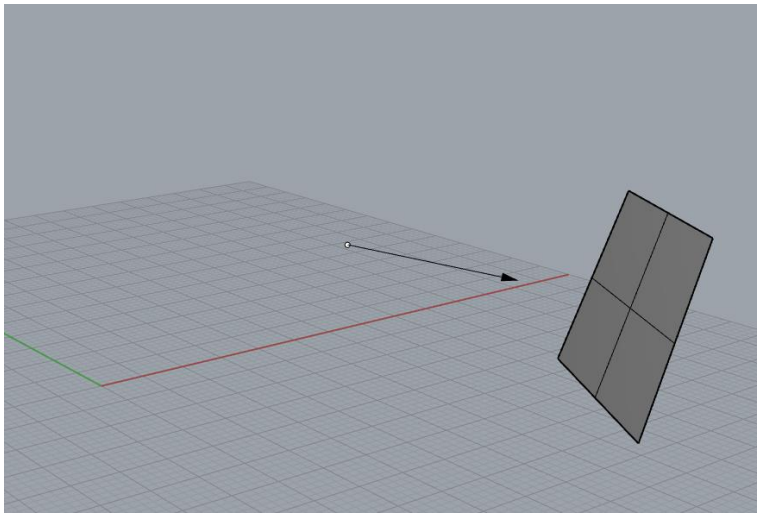
Ker želimo, da se igralec lahko premika in razgleduje naokrog po svetu, moramo najprej mnogokotnik, ki ga želimo narisati, zamakniti za trenutni položaj in rotacijo igralca [9]. Krajevni zamik je enostaven, vsaki točki mnogokotnika odštejemo pozicijo igralca, rotacijski zamik pa je težji. Ker želimo sposobnost gledati navpično in vodoravno, moramo mnogokotnik

zavrteti okoli dveh osi: z in y (vrstni red je pomemben). V ta namen uporabimo naslednji dve rotacijski matriki:

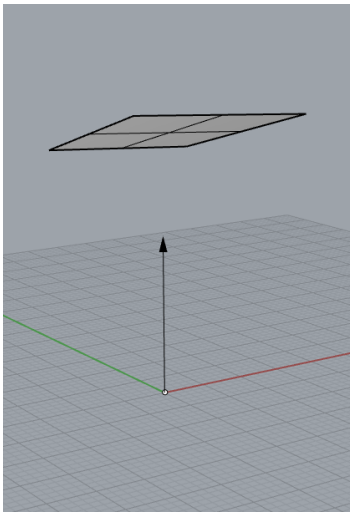
$$R_z = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\theta & -sin\theta \\ 0 & sin\theta & cos\theta \end{bmatrix}$$

$$R_y = \begin{bmatrix} cos\theta & 0 & sin\theta \\ 0 & 1 & 0 \\ -sin\theta & 0 & cos\theta \end{bmatrix}$$

Ko smo mnogokotnik pravilno zamaknili, lahko ravnamo, kot da se igralec nahaja v središču koordinatnega sistema in da gleda naravnost gor. Primer tega postopka je razviden na slikah 11a in 11b.



Slika 11a: Mnogokotnik pred zamikom



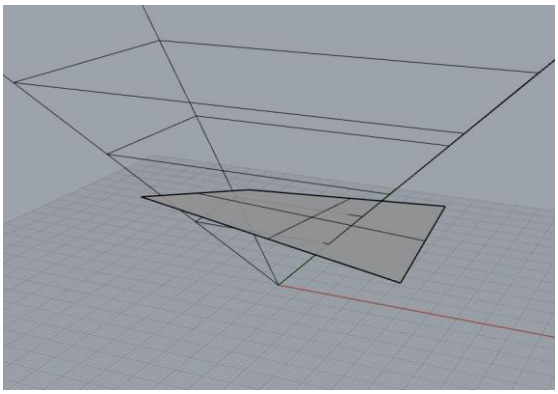
Slika 11b: Mnogokotnik po zamiku

6.3 Grafično obrezovanje

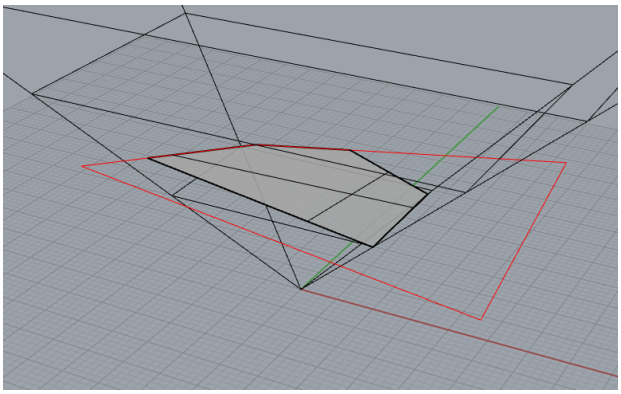
Sedaj moramo izrezati vse dele mnogokotnika, ki v končnem prikazu niso vidni.

6.3.1 Splošni opis

Za namen te raziskovalne naloge predpostavimo, da človekov pogled obsega neskončno visoko pravokotno piramido, ki ima glavno simetrijsko telesno os orientirano v smeri, v katero oseba trenutno gleda. Povsem želimo izločiti vse mnogokotnike, ki se ne nahajajo v tej piramidi, in primerno predelati tiste, ki se delno v njej nahajajo tako, da izrežemo vse dele, ki se v piramidi ne nahajajo. Prikaz slednjega je razviden na slikah 12a in 12b. Ta postopek se imenuje grafično obrezovanje ali po angleško "clipping" in ima dva glavna učinka: odstrani mnogokotnike, ki se nahajajo za igralcem in bi bili brez tega postopka vidni, in zagotovi, da ob koncu celotnega postopka ne bomo po nesreči risali izven ekrana [9].



Slika 12a: Mnogokotnik pred…



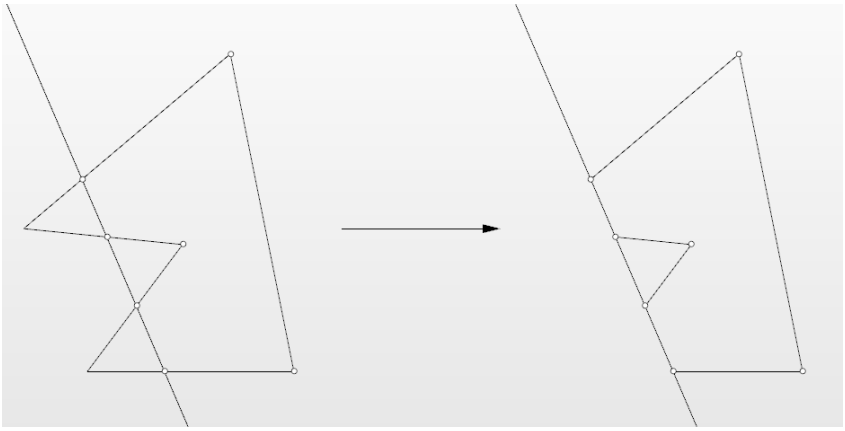
Slika 12b: …in po grafičnem obrezovanju

6.3.2 Postopek

Ideja je sledeča: Hipotetična piramida je sestavljena iz štirih stranic (osnovne ploskve ni treba upoštevati, saj je neskončno daleč). Za vsako stranico moramo izvesti operacijo, ki pravilno obreže mnogokotnike.

Operacija rezanja poteka takole: Vnaprej naredimo nov mnogokotnik, ki je trenutno brez točk. Za vsako točko prvotnega mnogokotnika izračunamo, na kateri strani ravnine – nosilke trenutne stranice – je. To naredimo tako, da izračunamo koordinato *z* točke, ki leži na ravnini in ima enaki koordinati *x* in *y* kot prvotna točka. Če je dobljeno število manjše od koordinate *z* prvotne točke, je točka na "notranji" strani ravnine. Nato zaporedno pregledamo točke, ki sestavljajo mnogokotnik. Če je trenutna točka na notranji strani ravnine, jo dodamo novemu mnogokotniku. Če trenutna in naslednja točka v zaporedju nista na isti strani ravnine, novemu mnogokotniku dodamo še presečišče ravnine in premice, ki jo ti dve točki določata. Tako ostane le del mnogokotnika, ki je na notranji strani ravnine. Dvodimenzionalen prikaz tega postopka je razviden na sliki 13.

Z dobljenim mnogokotnikom ponovimo ta postopek za ostale tri stranice. Ob koncu procesa ostane mnogokotnik, ki je presek piramide vidnega polja in prvotnega mnogokotnika.



Slika 13: Dvodimenzionalen prikaz postopka grafičnega obrezovanja

6.4 Perspektivna projekcija

V resničnem življenju se navidezna velikost predmeta manjša z razdaljo, ker bolj oddaljeni predmeti zavzamejo manjši delež našega vidnega polja. Če želimo, da umetni svet izgleda pravilno, mora prikaz to upoštevati. Ker je relacija med oddaljenostjo predmeta in njegovo velikostjo linearna, lahko za želeni učinek enostavno koordinati *x* in *y* vsake točke mnogokotnika delimo z njeno koordinato *z*, pomnoženo s tangensom polovičnega kota vidnega polja (za ta projekt sem izbral 120°) [9]. Tako je mnogokotnik skoraj že pripravljen za risanje na ekran, a treba je še upoštevati, da sta zaradi perspektivne preslikave *x* in *y* koordinati vseh točk v intervalu (-1, 1). Ta interval želimo spremeniti v [0, vodoravna ločljivost ekrana) za koordinato *x* in [0, navpična ločljivost ekrana) za *y*. V ta namen koordinati *x* prištejemo 1 in jo pomnožimo s polovico vodoravne ločljivosti. Podobno naredimo za koordinato *y*.

6.5 Pretvorba v množico točk na ekranu

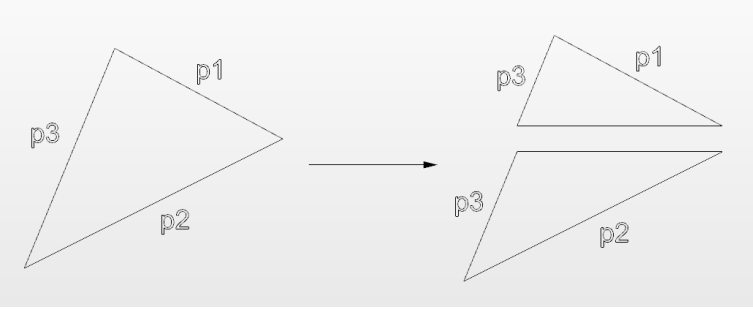
Mnogokotnik zdaj želimo narisati na ekran.

6.5.1 Splošni opis

V ta namen ga bomo najprej razdelili na trikotnike in jih narisali posamezno. Določiti moramo, katere točke na ekranu pripadajo danemu trikotniku in jih nato pravilno obarvati. To naredimo tako, da za vsako vrstico ekrana, ki leži na intervalu [najmanjša *y* koordinata trikotnika, največja *y* koordinata trikotnika], preračunamo vsa presečišča med stranicami trikotnika in obarvamo vse piksle med tema dvema točkama. Ta proces se po angleško imenuje "scan conversion."

6.5.2 Postopek

Pred začetkom risanja dani trikotnik razbijemo na dva manjša trikotnika tako, da ima en spodnjo in drugi zgornjo stranico vzporedno z abscisno (*x*) osjo, in ju nato ločeno narišemo. To naredimo zato, da se ukvarjamo le z dvema stranicama naenkrat. Preračunamo tudi enačbe vseh premic, ki so nosilke stranic prvotnega trikotnika. Te premice bodo odslej naprej označene, kot je razvidno na sliki 14. Med risanjem zgornjega trikotnika za vsako vrstico preračunamo *x* koordinati točk na premicah p1 in p3 v tisti vrstici. Med risanjem spodnjega pa p1 nadomestimo s p2. Tako na vsaki vrstici dobimo interval pikslov, ki jih je potrebno obarvati. S tem, kako se jih obarva, se bo ukvarjalo naslednje podpoglavje.



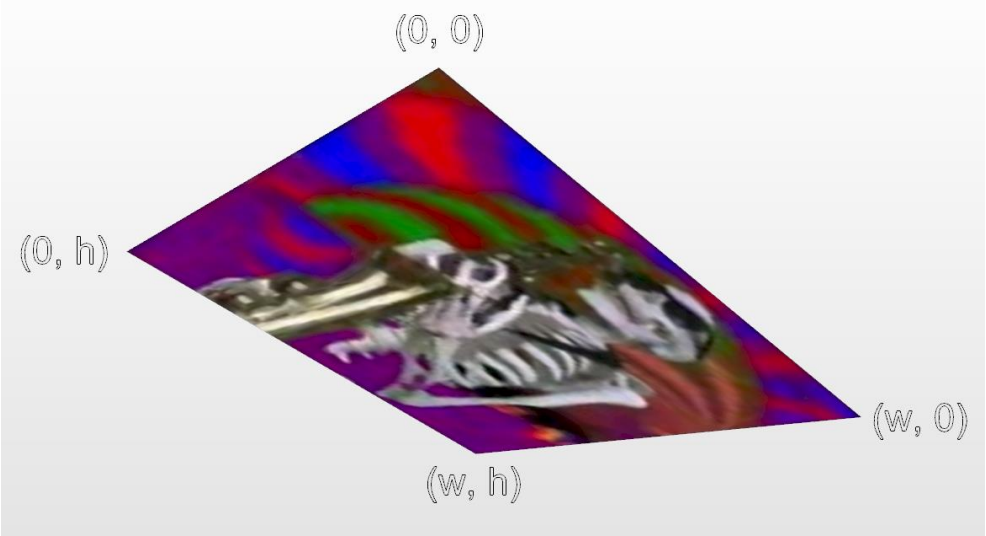
Slika 14: Prikaz razdelitve trikotnika

6.6 Teksture

Ker ne želimo enobarvnih mnogokotnikov, moramo dinamično izbirati barve, s katerimi obarvamo piksle.

6.6.1 Teksturne koordinate

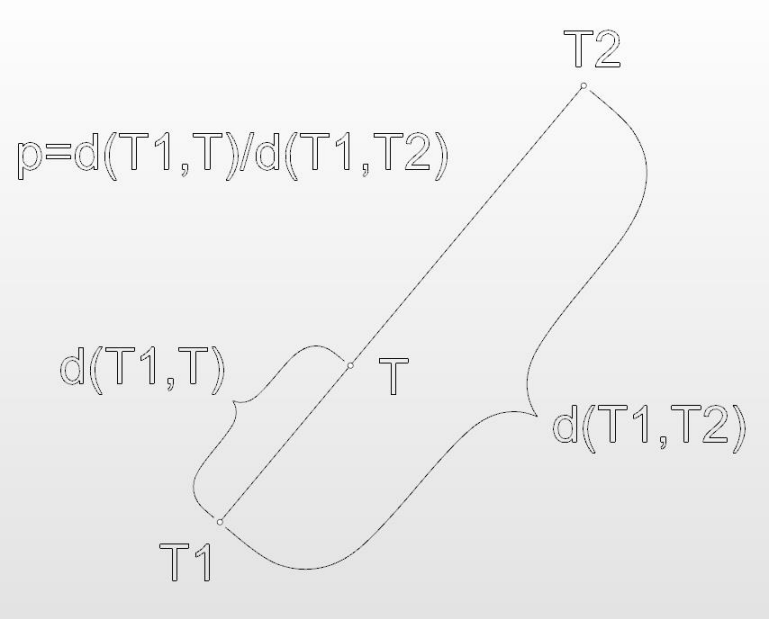
Če predpostavimo, da imamo sliko dimenzij *w* in *h*, lahko vsakemu oglišču mnogokotnika pripišemo, kateri točki na sliki naj bi pripadala (v enotah pikslov). Te številke se imenujejo teksturne koordinate [9]. Primer njihovega zapisa je razviden na sliki 15. Potrebujemo tak par teksturnih koordinat za vsako točko na mnogokotniku, ki jo želimo narisati. Vmesne vrednosti v teh ključnih točkah določimo z linearno interpolacijo.



Slika 15: Primer zapisa teksturnih koordinat

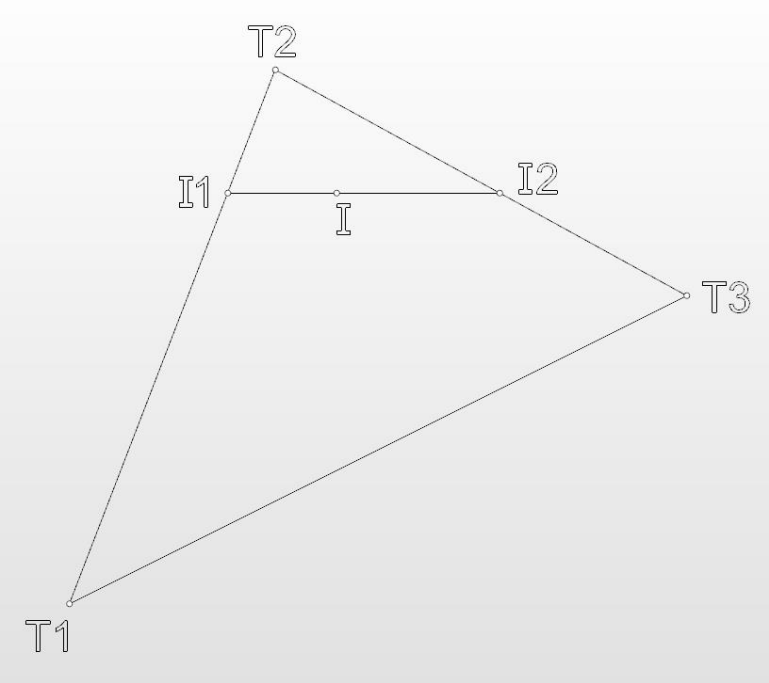
6.6.2 Postopek

Predpostavimo, da je za začetek in konec črte T1 in T2 podan položaj v prostoru in teksturne koordinate. Prikaz tega je razviden na sliki 16. Vrednost teksturnih koordinat katerekoli točke T na tej črti lahko dobimo s sledečim postopkom: Preračunamo razdaljo med točkama T in T1 ter jo ulomimo s razdaljo med točkama T1 in T2 (to lahko naredimo tudi s samo eno koordinato točk, saj vse ležijo na isti črti). To nam dá nekakšno mero tega, koliko blizu je T točkama T1 in T2. To vrednost imenujemo p. Če je p=0, je T=T1, in če je 1, je T=T2. Nato za vsako koordinato teksturnih koordinat izračunamo uteženo povprečje, kjer je utež prve koordinate (tiste, ki pripada točki T1) enaka p, utež druge pa 1-p (Ker je vsota uteži vedno 1, nam ni treba z njo na koncu deliti).



Slika 16: Prikaz črte s teksturnimi koordinatami

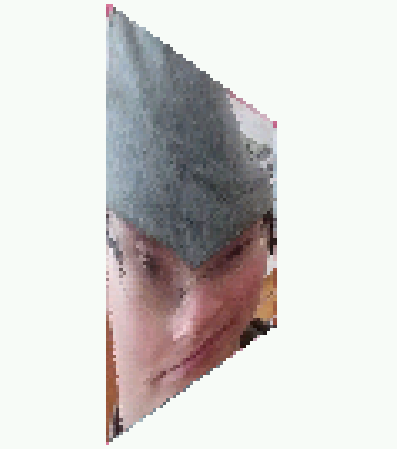
To nam omogoči dobiti teksturne koordinate katerekoli točke na črti, a mi želimo to uporabiti na ravnini. To lahko dosežemo tako, da enostavno večkrat interpoliramo. Ko računamo presek vodoravne črte in stranice trikotnika, kot je bilo opisano v prejšnjem podpoglavju, tudi izračunamo teksturne koordinate presečišč po naši novi metodi. Nato za vsak piksel med tema presečiščema še interpoliramo ti dve novi teksturni koordinati. Prikaz tega je razviden na sliki 17.



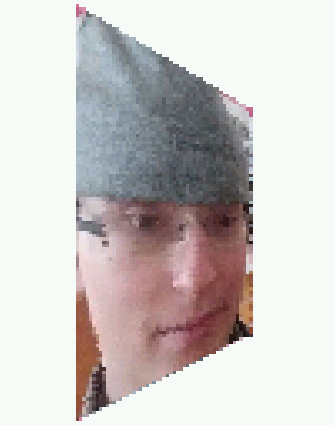
Slika 17: Prikaz interpolacije v dveh dimenzijah

6.6.3 Napaka zaradi projekcije

A ko poskusimo prej opisani proces implementirati, je slika popačena, ker smo pozabili, da smo v prejšnjem postopku perspektivne projekcije transformirali naš koordinatni sistem tako, da smo vsako točko pomnožili s faktorjem 1/z. Če želimo pravilno interpolacijo, moramo isto transformacijo izvesti tudi na teksturnih koordinatah. V praksi to pomeni, da jih pred začetkom risanja prav tako pomnožimo s tem faktorjem in nato med postopkom sproti interpolirati tudi ta faktor. Čisto na koncu, ko želimo iz teksturnih koordinat dobiti točko na sliki, interpolirane teksturne koordinate delimo še z interpolirano vrednostjo 1/z [10]. Primerjava stare in nove implementacije je razvidna na slikah 18a in 18b. Vidimo, da to popravi popačenost, a na žalost tudi zelo upočasni postopek, saj je deljenje necelih števil na računalnikih precej počasno, mi pa ga moramo izvesti za vsak piksel. Obstaja druga vrsta mnogokotnika, kjer je to računanje mnogo hitrejše. To bomo obravnavali v naslednjem podpoglavju.



Slika 18a: Prikaz popačenosti



Slika 18b: Prikaz nepopačenosti

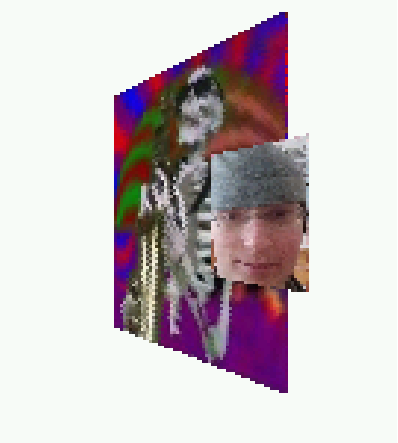
6.7 Zakrivanje

Do zdaj smo metodo preizkušali le z enim mnogokotnikom. Ko skušamo dodati še enega, se pojavi problem. Trenutno namreč nimamo načina presoditi, kateri mnogokotnik zakriva drugega, če je eden pred drugim. Prikaz tega je razviden na slikah 19a in 19b.

6.7.1 Globinski pomnilnik

Iščemo postopek, ki bi lahko to presojal. V ta namen uporabimo t. i. globinski medpomnilnik [9]. Ideja je sledeča: imamo medpomnilnik, ki je namenjen temu, da ima za vsako točko ekrana shranjeno njeno oddaljenost od igralca. Vsakič, ko želimo prepisati vrednost nekega piksla, najprej izračunamo, koliko je oddaljena točka na mnogokotniku, ki predstavlja ta piksel. Če je dobljena vrednost manjša od vrednosti, ki je za tisti piksel trenutno zapisana v globinskem medpomnilniku, barvo prepišemo in v globinski medpomnilnik zapišemo novo razdaljo.

Slika 19a: Prikaz mnogokotnikov z ene strani



Slika 19b: Prikaz mnogokotnikov z druge strani

6.7.2 Postopek

Samo razdaljo preračunamo tako, da ob začetku risanja izračunamo enačbo ravnine, kateri pripada mnogokotnik, preko uporabe vektorskega produkta. Nato enačbo preuredimo, da izrazimo z koordinato. Ostane linearna funkcija, kar omogoča hitro računanje.

6.7.3 Prednosti in slabosti

Obstajajo tudi drugi postopki, s katerimi se dá določiti prekrivanje, a sem izbral tega iz dveh glavnih razlogov: v vseh primerih deluje pravilno in omogoča paralelno računanje. Slabost je, da je v zameno za točnost v primerjavi z drugimi podobnimi postopki nekoliko počasen in da je treba ob začetku risanja tudi na novo popisati celoten globinski medpomnilnik. Poleg tega sam globinski medpomnilnik zavzame veliko spomina. Problem počasnosti nekoliko ublaži zelo visoka stopnja paralelizacije.

6.8 Uporaba več jeder

ESP32-S3 N8R8 nima le enega jedra, ampak kar dve, torej če napišemo program tako, da se izvaja več procesov naenkrat, mu lahko skoraj podvojimo hitrost delovanja. To naredimo tako, da enemu jedru dodelimo eno polovico mnogokotnikov, drugemu pa drugo. Ta metoda ne upošteva tega, da lahko eno jedro prej konča s risanjem in mora nato čakati drugega, a zaradi števila mnogokotnikov in njihove približno enake velikosti je kljub temu zadovoljiva.

To porazdelitev dela med procesorjema sem uporabil samo za postopek risanja, ker zavzame daleč največ programskega časa. Najbrž je možno kaj podobnega izvesti tudi pri drugih delih računanja igre (prvi, ki pride na pamet je zaznavanje trkov, ki je predstavljen v naslednjem (7.) poglavju), a tega nisem še izvedel.

7 Trki

Poleg grafičnega pogona je potrebno ustvariti sistem za zaznavanje in obravnavanje trkov.

7.1 Uvod

Zaradi omejenosti računalnikov v času izida Doom-a so bile njegove metode za zaznavanje trkov zelo enostavne. Trke se namreč obravnava predvsem v dveh dimenzijah. Tudi višina pride v poštev, a bolj kot naknadna misel. Igra je z vidika trkov sestavljena le iz kvadrov in črt. Za popolno zaznavo vseh trkov potrebujemo torej le tri postopke, ki določijo sledeče: ali se sekata dva kvadra, ali se sekata dve črti in ali se sekata črta in kvader. Te metode bodo podrobno opisane v 3. podpoglavju.

A zaznavanje trkov je le polovica problema. Treba jih je tudi pravilno obravnavati, ko jih zaznamo. V večini primerov naslednja akcija nima več zveze s trki (npr. če raketa zazna, da je zadela steno ali kaj drugega, naj se razstreli). V dveh primerih pa to ne velja: ko se bitje zadane v steno in ko se zadane v drugo bitje. Ta primera si bomo pogledali v 4. podpoglavju.

7.2 Kategorizacija objektov in predpostavke

V prvotnem Doom-u je bilo vso neživo okolje* razdeljeno na stene in tla/strop. Ta razdelitev je bila takrat tudi v pomoč pri optimiziranju risanja, a ker želimo ohraniti sposobnost gledati gor in dol, je relevantna le pri računanju trkov. Ključna porazdelitev je v tem, da so tla vedno vzporedna z ravnino, ki jo določita abscisna in ordinatna os, stene pa so glede na tla vedno navpične. Zato lahko veliko večino trkov obravnavamo zgolj v dveh dimenzijah, kar vse zelo poenostavi in postopoma tudi zelo pospeši.

Vsakemu elementu živega okolja, ki jim bomo od tod naprej rekli bitja, je bil predpisan kvader, ki naj bi približno predstavljal to bitje. Ključno je, da so stranice teh kvadrov vzporedne z osmi koordinatnega sistema.

Za vse črte se tudi predpostavi, da je z komponenta obeh točk, ki jo določata, enaka, torej da je črta vedno pri konstantni višini.

**Nekatere elemente neživega okolja se obravnava kot bitja.*

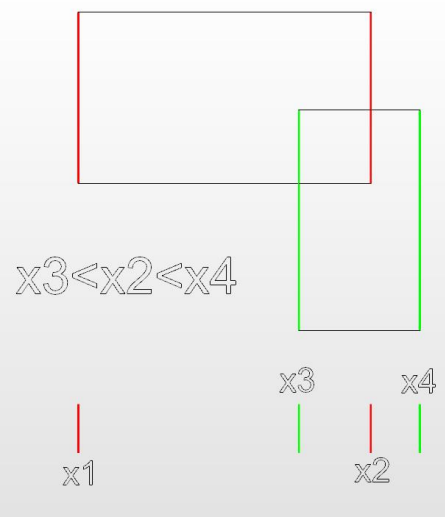
7.3 Načini zaznavanja trkov

V tem podpoglavju si bomo ogledali metode za določanje preseka dveh elementov igre.

7.3.1 Presek dveh kvadrov

Splošna različica tega problema je nekoliko bolj zapletena, a v tem primeru imamo luksuz prejšnje predpostavke, ki omogoči, da problem razbijemo na manjše kose. Namreč če to predpostavimo, moramo le za vsako dimenzijo posamezno pogledati, če se kvadra sekata in nato združiti rezultate. Če se kvadra v vseh treh dimenzijah neodvisno sekata, se tudi dejansko

sekata. Za vsako dimenzijo posamezno določimo, ali se sekata tako, da za obe stranici enega kvadra pogledamo, če je povsem vsebovana v drugem kvadru. Če to velja za vsaj eno stranico, se v tisti dimenziji kvadra sekata. Prikaz tega je razviden na sliki 20.



Slika 20: Prikaz sekanja pravokotnikov v eni dimenziji

7.3.2 Presek dveh črt

Ta vrsta računanja presekov je precej enostavnejša kot prejšnja. Za obe črti (ki sta podani z množico štirih točk) preračunamo funkcijo premice, ki je njena nosilka, in izračunamo njun presek. Pri tem pazimo tudi na posebna primera, da je ena izmed črt navpična in da sta si premici vzporedni.

7.3.3 Presek črte in kvadra

Tu ponovno problem razdelimo na manjše kose. Namreč da določimo, če črta seka kvader, moramo določite le, ali seka katerokoli izmed stranic tega kvadrata in ločeno, ali črta sploh po višini seka kvader. Torej le štirikrat uporabimo že definiran postopek za določanje preseka dveh črt. Možna je tudi optimizacija, kjer se namesto stranic računa z diagonalama osnove kvadra, a tega nisem izvedel zaradi časovne omejitve.

7.4 Obravnavanje trkov

Poglejmo si zdaj metode za obravnavanje trka med dvema elementoma igre.

7.4.1 Trk med bitjem in steno

Ta postopek bitje primerno potisne izven stene, v katero se je zaletelo. Ker naj bi se ta postopek izvedel le, ko je že bilo določeno, da se bitje in stena sekata, lahko tudi tu predpostavimo, da je to res. Razen posebnih primerov, ko je stena glede na tloris vodoravna ali navpična, je postopek sledeč: izračunamo, katera je "notranja" stran stene z istim postopkom, kot smo ga uporabili pri izrezovanju (6.3), razen da tokrat za notranjo stran določimo tisto, na katero ne kaže normalni

vektor stene (ki ga izračunamo preko vektorskega produkta). Nato za vsako izmed oglišč kvadrata, ki je osnova kvadra bitja, izračunamo, ali je na notranji strani stene. Tistim, ki so, izračunamo razdaljo do premice, ki je nosilka stene preko sledeče formule (dokaz v prilogi 1):

$$d = \frac{|y - (kx + n)|}{\sqrt{k^2 + 1}}$$

Izmed vseh izračunanih razdalj izberemo največjo, normalni vektor stene pomnožimo z njo in rezultat prištejemo koordinatam bitja.

7.4.2 Trk med dvema bitjema

Trk med dvema bitjema je na srečo nekoliko bolj enostaven. Določimo, če sta si bitji bolj oddaljeni po x ali y koordinati in izberemo večjo od obeh koordinat. Določimo tudi, v katero smer (pozitivno ali negativno) sta si oddaljena. Nato nastavimo pozicijo enega bitja na pozicijo drugega bitja in dodamo ali odštejemo vsoto polovic njunih debelin.

8 Notranje delovanje igre

Sedaj imamo vse gradnike, ki jih potrebujemo za začetek programskega dela. V tem poglavju si bomo pogledali notranja porazdelitev elementov igre in dejansko delovanje igre.

8.1 Uvod

A najprej je potrebno omeniti, da je zelo pomembna omejitev spomin. ESP32-S3 N8R8 ima zelo omejeno količino notranjega (hitrega) pomnilnika (512 KB). Ima tudi dodaten zunanji pomnilnik (8 MB), a ta je za naše potrebe mnogo prepočasen. Torej moramo varčevati s spominom, kjer se le dá. A ESP32-S3 N8R8 ima poleg pomnilnika tudi programski spomin, ki je podobno hiter, a mnogo večji (8 MB). Njegova slabost je to, da se iz njega lahko le bere. Torej, ko imamo vrsto elementa, ki se skozi delovanje programa ne spreminja, ga bomo postavili v programski spomin. To je razlog za nekatere tehnične odločitve.

8.2 Vpeljava objektov igre

Vsaka igra je sestavljena iz več elementov, ki se med seboj razlikujejo po načinu obravnave.

8.2.1 Površine

Površine so glavni sestavni elementi nežive okolice. Delijo se na stene in tla, ki ju predvsem na nivoju trkov povsem drugače obravnavamo. Ker se površine večinoma ne spreminjajo, jih lahko postavimo v programski spomin. Majhen del površin mora kljub temu biti sposoben gibanja (to so večinoma vrata in podobno), a ker je to tako majhen delež vseh površin, ne zavzamejo veliko spomina in jih lahko torej shranimo v notranjem pomnilniku. To površine razdeli na statične in dinamične. Iz površin bomo torej zgradili svet, v kateremu se bo igra odvijala.

Slika 21: Površina

8.2.2 Bitja

Bitja so najbolj dinamičen element in morajo torej biti vsa shranjena v notranjem pomnilniku. So tudi edini element, ki dinamično reagira na igralca (npr. če ga igralec ustreli, bo bitje poskusilo ustreliti nazaj). Zato so tudi najbolj kompleksen element, saj ne vplivajo le na igralca, ampak tudi drug na drugega.

Slika 22: Bitje

8.2.3 Vizualni efekti

Vizualni efekti so različica bitij, ki so toliko manj dinamični, da se jih izplača odcepiti. Ta razcepitev je tudi potrebna, ker jih je pogosto veliko več kot bitij in bi torej zelo upočasnili delovanje programa. Dinamično se ustvarjajo med delovanjem igre (npr. igralec ustreli v steno, zato se tam pojavi dim), torej se morajo nahajati v notranjem pomnilniku, a ker je obseg njihovega delovanja zelo majhen, zavzamejo malo spomina.

Slika 23: Vizualni efekt

8.2.4 Sprožilci

Sprožilci omogočajo dodatne dogodke v svetu, ki je drugače povsem statičen. Delujejo tako, da zaznajo, ko je izpolnjen nek pogoj (npr. igralec je pritisnil gumb za odpiranje vrat) in, če ga zaznajo, sprožijo nek drug dogodek (npr. odpiranje vrat). Nekaj dogodkov je tako pogostih, da imajo že vnaprej spisane funkcije, a če je željen bolj unikaten dogodek, se lahko za sprožilec naredi tudi namensko funkcijo. Ker so v osnovi tudi del nežive okolice, se nahajajo v programskem spominu. Zaradi njihove narave so sprožilci nevidni.

8.3 Glavna zanka

Igra je v osnovi samo zanka, ki se ponovi večkrat na sekundo in tako ustvari iluzijo neprekinjenosti. V tej zanki je potrebno pridobiti vnose igralca, preračunati vse dogodke v svetu in ga nato tudi narisati. Je pa potrebno upoštevati hitrost, s katero se vse to dogaja. Zavedam se neverjetnosti sledečega stavka, a kljub temu je še vedno resničen: mikrokrmilnik je prehiter. Hitrost odvijanja sveta je namreč neposredno povezana s hitrostjo računanja ESP-ja. Treba je torej omejiti hitrost zanke. Ta problem se obravnava 7. pod-podpoglavju. Velja tudi obratno, a zoper ta problem lahko le igro čim bolje optimiziramo. Sledeča pod-podpoglavja si sledijo v dejanskem vrstnem redu izvajanja v programu.

8.3.1 Podatki s tipkovnice

Podatke s tipkovnice pridobimo preko postopka, ki je opisan v 3. poglavju. Medtem ko program računa druge stvari, strojna naprava za protokol UART spremlja tipkovnico in shrani vsa sporočila, ki jih od nje pridobi. Program lahko torej ta sporočila pogleda, ko je to najbolj ugodno, na začetku nove izvedbe zanke.

8.3.2 Obravnavanje sprožilcev

Sprožilci se morajo sprožiti, ko zaznajo, da je njihov pogoj izpolnjen. V tej fazi preračunamo izpolnjenost teh pogojev in primerno aktiviramo tiste sprožilce, ki imajo pogoje izpolnjene. Poleg tega nadaljujemo z obravnavo tistih, ki so bili sproženi že prej in za svoje delovanje potrebujejo več kot eno izvedbo zanke.

8.3.3 Obravnavanje bitij

Tu se preračunajo trki tako, da to bitje primerjamo z vsemi ostalimi bitji in površinami. Trenutno ta postopek še ni zelo optimiziran, a je v načrtu. Preračuna se tudi splošno obnašanje bitij. V praksi to pomeni, da za vsako bitje pokličemo funkcijo, ki se odloči o njegovem naslednjem dejanju. Ta funkcija je za vsako vrsto bitja drugačna. Tukaj tudi obdelamo vnose, saj je tudi igralec bitje.

8.3.4 Risanje sveta

Ta postopek smo izčrpno obravnavali že v 6. poglavju. Vredno je omeniti le, da je to daleč strojno najzahtevnejši postopek v zanki. Zato je bil največji del optimizacije posvečen ravno risanju sveta in zato je trenutno tudi edini del, ki uporablja drugo jedro.

8.3.5 Risanje pomožnih elementov

Poleg trenutnega dogajanja v svetu želimo igralcu posredovati tudi pomožne podatke o njegovem stanju (npr. koliko je zdrav ali koliko nabojev še ima). To naredimo v odseku zaslona, ki je zaradi stilističnih odločitev prvotnih izdelovalcev Doom-a namenjen prav temu. To nam omogoči še eno optimizacijo, saj tega, kar je skrito pod tem odsekom, ni treba risati.

8.3.6 Čakanje

Zaradi prej omenjenega problema celotna zanka ne sme delovati prehitro. Tu enostavno čakamo, dokler ne izmerimo, da je od začetka izvajanja zanke pretekla ena tridesetinka sekunde. To naredimo s pomočjo vgrajene štoparice, ki smo jo vsakič na začetku zanke nastavili, da začne šteti čas. To tudi omogoča preračun, koliko časa je trajalo, da je program enkrat izvedel zanko, kar dá merilo trenutne zahtevnosti računanja igre.

8.3.7 Menjava zaslonskega in delovnega medpomnilnika

Na koncu vsake izvedbe zanke je treba to, kar smo narisali, prikazati na ekranu. Naiven način bi bil, da vsebino delovnega medpomnilnika prekopiramo v zaslonski medpomnilnik, a to je nepotrebno počasno, saj jima lahko enostavno zamenjamo vlogi. Novega delovnega medpomnilnika nam tudi ni treba brisati, saj je igra narejena na tak način, da se celoten zaslon tako ali tako vsakič preriše.

8.4 Pridobitev slik igre

To podpoglavje ni povezano z delovanjem same igre, a se mi je zdelo potrebno obrazložiti, kako sem pridobil zaslonske posnetke igre, če pa igra v resnici teče na ESP-ju in na emulatorju ali podobnem tehničnem pripomočku.

Ob pritisku določene tipke program igro začasno prekine in trenutni zaslonski medpomnilnik preko UART protokola pošlje računalniku, na katerega je priključen ESP. Tam čaka pomožni Python program, ki podatke sprejme in jih nato sestavi v sliko. Ta program je podan v prilogi 2. Na tak način so zajete slike 24, 25 in 26.

Slika 24: Prizor iz igre

Slika 25: Prizor iz igre

Slika 26: Prizor *iz igre*

9 Zaključek

Končni izdelek torej omogoča, da mikrokrmilnik izvaja naslednje:

- Generiranje VGA izhoda
- Branje tipkovnice preko PS/2
- Osnovna računalniška grafika v treh dimenzijah
- Računanje in obravnavanje trkov
- Vse našteto združi v koherentno igro

Nekaj ključnih lastnosti še manjka, kot na primer:

- Zvok
- Nivoji prvotne igre Doom
- Optimizacija risanja na nivoju sob
- Splošna optimizacija trkov

Končni izdelek je po mojem mnenju že precej podoben Doom-u, a manjka še nekaj ključnih lastnosti prvotne igre. Projekt nameravam torej še nadgraditi z vsem naštetim. Zvok bo poganjala strojna naprava za protokol I2S, originalne nivoje igre pa bom pridobil tako, da spišem Python skripto, ki bo pretvorila datoteke, ki vsebujejo podatke za nivoje prvotnega Doom-a, v moj format shrambe. To bo tudi omogočilo uporabo že obstoječih orodij za izdelavo lastnih Doom nivojev.

Kljub tem pomanjkljivostim smo vsekakor potrdili obe hipotezi:

- Mikrokrmilnik ESP32-S3 N8R8 res poganja Doom pri primerni hitrosti.
- Ob tem generira tudi VGA sliko.

Koda končnega izdelka je dosegljiva preko spletne povezave
https://github.com/MushySushy/Doom_na_mikrokrmilniku_ESP32 ali about.sushy.net.

Letošnjega decembra bo Doom star 30 let. V tem času smo doživeli tak tehnološki napredek, da igra, ki je pred daljnimi leti potrebovala popolno pozornost namiznega računalnika, zdaj lahko teče na mikrokrmilniku, ki ga držiš v dlani in stane manj kot 10 evrov.

Viri in literatura

[1] (2023). Doom (1993 video game), pridobljeno 28. 02. 2023, s https://en.wikipedia.org/wiki/Doom_(1993_video_game)

[2] Javier Valcarce. (2007). VGA Video Signal Format and Timing Specifications, pridobljeno 28. 02. 2023, s http://javiervalcarce.eu/html/vga-signal-format-timming-specs-en.html

[3] (2008). VGA Signal 640 x 400 @ 70 Hz timing, pridobljeno 28. 02. 2023, s http://tinyvga.com/vga-timing/640x400@70Hz

[4] Adam Chapweske. (1999). PS/2 Mouse/Keyboard Protocol, pridobljeno 28. 02. 2023, s http://www.burtonsys.com/ps2_chapweske.htm

[5] (?). PS/2 PC Keyboard Scan Sets Translation, Table, pridobljeno 28. 02. 2023, s http://www.vetra.com/scancodes.html

[6] (2022). ESP32-S3 Technical Reference Manual, pridobljeno 28. 02. 2023, s https://www.espressif.com/sites/default/files/documentation/esp32-s3_technical_reference_manual_en.pdf

[7] (2022). ESP32-S3 API Reference, pridobljeno 28. 02. 2023, s https://docs.espressif.com/projects/esp-idf/en/latest/esp32s3/api-reference/index.html

[8] Alan Wolke. (2015). Digital to Analog Conversion – The R-2R DAC, pridobljeno 28. 02. 2023, s https://www.tek.com/en/blog/tutorial-digital-analog-conversion-r-2r-dac

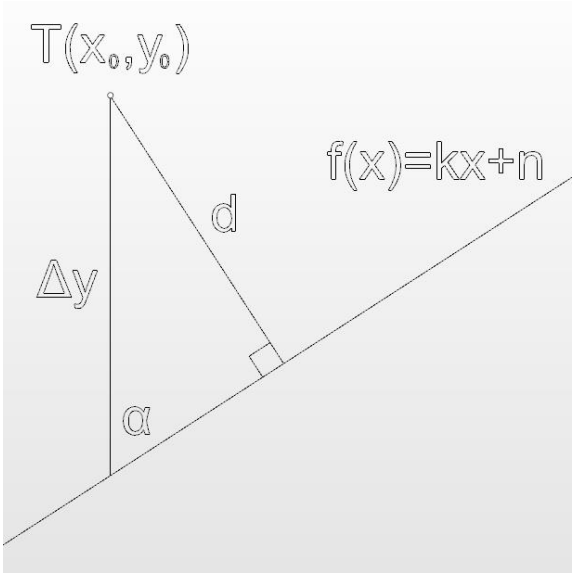
[9] Ken Joy. (2009). video serija Computer Graphics (posnetki 3, 5, 6, 7 in 10), pridobljeno 28. 2. 2023, s https://www.youtube.com/playlist?list=PL_w_qWAQZtAZhtzPI5pkAtcUVgmzdAP8g

[10] (2023). Texture mapping: Perspective correctness, pridobljeno 28. 02. 2023, s https://en.wikipedia.org/wiki/Texture_mapping#Perspective_correctness

Priloge

Priloga 1: Dokaz formule za razdaljo med točko in premico

Predpostavimo, da imamo točko $T(x_0, y_0)$ in premico $f(x)=kx+n$. Narišemo pravokotni trikotnik, kot je razvidno na sliki 27 in ga tudi primerno označimo.



Slika 27: Prikaz trikotnika, ki ga tvorita točka in premica

Vpeljemo sledeči formuli:

$$\Delta y = |y_0 - f(x_0)|$$

$$\alpha = \frac{\pi}{2} - \arctan k$$

Nato lahko zaradi lastnosti pravokotnih trikotnikov vpeljemo in predelamo sledečo formulo:

$$\sin \alpha = \frac{d}{\Delta y}$$

$$d = \Delta y \sin \alpha \ *$$

$$d = \Delta y \cos \arctan k \ **$$

$$d = \frac{\Delta y}{\sqrt{\tan^2 \arctan k + 1}}$$

$$d = \frac{|y_0 - (kx_0 + n)|}{\sqrt{k^2 + 1}}$$

* $\sin\left(\frac{\pi}{2} - x\right) = \cos x$

** $\tan^2 x + 1 = \frac{\sin^2 x + \cos^2 x}{\cos^2 x} = \frac{1}{\cos^2 x}$ $\qquad$ $\cos x = \frac{1}{\sqrt{\tan^2 x + 1}}$

Priloga 2: Python 3.7 program za sprejemanje slik

```
import serial, os
from PIL import Image

w=240
h=150
wn=640
hn=400
ser=serial.Serial(port="COM25",baudrate=115200*20,parity=serial.PARITY_NONE,stopbits
=serial.STOPBITS_ONE,bytesize=serial.EIGHTBITS,timeout=1)
print("ready")
s=""

while True:
	while s!=b"OwO":	#handshake
		s=ser.read(3)
	s=ser.read(w*h*2)
	img=Image.new("RGB",(w,h))
	img.putdata([((s[i]&0b11111)*255//32,
((((s[i+1]<<8)+s[i])>>5)&0b111111)*255//64, (s[i+1]>>3)*255//32) for i in
range(0,w*h*2,2)])
	#img=img.resize((wn,hn))

n=0
while os.path.isfile(str(n)+".png"):
	n+=1
img.save(str(n)+".png")
print("saved screenshot %d"%n)
```