UNIVERZA V LJUBLJANI

Fakulteta za elektrotehniko

Digitalna integrirana vezja in sistemi
3D grafični pospeševalnik

Projektno poročilo

Gal Nadrag

16. februar 2022

Kazalo

Kazalo slik

Kazalo preglednic

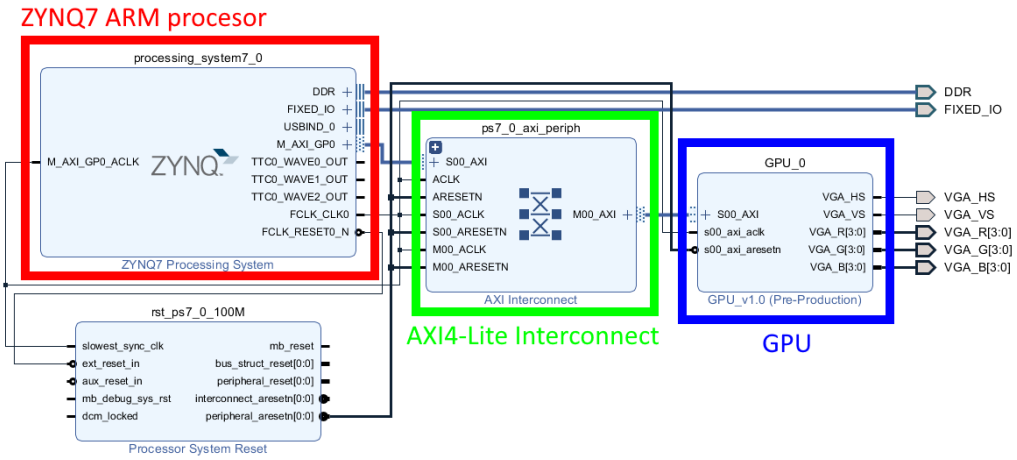
1 Povzetek

V poročilu predstavimo vezje za izris osvetljene krogle na VGA zaslonu. Resolucija zaslona je 800x600@72Hz [1]. Za generacijo slike uporabimo ZedBoard [2] z integriranim vezjem Xilinx Zynq-7000. Vezje vsebuje procesorski podsistem (PS) in programibilno logiko (PL).

1.1 Hierarhija

Celoten sistem je razdeljen na 3 pod-sklope:

- ZYNQ7 ARM procesor
- AXI4-Lite interconnect
- GPU



Slika 1.1: Hierarhija sistema

1.1.1 ZYNQ7 ARM procesor

Za PS poskrbi vgrajen ARM Cortex-A9, ki je povezan z PL prek AXI vodil. V našem sistemu ga uporabimo za generacijo koordinat luči, ki nam osvetljuje kroglo.

1.1.2 AXI4-Lite interconnect

Povezava med PS in PL je izvedena z AXI vodili. Grafični vmesnik uporablja poenostavljeno AXI4-Lite vodilo, za pretvorbo uporabimo kar PL.

1.1.3 GPU

Grafični pospeševalnik(GPU) je popolnoma izveden v PL delu vezja. Kot vhodni podatek sprejme koordinate luči, na izhodu nadzira VGA vmesnik z zaslonom.

2 Poročilo

2.1 Teorija

Za izris osvetljene krogle potrebujemo vedeti barvo vsakega piksla na zaslonu. To nalogo lahko razdelimo na 2 dela.

– Ali je piksel del krogle?
– Kako močno luč osvetljuje kroglo na poziciji tega piksla?

2.1.1 Ali je piksel del krogle?

Ta del izračuna je dokaj enostaven. Projekcija krogle na zaslon je preprosto krog. Da izvemo če smo znotraj kroga uporabimo formulo:

$$x^2 + y^2 < r^2$$

Kjer je r radij kroga v piksih.

2.1.2 Kako močno luč osvetljuje kroglo na poziciji tega piksla?

Osvetljenost krogle lahko razdelimo na več prispevkov:

– Ambientna svetloba: svetloba prisotna povsod, odbita od raznih predmetov v okolju
– Difuzna svetloba: svetloba, ki prihaja iz svetlobnega vira in se na površini razprši in odbije v vse smeri
– Spekularna svetloba: svetloba, ki prihaja iz svetlobnega vira in se na površini odbije pravokotno na vpadnico

To je precej bolj zapleten izračun. V našem primeru uporabimo Phongov odbojnostni model [3].

$$I_\mathrm{p} = k_\mathrm{a} i_\mathrm{a} + \sum_{m \in \text{ lights}} (k_\mathrm{d}(\hat{L}_m \cdot \hat{N}) i_{m,\mathrm{d}} + k_\mathrm{s}(\hat{R}_m \cdot \hat{V})^\alpha i_{m,\mathrm{s}})$$

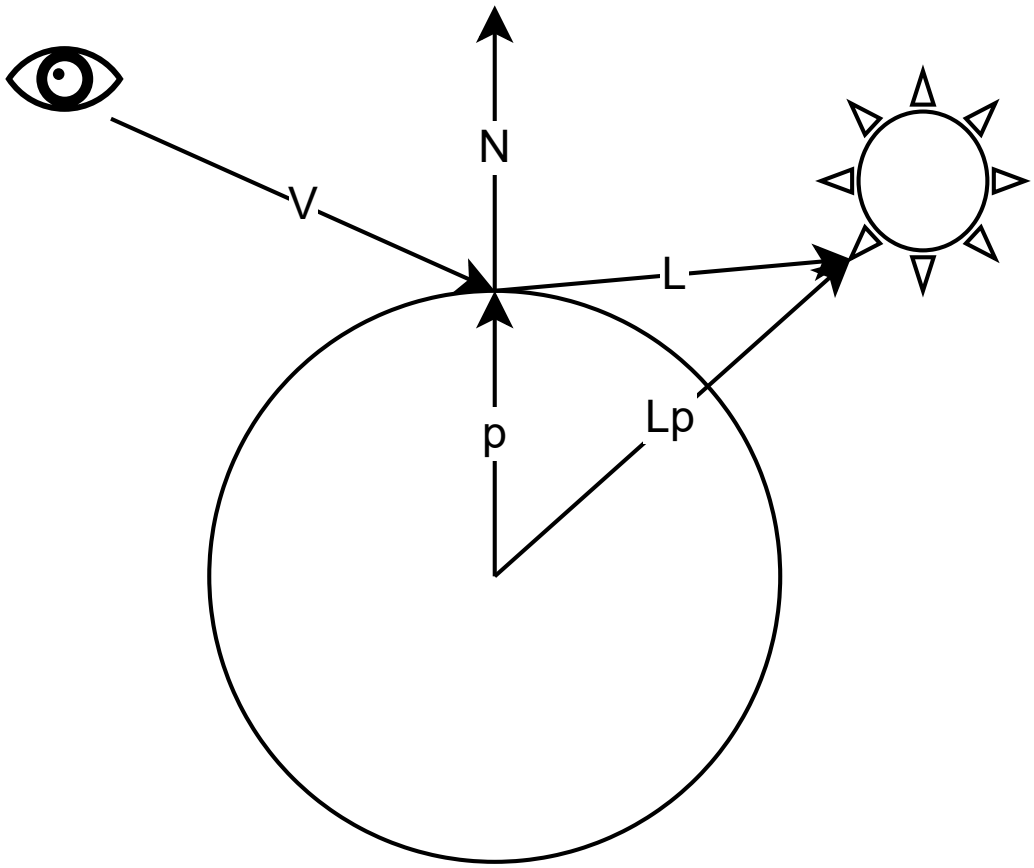
Da si poenostavimo izračun uporabimo samo en vir svetlobe in popolnoma difuzno površino krogle.

$$I_{\mathrm{p}} = I_{\mathrm{a}} + I_{\mathrm{d}}(\hat{L} \cdot \hat{N})$$

kjer predstavljajo:

I_{p}, predstavlja skupno osvetljenost točke.
I_{a}, predstavlja osvetljenost točke zaradi ambientne svetlobe.
I_{d}, predstavlja osvetljenost točke zaradi difuzne svetlobe svetlobe.

$\vec{p}$, predstavlja krajevni vektor točke na površini.
$\hat{L}$, predstavlja smerni vektor iz točke na površini proti viru svetlobe. [1]
$\vec{Lp}$, predstavlja krajevni vektor luči.
$\hat{N}$, predstavlja normalo točke na površini.



Slika 2.1: Diagram vektorjev

[1] Vektorji z strešico nad njimi npr. $\hat{L}$ so normalizirani, torej je njihova dolžina enaka 1

2.1.3 Poenostavitve

Da si prihranimo delo lahko naredimo nekaj poenostavitev

2.1.3.1 Pitagorov izrek

Da izračunamo če je naš piksel del krogle moramo izračunati x^2, y^2 in r^2. Ker je r^2 konstanta nam je kvadriranje prihranjeno. Nadalje si lahko prihranimo delo tako, da določimo $r = 1$.

2.1.3.2 Krajevni vektor

Za izračun krajevnega vektorja lahko uporabimo kar koordinate na našem (2D) zaslonu. Tako dobimo x in y koordinato. Da pridobimo z koordinato uporabimo formulo za točke na površini krogle:

$$x^2 + y^2 + z^2 = r^2$$

Če upoštevamo še $r = 1$ lahko izrazimo z koordinato kot

$$z = \sqrt{1 - x^2 - y^2}$$

Tako se efektivno izognemo računanju v 3D.

2.1.3.3 Normale

Če je krogla v središču našega koordinatnega sistema je smer normale preprosto enaka smeri krajevnega vektorja na površini krogle. Če predpostavimo $r = 1$ so naše normale tudi normalizirane.

2.1.3.4 Smerni vektor luči

Smerni vektor luči je izračunan po naslednji formuli:

$$\vec{L} = \vec{L_{\mathrm{p}}} - \vec{p}$$

Kjer predstavlja:

$\vec{L_{\mathrm{p}}}$, krajevni vektor vira svetlobe.
$\vec{p}$, krajevni vektor točke na površini.

Za pravilne izračune potrebujemo ta vektor normalizirati. Za to potrebujemo 3 deljenja in koren.

$$\hat{L} = \frac{\vec{L}}{|\vec{L}|}$$

$$\hat{L}.x = \frac{L.x}{\sqrt{L.x^2 + L.x^2 + L.y^2}}$$

$$\hat{L}.y = \frac{L.y}{\sqrt{L.x^2 + L.x^2 + L.y^2}}$$

$$\hat{L}.z = \frac{L.z}{\sqrt{L.x^2 + L.x^2 + L.y^2}}$$

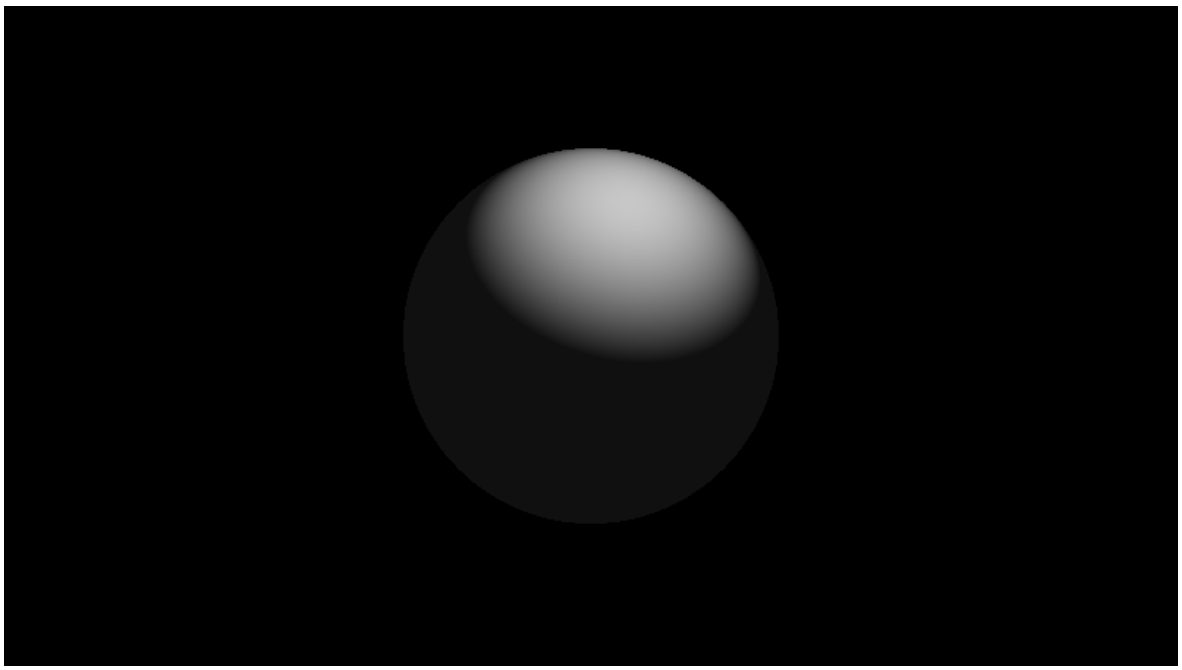
Da se temu izognemo, premikamo našo luč na fiksni razdalji od središča koordinatnega sistema. [2] [3]

[2] Kljub temu dobimo napako, ko smerni vektor luči in normala nista kolinearna, a napaka ni preveč moteča.

[3] Če vzamemo za razdaljo luči od središča $r = 2$, dobimo preveč svetlobe. To lahko popravimo z manjšo razdaljo, ker s tem zmanjšamo dolžino smernega vektorja luči. Dober kompromis za razdaljo je $r = \sqrt[3]{2}$

2.2 Programski model

Prvi korak načrtovanja je bila izdelava modela v višje-nivojskem jeziku. Za ta namen smo izdelali simulacijo v jeziku GLSL (OpenGL Shading Language) na spletni stani Shadertoy [4].
Koda je dostopna na slednjem spletnem naslovu: [5].



Slika 2.2: Slika modela

GLSL je jezik, ki nam omogoča izris slik s pomočjo grafičnega pospeševalnika. V njem zapišemo kodo, ki jo nato požene za vsak piksel na sliki. Kot vhodni podatki programu so koordinate trenutnega piksla in čas od zagona programa.

Program je napisan z uporabo števil z plavajočo vejico (float). V FPGA implementaciji, smo se odločili za fiksno natančnost. Pomembno je tudi dejstvo, da program uporablja 8-bitov za vsako barvo, medtem ko ima ZedBoard le 4-bite na barvo. Zato imamo v modelu zvezno barvo.

2.3 Implementacija

Implementacija se grobo deli na dva dela. Izvorna koda je dostopna na slednjem naslovu [6].

– Programski del
– GPU

Med deloma je povezava z AXI vodilom. V PL je tudi nekaj logike za prevedbo vodila v AXI4-Lite. Ta je generirana avtomatsko.

2.3.1 Programski del

Programski del je spisan v jeziku C. Njegova funkcija je prek AXI vodila posredovati koordinate naše luči GPU. Da bodo koordinate vedno na fiksni razdalji od središča jih rotiramo v krogu nad kroglo. Preden koordinate posredujemo GPU jih pretvorimo v fiksno natančnost (Q16.16). Luč vrtimo z hitrostjo ene stopinje na sliko (perioda $T = 5s$).

2.3.2 GPU

GPU je glavni del projekta, sestavljen iz pet logičnih pod-sklopov

– AXI vmesnik
– Izrisovalnik (renderer)
– Slikovni pomnilnik (framebuffer)
– VGA gonilnik

2.3.2.1 AXI vmesnik

GPU ima vhod za koordinate luči v obliki AXI4-Lite vmesnika. Uporabimo podrejeno (Slave) vodilo, ker podatke vedno posredujemo modulu. Prek vodila pišemo v 4 32-bitne registre (x,y,z in we)[4]. Koordinate sinhroniziramo z prikazom slike, da so koordinate za posamezno sliko vedno konstantne.

[4]we (write enable) omogoči pisanje v GPU registre. V našem primeru je vedno vklopljen

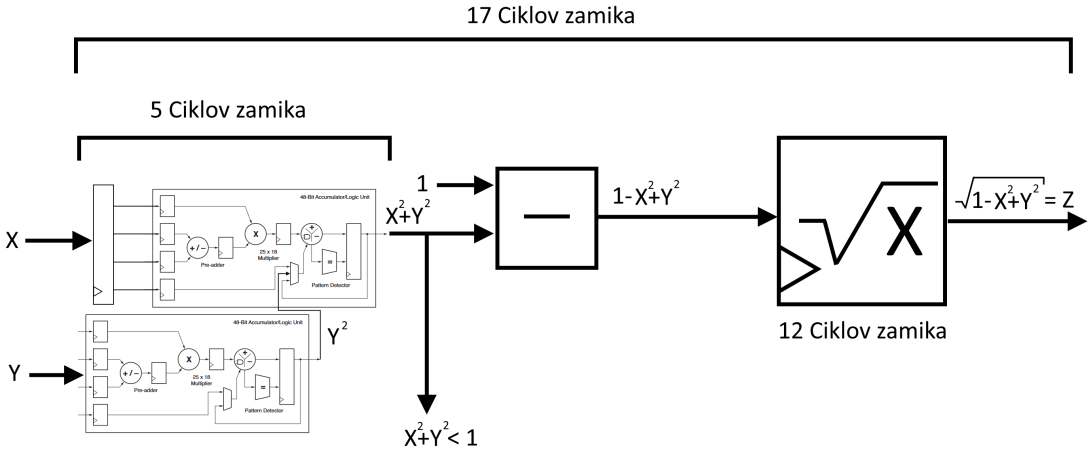
2.3.2.2 Izrisovalnik

Izrisovalnik ima 2 števca za x in y koordinato piksla, ki ga trenutno prikazujemo. Njegova naloga je, da določi barvo trenutnega piksla. Najprej postavimo koordinatno izhodišče na sredino zaslona. Torej namesto, da bi imele koordinate vrednosti $[800 \leftrightarrow 0{,}600 \leftrightarrow 0]$ jih imata $[400 \leftrightarrow -400{,}300 \leftrightarrow -300]$.

Nato poračunamo z koordinato z naslednjo formulo:

$$z = \sqrt{1 - x^2 - y^2}$$

Uporabimo vgrajene množilnike in lokalne povezave med njimi.



Slika 2.3: Arhitektura izračuna z koordinate

Za kvadratni koren uporabimo iterativni CORDIC algoritem.

Iz podatkov sedaj lahko vidimo če je naš piksel na krogli ali je del ozadja z naslednjo formulo:

$$x^2 + y^2 < 1$$

Uporabimo decimalna števila z fiksno natančnostjo. Za izračun koordinat uporabimo Q2.8, kar pomeni, da je radij naše krogle na zaslonu 256 pikslov.

Z z koordinato lahko sestavimo potrebne vektorje za izračun osvetljenosti I_p.

Krajevni vektor točke je jasno določen z x , y in z koordinatami. x , y in z prej še prevedemo v format Q16.16, saj v tem formatu dobimo koordinate luči.

$$\vec{p} = [x,y,z]$$

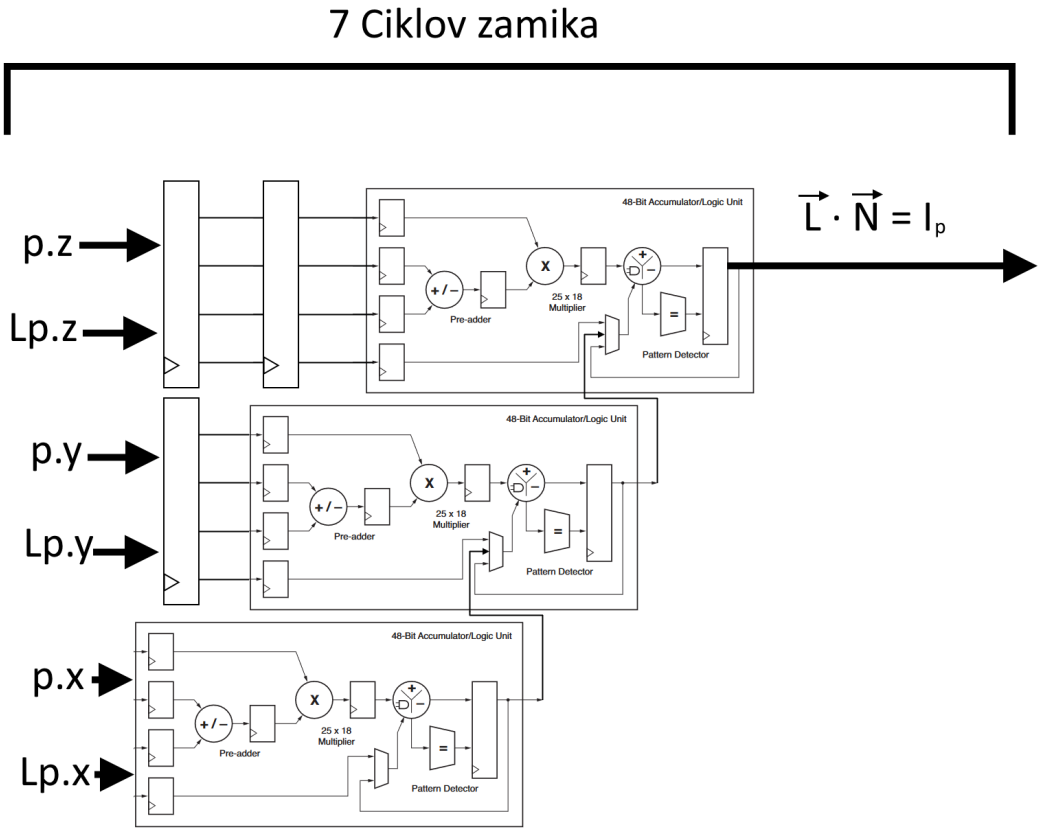
Normala je na krogli, kar enaka krajevnem vektorju.

$$\vec{N} = [x,y,z]$$

Pozicijo luči dobimo direktno iz PS. Smerni vektor luči poračunamo iz krajevnega vektorja točke in pozicije luči.

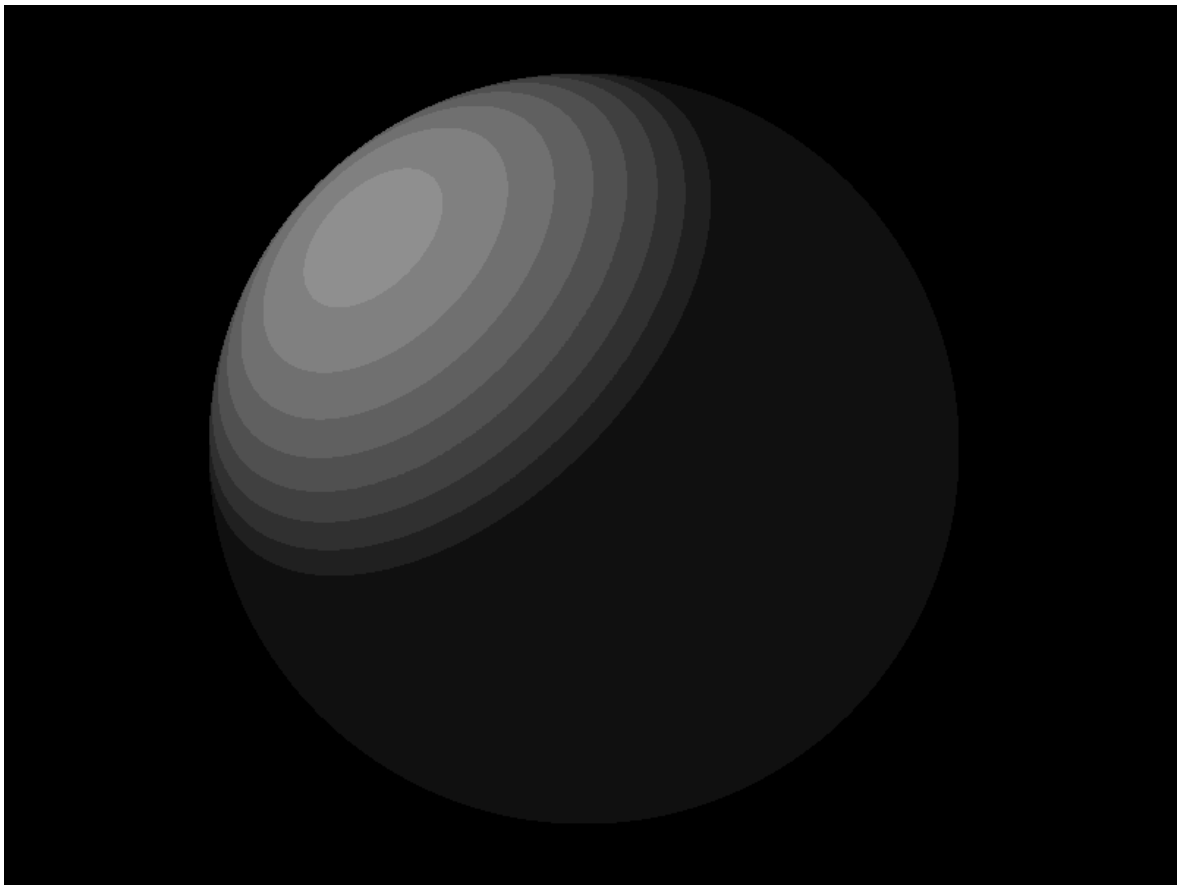
$$\vec{L} = \vec{L_{\mathrm{p}}} - \vec{p}$$

Ta vektor poračunamo kar v množilnikih, saj imajo notranje odštevalnike. Rezultat le še navzdol omejimo, da dobimo efekt ambientne osvetljenosti in barvo posredujemo slikovnem pomnilniku.



Slika 2.4: Arhitektura izračuna osvetljenosti piksla

Kot rezultat dobimo naslednjo sliko

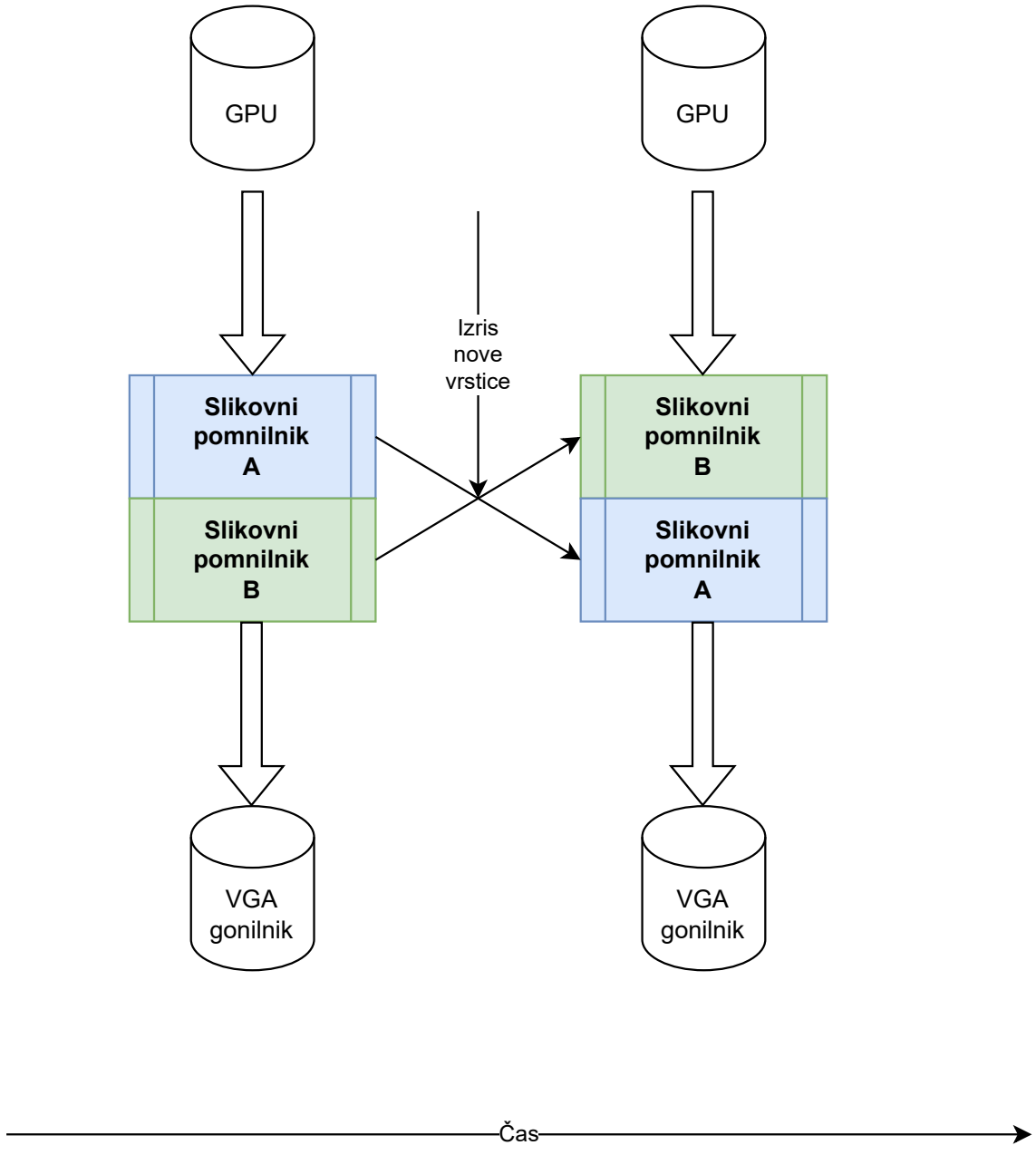


Slika 2.5: Izrisana krogla

Opazimo, da zaradi majhne barvne globine lahko jasno vidimo meje med odtenki. To odpravimo z poznejšim filtriranjem.

2.3.2.3 Slikovni pomnilnik

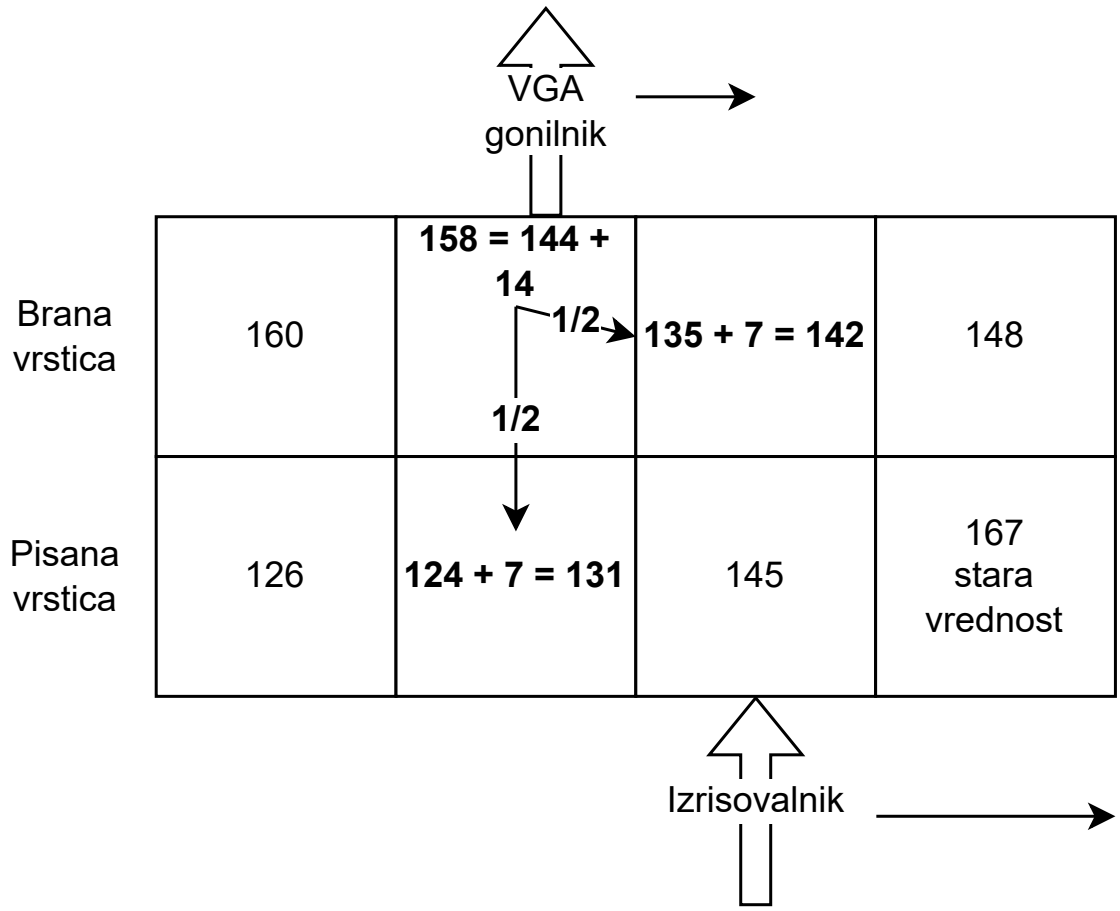
Slikovni pomnilnik nam sliko shrani medtem ko se izrisuje. Omogoča nam dodatno procesiranje preden sliko prikažemo.
Žal naš PL nima dovolj spominskih elementov, da bi shranil dve celotni sliki, zato shranimo dve vrstici. Medtem, ko eno prikazujemo, drugo izrisujemo.



Slika 2.6: Diagram slikovnega pomnilnika

S slikovnim pomnilnikom lahko opravimo dodatno funkcijo. Ker imamo na voljo le 4 bite na barvo, se na sliki vidijo posamezni prestopi med odtenki. To lahko rešimo z mešanjem odtenkov na prehodu (error diffusion dithering) [7].

Algoritem deluje na sledeč način:



Slika 2.7: Algoritem difuzije napake

Najprej barvo shranimo z večjo barvno globino, kot jo lahko prikažemo(v našem primeru 8 bitov). Nato vsak prikazan piksel zaokrožimo na 4 bite, napako ki pri tem nastane prištejemo sosednjim pikslom.

Pri tem uporabimo naslednji vzorec prerazporeditve:

$*$ predstavlja trenutni piksel.

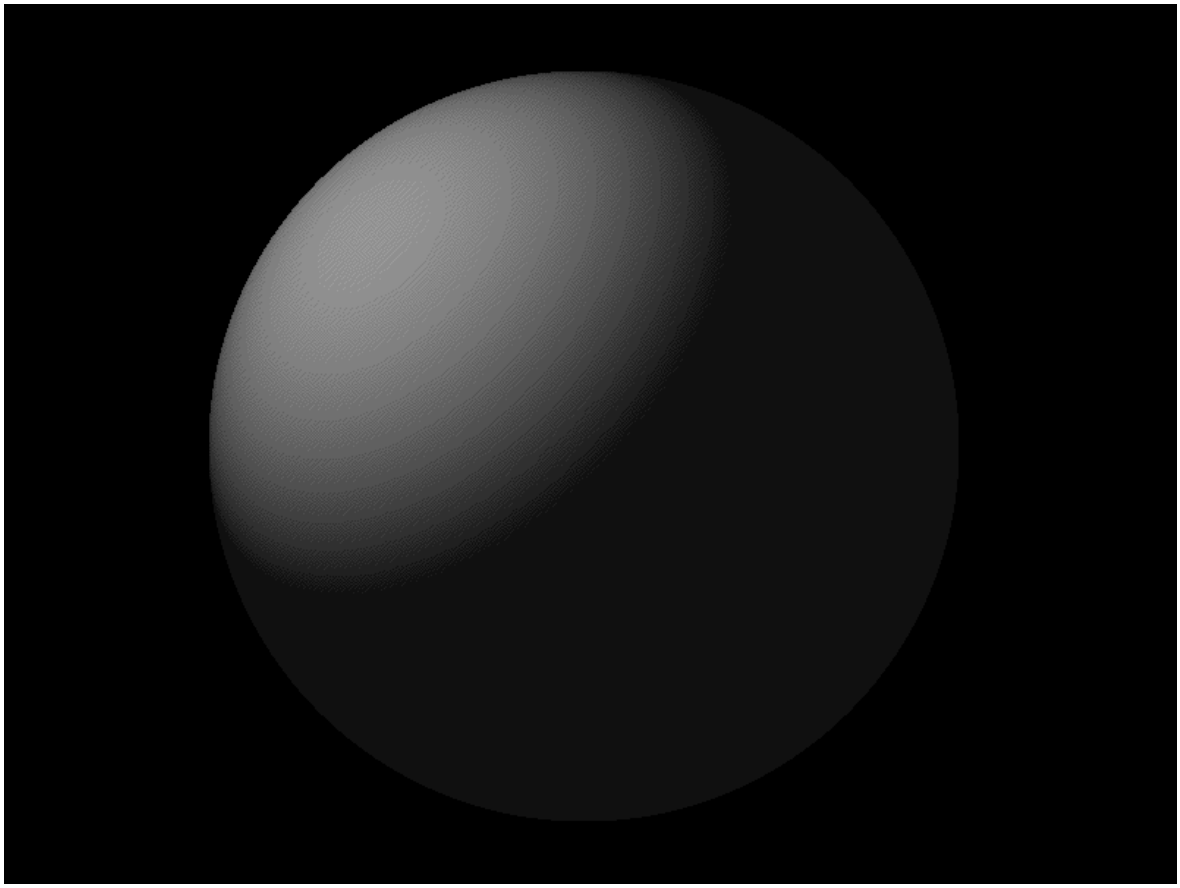
$$\begin{bmatrix} * & \frac{1}{2} \\ \frac{1}{2} & 0 \end{bmatrix}$$

Filtriranje izvedemo paralelno z branjem, torej potrebujemo dvo-kanalni spomin. [5] Iz brane vrstice naenkrat beremo trenutni piksel in popravljamo naslednjega. V pisano vrstico hkrati pišemo nove podatke in popravljamo stare podatke. Podatke lahko popravimo v enem urinem ciklu, saj lahko prek enega kanala hkrati beremo in pišemo.

Deljenje z 2 zlahka dosežemo prek bitnega zamika.

[5]Ker VGA gonilnik bere podatke z hitrostjo 50MHz, bi bilo mogoče shemo narediti tudi z enokanalnim spominom, za ceno znatne dodatne logike.

Da ta algoritem deluje moramo zagotoviti, da so sosednji piksli že izrisani, ko vrstico začnemo prikazovati. To dosežemo tako, da začnemo novo vrstico izrisovati že med sinhronizacijsko periodo (blanking period) zaslona. To nam tudi omeji globino izrisovalnega cevovoda.

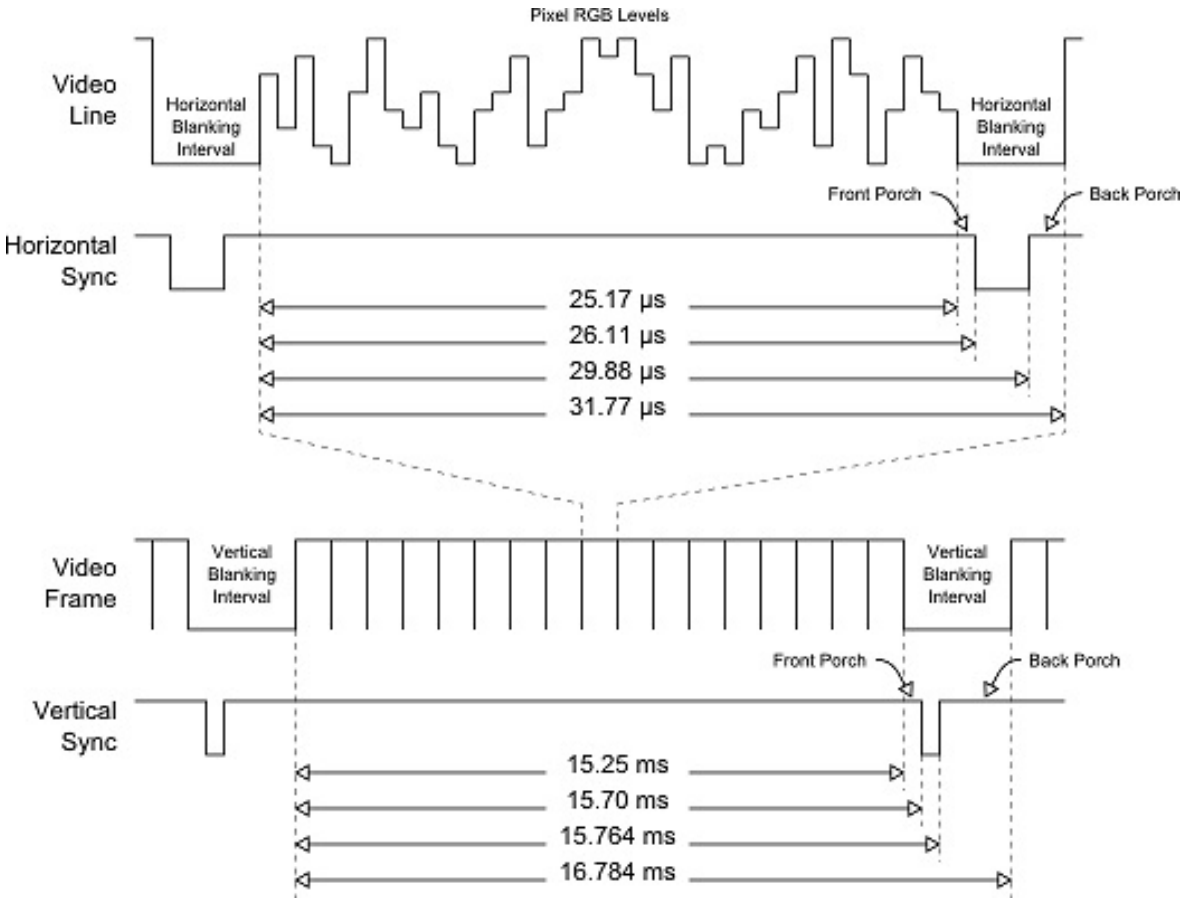


Slika 2.8: Izrisana popravljena krogla

2.3.2.4 VGA gonilnik

VGA gonilnik bere barvne vrednosti pikslov in jih prikazuje na zaslonu.

To stori z dvema števcema, za x in y koordinati zaslona. Na koncu vsake vrstice generira *hsync* signal za vodoravno sinhronizacijo in na koncu celotne slike še *vsync* sinhronizacijski signal.



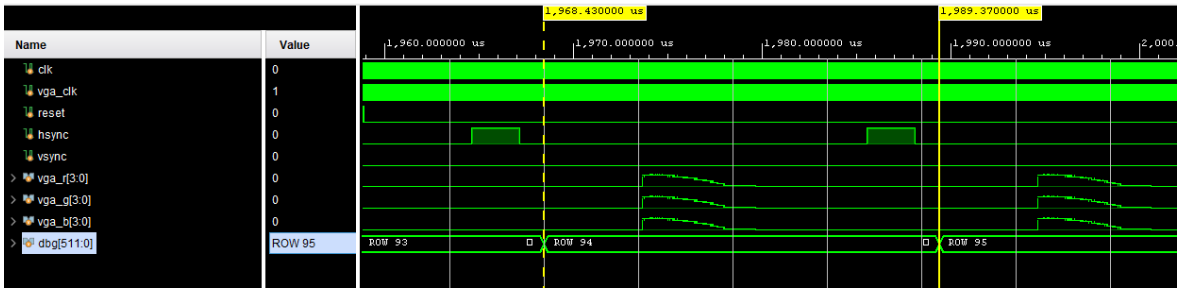
Slika 2.9: VGA časovni diagram

VGA uporabi analogen signal, ki ga pridobimo prek preprostega uporovnega delilnika iz našega digitalnega signala. Med prikazovanjem VGA gonilnik preprosto bere podatke iz slikovnega pomnilnika. Da dosežemo pravilni VGA signal, se morajo novi piksli pošiljati z hitrostjo 50MHz. Ker imamo glavno uro nastavljeno na 100MHz to storimo z preprostim 1 bitnim števcem (T Flip-Flop). [6]

[6]Deljeni urin signal uporabimo kot signal za omogočanje štetja in ne direktno kot urin signal. Vsa logika deluje na eni urini domeni.

2.4 Simulacija

Za simulacijo smo uporabili modul brez AXI vmesnika. V simulaciji prevedemo VGA izhod v slikovno datoteko formata .ppm. Izrišemo le eno sliko, da simulacija ne porabi preveč časa.



Slika 2.10: Simulacija vezja

3 Zaključek

Dosegli smo sprejemljivo animirano 3D sliko prikazano čez celoten zaslon z polno hitrostjo 72Hz. Če bi želeli prikazovati kaj bolj splošnega, bi bilo verjetno potrebno opustiti trenutni serijski izris slike in bi se morali poslužiti bolj splošne metode, kjer za vsak piksel izvedemo nek program na majhnem procesorju, izdelanem za ta namen. Verjetno bi morali opustiti tudi števila z fiksno vejico.

4 Literatura

[1] *TinyVGA — VESA Signal 800 x 600 @ 72 Hz timing*. 2022 URL `http://tinyvga.com/vga-timing/800x600@72Hz`, Dostopno 15-Februar-2022.

[2] *ZedBoard*. 2022 URL `https://www.avnet.com/wps/portal/us/products/avnet-boards/avnet-board-families/zedboard`, Dostopno 15-Februar-2022.

[3] Wikipedia: *Phong reflection model — Wikipedia, The Free Encyclopedia*. 2022 URL `https://en.wikipedia.org/wiki/Phong_reflection_model`, Dostopno 13-Februar-2022.

[4] *ShaderToy*. 2022 URL `https://www.shadertoy.com/`, Dostopno 14-Februar-2022.

[5] *ShaderToy*. 2022 URL `https://www.shadertoy.com/view/Nd2cDD`, Dostopno 14-Februar-2022.

[6] *GitHub — Sphere shading GPU for ZedBoard*. 2022 URL `https://github.com/6Kotnk/GPUv1`, Dostopno 16-Februar-2022.

[7] Wikipedia: *Error diffusion — Wikipedia, The Free Encyclopedia*. 2022 URL `https://en.wikipedia.org/wiki/Error_diffusion`, Dostopno 15-Februar-2022.