Generativna umjetnost u programskom jeziku Python

Šanovsky, Nikolina

SVEUČILIŠTE U DUBROVNIKU

ODJEL ZA ELEKTROTEHNIKU I RAČUNARSTVO

NIKOLINA ŠANOVSKY

GENERATIVNA UMJETNOST U PROGRAMSKOM JEZIKU PYTHON

DIPLOMSKI RAD

Dubrovnik, rujan 2021.

SVEUČILIŠTE U DUBROVNIKU

ODJEL ZA ELEKTROTEHNIKU I RAČUNARSTVO

NIKOLINA ŠANOVSKY

GENERATIVNA UMJETNOST U PROGRAMSKOM JEZIKU PYTHON

DIPLOMSKI RAD

Studij: Primijenjeno/poslovno računarstvo

Kolegij: Uvod u znanost o podacima

Mentor: izv.prof.dr.sc. Mario Miličević

Studentica: Nikolina Šanovsky

Dubrovnik, rujan 2021.

SAŽETAK

Ovaj diplomski rad opisuje pojam generative umjetnosti te izradu iste koristeći programski jezik Python. Rad se dijeli na dva moguća pristupa generativnoj umjetnosti koristeći *Python*, a to su kroz *Processing*, programski jezik i okruženje za stvaranje generative umjetnosti, te kroz primjenu dubokog učenja koristeći *TensorFlow* okruženje te Python-ove biblioteke, poput *Keras*-a, *Matplotlib*-a, i slično. U praktičnom djelu rada pišemo vlastite ili objašnjavamo postojeće primjere, dok u teoretskom navodimo definicije, povijest i domenu generative umjetnosti.

Ključne riječi: generativna umjetnost, *Python*, *Processing*, *TensorFlow*, duboko učenje

ABSTRACT

This thesis describes the concept of generative art and its creation using the Python programming language. The paper is divided into two possible approaches to generative art using *Python*, namely through *Processing*, a programming language and environment for creating generative art, and through the application of deep learning using the *TensorFlow* environment and *Python*'s libraries, such as *Keras*, *Matplotlib*, etc. In the practical part of the paper, we write our examples or explain existing ones, while in the theoretical part we state the definitions, history and domain of generative art.

Keywords: generative art, *Python*, *Processing*, *TensorFlow*, deep learning

SADRŽAJ

1 UVOD

U drugoj polovici 20. stoljeća, razvojem tehnologije, počelo se razmišljati o novim načinima izrade umjetnosti. Slobodnim pristupom velikim, laboratorijskim računalima znanstvenici su eksperimentiranjem došli do kodne umjetnosti, tj. umjetnosti koja se stvara programiranjem. Kodna umjetnost, potaknuta umjetničkim pravcima poput apstrakcionizma te 'pop' umjetnosti, bazirala se na geometrijskim oblicima i kombiniranjem boja. Uvođenjem slučajnosti, ponavljanja, uzoraka i dekompozicije, iz kodne umjetnosti nastala je podvrsta, generativna umjetnost, koja obuhvaća sve navedene značajke.

1.1 Definicija rada

Predmet istraživanja rada jest navođenje primjene *Python* programskog jezika u svrhu stvaranja generative umjetnosti. Prikazat će se i objasniti vlastiti i postojeći primjeri u okruženju *Processing.py* - a, a potom će se objasniti poveznica dubokih neuronskih mreža s generativnom umjetnosti u *Python*-u, tj. okruženju *TensorFlow*.

1.2 Svrha i ciljevi rada

Svrha rada jest približiti pojam generative umjetnosti publici, opisujući široki spektar primjene iste, dok je cilj rada kroz izradu i objašnjenje vlastitih te navođenje postojećih primjera, opisati kako točno izraditi generativnu umjetnost i koje elemente ona podrazumijeva.

1.3 Metodologija rada

U radu će se opisati okruženja poput *Processing*-a i *TensorFlow*-a, arhitektura Generalne kontradiktorne mreže (GAN), tj. arhitektura generatora i diskriminatora, te pojedine funkcije iz biblioteka korištenih za primjere kod dubokog učenja.

1.4 Struktura rada

Ovaj rad sastoji se od tri djela, od kojih je prvi dio uvod u rad i generativnu umjetnost općenito, kroz definiciju, povijest i alate. Drugi dio obuhvaća definiciju *Processing* okruženja te njegovog modula za *Python* te vlastite i postojeće primjere te njihova objašnjenja. Treći dio sadržava opis uloge dubinskog učenja u generativnoj umjetnosti, te postojeće primjere i objašnjenja, izrađene u *TensorFlow*-u koristeći *Python* i njegove biblioteke.

2 GENERATIVNA UMJETNOST

2.1. Definicija

Generativna umjetnost, vrsta kodne umjetnosti, zasniva se na autonomnom sustavu. Glavno obilježje autonomnog sustava je slučajnost, stoga, pri ponovnom pokretanju koda, očekujemo novo 'umjetničko' djelo, automatski se odričući potpune kontrole nad našim djelom. Takva umjetnost, kolaboracija je čovjeka koji napiše skriptu i računala, koje nam, slučajnim odabirom određenih značajki tog umjetničkog djela, daje nepredvidljive rezultate (vizuale, audio i slično). Osim slučajnosti, za generativnu umjetnost, bitni su još i algoritmi te geometrija. S algoritmima zadajemo određena pravila, dok nam je geometrija potrebna zbog učestalosti oblika i trigonometrije u ovakvoj vrsti umjetnosti.

2.2 Povijest

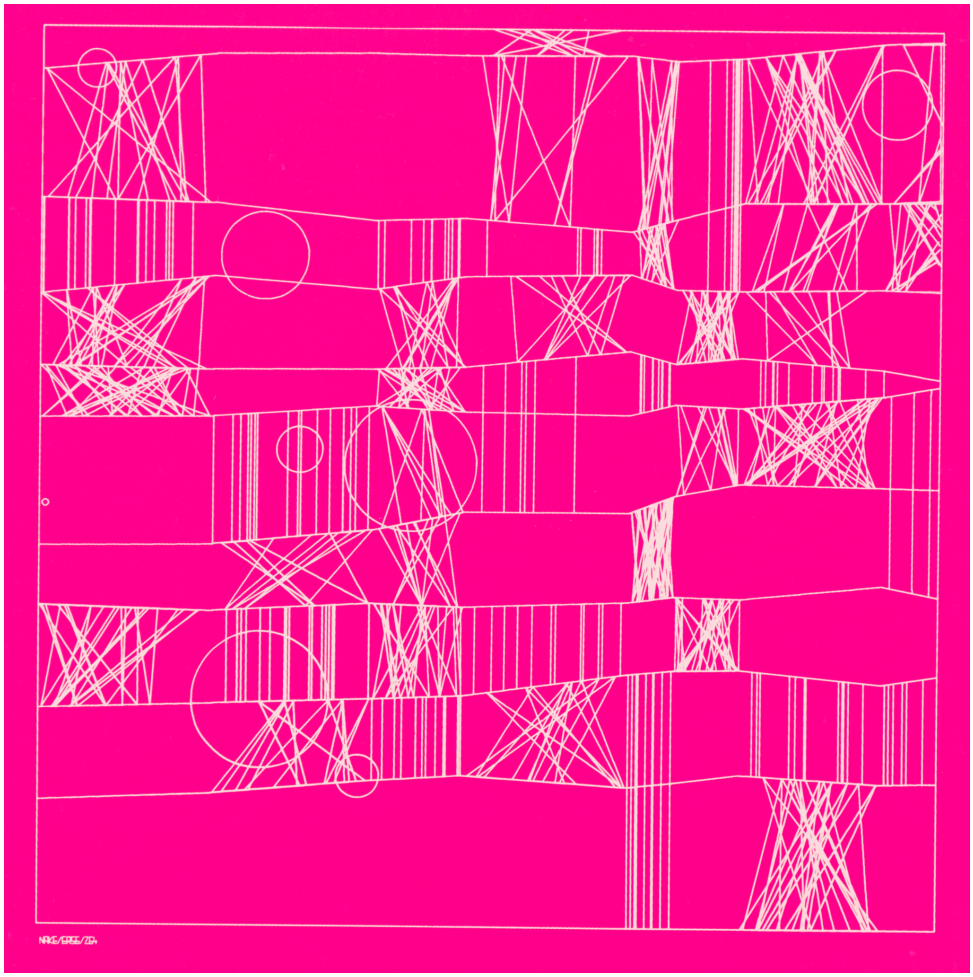
Premda je današnji pojam generativne umjetnosti definiran krajem dvadesetog stoljeća, prvotna inspiracija za istu potječe s kraja devetnaestog stoljeća, kada je Paul Cézanne postavio temelje kubizma, igrajući se s geometrijom. Početkom dvadesetog stoljeća, kulturni pokreti poput futurizma i konstruktivizma šire očaranost tehnologijom i strojevima te se već u tom razdoblju nazire suradnja tehnologije i umjetnosti. Umjetnički pravci s prve polovice 20. stoljeća, poput dadaizma, nadrealizma i apstraktnog ekspresionizma, ruše standardne norme i proširuju pojam umjetnosti, što postavlja dobre temelje za spajanje tehnologije i umjetnosti u nadolazećim godinama [1].

S pojavom prvih modernih računala, u razdoblju od 50-ih do 70-ih godina 20. stoljeća, započelo je eksperimentiranje s računalima veličine jedne prostorije. Prvi koji su eksperimentirali bili su znanstvenici s obzirom na to da su oni imali pristup laboratorijskim računalima, dok su umjetnici još neko vrijeme bili ograničenog pristupa istim.

Inženjer A. Michael Noll, bio je prvi koji je isprogramirao računalo za umjetničke svrhe. Napravio je prvu računalnu simulaciju uzoraka poznatih djela umjetnika poput Piet Mondrian-a te Bridget Riley [1].

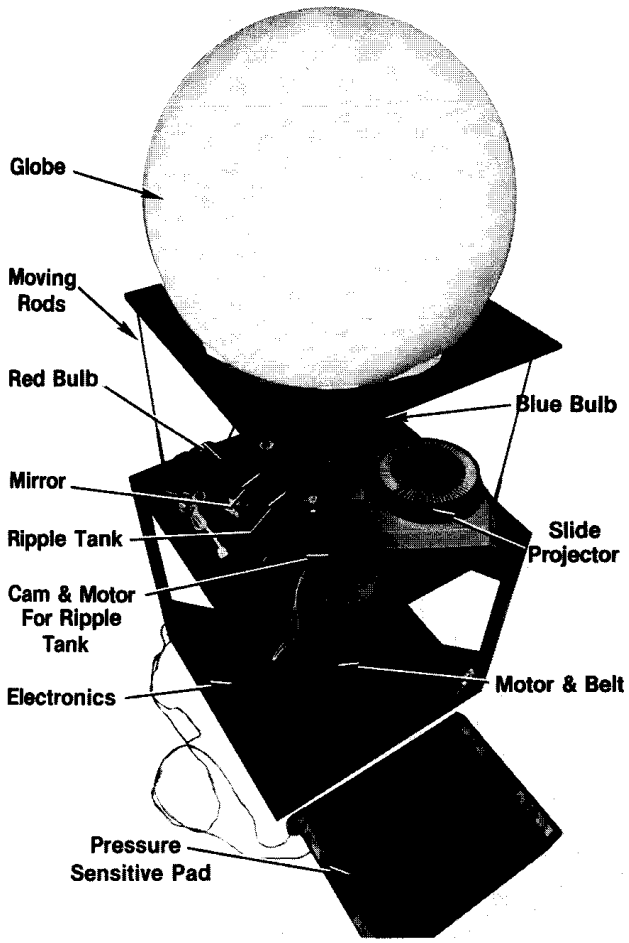
Prikaz rezultata, u to se vrijeme, vršio pomoću plotera, mehaničkog uređaja s nekom vrstom pisala koje bi, ovisno o skripti, iscrtavalo na podlogu. Ograničenost izlaznih resursa sužavala je domenu eksperimentiranja, stoga je umjetnost bila lišena animacija i drugih stvari koje su danas ključan aspekt generative umjetnosti.

Prvi čovjek koji je izložio svoj digitalni rad u galeriji bio je Frieder Nake (slika 1.), matematičar i informatičar. Većinu svojih radova učinio je pomoću preciznog plotera *Zuse Graphomat Z64* koristeći se kineskom tintom [1].



Slika 1. Frieder Nake, sitotisak, 1965. [2]

Lillian Schwartz prva je generativna umjetnica čiji se rad našao u Muzeju moderne umjetnosti u New York-u (MoMA) 1968. godine. Interaktivna skulptura *Proxima Centauri*, minimalne strukture s kupolom (slika 2.), sadržavala je jastučić na kojega bi promatrači stali, te bi pokretanjem motora uzrokovali različite, zanimljive efekte na kupoli (slika 3.).



Slika 2. Struktura *Proxima Centauri* skulpture [3]

Slika 3. *Proxima Centauri* (slika zaslona - okvir video zapisa)

Vera Molnar, Manfred Mohr te Herbert Franke još su neka od imena zaslužnih za uvođenje generative umjetnosti u primjenu.

Tijekom 70-ih godina 20. stoljeća, generativna umjetnost poprima neka od obilježja koja su danas prisutna, gdje se stvaranje istih širi i na prostore umjetničkih institucija. Povodom toga, 1970. godine, Škola umjetničkog instituta u Chicagu otvara odjel 'Generativni sustavi' koji obuhvaća umjetnost koja nastaje korištenjem tehnologije za prijenos, ispis i slično [1].

1988. Godine, teoretičar Henry Clauser izjavljuje da je sistemska autonomija ključni element generative umjetnosti te da takva umjetnost podrazumijeva dinamiku i kretnju koristeći se transformacijama i strukturama [1].

Generativna umjetnost je u 21. stoljeće, ušla pod vodstvom Jared-a Tarbell-a, suosnivača *Etsy*-a. Zamršenost i detalji, glavne su karakteristike njegovih djela, koja su dobila veliku popularnost (slika 4.).

Slika 4. Tarbell, Jared. *Happy Place*. 2004, *Processing Sketch* [4]

U sklopu *MIT Media Lab*-a i projekta *Dizajn brojevima*, 2001., Ben Fry i Casey Reas razvili su programski jezik *Processing*, koji se, za razliku od prethodnih softvera za animacije ili dizajn, odlikovao brzim radom. *Processing* je, za razliku od prethodnih alata vezanih za računalnu umjetnost, bio usmjeren na programere ili umjetnike koji žele stvarati pisanjem funkcija, a kako bi svima olakšali proces učenja, sam program je sadržavao primjere s osnovnim elementima potrebnim za napisati skriptu.

2.3 Glavni elementi

Generativna umjetnost, koja se stvara po zadanim pravilima, uključuje metode poput:

1. **Slučajnosti** - upravo je ona ta koja rad čini jedinstvenim, igrajući se s elementima poput boja, rotacije, veličine, oblika i slično. Bez slučajnosti, koristeći ostale metode, ne bi mogli stvoriti svaki put jedinstveno djelo.
2. **Ponavljanja** - u generativnoj umjetnosti ponavljanja su učestala zbog same sposobnosti da računala to odrade bez greške i u kratkom vremenu. Koristimo ih za stvaranje detaljnih struktura, mozaika i slično. Ponavljanja idu uz ruku slučajnosti, jer se koristeći

njome pri ponavljaju postižu zanimljivi rezultati koje viđamo u djelima generativne umjetnosti. Ponavljanja se najčešće vrše koristeći petlje.

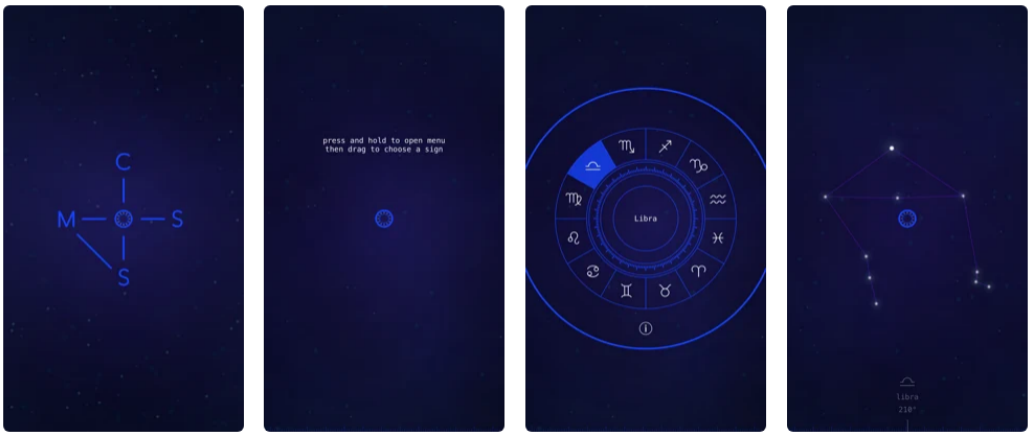
3. **Dekompozicija** - s tom metodom se veći oblici dijele na manje (trokuti, krugovi, kvadrati, itd.). U tu svrhu koristi se rekurzija.
4. **Konstrukcija** - ponavljajući izvedbu nekog pravila, oblik se povećava/širi. Suprotno od dekompozicije, ali također koristeći rekurziju.
5. **Prekidanje uzorka -** premda nije nužno, ova metoda se ponekad koristi kako bi unijeli nesavršenosti u rad, što ga čini zanimljivijim.
6. **Interakcija** - međudjelovanje objekta/oblika. Često kod animacija (odbijanje, privlačenje, sudaranje..)

2.4 Poznati alati za izradu generative umjetnosti

Za izradu generative umjetnosti koriste se različiti programski jezici i alati, a neki od poznatijih su:

1. ***Processing*** - trenutno najpoznatije programsko okruženje koje olakšava stvaranje 'kodne umjetnosti'. Baziran je na *Java* programskom jeziku, ali postoje i moduli za druge programske jezike poput *Python*-a, *JavaScript*-a, itd. *Processing* će biti detaljno razrađen u trećem poglavlju.
2. ***openFrameworks*** - besplatni razvojni okvir koji se koristi *C++* programskim jezikom u stvaranju kreativnih i eksperimentalnih projekata u stvarnom vremenu, koji uključuju audio, video, umrežavanje i računalni vid, a tome pridonosi veliki broj funkcija i klasa za rad s dvodimenzionalnim (2D) i trodimenzionalnim (3D) medijima. Sadrži mnogo dodataka vezanih za biblioteke, senzore te uređaje, što omogućuje napredne projekte. Kod se izvršava na svim popularnim platformama.
3. ***Cinder*** - biblioteka za kreativno kodiranje, također namijenjena za *C++* programski jezik. Kao i *openFrameworks*, može se izvršavati na svim bitnim platformama, poput *Windows*-a, *Linux*-a, *iOS*-a, *macOS*-a, itd. Pogodan je za eksperimentiranje, ali i za profesionalni rad.
4. ***C4*** - okruženje namijenjeno za kreativno kodiranje, koje za to svrhu koristi snagu izvornog *iOS* programiranja (*Swift* programski jezik), uz jednostavno sučelje koje

ubrzava proces. Glavne značajka *C4* je efikasnost, tj. jednostavnost izrade te brzina pokretanja različitih medija i animacija, za što su zaslužne kombinacija *UIKit*-a i *Core Animation*-a, te baza zasnovana na izvornom kodu. Najnovija verzija *C4* nudi i tutorial kako sam napraviti *C4SMOS* aplikaciju, vodeći nas kroz korake dizajna, izrade te objave iste. Već postojeća aplikacija *C4SMOS* je napravljena upravo u *C4*, a glavni cilj joj je kroz zanimljivu interakciju korisnika koji razgledava 'kozmos' pokazati što se sve može napraviti koristeći ovakvo okruženje (slika 5.)[5].



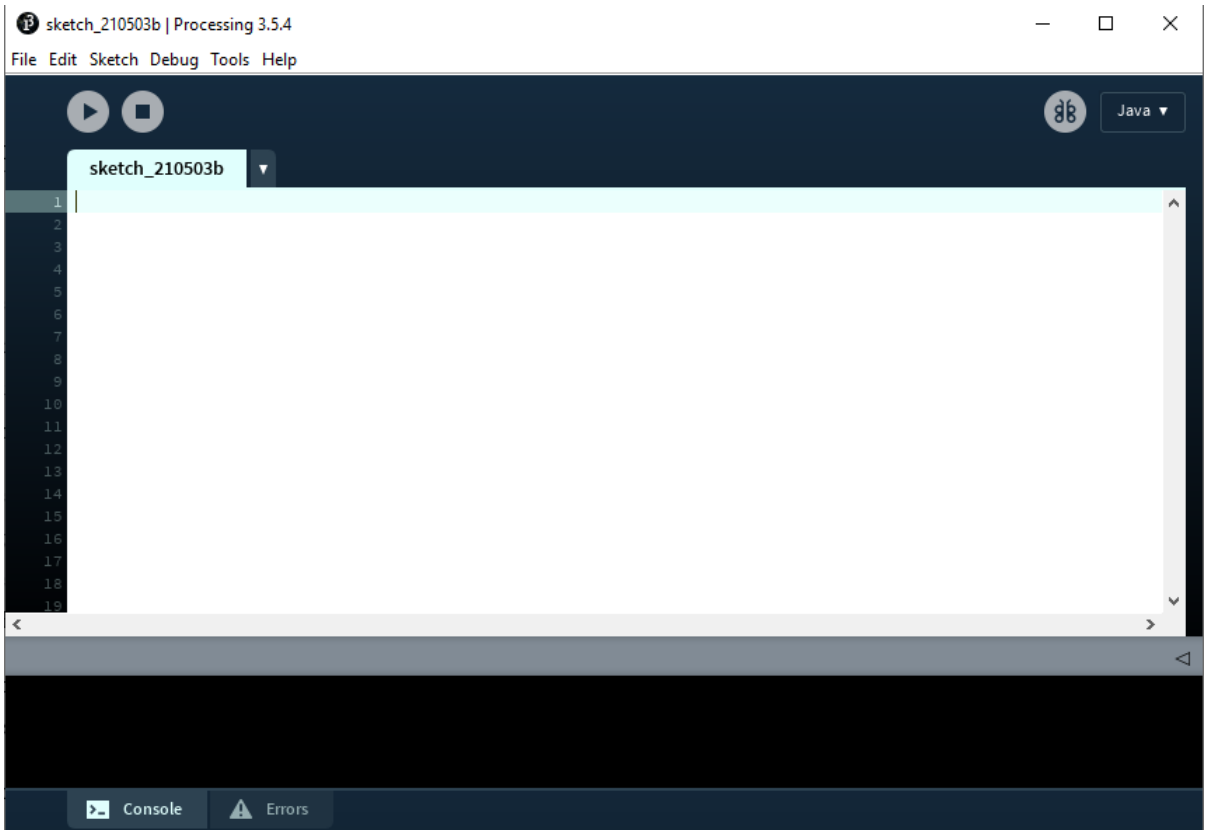
Slika 5. Slika zaslona *C4SMOS*-a s *App Store*-a [6]

5. ***Unity*** - višeplatformski softver za razvoj 2D/3D igara koji se služi *C#* programskim jezikom. Premda je softveru glavni fokus na računalne igre, sam alat ima veliki broj mogućnosti koje se mogu koristiti kod svih pojedinih polja, koje često obuhvaćaju i same igre, a to su: animacije, grafike, manipulacija zvukom i slično.

3 *PROCESSING*

Processing, kako navodi *Processing Foundation*, je razvojno okruženje osmišljeno da promovira pisanje koda u poljima koji se bave vizualima, te da promovira umjetnost i vizualizaciju u tehnološkim poljima, tj. da ohrabri ljude različitih interesa da nauče nove vještine koje mogu ukomponirati u svoje, već postojeće znanje [7].

Processing, programsko okruženje nastalo 2001. od grupe s *MIT*-a (Ben Fry i Casey Reas), baziran je na *Java* programskog jeziku, uz određene modifikacije i simplifikacije matematičkih funkcija i operacija. *Processing* ima svoje integrirano razvojno okruženje (engl. *Integrated Development Environment* - IDE) , a pojedini prozori unutar njega nazivaju se skice (engl. *sketch*), čime se naglašava razina eksperimentiranja ovog pristupa umjetnošću (slika 6.).

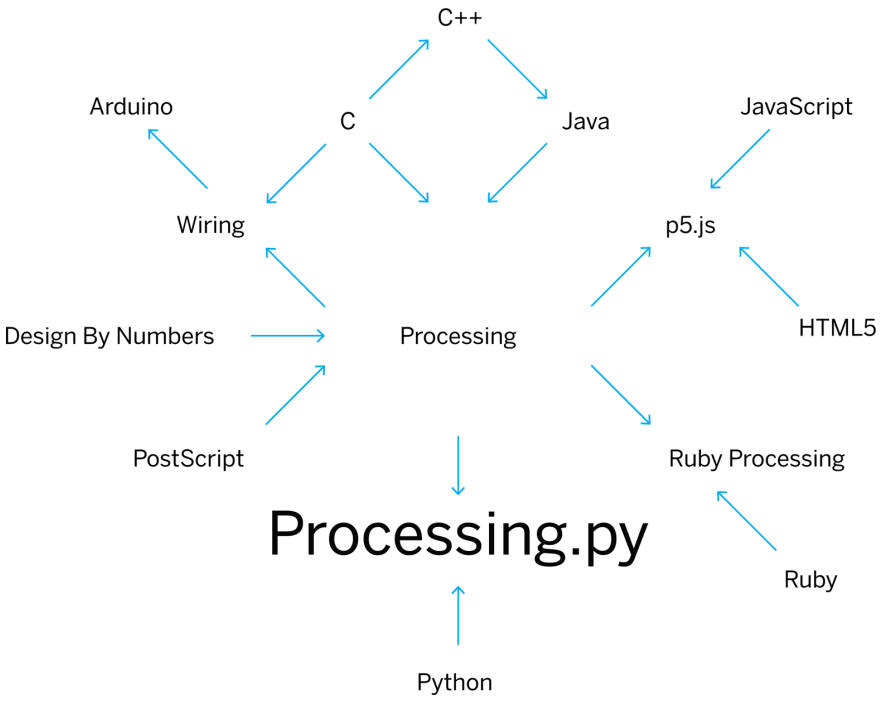


Slika 6. *Processing* IDE (snimka zaslona)

Skice podržavaju crtanje dvodimenzionalnih i trodimenzionalnih grafika. Zadano je crtanje u 2D-u, dok se odabirom P3D prikazivača omogućuje upravljanje kamerom, osvjetljenjem i materijalom.

Mogućnosti *Processing*-a su mnogobrojne zahvaljujući aktivnoj zajednici koja je pridonijela stvaranju stotinu biblioteka i alata, koji proširuju primjenu korištenja van osnovnog spektra. S tim dodacima omogućen je rad sa zvukovima, računalnim vidom te naprednom 3D geometrijom.

Tijekom godina razvijanja, *Processing* je razvio sučelja za programiranje u drugim programskim jezicima poput *JavaScript*-a (*P5.js*), *Python*-a (*Processing.py*), *Ruby* (slika 7.).



Slika 7. *Processing* i podržavani programski jezici [8]

3.1 *Processing.py*

Processing.py grafička je biblioteka bazirana na *Processing*-u, većinski razvijena od strane Jonathan-a Feinberg-a, uz pomoć kolega. Prva nezavisna implementacija puštena je u javnost 2010. godine, a 2014. godine dobiva podršku od *Google*-a. Dokumentacija i primjeri razvijeni su

u sklopu inicijative zvane ‘Integrirani dizajn, umjetnost i tehnologija’ (IDeATe) na Sveučilištu Carnegie Mellon i potporom Nacionalne zaklade za umjetnost.

3.1.1 Osnove

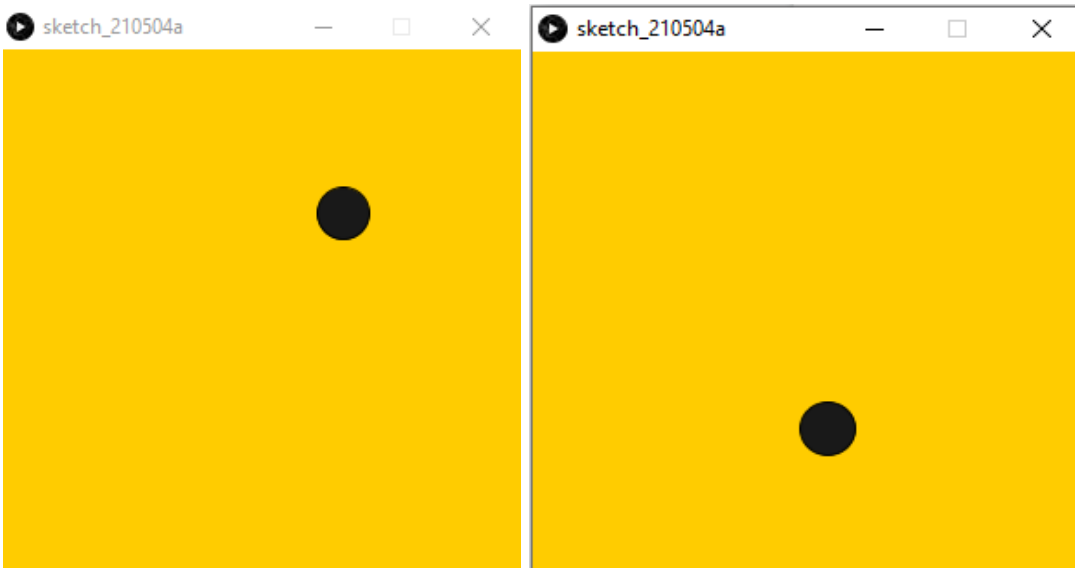
Dvije osnovne funkcije za pisanje bilo kojeg programa u *Processing*-u su `setup()` i `draw()`.

Funkcija `setup()` poziva se pri pokretanju programa, a sadrži sve potrebno za opisati okruženje programa poput veličine ekrana - `size(x,y)`, boje pozadine - `background(...)`, definiranja fontova i slično. Funkcija se ne smije pozivati više od jednog puta (isključivo na početku) te se varijable deklarirane unutar nje ne mogu pozivati u drugim funkcijama.

Funkcija `draw()` poziva se nakon `setup()` i vrti se sve dok program radi ili dok se ne pozove funkcija `noLoop()` - što znači da se `draw()` izvršava samo jednom. Broj ponavljanja te funkcije u sekundi može se definirati funkcijom `frameRate()`. U funkciji se često na početku definira boja pozadine kako bi oponašali kretanje nekog objekta/oblika bez tragova jer se kontinuiranim definiranjem pozadine briše staro stanje ekrana te tako možemo dobit privid gibanja, poput lopte koja odskače i slično (slika 8., slika 9.).

```
def setup():
    size(300,300)
    frameRate(4)

def draw():
    background(255,204,0)
    fill(25)
    circle(random(300),random(300), 30)
```

Slika 8. Primjer postavljanja `setup()` i `draw()` funkcije



Slika 9. Primjer `setup()` i `draw()` funkcije (slike zaslona)

Pozicija lopte u primjeru je nasumična, tj. `random()` funkcijom slučajno definiramo os x i os y, dok je treći parametar funkcije `circle()` širina kruga, koju smo jasno definirali, da bi stvorili privid da je to ista 'lopta'. Funkcijom `fill()` definiramo boju oblika koji slijede u nastavku koda, sve dok se opet ne definira `fill()` funkcija za neki drugi oblik. Za definiranje koristimo RGB (*red*, *green*, *blue*) ili HSB (*hue*, *saturation*, *brightness*) spektar boja.

Popis i objašnjenje svih funkcija i klasa nalazi se u repozitoriju službene stranice, a iste, korištene u primjerima, ćemo opisati naknadno.

3.2 Processing.py - Vlastiti primjeri generative umjetnosti

3.2.1. '*Screen-saver*' iz 2000-ih

Prvi primjer bit će imitacija jednostavne '*screen-saver*' animacije s početka 21. stoljeća, a skriptu za tu imitaciju možemo vidjeti na slici 10.

```
xSpeed = 10
ySpeed = 10

x = 0
y = 0
sizeCircle = 40

def setup():

    size(500,500)
    background(0,0,0)

def draw():

        global xSpeed, x, ySpeed, y, sizeCircle

        x = x + xSpeed
        y = y + ySpeed

        noStroke()
        col = color(random(255),random(255),random(255))

        circ = circle(x,y,sizeCircle)

        if(x >= width):
            fill(col)
            xSpeed = xSpeed * (-1)
            sizeCircle = random (20,50)

        if(y >= height):
            fill(col)
            ySpeed = random(-30,-10)
```

Slika 10a. Skripta za '*Screen-saver*' (1/2)

```
sizeCircle = random (20,50)

        if(x < 0):
            fill(col)
            xSpeed = random(10,30)
            sizeCircle = random (20,50)

        if(y < 0):
            fill(col)
            ySpeed = random(10,30)
            sizeCircle = random (20,50)
```

Slika 10b. Skripta za 'Screen-saver' (2/2)

Varijable `xSpeed` i `ySpeed` predstavljaju nam ubrzanje s kojim ćemo utjecati na poziciju kruga, ovisno o tome koji rub ekrana je dotakao. `x` i `y` također definiramo kao globalne varijable s vrijednošću 0, kako bismo ih naknadno mijenjali.

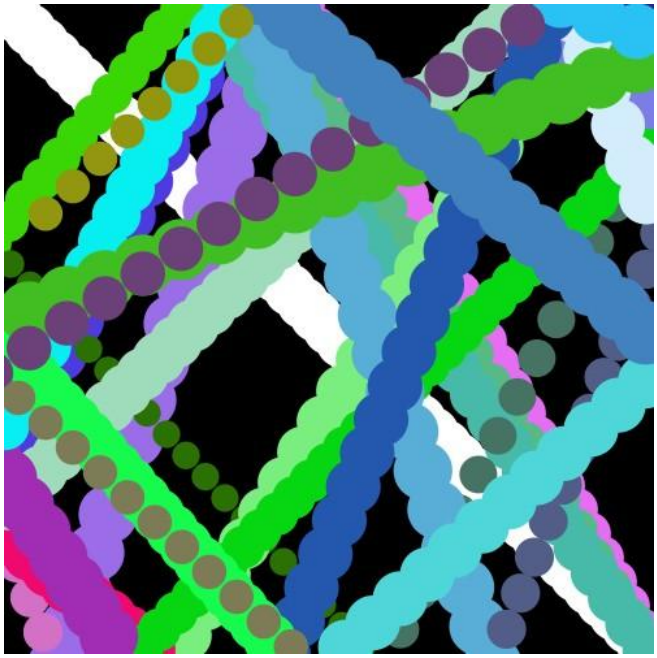
U funkciji `draw()`, `x` i `y` varijablu povećavamo za početnu vrijednost ubrzanja, što je 10, a u svakoj idućoj petlji ona će se slučajno mijenjati u odnosu na uvjete koje zadovoljava u okvirima neke domene. `noStroke()` funkcijom mičemo obrub kruga, a varijabli `col` pridodajemo, u svakom krugu, slučajnu boju u cijelom RGB spektru. Potom funkcijom `circle()` crtamo krug. Prvi krug počinje uvijek iz gornjeg lijevog kuta (10,10), a potom je ostatak vizuala slučajan ovisno o ubrzanju. Cilj je da svaki put kada krug dotakne bilo koji rub ekrana, krug promijeni boju i smjer.

Ako je koordinata x veća ili jednaka širini prozora `(width)`, `xSpeed` množimo s -1 tako da kada taj broj pribrojimo varijabli `x`, ona počne smanjivati.

Ako je koordinata y veća ili jednaka visini prozora (height) slučajno odabiremo negativan broj između -30 i -10 te mijenjamo varijablu `ySpeed`, kako bi se krug 'odbio' s dozom slučajnosti u kojem točno smjeru.

Ako je `x` ili `y` manji od nula, mijenjamo `xSpeed` ili `ySpeed` odabirom slučajnog broja između 10 i 30.

Dakle, veličinu kruga i boju mijenjamo isključivo kada krug dotakne jedan od rubova što možemo vidjeti na slici 11.



Slika 11. ‘Screen-saver 00’s’ (spremljeni okvir)

Ako na početak `draw()` funkcije dodamo definiranje boje pozadine, tj. `background(0)`, umjesto tragova, dobit ćemo privid lopte koja se odbija i pri tom mijenja boju i veličinu.

3.2.2 Američka krafna (engl. *Donut*)

U primjeru ‘američka krafna’ prikazano je korištenje transformacijskih funkcija kako bismo dobili zanimljiv vizual koji podsjeća na slaganje dodataka na američke krafne. U kodu možemo vidjeti nove funkcije kao što su `translate()`, `pushMatrix()`, `popMatrix()` i `rotate()` (slika 12.).

```
def setup():

    size(600,600)
    background(random(255),random(255),random(255))
    frameRate(0.8)

def draw():

    fill(random(255),random(255),random(255))
    translate(width/2,height/2)

    for r in range (0,360):

        x = random (2,100)
        y = random (100,200)

        pushMatrix()

        rotate(radians(r))
        circle(x,y,30)

        popMatrix()

def keyPressed():
    saveFrame("donut{}.jpg".format(frameCount))

def mousePressed():
    background(random(255),random(255),random(255))
```

Slika 12. Kod za ‘američke krafne’

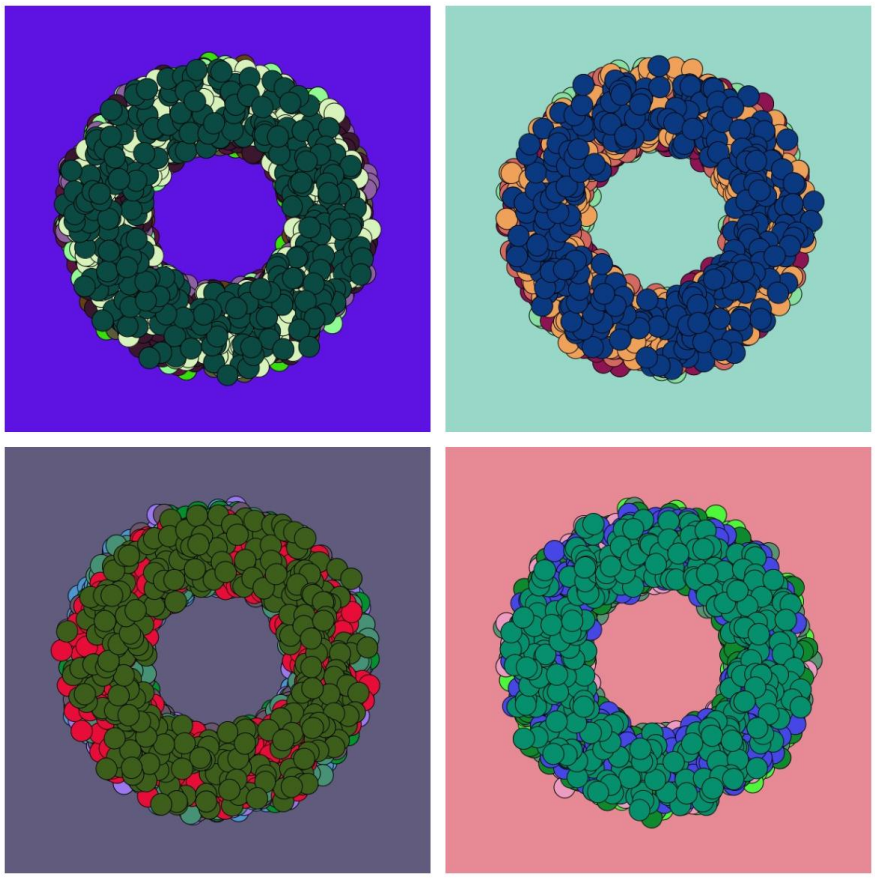
Funkciju `translate()` koristimo kada iz praktičnih razloga želimo pomaknuti koordinatni sustav, a u ovom primjeru cilj je točku (0,0) pomaknuti u sredinu, stoga ga premještamo na pola širine i visine.

Funkcije `pushMatrix()` i `popMatrix()` koristimo s transformacijskim funkcijama kada želimo kontrolirati granice transformacija. Funkciju `pushMatrix()` koristimo da trenutno stanje koordinatnog sustava spremimo u memoriju, dok `popMatrix()` koristimo da isti vratimo iz memorije. U ovom primjeru crtamo krugove tako da rotiramo koordinatni sustav, nacrtamo krug, a potom vratimo stari iz memorije.

Funkcija `rotate()`, kao što sama riječ kaže, služi za rotiranje koordinatnog sustava, u ovom slučaju stupanj po stupanj dok ne postignemo puni krug.

Za kraj smo napisali dvije funkcije koje nam omogućuju dodatne izbore kao što je spremanje trenutne slike pritiskom na bilo koju tipku - `keyPressed()` i slučajno mijenjanje boje pozadine klikom miša - `mousePressed()`.

U nastavku možemo vidjeti četiri spremljena primjera (slika 13.).



Slika 13. Četiri primjera spremljenih okvira 'američkih krafni' (vlastiti izvor)

3.2.3 Moderna grafika - nastavak prethodnog primjera

U ovom primjeru vidjet ćemo različite moderne grafike korištenjem i modifikacijom prethodnog koda (slika 14.).

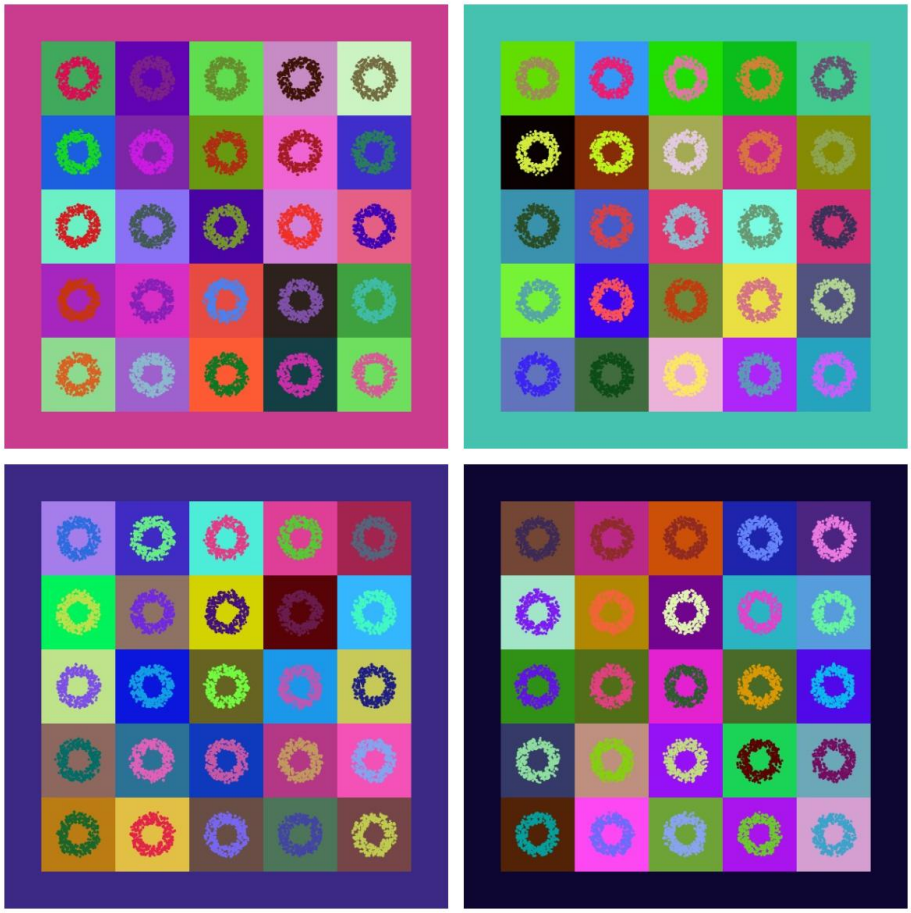
```
def setup():
    size(600,600)
    background(random(255),random(255),random(255))
    frameRate(0.7)
    rectMode(CENTER)

def draw():

    for i in range(100,width, 100):
        for j in range(100,height, 100):

            fill(random(255),random(255),random(255))
            noStroke()
            rect(i,j,100,100)

            pushMatrix()

            translate(i,j)
            scale(0.14)

            fill(random(255),random(255),random(255))
            #translate(width/2,height/2)

            for r in range (0,360):

                x = random (2,100)
                y = random (100,200)

                pushMatrix()

                rotate(radians(r))
                #fill(random(255),random(255),random(255))

                circle(x,y,random(20,30))
                popMatrix()
            popMatrix()

def keyPressed():
   saveFrame("grafika{}.jpg".format(frameCount()))

def mousePressed():
    background(random(255),random(255),random(255))
```

Slika 14. Moderna grafika - kod

U `setup()` funkciji postavljamo `rectMode(CENTER)`, što je drugačiji način definiranja kvadrata, koji je inače postavljen na `CORNER`, tj. da su `i`, `j` varijable gornji lijevi kut, a u ovom načinu predstavljaju sredinu kvadrata.

Prolaskom kroz petlju u petlji stvaramo mrežu kvadrata te unutar tih petlji privremeno koordinatni prostor pomičemo da počinje u središtu trenutnog kvadrata (`translate()`), te smanjujemo crtanje 'krafni' za x0.14 (`scale()`) kako bi odgovaralo veličini kvadrata. Zatim iskoristimo dio koda iz prethodnog primjera, gdje možemo birati hoćemo li ostaviti odabir slučajnih boja ili ćemo ići na *color-blocking* šemu, što je česti slučaj u modernoj umjetnosti.

Također imamo mogućnost spremanja grafika i mijenjanja boje pozadine (slika 15.).



Slika 15. Četiri primjera za modernu grafiku - *color blocking* (spremljeni okviri)

3.2.4 Solarni sustav (sunce-zemlja-mjesec) - animacija

U ovom primjeru osmišljena je imitacija solarnog sustava (sunce, zemlja, mjesec) s polu-slučajnim faktorima poput mijenjanja zraka i boje sunca (u žuto-narančastom spektru) te veličina samog mjeseca i zemlje (Slika 16.).

```
index = 0
orb_speed = 0

def setup():
    global width_circle, width_moon_circle
    size(600,600)
    width_circle = random(380,450)
    width_moon_circle = random(80,120)
    frameRate(1)

def draw():

    global orb_speed, index
    background(random(200),random(255),random(255))

    translate(width/2,height/2)

    #orbita oko sunca
    noFill()
    stroke(random(200,255),random(200,255),random(100))
    strokeWeight(3)
    circle(0,0,width_circle)

    #sunce
    fill(random(200,255),random(200,255),random(100))
    noStroke()
    circle(0,0,70)

    pushMatrix()

    rotate(orb_speed);

    for r in range (0,360,int(random(5,10))):

        y = random (100,150)
        pushMatrix()
```

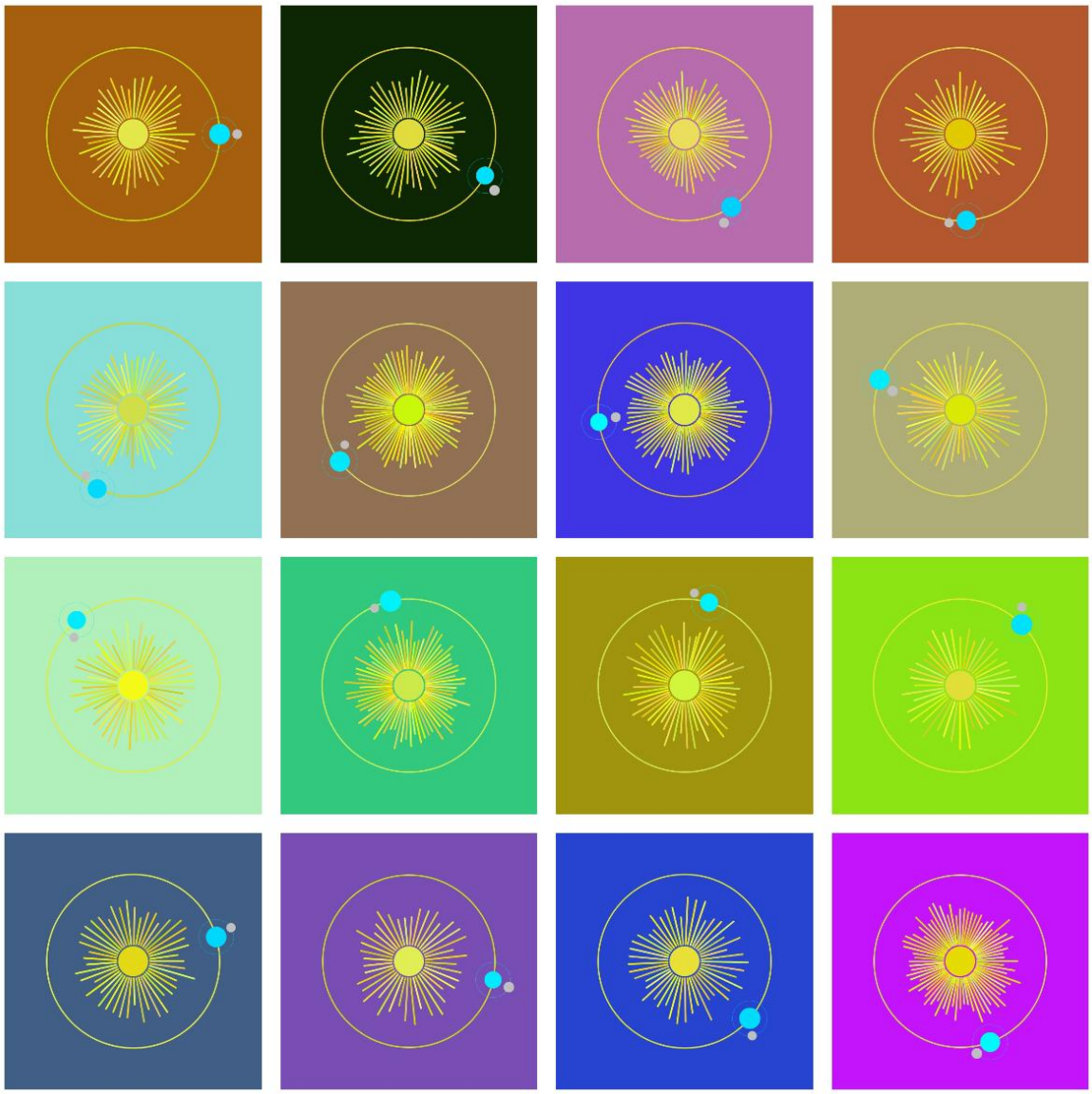
Slika 16a. Skripta za solarni sustav - kod (1/2)

```
  rotate(radians(r))
       #zrake sunca
       stroke(random(200,255),random(200,255),random(100))
       strokeWeight(4)
       line(40,0,y,0)
       popMatrix()
#orbita za mjesec
   translate(width_circle/2, 0);
   noFill()
   stroke(0, random(200,255), 255)
   strokeWeight(0.5)
   circle(0,0,width_moon_circle)

   pushMatrix();

   rotate(orb_speed);
   translate(width_moon_circle/2, 0);

   #mjesec
   noStroke()
   fill( "#C0C0C0");
   circle(0, 0, random(20,25));

   popMatrix()

   #zemlja
   noStroke()
   fill(0, random(200,255), 255)
   circle(0, 0, random(40,50))
   popMatrix()

   orb_speed = orb_speed + 0.5
   index = index + 1

   saveFrame("sun{}.jpg".format(index))
```

Slika 16b. Skripta za solarni sustav - kod (2/2)

Funkcije korištene u ovom primjeru već su objašnjene u prethodnim primjerima.

U svakom novom okviru (engl. *frame*) mjesec se okreće oko Zemlje, Zemlja oko Sunca, a Sunce poprima novi izgled (generiraju se nove zrake) uz novu, slučajnu, pozadinsku boju (slika 17 - 16 okvira).



Slika 17. 16 uzastopnih okvira solarnog sustava (vlastiti izvor)

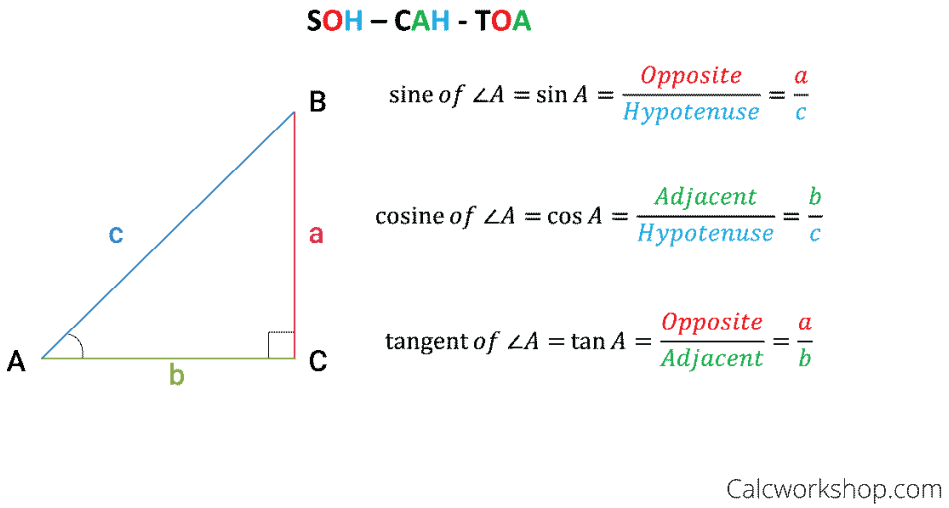
3.2.5 '*Sparkling*' oblik - animacija

Koristeći trigonometrijske funkcije i polarno definiranje koordinata napravljena je animacija stvaranja svjetlucajućeg oblika (engl. *Sparkling shape*)(slika 18.).

```
radius = random(100,200)
kut = PI/4
brzina = random(1,10)
ubrzanje = 0.005
i = 0

def setup():
    size(600,600)
    background(0)

def draw():
    global kut, brzina, ubrzanje,i,radius

    kut = kut + brzina
    brzina = brzina + ubrzanje
    brzina = constrain(brzina,1,10)

    stroke(255)
    strokeWeight(0.5)
    translate(width/2, height/2)

    x = radius * sin(radians(kut))
    y = radius * cos(radians(kut))
    line(x,0,0,y)

    pushMatrix()
    translate(width/4, height/4)

    x = radius/2 * sin(kut)
    y = radius/2 * cos(kut)
    line(x,0,0,y)

    popMatrix()
    pushMatrix()
    translate(-(width/4), -(height/4))

    x = radius/2 * sin(kut)
    y = radius/2 * cos(kut)
    line(x,0,0,y)

    popMatrix()

    i = i + 1
    saveFrame('shape{}.jpg'.format(i))
```

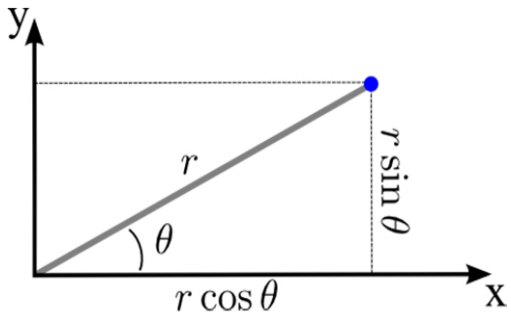
Slika 18. Svjetlucajući oblik - kod

Kako bismo postigli rotaciju, u ovom primjeru koristit ćemo se **SOH-CAH-TOA** metodom računanja sinusa, kosinusa i tangesa (slika 19.).



Slika 19. SOH-CAH-TOA metoda [9]

Ovu metodu primjenjujemo kod definiranja koordinata gdje je potrebno znati radijus i kut (polarno definiranje koordinata) umjesto uobičajenih x,y koordinata (slika 20.).



Slika 20. Polarne koordinate [10]

Slijedeći SOH-CAH-TOA metodu dobit ćemo da su sinus i kosinus:

$$sin\theta = \frac{y}{r} \quad (1)$$

$$cos\theta = \frac{x}{r} \quad (2)$$

, te da je množenjem jednadžbe s radijusom:

$$y = r * sin\theta \quad (3)$$

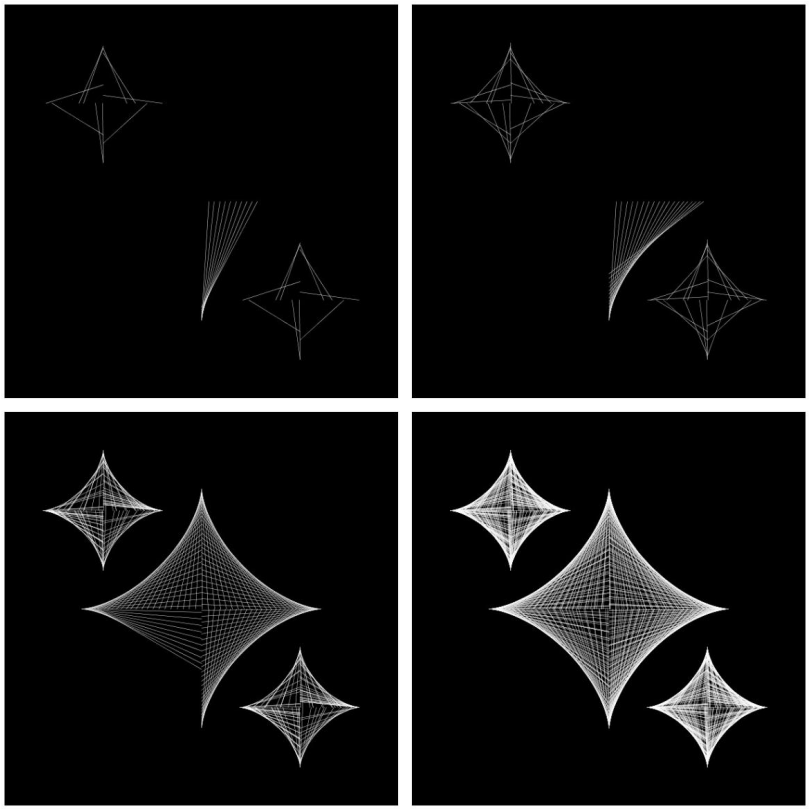
$$x = r * cos\theta \quad (4)$$

U ovom primjeru koristili smo se svojstvima kao što su brzina i ubrzanje kako bi utjecali na samo gibanje linija u animaciji, tj. na mijenjanje samog kuta iz kojeg proizlazi mijenjanje pozicije za crtanje.

Na početku programa definiramo slučajan radijus i brzinu (u željenoj domeni), kut, i ubrzanje. S funkcijom `constrain()` ograničavamo inkrementiranje brzine, tj. najviše može dosegnuti 10, a najmanja može biti 1.

Funkciju `translate()` koristit ćemo tri puta, jednom da bi premjestili koordinatni sustav u sredinu prozora, a potom dva puta da premjestimo u četvrtine po dijagonali, za dva manja svjetlucajuća oblika. Svaki oblik definiramo na isti način, samo dva manja definiramo s upola manjim radijusom.

Premda će, pri ponovnom pokretanju skripte, raspored oblika i ideja biti ista, s mijenjanjem brzine i radijusa animacija i krajnji oblik će ipak ispasti drugačiji (slika 21.).



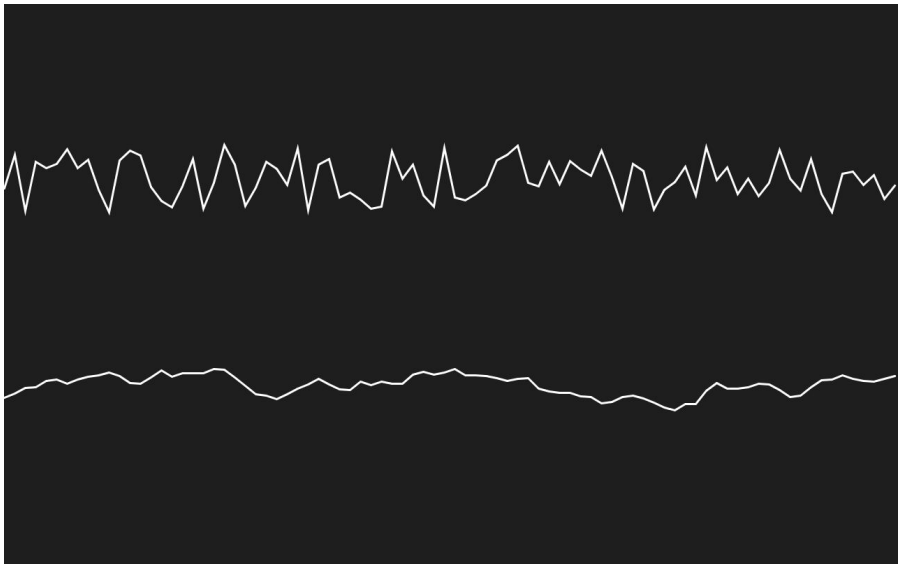
Slika 21. Animacija svjetlucajućih oblika (4/302 spremljena okvira - vlastiti izvor)

3.2.6 Zalazak sunca pomoću Perlinovog šuma

Nadolazeći primjer imitira zalazak sunca koristeći se Perlinovim šumom, matematičkim algoritmom, koji se koristi u računalnoj grafici kako bi se konstruirali različiti tereni, teksture i oblici, najčešće za računalne igre ili filmove.

3.2.6.1 Perlinov šum

Perlinov šum bitan je jer, za razliku od `random()` algoritma, ostvaruje prirodne rezultate, tj. rezultati ne odskaču previše jedan od drugoga, što ovaj algoritam čini pogodnim za imitiranje stvarnih tekstura i oblika, s obzirom na to da u prirodi rjeđe nailazimo na velike oscilacije (slika 22.).



Slika 22. Usporedba `random()` i `noise()` algoritma [11]

Za razliku od `random()` funkcije, koja prima minimalnu i maksimalnu vrijednost, `noise()` funkcija prima 'vrijeme', a broj argumenata ovisi o dimenziji koju želimo postići. Funkcija `noise()` uvijek vraća broj između 0 i 1, te će, u jednoj instanci programa za određeno vrijeme, uvijek davati isti rezultat. Kako bi dobili neku vrstu teksture rezultat ćemo mapirati funkcijom `map()` i/ili kombinirati s valnim funkcijama kao što su sinus i kosinus.

3.2.6.2 Primjer

U ovom primjeru smo dvostrukom primjenom Perlinovog šuma imitirali zalazak na horizontu (slika 23.).

```
index = 0

def setup():
    size(800,800, P3D)
    background(random(230),random(110),random(255))
    strokeWeight(0.3)
    frameRate(1000)

def draw():
    global index
    index = index + 1
    strokeWeight(0.3)

    for y in range (-20, height/2-10,13):
        stroke(random(200,255),random(200),random(100),70)
        drawPerlinLines(y)

    for y in range (height/2 - 20 ,height,13):
        stroke(random(170),random(170),random(210,255),70)
        drawPerlinLines(y)

    saveFrame("perlin_sunset{}.jpg".format(index))

def drawPerlinLines(y):
    x_pom = -50
    y_pom = y
    for x in range(0, width+1,10):

        y_novi = y + noise(x*0.01, y*0.01, frameCount*0.1)*60
        line(x_pom, y_pom, x, y_novi)
        x_pom = x
        y_pom = y_novi

def mousePressed():
    background(random(230),random(110),random(255))
     redraw()
```

Slika 23. Zalazak na horizontu pomoću Perlinovog šuma - kod

U dvije petlje (jedna za zalazak, druga za more) iscrtavali smo skup linija koji počinje udaljen jedan od drugog za 13 px.

Unutar svake petlje pozivamo definiranu funkciju `drawPerlinLines()`, kojoj predajemo trenutnu visinu za iscrtavanje skupa linija (`y`). U toj funkciji prolazimo kroz petlju u rangu širine prozora uzimajući svaki deseti `x`.

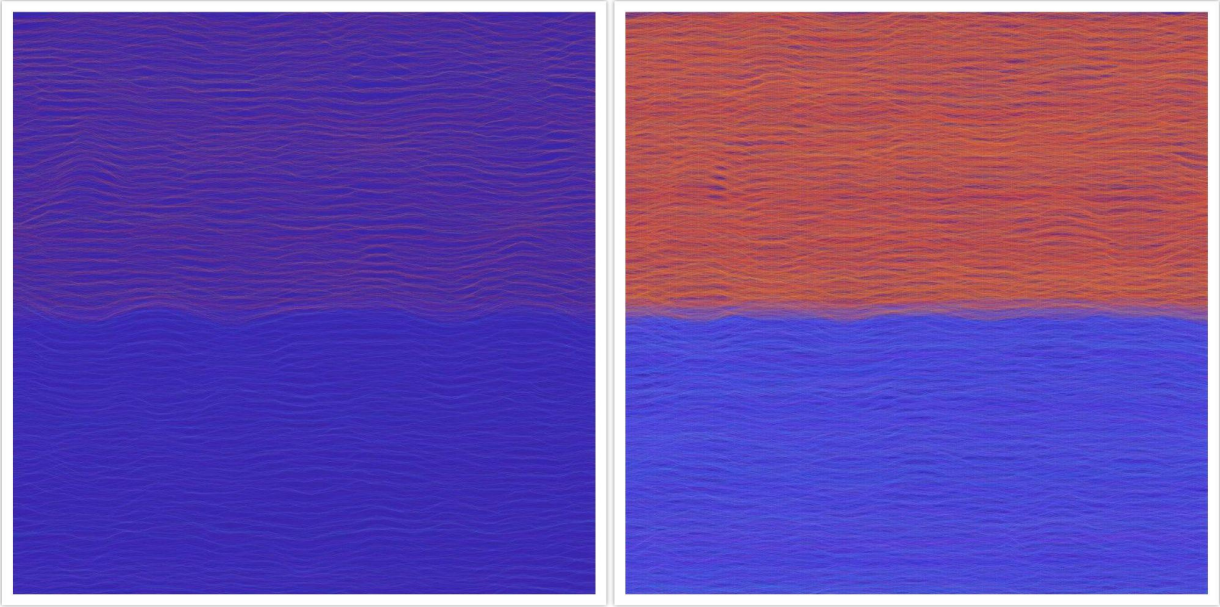
Liniju definiramo tako da su prva dva argumenta (`x_pom` i `y_pom`) pokazatelji na koordinate na kojima završava prethodna linija, treći argument je sami x, a zadnji argument (`y_novi`) definiramo tako da `y`-u (visini iscrtavanja) pribrajamo `noise()` funkciju kojoj predajemo tri argumenta: `x*0.01`, `y*0.01`, `frameCount*0.1`, a sve skupa množimo sa 60 (skaliranje). Argumente množimo s malim, decimalnim brojevima tako da noise funkcija daje što 'glađe' rezultate, a potom je skaliramo kako bismo vidjeli iste, jer bi nam se inače priviđalo da vidimo ravne linije, premda one to nisu. Množenje argumenata je proizvoljno, ali treba biti optimalno, tj. ne smiju se množiti ni s 1, ni s 0.0001 jer bi se tekstura u potpunosti izgubila.

Za lakše razumijevanje možemo ispisivati argumente koje predajemo `line()` funkciji (slika 24.).

```
('x_pom:', 760)
('y_pom:', 823.7894324064255)
('x:', 770)
('y_novi:', 819.198972940445)


('x_pom:', 770)
('y_pom:', 819.198972940445)
('x:', 780)
('y_novi:', 815.7898977994919)


('x_pom:', 780)
('y_pom:', 815.7898977994919)
('x:', 790)
('y_novi:', 815.0928810834885)


('x_pom:', 790)
('y_pom:', 815.0928810834885)
('x:', 800)
('y_novi:', 815.5043659210205)
```

Slika 24. Ispis argumenata tijekom četiri kruga petlje (slika zaslona)

Ako pratimo mijenjanje `y_novi` varijable možemo vidjeti da su oscilacije minimalne te zato dobivamo 'glatke' valove prikazane na slici 25.



Slika 25. Dva okvira animacije zalaska sunca (spremljeni okviri - vlastiti izvor)

3.2.7 Animacija terena pomoću Perlinovog Šuma

U ovom primjeru 'izrađujemo' teren koristeći se Perlinovim šumom, a isti teren, s modifikacijama i unaprjeđenjem, se može primijeniti za izradu računalne igrice (slika 26.).

```
scale_number = 15
w = 1200
h = 900

cols = w/scale_number
rows = h/scale_number

teren = [[] for i in range (cols)]

yoff = 0
move = 0

def setup():
    size(600,600,P3D)
    frameRate(12)
```

Slika 26a. Izrada terena pomoću Perlinovog šuma - kod (1/2)

```
def draw():
    global move,cols,rows,scale_number,w,h,yoff

    #background(135,206,235)
    background(0)

    teren[:]=[[] for i in range (cols)]
    move = move + 0.1
    yoff = move

    for cols_number in range (0,cols,1):
        xoff = 0
        for rows_number in range (0,rows,1):
            teren[cols_number].append(map(noise(xoff,yoff), 0, 1, -130, 130))
            xoff = xoff + 0.1
        yoff = yoff + 0.1

    noFill()
    #noStroke()
    translate(width/2, height/2)
    rotateX(PI/3)
    translate(-width/2, -height/2)

    for y in range (0,rows-1,1):
        beginShape(TRIANGLE_STRIP)
        for x in range (0,cols,1):

            z_index = teren[x][y]
            z_index_next = teren[x][y+1]

            if (z_index < -60):
                stroke(65,105,225)
            if(z_index > - 60 and z_index < 0 ):
                stroke(34,139,34)
            if(z_index > 0 and z_index < 30):
                stroke(207,185,151)
            if(z_index > 30):
                 stroke(255)

            vertex(x*scale_number - 200, y*scale_number, z_index)
            vertex(x*scale_number - 200, (y+1)*scale_number,  z_index_next)

    endShape()
    saveFrame('3dTerrain{}.jpg'.format(frameCount))
```

Slika 26b. Izrada terena pomoću Perlinovog šuma - kod (2/2)

Na početku programa definirali smo broj za skaliranje varijabli (`scale_number`) te širinu i visinu, tj. broj redaka i stupaca koji je obrnuto proporcionalan skaliranju. Također definiramo ugniježđenu listu koja će sadržavati jednak broj podlista koliko i stupaca.

Kako bi kasnije mogli definirati z koordinatu, u `size()` je, kao treći argument, potrebno dodati `P3D` (render za tro-dimenzionalni prostor).

Na početku izvođenja `draw()` funkcije, uvijek čistimo listu `teren[]` kako bismo u nju spremili nove podatke dobivene `noise()` funkcijom.

Varijable `move` i `yoff` služe nam kako bismo imitirali kretanje prostora, tako da ih pri svakoj petlji povećavamo za mali decimalni broj.

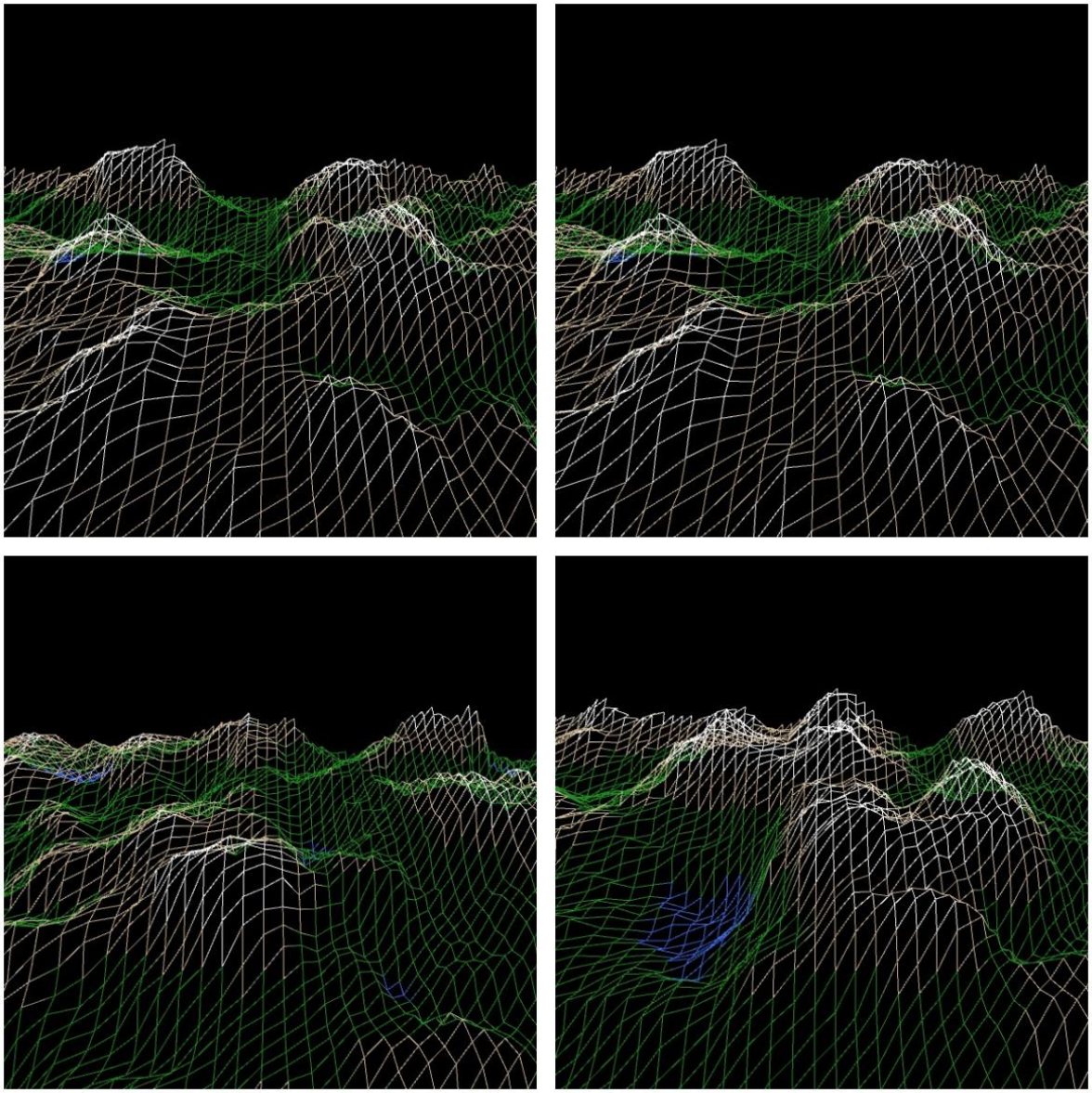
Kako bismo za svaku podlistu liste `teren[]` definirali elemente, za svaku podlistu vršimo funkciju `noise()` onoliko puta koliko ima i redaka. Funkcija `noise()` vrši se na varijablama `xoff` (povećava se prolaskom kroz svaki element podliste) i `yoff` (povećava se svakom petljom programa i svakom idućom podlistom), a potom se mapira u novu domenu [-130,130].

Kako bismo dobili trodimenzionalnu prostornost vršit ćemo premještanje koordinatnog prostora te rotaciju za $\frac{\pi}{3}$.

Teksturu prostora postići ćemo korištenjem `vertex()` funkcije koja nam omogućuje spajanje vrhova na različite načine kako bismo dobili određene oblike. Način na koji spajamo vrhove definiramo u `beginShape()` funkciji koju zovemo prije definiranja vrhova, a pozivanjem `endShape()` završavamo spajanje vrhova. U ovom primjeru koristit ćemo `TRIANGLE_STRIP` način spajanja kako bismo jasnije definirali prostornost, premda bi primjer ostvarivao svoj cilj i koristeći zadan način, a to je spajanje u pravokutnom obliku. Broj oblika ovisi o već postavljenom broju redaka i stupaca, gdje ćemo za svaki element retka (broj stupaca) definirati dva vrha pomoću tri argumenta.

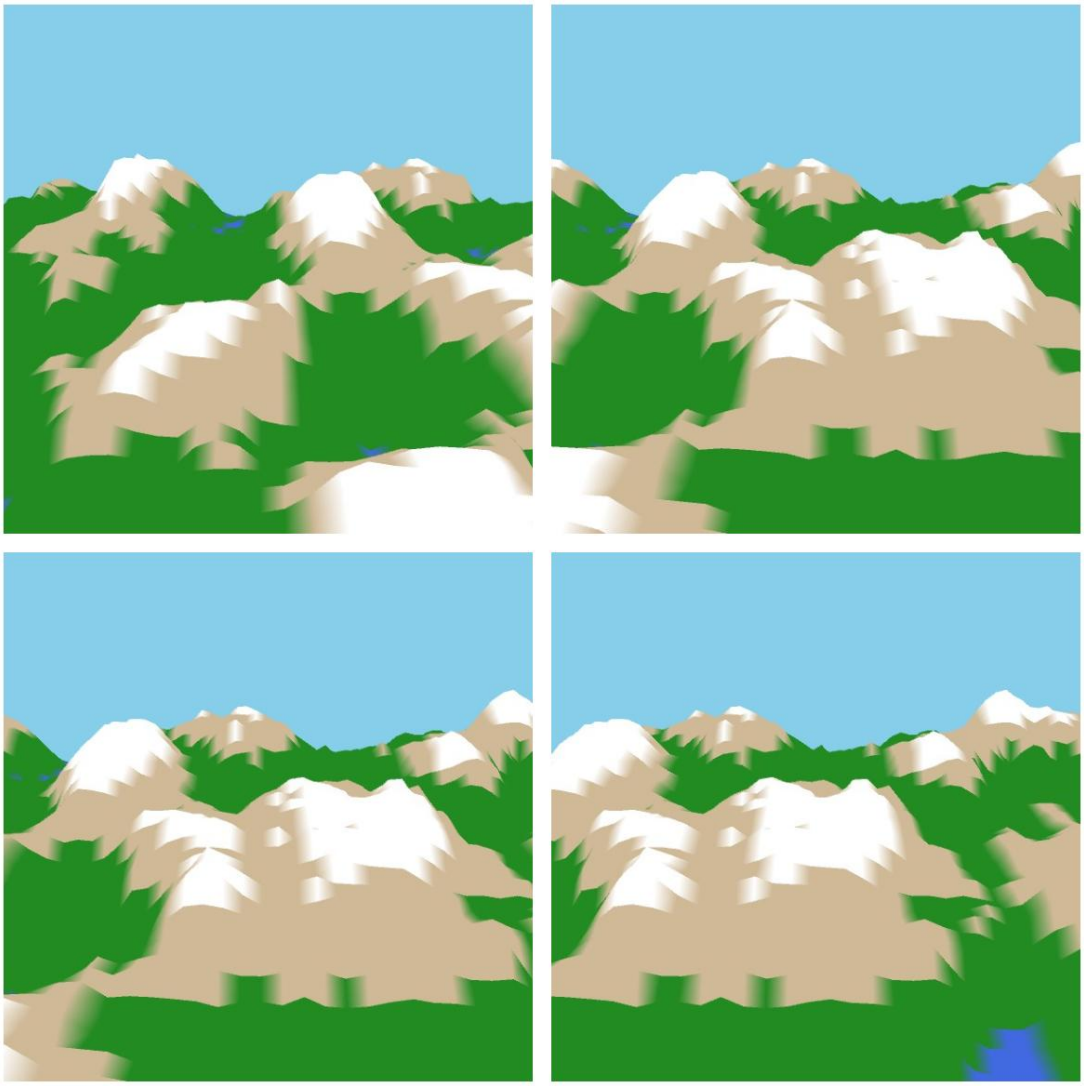
Elemente `x` i `y` množimo sa skalarom kako bismo dobili uvećani prikaz terena. `x` umanjujemo za 200 kako bi teren obuhvatio cijelu širinu ekrana, što je izgubljeno rotacijom koordinatnog sustava. Koordinatu z-osi dobivamo pristupanjem ugniježđenoj listi, čije smo elemente dobili `noise()` funkcijom, te nam upravo taj element daje raznolik, nepredvidljiv teren. Na temelju z

koordinate (`z_index`) definirali smo koje boje će oblici poprimiti, te tako dočarati reljef - mora/jezera, brežuljke i udoline, te vrhove planina pokrivene snijegom (slika 27.).



Slika 27. Animacija terena Perlinovim šumom (4/200 spremljenih okvira)

Ako želimo dobiti realističniji prikaz terena, zamijenit ćemo `stroke()` s `fill()` funkcijom, definirati boju pozadine te ukloniti rubove - `noStroke()` (slika 28.).



Slika 28. Realan prikaz reljefa (4/200 spremljenih okvira - vlastiti izvor)

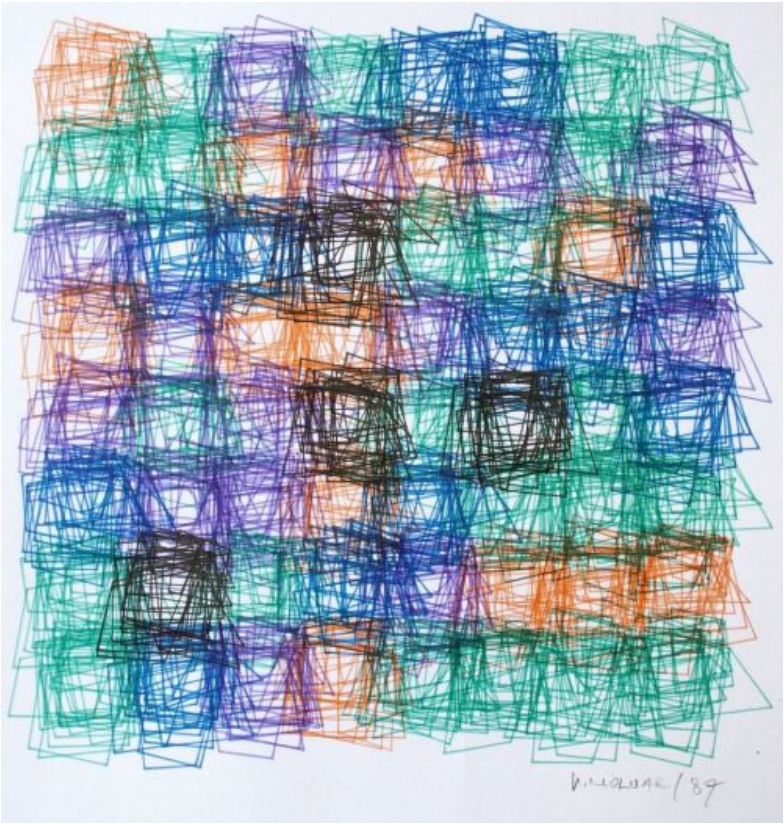
Važno je napomenuti da su varijable u ovom, a i drugim primjerima definirane po vlastitoj volji, a mijenjanjem varijable kao što su `scale_number` ili domene mapiranja, možemo utjecati na to koliko će 'glatko' izgledati rezultat.

3.3 Processing.py - Postojeći primjeri generative umjetnosti

3.3.1 Rekreacija algoritma Vere Molnar iz 1986. godine

Vera Molnar jedna je od poznatijih generativnih umjetnica prošlog stoljeća koja se bavila smišljanjem algoritama i prije nego je dobila pristup laboratorijskim računalima. Izjavila je da su njen život kvadrati, trokuti i linije jer su njeni radovi kombinacija geometrijskih oblika.

Ovim primjerom rekreirano je poznato djelo Vere Molnar iz serije radova *Structure de Quadrilatères* (slika 29.).



Slika 29. *Structure de Quadrilatères* - Vera Molnar [12]

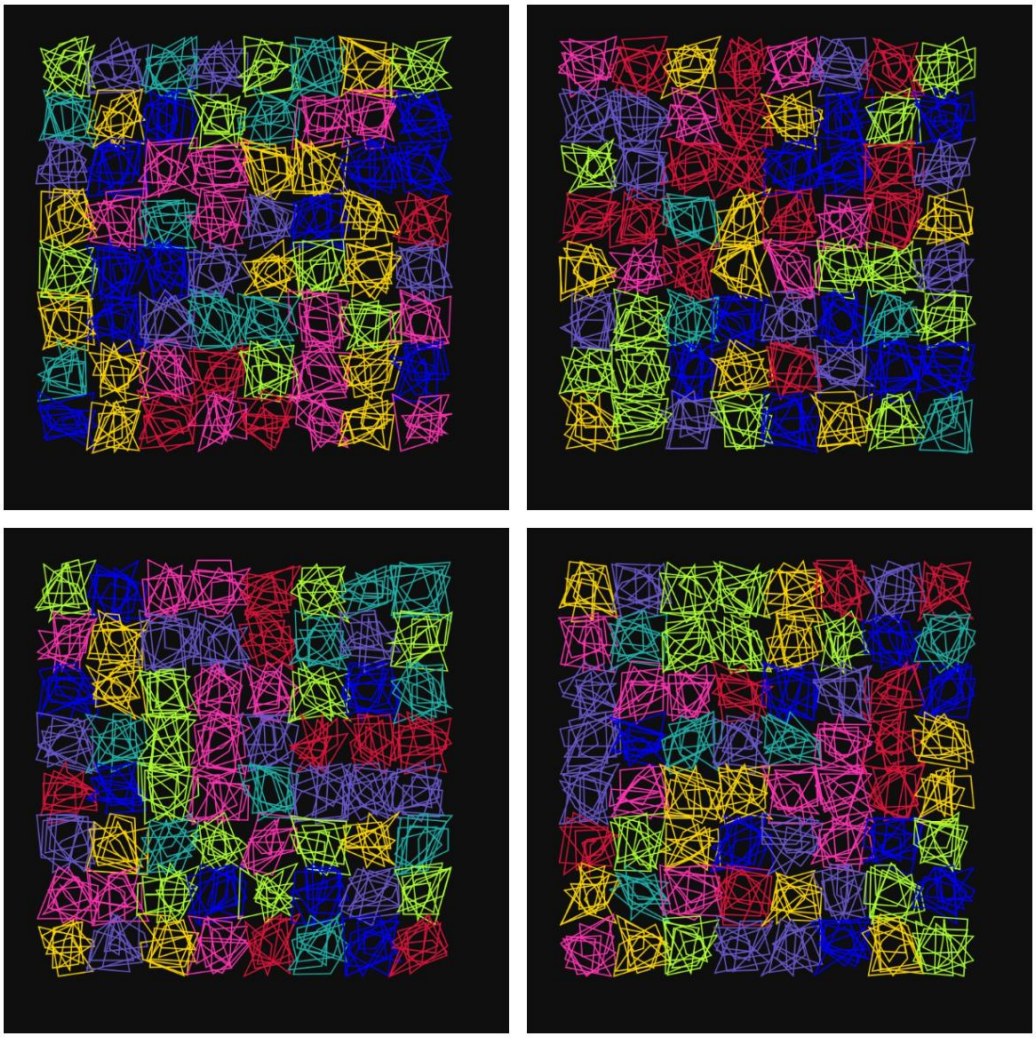
Rekreacija algoritma izvršena je u *Processing.py*-u uz određene preinake i samovoljan odabir određenih parametara (slika 30.).

```
lista_boja = [ '#0000FF', '#DC143C', '#FFD700', '#ADFF2F', '#20B2AA',
'#6959CD','#FF34B3' ]

def setup():
    size(800,800)
    frameRate(10)
    noLoop()

def draw():
    background(15)
    noFill()
    strokeWeight(2)

    for i in range(100, width-100, 80):
        for j in range(100,height-100, 80):

            boja = lista_boja[int(random(7))]
            for x in range (0,7):
                stroke(boja)
                pushMatrix()
                translate(i,j)
                quad(-random(50),-random(50),random(50),-random(50),random(50),
                     random(50),-random(50),random(50))
                popMatrix()
    saveFrame('scrabble{0}.jpg'.format(frameCount))

def keyPressed():
    redraw()
```

Slika 30. Rekreacija algoritma Vere Molnar - kod

U algoritmu je definirano sedam različitih boja napisanih u heksadecimalnom obliku. Pri svakom crtanju, tj. na svaki pritisak miša vrši se crtanje 8x8 matrice (u ovom slučaju), tako da je u svakom polju matrice nacrtano 7 različitih oblika iste boje, koja se bira iz liste boja za svako polje matrice. Oblici cu nacrtani pomoću `quad()` funkcije koji prima 8 parametara, tj. koordinate svake točke, s obzirom na to da služi za crtanje nepravilnih četverokutnih oblika. Za svaki par točaka kod te funkcije određena je drugačije kombinacija predznaka (+ i -) kako bi oblici ipak zauzimali djelomično kvadratast oblik jer je svaka od četiri točke ipak na svom kraju.

Na slici 31. možemo vidjeti više rezultata koje nam daje opisani algoritam.



Slika 31. Četiri okvira rekreiranog algoritma Vere Molnar (spremljene slike - vlastiti izvor)

3.3.2 *Color smoke*

U ovom primjeru prikazujemo animaciju koja se odvija pomoću već objašnjenog Perlinovog šuma, a istu animaciju napravio je korisnik imena **lsura** u *P5.js* modu, čiji kod možemo vidjeti na sljedećem izvoru [13]. U sljedećoj skripti, 'preveli' smo kod u *Processing.py* modul (slika 32.).

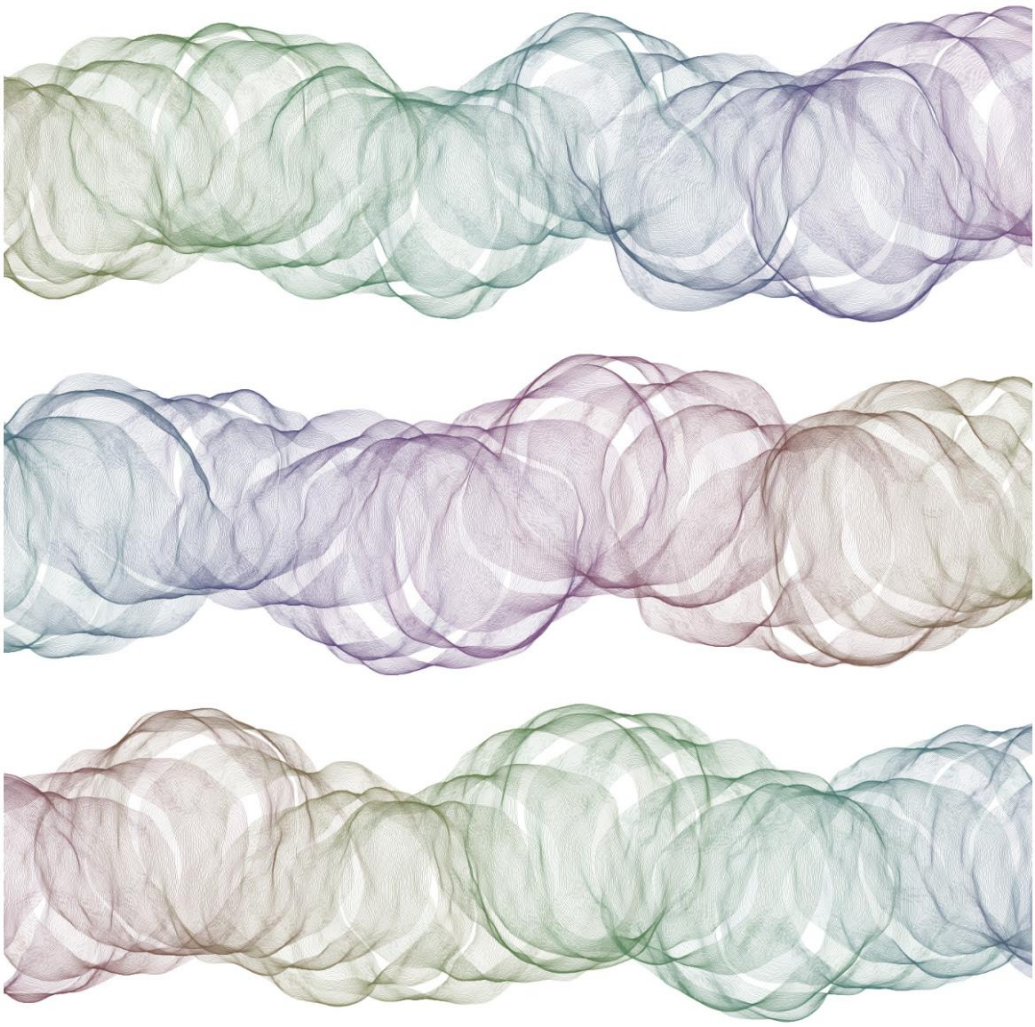
```
np = 300
startcol = random(255)
sx,sy = 0,0

def setup():
    size(1600,800)
    background(255)
    noFill()
    noiseSeed(int(random(1000)))
    frameRate(1000)
    noLoop()

def mousePressed():
    loop()

def mouseReleased():
    noLoop()

def draw():

    beginShape()

    for i in range(0,np):
        angle = map(i,0,np,0, TWO_PI)
        cx = (frameCount * 2) - 200
        cy = (height / 2)  + (50 * sin(frameCount /50))

        xx = 100 * sin(angle + (cx/10))
        yy = 100 * cos(angle + (cx/10))

        v = PVector(xx,yy)

        xx = (xx + cx) /150
        yy = (yy + cy) /150

        v.mult(1 + (1.5 * noise(xx,yy)))
        vertex(cx + v.x, cy + v.y)
        if (i == 0):
            sx = cx + v.x
            sy = cy + v.y
```

Slika 32a. *Color Smoke* u *Processing.py*-u - kod (1/2)

```
colorMode(HSB)
    hue_var = cx / 10 - startcol

    if (hue_var < 0):
        hue_var = hue_var + 255
    stroke (hue_var, 100, 120)
    strokeWeight(0.1)
    vertex(sx,sy)
    endShape()


    if(frameCount > (width + 500)):
       noLoop()

def keyPressed():
    save('ColorSmoke{0}.jpg'.format(random(100)))
```

Slika 32b. *Color Smoke* u *Processing.py*-u - kod (2/2)

Varijable `cx` i `cy` predstavljaju koordinate središta oblika, a kako bi stvorili privid gibanja, zadana je formula koja ovisno o promjenjivom parametru (`frameCount`) mijenja i koordinate središta svakog novog, nacrtanog oblika. Varijable `xx` i `yy` mijenjat će se u odnosu na kut, središte i trigonometrijske funkcije (sinus ili kosinus), a potom će se spremiti u vektor koristeći funkciju `PVector()`. Na vektoru vršimo multipliciranje sa skalarom, kojeg dobivamo korištenjem `noise()` funkcije, te tako postižemo da vrhovi budu raspoređeni u valovitom obliku. Kako bi nam bilo jasnije gdje su raspoređeni vrhovi koji stvaraju oblik, u `beginShape()` funkciju uvijek možemo dodati parametar `POINTS` i tako, umjesto povezanog oblika, vidjeti pojedine vrhove koji sačinjavaju taj oblik. Svaki oblik poprimit će drugu boju, ali prelaz između njih biti će gladak jer se koristeći HSB (*hue-saturation-brightness)* modom, u ovom primjeru zadanom formulom, mijenja samo nijansa (*hue*) (slika 33.).



Slika 33. Tri primjera animacije *Color Smoke* (spremljeni okviri - vlastiti izvor)

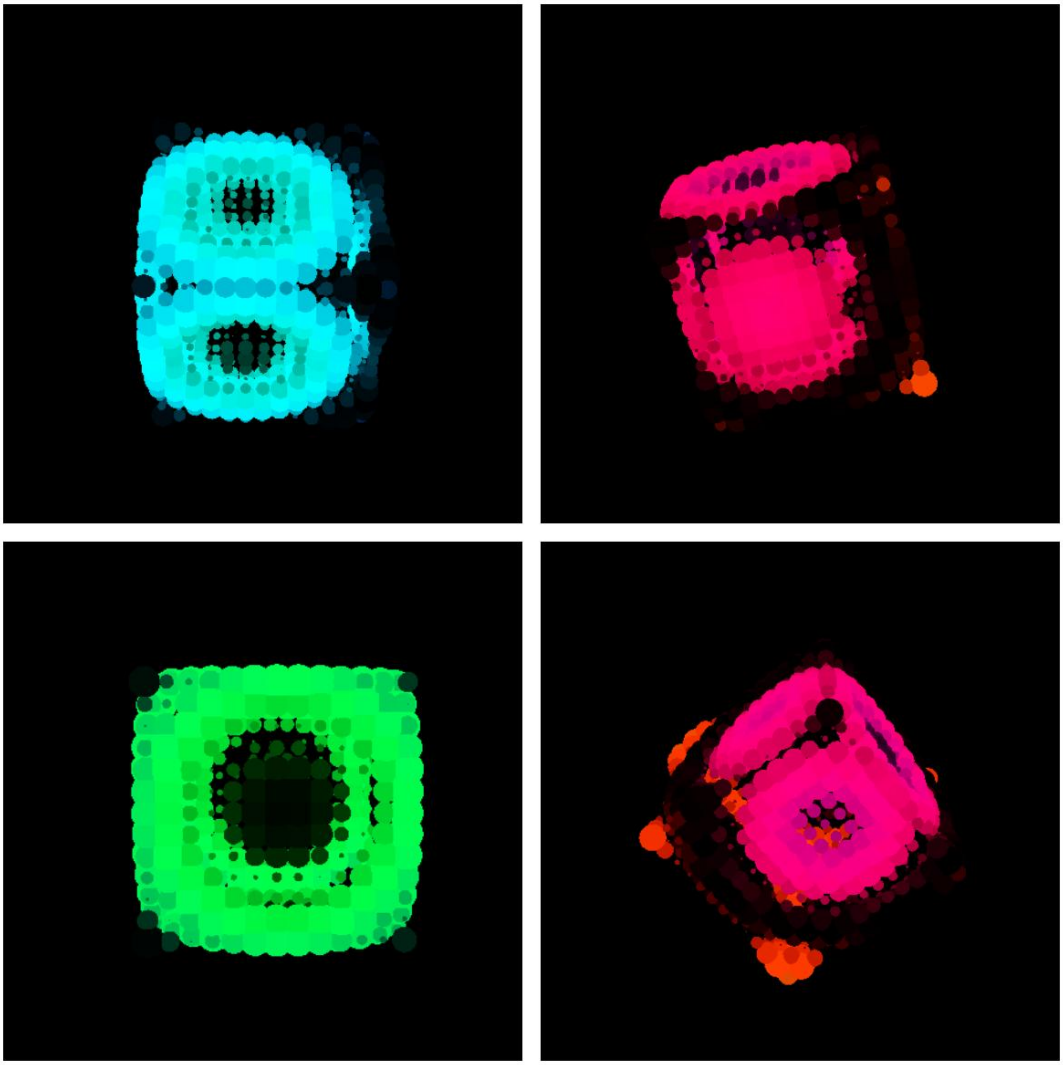
3.3.3 *Mutable ripples*

U ovom primjeru prikazat ćemo interaktivnu animaciju, koju je izradio korisnik Jason Labbe [14], ali rekreiranu u *Python* modu. Animacija predstavlja kocku, sačinjenu od sfera, tj. šupljih kugla, koja se nalazi u 3D prostoru. Kocka se rotira s pomicanjem miša, a nijanse se konstantno mijenjaju, što je kao i u prošlom primjeru, postignuto ovisnošću o neprekidno promjenjivoj varijabli `frameCount`. Ovisno o vrijednosti valne funkcije, mijenja se i svjetlina pojedinih kugla, što se postiže funkcijom `map()` (slika 34.).

```
cubeSize = 150
blockSize = 12

def setup():
  size(800, 800, P3D)
  colorMode(HSB, 255)

def draw():
  background(0)
  noStroke()

  translate(width / 2, height / 2, 200)
  rotateX(radians(map(mouseY, 0, height, 180, -180)))
  rotateY(radians(map(mouseX, 0, width, -180, 180)))

  noiseMult = 0.01
  nx = noise(frameCount * noiseMult) * cubeSize
  ny = noise(1000 + frameCount * noiseMult) * cubeSize
  nz = noise(5000 + frameCount * noiseMult) * cubeSize

  maxCount = cubeSize - (cubeSize % blockSize)
  for x in range(0, cubeSize+1, blockSize):
    for y in range(0, cubeSize+1, blockSize):
      for z in range(0, cubeSize+1, blockSize):
        if((x != 0) and (x != maxCount) and (y != 0) and (y != maxCount) and
          (z != 0) and (z != maxCount)): continue

        multi = dist(x, y, z, nx, ny, nz)
        offset = (frameCount + multi) * 0.1
        wave = sin(offset)

        pushMatrix()
        translate(x - cubeSize / 2, y - cubeSize / 2, z - cubeSize / 2)
        scale(wave)

        colorHue = (frameCount + multi * 0.5) % 255
        colorBright = map(wave, -1, 1, 255, 0)
        fill(colorHue, 255, colorBright)

        sphere(blockSize * 0.8)
        popMatrix()
```

Slika 34. *Mutable ripples* (Jason Labbe) rekreacija u *Processing.py*-u - kod

U nastavku možemo vidjeti kako izgleda mreškanje i pomicanje kocke u četiri različita okvira (slika 35.).



Slika 35. *Mutable ripples* - četiri okvira animacije (spremljeni okviri - vlastiti izvor)

3.3.4 Animacija stabala u zimskom okruženju

U ovom primjeru prikazat ćemo stvaranje stabala u zimskom okruženju, prateći kod Roberta D'Arcy-a zasnovan prema originalnom primjeru [15], rekreiran u *Processing.py*-u (slika 36.).

```
isDrawing = False;
numTrees = 0;

def setup():
  # size(1400, 750)
  fullScreen()
  smooth()
  drawBG()

def draw():
  global numTrees
  if (isDrawing == False and numTrees <= 70):
    numTrees = numTrees + 1
    sze = random(50, 250)
    branch(random(-50, width+50), height+5, -PI/2, sze, sze/10, 0)

def branch(startX, startY, ang, leng,  wt,  col):
  isDrawing = True
  numSegs = 0
  if (leng > 50):
    numSegs = 10
  elif(leng > 30 and leng <= 50):
    numSegs = 5
  elif (leng <= 30):
    numSegs = 1
  segLen = leng / numSegs
  nextSegWt = wt * 0.6

  col = col + 10

  for i in range(numSegs):
    varyAng = random(-PI/2, PI/2) * 0.05
    varySegLen = random(0.5, 1.5)
    endX = startX + cos(ang + varyAng) * (segLen *  varySegLen)
    endY = startY + sin(ang + varyAng) * (segLen *  varySegLen)
    segWeight = map(i, 0, numSegs, wt, nextSegWt)

    drawBranchSeg(startX, startY, endX, endY, segWeight, col)

    startX = endX
    startY = endY
```

Slika 36a. Stabla u zimskom okruženju - kod (1/3)

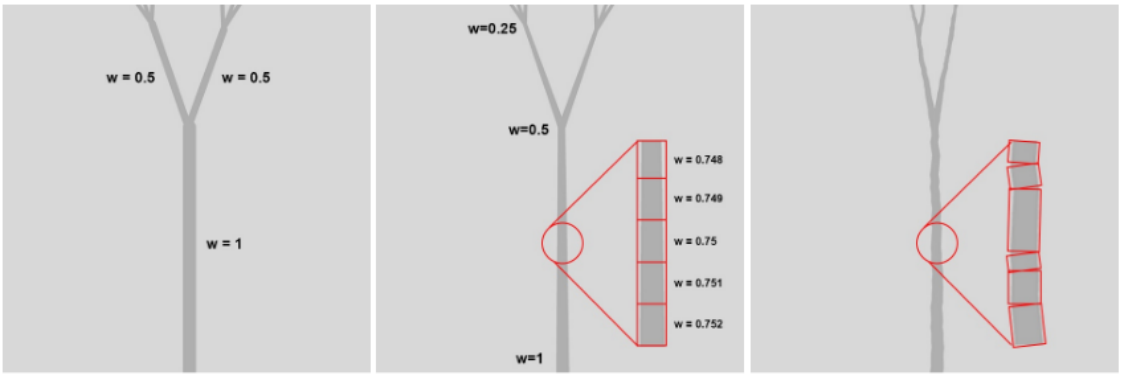
```
if (leng > 10):
    leng = leng * .66

    if (leng > 30):
      branch(startX, startY, ang + random(-PI/6, 0), leng, nextSegWt, col)
      branch(startX, startY, ang + random(PI/6), leng, nextSegWt, col)
    elif (leng <= 30):
      numBranches = int(random(2, 4))
      for i in range(numBranches):
        newAng = ang + random(-PI/4, PI/4)
        branch(startX, startY, newAng, leng, nextSegWt, col)

  isDrawing = False;

def drawBranchSeg( startX,  startY,   endX, endY,  wt,  col):

  leng = sqrt(sq(endX - startX) + sq(endY-startY))

  if (wt < 3):
    strokeWeight(wt)
    stroke(col+((180-col)/2), 200)
    line(startX, startY, endX, endY)
  elif (wt >= 3):
    pushMatrix()
    translate(startX, startY)
    rotate(-PI/2 + atan2(endY-startY, endX-startX))

    incr = (220 - col)/wt
    for i in range(int(wt)+1):
      x1 = -wt/2 + i
      y1 = 0
      x2 = -wt/2 + i
      y2 =  leng
      stroke(col + i * incr)
      strokeWeight(1)
      line(x1, y1, x2, y2)

    noStroke()
    fill(0)
```

Slika 36b. Stabla u zimskom okruženju - kod (2/3)

```
for j in range(int(wt)*2):
      x = random(-((wt/2)-(wt/20)), (wt/2)-(wt/20))
      y = random(leng)

      if (random(1) < 0.2):
        ellipse(x, y, wt * random( 0.1, 0.15), wt * random(0.1, 0.15))
      else:
        ellipse(x, y, wt * random(0.005, 0.1), wt * random (0.005, 0.1))

    popMatrix()

def drawBG():
  c1 = color(200)
  c2 = color(245)
  noFill()

  for i in range(height):
    stroke(lerpColor(c1, c2, i/height))
    rect(0, i, width, 1)

def mousePressed():
  drawBG()
  numTrees = 0
  redraw()
```

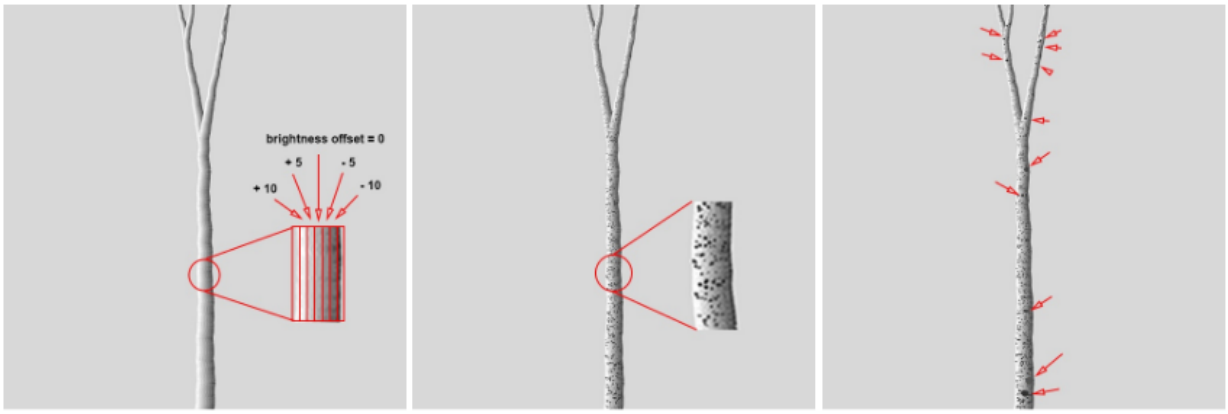
Slika 36c. Stabla u zimskom okruženju - kod (3/3)

Primjer se svodi na rekurzivno crtanje grančica, čime se, kao finalan produkt, dobiva izgled stabla. Ideja jest da svaki idući par grančica bude tanji i kraći, ali kako bi se zadobio realan izgled, svaka grančica je podijeljena u više segmenata (broj segmenata proporcionalan dužini grane), a svakom od tih segmenata bude dodijeljena različita debljina te mala rotacija (slika 37.).



Slika 37. Proces crtanja osnove stabla [15]

U funkciji `drawBranchSeg()` svaki pojedini segment činimo realnijim igrajući se s rastavljanjem na više vertikalnih linija koje će nam predstavljati gradijent te dočarati igru svjetla u prirodi. Također, slučajno ćemo raspodijeliti krugove različitih debljina kako bi dočarali teksturu stabla (slika 38.).



Slika 38. Rekreiranje prirodnog svjetla i teksture stabla [15]

Broj nacrtanih stabala određen je varijablom `numTrees`, a stabla se crtaju po segmentima kako ne bi došlo do prevelikog opterećenja programa. Klikom miša dodavamo iduću grupu stabala. Dobiveni rezultat možemo vidjeti na slici 39.

Slika 39. Crtež dobiven pokretanjem skripte za crtanje stabala (snimka zaslona)

3.3.5 *Brush drawing*

Primjer Olivera Brotherhood-a [16], rekreiran u *Processing.py*-u, predstavlja interaktivnu animaciju crtanja slike, koja je u stvari postepeno otkrivanje pozadinske slike (slika 40.).

```
balls = []
radiusLow = 10
radiusHigh = 20
rangeLow = .5
rangeHigh = 1

def setup():
  global img
  img = loadImage("vangogh.jpg")
  size(800,993)
  background(0)
  textAlign(CENTER)
  text("CLICK AND HOLD! :)", width/2, height/2)

def draw():
   for i in range(len(balls)):
      balls[i].drawBrush()
      balls[i].update()
      balls[i].changeColour()
      if(balls[i].radius<0):
          balls[i].radius = 0.2

  if (mousePressed):
      for i in range(5):
          balls.append(Ball(mouseX, mouseY,
          color(img.get(mouseX+int(random(2)), mouseY+int(random(2))))))
class Ball:
      def __init__(self, mX, mY,c):
          self.location = PVector(mX,mY)
          self.radius = random(10,20)
          self.r = red(c)
          self.g = green(c)
          self.b = blue(c)
          self.xOff = 0.0
          self.yOff = 0.0

def update(self):
          self.radius =  self.radius - random(0.15, 0.25)
          self.xOff = self.xOff + random(-.5, .5)
```

Slika 40a. Animacija crtanja slike - kod (1/2)

```
        self.nX = noise(self.location.x) * self.xOff

        self.yOff = self.yOff + random(-.5, .5)
        self.nY = noise(self.location.y) * self.yOff

        self.location.x = self.location.x + self.nX
        self.location.y = self.location.y + self.nY

    def changeColour(self):

        self.c = color(img.get(int(self.location.x), int(self.location.y)))
        self.r = red(self.c)
        self.g = green(self.c)
        self.b = blue(self.c)

    def drawBrush(self):

        noStroke()
        fill(self.r, self.g, self.b)
        ellipse(self.location.x, self.location.y, self.radius, self.radius)
```

Slika 40b. Animacija crtanja slike - kod (2/2)

Klasa `Ball` definirana je parametrima poput lokacije, boje, pomaka (engl. *offset*) te šuma. U sebi sadrži tri metode kao što su `update()`, `changeColour()` i `drawBrush()`. Klikom miša na ekran dodavamo novu instancu klase `Ball` u listu `balls[]`, iz koje, u `draw()` funkciji, izvlačimo svakog člana, nad kojim onda vršimo gore navedene metode definirane u klasi.

U `update()` metodi mijenja se lokacija te se sama 'loptica' postepeno smanjuje. Lokacija se mijenja primjenom šuma, kako bi dobili neočekivanu putanju, ali bez velikih odstupanja.

Funkcija `changeColour()` jednostavno izvlači boju pozadinske slike s trenutne lokacije koristeći `get()` metodu, a onda mijenja boju kuglice tako što joj pridoda RGB vrijednosti spremljene boje.

Funkcija `drawBrush()` izvršava samo crtanje loptice, tj. kruga, zadajući boju, radijus i lokaciju.

Na pritisak miša vrši se stvaranje pet novih instanci kuglica, te se iz te pozicije nastavlja širiti, tj. stvarati ako držimo miš pritisnutim (slika 41.).

Slika 41. Brush Drawing dio rezultata skripte (snimka zaslona)

4 DUBOKO UČENJE NA PODRUČJU GENERATIVNE UMJETNOSTI

Premda je strojno učenje osmišljeno kako bi stroj, učeći se na podacima, klasificirao iste ili donosio određena predviđanja, u zadnjih par godina, okrenulo se k sferama poput stvaranja, imitiranja i slično.

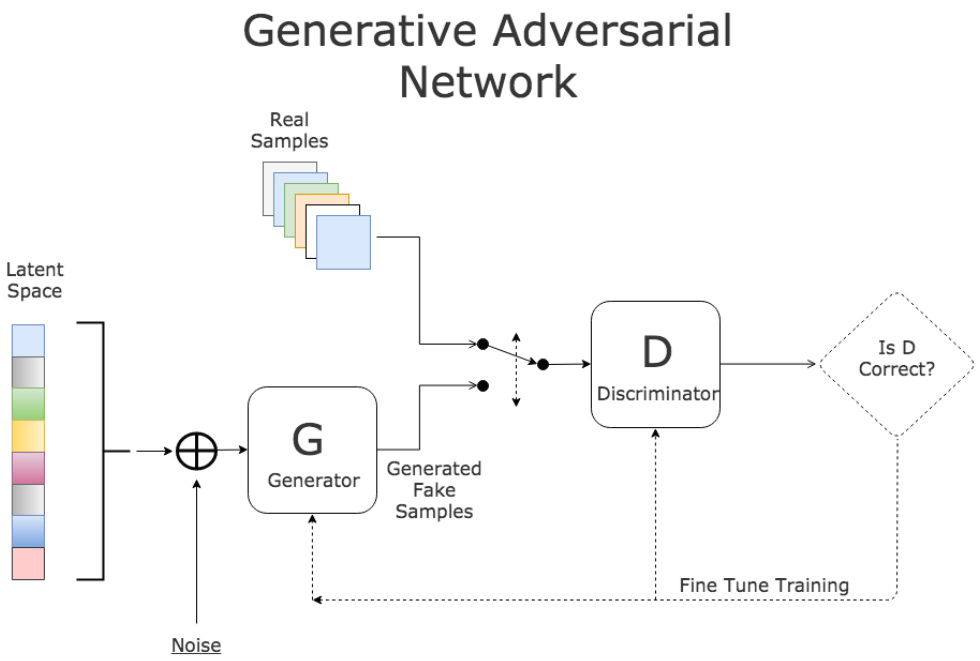
Jedna od prvih takvih aplikacija jest *FaceApp*, koja služi modificiranju fotografije tako što generira značajke poput brade, kose, osmijeha, i slično. Sličnu stvar možemo vidjeti i na stranici *This Person Does Not Exist* [17], gdje pri svakom ponovnom pokretanju stranice dobijamo strojno generirano lice osobe koja zapravo ne postoji.

Stvaranjem prethodno navedenih programa pokrenulo se pitanje o tome možemo li učiniti strojeve kreativnima, tj. ako su strojevi sposobni napraviti sliku, ton, riječi, jesu li sposobni napraviti i umjetničku sliku, smislenu sonatu i poemu [18].

Navedeni primjeri su proizašli koristeći se GAN (*Generative Adversarial Network*) modeliranjem.

4.1 GAN

GAN predstavlja novu arhitekturu u području strojnog učenja, osmišljenu od strane Iana Goodfellow-a i njegovih kolega sa sveučilišta u Montrealu 2014. godine. GAN je okarakteriziran kao način treniranja neuronske mreža bez nadzora, koji se odlikuje boljim performansama u odnosu na tradicionalne načine. GAN sadrži dvije odvojene mreže koje se međusobno ponašaju kao protivnici, a zovu se Generator (G) i Diskriminator (D). Dijagramom na slici 42. prikazana je osnova GAN-ova, gdje je ključno da je Diskriminator onaj koji prolazi obuku te potom vrši proces klasifikacije, dok je Generator onaj koji generira nasumične uzorke koji su nalik pravim i tako pokušava ‘podvaliti’ Diskriminatoru da donese krivu odluku [19].



Slika 42. GAN dijagram procesa [19]

Cilj ovog procesa jest unaprijediti obje neuronske mreže. U slučaju da Diskriminator pogodi, Generator se usavršava tako da generira realnije uzorke, a u suprotnom je Diskriminator onaj koji se usavršava tako da u buduće bolje prepozna razliku između pravog i lažnog uzorka, učeći se na dosadašnjim greškama. Proces se odvija sve dok se ne postigne ravnoteža u kojoj je trening Diskriminatora optimiziran.

4.1.1 Fréchet Inception Distance mjera (F.I.D.)

F.I.D. predstavlja mjernu vrijednost koja se dobiva računanjem udaljenosti vektora stvarne i generirane slike. Rezultat nam govori koliko su statistički slične značajke dvaju sirovih slika, a dobije se korištenjem modela *v3* za klasifikaciju slika. Niži rezultat nam govori da su dvije grupe slika sličnije, tj. da bi u slučaju savršene podudaranosti rezultat bio 0,0.

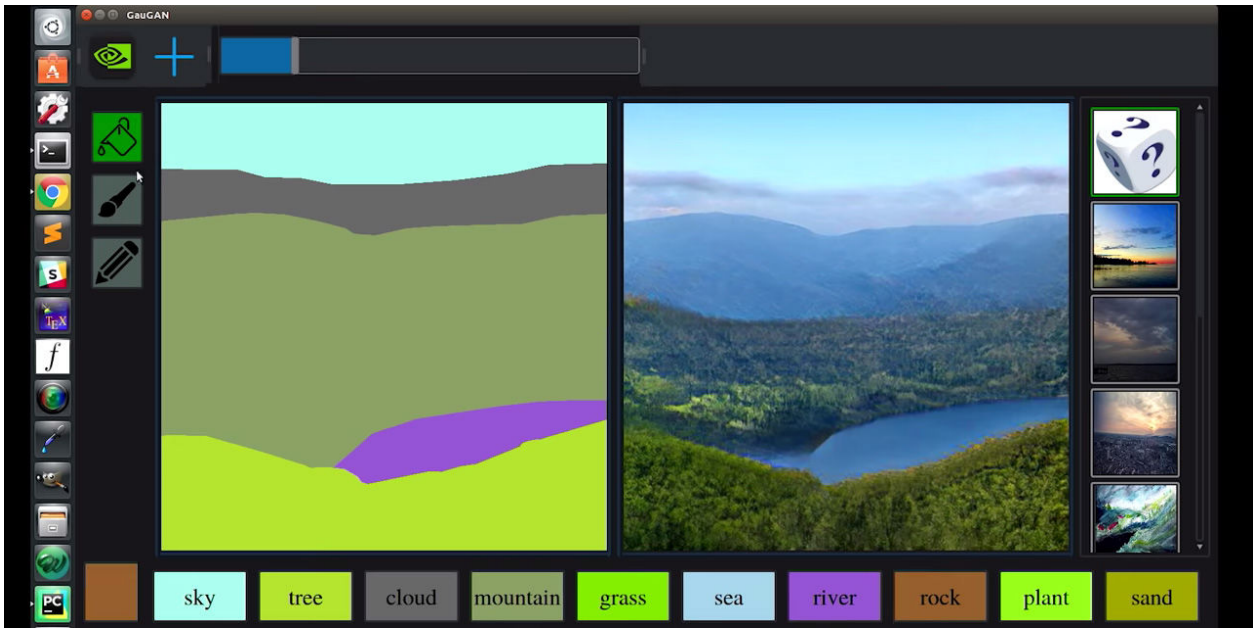
$$FID(x, g) = ||\mu_x - \mu_g||^2 + T_r(\Sigma_x + \Sigma_g - 2\sqrt{\Sigma_x * \Sigma_g}) \quad (5)$$

μ_x i μ_g predstavljaju srednje vrijednosti značajki stvarnih i generiranih slika, dok Σ_x i Σ_g predstavljaju matrice kovarijanci za značajke stvarnih i generiranih slika. T_r predstavlja operaciju iz linearne algebre, koja znači zbrajanje elemenata po dijagonali.

4.1.2 Poznati primjeri GAN-ova

Osim već prethodno navedena dva primjera GAN uporabe, postoji još nekolicina njih kojih će nam biti poznati, pogotovo ako se u slobodno vrijeme bavimo fotografijom, videoprodukcijom ili pratimo nove trendove u svijetu aplikacija i zabave.

GauGAN je tehnologija 'pametnog kista' osnovana od strane *NVIDIA Research*-a 2019. godine, koristeći *PyTorch*. Omogućuje korisnicima da potezima 'kista' rade svoje segmentacijske mape (npr. pijesak, more, nebo), koje se u stvarnom vremenu pretvaraju u fotorealistične prikaze istih segmenata (slika 43.).



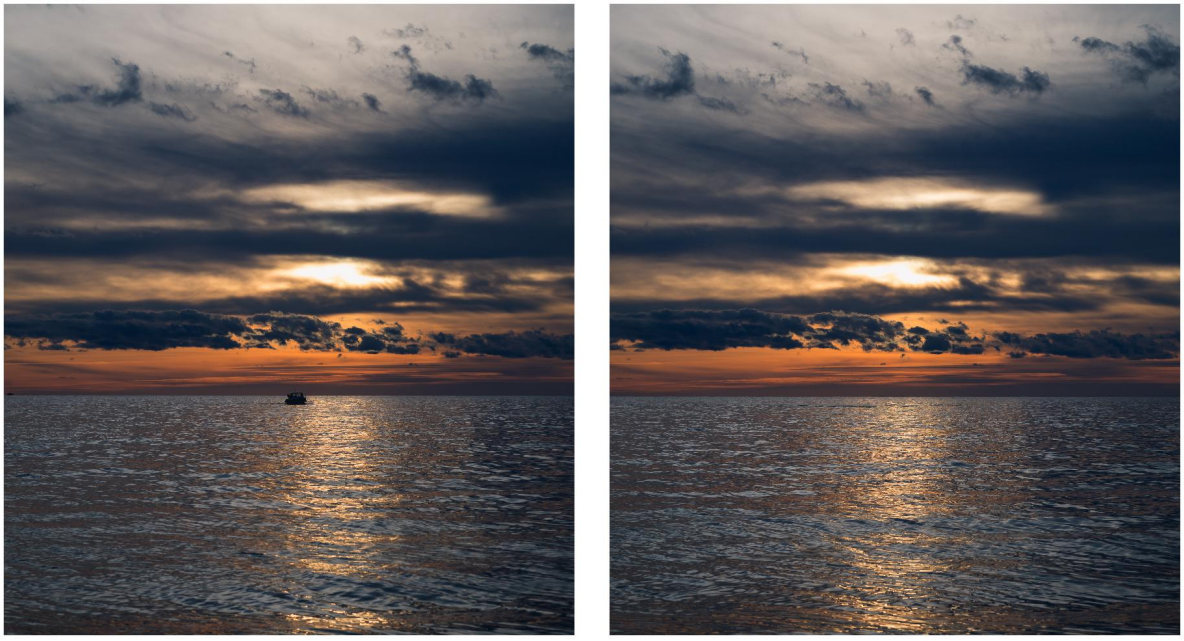
Slika 43. GauGAN primjer korištenja [20]

Koristeći duboko učenje, mijenjanjem jednog segmenta slike, mijenja se i ostatak scenarija. Ako napravimo jezero, vidit ćemo i refleksiju okolnih elemenata u njemu, a kada promijenimo travu u snijeg, scenarij cijele slike će se pretvoriti u zimsku idilu.

Osim popunjavanja tekstura i detalja, aplikacija omogućuje i preuzimanje umjetničkog stila nekog slikara na sliku te mijenjanje doba dana, čime se automatski mijenja i izgled same slike.

GauGAN nadovezuje se na saznanja o *Pix2Pix* sustavima za mapiranje ulaznih slika u nove, izlazne slike. Takav sustav opisan je u idućem potpoglavlju.

Pri uređivanju fotografija, bilo to da se maknu objekti koji odvlače pozornost ili da se popuni praznina pri namještanju perspektive ili skaliranja fotografije, česta je upotreba ***Content-Aware Fill*-a**, koji se s godinama unaprjeđivao uvodeći GAN kao metodu treniranja. Na slici 44. Možemo vidjeti gdje je pomoću *Adobe Photoshop*-a te *Content-Aware Fill* funkcije, neprimjetno maknuta barka s fotografije.



Slika 44. Prije i poslije Content-Aware Fill funkcije (vlastita fotografija i izrada)

Pozadinsko matiranje (engl. *Background matting*) omogućuje nam razdvajanje slike na pozadinu i subjekt, tj. prednji plan. Prednost matiranja naspram obične segmentacije jest u alfa vrijednosti. Kod obične segmentacije za svaki piksel se dodjeljuje binarna vrijednost, tj. 0 ili 1, a svaki piksel je izražen kao linearna kombinacija pozadinskih i prednjih boja:

$$I_i = \alpha_i F_i + (1 - \alpha_i) B_i, \quad \alpha_i \epsilon [0, 1] \tag{6}$$

Skalar α_i predstavlja alfa vrijednost, tj. prozirnost piksela, F_i predstavlja (engl. *Foreground*) prednju vrijednost i-tog piksela te B_i predstavlja vrijednost pozadinskog piksela (engl. *Background*).

Međutim, rubni pikseli često su 'miješani' te dodjeljivanjem 0 ili 1 tim pikselima može doći do defekata pri odjeljivanju pozadine i subjekta, stoga je α_i ključan u donošenju odluke.

Sustavi koji se zasnivaju na matiranju, poput *AlphaGAN*-a, koriste GAN kako bi poboljšali rezultate odjeljenja subjekta od pozadine. Generator je konvolucijska mreža enkodera i dekodera koja prima sliku sastavljenu od prednjeg plana, alfe, slučajne odabrane pozadine s trimapom (slika rastavljena na tri sloja - pozadinu, prednji plan te nepoznati prostor) kao četvrtim ulaznim kanalom te pokušava predvidjeti istinu alfu. Diskriminator pokušava razlikovati stvarne 4-kanalne ulaze i lažne ulaze, gdje su prva tri kanala sastavljena od prednjeg plana, pozadine i predviđene alfe [21].

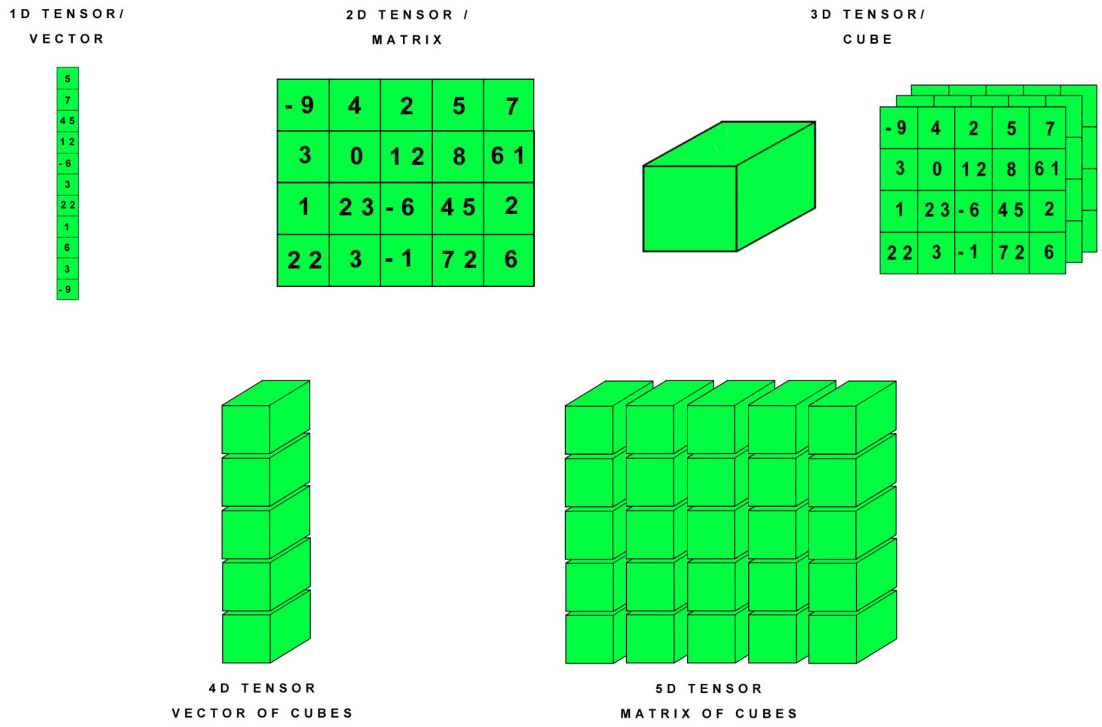
4.2 Python i duboko učenje u generativnoj umjetnosti

Python se odlikuje mnogim besplatnim bibliotekama namijenjenim za duboko učenje te je stoga pogodan za stvaranje umjetnosti pomoću neuronskih mreža. Za stvaranje idućih primjera korištene su *Python*-ove biblioteke *TensorFlow*, *Matplotlib*, *Keras*, *NumPy*, itd.

TensorFlow je okruženje za trening i zaključivanje dubokih neuronskih mreža, nastalo kao odgovor na sve veću popularnost dubokog učenja, koju je 2015. prepoznao i iskoristio *Google Brain* tim. Cilj tog okruženja je zajednički rad programera i istraživača na modelima umjetne inteligencije. *TensorFlow* je skup povezanih automatskih učenja te algoritama za dubinsko učenje. Programer u *TensorFlow*-u radi u *Python* programskom jeziku, što omogućuje lako shvaćanje i baratanje mnogim bibliotekama, međutim, same računske operacije napisane su u *C++* programskom jeziku što omogućuje visoke performanse.

TensorFlow nam omogućuje da, pomoću *Tensor*-a, više-linearnih funkcija nalik više-dimenzionalnim poljima (zvanih *dtype*), izrađujemo grafikone i strukture protoka podataka. Također omogućuje i konstruiranje dijagrama mogućih operacija nad tim ulaznim poljima.

Tenzori su nepromjenjivog tipa, a po strukturi sliče na *numpy* polje (*np.array*). Tenzori mogu imati i više osi kao što je prikazano na slici 45.



Slika 45. Tenzori s različitim brojem osi [22]

Keras je biblioteka napisana u Pythonu, koja u suradnji s *TensorFlow*-om, omogućuje eksperimentiranje i rješavanje problema u strojnom učenju. Odlike *Keras*-a su:

- Jednostavnost - okruženje je stvoreno da se program usredotoči na rješavanje problema
- Fleksibilnost - progresivna složenost, omogućuje da se jednostavni zadaci rješavaju lako, dok se kompliciraniji rješavaju nadovezujući se na već naučeno
- Snaga - omogućuje performanse i snagu i za velike industrije, poput *NASA*-e i *Youtube*-a [23].

4.2.1 Prijenos neuronskog stila (engl. *Neural style transfer*)

Prijenos neuronskog stila je proces pri kojem se stil jedne slike (jednog umjetnika) prebacuje na drugu sliku/fotografiju pomoću neuronskih mreža. Cilj se postiže optimizacijom funkcije gubitka koja se sastoji od:

1. **gubitka stila** - sastoji se od zbroja udaljenosti između Gramovih matrica prikaza osnovne i referentne slike stila, a definira se pomoću duboke konvolucijske, neuronske mreže. Služi za hvatanje informacija o boji i teksturi na različitim prostornim mjerilima.
2. **gubitka sadržaja** - udaljenost između značajki osnovne slike (izvučene iz dubokog sloja) i kombinirane slike, čime se generirana slika drži blizu izvorne
3. **ukupnog gubitka varijacije** - prostorni kontinuitet između piksela finalne slike koji joj daje vizualnu koherentnost [24]

Primjer koda preuzet je iz *Google Coolab*-a, dok će u nastavku biti minimalne izmijene kod komentiranja koda te priloženih slika [24].

Za početak rada bitno je uključiti biblioteke *NumPy*, *Keras* i *TensorFlow* te definirati putanje slika i dimenziju generirane slike (slika 46.).

```
import numpy as np
import TensorFlow as tf
from TensorFlow import keras
from TensorFlow.keras.applications import vgg19

base_image_path = keras.utils.get_file("dubrovnik.jpg",
"https://journication.de/wp-content/uploads/sites/110/2020/02/dubrovnik-croatia-kro
atien-altstadt-stari-grad-old-town-ocean-meer-walk-viewpoint-summer-fortress-walls-
festung-stadtmauer-orange-roofs-medieval-mittelalter-scaled.jpg")
style_reference_image_path = keras.utils.get_file(
    "starry_night.jpg", "https://i.imgur.com/9ooB60I.jpg")
result_prefix = "dubrovnik_generated"

# Težine različitih komponenti
total_variation_weight = 1e-6
style_weight = 1e-6
content_weight = 2.5e-8
```

Slika 46. Kod za prijenos neuronskog stila - postavke

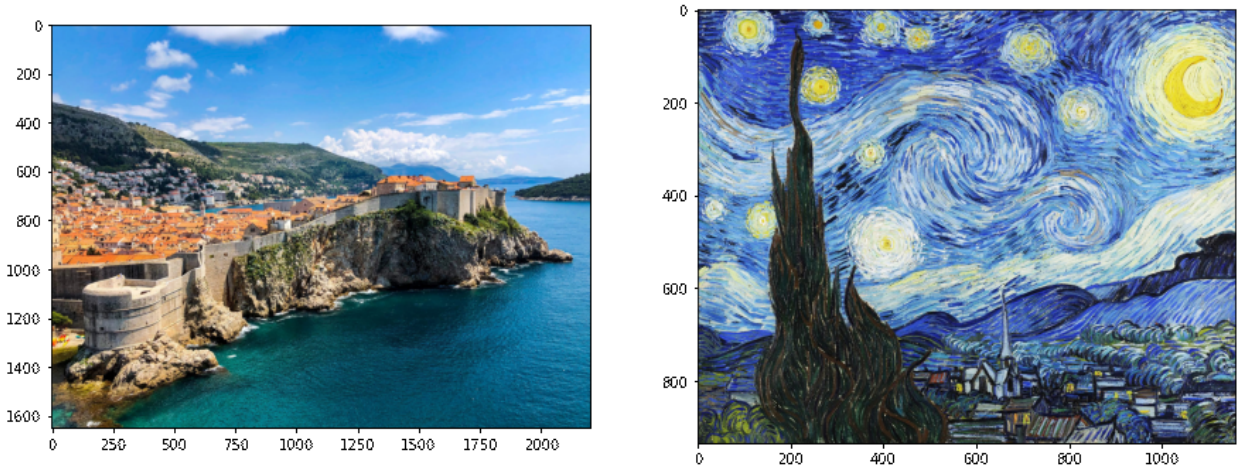
Funkcija vgg19 uključena iz *Keras* biblioteke predstavlja funkciju za konvolucijsku mrežu koja se sastoji od 19 slojeva.

Kako bi vidjeli priložene slike (originalnu i sliku referentnog stila) koristit ćemo se *Matplotlib* bibliotekom, koja nam omogućuje ispis slika u grafikonu (slika 47.).

```
import matplotlib.image as mpimg
import matplotlib.pyplot as plt

a = plt.imread(base_image_path)
b = plt.imread(style_reference_image_path)
f, axarr = plt.subplots(1,2, figsize=(15,15))
axarr[0].imshow(a)
axarr[1].imshow(b)
plt.show()
```

Slika 47. Kod za prijenos neuronskog stila - prikaz slika

Rezultat prethodnog djela koda možemo vidjeti na slici 48.



Slika 48. Kod za prijenos neuronskog stila - prikaz slika (ispis)

Nakon što smo učitali i provjerili slike definiramo funkcije za procese pretprocesiranja i reprocesiranja slika koristeći se *Keras* bibliotekom i njezinim funkcijama. pretprocesiranjem slika iste učitavamo, redimenzioniramo te oblikujemo, dok u reprocesiranju iz vektora izvlačimo pravu sliku i namještamo RGB vrijednosti (slika 49.).

```
def preprocess_image(image_path):
    # Util funkcije za otvaranje, redimenzioniranje i oblikovanje slika
    img = keras.preprocessing.image.load_img(
        image_path, target_size=(img_nrows, img_ncols)
    )
    img = keras.preprocessing.image.img_to_array(img)
    img = np.expand_dims(img, axis=0)
    img = vgg19.preprocess_input(img)
    return tf.convert_to_tensor(img)

def deprocess_image(x):
    # Util funkcija za pretvaranje tenzora(skalar/vektor) u pravu sliku
    x = x.reshape((img_nrows, img_ncols, 3))
    # Ukloniti nulti centar srednišnjim pikselom
    x[:, :, 0] += 103.939
    x[:, :, 1] += 116.779
    x[:, :, 2] += 123.68
    # 'BGR'->'RGB'
    x = x[:, :, ::-1]
    x = np.clip(x, 0, 255).astype("uint8")
    return x
```

Slika 49. Kod za prijenos neuronskog stila - procesiranje slika

Za računanje gubitka stila na sloju koristi se Gram matrica, za što trebamo izravnati (engl. *flatten*) sloj (Gram matrica se neće mijenjati bez obzira na broj filtera). Gram matrica dobiva se računanjem skalarnog produkta između vektora, a govori nam o sličnosti između filtera, tj. što je skalarni produkt manji to su dva vektora bliže i obrnuto. Na slici 50. prikazan je izračun gram matrice te sve tri, gore navedene, funkcije gubitka.

```
# Gram matrica tenzora slike (vanjski produkt značajki)

def gram_matrix(x):
    x = tf.transpose(x, (2, 0, 1))
    features = tf.reshape(x, (tf.shape(x)[0], -1))
    gram = tf.matmul(features, tf.transpose(features))
    return gram

# Gubitak stila osmišljen je za očuvanje stila referentne slike u generiranoj
# Baziran je na Gram matrici koje bilježe stil s mape značajke referentne slike i
generirane
```

Slika 50a. Kod za prijenos neuronskog stila - gram matrica i gubitci (1/2)

```
def style_loss(style, combination):

    S = gram_matrix(style)
    C = gram_matrix(combination)
    channels = 3
    size = img_nrows * img_ncols
    return tf.reduce_sum(tf.square(S - C)) / (4.0 * (channels ** 2) * (size ** 2))

# Pomoćna funkcija gubitka koja pomaže očuvati sadržaj originalne u generiranoj
slici

def content_loss(base, combination):
    return tf.reduce_sum(tf.square(combination - base))

# Ukupni gubitak varijacije osmišljen da održi generiranu sliku lokalno koherentnom

def total_variation_loss(x):
    a = tf.square(
        x[:, : img_nrows - 1, : img_ncols - 1, :] - x[:, 1:, : img_ncols - 1, :]
    )
    b = tf.square(
        x[:, : img_nrows - 1, : img_ncols - 1, :] - x[:, : img_nrows - 1, 1:, :]
    )
    return tf.reduce_sum(tf.pow(a + b, 1.25))
```

Slika 50b. Kod za prijenos neuronskog stila - gram matrica i gubitci (2/2)

Idući korak je postavljanje *VGG19* modela te postavljanje simboličnih izraza za svaki sloj u rječniku (*dictionary*) (slika 51.).

```
# Izrada VGG19 modela koji sadrži već trenirane ImageNet težine
model = vgg19.VGG19(weights="imagenet", include_top=False)

# Dohvatiti simbolične izlaze za svaki sloj
outputs_dict = dict([(layer.name, layer.output) for layer in model.layers])

# Postaviti model koji vraća vrijednosti za svaki sloj u VGG19 (dict)
feature_extractor = keras.Model(inputs=model.inputs, outputs=outputs_dict)
```

Slika 51. Kod za prijenos neuronskog stila - izrada *VGG19* modela

Nakon što smo postavili model, postavljamo slojeve za gubitak stila i sadržaja te definiramo funkciju za izračun ukupnog gubitka (slika 52.).

```
# lista slojeva koji se koriste za gubitak stila
style_layer_names = [
    "block1_conv1",
    "block2_conv1",
    "block3_conv1",
    "block4_conv1",
    "block5_conv1",
]
# Sloj koji se koristi za gubitak sadržaja
content_layer_name = "block5_conv2"

def compute_loss(combination_image, base_image, style_reference_image):
    input_tensor = tf.concat(
        [base_image, style_reference_image, combination_image], axis=0
    )
    features = feature_extractor(input_tensor)

    # Inicijalizacija gubitka
    loss = tf.zeros(shape=())

    # Dodavanje gubitka sadržaja
    layer_features = features[content_layer_name]
    base_image_features = layer_features[0, :, :, :]
    combination_features = layer_features[2, :, :, :]
    loss = loss + content_weight * content_loss(
        base_image_features, combination_features
    )
    # Dodavanje gubitka stila
    for layer_name in style_layer_names:
        layer_features = features[layer_name]
        style_reference_features = layer_features[1, :, :, :]
        combination_features = layer_features[2, :, :, :]
        sl = style_loss(style_reference_features, combination_features)
        loss += (style_weight / len(style_layer_names)) * sl

    # Dodavanje gubitka varijacije
    loss += total_variation_weight * total_variation_loss(combination_image)
    return loss
```

Slika 52. Kod za prijenos neuronskog stila - izračun gubitaka

Kako bi optimalno izračunali gubitak, potrebna nam je delta koju dobijemo izračunom izvedenica (derivacija), a sve se to postiže *TensorFlow* funkcijom `GradientTape()`, koja s obzirom na cijenu daje gradijent (slika 53.).

```
@tf.function
def compute_loss_and_grads(combination_image, base_image, style_reference_image):
    with tf.GradientTape() as tape:
        loss = compute_loss(combination_image, base_image, style_reference_image)
    grads = tape.gradient(loss, combination_image)
    return loss, grads
```

Slika 53. Kod za prijenos neuronskog stila - `GradientTape` funkcija

Kada smo definirali sve potrebne funkcije, vršimo proces treniranja, u ovom slučaju u 4000 koraka pri kojima je svako 100-i rezultat (slika) spremljen, a brzina učenja smanjuje se za 0.96 svakih 100 koraka (slika 54).

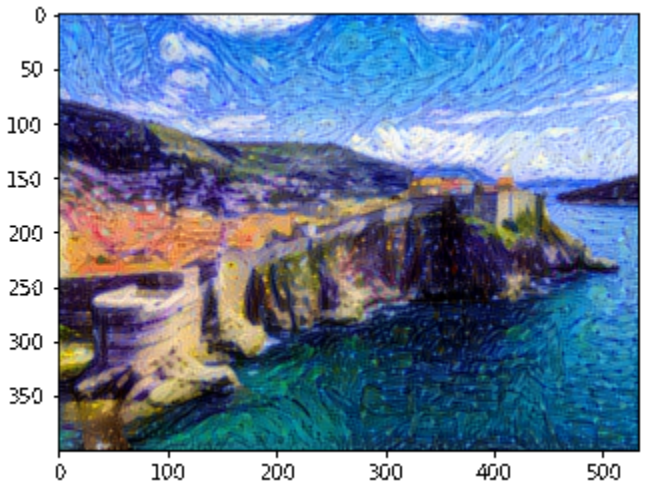
```
optimizer = keras.optimizers.SGD(
    keras.optimizers.schedules.ExponentialDecay(
        initial_learning_rate=100.0, decay_steps=100, decay_rate=0.96
    )
)

base_image = preprocess_image(base_image_path)
style_reference_image = preprocess_image(style_reference_image_path)
combination_image = tf.Variable(preprocess_image(base_image_path))

iterations = 4000
for i in range(1, iterations + 1):
    loss, grads = compute_loss_and_grads(
        combination_image, base_image, style_reference_image
    )
    optimizer.apply_gradients([(grads, combination_image)])
    if i % 100 == 0:
        print("Iteration %d: loss=%.2f" % (i, loss))
        img = deprocess_image(combination_image.numpy())
        fname = result_prefix + "_at_iteration_%d.png" % i
        keras.preprocessing.image.save_img(fname, img)
```

Slika 54. Kod za prijenos neuronskog stila - Treniranje

Za kraj želimo prikazati rezultate (slika 55.), a to ćemo postići ispisivajući sliku dobivenu nakon 4000 koraka (slika 56).

```
c = plt.imread(result_prefix + "_at_iteration_4000.png")
imgplot = plt.imshow(c)
plt.show()
```

Slika 55. Kod za prijenos neuronskog stila - Rezultat



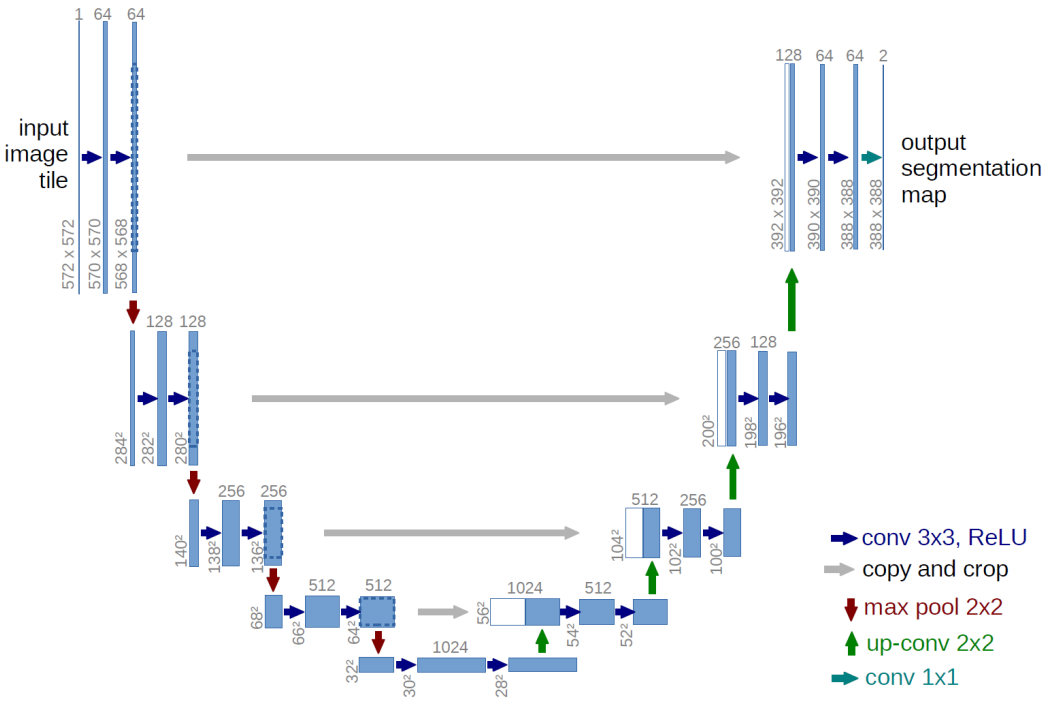
Slika 56. Kod za prijenos neuronskog stila - Rezultat (Ispis)

4.2.2 *Pix2Pix* (cGAN)

Pix2Pix naziv je uvjetovanog GAN-a (engl. *conditional GAN* - cGAN) koji uči preslikavanje od ulaznih do izlaznih slika. Širok je spektar korištenja istog, a tu spada sintetiziranje fotografije s karata, generiranje obojenih fotografija iz crno-bijelih, generiranje fotografije iz priloženih skica, te pretvaranje slika preuzetih iz Google-ove karte u zračne slike [25].

Preslikavanje slike u sliku, prije uvođenja cGAN-a, vršilo se klasificiranjem/regresijom piksela što je izlazni rezultat činilo nestrukturiranim, tj. svaki izlazni piksel bio je neovisan o ostalima. cGAN uči na strukturiranom gubitku te penalizira sve što odskače od očekivanog.

Pix2Pix sadrži generator U-net arhitekture te diskriminator koji je predstavljen konvolucijskim *PatchGAN* klasifikatorom. Na slici 57. Možemo vidjeti U-net arhitekturu, koja se sastoji od enkodera i dekodera koji vrše reduciranje uzorka (engl. *downsampling*) kako bi došli do konteksta slike, a potom 'širenje' uzorka (engl. *upsampling*) kako bi se vratile informacije poput prostornog razumijevanja.



Slika 57. U-net arhitektura [26]

Kod koji će se prikazati u nastavku preuzet je s *TensorFlow*-a, a pokrenut ćemo ga u *Google Coolab*-u s drugačijim podatkovnim skupom od izabranog u primjeru [25]. U primjeru su podaci bazirani na fasadama zgrada, dok smo mi odabrali skup podataka koji sadrži gradske pejzaže (ulice, ceste, prirodu, kuće).

Kao i uvijek, u početku učitavamo sve bitne biblioteke za rad s podacima (slika 58.)

```
import TensorFlow as tf

import os
import pathlib
import time
import datetime

from matplotlib import pyplot as plt
from IPython import display
```

Slika 58. *Pix2pix* kod - Učitivanje *Python* biblioteka

Potom odabiremo skup podataka te isti učitivamo koristeći *Keras* biblioteku (slika 59.)

```
dataset_name = "cityscapes" #@param ["cityscapes", "edges2handbags", "edges2shoes",
"facades", "maps", "night2day"]

_URL = f'http://efrosgans.eecs.berkeley.edu/pix2pix/datasets/{dataset_name}.tar.gz'

path_to_zip = tf.keras.utils.get_file(
    fname=f"{dataset_name}.tar.gz",
    origin=_URL,
    extract=True)

path_to_zip  = pathlib.Path(path_to_zip)

PATH = path_to_zip.parent/dataset_name
```

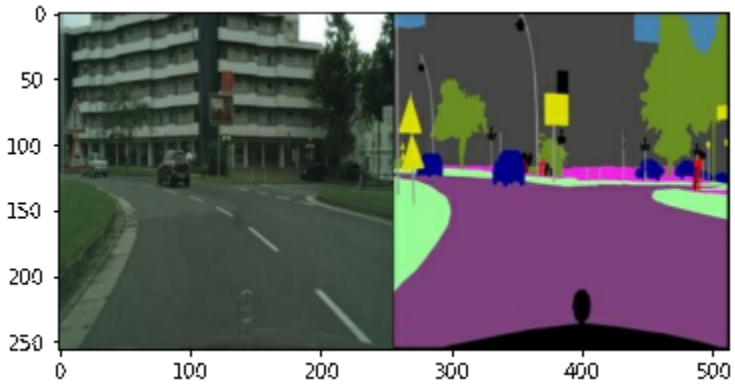
Slika 59. *Pix2Pix* kod - Učitavanje skupa podataka

Kako bi vidjeli primjer jednog uzorka ispisujemo isti na ekran koristeći se *Matplotlib* bibliotekom (slika 60.).

```
sample_image = tf.io.read_file(str(PATH / 'train/1.jpg'))
sample_image = tf.io.decode_jpeg(sample_image)
print(sample_image.shape)
plt.figure()
plt.imshow(sample_image)
```

Slika 60. *Pix2Pix* kod - Prikaz jednog primjerka

Svaki uzorak sastoji se od dvije slike 256x256 veličine - stvarne i ‘nacrtane’ (slika 61.).



Slika 61. *Pix2Pix* kod - Prikaz jednog primjerka (rezultat)

Potom je bitno razdvojiti svaku sliku u dva tenzora, kako bi se mogli pojedinačno koristiti (slika 62.).

```
def load(image_file):
  # Čitanje i dekodiranje slike u uint8 tenzor
  image = tf.io.read_file(image_file)
  image = tf.image.decode_jpeg(image)

  # Razdvajanje svakog tenzora u dva - 1 pravi, 1 označeni
  w = tf.shape(image)[1]
  w = w // 2
  input_image = image[:, w:, :]
  real_image = image[:, :w, :]

  # Pretvaranje obje slike u float32 tenzore
  input_image = tf.cast(input_image, tf.float32)
  real_image = tf.cast(real_image, tf.float32)

  return input_image, real_image
```

Slika 62. *Pix2Pix* kod - Razdvajanje uzorka u dva tenzora

Idući korak je primjenjivanje nasumičnog podrhtavanja (engl. *jitter*) i zrcaljenja kako bi obradili skup podataka za treniranje. Na slici 63. vidjet ćemo sljedeće:

1. Kod za definiranje veličine međuspremnika (BUFFER_SIZE), serije (BATCH_SIZE), te slike (IMG_WIDTH, IMG_HEIGHT).
2. Mijenjanje dimenzija slike iz 256x256 u 286x286
3. Slučajno kidanje slike na dimenziju 256x256
4. Slučajno horizonatalno okretanje slike (lijevo → desno)
5. Normaliziranje slike u područje [-1,1]

```
BUFFER_SIZE = 400
BATCH_SIZE = 1
IMG_WIDTH = 256
IMG_HEIGHT = 256

def resize(input_image, real_image, height, width):
  input_image = tf.image.resize(input_image, [height, width],
                                method=tf.image.ResizeMethod.NEAREST_NEIGHBOR)
  real_image = tf.image.resize(real_image, [height, width],
                               method=tf.image.ResizeMethod.NEAREST_NEIGHBOR)
  return input_image, real_image

def random_crop(input_image, real_image):
  stacked_image = tf.stack([input_image, real_image], axis=0)
  cropped_image = tf.image.random_crop(stacked_image, size=[2, IMG_HEIGHT,
  IMG_WIDTH, 3])

  return cropped_image[0], cropped_image[1]

def normalize(input_image, real_image):
  input_image = (input_image / 127.5) - 1
  real_image = (real_image / 127.5) - 1

  return input_image, real_image

@tf.function()
def random_jitter(input_image, real_image):
  Input_image, real_image = resize(input_image, real_image, 286, 286)

  input_image, real_image = random_crop(input_image, real_image)

  if tf.random.uniform(()) > 0.5:
    input_image = tf.image.flip_left_right(input_image)
    real_image = tf.image.flip_left_right(real_image)

  return input_image, real_image
```

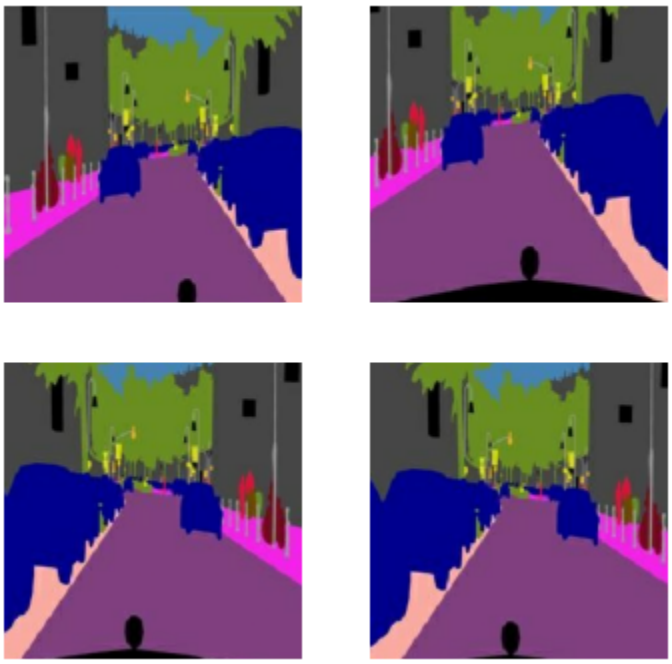
Slika 63. *Pix2Pix* kod - podrhtavanje i zrcaljenje u predobradi slike

U nastavku je kod za prikaz navedenih i definiranih radnji - zrcaljenje, kidanje slike (slika 64.).

```
plt.figure(figsize=(6, 6))
for i in range(4):
  rj_inp, rj_re = random_jitter(inp, re)
  plt.subplot(2, 2, i + 1)
  plt.imshow(rj_inp / 255.0)
  plt.axis('off')
plt.show()
```

Slika 64. *Pix2Pix* kod- zrcaljenje i kidanje slike

Rezultat vidimo u grafikonu na slici 65.



Slika 65. *Pix2Pix* kod- zrcaljenje i kidanje slike (rezultat)

Zatim definiramo pomoćne funkcije za učitavanje i pretprocesiranje skupova za učenje i testiranje, kao što možemo vidjeti na slici 66.

```
def load_image_train(image_file):
  input_image, real_image = load(image_file)
  input_image, real_image = random_jitter(input_image, real_image)
  input_image, real_image = normalize(input_image, real_image)

  return input_image, real_image

def load_image_test(image_file):
  input_image, real_image = load(image_file)
  input_image, real_image = resize(input_image, real_image,
                                   IMG_HEIGHT, IMG_WIDTH)
  input_image, real_image = normalize(input_image, real_image)

  return input_image, real_image

train_dataset = tf.data.Dataset.list_files(str(PATH / 'train/*.jpg'))
train_dataset = train_dataset.map(load_image_train,
                                  num_parallel_calls=tf.data.AUTOTUNE)
train_dataset = train_dataset.shuffle(BUFFER_SIZE)
train_dataset = train_dataset.batch(BATCH_SIZE)

try:
  test_dataset = tf.data.Dataset.list_files(str(PATH / 'test/*.jpg'))
except tf.errors.InvalidArgumentError:
  test_dataset = tf.data.Dataset.list_files(str(PATH / 'val/*.jpg'))

test_dataset = test_dataset.map(load_image_test)
test_dataset = test_dataset.batch(BATCH_SIZE)
```

Slika 66. *Pix2Pix* kod - učitavanje i pretprocesiranje skupova za učenje i testiranje

Sada slijedi stvaranje generatora, prethodno opisane, U-Net arhitekture. Blokovi u enkoderu su strukture: konvolucija → normalizacija serije → *Leaky ReLU* funkcija, dok je struktura blokova u dekoderu: transponirana konvolucija → normalizacija serije → Ispadanje (prva tri bloka) → *ReLU* funkcija (slika 67.) [25].

```
OUTPUT_CHANNELS = 3

def downsample(filters, size, apply_batchnorm=True):
  initializer = tf.random_normal_initializer(0., 0.02)

  result = tf.keras.Sequential()
  result.add(
      tf.keras.layers.Conv2D(filters, size, strides=2, padding='same',
                             kernel_initializer=initializer, use_bias=False))

  if apply_batchnorm:
    result.add(tf.keras.layers.BatchNormalization())

  result.add(tf.keras.layers.LeakyReLU())

  return result

down_model = downsample(3, 4)
down_result = down_model(tf.expand_dims(inp, 0))

def upsample(filters, size, apply_dropout=False):
  initializer = tf.random_normal_initializer(0., 0.02)

  result = tf.keras.Sequential()
  result.add(tf.keras.layers.Conv2DTranspose(filters, size, strides=2,
                                             padding='same',
                                             kernel_initializer=initializer,
                                             use_bias=False))

  result.add(tf.keras.layers.BatchNormalization())

  if apply_dropout:
      result.add(tf.keras.layers.Dropout(0.5))

  result.add(tf.keras.layers.ReLU())

  return result

up_model = upsample(3, 4)
up_result = up_model(down_result)
```

Slika 67. *Pix2Pix* kod - Izrada enkodera i dekodera u generatoru

Nakon što smo definirali funkcije za enkoder i dekoder, uvrstit ćemo ih u primjenu koristeći ih u novo definiranoj funkciji za generator (slika 68.).

```
def Generator():
  inputs = tf.keras.layers.Input(shape=[256, 256, 3])
  down_stack = [
    downsample(64, 4, apply_batchnorm=False),  # (batch_size, 128, 128, 64)
    downsample(128, 4),  # (batch_size, 64, 64, 128)
    downsample(256, 4),  # (batch_size, 32, 32, 256)
    downsample(512, 4),  # (batch_size, 16, 16, 512)
    downsample(512, 4),  # (batch_size, 8, 8, 512)
    downsample(512, 4),  # (batch_size, 4, 4, 512)
    downsample(512, 4),  # (batch_size, 2, 2, 512)
    downsample(512, 4),  # (batch_size, 1, 1, 512)
  ]

  up_stack = [
    upsample(512, 4, apply_dropout=True),  # (batch_size, 2, 2, 1024)
    upsample(512, 4, apply_dropout=True),  # (batch_size, 4, 4, 1024)
    upsample(512, 4, apply_dropout=True),  # (batch_size, 8, 8, 1024)
    upsample(512, 4),  # (batch_size, 16, 16, 1024)
    upsample(256, 4),  # (batch_size, 32, 32, 512)
    upsample(128, 4),  # (batch_size, 64, 64, 256)
    upsample(64, 4),  # (batch_size, 128, 128, 128)
  ]

  initializer = tf.random_normal_initializer(0., 0.02)
  last = tf.keras.layers.Conv2DTranspose(OUTPUT_CHANNELS, 4,
                                         strides=2,
                                         padding='same',
                                         kernel_initializer=initializer,
                                         activation='tanh')  # (batch_size, 256, 256,
                                         3)
  x = inputs

  skips = []
  for down in down_stack:
    x = down(x)
    skips.append(x)

  skips = reversed(skips[:-1])

  for up, skip in zip(up_stack, skips):
    x = up(x)
    x = tf.keras.layers.Concatenate()([x, skip])

  x = last(x)
  return tf.keras.Model(inputs=inputs, outputs=x)
```

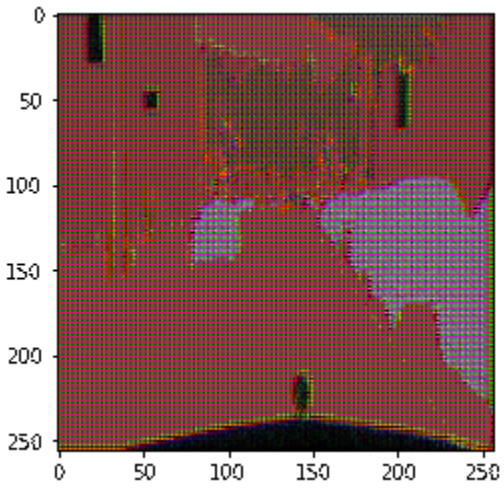
Slika 68. *Pix2Pix* kod - Definiranje funkcije generatora

Nakon što smo definirali generator, provjeravamo njegovu strukturu (zbog veličine rezultata pogledati originalan kod) te testiramo isti (slika 69.).

```
generator = Generator()
tf.keras.utils.plot_model(generator, show_shapes=True, dpi=64)

gen_output = generator(inp[tf.newaxis, ...], training=False)
plt.imshow(gen_output[0, ...])
```

Slika 69. *Pix2Pix* kod - provjera i testiranje generatora

Rezultat testiranja generatora vidimo na slici 70.



Slika 70. *Pix2Pix* kod - provjera i testiranje generatora (rezultat)

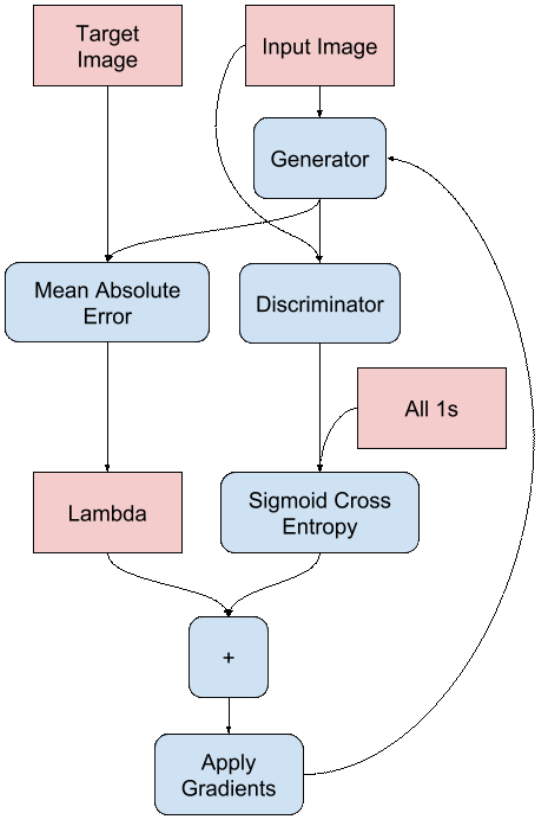
Gubitak generatora je ujedno gubitak sigmoidne unakrsne entropije, a također se spominje i gubitak L_1 koji je srednja apsolutna pogreška (MAE - *Mean Absolute Error*) između generirane i ciljane slike. Prateći gubitak omogućuje se strukturiranje generirane slike tako da bude sličnija ciljanoj [27].

Kod za navedene gubitke i prikaz modela treniranja generatora možemo vidjeti na slici 71.

```
LAMBDA = 100
loss_object = tf.keras.losses.BinaryCrossentropy(from_logits=True)

def generator_loss(disc_generated_output, gen_output, target):
  gan_loss = loss_object(tf.ones_like(disc_generated_output),
disc_generated_output)

  # Mean absolute error
  l1_loss = tf.reduce_mean(tf.abs(target - gen_output))

  total_gen_loss = gan_loss + (LAMBDA * l1_loss)

  return total_gen_loss, gan_loss, l1_loss
```

Slika 71. *Pix2Pix* kod - model treniranja i kod za gubitak generatora

Dijagram koji predstavlja proces treniranja generatora vidimo na slici 72.



Slika 72. *Pix2Pix* kod- dijagram modela treniranja

Diskriminator je, u ovom slučaju, konvolucijski *PatchGAN* klasifikator, što znači da pokušava klasificirati je li svaki slikovni isječak (engl. *patch*) stvaran ili ne.

Svaki blok u diskriminatoru sastoji se od konvolucije → normalizacije serije → *Leaky ReLU* funkcije. *Leaky ReLU* funkcija predstavlja propusnu *ReLU* funkciju, koja za razliku od osnovne, propušta gradient u neaktivnom području te smanjuje opterećenje i povećava kapacitet memorije.

Oblik izlaznog rezultata nakon zadnjeg sloja jest 30x30, a svaki taj slikovni isječak klasificira dio od 70x70 ulazne slike.

Diskriminator prima dva ulaza:

- Ulaznu sliku i ciljanu sliku, koju bi trebao klasificirati kao pravu
- Ulaznu sliku i generiranu sliku (izlaz od strane generatora), koju bi trebao klasificirati kao lažnu [25]

Na slici 73. vidimo funkciju za definiranje diskriminatora, u kojoj je navedeno sve što smo naveli da diskriminator radi.

```
def Discriminator():
  initializer = tf.random_normal_initializer(0., 0.02)

  inp = tf.keras.layers.Input(shape=[256, 256, 3], name='input_image')
  tar = tf.keras.layers.Input(shape=[256, 256, 3], name='target_image')

  x = tf.keras.layers.concatenate([inp, tar])  # (batch_size, 256, 256, channels*2)

  down1 = downsample(64, 4, False)(x)  # (batch_size, 128, 128, 64)
  down2 = downsample(128, 4)(down1)  # (batch_size, 64, 64, 128)
  down3 = downsample(256, 4)(down2)  # (batch_size, 32, 32, 256)

  zero_pad1 = tf.keras.layers.ZeroPadding2D()(down3)  # (batch_size, 34, 34, 256)
  conv = tf.keras.layers.Conv2D(512, 4, strides=1,
                                kernel_initializer=initializer,
                                use_bias=False)(zero_pad1)  # (batch_size, 31, 31,
                                512)

  batchnorm1 = tf.keras.layers.BatchNormalization()(conv)
  leaky_relu = tf.keras.layers.LeakyReLU()(batchnorm1)
  zero_pad2 = tf.keras.layers.ZeroPadding2D()(leaky_relu)  # (batch_size, 33, 33,
                                                             512)
  last = tf.keras.layers.Conv2D(1, 4, strides=1, kernel_initializer=initializer)
                               (zero_pad2)  #(batch_size, 30, 30, 1)

  return tf.keras.Model(inputs=[inp, tar], outputs=last)
```

Slika 73. *Pix2Pix* kod - definiranje funkcije diskriminatora

Kako bi predočili vizualizaciju diskriminatorove arhitekture pozivamo kod sa slike 74., a samu sliku arhitekture možemo pogledati u *Google Colab*-u [25].

```
discriminator = Discriminator()
tf.keras.utils.plot_model(discriminator, show_shapes=True, dpi=64)
```

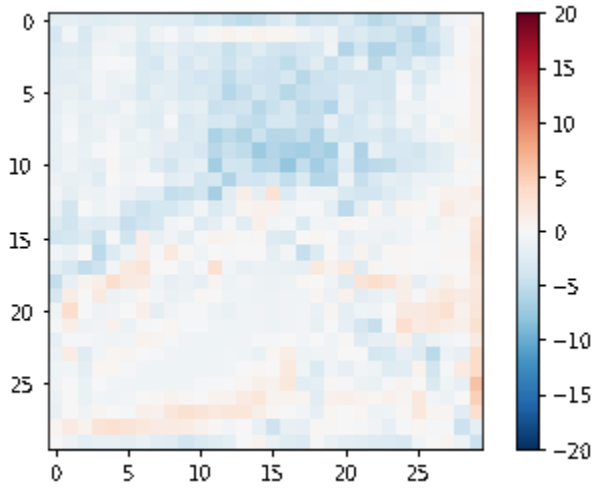
Slika 74. *Pix2Pix* kod - prikaz arhitekture diskriminatora

Testiramo diskriminator i vizualiziramo rezultat (slika 75.).

```
disc_out = discriminator([inp[tf.newaxis, ...], gen_output], training=False)
plt.imshow(disc_out[0, ..., -1], vmin=-20, vmax=20, cmap='RdBu_r')
plt.colorbar()
```

Slika 75. *Pix2Pix* kod - Testiranje i vizualizacija rezultata

Rezultat vizualizacije vidimo na slici 76.



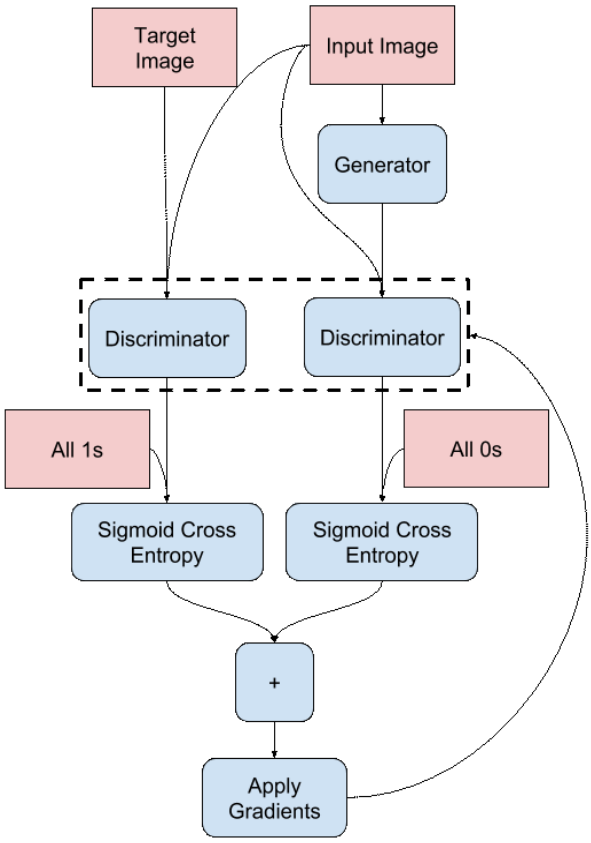
Slika 76. *Pix2Pix* kod - Testiranje i vizualizacija rezultata (rezultat)

Za proces učenja potrebno je definirati gubitak diskiriminatora, koji prima gubitak prave i gubitak generirane slike. Pravi gubitak je sigmoidna unakrsna entropija pravih slika i niza jedinica, a generirani gubitak je sigmoidna unakrsna entropija generiranih slika i niza nula (lažne slike). Ukupni gubitak diskriminatora je zbroj dva prethodno opisana gubitka (slika 79.).

```
def discriminator_loss(disc_real_output, disc_generated_output):
  real_loss = loss_object(tf.ones_like(disc_real_output), disc_real_output)

  generated_loss = loss_object(tf.zeros_like(disc_generated_output),
disc_generated_output)

  total_disc_loss = real_loss + generated_loss

  return total_disc_loss
```

Slika 77. Pix2pix kod - definiranje gubitka diskriminatora

Proces treniranja diskriminatora prikazan je na slici 78.



Slika 78. *Pix2Pix* - proces treniranja [25]

Za optimizaciju generatora i diskriminatora koristit će se *Adam* (*Adaptive Moment Estimation*) optimizator iz *Keras* biblioteke (slika 79.), koji omogućuje iterativno ažuriranje mreže na temelju naučenih podataka. Optimizator se odlikuje efikasnošću te primjenjivosti na velike podatkovne skupove ili skupove s velikim šumom.

```
tf.keras.optimizers.Adam(
    learning_rate=0.001,
    beta_1=0.9,
    beta_2=0.999,
    epsilon=1e-07,
    amsgrad=False,
    name="Adam",
    **kwargs
)
```

Slika 79. Hiperparametri *Adam* optimizatora [28]

Premda optimizator ima zadane vrijednosti, mi ćemo izmijeniti prva dva parametra. Na slici 80. vidimo da je prvi parametar koji prima stopa učenja (engl. *Learning rate*), a drugi `beta_1`, koji predstavlja stopu opadanja za prvi moment (srednja udaljenost od 0). Na istoj slici vidimo kod za spremanje kontrolnih točaka (engl. *checkpoint*).

```
generator_optimizer = tf.keras.optimizers.Adam(2e-4, beta_1=0.5)
discriminator_optimizer = tf.keras.optimizers.Adam(2e-4, beta_1=0.5)

checkpoint_dir = './training_checkpoints'
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
checkpoint = tf.train.Checkpoint(generator_optimizer=generator_optimizer,
                                 discriminator_optimizer=discriminator_optimizer,
                                 generator=generator,
                                 discriminator=discriminator)
```

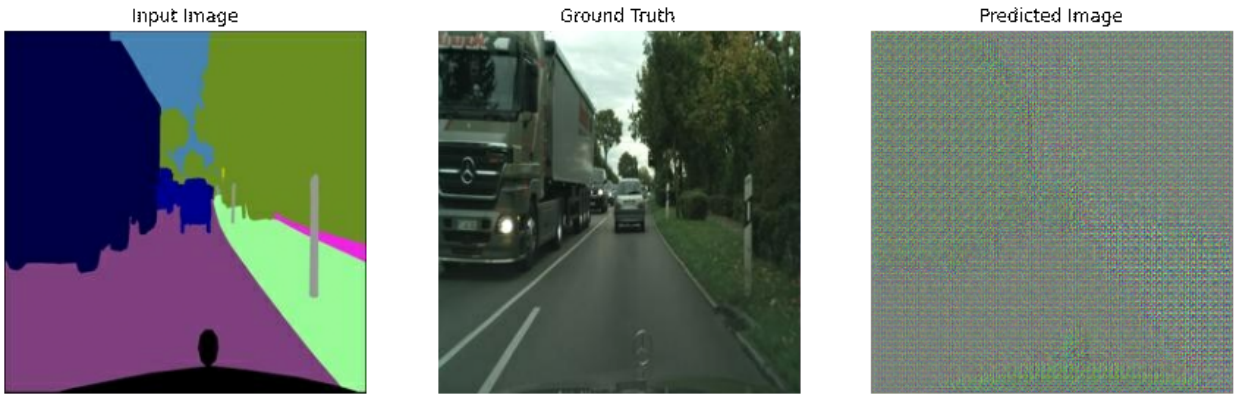
Slika 80. *Pix2Pix* kod - optimizacija generatora i diskriminatora te spremanje kontrolnih točaka

Kako bi generirali slike, moramo proslijediti slike iz testnog seta u generator, koji potom prebacuje ulazne slike u izlazne. Nakon toga crtamo predviđanje (slika 81.).

```
def generate_images(model, test_input, tar):
  prediction = model(test_input, training=True)
  plt.figure(figsize=(15, 15))

  display_list = [test_input[0], tar[0], prediction[0]]
  title = ['Input Image', 'Ground Truth', 'Predicted Image']

  for i in range(3):
    plt.subplot(1, 3, i+1)
    plt.title(title[i])
    # Getting the pixel values in the [0, 1] range to plot.
    plt.imshow(display_list[i] * 0.5 + 0.5)
    plt.axis('off')
  plt.show()

#testiranje funkcije
for example_input, example_target in test_dataset.take(1):
  generate_images(generator, example_input, example_target)
```

Slika 81. *Pix2Pix* kod - Generiranje slika

Jedan rezultat prethodnog testa funkcije vidimo na slici 82.



Slika 82. *Pix2Pix* kod - Generiranje slika (rezultat)

Trening se vrši prateći iduće korake (slika 83.):

1. Za svaki ulazni primjer generira se izlaz
2. Diskriminator prima dva ulaza (ulaznu i generiranu sliku, ulaznu i ciljanu sliku)
3. Računa se gubitak generatora te gubitak diskriminatora
4. Računa se gradientni gubitak, s obzirom na varijable generatora i diskriminatora, te se primijenjuju na optimizator
5. Spremiti izračunate gubitke u *TensorBoard*

```
@tf.function
def train_step(input_image, target, step):
  with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
    gen_output = generator(input_image, training=True)

    disc_real_output = discriminator([input_image, target], training=True)
    disc_generated_output = discriminator([input_image, gen_output], training=True)

    gen_total_loss, gen_gan_loss, gen_l1_loss =
generator_loss(disc_generated_output, gen_output, target)
    disc_loss = discriminator_loss(disc_real_output, disc_generated_output)

  generator_gradients = gen_tape.gradient(gen_total_loss,
                                          generator.trainable_variables)
  discriminator_gradients = disc_tape.gradient(disc_loss,
                                               discriminator.trainable_variables)

  generator_optimizer.apply_gradients(zip(generator_gradients,
                                          generator.trainable_variables))
  discriminator_optimizer.apply_gradients(zip(discriminator_gradients,
                                              discriminator.trainable_variables))

  with summary_writer.as_default():
    tf.summary.scalar('gen_total_loss', gen_total_loss, step=step//1000)
    tf.summary.scalar('gen_gan_loss', gen_gan_loss, step=step//1000)
    tf.summary.scalar('gen_l1_loss', gen_l1_loss, step=step//1000)
    tf.summary.scalar('disc_loss', disc_loss, step=step//1000)
```

Slika 83. *Pix2Pix* kod - trening

Petlja za treniranje iterira po broju koraka (u ovom primjeru mogu se koristiti različiti podatkovni skupovi stoga je bolje ići po koracima nego po epohama), te svako:

- 10 koraka ispisuje točku (.)
- 1000 koraka briše prikaz te ponovno pokreće generate_images() da prikaže trenutno stanje procesa
- 5000 koraka sprema kontrolnu točku

Kod za navedeno možemo vidjeti na slici 84.

```
def fit(train_ds, test_ds, steps):
  example_input, example_target = next(iter(test_ds.take(1)))
  start = time.time()

  for step, (input_image, target) in train_ds.repeat().take(steps).enumerate():
    if (step) % 1000 == 0:
      display.clear_output(wait=True)

      if step != 0:
        print(f'Time taken for 1000 steps: {time.time()-start:.2f} sec\n')

      start = time.time()
      generate_images(generator, example_input, example_target)
      print(f"Step: {step//1000}k")

    train_step(input_image, target, step)

    if (step+1) % 10 == 0:
      print('.', end='', flush=True)

    if (step + 1) % 5000 == 0:
      checkpoint.save(file_prefix=checkpoint_prefix)
```

Slika 84. *Pix2Pix* - petlja za treniranje

TensorFlow sadrži *TensorBoard*-ove, koji su interaktivni prikaz praćenja i vizualizacije mjerenja (gubitci i točnost), grafova, histograma i slično. Kako bi se mogli služiti *TensorBoard*-om moramo ga inicirati te spremiti podatke, a kod za navedeno možemo vidjeti na slici 85.

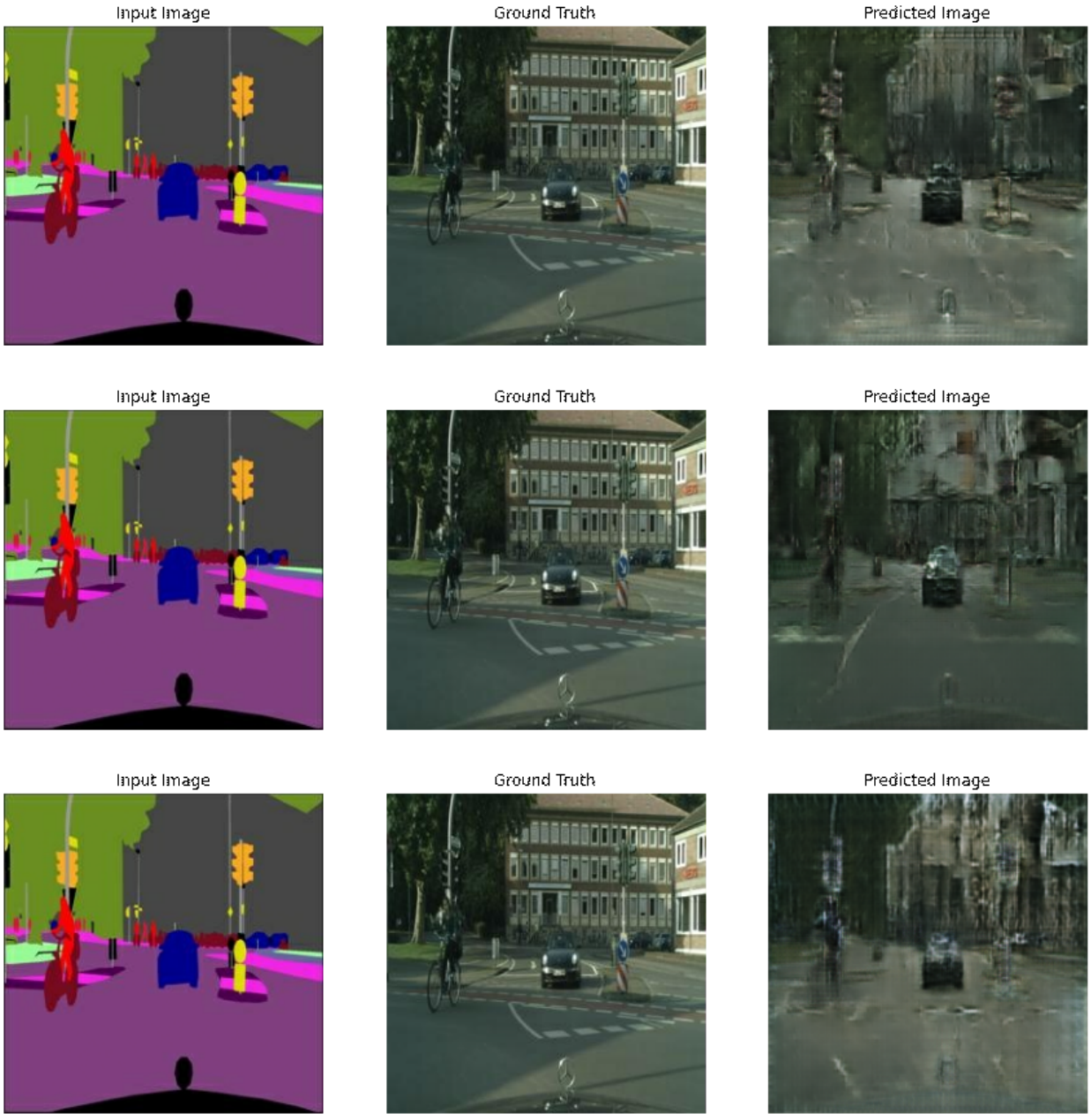
```
%load_ext tensorboard
%tensorboard --logdir {log_dir}
```

Slika 85. *Pix2Pix* kod **-** Učitavanje *TensorBoard*-a te spremanje informacija u zapisnik

Na slici 86. možemo vidjeti poziv funkcije za treniranje, a na slici 87. rezultate koje smo spremili na 10 000, 24 000 te 39 000 koraka.

```
fit(train_dataset, test_dataset, steps=40000)
```

Slika 86. *Pix2Pix* kod - petlja za treniranje



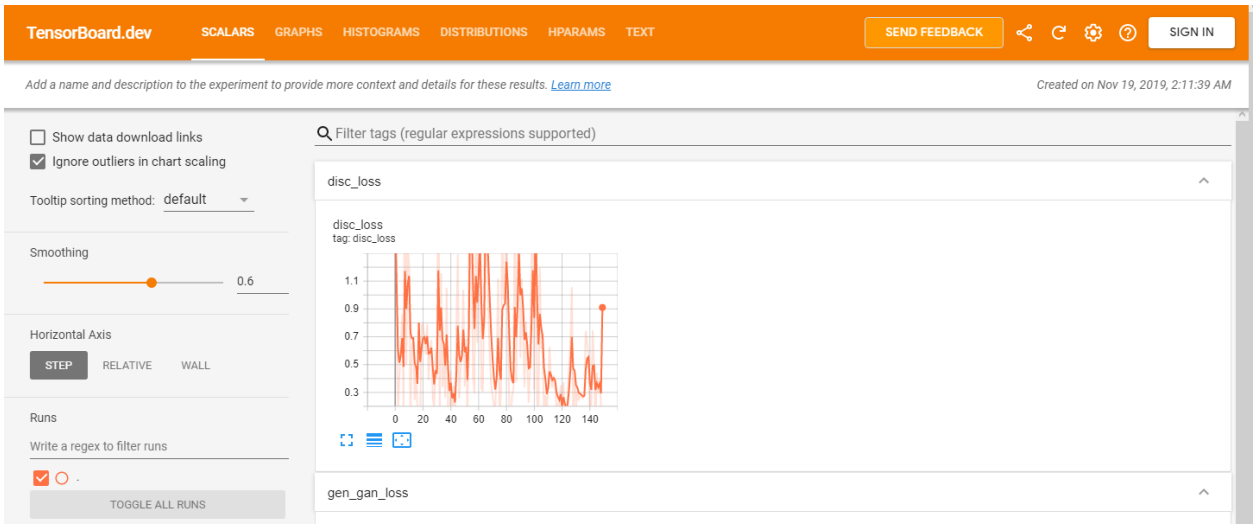
Slika 87. *Pix2pix* kod - tri spremljena rezultata petlje za treniranje (12,000, 24,000, 39,000 koraka)

Nakon što je proces treniranja završen možemo pratiti promjenu gubitka u kontrolnim točkama u *TensorBoard*-u (slika 88.).

```
display.IFrame(
    src="https://tensorboard.dev/experiment/lZ0C6FONROaUMfjYkVyJqw",
    width="100%",
    height="1000px")
```

Slika 88. *Pix2Pix* kod - vizualizacija *TensorBoard*-a

Vizualizacija je prikazana na slici 89.



Slika 89. *Pix2pix* kod - vizualizacija *TensorBoard*-a (rezultat)

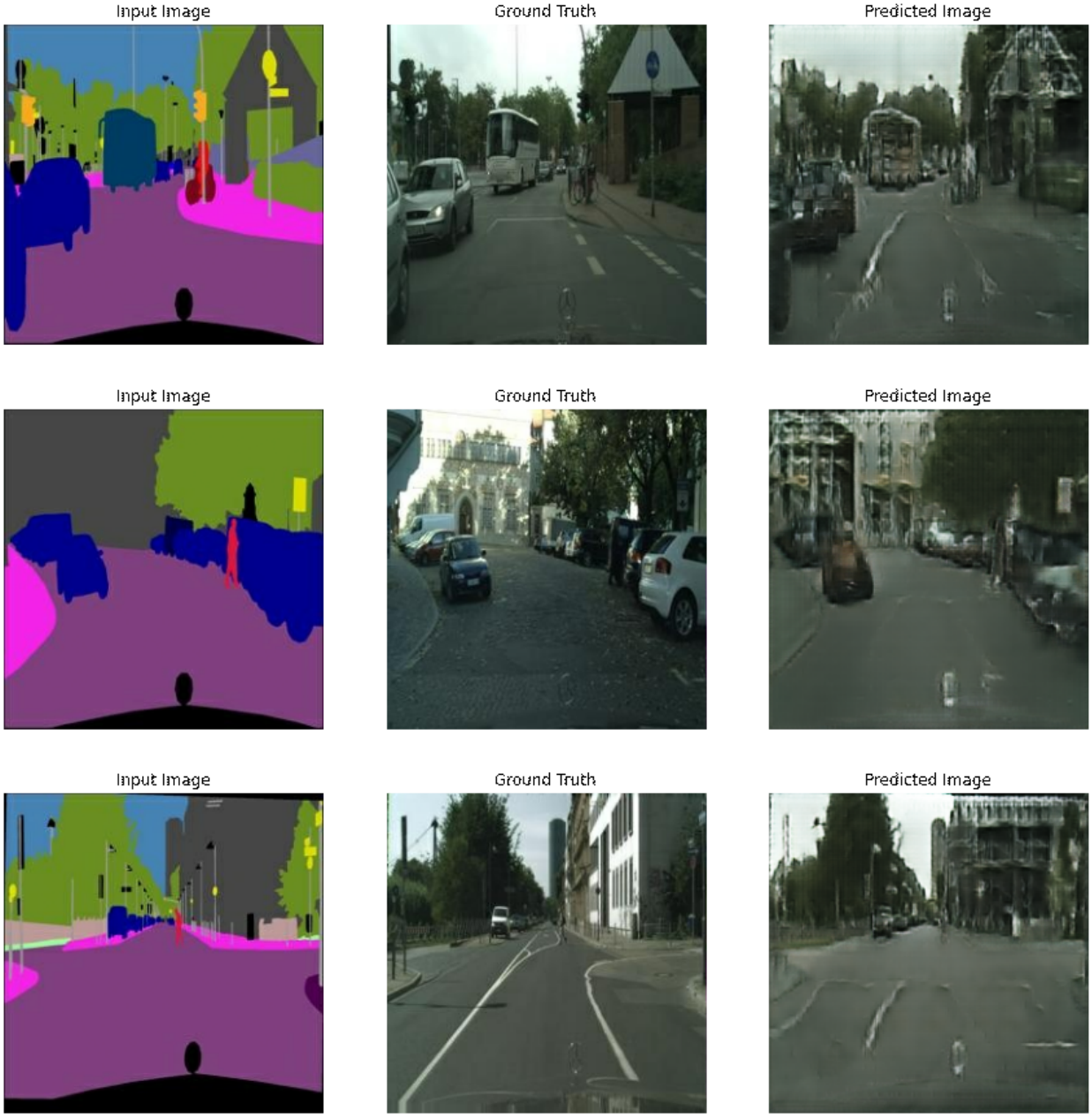
Ako vrijednost gubitka diskriminatora ili generatora padne ispod 0.69 u nekoj od točaka, znači da je u tom momentu, taj čija je vrijednost manja od navedene, u prednosti.

Za kraj, vraćamo se na zadnju kontrolnu točku i testiramo mrežu. Na slici 90. prikazan je poziv funkcija za navedeno.

```
!ls {checkpoint_dir}
checkpoint.restore(tf.train.latest_checkpoint(checkpoint_dir))
# Pokrećemo trenirani model na nekoliko primjeraka iz skupa za testiranje
for inp, tar in test_dataset.take(5):
  generate_images(generator, inp, tar)
```

Slika 90. *Pix2Pix* kod- vraćanje na zadnju kontrolnu točku

Testiranjem mreže dobijemo različita ‘predviđanja’ kao što možemo vidjeti na slici 91.



Slika 91. *Pix2Pix* kod - tri rezultata testiranja mreže

5 ZAKLJUČAK

Generativna umjetnost, vrsta kodne umjetnosti koja se zasniva na (djelomično) neovisnom sustavu, doživjela je veliku popularnost početkom 21. stoljeća. Kroz primjere i opise alata vidimo da se osim stvaranja umjetničkih slika, glazbe ili teksta, domena primjene generativne umjetnosti proširila na izradu animacija, terena za računalne igre, kopiranje stila sa slike, napredno uređivanje fotografija i videa, i slično.

Opisali smo *Processing*, programski jezik i okruženje nastalo upravo s ciljem stvaranja generative umjetnosti. Fokusirali smo se na modul za *Python* te kroz sedam vlastitih i pet postojećih primjera opisali smo najbitnije funkcije i metode koje *Processing* sadrži, a uvelike nam pomažu pri izradi generative umjetnosti.

Python-ova najpopularnija sfera je dubinsko učenje, te smo stoga u trećem djelu opisali poveznicu dubinskog učenja i generativne umjetnosti. Nakon što smo naveli popularne primjene, kao što je na primjer *Content-Aware Fill* te opisali što je točno GAN, naveli smo dva primjera s *TensorFlow*-a napisana u Pythonu, prijenos neuronskog stila te preslikavanje slika (*Pix2Pix*).

S obzirom na to da su ljudi dobro upoznati s dosta navedenih primjera možemo zaključiti da je generativna umjetnost već uzela maha u uobičajenom životu, barem na području zabave, multimedije te softvera za uređivanje fotografija/video snimaka, a s obzirom na široki spektar primjene očekujem daljnji rast i unaprjeđenje tog područja.

LITERATURA

[1] „Generative Art – Fascinating Things You Don't Know About", *Studio A N F*. [Mrežno] Dostupno na: https://studioanf.com/a-brief-history-of-generative-art/ (Pristupljeno 02. srpnja, 2021).

[2] „FRIEDER NAKE - 79 + 364", *DAM GALLERY*. [Mrežno] Dostupno na: https://damprojects.org/79plus364/?lang=en (Pristupljeno 02. srpnja, 2021).

[3] „Kinetic | Lillian F. Schwartz". [Mrežno] Dostupno na: http://lillian.com/kinetic/ (Pristupljeno 02. srpnja, 2021).

[4] „Interview With Generative Artist Jared Tarbell", *Artnome*. [Mrežno] Dostupno na: https://www.artnome.com/news/2020/8/24/interview-with-generative-artist-jared-tarbell (Pristupljeno 03. srpnja, 2021).

[5] „Features", *C4*. [Mrežno] Dostupno na: http://c4ios.com (Pristupljeno 19. srpnja, 2021).

[6] „C4SMOS", *App Store*. [Mrežno] Dostupno na: https://apps.apple.com/mn/app/c4smos/id985883701 (Pristupljeno 19. srpnja, 2021).

[7] „Processing Foundation". [Mrežno] Dostupno na: https://processingfoundation.org/ (Pristupljeno 19. srpnja, 2021).

[8] *1. Hello*. [Mrežno] Dostupno na: https://learning.oreilly.com/library/view/getting-started-with/9781457186820/ch01.html (Pristupljeno 30. srpnja, 2021.)

[9] „SOHCAHTOA Explained (19 Step-by-Step Examples!)", *Calcworkshop*. [Mrežno] Dostupno na: https://calcworkshop.com/triangle-trig/sohcahtoa/ (Pristupljeno 19. srpnja, 2021).

[10] „Motion basics: Difference between Cartesian and polar coordinate systems". [Mrežno] Dostupno na:

https://www.linearmotiontips.com/motion-basics-difference-between-cartesian-and-polar-coordinate-systems/ (Pristupljeno 19. srpnja, 2021.).

[11] „Randomization“. [Mrežno] Dostupno na: http://printingcode.runemadsen.com/lecture-randomization/ (pristupljeno svi. 19, 2021.).

[12] „Research / Pesquisa“. [Mrežno] Dostupno na: http://gallery.bananabanana.me/research/output/vera-molnar.html (Pristupljeno 31. srpnja, 2021.).

[13] „Color smoke - OpenProcessing“. [Mrežno] Dostupno na: https://openprocessing.org/sketch/408882 (Pristupljeno 03. rujna, 2021.).

[14] *Mutable ripples - OpenProcessing*. [Mrežno] Dostupno na: https://openprocessing.org/sketch/656607/ (Pristupljeno 03. rujna, 2021.).

[15] „How I drew a pencil-sketch-like forest with Processing | GenerativeART“.), G.-T. B. of Code. [Mrežno] Dostupno na: http://g-e-n-a-r-t.com/en/post.php?p=10 (Pristupljeno 04. kolovoza, 2021.).

[16] *Brush Drawing - OpenProcessing*. [Mrežno] Dostupno na: https://openprocessing.org/sketch/649427/ (Pristupljeno 07. rujna, 2021.).

[17] „This Person Does Not Exist“. [Mrežno] Dostupno na: https://thispersondoesnotexist.com/ (Pristupljeno 07. rujna, 2021.).

[18] M. Creativity, „Artificial Art: How GANs are making machines creative“, *Medium*. [Mrežno] Dostupno na: https://heartbeat.fritz.ai/artificial-art-how-gans-are-making-machines-creative-b99105627198 (Pristupljeno 12. kolovoza, 2021.).

[19] „Generative Adversarial Networks – Hot Topic in Machine Learning“, *KDnuggets*. [Mrežno] Dostupno na: https://www.kdnuggets.com/generative-adversarial-networks-hot-topic-in-machine-learning

.html/ (Pristupljeno 15. kolovoza, 2021.).

[20] „Nvidia's Gaugan tech turns simple colour blocks into realistic landscapes", *Digital Arts*. [Mrežno] Dostupno na: https://www.digitalartsonline.co.uk/news/creative-software/nvidias-gaugan-tech-turns-simple-colour-blocks-into-realistic-landscapes/ (Pristupljeno 16. kolovoza, 2021.).

[21] „AlphaGAN: Generative adversarial networks for natural image matting", S. Lutz, K. Amplianitis, i A. Smolic, 2014.

[22] „Investigating Tensors with PyTorch", *DataCamp Community*. [Mrežno] Dostupno na: https://www.datacamp.com/community/tutorials/investigating-tensors-pytorch (Pristupljeno 19. kolovoza, 2021.).

[23] „Keras documentation: About Keras", K. Team. [Mrežno] Dostupno na: https://keras.io/about/ (Pristupljeno 31. kolovoza, 2021.).

[24] „Google Colaboratory". [Mrežno] Dostupno na: https://colab.research.google.com/drive/1v-sS347JBN5yRPXJDGwZw7Qt4ooFefi-#scrollTo=7BkGc20Rfan6 (Pristupljeno 23. kolovoza, 2021.).

[25] „Google Colaboratory". [Mrežno] Dostupno na: https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/generative/pix2pix.ipynb#scrollTo=oeq9sByu86-B (Pristupljeno 24. kolovoza, 2021.).

[26] , „U-Net: Convolutional Networks for Biomedical Image Segmentation", O. Ronneberger, P. Fischer, i T. Brox. *2015.*

[27] „Image-to-Image Translation with Conditional Adversarial Networks", P. Isola, J.-Y. Zhu, T. Zhou, i A. A. Efros. *ArXiv161107004 Cs*, 2018. [Mrežno] Dostupno na: http://arxiv.org/abs/1611.07004 (Pristupljeno 24. kolovoza, 2021.).

[28] „Keras documentation: Adam", K.Team. [Mrežno] Dostupno na: https://keras.io/api/optimizers/adam/ Pristupljeno 28. kolovoza, 2021.).

POPIS SLIKA

IZJAVA

Izjavljujem pod punom moralnom odgovornošću da sam diplomski rad izradila samostalno, isključivo znanjem stečenim na studijima Sveučilišta u Dubrovniku, služeći se navedenim izvorima podataka i uz stručno vodstvo mentora izv.prof.dr.sc. Maria Miličevića, kome se još jednom srdačno zahvaljujem.

Nikolina Šanovsky