Volume 0 (1981), Number 0 pp. 1–4 COMPUTER GRAPHICS *forum*

Natural neighbor interpolation of images and volumes

Jan Živković

Univerza v Ljubljani Fakulteta za računalništvo in informatiko, Slovenia

Abstract

V seminarski nalogi se prikaže način generiranja slike z le nekaj vzorci točk in pripadajočo barvo na teh točkah. Glede na te podatke se zgenerira voronoi graf in nato za vsako pixel slike izračuna barva, da se lahko nato izriše slika, ki je bila zgenerirana s podanimi podatki. Slika je z majhnim številom vzorcev bolj 'zamegljena', z večjim številom vzorcev pa je vedno bolj podobna originalu. Opisana je samo 2D implementacija z optimizacijami, 3D implementacije pa ni implementirane.

1. Uvod

V seminarski nalogi je predstavljen način za naravno interpolacijo sosedov v 2D prostoru z slikami in 3D prostoru u volumni. S tem lahko z nekaj vzorci barv in lokacijo tega vzorca(x, y v 2D in x, y, z v 3D) preračunamo barvo z naravno interpolacijo.

2. Voronoi graf

Na začetku potrebujemo voronoi graf, ki ga zgeneriramo s pomočjo paketa scipy. Grafu se poda vzorčne točke, iz katerih nato izračuna voronoi regije. Primer voronoi grafa z neskončnimi robovi je prikazan na sliki 1.

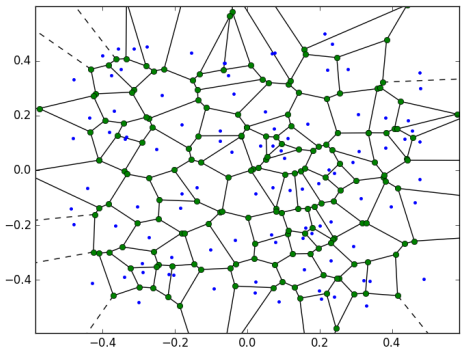

Figure 1: *Voronoi graf z neskončnimi robovi.*

2.1. Omejevanje grafa v 2D prostoru

V 2D implementaciji je bil problem, da je generirana slika imela črne robove, kjer je voronoi graf izračunal regije z neskončnimi robovi, kot je to razvidno na sliki 1. Zaradi tega je izračun barve vrnil črno barvo. Primer zgenerirane slike s črnimi robovi je na sliki 2.

Figure 2: *300x300 slika z 10000 vzorci s črnimi robovi.*

Graf je bilo potrebno torej omejiti, oziroma zgenerirati tako, da so vse točke, ki se nahajajo na sliki v zaprtih voronoi regijah in ne odprtih. Na začetku je bila ideja da ročno dodamo dodatne točke izven velikosti slike, a se ta način ni obnesel. Kasneje sem našel boljši način na 'stackoverflow', ki vzorčne točke preslika čez meje slike, torej če je slika velikosti 300x300, preslika točke v 4 smeri:

- preslika vse originalne vzorčne točke čez x=0
- preslika vse originalne vzorčne točke čez x=300
- preslika vse originalne vzorčne točke čez y=0
- preslika vse originalne vzorčne točke čez y=300

Primer zgeneriranega voronoi grafa s preslikanimi točkami je na sliki 3. S tem se število točk s katerimi se izračuna voronoi graf poveča za 5x, kot tudi čas izračuna grafa, a je le ta dokaj zanemarljiv(kot če bi sami med izvajanjem omejevali graf).

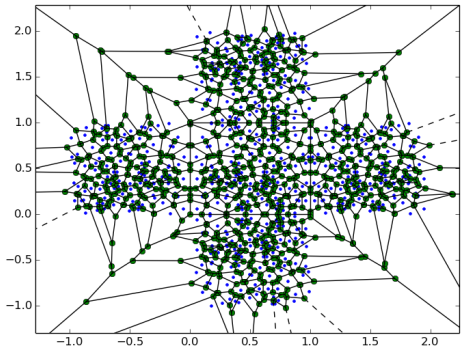

Figure 3: *Omejen voronoi graf s preslikanimi točkami.*

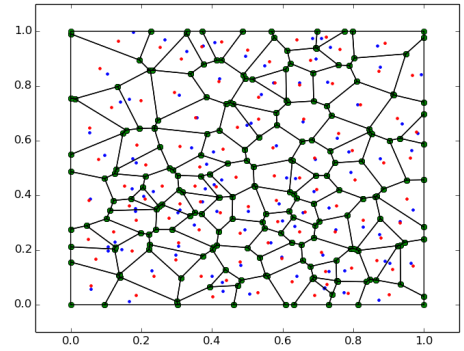

Figure 4: *Voronoi graf z omejenimi robovi.*

2.2. Omejevanje grafa v 3D prostoru

Po vsem raziskovanju, bi na isti problem kot na 2D naletel tudi pri 3D, le da bi tukaj moral preslikati točke čez vse ploskve volumna(oziroma čez bounding box volumna).

3. Implementacija

V sklopu seminarske naloge je potrebno implementirati 2D in 3D implementacijo. Tu je opisana in narejena le 2D implementacija, saj mi 3D ni uspelo implementirati.

3.1. 2D

Implementacijo sem izdelal v programskem jeziku Python3, za kar sem se odločil zato, ker ima že pakete za generacijo voronoi grafov, kot tudi za KDTree in olajša delo z algoritmi. Nekaj kode je tudi napisane v jeziku Cython [cyt], ki je kombinacija jezika Python in C, s čimer sem lahko pohitril dele kode.

Implementacijo sem začel z uporabo voronoi grafa, ki je opisan v 2 točki. Na začetku se z omejevanjem voronoi grafa nisem ukvarjal in sem le to pustil za zadnjo stvar. Ko sem imel voronoi graf izračunan sem začel z implementacijo funkcije za dodajanje točke v graf(prikazan v izseku kode 1). Algoritem deluje tako, da poišče najbližjo točko vstavljeni točki, nato izračuna pravokotni bisektor in najde robova(od regije najbližje točke), ki ju pravokotni bisektor seka. Pridobljena robova morata biti sortirana 'ccw'(s čimer povemo da generiramo robove vstavljene točke v nasprotni smeri urinega kazalca). Rob dobimo z kreiranjem črte, ki gre od točke kjer je sekan prvi rob, do točke kjer je sekan drugi rob. Sedaj ko imamo rob vstavljene točke, lahko izračunamo ploščino poligona, ki smo ga 'ukradli' najbližji točki. Ploščino bomo kasneje uporabili za izračun barve. Ta proces ponavljamo dokler ne zgeneriramo roba, ki robove sklene v zanko. Ko imamo zgeneriran celotni poligon, lahko izračunamo barvo vstavljene točke in le to vrnemo. Ta postopek se ponavlja toliko časa dokler ne izračunamo barve vseh pixlov slike. Na sliki 5 se vidi kako se izračuna ukraden voronoi poligon.

Pseudokoda algoritma je:

```
def add_point(self, point):
        list_of_area_and_color = []
        closest_point = self.get_closest_point(point)
        perpendicular_bisector = self.get_perpendicular_bisector(
                closest_point['point'], point
        )
        e1, e2 = self.get_edges_crossing_bisector(
                closest_point, perpendicular_bisector, point
        )
        list_of_area_and_color.append(
                self.calculate_area_taken((e1, e2), closest_point)
        )

        final_edge = e1
        prev_point = closest_point
        loop_counter = 0
        while not self.edge_equals(e2, final_edge):
                if loop_counter > 20: return 0
                other_point = self.get_other_point(e2, prev_point)
                perpendicular_bisector = self.get_perpendicular_bisector(
                        other_point['point'], point
                )
                e1, e2 = self.get_edges_crossing_bisector(
                        other_point, perpendicular_bisector, point
                )

                list_of_area_and_color.append(
                        self.calculate_area_taken((e1, e2), other_point)
                )
                prev_point = other_point
                loop_counter += 1

        return self.calculate_weighed_color(list_of_area_and_color)
```

Listing 1: *Pseudokoda algoritma za izračun barve točke*

3.1.1. Izračun barve za točko

Med izračunom regije za vstavljeno točko si tudi shranim barvo točke kateri sem 'ukradel' kos regije in pa samo ploščino ukradenega kosa regije. Nato se barva izračuna glede na procent ukradene ploščine poligona glede na celotno ploščino ukradenega poligona vstavljene točke. V izseku kode 2 je prikazana pseudokoda za izračun barve.

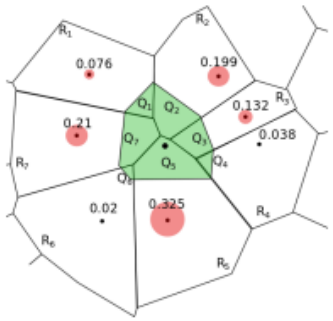

Figure 5: *Primer izračuna nove voronoi regije za vstavljeno točko.*

```
def calculate_weighed_color(list list_area_and_color not None, int n):
        cdef double full_area = 0
        cdef int i
        for i in range(n):
                full_area += list_area_and_color[i][0]
        cdef int r = 0, g = 0, b = 0
        cdef list t
        for i in range(n):
                t = list_area_and_color[i]
                percentage = t[0] / full_area
                r += t[1][0] * percentage
                g += t[1][1] * percentage
                b += t[1][2] * percentage
        cdef double output[3]
        output[0] = round(r)
        output[1] = round(g)
        output[2] = round(b)
        return output
```

Listing 2: *Pseudokoda algoritma za izračun barve točke*

Za izračun barve sem uporabil drugačen algoritem, kot je opisano v [Tsi16] z namenom hitrejšega izračuna, saj ni potrebno izračunati ploščine za vse regije vzorčnih točk. Razlika v izračunani barvi je minimalna in s prostim očesom neopazna, zato sem se odločil da uporabim svoj algoritem.

3.1.2. Optimizacije

V nalogi sem uporabil razne optimizacije, da bi program deloval kar se da hitro. Te optimizacije so naštete spodaj:

- Vračanje barve za točko, če je le ta točka ena izmed vzorčenih točk
- cKDTree (namesto naivne funkcije, ki gre čez vse točke da najde najbližjo točko iskani točki)
- Memoizacija (memoizacija funkcije za vračanje robov voronoi regije)
- Cython (optimizacija določenih funkcij z izvajanjem kode v C jeziku)

Funkcije optimizirane z Cython:

- get_line_intersection (največja optimizacija, iz 20 sekund na 0.7 sekunde pri 300x300 sliki z 10000 vzorčnimi točkami)
- ccw
- get_center_on_line
- get_perpendicular_bisector
- calculate_area_from_vertices
- calculate_weighed_color
- calculate_area_for_stolen_polygon

Vse optimizirane funkcije se nahajajo v datoteki 'optimized.pyx'. Za uporabo Cython kode je bilo potrebno namestiti razvojna orodja za Python in razvojna orodja za C/C++ jezik. Optimizirane funkcije se s Cython prevedejo v C kodo, katera se nato prevede v C program, ki ga Cython v Python kodo doda kot Python paket. Uporaba zgoraj naštetih optimizacij je izvajanje programa zmanjšala z približno 180 sekund na 'le' 21 sekund.

Nadaljne optimizacije bi lahko bile, da se program izvaja na grafični kartici z CUDA ali PYOpenCL vendar bi to potrebovalo prepis in reorganizacijo celotne kode, kot tudi veliko časa in potrpljenja.

3.1.3. Primeri izračunanih slik

Tu je prikazan primer izračunane slike z podano vhodno sliko.

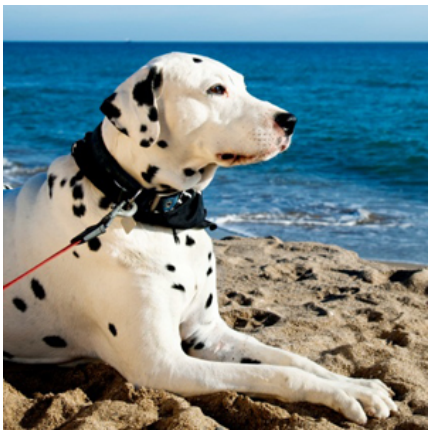

Figure 6: *300x300 slika na podlagi katere se izračunava.*

Figure 7: *300x300 slika narejena z 1000 vzorčnimi točkami.*

Figure 8: *300x300 slika narejena z 10000 vzorčnimi točkami.*

Figure 9: *1024x1024 slika na podlagi katere se izračunava.*

Figure 10: *1024x1024 slika narejena z 10000 vzorčnimi točkami.*

Figure 11: *1024x1024 slika narejena z 100000 vzorčnimi točkami.*

3.2. 3D

3D implementacija ni narejena, a bi po vsej verjetnosti bila narejena na isti način kot 2D, z razlikami kako bi sortirali točke in robove, namesto voronoi regij bi uporabljali voronoi poligone itd. To implementacijo bi bilo skoraj nujno implementirati na grafični kartici, saj bi se ta na procesorju izračunavala veliko počasneje.

4. Zaključek

Sama seminarska naloga je bila zelo zanimiva, celo bolj kot sem pričakoval. Implementacija naloge mi je vzela veliko časa, vmes sem imel veliko problemov, kot so recimo:

- napačno generiranje regije za vstavljeno točko v določenih primerih
- napačen izračun ploščine ukradenega poligona(včasih je izračunal ukraden kos poligona, včasih pa preostal kos poligona)
- napačen izračun barve(overflow modre barve v zeleno zaradi uporabe integer zapisa barve)

Samo debugiranje kode je bilo dokaj zahtevno, saj je do problema ponavadi prišlo globoko med izvajanjem, zato sem si ustvaril datoteko, kjer sem lahko imel vedno iste podatke za generiranje slike, jo naložil v program in tako vedno izvajal vstavljenje točke na kontroliranih podatkih, si izrisoval grafe in lažje našel napako.

Vsa izvorna koda celotnega seminarja je na voljo na spletnem naslovu [sou].

References

[cyt] Cython: C-extensions for python. URL: https://cython.org/. 2

[sou] Github repository with source code. URL: https://github.com/zivkovic/natural_neighbour_interpolation_FRI_NRG. 4

[Tsi16] TSIDAEV A.: Parallel algorithm for natural neighbor interpolation. *CEUR Workshop Proceedings 1729* (2016), 78–83. 3