Features from Accelerated Segment Test (FAST)
Algoritm pentru detectarea colțurilor

Adela Iosif

2024

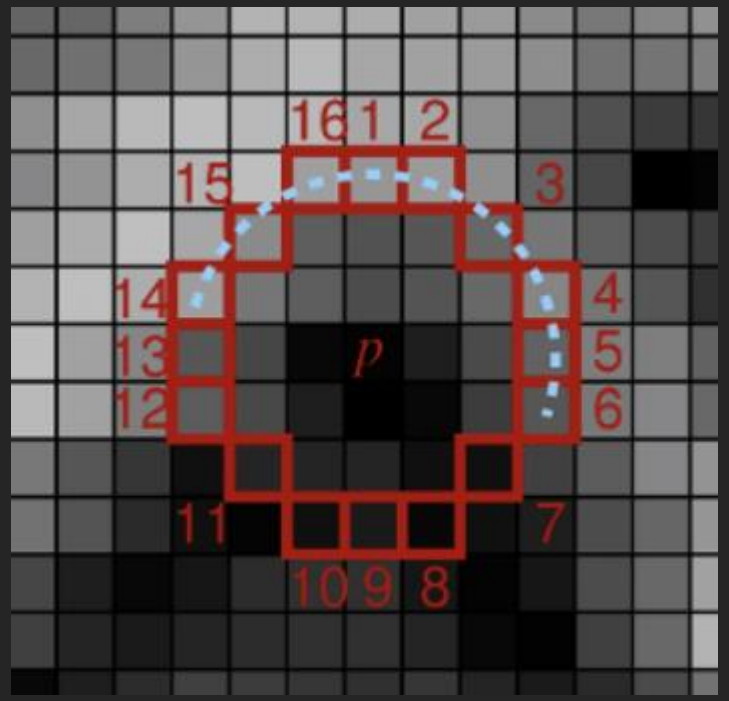
Descrierea proiectului

- Algoritmul FAST a fost propus în 2006 de Rosten și Drummond pentru a oferi o metodă rapidă și eficientă de detectare a colțurilor în imagini, fiind folosit în special în aplicații de vizualizare și urmărire a obiectelor în timp real.
- Proiectul meu implică identificarea cât mai precisă și rapidă a colțurilor din imagini grayscale.
- Detectarea acestor colțuri presupune salvarea lor într-un vector specific și colorarea lor pentru a fi cât mai vizibile.
- La intrare se primește o imagine digitală în tonuri de gri care conține obiecte pentru care se dorește detecția colțurilor.
- Ieșirea constă într-o nouă versiune a imaginii de intrare, având colțurile evidențiate pentru o vizualizare mai bună.

Research și publicații asemănătoare

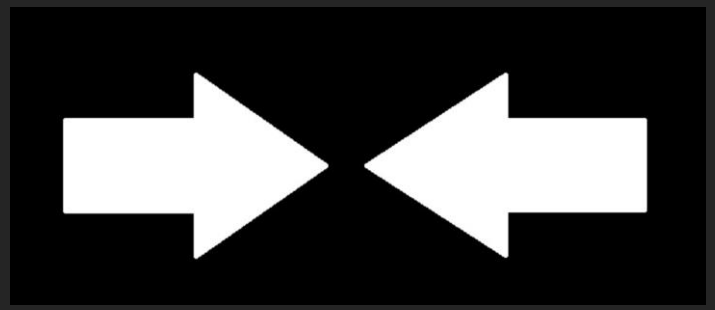
- Articolele de la care am pornit studiul acestei teme sunt: ***Machine learning for high-speed corner detection*** (Edward Rosten și Tom Drummond, Cambridge University, UK) și ***Sparse Least-Squares Support Vector Machines via Accelerated Segmented: a Dual Approach*** (Federal University of Ceará, Brazilia).
- În cadrul acestor articole, algoritmul FAST se abordează astfel:
  1. Se identifică regiunile de interes în imaginea de intrare (posibilele colțuri) și se numără pixelii care aduc variații de intensitate față de pixelul de test.
  2. Se verifică stabilitatea algoritmului – aceasta este dată de numărul de pixeli adiacenți care sunt semnificativ mai luminoși/întunecați decât pixelul de test.
  3. Se introduce un high-speed test pentru a exclude mai rapid foarte multe puncte care nu sunt colțuri: se examinează pixelii 1, 5, 9 și 13, iar p este colț doar dacă cel puțin 3 dintre aceste puncte respectă condiția de mai luminos/întunecat precizată mai devreme.
  4. Se determină colțurile și se marchează ca fiind colțuri în imaginea de intrare.

Abordare generală și arhitectura la nivel înalt

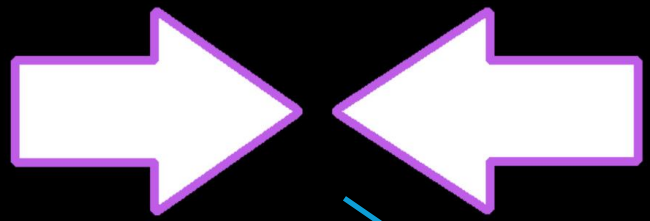


- Se ia un pixel p pentru a fi testat, cu intensitatea I_p
- Se consideră un cerc de 16 pixeli în jurul lui
- Se alege un prag t pentru a decide dacă diferența de intensitate între pixelul central și pixelii vecini este suficient de mare pentru a considera pixelul central colț; pragul se alege de obicei 20 (un prag prea mic poate duce la detectarea unui număr prea mare de colțuri, inclusiv detectări false – zgomot; iar un prag prea mare poate rata unele colțuri reale)
- Pixelul p este considerat colț dacă există un set de n pixeli consecutivi (de obicei, n = 12) de pe cerc care să fie mai luminoși decât I_p+t sau mai întunecați decât Ip-t

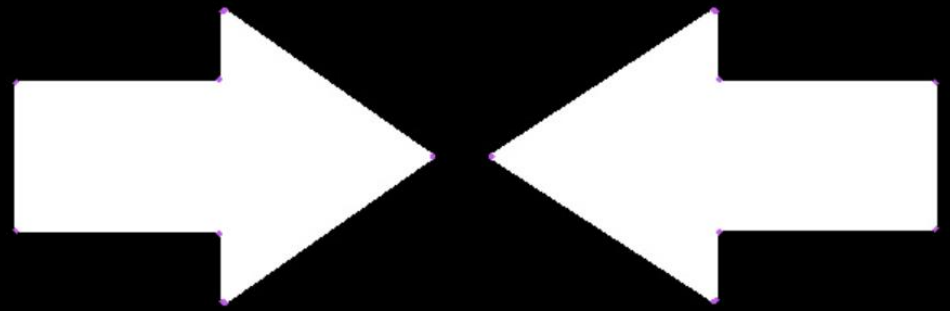
Abordare generală și arhitectura la nivel înalt



Zone în care se suspectează că ar fi colțuri, deoarece există o schimbare bruscă de intensitate în vecinătatea punctului (se consideră colțuri toți vecinii unde există schimbarea de intensitate).

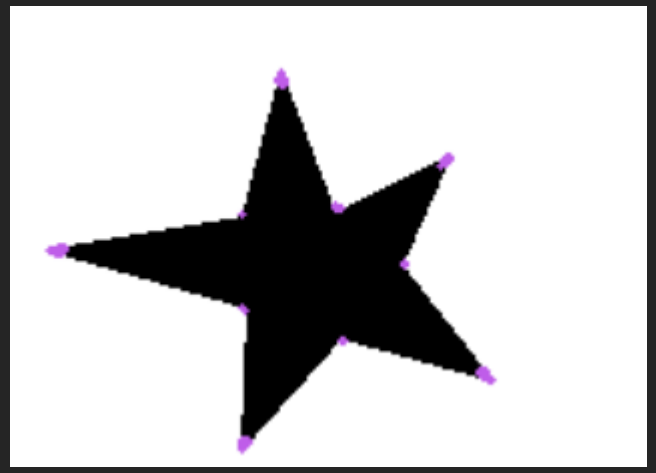


Imaginea după ce se face testul de stabilitate: se setează n = 12 pixeli consecutivi din vecinătate care trebuie să fie semnificativ mai luminoși/întunecați decât pixelul de test, pentru ca acesta să fie colț.



Abordare generală și arhitectura la nivel înalt

În abordarea în care se caută 12 pixeli adiacenți din vecinătate care să fie mai luminoși/întunecați, s-au găsit 60 de puncte colț.



Imaginea după testul high-speed

În abordarea mai rapidă în care se caută ca trei dintre pixelii 1, 5, 9 sau 13 din vecinătate să existe variații mari de intensitate, s-au detectat 63 de puncte colț (pentru același prag=20).

Detalii de implementare

- Se creează o funcție auxiliară pentru testarea dacă un pixel din imagine este colț.
- Se declară 2 vectori de deplasament (pe axele X și Y), aceste deplasamente fiind declarate în ordine trigonometrică (pentru a asigura că verificările ulterioare se fac pe pixeli consecutivi).
- Voi avea 2 variabile întregi pentru numărarea pixelilor mai deschiși, respectiv mai închiși decât pixelul de test.
- Cu o buclă for voi verifica toți cei 16 pixeli din jurul pixelului ales – de la poziția (rând+deplasamentX[i], coloană+deplasamentY[i]) dacă sunt semnificativ mai închiși, sau mai deschiși decât pixelul din mijloc.
- Dacă se găsesc n astfel de pixeli, atunci funcția va returna adevărat, iar în caz contrar, pixelul nu e considerat colț, deci funcția va returna fals.

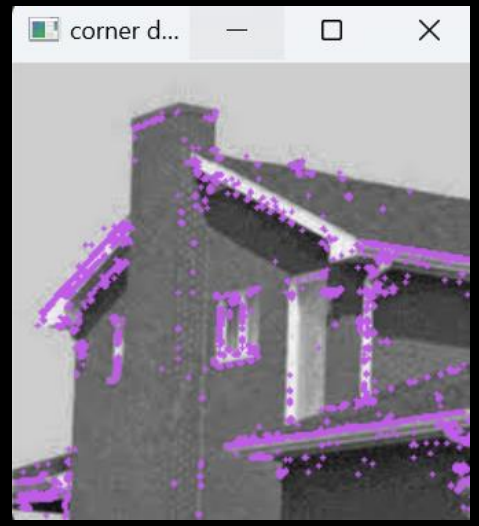
Detalii de implementare

- Odată ce am această funcție care verifică dacă un punct este sau nu colț, voi trece la implementarea propriu-zisă a algoritmului.
- Astfel, voi avea un vector de puncte în care voi reține pixelii detectați ca fiind colțuri, iar acest vector va fi returnat din funcția care implementează algoritmul FAST.
- Mai este nevoie de 2 variabile simple, una pentru prag și una pentru numărul consecutiv de pixeli, care vor fi transmise ca parametri în funcția de testare a colțurilor.
- În final, se parcurge imaginea și se adaugă colțurile în vectorul specific.
- Tot ce a mai rămas de făcut este marcarea pe imagine a acestor pixeli colț.
- Pentru o mai bună vizualizare, voi copia (în programul principal) imaginea grayscale într-o nouă imagine color (cu ajutorul funcției cvtColor) și voi desena puncte roz acolo unde s-au identificat colțurile din imagine.

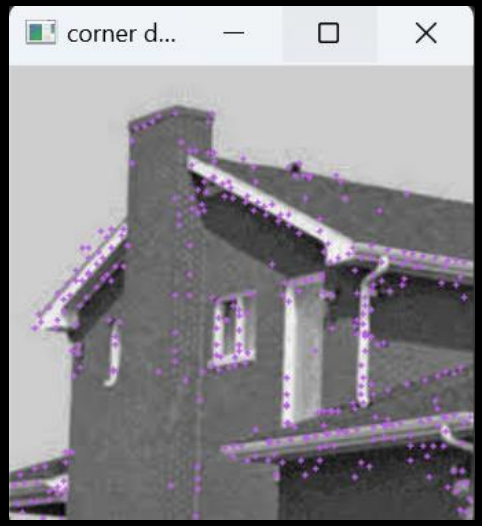
Detalii de implementare – alegerea maximelor dintr-o zonă

Am adăugat o etapă în care se aleg doar maximele dintre punctele detectate ca și colțuri. Pentru fiecare punct găsit, verific în vecinătatea lui dacă există altele și păstrez (în vectorul points_improved) doar punctele considerate maxime în zona lor.

Elimin punctele care sunt prea apropiate de altele, păstrând punctul-colț cu cel mai mare scor. Îmbunătățirea se observă mai ales la imagini din lumea reală.



1441 pixeli colț găsiți inițial

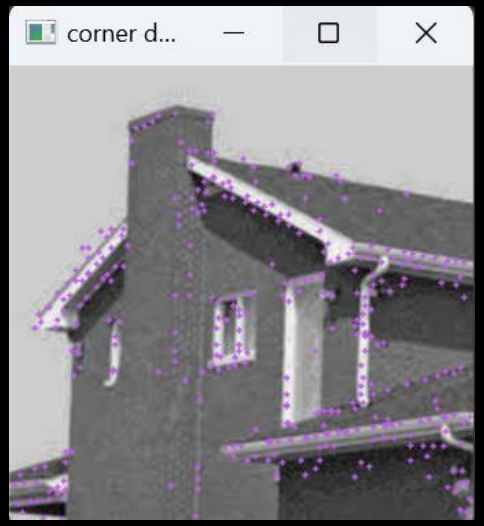


185 pixeli găsiți după eliminarea non-maximelor din zonă

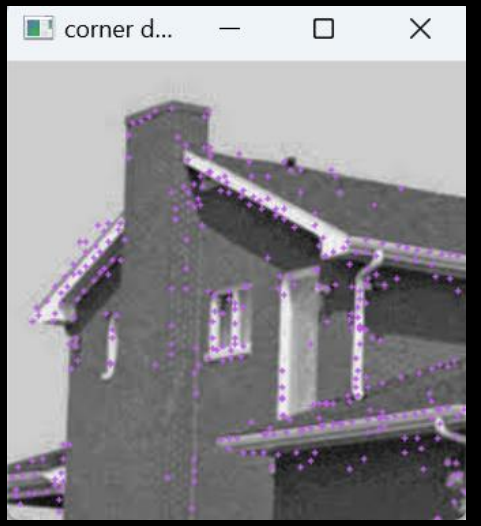
Detalii de implementare – verificare circulară a vecinilor

- Am încercat să verific într-o manieră circulară continuă pixelii vecini, continuând căutarea chiar dacă s-a ajuns la vecinul nr. 1 deja explorat.
- Astfel, verific și dacă, spre exemplu, pixelii cu nr. 13, 14, 15, 16, 1, 2, 3, 4, 5, 6, 7, 8 formează cei 12 pixeli consecutivi mai întunecoși/luminoși.
- De aceea, în loc să merg cu bulca for de la 0 la 15 cum o făceam înainte, acum merg de la 0 la 26 (inclusiv) – am adăugat 11, deoarece ultima variantă posibilă necuprinsă înainte ar fi pixelul 16 cu pixelii 1..11. Pentru asta, când accesez vecinul din deplX și deplY, voi face i%16 (pentru că for-ul începe de la 0, iar pe cerc am numerotarea de la 1 la 16; astfel, pixelul 1 de pe cerc, este pixelul de la i=0).

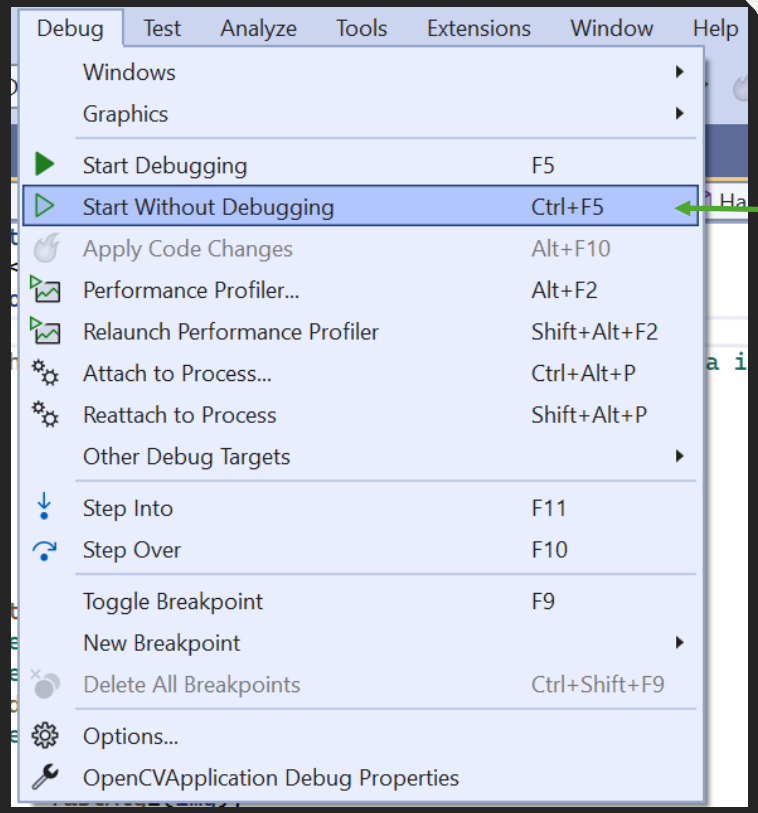
Am adăugat o verificare circulară a pixelilor din vecinătatea pixelului candidat, deci algoritmul a găsit mai mulți pixeli-colț.



185 pixeli găsiți înainte de verificarea circulară a pixelilor



330 pixeli găsiți după verificarea circulară



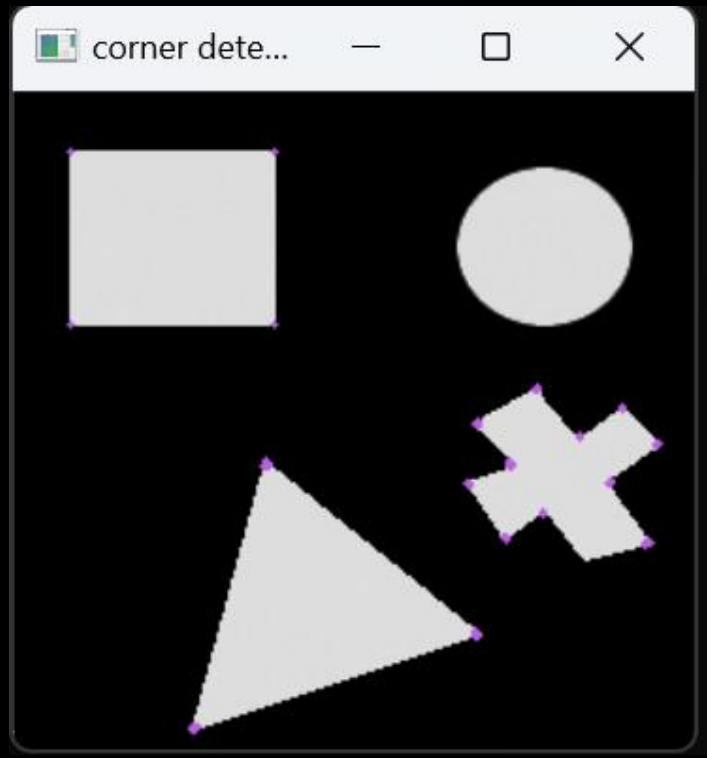
Manual de utilizare

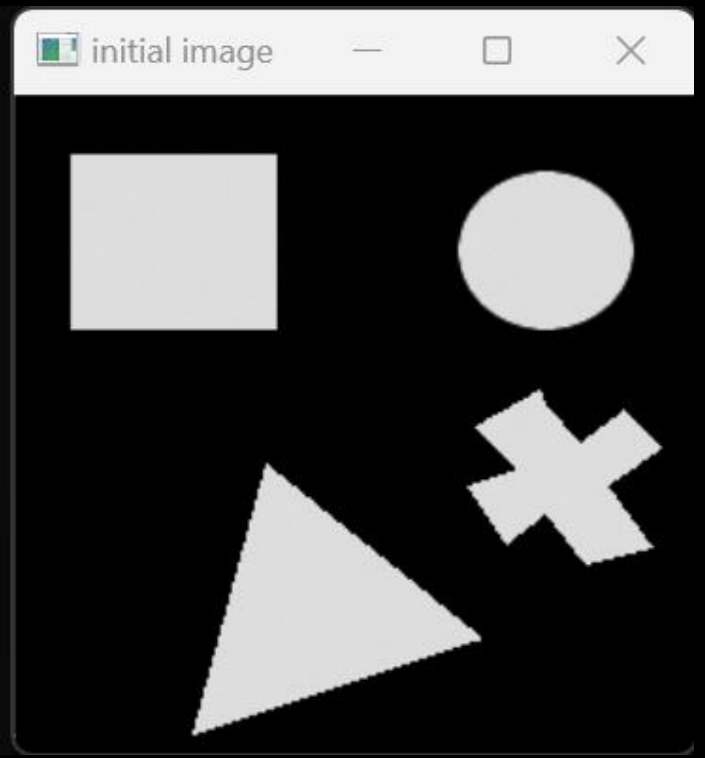
- Proiectul se poate porni din Visual Studio 2022, fără probleme.
- Se vor identifica colțurile din imaginea dată ca intrare în programul principal (a se schimba calea dacă aceasta nu coincide).

```
Mat_<uchar>img = imread("Images/star.bmp", IMREAD_GRAYSCALE);
```

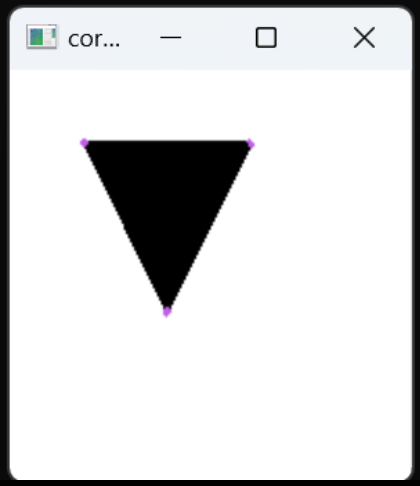
Demonstrarea rezultatelor

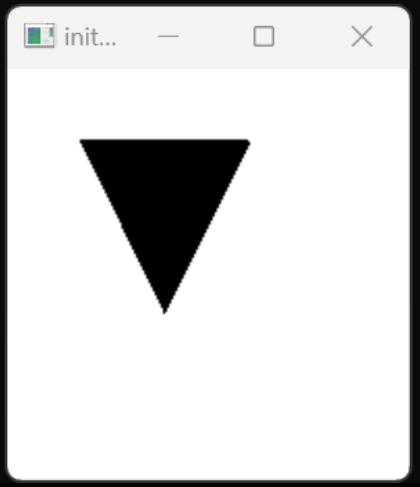
În imagini se pot observa pixelii de colț colorați în roz, iar la imaginea de jos, se pot vedea și coordonatele pixelilor detectați.





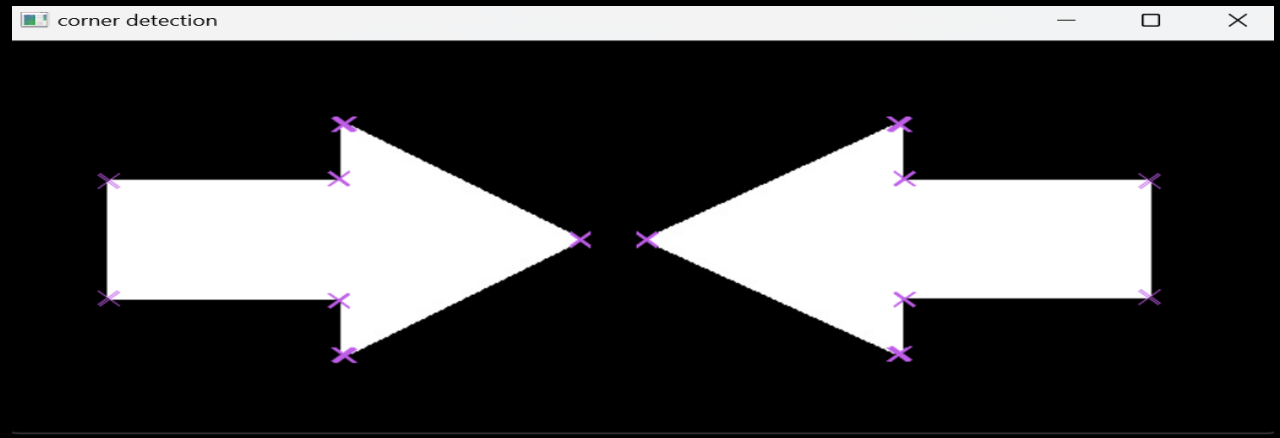

```
35.000000 34.000000
36.000000 34.000000
115.000000 34.000000
35.000000 35.000000
36.000000 35.000000
116.000000 35.000000
116.000000 36.000000
75.000000 115.000000
76.000000 115.000000
75.000000 116.000000
76.000000 116.000000
75.000000 117.000000
```

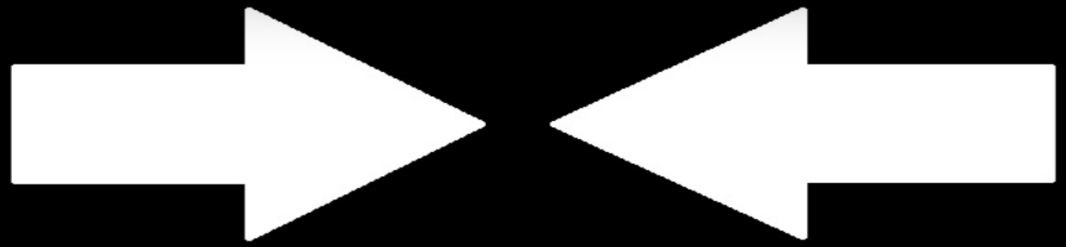


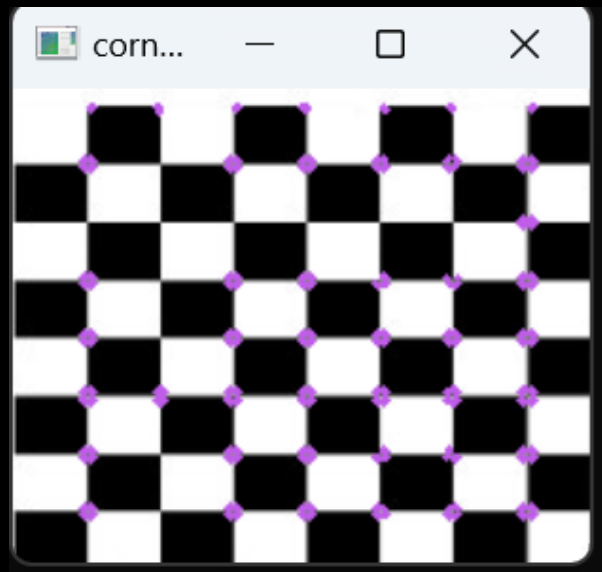


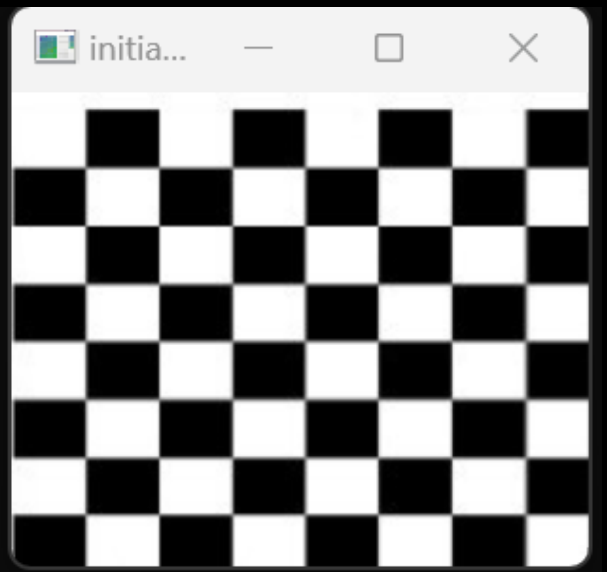
Demonstrarea rezultatelor

Imaginile nu sunt perfecte, există zgomot, de aceea sunt mai multe puncte de colț găsite, dar rezultatele sunt apropiate de cele reale.





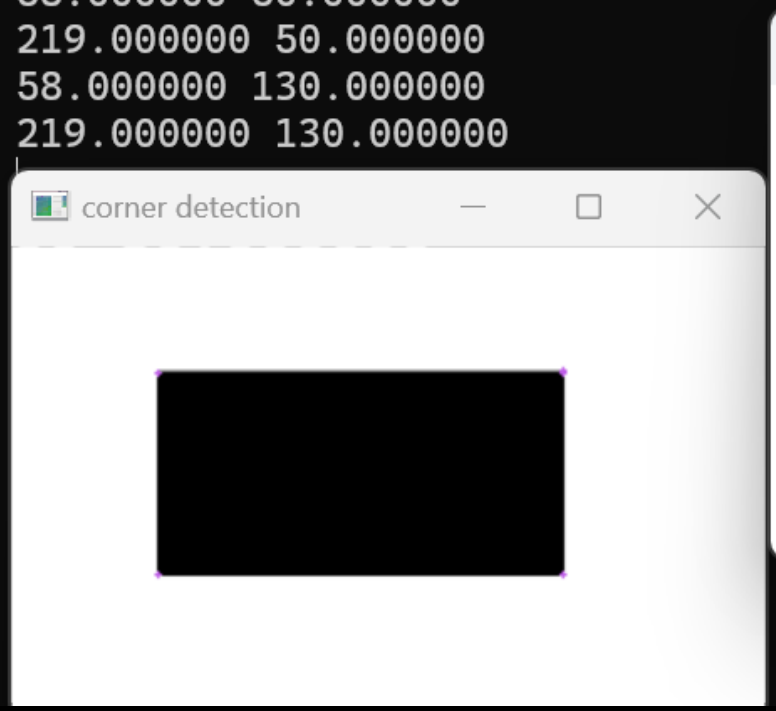


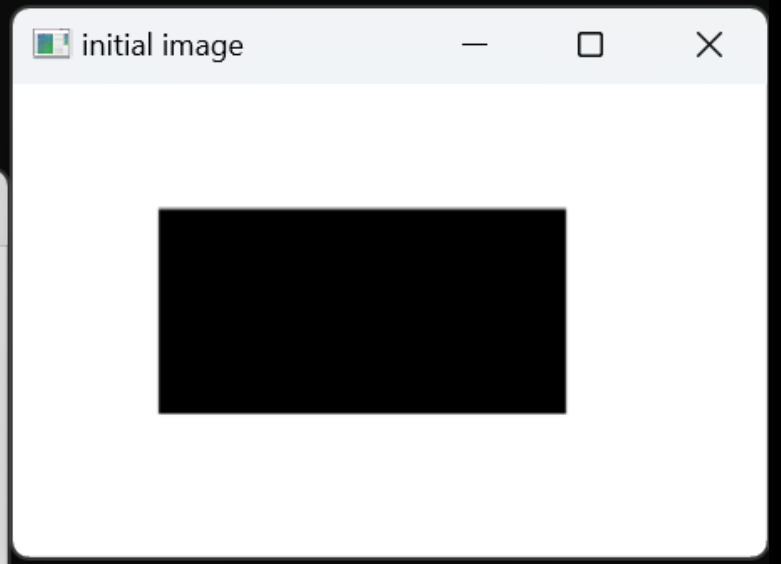


Demonstrarea rezultatelor

Totuși, mi-am creat o imagine în Paint, unde am adăugat un dreptunghi, iar dacă aleg numărul de vecini consecutivi n=11, algoritmul îmi va detecta exact cele 4 colțuri ale dreptunghiului.

```
219.000000 49.000000
58.000000 50.000000
219.000000 50.000000
58.000000 130.000000
219.000000 130.000000
```





High-speed test

- Pentru excluderea unui număr mai mare de ne-colțuri
- Acest test verifică numai patru pixeli de pe cercul Bresenham din jurul lui p: 1, 5, 9 și 13
- Dacă cel puțin 3 dintre acești pixeli sunt semnificativ mai luminoși/întunecați, p este colț

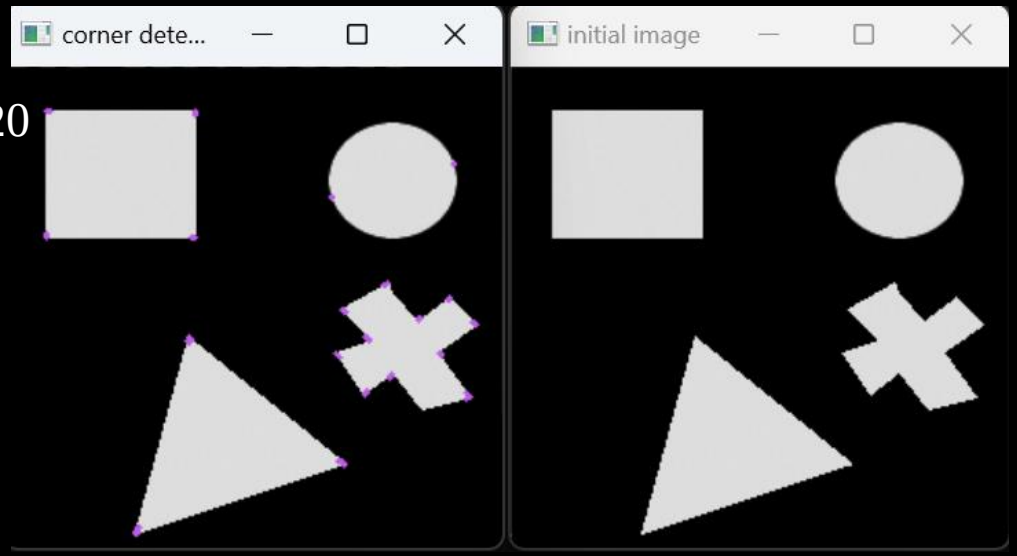
Am aplicat acest test astfel:

- Am creat o funcție asemănătoare cu cea inițială, pentru verificare dacă un pixel este colț
- La început, verific dacă pixelii 1 și 9 aparțin intervalului și dacă nu sunt mai luminoși sau mai întunecați, iar dacă se îndeplinesc condițiile, atunci pixelul e clasificat ca ne-colț (funcția returnează fals)
- Dacă funcția nu a returnat fals, verifică toți cei 4 pixeli (1,5,9 și 13), dacă sunt fie mai luminoși, fie mai întunecați
- Dacă minim 3 dintre aceștia satisfac condiția, pixelul este colț

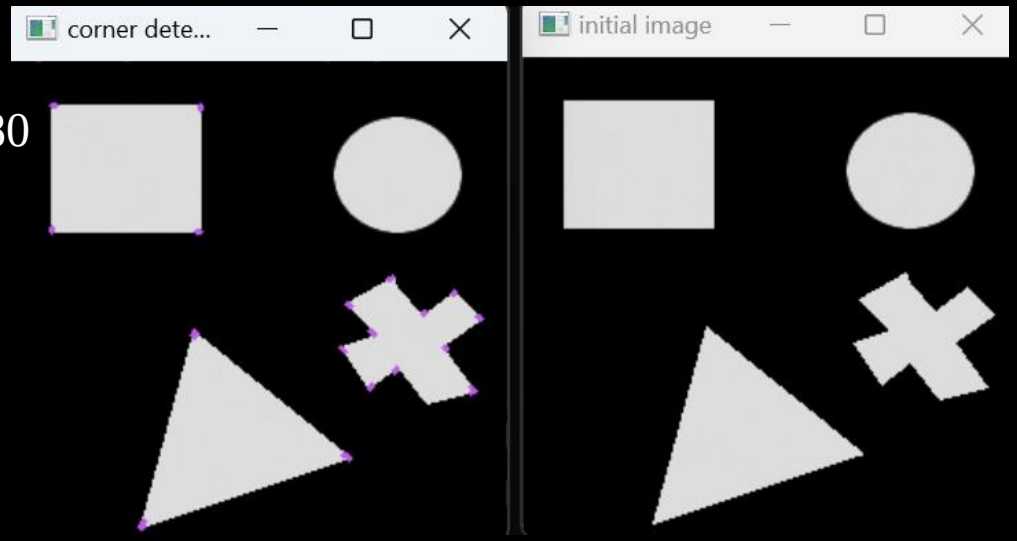
- Deși acest test accelerează procesul de detecție, acest algoritm detectează mai multe puncte colț decât înainte, dacă las același prag = 20 (variațiile locale de intensitate pot induce în eroare testul de mare viteză).
- Totuși, dacă cresc puțin pragul și nu îl las atât de permisiv, rezultatele sunt îmbunătățite.

High-speed test

Rezultate pentru prag=20



Rezultate pentru prag=80



High-speed test – îmbunătățire

- Am cronometrat ambele forme de implementare, rezultatele fiind cele așteptate.
- Testul de mare viteză accelerează procesul de detecție, prin eliminarea rapidă a pixelilor ne-colțuri.
- Fără high-speed test se fac, în cel mai rău caz, 16 comparații pentru fiecare pixel candidat, iar cu testul de mare viteză se fac aproximativ 4 comparații, ceea ce duce la economisirea de timp și resurse.

Rezultate obținute la măsurarea timpilor, pentru diferite imagini:

```
Timp pentru segment test detector = 203.147 [ms]

Timp cu high speed test = 42.906 [ms]
```

```
Timp pentru segment test detector = 212.367 [ms]

Timp cu high speed test = 51.904 [ms]
```

```
Timp pentru segment test detector = 175.572 [ms]

Timp cu high speed test = 30.902 [ms]
```

Concluzii

- În forma sa neîmbunătățită, FAST poate detecta multe colțuri false în prezența zgomotului.
- Algoritmul este unul rapid și simplu de implementat, dar poate suferi la precizia detectării.
- Prin introducerea testului de mare viteză reduce numărul de comparații, deci algoritmul devine mai rapid (se identifică mai rapid punctele care nu sunt colțuri și se elimină din procesarea ulterioară).