Features from Accelerated Segment Test

2016010873 박정욱

목차

서론

- 기존에도 SIFT, Harris, SUSAN 등 고성능 특징 검출기들이 있음

- 하지만 이들은 SLAM 등의 실시간 처리에 쓰이기엔 너무 '계산 집약적'
  - 계산 집약적 문제 : 컴퓨터로 해결하기에 시간과 자원이 너무 많이 드는 문제

- 특수한 경우가 아니라면 실시간 처리가 불가능하다고 알려져있음
  - 별도의 하드웨어 처리가 거의 필수적
  - GPU, FPGA(Field Programmable Gate Array), DSP(Digital Signal Processor), etc....

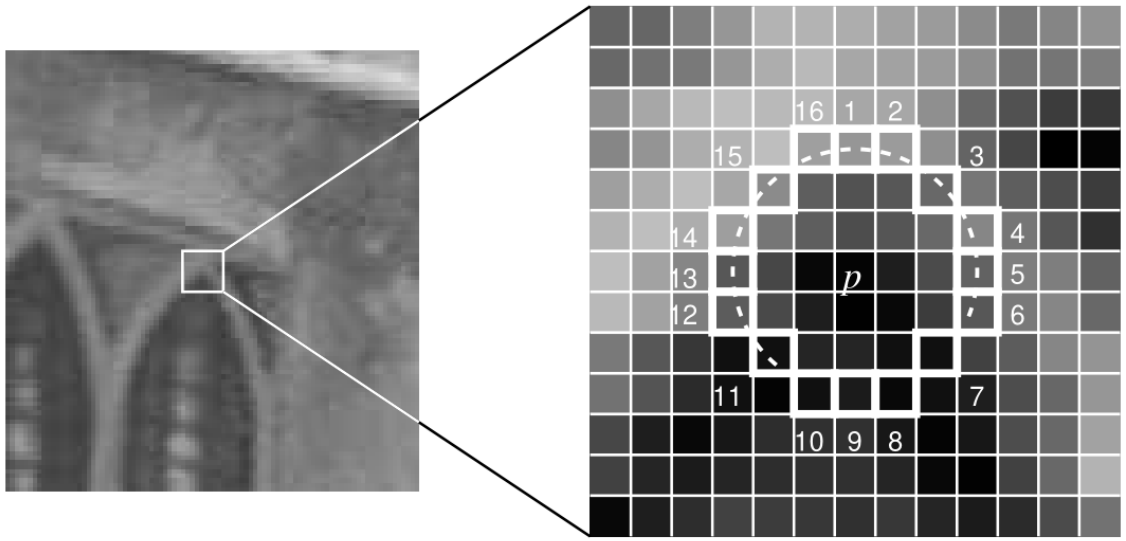
- 실시간 처리를 위한 새로운 특징 검출기가 필요!

서론

- 번역해보자면 “가속화된 세그먼트 검사로부터의 특징”
- 이름에 맞게, 특징을 검출할 때 영상의 특정 **부분**에 대해 **가속화**된 **검사**를 진행
- 기존의 SIFT와 Harris는 화소 강도의 **순간변화율**을 이용하는 방법론
  - 화소 강도 : 회색조 영상에서 밝기를 나타내는 정도. 흔히 8비트 정수로 표현됨
- SUSAN의 경우, 선택된 화소 주변에 비슷한 강도의 화소가 얼마나 **적은지**를 기준으로 함
- 계산이 필요한 변화율이 아니라 눈에 보이는 **지역적 특성**을 이용한다는 발상에 주목

고속 특징점 검출

- 특징점 후보인 화소 p를 중심으로 하는 Bresenham 원을 생각해보자.
  - Bresenham 원 = 중간점 원 그리기 알고리즘 + Bresenham 선 그리기 알고리즘
  - 실수 연산이 아닌 정수 연산을 이용하므로 압도적으로 효율적임



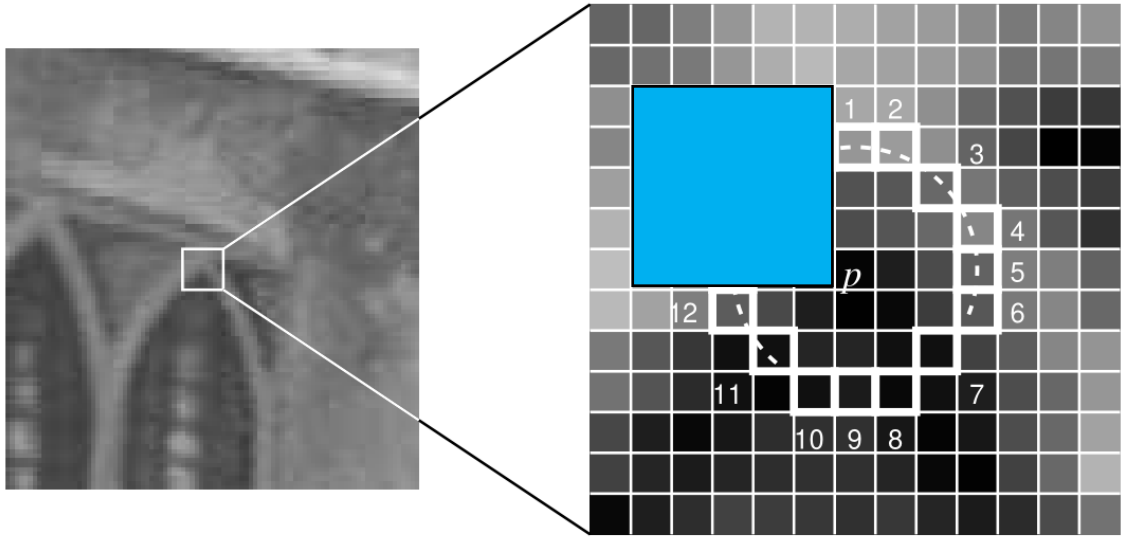
고속 특징점 검출

- 원 위의 화소 $x \in \{1..16\}$ 의 강도값 $I_{p \to x}$에 대해, 다음이 성립(단, t는 임의의 임계값)
- $I_p + t \leq I_{p \to x} \Rightarrow brighter,\ I_p - t \geq I_{p \to x} \Rightarrow darker$
- 이 때, n개의 화소가 **연속적**으로 p보다 밝거나 어둡다면 p는 특징점으로 분류됨
- 또한, n의 개수에 따라 알고리즘의 이름이 FAST-n과 같이 정해짐
  - FAST-9, FAST-12, etc….

고속 특징점 검출

- **가속화된 검사**(이후 '**검사**'로 표기)를 하기 위해 $n = 12$로 잡고, 4방위 화소 4개만 검사
  - 연속된 화소가 12개 존재하려면 **적어도 3개**가 $I_p + t$보다 밝거나 $I_p - t$보다 어두워야 함

- 이후 일차적으로 걸러진 특징점 후보군에 대해 전체 검사를 진행



약점 및 그 보완

- **검사**를 진행하는 이 알고리즘엔 몇 가지 약점이 존재

1. **검사**는 $n < 12$일 경우 일반적으로 적용하기 힘듦
   - 특징점 검출 조건이 비교적 느슨해지기 때문에, **검사** 이후 전체 검사 시 제외되는 후보군이 적어짐

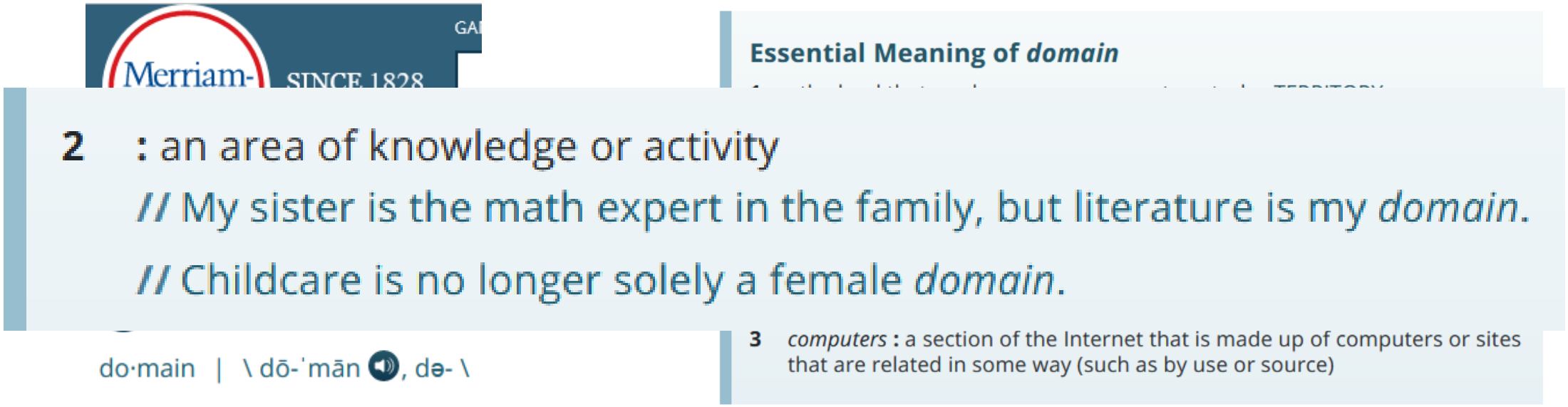
2. **검사**의 속도가 특징점의 분포 형태에 따라 결정됨
   - 어떤 화소를 우선적으로 검사할지? 어떤 방향으로 돌아가며 검사할지?

3. **검사** 결과는 후보군 제외 후 버려짐

4. 인접한 지역에서 여러 개의 특징점이 검출됨

약점 및 그 보완

- **1, 2, 3**번 약점을 보완하기 위해, 기계학습 도입
- 목표 **도메인**에 응용하기 위한 학습 영상을 준비



약점 및 그 보완

- n값과 적절한 임계값 t를 정해, 전체 검사를 진행하여 특징점을 추출

- 밝기 검사 식을 원 위의 모든 점 $x \in \{1..16\}$에 대해 적용할 경우:

$$S_{p\to x} = \begin{cases} d, & I_{p\to x} \le I_p - t & \text{(darker)} \\ s, & I_p - t < I_{p\to x} < I_p + t & \text{(similar)} \\ b, & I_p + t \le I_{p\to x} & \text{(brighter)} \end{cases}$$

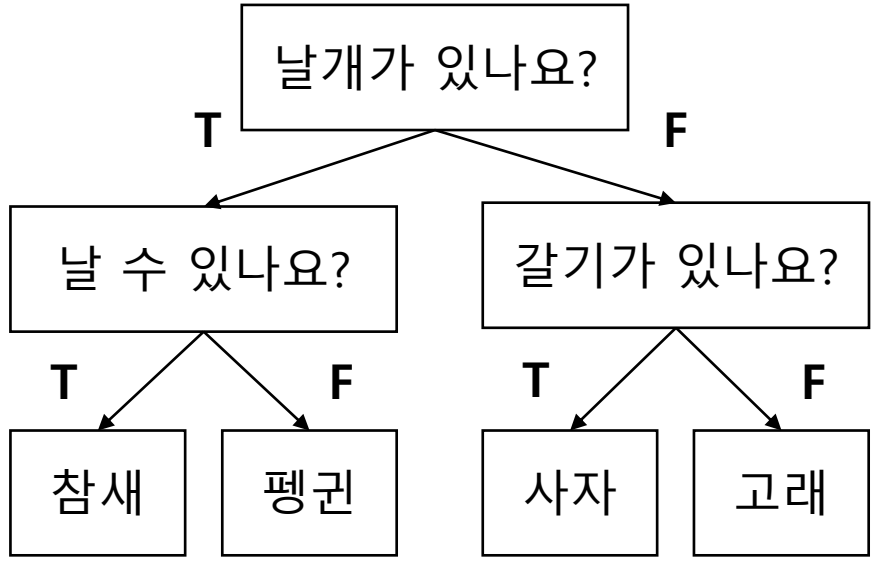
1. 임의의 화소 $p \in P$에 대한 16차원 벡터로 표현 가능

2. 전체 화소 집합 P는 세 부분집합 P_d, P_s, P_b로 **분할**됨(서로 같은 원소 없음)

ID3 알고리즘을 이용한 결정 트리

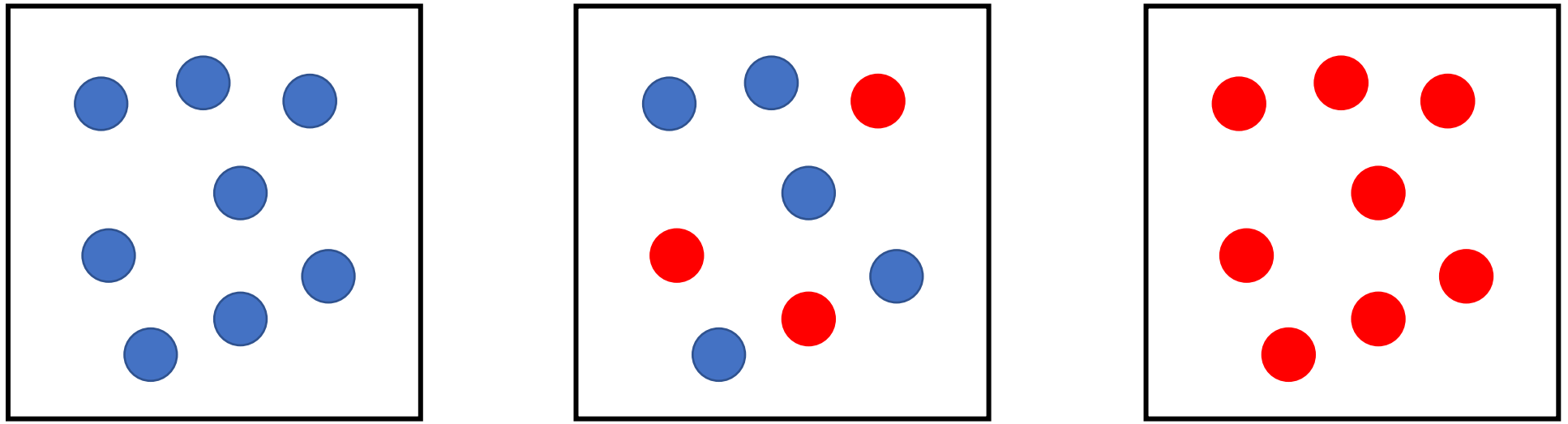
- 결정 트리
  - 결정 규칙과 그 결과를 트리 구조로 도식화한 도구
  - 스무고개 같은 형태를 생각하면 이해하기 편함

날개?	비행?	갈기?	동물
O	O	X	참새
O	X	X	펭귄
X	X	X	고래
X	X	O	사자



ID3 알고리즘을 이용한 결정 트리

- 결정 트리는 **불순도**가 낮아지는 방향으로



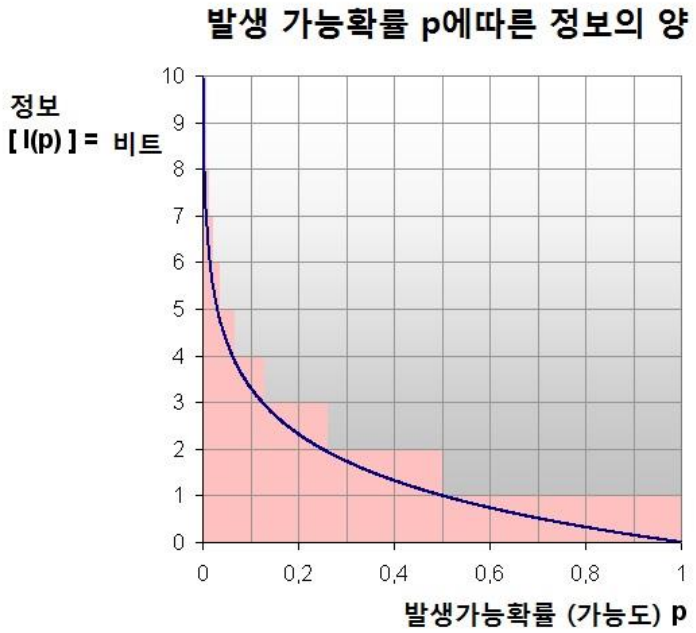
- **엔트로피**는 불순도를 수치화한 지표 중 하나이며, **정보량**의 기댓값

ID3 알고리즘을 이용한 결정 트리

- 사람은 많이 알고 있을수록 덜 놀란다!
- 일어나기 힘든 일을 알기 위해서는 많은 정보가 필요하다!
- = 정보량은 발생가능확률에 반비례
- 따라서 정보량을 다음과 같이 정의:

$$I(E) = \log_a \frac{1}{P(E)} = -\log_a P(E)$$

- 정보 이론에서 로그의 밑 a는 일반적으로 2를 사용
  - $P(E)$는 사건 E가 일어날 확률



ID3 알고리즘을 이용한 결정 트리

- 이산 확률 변수의 기댓값:

$$E(X) = \sum_i p_i x_i$$

- 정보량의 기댓값 – **엔트로피**:

$$H(x) = E\big(I(x)\big) = -\sum_i p(x_i) \log_2 p(x_i)$$

- 엔트로피는 불순도와 비례 관계
  - 불순도를 수치화한 지표
  - 기댓값이란 어떤 사건에 대한 평균으로 생각할 수 있으므로, 엔트로피 또한 평균 정보량으로 생각 가능

ID3 알고리즘을 이용한 결정 트리

- 결정 트리의 성능을 높이려면 엔트로피를 줄여야 함!
- ID3 알고리즘은 이를 위해 **정보 이득**이라는 개념을 도입
- 결정 트리에 따라 데이터를 세부 클래스로 분류할 때, 엔트로피는 점점 줄어듦
  - 정확한 클래스를 고를 확률이 점점 높아짐
- 정보 이득 = 상위 클래스일 때의 엔트로피와 세부 클래스일 때의 엔트로피의 차이

$$gain(A) = H(X) - H(X|A)$$

- 결국 ID3는 **정보 이득이 가장 큰 것**을 기준으로 데이터를 이분하여 분류하는 알고리즘!

ID3 알고리즘을 이용한 결정 트리

- **어떤 순서로 화소를 검사**하는 것이 좋을지에 대한 **결정 트리 생성**을 위해 ID3 적용!

- 화소 p의 특징점 여부를 저장하는 Boolean 변수 K_p의 집합 P에 대한 엔트로피:

$$
\begin{aligned}
H(P) &= (c+\bar{c})\log_2(c+\bar{c}) - c\log_2 c - \bar{c}\log_2 \bar{c} \\
\text{where} \quad c &= \left|\{p|K_p \text{ is true}\}\right| \quad \text{(number of corners)} \\
\text{and} \quad \bar{c} &= \left|\{p|K_p \text{ is false}\}\right| \quad \text{(number of non corners)}
\end{aligned}
$$

- 먼저 검사할 x를 고를 시 발생하는 정보 이득:

$$IG = H(P) - H(P_d) - H(P_S) - H(P_b)$$

ID3 알고리즘을 이용한 결정 트리

- 정보 이득이 가장 큰 x를 선택하고, 이 결정을 하위 부분집합 P_d, P_s, P_b에 재귀적으로 적용

- 부분집합의 엔트로피가 0이 되면 종료
  - 엔트로피가 0
  - 해당 부분집합의 모든 K_p값은 동등
  - 모두 특징점이거나, 모두 특징점이 아님

- 이렇게 만들어진 결정 트리는 C(혹은 다른 언어)의 if-else문으로 바뀌어 컴파일됨

- 특징점을 판별하기 위해 검사하는 화소들의 **가장 좋은 순서**를 고르는 것!

비최대 억제

- 4번 약점인 인접한 지역에서 여러 개의 특징점이 검출되는 현상을 보완해야 함
- 특징점 검출 시 차등을 두지 않았기 때문에, 각 특징점에 대해 **점수 함수** V를 계산
- 인접한 지역에 더 높은 V값이 존재하는 특징점들을 제거

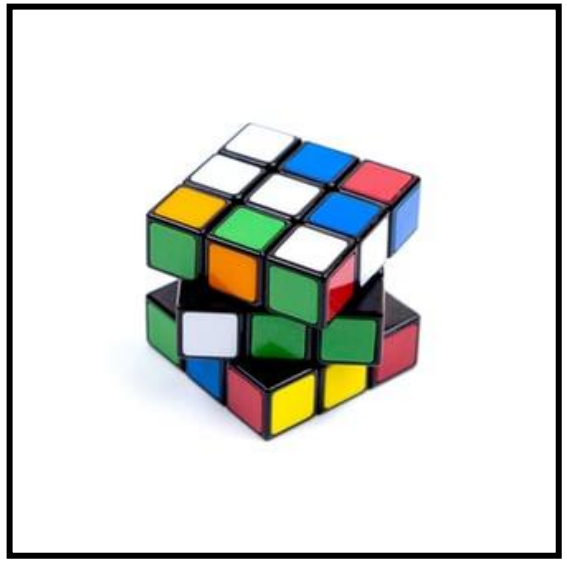
$$V = max\left(\sum_{x \in S_{bright}} |I_{p \to x} - I_p| - t\,,\ \sum_{x \in S_{dark}} |I_p - I_{p \to x}| - t\right)$$

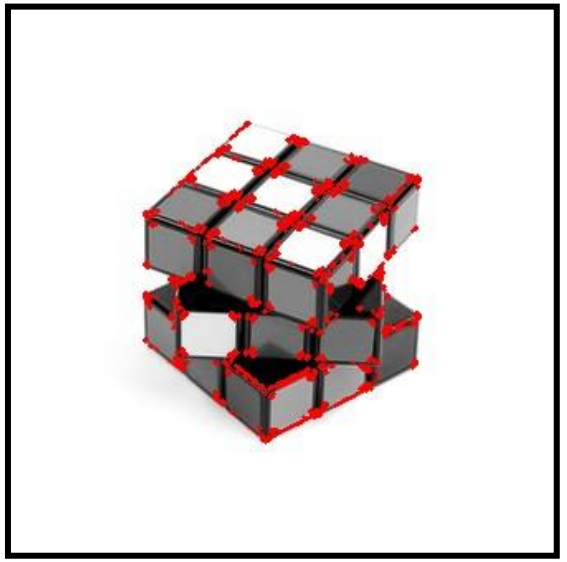
$$S_{p \to x} = \begin{cases} d, & I_{p \to x} \leq I_p - t \quad \text{(darker)} \\ s, & I_p - t < I_{p \to x} < I_p + t \quad \text{(similar)} \\ b, & I_p + t \leq I_{p \to x} \quad \text{(brighter)} \end{cases}$$

비최대 억제

- 함수 V가 의미하는 바는 곧 다음과 같음:
- 원 위의 화소 $x \in \{1..16\}$과 중심 화소 p 간의 **화소 강도 차이의 <u>절대값</u>**
- 원 위의 화소들이 중심 화소보다 밝을 때와 어두울 때 모두를 아우르겠다는 뜻
- 여러 특징점들 중, 중심 화소와 비교 화소 간 **화소 강도 차이가 큰 점들**만 남기겠다!

실행 결과





FAST in OpenCV

- https://github.com/opencv/opencv/blob/master/modules/features2d/src/fast.cpp

◆ FAST() [2/2]

```
void cv::FAST ( InputArray                  image,
                std::vector< KeyPoint > &   keypoints,
                int                         threshold,
                bool                        nonmaxSuppression,
                int                         type
              )
```

```
#include <opencv2/features2d.hpp>
```

Detects corners using the FAST algorithm.

Parameters

image	grayscale image where keypoints (corners) are detected.
keypoints	keypoints detected on the image.
threshold	threshold on difference between intensity of the central pixel and pixels of a circle around this pixel.
nonmaxSuppression	if true, non-maximum suppression is applied to detected corners (keypoints).
type	one of the three neighborhoods as defined in the paper: **FastFeatureDetector::TYPE_9_16**, **FastFeatureDetector::TYPE_7_12**, **FastFeatureDetector::TYPE_5_8**

Detects corners using the FAST algorithm by **[182]** .

정리

- 기존의 검출기들은 실시간 처리에 부적합, FAST의 등장
- 특징점 후보 화소와 그 주변 화소의 강도 차이를 이용해 검출하는 방식
- 처리 속도를 위해 가속화된 검사를 생각해보았으나 약점 존재
- 약점을 보완하기 위해 기계학습과 비최대 억제 알고리즘을 적용
- **ORB는 특징점 검출 시 FAST 검출 알고리즘을 이용하므로 알아둘 필요가 있음**
- 부가적으로, 기계학습이 완료된 검출기의 소스 코드와 프로그램은 링크에서 접근 가능

정리

- https://github.com/opencv/opencv/blob/master/modules/features2d/src/orb.cpp

◆ create()

static Ptr<ORB> cv::ORB::create (int nfeatures = 500 ,

```
// Detect FAST features, 20 is a good threshold
{
Ptr<FastFeatureDetector> fd = FastFeatureDetector::create(fastThreshold, true);
fd->detect(img, keypoints, mask);
}
```

)

static

Python:

cv.ORB_create([, nfeatures[, scaleFactor[, nlevels[, edgeThreshold[, firstLevel[, WTA_K[, scoreType[, patchSize[, fastThreshold]]]]]]]]) -> retval

Todo

- Oriented FAST
- BRIEF
- Rotated BRIEF
- SLAM