

[Intel] 엣지 AI SW 아카데미 / 객체지향 프로그래밍

영상처리 소프트웨어 구현

C++ 기반

이윤혁



목차

● [Intel] 엣지 AI SW 아카데미 / 객체지향 프로그래밍

● 1. 프로젝트 개요

● 2. 화소 점 처리

● 3. 화소 영역 처리

● 4. 기하학적 처리

● 5. 히스토그램 처리

● 6. 컬러 이미지 효과



[C++을 기반으로]

OpenCV 라이브러리 없이 직접!

C++와 Visual Studio 2022 MFC를 활용하여
GUI가 포함된 여러 영상처리 알고리즘을 구현

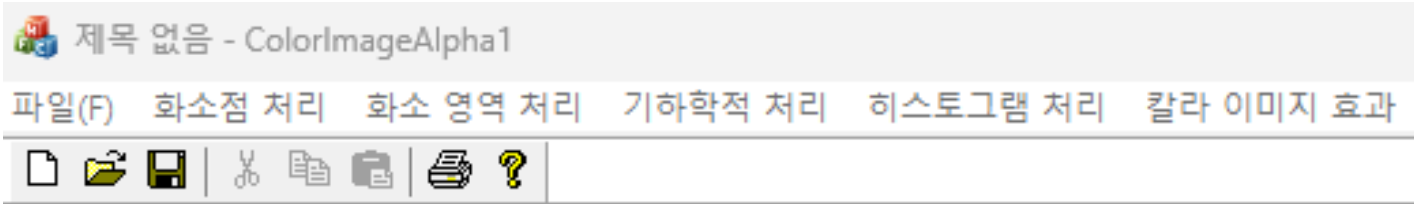
[개발 환경]

개발 OS, Tool, 언어

OS : Windows 11
Tool : Visual Studio 2022 MFC
Language : C++

프로젝트 개요 : 메뉴 화면 구성 및 추가 기능

메뉴 화면



기본 기능

새로 만들기(N)	Ctrl+N
열기(O)...	Ctrl+O
저장(S)	Ctrl+S
다른 이름으로 저장(A)...	
<hr/>	
인쇄(P)...	Ctrl+P
인쇄 미리 보기(V)	
인쇄 설정(R)...	
끝내기(X)	

화소 점 처리

동일 이미지	
그레이스케일	
밝기 조절	>
반전	
흑백	
감마 처리	
파라볼라 처리	>
연산자	>

화소 영역 처리

옴보싱
옴보싱(HSI)
블러
샤프닝
경계선 처리

기하학적 처리

확대	>
축소	
이동	
회전	>
반전	>

히스토그램 처리

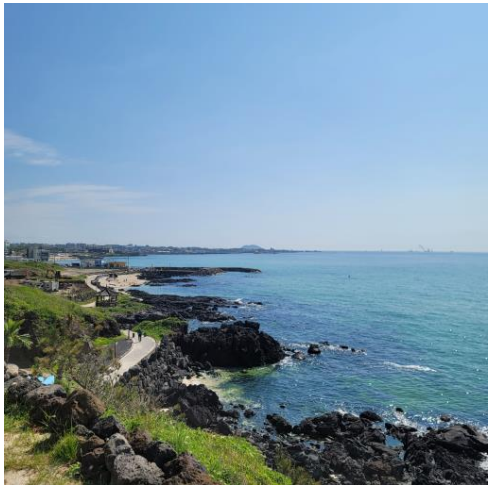
히스토그램 스트레칭
엔드-인
평활화

칼라 이미지 효과

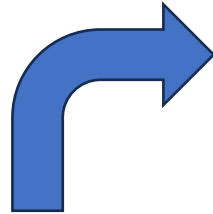
채도 변경
오렌지 추출

화소 점 처리

● 밝기 조절 (덧셈, 곱셈)



+100



X 2



1. 밝게 - 덧셈

```
if (m_inImageR[i][k] + value > 255)
    m_outImageR[i][k] = 255;
else
    m_outImageR[i][k] = m_inImageR[i][k] + value;
```

- ★ inImage(원본)에 입력 받은 정수 값을 더 함.
- ★ outImage 값이 255를 넘을 때를 고려하여 if 문 사용
- ★ 남은 G,B 값에 대해 반복해준다.

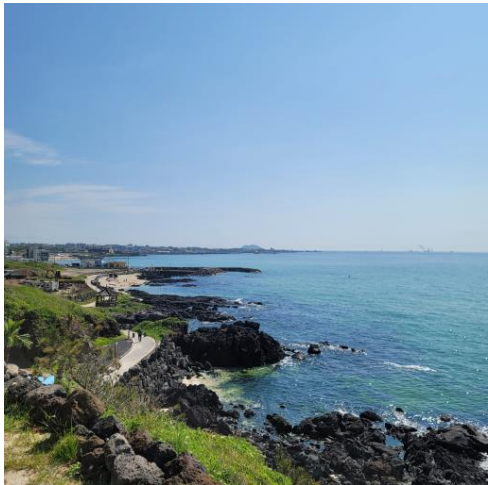
2. 밝게 - 곱셈

```
if (m_inImageR[i][k] * value > 255)
    m_outImageR[i][k] = 255;
else
    m_outImageR[i][k] = m_inImageR[i][k] * value;
```

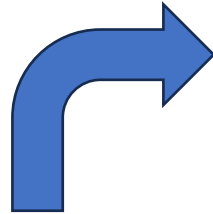
- ★ inImage(원본)에 입력 받은 정수 값을 곱함.
- ★ outImage 값이 255를 넘을 때를 고려하여 if 문 사용
- ★ 남은 G,B 값에 대해 반복해준다.
- ★ 덧셈 연산에 비해 보다 더 선명하게 밝기 조절이 된다.

화소 점 처리

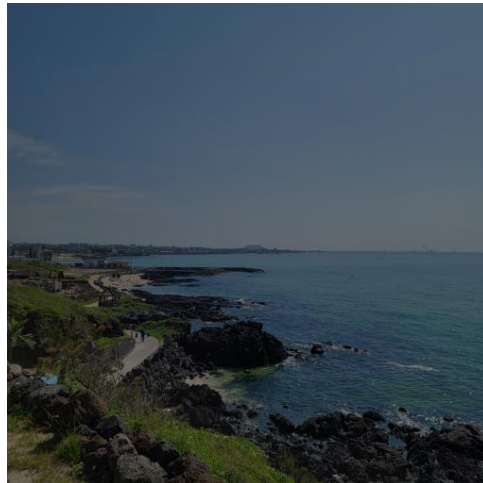
● 밝기 조절 (뺄셈, 나눗셈)



+100



X 2



3. 어둡게 - 뺄셈

```
if (m_inImageR[i][k] - val > 0)
    m_outImageR[i][k] = m_inImageR[i][k] - val;
else
    m_outImageR[i][k] = 0;
```

- ★ inImage(원본)에 입력 받은 정수 값을 뺌.
- ★ outImage 값이 0 미만이 될때를 고려하여 if 문 사용
- ★ 남은 G,B 값에 대해 반복해준다.

4. 어둡게 - 나눗셈

```
if (m_inImageR[i][k] / val > 0)
    m_outImageR[i][k] = m_inImageR[i][k] / val;
else
    m_outImageR[i][k] = 0;
```

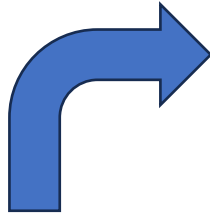
- ★ inImage(원본)에 입력 받은 정수 값을 뺌.
- ★ outImage 값이 0 미만이 될때를 고려하여 if 문 사용
- ★ 남은 G,B 값에 대해 반복해준다.
- ★ 뺄셈 연산에 비해 보다 더 선명하게 밝기 조절이 된다.

화소 점 처리

● 그레이 스케일, 흑백 처리



GrayScale



흑백 처리



5. GrayScale

```
avg = (m_inImageR[i][k] + m_inImageG[i][k] + m_inImageB[i][k]) / 3.0;  
m_outImageR[i][k] = m_outImageG[i][k] =  
m_outImageB[i][k] = (unsigned char)avg;
```

- ★ 원본의 RGB 값들의 평균을 구하여 outImage에 대입
- ★ 흑백 처리가 된 사진이 나온다.

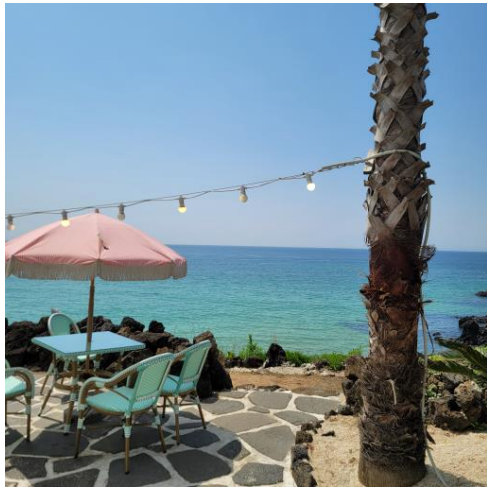
6. 흑백 처리

```
if (m_outImageR[i][k] < 127)  
    m_outImageR[i][k] = 0;  
else  
    m_outImageR[i][k] = 255;
```

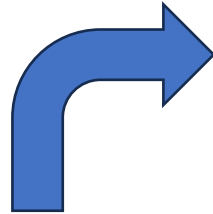
- ★ GrayScale 알고리즘을 이용하여 처리한 다음, 127을 기준으로 흑/백으로 나누어 처리한다.
- ★ 남은 G,B 값에 대해 반복해준다.

화소 점 처리

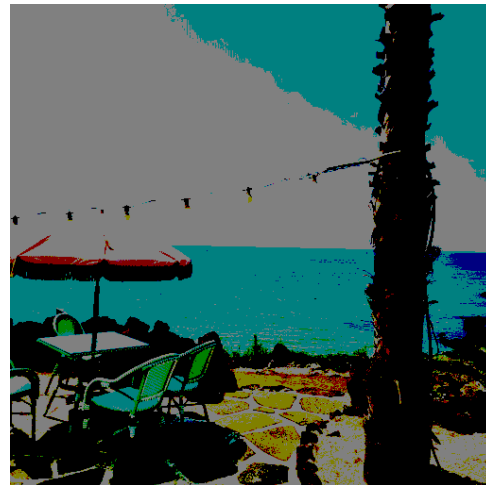
반전 처리, 연산자 (AND)



반전 처리



& 128



7. 반전 처리

```
for (int k = 0; k < m_inW; k++) {  
    m_outImageR[i][k] = 255 - m_inImageR[i][k];  
}
```

- ★ 각 픽셀의 값을 255에서 뺀다.
- ★ 반전 처리가 된 사진이 나온다.
- ★ 남은 G,B 값에 대해 반복해준다.

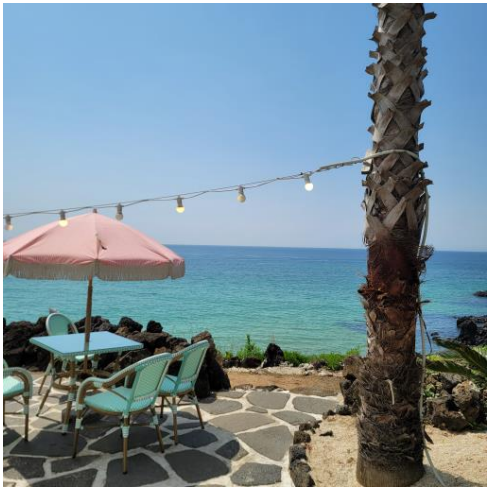
8. AND 연산

```
if ((m_inImageR[i][k] & (unsigned char)val) >= 255)  
    m_outImageR[i][k] = 255;  
else if ((m_inImageR[i][k] & (unsigned char)val) < 0)  
    m_outImageR[i][k] = 0;
```

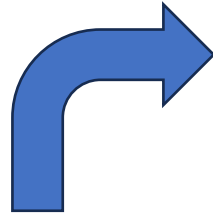
- ★ 0~255 범위 안에 있는 경우 각 픽셀에 & (AND) 연산자를 사용하여 연산
- ★ 남은 G,B 값에 대해 반복해준다.

화소 점 처리

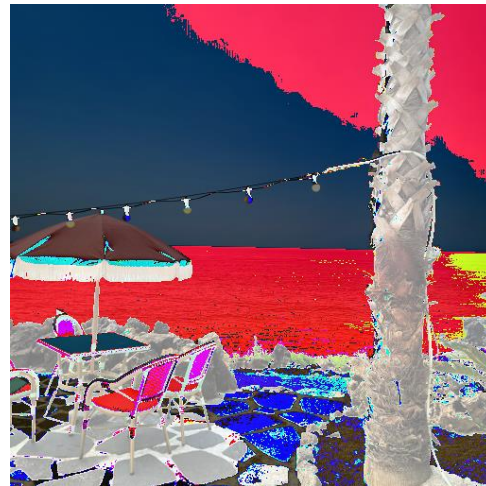
연산자 (OR, XOR)



| 128



^ 128



9. OR 연산

```
if ((m_inImageR[i][k] | (unsigned char)val) >= 255)
    m_outImageR[i][k] = 255;
else if ((m_inImageR[i][k] | (unsigned char)val) < 0)
    m_outImageR[i][k] = 0;
else
    m_outImageR[i][k] = (unsigned char)(m_inImageR[i][k] | (unsigned char)val);
```

- ★ 0~255 범위 안에 있는 경우 각 픽셀에 | (OR) 연산자를 사용하여 연산
- ★ 남은 G,B 값에 대해 반복해준다.

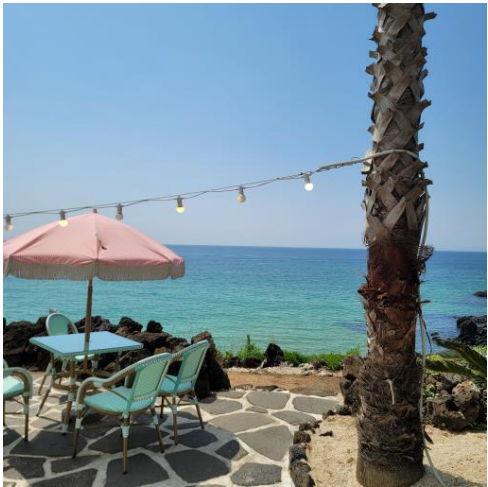
10. AND 연산

```
if ((m_inImageR[i][k] ^ (unsigned char)val) >= 255)
    m_outImageR[i][k] = 255;
else if ((m_inImageR[i][k] ^ (unsigned char)val) < 0)
    m_outImageR[i][k] = 0;
else
    m_outImageR[i][k] = (unsigned char)(m_inImageR[i][k] ^ (unsigned char)val);
```

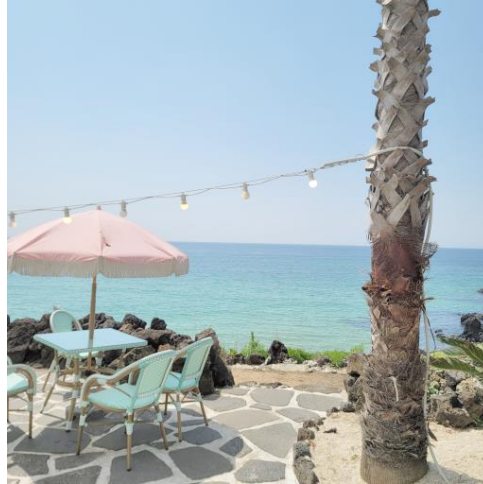
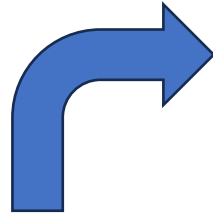
- ★ 0~255 범위 안에 있는 경우 각 픽셀에 ^ (XOR) 연산자를 사용하여 연산
- ★ 남은 G,B 값에 대해 반복해준다.

화소 점 처리

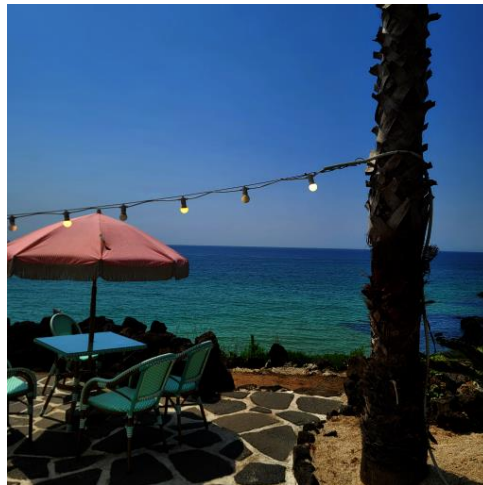
감마 처리



$\gamma = 0.5$

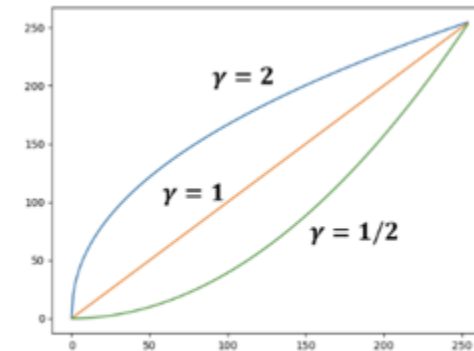


$\gamma = 2.5$



11. 감마 연산

```
float normalized_pixel = (float)inImage[i][k] / 255.0;  
float corrected_pixel = pow(normalized_pixel, gamma);
```

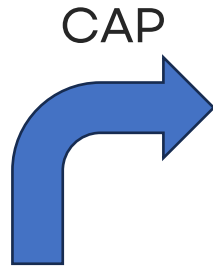


gamma 에 따른 pixel 값 변화

- ★ 픽셀 값을 0~1로 정규화 한 후 감마 함수 적용 후 보정된 값 계산
- ★ $\gamma > 1$ 이면 어두운 영역의 변화폭은 크고 밝은 영역의 변화폭은 작다
- ★ $\gamma < 1$ 이면 밝은 영역의 변화폭이 크고 어두운 영역의 변화폭은 작다
- ★ 남은 G,B 값에 대해 반복해준다.

화소 점 처리

● 파라볼라 처리 (CAP, CUP)



12. 파라볼라 CAP

```
m_outImageR[i][k] = 255 - 255 + pow((m_inImageR[i][k] / 127), 2);
```

- ★ Parabola CAP 함수를 사용하여 연산
- ★ 남은 G,B 값에 대해 반복해준다.

13. 파라볼라 CUP

```
m_outImageR[i][k] = 255 + pow((m_inImageR[i][k] / 127), 2);
```

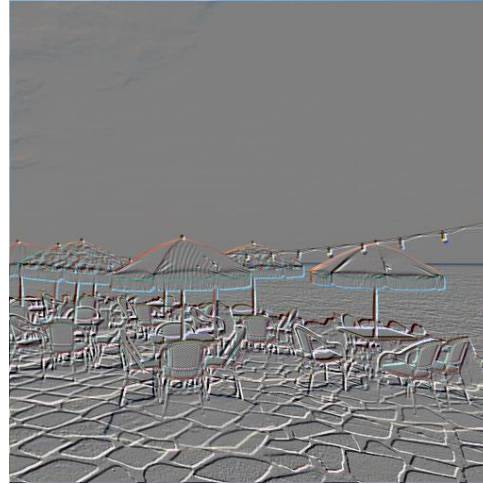
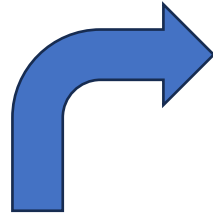
- ★ Parabola CUP 함수를 사용하여 연산
- ★ 남은 G,B 값에 대해 반복해준다.

화소 영역 처리

● 엠보싱 (1)



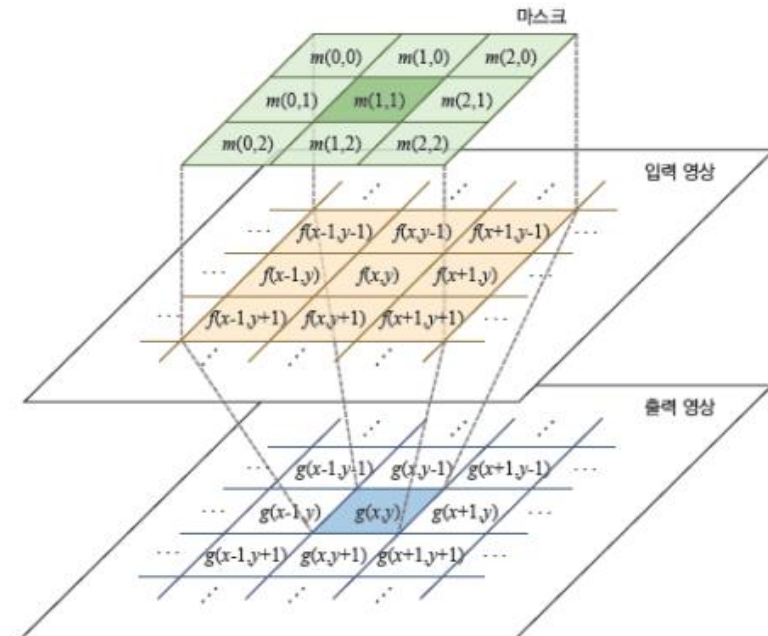
엠보싱



- ★ 해당 입력 화소뿐만 아니라 그 주위의 화소 값도 함께 고려함 - 영역, 공간 처리
- ★ 회선 기법이 추가됨.
- ★ 원시 화소와 이웃한 각 화소에 가중치를 곱한 합을 출력 화소로 생성
- ★ 엠보싱 - 입력 영상을 양각 형태로 보이게 하는 기술

1. 엠보싱

$$g(x, y) = m(0,0)f(x-1, y-1) + m(1,0)f(x, y-1) + m(2,0)f(x+1, y-1) \\ + m(0,1)f(x-1, y) + m(1,1)f(x, y) + m(2,1)f(x+1, y) \\ + m(0,2)f(x-1, y+1) + m(1,2)f(x, y+1) + m(2,2)f(x+1, y+1)$$

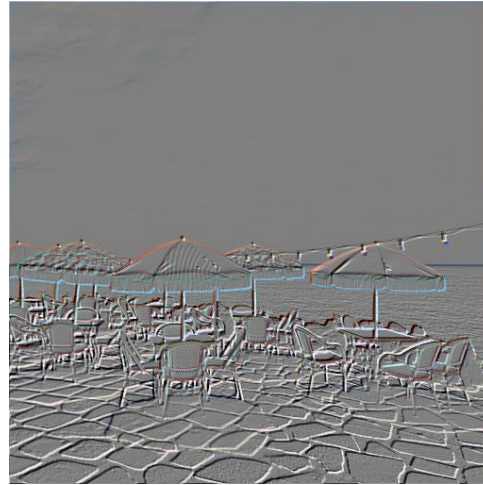
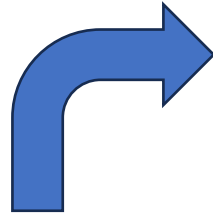


화소 영역 처리

엠보싱 (2)



엠보싱



```
double mask[MSIZE][MSIZE] =  
    { -1.0, 0.0, 0.0 },  
    {  0.0, 0.0, 0.0 },  
    {  0.0, 0.0, 1.0 } };
```

엠보싱 마스크

1. 엠보싱

```
// 마스크 (3x3) 와 한점을 중심으로한 3x3을 곱하기  
S = 0.0; // 마스크 9개와 입력값 9개를 각각 곱해서 합한 값.  
....  
    S += tmpInImage[i + m][k + n] * mask[m][n];  
}  
tmpOutImage[i][k] = S;
```

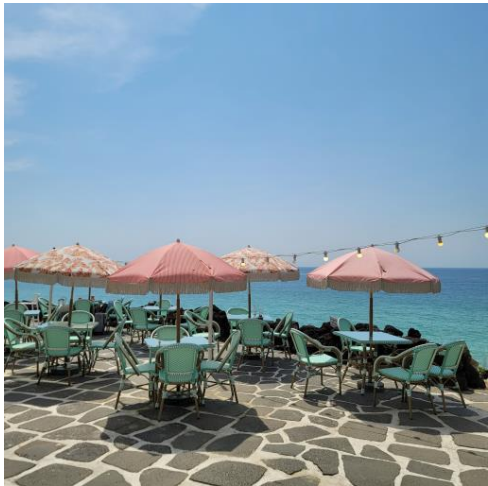
- ★ tmpInImage[i + 1][k + 1] = inImage[i][k];
→ 임시로 이미지의 가장자리를 확장.
- ★ RGB값 모두 수행해주어야함.

```
for (int m = 0; m < MSIZE; m++)  
    for (int n = 0; n < MSIZE; n++)  
        S += tmpInImageR[i + m][k + n] * mask[m][n];  
tmpOutImageR[i][k] = S;
```

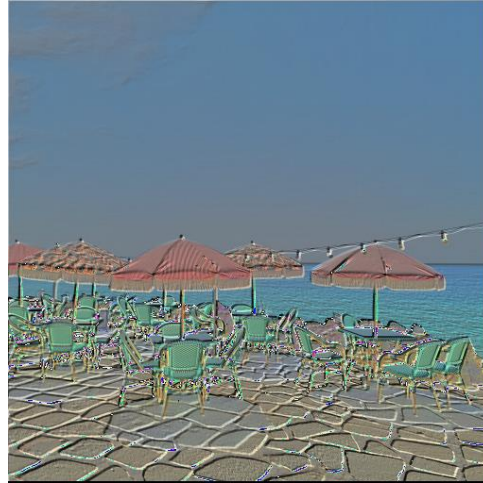
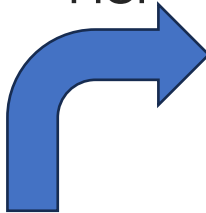
- ★ 회선 연산을 통하여 화소 영역 처리 알고리즘 수행.
- ★ 그 이후 후처리를 통하여 (ex. 각 픽셀에 +127)
이미지가 알맞게 나오도록 함.
- ★ RGB값 모두 수행해주어야함.

화소 영역 처리

● 엠보싱 (HSI)



엠보싱
HSI



- ★ 엠보싱에 HIS 알고리즘을 도입하여 원본의 RGB 값을 일부 보존
- ★ 일반적인 엠보싱 기법보다 색감이 더 남아있음

2. 엠보싱(HSI)

```
unsigned char R, G, B;  
R = tmpInImageR[i][k]; G = tmpInImageG[i][k]; B = tmpInImageB[i][k];  
hsi = RGB2HSI(R, G, B);
```

```
double H, S, I;  
H = hsi[0]; S = hsi[1]; I = hsi[2];  
tmpInImageH[i][k] = H; tmpInImageS[i][k] = S; tmpInImageI[i][k] = I;
```

★ RGB2HSI 함수를 통하여 RGB 값을 HSI 값으로 변환

```
unsigned char* rgb;  
double H, S, I;  
H = tmpInImageH[i][k]; S = tmpInImageS[i][k]; I = tmpInImageI[i][k];  
rgb = HSI2RGB(H, S, I);  
tmpOutImageR[i][k] = rgb[0]; tmpOutImageG[i][k] = rgb[1];  
tmpOutImageB[i][k] = rgb[2];
```

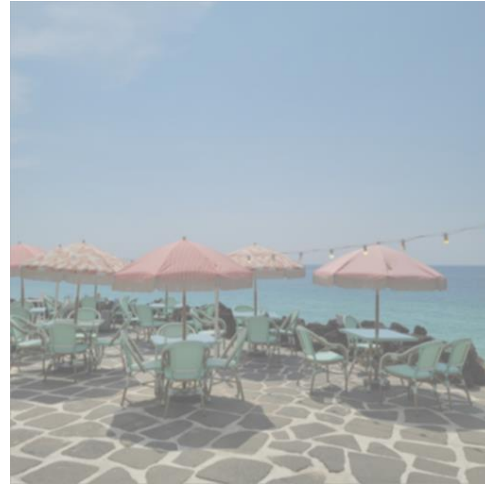
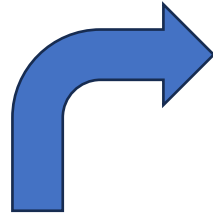
- ★ 이후 회선 연산을 통하여 화소 영역 처리 알고리즘 수행.
- ★ 그 이후 후처리를 통하여 (ex. 각 픽셀에 +127) 이미지가 알맞게 나오도록 함.
- ★ 그 후에 다시 HSI 값을 RGB 값으로 변환.
- ★ RGB값 모두 수행해주어야함.

화소 영역 처리

블러



블러



```
double mask[MSIZE][MSIZE] =  
{ {1.0 / 20, 1.0 / 20, 1.0 / 20},  
  {1.0 / 20, 2.0 / 20, 1.0 / 20},  
  {1.0 / 20, 1.0 / 20, 1.0 / 20} };
```

- ★ 영상의 세밀한 부분을 제거, 흐리게 하거나 부드럽게 하는 기술
- ★ 블러 마스크가 영상의 세밀한 부분인 고주파 부분을 제거함
- ★ 사용하는 회선 마스크는 저역통과필터 (Low Pass Filter)

3. 블러

```
// 마스크 (3x3) 와 한점을 중심으로한 3x3을 곱하기  
S = 0.0; // 마스크 9개와 입력값 9개를 각각 곱해서 합한 값.  
....  
    S += tmpInImage[i + m][k + n] * mask[m][n];  
}  
tmpOutImage[i][k] = S;
```

- ★ tmpInImage[i + 1][k + 1] = inImage[i][k];
→ 임시로 이미지의 가장자리를 확장.
- ★ RGB값 모두 수행해주어야함.

```
for (int m = 0; m < MSIZE; m++)  
    for (int n = 0; n < MSIZE; n++)  
        S += tmpInImageR[i + m][k + n] * mask[m][n];  
tmpOutImageR[i][k] = S;
```

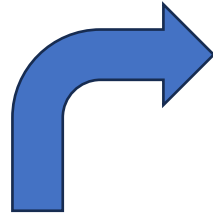
- ★ 회선 연산을 통하여 화소 영역 처리 알고리즘 수행.
- ★ 그 이후 후처리를 통하여 (ex. 각 픽셀에 +127) 이미지가 알맞게 나오도록 함.
- ★ RGB값 모두 수행해주어야함.

화소 영역 처리

샤프닝



샤프닝



```
double mask[MSIZE][MSIZE] =  
{ {0.0, -1.0, 0.0},  
  {-1.0, 5.0, -1.0},  
  {0.0, -1.0, 0.0} };
```

- ★ 블러링과는 반대로 상세한 부분을 더욱 강조
- ★ 샤프닝 마스크가 저주파 성분을 제거함
부분을 제거함
- ★ 사용하는 회선 마스크는 고역통과필터
(High Pass Filter)

4. 샤프닝

```
// 마스크(3x3) 와 한점을 중심으로한 3x3을 곱하기  
S = 0.0; // 마스크 9개와 입력값 9개를 각각 곱해서 합한 값.  
....  
    S += tmpInImage[i + m][k + n] * mask[m][n];  
}  
tmpOutImage[i][k] = S;
```

- ★ tmpInImage[i + 1][k + 1] = inImage[i][k];
→ 임시로 이미지의 가장자리를 확장.
- ★ RGB값 모두 수행해주어야함.

```
for (int m = 0; m < MSIZE; m++)  
    for (int n = 0; n < MSIZE; n++)  
        S += tmpInImageR[i + m][k + n] * mask[m][n];  
tmpOutImageR[i][k] = S;
```

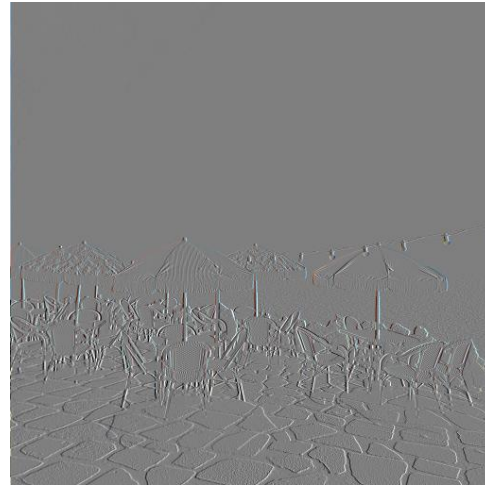
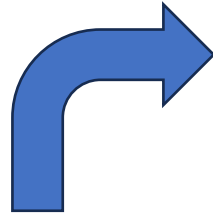
- ★ 회선 연산을 통하여 화소 영역 처리 알고리즘 수행.
- ★ 그 이후 후처리를 통하여 (ex. 각 픽셀에 +127)
이미지가 알맞게 나오도록 함.
- ★ RGB값 모두 수행해주어야함.

화소 영역 처리

● 경계선 검출



경계선
검출



```
const int MSIZE = 3;
double mask[MSIZE][MSIZE] =
{ {0.0, 0.0, 0.0},
  {-1.0, 1.0, 0.0},
  {0.0, 0.0, 0.0} };
```

- ★ 영상의 경계선을 찾아내는 기술
- ★ 경계선은 영상의 밝기가 변하는 지점에 있으므로 입력한 영상의 정보가 많이 필요.

5. 경계선 검출

```
// 마스크(3x3) 와 한점을 중심으로한 3x3을 곱하기
S = 0.0; // 마스크 9개와 입력값 9개를 각각 곱해서 합한 값.
....
    S += tmpInImage[i + m][k + n] * mask[m][n];
}
tmpOutImage[i][k] = S;
```

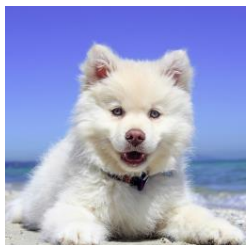
- ★ tmpInImage[i + 1][k + 1] = inImage[i][k];
→ 임시로 이미지의 가장자리를 확장.
- ★ RGB값 모두 수행해주어야함.

```
for (int m = 0; m < MSIZE; m++)
    for (int n = 0; n < MSIZE; n++)
        S += tmpInImageR[i + m][k + n] * mask[m][n];
tmpOutImageR[i][k] = S;
```

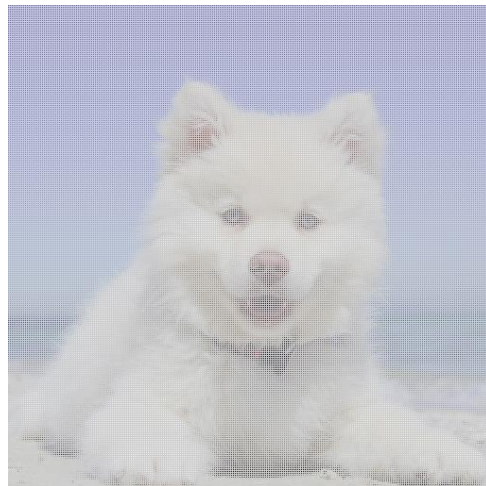
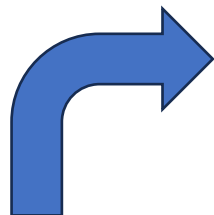
- ★ 회선 연산을 통하여 화소 영역 처리 알고리즘 수행.
- ★ 그 이후 후처리를 통하여 (ex. 각 픽셀에 +127) 이미지가 알맞게 나오도록 함.
- ★ RGB값 모두 수행해주어야함.

기하학적 처리

확대 - 포워딩 (1)



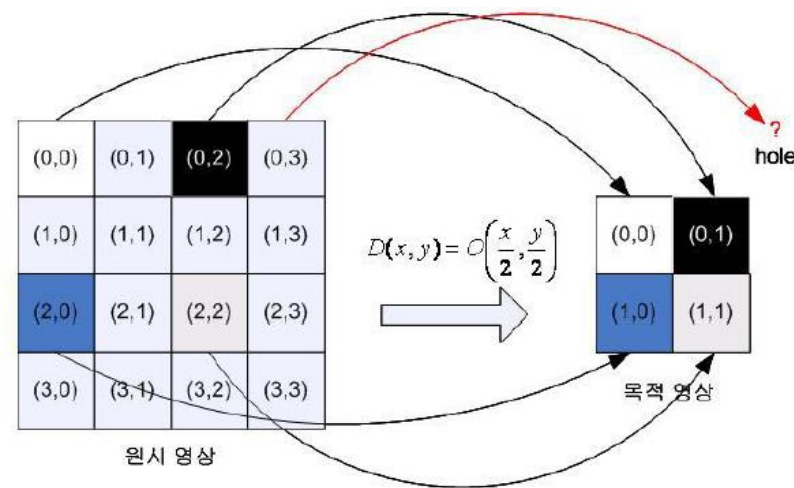
확대 X 2
(포워딩)



포워딩 확대 기법을 적용하여
홀이 생긴 모습

- ★ 영상을 구성하는 화소의 공간적 위치를 재배치
- ★ 전방향 사상 - 입력영상을 출력영상으로 화소 위치 변환

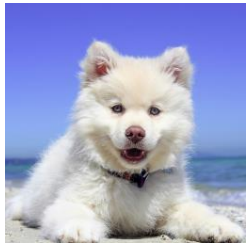
1. 확대 (포워딩)



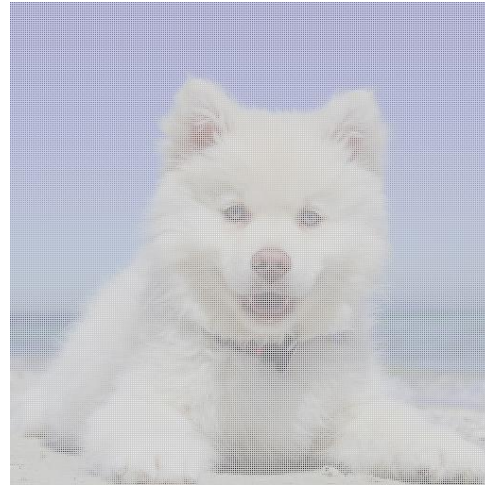
- ★ 전방향(포워딩) 사상에서는 원 영상의 좌표가 홀수면 목적 영상의 좌표 값에 소수점이 들어가 해당 좌표가 존재하지 않으므로 홀 문제가 발생

기하학적 처리

확대 - 포워딩 (2)



확대 X 2
(포워딩)

포워딩 확대 기법을 적용하여
홀이 생긴 모습

```
m_outH = (int)(m_inH * scale);  
m_outW = (int)(m_inW * scale);
```

출력 이미지의 크기 결정

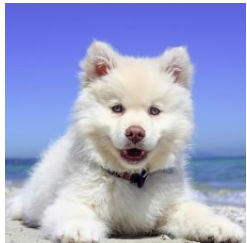
1. 확대 (포워딩)

```
m_outImageR[(int)(i * scale)][(int)(k * scale)]  
= m_inImageR[i][k];
```

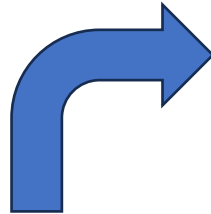
- ★ 포워딩 기법을 사용하여 픽셀 값 계산
- ★ RGB값 모두 수행해주어야함.

기하학적 처리

확대 - 백워딩 (1)



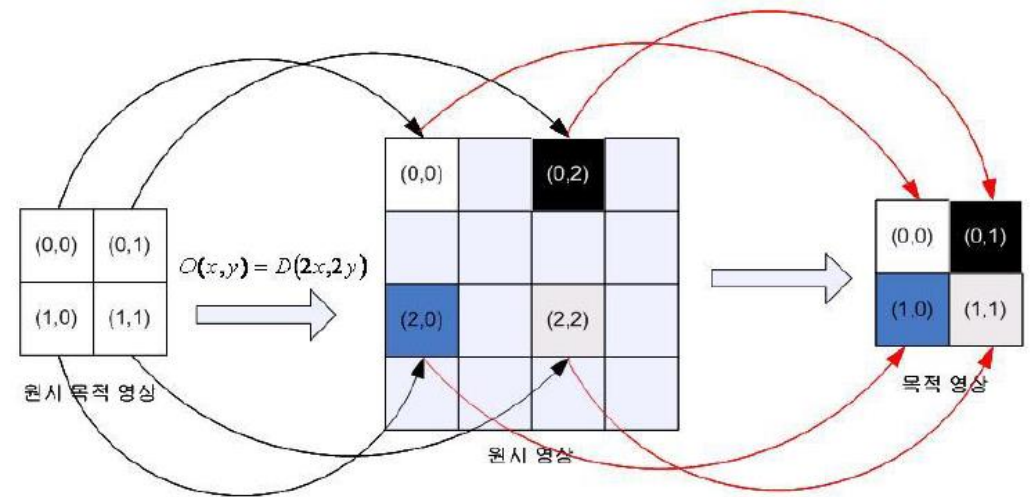
확대 X 2
(백워딩)



백워딩 확대 기법을 적용하여
홀이 생기지 않은 모습

- ★ 영상을 구성하는 화소의 공간적 위치를 재배치
- ★ 역방향 사상 - 출력영상을 입력영상으로 화소 위치 변환

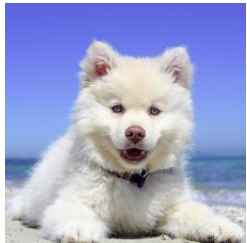
2. 확대 (백워딩)



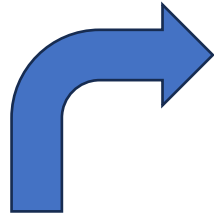
- ★ 역함수로 원 영상의 좌표 값을 계산, 그 좌표에 해당하는 좌표를 찾아 목적 영상에 할당.
- ★ 포워딩 사상에서 발생했던 홀 문제 발생 하지 않음

기하학적 처리

확대 - 백워딩 (2)



확대 X 2
(백워딩)



백워딩 확대 기법을 적용하여
홀이 생기지 않은 모습

```
m_outH = (int)(m_inH * scale);  
m_outW = (int)(m_inW * scale);
```

출력 이미지의 크기 결정

2. 확대 (백워딩)

```
m_outImageR[i][k] =  
m_inImageR[(int)(i / scale)][(int)(k / scale)];
```

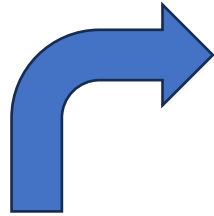
- ★ 백워딩 기법을 사용하여 픽셀 값 계산
- ★ RGB값 모두 수행해주어야함.

기하학적 처리

확대 - 양선형 보간법 (1)



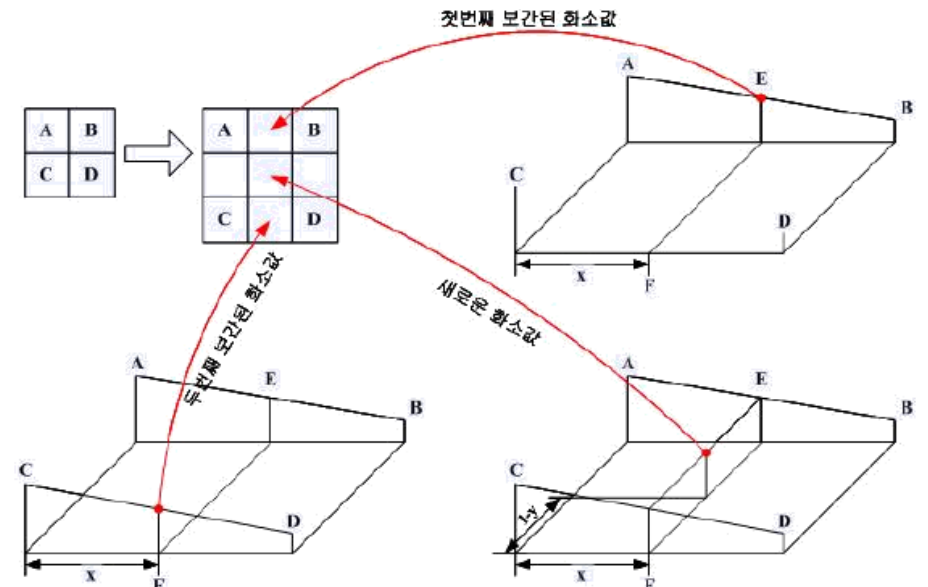
확대 X 2
(양선형 보간법)



양선형 보간법 기법을 적용하여
홀이 생기지 않은 모습

- ★ 영상을 구성하는 화소의 공간적 위치를 재배치
- ★ 양선형 보간법 - 원 영상의 화소 값 두개를 이용하여 원하는 좌표에서 새로운 좌표를 계산하는 방법

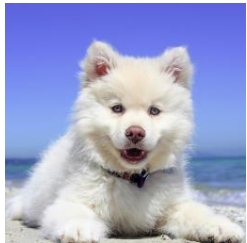
3. 확대 (양선형 보간법)



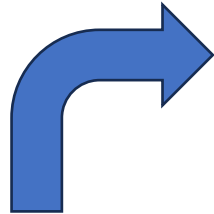
- ★ 선형 보간을 바탕으로 화소당 세 번 수행.
- ★ 새롭게 생성된 화소는 가장 가까운 화소 네 개에 가중치를 곱한 값을 합해서 얻음.
- ★ 각 가중치는 각 화소에서의 거리에 정비례하도록 선형적으로 선택

기하학적 처리

확대 - 양선형 보간법 (2)



확대 X 2
(양선형 보간법)



양선형 보간법 기법을 적용하여
홀이 생기지 않은 모습

3. 확대 (양선형 보간법)

```
int baseRow = (int)(round(i * rowRatio));  
int baseCol = (int)(round(k * colRatio));
```

★ 현재 픽셀의 위치를 기준으로 입력 이미지에서 가장 가까운 네 개의 픽셀을 찾음

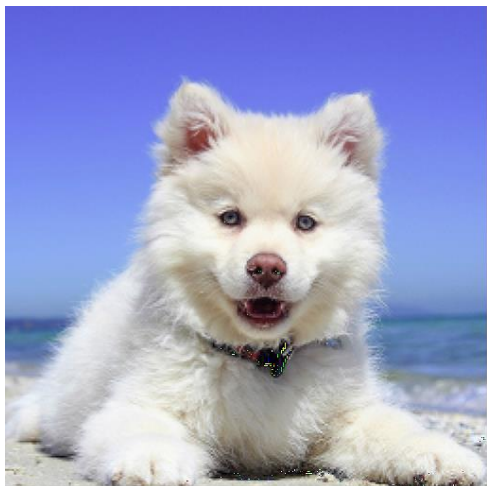
```
double dx = (i * rowRatio) - baseRow;  
double dy = (k * colRatio) - baseCol;
```

★ 현재 픽셀의 위치와 가장 가까운 네 개의 픽셀 사이의 거리를 계산

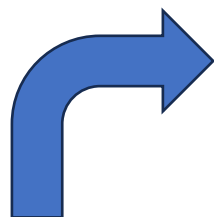
```
double interpolatedValue = (1 - dx) * (1 - dy) * m_inImageR[baseRow][baseCol] +  
dx * (1 - dy) * m_inImageR[baseRow + 1][baseCol] +  
(1 - dx) * dy * m_inImageR[baseRow][baseCol + 1] +  
dx * dy * m_inImageR[baseRow + 1][baseCol + 1];
```

★ 양선형 보간법을 사용하여 현재 픽셀 값 계산.

★ RGB값 모두 수행해주어야함.



축소 X 2



```
m_outH = (int)(m_inH / scale);  
m_outW = (int)(m_inW / scale);
```

★ 출력 이미지의 크기를 결정

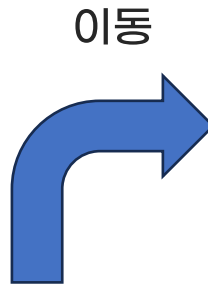
4. 축소

```
m_outImageR[(int)(i / scale)][(int)(k / scale)] = m_inImageR[i][k];  
m_outImageG[(int)(i / scale)][(int)(k / scale)] = m_inImageG[i][k];  
m_outImageB[(int)(i / scale)][(int)(k / scale)] = m_inImageB[i][k];
```

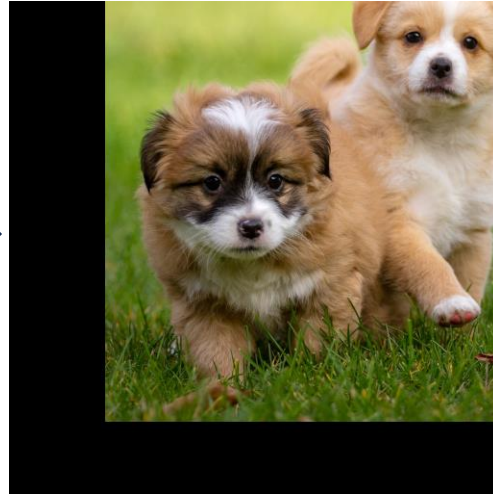
- ★ 입력 받은 값으로 축소 알고리즘 실행
- ★ RGB값 모두 수행해주어야함.

기하학적 처리

이동



이동



```
m_outH = m_inH;  
m_outW = m_inW;
```

★ 출력 이미지의 크기를 결정

5. 이동

```
if (i + a >= 0 && i + a < m_inH && k - b >= 0 && k - b < m_inW)  
    m_outImageR[i][k] = m_inImageR[i + a][k - b];  
else  
    m_outImageR[i][k] = 0;
```

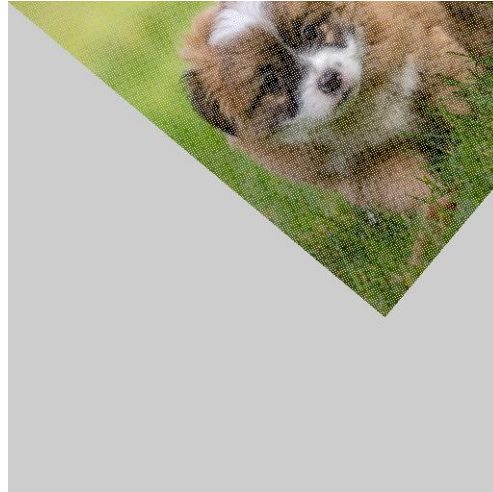
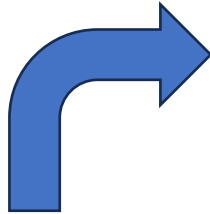
- ★ 입력 받은 값으로 이동 알고리즘 실행
- ★ RGB값 모두 수행해주어야함.

기하학적 처리

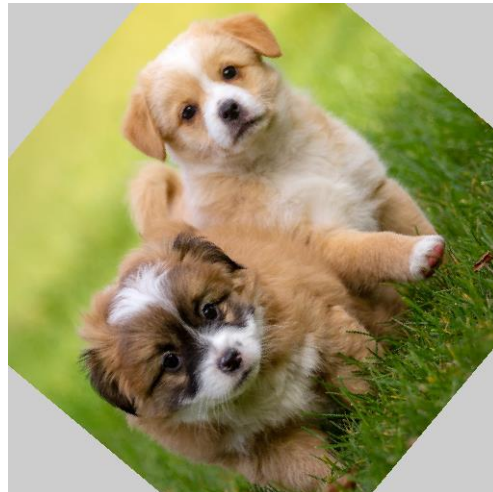
회전



회전



CUP



6. 회전

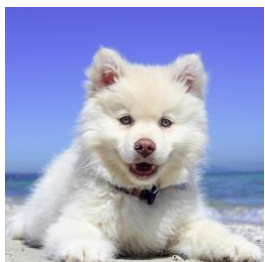
```
if ((0 <= xd && xd < m_outH) && (0 <= yd && yd < m_outW))  
    m_outImageR[xd][yd] = m_inImageR[xs][ys];
```

- ★ 입력 받은 각도 만큼 회전.
- ★ if ((0 <= xd && xd < outH) && (0 <= yd && yd < outW))
- ★ if문을 사용하여 벗어난 이미지에 대한 예외처리.
- ★ 하지만 중심이 맞지 않고, 화질 저하
- ★ RGB값 모두 수행해주어야함.

7. 회전 (중심 + 백워딩)

```
int cx = m_inH / 2;  
int cy = m_inW / 2;  
if ((0 <= xs && xs < m_outH) && (0 <= ys && ys < m_outW))  
    m_outImageR[xd][yd] = m_inImageR[xs][ys];
```

- ★ 입력 받은 각도 만큼 회전.
- ★ 중심 맞추는 코드 추가.
- ★ 백워딩을 적용하여 화질 보존
- ★ RGB값 모두 수행해주어야함.



8. 회전 (확대 + 백워딩)

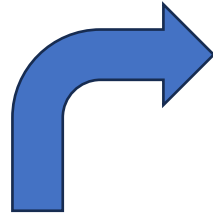
```
int xd = i / scale;
int yd = k / scale;
int xs = (int)(cos(radian) * (xd - cx) - sin(radian) * (yd - cy) + cx);
int ys = (int)(sin(radian) * (xd - cx) + cos(radian) * (yd - cy) + cy);

// 회전된 픽셀 위치의 유효성 검사
if (0 <= xs && xs < m_inH && 0 <= ys && ys < m_inW) {
    m_outImageR[i][k] = m_inImageR[xs][ys];
}
else {
    m_outImageR[i][k] = 255; // 흰색으로 설정
```

- ★ 입력 받은 각도 만큼 회전.
- ★ 백워딩 기법으로 확대를 하여 화질 저하 없이 확대
- ★ RGB값 모두 수행해주어야함.



상하



좌우



9. 상하 대칭

```
m_outImageR[i][k] = m_inImageR[m_inH - 1 - i][k];  
m_outImageG[i][k] = m_inImageG[m_inH - 1 - i][k];  
m_outImageB[i][k] = m_inImageB[m_inH - 1 - i][k];
```

- ★ 상하 대칭 반전 수행
- ★ 남은 G,B 값에 대해 반복해준다.

10. 좌우 대칭

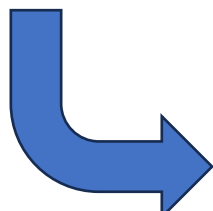
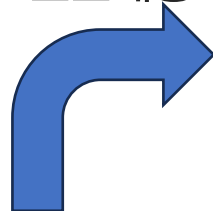
```
m_outImageR[i][k] = m_inImageR[i][m_inW - 1 - k];  
m_outImageG[i][k] = m_inImageG[i][m_inW - 1 - k];  
m_outImageB[i][k] = m_inImageB[i][m_inW - 1 - k];
```

- ★ 좌우 대칭 반전 수행
- ★ 남은 G,B 값에 대해 반복해준다.

히스토그램 처리

스트레칭, 엔드-인

히스토그램
스트레칭



엔드-인



1. 히스토그램 스트레칭

```
if (m_inImageR[i][k] < lowR)
    lowR = m_inImageR[i][k];
if (m_inImageR[i][k] > highR)
    highR = m_inImageR[i][k];
```

```
oldPixelR = m_inImageR[i][k];
```

- ★ 히스토그램의 최저 명도 값과 최고 명도 값을 구하는 함수를 삽입 후 기본 명암대비 스트레칭 수행 공식을 코드에 삽입
- ★ 남은 G,B 값에 반복 수행

2. 엔드-인

```
if (m_inImageR[i][k] < lowR)
    lowR = m_inImageR[i][k];
if (m_inImageR[i][k] > highR)
    highR = m_inImageR[i][k];
```

```
oldPixelR = m_inImageR[i][k];
```

```
if (newPixelR > 255)
```

```
    newPixelR = 255;
```

```
if (newPixelR < 0)
```

```
    newPixelR = 0;
```

```
m_outImageR[i][k] = newPixelR;
```

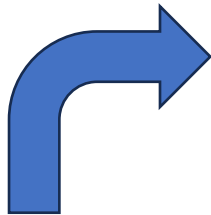
- ★ 좌우 대칭 반전 수행, 일정한 양의 화소를 흰색이나 검정색으로 지정하여 히스토그램의 분포를 좀더 균일하게 만들
- 엔드-인 탐색 수행 공식을 사용
- ★ 남은 G,B 값에 대해 반복해준다.

히스토그램 처리

히스토그램 평활화



회전+
확대



3. 히스토그램 평활화

```
int histo[256] = { 0. };  
  
....  
  
    histo[inImage[i][k]]++;
```

1단계 : 히스토그램 빈도수 세기

```
int sumHisto[256] = { 0, };  
sumHisto[0] = histo[0];  
...  
  
    sumHisto[i] = sumHisto[i - 1] + histo[i];
```

2단계 : 누적 히스토그램 생성

```
double normalHisto[256] = { 1.0, };  
.... {  
    normalHisto[i] = sumHisto[i] * (1.0 / (inH * inW)) * 255.0;
```

3단계 : 정규화 된 히스토그램 생성

```
outImage[i][k] = (unsigned char)normalHisto[inImage[i][k]];
```

4단계 : inImage를 정규화 된 값으로 치환

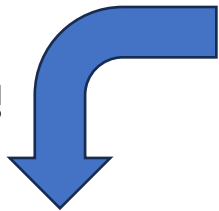
5단계 : G,B 값에 대해 반복 수행

칼라 이미지 효과

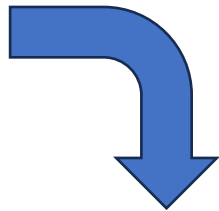
채도 변경, 색 추출



채도 변경



색 추출



1. 채도 변경

```
double* hsi = RGB2HSI(R, G, B);
H = hsi[0]; S = hsi[1]; I = hsi[2];
/// 채도(S) 흐리게
S = S - 0.3;
if (S < 0)
    S = 0.0;
// HSI --> RGB
unsigned char* rgb = HSI2RGB(H, S, I);
R = rgb[0]; G = rgb[1]; B = rgb[2];
m_outImageR[i][k] = R;
```

- ★ RGB 값을 HSI 값으로 변환시켜 채도 변경하는 알고리즘 처리
- ★ 남은 G,B 값에 반복 수행

2. 색 추출

```
double* hsi = RGB2HSI(R, G, B);
H = hsi[0]; S = hsi[1]; I = hsi[2];

/// 오렌지 추출 (H : 8~20)
if (8 <= H && H <= 20) {
    m_outImageR[i][k] = m_inImageR[i][k];
} else {
    double avg = (m_inImageR[i][k] + m_inImageG[i][k] + m_inImageB[i][k]) / 3.0;
    m_outImageR[i][k] = m_outImageG[i][k] = m_outImageB[i][k] = (unsigned char)avg;
```

- ★ RGB 값을 HSI 값으로 변환시켜 일정 범위의 색을 추출하는 알고리즘 처리
- ★ 남은 G,B 값에 대해 반복해준다.



다양한 로 본격적으로 이렇게 프로그램을
만들어 본 것은 처음이었다.
그 동안 포토샵 같은 프로그램들이 어떻게
작동하는지 원리를 전혀 몰랐는데
이 프로젝트를 진행하면서
많이 알아가게 되는 것 같다.

아직은 남아 있는 수 많은 영상 처리
알고리즘들을 다 담아내진 못하였다.
내가 코딩 실력이 부족함에 따라 시간이
남들보다 많이 걸렸기 때문이라 생각한다.
출발이 늦은 만큼 더더욱 노력 해야겠다.

향후에 이 프로그램을 더 발전 시킨다면,
우선 시간 관계상 코딩하지 못했던 추가 기
능들을 추가한다음, 궁극적으로 풀컬러 영상
에서도 완벽하게 동작하게 하는 프로그램을
만들어 보고 싶다.