

마우스, 키보드 자동화를 이용한 게임 매크로

2021204045 이성민

해당 문서는 함께 제공된 게임 매크로 Python 파일의 내용과 동작 방식을 설명하는 문서이다. 먼저, 해당 매크로는 학습 목적으로 개발되었으며, 개발 외의 성능적인 사용은 하지 않았음을 밝힌다. 본 파일은 다음과 같은 구성으로 이루어져 있다.

게임 매크로를 실행하는 python 파일 (Dragon(Nox540X960)가상키.py),

게임 매크로 실행 도중 게임 내부에서 실행되는 매크로 방지 프로그램을 파훼하는 python 파일 (Prevent_macro.py),

게임이 실행되어 있지 않을 경우에도 실행할 수 있는 디버깅용 매크로 방지 프로그램 파훼 python 파일 (Prevent_macro_debug.py),

혹시나 매크로 방지 프로그램 파훼가 실패할 경우 사용자에게 알림을 주는 mp3파일 (EnderDragonDie.mp3),

매크로 방지 프로그램이 실행되었을 때, 이를 촬영하여 저장하는 파일(file),

매크로 방지 파훼 프로그램 디버깅용 여분 촬영 파일(Folder),

실제 게임에 적용시켜 실행하는 영상(실행동영상.mp4) (용량 문제로 화질이 매우 낮음).

그리고 상단의 프로그램을 설명하는 본 문서 1부로 구성되어 있다.


1. 게임의 기본 설명





해당 화면은 게임의 메인 화면이다. 게임 매크로가 실행될 때 이 창이 무조건 실행되어 있어야 하며(창의 이름은 NoxPlayer이다.) 해당 화면을 화면에 띄워놓은 상태로 매크로를 실행해야 한다. 매크로의 1 사이클이 마무리될 때 오는 장 소이며, 매크로가 이를 인식하여 매크로를 시작/재시작하게 된다.


스마트폰 프로그램을 pc에서 사용할 수 있게 해주는 Nox에는 가상키라는 기능이 있는데, 사전에 설정해둔 키보드의 키를 누르면 해당 화면을 자동으로 터치하는 방식이다. 또한 영웅들의 위에 떠 있는 파란색 바가 바로 스킬을 사용할 수 있다는 표시이며, 본 매크로 프로그램은 화면의 색을 인식하여 자동으로 키보드를 눌러 매크로를 실행한다.


각 영웅의 역할은 다음과 같다.


 **도로시(hero1, 1), 엘리스(hero2, 2), 리사(hero3, 3):** 해골 마법사, 궁수, 전사를 소환하는 마녀 소환사. 마녀 3명이 모두 스킬을 사용할 수 있는 상태일 때, 높은 순위로 다른 마녀들과 함께 스킬을 사용한다.


 **크로노(hero4, Q):** 스킬을 사용할 시 **게임 속도가 빨라진다**. 본인의 스킬 사용이 가능할 때, 높은 순위로 스킬을 사용한다.


 **퓨어 위자드(hero5, W):** 스킬을 사용할 시 **영웅의 쿨타임이 빨리 찬다**. 다른 대부분의 영웅의 스킬 사용이 불가능하면서 본인의 스킬 사용이 가능할 때 스킬을 사용한다.

 **포세이돈(hero6, E):** 스킬을 사용할 시 공격하는 딜러. 본인의 스킬 사용이 가능할 때, 낮은 순위로 스킬을 사용한다.

 **클레릭(hero7, A), 다크 네크로맨서(hero8, S), 다크 스켈레톤(hero9, D):** 스킬을 사용할 시 각각 아군에게 버프를 주는 서포터, 적군에게 디버프를 주는 서포터, 적을 공격하는 딜러이다. 서로의 쿨타임이 비슷하여 3명이 모두 스킬을 사용할 수 있는 상태일 때, 높은 순위로 함께 스킬을 사용한다.

 **스미스 2세(hero10, Z):** 스킬을 사용할 시 **성체의 HP를 회복한다**. 성체의 HP가 얼마 없으면서 본인의 스킬 사용이 가능할 때, 매우 높은 순위로 스킬을 사용한다.

 **다크 엘프(hero11, X), 블루 디펜더(hero12, C):** 각각 성체의 HP를 지속적으로 회복하는 서포터, 성체의 최대 HP를 올려주는 서포터이다. 스킬이 존재하지 않아, 매크로 python 파일에는 변수 할당만 되어 있고 사용은 되지 않는다.

 **마을 아처1(villagearcher1, 4), 마을 아처2(villagearcher2, 5):** 스킬을 사용할 시 공격 속도가 증가하는 딜러. 둘 모두 스킬을 사용할 수 있는 상태일 때, 매우 낮은 순위로 함께 스킬을 사용한다.

2. 동작 방식



게임의 메인 화면이다. 이 곳에서 V키를 눌러 왼쪽 아래의 화면으로 진입한다.

왼쪽의 화면에서 다시 V키를 누르면 '레전더리 드래곤' 창으로 넘어가게 되고, 오른쪽의 화면에서 Space키를 누르면 전투 화면으로 진입하게 된다.

전투 화면으로 진입할 때 보통은 왼쪽의 화면으로 진입되어 **드래곤과의 전투**가 시작되지만, 약 1시간마다 오른쪽의 **매크로 방지 프로그램**으로 진입하게 된다.

먼저 START 버튼을 누르면, 가운데에 있는 다이아몬드가 랜덤한 통나무에 스며들며, 이 통나무들이 서로 섞인 뒤, 다이아몬드가 스며든 통나무를 찾아내 마우스로 클릭하는 방식이다.



해당 프로그램 파훼에 성공하면 왼쪽 위 화면으로 진입하여 다시 드래곤과의 전투가 시작된다. 매크로 프로그램에선 해당 통나무를 일명 'macrotree'라고 명명한다.



드래곤과의 전투가 승리로 마무리되면, 드래곤이 있던 자리에 보물상자가 생성되면서 위에 붉은 글씨로 시간초가 세어진다.

해당 시간초가 마무리되면, 최종적으로 드래곤을 잡은 보상으로 한 장비가 드롭된다. 여기서 N 버튼을 누르면 해당 장비를 분해할 수 있고, M 버튼을 누르면 해당 장비를 내 인벤토리에 받아들 수 있다. 이 단계를 마무리하면 다시 게임의 메인 화면으로 돌아가고, 이를 반복한다.

3. 코드 설명

대부분은 코드 내부에 주석으로 설명을 달아놓았으니, 해당 문서에선 전체적인 코드 동작 방식을 설명한다.



3-1. Dragon(Nox540X960)가상키.py

```
# 창 크기와 위치
window = pyautogui.getActiveWindow()

window_title = 'NoxPlayer' # NoxPlayer 앱이 켜져있어야 이 창의 크기를 확인하고 변수 좌표를 할당함

# 모든 창 목록을 가져옵니다.
windows = gw.getWindowsWithTitle(window_title)

# 창이 발견되면 위치를 출력합니다.
if windows: # NoxPlayer 앱이 켜져있다면
```

먼저 해당 프로그램을 실행하기 위해선 'NoxPlayer'라는 프로그램이 켜져 있어야 한다. 또한 이 름에서 보다시피 540 X 960 해상도를 기준으로 맞췄기에 해당 해상도를 맞추면 문제 없이 해당 매크로를 실행할 수 있다.

```
start_dragonhunt() # 드래곤 사냥을 시작함.
```

각종 변수(좌표)를 할당한 뒤에, 드래곤 사냥을 시작하는 함수를 실행시킨다.

```
def start_dragonhunt():
    start() # 메인 화면에서 드래곤 전투 화면으로 입장하기 위해 실행
    delay(0.5, 0.6) # 0.5 ~ 0.6초 랜덤 딜레이
```

함수 시작부터 start() 함수를 실행한다.

```
def start(): # 메인 화면에서 드래곤 전투 화면으로 입장하기 위해 실행되는 함수
    pyautogui.press('v') # 메인 화면의 드래곤 석상
    delay(0.2, 0.3)

    pyautogui.press('v') # 레전더리 드래곤
    delay(0.05, 0.1)

    pyautogui.press('space') # BATTLE
    delay(0.15, 0.2)
```

각각 메인 화면에서 드래곤 석상으로, 드래곤 석상에서 레전더리 드래곤으로, 레전더리 드래곤에서 BATTLE을 누르는 함수이다.



```
color_at_prevent_macro = check_color_at_position(prevent_macro) # prevent_macro 좌표의 색을 확인
macro_prevent = is_similar_to(color_at_prevent_macro, brown, 2) # 확인한 색이 brown과 2만큼 비슷한지 확인(거의 똑같아야 함)
if macro_prevent: # 색이 비슷하다고 확인된다면 macro_prevent는 True의 값을 가짐. 아니면 False.
    subprocess.run(['python', 'C:/Visual_Studio_Code/GrowCastle/Prevent_macro.py']) # Prevent_macro.py(매크로 방지 프로그램 파헤) 실행
    delay(0.1, 0.2)
    color_at_prevent_macro = check_color_at_position(prevent_macro)
    macro_prevent = is_similar_to(color_at_prevent_macro, brown, 2)
if macro_prevent: # Prevent_macro.py가 실행되었음에도 여전히 True라면(매크로 방지 프로그램이 닫히지 않았다면)
    print("매크로 파헤 실패")
    pygame.init()
    pygame.mixer.init()
    pygame.mixer.music.load('C:/Visual_Studio_Code/GrowCastle/EnderDragonDie.mp3') # 알람이 울림
    pygame.mixer.music.play()
    while pygame.mixer.music.get_busy():
        pygame.time.Clock().tick(10) # 알람이 끝날 때까지 기다림
    sys.exit(0)
# 알람이 끝난 뒤에 프로그램 종료
else: # 매크로 방지 프로그램 파헤에 성공하였다면
    monitor_heroes() # 본격적으로 드래곤과의 전투를 시작함
    delay(0.2, 0.25)
    start_dragonhunt() # 모든 단계가 끝난 뒤에 처음부터 다시 시작함
else: # 매크로 방지 프로그램이 실행되지 않았다면
    monitor_heroes() # 본격적으로 드래곤과의 전투를 시작함
    delay(0.2, 0.25)
    start_dragonhunt() # 모든 단계가 끝난 뒤에 처음부터 다시 시작함
```

start() 함수가 실행되고 나면 드래곤과의 전투 화면으로 넘어가거나 매크로 방지 프로그램으로 넘어가게 된다. 만약 매크로 방지 프로그램으로 넘어가게 될 경우, 내용이 매우 길어지기에 새로운 .py로 넘어가게 된다. 매크로 방지 프로그램 파헤가 성공하거나 매크로 방지 프로그램이 실행되지 않았다면 monitor_heroes() 함수가 시작된다. 또한 만에 하나 매크로 방지 프로그램 파헤가 실패한다면 이 상태에서 한 번 더 실패할 시 게임에서 매크로 판정을 받아 계정 정지에 이를 수 있기에 큰 효과음으로 사용자에게 알람을 주고 프로그램을 종료한다. 이 다음 매크로 방지 프로그램은 직접 풀어야 한다. prevent_macro2.py 는 내용이 많아 해당 py 파일을 모두 설명한 뒤에 설명하도록 하겠다.

def monitor_heroes(): # 본격적으로 드래곤과의 전투를 시작하는 함수

드래곤과의 전투 페이지에 진입하게 되면, 영웅들의 스킬 사용 가능 여부와 기타 정보에 따라 다른 알고리즘으로 영웅들의 스킬을 사용한다. 그저 영웅의 스킬 사용이 가능하다고 해서 마구 사용해버리면 매크로 판정을 받을 수 있는 데다가 드래곤을 잡을 때 효율 또한 좋지 못하기에 적당한 알고리즘을 구현하여 영웅들의 스킬을 자동으로 사용한다.

```
# hp 적을 때 스미스 2세 스킬 1순위로 사용
if hero10skill and not hpfull: # 스미스 2세의 스킬이 사용 가능하면서 성체 HP가 절반 이하라면
    press('z') # 스미스 2세의 스킬 사용(성체 HP 60% 회복)
    delay(0.001, 0.002)
```

예를 들어 스미스 2세는 스킬을 사용할 시 성체 HP의 60%를 회복한다. 그러나 성체의 HP가 많은 초반에 이 스킬을 사용하면 스킬 낭비로 이어질 수 있다. 그러니 성체의 HP가 절반 이하일 때 해당 스킬을 사용하면 해결될 것이다.

red = (232, 77, 77) # 성체 HP의 색

```
hpdanger = (((window.width)*0.5787)+window.left,
            ((window.height)*0.08)+window.top) # 성채 HP의 중간 어느 부분의 좌표이다. 해당 좌표의 색을 판별해 스미스 2세의 스킬
사용 여부를 결정한다.
```

```
color_at_hpdanger =check_color_at_position(hpdanger) # 성채 HP의 중간 부분의 좌표(hpdanger)의 색 확인.
hpfull =is_similar_to(color_at_hpdanger, red, 20) # 성채 HP의 중간 부분 좌표의 색이 red와 20만큼 비슷한지(성채 HP가 절반 이상
인지) 확인. 절반 이상이라면 True, 절반 이하라면 False.
```



가운데 상단 성채 HP 중간에 보이는 하얀색 점이 hpdanger의 좌표이다. 저 hpdanger의 색을 검사해서 red와 20만큼 비슷하다면 hpfull이 True, 아니라면 False가 된다. 이 값을 이용해 스미스 2세의 스킬 사용 여부를 결정한다.

```
color_at_dragonhp0 =check_color_at_position(dragonhp0)
dragonddie =is_similar_to(color_at_dragonhp0, cleared, 1) # dragon의 생사 여부 상태 갱신

if defeat: # 스킬 사용 사이사이에 꾸준히 패배 혹은 승리 확인.
    start_dragonhunt() # 패배했다면 다시 드래곤 사냥 시작
if dragonddie : # 드래곤과의 전투에서 승리했다면
    receive_compensation() # 보상을 받는 함수
```

함수 중간중간에 이런 코드를 꾸준히 볼 수 있는데, 해당 코드는 만약 영웅 스킬 사용 로직이 발동하고 있는 중간이라도 드래곤의 사망 혹은 플레이어의 패배를 감지할 수 있게 영웅 스킬 사용 로직 중간중간에 빼곡히 넣어 두었다. 다른 스킬 사용 코드는 주석에 설명을 넣어두었으므로 넘어가겠다.

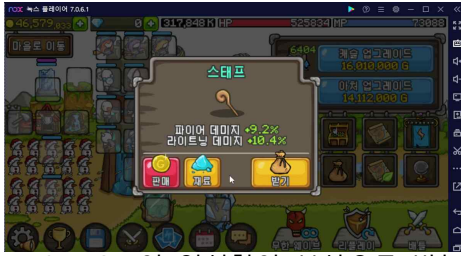


dragonhp0 좌표의 색이 cleared와 1만큼 비슷(거의 똑같음)하다는 것은, 드래곤을 해치워 보상을 기다리고 있다는 증거이다. 그러므로 dragonddie가 True라면 receive_compensation() # 보상을 받는 함수로 이동하게 된다.

```
def receive_compensation(): # 드래곤과의 전투 승리 시 보상을 받는 함수
    time.sleep(2.5)
    while True: # 다른 함수로 진입하지 않는 한(보상을 획득함), 계속 반복
        color_at_resultcall =check_color_at_position(resultcall)
        if is_similar_to(color_at_resultcall, brown, 1): # 드래곤을 해치우고 보상 창이 뜬다면
            delay(0.4, 0.5)
            color_at_rating =check_color_at_position(rating) # 보상의 등급 색을 확인
            if is_similar_to(color_at_rating, A_color, 20): # 보상의 등급 색이 A_color와 20만큼 비슷하다면(A등급이라면)
                pyautogui.press('n') # 보상을 가루로 분해함
                delay(0.1, 0.2)
                pyautogui.press('n') # 보상을 가루로 분해함(2차 확인)
            else : # 보상의 등급 색이 A_color와 20만큼 비슷하지 않다면(S등급이나 L등급이라면)
                pyautogui.press('m') # 보상을 인벤토리 내에 받음

            delay(0.2, 0.3)
            start_dragonhunt() # 모든 단계를 끝마치면 다시 드래곤 사냥 시작
```

보상을 받는 함수이다. 보상 창의 brown색을 인식하여 재료로 분해할지, 혹은 인벤토리로 받을지 결정하는 함수이다. 사진의 재료 위의 하늘색 가루가 보이는가? 저 곳의 색을



color_of_rating이 인식하여 보상으로 받는 장비의 등급을 판별한다. 사진의 등급은 **A급**이다.



만약 보상으로 받는 장비가 **S급**이나 **L급**이라면 가루는 왼쪽과 같은 색을 띄게 된다(해당 등급 장비의 등장 확률이 낮아 인벤토리의 사진으로 대체한다...). 그래서 만약 해당 가루의 색이 A_color와 20만큼 비슷하다면(A급의 장비라면) N 키를 눌러 장비를 분해하고, 비슷하지 않다면(S급이나 L급이라면) M키를 눌러 인벤토리로 장비를 받아온다. S_color, L_color 색 배열은 당장은 사용하지 않으나 나중에 더욱 상위 드래곤을 상대하는 매크로를 만들 때를 위해 미리 해당 색 배열을 마련해놓았다. 보상을 분해하거나 받는 데 성공하였다면, 다시 `start_dragonhunt()`의 처음으로 돌아가 상기한 단계들을 반복한다.

3-2. Prevent_macro.py



BATTLE 버튼을 눌렀을 때, 약 1시간마다 드래곤 대신 매크로 방지 프로그램이 실행된다. 상기한 방식으로, START 버튼을 누르면 가운데의 다이아몬드가 랜덤한 통나무로 스며들고, 통나무들이 서로 섞인 뒤 다이아몬드가 스며든 통나무를 선택(클릭)하면 해당 창이 닫히고 다시 드래곤과의 전투가 시작되는 방식이다.

```
color_at_prevent_macro = check_color_at_position(prevent_macro) # prevent_macro 좌표의 색을 확인
macro_prevent = is_similar_to(color_at_prevent_macro, brown, 2) # 확인한 색이 brown과 2만큼 비슷한지 확인(거의 똑같아야 함)
if macro_prevent: # 색이 비슷하다고 확인된다면 macro_prevent는 True의 값을 가짐. 아니면 False.
    subprocess.run(['python', 'C:/Visual_Studio_Code/GrowCastle/Prevent_macro.py']) # Prevent_macro.py(매크로 방지 프로그램 좌측) 실행
```

상기한 방식으로 해당 창의 배경인 brown을 인식하여 `Prevent_macro.py`를 실행한다.

```
# 저장할 이미지 수
num_images = 80

screenshot = pyautogui.screenshot()
# 지정된 영역 잘라내기
cropped_image = screenshot.crop((macrobar))

# 이미지 저장
cropped_image.save(f"C:/Visual_Studio_Code/GrowCastle/file/1.png")

click(macrobarstart)

# 이미지 생성 및 저장
for i in range(num_images):
    # 전체 화면 스크린샷 찍기
    screenshot = pyautogui.screenshot()
```

```
# 지정된 영역 잘라내기
cropped_image = screenshot.crop((macrobar))

# 이미지 저장
cropped_image.save(f"C:/Visual_Studio_Code/GrowCastle/file/{i + 1}.png")
```

여기서부터 `Prevent_macro.py`의 코드이다. macrobar는 매크로 방지 프로그램의 검색 창 범위이다. 또한 macrobarstart는 사진에 보이는 START 버튼의 범위이다. 해당 버튼을 클릭하고 80장을 연속적으로 찍어서 file 폴더에 저장한다. 이제부터 할 작업들은 모두 이렇게 찍힌 사진을 분석하는 과정이다. 왼쪽의 사진은 이러한 과정으로 거쳐 찍힌 사진들이다. 또한 본 문서와 함께 포함되어 있는 file 폴더와 Folder 폴더도 실제 이러한 과정을 통해 찍힌 사진들이며, 원한다면 함께 동봉된 `Prevent_macro debug.py`(디버깅용 결과 출력만 되고 클릭은 하지 않음)를 이용해 분석 결과를 직접 확인할 수 있다.

```
for i in range(8, 25):
    # 이미지 경로 설정
    image_path = f"C:/Visual_Studio_Code/GrowCastle/file/{i}.png"

    # 이미지 읽기
    image = cv2.imread(image_path)
```

file에 저장된 사진 중 8부터 시작하여 24까지 분석한다. 그러나 중간에 특정 조건을 만족하면 루프가 종료된다. 이 특정 조건은 밑에서 설명하겠다.

```
# 목표 색상과 허용 오차 설정
target_color = np.array([(145, 194, 214)]) # GBR 기준
diamond_color = np.array([(247, 195, 63)]) # GBR 기준
tolerance = 15 # target_color의 범위
tolerance2 = 100 # diamond_color의 범위

# 색상의 상한과 하한 설정
lower_bound = target_color - tolerance
upper_bound = target_color + tolerance

lower_diamond_bound = diamond_color - tolerance2
upper_diamond_bound = diamond_color + tolerance2

# 이미지에서 목표 색상과 가까운 부분 마스크 생성
mask = cv2.inRange(image, lower_bound, upper_bound)
diamond_mask = cv2.inRange(image, lower_diamond_bound, upper_diamond_bound)

# 마스크된 영역을 초록색으로 변경
image[mask != 0] = [0, 255, 0]
image[diamond_mask != 0] = [0, 0, 0]

# BGR에서 HSV로 변환
hsv_image = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
```



```

# 마스크에서 초록색 뭉치 찾기
contours, _=cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
diamondcontours, _=cv2.findContours(diamond_mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
# 최소 면적 필터 (큰 뭉치만)
min_area =130

# 검정색 뭉치들의 중심 좌표 계산 및 출력
count_black_blobs =0
for contour in diamondcontours:
    area =cv2.contourArea(contour)
    if area >=min_area:
        count_black_blobs +=1
        M =cv2.moments(contour)
        if M["m00"] !=0:
            cX =int(M["m10"] /M["m00"])
            cY =int(M["m01"] /M["m00"])
            cv2.circle(image, (cX, cY), 5, (0, 0, 255), -1)
        else:
            pass

```

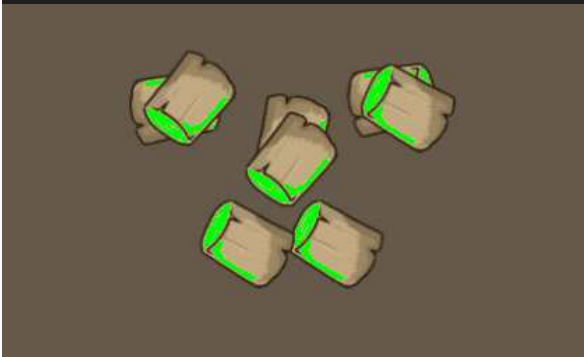


여기서 통나무의 윗면의 색을 인식하여 초록색 마스크를 입혀 표시하였고, 다이아몬드의 좌표 또한 cX, cY로 표시하였다. 그런데 여기서 통나무의 옆면도 같이 인식이 되어 초록색 뭉치의 개수가 16개가 되는(윗면 8개 + 옆면 8개) 참사가 발생하였다. 이를 해결하기 위해 min_area = 130를 지정하여 면적이 이보다 높은 초록색 뭉치만 분석에 활용하였다. 이렇게 하면 통나무의 옆면은 min_area를 넘지 않아 분석에 활용되지 않는다.

```

# 초록색 뭉치가 7개 미만이라면 루프를 끝냄
count = 0
for contour in contours:
    area =cv2.contourArea(contour)
    if area >=min_area:
        count +=1
if count < 7:
    break

```



여기서 통나무가 섞이는 과정에서 서로가 서로를 가리는 경우가 발생한다. 이 과정에서 통나무 8개 중, 6개 이하만 인식이 될 정도로 사진의 상태가 좋지 않다면, 분석 결과도 이상하게 나올 수 있기 때문에 루프를 종료한다. 왼쪽의 사진은 이 경우에 부합하는 사진으로, 해당 사진은 분석에 어려움이 있을 수 있어 분석에 활용하지 않고 루프를 끝낸다.

```

count =0
# 초록색 뭉치의 좌표 표시
for contour in contours:

```

```

area =cv2.contourArea(contour)
if area >=min_area:
    count +=1
    # 뭉치의 좌표들 중에서 가장 왼쪽과 가장 오른쪽 좌표 찾기
    leftmost =tuple(map(int, contour[contour[:, :, 0].argmin()][0])) # 가장 왼쪽 x 좌표
    rightmost =tuple(map(int, contour[contour[:, :, 0].argmax()][0])) # 가장 오른쪽 x 좌표
    # 가장 위쪽과 아래쪽 좌표 찾기
    topmost =tuple(map(int, contour[contour[:, :, 1].argmin()][0])) # 가장 위쪽 y 좌표
    bottommost =tuple(map(int, contour[contour[:, :, 1].argmax()][0])) # 가장 아래쪽 y 좌표

    if calculate_distance(topmost, bottommost) >calculate_distance(leftmost, rightmost):
        leftmost, rightmost =topmost, bottommost

    # 가장 왼쪽 좌표에 빨간색 점 표시
    cv2.circle(image, leftmost, 5, (count *30, count *30, 255), -1)
    # 가장 오른쪽 좌표에 빨간색 점 표시
    cv2.circle(image, rightmost, 5, (count *30, count *30, 255), -1)
    # 왼쪽과 오른쪽을 잇는 빨간색 선 그리기
    cv2.line(image, leftmost, rightmost, (count *30, count *30, 255), 2)

    # 선의 중점 계산
    midpoint = ((leftmost[0] +rightmost[0]) //2, (leftmost[1] +rightmost[1]) //2)

    # 중점에 빨간색 점 표시
    cv2.circle(image, midpoint, 5, (count *30, count *30, 255), -1)

```



인식된 초록색 뭉치의 가장 왼쪽과 가장 오른쪽, 그리고 중점을 출력한다(만약 가장 왼쪽과 가장 오른쪽을 이은 선보다 가장 위쪽, 가장 아래쪽을 이은 선이 더 길다면 해당 선으로 대체한다.). 그러나 이 중점은 초록색 뭉치의 중점이 아닌 통나무의 중점이 아니다. 아래의 코드를 통해 통나무의 중점을 구할 수 있다.

```

# 선의 각도 계산 (atan2 사용)
dx =rightmost[0] -leftmost[0]
dy =rightmost[1] -leftmost[1]
angle =math.atan2(dy, dx) # 각도 (라디안 단위)

# 수직 방향으로 선을 그리기 위한 각도 계산 (현재 각도에서 90도 회전)
perpendicular_angle =angle +math.pi /2 # 90도 (라디안) 추가

# 23 픽셀 떨어진 좌표 계산 (중점에서 수직으로)
offset_x =int(23 *math.cos(perpendicular_angle))
offset_y =int(23 *math.sin(perpendicular_angle))

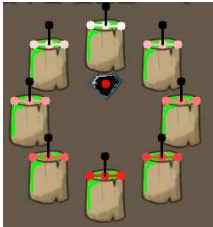
# 새로운 좌표 계산
new_point = (midpoint[0] +offset_x, midpoint[1] +offset_y)
new_point2 = (midpoint[0] -offset_x, midpoint[1] -offset_y)

rgb_color1 =image[new_point[1], new_point[0]]
rgb_color2 =image[new_point2[1], new_point2[0]]

distance1 =np.linalg.norm(midpoint_color -rgb_color1)
distance2 =np.linalg.norm(midpoint_color -rgb_color2)
cv2.circle(image, new_point2, 5, (0, 0, 0), -1)
cv2.line(image, midpoint, new_point2, (0, 0, 0), 2)

```

해당 사진은 통나무의 중점을 구한 사진이다. 그러나 후에 중점의 색을 구하는 부분이 있어



중점을 표시하지 않고 반대편을 표시했다. 검은색 선 부분은 중점의 반대편이며, 붉은 직선의 반대로 같은 길이, 같은 각도로 선을 그으면 중점을 알 수 있다. 아래의 코드는 중점으로서의 선을 긋는 코드이다.

```
cv2.circle(image, new_point, 5, (0, 0, 0), -1)
cv2.line(image, midpoint, new_point, (0, 0, 0), 2)
```

그러나 이 과정에서 문제가 발생한다. `new_point`와 `new_point2`중에 무엇을 중점으로 삼을지에 대한 코드가 없어, 조금만 통나무의 각도가 틀어져도 중점이 중구난방해진다. 아까 검정색 부분이 중점의 반대편이라고 하였는데, 검정색 부분이 중점인 통나무들은 통나무의 중점으로 이상한 좌표가 부여되어 분석에 어려움이 있을 수 있다. 이를 위해 `new_point`와 `new_point2`중에 무엇을 중점으로 삼을지에 대한 코드가 필요하다.



```
if np.array_equal(rgb_color1, background_color) or
np.array_equal(rgb_color2, background_color): # point 2개 중, 무엇이
background_color과 더 가까운지 체크
    if np.array_equal(rgb_color1, background_color):
        new_point, new_point2 = new_point2, new_point
        rgb_color1, rgb_color2 = rgb_color2, rgb_color1
    elif not np.array_equal(rgb_color1, background_color) and not np.array_equal(rgb_color2, background_color):
        # 더 가까운 색상 출력
        if distance1 > distance2:
            # print("바꾸기 실행")
            new_point, new_point2 = new_point2, new_point
            rgb_color1, rgb_color2 = rgb_color2, rgb_color1
            # print(f"중심 색: {rgb_color1}, 바깥쪽 색: {rgb_color2}")

if np.all(rgb_color1 == rgb_color2): # 서로 색이 같다면
    # 최소 거리 초기화
    min_distance_new_point = float('inf')
    min_distance_new_point2 = float('inf')

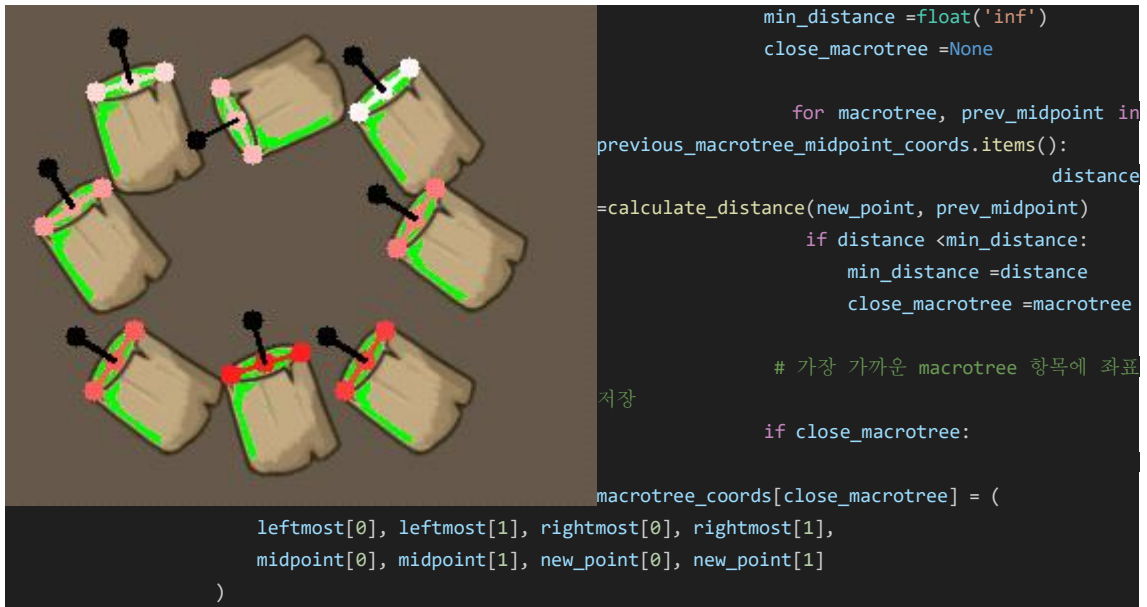
    # 이전 사진의 중점들과 거리 비교
    for name, coords in previous_macrotree_midpoint_coors.items():
        distance_to_new_point = calculate_distance(coords, new_point)
        distance_to_new_point2 = calculate_distance(coords, new_point2)

        min_distance_new_point = min(min_distance_new_point, distance_to_new_point)
        min_distance_new_point2 = min(min_distance_new_point2, distance_to_new_point2)

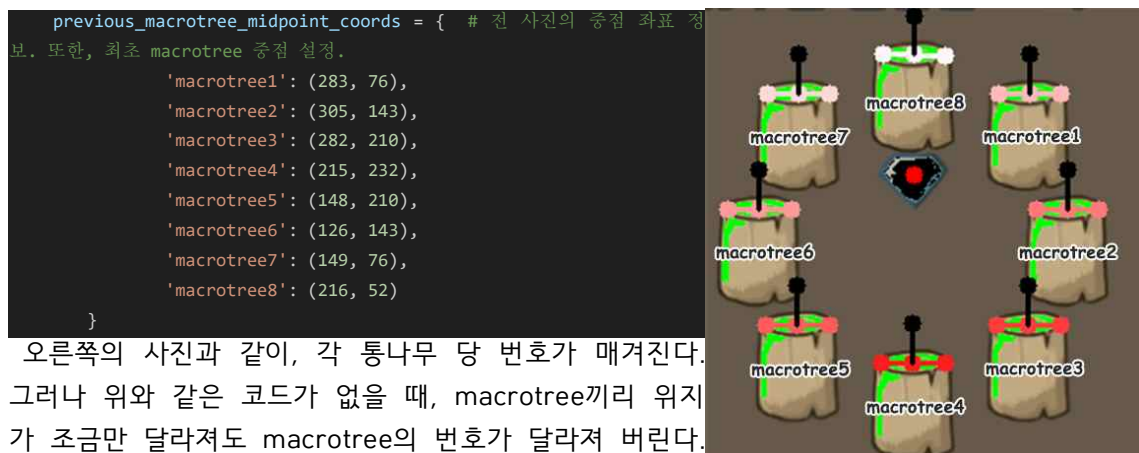
    # new_point와 new_point2 중 무엇이 더 이전 사진의 중점들과 가까운지
    if min_distance_new_point > min_distance_new_point2:
        new_point, new_point2 = new_point2, new_point
        rgb_color1, rgb_color2 = rgb_color2, rgb_color1
```

다시 말하지만, 검정색 부분은 중점의 반대 방향이다. 해당 코드를 실행하면 왼쪽의 사진처럼 제대로 중점이 부여된 모습을 볼 수 있다. 먼저, point 2개 중, 무엇이 background_color과 더 가까운지 체크한다(background_color는 brown을 GBR화 한 것과 매우 비슷하다). 더 비슷하지 않은 쪽을 중점으로 선정한다. 만약 두 point가 색이 똑같다면, 이전 사진의 중점의 좌표 `previous_macrotree_midpoint_coors`와의 거리를 각각 비교하여 더 가까운 중점을 진짜 중점으로 선정한다.

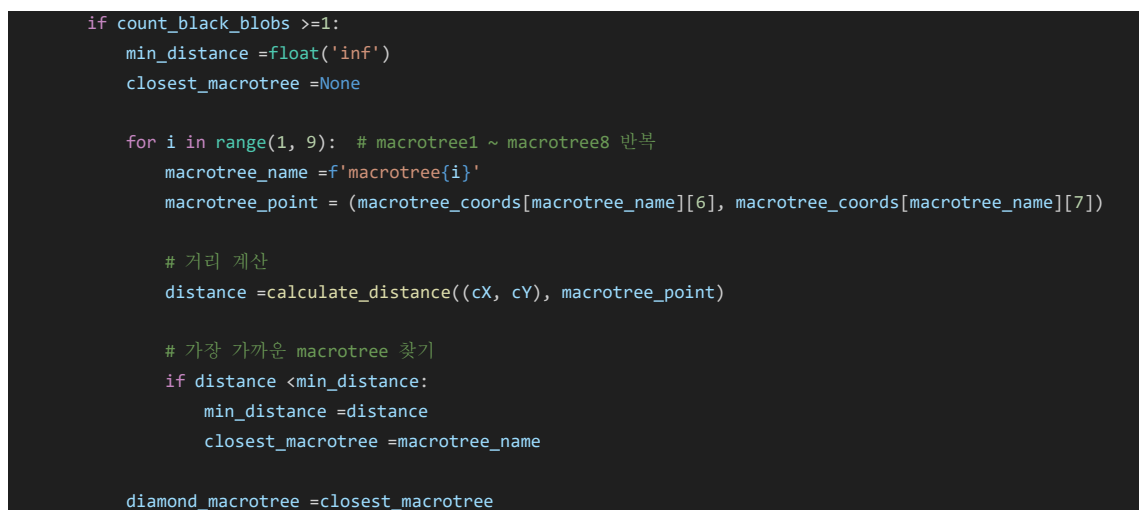
```
# new_point와 가장 가까운 previous_macrotree_midpoint_coors 찾기
```



해당 코드는 `previous_macrotree_midpoint_coors`와 가장 가까운 통나무의 중점에 같은 이름을 부여하는 과정이다.



오른쪽의 사진과 같이, 각 통나무 당 번호가 매겨진다. 그러나 위와 같은 코드가 없을 때, macrotree끼리 위치가 조금만 달라져도 macrotree의 번호가 달라져 버린다. 위치가 바뀌더라도 지속적으로 가장 가까운 중점의 macrotree에게 번호를 계승함으로써 번호가 바뀌지 않고 계속 유지될 수 있다.

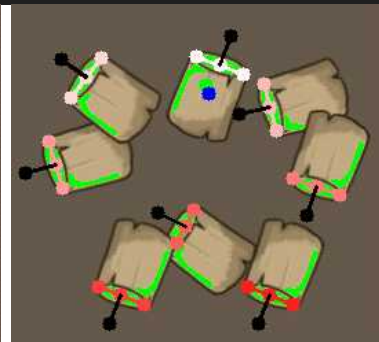
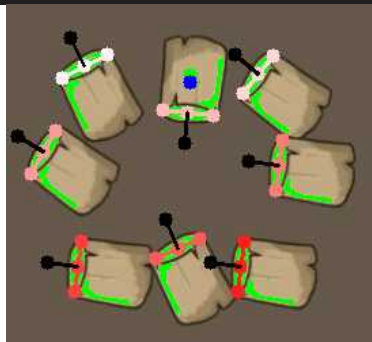
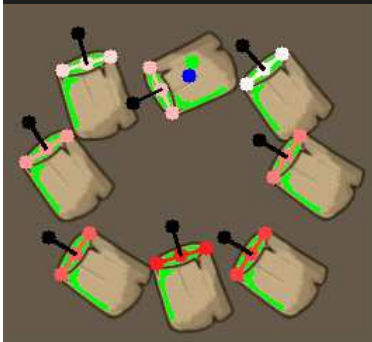



```
print(f"다이아몬드 트리는 {diamond_macrotree}")
```

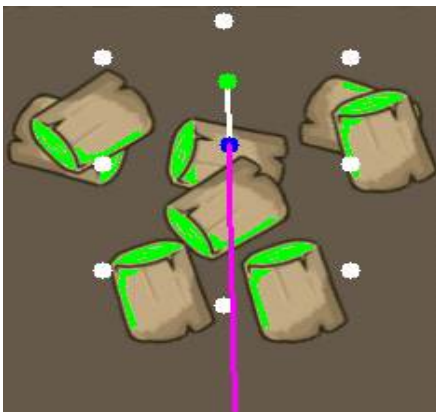
다이아몬드 트리는 **macrotree8**

해당 코드는 macrotree의 중점과 다이아몬드의 중점 좌표(cX, cY)의 거리를 비교하여 가장 가까운 macrotree를 선별한다. 이 다이아몬드가 스며든 macrotree의 이름을 diamond_macrotree에 저장한다.

```
if previous_macrotree_coords[diamond_macrotree][6] !=0 and previous_macrotree_coords[diamond_macrotree]
!=macrotree_coords[diamond_macrotree]:
    # 벡터 그리기
    cv2.arrowedLine(image, (previous_macrotree_coords[diamond_macrotree][6],
previous_macrotree_coords[diamond_macrotree][7]), (macrotree_coords[diamond_macrotree][6],
macrotree_coords[diamond_macrotree][7]), (255, 0, 0), 2)
    # 점을 그리기
    cv2.circle(image, (previous_macrotree_coords[diamond_macrotree][6],
previous_macrotree_coords[diamond_macrotree][7]), 5, (0, 255, 0), -1) # 시작점 (녹색)
    cv2.circle(image, (macrotree_coords[diamond_macrotree][6], macrotree_coords[diamond_macrotree][7]), 5, (255,
0, 0), -1) # 끝점 (파란색)
    print(f"시작: {(previous_macrotree_coords[diamond_macrotree][6],
previous_macrotree_coords[diamond_macrotree][7])} -> 끝: {(macrotree_coords[diamond_macrotree][6],
macrotree_coords[diamond_macrotree][7])}")
    if movecount ==0:
        start_coords =list(previous_macrotree_coords[diamond_macrotree][6:8]) # [6], [7]을 리스트로 변환
        movecount +=1
    lastvector =True
```



전 사진의 diamond_macrotree의 중점 좌표가 초록색 점, 현재 diamond_macrotree의 중점 좌표가 파란색 점이다. (참고로, 붉은 색 선의 색 차이는 macrotree의 번호와는 전혀 상관이 없다. `new_point`와 가장 가까운 `previous macrotree midpoint coords` 찾기를 참고하자.) 따라서 우리는 시간이 지남에 따른 diamond_macrotree의 이동 방향, 즉 **벡터**를 알 수 있게 되었다.



`for i in range(8, 25):`반복문이 끝나면서부터 코드의 끝 부분까지, 이 벡터를 구하고 온갖 변수(예: 섞이면서 전체적으로 위로 올라가는 것에 대한 보정치, 가운데를 기준으로 포물선을 그리면서 섞임 등)에 맞게 벡터를 가공하고, 이 벡터와 가장 가까운 하얀색 점을 구하여 아래와 같이 출력한다.

```
macrotree2 64.78016372986997
macrotree3 63.5839258699388
macrotree4 3.674227468704632
macrotree5 68.20412633165685
macrotree6 67.01600646972656
결과 macrotree: macrotree4
```

이렇게 나오면, `macrotree click coords`의 'macrotree4'를 클릭하게 된다. 아래는 클릭 코드이다.

```
if result_macrotree in macrotree_coords:
    coords =macrotree_click_coords[result_macrotree]
    pyautogui.moveTo(coords[0], coords[1])
    pyautogui.doubleClick()
```

```
time.sleep(0.01)
pyautogui.doubleClick()
else:
    print("오프: 잘못된 macrotree 이름입니다.")
```