소프트웨어개발도구및환경실습 최종 프로젝트 보고서

컴퓨터공학과 20211584 장준영

1. 실험 환경

OS	Windows 10 Pro 22H2, x64-based processor
Processor	AMD Ryzen 5 5600X 6-Core Processor 3.70GHz NVIDIA GeForce RTX 3070 Ti 8GB
IDE	Visual Studio 2022, openFrameworks(C/C++)

openFrameworks 라이브러리를 포함할 수 있는 디렉토리 내의 **Visual Studio 2022의 솔루션**에서 해당 프로그램을 빌드할 수 있음. windows.h 헤더가 포함되어 있으므로 반드시 **Windows 운영 체제**에서 빌드해야 함. 빌드된 프로그램은 솔루션 디렉토리 내의 **bin/prj2.exe**을 실행하면 됨.

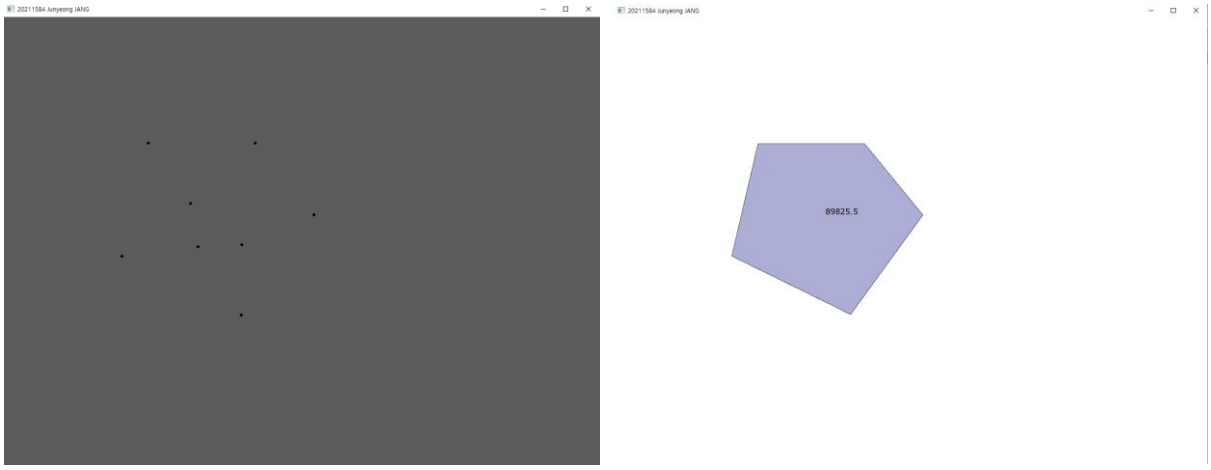
2. 프로젝트 목표

소프트웨어개발도구및환경실습 과목에서 학습한 내용을 두루 심화하여 최종 프로젝트에 녹여내고자 하였다. 주요 개념으로 학습하였던 (1)그래프 내의 경로 탐색(Maze : BFS, DFS), (2)기하 알고리즘(WaterFall : 선분 위 점 판정), (3)경로 추천(Tetris : Tree DFS) 등을 바탕으로, 최종 프로젝트에선 심화하여 '커스텀 필드에서의 최단 경로 추천 시스템'을 구현한다. 이를 위해 (1)Convex Hull, 선분 교차 판정, 래스터라이징 등의 기하 알고리즘, (2)생성된 커스텀 필드에서 두 점의 최단 경로를 찾기 위한 BFS 알고리즘을 설계하고 코드로 작성할 것이다. 또, 결과의 그래픽 표현을 위해 여러 자료 구조와 openFrameworks의 메써드를 적절히 활용할 수 있어야 한다.

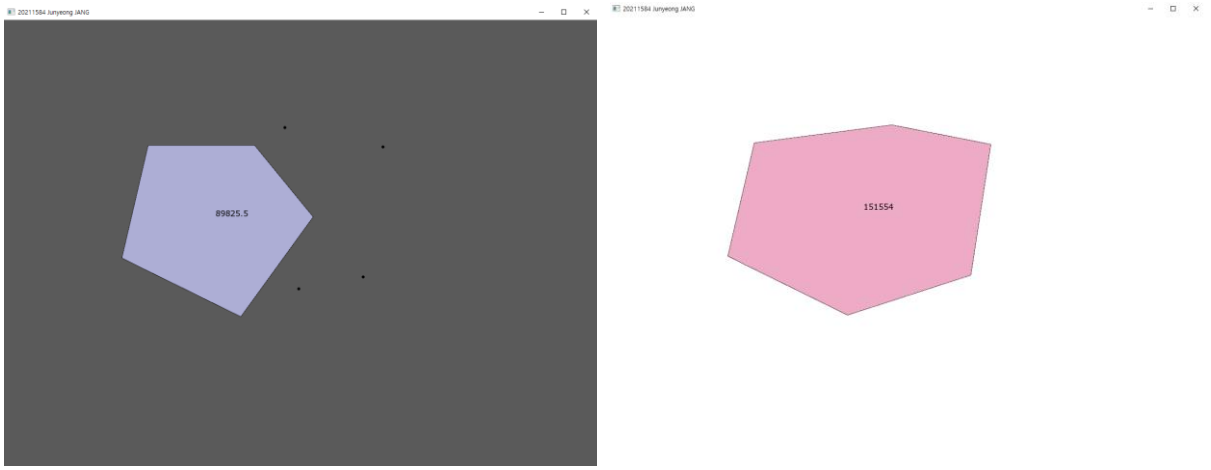
3. 프로젝트 실행 결과

(※ 자세한 동작은 발표 동영상의 데모에서 확인 바람.)

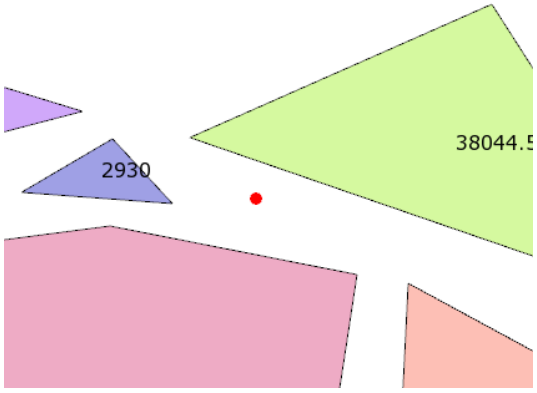
릴리즈 모드로 빌드된 executable file을 실행하면, 구현의 결과를 확인할 수 있다.



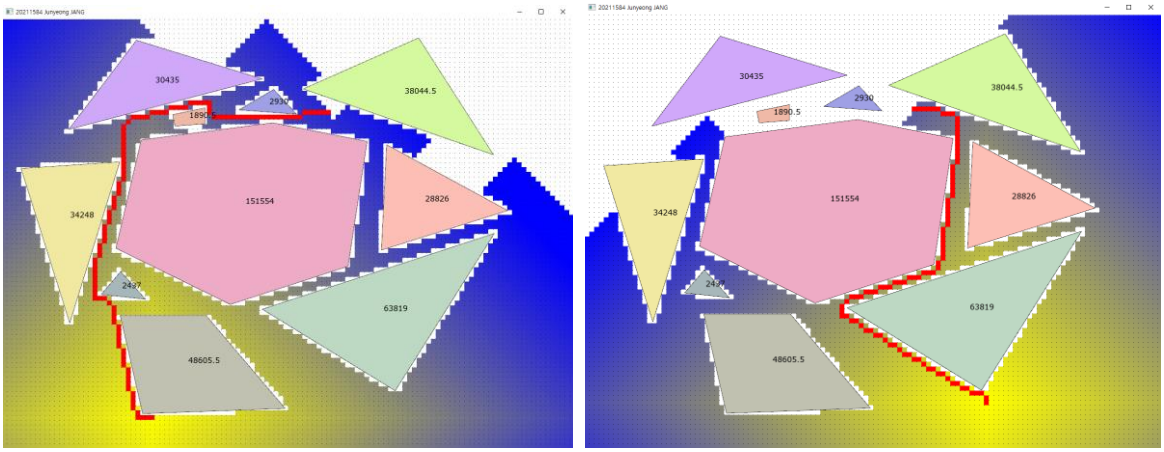
스페이스 바를 눌러 '점 추가 모드 / 폴리곤 생성 모드'를 토글링 할 수 있다. 배경색이 회색으로 일렁이고 배경 음악이 작아지면 점 추가 모드에 있는 상태로, **마우스 좌클릭**을 통해 필드에 최대 20개까지 점을 추가할 수 있다. 다시 **스페이스 바**를 눌러 폴리곤 생성 모드에 진입하면, 찍힌 점을 바탕으로 해당 점들의 convex hull 폴리곤을 필드에 생성한다. 폴리곤의 중심에는 해당 폴리곤의 면적이 쓰여 있다.



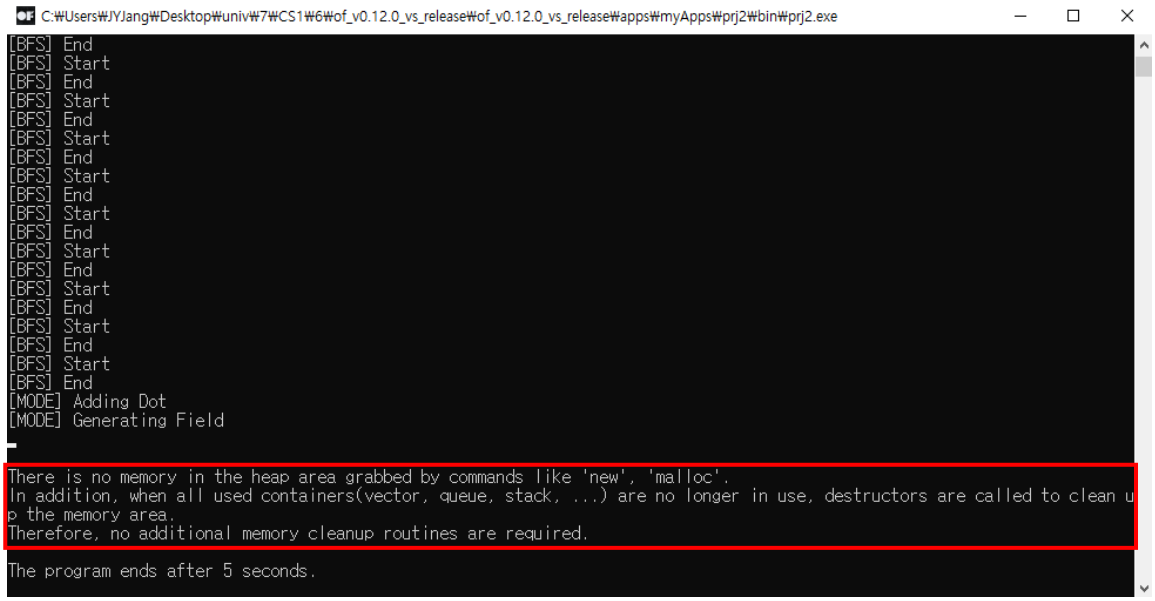
만약 생성될 예정인 폴리곤이 기존에 필드에 존재하던 폴리곤과 겹친다면, 겹치는 모든 폴리곤의 꼭짓점을 집합으로 하여 새로운 convex hull을 생성한다.



윈도우 내에 마우스 커서를 두고 **휠 버튼을 클릭**하면, 빨간 점이 생성된다. 해당 점은 도착점으로, 이후 BFS를 통해 해당 지점에 도달할 수 있는 최단 경로를 찾게 된다.



윈도우 내에 마우스 커서를 두고 **우클릭**을 하면, 필드를 $^{1}/_{10}$ 비율로 래스터라이징한 격자점 내에서 우클릭한 지점부터 BFS를 시작한다. BFS를 통해 방문한 점은 색이 칠해져 확인할 수 있다. 앞서 **마우스 휠 클릭**을 통해 지정한 도착점에 도달하면, 탐색을 멈추고 최단 경로를 보여준다. 추가적으로, 키보드에서 **R 키**를 누르면 필드의 모든 폴리곤과 찍은 점이 사라지고 초기 상태로 돌아간다.

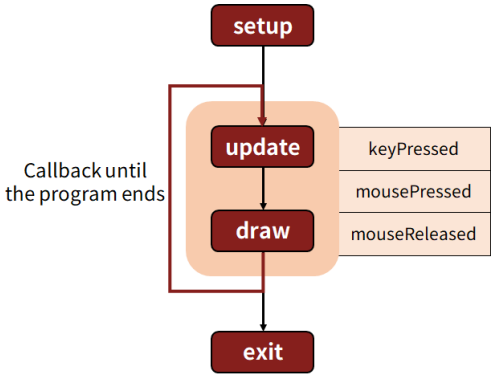


콘솔에서는 내가 현재 어떤 모드에 있는지, BFS의 진행 상황 등을 알 수 있다. 윈도우에서 **ESC 키**를 누르면 종료 안내 메시지와 함께 5초 뒤에 프로그램이 종료된다.

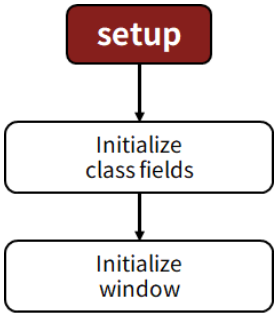
4. 프로젝트 내용

oF의 메써드를 필요한 상황에 알맞게 사용한다. 프레임워크에서 사용되는 메써드는 setup, update, draw, exit, keyPressed, mousePressed, mouseReleased이다. 자세한 자료 구조 및 알고리즘은 **5. 프로젝트 구현**에서 후술한다. 다음은 전체 프로그램과 사용된 메써드의 플로우 차트이다.

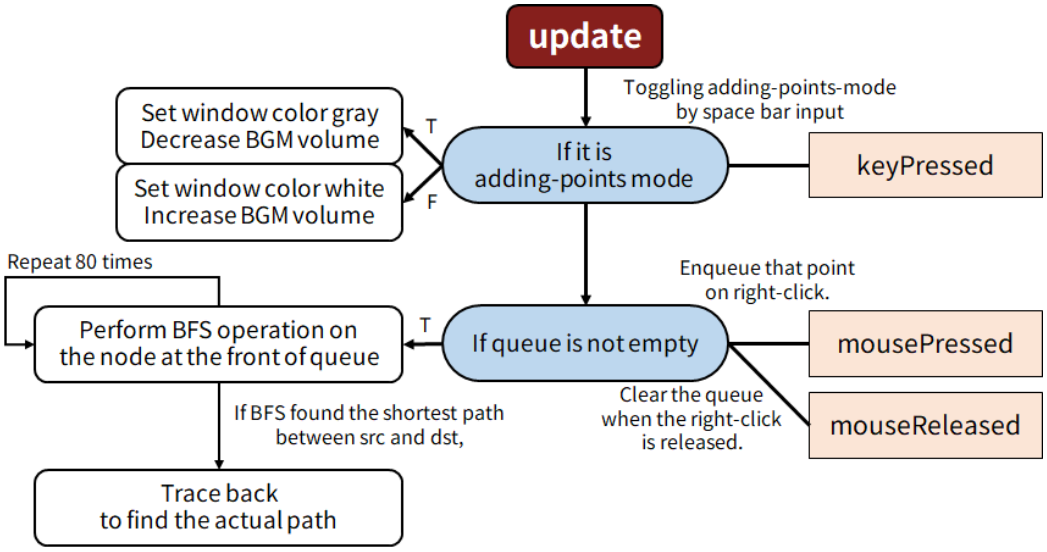
[1] openFrameworks 프로그램 전체



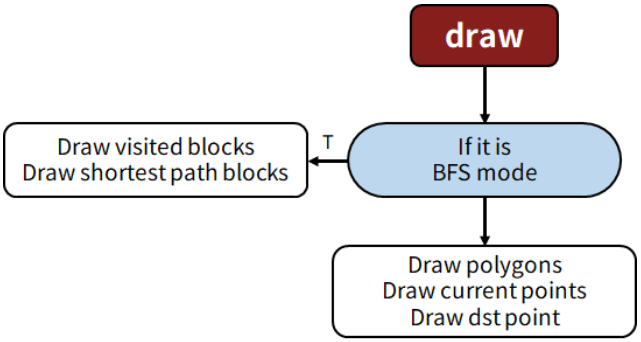
[2] setup 메써드



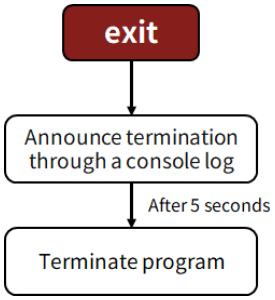
[3] update 메써드



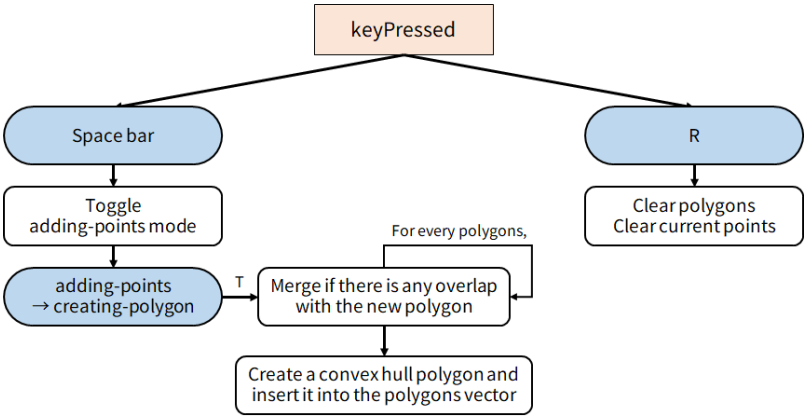
[4] draw 메써드



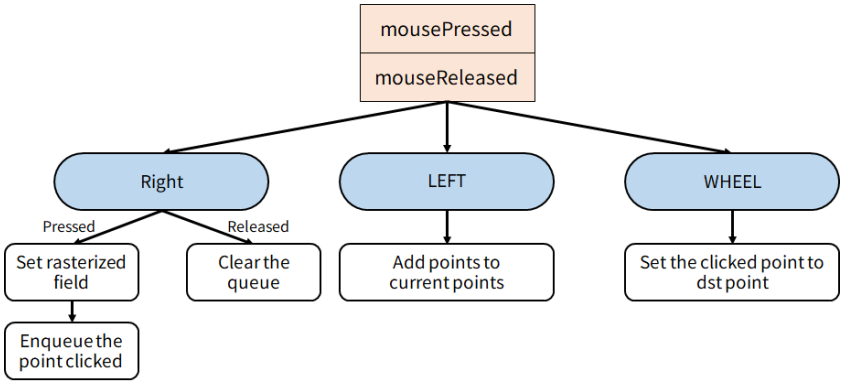
[5] exit 메써드



[6] keyPressed 메써드



[7] mousePressed 메써드

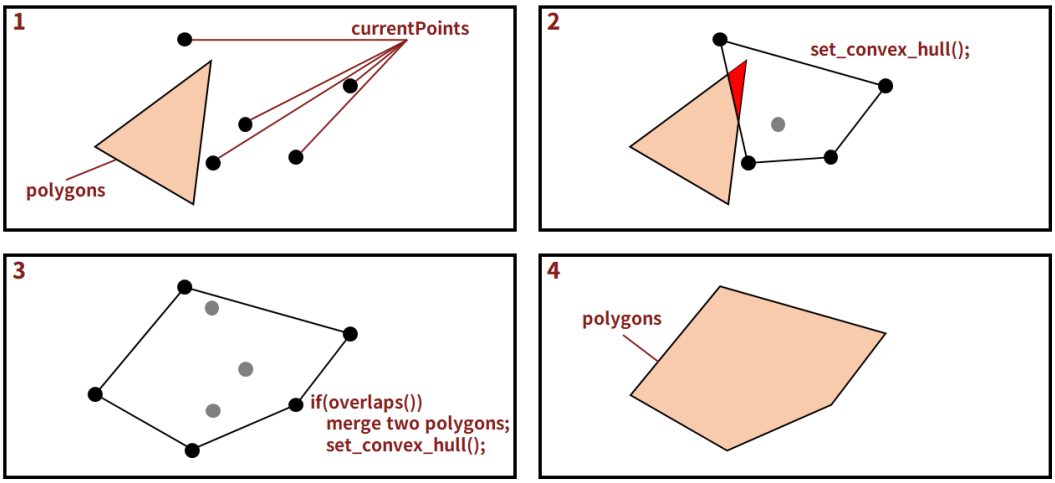


해당 플로우 차트를 통해 사용자와의 인터랙션이 어떻게 이루어지는지, 어떤 부분에서 필요한 알고리즘이 호출되는지를 알면 이후 설명될 알고리즘에 대해 이해하기 쉽다.

5. 프로젝트 구현

이 프로젝트는 총 세 단계의 알고리즘 진행으로 이루어져 있다. (1)사용자가 찍은 점을 바탕으로 convex hull 폴리곤을 생성하는 단계, (2)모든 폴리곤을 $^{1}/_{10}$ 비율의 격자점을 갖는 필드에 래스터라이징하는 단계, (3)시작점과 도착점을 잇는 최단 경로를 BFS로 찾는 단계가 있다.

[1] convex hull 폴리곤 생성



1단계의 과정은 위 이미지와 같다. 유저가 직접 점을 찍고 폴리곤 생성 모드로 변경했을 때, 해당 점을 감싸는 convex hull 폴리곤을 생성한다. 이후 필드에 존재하는 모든 폴리곤에 대해, 새로운 폴리곤과 겹치는 부분이 존재할 경우 두 폴리곤의 점을 하나의 집합으로 합쳐, 그 집합에서 다시 convex hull을 찾는다. 이 과정을 통해 폴리곤 생성을 마칠 수 있다.

```
struct POINT2D
{
    int x, y; /* coordinates of a point */
    POINT2D() : x(-1), y(-1) {}
    POINT2D(int X, int Y) : x(X), y(Y) {}
};

struct POLYGON
{
    vector<POINT2D> convexhull; /* points on a convex hull polygon */
    double area;                /* area */
    int r, g, b;                /* color */
    void set_convex_hull();
};
```

```cpp
void POLYGON::set_convex_hull()
{
    /* Calculate points and area on convex hull. */
    convexhull = convexHull(convexhull);
    area = polygonArea(convexhull);
    /* Randomize color. */
    r = 255 - ofRandom(100);
    g = 255 - ofRandom(100);
    b = 255 - ofRandom(100);
}
```

이 알고리즘에서 점은 POINT2D, 폴리곤은 POLYGON 타입으로 선언된다. POINT2D에는 점의 x, y 좌표 필드가 존재하고, POLYGON에는 convex hull을 이루는 점들의 벡터, 해당 폴리곤의 면적, 해당 폴리곤의 색(r, g b) 필드가 존재한다. POLYGON 타입의 convexhull 벡터에 임의의 점들을 추가하고 set_convex_hull() 메써드를 호출하면 자동으로 convex hull 위의 점만 남고, 면적과 색을 정해서 필드에 저장한다.

```cpp
/* convexHull - Funtion to store the points of the convex hull in a vector. */
vector<POINT2D> convexHull(vector<POINT2D> &points)
{
    int n = points.size();
    if (n < 3)
        return {};

    /* Find the bottommost point. */
    int ymin = points[0].y, minn = 0;
    for (int i = 1; i < n; i++)
    {
        int y = points[i].y;

        /* Pick the bottom-most or chose the left most point in case of tie. */
        if ((y < ymin) || (ymin == y && points[i].x < points[minn].x))
            ymin = points[i].y, minn = i;
    }

    /* Place the bottom-most point at first position. */
    swap(points[0], points[minn]);

    /* Sort n-1 points with respect to the first point. */
    p0 = points[0];
    sort(points.begin() + 1, points.end(), compare);

    /* Create an empty stack and push first three points to it. */
    vector<POINT2D> hull;
    hull.push_back(points[0]);
    hull.push_back(points[1]);
    hull.push_back(points[2]);

    /* Process remaining n-3 points. */
    for (int i = 3; i < n; i++)
    {
        /* Keep removing top while the angle formed
         * by points next-to-top, top, and points[i] makes a non-left turn.
         */
        while (hull.size() > 1 && ccw(hull[hull.size() - 2], hull[hull.size() - 1], points[i]) != 1)
            hull.pop_back();
        hull.push_back(points[i]);
    }

    return hull;
}
```

convexHull 함수는 Graham's scan 알고리즘을 사용한다. 집합 내의 모든 점을 특정 최외곽 점부터 (각도 면의) 회전하며 확인하면서, 포함시킨 점 중 가장 최근의 세 점이 항상 특정 방향을 이루도록 한다. 포함시킨 모든 점이 순서대로 한 방향만으로 흘러갈 때, 해당 집합이 convex hull이 된다. 이 알고리즘의 시간 복잡도는 다음과 같다.

시간 복잡도 : $O(N \log N)$

점을 각도 기준 정렬하는 연산이 시간 복잡도 면에서 가장 큰 연산이다. 정렬 이후엔 모든 점을 최대 두 번씩 참조하면서 연산을 마치기 때문에, subquadratic한 시간 복잡도 내에 convex hull을 찾을 수 있다.

```
/* polygonArea - Function to calculate the area of a polygon given its vertices. */
double polygonArea(vector<POINT2D> &points)
{
    int n = points.size();
    double area = 0.0;

    /* Loop through each vertex (except the first and last) */
    for (int i = 1; i < n - 1; i++)
        /* Calculate part of the determinant (shoelace formula) and add to the total area.
        * This part calculates the area of the triangle formed by points[0], points[i], and points[i+1]
        * and adds/subtracts it from the total area accordingly.
        */
        area += (points[i].x - points[0].x) * (points[i + 1].y - points[i].y) - (points[i].y - points[0].y) * (points[i + 1].x - points[i].x);
    /* The formula gives twice the actual area and might be negative depending on the vertices' order,
    * so take the absolute value and divide by 2 to get the actual area.
    */
    return abs(area / 2.0);
}
```

넓이를 계산하는 polygonArea() 함수는 가장 떠올리기 쉬운 shoelace formula를 사용한다. 폴리곤의 모든 점에 대해, 특정 기준점을 포함하는 삼각형으로 쪼개 그 삼각형 각각의 넓이를 계산하여 더한다. 이 알고리즘의 시간 복잡도는 다음과 같다.

시간 복잡도 : $O(N)$

모든 점을 최대 두 번씩 참조하여 연산을 마칠 수 있고, shoelace formula는 상수 시간 내에 계산이 가능한 단일식이기 때문에 선형 시간 복잡도를 갖는다.

```
vector<POINT2D> currentPoints; /* vector to store the current input points */
vector<POLYGON> polygons;      /* vector to store all polygons */
```

currentPoints는 유저가 점 추가 모드에서 찍은 점의 좌표를, polygons는 필드에 존재하는 모든 폴리곤의 정보를 저장한다. 해당 정보를 저장하고 활용하여 필드에 convex hull 폴리곤을 생성하고 그린다.

```cpp
/* Create a new convex hull from currentPoints. */
vector<POINT2D> mergedPoints = convexHull(currentPoints);

/* Loop through all polygons */
for (auto it = polygons.begin(); it != polygons.end();)
{
    /* If the newly created polygon(mergedPoints) overlaps an existing polygon, */
    if (overlaps(it->convexhull, mergedPoints))
    {
        /* Merge two polygons' points. */
        mergedPoints.insert(mergedPoints.end(), it->convexhull.begin(), it->convexhull.end());
        mergedPoints = convexHull(mergedPoints);
        /* Delete the overlapping polygon and update the iterator. */
        it = polygons.erase(it);
    }
    /* Otherwise, */
    else
        /* Move to the next polygon. */
        ++it;
}

/* Create a new polygon from the merged points. */
POLYGON newPolygon;
newPolygon.convexhull = mergedPoints;
newPolygon.set_convex_hull();
polygons.push_back(newPolygon);
```

점 생성 모드에 돌입하면, 앞서 설명한대로 모든 폴리곤에 대한 overlap 판정을 통해 최종 폴리곤을 결정한다.

```cpp
/* overlaps - Functions to check if two polygon regions overlap */
bool overlaps(const vector<POINT2D> &polygon1, const vector<POINT2D> &polygon2)
{

    /* Check if any point of polygon1 is inside polygon2. */
    for (const POINT2D &point : polygon1)
    {
        if (isPointInsidePolygon(polygon2, point))
            return true;
    }

    /* Check if any point of polygon2 is inside polygon1. */
    for (const POINT2D &point : polygon2)
    {
        if (isPointInsidePolygon(polygon1, point))
            return true;
    }

    /* Check for edge intersection between the two polygons. */
    for (int i = 0; i < polygon1.size(); i++)
    {
        for (int j = 0; j < polygon2.size(); j++)
        {
            /* Calculate the next vertex index for the current edge in polygon. */
            int next_i = (i + 1) % polygon1.size();
            int next_j = (j + 1) % polygon2.size();

            /* Check if the current edges from polygon1 and polygon2 intersect. */
            if (isSegmentsIntersect(polygon1[i], polygon1[next_i], polygon2[j], polygon2[next_j]))
                return true;
        }
    }

    return false;
}
```

overlap 판정은 (1)A 폴리곤의 점이 B 폴리곤의 내부에 있을 때, (2)B 폴리곤의 점이 A 폴리곤의 내부에 있을 때, (2) 두 폴리곤의 선분 중 교차하는 것이 있을 때 true를 반환하고, 아니면 false를 반환한다. 내부 점 판정은 레이 캐스팅을 통해, 선분 교차 판정은 CCW를 활용한 선분 교차 판정을 통해 계산한다. 폴리곤 생성의 총 시간 복잡도는 다음과 같다.

---

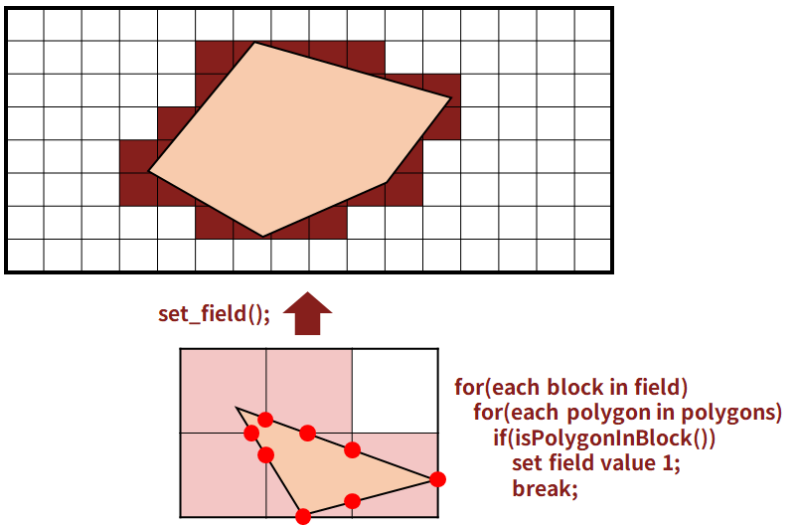
convex hull 생성 : $O(PN \log N)$

overlap 판정 : $O(P^2N)$

>> 시간 복잡도 : $\boldsymbol{O(P^2N) \approx O(P^2)}$

폴리곤 생성의 시간 복잡도가 cubic으로, 만약 P(폴리곤의 총 개수)와 N(폴리곤의 점의 개수)이 커지면 실시간으로 계산이 힘들어질 수도 있다. 실제 프로그램에선 N의 최대 값을 20으로 정하고 있어 quadratic 커버가 가능하다. 여러 기하 객체를 유지 관리하는 데에는 비용이 크게 들고, 이는 아무리 개선하여도 시간 복잡도와 공간 복잡도의 trade-off를 가질 수밖에 없다.

[2] 래스터라이징



2단계의 과정은 위 이미지와 같다. 10x10 크기의 블록으로 이루어진 필드에 대해, 해당 블록 위에 폴리곤이 존재하는지 판별하기 위해 아래와 같은 알고리즘을 사용한다. 폴리곤의 선분이 블록의 네 모서리 중 단 하나라도 교차하는 경우, 해당 블록에는 폴리곤이 존재한다(∵ Intermediate Value Theorem). 이를 판별하기 위해 역시 선분 교차 알고리즘을 사용한다.

```
char field[90][120];          /* block field */
void set_field();
```

1200x900 윈도우 크기에 대해, 120x90으로 축소된 블록 field를 set_field() 함수로 채운다.

```cpp
void ofApp::set_field()
{
    /* Loop through every block in field */
    for (int y = 0; y < 90; y++)
    {
        for (int x = 0; x < 120; x++)
        {
            bool flag = false;
            /* Loop through all polygons */
            for (const auto &polygon : polygons)
            {
                /* If a polygon exists in that block, */
                if (isPolygonInBlock(polygon, x, y))
                {
                    /* Set field blocked. */
                    field[y][x] = '#';
                    flag = true;
                    break;
                }
            }
            /* Otherwise, */
            if (!flag)
                /* Set field not blocked. */
                field[y][x] = '~';
        }
    }
#ifdef DEBUG_LOG
    printf("[ALERT] set_field complete\n");
#endif
}
```

```cpp
/* isPolygonInBlock - Function to check if the given polygon overlaps
 *                    with the block at grid position (i, j)
 */
bool isPolygonInBlock(POLYGON polygon, int i, int j)
{
    /* Get the corners of the block at grid position (i, j). */
    vector<POINT2D> blockCorners = getBlockCorners(i, j);
    /* Loop through each edge of the polygon's convex hull */
    for (size_t p = 0; p < polygon.convexhull.size(); p++)
    {
        /* Get the current vertex and the next vertex of the polygon. */
        POINT2D p1 = polygon.convexhull[p];
        POINT2D p2 = polygon.convexhull[(p + 1) % polygon.convexhull.size()];
        for (size_t b = 0; b < blockCorners.size(); b++)
        {
            /* Get the current vertex and the next vertex of the block. */
            POINT2D b1 = blockCorners[b];
            POINT2D b2 = blockCorners[(b + 1) % blockCorners.size()];
            /* Check if the current edge of the polygon intersects
             * with the current edge(corner) of the block.
             */
            if (isSegmentsIntersect(p1, p2, b1, b2))
                return true;
        }
    }
    return false;
}
```

모든 블록에 대해 어떤 한 폴리곤이라도 존재한다면 지나갈 수 없도록 필드를 설정한다. 시간 복잡도는 다음과 같다.

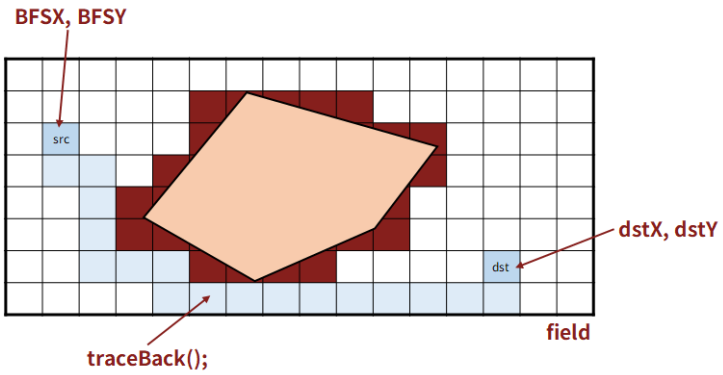
시간 복잡도 : $\boldsymbol{O(WHPN)} \approx \boldsymbol{O(WHP)}$

모든 필드 블록(WH)에 대해 모든 폴리곤의 선분 정보(PN)를 사용하므로 $WHPN$가 된다. 역시

N을 20 이하의 상수로 설정하고 있기 때문에 cubic 커버가 가능하다.

---

[3] BFS



2단계에서 생성된 필드를 통해 3단계에선 BFS로 최단 경로 찾기 찾기를 시작한다. Maze 프로젝트에서 진행했던 것과 아주 유사하게 BFS를 통해 경로를 찾으면 된다.

```
pair<int, int> prev[90][120]; /* vector to store previous path */
int BFSX, BFSY;               /* BFS starting point */
int dstX, dstY;               /* BFS destination point */
```

경로 찾기를 위한 prev, 시작점 (BFSX, BFSY), 도착점 (dstX, dstY)를 통해 BFS로 최단 경로를 찾는다.

```
    /** BFS **/
    /* Loop through max(q.size(), 100) */
    for (int i = 0; i <= 80 && !q.empty(); i++)
    {
        /* Retrieves the current position from the queue. */
        int x = q.front().second;
        int y = q.front().first;
        q.pop();

        for (int i = 0; i < 4; i++)
        {
            /* Move up, down, left, and right. */
            int nx = x + dx[i];
            int ny = y + dy[i];

            /* Check if the position is movable within the field's range. */
            if (nx >= 0 && nx < 120 && ny >= 0 && ny < 90)
            {
                if (field[ny][nx] == '~') /* not visited */
                {
                    /* Enqueue the movable position and mark it with '*'(visited). */
                    q.push({ny, nx});
                    field[ny][nx] = '*';
                    /* Store the previous coordinates moved from. */
                    prev[ny][nx] = {x, y};
                    /* If the destination is reached,
                     * clear the queue and find the shortest path.
                     */
                    if (nx == dstX && ny == dstY)
                    {
                        q = queue<pair<int, int>>();
#ifdef DEBUG_LOG
                        printf("[ALERT] traceBack start\n");
#endif
                        traceBack(dstX, dstY);
#ifdef DEBUG_LOG
                        printf("[ALERT] traceBack end\n");
#endif
```

BFS 진행 상황을 시간이 지남에 따라 확인할 수 있게 하도록 update 메써드에서 한 번에 80개의 노드만 탐색하도록 설정하였다. 이후 과정은 일반적인 BFS와 같다. 시간 복잡도는 다음과 같다.

시간 복잡도 : $\boldsymbol{O(WH)}$

최악의 경우 필드의 전체 노드(WH)를 탐색해야 하므로 quadratic 한 시간 복잡도를 갖는다.

이렇게 전 과정에 대한 시간 복잡도를 계산해보았다. 전체 자료 구조에 대한 공간 복잡도는 다음과 같다.

---

field, prev : $O(WH)$

polygons : $O(PN) \approx O(P)$

>> 공간 복잡도 : $\boldsymbol{O(WH + P)}$

(※ 코드에 주석이 상세하게 달려있으니 함수에 대한 매우 자세한 동작은 코드를 참고 바람.)

6. 느낀 점 및 개선 사항

역시 시간 복잡도가 큰 것이 가장 아쉬웠다. 이를 개선하기 위해 자료 구조를 여러 부분으로 나눠 meet in the middle과 같은 기법을 사용할 수 있다. 또, 폴리곤 별로 거리 기준 overlap 가능성이 있는 다른 폴리곤을 저장하고 있는 등 추가적인 공간을 사용해 탐색 수를 줄일 수도 있다. 하지만 역시 이 모든 방법은 시간 복잡도와 공간 복잡도의 trade-off를 동반한다. 기하 객체를 여럿 관리하면서도 시간과 공간 면에서 모두 효율적인 알고리즘을 학습해보면 좋을 것 같다.

*** 한 학기 동안 수고 많으셨습니다. ***