


Implementacja otoczki wypukłej i quad tree w prostej grze typu “Flappy Bird”

Omówienie algorytmu otoczki wypukłej (Grahama)

Algorytm Grahama (Otoczka wypukła):

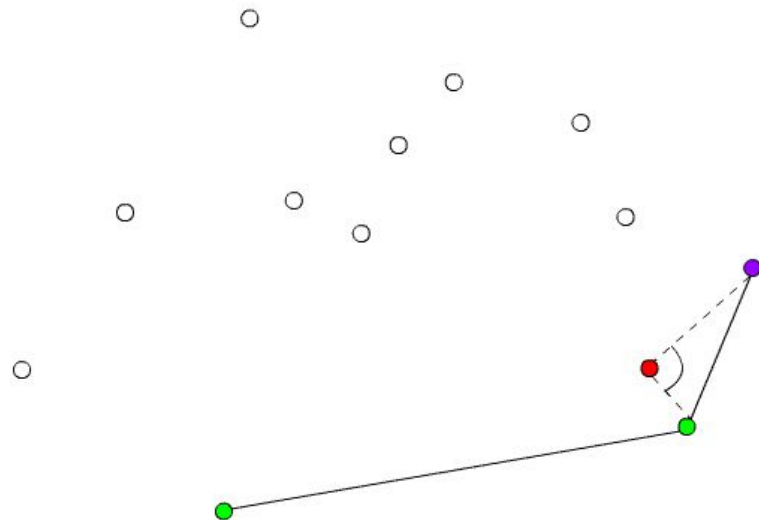
Cel: Znalezienie wielokąta wypukłego o najmniejszej ilości krawędzi, który zawiera wszystkie punkty z danej zbioru.

Etapy:

- Sortowanie punktów według współrzędnych.
 - Budowanie dolnej otoczki: Iteracja przez punkty i usuwanie tych, które nie są częścią otoczki.
 - Budowanie górnej otoczki: Proces analogiczny jak dla dolnej otoczki, ale w odwrotnej kolejności.
 - Scalenie dolnej i górnej otoczki w końcowy wynik.
- 

Działanie algorytmu grahama

```
def convex_hull(points):  
    points = sorted(set((p.x, p.y) for p in points))  
  
    if len(points) <= 1:  
        return points  
  
    def cross(o, a, b):  
        return (a[0] - o[0]) * (b[1] - o[1]) - (a[1] - o[1]) * (b[0] - o[0])  
  
    lower = []  
    for p in points:  
        while len(lower) >= 2 and cross(lower[-2], lower[-1], p) <= 0:  
            lower.pop()  
        lower.append(p)  
  
    upper = []  
    for p in reversed(points):  
        while len(upper) >= 2 and cross(upper[-2], upper[-1], p) <= 0:  
            upper.pop()  
        upper.append(p)  
  
    return lower[:-1] + upper[:-1]
```



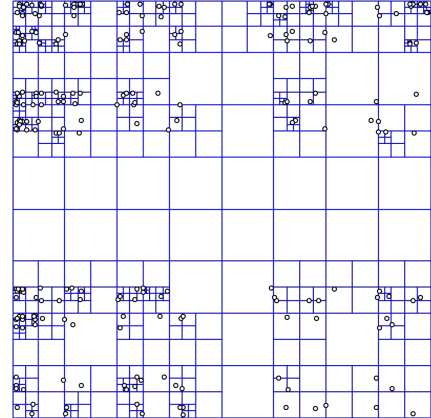
Quad Tree:

Cel: Struktura danych do efektywnego przechowywania i wyszukiwania punktów w 2D.

Podział przestrzeni: Kwadrat podzielony na cztery mniejsze kwadraty, z możliwością dalszego podziału, gdy liczba punktów przekroczy określoną wartość (pojemność).

Operacje:

- Wstawianie punktu: Sprawdzenie czy punkt należy do aktualnej przestrzeni, a następnie wstawienie go lub podział na mniejsze poddrzewa.
- Wyszukiwanie punktów: Przeszukiwanie odpowiednich poddrzew, które przecinają się z daną przestrzenią.



```
class QuadTree:
    def __init__(self, boundary, capacity):
        self.boundary = boundary # Rect: x, y, width, height
        self.capacity = capacity
        self.points = []
        self.divided = False
```

1 usage

```
def subdivide(self):
    x, y, w, h = self.boundary
    nw = (x, y, w / 2, h / 2)
    ne = (x + w / 2, y, w / 2, h / 2)
    sw = (x, y + h / 2, w / 2, h / 2)
    se = (x + w / 2, y + h / 2, w / 2, h / 2)
    self.northwest = QuadTree(nw, self.capacity)
    self.northeast = QuadTree(ne, self.capacity)
    self.southwest = QuadTree(sw, self.capacity)
    self.southeast = QuadTree(se, self.capacity)
    self.divided = True
```

5 usages

```
def insert(self, point):
    #x, y = point
    if not self.contains(self.boundary, point):
        return False

    if len(self.points) < self.capacity:
        self.points.append(point)
        return True
    else:
        if not self.divided:
            self.subdivide()

        if self.northwest.insert(point):
            return True
        elif self.northeast.insert(point):
            return True
        elif self.southwest.insert(point):
            return True
        elif self.southeast.insert(point):
            return True
```

5 usages

Wyszukiwanie w Quad Tree

```
3 usages
def query(self, range, found):
    if not self.intersects(self.boundary, range):
        return False
    else:
        for p in self.points:
            if self.contains(range, p):
                found.append(p)
        if self.divided:
            self.northwest.query(range, found)
            self.northeast.query(range, found)
            self.southwest.query(range, found)
            self.southeast.query(range, found)
    return found
```

```
2 usages
@staticmethod
def contains(rect, point):
    x, y, w, h = rect
    px, py = point.x, point.y
    return (x <= px <= x + w) and (y <= py <= y + h)

1 usage
@staticmethod
def intersects(rect1, rect2):
    x1, y1, w1, h1 = rect1
    x2, y2, w2, h2 = rect2
    return not (x1 > x2 + w2 or x1 + w1 < x2 or y1 > y2 + h2 or y1 + h1 < y2)
```

Krótkie omówienie gry

Mechanika: Gracz steruje małym ptakiem (kołem), który porusza się pionowo, unikając przeszkód.

Sterowanie: Skok ptaka realizowany poprzez naciśnięcie spacji.

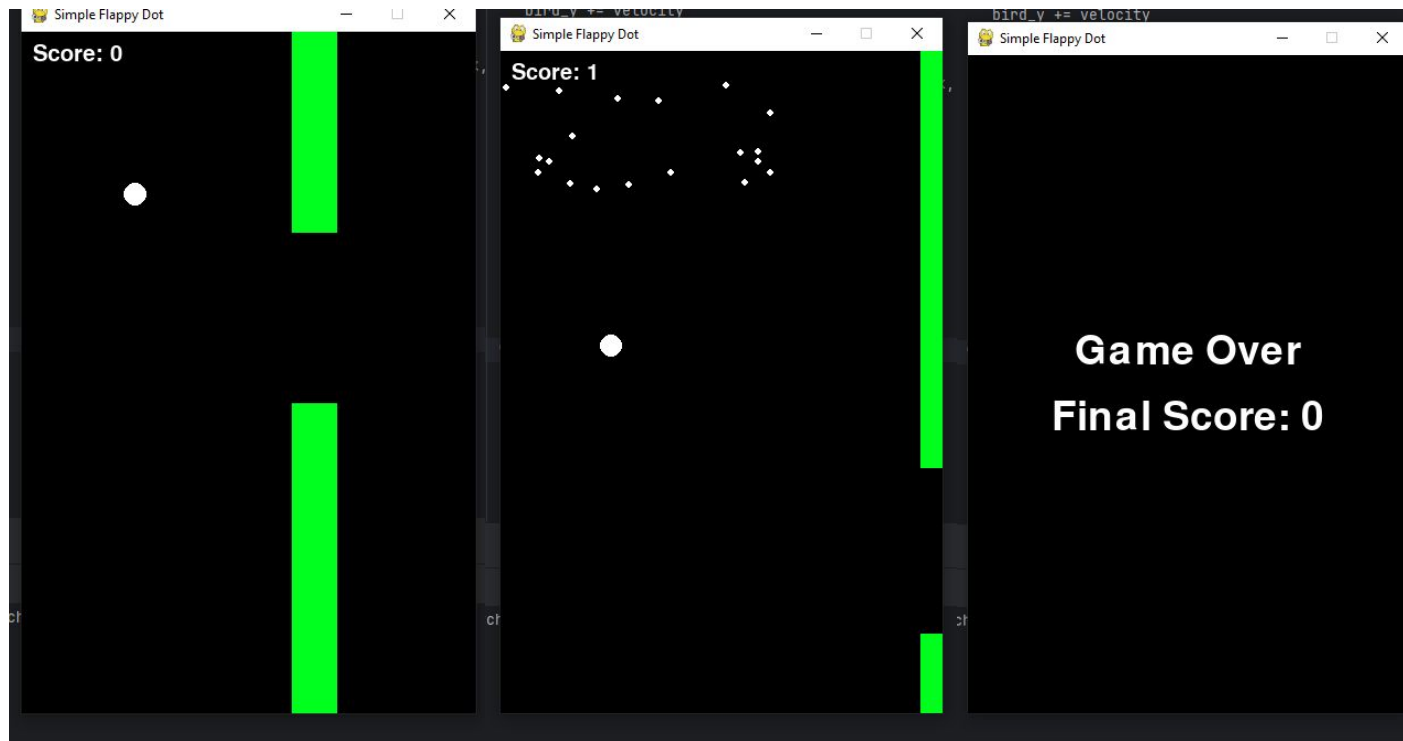
Elementy gry:

- Ptak: Porusza się w pionie pod wpływem grawitacji; gracz może kontrolować jego skoki.
- Przeszkody: Rury i chmury punktów, które gracz musi omijać.
- Punktacja: Gracz zdobywa punkty za każdą ominiętą (rurę).



przeszkodę

Screeny z gry



Implementacja otoczki wypukłej i Quad Tree w grze

Otoczka wypukła (Convex Hull):

Zastosowanie: Detekcja kolizji między ptakiem a przeszkodami.

Implementacja:

- Tworzenie otoczki wypukłej dla ptaka na podstawie punktów wyznaczających jego kształt.
- Tworzenie otoczki wypukłej dla każdej przeszkody (rur i chmur punktów).
- Sprawdzanie kolizji poprzez sprawdzenie przecięcia się otoczek wypukłych.



```
def collision_detection():

    if bird_y > height or bird_y < 0:
        show_game_over_screen()

    bird_points = [Point(bird_x + bird_radius * math.cos(theta), bird_y + bird_radius * math.sin(theta))
                   for theta in [math.radians(angle) for angle in range(0, 360, 30)]]
    bird_hull = convex_hull(bird_points)

    #rury
    for obstacle in obstacles:
        obstacle_up_points = [
            Point(obstacle['top']['x'], obstacle['top']['y']),
            Point(obstacle['top']['x'] + obstacle['top']['width'], obstacle['top']['y']),
            Point(obstacle['top']['x'], obstacle['top']['y'] + obstacle['top']['height']),
            Point(obstacle['top']['x'] + obstacle['top']['width'], obstacle['top']['y'] + obstacle['top']['height']),
        ]
        obstacle_down_points = [
            Point(obstacle['bottom']['x'], obstacle['bottom']['y']),
            Point(obstacle['bottom']['x'] + obstacle['bottom']['width'], obstacle['bottom']['y']),
            Point(obstacle['bottom']['x'], obstacle['bottom']['y'] + obstacle['bottom']['height']),
            Point(obstacle['bottom']['x'] + obstacle['bottom']['width'],
                  obstacle['bottom']['y'] + obstacle['bottom']['height'])
        ]
        obstacle_hull_up = convex_hull(obstacle_up_points)
        obstacle_hull_down = convex_hull(obstacle_down_points)
        if hull_intersection(bird_hull, obstacle_hull_down) or hull_intersection(bird_hull, obstacle_hull_up):
            show_game_over_screen()
```

Czy faktycznie nastąpiła kolizja?

```
2 usages
def is_point_in_hull(point, hull):
    if len(hull) < 3:
        return False
    for i in range(len(hull)):
        j = (i + 1) % len(hull)
        if (hull[j][0] - hull[i][0]) * (point.y - hull[i][1]) - (hull[j][1] - hull[i][1]) * (point.x - hull[i][0]) < 0:
            return False
    return True

3 usages
def hull_intersection(hull1, hull2):
    for p in hull1:
        if is_point_in_hull(Point(p[0], p[1]), hull2):
            return True
    for p in hull2:
        if is_point_in_hull(Point(p[0], p[1]), hull1):
            return True
    return False
```

Quad Tree:

Zastosowanie: Efektywne zarządzanie chmurami punktów.

Implementacja:

- Struktura Quad Tree dla przechowywania punktów chmur.
- Wstawianie punktów do Quad Tree.
- Wyszukiwanie punktów w obszarze ptaka w celu wykrywania kolizji.



Chmury punktów

```
# chmury i quad tree
if len(cloud_obstacles) < 1:
    return

boundary = (0, 0, width, height)
qtree = QuadTree(boundary, capacity=4)

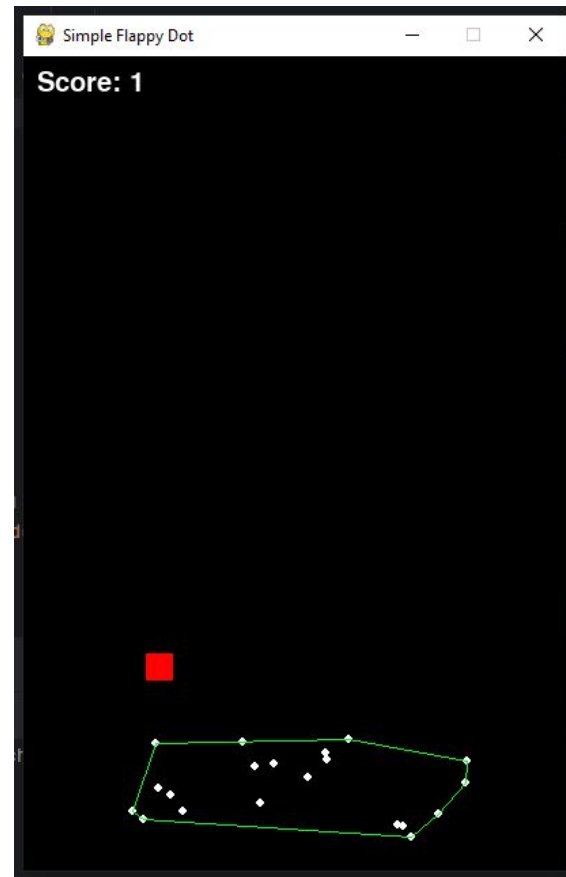
for cloud_obstacle in cloud_obstacles:
    for point in cloud_obstacle.points:
        qtree.insert(point)

bird_rect = pygame.Rect(bird_x - bird_radius, bird_y - bird_radius, bird_radius * 2, bird_radius * 2)
pygame.draw.rect(screen, red, bird_rect)
bird_query_rect = (bird_rect.x, bird_rect.y, bird_rect.width, bird_rect.height)

found_points = []
found = qtree.query(bird_query_rect, found_points)

if found:
    for found_point in found_points:
        hull = convex_hull(found_point.cloud.points)
        if hull_intersection(bird_hull, hull):
            show_game_over_screen()
```

Otoczka wypukła i obszar wyszukiwania w Quad Tree



Autorzy

- Patryk Krysta
- Jan Kubicius
- Krzysztof Lach
- Bartłomiej Karetko



Bibliografia

https://en.wikipedia.org/wiki/Quadtree#/media/File:Point_quadtree.svg

https://pl.wikipedia.org/wiki/Algorytm_Grahama

