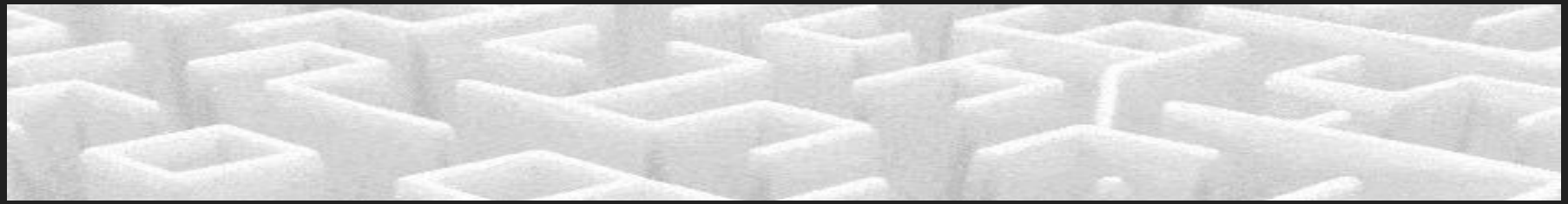
Патрулирование и навигация

Алгоритм поиска пути



Как бы человек искал выход, если бы его внезапно высадили посреди лабиринта?

Например, он мог бы воспользоваться правилом «одной руки», отмечать обследованные пути верёвкой или мелом, пока не найдёт выход.

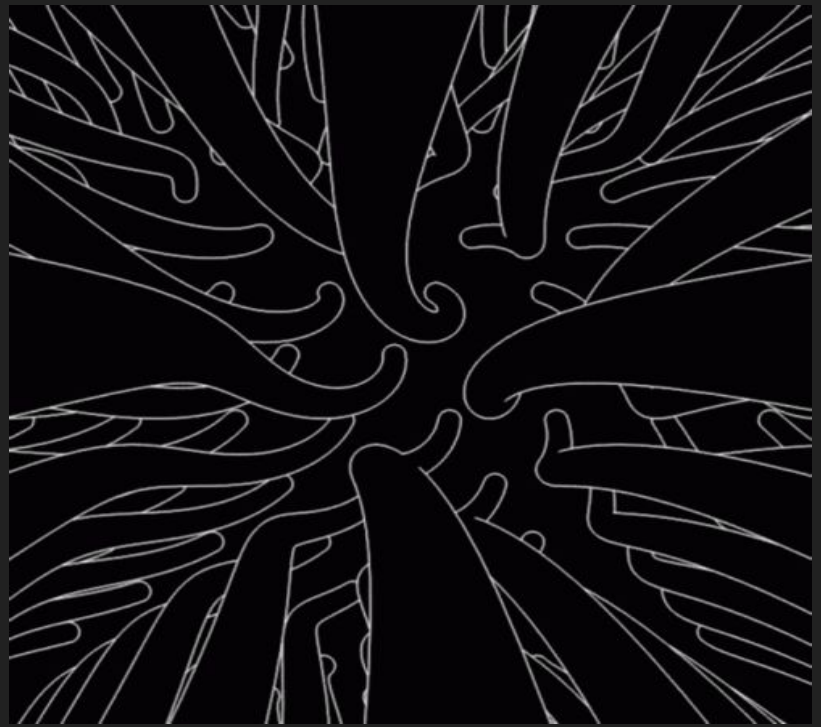
Большинство алгоритмов действуют по схожей схеме, за исключением того, что компьютер может «размножаться» и «ощупывать» одновременно несколько путей.

Различаются же алгоритмы точностью и скоростью получения результата.

Поиск в ширину

...предполагает исследование пути от начальной точки сразу во все стороны.

Сначала ощупываются соседние со стартом точки, потом соседние с ними и так далее, пока не найдётся конечная точка или поле не закончится.



Если же описывать работу в виде алгоритма, то получатся такие шаги:

1. Помещаем стартовую точку в список «на проверку».
2. Все точки из списка «на проверку» добавляем в список «исследованное».
3. Для всех точек из списка «на проверку» находим все соседние точки, на которые можно перейти (для обычной сетки это будут все соседние клетки без препятствий). Помещаем их в список «текущие».
4. Исключаем из списка «текущие» те точки, которые уже были исследованы и хранятся в списке «исследованное».
5. Список «на проверку» очищаем и помещаем туда все точки из списка «текущее». Текущий список тоже очищаем.
6. Повторяем алгоритм с шага 2, пока не будет найдена конечная точка или все доступные точки не будут проверены (то есть список «на проверку» окажется пустым).

Важно!

Такой алгоритм на каждом N цикле находит все точки, к которым можно дойти за N шагов. Чтобы получить путь, нужно запоминать не только проверенные точки, а ещё и последовательность точек до них.

Или хотя бы предыдущую точку, с которой мы пришли на эту, чтобы по шагам восстановить маршрут.

Поиск в ширину хорошо работает, если переход между соседними точками всегда занимает одинаковое количество времени (или другого параметра, по которому мы стараемся минимизировать путь).

Но если это не так, то он превращается в алгоритм Дейкстры.

Алгоритм Дейкстры

Алгоритм Дейкстры работает с моделями, в которых расстояния между точками могут быть разными.

Например, в Heroes of Might and Magic III проход по снегу тратит на 50% больше очков перемещения, так что условно можно посчитать, что проход по снегу занимает 1.5 шага вместо 1 по обычной земле.

В этом случае обход снега может быть быстрее, чем проход напрямую по нему, хоть визуально и будет пройдено большее расстояние.

От поиска в ширину алгоритм Дейкстры отличает пара моментов:

1. Кроме сохранения уже проверенных точек, сохраняется ещё и количество шагов, потраченных на то, чтобы добраться до них.
2. Точки исключаются из последующей проверки не тогда, когда были уже проверены, а только в случае, если предыдущий найденный путь до неё занимал меньшее количество «шагов».
3. Точки в списке рассматриваются не по порядку, сначала выбираются те, до которых меньше идти.

Алгоритм:

1. Помещаем стартовую точку и расстояние до неё в виде «0» в список «на проверку» и в список «исследованное».
2. Выбираем из списка «на проверку» точку с самым маленьким расстоянием до неё, удаляем её из списка.
3. Находим все соседние доступные точки и суммируем расстояние от выбранной точки до них с расстоянием до выбранной точки. Добавляем эту информацию в список «текущий».
4. Исключаем из списка «текущие» точки из списка «исследованное», расстояние до которых больше, чем значение расстояния в списке «исследованное».
5. Список «на проверку» очищаем и помещаем туда все точки из списка «текущее». Эти же точки помещаем в список «исследованное». Текущий список тоже очищаем.
6. Повторяем алгоритм с шага 2, пока не будет найдена конечная точка или все доступные точки не будут проверены (то есть список «на проверку» окажется пустым).

Эвристический алгоритм

Минусом как поиска в ширину, так и алгоритма Дейкстры является то, что компьютер ищет путь во все стороны сразу и тратит время на очевидно бесперспективные пути, которые отдаляют от цели, а не приближают к ней.

Эвристический алгоритм помогает это исправить.

Он работает точно так же, как и алгоритм Дейкстры, но с одним изменением. При выборе следующей точки на рассмотрение первой выбирается не та точка, которая ближе к началу пути, а та, что ближе к концу. Расстояние до конца рассчитывается приблизительно, например, просто как расстояние между двумя точками на карте.

Минус алгоритма в том, что найденный путь не обязательно будет самым кратким. Зато его нахождение потребует меньше времени.

YUKA JS API

https://mugen87.github.io/yuka/

Скачаем код

11Kotikov/tanks

Задать приоритет поведению танка

```
82  fleeBh.weight = 0.7;

88  wanderBh.active = false; // меняем с тру

89  wanderBh.weight = 0.4
```

Научим танк избегать препятствия

https://mugen87.github.io/yuka/docs/ObstacleAvoidanceBehavior.html

```
91  obstacalAvoidence = new YUKA.ObstacleAvoidanceBehavior

    vehicle.steering.add(obstacalAvoidence);
    obstacalAvoidence.active = true;
```

логика избегания препятствий

```
//функция создания препятствия
function makeObstacles() {
    //коробка препятствие
    let obstacleBoxMesh = new BABYLON.MeshBuilder.CreateBox();
    let obstacleMat = new BABYLON.StandardMaterial();
    obstacleMat.diffuseColor = new BABYLON.Color3(1, 0, 0);
    obstacleBoxMesh.material = obstacleMat;
}
```

Функция создания препятсвий

```
function makeObstacles() {
    //коробка препятствие
    let obstacleBoxMesh = new BABYLON.MeshBuilder.CreateBox();
    let obstacleMat = new BABYLON.StandardMaterial();
    obstacleMat.diffuseColor = new BABYLON.Color3(1, 0, 0);
    obstacleBoxMesh.material = obstacleMat;
    //создание препятсвия как сущность библиотеки ЮКА
    let obstacle = new YUKA.GameEntity();
    //синхронизируем и добавим компонет рендеринга
    obstacle.setRenderComponent(obstacleBoxMesh, sync);
    //позиция на карте
    obstacle.position = new YUKA.Vector3(5, 0, 5);

}
```

Добавим массив препятсвий

```
9. const obstacles = [];

function makeObstacles() {...
obstacleBoxMesh.rotationQuaternion = new BABYLON.Quaternion();


    obstacles.push(obstacle);
    entityManager.add(obstacle);
92  makeObstacles(); вызываем функцию и передаем параметры

    obstacalAvoidence = new YUKA.ObstacleAvoidanceBehavior(obstacles);

}
```

Создайте 5 препятствий

Это практическое задание:

Создайте 5 коробочек препятствий, необходимо сделать так, чтобы препятствия каждый раз появлялись в случайной позиции на карте

Создание пути по точкам (FollowPathBehavior)

```
let followPathBh = new YUKA.FollowPathBehavior();
    let path = new YUKA.Path();
    path.loop = true;
    path.add(new YUKA.Vector3(-8, 0, 0));
    path.add(new YUKA.Vector3(8, 0, 0));
    path.add(new YUKA.Vector3(5, 0, 0));
    path.add(new YUKA.Vector3(-5, 0, 0));
    path.add(new YUKA.Vector3(-10, 0, 0));
    followPathBh.path = path;
    followPathBh.active = true;
    vehicle.steering.add(followPathBh);
    vehicle.maxSpeed = 3;
```

Работа с мэшами

Навигационный Мэш

- упрощенная модель из многоугольников, располагающая информацией о “местах”, где можно проехать

Необходимо:

1. Загрузить плагин ...JS
2. Создадим карту
3. Генерируем навигационный мэш
4. Почистим код от лишних вызовов и старого поведения
5. Проложим путь танка по навигационному мэшу

1. Установим сторонний пакет

Установка с помощью NPM в проект:

npm i recast-detour

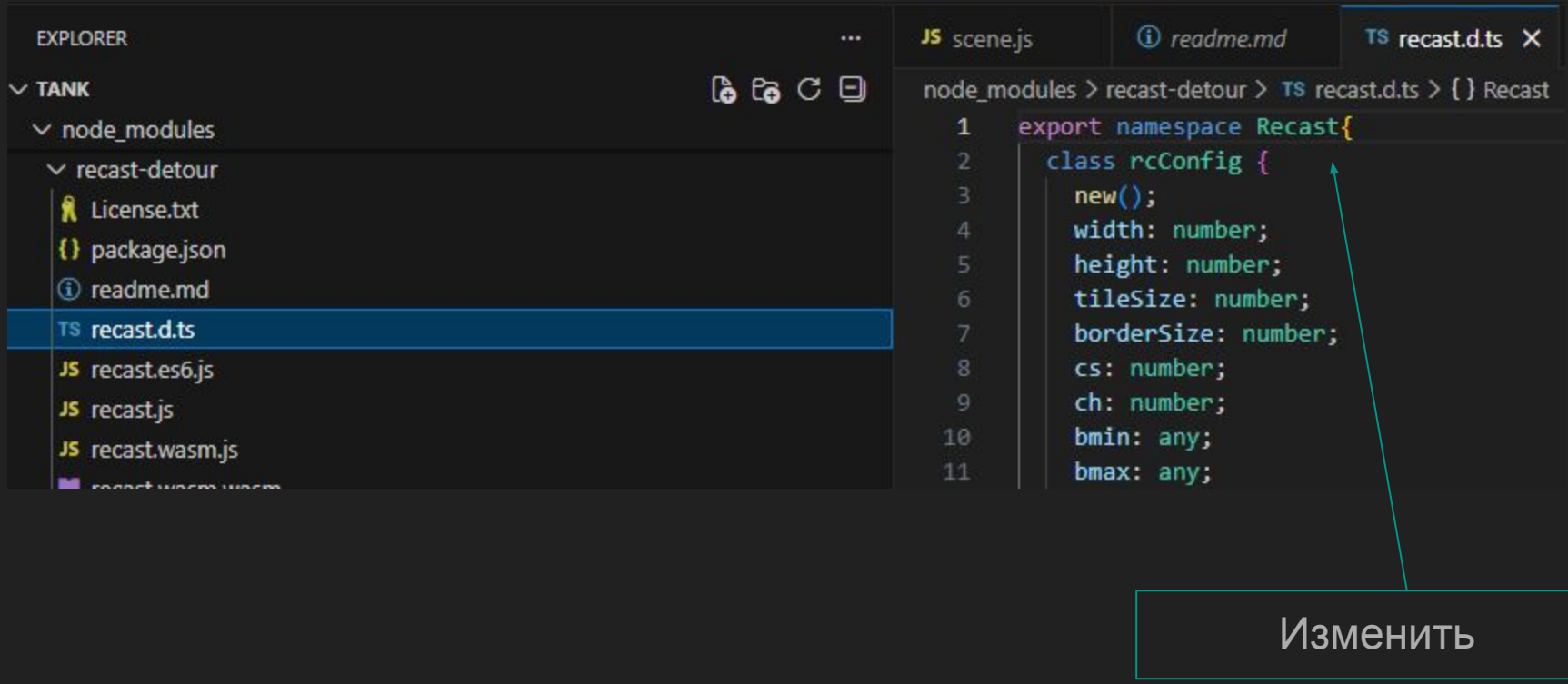
И сделать импорт в проект:

```
import Recast from "recast-detour";
const recast = await new Recast();
```

Ссылка на гитхаб:

https://github.com/recastnavigation/recastnavigation

Небольшая уловка для импорта в проект



2. Создадим “Карту”

```
let mapMeshes = await
BABYLON.SceneLoader.ImportMeshAsync(null,
'./models/','ground.glb', scene);
//отцепить мэш root так как он помешает для подготовки
навигационного мэша
let meshes = mapMeshes.meshes[0].getChildMeshes(false);
```

3. Навигационный МЭШ

```
let meshes = mapMeshes.meshes[0].getChildMeshes(false);
let navigationPlugin = new BABYLON.RecastJSPlugin(recast);
let navmeshParametres = {
    cs: 0.2,
    ch: 0.2,
    walkableSLopeAngle: 45,
    walkableHeight: 1.0,
    walkabLeCLimb: 1,
    walkabLeRadius: 2,
    maxEdgeLen: 12.,
    maxSimplificationError: 2.3,
    minRegionArea: 8,
    mergeRegionArea: 20,
    maxVertsPerPoly: 6,
    detailSampleDist: 6,
    detailSampleMaxError: 1,
}
navigationPlugin.createNavMesh(meshes, navmeshParametres);
```

Создадим видимую сетку для дебагга:

```
let navMeshDebug = navigationPlugin.createDebugNavMesh(scene);
navMeshDebug.position.y = 0.01;
let matDebug = new BABYLON.StandardMaterial();
matDebug.diffuseColor = new BABYLON.Color3(1, 0, 0);
matDebug.alpha = 0.8;
matDebug.wireframe = true;
navMeshDebug.material = matDebug;
```

5. Путь танка

```
let followPathBh = new YUKA.FollowPathBehavior();
let path = new YUKA.Path();
path.loop = false;
followPathBh.path = path;
followPathBh.active = false;

vehicle.steering.add(followPathBh);
vehicle.maxSpeed = 3;
```

В utils.js добавим две функции конвертации

```
export function y2b(v) {
    return new BABYLON.Vector3(v.x, v.y, v.z)
}

export function b2y(v) {
    return new YUKA.Vector3(v.x, v.y, v.z)
}
```

а в scene.js импорт

```
import {y2b,b2y} from './utils.js';
```

5. Путь танка

```
    scene.onPointerPick = (evt, info) => {
        if (evt.button == 0) {

        }
        //Правая кнопка - послать его
        if (evt.button == 2) {
            //смотрим ближайшую точку клика
            let closestPoint = navigationPlugin.getClosestPoint(info.pickedPoint);
            let pathPoints = navigationPlugin.computePath(y2b(vehicle.position), closestPoint);
            if(pathPoints.length > 0) {
                path.clear();
                pathPoints.forEach((p)=>{
                    path.add(b2y(p));
                })
                followPathBh.active=true;
            }
        }
    }
```