Лекция 5

- **Агент `Agent` - это моб, который ищет путь**
- **Сетка `Grid` - это способ представления мира**

Path Finding (Навигация)

Алгориты поиска

- `BFS` - поиск в ширину
- `DFS` - поиск в глубину
- `Dijkstra` - поиск кратчайшего пути в графе
- `Heuristic` - если знаем направление в каждой точке до цели. Какая точка будет ближе к цели
- `A*`
    - начинает на `Dijkstra`
    - добавляем дистанцию до точки к `Heuristic` в каждом проходе
- `JumpPoint Search` - идея в симмитричных путях
- `HPA` - максимально красивый, похожий на человеческий путь, для более детальных путей. Например, если посмотреть на карту России, то можно сказать, что из А в Б можно попасть таким-то путем, но он будет отличаться более детального пути, когда нам нужно знать как мы будет двигаться в каждом городе
    - Делим большую сетку для кластера
    - Для каждого кластера есть входы и выходы
    - Берем два кластера. Добавляем входы. К входу находим путь до выходов, с помощью поиска кратчайшего пути. Сохраняем входы и выходы
    - Далее по каждому кластеру ищем путь
    - Иерархический путь будет неидеальным, с погрешностью
- `HPAA` - для агентов больше одной клетки
    - Добавляем для каждой точки добавляем атрибут `clearence`, который говорит размер агента, который может поместиться в эту точку

NavMesh

- `NavMesh` дает более простой граф
- Как строить? (Примитив)
    - Берут части меша
    - Делят на выпуклые многоугольники
    - Строят из них граф
- Строится до начала игры
- Движение по `NavMesh`
    - По центрам полигонов
    - По вершинам (по умолчанию в Unity)
    - По середине ребра (чаще всего используют)

Сглаживание пути

- делается в runtime
- для `grid` - смотрим по каждому ребру и соединяем точки для оптимизации прямой
- для `navmesh` - для каждого движения свой `navmesh`

Добавления

- Писать свой `HPA` путь просто
- Если игра не киберспортивная, то можно воспользоваться средствами `NavMesh`

Unity

- Запекаем `NavMesh`
- Добавляем препятствиям `NavMeshObstacle`
- Добавляем компоненту `NavMeshAgent` для `unit`
- Пишем скрипт на котором обозначим куда идти `unit`: `_agent.destination`

AI - искуственный интелект

- `Decision Tree` - дерево принятия решения из серии вложенных `if`
- Конечный автомат (`State Machine`) - в один момент времени активно одно состояние и `state machine` переключается между состояниями.
    - Можно представить в виде графа, где вершины - действия, а ребра - условия перехода
    - Сильные стороны
        - Наглядность
    - Слабые стороны
        - Сложность масштабирования
            - Одно действие начинает занимать много состояний
- Деревья поведения (`Behaviour Tree`)
    - Узлы - поведения или задачи
    - Каждая задача находится в одном из состояний
        - Success
        - Fail
        - Running
    - Каждая задача отправляет своё состояние об изменении родителю
    - Движение с корня (сверху вниз) и слева направо
    - Типы узлов
        - Композитные - могут содержать несколько дочерних узлов.
            - Sequence - возвращает `fail`, если хотя бы один из дочерних узлов вернет `fail`. Например, действие пойти на работу требует проснуться, покушать, одеться: если не проснулся, то не идем на работу
            - Selector - возвращает `success`, если хотя бы один из дочерних `success`
        - Декоратор - может содержать только один узел
            - Например, инвертор - добавляем отрицание к действию при возвращении
        - Лист (т.к. дерево) - самый низкоуровневый узел, который содержит в себе действие, например, пойти открыть окно
    - Для большой логики лучше использовать именно этот тип
    - Легкая отладка
    - `Asset Store`: `Behaviour Designer` или `Behaviour machine`

- GOAP - целеориентированное планирование целей
  - во главе принятий решений стоит цель
  - ключевые элементы - действия
  - у каждого действия есть стоимость (сами задаем)
  - среди всех последовательностей достижения цели выбирается меньшая по стоимости
  - для действия важны условия выполнения и эффекты - что делает действие
  - планировщик принимает на вход условия выполнения и конечная цель