PATHFINDING, NAVMESHES Y WAYPOINTS

Dado a que estas 3 tareas están muy relacionadas entre sí, hemos decidido realizar un único entregable para mostrar su funcionamiento.

En el ejecutable, encontraremos un Agente que se encargará de navegar el Navmesh. Para decirle a dónde debe ir, simplemente tenemos que hacer clic derecho en alguna parte del Navmesh.

Controles del ejecutable:

Q ——— Cerrar

Movimiento de la cámara:

W ——— Adelante
Izquierda —— **A** **S** **D** ——— Derecha
|
Atrás

LEFT CTRL ——— Abajo
ESPACIO ——— Arriba
I ——— Invertir cámara en el eje Y

Debug visual:

F1 ——— Físicas: Muestra/Oculta las entidades de Bullet
N ——— Nodos: Muestra/Oculta los nodos del Navmesh
P ——— Path: Muestra/Oculta el camino
V ——— Vectores: Muestra/Oculta los vectores de fuerzas
C ——— Conexiones: Muestra/Oculta las conexiones entre nodos

Pathfinding

El algoritmo usado es A*, su implementación principal se encuentra en el fichero src/GameAI/Pathfinding.cpp — void A_Estrella(...).

En cuanto a la implementación, no hay mucho que destacar, una de las optimizaciones realizadas está en la función que calcula la heurística. En un principio era la distancia euclídea, pero quitamos el cálculo de la raíz cuadrada ya que si:

sqrt(x) > sqrt(y) entonces x > y

Con lo que, como sólo necesitamos hacer una comparación, podemos ahorrarnos un par de ciclos. Es lo que llamamos gg::FastDIST(...).

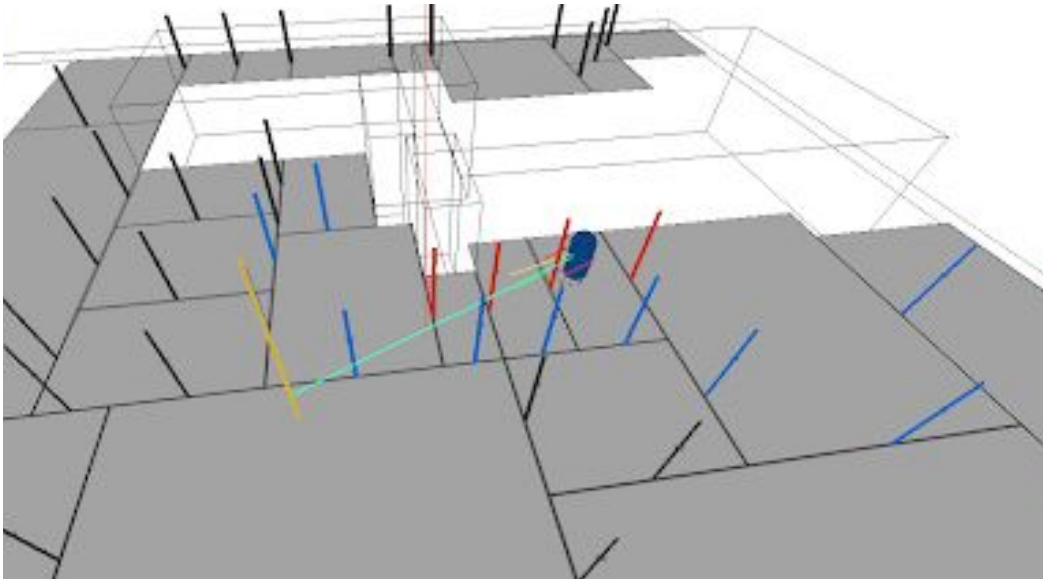
Si tenemos activados los Nodos, al hacer clic derecho, podemos ver como cambian de colores las líneas:

Las líneas **negras**, son nodos no visitados, por lo que no ha habido procesamiento.

Las líneas **azules**, son los nodos en la lista abierta.

Y las líneas **rojas**, son los nodos que han sido visitados.

La línea **dorada** es el destino. Y la **azul clarito**, marca el camino.

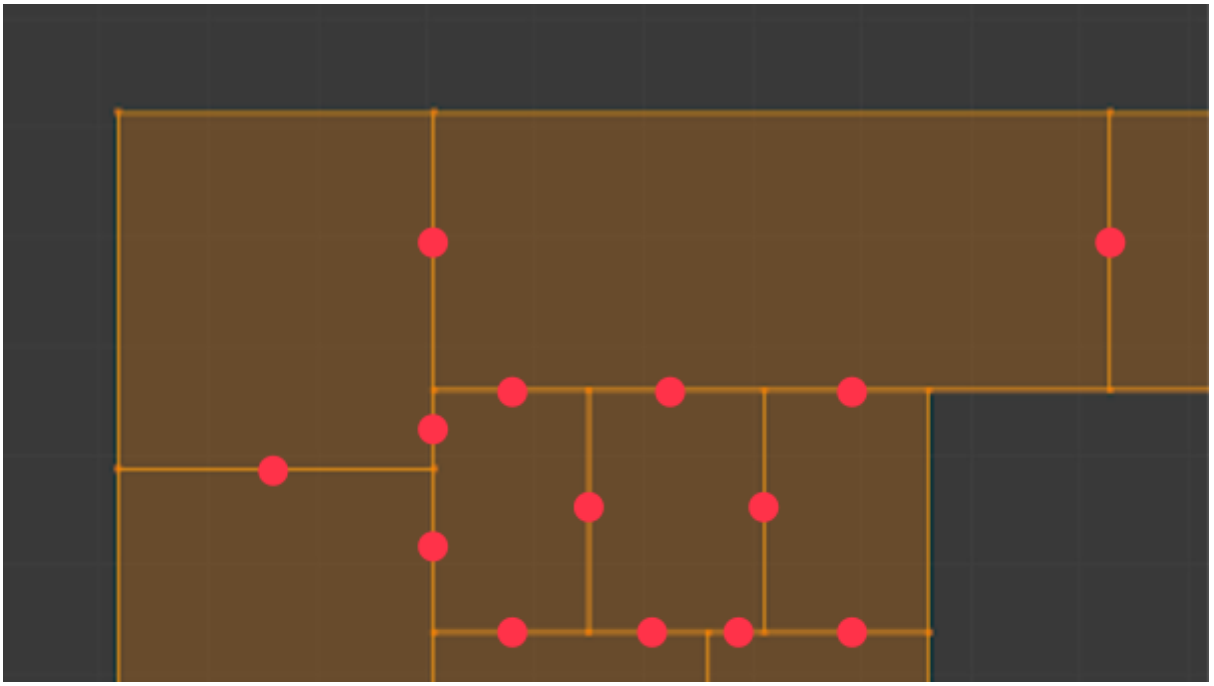


Navmeshes

En cuanto a las Navmeshes, decidimos hacerlas a mano en Blender.

Para acelerar cálculos los polígonos serán rectángulos, ya que para comprobar si un punto está dentro de un rectángulo, basta con 4 ifs.

En cuanto al lugar donde colocar los nodos, después de probar a colocarlos en el centro y en los vértices. Al final se usamos los puntos medios de las aristas adyacentes.



Una vez generado el .obj, con la ayuda de assimp, procesamos los datos (fuera del tiempo de ejecución), para exportarlos a un fichero binario, que leeremos dentro del juego después.

Aquí guardamos 3 datos importantes: los **nodos**, las **conexiones** entre nodos, y las **caras**. De este último dato, guardaremos las esquinas superior izquierda e inferior derecha, y a que Nodos tiene acceso, lo usaremos para saber en qué polígono está el origen y el destino.

Waypoints

Una vez tenemos todo lo necesario para realizar el algoritmo nos queda saber el camino que nuestro Agente debe seguir.

Lo primero que se hace, es iterar sobre todas las caras hasta encontrar en qué cara está el Agente, y el destino al cual queremos ir.

Una posible optimización aquí que tuvimos en cuenta, fue guardar la última cara en la que estaba el agente, y buscar en las caras adyacentes a partir de ahí, ya que es más probable que se encontrara en una de ellas.

Pero puesto que luego mejoraremos el camino, es posible que nuestro agente se encuentre en una cara que no formaba parte del camino original. Con lo que estaríamos accediendo al vector de caras de manera aleatoria, por lo que pensamos que sería más rápido recorrerlo secuencialmente desde el principio. Además, no tenemos pensado que el número de caras sea tan elevado como para que suponga un problema.

Una vez encontradas las caras de origen y destino, primero comprobamos si el agente y el destino están en la misma cara, si es así, simplemente vamos en línea recta.

Si no lo está, calculamos que nodo (que está dentro de esa cara) es el más cercano, tanto para el origen como para el destino.

Ahora que tenemos ambos nodos, hacemos hacemos A*, lo que nos devuelve un camino.

Una vez tenemos el camino, nuestro Agente ya puede empezar a trabajar.

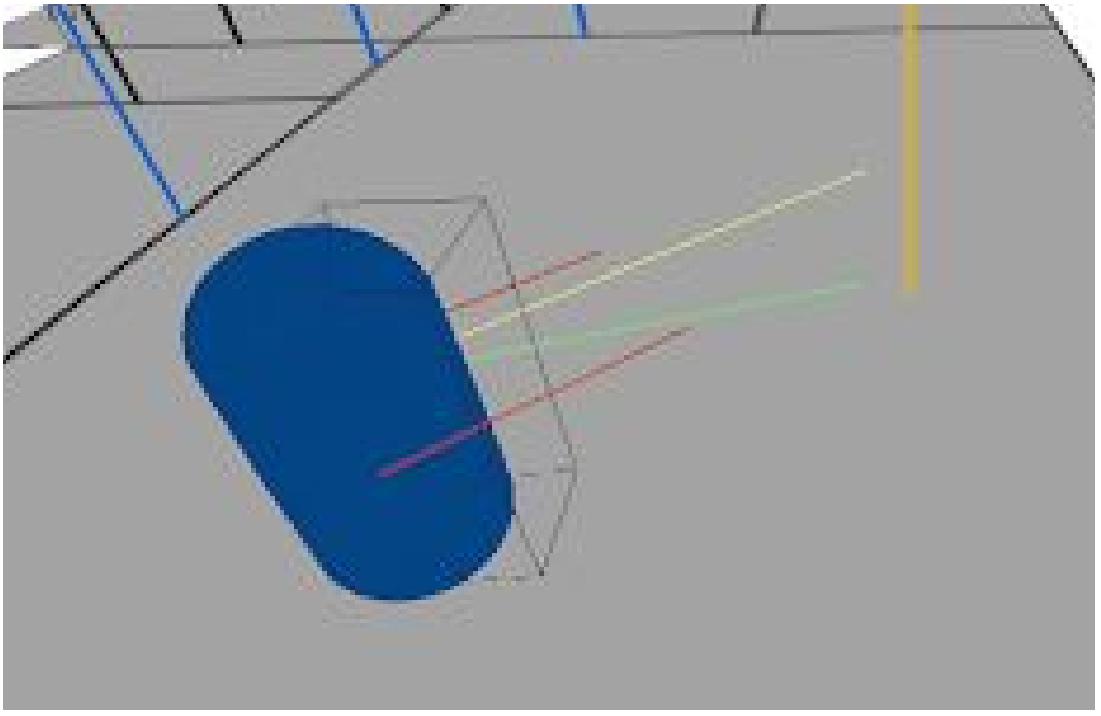
Movimiento

Todo el código del movimiento lo podemos encontrar en el fichero src/ComponentArch/Components.cpp — void FixedUpdate( ).

Para movernos, simplemente aplicamos una fuerza al Agente en la dirección del siguiente Waypoint. Cuando se produce un giro, aplicamos una fuerza de frenado dependiendo del ángulo que formen el vector de velocidad y vector de dirección, para evitar que derrape en las curvas.

Si activamos los vectores, podemos ver que aparecen unas cuantas líneas:

Verde: Vector desde el Agente al destino
Amarilla: Vector velocidad
Rojas: Sensores para el Reactive Path following, que aplicarán una fuerza a izquierda o derecha.



Reactive Path following

Para ver si podemos saltarnos algunos nodos del camino, realizamos 2 comprobaciones cada 0.15 segundos (0.15 ya que no es necesario que se haga constantemente).

Comprobamos primero, si el siguiente nodo se encuentra dentro de la vista de alcance del Agente, si es así, realizamos un Raycast desde la posición del agente, al siguiente nodo.

Si no hay colisión, podemos ir en línea recta, y repetimos para el siguiente nodo. Hasta que alguna condición falle, o nos quedemos sin nodos.

Por último, si uno de los sensores (también formados por Raycasts, aunque de menor alcance), detecta un obstáculo, se aplica una fuerza del lado del obstáculo. Si tenemos activados los vectores, podemos ver un par de líneas **moradas**, parpadeando cuando esto suceda, indicando la dirección de la fuerza.

Posibles optimizaciones futuras

Por ahora el algoritmo funciona bien, pero si en el futuro experimentamos bajones, y necesitamos ahorrar un par de ciclos de alguna parte, podemos cambiar el algoritmo por una Lookup Table, eliminando así todo procesamiento, aunque aumentando el tamaño en memoria.