Generación de caminos A*

El algoritmo A* ha sido implementado siguiendo el algoritmo proporcionado en las transparencias. Para poder calcular un camino los agentes deben solicitárselo a un agente especial, el Path Finder.

Path Finder

Dicho agente se encarga de evitar que pueda calcularse más de un camino de tipo A* a la vez. Para conseguirlo utiliza el sistema de mensajería para agentes que se ha implementado y una aproximación de inteligencia artificial de agentes de tipo body y mind. Al ser una entidad abstracta, no tiene representación como tal en el mundo de juego, su body no hace nada. Sin embargo su función de mind es la que se encarga de evitar que más de un camino pueda suceder a la vez y de calcular el camino sin bloquear la ejecución del juego.

```
if(actual_state_ == PFAgentState::k_Waiting)
{
  for (uint32_t i = 0; i < num_agents_; i++)
  {
    if((mail_box_ + i)->type == AgentMessageType::k_AskForPath)
    {
      requestor_ = i;
      actual_state_ = PFAgentState::k_Calculating;
      origin_ = (mail_box_ + i)->position;
      dst_ = (mail_box_ + i)->dst;
      path_ = (mail_box_ + i)->path;
      (mail_box_ + i)->type = AgentMessageType::k_Nothing;
      (mail_box_ + i)->position = Float2(0.0f, 0.0f);
      break;
    }
  }
}
if(actual_state_ == PFAgentState::k_Calculating)
{
  const s32 status = generatePath(path_, origin_, dst_, dt);
  if(status == kErrorCode_Ok)
  {
    AgentMessage msg;
    msg.type = AgentMessageType::k_PathIsReady;
    msg.position = Float2(0.0f, 0.0f);
    msg.path = nullptr;
    GameState::instance().agents_[requestor_ - 1]->sendMessage(msg, id_);
    actual_state_ = PFAgentState::k_Waiting;
  }else if(status != kErrorCode_Timeout)
  {
    AgentMessage msg;
    msg.type = AgentMessageType::k_PathNotFound;
    msg.position = Float2(0.0f, 0.0f);
    msg.path = nullptr;
    GameState::instance().agents_[requestor_ - 1]->sendMessage(msg, id_);
    actual_state_ = PFAgentState::k_Waiting;
  }
}
```

Interior de la función updateMind del PathFinder

Como podemos observar, mientras el PathFinder no esté ocupado revisará si le ha llegado alguna petición de cálculo de caminos por parte de algún agente. Si por casualidad recibiese un cálculo de camino cambiaría su estado a k_Calculating, limpiará el buzón del agente que le ha pedido el camino y comenzará a calcular el camino. Nótese que las comprobaciones de estado no son excluyentes, primero comprueba si está libre y después si está calculando en un mismo ciclo del bucle de simulación. Esto se ha realizado intencionadamente para que el cálculo del camino comience a calcularse lo antes posible.

En caso de encontrarse en un estado de cálculo de camino irá lanzando un generatePath en cada ciclo de iteración hasta que el camino haya sido encontrado (en cuyo caso la función generatePath devolverá kErrorCode_Ok) o hasta que haya finalizado sin un error de timeout, lo cual querrá decir que el cálculo ha acabado pero o no había camino u ocurrió algún error durante la búsqueda. Dependiendo del estado recibido el Path finder mandará un mensaje u otro devuelta al agente que lo había solicitado y cambiará su estado a esperando de nuevo. Pudiendo de esta forma encargarse de cualquier camino pendiente en el buzón o de uno nuevo que le llegue.

El Path finder puede calcular el camino de dos forma.

- Por partes: Es la comentada hace un momento. Irá calculando el camino en su mente en cada iteración del bucle de simulación hasta que lo encuentre o la función acabe por otro motivo que no sea timeout, momento en el cual notificará al agente solicitante.

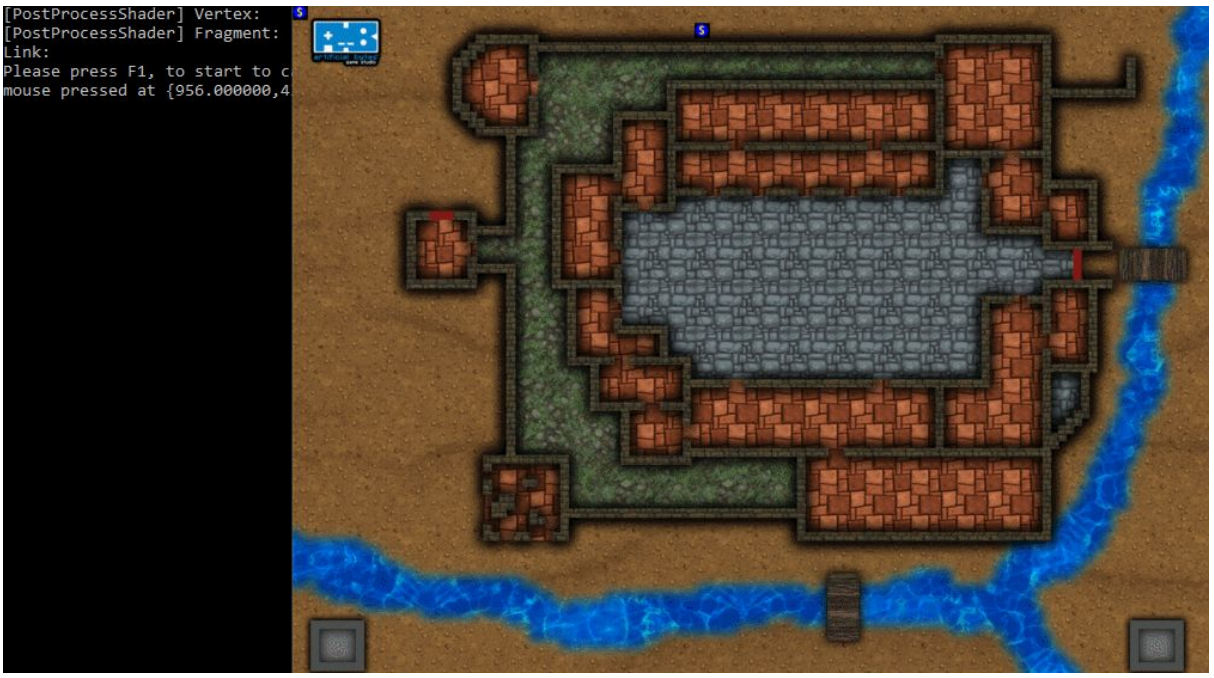
- De una sola vez: Forma inicial en la que se calculaba el camino, al solicitarse un camino este se calcula hasta obtenerlo sin interrumpir la ejecución de la función.

Pese a que el Path Finder es un agente este no comparte herencia de clases con los agentes FSM. Se ha realizado de esta forma para realizar el caso de una forma sencilla. Sin embargo, esto crea la necesidad de indicarle al Path Finder cuántos agentes existen en el mundo de juego, para que así pueda iniciar los buzones en los cuales dichos agentes dejarán sus solicitudes al Path Finder. Esto podría haberse solucionado con herencia. De forma que existiera una clase abstracta de agente, con las funcionalidades de update virtuales para que la puedan implementar sus descendientes, que serían el path finder y la clase agente actual. De esta forma el Path finder sería contabilizado con el sistema de ids único que implementan actualmente los agentes.

```
void Agent::init(const float x, const float y)
{
  position_ = Float2(x, y);
  velocity_ = Float2(0, 0);
  target_position_ = position_;
  id_ = total_agents;
  total_agents++;
  initialized_ = false;
  representation_ = nullptr;
  path_ = new Path();
}
```

Función de inicialización actual que usan los agentes

Dicha función podría incluso mejorarse para seguir un patrón de factorías a la hora de crear a los agentes. Lo que es un hecho es que de esta forma el Path Finder sabría cuántos agentes hay y podría independizarse a los agentes completamente del main.



Demostración de caso con 2 agentes solicitando un camino a la vez al Path Finder

MapData

La clase A* se apoya con una clase de tipo mapa. Dicho mapa dispone de una matriz de colisión (lineal en la cual nos indexamos mediante la fórmula x + (y*ancho)) que se crea en base a una imagen especial de colisiones.

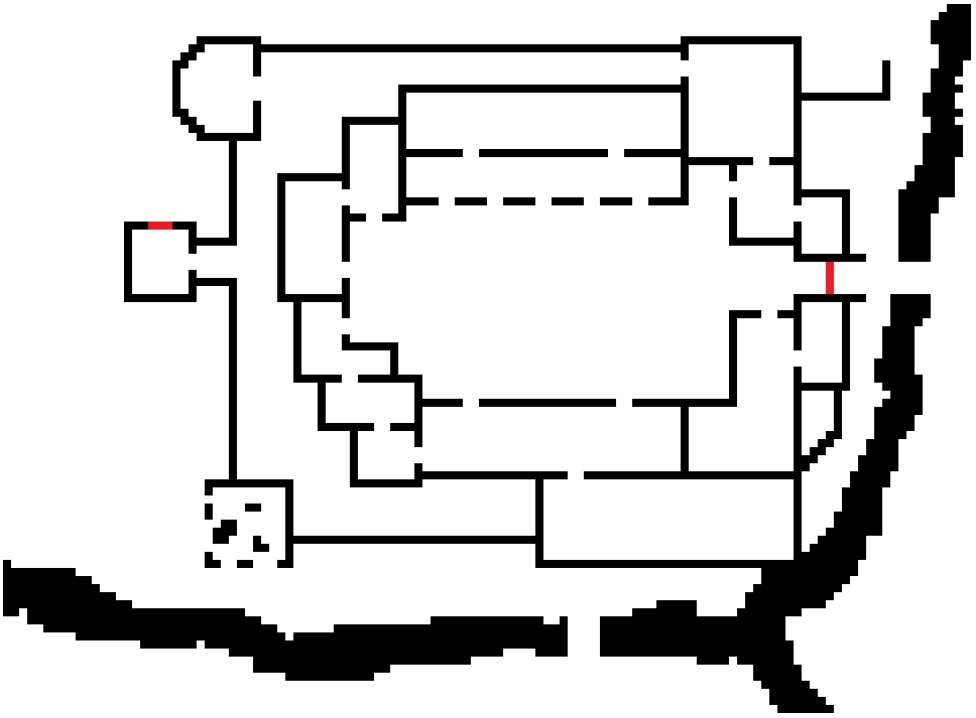


Imagen con información de colisiones

La matriz se crea en base a la altura y anchura de dicha imagen, el mapa también se encarga de guardar el ratio con la imagen original de mapa de la que dispone. De esta forma se da soporte a mapas de diferentes dimensiones para simplificar el coste del A*.

```
s16 Map::loadMap(const char* src, const char* background)
{
  if (!src || !background) return kErrorCode_InvalidPointer;
  //If there's already data loaded we free it
  if (collision_data_) freeResources();

  s32 bpp;

  unsigned char* background_image = stbi_load(background, &original_width_, &original_height_, &bpp, 1);

  if (!background_image) return kErrorCode_Memory;

  unsigned char* image_data = stbi_load(src, &width_, &height_, &bpp, 1);

  if (!image_data) {
    stbi_image_free(background_image);
    return kErrorCode_Memory;
  }

  const int number_of_elements = height_*width_;
  collision_data_ = static_cast<bool*>(malloc(sizeof(bool)*number_of_elements));

  if (!collision_data_) {
    stbi_image_free(background_image);
    stbi_image_free(image_data);
    return kErrorCode_Memory;
  }

  for (int i = 0; i < width_*height_; i++)
  {
    const bool result = image_data[i] == 0xff;
    collision_data_[i] = result;
  }
  ratio_ = Float2(static_cast<float>(original_width_) / width_, static_cast<float>(original_height_) / height_);

  background_ = ESAT::SpriteFromFile(background);

  stbi_image_free(background_image);
  stbi_image_free(image_data);

  return kErrorCode_Ok;
}
```

Función LoadMap de la clase Map

Como se puede observar en base al color del pixel de la imagen rellenamos nuestra información de colisiones. De esta forma utilizando las funciones isValidPosition e isOccupied el algoritmo A* podrá comprobar si el agente puede pasar por un punto. La opción de permitir cargar mapas de diferentes niveles de complejidad permite una ejecución del A* mucho más rápida, pues estamos disminuyendo el dominio total de nodos que podrán generarse, ya que no podrá haber más de una cantidad de widthXheight del mapa de costes de nodos diferentes. Sin embargo, esto puede conllevar algunos problemas. Pues un mapa con unas dimensiones menores puede reconocer como posiciones ocupadas algunos bordes del mapa. Sin embargo, salvo que nuestro agente tenga que implementar un comportamiento especial, que se pegue a las paredes para algo muy específico el jugador no llegaría a notar algo tan sustancial. Sin embargo, es importante tenerlo en cuenta pues dependiendo del tipo de juego podríamos encontrarnos en una situación en la que debamos usar un mapa de costes mayor que otro.

A* por partes

Para poder realizar el algoritmo A* sin interrumpir el bucle principal se comprueba si se ha excedido el tiempo cada vez que se va a expandir un nuevo nodo current. De esta forma si el tiempo disponible ya ha expirado la función devolverá un código de error por timeout, el cual como se ha comentado anteriormente el Path Finder interpreta como que tiene continuar con el algoritmo en su próxima interacción de mente.

Comportamiento inteligente

Los agentes implementados para el juego de tipo dungeon mantienen un body puramente físico. Este avanza hacia el objetivo marcado por la mente en target_position_. El cuerpo informará a esta de que ha llegado ha dicho objetivo mediante la variable target_reached_. Se ha aprovechado que la mente no necesita actualizarse en cada frame para actualizarla con un tiempo de refresco diferente a la del cuerpo, en específico se refresca cada segundo.

Por tanto lo primero que realiza la mente es mirar si ha pasado el tiempo suficiente para que le toque actualizarse. Si es así y el agente aún no ha sido inicializado lo inicializa. Aquí se fijarán los parámetros específicos de cada tipo de agente, veamos un ejemplo:

```
case AgentType::k_Normal:
  move_type_ = MovementType::k_MovRandom;
  next_random_time_ = 5000; //5s
  accum_time_random_ = 0;

  tracking_retarget_time_ = 1500; //1.5s

  speed_ *= 1.0f;
  epsilon_ = kEpsilonFactor * speed_;
  representation_ = ESAT::SpriteFromFile("../../../data/gfx/agents/normal_agent.png");
  break;
```

Inicialización del agente Normal

En el proyecto final del curso se ha implementado el sistema de comportamiento inteligente mediante cuerpo y mente al enemigo torreta. Debido a que la torreta era un enemigo bastante diferente.

En caso de que ya esté actualizado, lo primero que hace la mente del agente es revisar su buzón y comprobar si le ha llegado algún mensaje. En nuestro caso el agente mirará si tiene algún mensaje de que su camino A* está listo. Por último realizará su acción FSM en base al estado en el que se encuentre:

```
for (u32 i = 0; i < total_agents; i++)
{
  if ((mail_box_ + i)->type == AgentMessageType::k_PathIsReady)
  {
    move_type_ = MovementType::k_MovAStar;
    target_reached_ = true;
    (mail_box_ + i)->type = AgentMessageType::k_Nothing;
    break;
  }
  (mail_box_ + i)->position = Float2(0.0f, 0.0f);
}

switch (actual_state_) {
case FSMStates::k_Working:
  FSM_Working(dt);
  break;
case FSMStates::k_Chasing:
  FSM_Chasing(dt);
  break;
case FSMStates::k_Fleeing:
  FSM_Fleeing(dt);
  break;
case FSMStates::k_Resting:
  FSM_Resting(dt);
  break;
default:
  break;
}
```

Revisión del buzón y actualización de la máquina de estados

Estas funcionalidades tienen implementado un sistema de debugeo que sólo se activa cuando ejecutamos en modo debug y actualizan al agente en base a las percepciones de la mente en dicho momento. Por ejemplo la FSM de resting, actualizará el tiempo que lleva descansando y si este excede el tiempo de descanso volverá a activar el movimiento básico en base al tipo del agente y cambiará al estado working. Si por otra parte mientras descansa avista a otro agente más grande que él entrara en el estado fleeing.

```
void Agent::FSM_Resting(const u32 dt) {
  time_rested_ += dt;
  if(time_rested_ >= resting_time_)
  {
#ifdef DEBUG
    printf("Time to work a bit. \n");
#endif
    time_rested_ = 0;
    actual_state_ = FSMStates::k_Working;
    switch(type_agent_)
    {
    case AgentType::k_Small:
      move_type_ = MovementType::k_MovPattern;
      break;
    case AgentType::k_Normal:
      move_type_ = MovementType::k_MovRandom;
      break;
    case AgentType::k_Huge:
      move_type_ = MovementType::k_MovDeterminist;
      break;
    default:
      move_type_ = MovementType::k_MovAStar;
      break;
    }
  }

  for (Agent* agent : GameState::instance().agents_)
  {
    Float2 distance = agent->position_ - this->position_;
    const float d = distance.Length();
    //We check if the agent is in flee range and is smaller
    if (agent != this && isSmaller(agent) && d <= flee_distance_)
    {
      actual_state_ = FSMStates::k_Fleeing;
      objective_ = agent;
      move_type_ = MovementType::k_MovTracking;
      return;
    }
  }
}
```

FSM Resting

Torreta en Unreal utilizando una FSM

En el proyecto final del curso se ha implementado el sistema de comportamiento inteligente mediante cuerpo y mente al enemigo torreta. Debido a que la torreta era un enemigo bastante diferente al resto, al no ser humanoide, se decidió programarlo independientemente de los demás y animarla completamente por programación. Para realizar su IA no se ha usado ninguna clase de Unreal por lo que está realizada completamente en c++. Como se ha mencionado anteriormente se ha seguido un

procedimiento de FSM junto al paradigma de update mediante cuerpo y mente visto con Toni en clase de Inteligencia Artificial.

Debido a que el cuerpo es una componente física debe actualizarse en cada update de nuestro bucle de simulación, lo cual en palabras de Unreal se traduce como el Tick(). Sin embargo debido a que nuestra mente no necesita una actualización constante junto al cuerpo ésta sólo la actualizamos cada cierto tiempo (indicado por una variable TurretMindSpeed y completamente configurable por el diseñador). De esta forma la torreta en cada tick de cuerpo actualizará su animación y en caso de que la mente localice al enemigo cambiará su estado para que el cuerpo reaccione de forma acorde.

```
void AP_EnemyTurret::Tick(float DeltaTime)
{
  Super::Tick(DeltaTime);
  IsEnemyAtRange();
  MindUpdateAccum += DeltaTime;

  //Turret mind
  if(MindUpdateAccum >= TurretMindSpeed)
  {
    if (CurrentStatus == ETurretStatus::HTE_Idle && IsEnemyAtRange()) {
      //Player spotted
      OnTurretSightEffect();
      VisionLight->SetLightColor(SpottedLightColor);
      CurrentStatus = ETurretStatus::HTE_ChangingAttackMode;
    }else if(CurrentStatus == ETurretStatus::HTE_Idle && ObjectiveReached){
      AP_Waypoint* nextObjective = GetNextPatrolPoint();
      if (nextObjective) NextPatrolPosition = nextObjective->GetActorLocation();
      ObjectiveReached = false;
    }/*else if(CurrentStatus != ETurretStatus::HTE_Idle && !IsEnemyAtRange())
    {
      OnTurretUnsightEffect();
      CurrentStatus = ETurretStatus::HTE_ChangingIdleMode;
    }*/
    MindUpdateAccum = 0;
    //We update the position where we know our objective is
    if (Objective != nullptr) ObjectiveCurrentPosition = Objective->GetActorLocation();
  }

  //Turret Body
  switch(CurrentStatus)
  {
  case ETurretStatus::HTE_Idle:
    DoIdleAnim();
    break;
  case ETurretStatus::HTE_Attacking:
    Attack(DeltaTime);
    break;
  case ETurretStatus::HTE_Reloading:
    ReloadTurret(DeltaTime);
    break;
  case ETurretStatus::HTE_ChangingAttackMode:
    ChangeToAttackModeAnim();
    break;
  case ETurretStatus::HTE_ChangingIdleMode:
    ChangeToIdleModeAnim();
    break;
  }
}
```

“Update” de Unreal con la parte de body y mind

Al separar cuerpo y mente evitamos que algunos de los cálculos, bastante caros, como el DotProduct que se realiza para saber si el jugador está en nuestro rango de visión y luego lanzar un ray tracing para saber si lo vemos no se realicen en cada frame. Demostrando de

esta forma como hemos visto en clase de inteligencia artificial que no es siempre necesario complicarse con el diseño de una IA utilizando behaviour trees. Puesto que si esta es lo suficientemente sencilla una FSM es más que suficiente.

Torreta en Unreal utilizando una FSM

Se ha realizado la gestión del proyecto mediante el uso de un tablero en Trello. A continuación se puede encontrar el enlace a dicho tablero para cualquier consulta.

Enlace al tablero Trello