Programmation Objet Avancée - Rapport

Binôme : PENUCHOT Jules | BIARD David

Requirements

- CMake 2.8 ou plus récent
- Clang 5.0 ou plus récent

Attention : Le projet repose sur des fonctionnalités présentes à partir de C++ 17, veillez à compiler le programme avec un compilateur qui les supporte.

Tour des features

Ce qui marche

- Chargement des niveaux
- Affiches
- Gestion de la vie
- Détection de l'entité précise sur laquelle on tire
- Suppression d'entités quelconques
- Pathfinding optimal (A* avec heuristique euclidienne)
- Collisions
- Distance au trésor (pour ajustement de la difficulté)
- Fin du jeu (trésor atteint -> appel d' `exit(0)`)

Ce qui ne marche pas

- Déplacements buggés (collisions avec les coins, sûrement à cause de l'angle de déplacement)
- Niveaux de difficultés
- Tirs automatiques sur le chasseur
- Réseau

Précisions sur le code

Structure du projet, compilation...

La compilation du projet est prise en charge par le script `build.sh` grâce à CMake. Le script `CMakeLists.txt` effectue une recherche récurcive dans `src/` pour récupérer l'intégralité des .cpp et faire de la compilation modulaire. Les objets externes sont spécifiés et linkés dans le `CMakeLists.txt` également.

CMake est un générateur de makefile, il permet à partir de code haut niveau de générer des makefiles performants et de gérer les librairies et sous-projets plus facilement.

Les scripts (bash/cmake) sont faits maison (voir https://github.com/JPenuchot/cpp-template).

Utilisation de C++ moderne

Nous avons utilisé intensivement la STL dans tout le programme : lambdas, fonctions templates, `std::regex` . L'accent a donc été mis sur la qualité et la lisibilité du code.

L'algorithme de pathfinding illustre cet exemple : l'algorithme repose sur la structure `std::priority_queue` qui implémente un tas paramétrable par le type contenu et la fonction de comparaison. La fonction de comparaison, elle, compare de

manière dynamique non pas le coût d'un chemin (comme dans Dijkstra) mais directement la somme du coût du chemin et de l'heuristique (minimisante).

Dans mapgen également, les lambdas sont utilisées de manière intensive pour nommer des actions répétées à travers l'algorithme au début de la fonction, pour avoir un "coeur algorithmique" plus clair et concis par la suite.

Les lambdas permettent donc de décrire des actions locales de manière plus légère et contextuelle que des fonctions externes, permettant de rendre les algorithmes plus clairs, d'éviter des bugs (en factorisant) et optimisés (les lambdas sont très bien optimisées par le compilateur, le plus souvent inlinées).

```
bool Labyrinthe::findPath(pos_int from, pos_int to, queue<pos_int>& res)
{
  /**
   * Implémentation de l'algorithme A*
   */

  //  Retourne les 4 voisins d'une position
  auto neighborsOf = [](pos_int& p)
  {
    return array<pos_int, 4> (
      { make_pair(p.first + 1, p.second)
      , make_pair(p.first - 1, p.second)
      , make_pair(p.first, p.second + 1)
      , make_pair(p.first, p.second - 1)
      });
  };

  //  Heuristique
  auto h = [](pos_int& a, pos_int& b)
  {
    float dx = b.first - a.first, dy = b.second - a.second;
    return (dx * dx) + (dy * dy);
  };

  //  Stockage des coûts
  map<pos_int, int> costMap;

  //  Comparateur (coût + heuristique d'une arête)
  auto cmp = [&](pos_int& a, pos_int& b)
  {
    return (costMap[a] + h(a, from)) > (costMap[b] + h(b, from));
  };

  //  Queue
  priority_queue<pos_int, vector<pos_int>, decltype(cmp)> q(cmp);

  //  On commence de la destination
  //  (le résultat sera rempli depuis la destination)
  q.push(to);
  costMap[to] = 0;

  //  L'algo
  while((!q.empty()) && costMap.find(from) == costMap.end())
  {
    //  On défile la position (avec priority_queue et le comparateur maison,
    //  on obtient donc celle qui a la somme coût + heuristique la plus faible)
    auto curr_pos  = q.top();
    auto curr_cost = costMap[curr_pos];
    q.pop();

    //  On définit l'ensemble des voisins
    auto neighbors = neighborsOf(curr_pos);

    for(auto& n : neighbors)
    {
      //  On saute les murs et les éléments déjà vus
      if(   !isValid(n) || (!this->walkable(n))
        ||  costMap.find(n) != costMap.end() )
        continue;

      //  On définit le coût du sommet (de la case)
      costMap[n] = curr_cost + 1;
```

```
      //  On ajoute à la priority_queue qui se charge de calculer la somme
      //  coût + heuristique grâce au comparateur cmp
      q.push(n);
    }
  }

  //  Si on n'a aucun chemin (pas trouvé from depuis to), on renvoie false
  if(costMap.find(from) == costMap.end())
    return false;

  //  Backtrack : on part de from et on remonte jusqu'à "to"

  //  On part de la destination et on remonte jusqu'à l'origine
  //  (rappel : elles ont été inversées pour que le chemin soit
  //            enfilé dans le bon ordre)

  auto next = from;

  //  Tant qu'on ne tombe pas sur l'origine...
  while(next != to)
  {
    auto curr_cost = costMap[next];
    auto neighbors = neighborsOf(next);

    //  Pour chaque voisin (sauf si non exploré) on prend celui
    //  qui se rapproche le plus de l'origine
    for(auto& n : neighbors)
      if( costMap.find(n) != costMap.end() && costMap[n] <= costMap[next] )
        next = n;

    //  On ajoute la position en cours au résultat
    res.push(next);
  }

  //  On a trouvé le chemin, on renvoie true
  return true;
}
```

Notre algorithme A*

Types & Abstractions

De nombreuses abstractions ont été faites sur la structure fournie, notamment les pointeurs de gardiens et chasseurs sont stockés dans des `std::vector` pour une meilleure gestion de la mémoire et surtout une distinction des types.

Deux types ont été créés pour représenter les coordonnées (`pos_int` et `pos_float` , créés à partir de `std::pair`) et les abstraire notamment dans les algorithmes de pathfinding.

Autres

Nous nous sommes rendus compte que `Environnement::data(i, j)` n'était jamais linkée... Nous nous en sommes donc passés.

La fonction partie_terminee n'étant pas présente dans les headers, elle a été remplacée par `exit(0)` .