Project Report - Third assignment - GAME ENGINES

Andrea Distler

10 December, 2012

Contents

1 Introduction

This is the project report for the third assignment of the Game Engines course.

A short description of the source code and the used libraries can be found in section 2. The files are available on GitHub.

In section 3 the outcome is described. In section 4 used mechanics and an overview of the classes can be found.

2 The Files

All files can be found on GitHub.
`https://github.com/JungleJinn/GE---SimplePlatformerEngine`.

The project was written in C++, without using additional libraries.

3 Outcome

The assignment was to implement the A* search algorithm on a grid-formed graph. There should be two "NPCs", which navigate using a finite state machine. In my project, the NPCs have each two states: a chase state and a flee state. The NPC in the chase state is represented by an upper-case letter, the NPC in the flee state is depicted as a lower-case letter. An "X" marks which grid nodes are blocked. Figure 1 shows the output of the application.

4 Classes and Mechanics

Graph and Node. The Graph and Node classes are the basic classes used to form the structure of the grid and the gridnodes. They are also tools to visualize that the grid is handled like a graph.

Link. The Link class links two nodes of an arbitrary type. It can also store an edge travel cost.

Grid. The Grid class basically manages everything. It derives from the Graph class and contains a list of all its nodes. It also contains the two NPCs. It naturally handles the creation of the nodes and their links.

GridNode. The GridNodes play an essential role in the calculation of paths. They store their temporary costs and heuristic costs. This means that there could not be two A* path-finding algorithms running at the same time.

FiniteStateMachine(FSM). In listing 1 the finite state machine is shown. It holds the current state and handles state changes by calling the enter and exit methods of the states.

Listing 1: The finite state machine.

```
template <typename T>
public class FSM
{
public:

               FSM(FSMState<T> *state)
               {
                 lastState = NULL;
                 currentState = state;
               }

  void           ChangeState(FSMState<T> *newState)
  {
    if (currentState != newState)
    {
      lastState = currentState;

      lastState->Exit();

      currentState = newState;
      currentState->Enter();
    }
  }

protected:
  FSMState<T> *     lastState;
  FSMState<T> *     currentState;
};
```

FSMState. The FSM contains only states of a certain pattern, like in listing 2. Both the ChaseState and the FleeState derive from this class.

Listing 2: A finite state machine state.

```
template <typename T>
public class FSMState
{
public:
                FSMState(T *_owner)
                {
                  owner = _owner;
                }

  virtual void      Enter() = 0;
  virtual void      Update(/*float deltaTime*/) = 0;
  virtual void      Exit() = 0;

  //char*          Name;

protected:
  T*             owner;
};
```

NPC. The NPC class derives from the FSM, as shown in listing 3. It stores instances of the states it needs. The best next node to move to is calculated in the ChaseState. The FleeState doesn't do much more than randomly choosing new nodes after five turns.

Listing 3: Curiously recurring template pattern.

```
class NPC : public FSM<NPC>
{
public:

  ChaseState*       chaseState;
  FleeState*        fleeState;

  .
  .
  .
};
```

Listings

List of Figures

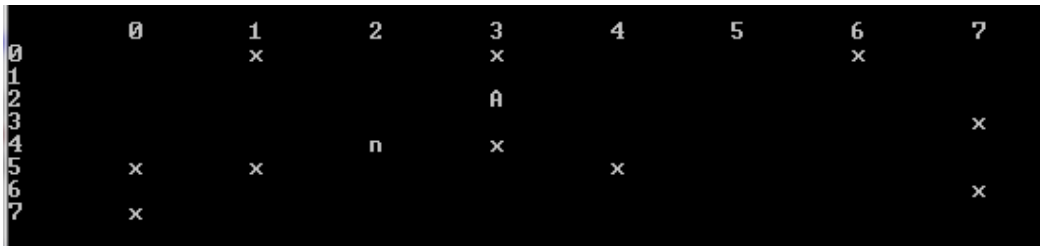

Figure 1: A screenshot of the result.