CAPIRSolver – Technical Specification

Remarks: In AI Game Engine, I used two third-party code portions.

1. RapidXML: a single-file, simple and fast XML parser
   1. Author: Marcin Kalicinski
   2. URL: http://rapidxml.sourceforge.net
   3. License: Boost or MIT License
2. String compression/decompression code snippet using ZLib
   1. Author: Timo Bingmann
   2. URL: <http://idlebox.net/2007/0328-ZLibString.blog>
   3. License: None (agreed by the author via email)

Table of Contents

Short introduction 3

Class Hierarchy 3

Some notable design details 4

MazeWorldDescription 4

Must-know Structs 4

State 4

Code 4

AbstractState 5

Components 5

Extending classes 6

Player 6

Routines to extend 6

Monster 6

Summary 6

Routines to extend 7

Maze 7

Summary 7

Routines to extend 8

MazeWorld 8

Summary 8

Routines to extend 8

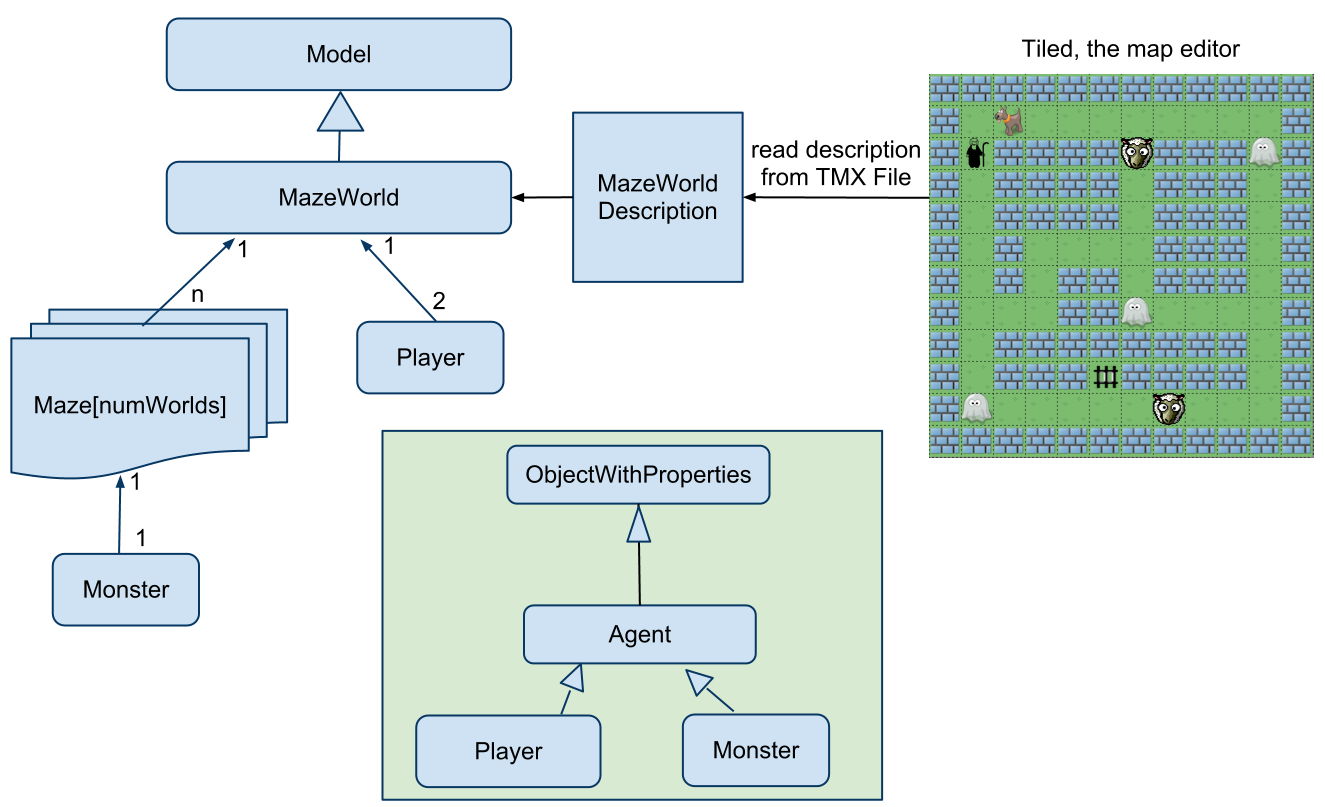
Game Solver 8

# Short introduction

The engine is targeted for Collaborative Puzzle Role Playing Games, in which each level of a game consists of a set of puzzles that a human player and his sidekick need to solve. Generally, a puzzle is a task that produces rewards when completed such as bringing a sword to the castle, killing a dragon, herding a sheep into a pen, or as simple as opening an important door. In the following sections, we use the terms ***maze***, ***world*** and ***puzzle*** interchangeably to indicate an individual puzzle in the multi-puzzle model that represents one level of a game.

The engine acts as a toolkit to create games in which AI is done by solving the MDPs (Markov Decision Processes) formalized in the game and not by writing AI scripts. Each puzzle is encoded by an MDP, which specifies how an NPC behave with respect to its relative distance to the players, and how rewarding certain game states are. These two pieces of information play a crucial role for the AI in determining what the most rewarding course of actions to execute. Therefore, to make use of this AI automated generation there are classes that need extending for any new game.

# Class Hierarchy



MazeWorld (a level of the game) consists of:

1. Two Player objects, and
2. Multiple Maze objects, each of which consists of one NPC (Monster).

# Some notable design details

## MazeWorldDescription

MazeWorldDescription (MWD) stores information about the map, as parsed from the TMX file created using Tiled the map editor. A game level must be able to take an MWD struct and allocate memory as well as initialize its components, such as human, assistant and mazes as designed in Tiled.

typedef vector<long> ID\_PosX\_PosY;

typedef pair< string, string > Property;

typedef pair< long, Property > ID\_Property;

struct MazeWorldDescription

{

// this stores the list of triple (id, x, y) of characters on the map.

ID\_PosX\_PosY characters;

// this stores the list of pair <id, Property> found in the map

vector<ID\_Property> properties;

// solver related data

………

};

The two arrays most probably needed are ***characters*** and ***properties***.

* ***characters*** is an array of type long, consisting of triple (tile\_id, coordX, coordY). It stores the positions of all tiles that are not ground tile or wall tile (whose ids are preset). Therefore, this array tells the initial positions of human, assistant, all monsters and specialLocations.
* ***properties*** is an array of pair (tile\_id, Property) where Property is pair (property\_name, property\_value). It stores the properties of all tiles identified by tile\_id.

# Must-know Structs

When formalized as MDPs, the game evolves from one state to another. Extending classes will have to deal with these game states quite frequently, so a good grasp of their structures would be greatly beneficial.

## State

Usage   
***State*** contains information related to the two players and all mazes at one point of time.

### Code

struct State

{

// playerProperties[i]: (coordX, coordY, <additional properties>)

vector<long> playerProperties[2];

// mazeProperties[i]: (coordX, coordY, <additional properties>)

vector< vector<long> > mazeProperties;

};

Components  
playerProperties: **playerProperties**[0] contains human’s data and **playerProperties**[1] contains AI assistant’s data. Exact location data is compulsory while other additional properties are optional.

mazeProperties: **mazeProperties**[i] stores data of the i-th maze. The first two items in **mazeProperties**[i] store X and Y coordinates of the monster in grid map while the rest are additional properties of that Monster such as hit points if any.

## AbstractState

Usage   
***AbstractState*** is used at planning stage in one maze (each maze may have one distinct set of AbstractStates). The most notable difference from ***State*** is that a location abstraction can be in use or not. When abstraction is used, the players’ positions are abstracted to the region index, instead of exact locations. A ***region*** is either a junction or a corridor. Note that the monster’s position is not abstracted. Our reason for this type of abstraction is that we assume a monster’s behaviors are affected by whether it sees one of the players or both of them, and knowing which region the players are in is enough to plan for actions (if a player is in a region not seen by the monster and wants to be seen by the monster, he can execute a move action towards the monster’s position). When abstraction is not used, AbstractState ignores region information.

Code

struct AbstractState

{

// playerProperties[i]: (regionId, coordX, coordY, <additional properties>)

vector<long> playerProperties[2];

// if monsterProperties is not empty:

// (coordX, coordY, seesHuman, seesAssistant, <additional properties>)

vector<long> monsterProperties;

};

### Components

playerProperties: Similar to State, playerProperties[0] stores human’s data and playerProperties[1] stores assistant’s data.

Note that when the Solver is run with abstraction on, among regionID, coordX and coordY, most of the time only regionID is used for planning. coordX and coordY is used when player is in the same region with monster and there is a need to determine their relative position. In that case (coordX, coordY) take the value of the end of the region the player is at.

monsterProperties: The first and second items store the exact coordinates of the monster, while the third and fourth indicate whether monster sees any of the players. The rest is the monster’s additional attributes.

# Extending classes

Most classes are customizable and can be extended as much as desired by overriding default implementation. We will summarize the meaning of important classes and routines.

## Player

Summary

This is the base class for both the Human-controlled player and AI-controlled assistant. The corresponding child classes need to specify the ability of the character, as well as any additional attributes it may have. By default, a player has five actions {no\_move, E, S, W, N}, i.e., clockwise from East, and no attribute.

* The movement actions are mapped to {0,1,2,3,4}

### Routines to extend

1. Constructor
2. **(Optional)** Dynamic routines for special actions: Note that by special actions, we mean actions that do not alter the positions of related characters but their properties only, for instance, shooting, picking up things, talking, etc. These special actions should take value from 5 onwards. Note that with special actions
   1. void absExecuteSpecialAct( vector< pair<AbstractState, double> >& input, long act, double probAct, vector< pair<AbstractState, double> >& output)
      * This routine deals with AbstractState, used for planning
      * Inputs include:
        1. Array of AbstractState with probability ***input***.
        2. The special action ***act*** with probability ***probAct***
      * Output is an array of AbstractState with probability ***output***.

## Monster

### Summary

Every type of monsters (NPCs) should have a class of their own. Our assumption is that monsters’ behaviors differ depending on whether they see one player or two players at the same time or none at all. For example, a sheep moves randomly when not seeing any of the players, runs away from the sole player it sees, or runs away from the nearer player between the two players it sees. Besides, similar to the players, they could have special actions as well and we also suppose their decisions on when to do which special action also depends on whether they are threatened by the sight of players.

By default, our monsters run away from any player on sight and wander randomly otherwise. Nonetheless, if other types of dynamics are required, game designers can make use of our set of predefined behaviors such as

* moveActsTowardsLocation: Getting the set of move actions towards a certain location
* moveActsAwayFromLocation: Getting the set of move actions away from a certain location
* moveActsFarthestFromBothPlayers: Getting the set of move actions away from both players
* moveActsFarthestFromPlayer: Getting the set of move actions away from a specific player
* randomMoveActs: Getting the set of available move actions

### Routines to extend

1. Constructor
2. **(Optional)** Dynamic routines: As explained above, if the monster is designed to react differently from the default behaviors, these routines must be supplied. For example,
   1. void absGetActNoneSeen(const AbstractState& absState, long humanAct, long aiAct, const AbstractState& prevAbsState, vector<pair<long,double> >& action\_prob)
      * This routine deals with AbstractState, used for planning when the monster does not see any of the players.
      * Inputs include:
        1. Current AbstractState ***absState***. This AbstractState already incorporates the effects of ***humanAct*** and ***aiAct***.
        2. The players’ actions ***humanAct*** and ***aiAct***.
        3. Previous AbstractState ***prevAbsState***. This AbstractState is before humanAct and aiAct are executed.
      * Output is an array of actions with probability ***action\_prob***.
      * Similarly are absGetActHumanAgentSeen and absGetAct1AgentSeen.
   2. void absGetAffectedByPlayersActions(vector< pair< AbstractState, double> >& input, long humanAct, long aiAct, vector< pair<AbstractState, double> >& output)
      * This routine is used when the special actions of players have effect on the NPC.
      * For instance, a long-range regional attack could send a signal to all NPCs around the offending player, but only certain kinds of NPCs receive HP loss. Such effects can be encoded in this routine of pertinent NPCs.

## Maze

### Summary

Each Maze consists of at most one Monster and one SpecialLocation and at least one of the two. Besides it can contain other properties.

### Routines to extend

1. Constructor
2. Initialization routines: There’s no format for these routines, because each maze could require different sets of parameters to initialize.
3. double absGetReward(AbstractState& absState)
   * This routine returns the reward or penalty of a game state.

## MazeWorld

### Summary

MazeWorld is the base class for levels of the game.

### Routines to extend

1. Constructor
2. void initializeHumanAssistantMazes(MazeWorldDescription& desc)
   * Initialization routine which takes a MazeWorldDescription and initializes human, assistant and all mazes involved.
3. char\* getWorldRepresentation()
   * Returns the char array that represents the whole level.
   * This array will be sent to the game GUI to initialize the game level graphically.

# Game Solver

This executable orchestrates aforementioned classes and computes AI for a given game level. The structure of this file can be copied directly from the sample GhostBustersSolver in /src folder provided along with the game engine’s source code. There are some customizable parameters used by the MDP solver, if game designers so desire, although there is generally no need to do so.

Please refer to the HTML API specification for more in-depth understanding.