PART 17: "AI"

Introduction

Like physics, AI (*Artificial Intelligence*) is an incredibly complex field, and it's always changing. Some companies allocate entire teams to the subject to make their game provide challenging game play. So we will only cover a few basic concepts in this tutorial.

A key point about AI is that it is designed to provide "intelligent" players in a game. But don't make the mistake of assuming 'intelligent' means 'perfect'. The best AI should always introduce an element of "error" in its approach, because if you are trying to emulate another human player/controller… humans are rarely perfect.

AI Engines

Though not as commonplace as Physics or graphic engines, there are a few AI engines available that provide some basic AI features. Despite this, AI very much depends on the type of game you are playing and what kind of gameplay you want your computer controlled players to do, therefore you will need to carefully consider how you make a computer opponent function.

Let's start at the beginning. One of the simplest forms of AI is to get an enemy (or ally) to chase you. In an open playfield, it is a simple matter to determine if a player is above or below you, or to the left or right and move in the direction to intercept.

Let's do this with our trusty tank game from previous episodes. Load up the template and create two tanks. One of which you can move around with key control using a speed variable you can adjust. The other needs to be controlled by a simple AI routine called TankMove(). Which you can leave blank for now.

Once you have your tank moving under player control, we need to consider what the process is. Consider the following code:

```
enum {up,down,left,right}; //these are our list of possible commands/directions
int direction;
int speed = 2; // make sure it is slower than the speed of the player tank

// this is our decision phase, work out what command(direction) we want
if (player.x < cpu.x) direction = left; else direction = right;
if (player.y < cpu.y) direction = up; else direction = down;

//we've set up our command, let's implement it.
switch (direction)
{
case up:    cpu.y -= speed; break;
case down:  cpu.y += speed; break;
case left:  cpu.x -= speed; break
case right: cpu.x += speed; break;
}
```

Consider what we are doing here. It's a simple movement system which first decides on a direction, then moves based on that decision. It does not currently make any attempt to prevent the CPU tank being on top of the player tank, and whenever you move the player tank the CPU tank should move to the same place and hover around that spot if the player tank stops moving.

Code this up, in your TankMove() routine; also to prevent hovering/shudder, add some code to not move when the CPU tank is over the player tank. (hint: add a command to your list called `stay`).

Constrained AI

Your simple tank movement is an example of an unconstrained logic decision. The tank simply moves left/right/up/down as it needs and does not give any consideration to anything else.

Constraints can be added to this. For example: suppose we do not allow our CPU tank to go outside a certain boundary in the middle of the screen. Our human controlled tank can still move all over the screen.

Go back to your code, and in the movement part of your code, prevent the CPU tank from moving to x positions less than 100 or greater than 500, and the y positions less than 150 and greater than 450. Why do you think you do this check in the movement part and not in the decision part? (**answer at the end!*). Note: it is important to make sure your CPU tank starts within these boundaries.

Now as it stands, your CPU tank has to obey certain simple boundary rules which prevent its movement, but once those rules do not interfere, it will move towards you and pounce if you enter its range. Now add some code to prevent our CPU tank moving at all unless the player enters this range.

Too perfect?

Despite having a boundary limit, our CPU tank is a pretty hard opponent to avoid, as soon as you go into his range he will home in to you. Imagine if he were able to move diagonally, or even in a vector movement, he'd be pretty hard to sneak past…

It might be helpful if we were to introduce a little inconsistency in his movement. There are 2 simple ways to do this

1. *Reduce the frequency of decisions*
   At the moment the tank is making a decision and moving every frame. Suppose he only made a decision every 8th frame. Try adding a timer that counts down each time the decision code is called, and only allows the decision part of your code to function when the timer is 0. Remember to reset the timer when you make a decision.

2. *Add some degree of error in decisions*

The easiest way to add error, is to provide a degree of randomness to the decision process. After your decisions are made, add some code to overwrite the direction choice with a random value 0-3 (which will correspond to a direction). Note though: doing this each frame will be very unsatisfactory but try it and see. How could you make this randomness more effective? (***Answer later*)

There are of course many other methods to add inconstancy, but these work well for most things.

Pacman AI

Pacman was probably one of the first arcade games ever to make use of AI. We can approximate its ghost movement with a variation of the logic presented here.

Essentially Pacman AI operates like this: if the ghost is above you, it moves down, and if left of you, it moves right. But there were more constraints in place and also some forced decisions.

For example, in Pacman the ghosts all have to move inside the maze and obey the collision systems. They also had different levels of intelligence, ranging from very clever to downright stupid. As an example of a forced decision, it also has enough intelligence to not get stuck in corners/dead ends, should the Pacman not move and provide updated homing information. If a ghost is moving in a direction which has a dead end, it is forced into choosing another direction, sometimes randomly, sometimes the most effective direction.

The biggest constraint imposed on a Pacman ghost though was the maze. The ghosts must be able to home in on the hero but they have to move within the maze.

The key to the ghost's movement though is deciding *when* to move. We could use the timer we already have, but we already know we can only move in spaces where the maze allows movement, i.e. at junction points. Therefore rather than a timer, what we need is a way to tell if we are at a junction point.

Assignment

Using tiles and a 2D array, create a Pacman style maze (no need for dots). You just need 2 values in the array and on screen, so you can create walls and spaces. Position your tank on screen, and from then on reference the array to decide if it can move around the maze.

Now place another tank on screen, as your chase ghost. The logic is essentially the same as we did previously but rather than x and y boundaries, test your array to see if there is space in the chosen direction, and if so allow the movement.

Calling this update routine every frame will make your tank super effective at hunting you down as you move inside your maze. Try adapting your AI to detect if you are at a junction point (at least 1 point adjacent to you in the array is blank) and then make your decisions to move.

Once this is working, add a small counter, and make your ghost change direction every other junction point.

You can add many refinements to this basic chase code. Feel free to add more “ghosts”, change the frequency of their decision process, and their speed. Also make sure any random choices you do make in forced situations result in valid decisions.

*Answer: At the point of decision making, you do not yet know which direction you are moving, therefore you can only use the current x and y coordinates. If either of those are out of the specified range, it could choose the stay command, which prevents any further movement.
** Answer: Calling random every frame is chaotic and not “intelligent” in anyway, however by “occasionally” choosing a random number, combined with a reduced frequency decision process, you can throw in odd movements. “Occasionally”,suggests a process to decided when to do this, you can use another timer or test a random number for a certain range to trigger these “errors”

END OF PART 17

Next part: “TBD”