DM810

Computer Game Programming II: AI

Lecture 11

Decision Making

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

Resume

- Decision trees
- State Machines
- Behavior trees
- Fuzzy logic
- Goal-Oriented Behavior
- Rule-Based Systems
- BlackBoard Architectures

Outline

Outline

Scripting

- early AI in games: hard-coded using custom written code to make decisions
  - good for small development teams
  - still the dominant model for platforms with modest development needs (i.e., smart phones)
- nowadays tendency to separate the content from the engine:

  scripts: data files in a language simple enough for behavior/level designers
  - behavior of game levels (which keys open which doors)
  - programming the user interface
  - rapidly prototyping character AI

Scripting makes possible:

- Mods or modifications of original game:
  new items, weapons, characters, enemies, models, textures, levels, story lines, music, and game modes
  - partial conversions add new content to the underlying game
  - total conversions create an entirely new game.

Features for Language Selection

- Speed: meet the needs of rendering, though it may be running over multiple frames
- Compilation and Interpretation: broadly interpreted, byte-compiled (eg. Lua), or fully compiled
- Extensibility and Integration: function calls, embedding
- Re-Entrancy: ie. pause execution, or force completion

Using an existing language has the advantage:

- reuse debugging and extensions from the community

Developing your own language has the advantage:

- specialization and simplicity

Open-source software: rights depend on specific license, consult experts if plan to redistribute

Languages

- Lua, simple procedural language, has a small number of core libraries that provide basic functionality, which is good.
  Does not support re-entrant functions
  events and tags: events occur at certain points in a script's execution; tag routines are called when the event occurs, allowing the default behavior of Lua to be changed.
  impressive performance in speed
  syntax not very simple
  most used pre built scripting language

- Scheme, functional language, integration of code and data

- Python, re-entrant function generators, speed (hash lookup table) and size (heavy linkable libraries) are disadvantages, pygame

Game Programming with Python, Lua, and Ruby, Tom Gutschmidt

Your Own Implementation

- tokenizing (Lex)
- parsing (Yacc)
- compiling
- interpreting
- just-in-time compiling

Types of Actions

- State change actions: changing some piece of the game state (visible effects or hidden variables)
- Animations: visual feedback
- Movement through the game level: calling the appropriate algorithms and passing the results onto the physics or animation layer
- AI requests: complex characters, a high-level decision maker may be tasked with deciding which lower level decision maker to use

Actions may combinae more of these types.
A general action manager will need to cope with actions that take time

- Decision maker keeps scheduling the same action every frame, need actions that can send a signal when finished.
- Interrupting Actions: Our action manager should allow actions with higher importance to interrupt the execution of others.
- Compound Actions: actions carried out together
  action of a character is typically layered
  split and generated by different decision making
  need for accumulating all these actions and determine which ones can be layered
  alternative is to have decision makers that output compound actions
- Scripted Actions: pre-programmed actions that will always be carried out in sequence by a character
  problematic with changing environment
  sequence delegated to the decision making to which one can return if changes occur.

Every action has:

- expiry time
- priority
- methods to check if it has completed
- if it can be executed at the same time as another action
- if it should interrupt currently executing actions
- track of component action that is currently active

Action Manager:

- queue where actions are initially placed and wait until they can be executed
- active set: actions that are currently being executed.

Action Manager

1. Actions are moved to the active set, as many as can be executed at the same time, in decreasing order of priority.

2. At each frame, active actions are executed, and if they complete, they are removed from the set.

3. If a new item has higher priority than the currently executing actions, it is allowed to interrupt and is placed in the active set.

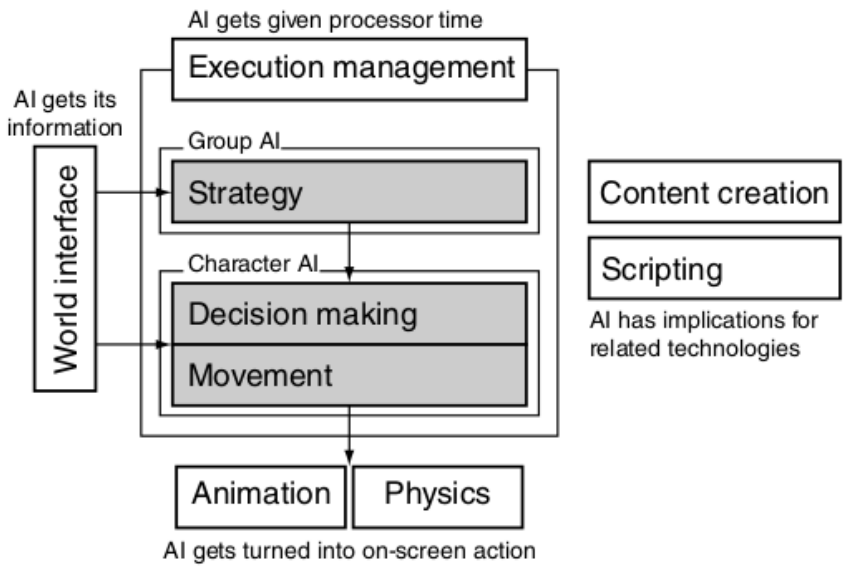
Outline

Tactical and Strategic AI

So far: single character and no use of knowledge from prediction of the whole situation.

Now:

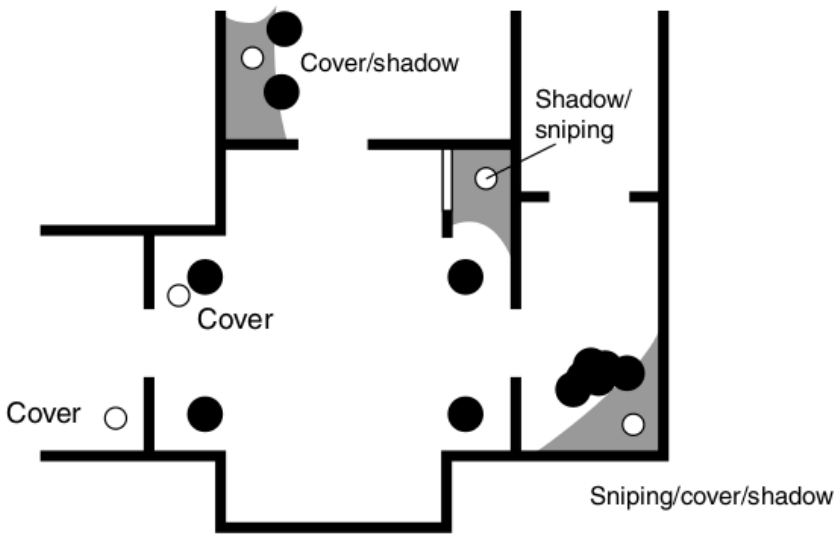
- deduce the tactical situation from sketchy information
- use the tactical situation to make decisions
- coordinate between multiple characters



Waypoint Tactics

Waypoint tactic (aka rally points): positions in the level with unusual tactical features. Eg: sniper, shadow, cover, etc

Often pathfinding graph and tactical location set are kept separated



Outlook

Waypoint tactical properties

- compound tactics
- connections
- continuous properties
- context sensitivity

Using tactical waypoints

Automation:

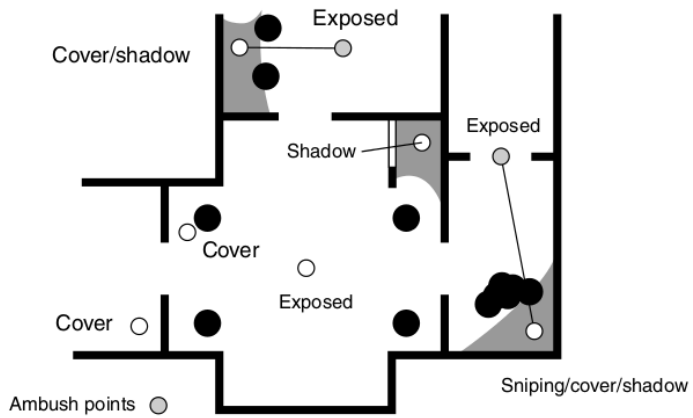
- assess tactical properties
- determine locations

Primitive and Compound Tactics

Store primitive qualities: cover, shadow, and exposure

ambush needs cover and shadow and exposure within a certain ray $\rightsquigarrow$ we can decide which locations are worth

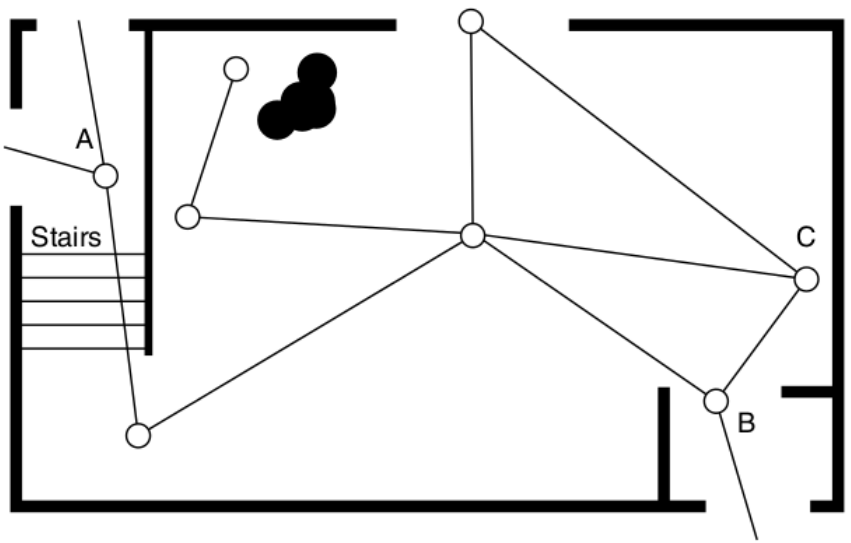
- limit the number of different tactical qualities: support a huge number of different tactics without making the level designer's job hard or occupying large memory
- slower: need to look for points and then whether they are close
- speed is less critical in tactics
- pre-processing offline to identify compound qualities: increase memory but still relieves the designer from the task. algorithms can also be used to identify qualities



Waypoint Topology

Example of topological analysis $\rightsquigarrow$ need for connections:

- we're looking for somewhere that provides a good location to mount a hit and run attack
- we need to find locations with good visibility, but lots of exit routes.



Mostly resolved by level designer

Continuous Tactics

Qualities have continuous degrees.
We can use fuzzy logic
Eg:

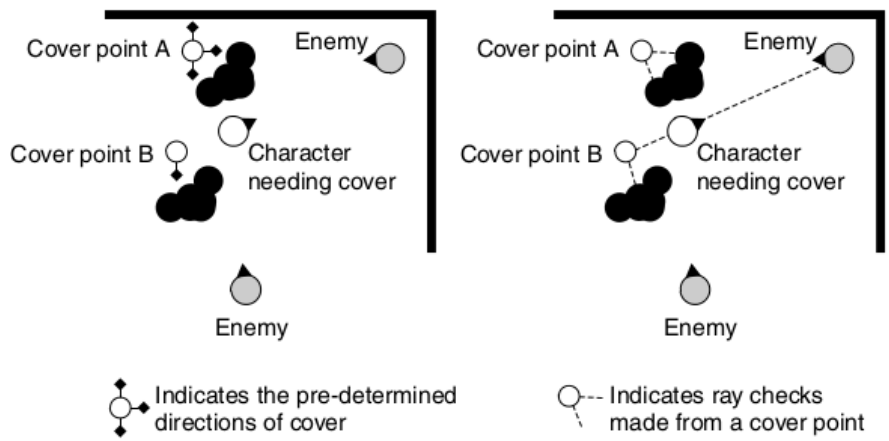
$$\text{sniper} = \text{cover AND visibility}$$

$$\begin{aligned} m_{\text{sniper}} &= \min(m_{\text{cover}}, m_{\text{visibility}}) \\ &= \min\{0.9, 0.7\} \\ &= 0.7. \end{aligned}$$

Context Sensitivity

The tactical properties of a location are almost always sensitive to actions of the character or the current state of the game.

- store multiple values for each node (eg. for different directions)
- keep one single value but add an extra step (post-processing) to check if it is appropriate (line of sight, influence map, heuristic for sniper points: has fired there? etc.)



If 4 directions + both standing and crouched + against any of five different types of weapons $\rightsquigarrow$ 40 states for a cover waypoint.

Using Tactical Waypoints

- controlling tactical movement
- incorporates tactical information into the decision making process
- tactical information during pathfinding

Simple Tactical Movement

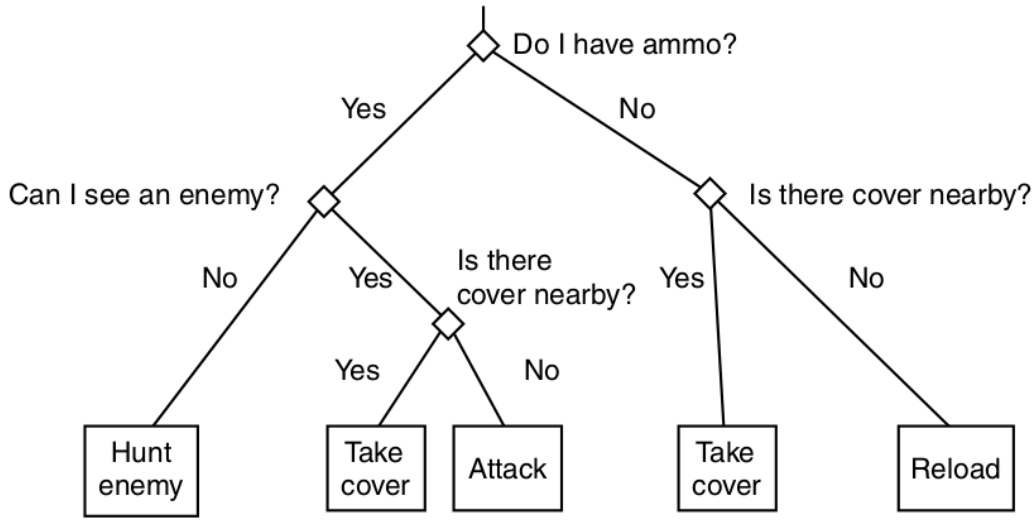
Action decision: reload, under cover.
Tactical decision: querying the tactical waypoints in the immediate vicinity. (Suitable waypoints are found, and any post-processing done)
The character then chooses a suitable location

- nearest suitable location,
- numerical value for quality of location
- combination

Issue: tactical information is not included in decision making hence we may end up discovering the decision is foolish

Use tactical info in Decision Making

Example with decision tree:



Similarly, with state machine we might only trigger certain transitions based on the availability of waypoints.

Tactical Information in Fuzzy Logic Decision Making: take account of the quality of a tactical location

IF cover-point THEN lay-suppression-fire
IF shadow-point THEN lay-ambush

lay-suppression-fire: membership = 0.7
lay-ambush: membership = 0.9

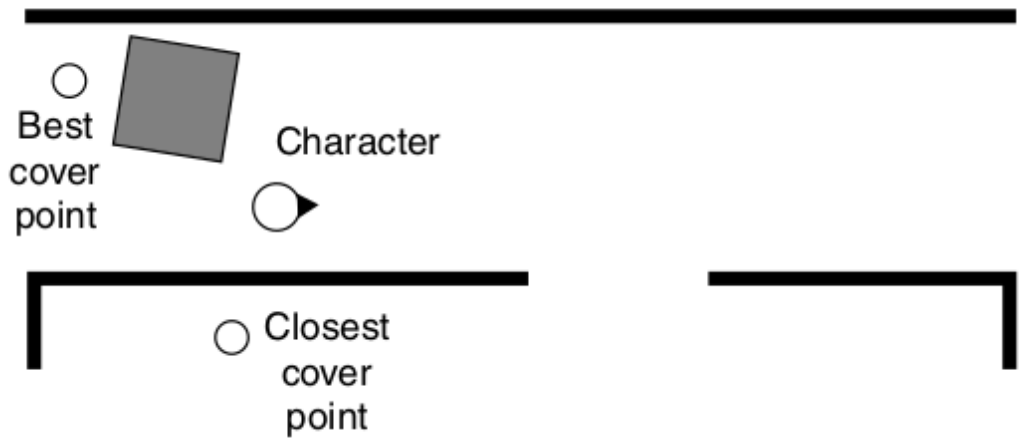
IF cover-point AND friend-moving THEN lay-suppression-fire
IF shadow-point AND no-visible-enemies THEN lay-ambush

friend-moving = 0.9
no-visible-enemies = 0.5

lay-suppression-fire: membership = min(0.7, 0.9) = 0.7
lay-ambush: membership = min(0.9, 0.5) = 0.5

Given the location of a character, we need a list of suitable waypoints in order of distance.

Use appropriate data structures: quad-trees, binary space partitions, multi-resolution maps (a tile-based approach with a hierarchy of different tile sizes)



- level designer may not use thin walls in a game level.
- use pathfinding to generate the distance (prune, when distance exceed best so far)

Generating the Tactical Properties

Level designer may express tactical location but rarely also define properties
$\rightsquigarrow$ Tools supporting the designer (preprocessing or online)

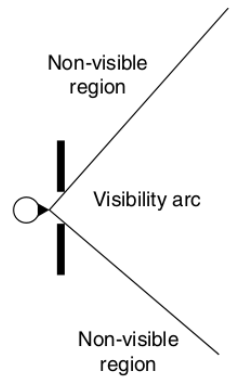
Algorithms for calculating a tactical quality depends on the type of tactic:

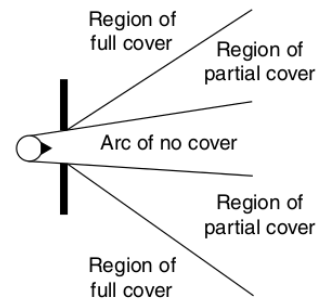
- Cover points
  testing how many different incoming attacks might succeed (line-of-sight)
  check attacks at regular angles around the point, from the location selected, a ray is cast toward the candidate cover point (random point in a person-sized volume)

- Visibility points
  line-of-sight tests: quality is related to average length of the rays sent out

- Shadow points
  samples from character-sized volume around the waypoint and ray casts to nearby light sources or looking up data from the global illumination model.
  Take maximum lightness

```
for i in 0..iterations:

   # Create the from location
   from = location
   from.x += RADIUS * cos(theta) + randomBinomial() * RANDOM_RADIUS
   from.y += rand() * 2 * RANDOM_RADIUS
   from.z += RADIUS * sin(theta) + randomBinomial() * RANDOM_RADIUS

   # Check for a valid from location
   if not inSameRoom(from, location): continue
   else: valid++

   # Create the to location
   to = location
   to.x += randomBinomial() * characterSize.x
   to.y += rand() * characterSize.y
   to.z += randomBinomial() * characterSize.z

   # Do the check
   if doesRayCollide(from, to): hits++

   # Update the angle
   theta += ANGLE

return float(hits) / float(valid)
```





Automatically Generating Waypoints

Similar to waypoints for pathfinding

- Watching Human Players
- Condensing a Waypoint Grid
  for each property test valid locations in the level and choose the best.
  Assess with real-valued tactical qualities.
  Tactical locations compete against one another for inclusion into final set. We want either high quality or a long distance from any other waypoint in the set (high discrepancy)