DM842

Computer Game Programming: AI

Lecture 8

Tactical and Strategic AI

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

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

Outline

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 combine 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

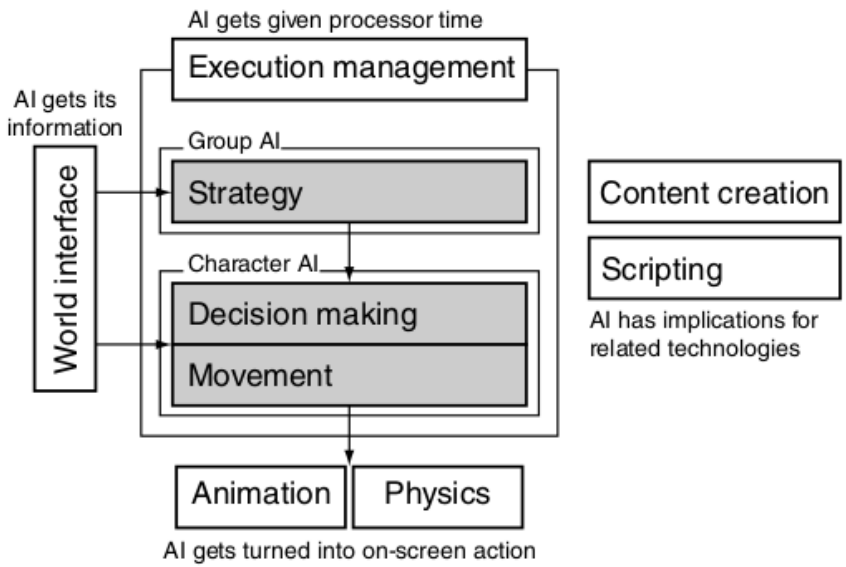
Outline

Tactical and Strategic AI

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

Now:

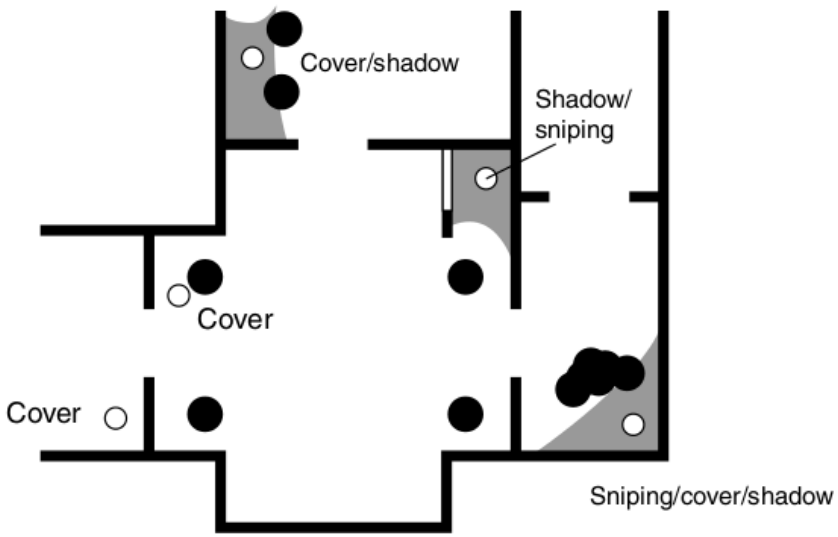
- deduce the tactical situation from sketchy information
- use the tactical situation to make decisions
- coordinate between multiple characters



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:

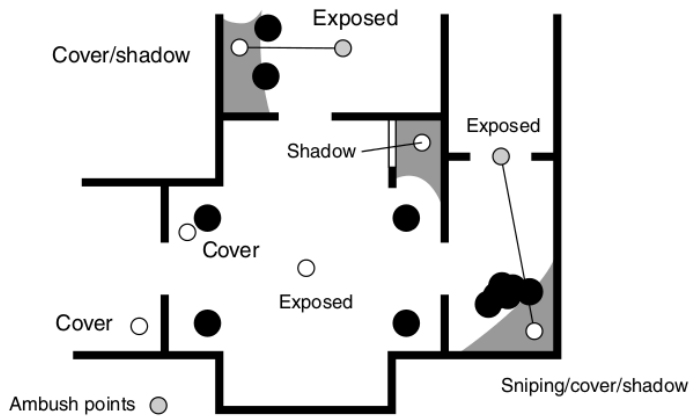
- assess tactical properties
- determine locations

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

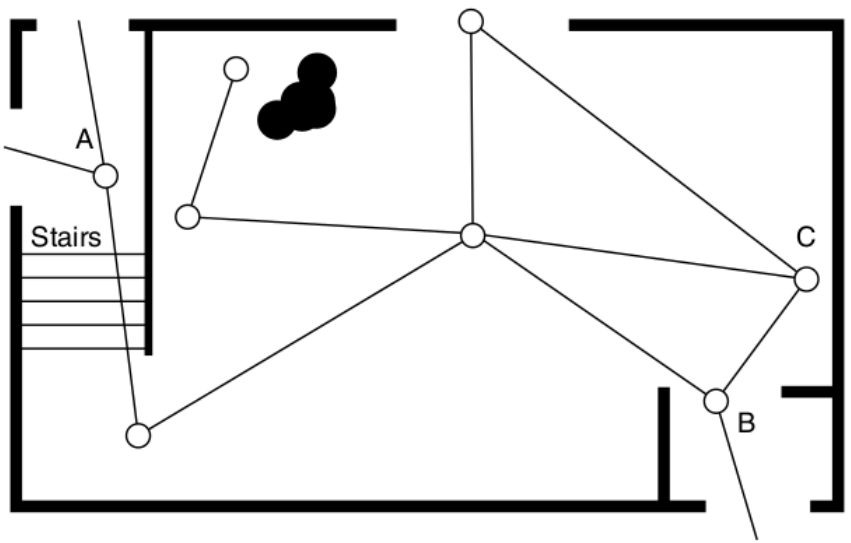
- limit the number of different tactical qualities: support a huge number of different tactics without making the level designer's job hard or occupying large memory
- slower: need to look for points and then whether they are close
- speed is less critical in tactics
- pre-processing offline to identify 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:

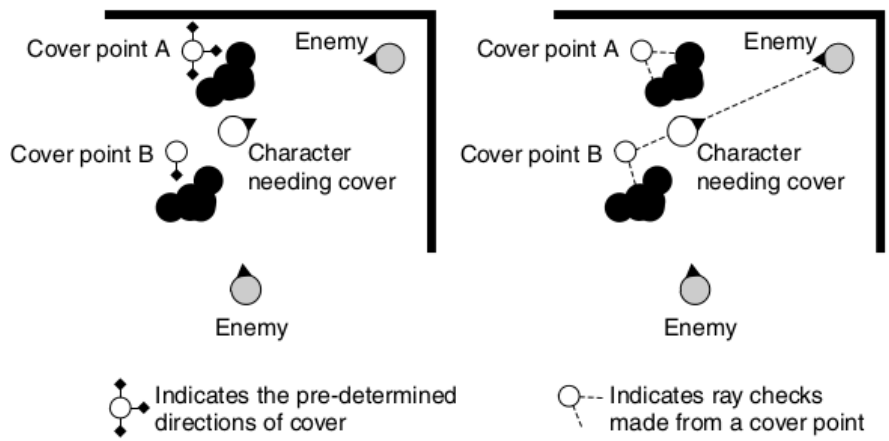
$$\text{sniper} = \text{cover AND visibility}$$

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

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

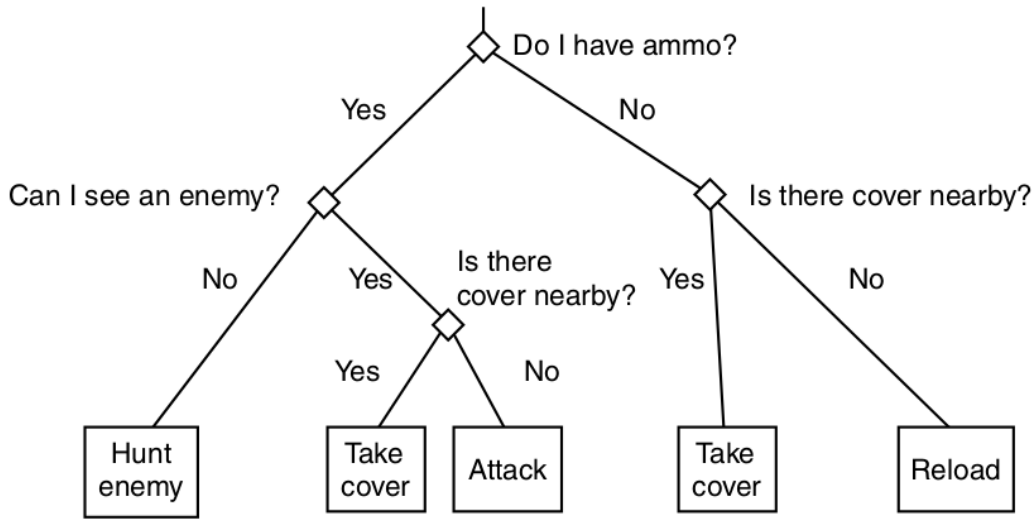
Action decision: reload, under cover.
Tactical decision: querying the tactical waypoints in the immediate vicinity.
(Suitable waypoints are found, and any post-processing done)
The character then chooses a suitable location

- 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

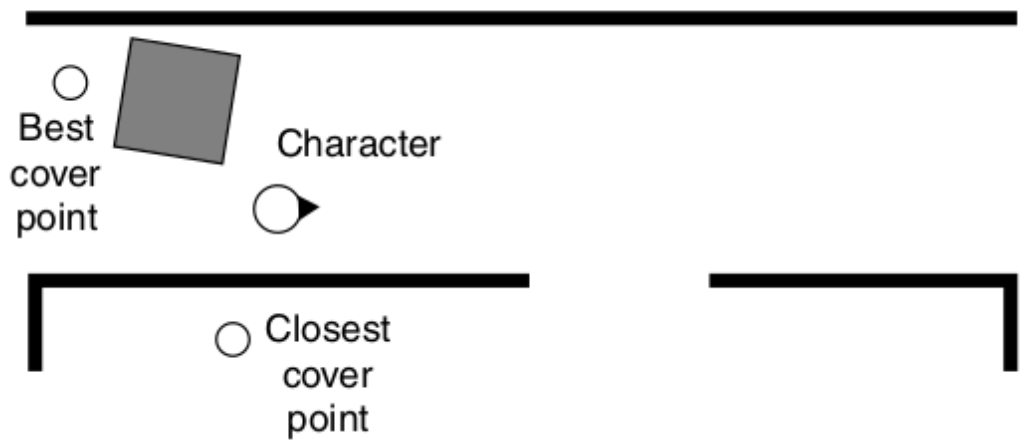
lay-suppression-fire: membership = min(0.7, 0.9) = 0.7
lay-ambush: membership = min(0.9, 0.5) = 0.5

With Pathfinding

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)

- problems with the situation below:



- use pathfinding to generate the distance (stop, when distance exceed best so far)

Generating the Tactical Properties

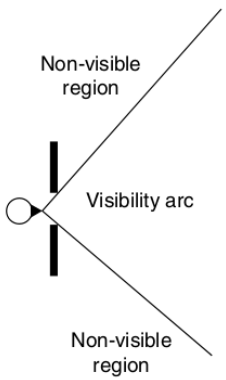
Level designer may express tactical location but rarely also define properties
$\rightsquigarrow$ Tools supporting the designer (preprocessing or online)

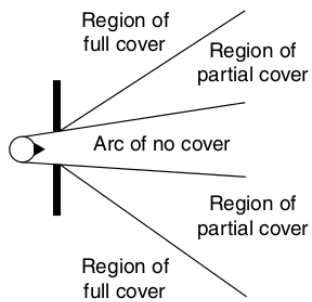
Algorithms for calculating a tactical quality depend on the type of tactic:

- Cover points
  testing how many different incoming attacks might succeed (line-of-sight)
  check attacks at regular angles around the point (with random offsets for the height). From the location selected, a ray is cast toward the candidate cover point

- 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






```
def getCoverQuality(location, iterations, characterSize):
    theta = 0 # Set up the initial angle
    hits = 0 # We start with no hits
    valid = 0 # Not all rays are valid
    for i in 0..iterations:
        # Create the from location
        from = location
        from.x += RADIUS * cos(theta) + rand(-1,1) *
            RANDOM_RADIUS
        from.y += rand(0,1) * 2 * RANDOM_RADIUS
        from.z += RADIUS * sin(theta) + rand(-1,1) *
            RANDOM_RADIUS
        # Check for a valid from location
        if not inSameRoom(from, location): continue
        else: valid++
        # Create the to location
        to = location
        to.x += rand(-1,1) * characterSize.x
        to.y += rand(0,1) * characterSize.y
        to.z += rand(-1,1) * characterSize.z
        if doesRayCollide(from, to): hits++
        theta += ANGLE # the smaller iterations the
            larger 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)

Outline

Tactical Analysis

Primarily present in Real-time strategy (RTS)
(as opposed to turn-based strategy) games

- influence mapping: determining military influence at each location
- terrain analysis: determining the effect of terrain features at each location

But other tactical information too: eg, regions with lots of natural resources to focus harvesting/mining activities.

Influence Maps

- game level split into chunks made of locations of roughly same properties for any tactics we are interested in
(like in pathfinding: Dirichlet domains, floor polygon, tile-based grid and imposed grid for non-tile-based levels, ...)

- an influence map keeps track of the current balance of military influence at each location in the level

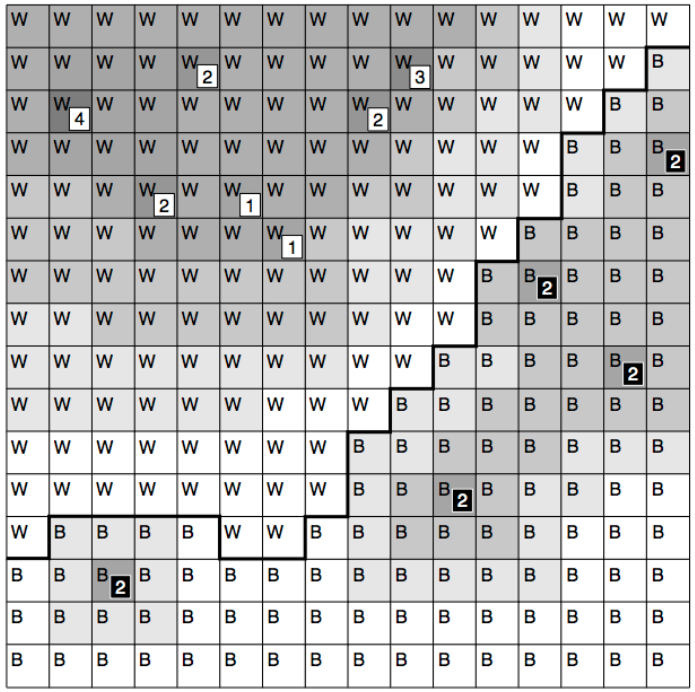
- simplifying assumption: military influence is a factor of the proximity of enemy units and bases and their relative military power (there may be many more)

- Influence is taken to drop off with distance. Let I_0 be intrinsic military power of unit

$$I_d = I_0 - d \qquad I_d = \frac{I_0}{\sqrt{1+d}} \qquad I_d = \frac{I_0}{(1+d)^2}$$

Influence Maps

- values of intrinsic influence set by level designers by visualizing the influence map, tuning needed
- influence at one location by one unit is the drop off formula
- influence of a whole side on a location is the sum of the influence of each unit belonging to that side.
- side with the greatest influence on a location has control over it
- degree of control: difference between the winning influence value and the influence of the second placed side
- degree of control high $\rightsquigarrow$ secure



Calculating Influence values

Three approaches to reduce the $O(mn)$ complexity:

- limited radius of effect
  each unit has intrinsic influence I_0 + radius of effect
  (or threshold value, $r = I_0 - I_t$)

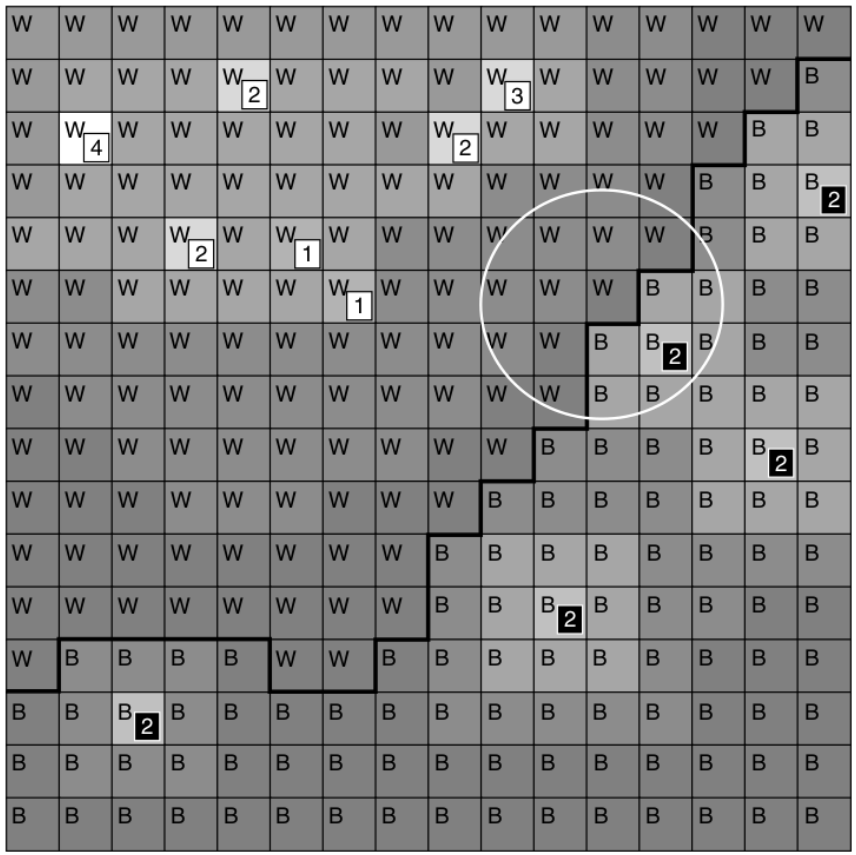
- convolution filters
  filter: a rule for how a location's value is affected by its neighbors;
  influence blurs out; expensive but graphics helps; (eg. Gaussian)

- map flooding
  influence of each location is equal to the largest influence contributed by any unit
  can use Dijkstra algorithm

Calculations can be interrupted and split over frames. Not essential that they are up-to-date.

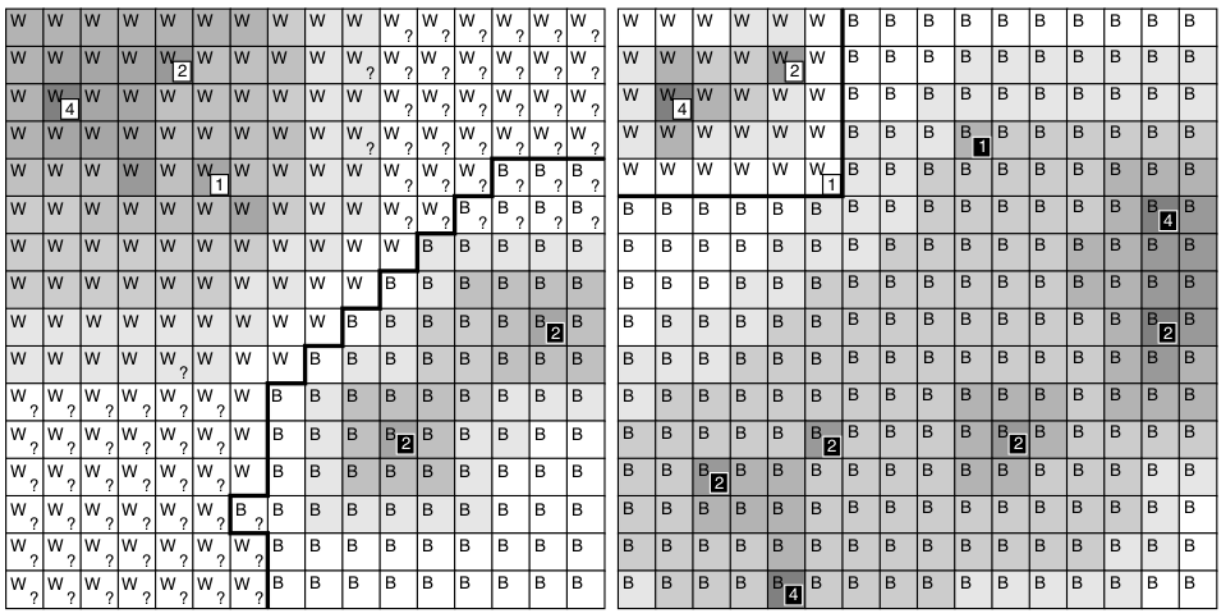
Applications

- which areas of the game are secure
- which areas to avoid
- where the border between the teams is weakest



Lack of Full Knowledge

- some units may not be in line-of-sight
- partial information $\rightsquigarrow$ the two teams may create different influence maps
  $\rightsquigarrow$ one influence map per player
- simplifications assuming full knowledge may be disappointing
- learning to predict from signs



Terrain Analysis

Similar to tactical waypoint analysis but in outdoor environments

Extract useful data from the structure of the landscape:

- difficulty of the terrain (for pathfinding or other movement)
- visibility of each location (to find good attacking locations and to avoid being seen)
- shadow, cover, ease of escape

Calculated on each location by an analysis algorithm depending on information

Terrain Difficulty

1. Each location has a type and each unit has a level of difficulty moving through terrain types.

2. ruggedness of the location: difference in height with neighboring locations (calculated offline)

Combination of 1. and 2., eg. by weighted linear sum

Visibility Maps

- check line-of-sight between location and other significant locations in the level.
- number of locations that can be seen, or average ray length if we are shooting out rays at fixed angles
- need to reduce the number of locations

Learning with Tactical Analysis

Instead of running algorithms at locations:
during the game, whenever an interesting event happens, change the values of some locations in the map.

Frag-maps learned offline during testing. In the final game they will be fixed.

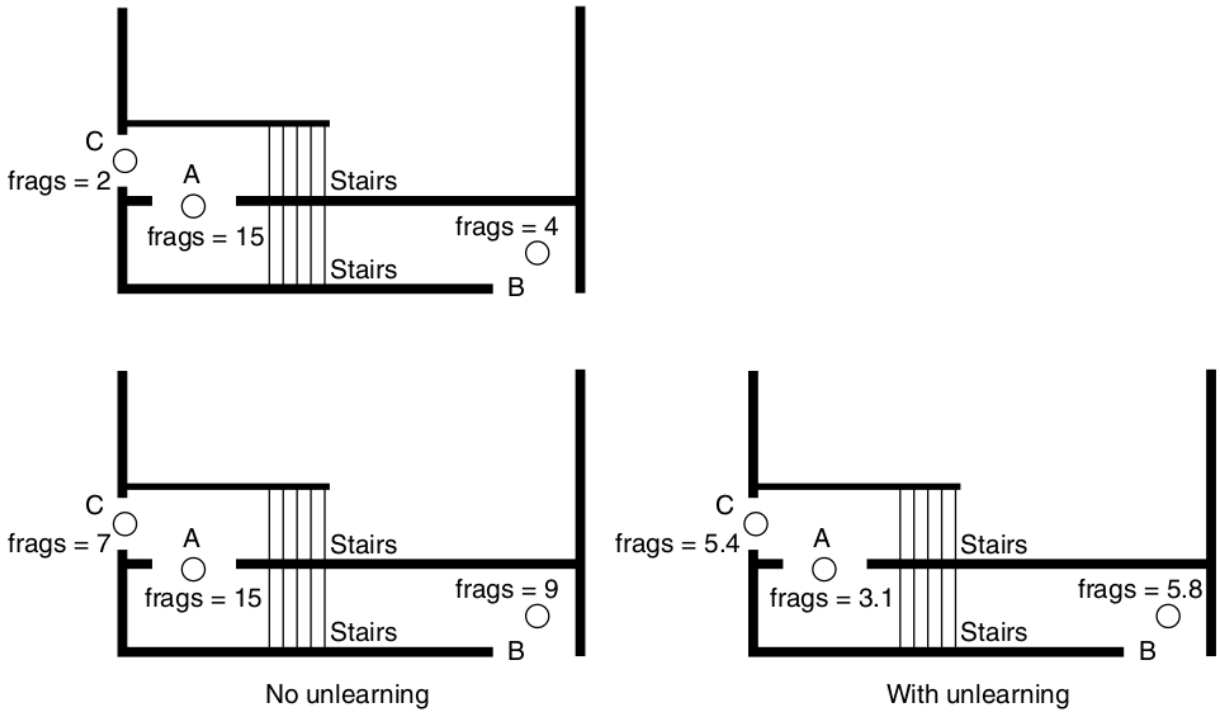
Frag-map per character:

- initially each location gets a zero value
- each time a character gets hit (including the char. itself), subtract a number from the location in the map corresponding to the victim
- if a character hits another character, increase the value of the location corresponding to the attacker.

filtering can be used to expand values out to form estimates for locations we have no experience of.

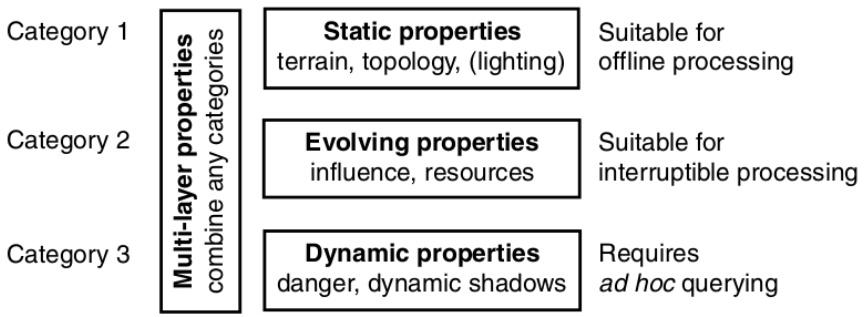
They can be adapted online, forgetting old information

- best location to ambush from
- A is exposed from two directions (locations B and C).
- character gets killed 10 times in location A by 5 attacks from B and C.
- forgetting factor 0.9



Structure for Tactical Analysis

We may need different information, not only influence and terrain analysis.



Updating tactical analysis for the whole level at each frame is too time consuming.

limit the recalculation to those areas that we are planning to use:

- neighborhood of the characters
- use second-level tactical analysis

Multi-Layer Analysis

Combine tactical information

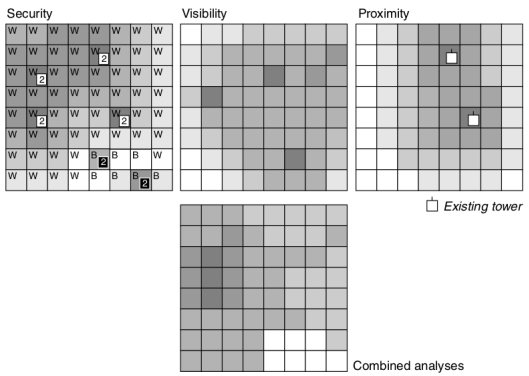
Example: RTS game where the placement of radar towers is critical to success
Relevant properties:

- Wide range of visibility (to get the maximum information)
- In a well-secured location (towers are typically easy to destroy)
- Far from other radar towers (no point duplicating effort)

Combination:

$$\text{Quality} = \text{Security} \times \text{Visibility} \times \text{Distance}$$

$$\text{Quality} = \frac{\text{Security} \times \text{Visibility}}{\text{Tower Influence}}$$



Map flooding

Map flooding: calculates Dirichlet domains on tile-based levels. Which tile locations are closer to a given location than any other

Example: a location in the game belongs to the player who has the nearest city to that location
- each city has a strength
- the region of a city's influence extends out in a continuous area.
- cities have maximum radius of influence that depends on the city's strength.

Dijkstra algorithm

- open list: set of city locations.
- labels: controlling city + strength of influence
- process location with greatest strength
- processing: calculate the strength of influence for each location's neighbor for just the city recorded in the current node.
- updates: highest strength wins, and the city and strength are set accordingly; processed locations moved in closed list; changed strength moved to open list.

Convolution Filters

- update matrix
- size of the filter: number of neighbors in each direction.
- new value by multiplying each value in the map by the corresponding value in the matrix and summing the results.
- two copies of the map. One to read and one to write.
- If the sum total of all the elements in our matrix is one, then the values in the map will eventually settle down and not change over additional iterations.
- processing several locations at the same time (SIMD)
- in games we limit the number of pass (one per frame) for speed
- **boundaries**: adapt the matrix but matrix switching is not good.
  adapt the map adding margin numbers that are updated as well

Gaussian blur

smooths out differences

elements of the binomial series

$$
\begin{bmatrix} 1 & 2 & 1 \end{bmatrix}
$$

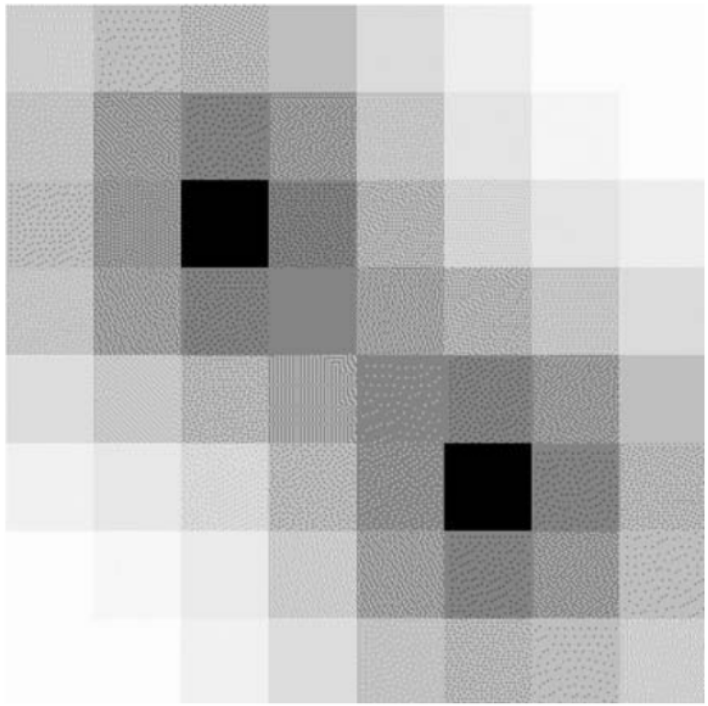
$$
\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \end{bmatrix}
$$

$$
\begin{bmatrix} 1 & 6 & 15 & 20 & 15 & 6 & 1 \end{bmatrix}
$$

$$
\begin{bmatrix} 1 & 8 & 28 & 56 & 70 & 56 & 28 & 8 & 1 \end{bmatrix}.
$$

$$
\begin{bmatrix} 1 \\ 4 \\ 6 \\ 4 \\ 1 \end{bmatrix} \times \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}
$$

$$
M = \frac{1}{256} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}
$$



Separable: first only vertical and then only horizontal vector

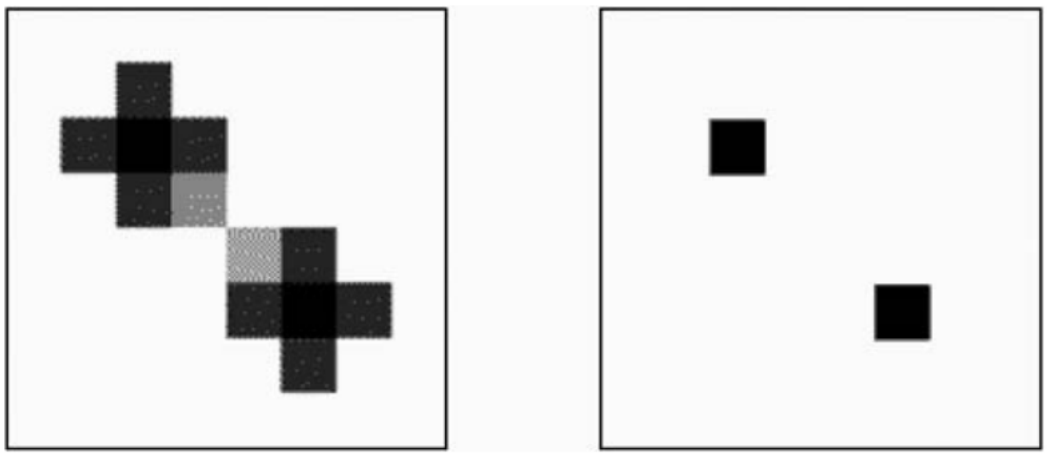
Evaporation at each iteration. The total removed influence is the same as the total influence added

Sharpening filter

concentrates the values in the regions that already have the most

central value will be positive, and those surrounding it will be negative

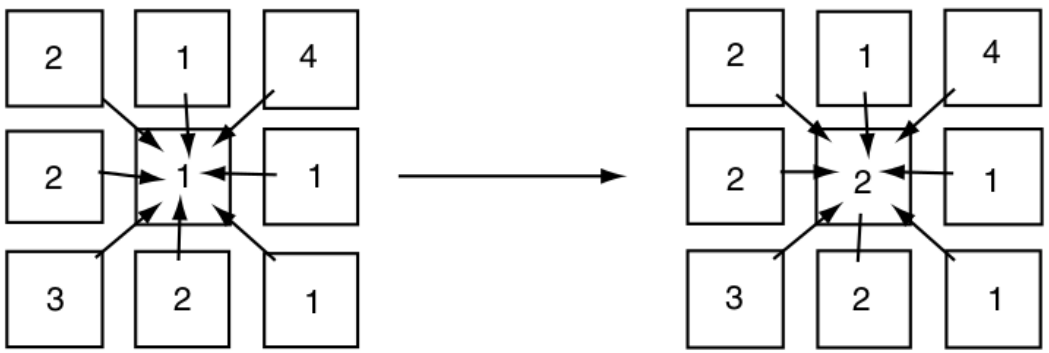
$$\frac{1}{a}\begin{bmatrix} -b & -c & -b \\ -c & a(4b+4c+1) & -c \\ -b & -c & -b \end{bmatrix}$$



Never run to steady state

Cellular Automata

- update rules generate the value at one location in the map based on values of other surrounding locations (eg. Conway's"The Game of Life")
- at each iteration values are calculated based on the surrounding values at the previous iteration.
- values in each surrounding location are first split into discrete categories
- update for one location depends only on the value of locations at the previous iteration $\rightsquigarrow$ need two copies of the tactical analysis



IF two or more neighbors with higher values,
THEN increment

IF no neighbors with as high a value,
THEN decrement

Rules

- output category, based on the numbers of its neighbors in each location (If two neighboring locations change their category based on each other, then the changes can oscillate backward and forward)
- in other applications maps are considered to be either infinite or toroidal but not in games
- rules that are based on larger neighborhoods (not just locations that touch the location in question) and proportions rather than absolute numbers.
  Eg: if at least 25% of neighboring locations are high-crime areas then a location is also high crime
- In most cases, rules are heavily branched: lots of switch or **if** statements, which are not easily parallelized
- rule of thumb to avoid dynamism: set only one threshold

Applications

Areas of Security
A location is secure if at least four of its eight neighbors (or 50% for edges) are secure

Building a City
way buildings change depending on their neighborhood
a building appears on an empty patch of land when it has one square containing water and one square containing trees
Taller buildings come into being on squares that border two buildings of the next smaller size, or three buildings of one size smaller, or four buildings of one size smaller still.
buildings are not removed
In RTS reasoning on a flow of resources

Resume

1. Decision Making
   - Scripting
   - Action Management

2. Tactical and Strategic AI
   - Waypoint Tactics
   - Tactical Analysis