CMSC 498M: Chapter 8
Artificial Intelligence for Games

Reading:
- Artificial Intelligence: Agents, Architecture and Techniques, by S. Rabin (Chapt. 5.3 in "Introduction to Game Development" by S. Rabin)
- Gamasutra tutorial: http://www.gamasutra.com/features/19990212/sm_01.htm
- Boids page (http://www.red3d.com/cwr/boids/) by Craig Reynolds

Overview:
- Basic AI concepts
- Motion
- Planning and coordination

What is AI?

What is Artificial Intelligence?
- "The study of computational systems that exhibit intelligence"
- Theories and computational models

What is Intelligence?
- "It's whatever people do"
  - Leads to the "Turing Test": If a person can't distinguish its behavior from a human, then it is intelligent
  - This is not rigorous—"I know intelligence when I see it"

Behaving in a rational manner:
- Use available knowledge to maximize goal achievement
- Often leads to optimization techniques

Demonstrating complex capabilities:
- Problem solving
- Learning
- Planning, ...
Roles of AI in Computer Games

Roles of Game AI: Complex behaviors not specified by player
- Nonplayer Opponents: Realistic attach behavior
- Nonplayer Teammates: Coordinated supportive behavior
- Support and Autonomous Characters: Crowd/flock behavior
- Commentators/Instruction: Does the player need a hint?
- Camera Control: Finding the best viewpoint

What AI is not: There is not a clear distinction, but...
- Determined by physical laws: E.g., path of a tennis ball
- Purely random: E.g., which block falls next in Tetris
- Direct response to game rules/user inputs: E.g., shoot gun when space-bar hit, scripted instruction sequences

AI versus Animation

AI System: AI determines what to do and the animation does it
- AI drives animation: AI system decides what action to take. Animation renders the result.
- Scenario 1: AI system issues orders: "move from A to B" and animation system does the rest
- Scenario 2: AI system controls everything, down to the animation clip to play

Which scenario?
- Depends on the nature of the AI system and the nature of the animation system
- Is the animation system based on motion capture, or physics, or something else?
- Does the AI perform collision avoidance? Does it do detailed planning?

Chapter 8, Slide 3
Copyright © David Mount and Amitabh Varshney
Properties of a Good AI System

Goals:

- **Goal driven**: AI system decides what to do, and then figures out how to do it.
- **Responsive**: AI system responds immediately to changes in the world.
- **Smart**: (knowledge intensive) AI system knows a lot about the world and how it behaves, and uses this knowledge in its own behavior.
- **Consistent**: Embodies a believable, consistent character.
- **Efficient**: Low CPU and memory usage.
- **Practical**: Fast and easy development.

Tradeoffs:

- Unfortunately, many of these goals conflict fundamentally.

Part I - Basic AI Concepts

- **AI Basics**
- **Agents**
- **Rule-based Approaches**
- **Finite-State Machines**
General Structure: AI Update Step

Sensing:
- Determines the state of the world
- May be simple: State changes all come by message passing
- Or complex: Figure out what is visible, where your team is, etc.

Thinking:
- What to do next, given world state/goals
- The core of the AI system

Acting:
- Tells the animation system what to do
- Translate low-level goals into specific motion (velocities, joint-angles)
- Can be messy/tedious

General Structure: Polling or Event Driven?

Polling:
- The AI gets called at fixed time intervals
- Senses: Look to see what has changed in the world. For instance:
  - Queries what it can see
  - Acts on it

Event Driven:
- Acts in response to world events
  - Events sent by messages (e.g. invoke a callback function)
- Examples:
  - You heard a sound
  - A threat just entered your field of view
  - You have just been hit

Real systems are a mix:
- Something just changed (event)...so do some sensing (polling)
AI Techniques in Games

Basic problem:
- Given the state of the world, what should the agent do next?

Range of (real-time) solutions in games:
- Rule-based systems
- Finite-state machines
- Decision trees
- Neural networks
- Fuzzy logic

Wider range of solutions in the academic world:
- Planning systems
- Logic programming
- Genetic algorithms
- Bayes nets
- Currently, too slow for games (but this may change in time)

Two Measures of Complexity

Execution Complexity:
- How fast does it run as more knowledge is added?
- How much memory is required as more knowledge is added?
- Determines the run-time cost of the AI

Specification Complexity:
- How hard is it to write the code?
- As more knowledge is added, how much more code needs to be added?
- Determines the development cost, and risk
Part I - Basic AI Concepts

- AI Basics
- Agents
- Rule-based Approaches
- Finite-State Machines

Agents: Goals of an Opponent

Provide a challenging (but flawed) opponent:
- Should be beatable (but not too easily)
- Should be entertaining and fun

Not too challenging:
- Should not be superhuman in accuracy, precision, sensing, ...

No unintended weaknesses:
- No "golden path" to defeating opponent every time
- Must not fail miserably or look dumb (e.g., getting lost)

Not be too predictable:
- Through randomness
- Through multiple, fine-grained responses
- Through adaptation and learning
Agents: What are They?

Definition:

"An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future." (Stan Franklin and Art Graesser)

Structure of Agent Action:

- **Sensing**: perceive features of the environment
- **Thinking**: decide what action to take to achieve its goals, given the current situation and its knowledge
- **Acting**: doing things in the world

Issues:

- Thinking can make up for limitations in sensing and acting (e.g., extrapolating future motion)
- The more accurate the models of sensing and acting, the more realistic the behavior

Agents: Sensing Limitations & Complexities

**Sensing Issues**:

- **Limited sensor distance/field of view**: Can sense local environment
- **Obstacles**: Agent cannot see through walls
- **Improving the view**: Detecting and computing paths to doors
- **Realistic behavior**: Sensitivity to motion, fooled by camouflage
- **Reaction time**: Sensed information is not processed instantly

**Random noise in sensors**:

- Mimicking human limitations

**Different Sensors**:

- **Sound**: Omni-directional, gives direction, distances, speech, ...
- **Vision**: Limited field of view, 2.5D, color, texture, motion, ...
- **Smell**: Omni-directional, chemical makeup

**Integration**:

- Need to integrate different sources to build complete picture
Agents: Simple Action Strategies

Random motion: ("unintelligent" AI)
- Roll the dice to select when and where to move

Regular pattern:
- Follow invisible tracks: E.g., Galaxian

Tracking/Pursuit Strategies:
- Direct Pursuit: Move toward prey’s position
  (E.g., heat-seeking missile)
- Lead Pursuit: Move toward a spot ahead of prey
- Interception: Move to where prey is expected to be

Evasion Strategies:
- Direct Evasion: Move directly away from (slow) pursuer
- Side-step: Head perpendicular to (fast) pursuer’s current bearing
- Weave: Every N seconds move X degree off pursuer’s bearing

Part I - Basic AI Concepts

- AI Basics
- Agents
- Rule-based Approaches
- Finite-State Machines
Rule-based approaches

Rule-Based Systems: Popular because:
- Familiar
- Predictable and, hence, testable
- Many game designers don’t know much about AI :-)

Rules specify the action depending on circumstances:
- **Deterministic**: One action for each situation
- **Random**: Multiple possible actions in the same situation
- **Weighted**: Certain actions have a higher probability

**Based on State/Environment Information:**
Take the direction that leads you to the opponent

---

**Example:**

- A simple rule-based system for moving a ghost agent in **Pac-Man**

<table>
<thead>
<tr>
<th>Ahead</th>
<th>Right</th>
<th>Left</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open</td>
<td>—</td>
<td>—</td>
<td>Go ahead</td>
</tr>
<tr>
<td>Blocked</td>
<td>Open</td>
<td>—</td>
<td>Turn Right</td>
</tr>
<tr>
<td>Blocked</td>
<td>Blocked</td>
<td>Open</td>
<td>Turn Left</td>
</tr>
<tr>
<td>Blocked</td>
<td>Blocked</td>
<td>Blocked</td>
<td>Go backwards</td>
</tr>
</tbody>
</table>

**Note:**
- Easy to add randomness, if we like
- Does not consider the location/state of the player
Part I - Basic AI Concepts

- AI Basics
- Agents
- Rule-based Approaches
- **Finite-State Machines**

---

**Finite-state machines (FSMs)**

**Finite-State Machines:**
- A (finite) set of possible states that the agent can be in
- Connected by transitions that are triggered by a events or changes in the world or user input
- Normally represented as a directed graph, with the edges labeled with the transition event

**Ubiquitous:**
- Almost all computer game AI systems support FSMs

**Déjà vu?**
- FSMs are used in language processing (regular languages)
- You might have seen them in language theory and compilers

**Issues:**
- Provide a clean graphical way to describe state transitions/actions
- Uniform table/graph structure
Quake-Bot Example

Types of behavior to capture:
- If you don’t see an enemy, wander randomly
- When see enemy, attack
- When hear an enemy, chase enemy
- On dying, re-spawn
- Extras: If health is low and seeing an enemy, retreat

Finite-state machines (FSMs)

Actions: On entering or exiting a state we invoke a callback function

Events:
E=Enemy Seen
S=Sound Heard
D=Die

Problem: There is no transition from Attack to Chase
Finite-state machines (FSMs)

Possible Fix: Create an extra attack state if sound is heard

Problem: This can result in the proliferation of a huge number of states

Solution: Rather than creating new states, make limited use of additional variables

Events:
E=Enemy Seen
S=Sound Heard
D=Die

Hierarchical FSMs

Expand complex states into their own FSM

Use a stack to save current state
**Nondeterministic FSMs**

- Add multiple transitions and associated probabilities

```
Start

Aim & Slide Left & Shoot

Aim & Jump & Shoot

Approach

Attack

No enemy

Wander

Aim & Slide Right & Shoot

.3

.3

.3

.4

.4

Die

Start
```

These are essentially Markov chains

**Efficient FSM Implementation**

**Basic Implementation:**
- Compile into an array indexed by [state-name, event]
- Transition: [state-name]_{i+1} \leftarrow \text{array}[\text{state-name}_i, \text{event}]
- Switch based on state-name to call execution logic

**Hierarchical FSMs:**
- Create array for every FSM
- Have stack of states:
  - Classify events according to stack
  - Update state which is sensitive to current event

**Nondeterministic FSMs:**
- Have array of possible transitions for every (state-name, event) pair
- Choose one at random, based on transition probabilities
Part II – Motion

- Motion Planning
- Configuration spaces
- Waypoints
- Path planning algorithms
- Navigation Interfaces

### Motion Planning

**Motion planning**: Move an agent from one position to another
- Avoiding (fixed) obstacles
- Avoiding other moving objects
- **Goals**: Smooth, compact, natural motion. Robustness

**Motion planning in games**:
- **First-Person Shooter**: How does the AI get from room to room?
- **Real-Time Strategy**: User directs an agent to go somewhere. How do they get there?
- **Many others**: Sports, racing, ...

**Computational issues**:
- Coordinating the motion of multiple entities with many degrees of freedom is computationally very hard
Motion Planning

**Academic Formulation:** Given a start point \( s \) and a destination point \( t \), find a path from \( s \) to \( t \) that avoids obstacles

- **Minimize cost:** Distance, travel time, ...
  - Travel time may depend on terrain, for instance
- **Dynamic environments:** Paths being blocked or removed
- **Limited knowledge:** No map; just what sensors tell us

**Game Formulation:** Find a reasonable path from \( s \) to \( t \)

- **Need not be optimal:** Not shortest, but not too long
- **Need not be perfect:**
  - OK to briefly intersect obstacles
  - OK to backtrack sometimes
- **Dynamic environments:** But you usually have complete knowledge of the current environment

---

**Part II – Motion**

- Motion Planning
- **Configuration spaces**
- Waypoints
- Path planning algorithms
- Navigation Interfaces
Configuration Space: Simple Example

Point motion planning:
- Much easier to plan the motion of a point than a geometric object
- Can we reduce object motion planning to point motion planning?

Simple example:
- We want to plan the motion of a circular disk of radius \( r \) from starting point \( s \) to destination point \( t \)
- Can we convert this into a motion planning problem for a point?

Equivalent problem:
- Grow each obstacle \( o_i \) by radius \( r \). Call the grown obstacle \( o'_i \).
- Shrink the disk down to a point
- Plan the motion of a point from \( s \) to \( t \)

Why this works?
A disk centered at a point \( p \) intersects obstacle \( o \)
\( \iff \) \( p \) is within distance \( r \) of \( o \)
\( \iff \) \( p \) intersects the grown obstacle \( o' \)

Can we generalize this idea to more complex objects?
Configuration Space: Basics

General motion planning:
- How can we reduce the motion of an arbitrary geometric object to point motion planning?

Configuration vector:
- The exact position of each moving entity can be described by a vector of numbers, e.g.:
  - Position of its center of mass: \((x, y, z)\)
  - 3D rotational orientation (e.g. using Euler angles): \((\varphi, \theta, \psi)\)
  - Joint angles: \((\theta_1, \theta_2, \theta_3, \ldots, \theta_k)\)
- Thus, the position of an object can be modeled as a multi-dimensional configuration vector: \((x, y, z, \varphi, \theta, \psi, \theta_1, \theta_2, \theta_3, \ldots, \theta_k)\)

Configuration space:
- Can model location of an object in 3-space as a point \((6+k)\)-space

Configuration Space: Free/Forbidden Placement

Free or Forbidden: Each point in configuration space is either:
- Free: Intersects no obstacles
- Forbidden: Intersects some obstacle

Example: A 3x2 rectangle in 2D that translates and rotates:
- Center of Mass Position: \((x, y)\)
- Rotation: \(\theta\)
- Configuration point (3D): \((x, y, \theta)\)

Suppose that we require the rectangle to lie above the line \(y = 0\)

\[(4, 3, 0):\ Free\]
\[(8, 1, 45^\circ):\ Forbidden\]
Configuration Space: Configuration Obstacle

Configuration obstacles:
The set of all forbidden points in configuration space defines a forbidden region. These regions are called configuration obstacles.

Equivalence:
- Computing the motion of an object in "regular space" from one free placement to another is equivalent to...
- Computing the motion of a point in configuration space, from one free placement to another

![Configuration obstacles diagram](image1)

Configuration Space: Issues

Further issues:
- **Configuration distance**: Is not Euclidean because configuration coordinates are generally not homogeneous (E.g., x-translation in meters, rotation in radians)
- **Curse of Dimensionality**: Computational complexity increases exponentially with the dimension of the configuration space
- **Coordinated Motion**: Can be done by concatenating the configuration vectors of all moving objects. Very hard!!!

Reducing complexity:
- **Constrained motion**: e.g., no rotation
- **Simplify models**: e.g., bounding boxes
- **Prioritized planning**: e.g., plan object 1's motion first, then object 2, etc.
Part II - Motion

- Motion Planning
- Configuration spaces
  - Waypoints
- Path planning algorithms
- Navigation Interfaces

Path Planning

Point path planning:
- Henceforth we assume (by reduction to configuration space) that we need only plan motion of a point

Common methods:
- Discrete (graph-based) search algorithms:
  - BFS, Dijkstra's algorithm, A* search, ...
- Potential fields:
  - Put a "force field" around obstacles, and follow the "potential valleys"
- Pre-computed plans with dynamic re-planning:
  - Pre-compute answer and modify as required
- Special purpose algorithms:
  - E.g., Given a fixed start point, fast ways to find paths around polygonal obstacles
Discrete (Graph-Based) Algorithms

Curved surfaces and curved paths:
- Generally obstacles (especially in configuration space) are curved
- Generally optimal paths are curved (Imagine running around a corner)

Discretize the search space:
- Restrict the start and goal points to a finite set of waypoints
- Restrict the path to lie on line segments connecting waypoints

Graph search:
- Waypoints are nodes
- Connecting segments are the edges
- Distance/cost modeled as edge weights
- Search reduces to the well studied problem of computing shortest paths in a graph

Waypoints

Waypoint:
- In navigation this is a mapped reference point used for navigation. (E.g. a buoy or radio beacon)

Where to put waypoints?
- There are many possibilities, as we shall see

How to connect them?
- Depends on what paths you are willing to accept - almost always assume straight lines (e.g., along line of sight)

Answers depend on the game:
- Environment: open fields, enclosed rooms, etc...
- Game Style: covert hunting, open warfare, friendly romp, ...
Where would you place Waypoints?

**Room Structured**

**Open Terrain**

Strategies for Placing Waypoints

**By hand:**
- Best control, but most time consuming

**Heuristics:**
- In doorways, because characters have to go through doors and straight lines joining rooms always go through doors
- Along walls, for characters seeking cover
- At other discontinuities in the environments (edges of rivers, for example)
- At corners, because shortest paths go through corners

**Importance of good waypoints:**
- Good waypoint selection results in more natural, varied motion
- Automatic obstacle avoidance (by providing sufficient clearance)
**Grid-Based Placement**

**Grid-based waypoints:**
- Place a grid over the world
- Put a waypoint at every grid point that is obstacle-free

**Joining waypoints:**
- Join waypoints if the segment between them hits no obstacle
- Only check immediate neighboring grid points
  E.g., 4-neighbors (N,E,S,W) or diagonal 8-neighbors

**Pros & Cons:**
- Easy to implement
- Insensitive to variations in resolution
- Problems with tight spaces

---

**Polygon-Based Placement**

**Polygon-based waypoints:**
- Choose waypoints based on the floor polygons in your world
- Further subdivide large/complex polygons into smaller, convex regions for more complete waypoint placement

**Where to place waypoints?**
- There are two obvious options (next slide)
### Polygon-based Placement

**Bad news:** May not be connected by line of sight

- **Waypoints within faces**
- **Waypoints along edges**

### Waypoints around Obstacles

**Obstacles:**
- Place waypoints a small distance from obstacle corners
- Nearby waypoints are connected if they are mutually visible

**Pros & Cons:**
- Tends to produce the shortest paths
- Motion "scraping" along walls is not very natural. Real people tend to find maximum clearance paths
Getting On and Getting Off

Problem:
- Characters may not start and end at waypoints

Solution:
- When the character starts, find the closest visible waypoint and move to that first

Better:
- Find the visible waypoint closest to the direction you think you need to go

Even Better:
- Try all visible waypoints and see which gives the shortest path, most direct path

Best option:
- Add a new, temporary waypoint at the precise start and goal point, and join it to nearby waypoints. Then invoke path planning algorithm.

Chapter 8, Slide 47
Copyright © David Mount and Amitabh Varshney

Getting On and Getting Off

Chapter 8, Slide 48
Copyright © David Mount and Amitabh Varshney
Part II – Motion

- Motion Planning
- Configuration spaces
- Waypoints
- Path planning algorithms
- Navigation Interfaces

Search Algorithms

Generic search algorithm: From source s to destination t

- Initialization:
  - Mark s as discovered, and mark all other nodes as undiscovered
  - \( \text{dist}[s] \leftarrow 0 \), and \( \text{dist}[u] \leftarrow \infty \) for all other nodes
  - Create a priority queue containing only s
- Repeat until reaching t:
  - Extract: Remove the node \( u \) of highest priority from the priority queue
  - Process: For each unfinished neighbor \( v \) of \( u \):
    - \( \text{dist}[v] \leftarrow \min( \text{dist}[v], \text{dist}[u] + \text{weight}(u, v) ) \)
    - Mark \( v \) as discovered and add to/update priority queue
  - Finish: Mark \( u \) as finished

Algorithm invariant: At any time there are three types of nodes:
- Undiscovered: Haven’t seen this node yet
- Discovered: A neighbor has seen you. You are in the queue.
- Finished: You have completed processing. dist value will not change.
Search Algorithms

Obtaining the shortest path:
- Store a pointer for each node $u$ to the previous node along the shortest path to $u$
- Update pointers as nodes are added to the visited set and as nodes are processed (whenever the dist changes)
- To find path to goal, trace pointers back from goal nodes

Making the generic algorithm concrete:
- The only element that has not been spelled out is how priorities are assigned to the discovered but unfinished nodes
- The choice of priority affects the correctness and efficiency of the search algorithm

Common Search Algorithms:
- Breadth-First Search and Dijkstra’s Algorithm
- Best-first (Local Greedy) Search
- A* Search

Dijkstra’s Algorithm

Dijkstra’s Algorithm:
- The priority of node $u$ is the distance estimate, $\text{dist}[u]$
- The unvisited node $u$ with the smallest $\text{dist}[u]$ is processed next
- Under the (reasonable) assumption that edge weights are non-negative, this correctly finds the shortest path
- The order in which vertices are visited is independent of the location of the destination node

Breadth-First Search:
- When there are no edge weights, can replace the priority queue with a simple FIFO queue
Best-First Search: A Greedy Heuristic

Best-First Search:
- The priority of a node is its Euclidean distance from $t$
- Next node to process is the discovered node that is closest to $t$
- This heads straight to $t$, and so is potentially much faster than Dijkstra's algorithm or Breadth-First Search

Bad news:
- This heuristic may fail when obstacles are present

Best-First Search: Getting Caught

Best-First Search vs Dijkstra:
- When obstacles are present, best-first search may get trapped, and will need to backtrack. The result is a suboptimal path.
- Dijkstra's algorithm visits more nodes, but gets the correct path
A* Algorithm

- Combines the best aspects of Best-First Search and Dijkstra
- The priority of a node is the sum of two functions:
  \[ f(u) = g(u) + h(u), \]
  where:
  \[ g(u) = \text{dist}[u] \] (current distant estimate to u)
  \[ h(u) = ||u - t|| \] (Euclidean distance from u to t)
- Like Best-First, it seeks the destination first
- Like Dijkstra, it computes the correct shortest path

Key:
- \( h(u) \) should be a lower bound on the remaining distance to \( t \)

This is the algorithm of choice

Summary of Graph-Based Search Methods

Summary:

Dijkstra:
- Always correct
- Does not seek the destination
- Efficient in the worst-case, but not on average

Breadth-First Search:
- Simplified version of Dijkstra for when edges have uniform weight
  (Cost of a path is number of edges on the path)
- As with Dijkstra, does not seek the destination

Best-First Search:
- Greedily seeks the destination
- Very fast when few objects are present
- May not produce the shortest path

A* Search:
- Always correct
- Seeks the destination, and so is efficient on average
- Method of choice for game programming
Part II – Motion

- Motion Planning
- Configuration spaces
- Waypoints
- Path planning algorithms
- Navigation Interfaces

Navigation Interfaces

Motivation:
- Agent AI has a certain level of complexity in its objectives and goals
- Creating intelligent movement involves transmitting these objectives and goals to the navigation system

Navigation Interface:
- Provides a structure for conveying information about agent objectives and goals

Design Choices:
- Tradeoffs: Choosing an interface involves making a compromise between focus and flexibility
- Highly focused: Can be better tuned to a specific problem
- Flexible: Can have the expressiveness to handle a wider variety of scenarios
Navigation Interfaces

Levels of Expressiveness:
- Single Pair: (Lowest level)
  - A origin point and destination point are given
  - Return the lowest cost path between them
- Weighted Destinations:
  - Instead of a single destination, provide multiple goals, each with its own reward
  - Optimal path maximizes the net pay-off, (reward – cost)
- Goals: (Highest level)
  - Specify the movement in terms of the agent’s goals
  - May not involve spatial specifications (e.g., find armor or a weapon)

AI Paradigms for Movement:
- Reactive Behaviors:
  - Maps sensory input directly to motion (e.g., Danger! Run away!)
  - Fine for simple situations: obstacle avoidance, seeking, fleeing
  - Cannot handle traps or complex environments (e.g., running into a closet)
- Deliberative Planning:
  - Provably optimal paths (e.g., Dijkstra or A* search)
  - Higher-level of spatial intelligence
  - Plans based on an environmental model
- Hybrid Systems:
  - Combine "common sense" of reactive behaviors with "intelligence" of planning
Navigation Interfaces: Implementation

Reactive Behaviors:
- Steering behaviors based on explicit mathematical formulas
- Two problems:
  - Integration of multiple behaviors
  - Realism
- Prioritize behaviors and use fuzzy logic to simulate realism

Deliberative Planning:
- Not necessarily a search. Other techniques can be more efficient.
  - Precompute everything (like look-up tables)
  - Reactive approximation techniques to build near optimal paths without a search
- Dynamic environments make things tricky
  - Trigger a re-plan when the conditions have changed sufficiently
- "Quality of service" algorithms

Part III - Planning and Coordination

- Dynamic planning
- Potential-based path planning
- Flocking
- Particle systems
Dynamic Path Planning

Dynamic path planning: What happens when the environment changes after the plan has been made?
- The player alters the environment (e.g., blows up a bridge)
- Other agents get in the way
- Moving objects (e.g., falling debris) blocks the way

Possible approaches:
- Try to avoid the problem. (e.g. control your agents/world better)
- Re-plan when something goes wrong
- Reactive planning

Avoiding Plan Changes

Partial planning: Only plan short segments of path at a time
- Stop A* after a path of some length is found, even if the goal is not reached - use best estimated path found so far
  - Extreme: Use greedy search and only plan one step at a time
  - Common: Hierarchical planning and only plan low level when needed
- Underlying idea is that a short path is less likely to change than a long path
- Optimality will be sacrificed
- Side Benefit: more uniform frame rates

Re-Planning: If your plan has gone wrong, create a new one,
- Assumes that the dynamic changes are now permanent,
- Usually used in conjunction with avoidance strategies:
  - Re-planning is expensive, so try to avoid having to do it
  - No point in generating a plan that will be redone
Part III – Planning and Coordination

- Dynamic planning
- **Potential-based path planning**
- Flocking
- Particle systems

---

**Reactive Planning**

**Reactive Agent:**
- Plans one step at a time
- Use only local information (not globally optimal)

**Example:** Potential-field path planning
- Set up a repulsive field around obstacles (and other agents)
- Set up an attractive field toward the goal
- The agent follows the gradient downhill to the goal, while the force field pushes it away from obstacles
- Can also model velocity and momentum – potential field defines a force

**Why is this reactive?**
- Agent’s behavior depends only the local gradient at any instant

Widely used in low-level robotics navigation
### Potential Field Navigation

**Key:**
- **Red:** Start point
- **Blue:** Goal point
- **Green:** Obstacles

**Field Strength:** (Shown in gray scale)
- **Dark:** Attractive
- **Light:** Repulsive

**Potential Function:** is the sum of:
- **Repulsive force** around the obstacles (including the world boundary)
- **Attractive force** at the goal point

---

### Path Planning with Potential Fields

**Potential-Field Navigation:**

- **Intuition:** Moving in 2D space, the potential field is a terrain. It is hard to move uphill, and easy to slide downhill.

- **Path Plan:** Agent follows laws of physics and rolls downhill

**More Formally:** Your agent is moving subject to a **potential field** \( U(p) \), that is defined at every point \( p \) of the environment. \( U(p) \) defines the **strength** of the field (height of the terrain) at point \( p \).

**Gradient:** Is a vector that points in the direction of **steepest descent**:

\[
-\nabla U(p) = \left( -\frac{\partial U}{\partial x}, -\frac{\partial U}{\partial y} \right)
\]

**Force:** The gradient behaves like a **force vector** \( F(p) = -\nabla U(p) \), impelling the agent to steer downhill.
Defining the Fields

Attraction towards Goal:

Naïve: Simple linear potential based on the distance from the goal:

\[ U_{goal}(p) = c \| p - p_{goal} \| \]

Where \( c \) is an adjustment parameter. Potential function is a cone. The gradient is constant vector directed towards the goal.

Smarter: Use a parabolic well potential:

\[ U_{goal}(p) = \frac{c}{2} \| p - p_{goal} \|^2 \]

The gradient still points towards the goal, but the magnitude of the gradient vector increases linearly with the distance from the goal.

\[ \nabla U_{goal}(p) = \frac{c}{2} \| p - p_{goal} \| \nabla \| p - p_{goal} \| = \frac{c}{2} \| p - p_{goal} \| \nabla (\| p - p_{goal} \|) \]

\[ = c \| p - p_{goal} \| (\nabla \| p - p_{goal} \|) \]

Defining the Fields

Repulsion from Obstacles:

- Obstacle \( i \) contributes a field strength based on the distance from its boundary: \( U_i(p) = f(|p - \text{obst}_i|) \), where \( f \) is an decreasing function of distance.

- Typical potential functions: (\( c \) is adjustment parameter)
  - Quadratic/Polynomial: \( f(d) = \frac{c}{d^2} \), or generally \( f(d) = \frac{c}{d^p} \)
  - Negative Exponential: \( f(d) = c \cdot e^{-d} \)

- Taper so that field at some distance is zero. Why?

- Strength determines how likely the agent is to avoid it

Total Field Strength: Add the sub-fields together

(Does this remind you of a modeling technique we discussed?)
Following the Field

**Steepest Descent:**
- At each step, the agent needs to know which direction is "downhill"

**Total field gradient:**
- Compute: Gradients of each sub-field and sum
- Gradient: The vector of partial derivatives in x, y, and z

**Gradient induces a Force:** (and hence, acceleration)
- Automatically avoids sharp turns and provides smooth motion
- Higher mass can make large objects turn more slowly
- Easy to make frame-rate independent
- But, high velocities can cause collisions if field is not strong enough to turn the object in time
- One solution is to limit velocity - want to do this anyway because the field is only a guide, not a true force

---

Following the Field: Examples

**No momentum** - Go in direction of lowest field strength

**Momentum** - but with linear obstacle field strength and moved goal
Discrete Approximation

Discretization: Compute the field on a grid
- Can precompute fixed sub-fields, e.g., for fixed obstacles
- Sub-fields for moving obstacles computed with each time step

Flow:
- Go to neighboring node with smallest field value

Pros & Cons:
- Faster
- Requires more space for grid
- Only approximate

Potential Problems with Potential Fields

Requires lots of Tuning:
- Strength of the field around each obstacle
- Function for field strength around obstacle (quadratic, exp, etc)
- Steepness of force toward the goal
- Maximum velocity limit

Goals can conflict:
  - High field strength: Avoids collisions, but produces big forces and hence jerky motion
  - Low field strength: Smoother paths, but increases likelihood of collisions due to overshooting

Local minima:
- Cannot see the “big picture.” All decisions are purely local.
- May be arbitrarily far from optimal.
- Easy to get trapped in concavities
Potential-Field Problems: Examples

Field too weak:
Obstacle collision

Field too strong:
Jerky motion

Getting stuck in local minima
Local Minima Problem

**Problem Source:**
- Path planning is an optimization problem, typically with many local minima.
- Potential field planning is a form of gradient descent optimization.
- Gradient descent methods seek local minima, and will get stuck unless other techniques are applied.

**Possible Solutions:**
- **Virtual Obstacles**: Created at the point where the search gets stuck.
- **Virtual Sub-Goals**: Backtrack along the path and attract it in a new direction.
- **Randomized Potential Field**: Add a random noise to potential field values and try again. (Mixture of random walk and potential field.)

---

Part III - Planning and Coordination

- Dynamic planning
- Potential-based path planning
- **Flocking**
- Particle systems
Flocking

Motion of multiple agents: Objectives:
- Collision avoidance
- Cohesion:
  - Tendency to stick together
  - Examples: School of fish, flock of birds, herd of cattle
- Common goal?
  - Yes → Group should seek a common objective: A squadron of soldiers
  - No → Motion is independent: People walking in a crowded street

Emergent behavior:
- Define simple rules on individuals based on purely local information
- Each rule induces a force. Total force affects acceleration.
- Interesting global behavior naturally emerges as a result

Flocking Rules

Boids: (virtual birds)
- Term coined by Craig Reynolds (1986)
- Behavior determined by the following 4 rules
  Separation:
  - Avoid collisions with nearby boids
  - E.g., each boid generates a repulsive potential field of limited radius
  Alignment:
  - Align flight direction with neighboring boids
  Cohesion:
  - Attraction to centroid (center of mass) of the flock
    E.g., flock centroid generates an attractive potential field
  Avoidance:
  - Avoid obstacles in the environment
    E.g., each obstacles generates a repulsive potential field
Flocking Rules

Separation:
Steer away from neighbors that are too close

Alignment:
Steer towards average heading

Cohesion:
Steer towards flock centroid

Avoidance:
Steer away from obstacles

Combining Commands

Steering Rule:
- Steering rules act as forces; alter acceleration
- Set a maximum acceleration to avoid jerky behavior

Tuning Behavior:
- Assign a weight to each of the flocking rules
- Avoidance should be high
- Cohesion should be lower

Option 1: Apply rules in decreasing weight order, until max acceleration is reached
  Responsive: Ensures that high priority forces are applied

Option 2: Take weighted sum and truncate to max acceleration
  Fair: Ensures that all forces affect final motion

Demo: http://www.red3d.com/cwr/boids/
Flocking Evaluation

**Advantages:**
- **Simple:** Complex behavior from simple rules
- **Flexible:** Many varieties of behavior from a small set of different rules and varying parameter settings

**Disadvantages:**
- **Tuning:** Can be difficult to set parameters to achieve desired result
- **Local Minima:** All the problems of potential fields regarding strength of forces

Part III - Planning and Coordination

- Dynamic planning
- Potential-based path planning
- Flocking
- **Particle systems**
General Particle Systems

Particle system:
- A variation on the theme of flocking
- Particles are light-weight objects: E.g., point masses
- Simple rules control how particles:
  - are born
  - move, grow/shrink, change appearance
  - die

Flexible: Can model many complex, distributed phenomena
- Fireworks
- Waterfalls, spray, foam
- Explosions (smoke, flame, chunks of debris)
- Clouds
- Crowds, flocks, herds

Widely used in movies and games

Particle System Demos

Demo source: Flow particle animation application by Mark B. Allan.
http://users.rcn.com/mba.dnai/software/flow/

Demos:
- Kettle
- Face
- Smoke Ball
- Lava2
- Molten
Particle System Steps

1. **Inject any new particles** into the system and assign them their **initial attributes**
   - There may be multiple sources
   - Particles might be generated at random (clouds), in a constant stream (waterfall), or according to a script (fireworks)

2. **Remove** (kill) particles that have exceeded their **lifetime**
   - May have a fixed lifetime, or die on some condition

3. **Move** all the current particles according to their **scripts**
   - Script typically refers to the neighboring particles and the environment

4. **Render** all the current particles
   - Many options for rendering

---

Particle System Example: Puffs of Smoke

**Birth:**
- Particles are spawned at a constant rate
- They have zero initial velocity, or maybe a small velocity away from the rocket

**Update Rules:**
- Particles rise or fall slowly (drift with the wind)
- Current particle state described by parameter, size, that grows quickly then falls over time

**Death:**
- Kill particle when size becomes very small

**Render:**
- Scale size depending on the value of size
Particle System Example: Puffs of Smoke

Birth:
- System starts when the target is exploded
- Target is broken into many small pieces
- One particle assigned for each piece
- Each particle is assigned an initial velocity away from the center of the explosion

Update Rules: Doesn’t need to be perfect, just convincing
- Move continuously unless there is a collision
- Model accurate rigid body rotation, or just do random rotation
- Collisions handled as with colliding balls

Death: (not applicable)

Rendering:
- Draw the particle at the current position and orientation

Particle System Example: Explosions
Summary

Part I: Basic AI Concepts
- AI Basics
- Agents
- Rule-based Approaches
- Finite-State Machines

Part II: Motion
- Configuration spaces
- Waypoints
- Path planning algorithms
- Navigation Interfaces

Part III: Planning and Coordination
- Dynamic planning
- Potential-based path planning
- Flocking
- Particle systems