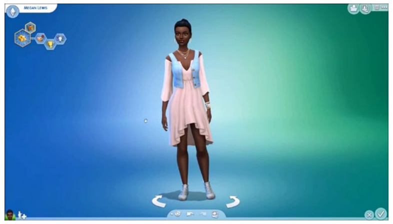
CS4386 AI Game Programming

Lecture 06
Goal-Oriented Behavior

Semester B, 2020-2021
Department of Computer Science
City University of Hong Kong

Motivation for Goal-Oriented Behavior

- In a game such as The Sims, characters need to demonstrate their emotional and physical state by choosing appropriate actions
  - They should eat when hungry, sleep when tired, chat to friends when lonely, and hug when in need of love
  - We could simply run a decision tree that selects available actions based on the current emotional and physical parameters of the character
  - This would lead to a very big decision tree as there are literally hundreds of parameterized actions to choose from for every character
- A better approach would be to present the character with a suite of possible actions and have it choose the one that best meets its immediate needs
- This is goal-oriented behavior (GOB), explicitly seeking to fulfill the character's internal goals



https://www.youtube.com/watch?v=QhC_hcnq0-s

Goals

- A character may have one or more goals, also called motives
  - There may be hundreds of possible goals, and the character can have any number of them currently active
  - They might have goals such as eat, regenerate health, or kill enemy
- Each goal has a level of importance (often called insistence) represented by a number
  - A goal with a high insistence will tend to influence the character's behavior more strongly
  - E.g., in The Sims, the character's physical and emotional parameters can be interpreted as goal values. A character might have a hunger motive: the higher the hunger value, the more eating becomes a pressing goal

Insistence

- The character will try to fulfill the goal or to reduce its insistence
  - Some games allow goals to be completely satisfied (such as killing an enemy)
  - Other games have a fixed set of goals that are always there, and they simply reduce the insistence when the goal is fulfilled (a character might always have a goal of "get healthy," for example, but at a low insistence when they are already healthy)
  - A zero value for insistence is equivalent to a completely satisfied goal
- We could easily implement goals without insistence values, but it would become more difficult to choose which goals to focus on

Actions (1)

- In addition to a set of goals, we need a suite of possible actions to choose from
  - In The Sims world, a kettle adds a "boil kettle" action and an empty oven adds an "insert raw food" action to the list of possibilities
  - In an action game an enemy might introduce an "attack me" action, while a door might expose a "lock" action
- The available actions depend on the current state of the game
  - The empty oven might check if the character is carrying raw food before positing its "insert" action. An oven containing raw food does not allow more food to be added; it exposes a "cook food" action
  - The door exposes an "unlock" action if it is already locked or maybe an "insert key" action first before unlocking is allowed

Actions (2)

- As the actions are added to the list of options, they are rated against each motive the character has
- This rating shows how much impact the action would have on a particular motive, e.g.,
  - The "playing with the games console" action might increase happiness a lot but also decrease energy
  - A "shoot" action can fulfill a kill enemy goal
- The goal that an action promises to fulfill might be several steps away, e.g.,
  - A piece of raw food might offer to fulfill hunger
  - If the character picks it up, it will not become less hungry, but now the empty oven will offer the "insert" action
  - The same thing continues through a "cook food" action, a "remove food from oven" action, and finally an "eat" action

Simple Selection

- We have a set of possible actions and a set of goals
- Certain actions may fulfill specific goals, e.g.,

  Goal: Eat = 4

  Goal: Sleep = 3

  Action: Get-Raw-Food (Eat – 3)

  Action: Get-Snack (Eat – 2)

  Action: Sleep-In-Bed (Sleep – 4)

  Action: Sleep-On-Sofa (Sleep – 2)

  The change in goal insistence is a heuristic estimate of its utility (how useful it might be to the character)

  – Which action should be chosen?

  – A simple approach would be to choose the most pressing goal (the one with the largest insistence) and find an action that either fulfills it completely or provides it with the largest decrease in insistence

  – If more than one action can fulfill a goal, we could choose between them at random or simply select the first one we find

Pseudo-Code: Simple Selection

```
def chooseAction(actions, goals):

  # Find the goal to try and fulfil
  topGoal = goals[0]
  for goal in goals[1..]:
    if goal.value > topGoal.value:
      topGoal = goal

  # Find the best action to take
  bestAction = actions[0]
  bestUtility = -actions[0].getGoalChange(topGoal)
  for action in actions[1..]:

    # We invert the change because a low change value
    # is good (we want to reduce the value for the goal)
    # but utilities are typically scaled so high values
    # are good.
    utility = -action.getGoalChange(topGoal)

    # We look for the lowest change (highest utility)
    if thisUtility > bestUtility:
      bestUtility = thisUtility
      bestAction = action

  # Return the best action, to be carried out
  return bestAction
```

It is simply two max()-style blocks of code, one for the goal and one for the action

Performance

- Performance
  - O(n+m) in time

    where n is the number of goals, and m is the number of possible actions
  - O(1) in memory
- Pros
  - Simple and fast
  - Can give surprisingly sensible results, especially in games with a limited number of actions available
- Cons
  1. it fails to take account of side effects that an action may have
  2. it does not incorporate any timing information

Another Example

- Insistence here is measured on a 5-point scale:

  Goal: Eat = 4

  Goal: Bathroom = 3

  Action: Drink-Soda (Eat − 2; Bathroom + 2)

  Action: Visit-Bathroom (Bathroom − 4)

- Using the simple section algorithm, the Drink-Soda action will be chosen. However, it will lead to the situation where the need for the toilet is at the top of the 5-point scale
  - Sometimes this could be fatal, e.g., a character in a shooter might have a pressing need for a health pack, but running right into an ambush to get it is not a sensible strategy
  - We need to consider the side effects of actions
  - A new measure to quantify the discontentment of the character can be introduced to address this issue

Discontentment

- Discontentment is calculated based on all the goal insistence values, where high insistence leaves the character more discontent
- The aim of the character is to reduce its overall discontentment level so it will focus on the whole set of goals rather than a single one
- If we simply add all the insistence values to give the discontentment, a bunch of medium values may swamp one high goal
- Thus the insistence is scaled so that higher values of insistence contribute disproportionately high discontentment values, such as formulating the discontentment as the sum of squares of insistence values after each action

Discontentment Example

- Back to the previous example:

  Goal: Eat = 4

  Goal: Bathroom = 3

  Action: Drink-Soda (Eat − 2; Bathroom + 2)

  Action: Visit-Bathroom (Bathroom − 4)

  - After Drink-Soda action:

    Eat = 2, Bathroom = 5, Discontentment = 2^2+5^2 = 29
  - After Visit-Bathroom action:

    Eat = 4, Bathroom = 0, Discontentment = 4^2+0^2 = 16
  - The Visit-Bathroom action will be chosen as it leads to the lowest discontentment
- In some literature, discontentment may be known as an energy metric

Pseudo-Code: Discontentment

```
def chooseAction(actions, goals):

  # Go through each action, and calculate the
  # discontentment.
  bestAction =  actions[0]
  bestValue = calculateDiscontentment(actions[0], goals)

  for action in actions:
    thisValue = calculateDiscontentment(action, goals)
    if thisValue < bestValue:
      bestValue = thisValue
      bestAction = action

  # return the best action
  return bestAction

def calculateDiscontentment(action, goals):

  # Keep a running total
  discontentment = 0

  # Loop through each goal
  for goal in action:
    # Calculate the new value after the action
    newValue = goal.value + action.getGoalChange(goal)

    # Get the discontentment of this value
    discontentment += goal.getDiscontentment(value)
```

Some goals may be really important and have very high discontentment values for large values (such as the stay-alive goal, for example); they can return their insistence cubed, for example, or to a higher power.

Others may be relatively unimportant and make a tiny contribution only

In practice, this will need some tweaking in your game to get it right

Performance

- Performance
  - O(nm) in time

    where n is the number of goals, and m is the number of possible actions
  - O(1) in memory

Timing (1)

- In order to make an informed decision as to which action to take, the character needs to know how long the action will take to carry out
  - It may be better for an energy-deficient character to get a smaller boost quickly (by eating a chocolate bar, for example), rather than spending eight hours sleeping
- Timing is often split into two components
  1. Actions typically take time to complete
  2. In some games it may also take significant time to get to the right location and start the action
- Because game time is often extremely compressed in some games, the length of time it takes to begin an action becomes significant

Timing (2)

- If it is needed, the length of journey required to begin an action cannot be directly provided by the action itself. It must be either
    1. provided as a guess (a heuristic such as “the time is proportional to the straight-line distance from the character to the object”); or
    2. calculated accurately (by pathfinding the shortest route)
- There is significant overhead for pathfinding on every possible action available
    - For a game level with hundreds of objects and many hundreds or thousands of possible actions, pathfinding to calculate the timing of each one is impractical
    - A heuristic must be used

Utility Involving Time

- In some games, goal values change over time:
  - a character might get increasingly hungry unless it gets food
  - a character might tend to run out of ammo unless it finds an ammo pack
  - a character might gain more power for a combo attack the longer it holds its defensive position
- When goal insistences change on their own, not only does an action directly affect some goals, but also the time it takes to complete an action may cause others to change naturally
  - If we know how goal values will change over time, then we can factor those changes into the discontentment calculation

Example

Goal: Eat = 4 changing at + 4 per hour

Goal: Bathroom = 3 changing at +2 per hour

Action: Eat-Snack (Eat − 2) 15 minutes

Action: Main-Meal (Eat − 4) 1 hour

Action: Visit-Bathroom (Bathroom − 5) 15 minutes

- After Eat-Snack action:
  Eat = 2, Bathroom = 3.5, Discontentment = $2^2+3.5^2$ = 16.25
- After Main-Meal action:
  Eat = 0, Bathroom = 5, Discontentment = 0^2+5^2 = 25
- After Visit-Bathroom action:
  Eat = 5, Bathroom = 0, Discontentment = 5^2+0^2 = 25

Pseudo-Code: Utility Involving Time

```
def calculateDiscontentment(action, goals):

  # Keep a running total
  discontentment = 0

  # Loop through each goal
  for goal in action:
     if (action.getGoalChange(goal) != 0)
        newValue = goal.value + action.getGoalChange(goal)
     else
        newValue = action.getDuration() * goal.getChange()


    # Get the discontentment of this value
    discontentment += goal.getDiscontentment(newValue)
```

It works by modifying the expected new value of the goal by either the action or the normal rate of change of the goal multiplied by the action's duration

Performance

- Performance
  - $O(nm)$ in time

    where n is the number of goals, and m is the number of possible actions
  - $O(1)$ in memory

Calculating the Goal Change Over Time

- In some games the change in goals over time is fixed and set by the designers
  - E.g., The Sims has a basic rate at which each motive changes. Even if the rate is not constant, but varies with circumstance, the game still knows the rate, because it is constantly updating each motive based on it
- In some situations we may not have any access to the value
  - In a shooter, where the “hurt” motive is controlled by the number of hits being taken, we do not know in advance how the value will change as it depends on what happens in the game
  - In this case, we need to approximate the rate of change
  - The simplest and most effective way to do this is to regularly take a record of the change in each goal. Each time the GOB routine is run, we can quickly check each goal and find out how much it has changed

Recency-Weighted Average

```
rateSinceLastTime = changeSinceLastTime / timeSinceLast
basicRate = 0.95 * basicRate + 0.05 * rateSinceLastTime
```

where the 0.95 and 0.05 can be any values that sum to 1.

The `timeSinceLast` value is the number of units of time that has passed since the GOB routine was last run

- The recency-weighted average provides a very simple degree of learning
  - If the character is taking a beating, it will automatically act more defensively (because it will be expecting any action to cost it more health), whereas if it is doing well it will start to get bolder

The Need for Planning (1)

- Let's imagine a fantasy role-playing game
  - A magic-using character has 5 fresh energy crystals in their wand
  - Powerful spells take multiple crystals of energy
  - The character is in desperate need of healing and would also like to fend off the large ogre descending on her

Goal: Heal = 4

Goal: Kill-Ogre = 3

Action: Fireball (Kill-Ogre – 2) 3 energy-slots

Action: Lesser-Healing (Heal – 2) 2 energy slots

Action: Greater-Healing (Heal – 4) 3 energy slots

  - Using the algorithm introduced before, the mage will choose Greater-Healing action because it gives the lowest discontentment. However, the mage will not have enough energy crystals left to fight with the ogre
  - The best combination should really be to cast the "lesser healing" spell, followed by the "fireball" spell, using the 5 energy slots exactly

The Need for Planning (2)

- In this example, we could include the magic in the wand as part of the motives (we are trying to minimize the number of slots used), but in a game where there may be many hundreds of permanent effects (doors opening, traps sprung, routes guarded, enemies alerted), we might need many thousands of additional motives
- To allow the character to properly anticipate the effects and take advantage of sequences of actions, a level of planning must be introduced
  - Goal-oriented action planning (GOAP) extends the basic decision making process
  - It allows characters to plan detailed sequences of actions that provide the overall optimum fulfillment of their goals

Overall Utility GOAP

- We can extend the utility-based GOB scheme to consider more than one action in a series
- Suppose we want to determine the best sequence of four actions
  - We can consider all combinations of four actions and predict the discontentment value after all are completed
  - The lowest discontentment value indicates the sequence of actions that should be preferred, and we can immediately execute the first of them
- GOAP: we consider multiple actions in sequence and try to find the sequence that best meets the character's goals in the long term
  - In this case, we are using the discontentment value to indicate whether the goals are being met

Pseudo-Code: Depth-first GOAP

```
def planAction(worldModel, maxDepth):
  # Create storage for world models at each depth, and
  # actions that correspond to them
  models = new WorldModel[maxDepth+1]
  actions = new Action[maxDepth]

  # Set up the initial data
  models[0] = worldModel
  currentDepth = 0

  # Keep track of the best action
  bestAction = None
  bestValue = infinity

  # Iterate until we have completed all actions at depth
  # zero.
  while currentDepth >= 0:

    # Calculate the discontentment value, we'll need it
    # in all cases
    currentValue =
      models[currentDepth].calculateDiscontentment()

    # Check if we're at maximum depth
    if currentDepth >= maxDepth:

      # If the current value is the best, store it
      if currentValue < bestValue:
        bestValue = currentValue
        bestAction = actions[0]

        # We're done at this depth, so drop back
        currentDepth -= 1

        # Jump to the next iteration
        continue

    # Otherwise, we need to try the next action
    nextAction = models[currentDepth].nextAction()
    if nextAction:

      # We have an action to apply, copy the current model
      models[currentDepth+1] = models[currentDepth]

      # and apply the action to the copy
      actions[currentDepth] = nextAction
      models[currentDepth+1].applyAction(nextAction)

      # and process it on the next iteration
      currentDepth += 1

    # Otherwise we have no action to try, so we're
    # done at this level
    else:

      # Drop back to the next highest level
      currentDepth -= 1

  # We've finished iterating, so return the result
  return bestAction
```

This is a typical depth-first search technique, implemented without recursion

Performance

- Performance
  - O(nm^k) in time

    where n is the number of goals, m is the mean number of available actions, and k is the maximum depth
  - O(k) in memory
- Heuristic
  - To speed up the algorithm we can use a heuristic: we demand that we never consider actions that lead to higher discomfort values
  - To implement this heuristic we need to calculate the discomfort value at every iteration and store it. If the discomfort value is higher than that at the previous depth, then the current model can be ignored, and we can immediately decrease the current depth and try another action

Reference

- Artificial Intelligence for Games
  - Chapter 5.7