
Class 6
Pathfinding and AI

Game A.I.

- Artificial Intelligence is what makes computer-controlled entities in a game behave reasonably
- Can be used to control enemy entities, or used to support the human player
- Usually, we want AI-controlled entities to behave as if a human is controlling them
- The goal of game AI is to provide a fun challenge, not necessarily to make optimal decisions

Pathfinding

- Pathfinding is the most common primitive used in game AI
- A path is a list of instructions for getting from one location to another
  - Not just locations: instructions could include “jump” or “climb ladder”
- A hard problem!
  - Bad path planning breaks the immersive experience
  - Many games get it wrong

3D world representation

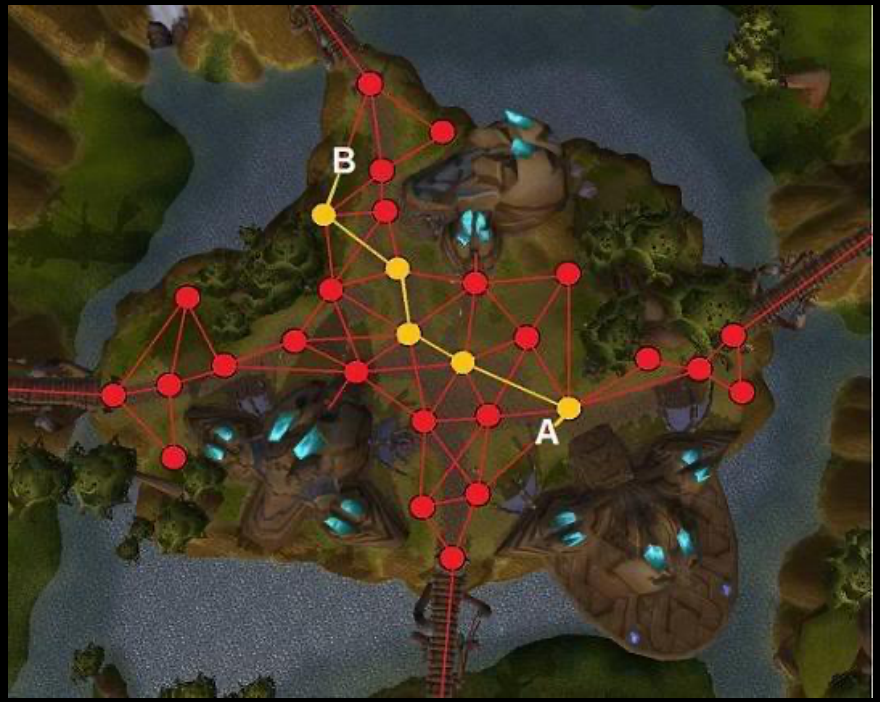
- Need an efficient encoding of relevant information in the world
  - Navigable space
  - Important locations (health, safety, bases, mission objective)
- Graph-based approaches
  - Waypoints
  - Navigation meshes

Decision Making

- World is represented as a graph
  - Nodes represent open space
  - Edges represent ways to travel between nodes
  - Use graph search algorithms to find paths
- Two common types
  - Waypoint graphs
  - Navigation meshes

Waypoint graphs

- Represents a fixed set of paths through the world
- Nodes are waypoints
- Edges represent a path between adjacent nodes



Disadvantages of waypoint graphs

- Optimal path is likely not in the graph
  - Paths will zig-zag to destination
  - Good paths require huge numbers of waypoints and/or connections, which can be expensive
- No model of space in between waypoints
  - No way of going around dynamic objects without recomputing the graph
- Awkward to handle entities with different radii
  - Have to turn off certain edges and add more waypoints

Navigation meshes

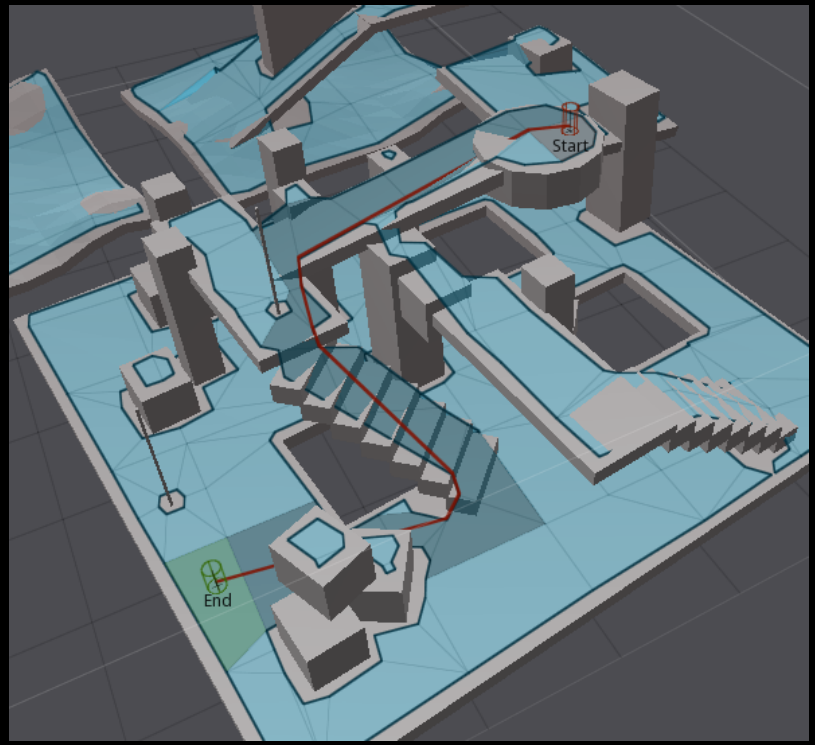
- Convex polygons as navigable space
- Nodes are polygons
- Edges show which polygons share a side

Advantages of navigation meshes

- More efficient and compact representation
  - Equivalent waypoint graph would have many more nodes and would take longer to traverse
- Models entire navigable space
  - Can plan path from anywhere inside nav mesh
  - Paths can be planned around dynamic obstacles
  - Zig-zagging can be avoided
- Naturally handles entities of different radii
  - Don't go through edges less than 2 * radius long
  - Leave at least a distance of radius when moving around nav mesh vertices

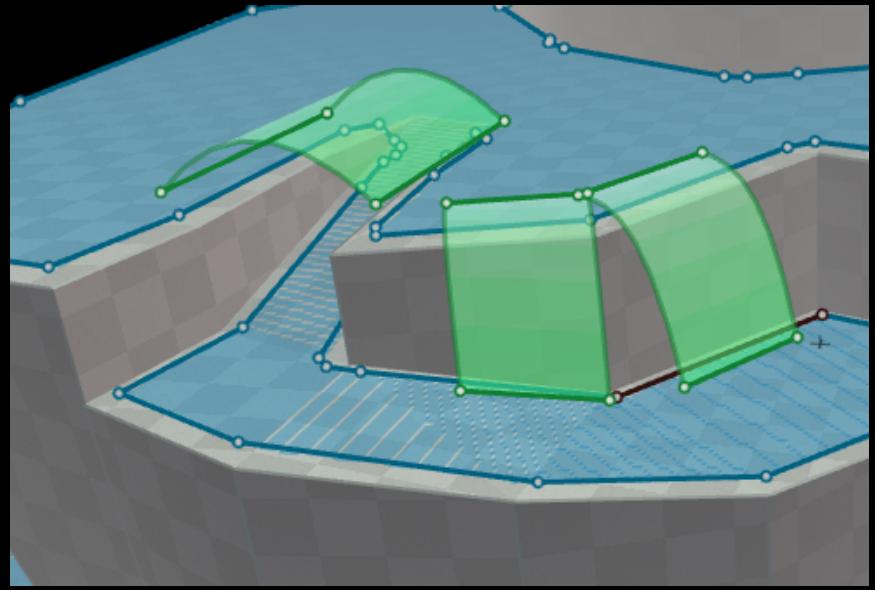
Navigation meshes

- Different from collision mesh
  - Only contains walkable faces
  - Stairs become a single, rectangular polygon
  - Polygons are usually smaller to account for player radius



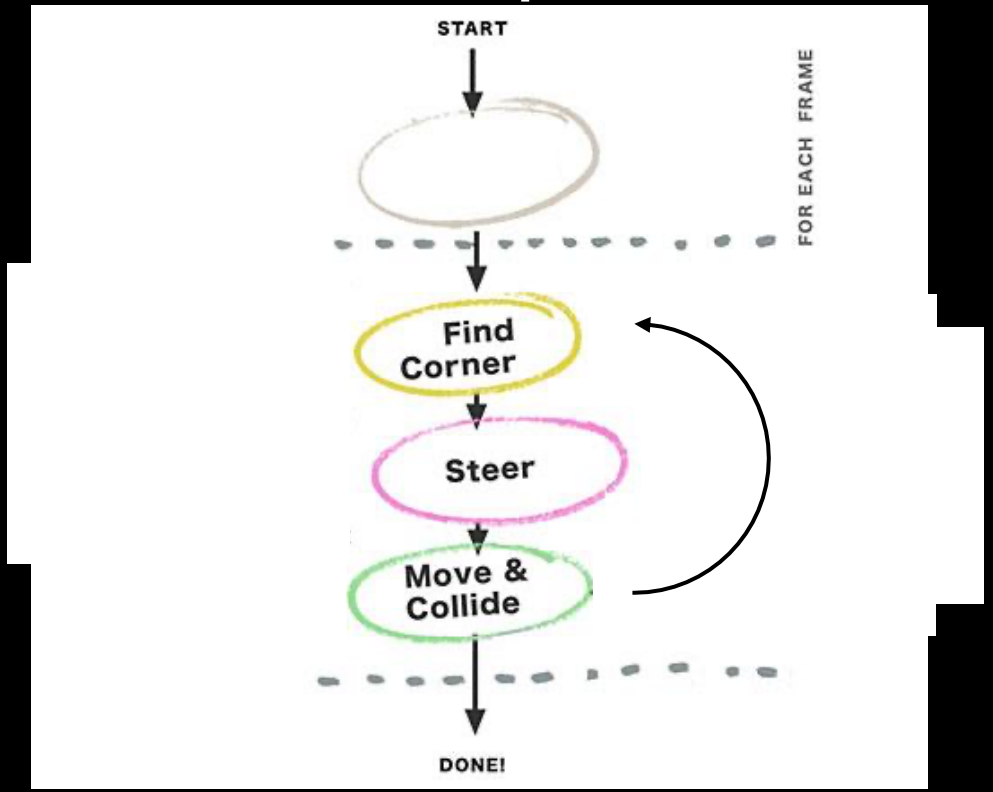
Navigation meshes

- Annotate special regions
  - Can have regions for jumping across, falling down, crouching behind, climbing up, …
  - Regions are usually computed automatically



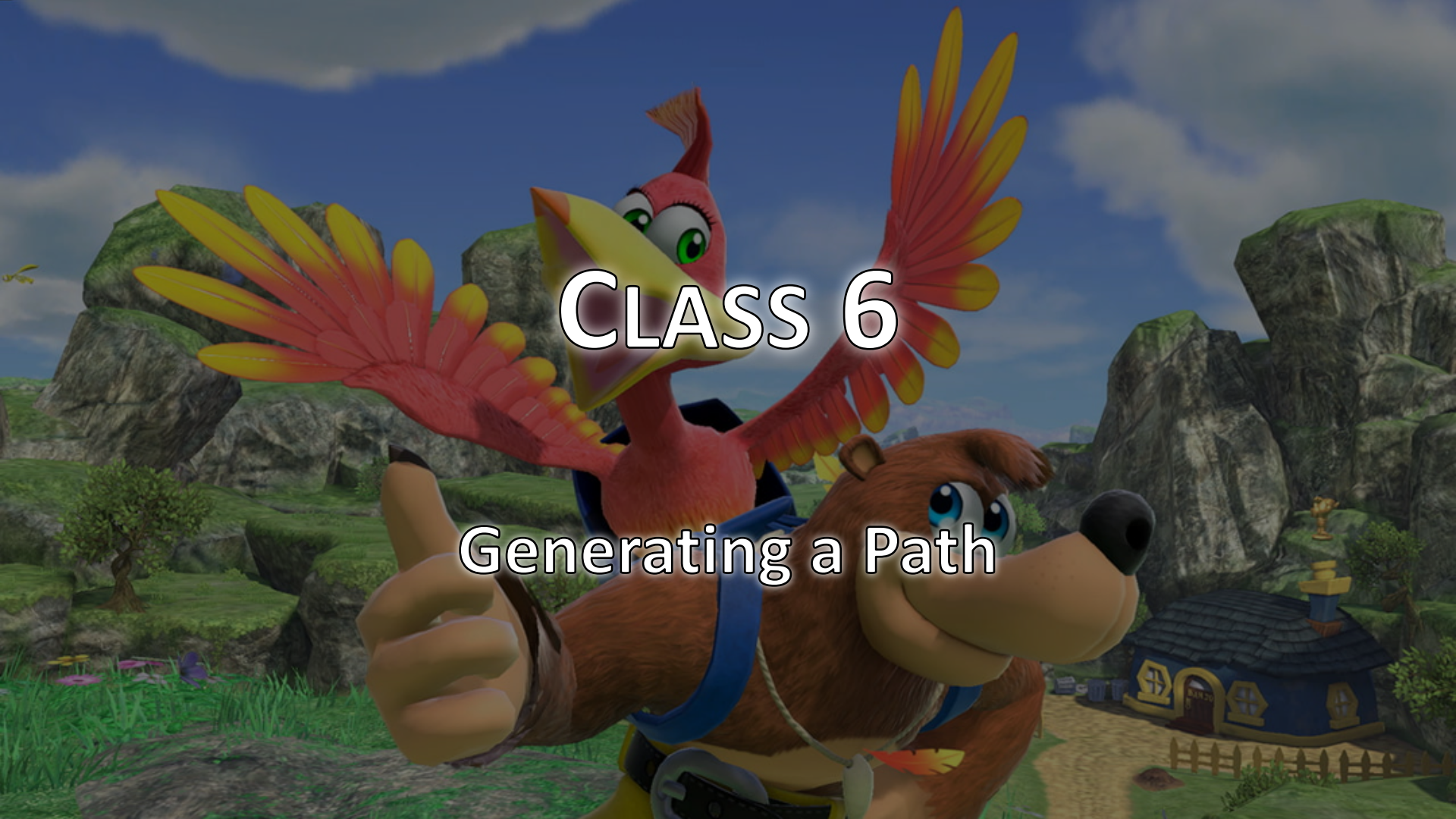
Navigation loop

- Process for robust path navigation on a navigation mesh:
  - Find sequence of polygons (corridor) using graph algorithm
  - Find corner using string pulling (funnel algorithm)
  - Steer using smoothing
  - Actually move/collide entity



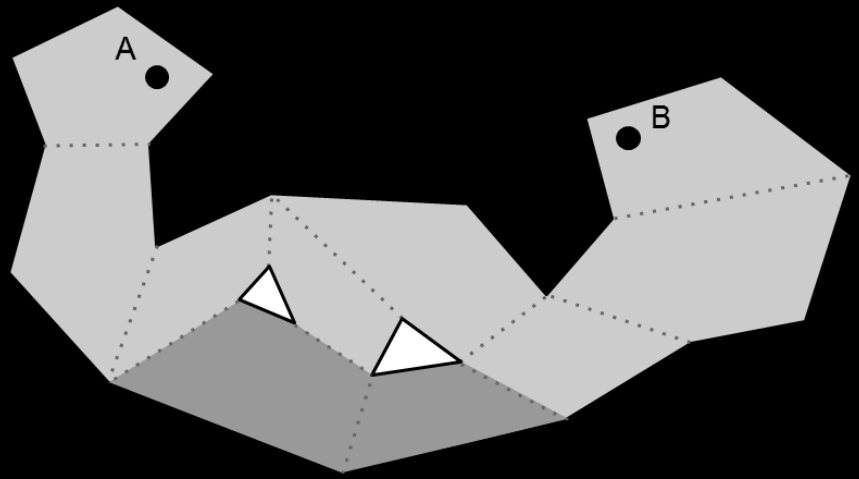
Graph search

- First step in finding a path
- Graph search problem statement
  - Given starting point A, target point B and a nav mesh
  - Generate a list of nav mesh nodes from A to B (called a corridor)
- Simplest approach: Breadth-first search
  - Keep searching until target point is reached
  - Each edge has equal weight
- Most common approach: A-star
  - Variable edge weights (mud or steep surfaces may have higher cost)
  - Uses a heuristic to arrive at an answer faster


Class 6
Generating a Path

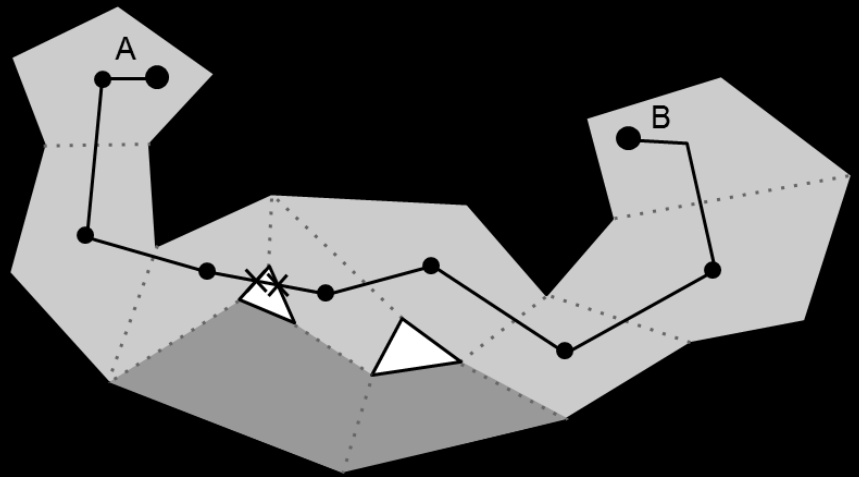
Path generation: problem statement

- Given a list of polygons (output of a graph search)
  - The light polygons
- Construct the shortest path for the agent
  - Where a path is a sequence of connected segments
  - The path must lie entirely in the list of polygons



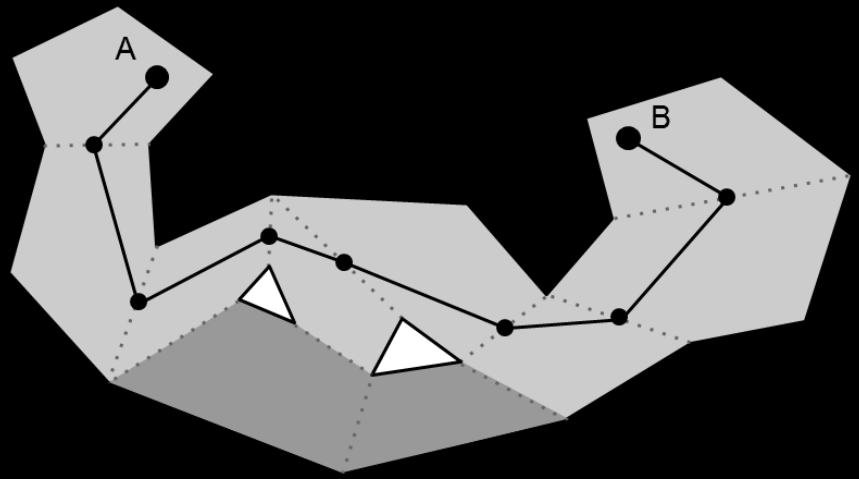
Path generation: first attempt

- Can we just connect polygon centers?
- No: the path might not even be within the polygons
- Polygons' convexity only tells us that any 2 points in a single polygon can be connected with a segment



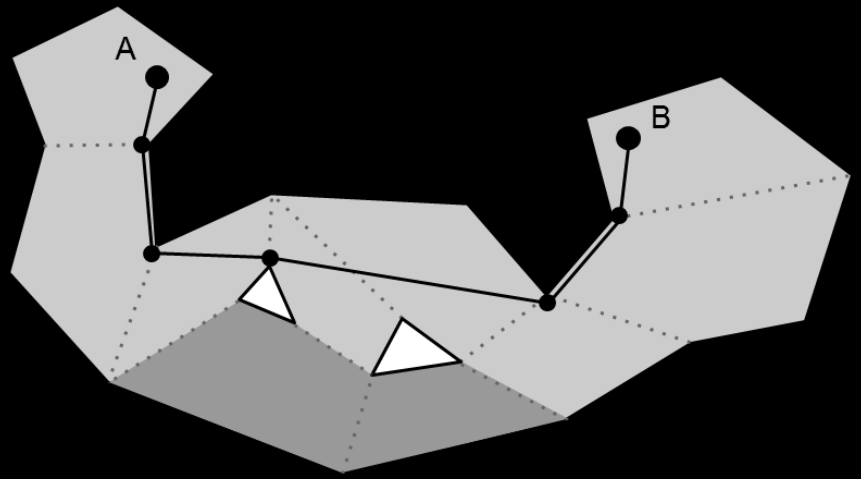
Path generation: second attempt

- Can we just connect centers of polygon sides?
- This always produces a valid path (within the polygons)
- But not always the optimal path (zig-zagging)
- This is just a waypoint graph!



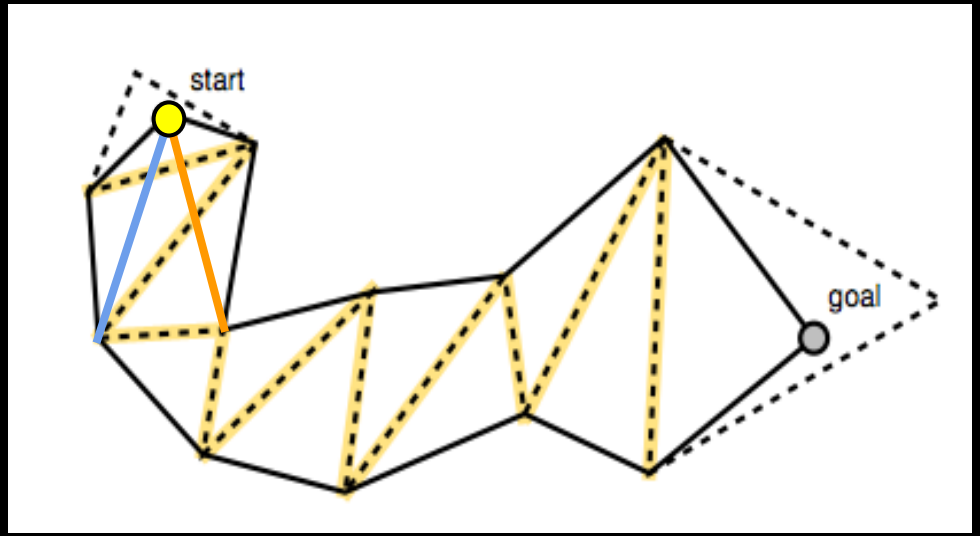
Path generation: third attempt

- The Funnel algorithm finds the optimal path
- Hugs corners
- Is like “pulling a string” that connects A and B until it is taut

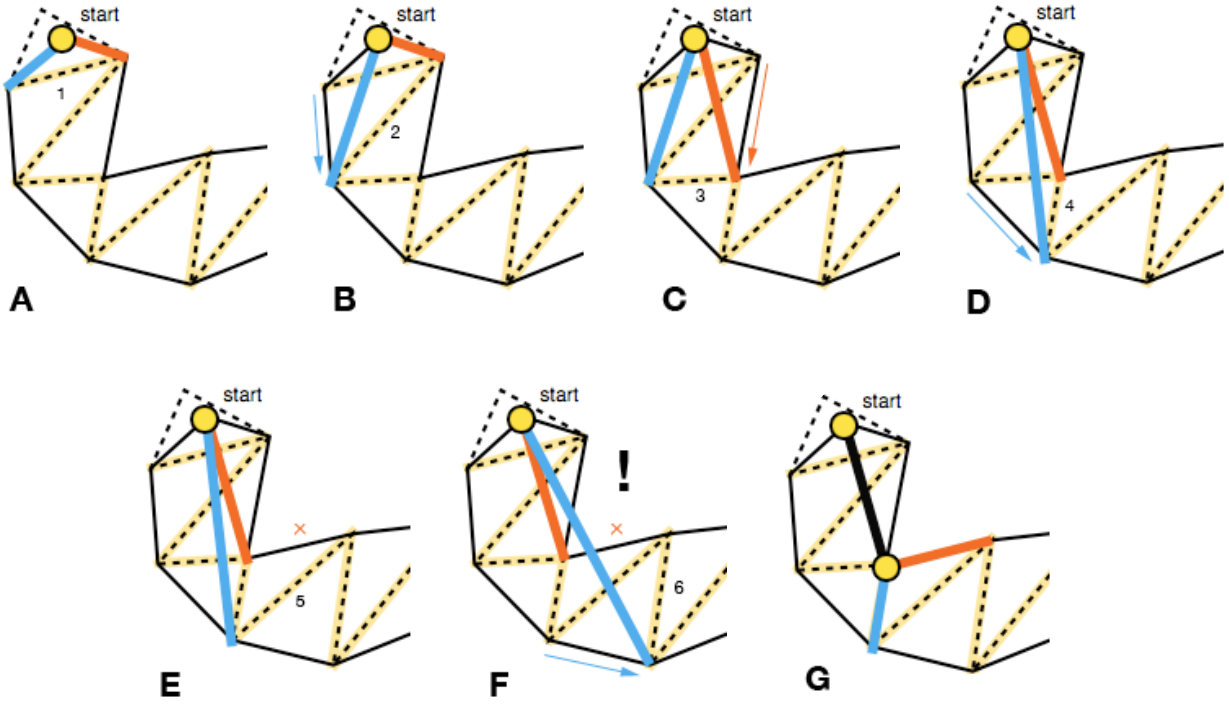


Funnel algorithm

- Traverses through a list of polygons connected by shared edges (portals)
- Keeps track of the leftmost and rightmost sides of the "funnel" along the way
- Alternates updating the left and right sides, making the funnel narrower and narrower
- Add a new point to the path when they cross



Funnel Algorithm



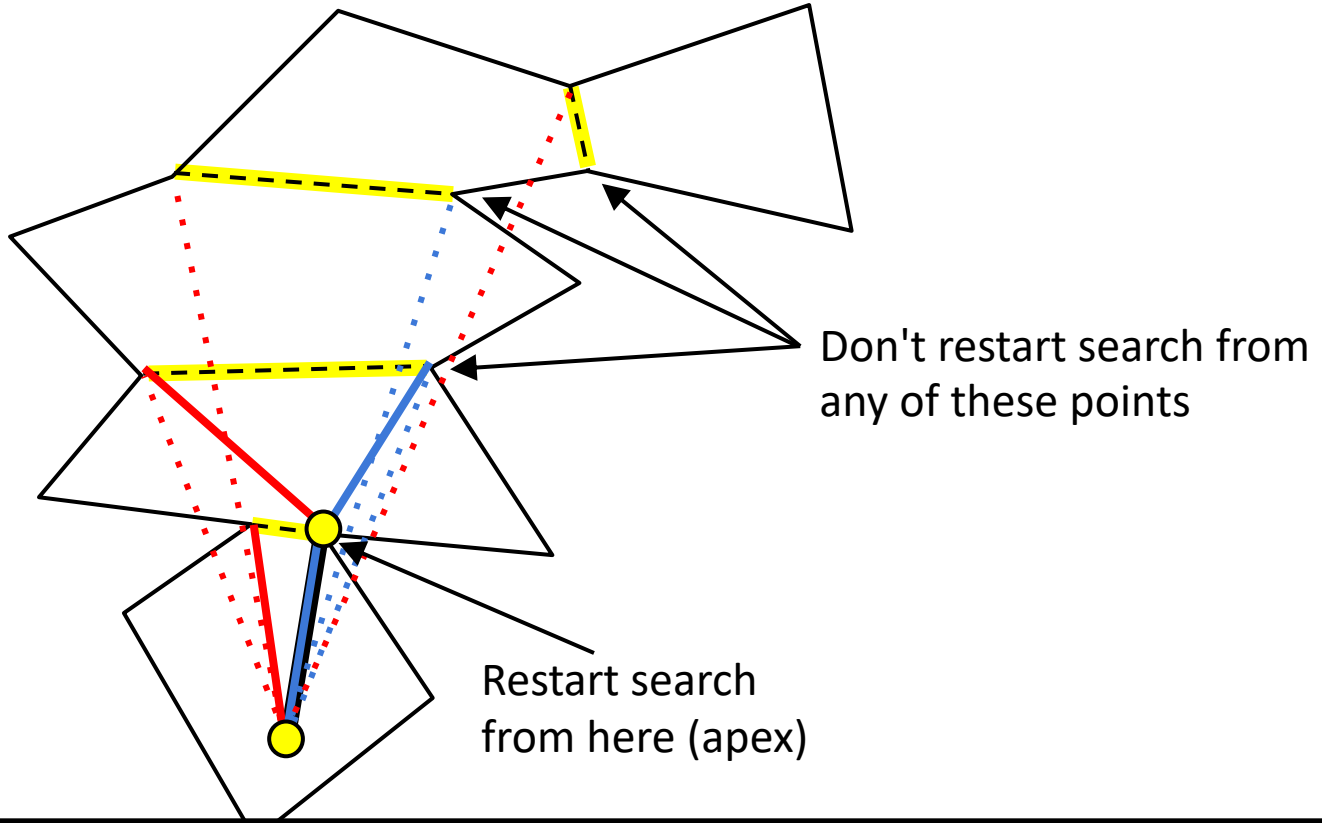
Funnel Algorithm

- Start
  - Apex point = start of path
  - Left and right points = left and right vertices of first portal
- Step
  - Advance to the next portal
  - Try to move left point to left vertex of next portal
    - If inside the funnel, narrow the funnel (C-D in previous slide)
    - If past the right side of the funnel, turn a corner (E-G in previous slide)
      - Add right point to path
      - Set apex point to right point
      - Restart at portal where right point came from
  - Try to move right point to right vertex of next portal
    - Similar to left point

Edge cases

- Zero-length funnel side (portals that share a vertex)
  - Always use left*0.99+right*0.01 for the left and left*0.01+right*0.99 for the right (shrinks portal slightly)
- End iteration of the funnel algorithm
  - End point is portal of size 0, need to check for potential restart like other portals

Funnel algorithm example



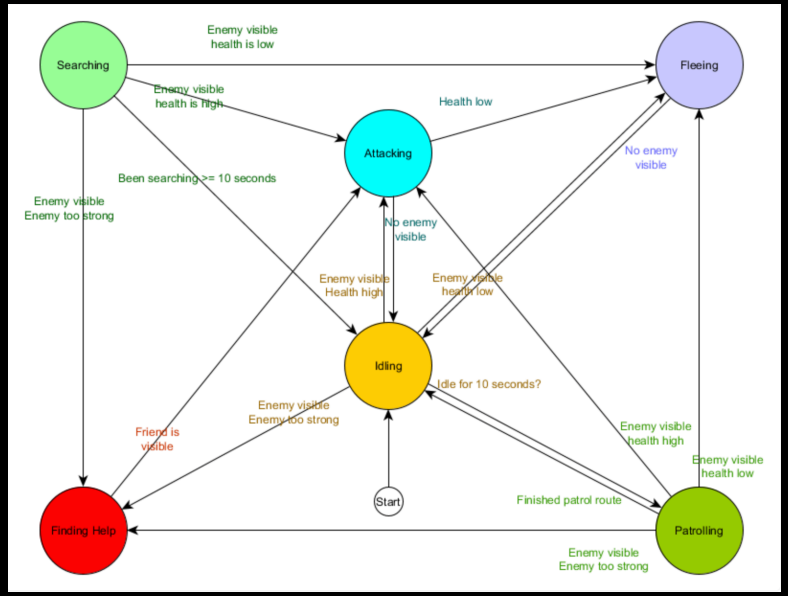

Class 6
Finite/Hierarchical State Machines

Decision Making

- NPC’s should do something, but what?
- Could hardcode the logic
  - Game-specific
  - Likely involves copied code
- We want a structured way for NPC’s to make decisions
  - Based on game state, unit state, random values, etc…

Finite State Machines

- A finite state machine (FSM) defines a set of states for an entity and the conditions that cause the entity to change states
- The FSM can be represented by a directed graph where the nodes are states and the edges are transitions between states
- At every tick, we check whether we need to move to a different state given the state we are currently in and the transitions to other states that we have defined



Finite State Machine Example

- This is an FSM for a patrolling guard AI
- Note how a lot of logic is copied between the edges in the graph
- To implement this FSM, we would have to do a lot of copy-and-pasting and hardcoding
- Can we formulate a better approach that creates the same behavior?

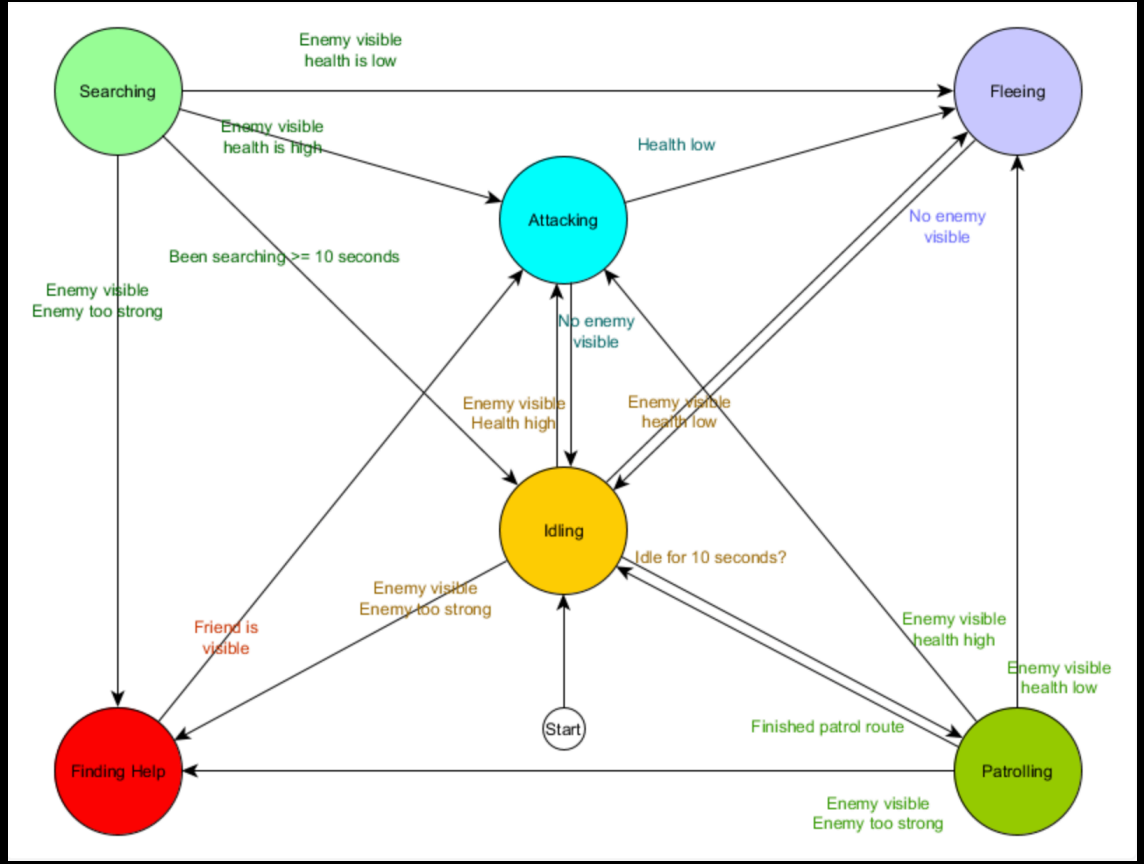
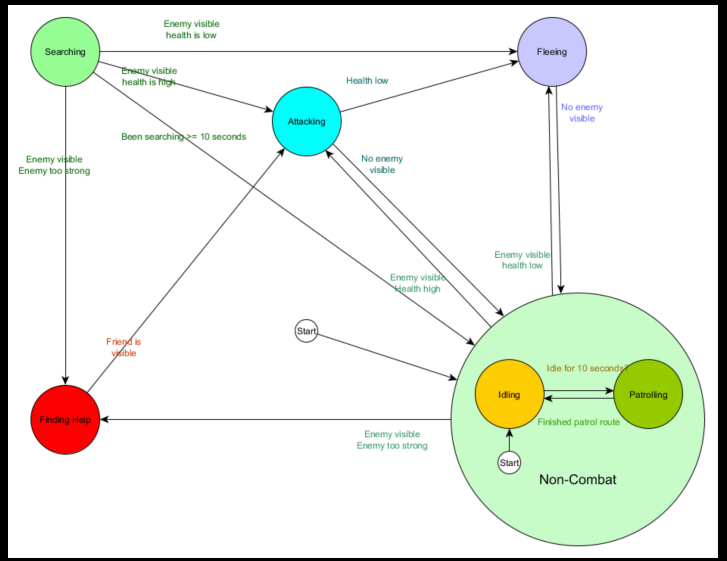


Image from:
https://www.gamedev.net/tutorials/programming/artificial-intelligence/the-total-beginners-guide-to-game-ai-r4942/

Hierarchical State Machines

- A hierarchical state machine (HSM) defines a state hierarchy
- States may have substates within them, (which also may have substates within them)
- Transitions between states are defined between nodes in the hierarchy (non-combat) as well as leaf substates (attacking, idling)
- It is easier to have an entity perform two behaviors at once if we use an HSM rather than a FSM



Hierarchical State Machine Example

- The is an HSM describing the same functionality as the FSM for the patrolling guard
- Notice how using an HSM simplifies the transitions in the graph
- We still would have to define a lot of functionality game-side to make this work

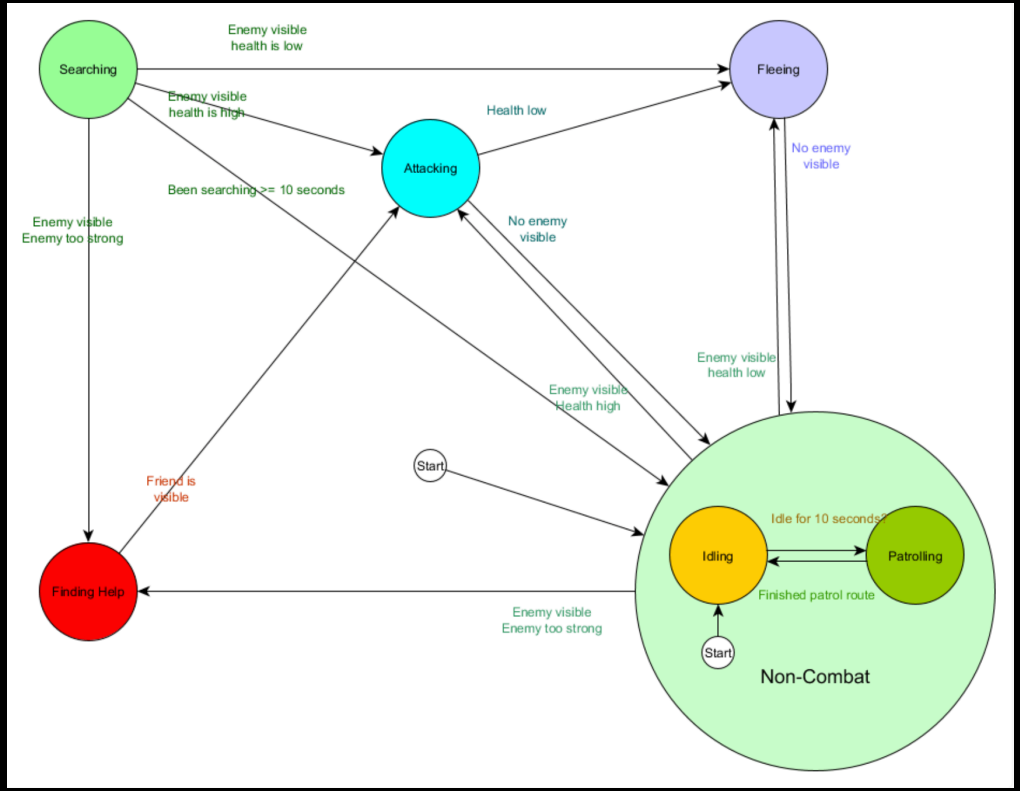
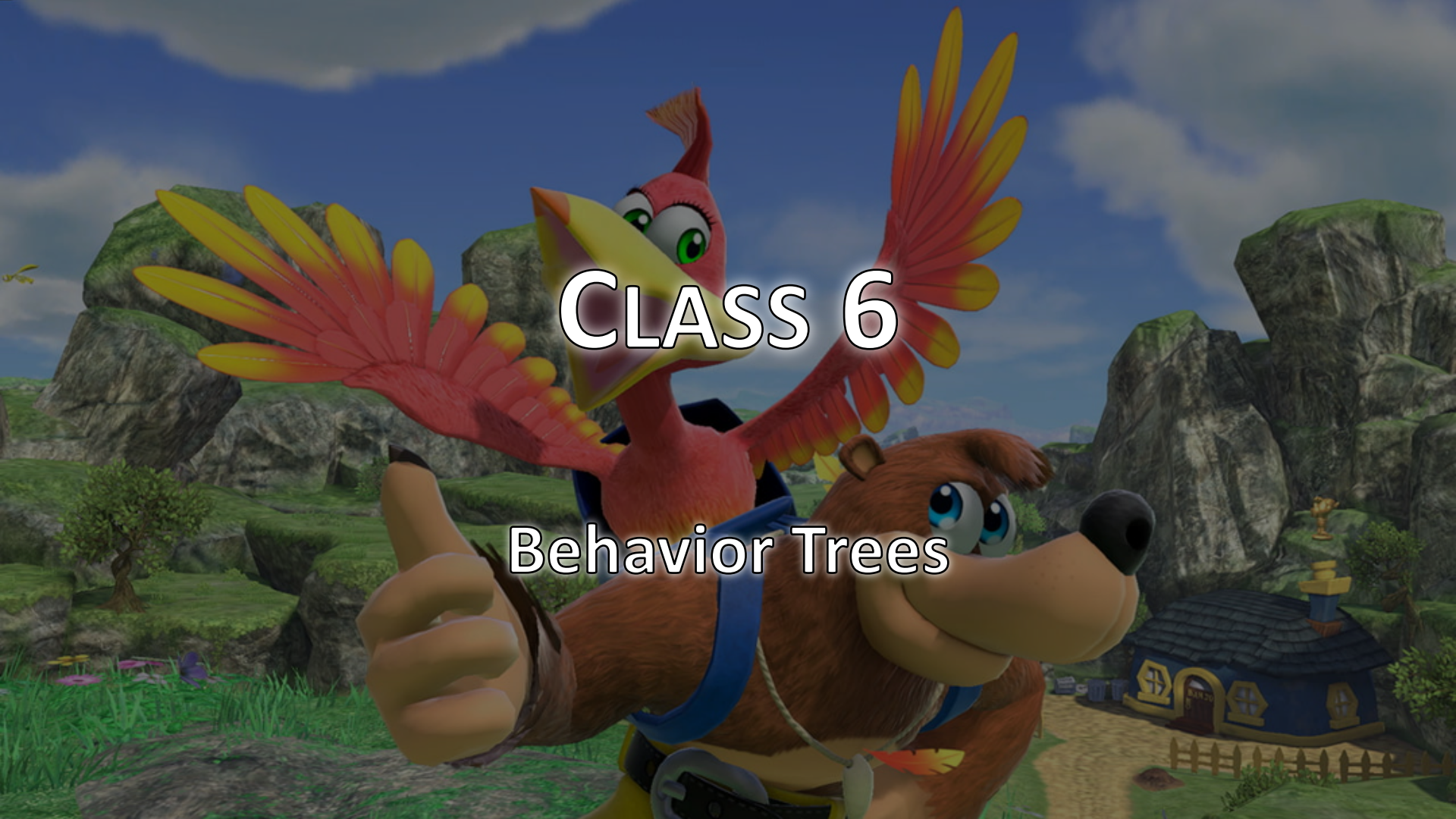


Image from:
https://www.gamedev.net/tutorials/programming/artificial-intelligence/the-total-beginners-guide-to-game-ai-r4942/

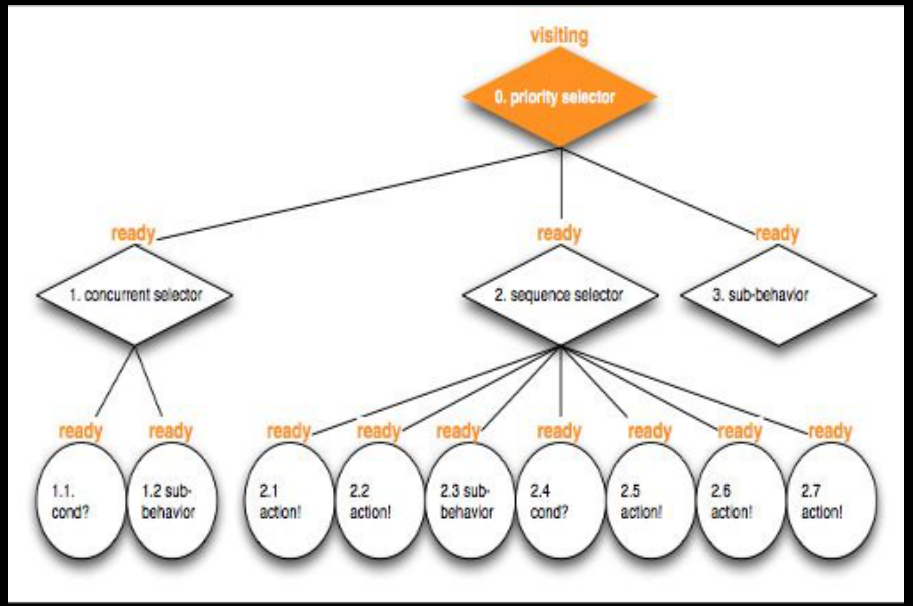

Class 6
Behavior Trees

Behavior Trees

- Popularized by Halo 2
- Core functionality is engine-general!

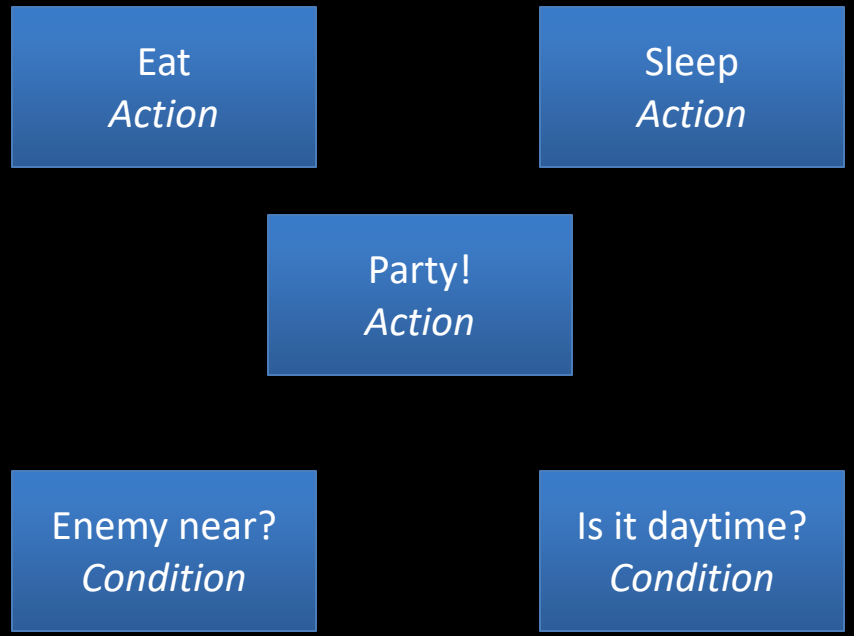
Structure

- It's a tree!
- Every tick, the root node is updated
- Each node returns a status when it's updated
  - SUCCESS, FAIL, RUNNING
- Nodes will update their children and return a status based on responses



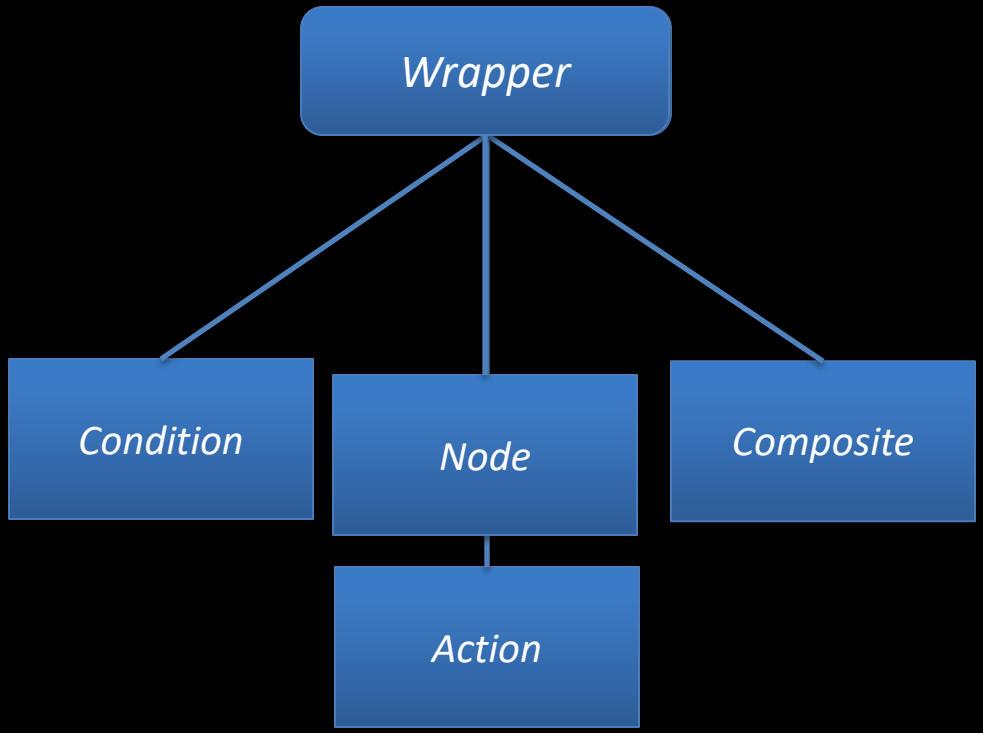
The Leaves

- Leaf nodes of the tree are Actions and Conditions
- Actions do things
  - Make a unit move or attack
  - Return SUCCESS or FAIL based on result of Action
  - Return RUNNING if Action is still if progress
- Conditions check some game state
  - Returns SUCCESS if the condition is true, or FAIL if the condition is false



The Internal Nodes

- Internal nodes are Composites and Wrappers/Decorators
- Composites have multiple children nodes
- Wrappers wrap a single child node and execute their child if they succeed
- Composites and Wrappers dictate the traversal of the tree on an update

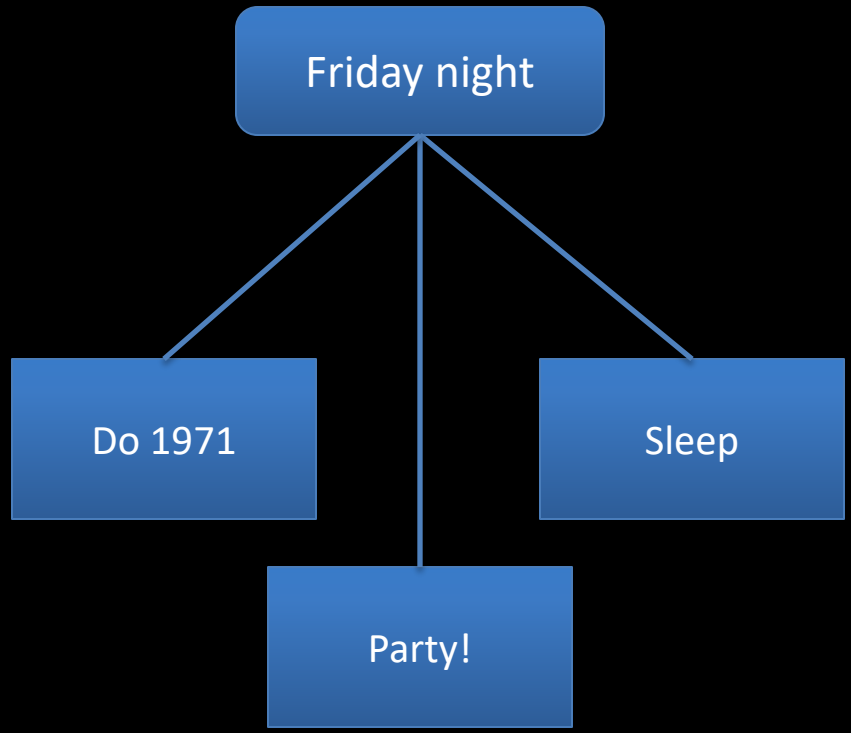


The Composites

- Maintain a list of children nodes
- Update by updating the children nodes (usually in a particular order)
- Return RUNNING if a child returns RUNNING
- Return SUCCESS/FAIL under other circumstances depending on the type of composite

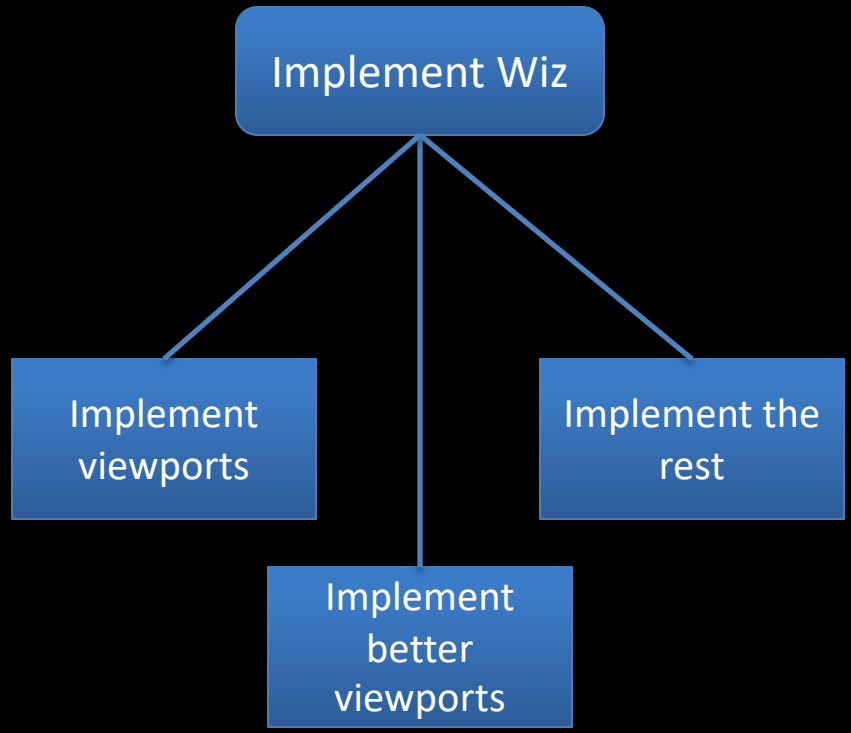
The Selector

- On update, updates each of its children in order until one of them *doesn't* fail
  - Hence "select", as this child has been "selected"
- Returns FAIL only if all children fail
- Kind of like an if else statement or block of or's
  - If child 1 succeeds, else if child 2 succeeds, etc…



The Sequence

- On update, updates each of its children in order until one *does* fail
- Returns SUCCESS if the entire sequence completes, else FAIL
- If one behavior fails then the whole sequence fails, hence “sequence”
- Similar to a bunch of and’s



Other Nodes

- Wrappers contain a single child and modify its behavior. Examples include:
  - Invert child
  - Repeatedly update child X times until FAIL or SUCCESS
- Random Selectors update its children in random order
  - For unpredictable behavior
  - Harder to debug though
- Some engines define composites that allow for multiple tasks to be executed simultaneously

Behavior Tree Node

- Just needs to be updated and reset
- Sample contract:

```
enum Status { SUCCESS, FAIL, RUNNING };
class BTNode {
public:
	virtual Status update(float seconds) = 0;
	virtual void reset() = 0;
}
```

Composites

- Needs a list of children
- Also should keep track of what child was running
- Sample contract:

```
class Composite: public BTNode {
    std::vector<BTNode *> m_children;
    BTNode *m_lastRunning;
}
```

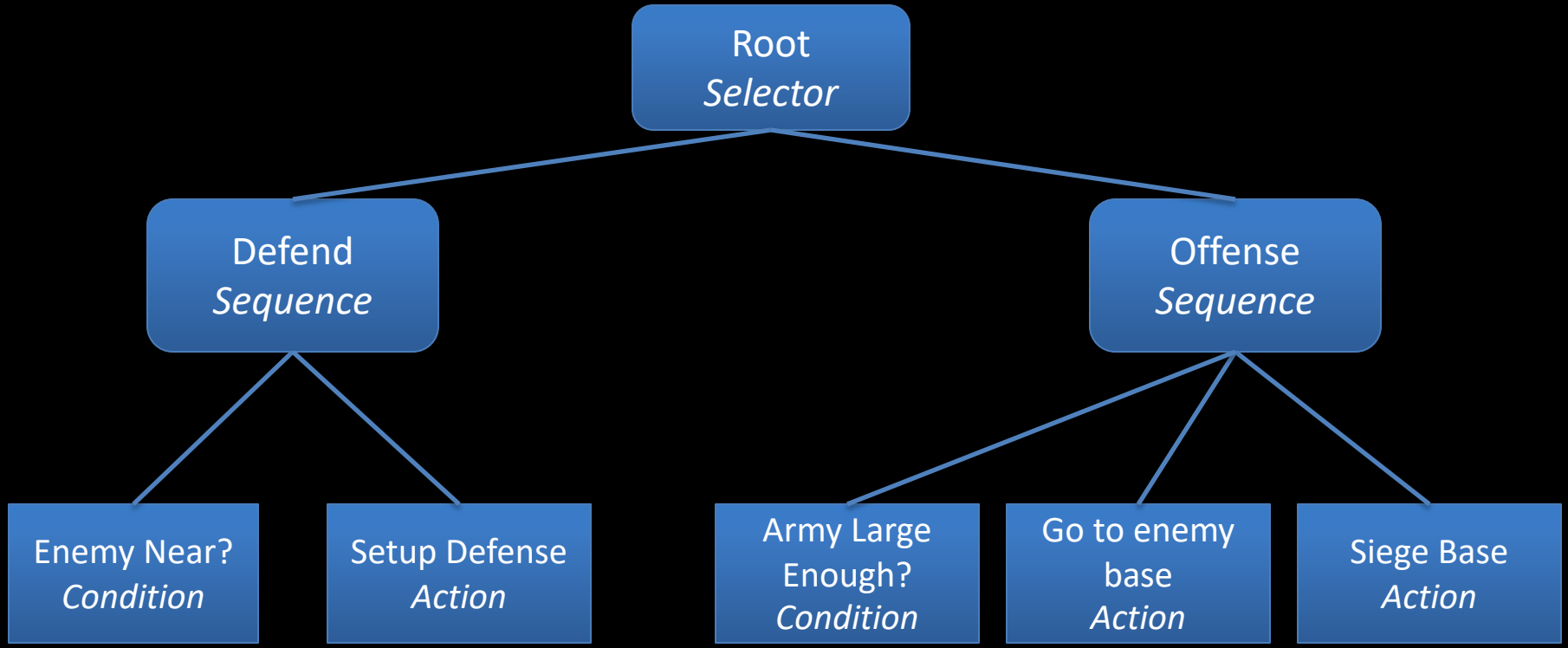
Note about Composites

- Sequences start updating from the previously RUNNING child
  - Previously running child should be left intact after returning, unless the entire sequence was completed
  - Goal is to complete the entire sequence – “I was in the middle of something and should continue where I left off”
- Selectors should always update from the first child
  - Should reset the previously running child if a child before it starts RUNNING
  - Children have priority – “I should always go back to defend my base, even if I’m in the middle of an offensive sequence”

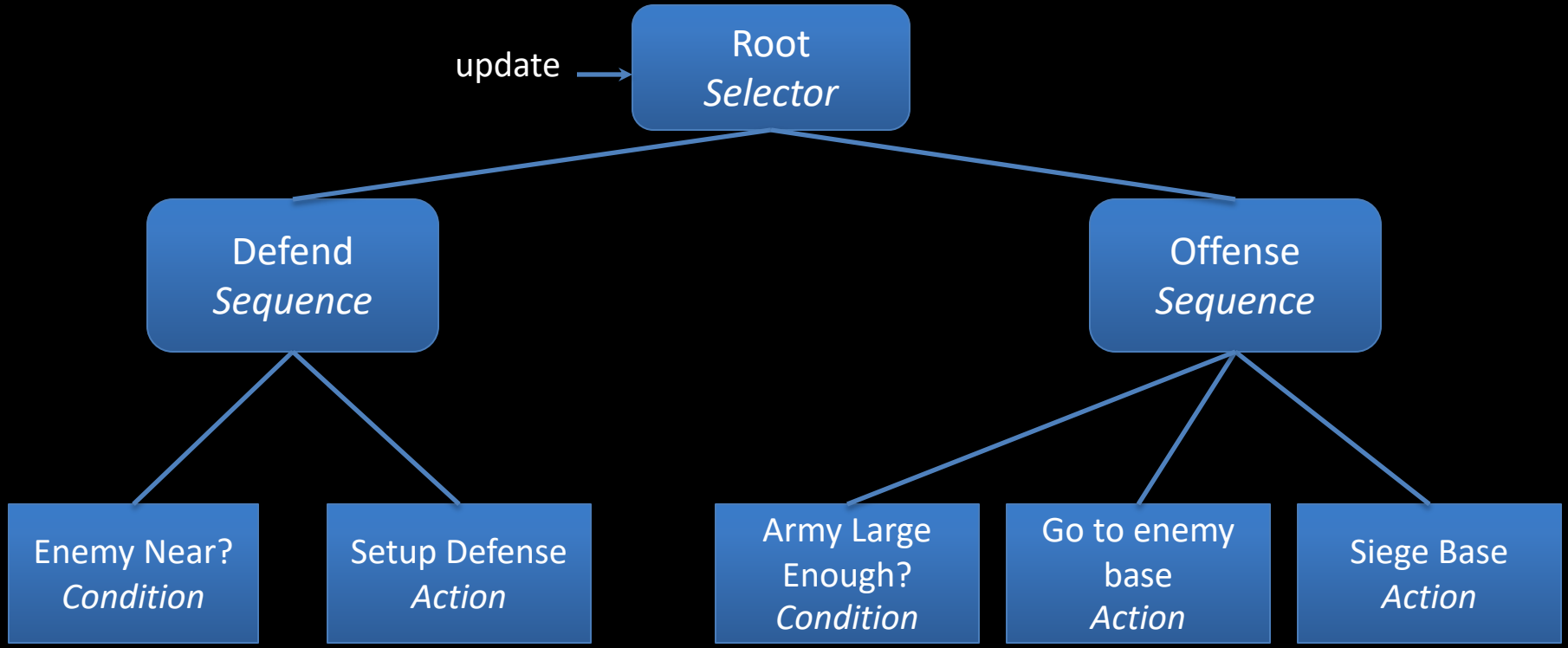
Behavior Trees

QUESTIONS?

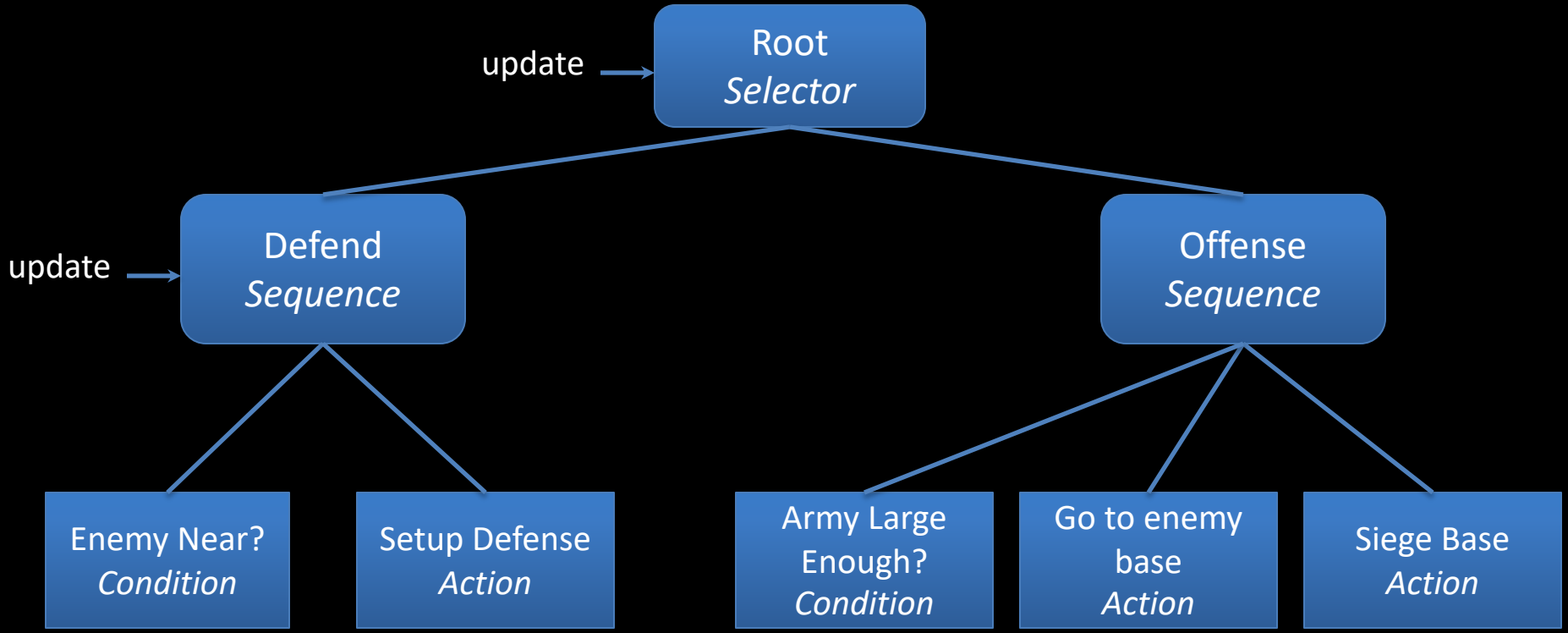
Example


Root
Selector
Defend
Sequence
Offense
Sequence
Enemy Near?
Condition
Setup Defense
Action
Army Large Enough?
Condition
Go to enemy base
Action
Siege Base
Action

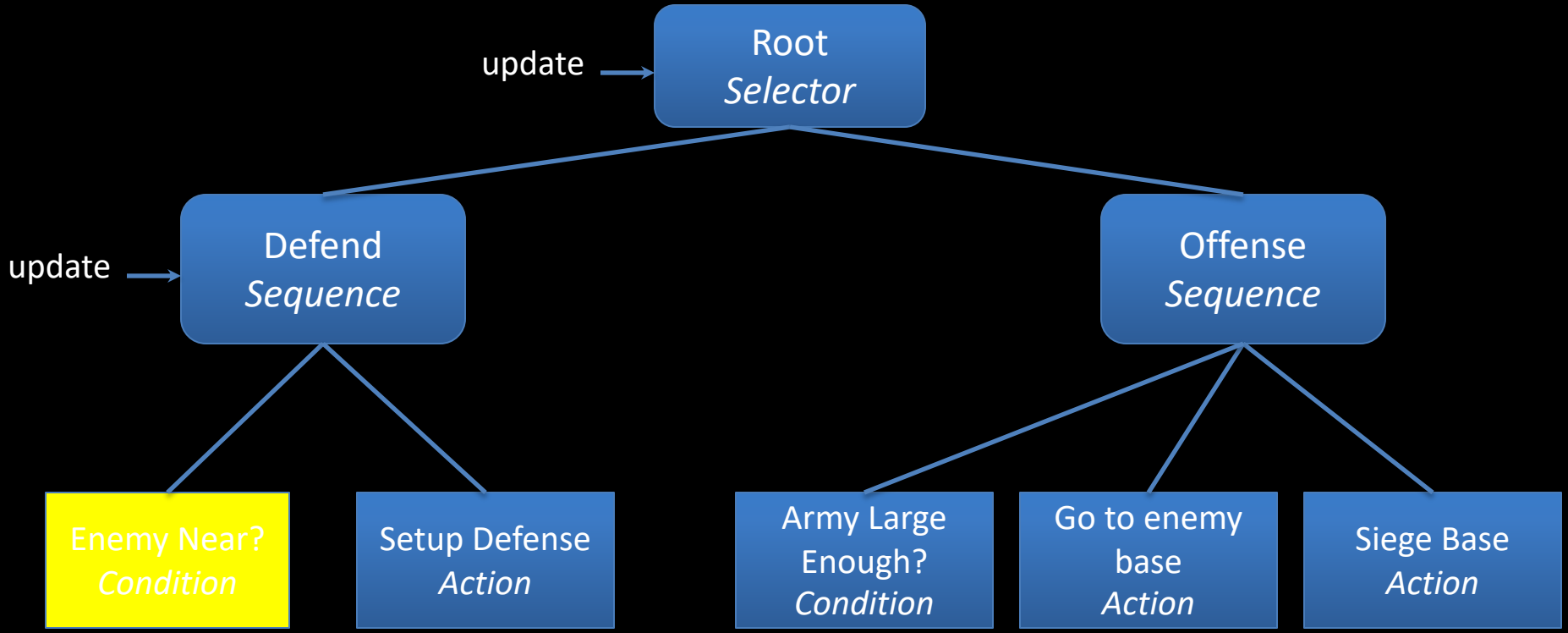
Example


update
Root
Selector
Defend
Sequence
Offense
Sequence
Enemy Near?
Condition
Setup Defense
Action
Army Large Enough?
Condition
Go to enemy base
Action
Siege Base
Action

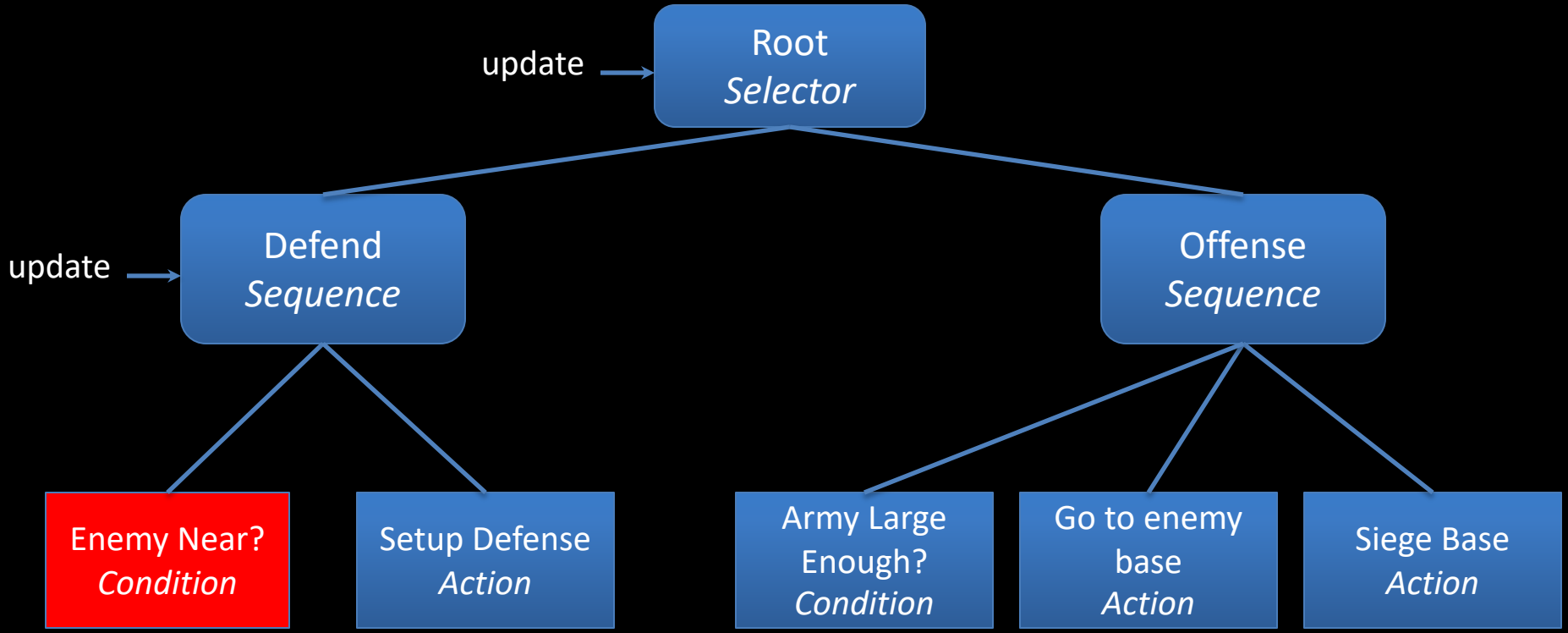
Example


update
Root
Selector
update
Defend
Sequence
Offense
Sequence
Enemy Near?
Condition
Setup Defense
Action
Army Large Enough?
Condition
Go to enemy base
Action
Siege Base
Action

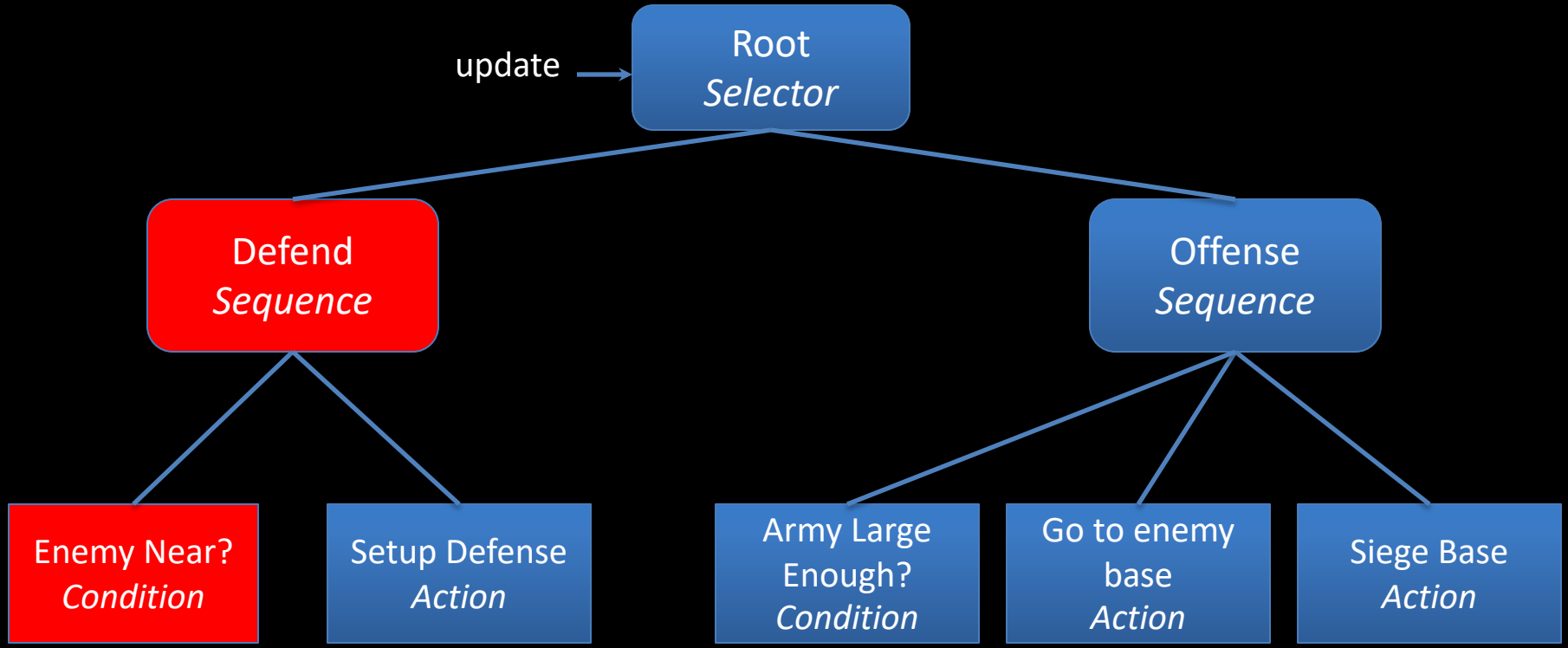
Example


update
Root
Selector
update
Defend
Sequence
Offense
Sequence
Enemy Near?
Condition
Setup Defense
Action
Army Large
Enough?
Condition
Go to enemy
base
Action
Siege Base
Action

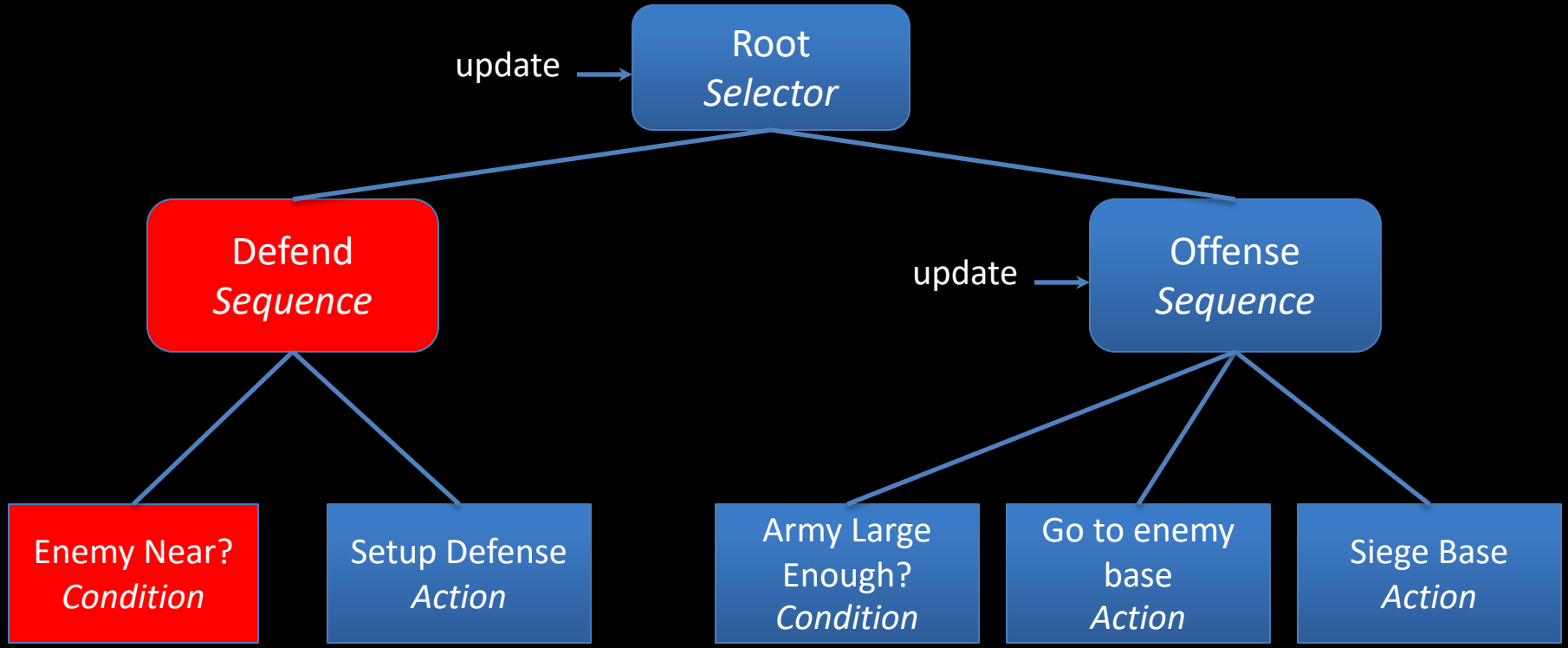
Example


update
Root
Selector
update
Defend
Sequence
Offense
Sequence
Enemy Near?
Condition
Setup Defense
Action
Army Large
Enough?
Condition
Go to enemy
base
Action
Siege Base
Action

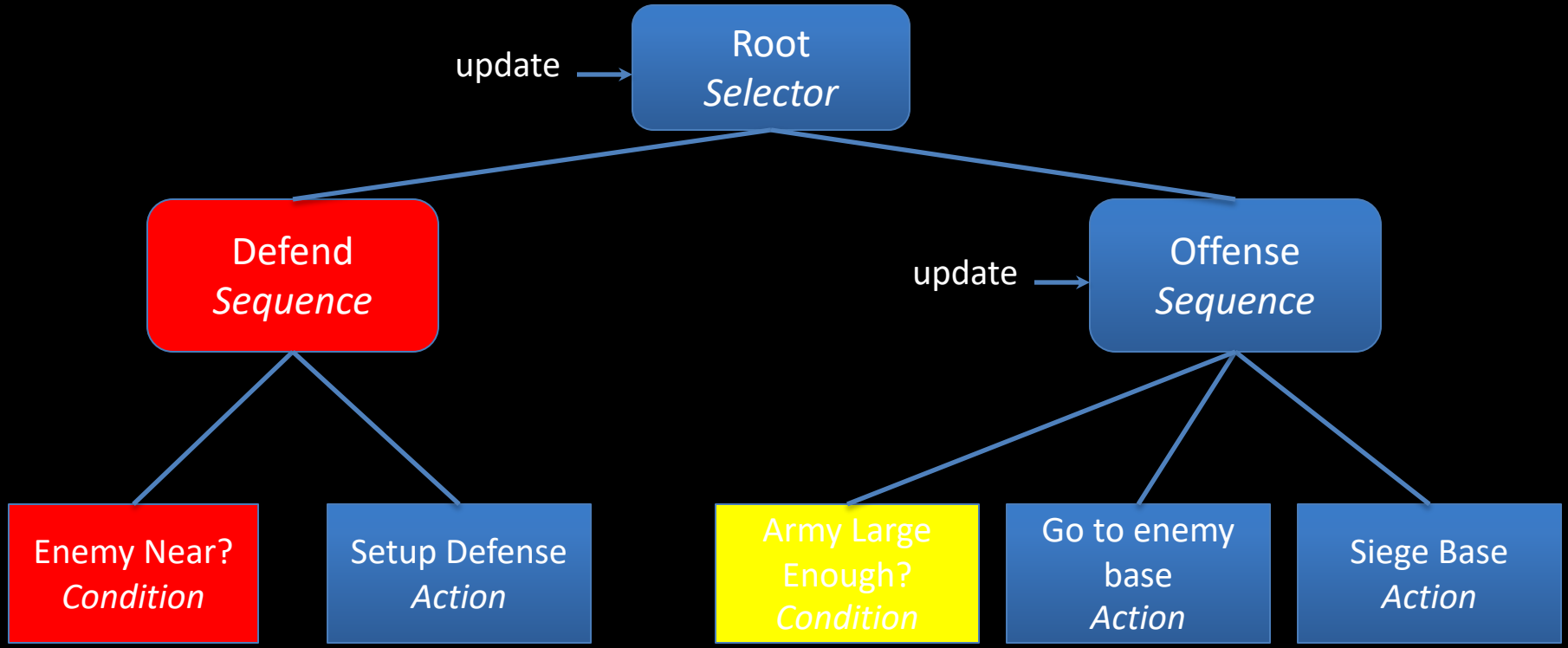
Example


update
Root
Selector
Defend
Sequence
Offense
Sequence
Enemy Near?
Condition
Setup Defense
Action
Army Large Enough?
Condition
Go to enemy base
Action
Siege Base
Action

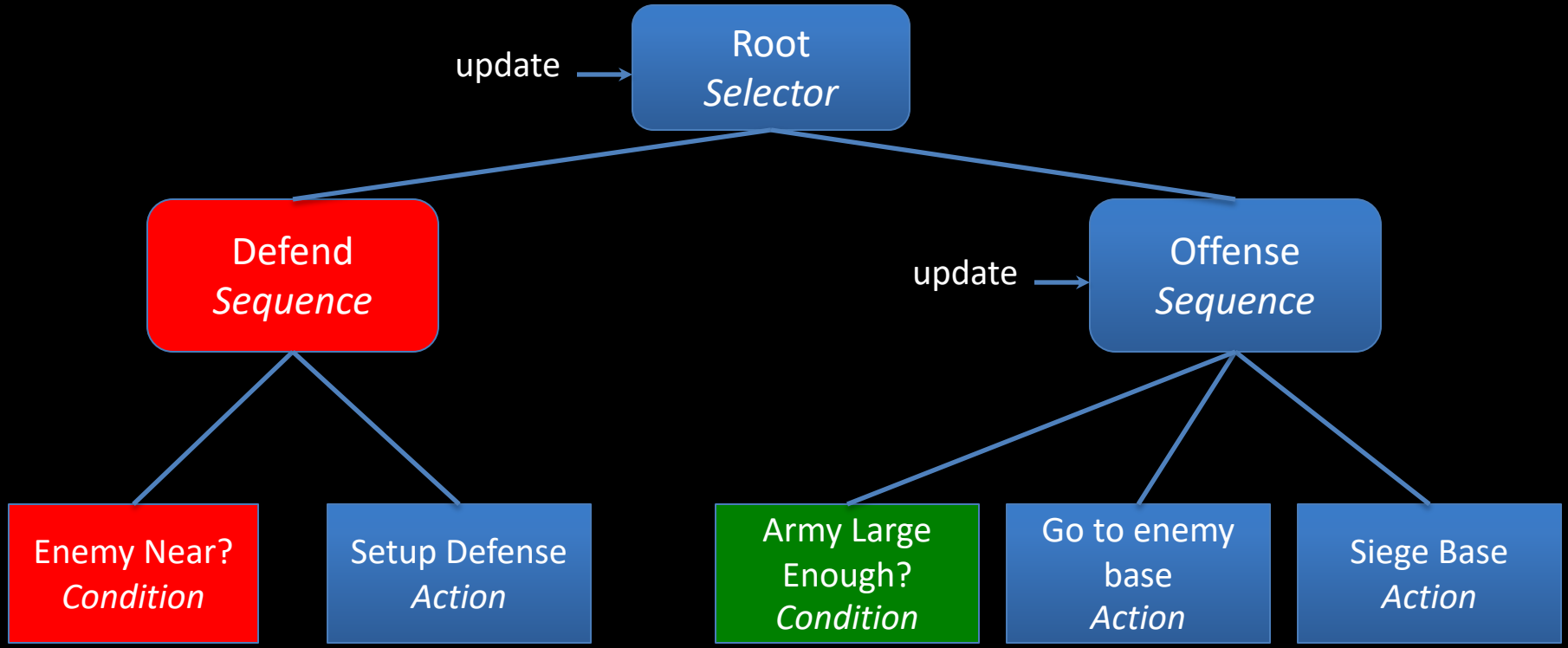
Example


update
Root
Selector
Defend
Sequence
update
Offense
Sequence
Enemy Near?
Condition
Setup Defense
Action
Army Large Enough?
Condition
Go to enemy base
Action
Siege Base
Action

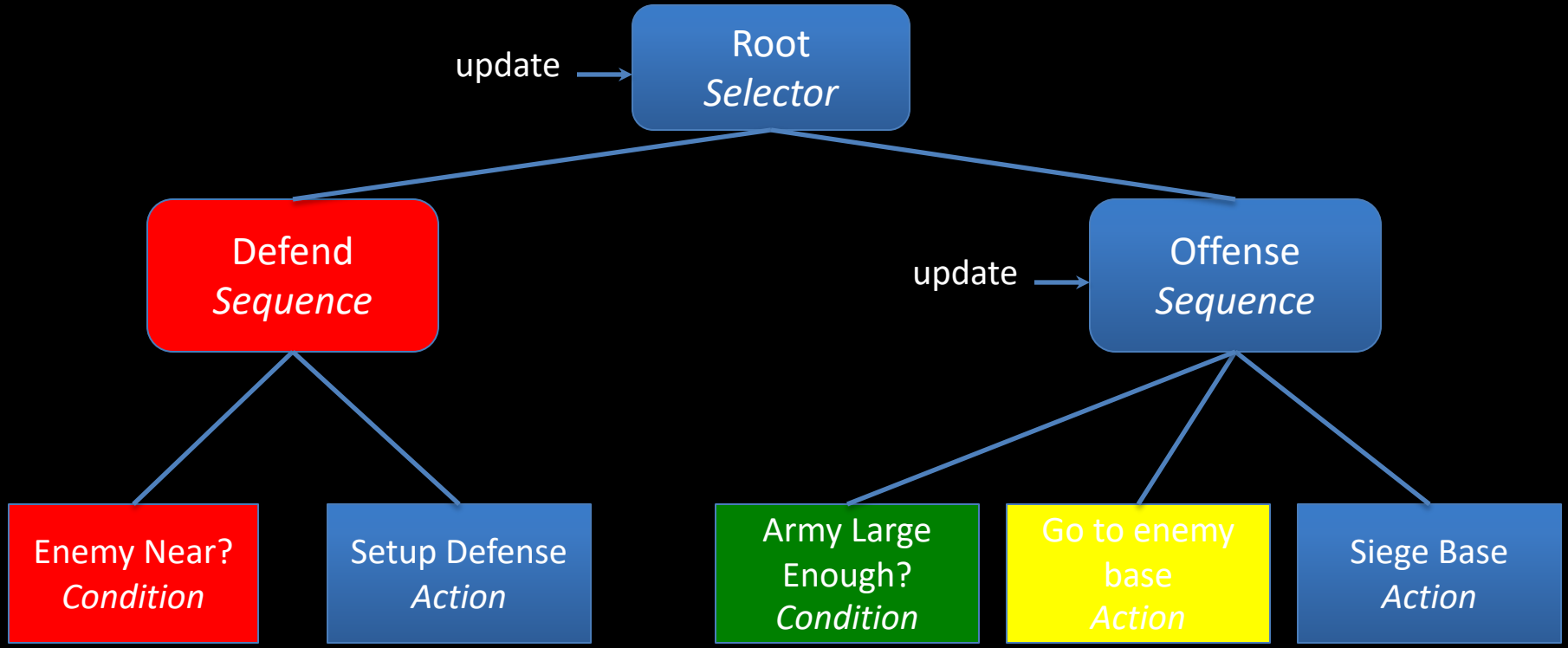
Example


update
Root
Selector
Defend
Sequence
update
Offense
Sequence
Enemy Near?
Condition
Setup Defense
Action
Army Large Enough?
Condition
Go to enemy base
Action
Siege Base
Action

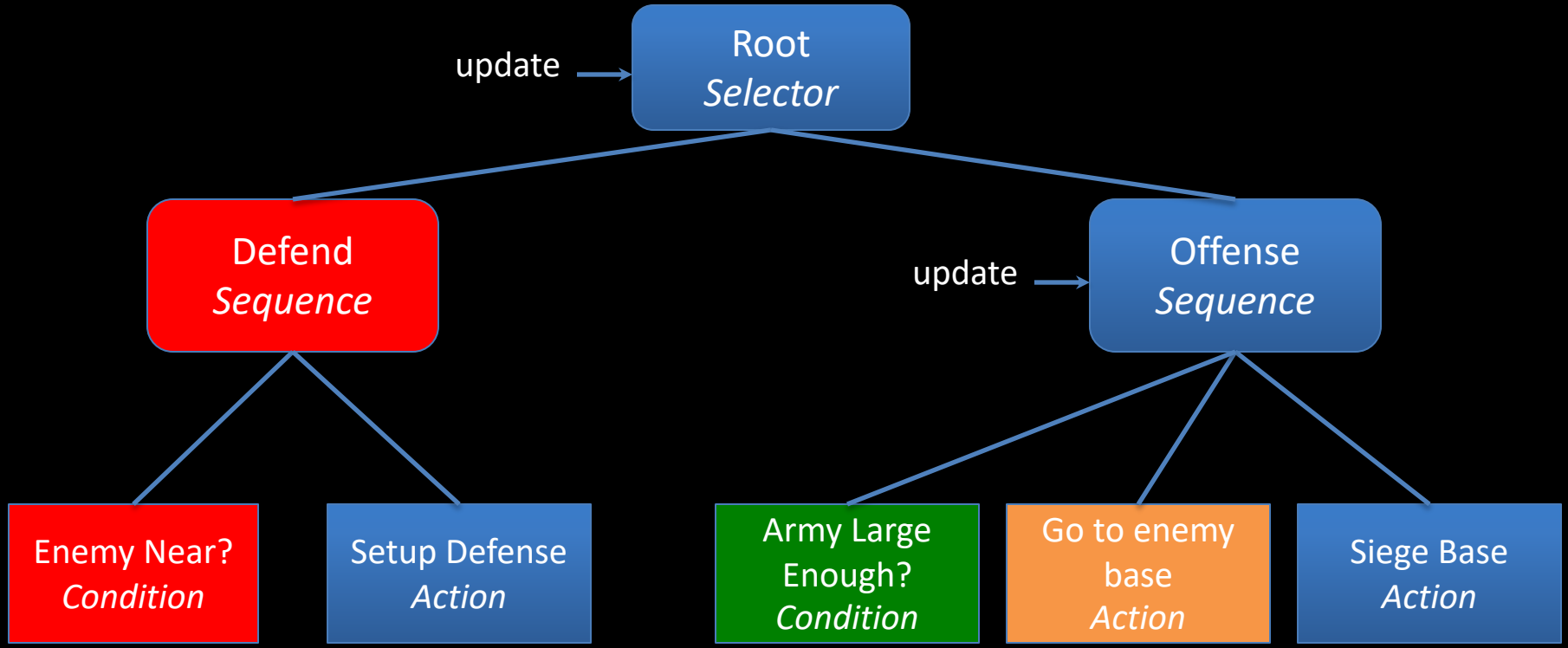
Example


update
Root
Selector
Defend
Sequence
update
Offense
Sequence
Enemy Near?
Condition
Setup Defense
Action
Army Large
Enough?
Condition
Go to enemy
base
Action
Siege Base
Action

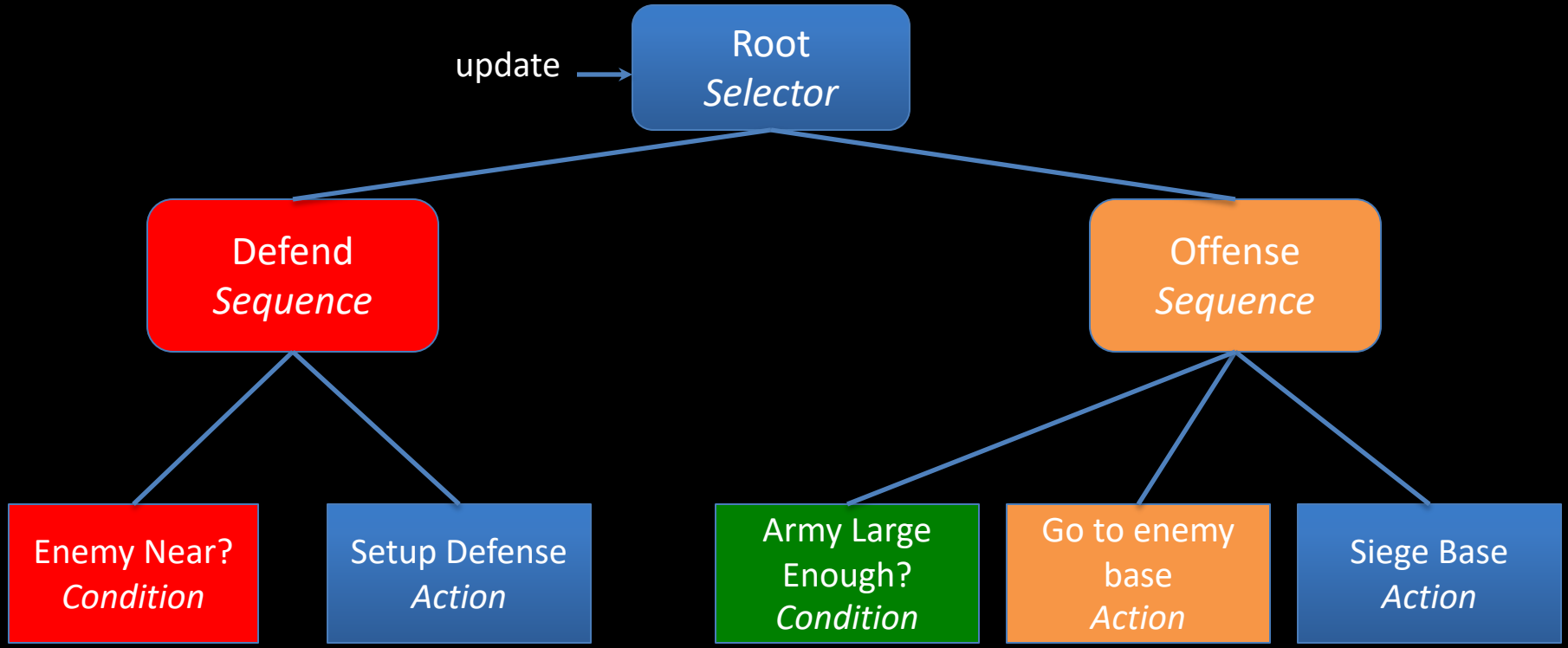
Example


update
Root
Selector
Defend
Sequence
update
Offense
Sequence
Enemy Near?
Condition
Setup Defense
Action
Army Large
Enough?
Condition
Go to enemy
base
Action
Siege Base
Action

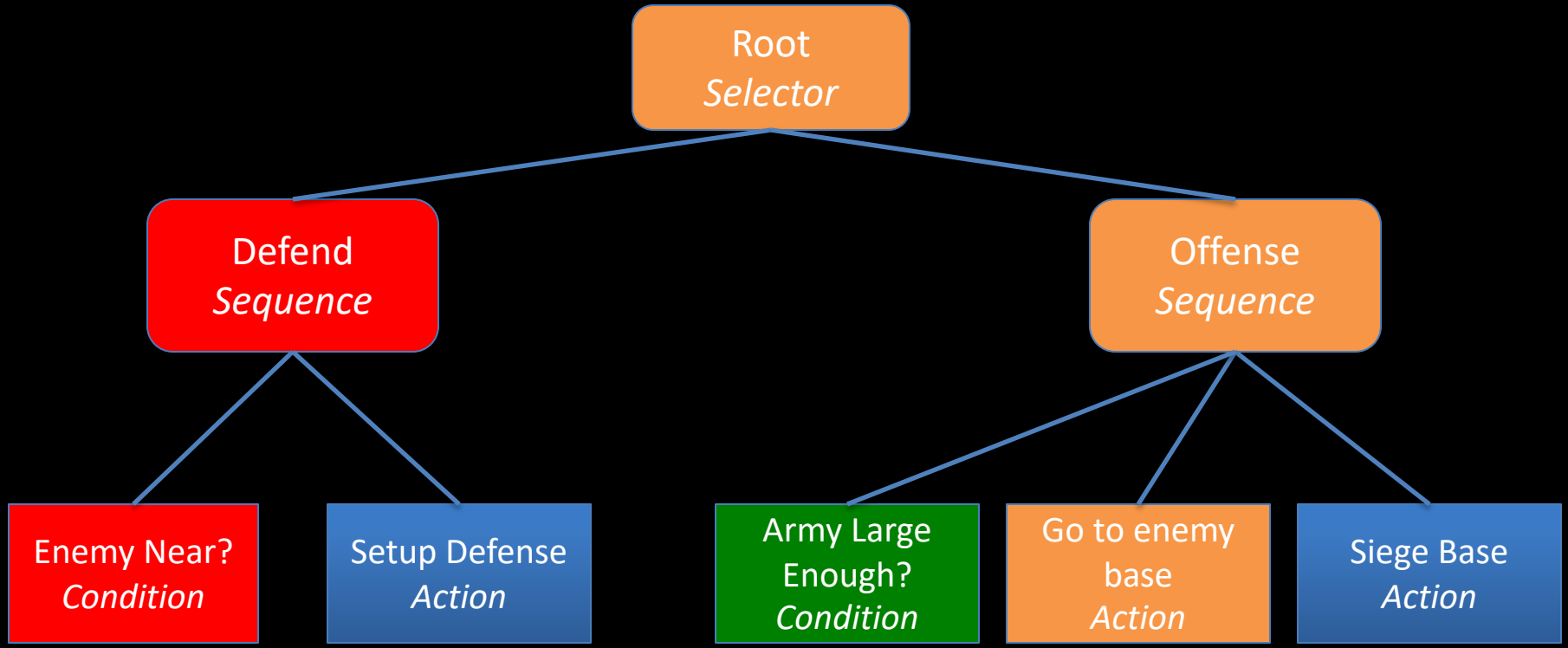
Example


update
Root
Selector
Defend
Sequence
update
Offense
Sequence
Enemy Near?
Condition
Setup Defense
Action
Army Large Enough?
Condition
Go to enemy base
Action
Siege Base
Action

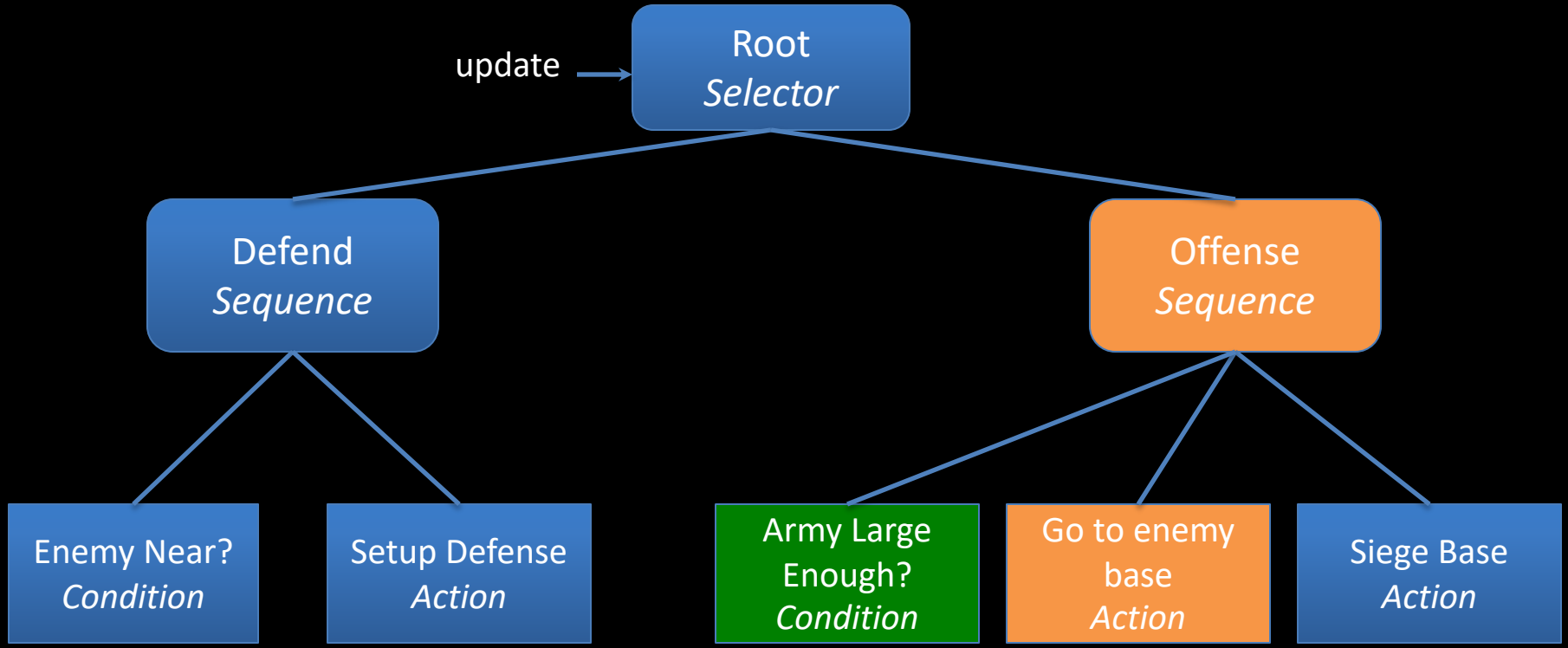
Example


update
Root
Selector
Defend
Sequence
Offense
Sequence
Enemy Near?
Condition
Setup Defense
Action
Army Large Enough?
Condition
Go to enemy base
Action
Siege Base
Action

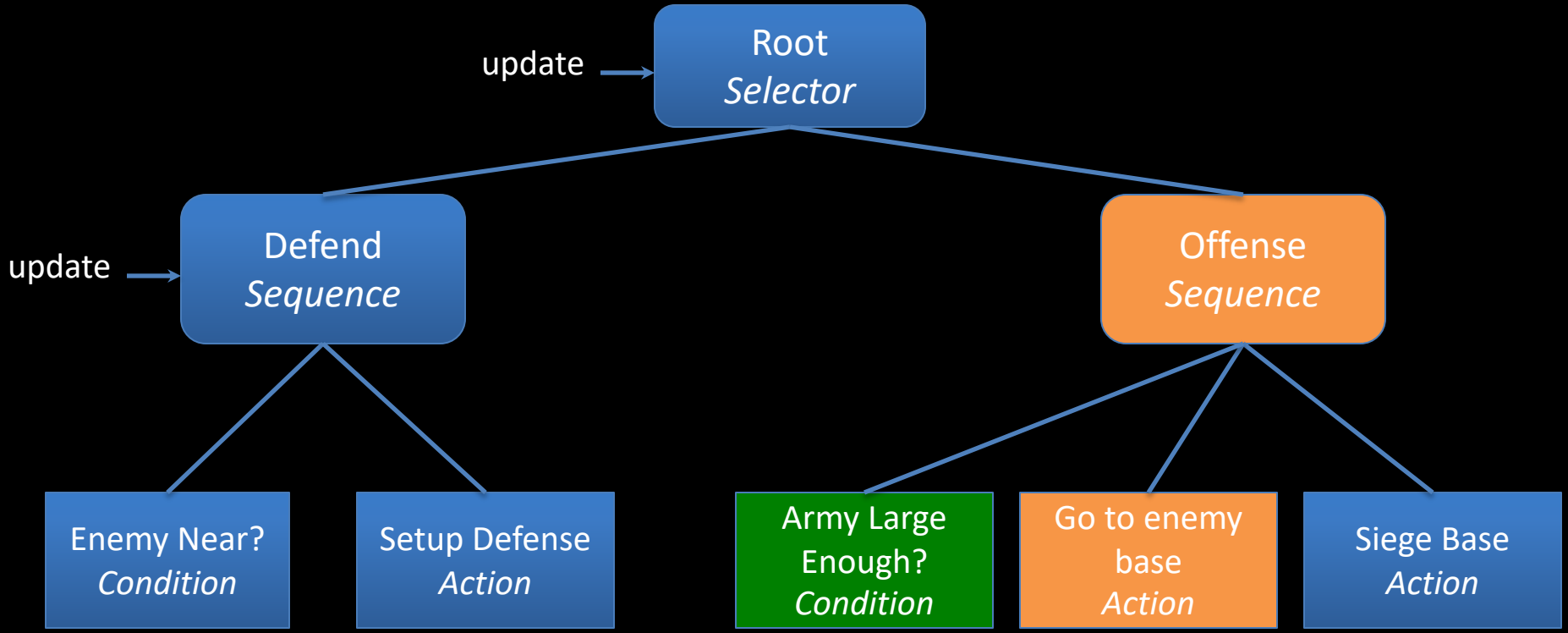
Example


Root
Selector
Defend
Sequence
Offense
Sequence
Enemy Near?
Condition
Setup Defense
Action
Army Large Enough?
Condition
Go to enemy base
Action
Siege Base
Action

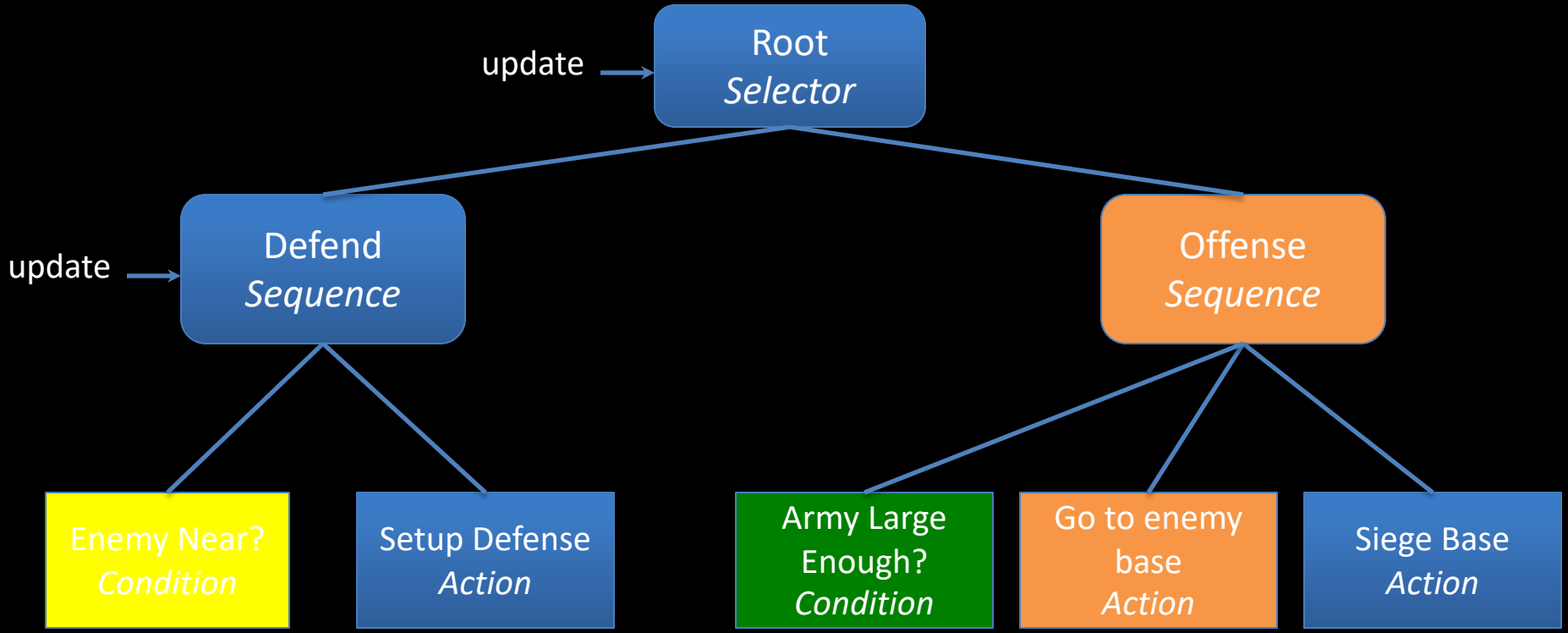
Example


update
Root
Selector
Defend
Sequence
Offense
Sequence
Enemy Near?
Condition
Setup Defense
Action
Army Large Enough?
Condition
Go to enemy base
Action
Siege Base
Action

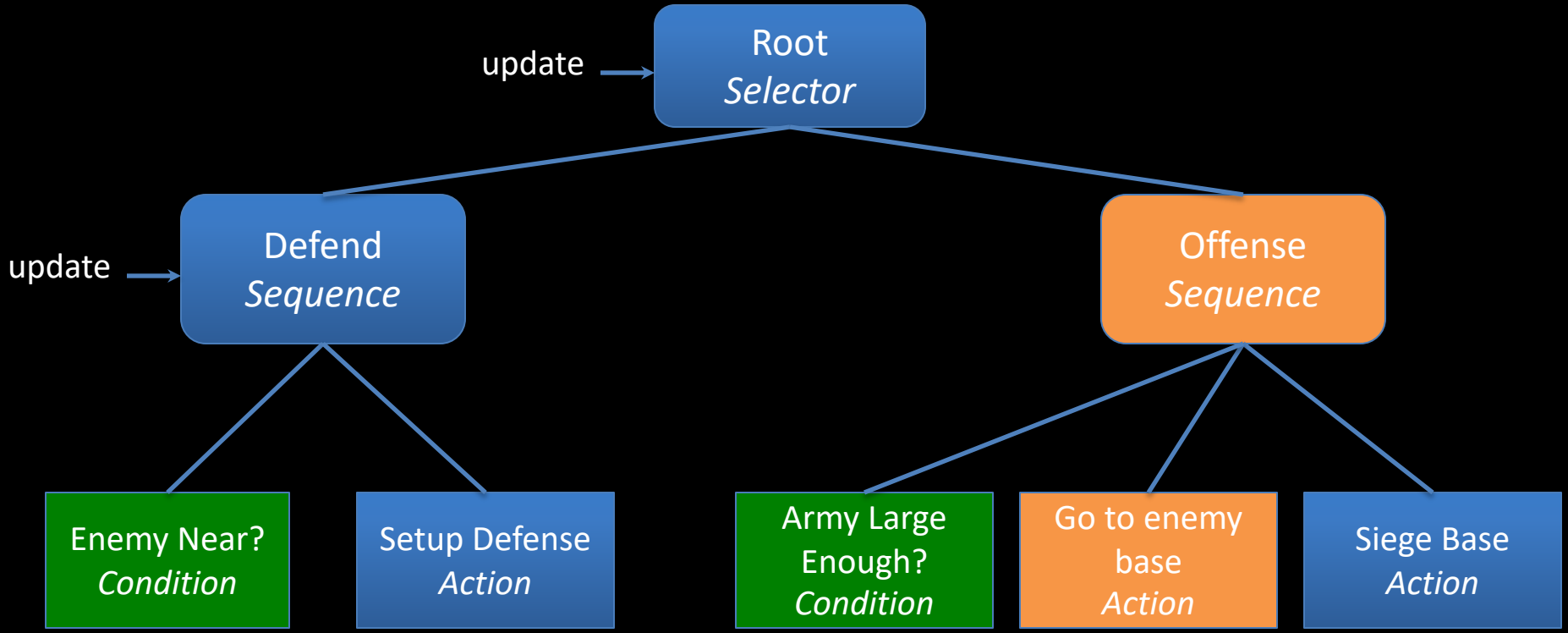
Example


update
Root
Selector
update
Defend
Sequence
Offense
Sequence
Enemy Near?
Condition
Setup Defense
Action
Army Large Enough?
Condition
Go to enemy base
Action
Siege Base
Action

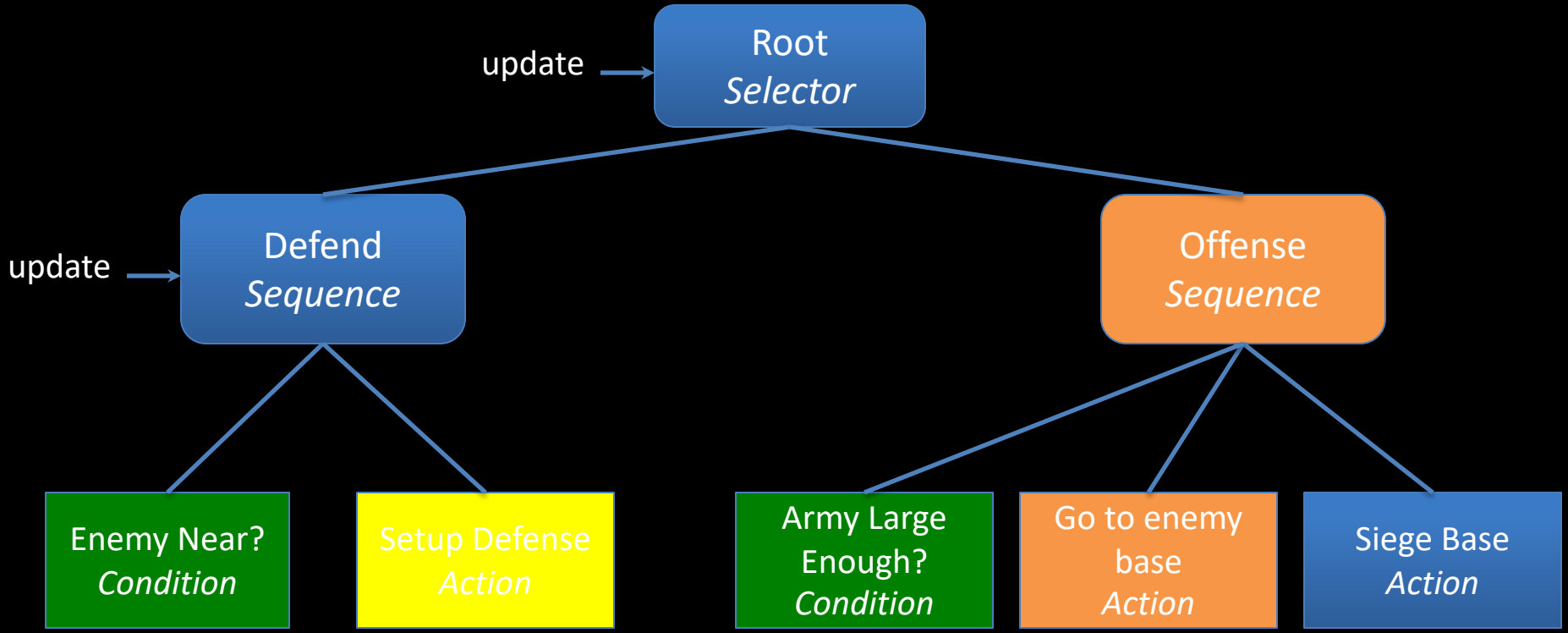
Example


update
Root
Selector
update
Defend
Sequence
Offense
Sequence
Enemy Near?
Condition
Setup Defense
Action
Army Large
Enough?
Condition
Go to enemy
base
Action
Siege Base
Action

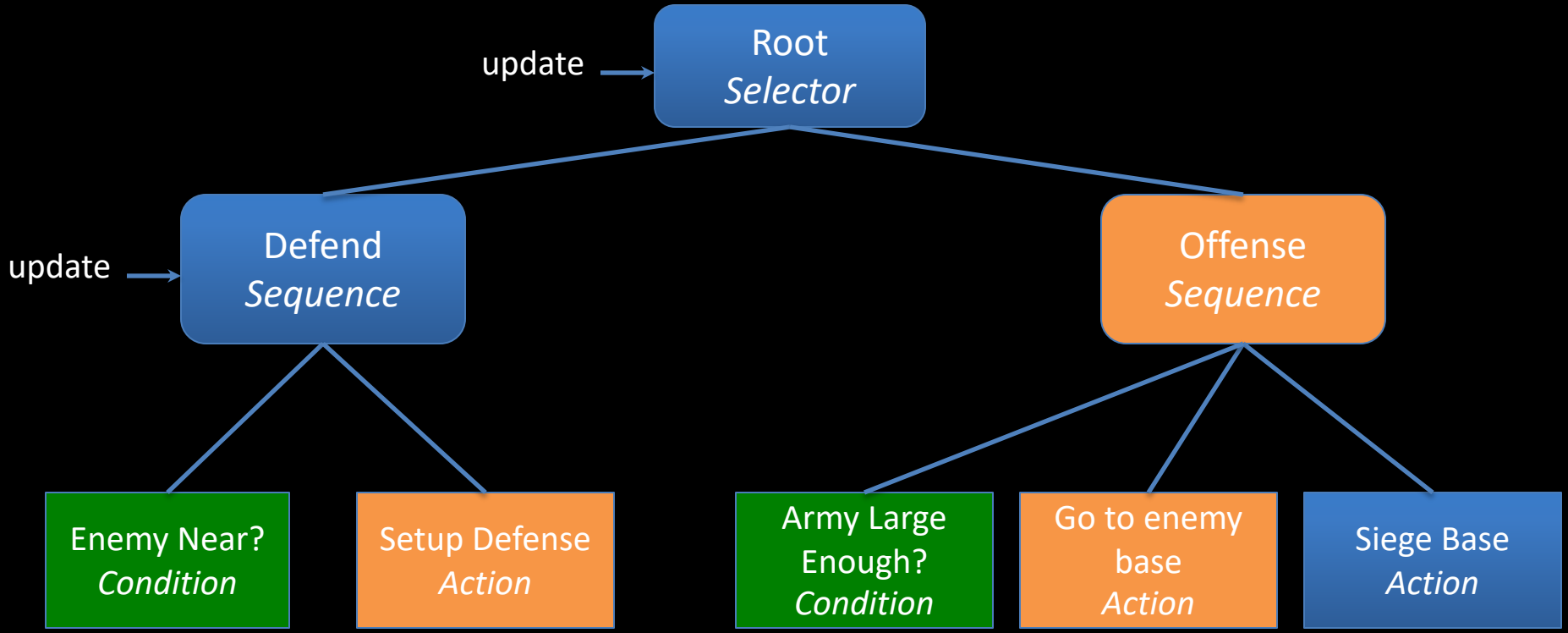
Example


update
Root
Selector
update
Defend
Sequence
Offense
Sequence
Enemy Near?
Condition
Setup Defense
Action
Army Large
Enough?
Condition
Go to enemy
base
Action
Siege Base
Action

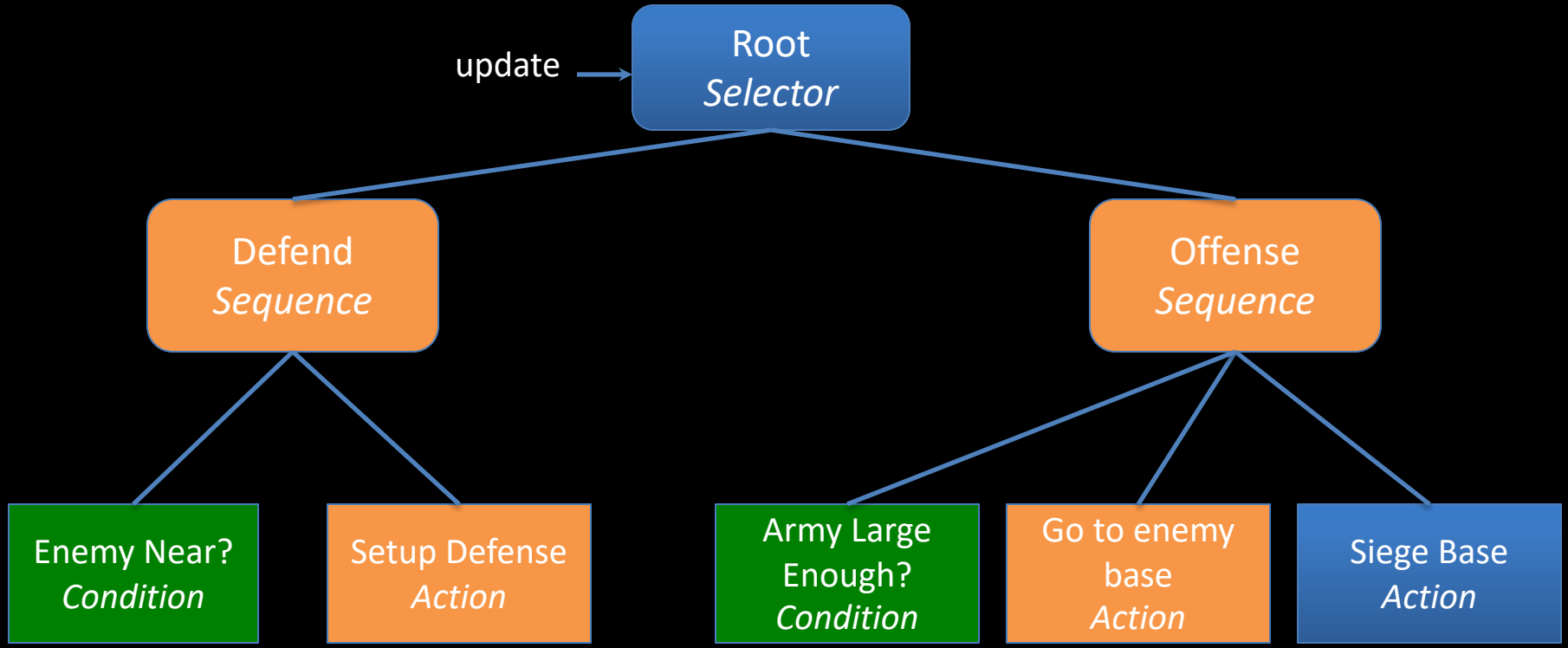
Example


update
Root
Selector
update
Defend
Sequence
Offense
Sequence
Enemy Near?
Condition
Setup Defense
Action
Army Large Enough?
Condition
Go to enemy base
Action
Siege Base
Action

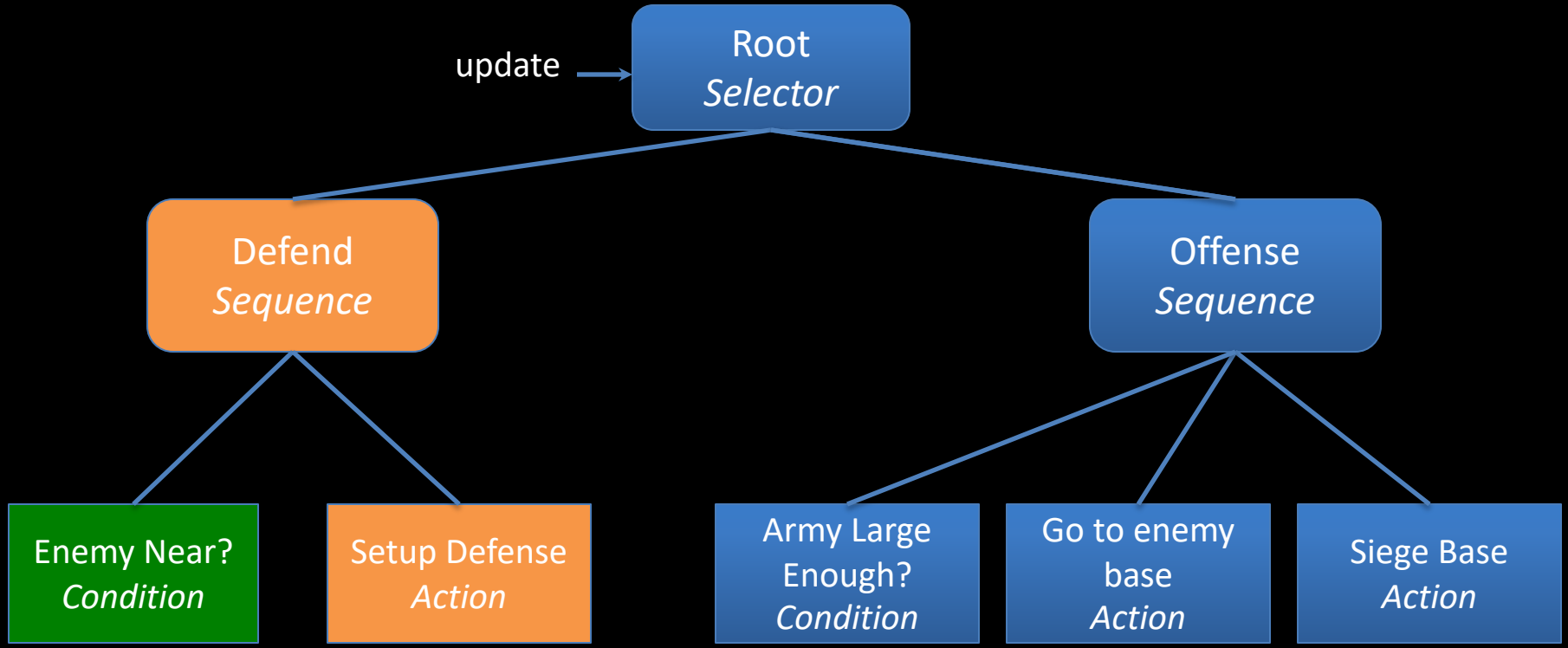
Example


update
Root
Selector
update
Defend
Sequence
Offense
Sequence
Enemy Near?
Condition
Setup Defense
Action
Army Large Enough?
Condition
Go to enemy base
Action
Siege Base
Action

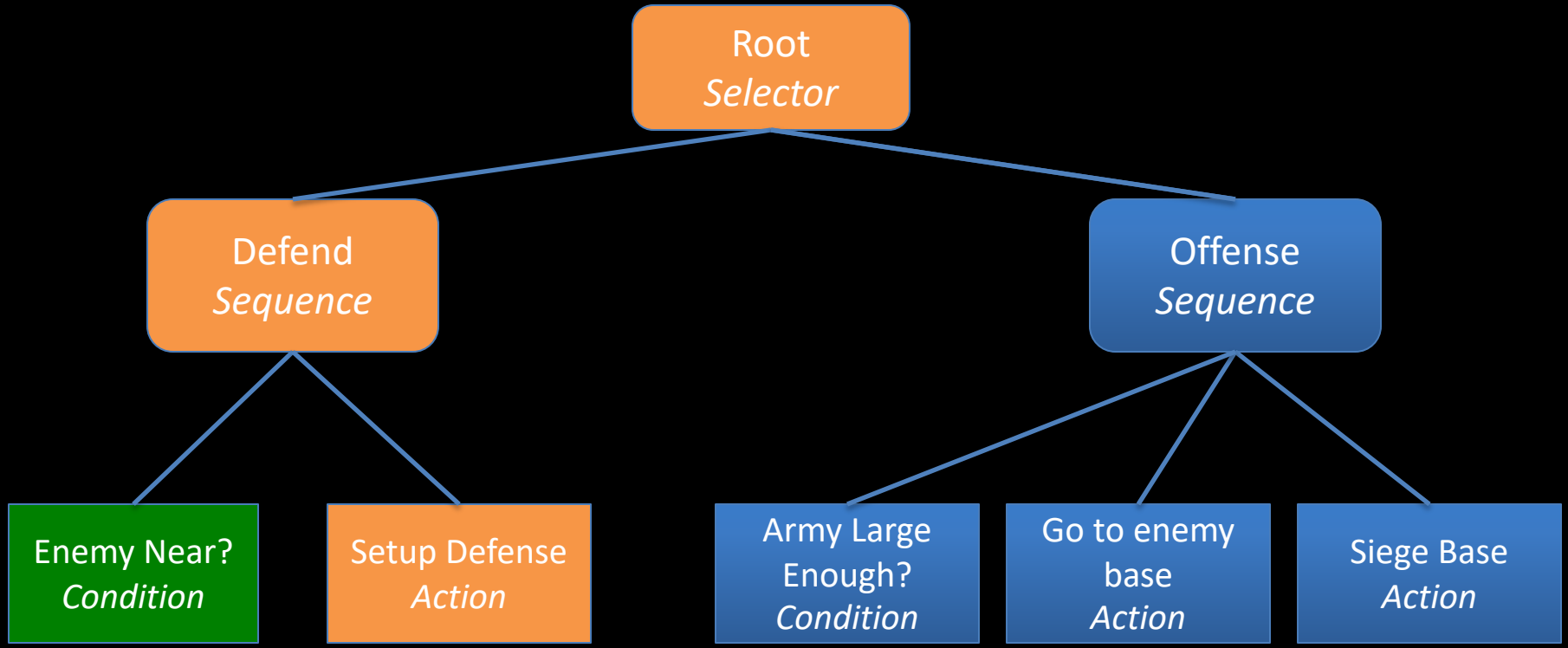
Example


update
Root
Selector
Defend
Sequence
Offense
Sequence
Enemy Near?
Condition
Setup Defense
Action
Army Large Enough?
Condition
Go to enemy base
Action
Siege Base
Action

Example

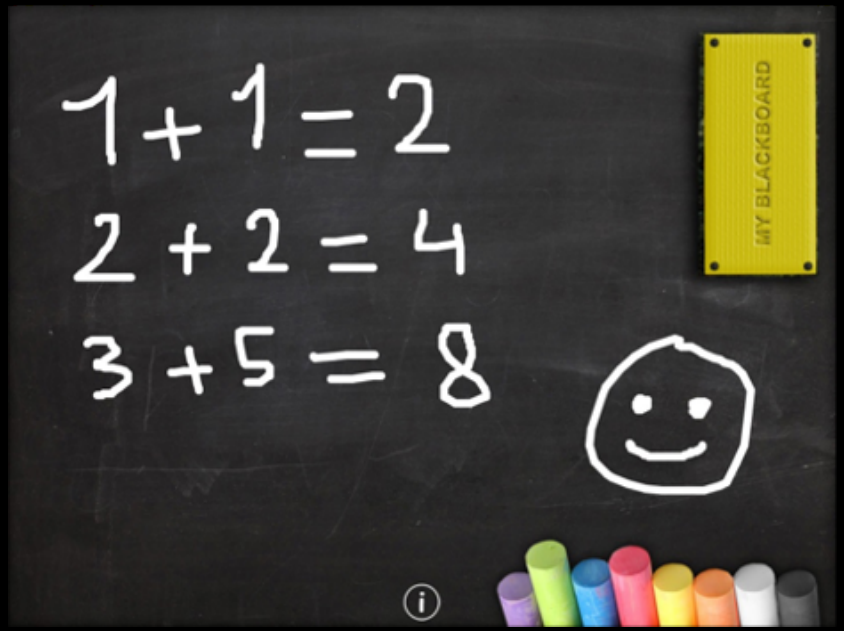

update
Root
Selector
Defend
Sequence
Offense
Sequence
Enemy Near?
Condition
Setup Defense
Action
Army Large Enough?
Condition
Go to enemy base
Action
Siege Base
Action

Example


Root
Selector
Defend
Sequence
Offense
Sequence
Enemy Near?
Condition
Setup Defense
Action
Army Large Enough?
Condition
Go to enemy base
Action
Siege Base
Action

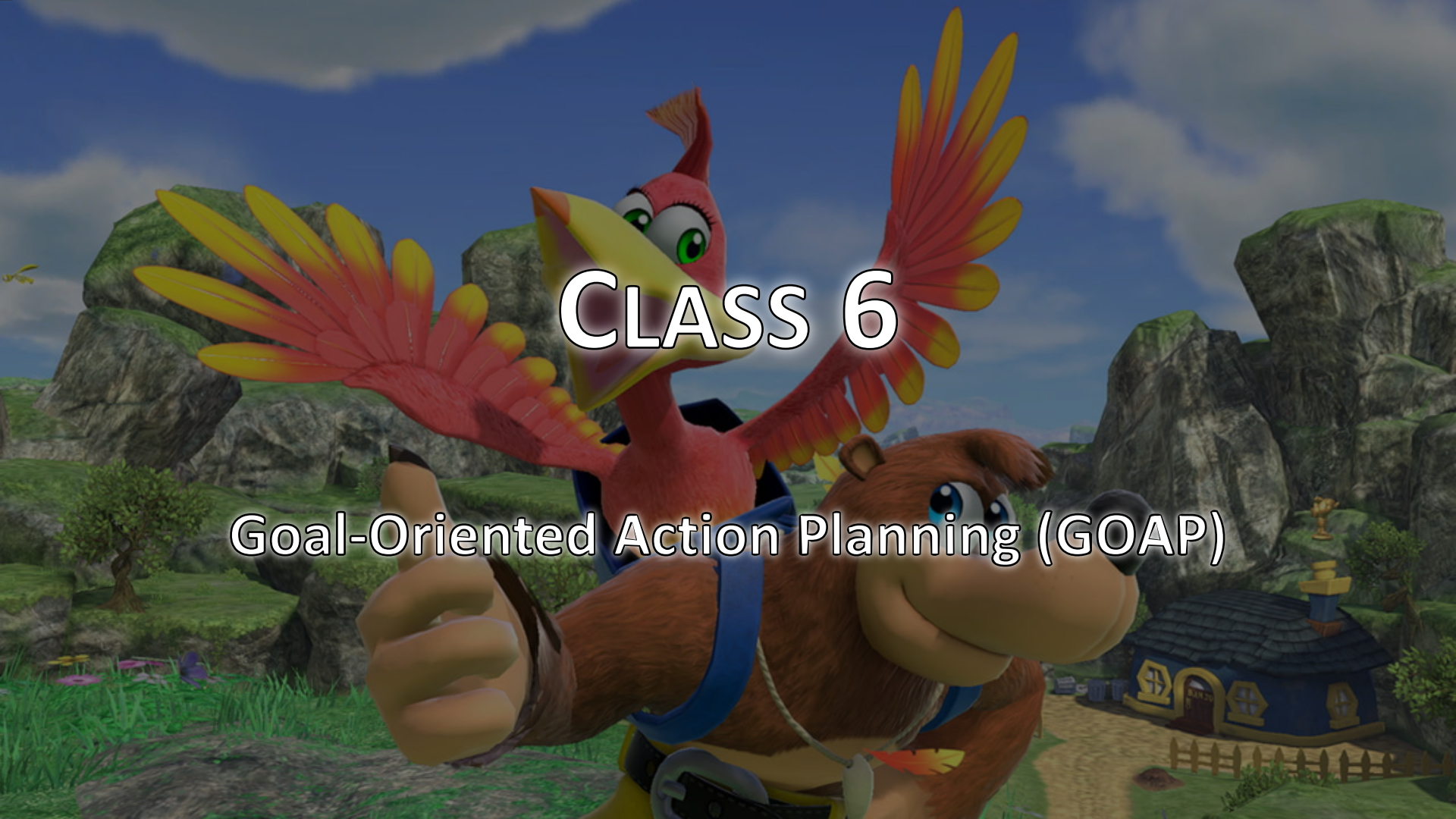
Data Persistence

- Your behavior tree nodes might need to communicate somehow
  - Finding a target, going to the target are separate nodes
- How to share data?
- Blackboard: shared object that holds information, that nodes can write and read from
  - Minimally, a `map<string, ???>`
- Certain groups of nodes can share different blackboards



In Summary

- Interfaces/abstract classes for:
  - BTNode
  - Composite
  - Condition/Action
- Full classes for:
  - Sequence
  - Selector
  - Other wrappers
- Game-specific classes extending Condition/Action

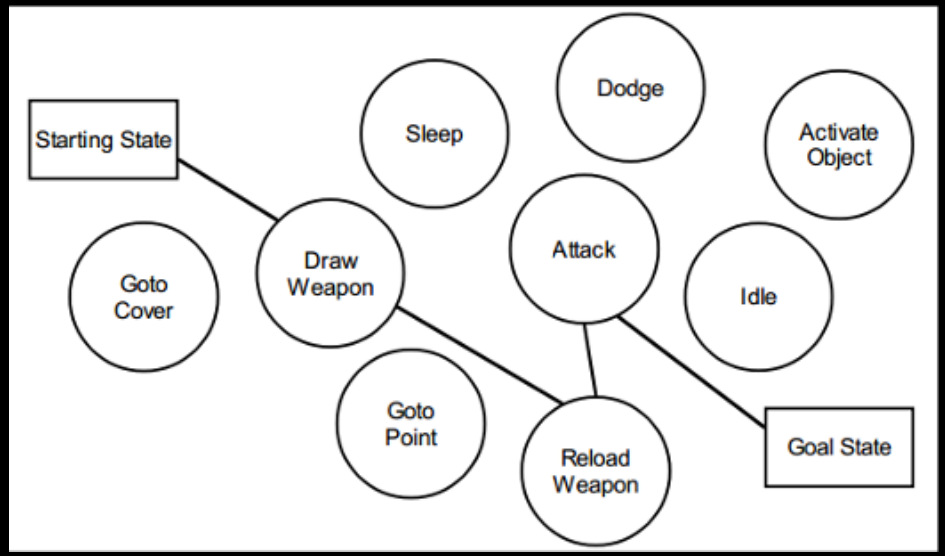

Class 6
Goal-Oriented Action Planning (GOAP)

Issues with BTs

- Behavior trees aren’t perfect
- Lots of enemies
  - Too much work to code each
  - Minor tweaks change a lot of code
- Procedurally generated enemies
  - Behavior trees usually aren’t expressive enough

What is GOAP?

- GOAP is a graph of game states
- We can search over it
  - A*



The Nodes

- Each node is a GameState
- GameStates are probably a map of string tags to booleans or integers
- The tags and their meaning are determined game-side

```
class GameState {
    std::map<std::string, int> m_props;
}
```

The Edges

- Each edge is an Action the AI can take
- Each Action has a cost and a Condition
- Actions also change the GameState

```
class Action {
public:
    virtual void changeState(GameState &s) = 0;

private:
    std::vector<Condition *> m_conditions;
    float m_cost;
}
```

Planning

- Goal
  - Generate a plan or “path” of actions
  - This plan should take you from start state to end state
- Just use A* !

Planning contd.

- Start at a state
- Add neighboring states to priority queue
  - Go through all actions
  - All actions whose conditions are true from the current state are allowed
  - Generate a neighbor for each by applying the corresponding action to a copy of the game state
- Pop lowest cost state from priority queue
- Continue
- Return “path” or list of actions that took you from start to end state

Actions

- Just like behavior trees, GOAP has actions
- Actions are much simpler in GOAP
  - Change one or more of the tags in the game state

Conditions

- Just like behavior trees, GOAP has conditions
- Conditions are also much simpler
  - Return true or false
  - Determined entirely by GameState

Conditions

```
class Condition {
public:
	virtual bool is_met(GameState &s) = 0;
}
```

GOAP

- The game defines a start state based on the current game world
- The game also defines a goal (Condition)
- Once the search is done, you need to map the list of actions to some real game effect
- Usually only the first action is executed before GOAP is run again
  - The action might not be completed before a new plan is generated
  - E.g., following the player

Goal Oriented Action Planning

QUESTIONS?

Problems

- Depending on the Actions available, GOAP can generate an infinite graph without any goal states
- This can be handled by any of the following:
  - Allow each action to be used once/max # of times
  - Specify a maximum cost

Problems

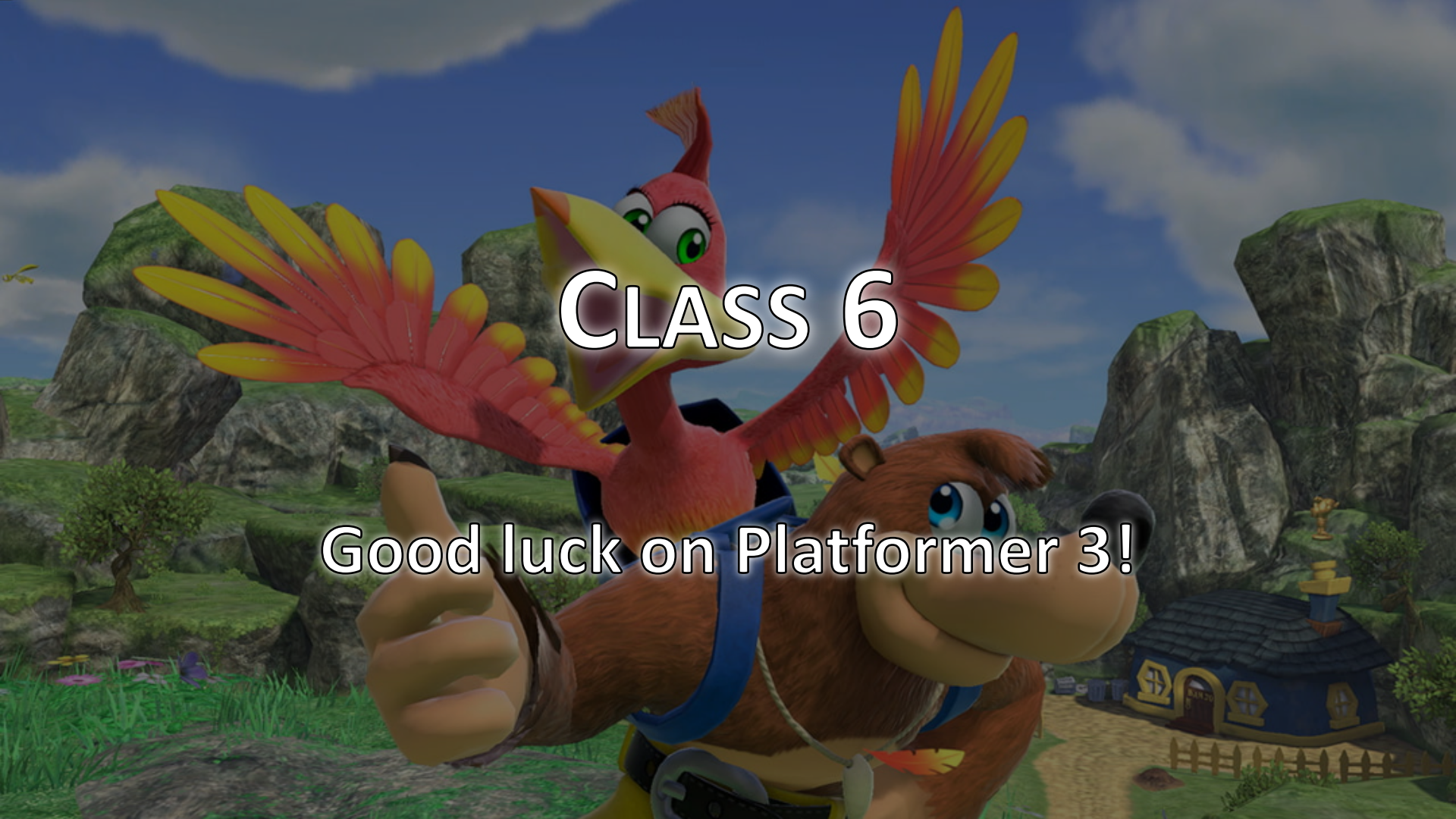
- With lots of actions and a distant goal, GOAP can be really slow
- GOAP is best used to solve small problems

Problems

- GOAP optimizes over a single parameter (time, cost, etc.)
- GOAP is good for short, discrete problems:
  - Which combo should I use?
  - Which route should I take?
- GOAP is bad for long-term, strategic problems:
  - How do I optimize my economy?
  - Which item will maximize my options next level?

Mix and Match

- Behavior trees and GOAP don't have to be mutually exclusive
- Behavior tree can determine the strategy (setting up which actions are available, how much each is weighted, what the goal is, etc.)
- GOAP can determine the plan to execute that strategy
- Behavior tree turns that plan into concrete actions
  - e.g., sequence


Class 6
Good luck on Platformer 3!