UNIVERSITÀ DEGLI STUDI
DI MILANO

Decision Making Basics

A.I. for Video Games

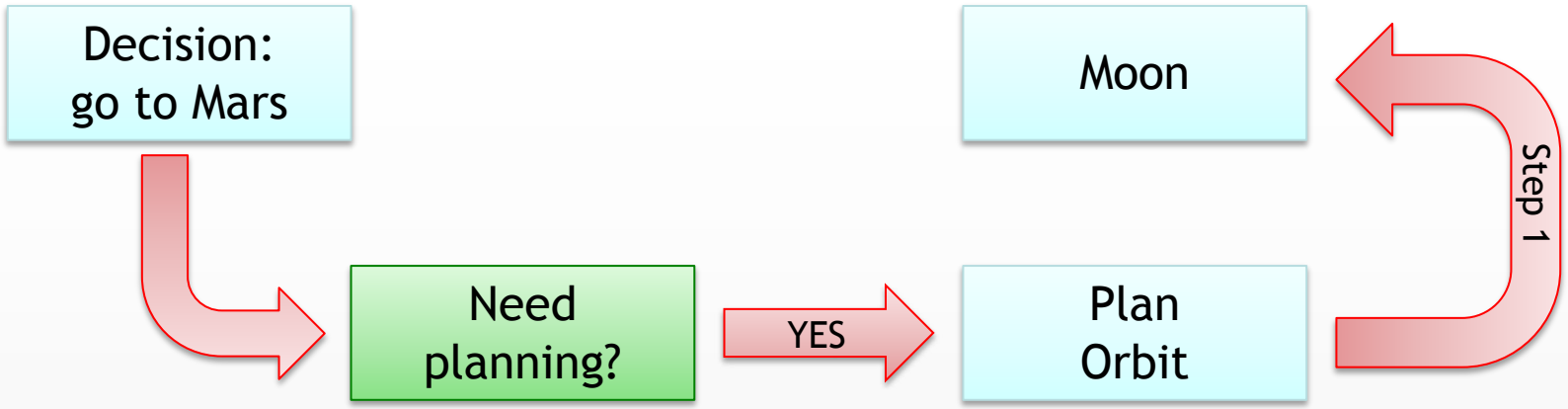
Davie Gadia (gadia@di.unimi.it) and Dario Maggiorini (dario@di.unimi.it)

Playlab fOr inNovation in Games

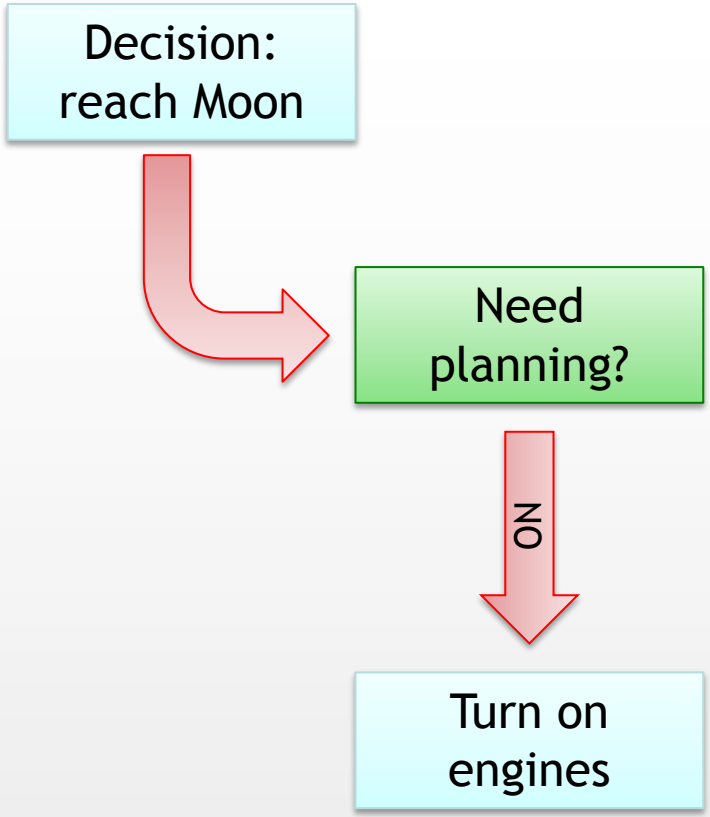
Relationship Between Planning and Decision

- So far, we saw pathfinding ... which is a kind of planning
  - We understand what to traverse in order to reach a destination
- Pathfinding is NOT a decision-making process
- To decide to go somewhere is a completely different thing from planning how to go there
  - The actual decision process is about picking the destination

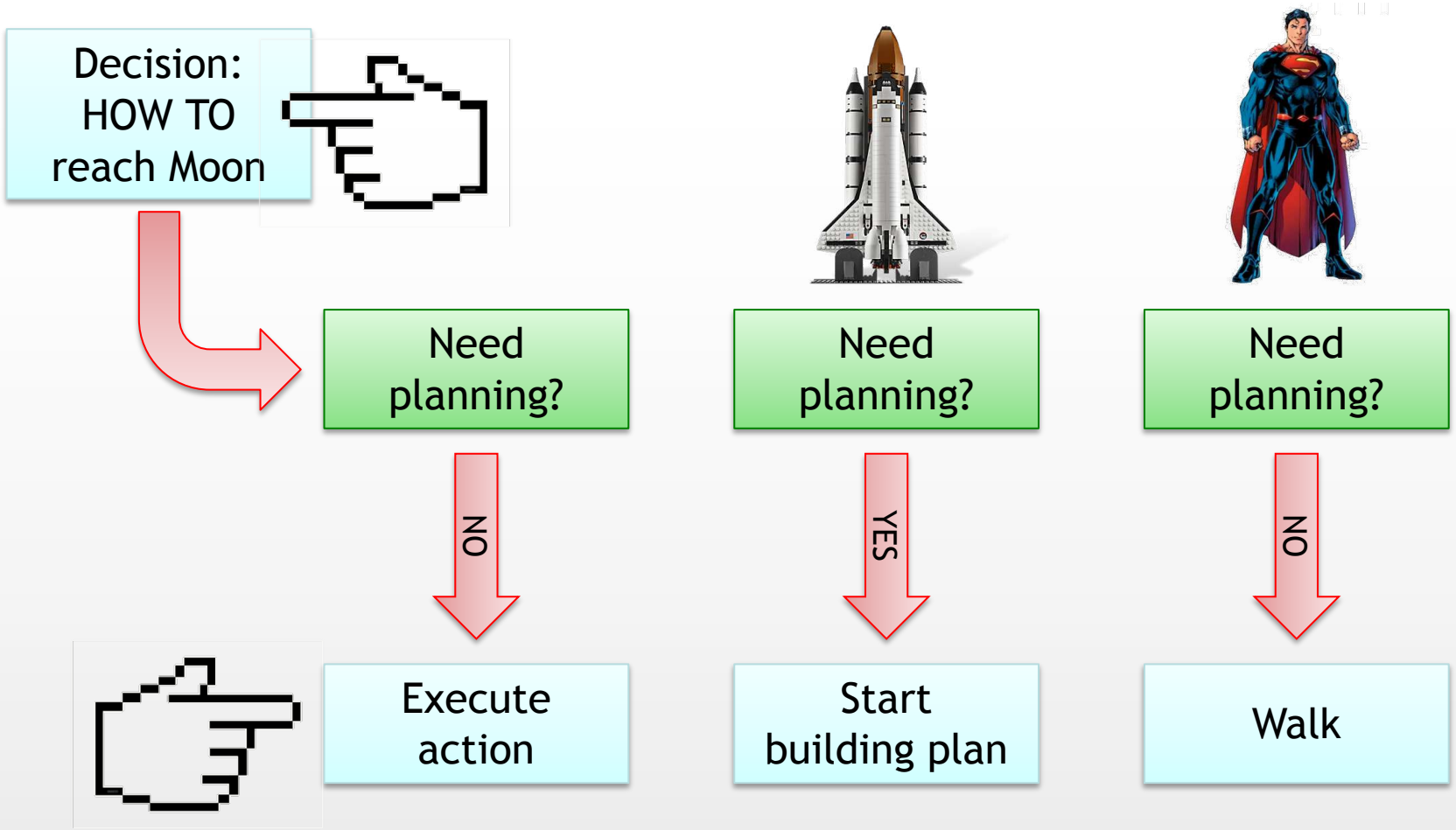
A Life Based on Decisions



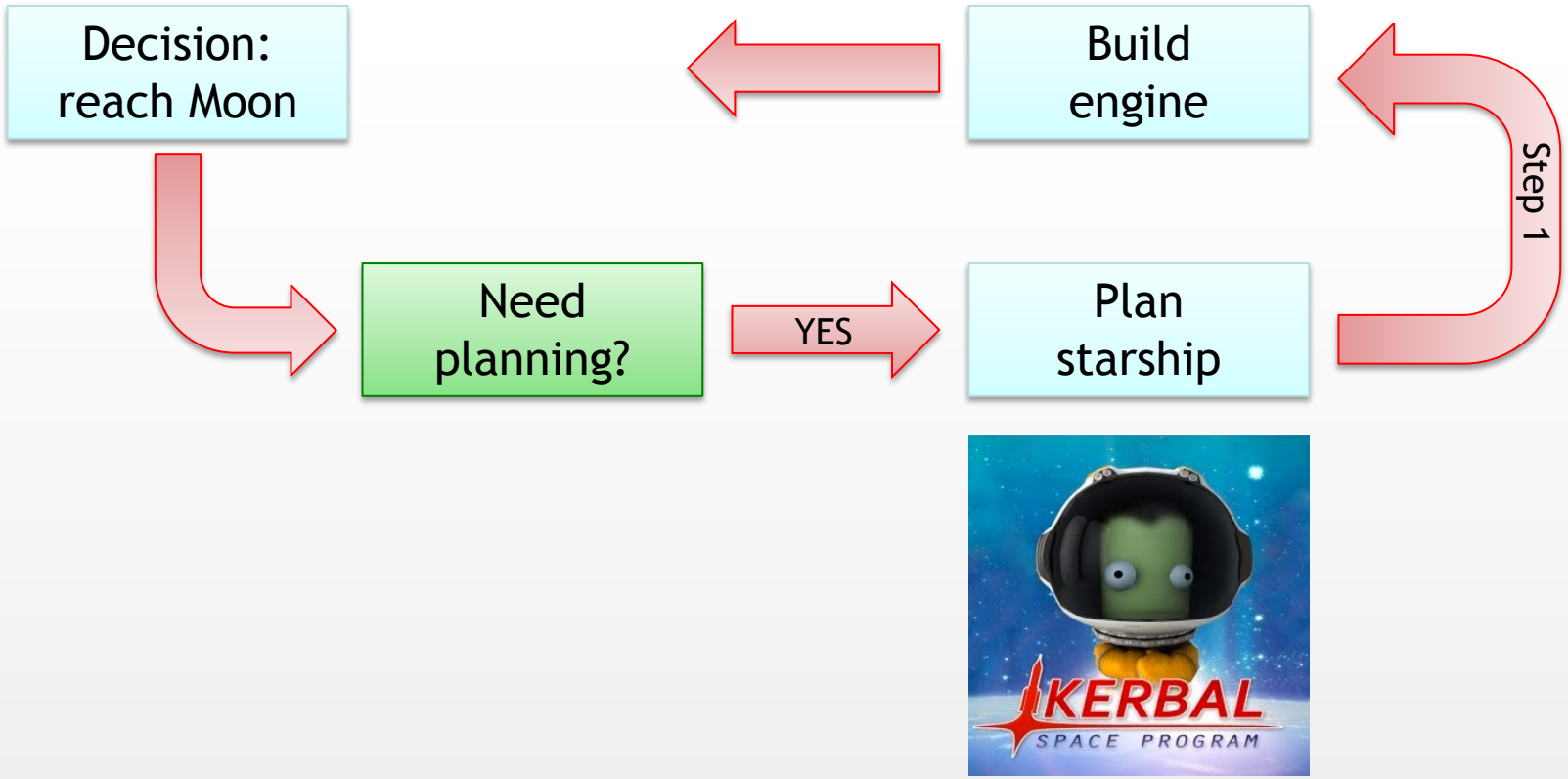
A Life Based on Decisions



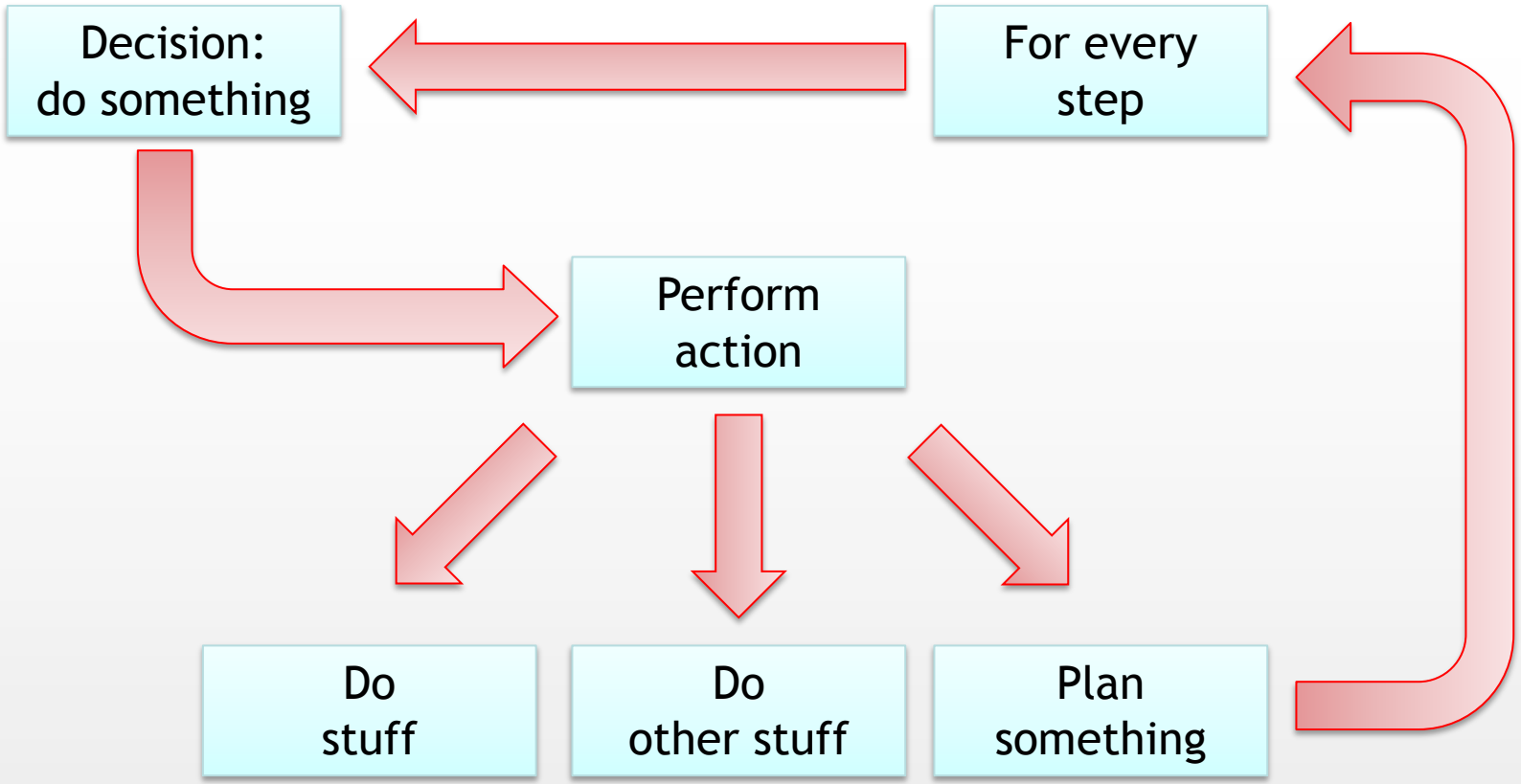
A Life Based on Decisions



A Life Based on Decisions



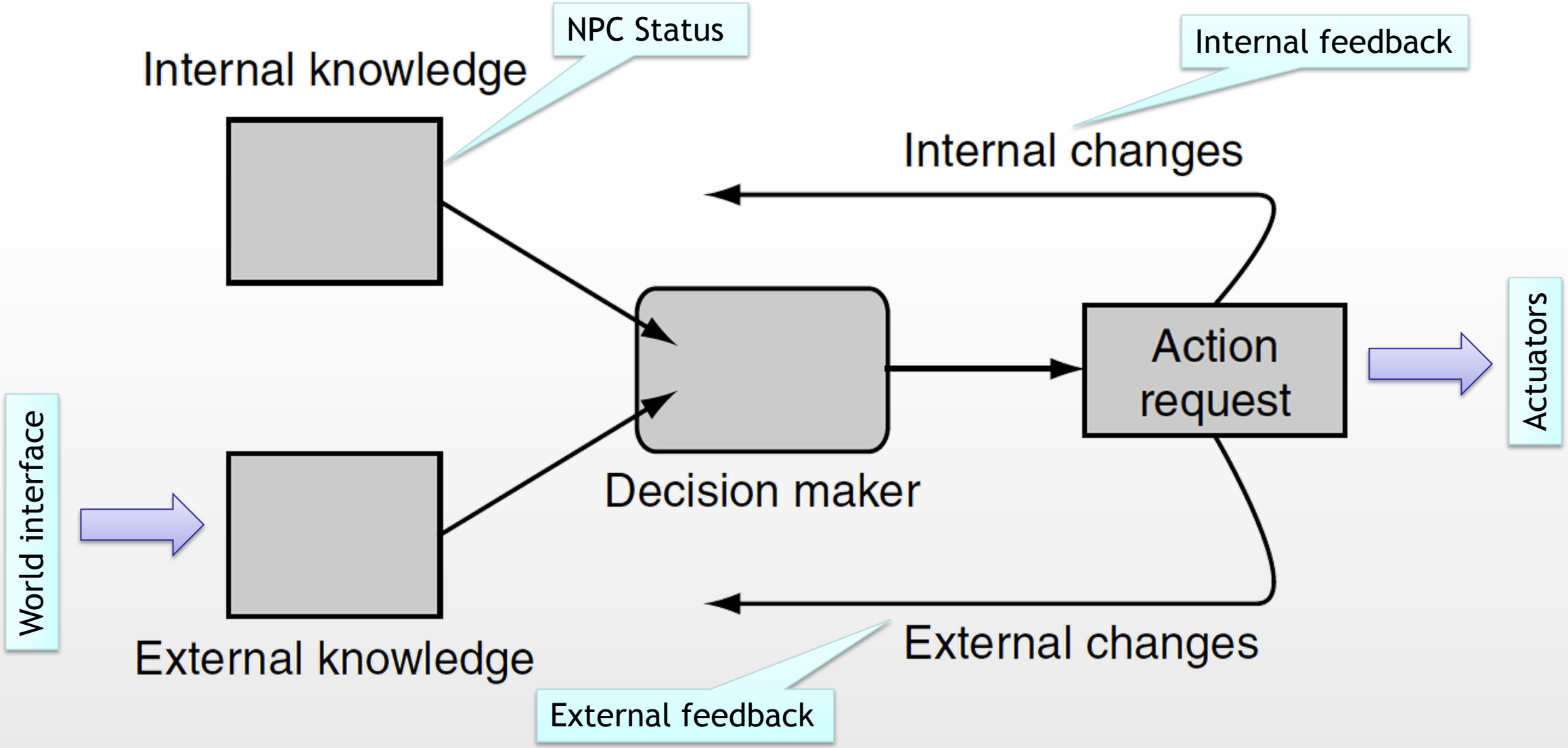
A Better Description



Decision Making in a Nutshell

- It is about having a character to decide “what to do” and NOT “how to do it”
- Might be inter- and intra-character
- A possible action might be “planning”
- The ability to carry out the decision is given for granted

How does it Work?



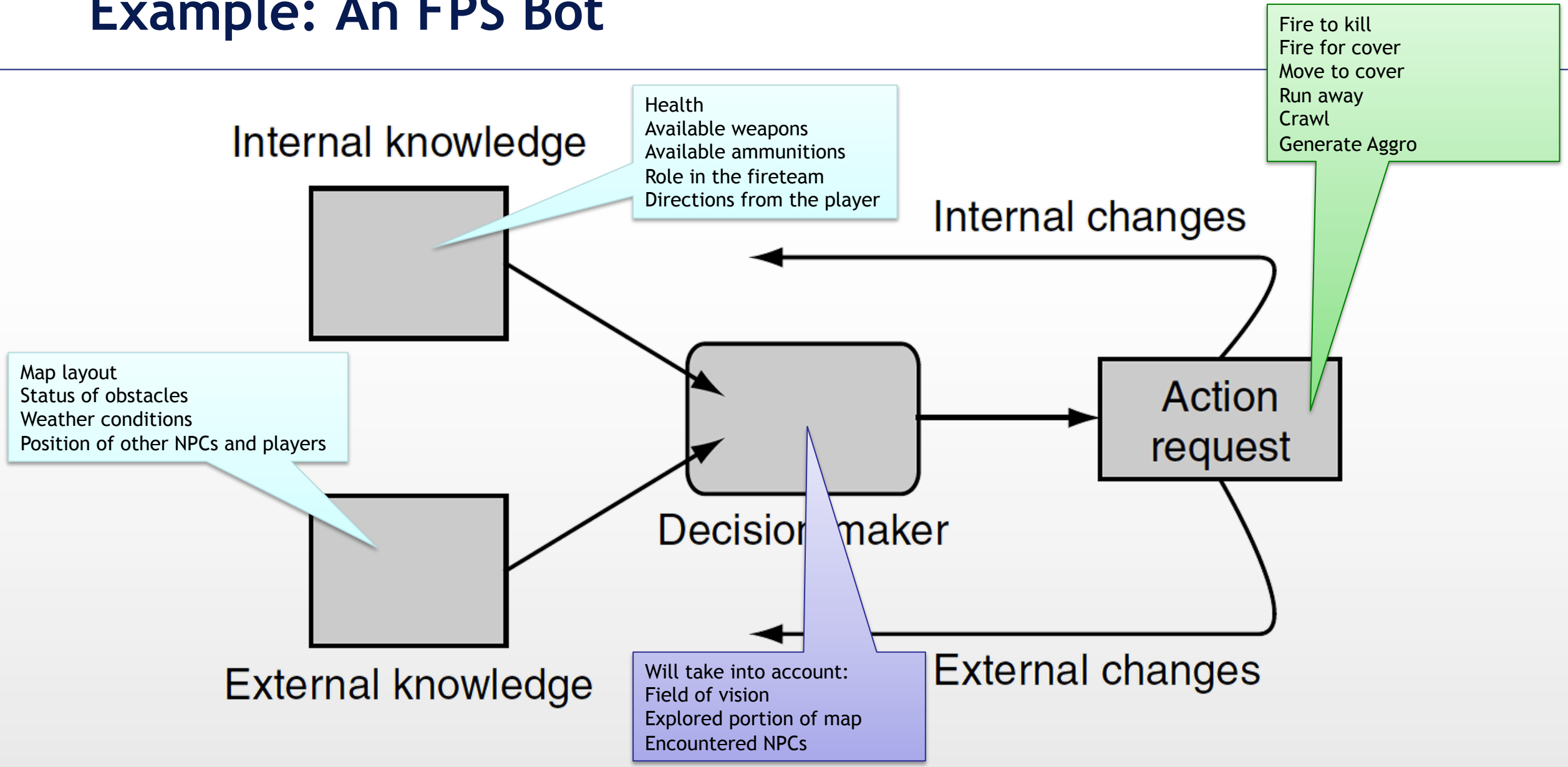
Internal and External Knowledge

- Internal
  - Available resources
  - Own goals
  - Personal behavior
  - History
- External
  - Position of other NPCs
  - Level layout
  - Status of objects
  - Whatever is coming from sensors

We know EVERYTHING

but it is up to the algorithm to decide WHAT to use

Example: An FPS Bot



UNIVERSITÀ DEGLI STUDI
DI MILANO

Decision Trees

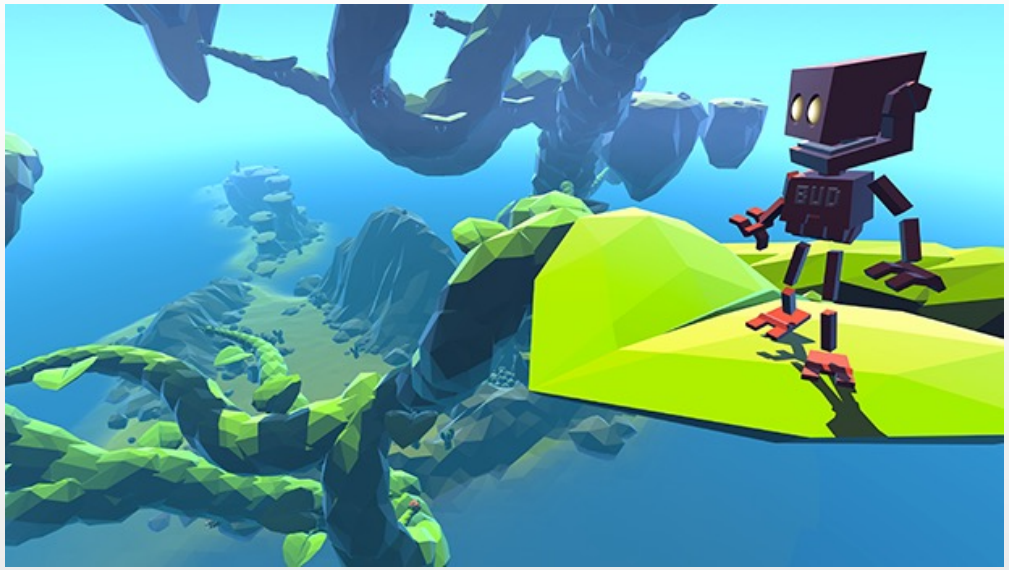
A.I. for Video Games

Davie Gadia (gadia@di.unimi.it) and Dario Maggiorini (dario@di.unimi.it)

Playlab fOr inNovation in Games

Decision Trees

- They are the simplest (and most used) approach to take a decision
  - Easy data structure
  - Fast to walk
  - Not only for characters
  - Very popular for animation control



Goal of Decision Trees

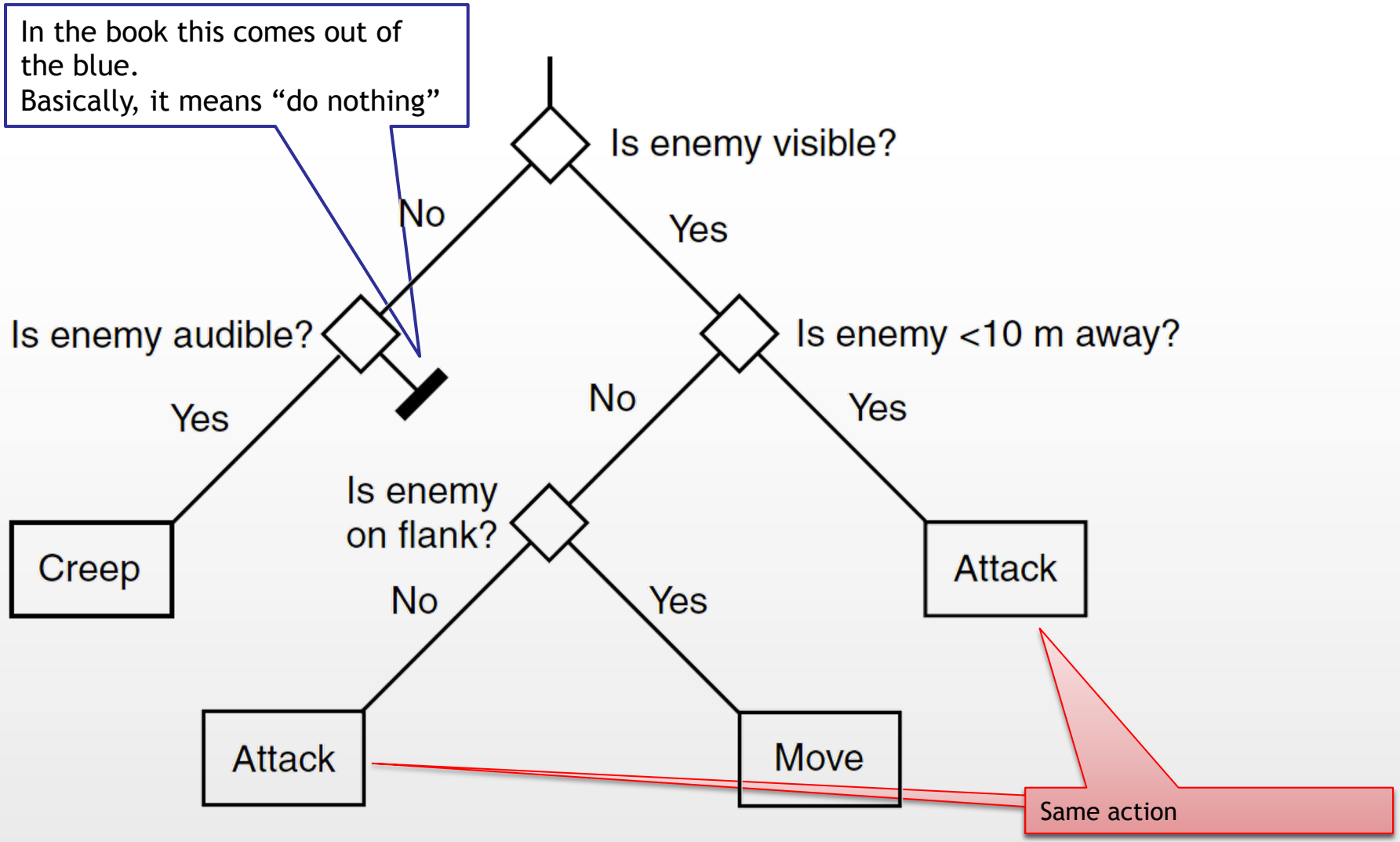
- Given a set of knowledge, we must select an action from a set of available actions
  - Mapping may be complex
  - The same action may be used for several (disjoint) set of inputs
  - Small changes in the input may be important
  - Some inputs may be more important than others

Decision Trees

- Structure
  - We have a set of decision points
    - Usually, we are asked for a quite simple decision at each node
    - Best practices suggest with two or three possible outcomes maximum
  - One decision point is the “beginning” of each decision (root node)
  - All decision points are linked in an acyclic graph
    - This will guarantee termination and bound execution time
  - Leaves declare actions to perform

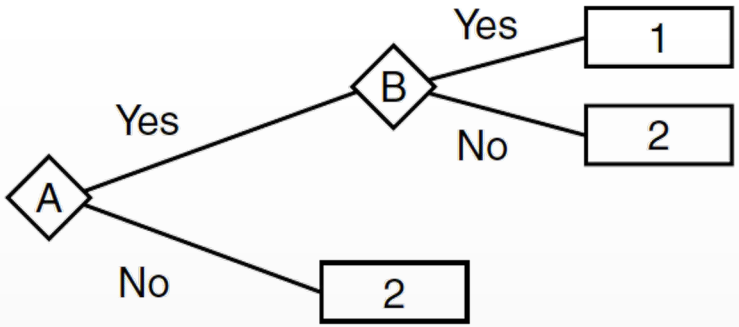
- Usage
  - Starting from the root node, we must select one of a set of outgoing options (links)
  - Each decision is taken based on the character’s knowledge
  - The algorithm continues until a leaf is reached
  - As soon as a leaf is reached, its action is performed

A Simple Decision Tree



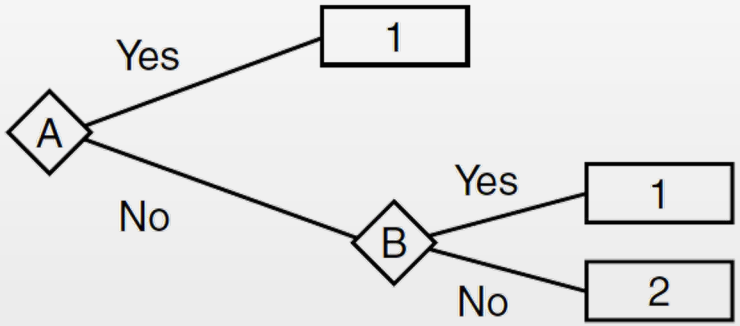
Combining Decisions

If A AND B then action 1, otherwise action 2

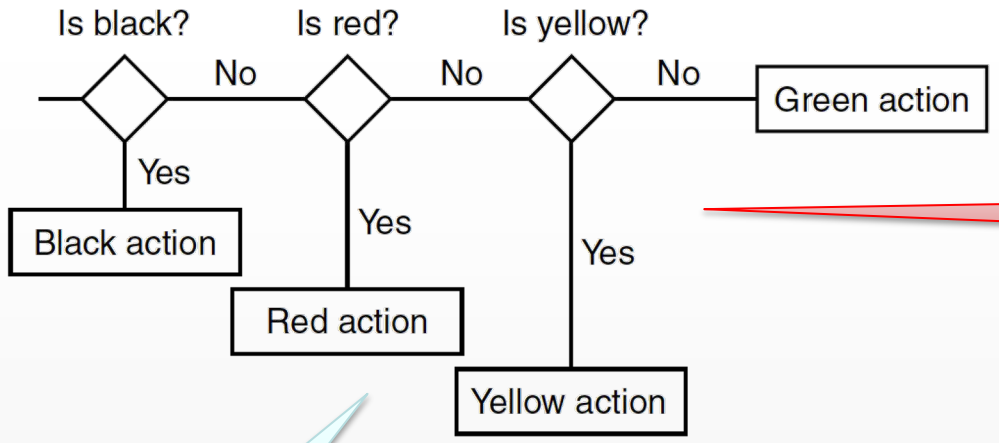


Decisions can be combined as for Boolean algebra

If A OR B then action 1, otherwise action 2

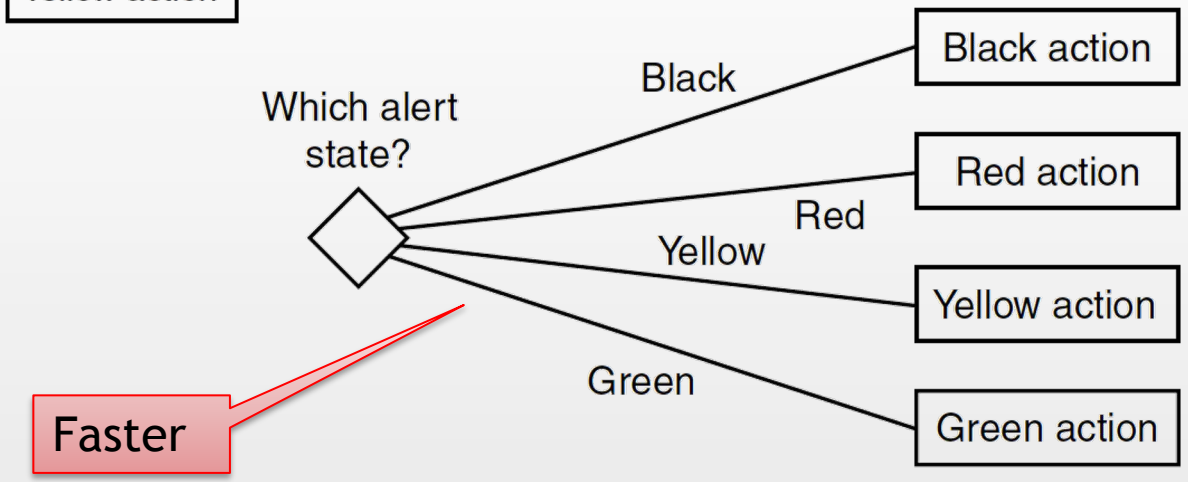


Working on Complexity



Easier to implement

More common



Faster

About DT Performances

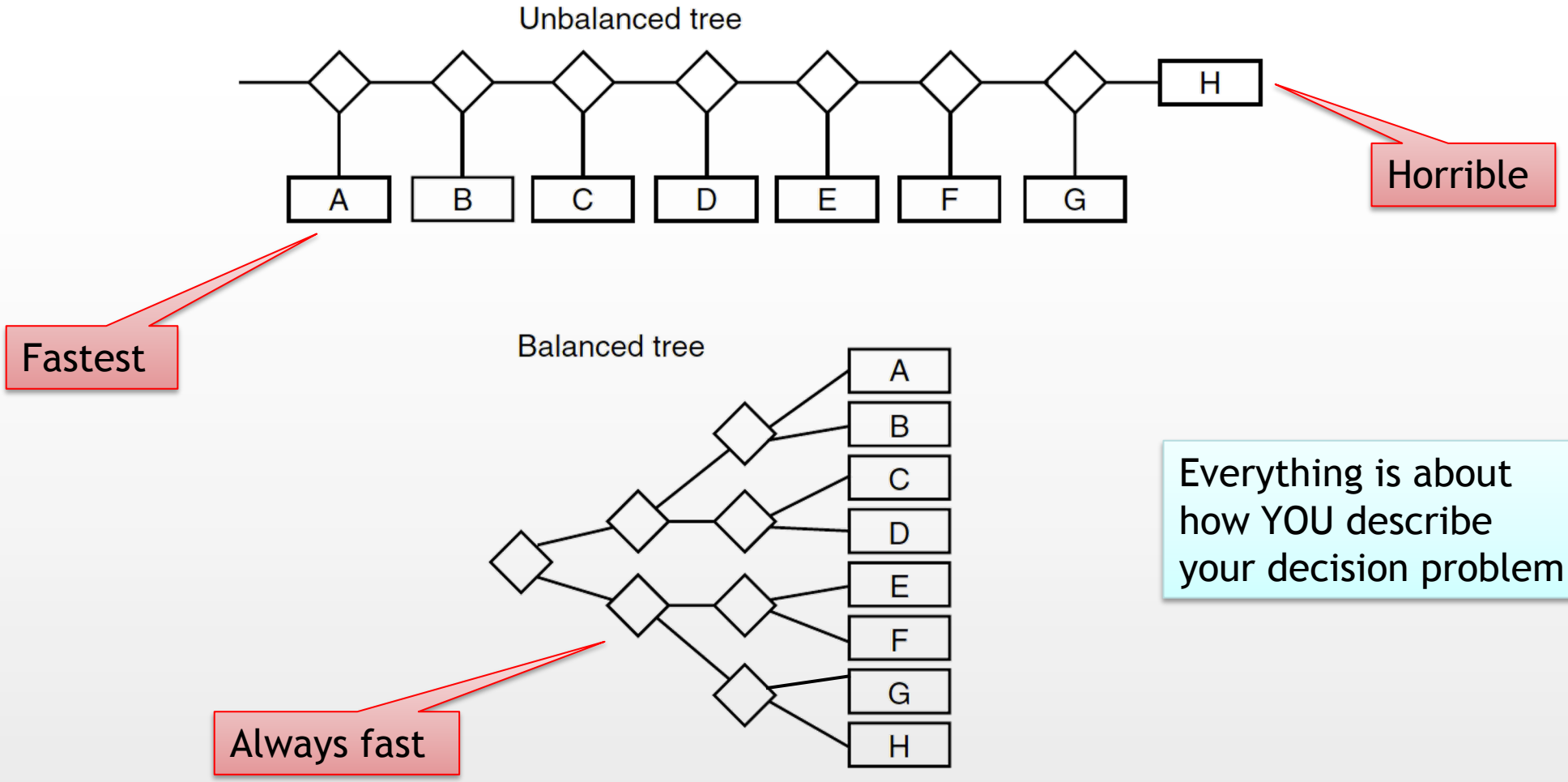
- Features of the execution
  1. Takes no memory
     - Each node in independent from the previous one
  2. We can assume a decision is taking a constant amount of time
     - If we have all simple and/or binary decision this is a safe assumption

- Time is linear with number of visited nodes
  - Thanks to point 2 above

- Complexity is the same as a tree visit algorithm
  - If the linear time condition above holds
  - Optimal case: a balanced z-branched tree - $O(\log_z n)$ where n is the number of nodes in the tree

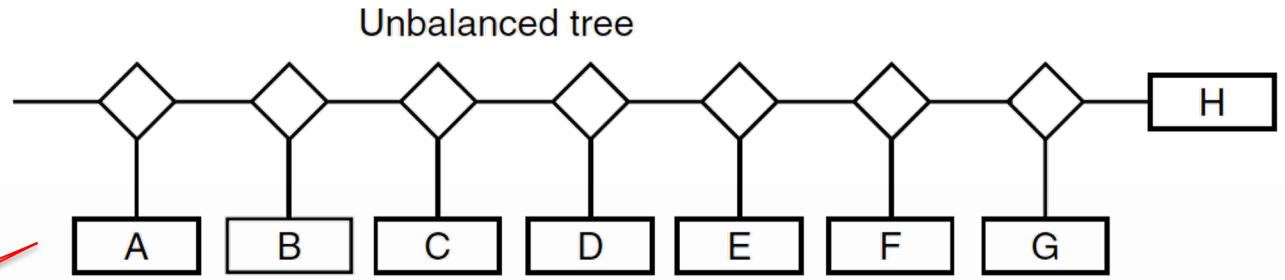
- Mind the bottlenecks!
  - Some decisions may take sensibly more if you need data from outside the game e.g., computing a complex problem or using a hardware sensor such as GPS on a phone

Balancing Trees



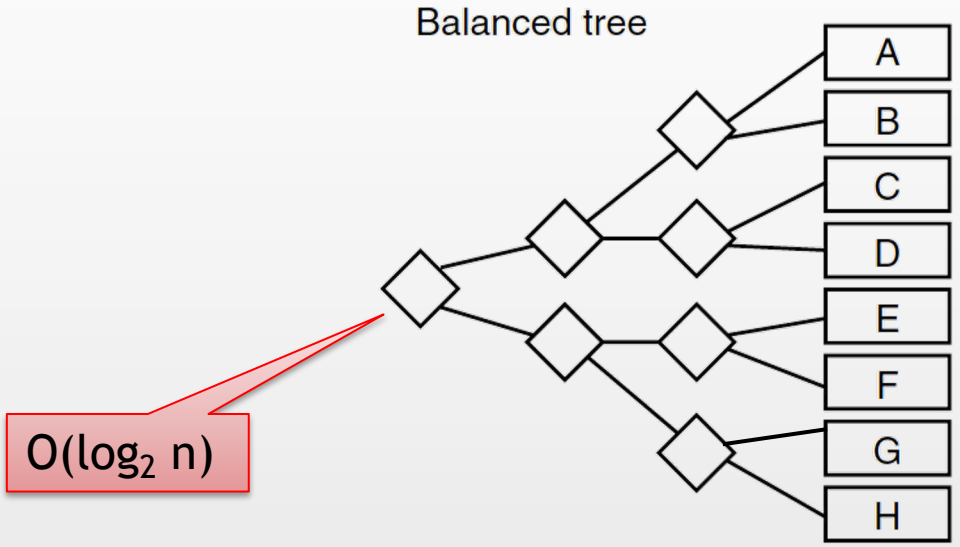
Everything is about how YOU describe your decision problem

Balancing Trees



O(n)

Average: 4.5 decisions



$O(\log_2 n)$

Average: 3 decisions

Not All Decisions are the Same

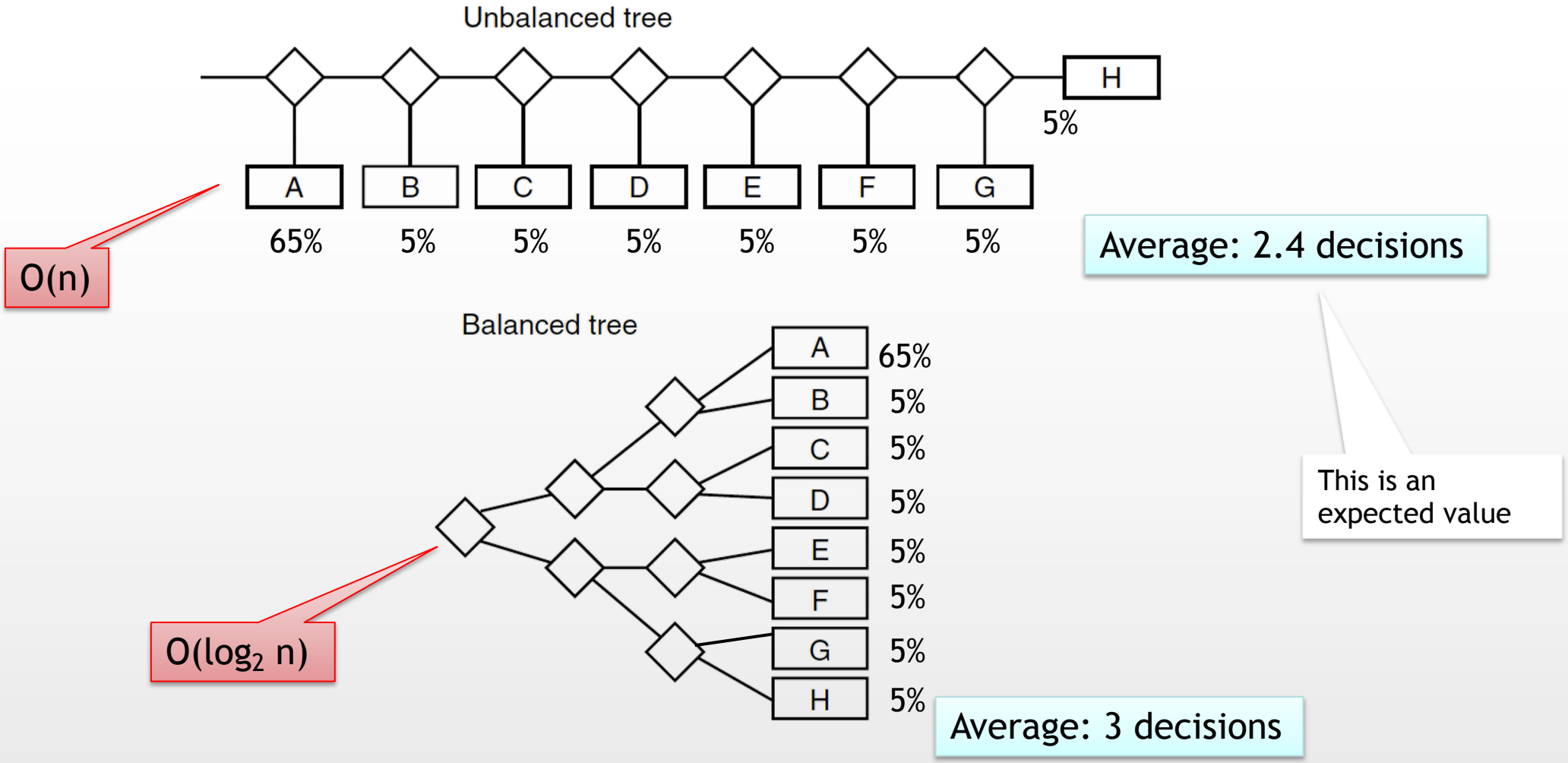
- Usual assumption when balancing a tree:

all decisions have the same probability to be taken

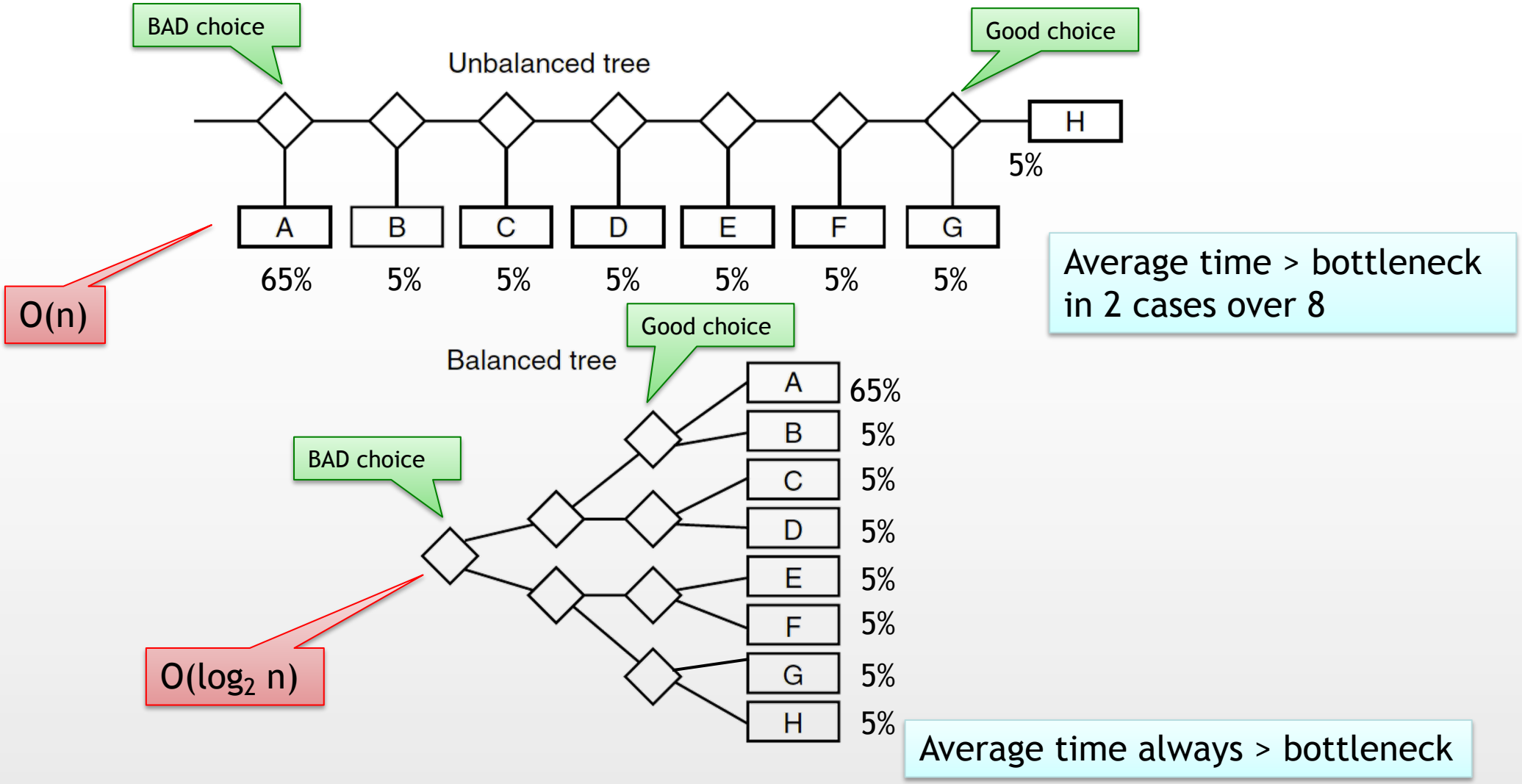
WRONG!

- An action (leaf) is taken (reached) very often?
  - Put it at the beginning of the queue
- A decision (node) is creating a bottleneck (slow to compute)?
  - Put it at the end of the queue

Very Often


Unbalanced tree
A
B
C
D
E
F
G
H
65%
5%
5%
5%
5%
5%
5%
5%
O(n)
Average: 2.4 decisions
This is an expected value
Balanced tree
A
B
C
D
E
F
G
H
65%
5%
5%
5%
5%
5%
5%
5%
O(log2 n)
Average: 3 decisions

Bottleneck (Where to Place It)

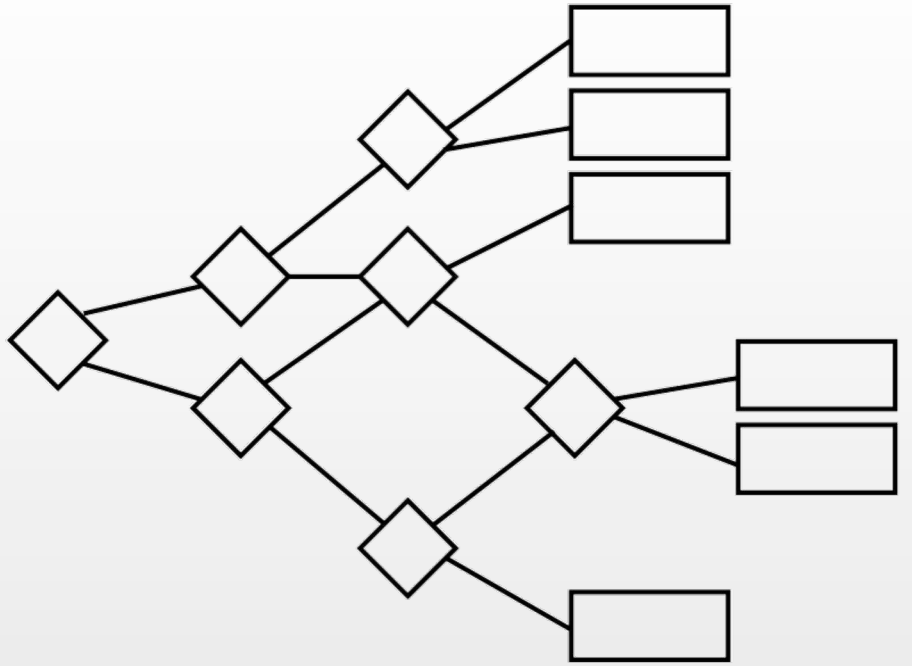


Not Strictly Trees

- To save space and avoid defining multiple time the same action we might define decision structures that are not directed graphs

- They are working, but are also
  1. More difficult to draw (crossing links)
  2. More difficult to debug

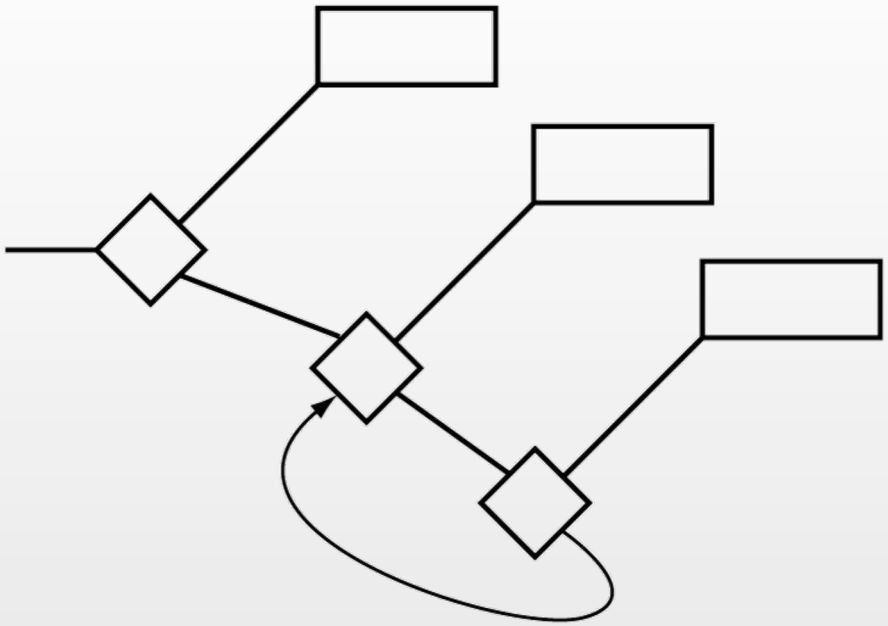
- Personal Suggestion:
  Always merge branches on leaves, never do that on nodes



Mind the Loops

- Inserting a loop **is always a huge mistake**
  - Termination is not guaranteed any more
  - Processing time will neither be linear nor bounded

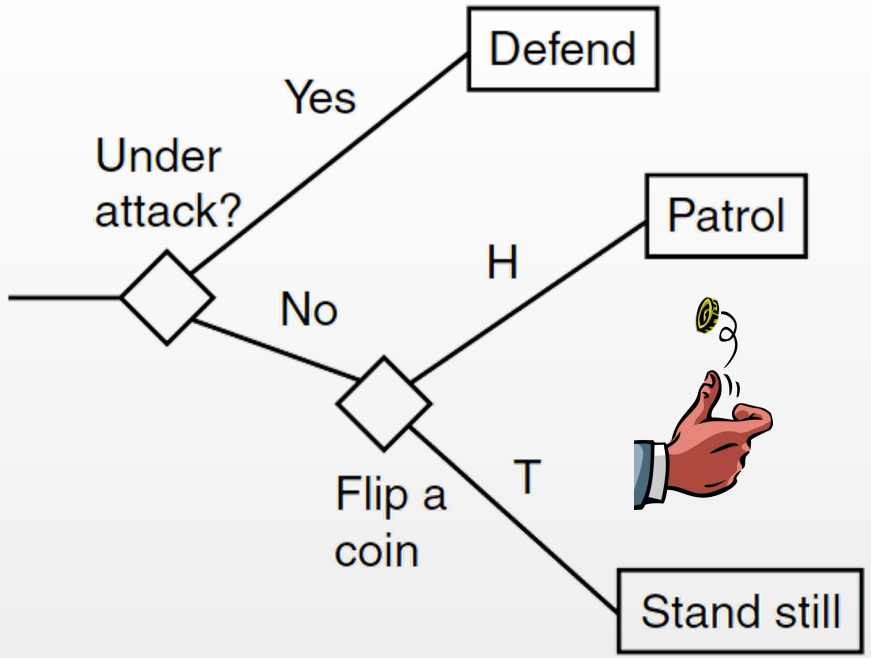
- In large (branch-merging) trees the loop may be long and very difficult to spot!



Adding Randomness

- May be helpful to spice up your game
  - Add unpredictability in NPCs behaviour
  - Variate gameplay
- ... and is easy to implement

- Computational warning!
  - Random is CPU intensive
    - Do not roll at every frame
    - Do not roll when not needed



Implementing Trees

- Data Representation
  - Trees use the same data types as the rest of the game
    - This makes your A.I. quick and easy to code
  - But change in the underlying code (or logic) might break your AI
    - These bugs are very difficult to squash

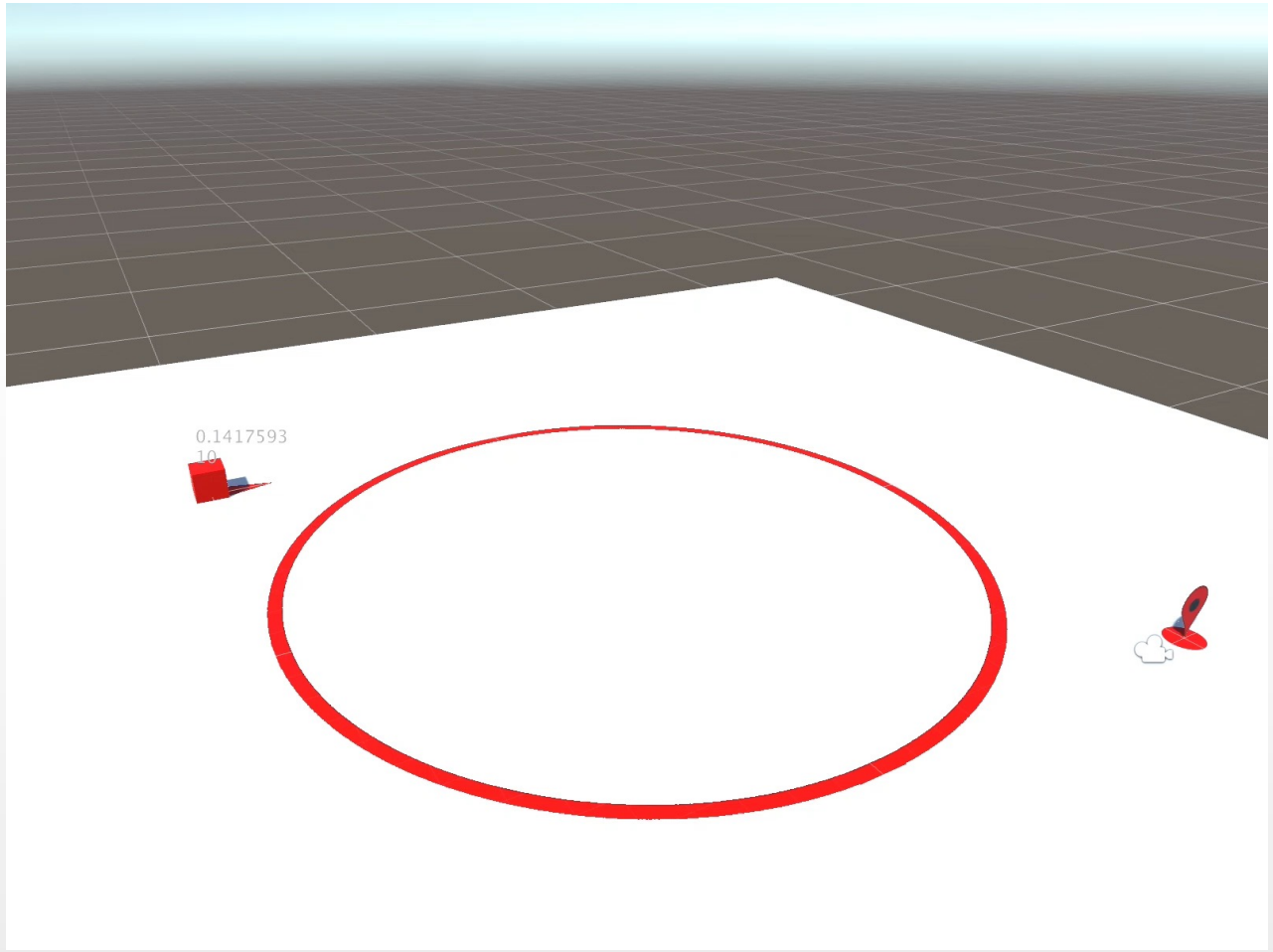
- Code integration
  - TIP: make no distinction between decisions and actions
  - Mileage varies with languages
    - C++ - use virtual functions
    - Java/C# - use runtime types

The Sentinel (Example in Unity)

- A sentinel is hidden and keeps looking around
- If an intruder is within range it pops out
- If the intruder is still there when the sentinel is out an alarm is activated on the next (AI) frame
- When the intruder leaves, the alarm is turned off and the sentinel goes back hiding

Spoiler

Scene: Sentinel With DT
Folder: Decision Trees



Comment on the Delay

- The detection delay is bounded by the AI frame resolution
  - The same problem we had when chasing the player on a navigation mesh
- To make the sentinel more responsive, just lower the duration of an AI frame, but the final result will be different
  - Mind, this will also increase CPU usage
- Also here, game mechanics and user experience can be affected by a technical decision
  - Pay a lot of attention on this!

Bad and Good News

- Bad news
  - Decision trees are not native in unity
  - There are some (visual) plugins in the asset store

- Good news
  - Implementing a decision tree by yourself is very simple

A Library for Decision Tree Management

Source: DecisionTree
Folder: Decision Trees

First, we define an interface for all nodes in the tree. The only operation required in the interface is to process (walk) the node

```
// Interface for both decisions and actions
// Any node belonging to the decision tree must be walkable
public interface IDTNode {
    DTAction Walk();
}

// This delegate will defer functions to both
// making decisions and performing actions
public delegate object DTCall(object bundle);
```

Always remember, in C#, *object* is a superclass for basic type also.
This delegate can return ANYTHING

With this delegate we will hook external code to both run decisions on single nodes and to perform actions

A Library for Decision Tree Management

A decision node is a node. So, we use the node interface

```
// Decision node
public class DTDecision : IDTNode {

    // The method to call to take the decision
    private DTCall Selector;

    // The return value of the decision is checked against
    // a dictionary and the corresponding link is followed
    private Dictionary<object, IDTNode> links;

    public DTDecision(DTCall selector) {
        Selector = selector;
        links = new Dictionary<object, IDTNode>();
    }

    // Add an entry in the dictionary linking a possible output
    // of Selector to a node
    public void AddLink(object value, IDTNode next) {
        links.Add(value, next);
    }

    // We call the selector and check if there is a matching link
    // for the return value. In such case, we Walk() on the link
    // No link means no action and null is returned
    public DTAction Walk() {
        object o = Selector(null);
        return links.ContainsKey(o) ? links[o].Walk() : null;
    }
}
```

Selector is the hook to the external code to perform the selection

This dictionary links following nodes through a key object. Once again, the key can be ANYTHING

The node is created with a selector and an empty dictionary

Add a link to an external state through a key

To walk the node, we call the selector. If there is a key equal to the returned value, then we walk the linked node. Otherwise we return null

The return value for walking the root node will be the action to perform ... or null

A Library for Decision Tree Management

An action (leaf) is also a node. So, we use the same interface

Action is the hook to the external code to perform the action

```
// Action node
public class DTAction : IDTNode {

    // The methos to perform the action
    public DTCall Action;

    public DTAction(DTCall callee) {
        Action = callee;
    }

    // We are an action, we are the one to be called
    public DTAction Walk() { return this; }
}
```

The action is created with just a reference to the code to run

NO! We cannot call Action and return its result. Otherwise, if the action itself is returning null, the root node will never know if the decision process produced a result or not

If we walked this node, this is the final stage of the decision process, and this is the action to perform. Then, we return ourselves

A Library for Decision Tree Management

A decision tree is a container for the root node; where all the other node are linked

```
// This class is holding our decision structure
public class DecisionTree {

    private IDTNode root;

    // Create a decision tree with starting from a root node
    public DecisionTree(IDTNode start) {
        root = start;
    }

    // Walk the structure and call the resulting action (if any)
    // anull means no action is required.
    public object walk() {
        DTAction result = root.Walk();
        if (result != null) return result.Action(null);
        return null;
    }
}
```

Walking the decision three is the same as walking the root node

The return value from walking the root node is a reference to an action that (if not null) will be executed

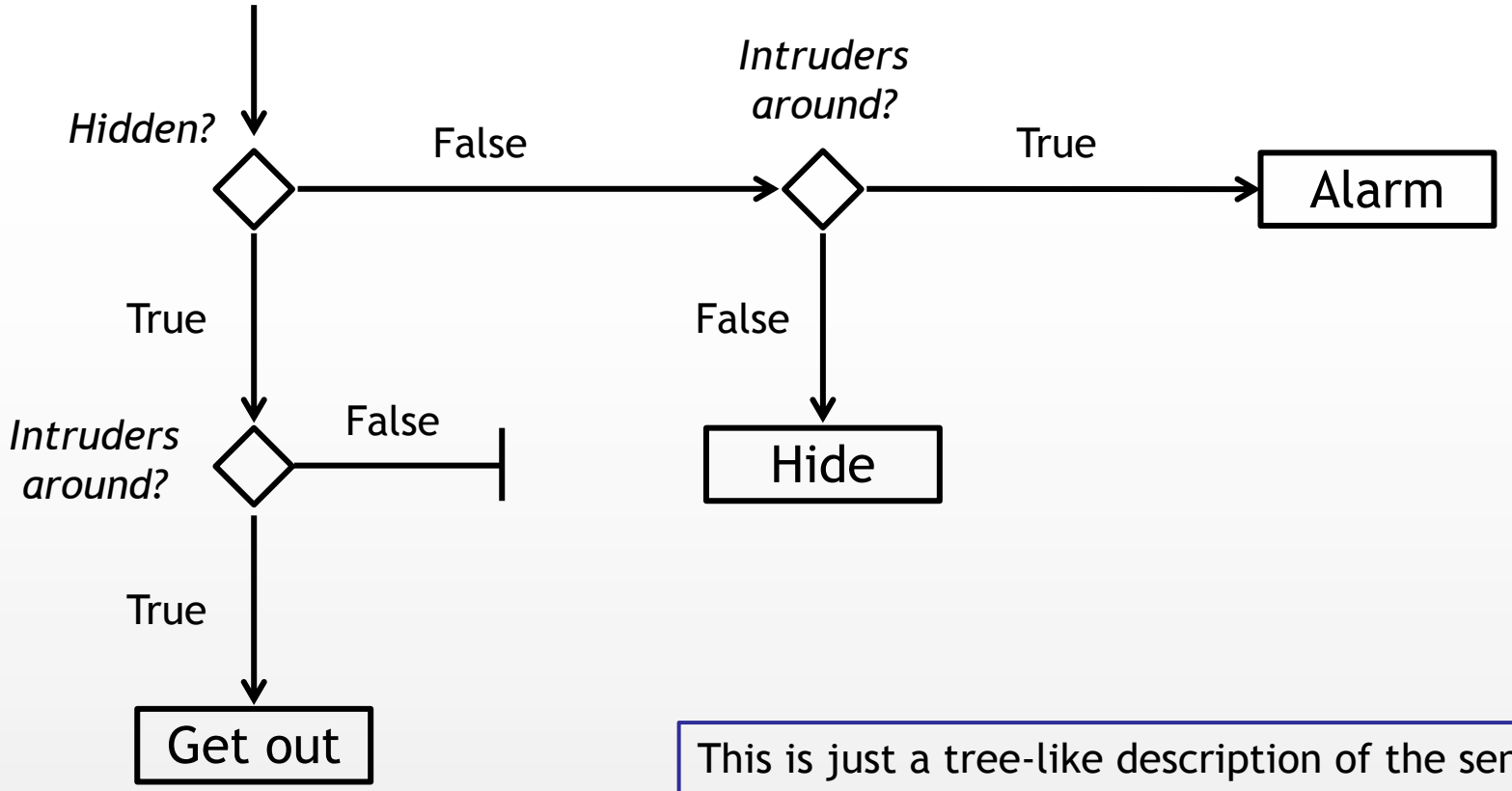
Let’s Make it Clear

- This implementation sucks (big time!)

  1. No error checking
  2. No sanity in parameters
  3. No optimization (uses recursion)

  … “*it just works*”

Planning the Tree



This is just a tree-like description of the sentinel behavior we expressed in text at the beginning

Actions

Source: DTSentinel
Folder: Decision Trees

Hiding means:
1. setting the light back to the original color
2. disabling the mesh rendered component (not be drawn on the scene)
3. setting the internal knowledge to “I am hiding”

Showing up means enabling the mesh rendered and setting the internal knowledge to “I am not hiding”

Pro tip: the hidden variable can be substituted by checking the status of the mesh rendered. Nevertheless, this will make the code less reusable and will take more CPU (and time) to run

```
// ACTIONS

public object Hide(object o) {
    alarmLight.color = baseColor;
    GetComponent<MeshRenderer> ().enabled = false;
    hidden = true;
    return null;
}

public object Show(object o) {
    GetComponent<MeshRenderer> ().enabled = true;
    hidden = false;
    return null;
}

public object Alarm(object o) {
    alarmLight.color = alarmColor;
    return null;
}
```

Turning on the alarm means setting the ambient color to the parameter set in the component interface

Decisions

To query the agent status (check if we are hiding) we use the internal knowledge

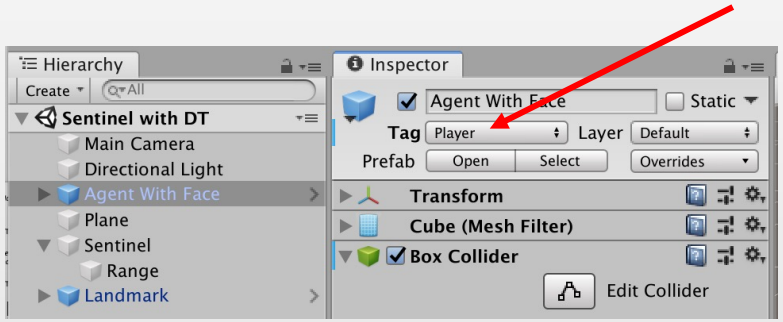
```
// DECISIONS

// Check if we are hidden or not
public object GetStatus(object o) {
    return hidden;
}

// Check if there are enemies in range
public object ScanField(object o) {
    foreach (GameObject go in GameObject.FindGameObjectsWithTag(targetTag)) {
        if ((go.transform.position - transform.position).magnitude <= range) return true;
    }
    return false;
}
```

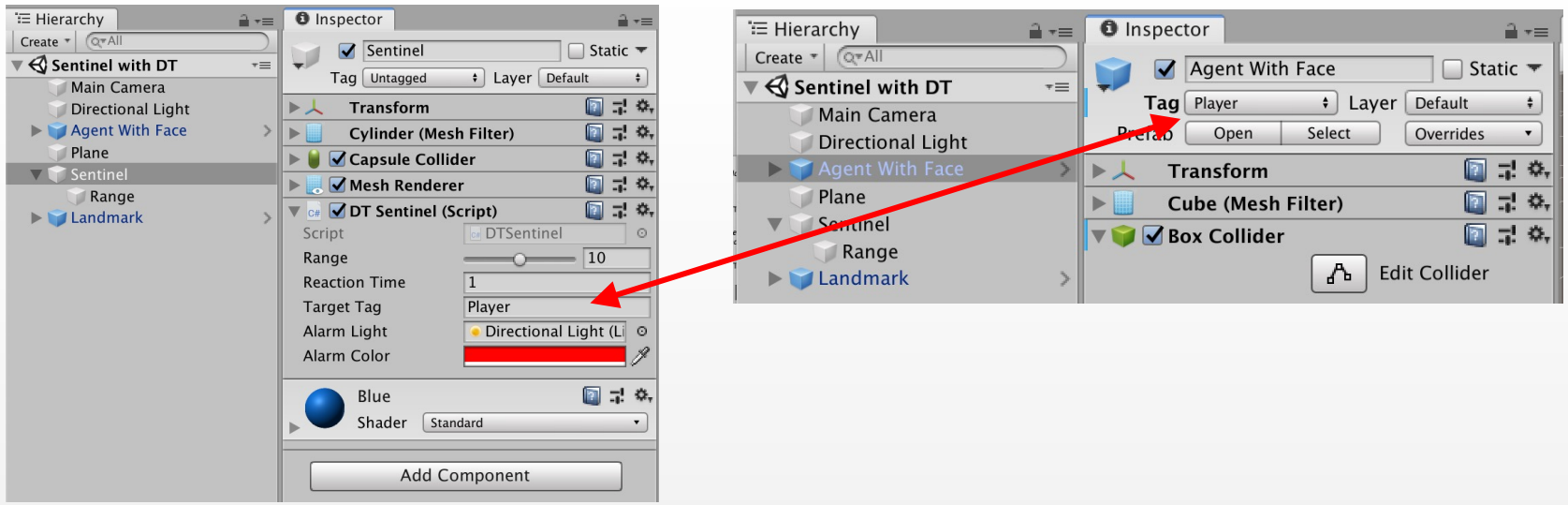
Actually, not all objects but only those with a specific tag. This will boost performances and avoid triggering for objects that will be always there. E.g., the ground

To scan the field we iterate on all the objects in the scene and check if there is at least one closer than the sensing range



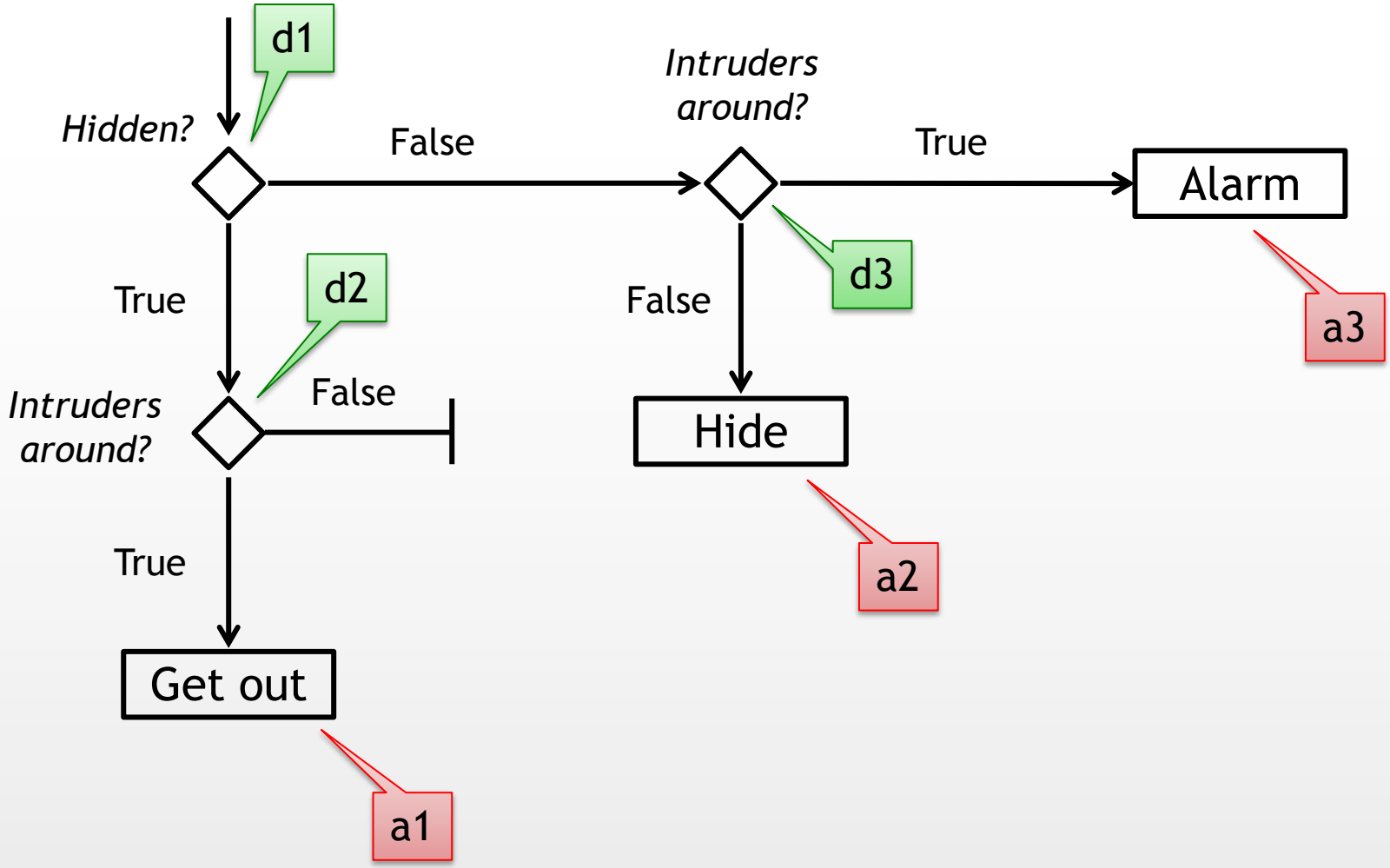
Beware of the Tags!

- Because on one side they are a drop down, while on the other they are a string



- It is very easy to break your build if you are careless

Code References



Implementation in a MonoBeaviour

First: assign actions

Second: assign decisions

Third: link decisions and actions

Fourth: create a decision tree on the root node (d1)

Fifth: start a coroutine to run the decision tree

```
void Start () {

    // Define actions
    DTAction a1 = new DTAction(Show);
    DTAction a2 = new DTAction(Hide);
    DTAction a3 = new DTAction(Alarm);

    // Define decisions
    DTDecision d1 = new DTDecision(GetStatus);
    DTDecision d2 = new DTDecision(ScanField);
    DTDecision d3 = new DTDecision(ScanField);

    // Link action with decisions
    d1.AddLink(true, d2);
    d1.AddLink(false, d3);

    d2.AddLink(true, a1);

    d3.AddLink(true, a3);
    d3.AddLink(false, a2);

    // Setup my DecisionTree at the root node
    dt = new DecisionTree(d1);

    // Set to hidden status
    baseColor = alarmLight.color;
    Hide(null);
    // same as - a2.Action();

    // Start patroling
    StartCoroutine(Patrol());
}
```

We could avoid hiding the sentinel at startup by disabling the mesh renderer in the editor. This way, the sentinel status would hidden by default.
Nevertheless, editing the scene is a pain when you cannot see the objects you are working with

Set the system at start condition: backup the default environment light color and and hide the sentinel

… And Number 5 is Alive

```
// Take decision every interval, run forever
public IEnumerator Patrol() {
    while(true) {
        dt.walk();
        yield return new WaitForSeconds(reactionTime);
    }
}
```

Keep walking the decision tree every *reactionTime* seconds and performn the returned action if needed

... and we are done!

References

- On the textbook
  - § 5.1
  - § 5.2 (excluding 5.2.3 and 5.2.4)