CPSC 427
Video Game Programming

Game Play and AI

Helge Rhodin

Overview

Today:

- ***Making decisions (short term)***
- ***State Machines***
- ***Behaviour Trees***
- and their implementation

Next:

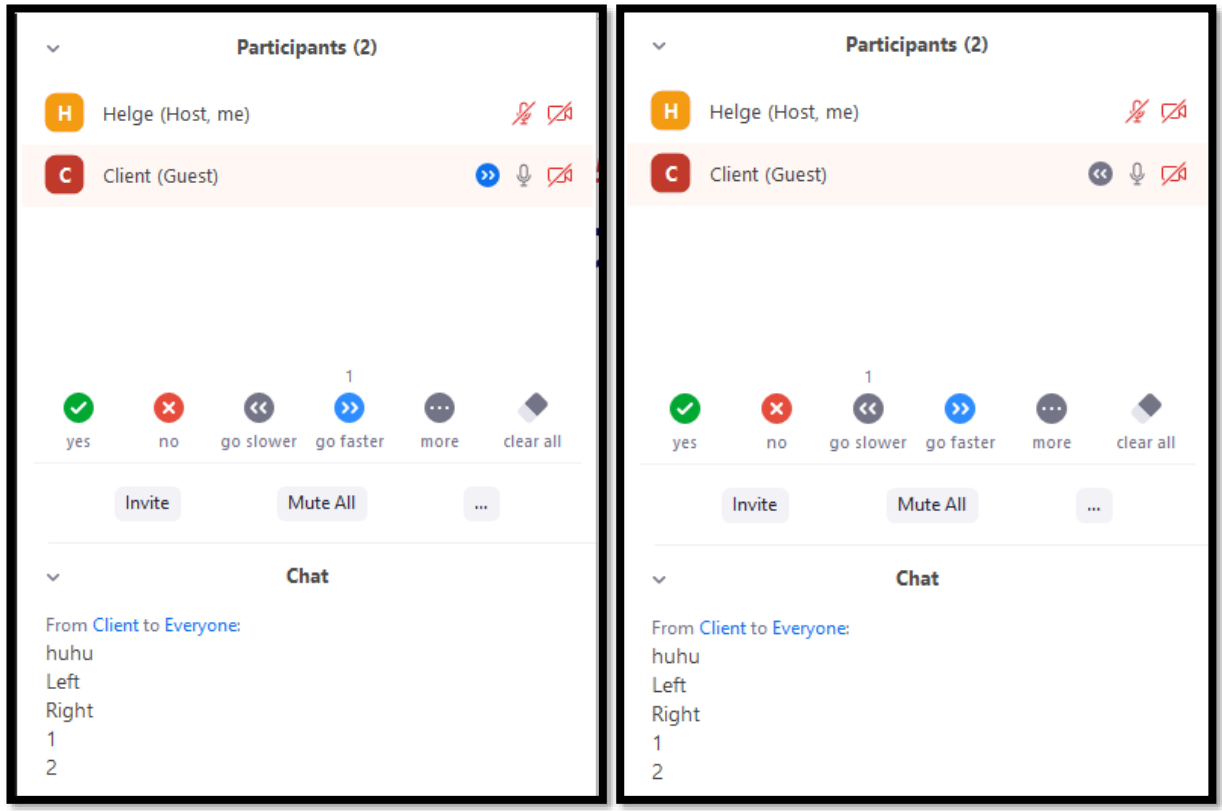
- ***Planning (long term)***

M1 submission

don’t forget your progress report!

Recap: Read the zoom chat

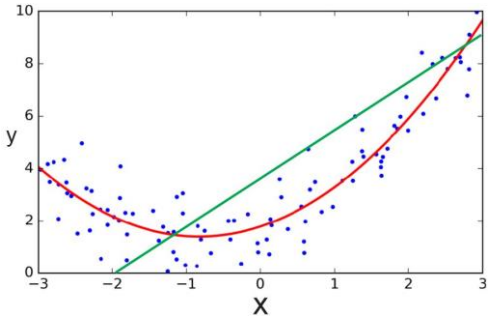
- ***Capture the screen***
  - *https://github.com/smasherprog/screen_capture_lite*
- ***Search for the zoom window***
- ***Check for colored symbol***
  - red, green, gray, blue?
    - *only need to read a few pixels*
      - its fast!
- ***Recognize numbers?***
  - only 10 different ones, brute force?

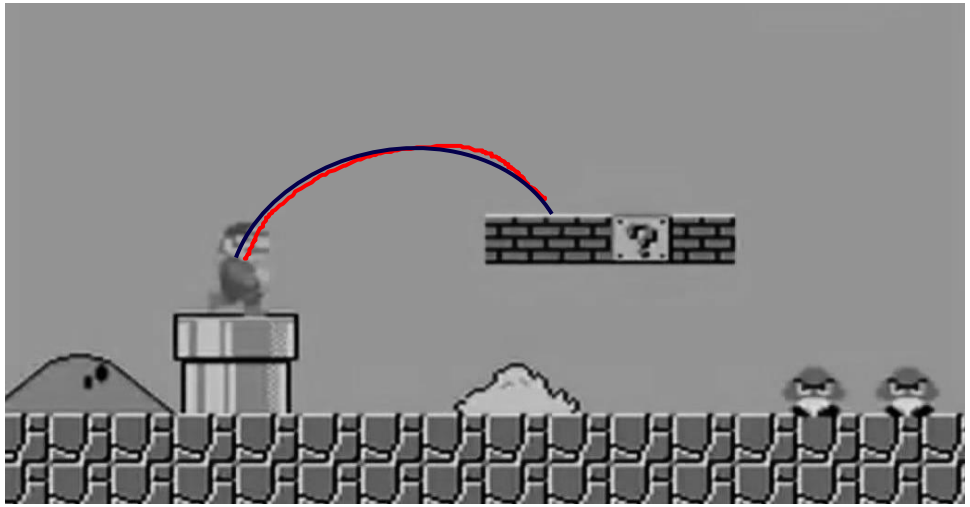


Feature: Mouse gestures

Regression

- ***least squares fit***
  - linear, polynomial, and other parametric functions



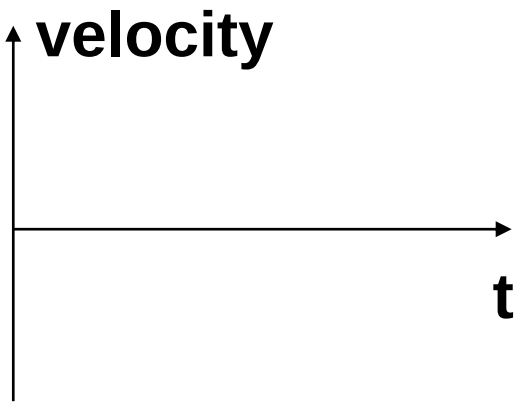


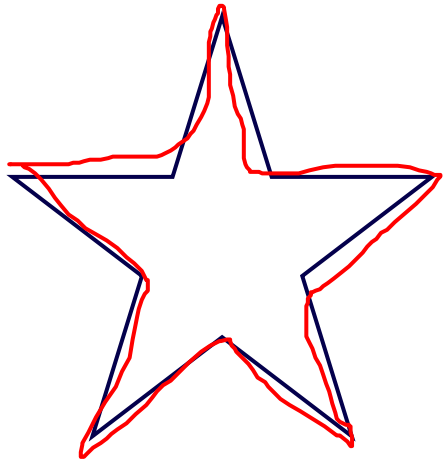
Search

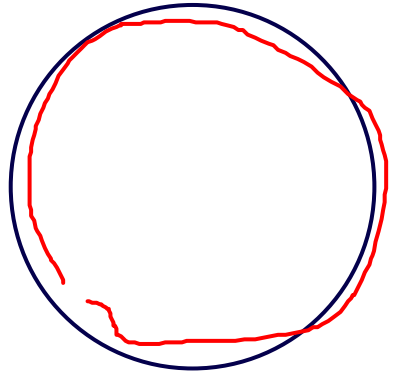
- brute force?
- binary search?

Detection

- key events
- pattern matching







Feature: Level Loading with JSON

Libraries:

- ***https://sourceforge.net/projects/libjson/***
- ***https://github.com/nlohmann/jso***
- ***others?***

Loading Entities and Components

- Outer list of entities
- Inner list of components
- Create a factory that instantiates each component type
- Equip components with toJSON(…) and fromJSON(…) functions

```
“entities": [
  {
    "position": {
      "x": -1.7193701,
      "y": -0.09165986
    },
    "velocity": {
      "x": 0,
      "y": 0
    },
    "color": {
      "x": 0.453125,
      "y": 0.453125,
      "z": 0.453125
    },
    "type": “Water Animal"
  },
  {
    "position": {
      "x": 2.2221813,
      "y": -1.2671415
    },
    "velocity": {
      "x": 0,
      "y": 1
    },
    "radius": 0.9300000000000006,
    "color": {
      "x": 0.40625,
      "y": 0.40625,
      "z": 0.40625
    },
    "type": ”Land Animal”
  }
]
```

Factory from JSON

Factory:

```
void ComponentfromJson(Entity e, JsonObject json)
{
  if(str1.compare(“Motion") != 0) {
     Motion& motion = Motion::fromJson(json);
     registry->insert(e, motion);
  }
  else if(str1.compare(“Salmon") != 0)
     Motion& component = Motion::fromJson(json);
     registry->insert(e, component);
  }
  …
}
```

Issues?

Component from JSON

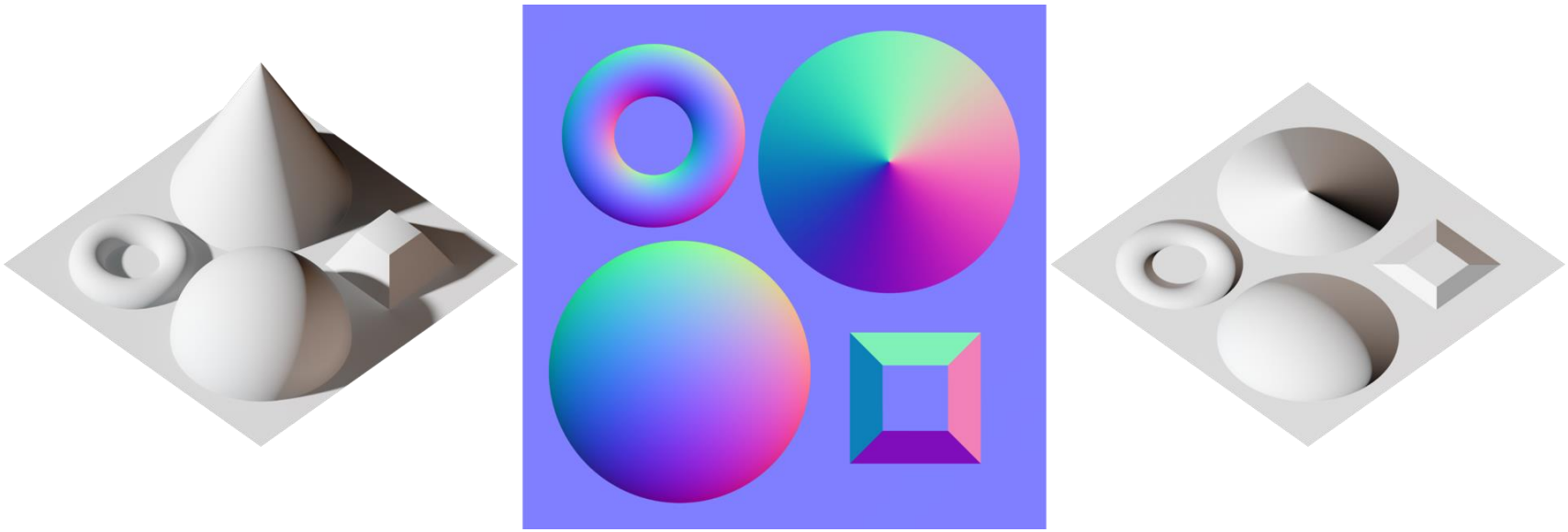
Component to/from:

```
class Vector2D
{
    float x,y;
    public:
    JsonObject toJson()
    {
       JsonObject json = Json.object();
       json.add("x", x);
       json.add("y", y);
       return json;
    }

    static Vector2D fromJson(JsonObject json)
    {
       double x = json.getFloat("x", 0.0f);
       double y = json.getFloat("y", 0.0f);
       return Vector2D(x,y);
    }
}
```

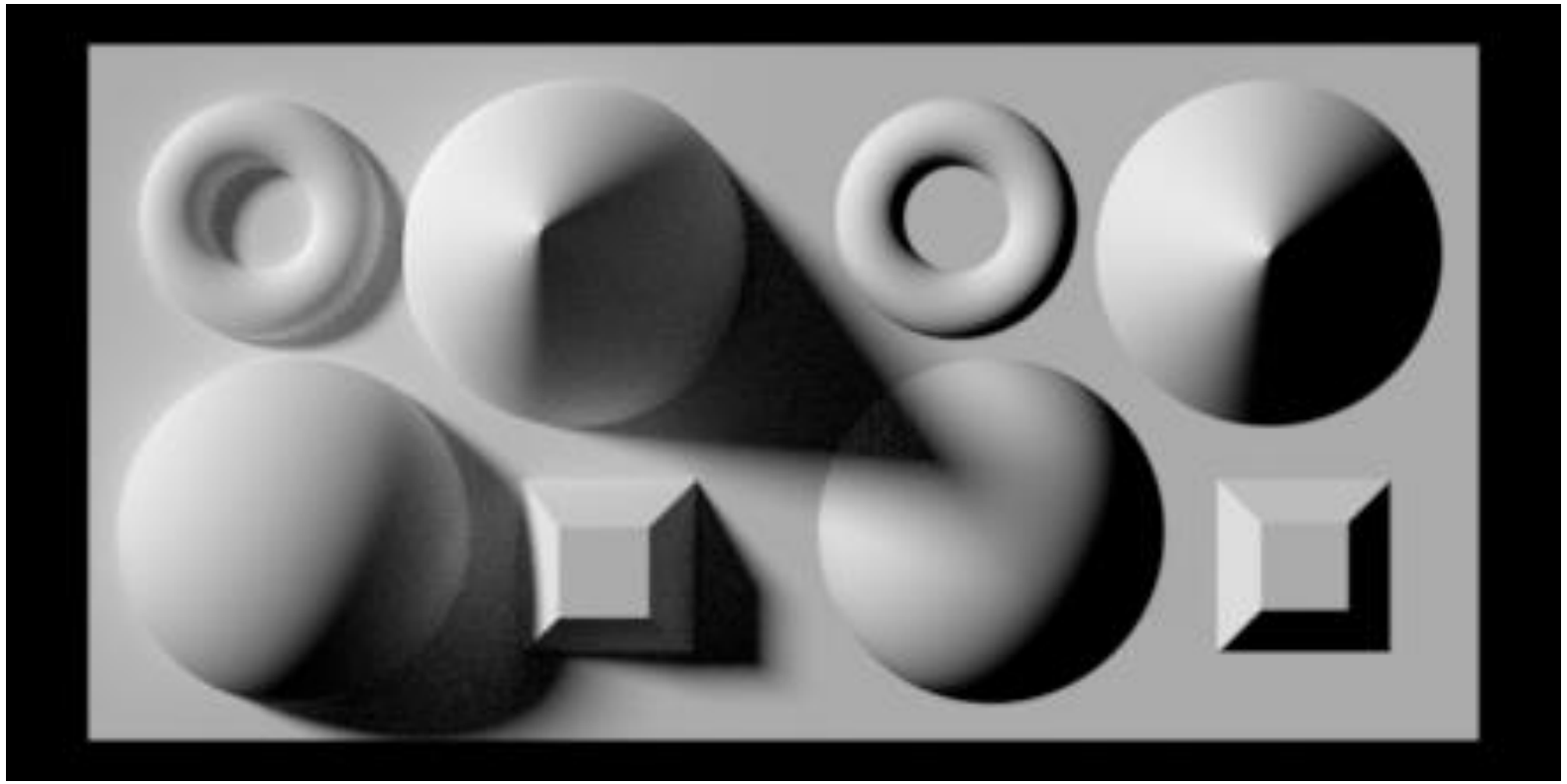
Feature: Normal maps

A way to fake 3D details



Perfect for illumination in 2D games

- ***What do you observe?***



How to implement?

Either:

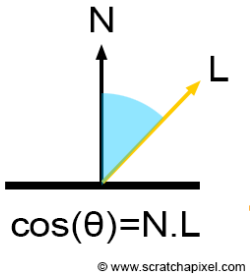
1. Include shading into every shader

- Load and sample from RGB texture
- Load and sample from normal map (the new aspect)
- Compute shading

2. Two-pass rendering

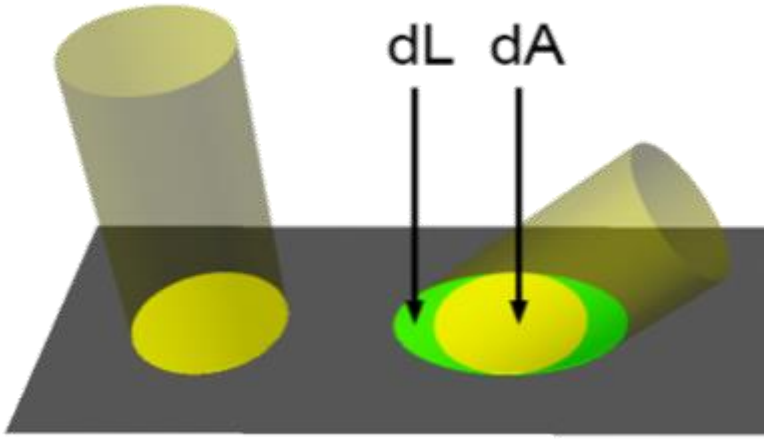
- Render color in one pass
- Render the normal in a second pass
- Compute shading in a separate pass, as for the water shader

Shading equation?



- ***Single light source:***
  - *Dot product of normal and light direction*
    - Light direction: computed from light source (L) and pixel location (x)
    - Normal direction: load from normal map

*color = texture(x) * dot(normal(x), normalized(x-L))*



- Multiple light sources? Specular highlights?

CPSC 427
Video Game Programming

State machines

Helge Rhodin

Gameplay

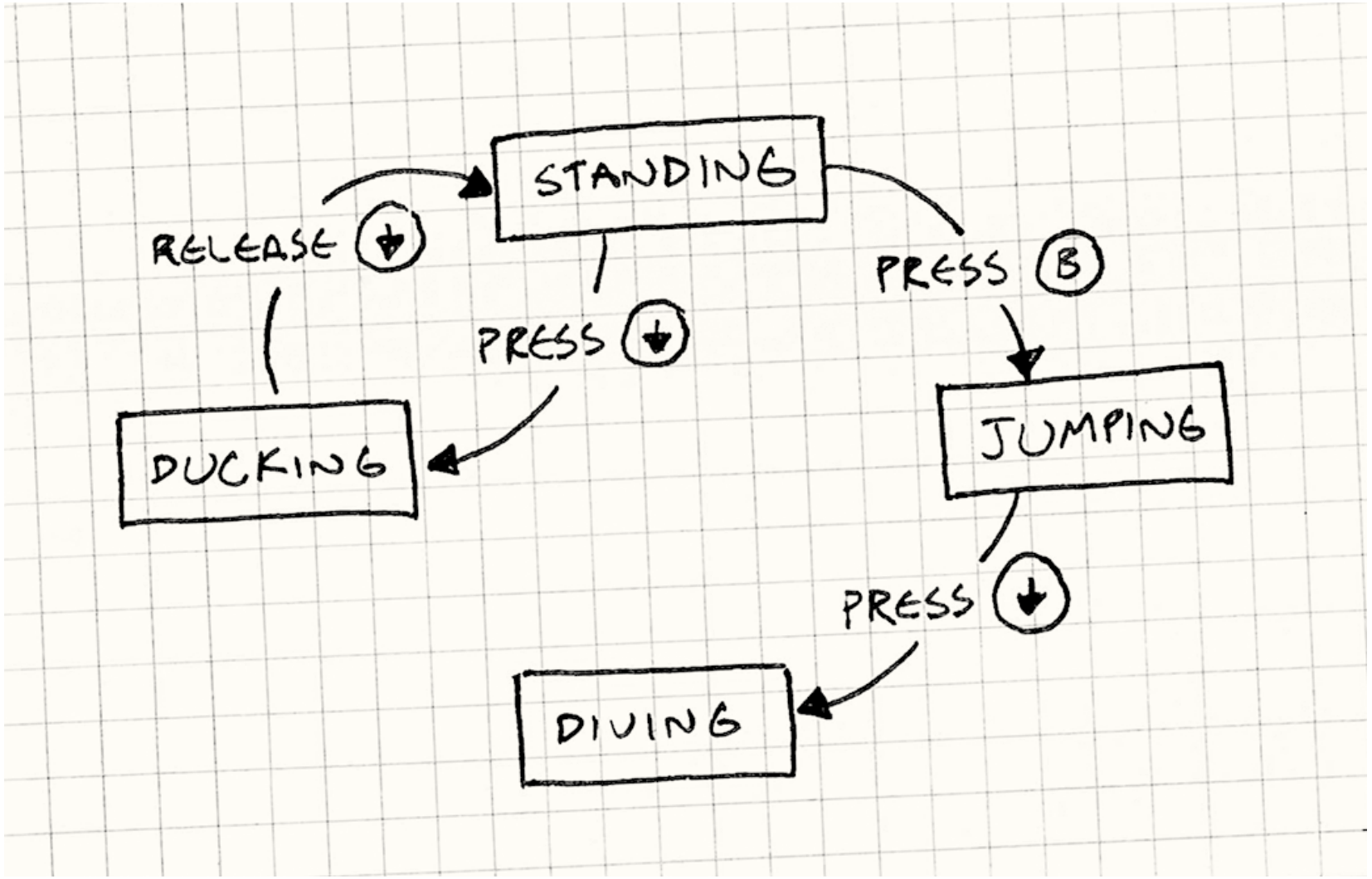
```
// start
if (!walking && wantToWalk)
{
    PlayAnim(StartAnim);
    walking = true;
}

// walk loop
if (IsPlaying(StartAnim) && IsAtEndOfAnim())
{
    PlayAnim(WalkLoopAnim);
}

// stop
if (walking && !wantToWalk)
{
    PlayAnim(StopAnim);
    walking = false;
}
```

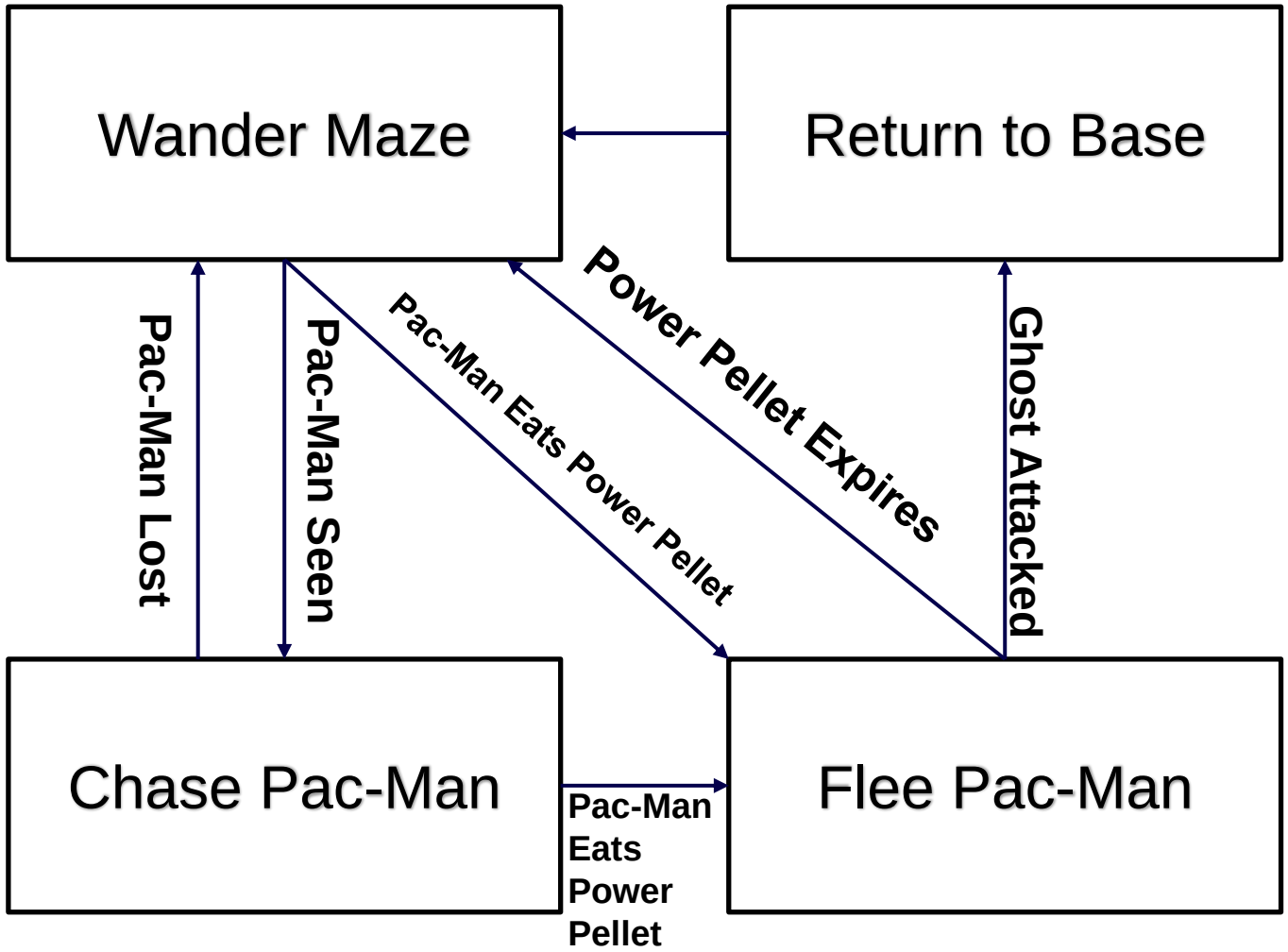
From http://twvideo01.ubm-us.net/o1/vault/gdc2016/Presentations/Clavet_Simon_MotionMatching.pdf

Finite State Machines: States + Transitions



FSM Example: Pac-Man Ghosts

FSM Example: Pac-Man Ghosts



Ghost AI in PAC-MAN

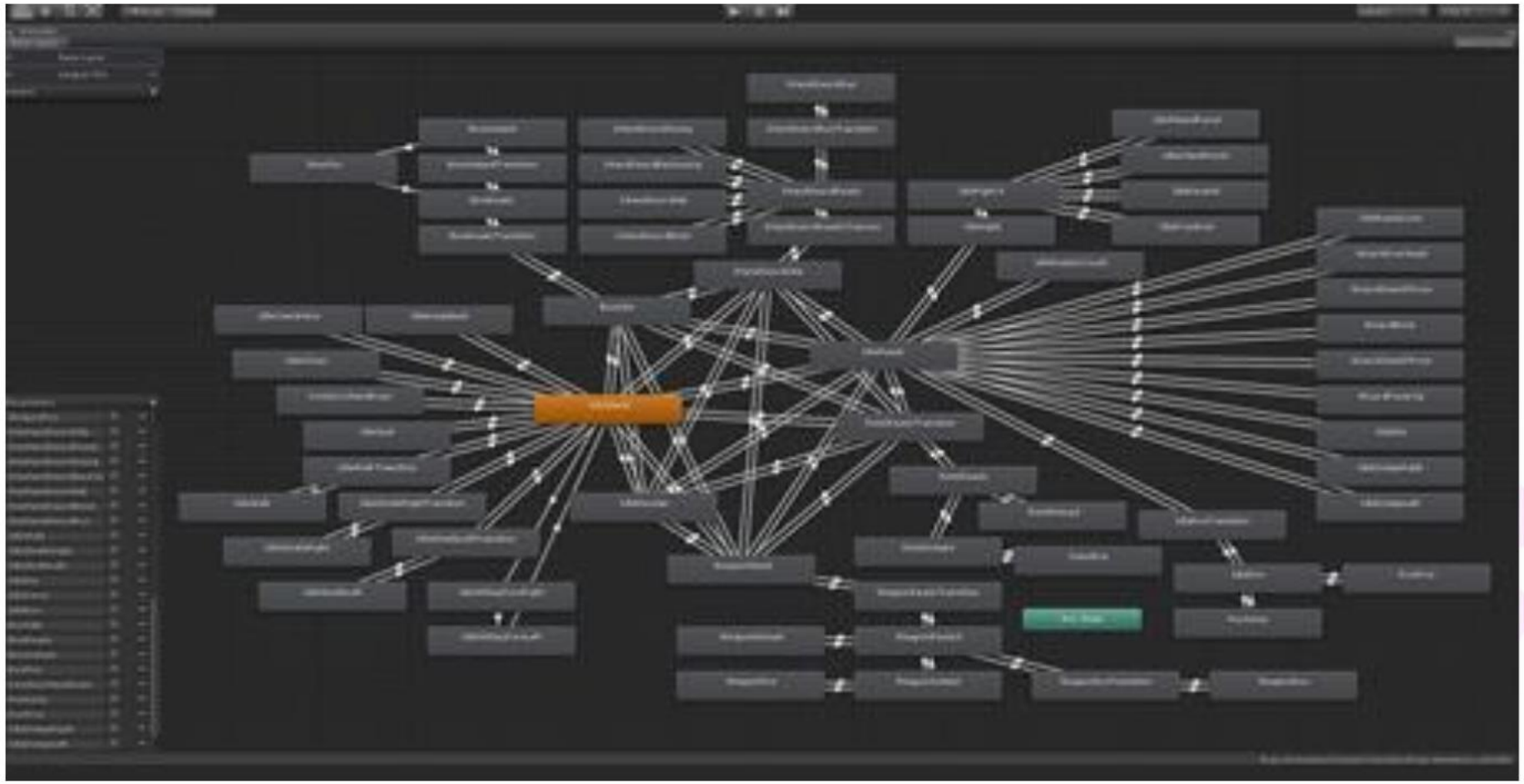
Is the AI for Pac-Man basic?

- chase or run.
- binary state machine?
- Toru Iwatani, designer of Pac-Man explained: “wanted each ghostly enemy to have a specific character and its own particular movements, so they weren’t all just chasing after Pac-Man... which would have been tiresome and flat.”
- the four ghosts have four different behaviors
  - different target points in relation to Pac-Man or the maze
  - attack phases increase with player progress
  - More details: http://tinyurl.com/238l7km

Finite State Machines (FSMs)

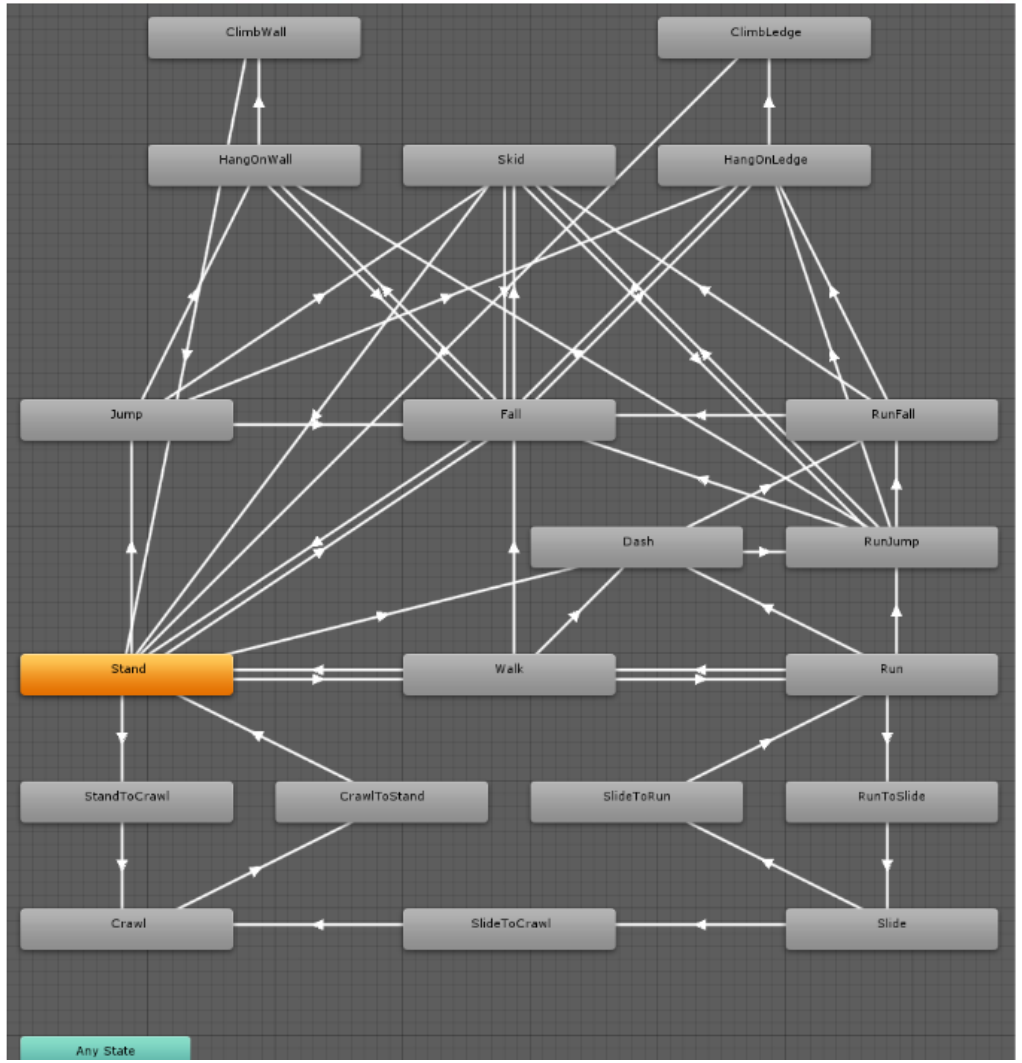
- ***Each frame:***
  - Something (the player, an enemy) does something in its state
  - It checks if it needs to transition to a new state
    - *If so, it does so for the next iteration*
    - *If not, it stays in the same state*
- ***Applications***
  - Managing input
  - Managing player state
  - Simple AI for entities / objects / monsters etc.

FSMs: States + Transitions



From http://twvideo01.ubm-us.net/o1/vault/gdc2016/Presentations/Clavet_Simon_MotionMatching.pdf

FSMs: Failure to Scale



No way to do long-term planning

No way to ask "How do I get here from there?"

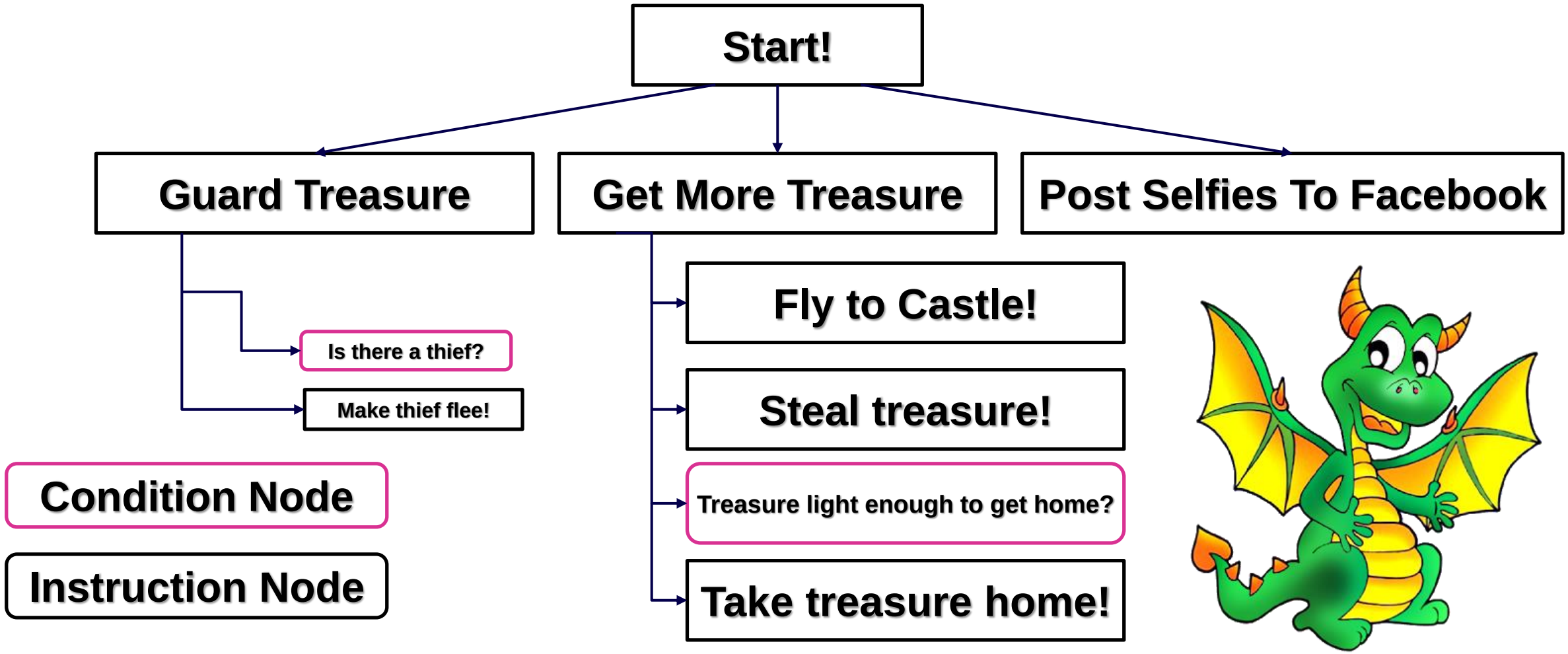
No way to reason about long-term goals

FSMs can get large and hard to follow

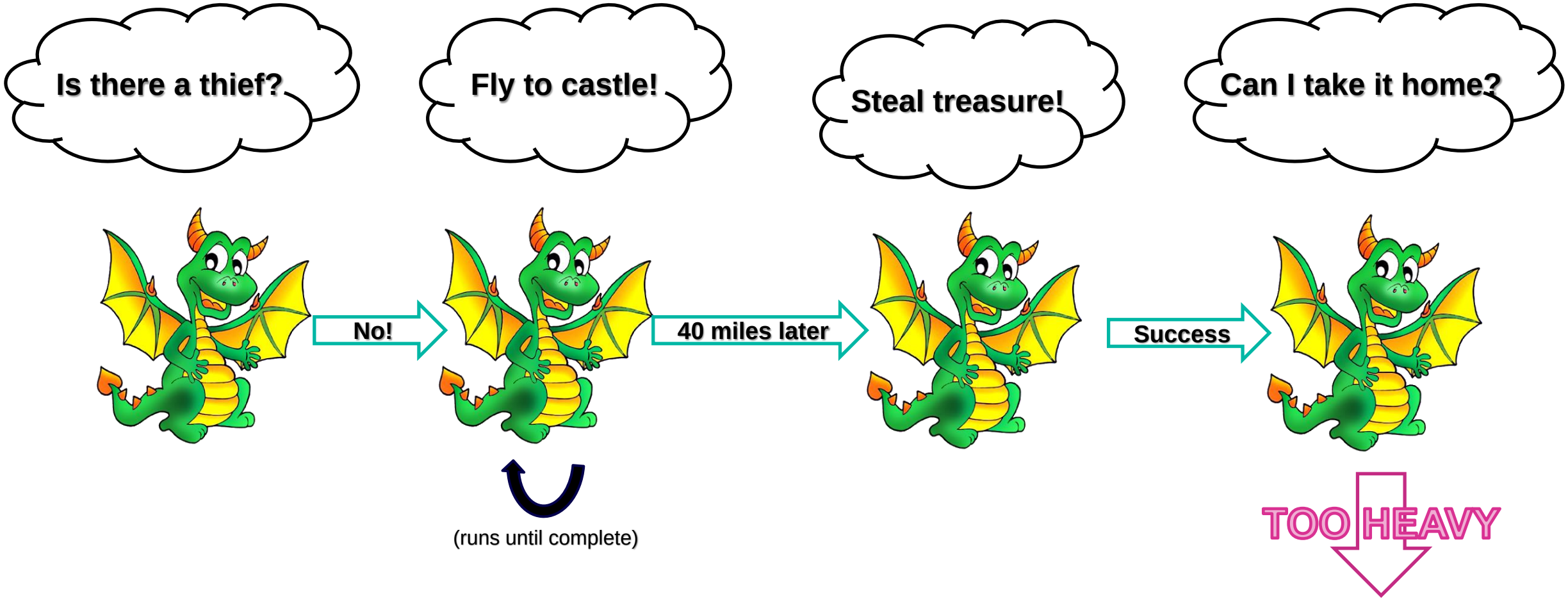
Can't generalize for larger games

From http://twvideo01.ubm-us.net/o1/vault/gdc2016/Presentations/Clavet_Simon_MotionMatching.pdf

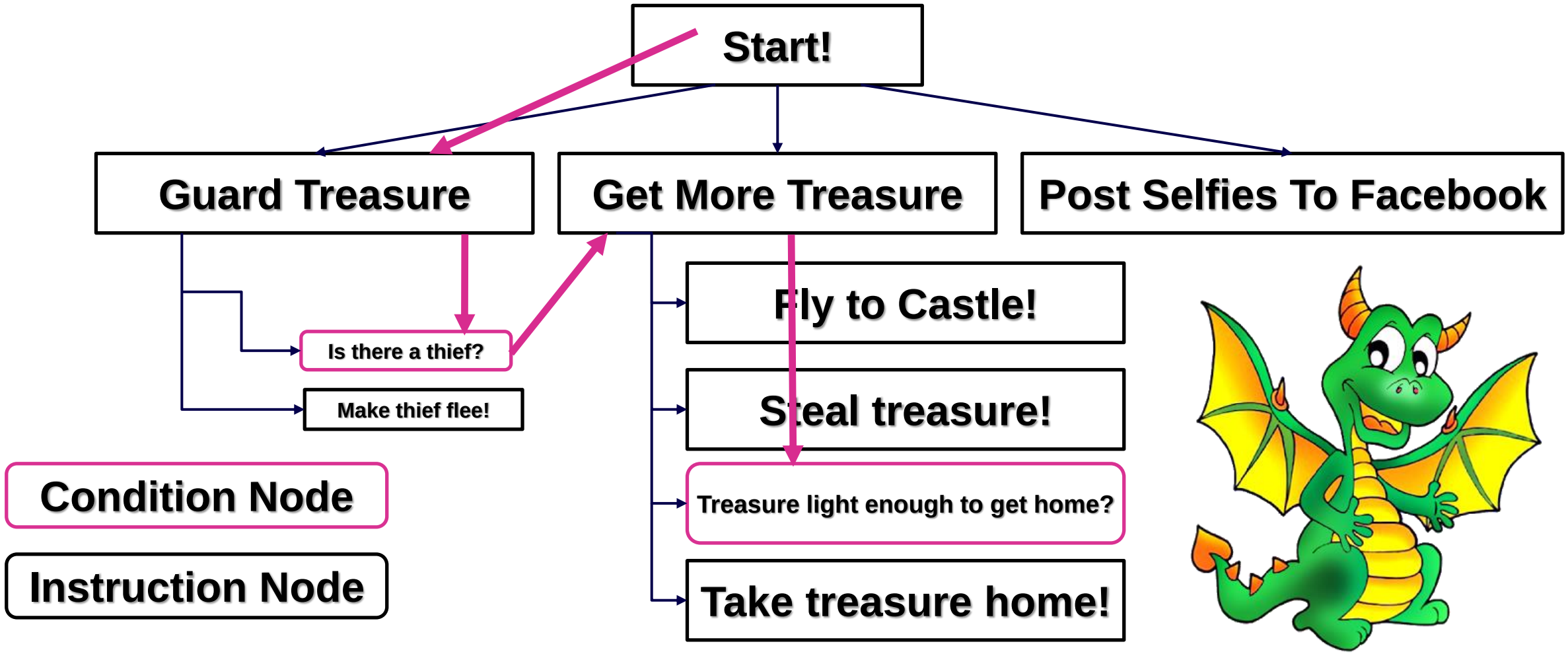
Behaviour Trees:
How To Simulate Your Dragon



Start!

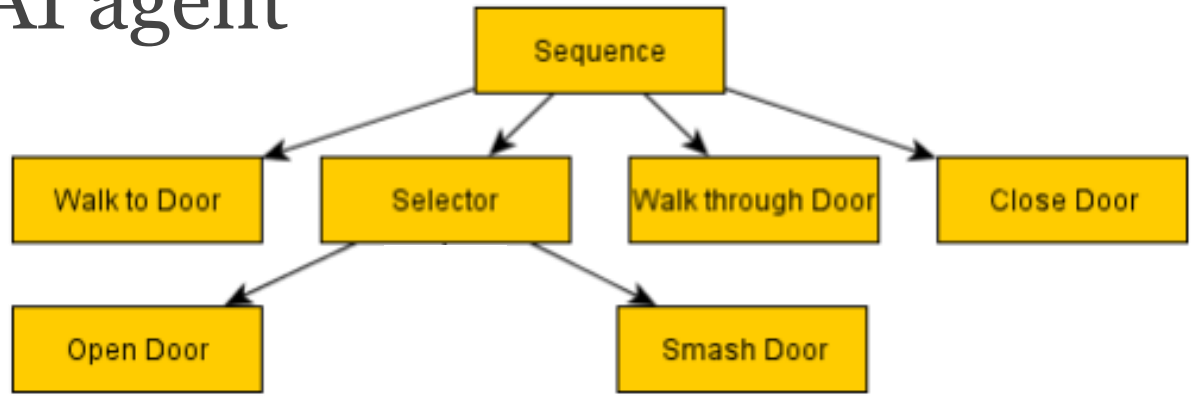


Behaviour Trees:
How To Simulate Your Dragon



Behaviour Trees

- flow of decision making of an AI agent
- tree structured
- ***Each frame:***
  - Visit nodes from root to leaves
    - *depth-first order*
    - *check currently running node*
      - succeeds or fails:
        - return to parent node and evaluate its Success/Failure
        - the parent may call new branches in sequence or return Success/Failure
      - continues running: recursively return Running till root (usually)



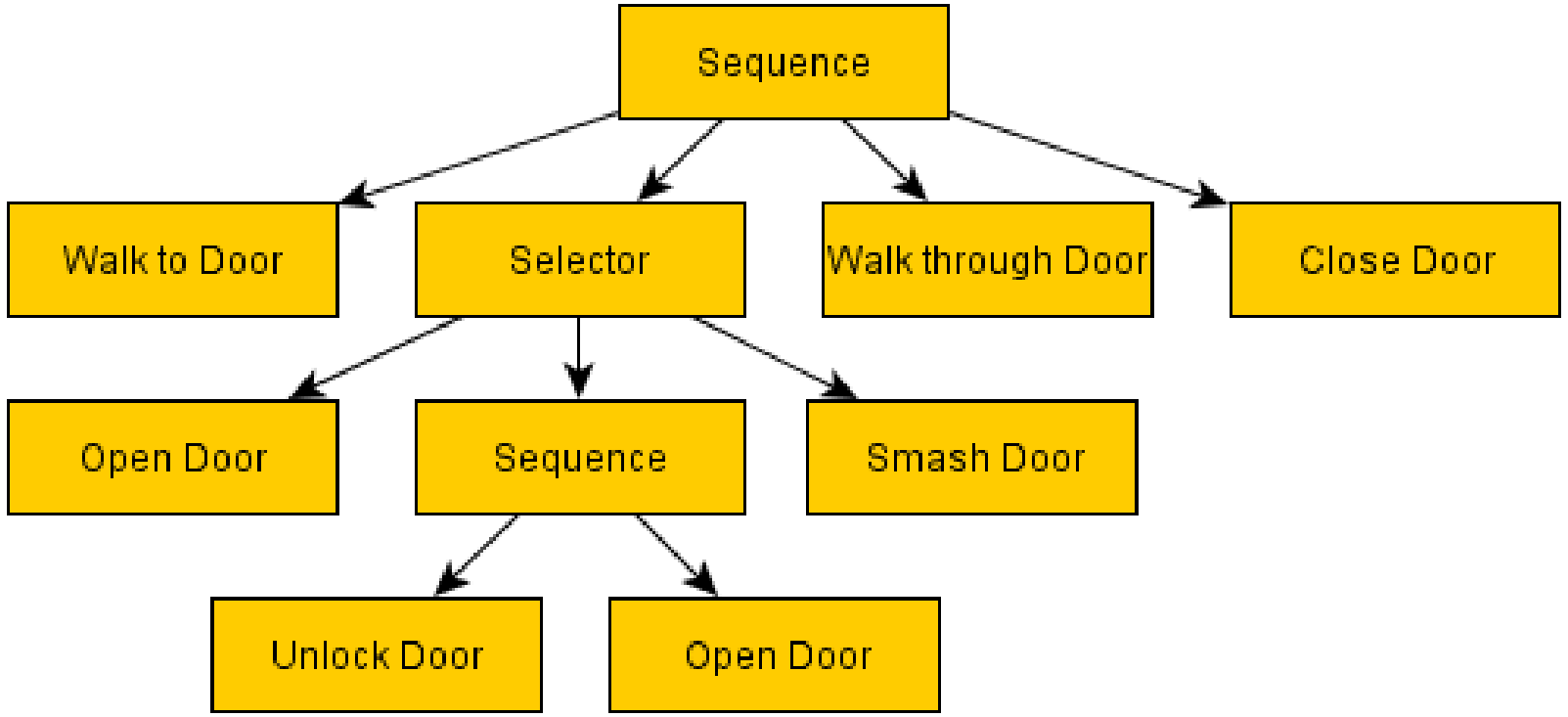
Behaviour Tree Elements

- leaves, are the actual commands that control the AI entity
  - upon tick, return: Success, Failure, or Running

- branches are utility nodes that control the AI's walk down the tree
  - loop through leaves: first to last or random
  - inverter: turn Failure -> Success
  - to reach the sequences of commands best suited to the situation

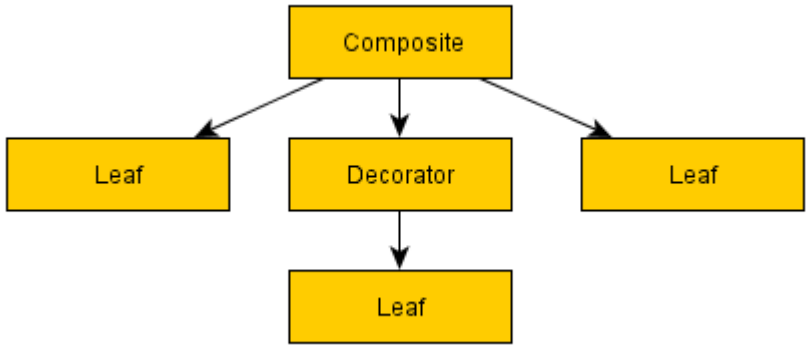
- trees can be extremely deep
  - nodes calling sub-trees of reusable functions
  - libraries of behaviours chained together

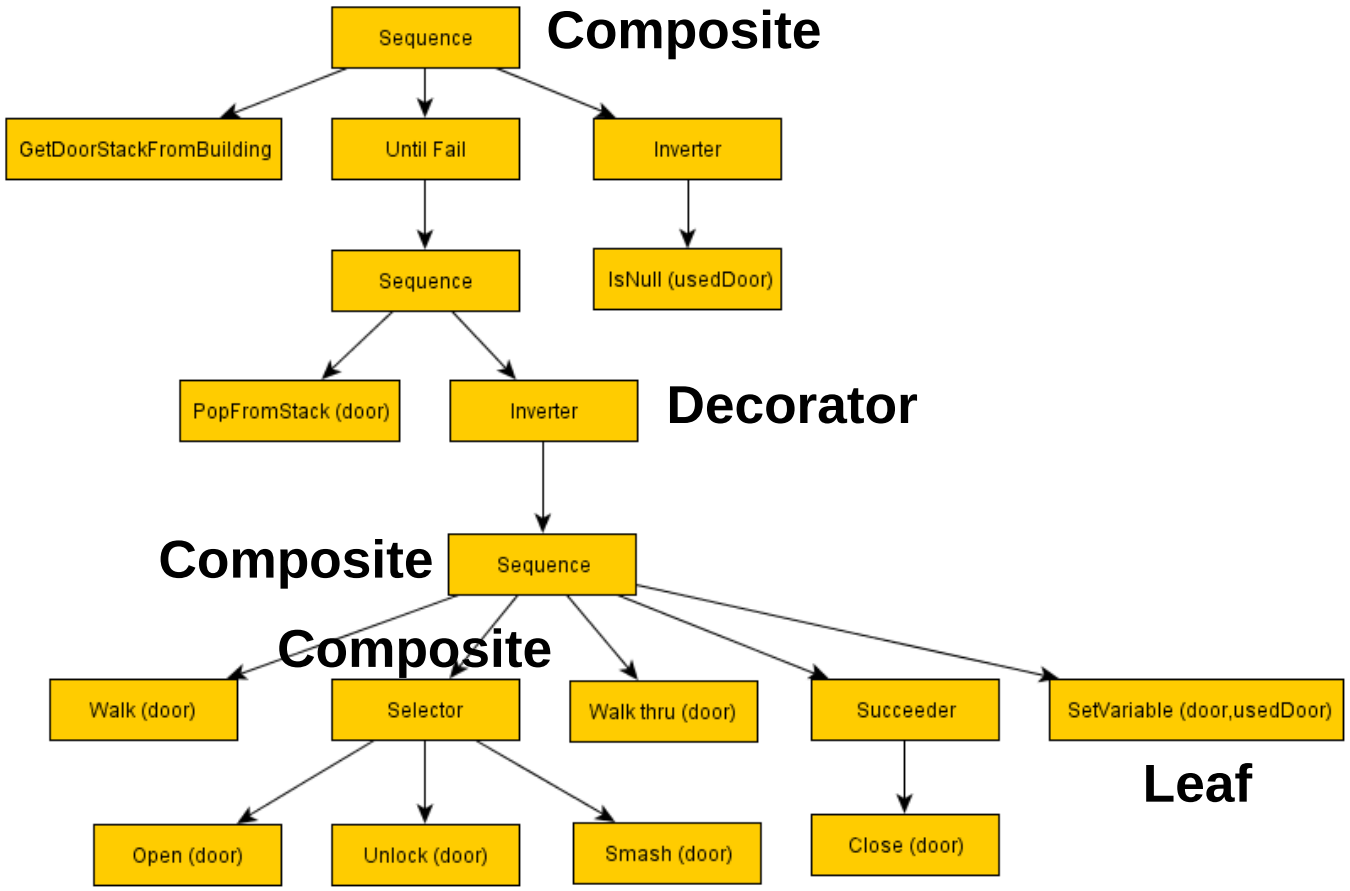
Schematic examples



https://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php

Types

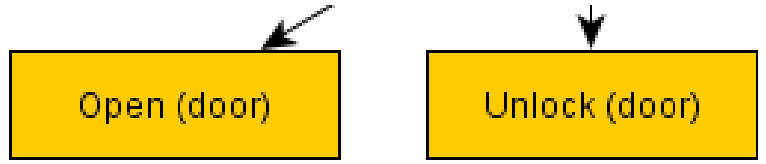




Behaviour Tree Elements

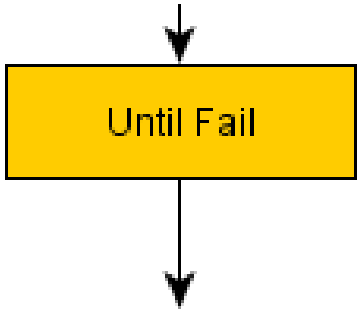
Leaf node

- A custom function, does the actual work
- Returns Running/Success/Failure



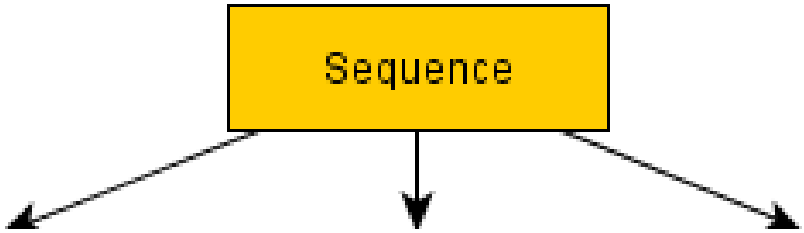
Decorator node

- has a single child
- Passes on Running/Success/Failure from child
  - may invert Success/Failure



Composite node

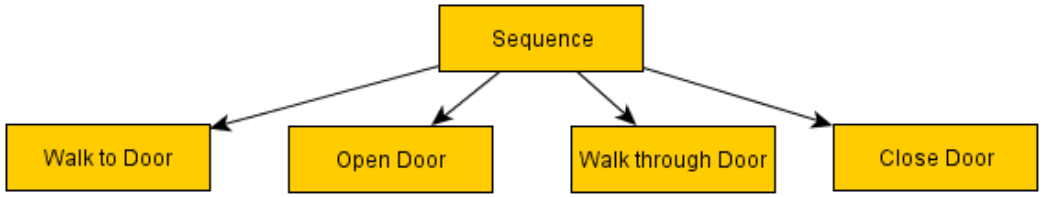
- has one or more children
- returns 'Running' until children stopped running



Useful Composites

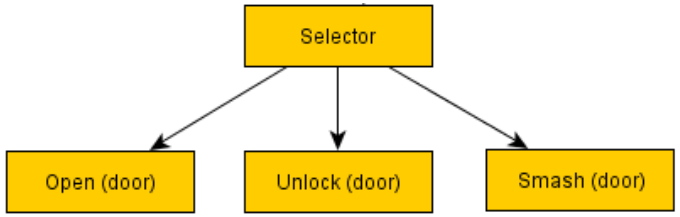
Sequence

- *execute all children in order*
- *Success if all children succeed (= AND)*



Selector

- *execute all children in order*
- *return Success if any child succeeded (= OR)*



Random Selectors / Sequences

- Randomized order of above composites

Useful Decorators

Inverter

- *Negates success/failure*

Succeeder

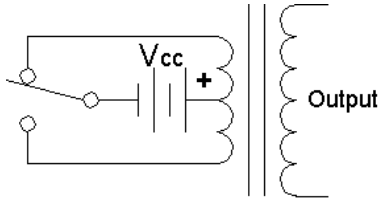
- always returns success

Repeater

- Repeat child N times

Repeat Until Fail

- Repeat until child fails



return “Success”;

Leaf Nodes

Functionality

- ***init(…)***
  - *Called by parent to initialize*
  - *Sets state to Running*
  - *Not called gain before returning Success/Failure*
- ***process()***
  - *Called every frame/tick the node is running*
  - *Does internal processing, interacts with the world*
  - *Returns Running/Success/Failure*

Example: *Walk to goal location*

- *Sets goal position for path finding*

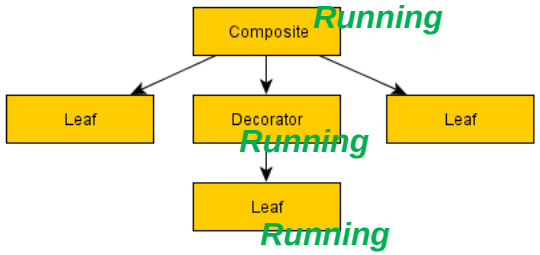
- Computes shortest path
- Sets character velocity
- Returns
  - success: Reached destination
  - failure: No path found
  - running: En route

Early exit?

- ***All parents of the currently running leaf node are running too***
- ***A node early in the tree can return Success/Failure***
  - Terminates children implicitly

- **Trying again?**
  - Re-initialize children with new parameters to init(…)

Example



- ***upon alarm***
  - abort sleeping
  - init running node

- ***try to sleep if alarm is off***
  - init sleeping node

Implementation example

Basics:

```
// The return type of behaviour tree processing
enum class BTState {
    Running,
    Success,
    Failure
};


// The base class representing any node in our behaviour tree
class BTNode {
public:
    virtual void init(Entity e) {};

    virtual BTState process(Entity e) = 0;
};
```

An if condition (inflexible)

```
// A general decorator with lambda condition
class BTIfCondition : public BTNode
{
public:
    BTIfCondition(BTNode* child)
        : m_child(child) {
    }

    virtual void init(Entity e) override {
        m_child->init(e);
    }

    virtual BTState process(Entity e) override {
        if (registry.motions.has(e)) // hardocded
            return m_child->process(e);
        else
            return BTState::Success;
    }
private:
    BTNode* m_child;
};
```

Implementation example II

A leaf node

```
class TurnAround : public BTNode {
private:
    void init(Entity e) override {
    }

    BTState process(Entity e) override {
        // modify world
        auto& vel = registry.motions.get(e).velocity;
        vel = -vel;

        // return progress
        return BTState::Success;
    }
};
```

Behaviour Trees are Modular!

- Can re-use behaviours for different purposes
- Can implement a behaviour as a smaller FSM
- Can be data-driven (loaded from a file, not hard coded)
  - *JSON?!*
- Can easily be constructed by non-programmers
- Can be used for *goal based programming*