Game Play and AI

Helge Rhodin
Overview

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

Next:
- *Planning (long term)*
‘Modern’ AI?

Machine learning has the problem of 1. training, 2. testing
• Takes ages for large models
• Can be real-time for small models (linear regression)

Opportunity of large language models (LLMs)
• General purpose
• Text is a very flexible interface
  • Understood by humans
  • Understood by machines
  • No need to specify the interface (what your game needs) in advance
‘Modern’ AI?

• **Use ChatGPT?**

• **Chat GPT provides a text-based interface**
  - *Summarise your game state as text (automatically)*
    - “User is at a distance of 10m, you have an arrow and a sword. Which one should you use? Answer with a single word.”
    - If(output == “sword”) …
State machines
Gameplay

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

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

// stop
if (walking && !wantToWalk)
{
    PlayAnim(StopAnim);
    walking = false;
}
```
Finite State Machines: States + Transitions
FSM Example: Pac-Man Ghosts
FSM Example: Pac-Man Ghosts

- Wander Maze
- Return to Base
- Chase Pac-Man
- Flee Pac-Man

Transitions:
- Pac-Man Lost → Wander Maze
- Pac-Man Seen → Power Pellet Expires
- Pac-Man Eats Power Pellet → Ghost Attacked
- Ghost Attacked → Flee Pac-Man
- Pac-Man Eats Power Pellet → Flee 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!

Guard Treasure
- Is there a thief?
- Make thief flee!

Get More Treasure
- Fly to Castle!
- Steal treasure!
- Treasure light enough to get home?
- Take treasure home!

Post Selfies To Facebook
Start!

Is there a thief?  
No!

Fly to castle!

40 miles later

Steal treasure!

Can I take it home?

Success

(runs until complete)

TOO HEAVY

© Alla Sheffer, Helge Rhodin
Behaviour Trees: How To Simulate Your Dragon

Start!

Guard Treasure
- Is there a thief?
  - Make thief flee!

Get More Treasure
- Fly to Castle!
  - Steal treasure!
    - Treasure light enough to get home?
      - Take treasure home!

Post Selfies To Facebook
BTs are state machines

- With structure (tree)
- With well-defined interfaces (fail-success-running)
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
  - e.g., walk one step
  - upon tick, return: Success, Failure, or Running
- branches are utility nodes that control the AI’s walk down the tree
  - e.g., door unlocked?
  - loop through children: 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
Types

Composite

Leaf

Decorator

Composite

Leaf

Decorator

Composite

Sequence

GetDoorStackFromBuilding

Until Fail

Inverter

Sequence

isnull(usedDoor)

PopFromStack(door)

Inverter

Sequence

Walk(door)

Selector

Walk thru(door)

Succesor

SetVariable(door, usedDoor)

Open(door)

Unlock(door)

Smash(door)

Close(door)
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

```java
return "Success";
```

![Diagram of Inverter](image)
Leaf Nodes

**Functionality**

- **init(...)**
  - Called by parent to initialize
  - Sets state to **Running**
  - Not called again 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
How to implement a tree in C++?
Implementation example

Basics:

```cpp
// 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)

```cpp
// 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)) // hardcoded
            return m_child->process(e);
        else
            return BTState::Success;
    }
private:
    BTNode* m_child;
};
```
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
Modular design?
Modular design?

Tree construction

```java
// Tree construction
// leaf nodes
RunNSTeps run3(3);
TurnAround turn;

// conditional turn sub-tree
BTIfCondition turn_right = BTIfCondition(&turn,
    [](Entity e) {return registry.motions.get(e).velocity < 0; });
BTRunPair root = BTRunPair(&turn_right, &run3);
```

Game loop

```java
Entity human;
root.init(human);
for (int i = 0; i < 100; i++)
    BTState state = root.process(human);
```
Decorators - Conditions

```cpp
class BTIfCondition : public BTNode {
    std::shared_ptr<BTNode> m_child;
    std::function<bool(ECS::Entity)> m_condition;

public:
    BTIfCondition(std::shared_ptr<BTNode> child, std::function<bool(ECS::Entity)> condition)
        : m_child(std::move(child)), m_condition(condition) {}

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

    virtual BTState process(ECS::Entity e) override {
        if (m_condition(e))
            return m_child->process(e);
        else
            return BTState::Success;
    }
};

Decorators - Conditions

Instantiation

BTNode standing = BTIfCondition(child_ptr, [](ECS::Entity e) { return ECS::registry<Motion>::get(e).velocity == 0; })
```
AND Sequences

- Iterate through children until end or until child returns **Failure**
- Similar to ‘and’ in ‘if(child[0] && child[1] && …)’
  - Expressions following the first ‘false’ will be ignored

- Further useful composites:
  - Repeat N times
  - Repeat indefinitely
  - Negate **Success/Failure**
  - OR Sequence
  - If … else
  - Exit condition
  - What else???
Leaf Nodes – Generic Version

How can we apply the same BT on different entities?

• How to store internal states?
  • store the state for every entity
  • use an std::map

Minor addition to ECS::Entity
// Comparator to use as key in std::map
bool operator <(const Entity& rhs) const
{
    return id < rhs.id;
}

class RunThreeMeters : public BTNode
{
    std::map<ECS::Entity, int> n;
    void init(ECS::Entity e) {
        n[e] = 3;
    }

    BTState process(ECS::Entity e) {
        // update internal state
        n[e]--;

        // modify world
        ECS::registry<Motion>.get(e).position += ECS::registry<Motion>.get(e).velocity;

        // return progress
        if (n[e] > 0)
            return BTState::Running;
        else
            return BTState::Success;
    }
};
ECS solves every problem?

**Entity**

**Component**

**System**

When not to use ECS?

- **When information is not shared across Systems**
- **AND ECS does not fit naturally**
  - multiple components of the same type associated to the same entity
    - previous slide: multiple class instances store the same information type in a different context
  - Entities and Components are still be useful locally
    - Storing Components in ECS instead of locally is equally performant. Use ECS whenever possible!
    - The unique Entity ID can still be useful to associate local information to a global entity!