# 07 - Artificial Intelligence and Non-Player Characters

CS 3160 - Game Programming

Max Gilson

# Artificial Intelligence (AI)

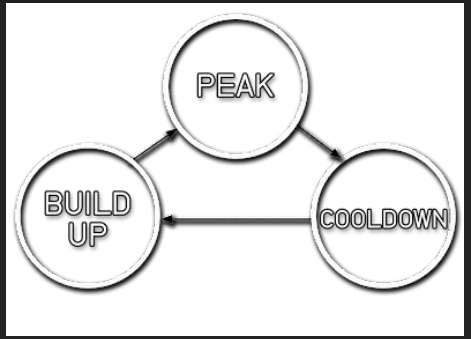
- In most realms of computer science “AI” often refers to some type of machine learning
- In games, AI refers to the ability for characters or entities in your game to make decisions and take actions
  - Some games may use some form of machine learning for characters, but this is uncommon
- Adding AI to your game can make your game feel more dynamic, allow for emergent gameplay, and immerse the player
  - An “unintelligent” AI that only runs towards the player is very boring and predictable

# Artificial Intelligence (AI) (cont.)

- AI can be utilized in many different ways in games:
  - Non-player characters (NPCs)
  - Cooperative agents to help the player
  - Challenging opponents that are hard to predict
  - World simulation where characters have their own motivations and routines
  - Dynamically adjusting how the game plays and providing more challenge
  - Faction and reputation systems to make player's decisions feel more meaningful
  - Group dynamics for squads to coordinate with each other
  - Reducing decisions the player has to make
  - And many more: https://www.youtube.com/@AIandGames/videos

# Artificial Intelligence Examples

- The Sims allows the player to control characters (Sims) in a home/world simulation game
    - These Sims also make their own decisions based on their current stats (hunger, fun, tiredness, etc.)
- Fear 1 and Fear 2 allows for opponents to operate within a squad to strategize
    - Enemy squads can communicate with each other if they spot something, out-maneuver the player, regroup, etc.
- Left 4 Dead uses an AI director that actively balances how the game plays
    - If the players are playing too well, the AI director dynamically adds more zombies

# Goals of Artificial Intelligence in Games

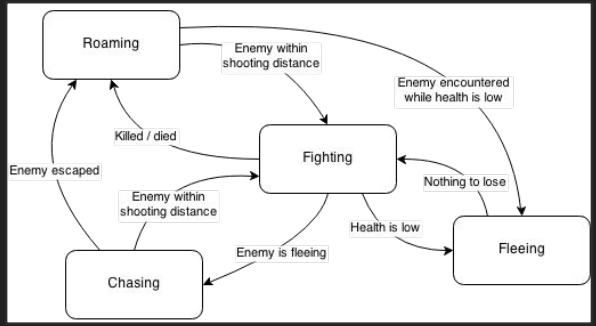
- In academic settings, we want AI to be smarter and better in every way
- In games, AI that is smarter and better than the player all the time is not fun
  - Imagine an AI opponent that is so smart and accurate that it is impossible to beat
- Some goals of our AI systems may be:
  - Enhance realism or believability of characters
  - Increase or decrease the challenge for the player
  - Adapt to player behavior to keep the gameplay interesting
  - Innovate or provide entirely new gameplay elements
- Our players are the ones playing the game, not our AI characters
- The AI should be designed to make the game more fun for the player
  - Making the AI too smart is not our goal

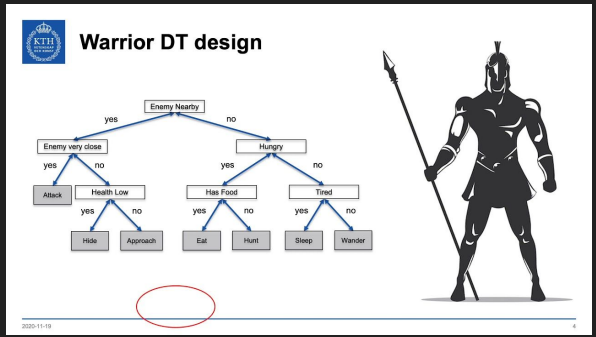
# The Player’s Perception of AI

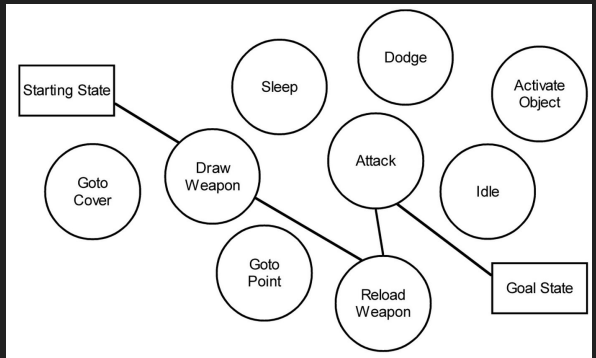
- The player is not usually thinking about the characters in your game as a bunch of lines of code the reality is:
  - Players have certain expectations of your characters
    - “This person must have a story to tell or some history” even if they are a generic placeholder character
    - Players expect AI to make somewhat realistic decisions
  - Players will use real world explanations for emergent gameplay that was never programmed
    - My companion is triggered a trap that hurt me they must be out to get me
    - https://www.youtube.com/watch?v=3cKQIjpTLog
  - Players ability to perceive actual intelligence is poor
    - Characters that do more/less damage are perceived as smarter/dumber
    - https://youtu.be/kda7rz5qFtI?si=D-luEtwrfuneW8e6&t=2010
    - Multiplayer bots often fool players into thinking they are battling other real players
- Your AI should maintain an illusion of intelligence

# Common AI Implementations

- Finite State Machine
  - What's my current state?
  - What must happen to transition me to a new state?
- Behavior Trees
  - If this then … over a tree
- Goal Oriented Action Planning
  - Initial state + end goal state + a set of actions
  - Construct a sequence of actions that achieves the end goal
- Hierarchical Task Network Planner
- Rule-Based
  - Tic-Tac-Toe
- Utility AI

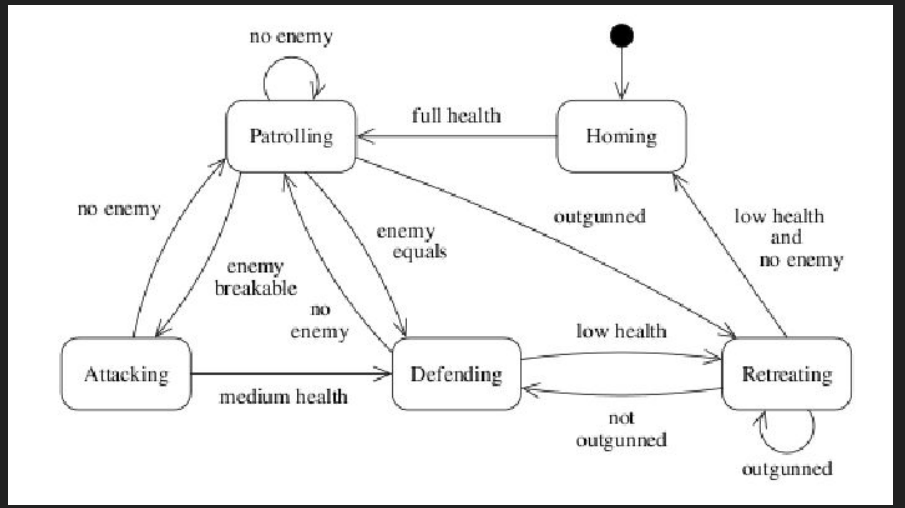


# Finite State Machine (FSM)

- FSMs have few key items:
  - The current state
  - A list of states
  - Transitions from each state
- The state is the action being performed by the AI
- The AI is constantly checking if it needs to transition out of the current state
- Can make for very complex AI interactions
- Can be very complicated to build if there are many states (not very scalable)

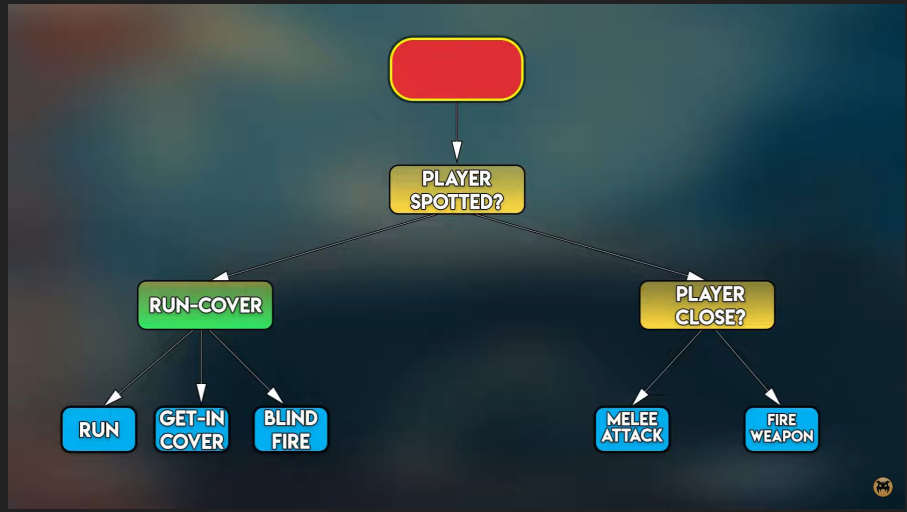



# FSMs in Games

- Half Life
- Quake
- Doom
- Batman: Arkham Asylum

# Behavior Tree

- A directed acyclic graph: fixed direction, no looping
- Node types:
  - Root node - the beginning of the AI's decision making
  - Leaf node - the action performed by the AI
  - Selector - selects a leaf from children based on a condition
  - Sequence - leafs get executed in sequence
- Behavior trees allow for easily visualizing the decision making of the AI
- Easier for non-programmers to work with
- Reusable amongst many AI agents

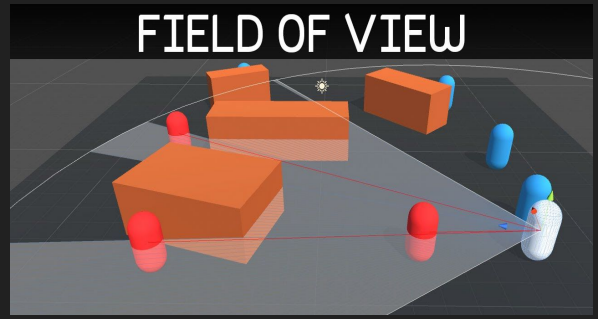


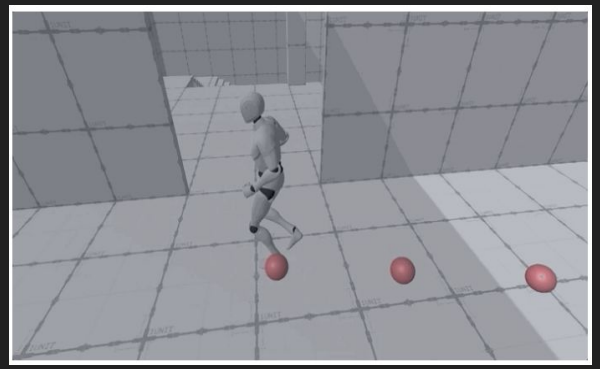
# Behavior Trees in Games

- Halo
- Alien Isolation
- Far Cry
- Bioshock


# AI Sensing The World

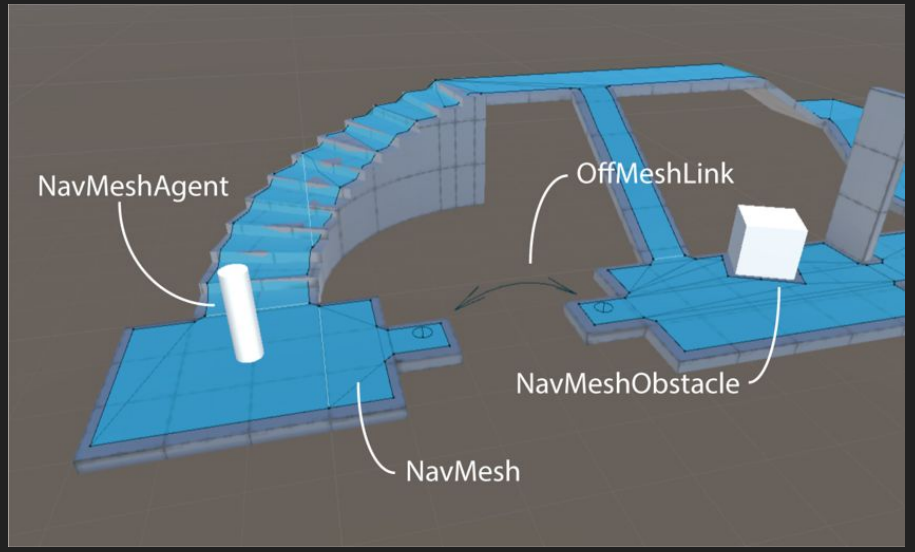
- The AI has to sense to world to react to it and make decisions:
  - Line of sight and FOV
    - Raycasts, colliders, view cones
  - Sound
    - Colliders/triggers
    - Breadcrumbs
  - Light level (stealth games)
  - Objects moved/manipulated
  - Blackboard
    - Shared list of information that NPCs share i.e. last known player position

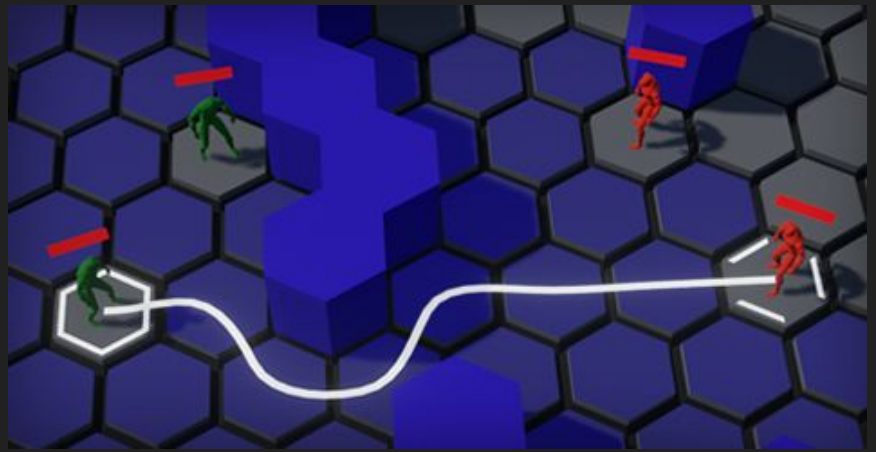


# AI Navigating The World

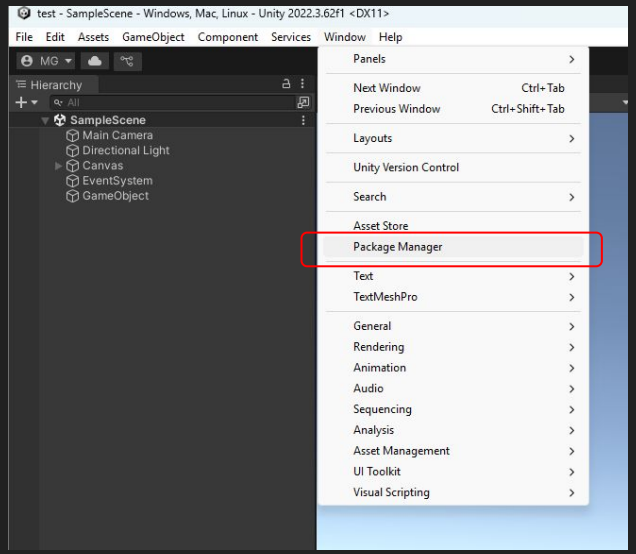
- AI has to know what parts of the world are traversable
- Just because you build stairs in your game does not mean the AI knows how to use them by default!
  - Navigation Meshes (Nav Mesh)
    - Mesh weights
    - Transitions between meshes
    - Obstacles
  - Grid based
- Path Planning
  - Once an AI know what is traversable, how does it find a path to it's target location?
  - A* (great for grids and efficient)
  - RRT (sampling based)
  - https://www.kaufer.org/WebRRT/#
  - https://qiao.github.io/PathFinding.js/visual/
  - https://www.reddit.com/r/Unity3D/comments/vfghfr/pathfinding_with_sparse_voxel_octrees/

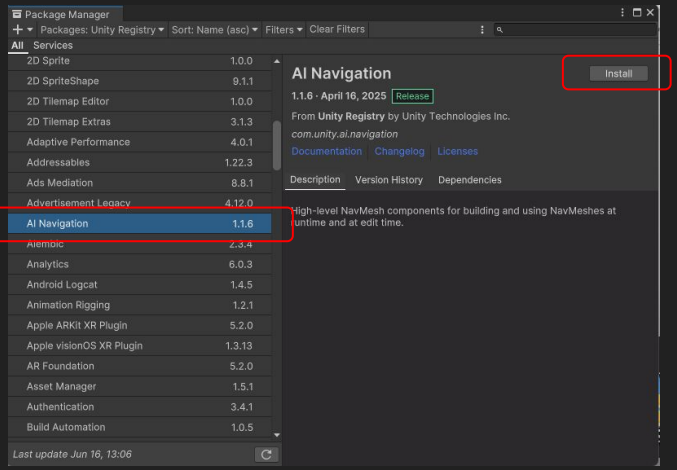


# AI In Unity

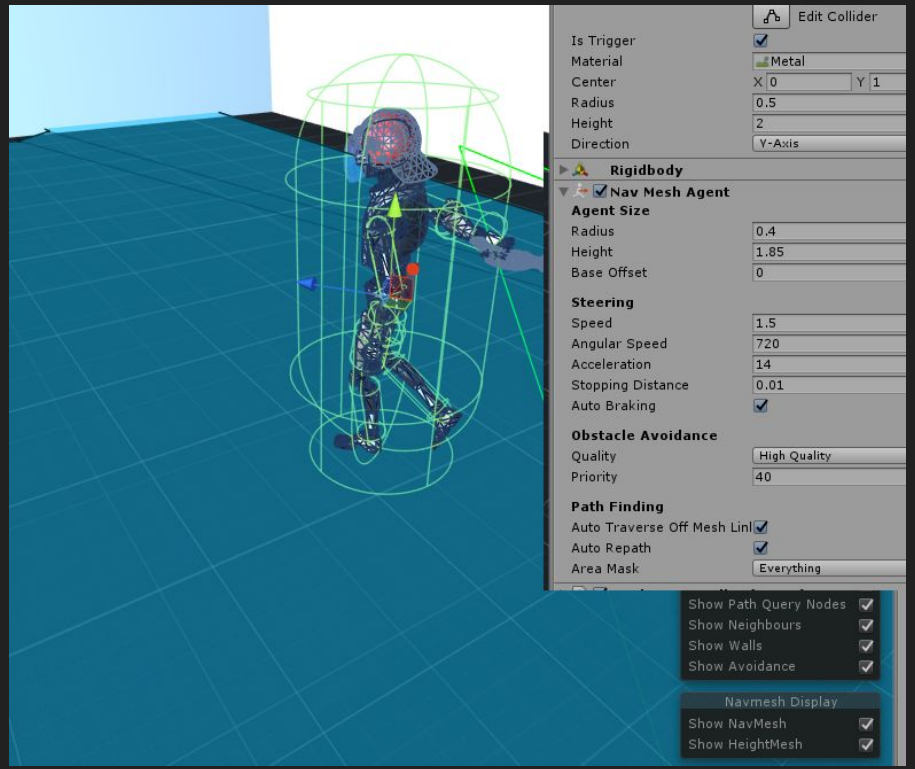
- AI navigation can easily be added to Unity
- AI behaviors and logic has to be custom
- Requires installing “AI Navigation” package



# AI In Unity (cont.)

- Navigation in Unity:
- NavMeshSurface - creates a navigation mesh area
  - Can have different agent restrictions and area settings
- NavMeshAgent - creates an AI agent that can path plan
  - Can have radius, height, and speeds set
- NavMeshObstacle - creates an obstacle that breaks up the nav mesh surface



# Custom FSM in Unity

EnemyStateMachine.cs:

```
using UnityEngine;
using UnityEngine.AI;

public class EnemyStateMachine : StateMachine
{
    [HideInInspector]
    public NavMeshAgent navMeshAgent;

    public void Start()
    {
        // When the enemey first spawns, go into the chase state
        navMeshAgent = GetComponent<NavMeshAgent>();
        SetCurrentState(new Chase(this));
    }

    public void FixedUpdate()
    {
        // Every fixed update execute the current state
        ExecuteCurrentState();
    }

    public void Update()
    {
        // Every update check if we need to transition to a new state
        CheckForTransitions();
    }
…
```

EnemyStateMachine.cs (continued):

```
…
    public override void CheckForTransitions()
    {
        // All the transitions between states are defined here
        switch (currentState)
        {
            // Transitions from Evade to other states
            case Evade:
                // If we are in the Evade state and the '1' key is pressed
                if (Input.GetKeyDown(KeyCode.Alpha1))
                {
                    // Transition to the Chase state
                    SetCurrentState(new Chase(this));
                }
                break;
            // Transitions from Chase to other states
            case Chase:
                // If we are in the Chase state and the '2' key is pressed
                if (Input.GetKeyDown(KeyCode.Alpha2))
                {
                    // Transition to the Evade state
                    SetCurrentState(new Evade(this));
                }
                break;
        }
    }
}
```

# Custom FSM in Unity (cont.)

StateMachine.cs:

```
public abstract class StateMachine : MonoBehaviour
{
    protected State currentState;

    public void SetCurrentState(State state)
    {
        // Whenever a new state is set
        var message = "";
        if (currentState != null)
        {
            // Exit the old one if it exists
            // Note: The currentState will be null if we just started the state machine
            message += $"Exiting {currentState.GetType()} state... ";

            // Call exit state to clean up the state if necessary
            currentState.ExitState();
        }

        // Set the current state

        currentState = state;
        message += $"Entering {state.GetType()} state";
        Debug.Log(message);

        // Call enter state to setup the state if necessary
        currentState.EnterState();

    }
...
```

StateMachine.cs (continued):

```
...

    public void ExecuteCurrentState()
    {
        // Execute the current state
        currentState.ExecuteState();
    }

    public virtual void CheckForTransitions()
    {
        // Nothing is defined here... on purpose
        // We want each class derived from StateMachine
        // to make their own decisions on how to transition
        // to other states
    }
}
```

# Custom FSM in Unity (cont.)

StateMachine.cs (continued):

```
public class Chase : State
{
    public Player player;
    public Chase(StateMachine stateMachine) : base(stateMachine)
    {

    }

    public override void EnterState()
    {
        // When we enter the Chase state, find the player
        player = Transform.FindFirstObjectByType<Player>();
    }

    public override void ExecuteState()
    {
        // Check if the player was not found, if so, there's no chasing to do, so return
        if (player == null) return;

        // Check if our state machine is an enemy state machine
        // This allows us to define different "Chase" state behaviors for different state machines
        // Imagine we have a "CompanionStateMachine", we could add an additional if statement for it
        // Then, inside that if statement, we would define custom behaviour for companions
        if (stateMachine.GetType() == typeof(EnemyStateMachine))
        {
            // Fetch the NavMeshAgent from the state machine
            var navMeshAgent = ((EnemyStateMachine)stateMachine).navMeshAgent;
            // Get the position of the player so we can chase them
            var chasePosition = player.transform.position;
            navMeshAgent.SetDestination(chasePosition);
        }
    }

    public override void ExitState()
    {

    }
}
```

StateMachine.cs (continued):

```
public class Evade : State
{
    public Player player;
    public Evade(StateMachine stateMachine) : base(stateMachine)
    {
    }

    public override void EnterState()
    {
        // When we enter the Evade state, find the player
        player = Transform.FindFirstObjectByType<Player>();
    }

    public override void ExecuteState()
    {
        // Check if the player was not found, if so, there's no evading to do, so return
        if (player == null) return;

        // Check if our state machine is an enemy state machine
        if (stateMachine.GetType() == typeof(EnemyStateMachine))
        {
            // Fetch the NavMeshAgent from the state machine
            var navMeshAgent = ((EnemyStateMachine)stateMachine).navMeshAgent;
            // Calculate the direction we must evade in and set it as our destination
            var evadeDirection = navMeshAgent.transform.position - player.transform.position;
            var evadePosition = navMeshAgent.transform.position + evadeDirection;
            navMeshAgent.SetDestination(evadePosition);
        }
    }

    public override void ExitState()
    {

    }
}
```

# Custom FSM in Unity (cont.)

StateMachine.cs (continued):

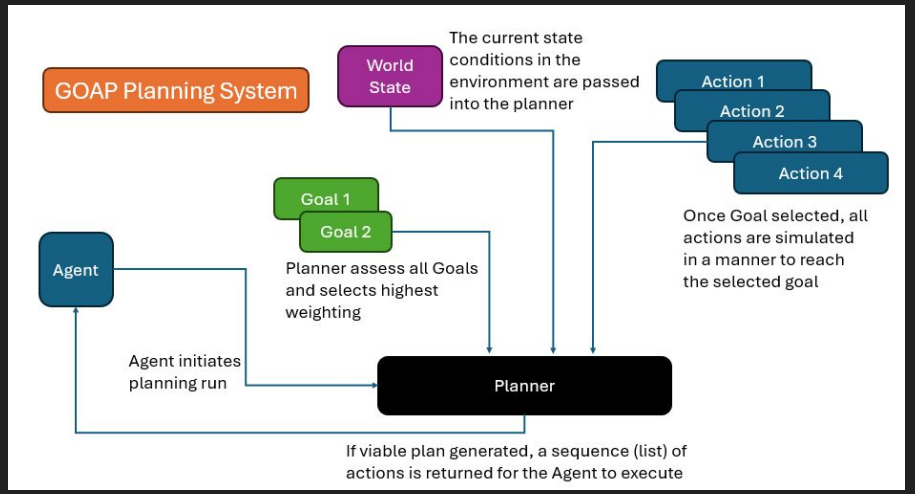
```
public abstract class State
{
    protected StateMachine stateMachine;

    // Allow every state to hold a reference to the state machine
    // This is used for grabbing data i.e. NavMeshAgents or Rigidbodies
    // that are attached to the AI
    public State(StateMachine stateMachine)
    {
        this.stateMachine = stateMachine;
    }

    // Abstract methods that all states must implement
    public virtual void EnterState()
    {
    }

    public virtual void ExecuteState()
    {
    }

    public virtual void ExitState()
    {
    }
}
```

# AI Case Study 1 - The Sims

- The Sims is a classic example of revolutionary AI technology called “Utility AI”
- Characters make decisions based on highest score of needs
  - Each need has a score based on the level of the need
  - Some needs have higher scores at the same levels than others
    - Example: hunger at 50% might have a score of 100 but social at 50% might have a score of 25
- Allows for emergent gameplay because even slightly different needs levels drastically changes what actions the characters take
- https://www.youtube.com/watch?v=9gf2MT-IOsg


# AI Case Study 2

- The game Fear also implemented groundbreaking AI with GOAP
- Enemies analyzed a handful of actions they had and planned a series of actions to take to accomplish a goal
    - Goals - i.e. kill player
    - Actions - i.e. shoot
    - Preconditions - i.e. has ammo
- Enemies also shared information with each other via a blackboard
    - If one enemy yells “he’s flanking us” all of the other enemies can be informed and change their actions
- https://www.youtube.com/watch?v=PaOLBOuyswI



# Machine Learning in Games

- Machine learning is typically not used in games for performance reasons but is still useful in some ways:
  - AI generated art assets
  - LLM AI conversations
    - https://www.youtube.com/watch?v=aihq6jhdW-Q
  - Clustering player behavior for analysis
  - Training AI for NPC interactions or QA