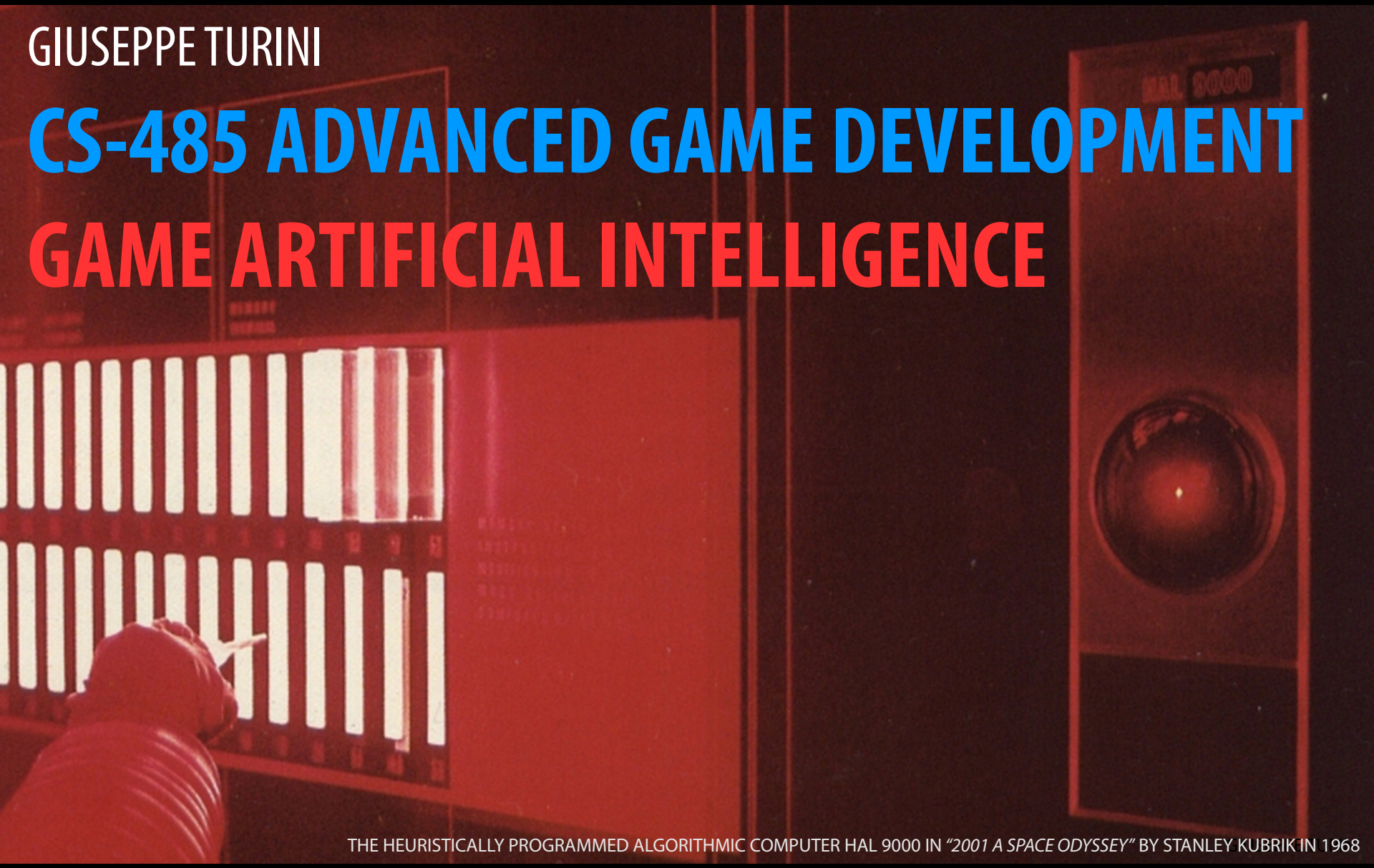
GIUSEPPE TURINI

CS-485 ADVANCED GAME DEVELOPMENT

GAME ARTIFICIAL INTELLIGENCE



THE HEURISTICALLY PROGRAMMED ALGORITHMIC COMPUTER HAL 9000 IN *"2001 A SPACE ODYSSEY"* BY STANLEY KUBRIK IN 1968

HIGHLIGHTS

Artificial Intelligence in Video Games
Most Influential Artificial Intelligence Video Games
Finite State Machines
- Definition and Main Concepts of Finite State Machines
- Implementation of a Simple FSM in Unity
- Implementation of an Advanced FSM in Unity
- Using a Finite State Machine Framework

Game AI Sensory Systems
- Introduction to Sensory System Design Pattern and Senses and Aspects
- Implementation of a Basic AI Sensory System in Unity

Behavior Trees
- Behavior Tree Definition, Structure, and Elements
- Control Flow and Execution Nodes, Tree Traversal and Ticks

Using a Game AI Engine (RAIN) in Unity

AI IN VIDEO GAMES

In video games, artificial intelligence (AI) is used to generate intelligent actions.
It is mainly used for Non-player Characters (NPC) to simulate human-like behaviors.

Game AI often refers to a broad set of algorithms/techniques from the following fields:

- Artificial Intelligence
- Control Theory
- Robotics
- Computer Graphics
- Computer Science

Game AI is focused on appearance of intelligence and good gameplay.
Workarounds and cheats are acceptable and downgrading is sometimes necessary.

MOST INFLUENTIAL AI VIDEO GAMES 1

Sim City by Maxis - 1989

A simulation and a city-building game.

Innovations in game AI:

Pioneer innovative gameplay: controlling complex sim.

Each element of the city is modeled in a realistic (AI) way.

Sim system properties perfectly balanced to improve the gameplay.

Creatures by Millenium Interactive - 1996

An artificial life program to breed and teach weird animals how to behave.

Innovations in game AI:

Pioneer application of machine learning into an interactive game.

Neural networks are used by the creatures to learn what to do.

Breakthrough in research for the modeling of creature behaviors.

MOST INFLUENTIAL AI VIDEO GAMES 2

Half Life by Valve - 1998

A sci-fi first person shooter (FPS) set in a contaminated research facility.

Innovations in game AI:

Interactive cut scenes (scripting and AI).

A guard (AI NPC) accompanies the player through some levels.

Effective pioneer squad AI in the latter stages of the game.

AI perfectly tweaked and integrated into the storyline.

Thief by Looking Glass Studios - 1998

A single player stealth-based first-person game.

Innovations in game AI:

Accurate sensory model allows AI units to react to light and sounds.

AI units use audio recordings to tell the player their current state.

MOST INFLUENTIAL AI VIDEO GAMES 3

Total War by The Creative Assembly - 2000

3D turn-based strategy battle game on a risk-like map with real-time tactics.

Innovations in game AI:

Thousands of soldiers (AI units) without performance problems.

Emotions (logic) of groups of soldiers modeled in the game.

AI engine used on TV ("Decisive Battles" series by History Channel).

The Sims by Maxis - 2000

An everyday life-simulation of a family of virtual characters in an house.

Innovations in game AI:

Smart objects help to implement behaviors (decentralized logic).

Each Sim has basic desires which drive their choice of actions.

Emotional interactions (relationships) between characters.

MOST INFLUENTIAL AI VIDEO GAMES 4

Halo by Bungie - 2001

A first-person shooter where the player battles aliens on foot or in vehicles.

Innovations in game AI:

AI enemies use cover wisely and employ suppressive fire and grenades.

Squad current status affects individuals (flee on leader death).

Its "behavior tree" technology has become very popular in games (Halo 2).

Façade by Procedural Arts - 2005

An interactive story set in an apartment during a relationship breakdown.

Innovations in game AI:

Player interacts by typing text into a natural language parser.

Its behavior language allows to specify the behavior of characters.

MOST INFLUENTIAL AI VIDEO GAMES 5

F.E.A.R. by Monolith Productions - 2005

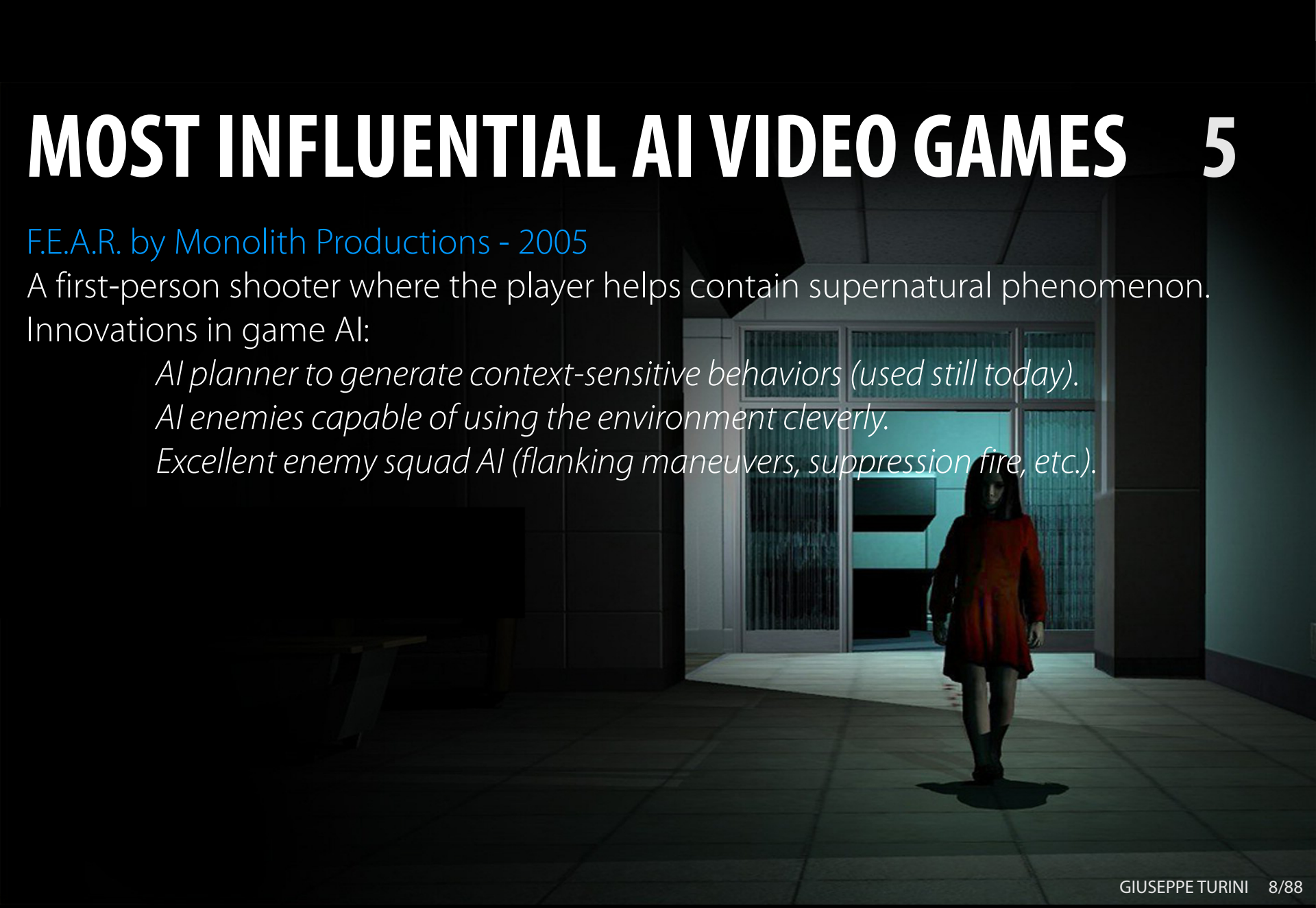
A first-person shooter where the player helps contain supernatural phenomenon.

Innovations in game AI:

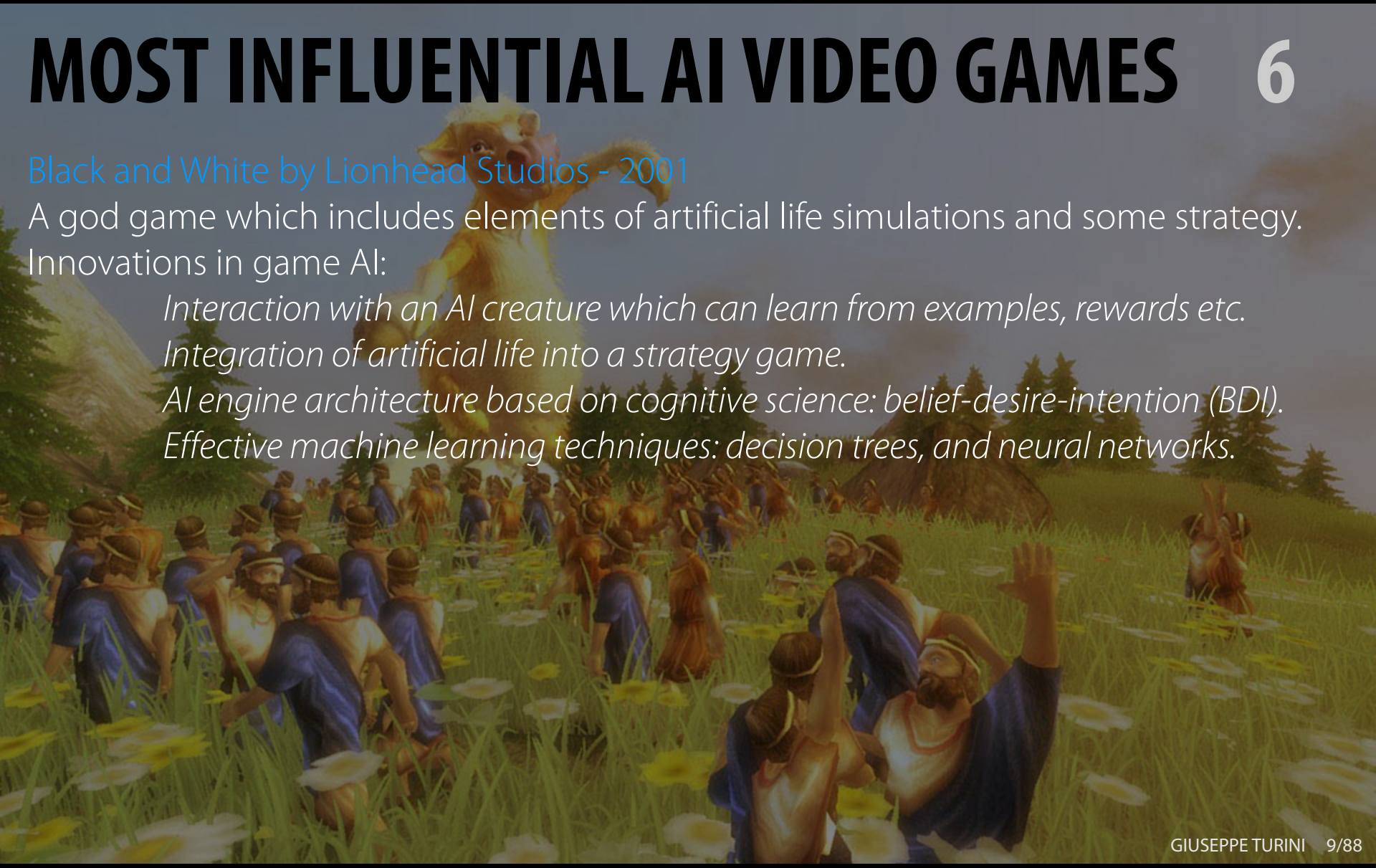
AI planner to generate context-sensitive behaviors (used still today).

AI enemies capable of using the environment cleverly.

Excellent enemy squad AI (flanking maneuvers, suppression fire, etc.).



MOST INFLUENTIAL AI VIDEO GAMES 6



Black and White by Lionhead Studios - 2001

A god game which includes elements of artificial life simulations and some strategy.

Innovations in game AI:

Interaction with an AI creature which can learn from examples, rewards etc.

Integration of artificial life into a strategy game.

AI engine architecture based on cognitive science: belief-desire-intention (BDI).

Effective machine learning techniques: decision trees, and neural networks.

MOST INFLUENTIAL AI VIDEO GAMES 7

S.T.A.L.K.E.R. by GSC Game World - 2007

A sci-fi first-person shooter survival horror game.

Innovations in game AI:

AI enemies use complex combat tactics (healing allies, giving orders etc.).

MOST INFLUENTIAL AI VIDEO GAME 7

Far Cry 2 by Ubisoft - 2008

A first-person shooter where the player fights numerous mercenaries and assassins.

Innovations in game AI:

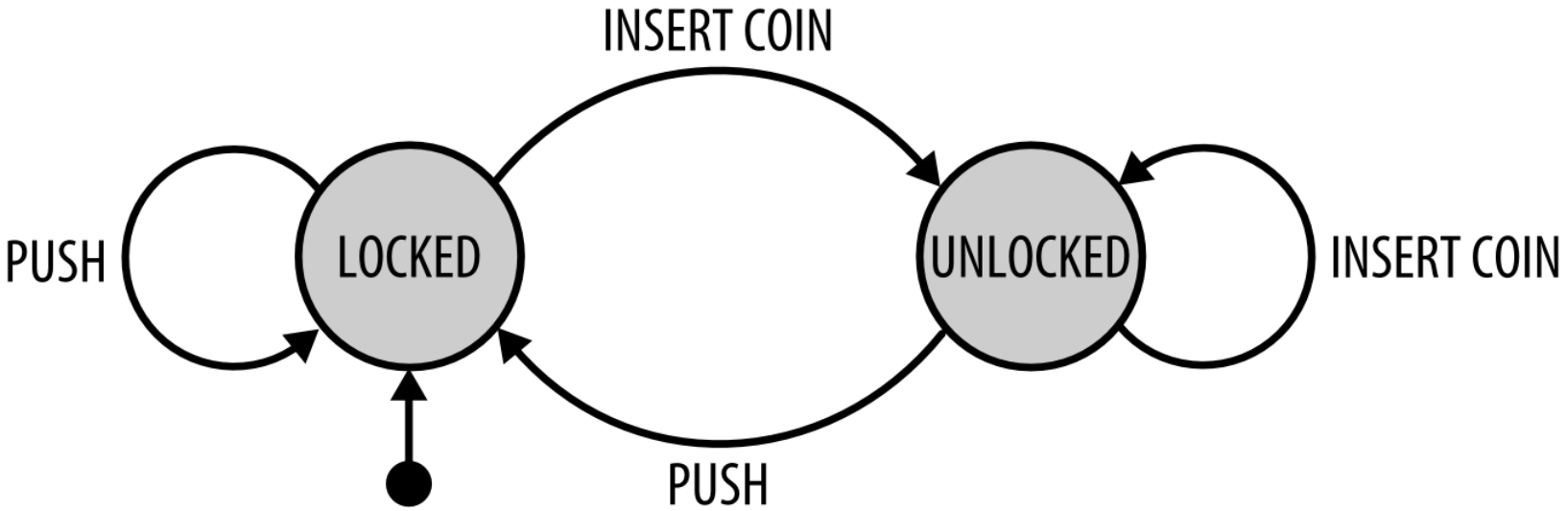
Behavior-based AI which uses action selection.

AI unpredictable in many situations.

AI enemies respond to sounds and visual distractions (fire, explosions etc.).

AI including social interfaces (not conversational, but reaction-based).

FINITE STATE MACHINES 1



TURNSTILE - STATE DIAGRAM

FINITE STATE MACHINES 2

Definition of a Finite State Machine - Part 1

A finite-state machine (FSM) is a mathematical model of computation.
It is used to design both computer programs and sequential logic circuits.

A FSM is an abstract machine that can be in one of a *finite number of states*.
A FSM is *in only one state at a time (the current state)*.
A FSM can change from one state to another when an event happens (*transition*).
A FSM is defined by a list of its states, and the condition for each transition.

Considered as an abstract model of computation, the finite state machine is weak.
For example: The FSM has less computational power than the Turing machine.
There are tasks which no FSM can do, but some Turing machines can.
The FSM has limited memory (finite number of states).

FINITE STATE MACHINES 3

Definition of a Finite State Machine - Part 2

State: Description of the status of a system that is waiting to execute a transition.

Transition: Set of actions to be executed when a condition is fulfilled.
In some FSM representations you can associate actions with a state:

- *Entry Action*: Action performed when entering the state.
- *Exit Action*: Action performed when exiting the state.

State Diagram: Diagram used to visually describe the behavior of a FSM.
This diagram is usually a *directed graph*.

FINITE STATE MACHINES 4

Simple Implementation of a Finite State Machine in Unity - Part 1

Player tank tagged as "Player" and with a Rigidbody Collider.
Player tank controlled via its PlayerTankController.cs script component.

Enemy tank with a Rigidbody Collider and 2 children: Turret and BulletSpawnPoint.
Enemy tank controlled via its SimpleFSM.cs script component (see also FSM.cs).
The FSM.cs script implements a generic finite state machine class.
The SimpleFSM.cs script derives FSM and implements a specific FSM class.

Bullet with automatic animation and destruction.

Waypoints as empty gameobject and tagged as "Waypoint".

FINITE STATE MACHINES 5

Simple Implementation of a Finite State Machine in Unity - Part 2

See the PlayerTankController.cs script.

```
public class PlayerTankController : MonoBehaviour {
    public GameObject bullet;
    private Transform turret;
    private Transform bulletSpawnPoint;
    private float currSpeed;
    private float targetSpeed;
    private float rotSpeed;
    private float turretRotSpeed = 10.0f;
    private float maxForwardSpeed = 50.0f;
    private float maxBackwardSpeed = -50.0f;
    protected float shootRate; // Tank shooting rate.
    protected float elapsedTime; // Elapsed time since the last shot.
```

FINITE STATE MACHINES 6

Simple Implementation of a Finite State Machine in Unity - Part 3

```
void Start() {
    rotSpeed = 150.0f;
    turret = gameObject.transform.GetChild(0).transform;
    bulletSpawnPoint = turret.GetChild(0).transform; }

void OnEndGame() { this.enabled = false; }

void Update() { UpdateControl(); UpdateWeapon(); }

// Implement aiming with mouse and the move/rotate with keyboard.
void UpdateControl() {
    // Create a plane intersecting the tank position with upward normal.
    Plane playerPlane = new Plane( Vector3.up, transform.position );
    // Generate a ray from the cursor position.
    Ray r = Camera.main.ScreenPointToRay( Input.mousePosition );
    ...
```

FINITE STATE MACHINES 7

Simple Implementation of a Finite State Machine in Unity - Part 3

```
// Determine the point where the cursor ray intersects the plane.
float hitDist = 0.0f;
// Check if the ray hits the plane.
if( playerPlane.Raycast( r, out hitDist ) ) {
    // Get a point along the ray using the hit distance.
    Vector3 rayHitPoint = r.GetPoint( hitDist );
    Quaternion targetRotation = Quaternion.LookRotation(
                                    rayHitPoint - transform.position );
    turret.rotation = Quaternion.Slerp( turret.transform.rotation,
                targetRotation, Time.deltaTime * turretRotSpeed ); }
// Check keyboard inputs for movement.
if( Input.GetKey( KeyCode.W ) ) { targetSpeed = maxForwardSpeed; }
else if( Input.GetKey( KeyCode.S ) ) {
    targetSpeed = maxBackwardSpeed; }
else { targetSpeed = 0.0f; }
...
```

FINITE STATE MACHINES 8

Simple Implementation of a Finite State Machine in Unity - Part 4

```
        // Check keyboard inputs for rotation.
        if( Input.GetKey( KeyCode.A ) ) {
            transform.Rotate( 0.0f, -rotSpeed * Time.deltaTime, 0.0f ); }
        else if( Input.GetKey( KeyCode.D ) ) {
            transform.Rotate( 0.0f, rotSpeed * Time.deltaTime, 0.0f ); }
        // Determine current speed.
        currSpeed = Mathf.Lerp(currSpeed,targetSpeed,7.0f*Time.deltaTime);
        // Move forward.
        transform.Translate(Vector3.forward*Time.deltaTime*currSpeed); }

    // Implement the firing with the mouse.
    void UpdateWeapon() {
        if((Input.GetMouseButtonDown(0))&&(elapsedTime>=shootRate)) {
            elapsedTime = 0.0f; // Reset elapsed time since our last shot.
            Instantiate(    bullet, bulletSpawnPoint.position,
                            bulletSpawnPoint.rotation ); } }
}
```

FINITE STATE MACHINES 9

Simple Implementation of a Finite State Machine in Unity - Part 5

See the FSM.cs script.

```
public class FSM : MonoBehaviour {
    public Transform turret { get; set; } // Tank turret transform.
    public Transform bulletSpawnPoint { get; set; } // Bullet spawn point.
    protected Transform playerTransform; // Player transform component.
    protected Vector3 destPos; // Next destination position of the NPC tank.
    protected GameObject[] pointList; // List of points for patrolling.
    protected float shootRate; // Tank shooting rate.
    protected float elapsedTime; // Elapsed time since the last shot.
    protected virtual void FSMInit() {}
    protected virtual void FSMUpdate() {}
    protected virtual void FSMFixedUpdate() {}
    void Start() { FSMInit(); }
    void Update() { FSMUpdate(); }
    void FixedUpdate() { FSMFixedUpdate(); }
}
```

FINITE STATE MACHINES 10

Simple Implementation of a Finite State Machine in Unity - Part 6

See the SimpleFSM.cs script.

```
public class SimpleFSM : FSM {
    public enum FSMState { None, Patrol, Chase, Attack, Dead };
    public FSMState currState; // FSM current state.
    public GameObject bullet; // Bullet prefab.
    private float currSpeed; // Tank current speed.
    private float currRotSpeed; // Tank current rotation speed.
    private int health; // Tank life points.
    ...
```

FINITE STATE MACHINES 11

Simple Implementation of a Finite State Machine in Unity - Part 7

```
// Init the FSM for the NPC tank.
protected override void FSMInit() {
    currState = FSMState.Patrol; currSpeed = 150; currRotSpeed = 2;
    elapsedTime = 0; shootRate = 3; health = 100;
    // Get the list of waypoints.
    pointList = GameObject.FindGameObjectsWithTag( "Waypoint" );
    FindNextWaypoint(); // Set a random waypoint as next destination.
    // Get the player transform component.
    playerTransform = FindGameObjectWithTag( "Player" ).transform;
    // Safeguard.
    if( !playerTransform ) { Debug.Log( "Player doesn't exist!" ); }
    // Get the tank turret child transform component.
    turret = gameObject.transform.GetChild(0).transform;
    // Get the tank bullet spawn point child transform component.
    bulletSpawnPoint = turret.GetChild(0).transform; }
```

FINITE STATE MACHINES 12

Simple Implementation of a Finite State Machine in Unity - Part 8

```
// Update the FSM for the NPC tank.
protected override void FSMUpdate() {
    switch(currState) {
        case FSMState.Patrol: { UpdatePatrolState(); break; }
        case FSMState.Chase: { UpdateChaseState(); break; }
        case FSMState.Attack: { UpdateAttackState(); break; }
        case FSMState.Dead: { UpdateDeadState(); break; } }
    // Update elapsed time since last shot.
    elapsedTime += Time.deltaTime; }
```

FINITE STATE MACHINES 13

Simple Implementation of a Finite State Machine in Unity - Part 9

```
// Implementation of the Patrol state.
protected void UpdatePatrolState() {
    // Find another random waypoint if current waypoint is reached.
    if( Vector3.Distance( transform.position, destPos ) <= 100.0f ) {
        FindNextWaypoint(); }
    // Check player-tank distance, if too low switch to Chase state.
    else if( Vector3.Distance(  transform.position,
                                playerTransform.position ) <= 300.0f ) {
        currState = FSMState.Chase; }
    // Rotate toward the target point.
    Quaternion targetRotation = Quaternion.LookRotation(
                                        destPos - transform.position );
    transform.rotation = Quaternion.Slerp(  transform.rotation,
            targetRotation, Time.deltaTime * currRotSpeed );
    transform.Translate(Vector3.forward*Time.deltaTime*currSpeed); }
```

FINITE STATE MACHINES 14

Simple Implementation of a Finite State Machine in Unity - Part 10

```
// Implementation of the Chase state.
protected void UpdateChaseState() {
    // Set target position as player position.
    destPos = playerTransform.position;
    // Compute the distance tank-player.
    float dist = Vector3.Distance( transform.position,
                                   playerTransform.position );
    // Switch state accordingly to the distance tank-player.
    if( dist <= 200.0f ) { currState = FSMState.Attack; }
    else if( dist >= 300.0f ) { currState = FSMState.Patrol; }
    transform.Translate(Vector3.forward*Time.deltaTime*currSpeed); }
```

FINITE STATE MACHINES 15

Simple Implementation of a Finite State Machine in Unity - Part 11

```
// Implementation of the Attack state.
protected void UpdateAttackState() {
    destPos = playerTransform.position; // Set target as player pos.
    // Compute the distance tank-player.
    float dist = Vector3.Distance( transform.position,
                                   playerTransform.position );
    // Switch state accordingly to the distance tank-player.
    if( ( dist >= 200.0f ) && ( dist < 300.0f ) ) {
        // Rotate toward the target point.
        Quaternion targetRotation = Quaternion.LookRotation(
                                        destPos - transform.position );
        transform.rotation = Quaternion.Slerp( transform.rotation,
                        targetRotation, Time.deltaTime * currRotSpeed );
        transform.Translate(Vector3.forward*Time.deltaTime*currSpeed);
        currState = FSMState.Chase; } // Switch state.
    ...
```

FINITE STATE MACHINES 16

Simple Implementation of a Finite State Machine in Unity - Part 12

```
        else if( dist >= 300.0f ) { currState = FSMState.Patrol; }
        // Turn the turret towards the player.
        Quaternion turretRotation = Quaternion.LookRotation(
                                                destPos - turret.position );
        turret.rotation = Quaternion.Slerp( turret.rotation, turretRotation,
                                            Time.deltaTime * currRotSpeed );
        FireBullet(); }

    // Implementation of the Dead state.
    protected void UpdateDeadState() {
        // Show the dead animation with some physics effects.
        Explode(); }
```

FINITE STATE MACHINES 17

Simple Implementation of a Finite State Machine in Unity - Part 13

```
// Fire a bullet from the spawn point.
private void FireBullet() {
    if( elapsedTime >= shootRate ) {
        // Create a clone of the bullet prefab on-the-fly.
        Instantiate(    bullet, bulletSpawnPoint.position,
                        bulletSpawnPoint.rotation );
        elapsedTime = 0.0f; } }

// Manage the collision with a bullet.
void OnCollisionEnter( Collision c ) {
    // Check if the colliding object is a bullet.
    if( c.gameObject.tag == "Bullet" ) {
        // Update tank life points accordingly to bullet power.
        health -= c.gameObject.GetComponent<Bullet>().damage;
        // Switch to Dead state if the tank has no life points left.
        if( health <= 0 ) { currState = FSMState.Dead; } } }
```

FINITE STATE MACHINES 18

Simple Implementation of a Finite State Machine in Unity - Part 14

```
// Find the next semi-random waypoint.
protected void FindNextWaypoint() {
    int rndIndex = Random.Range( 0, pointList.Length ); // Random index.
    float rndRadius = 10.0f; // Radius for waypoint proximity check.
    destPos = pointList[ rndIndex ].transform.position; // Target point.
    // Range check to prevent to select a waypoint too close.
    if( IsInCurrentRange( destPos ) ) {
        Vector3 rndPosition = new Vector3(
                                    Random.Range(-rndRadius,rndRadius),
                                    0.0f,
                                    Random.Range(-rndRadius,rndRadius) );
        destPos = pointList[rndIndex].transform.position+rndPosition; } }
```

FINITE STATE MACHINES 19

Simple Implementation of a Finite State Machine in Unity - Part 15

```
    // Check if input position is too close to the tank position.
    protected bool IsInCurrentRange( Vector3 pos ) {
        float xPos = Mathf.Abs( pos.x - transform.position.x );
        float zPos = Mathf.Abs( pos.z - transform.position.z );
        if( ( xPos <= 50 ) && ( zPos <= 50 ) ) { return true; }
        return false; }

    // Implement the animation for the tank explosion.
    protected void Explode() {
        float rndX = Random.Range(10,30); float rndZ = Random.Range(10,30);
        for( int i = 0; i < 3; i++ ) {
            rigidbody.AddExplosionForce( 10000.0f, transform.position -
                            new Vector3( rndX, 10.0f, rndZ ), 40.0f, 10.0f );
            rigidbody.velocity = transform.TransformDirection(
                                        new Vector3( rndX, 20.0f, rndZ ) ); }
        Destroy( gameObject, 1.5f ); }
}
```

FINITE STATE MACHINES 20

Advanced Implementation of a Finite State Machine in Unity - Part 1

In this advanced FSM design pattern it is the responsibility of each state to provide the FSM with an output state in reply to an input transition.

Tank AI in the NPCTankController.cs script, derived by the AdvancedFSM class.
The AdvancedFSM class implements a general FSM with generic states and transitions.
This class manages only the list of states and the execution of a transition.

The FSMState.cs script implement a generic FSM state.
It has a map to store output states associated with specific input transitions.
Each derived state should implement a Reason and an Act methods.

The Reason method performs the checks required to trigger transitions.
The Act method is the implementation of the state behavior.

FINITE STATE MACHINES 21

Advanced Implementation of a Finite State Machine in Unity - Part 2
See the NPCTankController.cs script.

```
public class NPCTankController : AdvancedFSM {

    public GameObject bullet;
    private int health;

    // Initialize the finite state machine for the NPC tank.
    protected override void FSMInit() {
        health = 100; elapsedTime = 0.0f; shootRate = 2.0f;
        GameObject objPlayer = GameObject.FindGameObjectWithTag( "Player" );
        playerTransform = objPlayer.transform;
        if( !playerTransform ) { Debug.Log( "Player doesn't exist!" ); }
        turret = gameObject.transform.GetChild(0).transform;
        bulletSpawnPoint = turret.GetChild(0).transform;
        // Init the finite state machine for the NPC tank;
        ConstructFSM(); }
```

FINITE STATE MACHINES 22

Advanced Implementation of a Finite State Machine in Unity - Part 3

```
protected override void FSMUpdate() {
    // Update the elapsed time since the last shot.
    elapsedTime += Time.deltaTime; }

protected override void FSMFixedUpdate() {
    CurrentState.Reason( playerTransform, transform );
    CurrentState.Act( playerTransform, transform ); }

public void SetTransition( Transition t ) { PerformTransition(t); }
```

FINITE STATE MACHINES 23

Advanced Implementation of a Finite State Machine in Unity - Part 4

```
// Init the FSM for the NPC tank.
private void ConstructFSM() {
    // Init waypoints.
    pointList = GameObject.FindGameObjectsWithTag( "Waypoint" );
    Transform[] waypoints = new Transform[ pointList.Length ];
    int i = 0;
    foreach( GameObject obj in pointList ) {
        waypoints[i] = obj.transform;
        i++; }
    // Init Patrol state.
    PatrolState patrol = new PatrolState(waypoints);
    patrol.AddTransition( Transition.SawPlayer, FSMStateID.Chasing );
    patrol.AddTransition( Transition.NoHealth, FSMStateID.Dead );
    ...
```

FINITE STATE MACHINES 24

Advanced Implementation of a Finite State Machine in Unity - Part 5

```
// Init Chase state.
ChaseState chase = new ChaseState(waypoints);
chase.AddTransition( Transition.LostPlayer, FSMStateID.Patrolling );
chase.AddTransition( Transition.ReachPlayer, FSMStateID.Attacking );
chase.AddTransition( Transition.NoHealth, FSMStateID.Dead );
// Init Attack state.
AttackState attack = new AttackState(waypoints);
attack.AddTransition(Transition.LostPlayer, FSMStateID.Patrolling);
attack.AddTransition( Transition.SawPlayer, FSMStateID.Chasing );
attack.AddTransition( Transition.NoHealth, FSMStateID.Dead );
// Init Dead state.
DeadState dead = new DeadState();
// Add states to the FSM.
AddFSMState(patrol); AddFSMState(chase); AddFSMState(attack);
AddFSMState(dead); }
```

FINITE STATE MACHINES 25

Advanced Implementation of a Finite State Machine in Unity - Part 6

```
    // Manage collisions with bullets.
    void OnCollisionEnter( Collision c ) {
        if( c.gameObject.tag == "Bullet" ) { // Colliding with a bullet?
        // Update life points and check to trigger the NoHealth transition.
        health -= 50;
        if( health <= 0 ) {
            SetTransition( Transition.NoHealth );
            Explode(); } } }

    // Perform the animation for the explosion of the tank.
    protected void Explode() { ... }

    // Fire the bullet from the turret (bullet spawn point).
    public void FireBullet() { ... }

}
```

FINITE STATE MACHINES 26

Advanced Implementation of a Finite State Machine in Unity - Part 7

See the AdvancedFSM.cs script.

```
public enum Transition {None, SawPlayer, ReachPlayer, LostPlayer, NoHealth};
public enum FSMStateID {None, Patrolling, Chasing, Attacking, Dead};

public class AdvancedFSM : FSM {

    public FSMStateID CurrentStateID { get { return currentStateID; } }
    public FSMState CurrentState { get { return currentState; } }

    private List<FSMState> states; // States changed via transitions.
    private FSMStateID currStateID;
    private FSMState currState;

    // Constructor.
    public AdvancedFSM() { fsmStates = new List<FSMState>(); }
```

FINITE STATE MACHINES 27

Advanced Implementation of a Finite State Machine in Unity - Part 8

```
// Add a new state into the list.
public void AddFSMState( FSMState s ) {
    // Check for Null reference before deleting.
    if( s == null ) { Debug.LogError( "FSM ERROR: Null reference!" ); }
    // First State inserted is also the Initial state.
    if( states.Count == 0 ) {
        states.Add(s); currState = s; currStateID = s.ID; return; }
    // Add the state to the list if it is not already contained.
    foreach( FSMState state in states ) {
        if( state.ID == s.ID ) {
            Debug.LogError( "FSM ERROR: ..." ); return; } }
    // Add the state to the list.
    states.Add(s); }
```

FINITE STATE MACHINES 28

Advanced Implementation of a Finite State Machine in Unity - Part 9

```
// Delete a state from the list.
public void DeleteState( FSMStateID sID ) {
    // Check if the state is valid before deleting.
    if( sID == FSMStateID.None ) {
        Debug.LogError( "FSM ERROR: State ID not valid!" ); return; }
    // Delete the state if the list contains it.
    foreach( FSMState state in states ) {
        if( state.ID == sID ) { states.Remove(state); return; } }
    Debug.LogError( "FSM ERROR: ..." ); }
```

FINITE STATE MACHINES 29

Advanced Implementation of a Finite State Machine in Unity - Part 10

```
    // Perform a transition changing the FSM state.
    public void PerformTransition( Transition t ) {
        // Check for if the transition is valid.
        if( t == Transition.None ) {    Debug.LogError( "FSM ERROR: ..." );
                                        return; }
        // Check if the current state can process the input transition.
        FSMStateID id = currState.GetOutputState(t);
        if( id == FSMStateID.None ) {
            Debug.LogError( "FSM ERROR: ..." ); return; }
        // Update the current state ID and the current state.
        currentStateID = id;
        foreach( FSMState state in states ) {
            if( state.ID == currStateID ) { currState = state; break; } } }

}
```

FINITE STATE MACHINES 30

Advanced Implementation of a Finite State Machine in Unity - Part 11

See the FSMState.cs script.

```
// A state for the AdvancedFSM finite state machine class.
// Each state has a dictionary storing pairs (transition-state).
// Each pair shows the next state if a transition is fired in this state.
public abstract class FSMState {

    public FSMStateID ID { get { return stateID; } }

    protected Dictionary<Transition, FSMStateID> map =
                                    new Dictionary<Transition, FSMStateID>();
    protected FSMStateID stateID;
    protected Vector3 destPos;
    protected Transform[] waypoints;
    protected float currRotSpeed;
    protected float currSpeed;
```

FINITE STATE MACHINES 31

Advanced Implementation of a Finite State Machine in Unity - Part 12

```
public void AddTransition( Transition t, FSMStateID sID ) {
    // Check if any of the input arguments is invalid.
    if( ( t == Transition.None ) || ( sID == FSMStateID.None ) ) {
        Debug.LogWarning( "FSMState ERROR: ..." ); return; }
    // Check if input transition is already inside the dictionary.
    if( map.ContainsKey(t) ) {
        Debug.LogWarning( "FSMState ERROR: ..." ); return; }
    map.Add( t, sID ); }

public void DeleteTransition( Transition t ) {
    // Check if the input transition is valid.
    if( t == Transition.None ) {
        Debug.LogError( "FSMState ERROR: ..." ); return; }
    // Check if the pair is in the dictionary before deleting.
    if( map.ContainsKey(t) ) { map.Remove(t); return; }
    Debug.LogError( "FSMState ERROR: ..." ); }
```

FINITE STATE MACHINES 32

Advanced Implementation of a Finite State Machine in Unity - Part 13

```
// Return the next state if the FSM receives a transition in this state.
public FSMStateID GetOutputState( Transition t ) {
    // Check if the input transition is valid.
    if( t == Transition.None ) {
        Debug.LogError("FSMState ERROR: ..."); return FSMStateID.None; }
    // Check if transition is in the dictionary.
    if( map.ContainsKey(t) ) { return map[t]; }
    Debug.LogError( "FSMState ERROR: ..." ); return FSMStateID.None; }

// Determine the transition to fire.
public abstract void Reason( Transform player, Transform npc );

// State implementation to control the associated behavior.
public abstract void Act( Transform player, Transform npc );
```

FINITE STATE MACHINES 33

Advanced Implementation of a Finite State Machine in Unity - Part 14

```
    // Find the next semi-random waypoint.
    public void FindNextWaypoint() {
        int rndIndex = Random.Range( 0, waypoints.Length );
        Vector3 rndPosition = Vector3.zero;
        destPos = waypoints[ rndIndex ].position + rndPosition; }

    // Check if next random waypoint is too close to the current position.
    protected bool IsInCurrentRange( Transform trans, Vector3 pos ) {
        float xPos = Mathf.Abs( pos.x - trans.position.x );
        float zPos = Mathf.Abs( pos.z - trans.position.z );
        if( ( xPos <= 50 ) && ( zPos <= 50 ) ) { return true; }
        return false; }

}
```

FINITE STATE MACHINES 34

Advanced Implementation of a Finite State Machine in Unity - Part 15

See the PatrolState.cs script.

```
public class PatrolState : FSMState {

    // Constructor.
    public PatrolState( Transform[] wps ) {
        waypoints = wps; stateID = FSMStateID.Patrolling;
        currRotSpeed = 1.0f; currSpeed = 100.0f; }

    public override void Reason( Transform player, Transform npc ) {
        // Check distance player-tank.
        if( Vector3.Distance( npc.position, player.position) <= 300.0f ) {
            // Trigger the SawPlayer transition.
            npc.GetComponent<NPCTankController>().SetTransition(
                                                Transition.SawPlayer ); } }
```

FINITE STATE MACHINES 35

Advanced Implementation of a Finite State Machine in Unity - Part 16

```
    public override void Act( Transform player, Transform npc ) {
        // Find a random waypoint if we reach the current waypoint.
        if( Vector3.Distance( npc.position, destPos ) <= 100.0f ) {
            FindNextWaypoint(); }
        // Rotate toward the target.
        Quaternion targetRotation = Quaternion.LookRotation(
                                                    destPos - npc.position );
        npc.rotation = Quaternion.Slerp(    npc.rotation, targetRotation,
                                            Time.deltaTime * currRotSpeed );
        // Move forward.
        npc.Translate( Vector3.forward * Time.deltaTime * currSpeed ); }

}
```

FINITE STATE MACHINES 36

Advanced Implementation of a Finite State Machine in Unity - Part 17

See the ChaseState.cs script.

```
public class ChaseState : FSMState {

    public ChaseState( Transform[] wps ) {
        waypoints = wps; stateID = FSMStateID.Chasing; currRotSpeed = 1.0f;
        currSpeed = 100.0f; FindNextWaypoint(); }

    public override void Reason( Transform player, Transform npc ) {
        destPos = player.position; // Set target pos as player pos.
        float dist = Vector3.Distance( npc.position, destPos );
        if( dist <= 200.0f ) { // Trigger the ReachPlayer transition.
            npc.GetComponent<NPCTankController>().SetTransition(
                                                Transition.ReachPlayer ); }
        else if( dist >= 300.0f ) { // Trigger the LostPlayer transition.
            npc.GetComponent<NPCTankController>().SetTransition(
                                                Transition.LostPlayer ); } }
```

FINITE STATE MACHINES 37

Advanced Implementation of a Finite State Machine in Unity - Part 18

```
    public override void Act( Transform player, Transform npc ) {
        // Rotate to the target point
        destPos = player.position;
        // ...
        Quaternion targetRotation = Quaternion.LookRotation(
                                                    destPos - npc.position );
        npc.rotation = Quaternion.Slerp(    npc.rotation, targetRotation,
                                            Time.deltaTime * currRotSpeed );
        // Move forward.
        npc.Translate( Vector3.forward * Time.deltaTime * currSpeed ); }

}
```

FINITE STATE MACHINES 38

Advanced Implementation of a Finite State Machine in Unity - Part 19

See the AttackState.cs script.

```
public class AttackState : FSMState {

    public AttackState( Transform[] wps ) {
        waypoints = wps; stateID = FSMStateID.Attacking;
        currRotSpeed = 1.0f; currSpeed = 100.0f; FindNextWaypoint(); }
```

FINITE STATE MACHINES 39

Advanced Implementation of a Finite State Machine in Unity - Part 20

```
public override void Reason( Transform player, Transform npc ) {
    float dist = Vector3.Distance( npc.position, player.position );
    if( ( dist >= 200.0f ) && ( dist < 300.0f ) ) {
        // Rotate and move toward the target point.
        Quaternion targetRotation = Quaternion.LookRotation(
                                                destPos - npc.position );
        npc.rotation = Quaternion.Slerp( npc.rotation, targetRotation,
                                        Time.deltaTime * currRotSpeed );
        npc.Translate( Vector3.forward * Time.deltaTime * currSpeed );
        // Trigger the SawPlayer transition.
        npc.GetComponent<NPCTankController>().SetTransition(
                                                Transition.SawPlayer ); }
    else if( dist >= 300.0f ) {
        // Trigger the LostPlayer transition.
        npc.GetComponent<NPCTankController>().SetTransition(
                                            Transition.LostPlayer ); } }
```

FINITE STATE MACHINES 40

Advanced Implementation of a Finite State Machine in Unity - Part 21

```
    public override void Act( Transform player, Transform npc ) {
        // Set the target position as the player position.
        destPos = player.position;
        // Turn the turret towards the player.
        Transform turret = npc.GetComponent<NPCTankController>().turret;
        Quaternion turretRotation = Quaternion.LookRotation(
                                                destPos - turret.position );
        turret.rotation = Quaternion.Slerp( turret.rotation, turretRotation,
                                            Time.deltaTime * currRotSpeed );
        // Fire the bullet.
        npc.GetComponent<NPCTankController>().FireBullet(); }

}
```

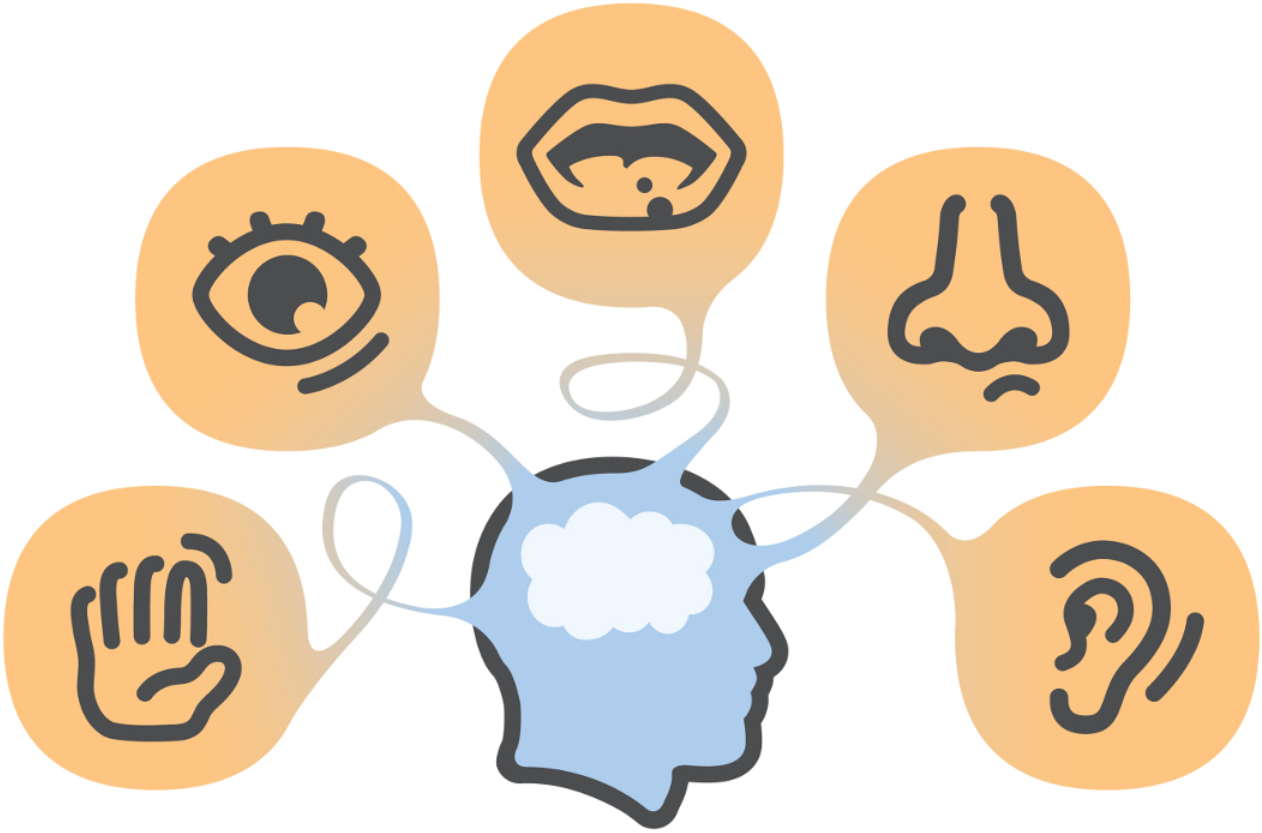
FINITE STATE MACHINES 41

Advanced Implementation of a Finite State Machine in Unity - Part 22

See the DeadState.cs script.

```
public class DeadState : FSMState {

    public DeadState() { stateID = FSMStateID.Dead; }

    public override void Reason( Transform player, Transform npc ) {}

    public override void Act( Transform player, Transform npc ) {}

}
```

GAME AI SENSORY SYSTEMS 1



GAME AI SENSORY SYSTEMS 2

AI Sensory Systems

An AI sensory system aims to emulate senses and usually has 2 components:

- Aspect: The component that triggers a sense: appearance, sound etc.
- Sense: The component implementing a sense: perception, smell, touch etc.

Simple Implementation of a Sensory System in Unity - Part 1

Player tank tagged as "Player" and with a Kinematic Rigidbody Trigger Collider.
Player tank controlled via the PlayerTank.cs script and a NavMeshAgent component.
Player tank implementing both Perception and Touch Aspects.

Enemy tank tagged as "Enemy" and with a Kinematic Rigidbody Trigger Collider.
Enemy tank controlled via the Wander.cs script and a NavMeshAgent component.
Enemy tank implementing both Perception and Touch Senses.

GAME AI SENSORY SYSTEMS 3

Simple Implementation of a Sensory System in Unity - Part 2

See the Aspect.cs script.

```
public class Aspect : MonoBehaviour {

    public enum PerceptionAspect { None, Player, Enemy };
    public enum SmellAspect { None, Smoke, Gas };
    public enum TouchAspect { None, Hard, Soft };

    public PerceptionAspect perception = PerceptionAspect.None;
    public SmellAspect smell = SmellAspect.None;
    public TouchAspect touch = TouchAspect.None;

}
```

GAME AI SENSORY SYSTEMS 4

Simple Implementation of a Sensory System in Unity - Part 3

See the Sense.cs script.

```
public class Sense : MonoBehaviour {

    public bool debug = true;
    public float detectionRate = 1.0f;

    protected float elapsedTime = 0.0f;

    protected virtual void InitSense() {}
    protected virtual void UpdateSense() {}

    void Start() { elapsedTime = 0.0f; InitSense(); }

    void Update() { UpdateSense(); }

}
```

GAME AI SENSORY SYSTEMS 5

Simple Implementation of a Sensory System in Unity - Part 4

See the TouchSense.cs script.

```
public class TouchSense : Sense {

    void OnTriggerEnter( Collider c ) {
        Aspect aspect = c.GetComponent<Aspect>();
        // Check the aspect.
        if( ( aspect != null ) &&
            ( aspect.touch != Aspect.TouchAspect.None ) ) {
            Debug.Log( "..." ); } }

}
```

GAME AI SENSORY SYSTEMS 6

Simple Implementation of a Sensory System in Unity - Part 5

See the PerspSense.cs script.

```
public class PerspSense : Sense {

    public Aspect.PerceptionAspect percAsp = Aspect.PerceptionAspect.None;
    public int fov = 45;
    public int dist = 100;

    private Transform player;
    private Vector3 rayDir;

    protected override void InitSense() {
        player = GameObject.FindGameObjectWithTag("Player").transform; }

    protected override void UpdateSense() {
        elapsedTime += Time.deltaTime;
        if( elapsedTime >= detectionRate ) { DetectAspect(); } }
```

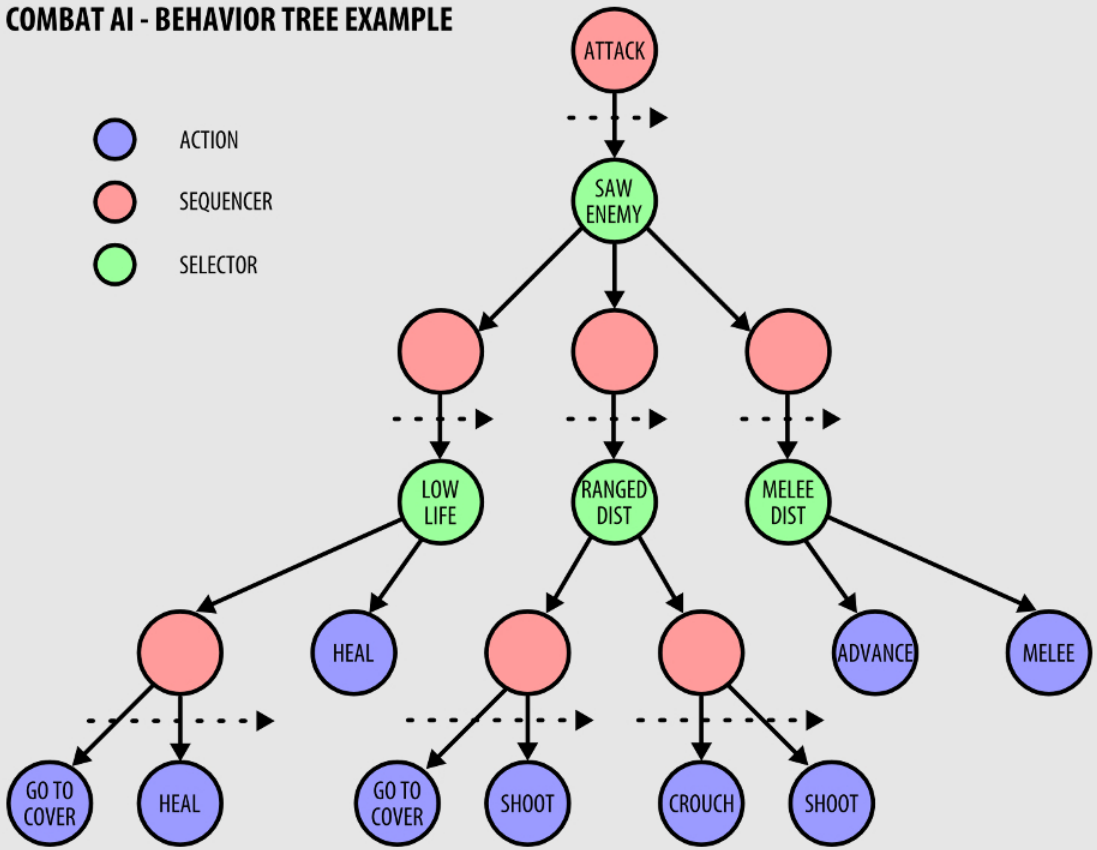
GAME AI SENSORY SYSTEMS 7

Simple Implementation of a Sensory System in Unity - Part 6

See the PerspSense.cs script.

```
// Detect perspective field of view for the AI Character.
void DetectAspect() {
    RaycastHit hit;
    rayDir = player.position - transform.position;
    if( ( Vector3.Angle( rayDir, transform.forward ) ) < fov ) {
        // Detect if player is within the field of view.
        if(Physics.Raycast(transform.position, rayDir, out hit, dist)) {
            Aspect aspect = hit.collider.GetComponent<Aspect>();
            // Check the aspect.
            if( (aspect != null) && (aspect.perception == percAsp) ) {
                Debug.Log( "..." ); } } } }
```

BEHAVIOR TREES 1



BEHAVIOR TREES 2

Until 2004 decision making problems in games were based on FSM, and FSM are:

easy to implement, intuitive, and with good performance; but
hard to maintain (not easy to add/remove a state), and not so expressive.

"Halo 2" (2004) was the first game using Behavior Trees (BTs) for Game AI.
Today BTs are the most used AI technique in games (Halo, Bioshock, Crysis, Spore etc.).
BTs are more intuitive and less error-prone than FSM, so they are popular in Game AI.

Behavior Trees - Definition

A Behavior Tree is a mathematical model of plan execution.
BTs describe switchings between a finite set of tasks in a modular fashion.
They are able to describe very complex tasks composed of simple tasks.
BTs are similar to FSM, but their behavior element is a task instead of a state.

BEHAVIOR TREES 3

Behavior Trees - Structure and Elements

A BT is a graph represented as a directed tree, with nodes classified as:
Root, Control Flow Nodes, or Execution Nodes (leaves, aka Tasks).

Each pair of connected nodes has: a *parent* (out-node), and a *child* (in-node).
The Root has no parents and exactly one child.
The Control Flow Nodes have one parent and at least one child.
The Execution Nodes have one parent and no children.

BT execution starts from the Root sending ticks (using frequency) to its child.

A tick is an enabling signal that allows the execution of a child.

When the execution of a node is allowed, it returns to the parent a status:
running if its execution has not finished yet (multiple ticks),
success if it has achieved its goal, or
failure otherwise.

BEHAVIOR TREES 4

Behavior Trees - Control Flow Nodes

A Control Flow Node is used to run each of its subtasks in turn.
When a subtask is done, the node decides to execute the next subtask or not.
Two types of Control Flow Nodes exist:

Selector Nodes (aka Fallback Nodes)

Used to find and execute the first child that does not fail.
Returns success/running as soon as a child returns success/running.
Their children are ticked in order of importance (from left to right).
Random Selector Nodes also exist.

Sequence Nodes (aka Sequencers)

Used to find and execute the first child that has not yet succeeded.
Returns failure/running as soon as a child returns failure/running.
Their children are ticked in order of importance (from left to right).
Random Sequence Nodes also exist.

BEHAVIOR TREES 5

Behavior Trees - Data Driven vs Code Driven

Data Driven: BT defined externally e.g. in a XML file.

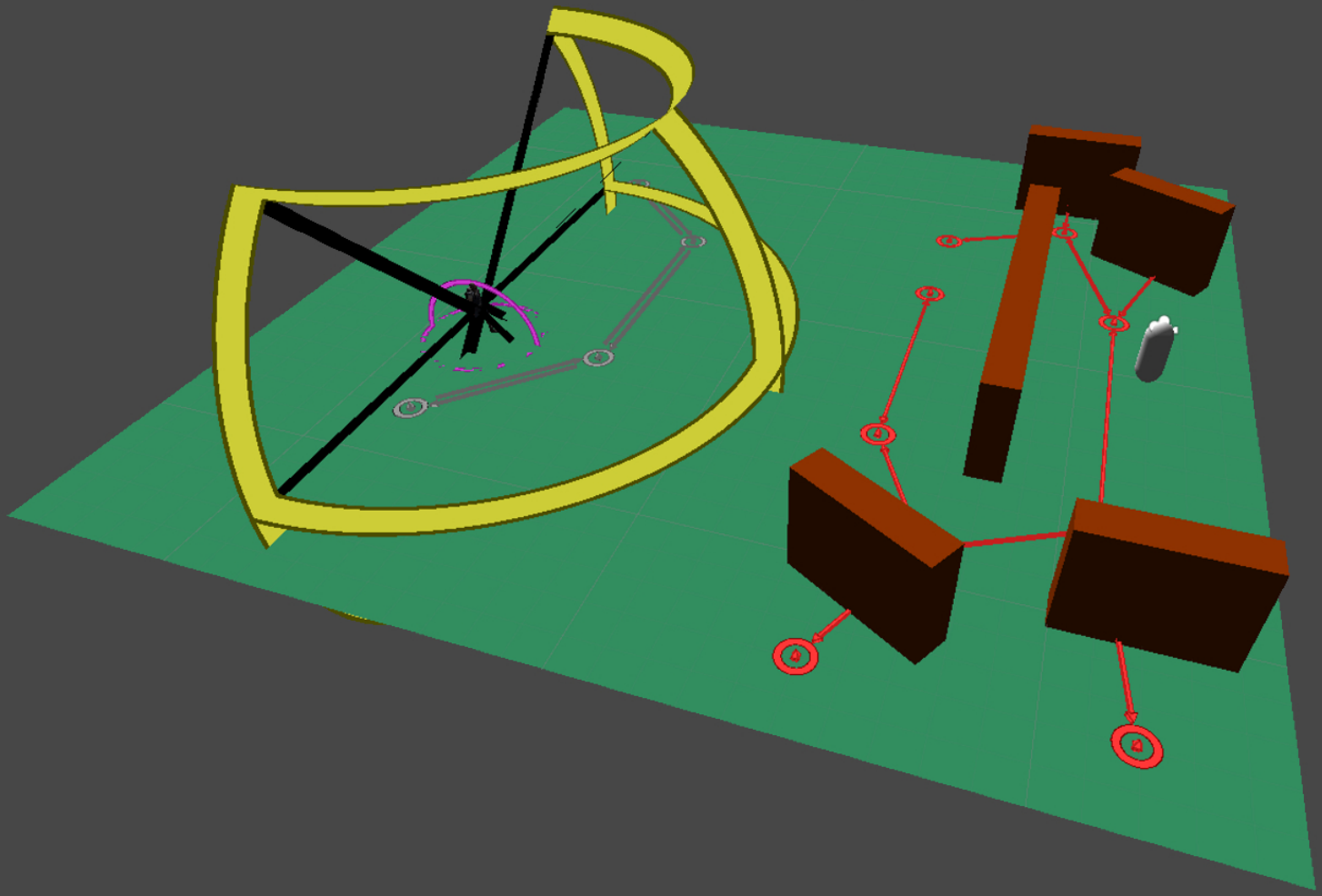
Code Driven: BT structure defined directly via nested class instances.

Behavior Trees - Traversal

We need to traversing the tree from the root to the active nodes every tick.

Optimize this task storing currently processing nodes to tick them directly.

USING A GAME AI ENGINE IN UNITY 1



USING A GAME AI ENGINE IN UNITY 2

Configuration of the RAIN AI Engine in Unity

Download the *"RAIN Starter Project 1"* from: rivaltheory.com/community/tutorials
Once the file (RAINStarter.unitypackage) is complete, create a new Unity project.
Import the RAIN package with: Assets + Import Package + Custom Package...

Implementing a Patrolling Behavior Using the RAIN AI Engine - Part 1

We start creating an AI-controlled unit using a 3D model and a special RAIN component.
Use "Max" as your AI controlled unit or another 3D model.

We are going to use 4 animations in our BT (idle, walk, run, and punch).

Select your unit + RAIN menu + Create AI.

Your unit now has a "AI" child gameobject with a AIRig script component.

This AIRig component is going to control the AI of your unit.

USING A GAME AI ENGINE IN UNITY 3

Implementing a Patrolling Behavior Using the RAIN AI Engine - Part 2

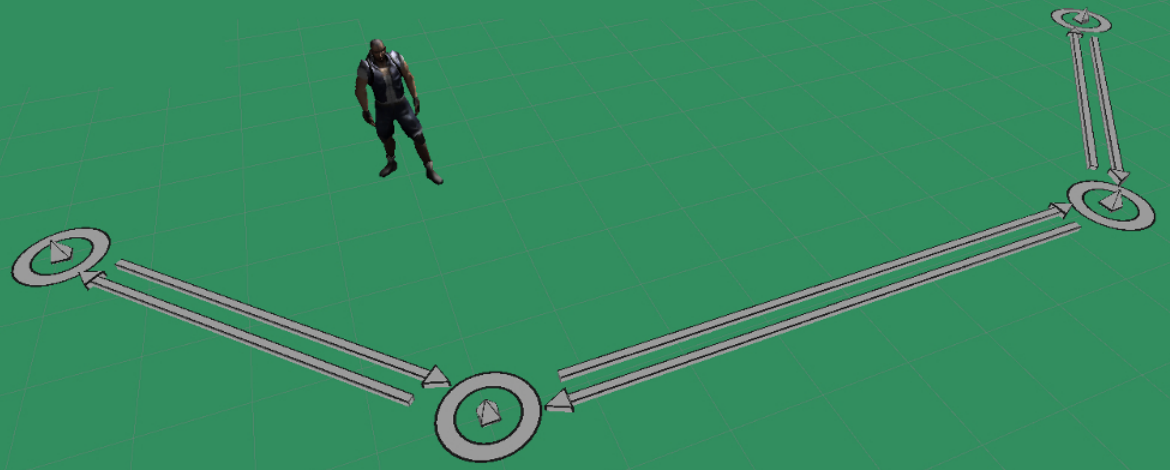
We start creating a simple patrolling behavior using a BT and waypoints.

Create a waypoint route with: RAIN menu + Create Waypoint Route.

This will add a gameobject to the scene with a WaypointRig script component.

Rename this gameobject as "PatrolWaypointRoute".

Add and configure some waypoints to describe a simple route.



USING A GAME AI ENGINE IN UNITY 4

Implementing a Patrolling Behavior Using the RAIN AI Engine - Part 3

Select the Mind tab of the AIRig script in your unit and click: Open Behavior Editor.

From the drop-down menu select: Create New Behavior Tree.

Rename the new BT as "MaxBehaviorTree".

By default the BT root is a Sequencer node.

Select the root and RMB click to open a context menu.

Do: Create + Actions + Choose Patrol Waypoints.

Rename the new node as "patrolWaypointRoute".

Set the Waypoint Route in this node to our route.

Set the Loop Type of this node to "Loop".

Set Move Target Variable name as "varMoveTarget".

This variable stores the next WP position.

Test the game while looking the Behavior Tree! See the file: MaxBehaviorTree01.xml

USING A GAME AI ENGINE IN UNITY 5

Implementing a Patrolling Behavior Using the RAIN AI Engine - Part 4

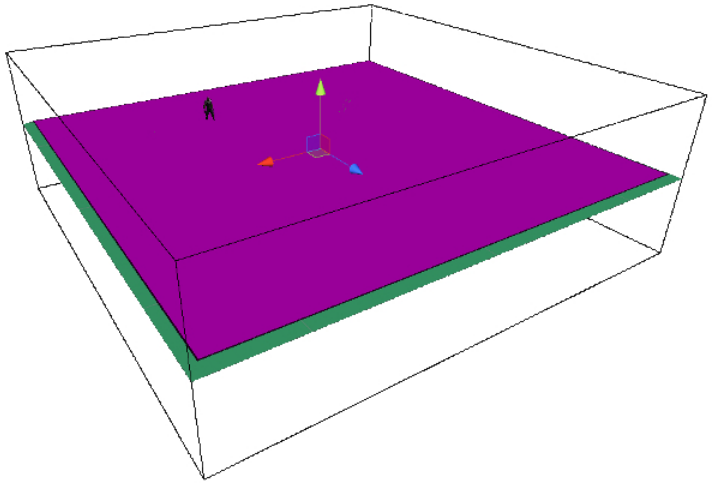
Select your unit AIRig and assign the new BT using the Mind tab.

Now create a RAIN Navigation Mesh: RAIN menu + Create Navigation Mesh.

Rename the newly created gameobject as "RAIN Navigation Mesh".

Rescale it to cover completely the ground.

In its NavMeshRig component click Generate Navigation Mesh to bake it.



USING A GAME AI ENGINE IN UNITY 6

Implementing a Patrolling Behavior Using the RAIN AI Engine - Part 5

Configure the Animation tab of our unit AIRig script component.

It should match the animation component of our unit.

Click the button Add Existing Animations.

Set Wrap Mode to Loop for the: idle, walk, run, and punch.

Edit the BT adding a Move action node as child of the "patrolWaypointRoute" node.

Name the Move node as "moveToPatrolWP".

Move node Move Target set to the var used in the parent ("varMoveTarget").

Move node Move Speed to scale the velocity of positioning (2).

Test the game while looking the Behavior Tree! See the file: MaxBehaviorTree02.xml

USING A GAME AI ENGINE IN UNITY 6

Implementing a Patrolling Behavior Using the RAIN AI Engine - Part 5

Now we need to integrate a "walk" animation into the patrolling.

Switch the root node to a Parallel node.

Add a Animate action node as a child of the root node.

Set the Animate node Animation State to "walk".

Test the game while looking the Behavior Tree! See the file: MaxBehaviorTree03.xml

USING A GAME AI ENGINE IN UNITY 7

Implementing an Attacking Behavior Using the RAIN AI Engine - Part 1

Implement the detection of the player using the RAIN sensory system (sensor-aspect).

Select the Perception tab in the AIRig script component of our unit.

- Add a Visual Sensor using proper settings accordingly to your AI unit.
  - Sensor Name set to "sensorVisualFar" and Position Offset to (0,1,0).
  - Horizontal Angle to 180 deg and Vertical Angle to 90 deg.
  - Range to 10 and Require Line of Sight set to ON.

Create the player importing the Character Controller package (Standard Asset).

- Delete the Main Camera (we are going to use the FPC camera).
- Drag-and-drop the FPC from the Standard Asset folder into the scene.

Add an Entity (EntityRig script) to the FPC: RAIN menu + Create Entity.

- An Entity in RAIN is a script component describing a set of Aspects.
- Name the Entity "entityPlayer".
- Add a Visual Aspect, name it "aspectVisualPlayer" and set its Size to 1.

USING A GAME AI ENGINE IN UNITY 8

Implementing an Attacking Behavior Using the RAIN AI Engine - Part 2

Edit the BT adding a Detect action node to the root and name it "detectVisualFar".

Move the Detect action node as first child of the root and set its properties.

Set Repeat to Forever.

Set Sensor to "sensorVisualFar" (*use quotations!*).

Set Aspect to "aspectVisualPlayer" (*use quotations!*).

Set Form Variable to "varPlayer".

This variable stores the sensing result (player position).

USING A GAME AI ENGINE IN UNITY 9

Implementing an Attacking Behavior Using the RAIN AI Engine - Part 3

Edit the BT and add a Selector node as child of the root.

Name this Selector node as "selectorPatrolOrAttack".

Add 2 Constraint nodes as children of the "selectorPatrolOrAttack" node.

First Constraint node as "constraintPlayerNotDetected": varPlayer == null

Second Constraint node as "constraintPlayerDetected": varPlayer != null

Add a Parallel node as child of the "constraintPlayerNotDetected" node.

Move the "patrol subtree" and the "animateWalk" node into this Parallel node.

Add a Parallel node as child of the "constraintPlayerDetected" node.

Add an Animate and a Move action nodes to this Parallel node.

Animation node as "animateRun" with State to "run".

Move node as "moveToPlayer" with Target to "varPlayer" and Speed 3.

Test the game while looking the Behavior Tree! See the file: MaxBehaviorTree04.xml

USING A GAME AI ENGINE IN UNITY 10

Implementing an Attacking Behavior Using the RAIN AI Engine - Part 4

We want to stop the "run" animation as soon as the unit reaches the player.

Add a Visual Sensor in the Perception tab of the AIRig component of the unit.

Name the sensor "sensorVisualNear" and set its Range to 2.

Add a Detect action node to the BT as second child of the root.

Name the new node "detectVisualNear" and set the Repeat to Forever.

Set Sensor to "sensorVisualNear" (*use quotations!*).

Set Aspect to "aspectVisualPlayer" (*use quotations!*).

Set Form Variable to "varDetectVisualNear".

USING A GAME AI ENGINE IN UNITY 11

Implementing an Attacking Behavior Using the RAIN AI Engine - Part 5

Add a Selector node "selectorAttackOrPunch" to the "constraintPlayerDetected" node.

Add 2 Constraint nodes to the "selectorAttackOrPunch" node.

First Constraint node as "constraintPlayerNotNear".

varDetectVisualNear == null

Second Constraint node as "constraintPlayerNear".

varDetectVisualNear != null

Move the "attack subtree" into the "constraintPlayerNotNear" node.

Add an Animate action node to the "constraintPlayerNear" node.

Animation node as "animatePunch" with State to "punch".

Test the game while looking the Behavior Tree! See the file: MaxBehaviorTree05.xml

USING A GAME AI ENGINE IN UNITY

Implementing an Wandering Behavior Using the RAIN AI Engine - Part 1

Add some randomness to the patrolling behavior (10 secs patrolling and 3 secs pause).

Add a Selector node "selectorPausedOrNot" as child of "constraintPlayerNotDetected".

Add 2 Constraint nodes to the "selectorPausedOrNot" node.

First Constraint node as "constraintNotPaused": !isPaused

Second Constraint node as "constraintPaused": isPaused

Add a Sequencer node as child of the "constraintNotPaused" node.

Add a Wait for Timer action node as the Sequencer 1st child.

Set the Time to 10 seconds.

Add an Evaluate Expression action node as the Sequencer 2nd child.

Expression: isPaused = true

Move "patrol subtree" (with Parallel node) as child of "constraintNotPaused".

Reset the Parallel node children: Sequencer and then Waypoints.

USING A GAME AI ENGINE IN UNITY

Implementing an Wandering Behavior Using the RAIN AI Engine - Part 2

Add a Parallel node as child of the "constraintPaused" node.

Add a Sequencer node as 1st child of the new Parallel node.

Add a Wait for Timer action node as Sequencer 1st child.

Set the Time to 3 seconds.

Add an Evaluate Expression action node as Sequencer 2nd child.

Expression: isPaused = false

Add an Animate node "animateIdle" as 2nd child of the Parallel node.

Set its Animation State to "idle".

Test the game while looking the Behavior Tree! See the file: MaxBehaviorTree06.xml

USING A GAME AI ENGINE IN UNITY 14

Implementing a Search for Player Behavior Using the RAIN AI Engine - Part 1

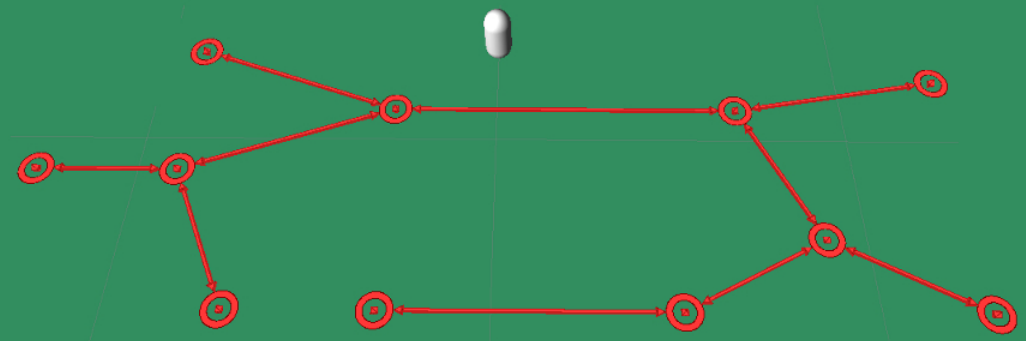
Add a "search for player" task in the attack behavior subtree ("constraintPlayerDetected").

Enlarge the floor and ensure the RAIN Navigation Mesh covers it (*regenerate it!*).

- By default the FPC is included in the baking (*so it looks like an obstacle!*).
- Create a custom layer ("Player") for the FPC and assign the layer to the FPC.
- Deselect the new "Player" layer in the RAIN Navigation Mesh settings.

Create a Waypoint Network "WanderWaypointNetwork" using the RAIN menu.

- *Waypoint Network is different than a Waypoint Route (just a sequence of WPs).*
- Add waypoints and build a simple network with few branches.



USING A GAME AI ENGINE IN UNITY 15

Implementing a Search for Player Behavior Using the RAIN AI Engine - Part 2

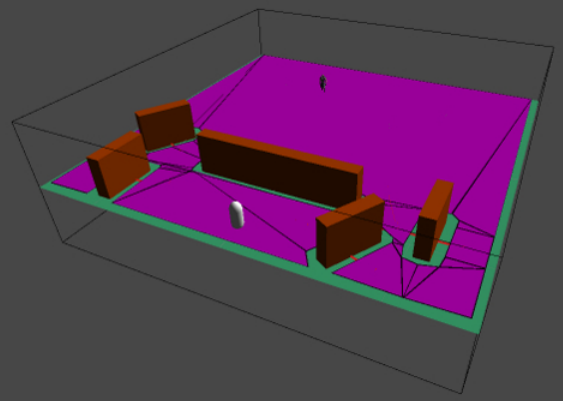
The idea is to let our unit to jump into this network as soon as it detects the player.

Add walls between the network WPs.

- Each wall should block the view hiding the player to the AI unit.
- Each wall needs a Collider (line of sight) and a MeshRenderer (nav mesh).
- Ensure a WP behind each wall to enable the unit to search behind the wall.

Rebake the RAIN Navigation Mesh to integrate the walls (as obstacles).

- Ensure the RAIN navigation mesh enables the unit to reach each WP.



USING A GAME AI ENGINE IN UNITY 16

Implementing a Search for Player Behavior Using the RAIN AI Engine - Part 3

Edit the BT adding a Constraint node to the "selectorPatrolOrAttack".

Name the new Constraint node "constraintPlayerNotDetectedAndSearching".

varPlayer == null && isSearching

Modify the "constraintPlayerNotDetected" node accordingly.

varPlayer == null && !isSearching

Rename "constraintPlayerNotDetected" as:

"constraintPlayerNotDetectedAndNotSearching"

USING A GAME AI ENGINE IN UNITY 17

Implementing a Search for Player Behavior Using the RAIN AI Engine - Part 4

Add a Parallel node as child of the "constraintPlayerNotDetectedAndSearching" node.

Add an Choose Path Waypoints action node as child of the Parallel node.

Set the Waypoint Network name (*use quotations!*).

Set the the Path Target as "varPathTarget".

Set the Move Target Variable as "varMoveTarget".

Add a Move action node "moveToWanderWP" as child of the waypoint node.

Set Move Target as "varMoveTarget" and Move Speed to 2.

Add an Animate action node "animateWalk" as a child of the Parallel node.

Set its Animation State to "walk".

Add an Evaluate Expression action node as a child of the "constraintPlayerDetected".

Name the Expression node "expressionSearchingOn": isSearching = true

Move "expressionSearchingOn" as the 1st child of "constraintPlayerDetected".

USING A GAME AI ENGINE IN UNITY

Implementing a Search for Player Behavior using the RAIN AI Engine - Part 5

Add a Custom Action as 1st child of "constraintPlayerNotDetectedAndSearching" node.

For its Class choose Create Custom Action.

Name the script as "RandomWanderLocation" (C#).

RMB click on the Custom Action to edit the script.

See the script: RandomWanderLocation.cs

Edit the "wanderWaypointNetwork" action node.

Path Target as: "varPathTarget"

In the script: ai.WorkingMemory.SetItem<Vector3>("varPathTarget", loc);

Edit the Parallel nodes: root and the "constraintPlayerNotDetectedAndSearching" child.

Set their Succeed property to Any.

Add a timer to the script to set isSearching to false after a while.

Test the game while looking the Behavior Tree! See the file: MaxBehaviorTree07.xml

USING A GAME AI ENGINE IN UNITY

Implementing a Search for Player Behavior using the RAIN AI Engine - Part 6

See the RandomWanderLocation.cs script.

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using RAIN.Core;
using RAIN.Action;
using RAIN.Navigation;
using RAIN.Navigation.Graph;

[RAINAction]
public class RandomWanderLocation : RAINAction {
    private static float _startTime = 0f;
    public RandomWanderLocation() { actionName = "RandomWanderLocation"; }
```

USING A GAME AI ENGINE IN UNITY 20

Implementing a Search for Player Behavior using the RAIN AI Engine - Part 7

```
public override void Start( AI ai ) {
    base.Start(ai);
    _startTime += Time.time; }

public override ActionResult Execute( AI ai ) {
    Vector3 loc = Vector3.zero; // Default.
    // Create a navigation graph collection.
    List<RAINNavigationGraph> found = new List<RAINNavigationGraph>();
    // Create a vector location based on our AI current location,
    // plus a random range values for X and Z coordinates.
    do{
        loc = new Vector3(  ai.Kinematic.Position.x+Random.Range(-4,4),
                            ai.Kinematic.Position.y,
                            ai.Kinematic.Position.z+Random.Range(-4,4));
```

USING A GAME AI ENGINE IN UNITY 21

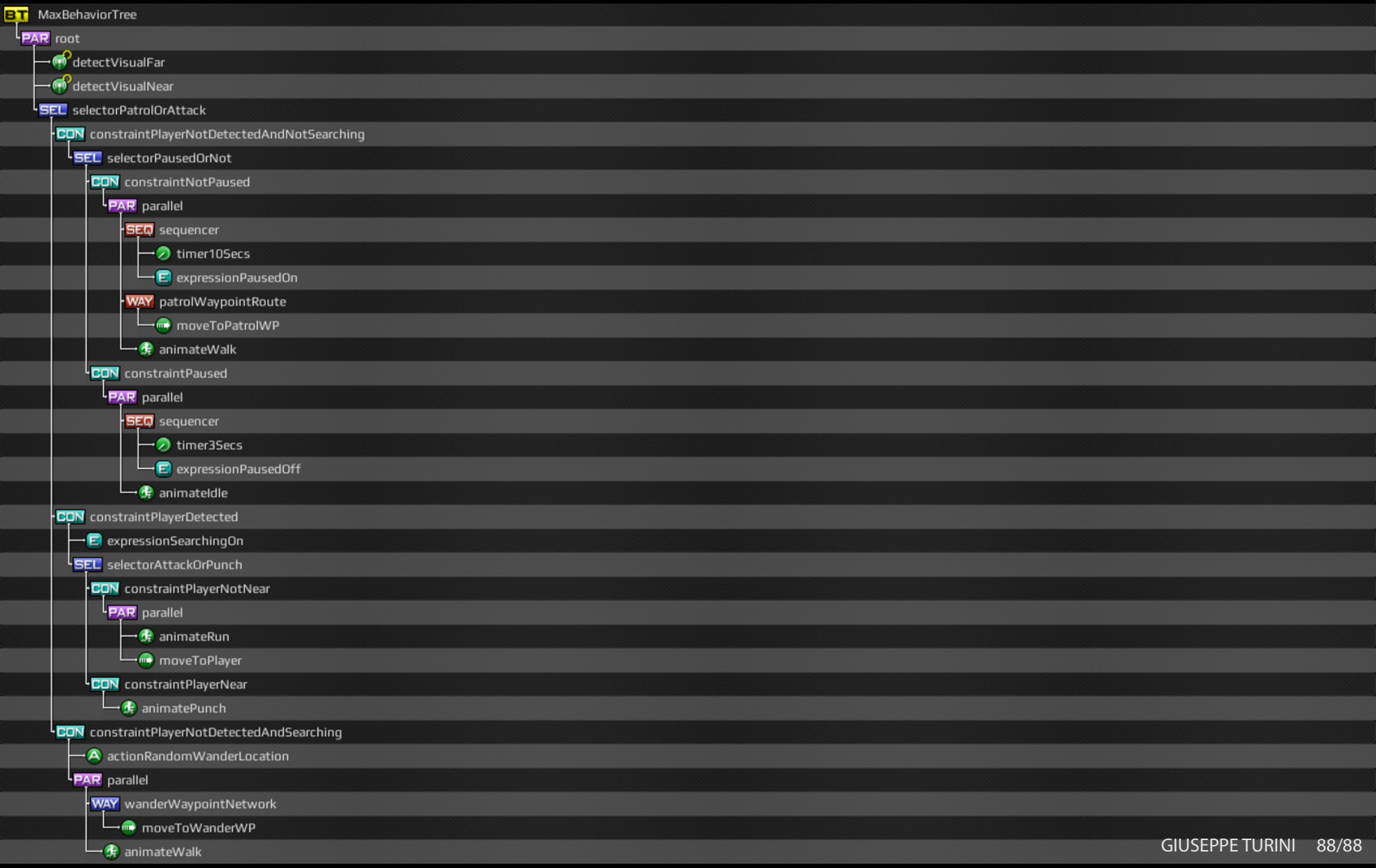
Implementing a Search for Player Behavior using the RAIN AI Engine - Part 8

```
        // Create navigation points using the above calculated value,
        // the current positon ensuring it is inside the nav mesh.
        found = NavigationManager.Instance.GraphsForPoints(
                                ai.Kinematic.Position, loc,
                                ai.Motor.StepUpHeight,
                                NavigationManager.GraphType.Navmesh,
                                ((BasicNavigator)ai.Navigator).GraphTags );
        // Ensure the location found is not too close,
        // so we don't pick anything to close or the same one.
    } while(    ( Vector3.Distance(ai.Kinematic.Position,loc) < 2.0f ) ||
                ( found.Count == 0 ) );
    // Define a runtime variable in the AIRig memory element panel.
    // Select this in your inspector to see the output at runtime.
    ai.WorkingMemory.SetItem<Vector3>( "varPathTarget", loc );
```

USING A GAME AI ENGINE IN UNITY

Implementing a Search for Player Behavior using the RAIN AI Engine - Part 9

```
        // Timer.
        if( _startTime > 100.0f ) {
            ai.WorkingMemory.SetItem( "isSearching", false );
            _startTime = 0; }
        // Action result.
        return ActionResult.SUCCESS;
    }

    public override void Stop( AI ai ) { base.Stop(ai); }

}
```


BT MaxBehaviorTree
PAR root
detectVisualFar
detectVisualNear
SEL selectorPatrolOrAttack
CON constraintPlayerNotDetectedAndNotSearching
SEL selectorPausedOrNot
CON constraintNotPaused
PAR parallel
SEQ sequencer
timer10Secs
expressionPausedOn
WAY patrolWaypointRoute
moveToPatrolWP
animateWalk
CON constraintPaused
PAR parallel
SEQ sequencer
timer3Secs
expressionPausedOff
animateIdle
CON constraintPlayerDetected
expressionSearchingOn
SEL selectorAttackOrPunch
CON constraintPlayerNotNear
PAR parallel
animateRun
moveToPlayer
CON constraintPlayerNear
animatePunch
CON constraintPlayerNotDetectedAndSearching
actionRandomWanderLocation
PAR parallel
WAY wanderWaypointNetwork
moveToWanderWP
animateWalk