AI Core Documentation

**Abstract class Action**

* Actions are various things that can be done in the game

Methods:

isComplete()

return type: bool

parameters : none

Must override this function, determines when the action is complete

canInterrupt()

return type: bool

parameters : none

returns if this action can be interrupetd

act()

return type: none

parameters : none

Must override this function, this is the actual actions that happens(such as play animations and/or sounds)

getPriority()

return type: float

parameters : none

reutrns the priority of the action

**Class ActionSequence**

* Holds multiple actions to execute in order

Methods:

ActionSequenc()

return type: AcitonSequence

Parameters : none

Default constructor

~ActionSequenc()

return type: none

Parameters : none

Default destructor

canInterrupt()

return type: bool

parameters : none

tests if any of the actions can be interrupted, if one action can be then the entire sequence can be interrupted

isComplete()

return type: bool

parameters : none

tests to see if all the actions in the sequence is complete

execute()

return type: none

parameters : none

Goes through each action and performs one individually

addAction()

return type: none

parameters : Action\*

adds an individual action to the sequence

addAction()

return type: none

parameters : ActionSequence

adds a sperate action sequence to the this sequence

getAction()

return type: none

parameters : list<Action\*>

Takes in a list of actions and fills it out with its current actions

**Abstract Class DecisionTreeNode**

* Can inherit from this class to create a decision tree

Methods:

makeDecision()

return type: DecisionTreeNode\*

parameters : none

Must override this function, makes a decision and decides which branch to follow on the tree

**Abstract Class Decision**

* Can inherit from this class to create a binary decision

Methods:

getBranch()

return type: DecisionTreeNode\*

parameters : none

Must override this function, decides which branch to go down

makeDecision()

return type: DecisionTreeNode\*

parameters : none

Makes a decision and decides and travels to next branch

**Abstract Class MultiDecision**

* Can inherit from this class to create a decision with multiple outcomes

Methods:

getBranch()

return type: DecisionTreeNode\*

parameters : none

Must override this function, decides which branch to go down

makeDecision()

return type: DecisionTreeNode\*

parameters : none

Makes a decision and decides and travels to next branch

**Abstract Class DecisionTreeAction**

* T his is at the end of the branches and is the outcome of the decisions

Methods:

makeDecision()

return type: DecisionTreeNode\*

parameters : none

returns itself giving the action at the end of the decision tree

**Abstract Class Condition**

* Used to check whether it is triggered

Methods:

test()

return type: Bool

parameters : none

Must override this function, determines the test of if this condition is met

**Class FSM\_State**

* State for the Finite State Machine to manage

Methods:

FSM\_State()

return type: none

parameters : none

Empty constructor

~FSM\_State()

return type: none

parameters : none

Empty destructor

FSM\_State()

return type: none

parameters : ActionSequence, ActionSequence, ActionSequence, list<Transition>

Constructor that takes 3 action sequences and list of type Transition

Takes in and sets entry actions, exit actions, and active actions and

a list of transitions to other states

getActions()

return type: ActionSequence

parameters : none

returns the active actions of this state

getEntryAction()

return type: ActionSequence

parameters : none

returns the entry actions of this state

getExitAction()

return type: ActionSequence

parameters : none

returns the exit actions of this state

getTransition()

return type: Void

parameters : list<Transition>

takes in a list and fills it out with the state's transitions

**Class Transition**

* This is used to flag the state machine to change states

Methods:

Transition()

return type: none

parameters : none

Constructor

~Transition()

return type: none

parameters : none

Destructor

Transition()

return type: none

parameters : ActionSequence, FSM\_State, Condition\*

Constructor that takes in ActionSequence of actions, a FSM\_State for the taget state and Condition\* for its condition

getActions()

return type: ActionSequence

parameters : none

returns actions

getTargetState()

return type: FSM\_State

parameters : none

returns target state to transition to if this transition is triggered

isTriggered()

return type: Bool

parameters : none

tests whether or not this transition's condition is met

**Class FiniteStateMachine**

* This is used to manage states and transition between them

Methods:

FiniteStateMachine()

Return type: none

parameter: FSM\_State initial state

Constructor that initializes the state machine with an initialState

update()

return type: actionSequence

parameters : none

checks the if the state should change and returns an ActionSequence of actions to execute

getCurrentState()

return type: FSM\_State

parameters : none

returns the current state of the Finite State Machine

**Class Kinematic**

* Used for simple kinematic behaviors(Seek, Flee, Wander)

Methods:

Kinematic()

return type: none

parameters : none

Default constructor

~Kinematic()

return type: none

parameters : none

Destructor

Kinematic()

return type: none

parameters : Vector2D, Float, Vector2D, Float, Float, Float)

sets parameters to corresponding data members

Position()

return type: Vector2D

parameters : none

returns the position

Rotation()

return type: float

parameters : none

returns the rotation

Target()

return type: Vector2D

parameters : none

returns the target

MaxSpeed()

return type: float

parameters : none

returns the max speed

SlowRadius()

return type: float

parameters : none

return the radius that the object starts to slow

WanderRotation()

return type: float

parameters : none

returns the rotation of the wander

setPosition()

return type: none

parameters : Vector2D

sets the position

setRotation()

return type: none

parameters : float

sets the rotation

setTarget()

return type: none

parameters : Vector2D

sets the target

setMaxSpeed()

return type: none

parameters : float

sets the max speed

setSlowRadius()

return type: none

parameters : float

sets the radius to start slowing when you seek something

setWanderRotation()

return type: none

parameters : float

sets the angle to wander

Seek()

return type: SteeringOutput

parameters : none

Returns a linear velocity and angular velocity to go from your position to your taget

Flee()

return type: SteeringOutput

parameters : none

Returns a linear velocity and angular velocity to get away from your taget

Wander()

return type: SteeringOutput

parameters : none

Returns a linear velocity and angular velocity that simulates wandering

**Class PathNode**

* Individual node that combined with other nodes and connections make a graph

Methods:

PathNode()

return type: none

parameters : none

Constructor

isOpen()

return type: Bool

parameters : none

returns a bool of whether or not this node is currently open

addConnection()

return type: Void

parameters : PathNode\*, Float

creates a connections between this and another node with the specified cost, if the connection already exists it skips over it

getConnections()

return type: Void

parameters : list<Connection\*> \*

fills out the list of connections that it was passed with all connections from this node

closeNode()

return type: Void

parameters : none

Closes this node

openNode()

return type: Void

parameters : none

Opens this node

**Class Connection**

* A connection between two nodes in a graph

Methods:

Connection()

return type: Void

parameters : PathNode\*, PathNode\*, Float

Constructor, creates a connection from fromNode to toNode with the specified cost

getCost()

return type: Float

parameters : none

returns the cost of this connection

getToNode()

return type: PathNode\*

parameters : none

Returns the toNode from this connection

getFromNode()

return type: PathNode\*

parameters : none

Returns the fromNode from this connection

**Class Graph**

* Stores a set of nodes and can traverse through them

Methods:

Graph()

return type: none

parameters : none

constructor

addNode()

return type: Void

parameters : PathNode\*

Adds a node the graph

traverse()

return type: Bool

parameters : PathNode\*, PathNode\*, list<Connection\*>\*

Using the dijkstra algorithm, finds a path of lowest cost from start to end and fills out path with connections if a path was found

Will return true if a path was found, false if it wasn't