# Modelling of NPCs

With the use of interacting statecharts

Universiteit Antwerpen

# Overview

- Why statecharts?
- Related work
- My contribution
- Conclusion

# Why Statecharts?

# Turn Based Games

- Popular examples include computerized board games like Chess and Connect-Four
- Game state does not change until a player makes a move
- Waiting several seconds for (computer-controlled) opponent is acceptable
- “Simple” algorithms within programming language suffice

# Real Time Games

- Examples : your favorite FPS or MMORPG
- Game state changes continuously
- Goal : make NPCs' actions and reactions look as intelligent and natural as possible
- More realism when NPC can :
  - Analyze situation
  - Evaluate different options
  - Take into account game history

→ Writing consistent, re-usable and efficient AI code becomes very hard

# Solution

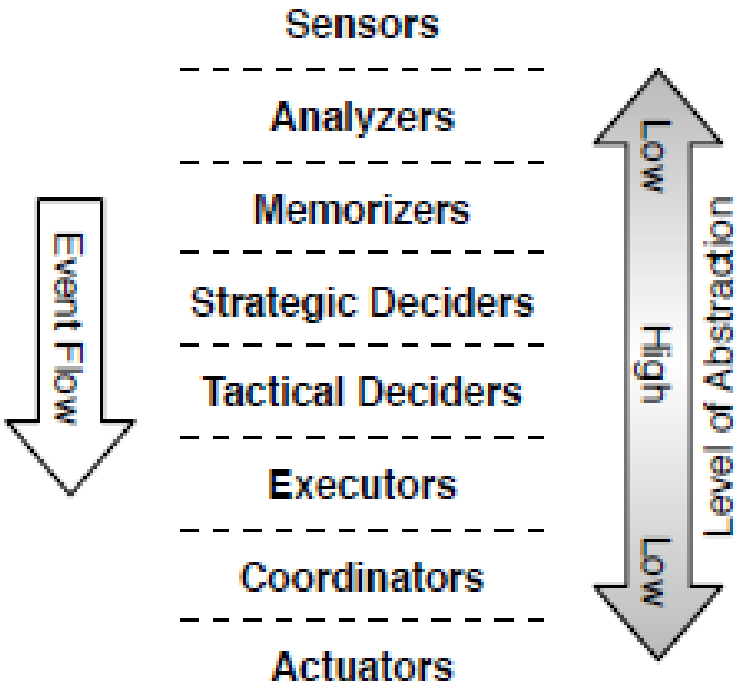
- Specification of such advanced AI should not be done within programming language
- Instead : higher level of abstraction using visual modelling language
- Main focus in Game AI is to define reactions to game events

→ An Event based formalism like Statecharts seems appropriate

# Related work

# Model-Based Design Of Computer-Controlled Game Character Behavior

by Jörg Kienzle, Alexandre Denault & Hans Vangheluwe



Sensors

Analyzers

Memorizers

Strategic Deciders

Tactical Deciders

Executors

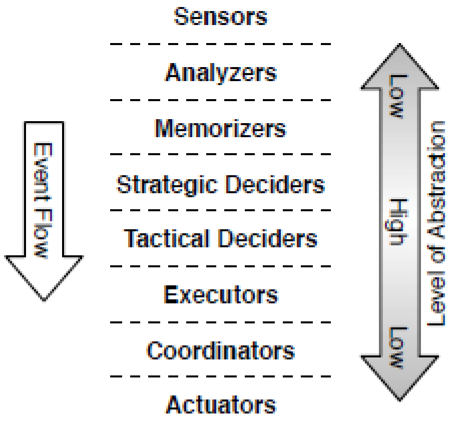
Coordinators

Actuators

## The layered architure of the AI model

As described in the paper "Model-Based Design Of Computer-Controlled Game Character Behavior" by Jörg Kienzle, Alexandre Denault & Hans Vangheluwe

# Architecture

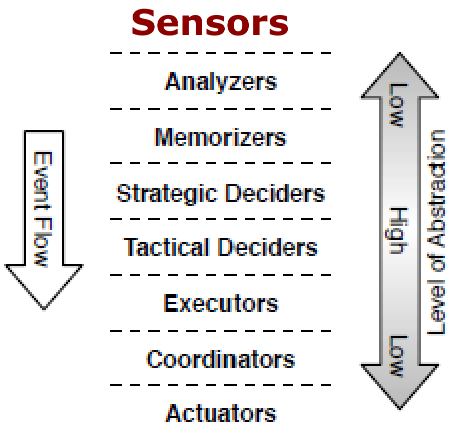


- Character perceives the environment through his sensors
- Input gets transformed by components from the layers
- Eventually reaction by the actuators
- Communication with asynchronous events ( event flow downwards)
- Example : Detecting obstacle

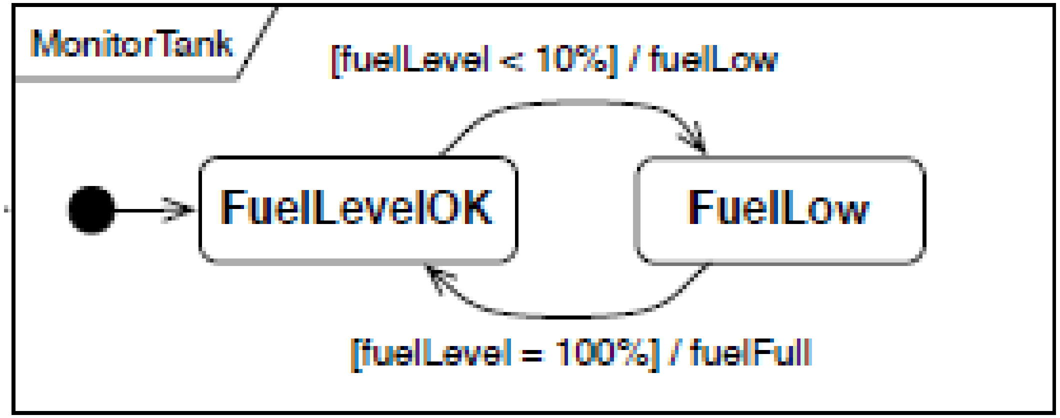
  → turning left to avoid collision.

# Sensors

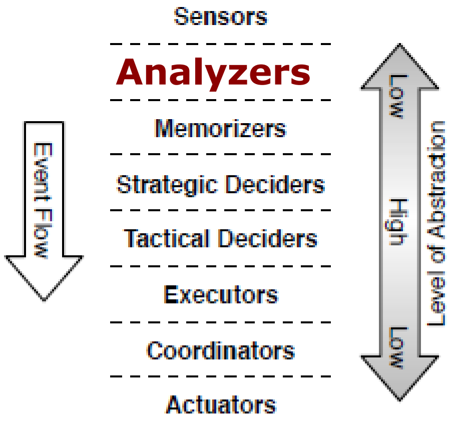
- Extract information from the state of the tank ( evolves continuously)
- Send events accordingly



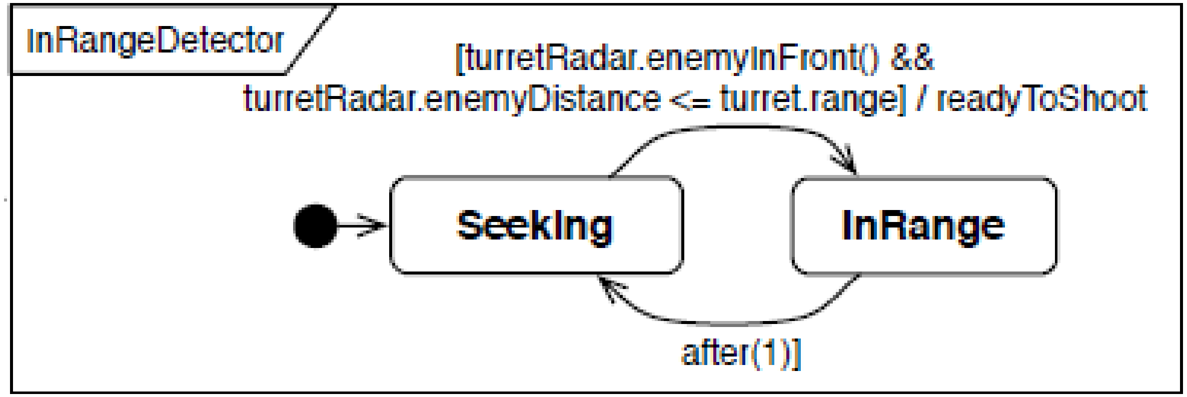
- Example :



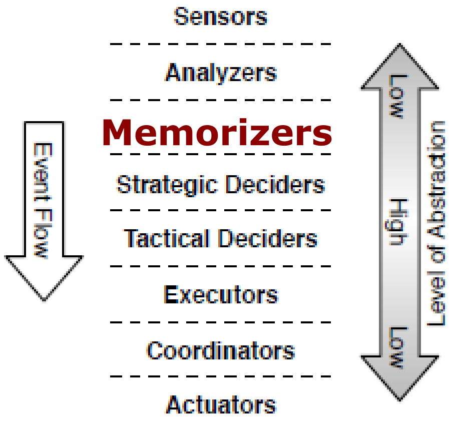
# Analyzers



- Detect significant events that can only be calculated based on the state of several components
- Example - To determine whether the enemy is in range, information from the turret and the turret radar is needed :

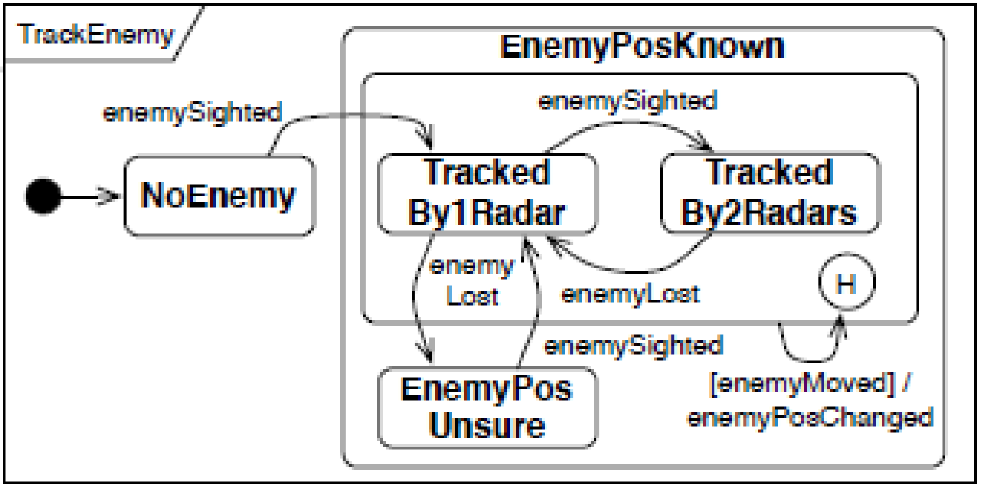

InRangeDetector
[turretRadar.enemyInFront() &&
turretRadar.enemyDistance <= turret.range] / readyToShoot
Seeking
InRange
after(1)]

# Memorizers



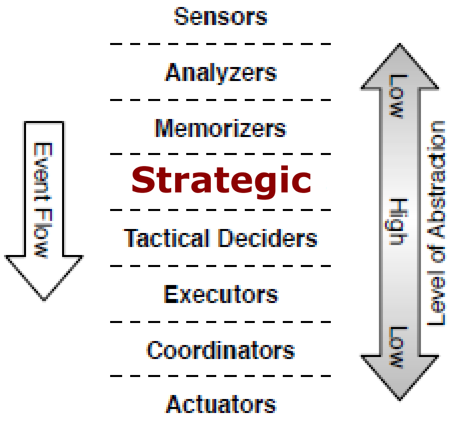
- Pilot takes events/state from the past also in consideration

  → Memory needed
- Example – Enemy Tracker remembers enemy position, even when it got out of sight :



# Strategic Deciders

- Deciding on a high level goal
- Strategies : Exploring, Attacking, Repairing, Fleeing, Refueling


Event Flow

Sensors

Analyzers

Memorizers

**Strategic**

Tactical Deciders

Executors

Coordinators

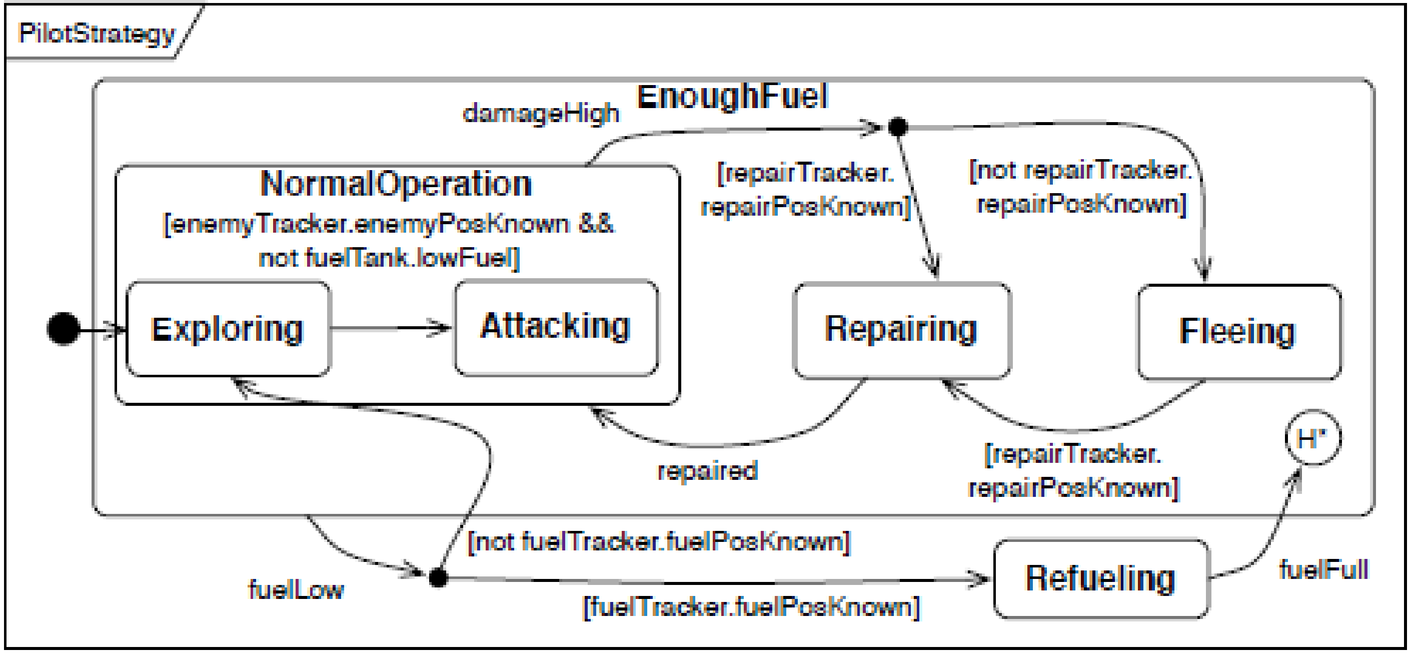
Actuators

Low

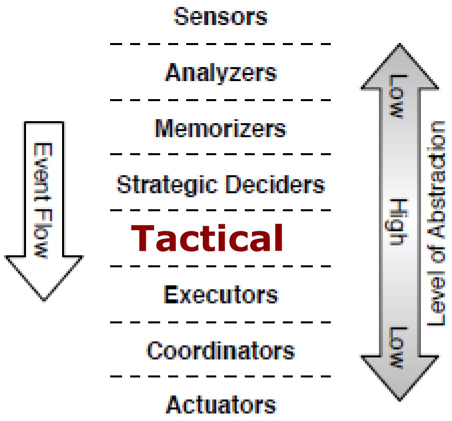
High

Low

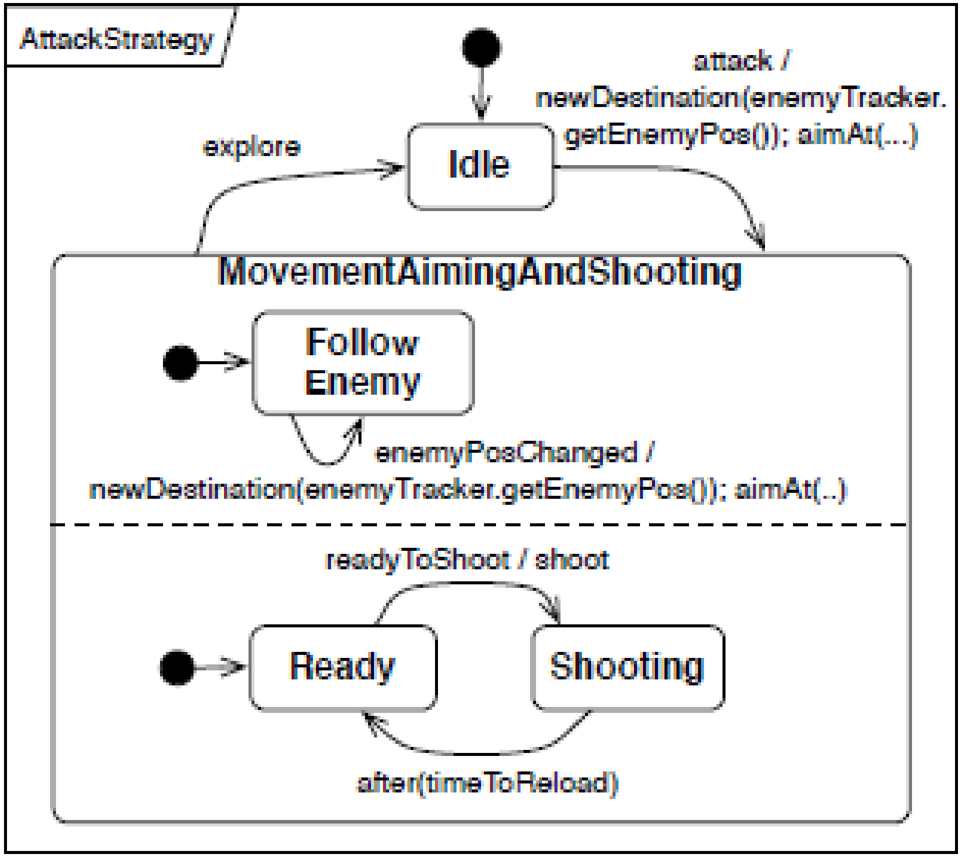
Level of Abstraction


PilotStrategy
EnoughFuel
damageHigh
NormalOperation
[enemyTracker.enemyPosKnown && not fuelTank.lowFuel]
Exploring
Attacking
[repairTracker. repairPosKnown]
[not repairTracker. repairPosKnown]
Repairing
Fleeing
repaired
[repairTracker. repairPosKnown]
H*
[not fuelTracker.fuelPosKnown]
fuelLow
[fuelTracker.fuelPosKnown]
Refueling
fuelFull

# Tactical Deciders



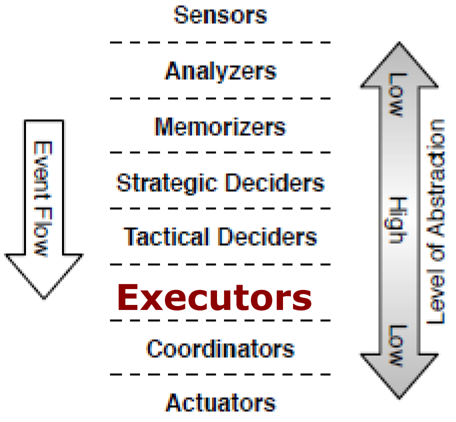
- Translate high level goals into low level commands
- Each strategy should have his own planner.

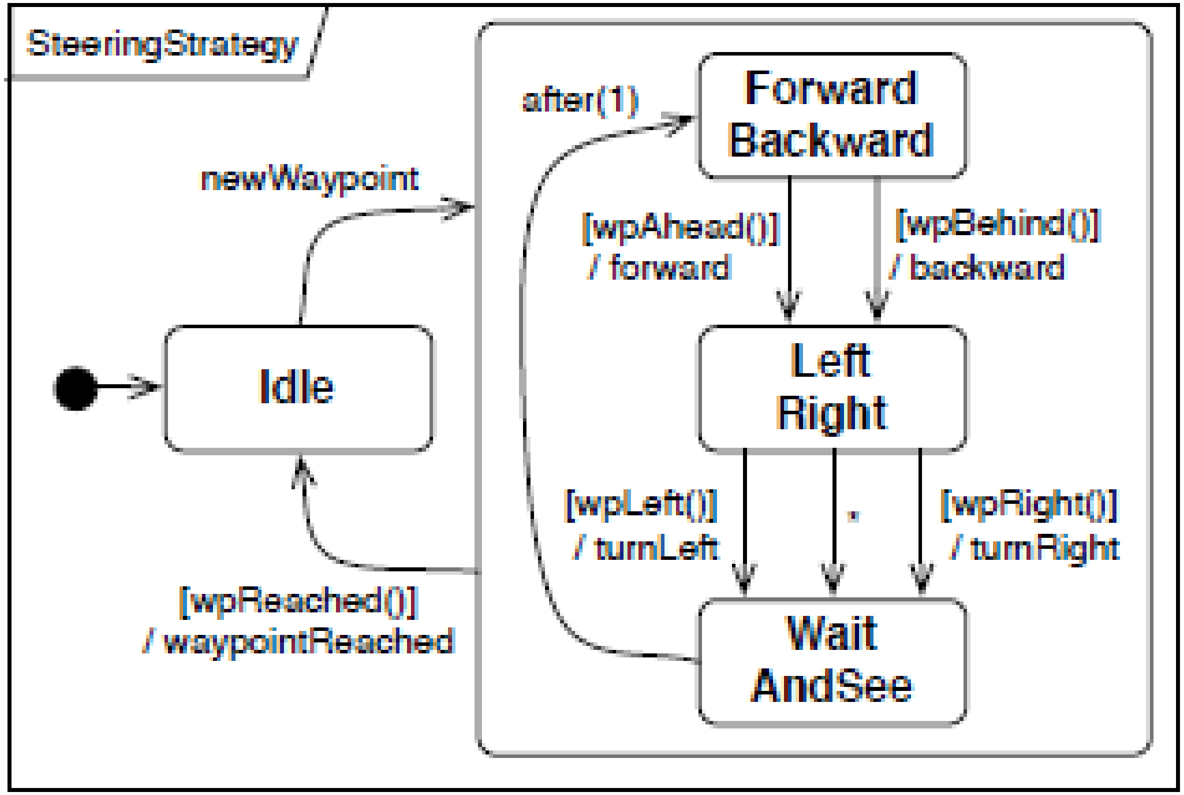

AttackStrategy
attack /
newDestination(enemyTracker.
getEnemyPos()); aimAt(...)
explore
Idle
MovementAimingAndShooting
Follow
Enemy
enemyPosChanged /
newDestination(enemyTracker.getEnemyPos()); aimAt(..)
readyToShoot / shoot
Ready
Shooting
after(timeToReload)

Planner for the attack strategy

# Executors

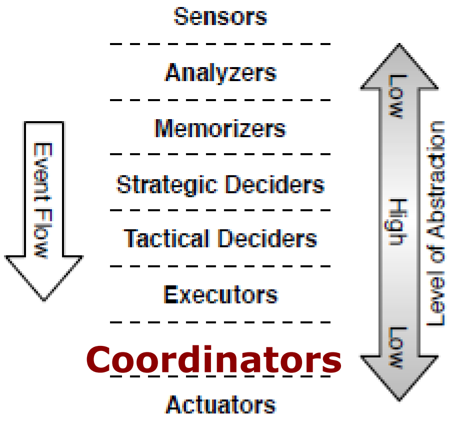
- Map the decisions to events the actuators can understand

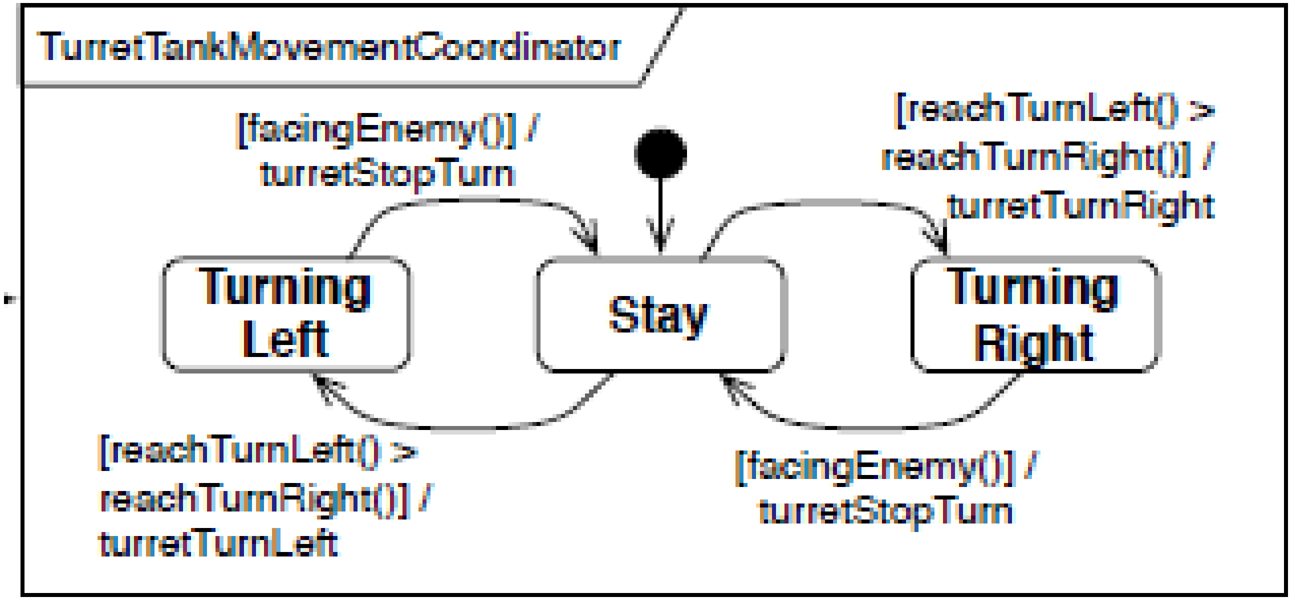




# Coordinators

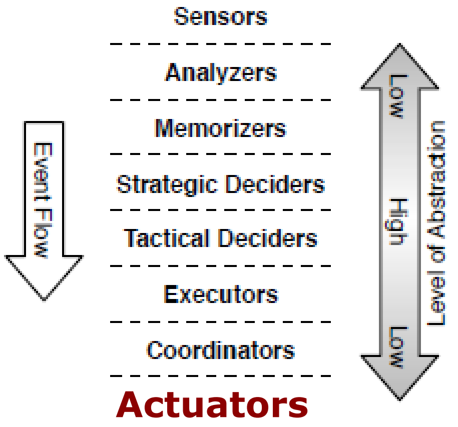
- Handle incorrect behaviour when the effects of actuators are correlated
- Example : Simultaneously turning tank and cannon




TurretTankMovementCoordinator
[facingEnemy()] / turretStopTurn
[reachTurnLeft() > reachTurnRight()] / turretTurnRight
Turning Left
Stay
Turning Right
[reachTurnLeft() > reachTurnRight()] / turretTurnLeft
[facingEnemy()] / turretStopTurn

# Actuators

- Execute the low level commands such as turn left or move forward



Sensors

Analyzers

Memorizers

Strategic Deciders

Tactical Deciders

Executors

Coordinators

**Actuators**

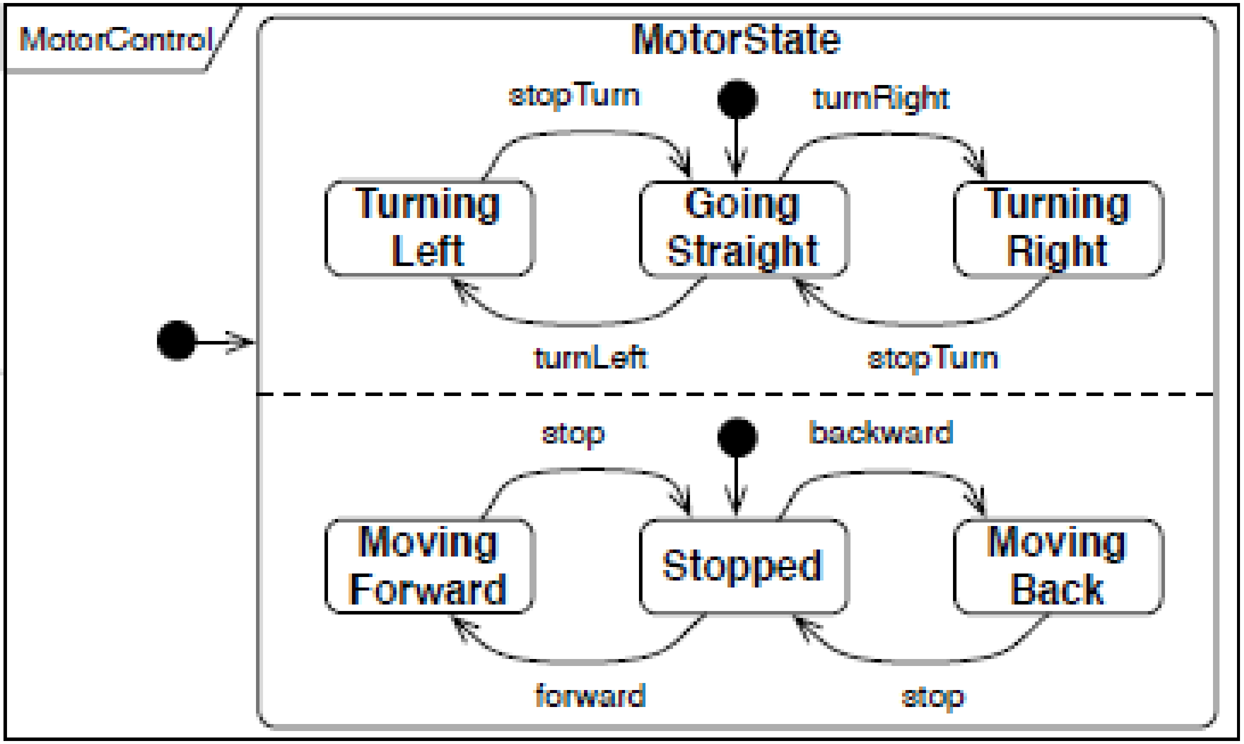
Event Flow

Low

High

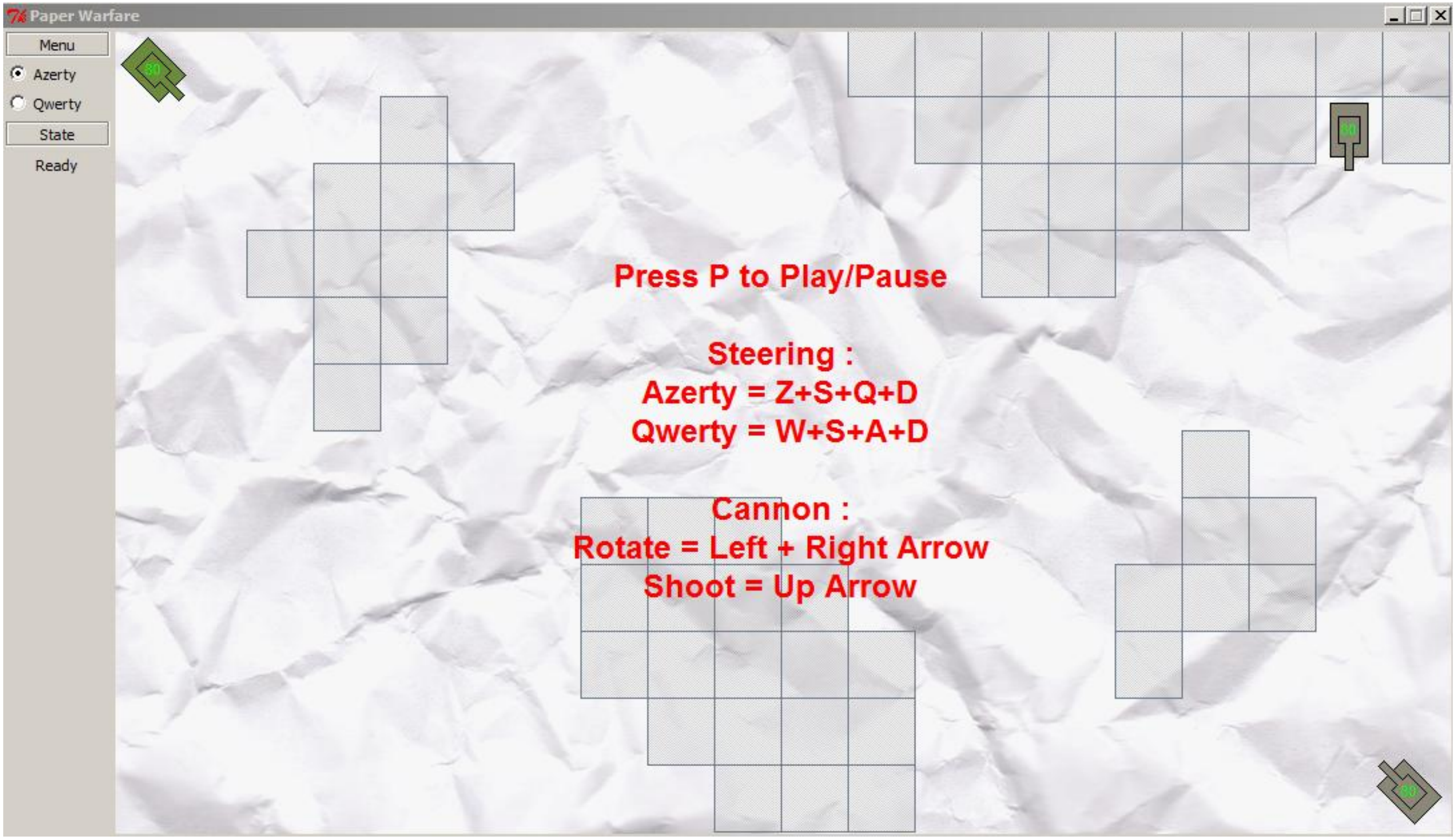
Low

Level of Abstraction



# My contribution

# Example Game : Paper Warfare



Press P to Play/Pause

Steering :
Azerty = Z+S+Q+D
Qwerty = W+S+A+D

Cannon :
Rotate = Left + Right Arrow
Shoot = Up Arrow

# Modelling

- As modelling environment AToM³ is used, in combination with the DCharts formalism and statechart compiler of Huining Fen

  [2] AToM3, http://atom3.cs.mcgill.ca/

  [3] H. Feng, DCharts, a formalism for modeling and simulation based design of reactive software system, http://msdl.cs.mcgill.ca/people/tfeng/thesis/thesis.html (2004).

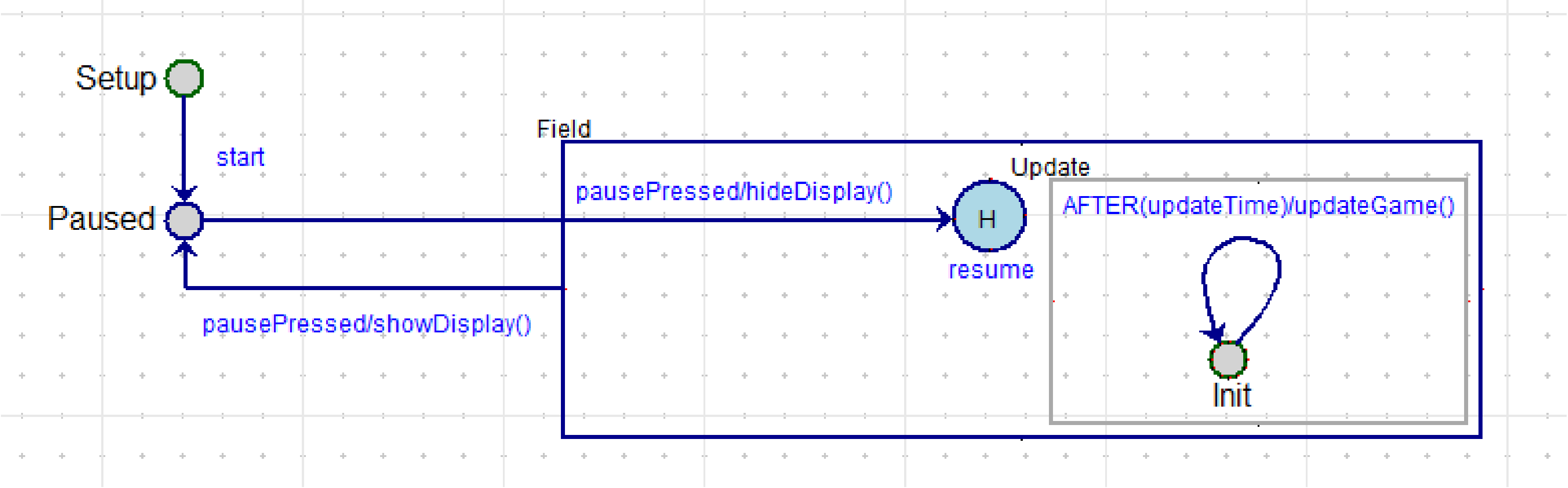
- User Interface with Tkinter

# Modelling

- A component with modelled behaviour consists out of :
  - A dynamic part : The statechart
  - A static part : Implements certain functionality which can be called by the statechart
  - A controller : For communication between the two parts
- Next to the NPCs, also other elements with modelled behaviour
- Should we model everything we can model?

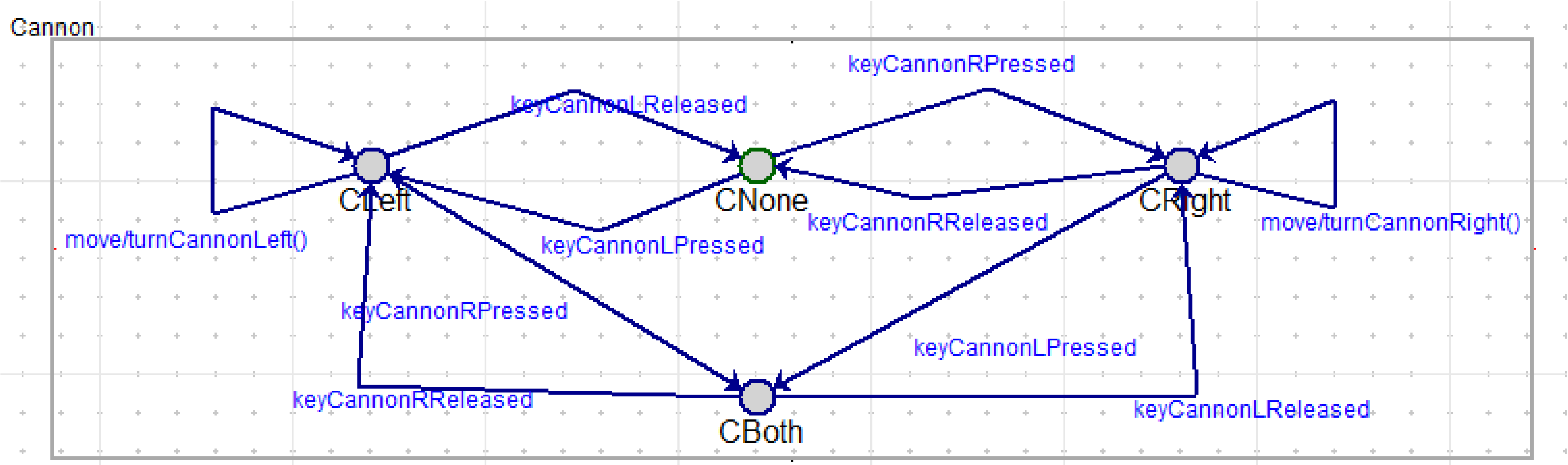
# Environment

- Field repeatedly updates all objects in game e.g. Bullet movement and collision detection → Would a separate statechart for a bullet be beneficial ?
- Pausing/resuming displays/hides a message


Setup
start
Field
Paused
pausePressed/hideDisplay()
Update
H
AFTER(updateTime)/updateGame()
resume
pausePressed/showDisplay()
Init

# Player

- Comparable to the executor & actuator layers of the AI -> input from the user is translated into actions
- Example – When the right arrow key is pressed the event "keyCannonRPressed" will be generated, resulting in the cannon turning right :

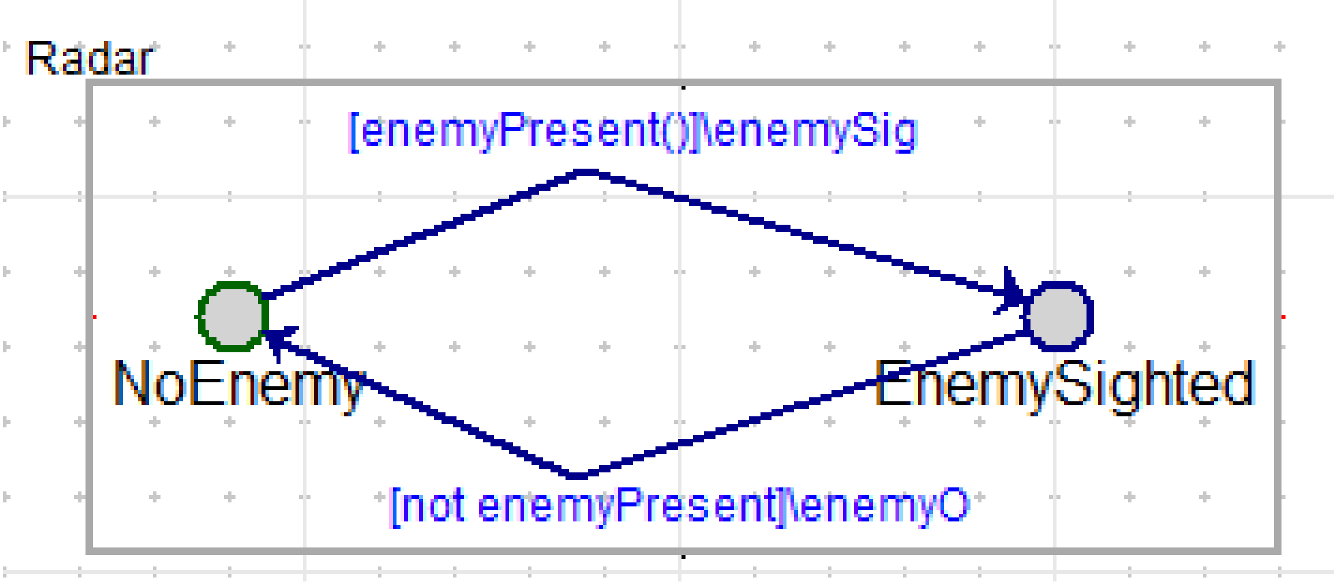


# Non-Player Character

- Same layered approach as paper in related work but different target game and platform
- Only interesting components will be shown (lots of trivial and similar components)

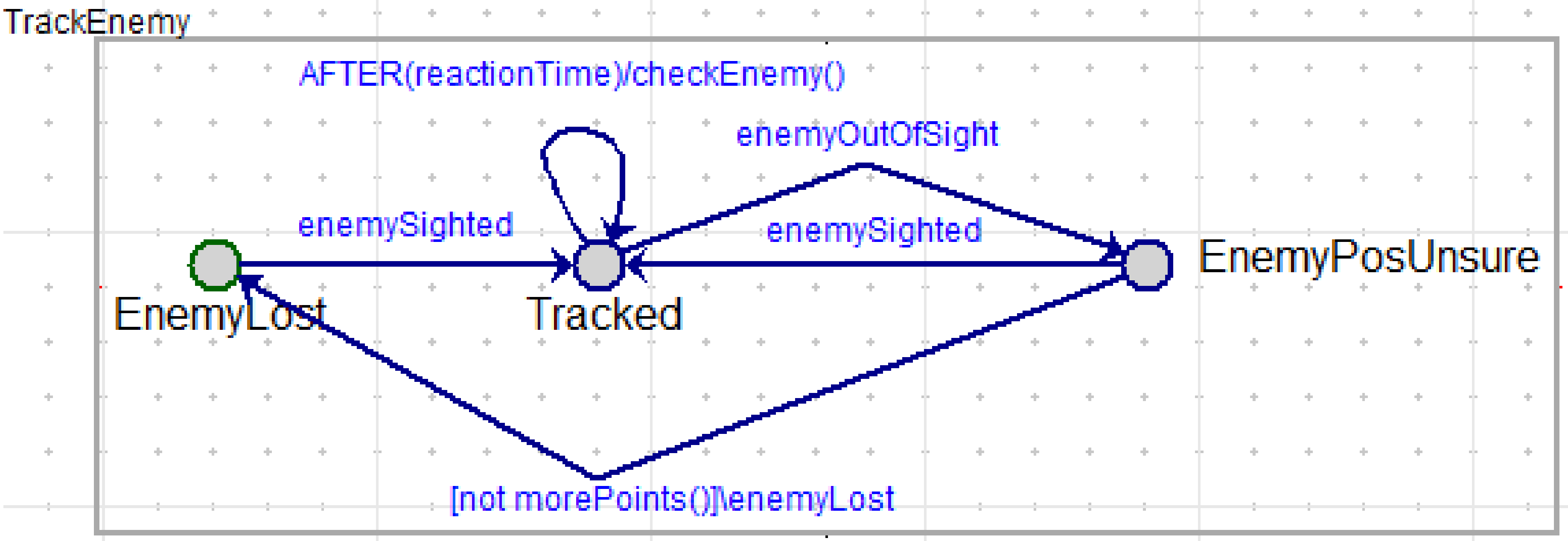
# Enemy Detection

- If enemy present, send "*enemySighted*" event and progress to EnemySighted state
- In this state keep checking for enemies, if no more enemies are present, send "*enemyOutOfSight*" event.



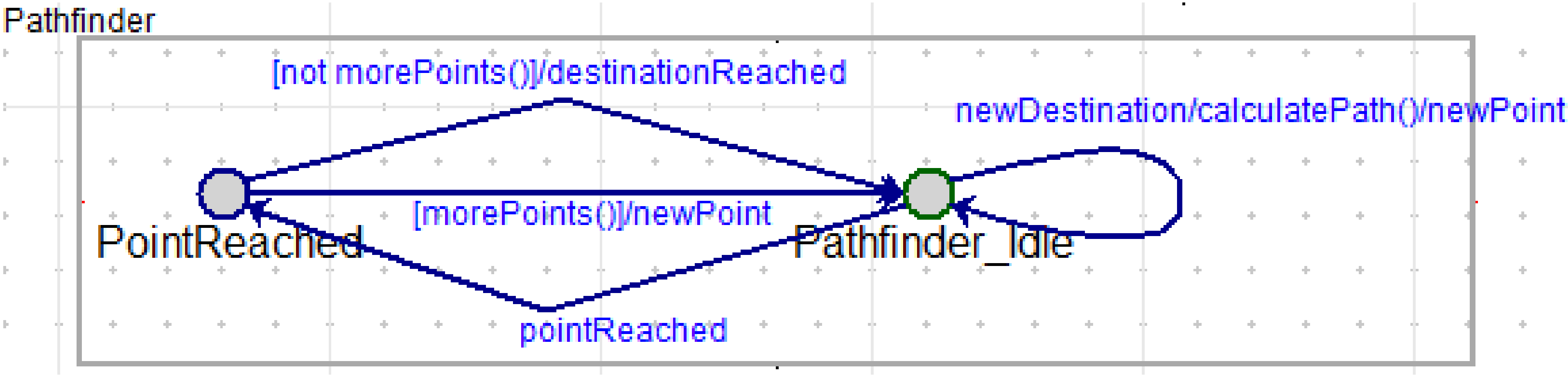
# Enemy Tracker

- Memorizer to pinpoint the enemy’s position
- Repeatedly update position of enemy
- If enemy out of sight and no waypoint left to travel to → Enemy lost, continue exploring



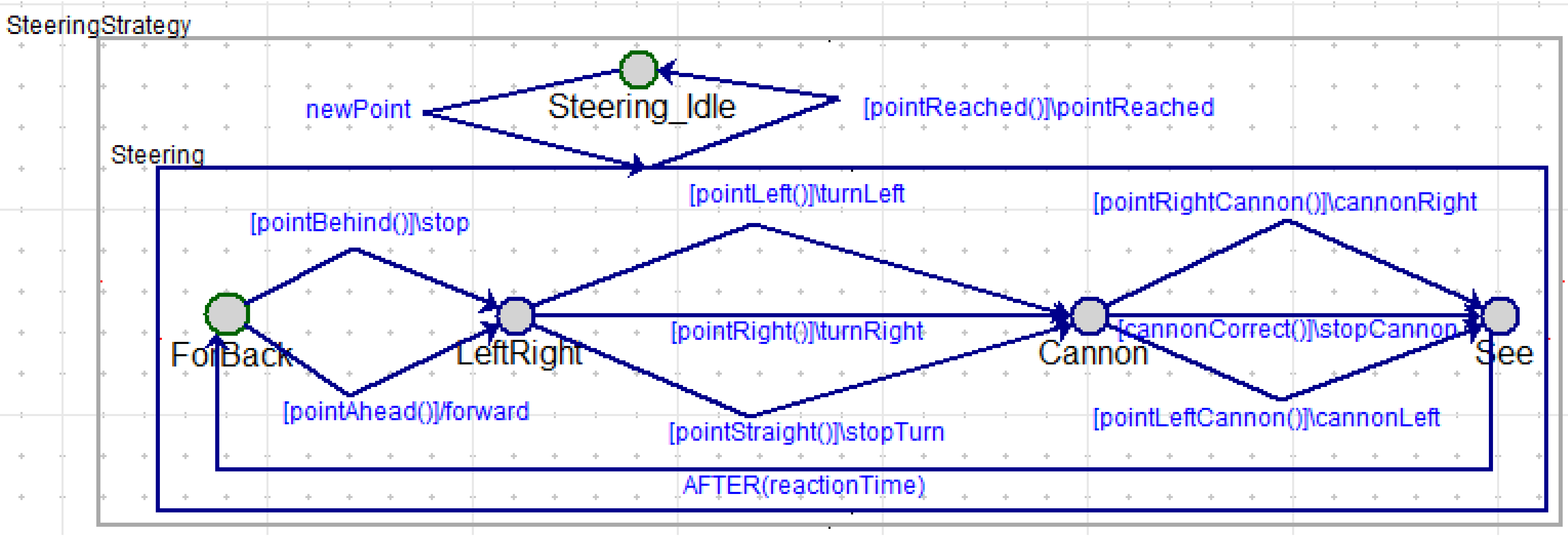
# Path Finder

- Determines route using waypoints when "*newDestination*" event comes in
- When point reached, checks if more points are left. If so, a "*newPoint*" event is send, else a "*destinationReached*" event.


Pathfinder
[not morePoints()]/destinationReached
newDestination/calculatePath()/newPoint
PointReached
[morePoints()]/newPoint
Pathfinder_Idle
pointReached

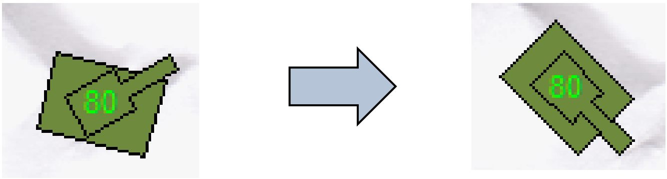
# Steering Strategy

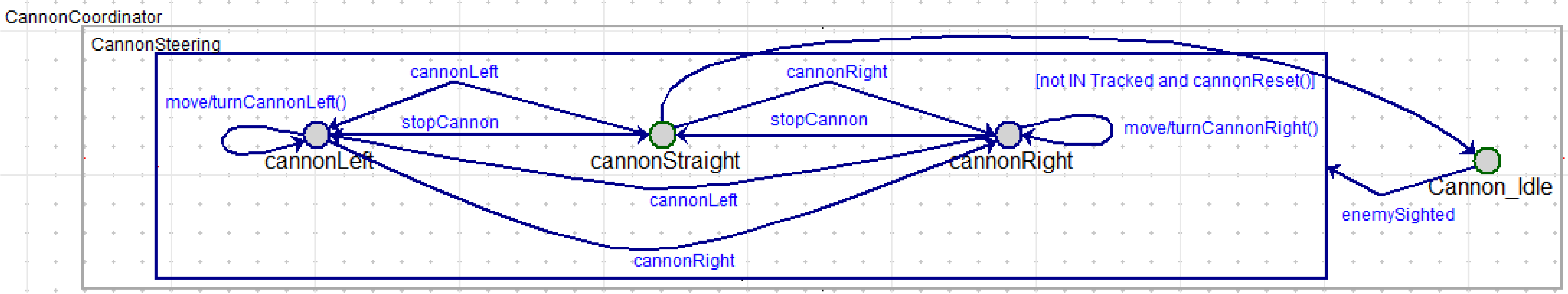
- This executor shoots in action when a new target point is set
- Checks where that point is located in relation to itself and propagates events accordingly.



# Cannon Coordinator

- Next to enforcing the desired behaviour of the cannon, it also attempts to reset the cannon to a zero angle difference with the tank when the attack state is left.





# Demo Time

# Conclusion

# Conclusion

- Statechart modelling = good way to obtain structured and easy-to-understand AI
- Usefull in other cases where keeping track of state is needed (e.g. what key is pressed / pausing game)
- Degrades performance → Structure, Consistency & Re-usability vs. Performance

- (Tkinter is not well suited for games)

Any questions?

# Thank you for listening