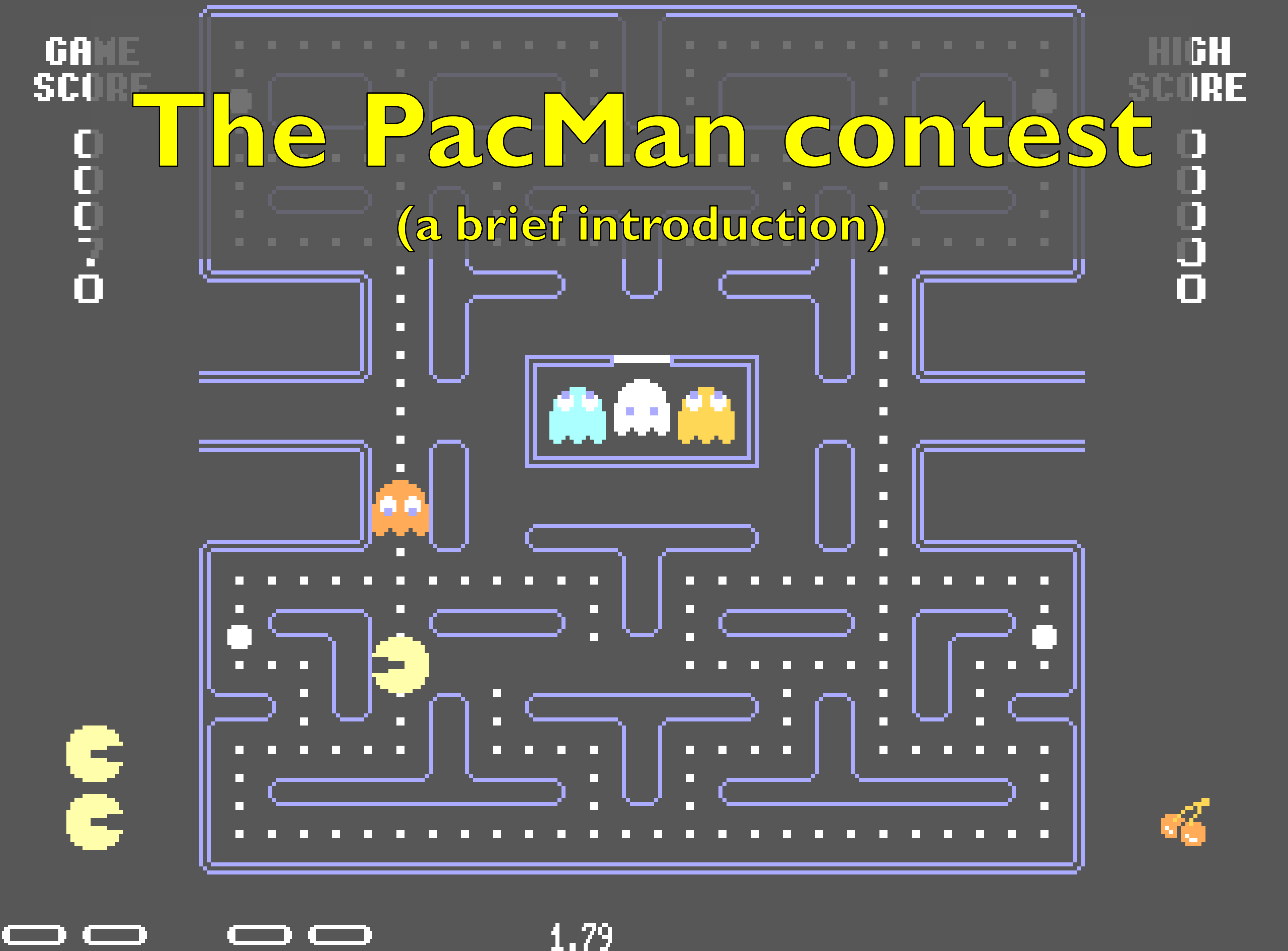
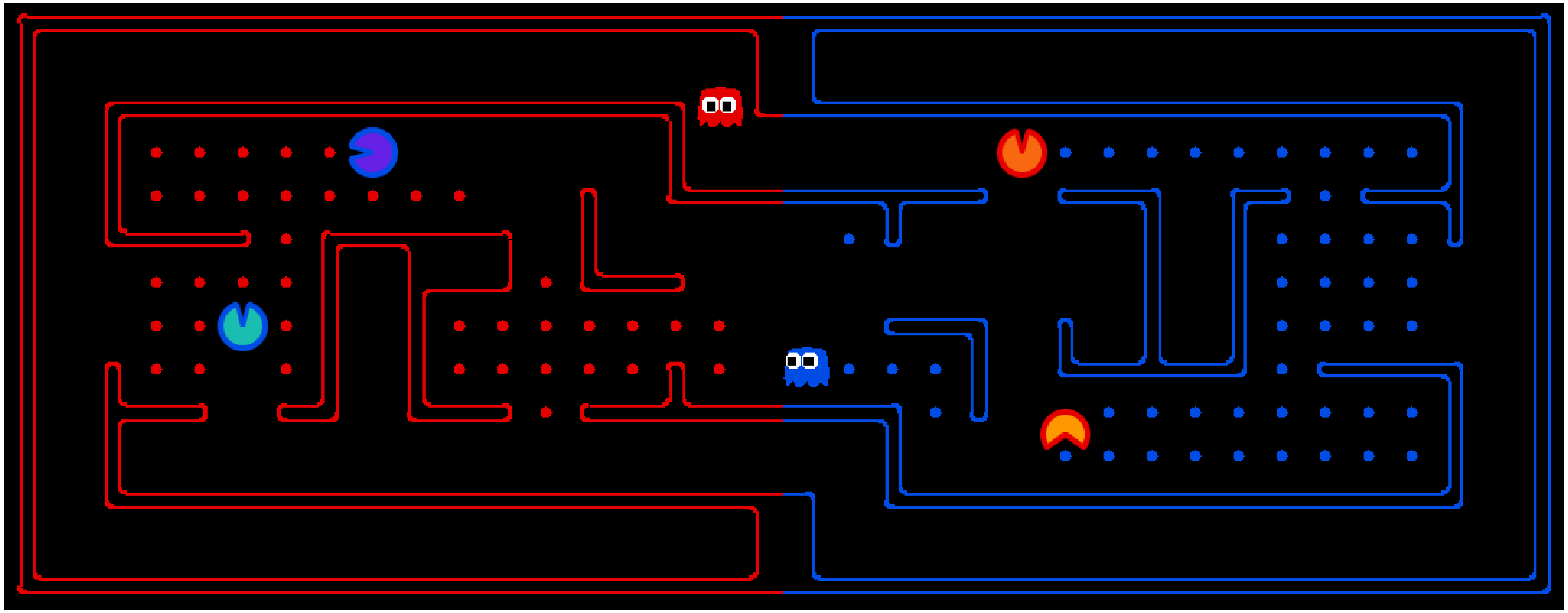
The PacMan contest

(a brief introduction)



PacMan capture-the-flag



The rules

- **Scoring:** When a Pacman eats a food dot, the food is permanently removed and one point is scored for that Pacman's team. Red team scores are positive, while Blue team scores are negative.
- **Eating Pacman:** When a Pacman is eaten by an opposing ghost, it returns to its starting position (as a ghost). No points are awarded for eating an opponent.
- **Winning:** A game ends when either one team eats all of the opponents' dots, or after 3000 agent moves. A final positive score means that the Red team wins, a negative one means that Blue wins.
- **Observations:** Agents can only observe an opponent's configuration (position and direction) if they or their teammate is within 5 squares (Manhattan distance). In addition, an agent always gets a noisy distance reading for each agent on the board, which can be used to approximately locate unobserved opponents.

The tournament

- On Day 3, we'll have some practice rounds for those who have agents ready to test
- On Day 4, we'll have a all-against-all tournament
- The mazes for the final tournament will vary, test your agents with different layouts

Running a game

- Code in `summerschool/project/pacman`
- Warning: the style of the PacMan code is not an example to follow!
  - 2-spaces indentation, and camelCaseNames are bad style!
  - Stick to the Python standard, i.e., 4-spaces, underscore_separated_names
- To run a match :

```
python capture.py -r MyAgentFactory
                  -b YourAgentFactory
                  -l layout_name --fps=100
```

other options:

```
python capture.py --help
```

Writing agents 101 – AgentFactory

- Called by main application, given an agent index returns an Agent instance:
  `python capture.py --red MyAgentFactory`
- Looks in all *gents.py files in your PYTHONPATH

```
class OffenseDefenseAgents(AgentFactory):
 """ Returns one defensive agent and one offensive agent"""

  def __init__(self, **args):
    AgentFactory.__init__(self, **args)
    self.offense = False

  def getAgent(self, index):
    self.offense = not self.offense
    if self.offense:
      return OffensiveReflexAgent(index)
    else:
      return DefensiveReflexAgent(index)
```

Writing agents 101 – Agent

```
class Agent:
  def __init__(self, index=0):
    self.index = index

  def getAction(self, game_state):
    """
    The Agent will receive a GameState and
    must return an action from
    game.Directions.{NORTH,SOUTH,EAST,WEST,STOP}
    """
    pass
```

Every agent is identified by an index.

Writing agents 101 – basic_agents.BasicAgent

- We recommend to use our subclass, basic_agents.BasicAgent, which is more pythonic and defines helpful methods to analyze the game state
- (wiki)

Writing agents 101 – capture.GameState

- Represents the state of the game, can be asked for useful information
- (wiki)

Writing agents 101 – Example agent

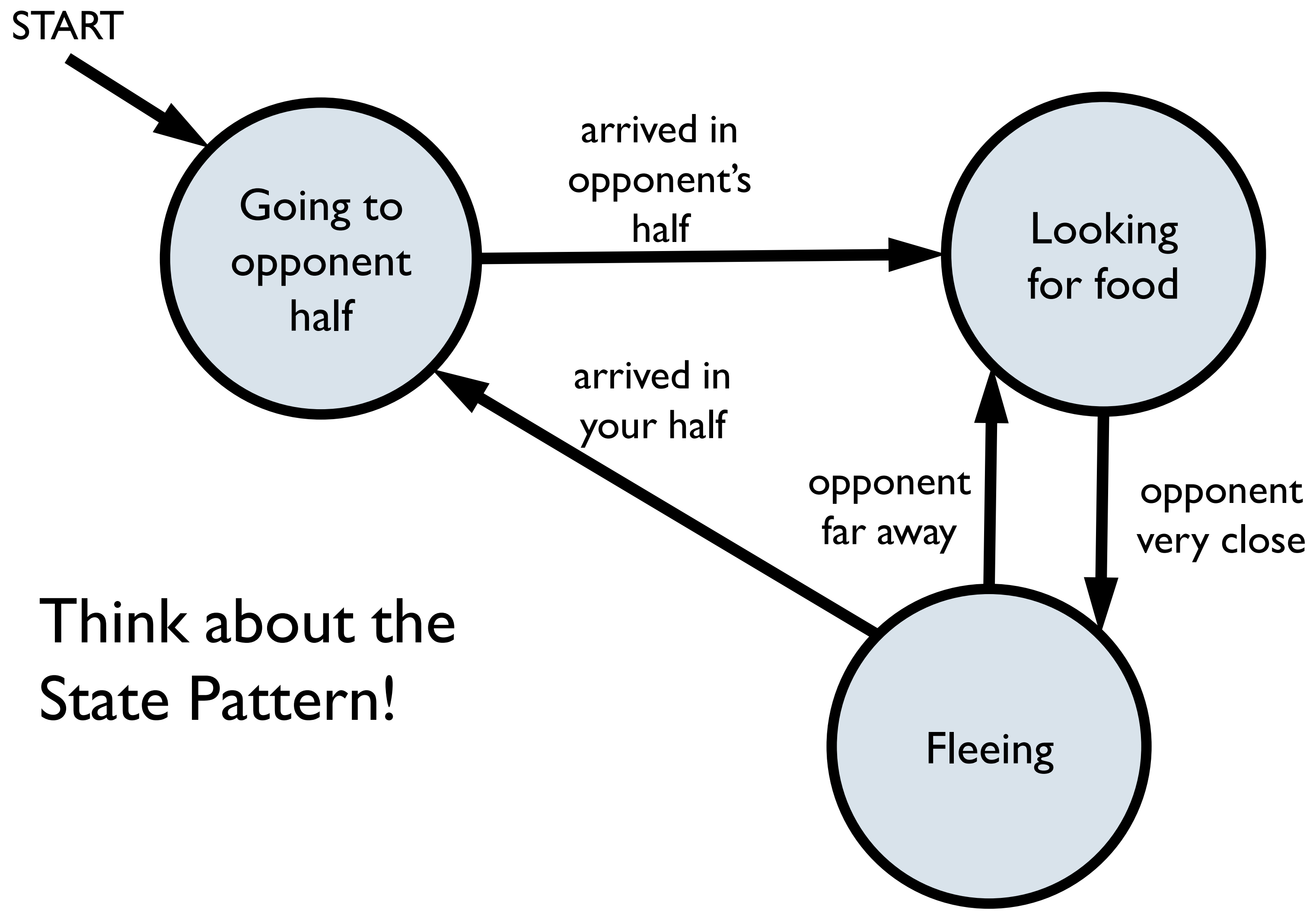
```
import random
from basic_agents import BasicAgent, BasicAgentFactory

class DrunkAgent(BasicAgent):
    def choose_action(self, game_state):
        self.say(random.choice(['Burp', 'Blah', 'Mrmmmf']))
        actions = game_state.getLegalActions(self.index)
        return random.choice(actions)
```

More in `summerschool/project/agents`

Writing agents 101 – Testing agents

- Very useful: the alternative is to run games, hope that the agents end up in the right situation, guess from looking at the screen if it behaved correctly
- More sophisticated testing scenario: you need to set up a fake game ("mock" game), put the agents in the correct situation, then run them and analyze their behavior
- (wiki)

Basic agent behaviors – Finite States Machines



Think about the State Pattern!

Basic agent behaviors – Value-maximizer

- Agent has a function that gives a value to a given game state according to several criteria, e.g.
  *value(game_state) = -1*distance_from_nearest_food*
  *+100*score*
- At each turn:
  - get the legal actions
    `game_state.getLegalActions(self.index)`
  - request the future game state given one of the actions
    `game_state.generateSuccessor(self.index, action)`
  - compute the value of future states
  - pick the action that leads to the state with the highest value

Learning

- Plenty of opportunities for learning
  - Adapt parameters according to final score
  - Reinforcement Learning (similar to learning weights in the value-maximizing agent)
  - Collect statistics on opponents
  - Ambitious: Genetic Programming
  - …

Things that we've found to be useful

- Shortest-path algorithm
- Algorithm to keep track of opponents
- Rike: communication between agents
- …

- Code re-use is encouraged
- More important than fancy strategies is the quality of your code: Is it well tested? Does it conform to standards? Apply agile development techniques

Let’s start!

- Form 5 teams of 6 people (wiki)
- Test that you can write and run matches with simple agents
  - your PYTHONPATH should contain
    ```
    export PYTHONPATH=$HOME/summerschool/project/pacman;
              $HOME/summerschool/project/agents
    ```
  - set up your project directory, put in the PYTHONPATH
  - write a RandomAgent and corresponding AgentFactory, try to have a few matches with different layouts
  - write an agent that picks a random direction at junctions
- Organize team work
- Have fun!