ART submission (MARL)

October 23, 2023

1 Joyee Chen's ART submission: MARL

This is my ART submission, focusing on the three-point MARL topic. I desire the structure and focus given by submitting any future ART project to an organized contest or effort in RL, and after talking to Mario Peng, I believe the AICrowd Melting Pot Challenge (https://www.aicrowd.com/challenges/meltingpot-challenge-2023) is quite well aligned to our purposes.

Mario allowed me to use the AICrowd gridworld, which I will do with explanations of each module and class, in the first part of this report in lieu of creating a gridworld myself. If I have time, in the second, I will set up some agents and attempt to find cooperation and compensation incentives.

2 Part 1: the AICrowd codebase, explained

The gridworlds used in this competition are all 2D "terrains" of some kind. Each type of gridworld is called a "substrate", and there are four. Let us focus on the first one, "allelopathic_harvest__open": https://github.com/rstrivedi/Melting-Pot-Contest-2023/blob/main/meltingpot/configs/substrates/allelopathic_harvest.py From the description, it is basically a situation where agents are rewarded most with just one type of many types of berries, and are tasked with choosing which berries to plant where, with the failure modes of misjudging the relative values of berries and free-riding by eating without planting.

The overall codebase is at https://github.com/rstrivedi/Melting-Pot-Contest-2023/tree/main

[ ]:
```
# Copyright 2022 DeepMind Technologies Limited.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
"""Configuration for Allelopathic Harvest.
```

```
This substrate contains three different varieties of berry (red, green, & blue)
and a fixed number of berry patches, which could be replanted to grow any color
variety of berry. The growth rate of each berry variety depends linearly on the
fraction that that color comprises of the total. Players have three planting
actions with which they can replant berries in their chosen color. All players
prefer to eat red berries (reward of 2 per red berry they eat versus a reward
of 1 per other colored berry). Players can achieve higher return by selecting
just one single color of berry to plant, but which one to pick is, in principle,
difficult to coordinate (start-up problem) -- though in this case all prefer
red berries, suggesting a globally rational chioce. They also always prefer to
eat berries over spending time planting (free-rider problem).

Allelopathic Harvest was first described in Koster et al. (2020).

Köster, R., McKee, K.R., Everett, R., Weidinger, L., Isaac, W.S., Hughes, E.,
Duenez-Guzman, E.A., Graepel, T., Botvinick, M. and Leibo, J.Z., 2020.
Model-free conventions in multi-agent reinforcement learning with heterogeneous
preferences. arXiv preprint arXiv:2010.09054.
"""

from typing import Any, Dict, Mapping, Sequence

from meltingpot.utils.substrates import colors
from meltingpot.utils.substrates import game_object_utils
from meltingpot.utils.substrates import shapes
from meltingpot.utils.substrates import specs
from ml_collections import config_dict
```

The code above, the first lines of the file, seem quite boilerplate: they import classes that provide distinguishing detail in the environment. We might think that meltingpot/utils/substrates itself deserves a closer look. I've identified game_object_utils.py within it as particularly useful: it defineds the notion of position (2d integer grid), orientation (compass rose), and the methods to get lists of various named components. More important are such general methods as getting game object positions (and orientations) from the given map, getting game objects themselves from maps, and most of all, creating objects. All at https://github.com/rstrivedi/Melting-Pot-Contest-2023/blob/main/meltingpot/utils/substrates/game_object_utils.py

[ ]:
```
PrefabConfig = game_object_utils.PrefabConfig

# Warning: setting `_ENABLE_DEBUG_OBSERVATIONS = True` may cause slowdown.
_ENABLE_DEBUG_OBSERVATIONS = False

# How many different colors of berries.
NUM_BERRY_TYPES = 3

DEFAULT_ASCII_MAP = """
```

```
333PPPP12PPP322P32PPP1P13P3P3
1PPPP2PP122PPP3P232121P2PP2P1
P1P3P11PPP13PPP31PPPP23PPPPPP
PPPPP2P2P1P2P3P33P23PP2P2PPPP
P1PPPPPPP2PPP12311PP3321PPPPP
133P2PP2PPP3PPP1PPP2213P112P1
3PPPPPPPPPPPPP31PPPPPP1P3112P
PP2P21P21P33PPPPPPP3PP2PPPP1P
PPPPP1P1P32P3PPP22PP1P2PPPP2P
PPP3PP3122211PPP2113P3PPP1332
PP12132PP1PP1P321PP1PPPPPP1P3
PPP222P12PPPP1PPPP1PPP321P11P
PPP2PPPP3P2P1PPP1P23322PP1P13
23PPP2PPPP2P3PPPP3PP3PPP3PPP2
2PPPP3P3P3PP3PP3P1P3PP11P21P1
21PPP2PP331PP3PPP2PPPPP2PP3PP
P32P2PP2P1PPPPPPP12P2PPP1PPPP
P3PP3P2P21P3PP2PP11PP1323P312
2P1PPPPP1PPP1P2PPP3P32P2P331P
PPPPP1312P3P2PPPP3P32PPPP2P11
P3PPPP221PPP2PPPPPPPP1PPP311P
32P3PPPPPPPPPP31PPPP3PPP13PPP
PPP3PPPPP3PPPPPP232P13PPPPP1P
P1PP1PPP2PP3PPPPP33321PP2P3PP
P13PPPP1P333PPPP2PP213PP2P3PP
1PPPPP3PP2P1PP21P3PPPP231P2PP
1331P2P12P2PPPP2PPP3P23P21PPP
P3P131P3PPP13P1PPP222PPPP11PP
2P3PPPPPPPP2P323PPP2PPP1PPP2P
21PPPPPPP12P23P1PPPPPP13P3P11
"""
SPRITE_SIZE = 8
```

A PrefabConfig is initialized. But what is it? Going back to game_object_utils.py, we see that it's a "recursive string mapping"…that is, an abstract class (native to python) that supports associating certain key-value pairs. We see there are 3 berry types, and a default map of the gridworld initialized, with its default size of sprites. The particular characters in the map each represent what type of thing that position holds…the characters being defined later. Sprites aren't themselves actors, but different "textures" or types of surfaces given a single position.

[1]:
```
# Map a character to the prefab it represents in the ASCII map.
CHAR_PREFAB_MAP = {
    "P": {"type": "all", "list": ["floor", "spawn_point"]},
    "W": "wall",
    "1": {"type": "all", "list": ["soil", "berry_1"]},
    "2": {"type": "all", "list": ["soil", "berry_2"]},
    "3": {"type": "all", "list": ["soil", "berry_3"]},
```

```
}


# These need to be orthogonal, same intensity and variance.
COLORS = [
    (200, 10, 10, 255),  # 'Red'
    (10, 200, 10, 255),  # 'Green'
    (10, 10, 200, 255),  # 'Blue'
]

ROLE_TO_MOST_TASTY_BERRY_IDX = {
    "player_who_likes_red": 0,
    "player_who_likes_green": 1,
    "player_who_likes_blue": 2,
}

MARKING_SPRITE = """
oxxxxxxo
xoxxxxox
xxoxxoxx
xxxooxxx
xxxooxxx
xxoxxoxx
xoxxxxox
oxxxxxxo
"""
```

What is the role of ROLE_TO_MOST_TASTY_BERRY_IDX? It's a fairly straightforward mapping between a qualitative attribute and its corresponding signifying index.

[ ]:
```
def get_marking_palette(alpha: float) -> Dict[str, Sequence[int]]:
  alpha_uint8 = int(alpha * 255)
  assert alpha_uint8 >= 0.0 and alpha_uint8 <= 255, "Color value out of range."
  return {"x": shapes.ALPHA, "o": (0, 0, 0, alpha_uint8)}

_NUM_DIRECTIONS = 4  # NESW

FLOOR = {
    "name": "floor",
    "components": [
        {
            "component": "StateManager",
            "kwargs": {
                "initialState": "floor",
                "stateConfigs": [{
                    "state": "floor",
                    "layer": "background",
```

```
                    "sprite": "Floor",
                }],
            }
        },
        {
            "component": "Appearance",
            "kwargs": {
                "renderMode": "ascii_shape",
                "spriteNames": ["Floor",],
                "spriteShapes": [shapes.DIRT_PATTERN],
                "palettes": [{
                    "x": (55, 55, 55, 255),
                    "X": (60, 60, 60, 255),
                }],
                "noRotates": [True]
            }
        },
        {
            "component": "Transform",
        },
    ]
}

SOIL = {
    "name": "soil",
    "components": [
        {
            "component": "StateManager",
            "kwargs": {
                "initialState": "soil",
                "stateConfigs": [{
                    "state": "soil",
                    "layer": "background",
                    "sprite": "Soil",
                }],
            }
        },
        {
            "component": "Appearance",
            "kwargs": {
                "renderMode": "ascii_shape",
                "spriteNames": ["Soil",],
                "spriteShapes": [shapes.SOIL],
                "palettes": [{
                    "D": (40, 40, 40, 255),
                    "d": (50, 50, 50, 255),
                    "X": (60, 60, 60, 255),
```

```
                    "x": (70, 70, 70, 255)}],
                "noRotates": [False]
            }
        },
        {
            "component": "Transform",
        },
    ]
}

WALL = {
    "name": "wall",
    "components": [
        {
            "component": "StateManager",
            "kwargs": {
                "initialState": "wall",
                "stateConfigs": [{
                    "state": "wall",
                    "layer": "upperPhysical",
                    "sprite": "Wall",
                }],
            }
        },
        {
            "component": "Transform",
        },
        {
            "component": "Appearance",
            "kwargs": {
                "spriteNames": ["Wall"],
                # This color is a dark shade of grey.
                "spriteRGBColors": [(40, 40, 40)]
            }
        },
        {
            "component": "BeamBlocker",
            "kwargs": {
                "beamType": "zapHit"
            }
        },
        {
            "component": "BeamBlocker",
            "kwargs": {
                "beamType": "fire_1"
            }
        },
```

```
        {
            "component": "BeamBlocker",
            "kwargs": {
                "beamType": "fire_2"
            }
        },
        {
            "component": "BeamBlocker",
            "kwargs": {
                "beamType": "fire_3"
            }
        },
    ]
}

SPAWN_POINT = {
    "name": "spawnPoint",
    "components": [
        {
            "component": "StateManager",
            "kwargs": {
                "initialState": "spawnPoint",
                "stateConfigs": [{
                    "state": "spawnPoint",
                    "layer": "logic",
                    "groups": ["spawnPoints"]
                }],
            }
        },
        {
            "component": "Transform",
        },
    ]
}
```

The point of the get_marking_palette function is somewhat bureaucratic rather than mathematical: it defines the properties of the essential boundaries of the gridworld, including their appearances.

[ ]:
```
def create_berry_prefab(lua_index: int):
  """Return a berry prefab for the given color index (initial color)."""
  berry = {
      "name": "berry",
      "components": [
          {
              "component": "StateManager",
              "kwargs": {
```

```
                "initialState": "unripe_{}".format(lua_index),
                "stateConfigs": [
                    # Unripe states.
                    {
                        "state": "unripe_1",
                        "layer": "lowerPhysical",
                        "sprite": "UnripeBerry_1",
                        "groups": ["unripes"]
                    },
                    {
                        "state": "unripe_2",
                        "layer": "lowerPhysical",
                        "sprite": "UnripeBerry_2",
                        "groups": ["unripes"]
                    },
                    {
                        "state": "unripe_3",
                        "layer": "lowerPhysical",
                        "sprite": "UnripeBerry_3",
                        "groups": ["unripes"]
                    },
                    # Ripe states.
                    {
                        "state": "ripe_1",
                        "layer": "lowerPhysical",
                        "sprite": "RipeBerry_1",
                        "groups": []
                    },
                    {
                        "state": "ripe_2",
                        "layer": "lowerPhysical",
                        "sprite": "RipeBerry_2",
                        "groups": []
                    },
                    {
                        "state": "ripe_3",
                        "layer": "lowerPhysical",
                        "sprite": "RipeBerry_3",
                        "groups": []
                    },
                ]
            }
        },
        {
            "component": "Transform",
        },
        {
```

```
                "component": "Appearance",
                "kwargs": {
                    "renderMode": "ascii_shape",
                    "spriteNames": [
                        "UnripeBerry_1",
                        "UnripeBerry_2",
                        "UnripeBerry_3",
                        "RipeBerry_1",
                        "RipeBerry_2",
                        "RipeBerry_3",
                    ],
                    "spriteShapes": [
                        shapes.BERRY_SEEDS,
                        shapes.BERRY_SEEDS,
                        shapes.BERRY_SEEDS,
                        shapes.BERRY_RIPE,
                        shapes.BERRY_RIPE,
                        shapes.BERRY_RIPE,
                    ],
                    "palettes": [
                        # Unripe colors
                        {
                            "o": COLORS[0],
                            "O": shapes.scale_color(COLORS[0], 1.5),
                            "x": (0, 0, 0, 0)
                        },
                        {
                            "o": COLORS[1],
                            "O": shapes.scale_color(COLORS[1], 1.5),
                            "x": (0, 0, 0, 0)
                        },
                        {
                            "o": COLORS[2],
                            "O": shapes.scale_color(COLORS[2], 1.5),
                            "x": (0, 0, 0, 0)
                        },
                        # Ripe colors
                        {
                            "d": COLORS[0],
                            "O": shapes.scale_color(COLORS[0], 1.5),
                            "o": shapes.scale_color(COLORS[0], 1.25),
                            "x": (0, 0, 0, 0)
                        },
                        {
                            "d": COLORS[1],
                            "O": shapes.scale_color(COLORS[1], 1.5),
                            "o": shapes.scale_color(COLORS[1], 1.25),
```

```
                    "x": (0, 0, 0, 0)
                },
                {
                    "d": COLORS[2],
                    "O": shapes.scale_color(COLORS[2], 1.5),
                    "o": shapes.scale_color(COLORS[2], 1.25),
                    "x": (0, 0, 0, 0)
                },
            ],
            # Note: the berries do not rotate in this version (unlike in
            # the original allelopathic_harvest version, where they do).
            "noRotates": [True] * (NUM_BERRY_TYPES * 2)
        }
    },
    {
        "component": "Berry",
        "kwargs": {
            "unripePrefix": "unripe",
            "ripePrefix": "ripe",
            "colorId": lua_index,
        }
    },
    {
        "component": "Edible",
        "kwargs": {
            "name": "Edible",
            "eatingSetsColorToNewborn": True,
        }
    },
    {
        "component": "Regrowth",
        "kwargs": {
            "minimumTimeToRipen": 10,
            "baseRate": 5e-6,
            "linearGrowth": True,
        }
    },
    {
        "component": "Coloring",
        "kwargs": {
            "numColors": NUM_BERRY_TYPES,
        }
    },
  ]
}
return berry
```

Obviously, this is the berry-instantiation module, and we begin to get a better sense of what a "prefab" concept is: an abstraction layer, that takes in humanlike textual properties and associates them with abstract machine-parseable ones. Still its first half is a fairly bureaucratic defining of properties (three ripe and three unripe states), though it does include some fairly important numerical details about regrowth.

[ ]:
```python
def create_avatar_object(player_idx: int,
                         most_tasty_berry_idx: int) -> Dict[str, Any]:
  """Return the avatar for the player numbered `player_idx`."""
  # Lua is 1-indexed.
  lua_index = player_idx + 1

  lua_most_tasty_berry_idx = most_tasty_berry_idx + 1

  live_state_name = "player{}".format(lua_index)
  avatar_sprite_name = "avatarSprite{}".format(lua_index)
  avatar_object = {
      "name": "avatar",
      "components": [
          {
              "component": "StateManager",
              "kwargs": {
                  "initialState": live_state_name,
                  "stateConfigs": [
                      # Initial player state.
                      {"state": live_state_name,
                       "layer": "upperPhysical",
                       "sprite": avatar_sprite_name,
                       "contact": "avatar",
                       "groups": ["players"]},

                      # Player wait type for when they have been zapped out.
                      {"state": "playerWait",
                       "groups": ["playerWaits"]},
                  ]
              }
          },
          {
              "component": "Transform",
          },
          {
              "component": "Appearance",
              "kwargs": {
                  "renderMode": "ascii_shape",
                  "spriteNames": [avatar_sprite_name],
                  "spriteShapes": [shapes.CUTE_AVATAR],
                  # This color is white. It should never appear in gameplay. So
```

```
                # if a white colored avatar does appear then something is
                # broken.
                "palettes": [shapes.get_palette((255, 255, 255))],
                "noRotates": [True]
            }
        },
        {
            "component": "Avatar",
            "kwargs": {
                "index": lua_index,
                "aliveState": live_state_name,
                "waitState": "playerWait",
                "speed": 1.0,
                "spawnGroup": "spawnPoints",
                "actionOrder": ["move",
                                "turn",
                                "fireZap",
                                "fire_1",
                                "fire_2",
                                "fire_3"],
                "actionSpec": {
                    "move": {"default": 0, "min": 0, "max": _NUM_DIRECTIONS},
                    "turn": {"default": 0, "min": -1, "max": 1},
                    "fireZap": {"default": 0, "min": 0, "max": 1},
                    "fire_1": {"default": 0, "min": 0, "max": 1},
                    "fire_2": {"default": 0, "min": 0, "max": 1},
                    "fire_3": {"default": 0, "min": 0, "max": 1},
                },
                "view": {
                    "left": 5,
                    "right": 5,
                    "forward": 9,
                    "backward": 1,
                    "centered": False
                },
            }
        },
        {
            "component": "Zapper",
            "kwargs": {
                "cooldownTime": 4,
                "beamLength": 3,
                "beamRadius": 1,
                "beamColor": (253, 253, 253),  # the zapper beam is white.
                "framesTillRespawn": 25,
                "penaltyForBeingZapped": 0,  # leave this always at 0.
                "rewardForZapping": 0,  # leave this always at 0.
```

```
                # GraduatedSanctionsMarking handles removal instead of Zapper.
                "removeHitPlayer": False,
            }
        },
        {
            "component": "ReadyToShootObservation",
        },
        {
            "component": "Taste",
            "kwargs": {
                "mostTastyBerryId": lua_most_tasty_berry_idx,
                "rewardMostTasty": 2,
            }
        },
        {
            "component": "ColorZapper",
            "kwargs": {
                "cooldownTime": 2,
                "beamLength": 3,
                "beamRadius": 0,
                "numColorZappers": NUM_BERRY_TYPES,
                "beamColors": COLORS,
                # When `eatingSetsColorToNewborn` and `stochasticallyCryptic`
                # are both true than stochastically change back to the
                # newborn color after eating a berry with probability
                # inversely related to the monoculture fraction. So larger
                # monoculture fractions yield lower probabilities of changing
                # back to the newborn color.
                "stochasticallyCryptic": True,
            }
        },
    ]
}
if _ENABLE_DEBUG_OBSERVATIONS:
  avatar_object["components"].append({
      "component": "LocationObserver",
      "kwargs": {"objectIsAvatar": True, "alsoReportOrientation": True},
  })
  avatar_object["components"].append({
      "component": "AvatarMetricReporter",
      "kwargs": {
          "metrics": [
              {
                  "name": "COLOR_ID",
                  "type": "Doubles",
                  "shape": [],
                  "component": "ColorZapper",
```

```
                    "variable": "colorId",
                },
                {
                    "name": "MOST_TASTY_BERRY_ID",
                    "type": "Doubles",
                    "shape": [],
                    "component": "Taste",
                    "variable": "mostTastyBerryId",
                },
            ]
        },
    })
    avatar_object["components"].append({
        "component": "AvatarIdsInViewObservation",
    })
    avatar_object["components"].append({
        "component": "AvatarIdsInRangeToZapObservation",
    })
  return avatar_object
```

From this instantiation method for avatars (which are basically the identity of the player, which we can deduce is ultimately identified by an integer), we can see interesting properties defined: the idea that berries can change back to the same or different colors, randomly, after being eaten, the idea that each player has a different range of view in different direction (after all, this is a POMDP), and the use of white as a color denoting errors/bugs. Note that zapping itself (unlike color zapping) seems to be implied to not exist here with its zero penalty/reward.

[ ]:
```
# PREFABS is a dictionary mapping names to template game objects that can
# be cloned and placed in multiple locations accoring to an ascii map.
PREFABS = {
    "floor": FLOOR,
    "soil": SOIL,
    "wall": WALL,
    "spawn_point": SPAWN_POINT,
    "berry_1": create_berry_prefab(1),
    "berry_2": create_berry_prefab(2),
    "berry_3": create_berry_prefab(3),
}

# PLAYER_COLOR_PALETTES is a list with each entry specifying the color to use
# for the player at the corresponding index.
NUM_PLAYERS_UPPER_BOUND = 60
PLAYER_COLOR_PALETTES = []
for i in range(NUM_PLAYERS_UPPER_BOUND):
  PLAYER_COLOR_PALETTES.append(shapes.get_palette(colors.palette[i]))

# Primitive action components.
```

```
# pylint: disable=bad-whitespace
# pyformat: disable
NOOP       = {"move": 0, "turn":  0, "fireZap": 0, "fire_1": 0, "fire_2": 0, "fire_3": 0}
FORWARD    = {"move": 1, "turn":  0, "fireZap": 0, "fire_1": 0, "fire_2": 0, "fire_3": 0}
STEP_RIGHT = {"move": 2, "turn":  0, "fireZap": 0, "fire_1": 0, "fire_2": 0, "fire_3": 0}
BACKWARD   = {"move": 3, "turn":  0, "fireZap": 0, "fire_1": 0, "fire_2": 0, "fire_3": 0}
STEP_LEFT  = {"move": 4, "turn":  0, "fireZap": 0, "fire_1": 0, "fire_2": 0, "fire_3": 0}
TURN_LEFT  = {"move": 0, "turn": -1, "fireZap": 0, "fire_1": 0, "fire_2": 0, "fire_3": 0}
TURN_RIGHT = {"move": 0, "turn":  1, "fireZap": 0, "fire_1": 0, "fire_2": 0, "fire_3": 0}
FIRE_ZAP   = {"move": 0, "turn":  0, "fireZap": 1, "fire_1": 0, "fire_2": 0, "fire_3": 0}
FIRE_ONE   = {"move": 0, "turn":  0, "fireZap": 0, "fire_1": 1, "fire_2": 0, "fire_3": 0}
FIRE_TWO   = {"move": 0, "turn":  0, "fireZap": 0, "fire_1": 0, "fire_2": 1, "fire_3": 0}
FIRE_THREE = {"move": 0, "turn":  0, "fireZap": 0, "fire_1": 0, "fire_2": 0, "fire_3": 1}
# pyformat: enable
# pylint: enable=bad-whitespace

ACTION_SET = (
    NOOP,
    FORWARD,
    BACKWARD,
    STEP_LEFT,
    STEP_RIGHT,
    TURN_LEFT,
    TURN_RIGHT,
    FIRE_ZAP,
    FIRE_ONE,
    FIRE_TWO,
    FIRE_THREE,
)
```

The prefabs' goal as abstraction layers is clear here. PLAYER_COLOR_PALETTES is a bureaucratic identification of colors to players. The action set covers all the basic forward, rotational, and mixed motions, plus firing motions (though I can't quite figure out what that has to do with either Zapper or ColorZapper). Note that the dictionary structure above means that from a given qualitative command, you can isolate exact numerical values of movement along certain dimensions.

[ ]:

```python
# The Scene objece is a non-physical object, it components implement global
# logic. In this case, that includes holding the global berry counters to
# implement the regrowth rate, as well as some of the observations.
def create_scene(num_players: int):
  """Creates the global scene."""
  scene = {
      "name": "scene",
      "components": [
          {
              "component": "StateManager",
              "kwargs": {
                  "initialState": "scene",
                  "stateConfigs": [{
                      "state": "scene",
                  }],
              }
          },
          {
              "component": "Transform",
          },
          {
              "component": "GlobalBerryTracker",
              "kwargs": {
                  "numBerryTypes": NUM_BERRY_TYPES,
                  "numPlayers": num_players,
              }
          },
          {
              "component": "GlobalZapTracker",
              "kwargs": {
                  "numBerryTypes": NUM_BERRY_TYPES,
                  "numPlayers": num_players,
              }
          },
          {
              "component": "GlobalMetricHolder",
              "kwargs": {
                  "metrics": [
                      {"type": "tensor.Int32Tensor",
                       "shape": (num_players, num_players),
                       "variable": "playerZapMatrix"},
                  ]
              }
          },
      ]
  }
  if _ENABLE_DEBUG_OBSERVATIONS:
```

```
scene["components"].append({
    "component": "GlobalMetricReporter",
    "kwargs": {
        "metrics": [
            {
                "name": "RIPE_BERRIES_BY_TYPE",
                "type": "tensor.Int32Tensor",
                "shape": (NUM_BERRY_TYPES,),
                "component": "GlobalBerryTracker",
                "variable": "ripeBerriesPerType",
            },
            {
                "name": "UNRIPE_BERRIES_BY_TYPE",
                "type": "tensor.Int32Tensor",
                "shape": (NUM_BERRY_TYPES,),
                "component": "GlobalBerryTracker",
                "variable": "unripeBerriesPerType",
            },
            {
                "name": "BERRIES_BY_TYPE",
                "type": "tensor.Int32Tensor",
                "shape": (NUM_BERRY_TYPES,),
                "component": "GlobalBerryTracker",
                "variable": "berriesPerType",
            },
            {
                "name": "COLORING_BY_PLAYER",
                "type": "tensor.Int32Tensor",
                "shape": (NUM_BERRY_TYPES, num_players),
                "component": "GlobalBerryTracker",
                "variable": "coloringByPlayerMatrix",
            },
            {
                "name": "EATING_TYPES_BY_PLAYER",
                "type": "tensor.Int32Tensor",
                "shape": (NUM_BERRY_TYPES, num_players),
                "component": "GlobalBerryTracker",
                "variable": "eatingTypesByPlayerMatrix",
            },
            {
                "name": "BERRIES_PER_TYPE_BY_COLOR_OF_COLORER",
                "type": "tensor.Int32Tensor",
                "shape": (NUM_BERRY_TYPES, NUM_BERRY_TYPES + 1),
                "component": "GlobalBerryTracker",
                "variable": "berryTypesByColorOfColorer",
            },
            {
```

```
            "name": "BERRIES_PER_TYPE_BY_TASTE_OF_COLORER",
            "type": "tensor.Int32Tensor",
            "shape": (NUM_BERRY_TYPES, NUM_BERRY_TYPES),
            "component": "GlobalBerryTracker",
            "variable": "berryTypesByTasteOfColorer",
        },
        {
            "name": "PLAYER_TIMEOUT_COUNT",
            "type": "tensor.Int32Tensor",
            "shape": (num_players, num_players),
            "component": "GlobalZapTracker",
            "variable": "fullZapCountMatrix",
        },
        {
            "name": "COLOR_BY_COLOR_ZAP_COUNTS",
            "type": "tensor.Int32Tensor",
            "shape": (NUM_BERRY_TYPES + 1, NUM_BERRY_TYPES + 1),
            "component": "GlobalZapTracker",
            "variable": "colorByColorZapCounts",
        },
        {
            "name": "COLOR_BY_TASTE_ZAP_COUNTS",
            "type": "tensor.Int32Tensor",
            "shape": (NUM_BERRY_TYPES + 1, NUM_BERRY_TYPES),
            "component": "GlobalZapTracker",
            "variable": "colorByTasteZapCounts",
        },
        {
            "name": "TASTE_BY_TASTE_ZAP_COUNTS",
            "type": "tensor.Int32Tensor",
            "shape": (NUM_BERRY_TYPES, NUM_BERRY_TYPES),
            "component": "GlobalZapTracker",
            "variable": "tasteByTasteZapCounts",
        },
        {
            "name": "TASTE_BY_COLOR_ZAP_COUNTS",
            "type": "tensor.Int32Tensor",
            "shape": (NUM_BERRY_TYPES, NUM_BERRY_TYPES + 1),
            "component": "GlobalZapTracker",
            "variable": "tasteByColorZapCounts",
        },
        {
            "name": "WHO_ZAPPED_WHO",
            "type": "tensor.Int32Tensor",
            "shape": (num_players, num_players),
            "component": "GlobalMetricHolder",
            "variable": "playerZapMatrix",
```

```
                },
            ]
        }
    })
return scene
```

Obviously, this instantiates the global environment…teaching us that if only we enabled the flag for debugging, we could access tensors for a wide range of statistics across different types of berries and the different players.