Creating New Env in Multigrid

Step 1: Create new world

agents.py

- If existing action classes don't suffice, add a new world's action class

constants.py

- If there are object-specific states, define a state_to_idx_{yourChosenName} dictionary
- If your env requires objects not yet defined, add an entry for the new object to the dictionary OBJECT_TO_STR

object.py

- Define a class for the new object(s), if any
- Objects require: type, colour, position, encode, decode, render
- Object attributes are defined through: can_overlap, can_pickup, can_contain, see_behind
- Objects can have: contains, toggle

world.py

- Add a new world if existing worlds don't suffice
- A world defines the objects, colours, and encoding size
- Note: Encode layers are to capture different things. Layer 1 is for the type of object in the cell, layer 2 is for color, and 3 is for agent orientation. We haven't used 4, 5, and 6 but can be used for more characteristics if needed.

grid.py

- This is the underlying structure of the new env
- Methods include: copy, get, set, rotate, slice, render, encode
- Also have: horz_wall, vert_wall, wall_rect

Step 2: Create env

Create a file {yourChosenName}.py at gym_multigrid/gym_multigrid/envs/. Write a class for your environment that inherits from MultigridEnv.

__init__

When calling super's init, you should specify:

- the list of agents
- grid dimensions
- whether you are using full or partial observability
- the number of timesteps per episode
- the actions and world classes defined above

```
class MultiGridEnv(gym.Env):
    """
    2D grid world game environment
    """

    metadata = {"render_modes": ["human", "rgb_array"], "video.frames_per_second": 10}

    def __init__(
        self,
        agents: list[AgentT],
        grid_size: int | None = None,
        width: int | None = None,
        height: int | None = None,
        max_steps: int = 100,
        see_through_walls: bool = False,
        partial_obs: bool = False,
        agent_view_size: int = 7,
        actions_set: Type[ActionsT] = DefaultActions,
        world: WorldT = DefaultWorld,
        render_mode: Literal["human", "rgb_array"] = "rgb_array",
        uncached_object_types: list[str] = [],
    ):
```

You likely will want to initialize/define other private variables relevant to your env. For example, in the collect game we need to keep track of the following:

```
class CollectGameEnv(MultiGridEnv):
    def __init__(
        self.num_balls = num_balls
        self.collected_balls = 0
```

_gen_grid

You *must* implement this method as it is not defined by the MultiGridEnv parent class. This method gets called by default during env.reset(). Here, you need to place all objects and agents present in the gridworld.

For example, in collect game we define the four boundary walls, place the balls, and then place the agents.

```
class CollectGameEnv(MultiGridEnv):

    def _gen_grid(self, width, height):
        self.grid = Grid(width, height, self.world)

        # Generate the surrounding walls
        self.grid.horz_wall(0, 0)
        self.grid.horz_wall(0, height - 1)
        self.grid.vert_wall(0, 0)
        self.grid.vert_wall(width - 1, 0)

        for number, index, reward in zip(
            self.num_balls, self.balls_index, self.balls_reward
        ):
            for i in range(number):
                self.place_obj(Ball(self.world, index, reward))

        # Randomize the player start position
        for a in self.agents:
            self.place_agent(a)
```

The place_obj() method is defined by the parent class and has the following parameters:

```
class MultiGridEnv(gym.Env):

    def place_obj(
        self,
        obj: WorldObjT,
        top: Position | None = None,
        size: tuple[int, int] | None = None,
        reject_fn: Callable[["MultiGridEnv", NDArray], bool] | None = None,
        max_tries: float = math.inf,
    ):
        """
        Place an object at an empty position in the grid

        :param top: top-left position of the rectangle where to place
        :param size: size of the rectangle where to place
        :param reject_fn: function to filter out potential positions
        """
```

By default, the method tries to place the object in the grid by sampling locations uniformly at random repeatedly until a free grid cell is found.

If you know the coordinates for the object's location, you should instead use this method:

```
class MultiGridEnv(gym.Env):

    def put_obj(self, obj: WorldObjT, i: int, j: int):
```

For placing agents, the above two methods are called as appropriate using this method:

```
class MultiGridEnv(gym.Env):

    def place_agent(
        self,
        agent: AgentT,
        pos: Position | None = None,
        top: Position | None = None,
        size: tuple[int, int] | None = None,
        rand_dir: bool = False,
        max_tries: float = math.inf,
    ) -> Position:
```

_reward

The default reward function defined is as follows:

```
class MultiGridEnv(gym.Env):

    def _reward(self, current_agent, rewards, reward=1):
        """
        Compute the reward to be given upon success
        """
        rewards[current_agent] += reward - 0.9 * (self.step_count / self.max_steps)
```

This gets called when a goal state is reached. current_agent specifies which agent receives the reward.

You should override this if your env has a different reward structure.

step

This method is crucial to the dynamics of your env. You should define this and can call the MultiGridEnv's step method as well if it handles the execution of actions the way you need for your env.

The only required parameter to the step method is the list of actions to execute. By default, these are executed in random order:

```
class MultiGridEnv(gym.Env):
    def step(self, actions):

        order = np.random.permutation(len(actions))
```

If your agent action class has actions that are not movement-related, this is where you would call methods such as _handle_pickup, _handle_build, _handle_drop, _handle_switch, and _handle_special_moves as necessary. None of those methods have a default implementation, so you ***must*** implement them yourself. For example, to collect the ball object:

```
class CollectGameEnv(MultiGridEnv):
    def _handle_pickup(self, i, rewards, fwd_pos, fwd_cell):
        if fwd_cell:
            if fwd_cell.can_pickup():
                fwd_cell.pos = np.array([-1, -1])
                ball_idx = self.world.COLOR_TO_IDX[fwd_cell.color]
                self.grid.set(*fwd_pos, None)
                self.collected_balls += 1
                if ball_idx == 0:
                    self._reward(i, rewards, fwd_cell.reward)
```

The step method is also where the done flag is typically set to true as appropriate. This can be used for early termination:

```
class MultiGridEnv(gym.Env):
    def step(self, actions):
            elif actions[i] == self.actions.forward:
                if fwd_cell is not None:
                    if fwd_cell.type == "goal":
                        done = True
                        rewards = self._reward(i, rewards, 1)
```

or otherwise to end the episode when the maximum number of timesteps has been reached:

```
class MultiGridEnv(gym.Env):
    def step(self, actions):
        if self.step_count >= self.max_steps:
            done = True
```

reset

As with the step method, you should implement one for your env and can call MultiGridEnv's reset method as well since it resets other variables.

For example, in the collect game we reset the number of collected_balls and the info dictionary:

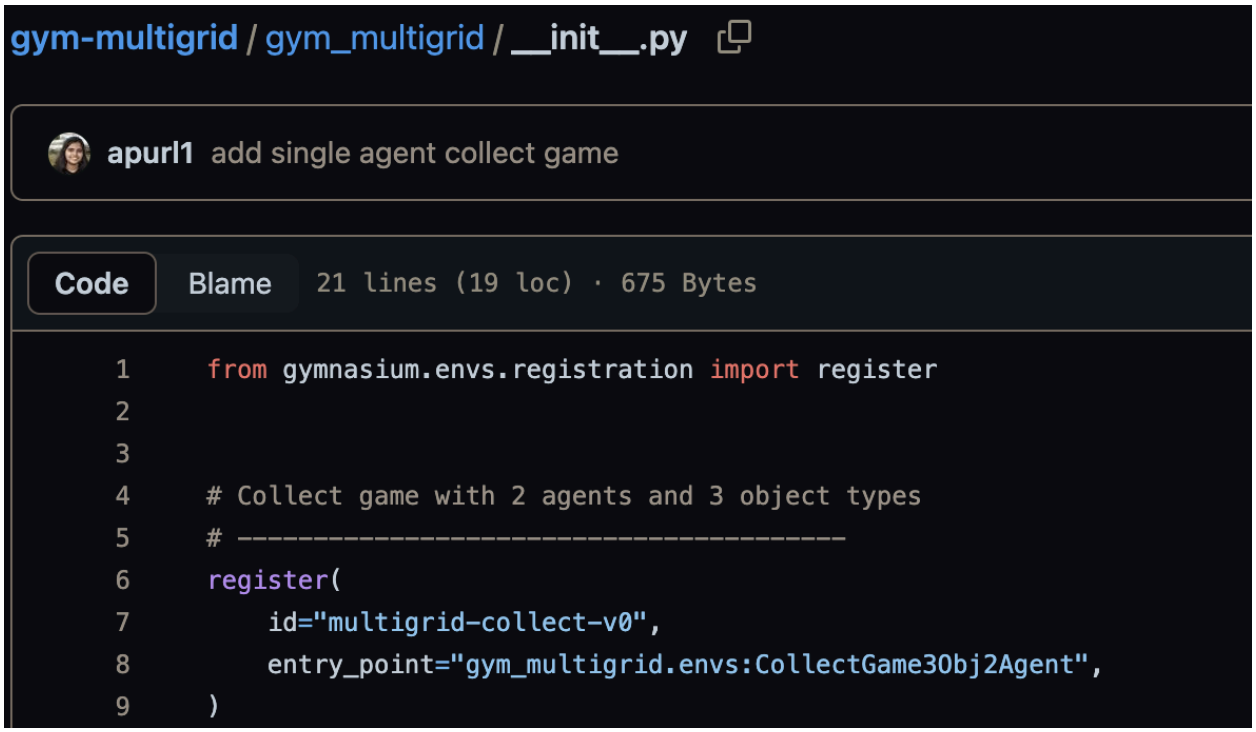
```
def reset(self, seed=None):
    self.collected_balls = 0
    self.info = {
        "agent1ball1": 0,
        "agent1ball2": 0,
        "agent1ball3": 0,
        "agent2ball1": 0,
        "agent2ball2": 0,
        "agent2ball3": 0,
    }
    super().reset()
    state = self.grid.encode()
    return state, {}
```

state/obs encoding

The default grid encoding is a numpy array of shape height x width x encode_dim. The method also accounts for partial observability. You may want to write a method to transform this default encoding into a format that works best for your env and agent algorithm.

Step 3: Register env

Add a line to gym_multigrid/gym_multigrid/__init.py__ to register the newly created env on gymnasium



Step 4: Write tests

It's a good idea to test the new env for bugs continuously as you make updates.

Here is an example of how to write a pytest to run through the env steps until an episode terminates by using random actions.

```
@pytest.mark.parametrize("env_id", ["gym_multigrid:multigrid-collect-v0"])
def test_collect_game(env_id) -> None:
    """Test collect_game()"""
    env = gym.make(env_id)

    obs, _ = env.reset()
    while True:
        action = [env.action_space.sample()]
        obs, reward, terminated, truncated, info = env.step(action)
        if terminated or truncated:
            break
```