Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Puebla

Modelación de sistemas multiagentes con gráficas computacionales - TC2008B.1

Profesores:

Jose Eduardo Ferrer Cruz

Luciano García Bañuelos

Actividad Integradora

Daniel Flores Rodríguez A01734184

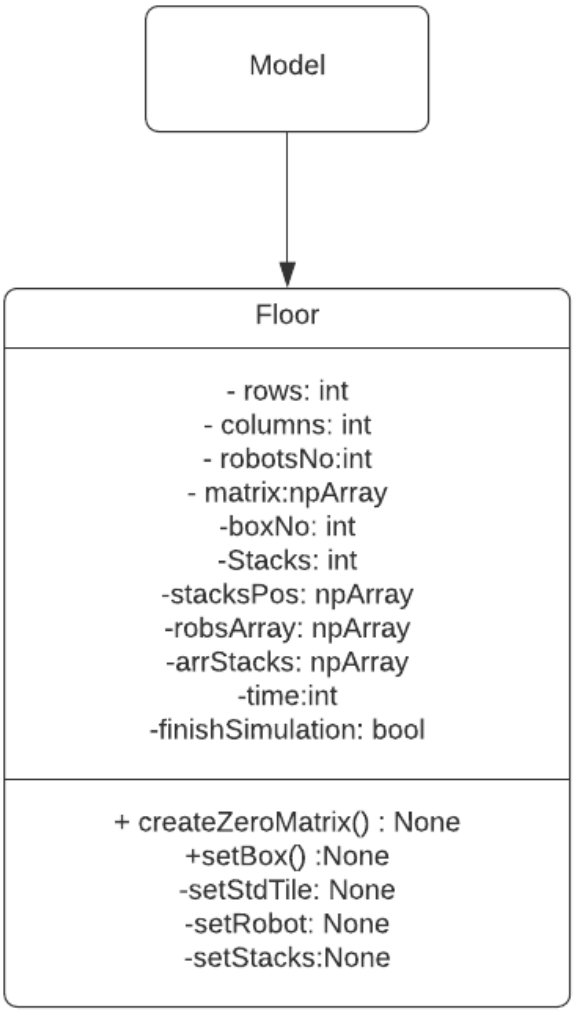
Fecha:

1 de diciembre del 2021

Link repositorio GitHub:

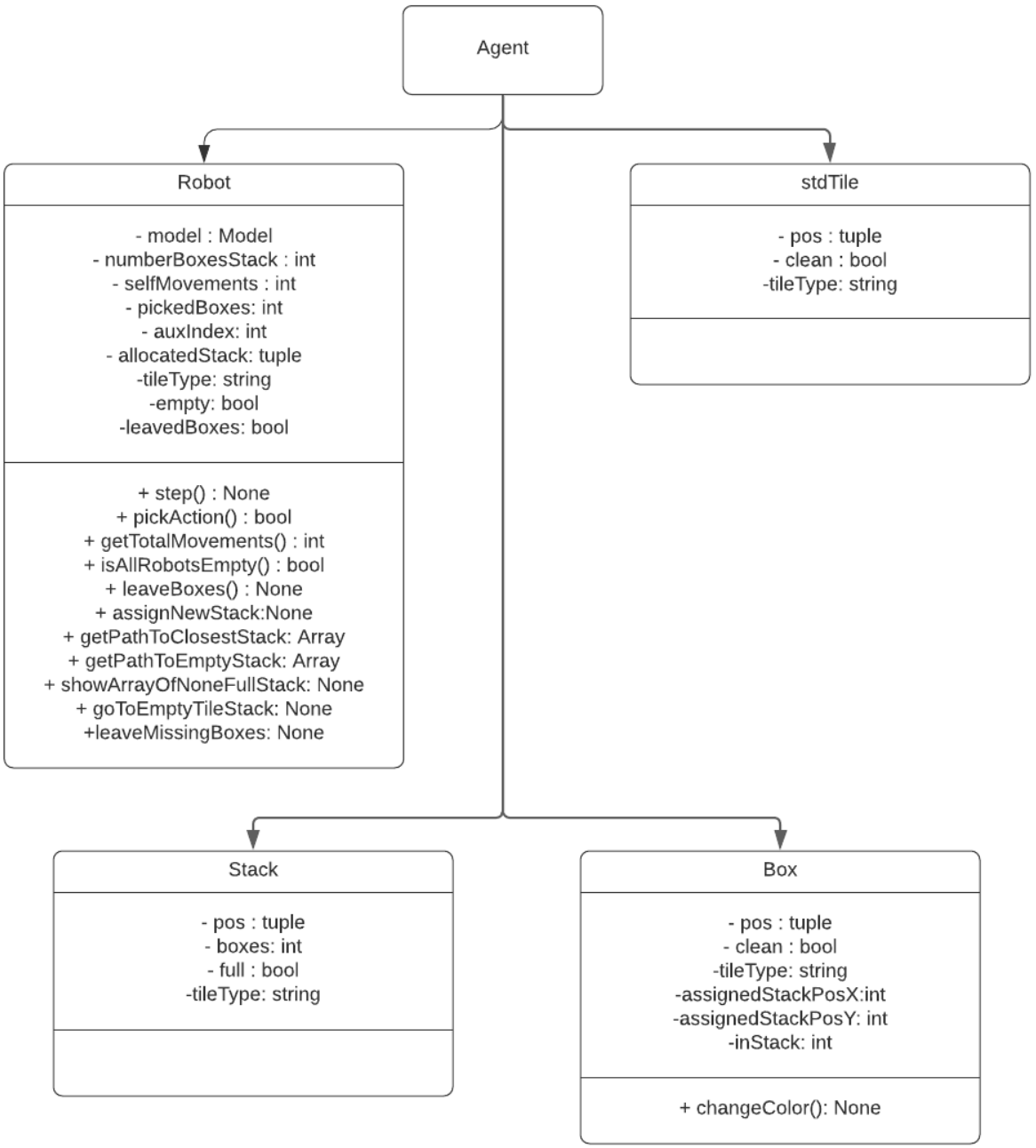
https://github.com/danoman17/PickingRobots_MultiAgent_Simulation/tree/master

Diagramas de clases de los Agentes



En este diagrama podemos observar el los atributos y métodos usados en el modelo de el código de python usando el framework de mesa, podemos ver que todos sus atributos nos son de utilidad para poder acceder a variables o información que podría considerarse global de cierto modo, pues a llamando al método de ".model", podemos acceder a estas en cualquier seccion del codigo

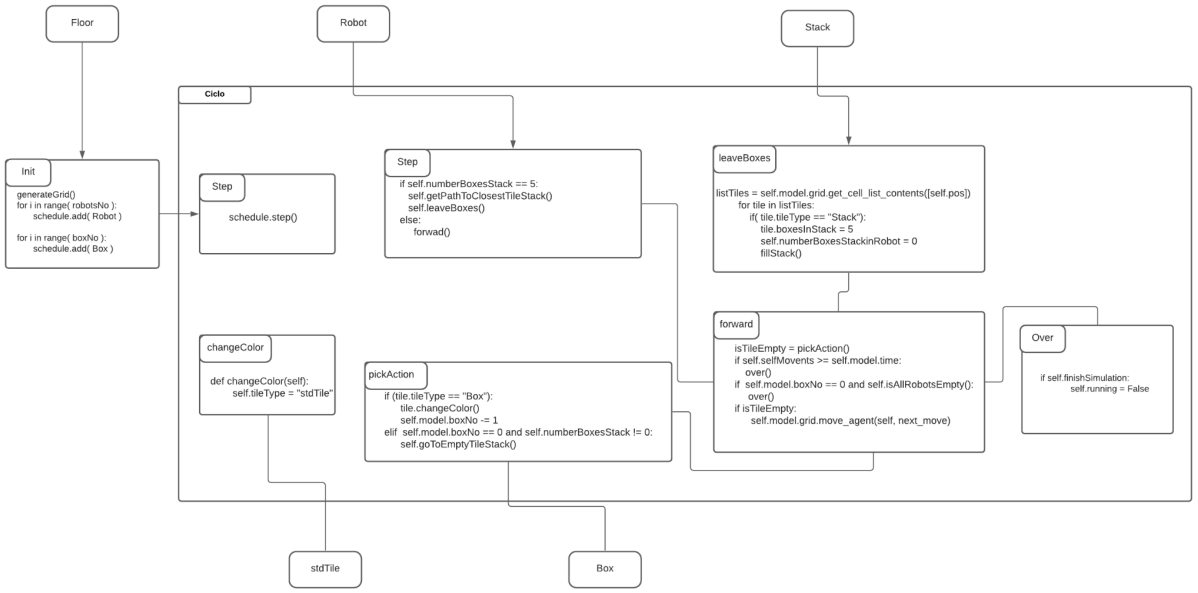
de mesa, por lo tanto, es entendible que aquí se almacenen arrays e indicadores para poder acceder más tarde a ellos desde otras clases de Agentes.



En este caso, podemos observar todas las clases que heredan de la clase de mesa Agent, con lo cual podríamos considerar a todas estas agentes que participaran activamente en la simulación computacional, podemos ver que tenemos 4 tipos de agentes diferentes, las cuales podemos destacar el agente caja (Box), el

agente robot (Robot) y el agente de pilas de cajas (Stack). Todos sus atributos y métodos son mencionados en los diagramas, cada método es usado, principalmente, en la función de step() del agente robot, en donde se podría decir que se concentra la mayor parte de acciones realizadas en cada step por la simulación computacional.

Diagramas de protocolo de Agentes



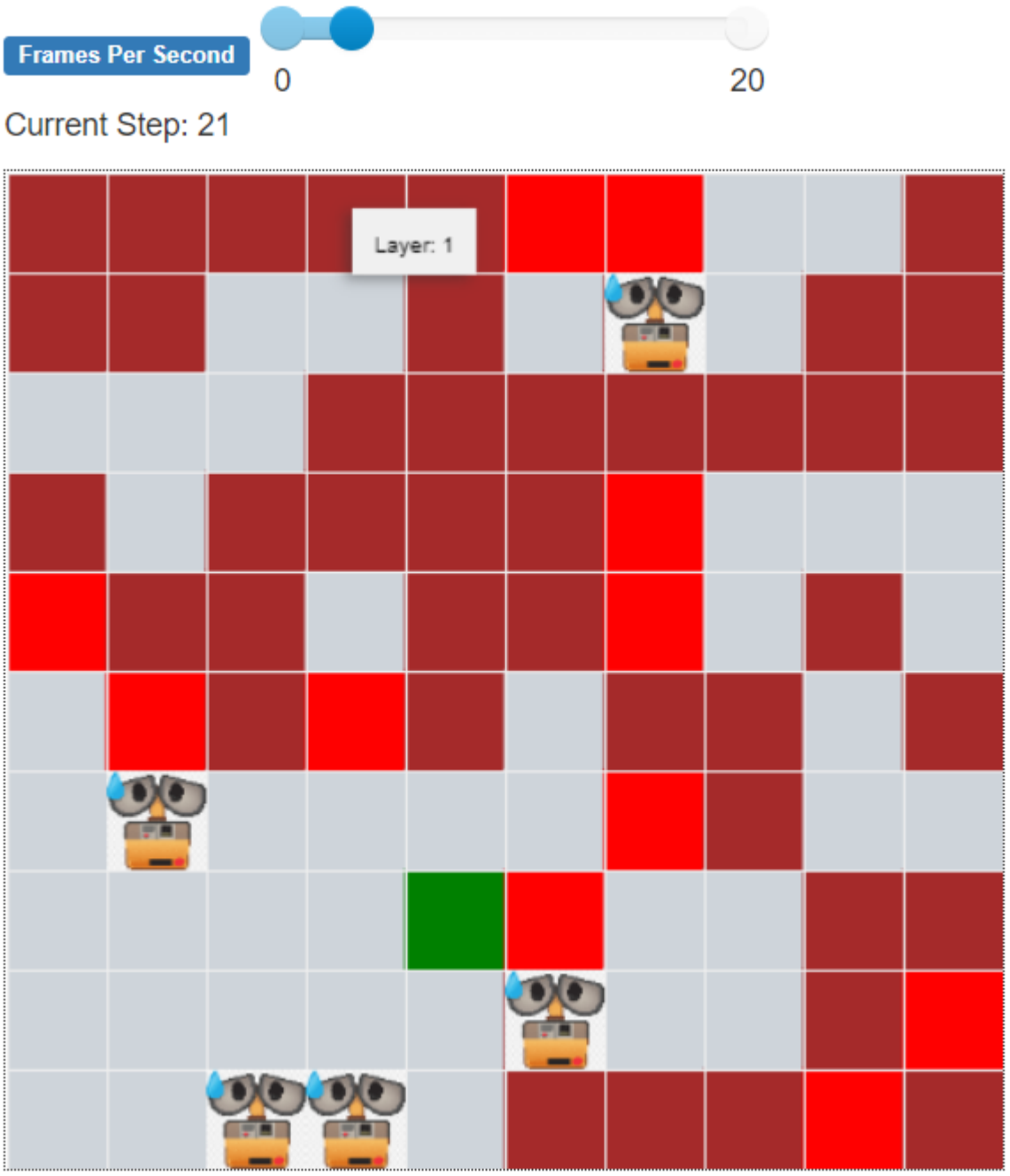
Podemos observar el diagrama de protocolo de los agentes que participan en la simulación, en este se muestran las acciones principales que realizan los agentes al momento de realizar un step() o paso, y como es que la acción del agente "Robot" afecta de cierta manera a los otros agentes.

Estrategia cooperativa para la solución de problema

Como en clases anteriores ya se había realizado algo similar pero con agentes determinados como aspiradoras que limpiaban la suciedad de losetas en un espacio determinado, puede relacionar dicha problemática con esta situacion problema, pues analizando de mejor manera, se nos plantea un escenario

similar, solo que en vez de ser suciedad la que tienen que “limpiar” estos agentes, para esta ocasión tienen que recoger cajas regadas sobre un espacio determinado para después apilarlas.

Para esto, lo que se hizo fue definir 4 tipos de agentes y 1 modelo, el primer agente y el más importante, se trata de un robot , el cual a través de ciertas funciones implementadas, tiene la capacidad de identificar cuando este llega a una posición en donde una caja se encuentra, para posteriormente levantarla y llevarla a un lugar determinado,, cuando esta cantidad de cajas acomuladas en dicho robot es igual a 5. Para realizar esta acción, lo que se realizó fue crear 3 tipos agentes los cuales son de utilidad al momento de identificar por cada sección recorrida si lo que se está ahí es una sección vacía (stdTile), es una sección en donde se encuentra una caja (Box) o bien, es una sección en donde se encuentra una sección para poder apilar cajas que lleva cargando un robot (Stack). Para realizar esto, lo que se tuvo que realizar primero es la programación de la lógica que cada uno de estos agentes iban a obedecer, a traves de el lenguaje de programación Python y el framework “mesa”, derivado de este mismo lenguaje, el cual nos permitió de manera más fácil, programar y definir cada uno de estos agentes mencionados a través de clases, en las cuales se definen cada uno de sus atributos y métodos (acciones) de cada uno de estos, a manera en como se realizaria en la realización/diseño de un videojuego, en donde la mayoría de las cosas son tratadas como un objeto. Al finalizar este código, pudimos probar y aprobar el comportamiento de todos estos agentes para posteriormente, implementar este comportamiento en una simulación computacional en 3D a través de Unity. El resultado de este primer paso fue el siguiente:



Teniendo la lógica definida, el siguiente paso fue realizar una API la cual tendría la tarea de comunicar o conectar el funcionamiento y propiedades del framework “Mesa” con la interfaz de Unity, por lo cual, se decidió hacerla a traves de otro framework llamado flask, el cual nos ayuda a la creación de aplicaciones web de manera fácil y sencilla. En este código, lo que se realizó fue definir la la dirección de peticiones POST y GET, así como aquellos datos

que se ocupan al momento de realizar la simulación en Unity (coordenadas e indicadores de cambio de estado, principalmente), a través de un formato JSON, el cual es sencillo de interpretar y útil al momento de trabajar con información enviada desde otro entorno diferente en el que nos encontramos trabajando.

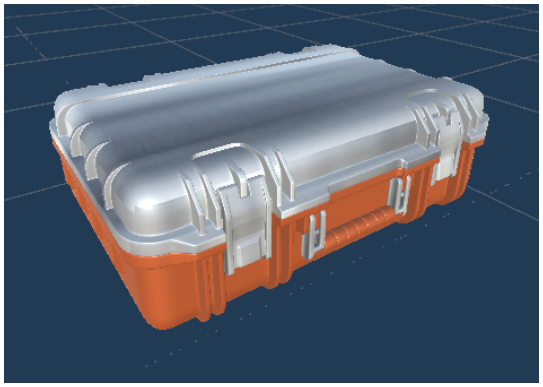
Para este caso, se hizo uso del siguiente código:

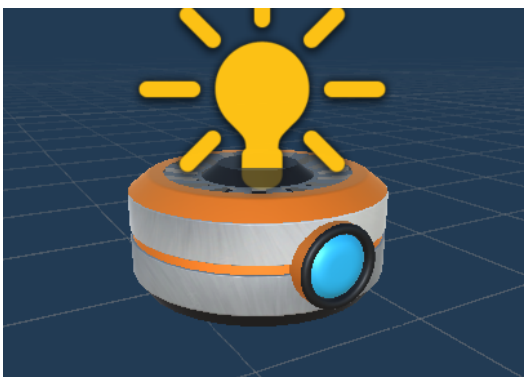
```
import flask
from flask.json import jsonify
import uuid
import robots2
from robots2 import Floor

games = {}

app = flask.Flask(__name__)

@app.route("/mesa", methods=["POST"])
def create():
    global games
    id = str(uuid.uuid4())
    games[id] = Floor() # instanseando el juego con la clase de Model
    print("id: ", id)
    return "ok", 201, {'Location': f"/mesa/{id}"}


@app.route("/mesa/<id>", methods=["GET"])
def queryState(id):
    global model
    model = games[id]
    model.step()
    listaRobots = []

    for i in range( len( model.schedule.agents ) ):
        agent = model.schedule.agents[i]

        if type(agent) is robots2.Robot:
                listaRobots.append({"x":agent.pos[0], "y": agent.pos[1],
"tipo":"Robot","cajasActualesRobot":agent.numberBoxesStack,
"leavedBoxes":agent.leavedBoxes })
        elif type(agent) is robots2.Box:
                listaRobots.append({"x":agent.pos[0], "y": agent.pos[1],
"tipo":"Caja",                  "limpio":                  agent.clean,
```

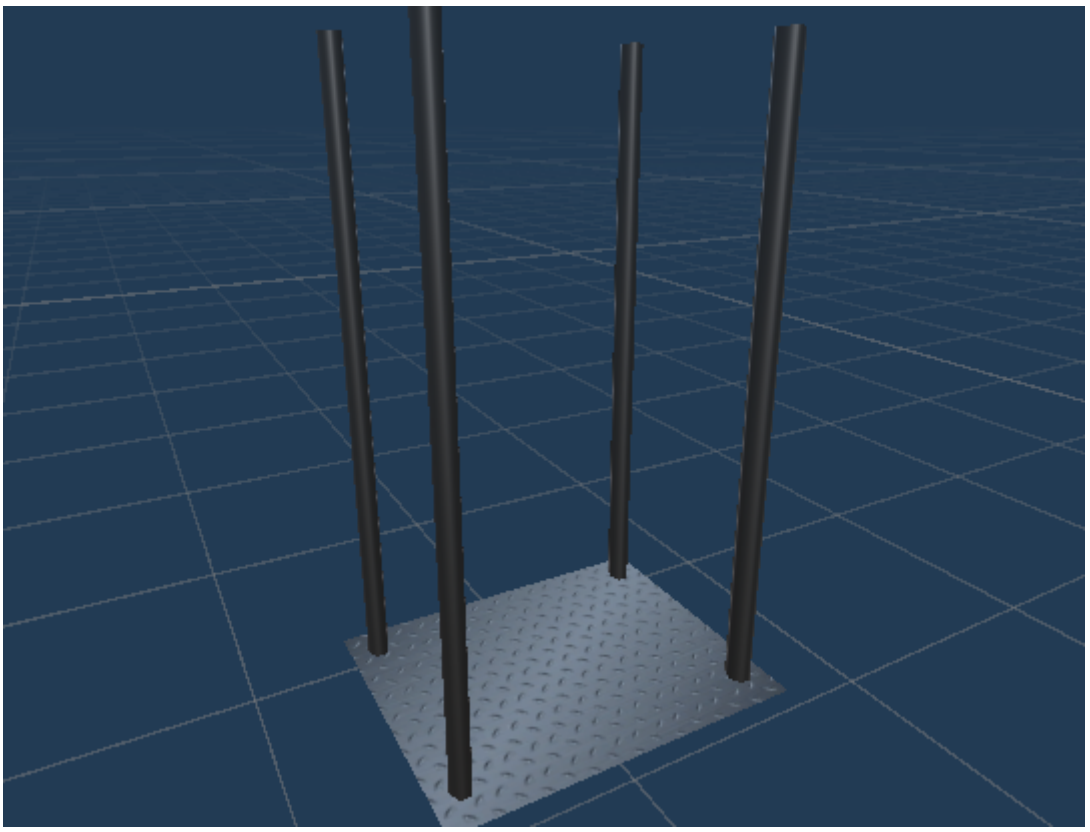
```
"stackAsignadoX":agent.assignedStackPosX,
"stackAsignadoY":agent.assignedStackPosY, "inStack":agent.inStack})
        elif type(agent) is robots2.Stack:
                listaRobots.append({"x":agent.pos[0], "y": agent.pos[1],
"tipo":"Stack", "boxesInStack": agent.boxes})
        else:
            i = i - 1
    return jsonify({"Items": listaRobots})

app.run()
```

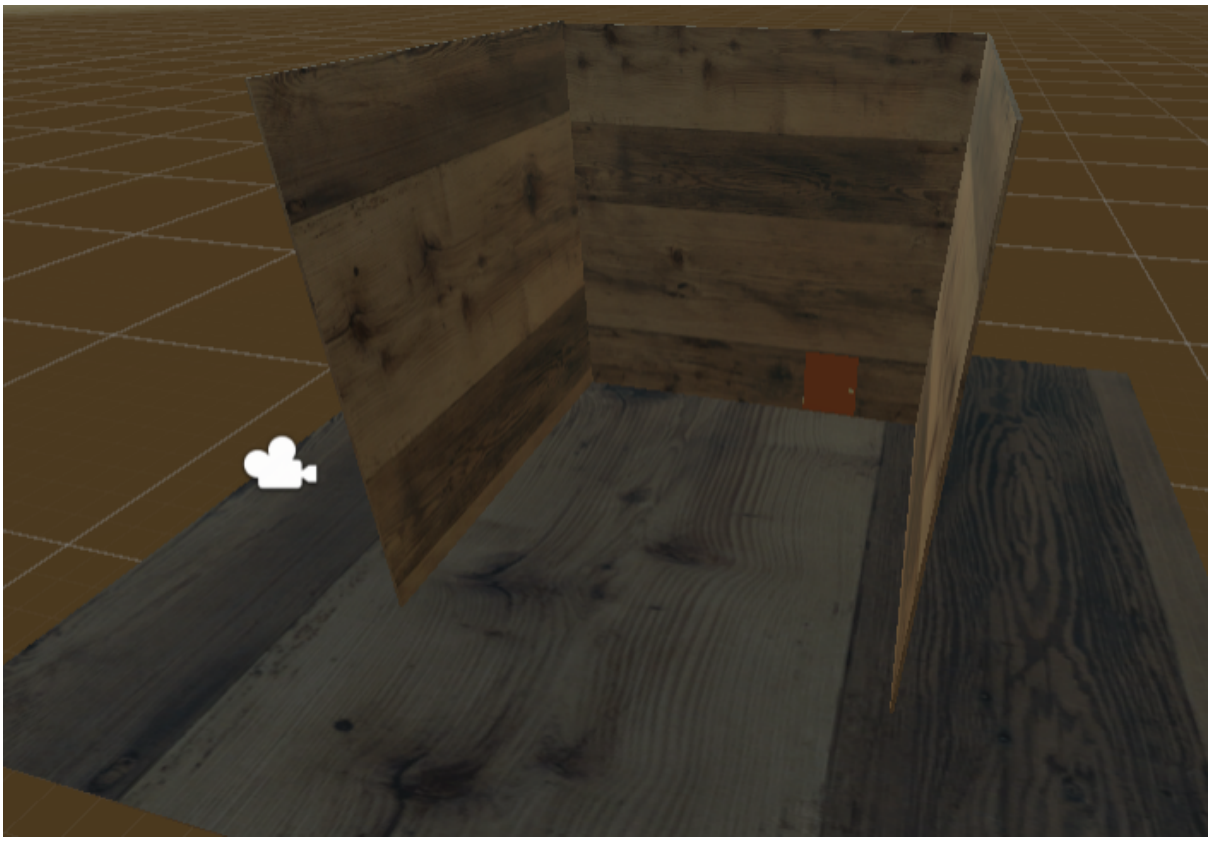
Finalmente, para realizar la simulación en Unity 3D, lo primero que se tuvo que realizar fué crear el entorno / ambiente en donde la simulación iba a ocurrir, para este caso en específico se realizó un cuarto sencillo con 4 paredes, simulando ser un almacén, este se modeló a través de la herramienta de ProBuilder de Unity. Posteriormente para dar vida a los agentes se escogieron varios prefabs de robots y cajas, los cuales fueron obtenidos en páginas de creative commons de modelados 3D y texturas, las cuales fueron texturizadas a través de la herramienta de ProBuilder: UV Editor.

Finalmente, para dar vida a esta simulación lo que se realizó fue un código en C#, donde se definen todos los prefabs que se usarán en la simulación, así como su comportamiento en el entorno a través de los objetos de GameObjects y lógicamente, a través de los datos recibidos por la API mencionada previamente, con la cual pudimos obtener la información deseada de cada objeto en el entorno 3D. A continuación se muestran algunos de estos modelos realizados y usados para dar vida a la simulación:









Código usado:

```
from typing import Text
from mesa import Agent, Model
from mesa.space import MultiGrid
from mesa.time import RandomActivation
from mesa.visualization.modules import CanvasGrid
from mesa.visualization.ModularVisualization import ModularServer

from pathfinding.core.diagonal_movement import DiagonalMovement
from pathfinding.core.grid import Grid as PathGrid
from pathfinding.finder.a_star import AStarFinder


class Robot(Agent):
    def __init__(self, model, pos):
        super().__init__(model.next_id(), model)

        self.pos = pos # robot coordenates
        self.numberBoxesStack = 0 # no. of boxes that are been carried
by the robot
        self.selfMovents = 0 # no. of movements that this robot is
making
        self.pickedBoxes = 0 # no. of picked boxes since the start of
the simulation individualy
        self.auxIndex = 0 # Auxilar index to handle the positions
traversed in our path to an empty closest stack tile.
        self.allocatedStack = (-1,-1) # Assigned Stack tile where the
robot most get the picked boxes
        self.tileType = "Robot"
        self.empty = True # attribute that help to leave boxes in a
stack tile
        self.leavedBoxes = False # attribute for APIs purposes



    def step(self):

        # we use the current neighbors
        next_moves = self.model.grid.get_neighborhood(self.pos,
moore=True)
        # we chose randomly some of these
```

```
        next_move = self.random.choice(next_moves)


        # check if the current cargo of a robot is full
        print("en stack del robot actual: ", self.numberBoxesStack)
        if self.numberBoxesStack == 5:
            isfullStack = True
        else:
            isfullStack = False


        if isfullStack == False:

            self.leavedBoxes = False # animation purposes
            isTileEmpty = self.pickAction() # we check the curent tile
is empty or not

            if self.selfMovents >= self.model.time: # we finished if
the time is overpassed
                print("Time's over, total movents: ", self.selfMovents
* self.model.robotsNo)
                self.model.finishSimulation = True
            if(self.model.boxNo == 0 and self.isAllRobotsEmpty()): # we
finished if there aren't any boxes left and each robot is empty
                print("All boxes have been picked up, adding up to: ",
self.getTotalMovements(), " total movents")
                self.model.finishSimulation = True
            if( self.selfMovents < self.model.time and isTileEmpty ): #
but if the time isn't over and the current tile is empty
                self.selfMovents += 1 # encrease self movemts of the
robot
                self.model.grid.move_agent(self, next_move)
        else:

            pathToStack = self.getPathToClosestTileStack() # we
calculate and recalculate the shortest path
            #print("path: ",pathToStack, "len: ", len(pathToStack),
"auxIndex: ", self.auxIndex)
            if( self.auxIndex < len(pathToStack) ):
                self.auxIndex = 1 # always select the second position
due to the curent position is the first (index:0)
                next_move = pathToStack[self.auxIndex]
```

```
                self.model.grid.move_agent(self, next_move) #move to
second position in our path array
            else: # when we get in here, we left the curent picked
boxes by the robot in the tile
                self.auxIndex = 0
                self.numberBoxesStack = 0
                self.leaveBoxes() # we call our function to do the
action mentioned previuosly
                self.leavedBoxes = True
                self.assignNewStack() # finally, we assiged other stack
for the curent robot.

            isTileEmpty = True # change this value so the robot will be
able to repeat the cycle.



    #function that helps us by probe if the current tile is empty or
not
    def pickAction(self):

        listTiles = self.model.grid.get_cell_list_contents([self.pos])
# we obtain the current tile member/s

        for tile in listTiles:

            if self.model.boxNo == 0:
                 # if there aren't any boxes to be picked up...
                if self.numberBoxesStack == 0: # and we got no robots
with box so far
                    self.empty = True # we marked that robot as empty
                    return False
                elif self.numberBoxesStack != 0: # but if the robot
still have boxes
                    self.showArrayOfNonFullStack() # we show how many
stacks didn't filled up
                    self.goToEmptyTileStack() # we go ahead to the new
empty stack

            if (tile.tileType == "Box"): # if the current tile is a Box

                tile.clean = True # we clean/picked the tile
                #print("alocatedStackX",self.allocatedStack)
```

```
                #print("alocatedStackY",self.allocatedStack)
                tile.assignedStackPosX = self.allocatedStack[0]
                tile.assignedStackPosY = self.allocatedStack[1]
                tile.changeColor() # doing so, we change the appereance
of the tile and it's own attributes
                self.pickedBoxes += 1 # we increment the global counter
for picked boxes
                self.numberBoxesStack += 1 # we increment the curent
cargo quantity
                tile.inStack = self.numberBoxesStack
                #print("Bloques recogidos: ", self.numberBoxesStack)
                self.model.boxNo -= 1 # we decrease the number of total
boxes
                #print("bloques restantes: ", self.model.boxNo)
                return False
            else:
                return True

    #function that calculate the total numbers of movents done by the
robots
    def getTotalMovements(self):
        movs = 0
        for robot in self.model.robsArray:
            movs += robot.selfMovents
        return movs

    # check if all robots are empty
    def isAllRobotsEmpty(self):
        for robot in self.model.robsArray:
            if robot.empty == False:
                return False
        return True

    #function that helps us to leave the cargo in a stack tile
    def leaveBoxes(self):
        listTiles = self.model.grid.get_cell_list_contents([self.pos])
        for tile in listTiles:
            if( tile.tileType == "Stack"):
                tile.boxes = 5
                self.numberBoxesStack = 0
                self.auxIndex = 0
```

```
    # function that helps us to assign another stacktile due to the
robot disassemble the cargo in.
    def assignNewStack(self):
        if( len( self.model.stacksPos ) > 0 ):
            stackIndex = self.random.randint(0, len(
self.model.stacksPos ) - 1) # we select radomly another stack to be
assign from our stack array
            tupleOfStack = self.model.stacksPos[stackIndex] # we create
a tuple, with new coords

            self.allocatedStack = tupleOfStack # reassigned the new
coords and the new stack aswell
            del self.model.stacksPos[stackIndex] # we delete that stack
from the original array


     #funcion auxiliar que saca la ruta hacia el stack mas cercano

    # Function that calculates the shortest path to a assigned
stackTile
    def getPathToClosestTileStack(self):
      grid = PathGrid(matrix=self.model.matrix)
      #print(self.model.matrix)
      grid.cleanup()
      start = grid.node(self.pos[0],self.pos[1])
      #print("start x:", self.pos[0]," y: ", self.pos[1])
      end = grid.node(self.allocatedStack[0],self.allocatedStack[1])
      #print("end x:", self.allocatedStack[0]," y: ",
self.allocatedStack[1])
      finder = AStarFinder(diagonal_movement=DiagonalMovement.never)
      path, runs = finder.find_path(start, end, grid)
      return path

    # Function that calculates the shortest path to a empty stackTile
    def getPathToEmptyTileStack(self, stack):

        grid = PathGrid(matrix=self.model.matrix)
        #print(self.model.matrix)
        grid.cleanup()
        start = grid.node( self.pos[0], self.pos[1] )
        end = grid.node( stack.pos[0], stack.pos[1] )
        finder = AStarFinder( diagonal_movement=DiagonalMovement.never
)
```

```
        path, runs = finder.find_path(start, end, grid)
        return path

    # delete and show from original array of stacks, those stacks that
haven't filled yet
    def showArrayOfNonFullStack(self):
        indexAux = 0
        for stack in self.model.arrStacks:
            if stack.full:
                del self.model.arrStacks[indexAux]
            indexAux += 1

        for stack in self.model.arrStacks:
            print("Stacks without been filled yet: ", stack.boxes, "
pos: ", stack.pos)

    # function that calcualtes the path and lead to the new empty stack
    def goToEmptyTileStack(self):

        if ( len(self.model.arrStacks) > 0 ):
            path =
self.getPathToEmptyTileStack(self.model.arrStacks[0]) #we
calculate/recalculate the path
            if( self.auxIndex < len( path ) ): # if we get in here, we
haven't arrive to the destination tileStack
                next_move = path[self.auxIndex]
                self.auxIndex = 1  # we alwsays took the second coord
due to the first (index:0) is the curent position
                self.model.grid.move_agent(self, next_move)
            else: # when we get in here when we arrive to the tile
                self.auxIndex = 0
                self.leaveMissingBoxes() # we left the boxes there

    #function that helps us to decide, if the robot could leave the
cargo in a tileStack
    def leaveMissingBoxes(self):

        # we select the agent types at the current tile
        listTiles = self.model.grid.get_cell_list_contents([self.pos])

        for tile in listTiles:

            if( tile.tileType == "Stack" ) :
```

```
                if( tile.boxes < 5 ):
                    if( ( tile.boxes + self.numberBoxesStack ) < 5 ):
                        tile.boxes = tile.boxes + self.numberBoxesStack
# reassign the new value of boxes in that stack
                        self.empty = True # we disassemble the cargo
                        self.numberBoxesStack = 0 # we restart the
curent cargo
                    elif ( ( tile.boxes + self.numberBoxesStack ) == 5
):
                        tile.boxes = 5 # we fill the current tileStack
                        self.numberBoxesStack = 0 #  we restart the
curent cargo
                        tile.full = True # and we set ass fill the
current tile
                    else: # if we get in here, that means we got a
spare boxes
                        spareBoxes = ( tile.boxes +
self.numberBoxesStack ) - 5 # we calculate the spare
                        tile.boxes = 5 # we fill the curent tile
                        tile.full = True
                        self.numberBoxesStack = spareBoxes #we left the
robot cargo the boxes spare
                else:
                    print("shouldn't get here")

# Usamos esta clase para crear las celdas sucias (las identificamos
como agentes sin movimiento)
class Box(Agent):
    def __init__(self, model, pos):
        super().__init__(model.next_id(), model)
        self.pos = pos
        self.tileType = "Box"
        self.clean = False
        self.assignedStackPosX = -1
        self.assignedStackPosY = -1
        self.inStack = -1

    def changeColor(self):
        self.tileType = "stdTile"


class stdTile(Agent):
    def __init__(self, model, pos):
```

```
        super().__init__(model.next_id(), model)
        self.pos = pos
        self.clean = True
        self.tileType = "stdTile"


class Stack(Agent):
    def __init__(self, model, pos):
        super().__init__(model.next_id(), model)
        self.pos = pos
        self.boxes = 0
        self.full = False
        self.tileType = "Stack"


class Floor(Model):
    #se asignan las variables modificables por el usuario siendo filas,
columnas, robots, tiempo de ejecucion y el numero de bloques sucios
    def __init__(self, rows =10,columns = 10, robotsNo = 5, time =
20000, boxNo = 60):
        super().__init__()

        self.schedule = RandomActivation(self)

        self.rows = rows # Num of rows
        self.columns = columns # Num of columns
        self.robotsNo = robotsNo # Num of robots
        self.matrix = [] # variable that store our grid
        self.boxNo = boxNo # Num of boxes
        self.Stacks = (boxNo // 5) # we calculate the number of
StackTiles by dividing the num of box by 5
        self.stacksPos = [] #variable that stores the stack positions
        self.robsArray = [] # arrasy to store our robots
        self.arrStacks = [] # array to store the stacksTiles
        self.time = time # max time to execute the simulation

        self.finishSimulation = False

        self.grid = MultiGrid(self.columns, self.rows, torus=False)

        #create a matrix full of zeros.
        self.createZeroMatrix()
```

```
        #We create the stacksTiles, robots, boxes and normal Tiles in
our grid
        self.setStacks()
        self.setRobot()
        self.setBox()
        self.setStdTile()

    def step(self):
        self.schedule.step()

        if self.finishSimulation:
            self.running = False


    @staticmethod
    def count_type(model):
        return model.boxNo

    def createZeroMatrix(self):
        for i in range(0, self.rows):
            zeros = []
            for j in range(0, self.columns):
                zeros.append(0)
            self.matrix.append(zeros)

    def setBox(self):
        tiles = self.boxNo
        while tiles > 0:
            randomPosX = self.random.randint(0, self.rows - 1)
            randomPosY = self.random.randint(0, self.columns - 1)
            while self.matrix[randomPosX][randomPosY] == 1:
                randomPosX = self.random.randint(0, self.rows - 1)
                randomPosY = self.random.randint(0, self.columns - 1)
            tile = Box( self, (randomPosY, randomPosX) )
            self.grid.place_agent( tile, tile.pos )
            self.matrix[randomPosX][randomPosY] = 1
            tiles -= 1
            self.schedule.add(tile)

    def setStdTile(self):
        for _, x, y in self.grid.coord_iter():
            if self.matrix[y][x] == 0:
                tile = stdTile( self, (x,y) )
```

```
                self.grid.place_agent(tile, tile.pos)
                self.schedule.add(tile)

    def setRobot(self):
        for _ in range(0, self.robotsNo):
            rob = Robot(self,(1,1))

            if( len(self.stacksPos) > 0 ):
                stackIndex = self.random.randint( 0,
len(self.stacksPos) - 1)
                stackTuple = self.stacksPos[stackIndex]
                rob.allocatedStack = stackTuple
                del self.stacksPos[stackIndex]

            self.robsArray.append(rob)
            self.grid.place_agent(rob, rob.pos)
            self.schedule.add(rob)



    def setStacks(self):
        for _ in range(0,self.Stacks):
            randomPosX = self.random.randint(0, (self.rows) -1)
            randomPosY = self.random.randint(0, (self.columns) -1)

            while self.matrix[randomPosX][randomPosY] == 1:
                randomPosX = self.random.randint(0, (self.rows) -1)
                randomPosY = self.random.randint(0, (self.columns) -1)

            stack = Stack(self, ( randomPosY, randomPosX ))
            self.grid.place_agent( stack, stack.pos )
            self.matrix[randomPosX][randomPosY] = 1

            self.stacksPos.append(stack.pos)
            self.arrStacks.append(stack)
            self.schedule.add(stack)

def agent_portrayal(agent):
    if( agent.tileType == "Robot" ):
        return {"Shape": "walle.png", "Layer": 0}
    elif( agent.tileType == "Box" ):
        return {"Shape": "rect", "w": 1, "h": 1, "Filled": "true",
"Color": "brown", "Layer": 1}
```

```
    elif( agent.tileType == "stdTile"):
        return {"Shape": "rect", "w": 1, "h": 1, "Filled": "true",
"Color": "#ced4da", "Layer": 1}
    elif( agent.tileType == "Stack" ):
        if agent.boxes == 5:
            agent.full = True
            return {"Shape": "rect", "w": 1, "h": 1, "Filled": "true",
"Color": "green", "Layer": 1}
        return {"Shape": "rect", "w": 1, "h": 1, "Filled": "true",
"Color": "red", "Layer": 1}

grid = CanvasGrid( agent_portrayal, 10, 10, 450, 450 )

server = ModularServer( Floor, [grid], "Store", {} )
server.port = 8524
server.launch()
```