Szymon Majkut

"Golce" – kooperacyjna sieciowa gra labiryntowa

Funkcjonalny opis systemu

Celem projektu jest zaprojektowanie oraz zaimplementowanie gry sieciowej opartej na architekturze klient-serwer, w której jeden lub więcej graczy, używając dedykowanej aplikacji klienckiej łączy się do wspólnej aplikacji serwerowej, odpowiedzialnej za synchronizację i wymianę danych. Tematem przewodnim samej rozgrywki będzie grywalne zasymulowanie cyklu życia kolonii golców piaskowych – niewielkich ssaków z rzędu gryzoni, żyjących w kilkunastoosobowych koloniach, z podziałem na kasty oraz dosyć długim czasem życia, jak na zwierzęta tego rozmiaru.

Bardzo istotnym elementem życia golców piaskowych jest struktura eusocjalna ich kolonii, występuje tam tylko jedna samica, odpowiedzialna za rozród (tak zwana królowa), pozostałe osobniki zajmują się pozyskiwaniem żywności, budują tunele oraz zdobywają żywność. W projekcie gry, gracze wcielają się z osobnika niższej kasty, który dla dobra całej kolonii, musi na bieżąco dostarczać żywność, razem z innymi graczami, tak aby kolonia była w stanie przetrwać jak najdłużej. Z założenia gracze poruszają się po przygotowanych już wcześniej tunelach, które na początku każdej rozgrywki zostają wygenerowane losowo, wraz z ustaleniem jednego punktu centralnego – głównej komory, do której należy sprowadzać żywność dla kolonii.

Golce piaskowe przystosowały się do życia w warunkach ciemnych i wąskich tuneli, poprzez zatracenie zmysłu wzroku, ale znaczne wyczulenie pozostałych zmysłów. Są w stanie precyzyjnie odbierać i reagować na wszystko, co dzieje się w tunelach, rozpoznając zagrożenia po zapachu lub dźwięku, ta cecha będzie stanowiła ważny element rozgrywki, gracz będzie znał precyzyjnie tylko najbliższe otoczenie, natomiast pozostała część informacji będzie pojawiała się wokół podróżującego golca w pewnej odległości od niego i zawsze będzie stanowiła sygnał zdobyty innym zmysłem.

Gracze będą mogli wpływać na otoczenie przez zamknięty zestaw komunikatów - dźwięków, symulujących nieustanne chrząkania golców, dzięki temu będą w stanie nawigować się nawzajem w tunelach. Dodatkowo każdy golec pozostawia ślad zapachowy w tunelu, dzięki czemu gracz jest w stanie określić czy znajduje się jeszcze blisko miejsca, w którym istnieje szansa spotkania jakiegoś golca czy może zabłądził gdzieś daleko od komory głównej.

Głównym zadaniem każdego gracza jest przynoszenie do komory głównej pożywienia – głównie korzenie oraz bulwy roślin, które mogą pojawiać się w tunelu w losowych momentach, ze zmiennym prawdopodobieństwem, im dalej dana odnoga tunelu znajduje się od komory głównej, tym większa szansa znalezienia pożywienia.

Komunikacja

Po uruchomieniu serwera oraz aplikacji klienckiej każdy z graczy poprzez aplikację kliencką podaje namiary na serwer oraz port i wybiera pseudonim. W pierwszym etapie serwer oczekuje na zgłoszenia graczy oraz otrzymanie komunikatu o rozpoczęciu rozgrywki. Wszyscy gracze otrzymują informacje o innych graczach oczekujących w poczekalni i mają możliwość zainicjowania rozgrywki.

Gdy serwer otrzymuje polecenie rozpoczęcia nowej gry, następuje generacja

labiryntu oraz początkowe umiejscowienie graczy w komorze głównej. Serwer przez cały czas trwania rozgrywki przechowuje oraz aktualizuje strukturę danych, w której zapisane są wszystkie szczegóły dotyczące mapy rozgrywki oraz położenie wszystkich graczy. Następnie serwer przygotowuje odpowiedź dla każdego z graczy.

Odpowiedź serwera w trakcie gry składa się z:

- struktury danych zawierającej wycinek obszaru gry, bezpośrednio otaczający gracza wraz z wszystkimi efektami dalszymi od niego, które byłby w stanie dostrzec przy pomocy innych zmysłów,
- czasu pozostałego do przetrwania roju (służy do zaktualizowania wartości licznika działającego w aplikacji klienckiej),
- informacji o znajdowaniu się w komory głównej – komory królowej.

W dalszej komunikacji klient jest w stanie wysłać wybraną przez siebie komendę, po czym otrzymuje odpowiedź servera, ze zaktualizowanym stanem gry. Możliwe jest również, że w danej rundzie gracz nie wykona ruchu, server nadal będzie wtedy odpowiadał użytkownikowi, na wypadek, gdyby któryś z innych graczy zmienił stan gry. Gracz może wybrać jedną z możliwych komend:

- ruch wraz z podaniem wybranego kierunku (góra, dół, lewo, prawo),
- wydanie odgłosu z podaniem wybranego z dostępnych,
- zebranie żywności – odstawienie żywności.

W sytuacji, gdy gracz wysyła więcej niż jedną komendę, ostatnia jest brana pod uwagę w trakcie aktualizacji stanu gry. Serwer oczekuje na komendy, a po ustalonym odstępie czasu, aktualizuje strukturę danych przechowującą szczegóły gry, zgodnie z kolejnością wybranych komend, dzięki czemu przy rozsyłaniu każdemu z graczy odpowiedzi każdy z nich otrzymuje spójny stan gry danej rundy, w postaci wycinka struktury gry wokół danego gracza.

Po każdej otrzymanej odpowiedzi serwera klient aktualizuje stan gry z punktu widzenia samego gracza i zaczyna nasłuchiwać i natychmiast przesyłać wszystkie kolejne komendy.

Gra kończy się w momencie, gdy ilość rund pozostała do przetrwania roju spadnie do zera. Ilość rund pozostałych do przetrwania wydłużana jest przez każde przyniesione przez graczy i pozostawione w komorze głównej pożywienie. Ostatecznie serwer zwraca wynik gry jako ilość rund, które przeżył rój do każdego z graczy, co kończy rozgrywkę.

Przewodnik użytkownika

Użytkownikiem aplikacji jest gracz, który łączy się z już ustalonym serverem. Gracz, który chce uruchomić grę w trybie okienkowym musi posiadać zainstalowany pakiet pygame, wtedy wystarczy, że uruchomi plik molerat_gui_client.py. Jeżeli server został uruchomiony z innym adresem lub portem niż domyślne, należy dostosować zawartość pliku zmieniając wartość globalnej zmiennej addr, ustalając te, z którymi uruchomiony został server, do którego gracz planuje się dołączyć.

Po uruchomieniu gry gracz zostanie poinformowany o sterowaniu grą oraz otrzymuje uproszczony opis wyglądu gry. W niewielkich odstępach czasu, pojawiają się również informacje o innych graczach znajdujących się w poczekalni. Każdy z graczy może uruchomić grę poprzez naciśnięcie przycisku spacji. Następnie gracz może sterować grą

wykorzystując klawisze wyszczególnione w opisie z poprzedniego ekranu, a na ekranie obserwować zaktualizowany obraz gry.

Możliwe jest również uruchomienie klienta w pełni konsolowego poprzez uruchomienie pliku molerat_client.py i jeżeli zachodzi taka potrzeba, dostosowania adresu oraz portu servera. Gracz jest w stanie obserwować wtedy słownik stanu gry, zalecane jest dostosowanie servera gry, aby jego odpowiedzi następowały po dłuższym czasie.

Reference Manual

Aplikacja działa w oparciu o model client-server. Aplikacja serverowa pakiet socket w celu utworzenia kanału komunikacji z klientami oraz pakiet _thread, dzięki któremu możliwe jest obsłużenie więcej niż jednego klienta jednocześnie, w sposób równoległy aktualizując stan gry. Dla każdego klienta zostaje otworzony osobne gniazdo do komunikacji, a sama komunikacja odbywa się w ramach nowo utworzonego wątku, pozostawiając aktualizowanie gry dla głównego wątku, w oparciu o komendy przychodzące od klientów.

Dodatkowo wykorzystywane są standardowe pakiety Python'a, pakiet time – do uśpienia wątków, których praca aktualnie nie jest wymagana, pakiet random – wykorzystywany w momencie generacji tuneli oraz pożywienia oraz pakiet json wykorzstywany przy komunikacji z klientami.

Aplikacja klienka jest przykładem "cienkiego klienta", posiada dwie główne funkcjonalności, również zaimplementowane z użyciem pakietów socket oraz _thread, pierwszą jest ciągłe odbieranie wejścia od użytkownika i jeżeli to możliwe, przetwarzanie go na komendę dla aplikacji serverowej, po czym wysyłanie tej komendy, drugą ciągłe wyświetlanie użytkownikowi aktualnego stanu gry, otrzymywanego z aplikacji serverowej.

Wykorzystany został uproszczony wzorzec projektowy Komendy, każda z komend otrzymywana przez aplikację serverową, zostaje zapisana jako osobna klasa, która podczas uruchomienia funkcji execute_command(), jest w stanie zaktualizować stan gry, enkapsulując swoją własną logikę.

Generacja labiryntu przeprowadzana jest w pierwszej fazie rozgrywki, tuż po rozpoczęciu gry przez jednego z graczy. Wybrany został uproszczony algorytm losowej ścieżki. Po wybraniu punktu na mapie oraz kierunku głównego, tworzony jest tunel, który może odbijać w dozwolonych kierunkach w sposób losowy, dając największe prawdopodobieństwo kierunkowi głównemu.

Instalacja i administracja

Pierwszym krokiem do zainstalowania aplikacji jest sklonowanie repozytorium, do którego link znajduje się na końcu tego dokumentu. Możliwe jest również udostępnienie jedynie kodu źródłowego klientów, dla samych graczy oraz przechowywanie kodu źródłowego servera tylko na maszynie, uruchamiającej aplikację serverową. Jedynym pakietem spoza biblioteki standardowej Python'a, jest pakiet pygame, który uruchamiany jest jedynie w kliencie okienkowym aplikacji.

Administrator, chcąc uruchomić aplikację serverową, po sklonowaniu repozytorium lub uzyskaniu aplikacji inną drogą, może dostosować adres oraz port usługi modyfikując zmienną addr, oraz dostosować czas gry, poprzez manipulowanie wartością wysyłaną do funkcji time.sleep() wewnątrz funkcji communicate() oraz game_loop(). W razie zmodyfikowania tej wartości, analogicznie powinny zostać one zmodyfikowane w

udostępnianym klientom kodzie aplikacji klienckiej.

Przed uruchomieniem aplikacji klienckich, administrator jest odpowiedzialny za uruchomienie pliku molerat_server.py i poinformowanie klientów, że mogą już łączyć się z serverem. Po zakończeniu gry administrator jest odpowiedzialny za zatrzymanie aplikacji i jeżeli występuje taka potrzeba, ponowne uruchomienie gry.

Kod źródłowy

Kod źródłowy zostaje dołączony do dokumentacji w postaci trzech plików: molerat_server.py, molerat_gui_client.py oraz opcjonalnego molerat.client.py oraz pliku Readme.md, zawierającego skrócony opis administracji. Kod źródłowy wraz z dokumentacją został również umieszczony w zdalnym repozytorium git i jest dostępny pod adresem: https://github.com/SzymonMajk/molerats

molerat_server.py

```
import socket
import _thread
import time
import sys
import random
import json


initial_reserves = 50
board_size = 100
vision_render = 3
audition_render = 8
smell_render = 15
food_probability = 0.002
service_address = ('', 65420)

class NoopCommand:
    def execute(self, player, board):
        pass

class MoveCommand:
    def __init__(self, direction):
        self.direction = direction

    def execute(self, player, board):
        new_x_position = player.x_position
        new_y_position = player.y_position

        if self.direction == "north":
            new_y_position = new_y_position + 1
        elif  self.direction == "east":
            new_x_position = new_x_position - 1
```

```
        elif  self.direction == "west":
            new_x_position = new_x_position + 1
        elif self.direction == "south":
            new_y_position = new_y_position - 1

        if board.can_move(new_x_position, new_y_position):
            player.move(new_x_position, new_y_position)
            board.left_pheromones(player.nick, player.x_position, player.y_position)

class SoundCommand:
    def __init__(self, sound):
        self.sound = sound

    def execute(self, player, board):
        board.add_sound(self.sound, player.x_position, player.y_position)

class CollectCommand:
    def execute(self, player, board):
        if player.inventory_reserves <= 0:
            player.inventory_reserves = board.collect_food(player.x_position, player.y_position)
        elif board.inside_queen_chamber(player.x_position, player.y_position):
            board.left_reserves(player.inventory_reserves)
            player.inventory_reserves = 0

class Player:
    def __init__(self, addr, nick):
        self.addr = addr
        self.x_position = board_size / 2
        self.y_position = board_size / 2
        self.inventory_reserves = 0
        self.nick = nick
        self.current_command = NoopCommand()

    def move(self, new_x_position, new_y_position):
        self.x_position = new_x_position
        self.y_position = new_y_position

    def execute_command(self, board):
        self.current_command.execute(self, board)

class Pheromone:
    def __init__(self, nick, x_position, y_position):
        self.nick = nick
        self.time_to_live = 10
        self.x_position = x_position
        self.y_position = y_position

    def list_serialize(self):
```

```
        return [self.nick, self.x_position, self.y_position]

class Sound:
    def __init__(self, type, x_position, y_position):
        self.type = type
        self.time_to_live = 3
        self.x_position = x_position
        self.y_position = y_position

    def list_serialize(self):
        return [self.type, self.x_position, self.y_position]

class Food:
    def __init__(self, value, x_position, y_position):
        self.value = value
        self.x_position = x_position
        self.y_position = y_position

    def list_serialize(self):
        return [self.value, self.x_position, self.y_position]

class GameBoard:
    def __init__(self, size, probability):
        self.size = size
        self.probability = probability
        self.fields = {}
        self.foods = []
        self.sounds = []
        self.pheromones = []
        self.collected_food = 0
        self.generate_board()

    def generate_board(self):
        for row in range(0, self.size):
            self.fields[row] = {}
            for col in range(0, self.size):
                if (self.inside_queen_chamber(row, col)):
                    self.fields[row][col] = 'F'
                else:
                    self.fields[row][col] = 'W'

        x_dig = self.size / 2
        y_dig = self.size / 2

        for i in range(0, self.size):
            random_value = random.random()
            if random_value < 0.25 and y_dig < self.size - 1:
                y_dig = y_dig + 1
```

```
    elif random_value >= 0.25 and random_value <= 0.75 and x_dig < self.size - 1:
        x_dig = x_dig + 1
    elif random_value > 0.75 and y_dig > 0:
        y_dig = y_dig - 1

    self.fields[x_dig][y_dig] = 'F'

x_dig = self.size / 2
y_dig = self.size / 2

for i in range(0, self.size):
    random_value = random.random()
    if random_value < 0.25 and y_dig < self.size - 1:
        y_dig = y_dig + 1
    elif random_value >= 0.25 and random_value <= 0.75 and x_dig > 0:
        x_dig = x_dig - 1
    elif random_value > 0.75 and y_dig > 0:
        y_dig = y_dig - 1

    self.fields[x_dig][y_dig] = 'F'

x_dig = self.size / 2
y_dig = self.size / 2

for i in range(0, self.size):
    random_value = random.random()
    if random_value < 0.25 and x_dig < self.size - 1:
        x_dig = x_dig + 1
    elif random_value >= 0.25 and random_value <= 0.75 and y_dig < self.size - 1:
        y_dig = y_dig + 1
    elif random_value > 0.75 and x_dig > 0:
        x_dig = x_dig - 1

    self.fields[x_dig][y_dig] = 'F'

x_dig = self.size / 2
y_dig = self.size / 2

for i in range(0, self.size):
    random_value = random.random()
    if random_value < 0.25 and x_dig < self.size - 1:
        x_dig = x_dig + 1
    elif random_value >= 0.25 and random_value <= 0.75 and y_dig > 0:
        y_dig = y_dig - 1
    elif random_value > 0.75 and x_dig > 0:
        x_dig = x_dig - 1
```

```
            self.fields[x_dig][y_dig] = 'F'

    def generate_food(self):
        for row in range(0, self.size):
            for col in range(0, self.size):
                random_value = random.random()
                if random_value <= self.probability and not self.inside_queen_chamber(row, col)
and self.fields[row][col] == 'F':
                    self.foods.append(Food(int((1 - random_value) * 50), row, col))

    def update_sounds(self):
        for sound in list(self.sounds):
            sound.time_to_live = sound.time_to_live - 1

            if sound.time_to_live <= 0:
                self.sounds.remove(sound)

    def update_pheromones(self):
        for pheromone in list(self.pheromones):
            pheromone.time_to_live = pheromone.time_to_live - 1

            if pheromone.time_to_live <= 0:
                self.pheromones.remove(pheromone)


    def can_move(self, x_position, y_position):
        if x_position in self.fields.keys():
            if y_position in self.fields[x_position].keys():
                return self.fields[x_position][y_position] == 'F'

    def add_sound(self, type, x_position, y_position):
        self.sounds.append(Sound(type, x_position, y_position))

    def left_pheromones(self, nick, x_position, y_position):
        self.pheromones.append(Pheromone(nick, x_position, y_position))

    def collect_food(self, x_position, y_position):
        for food in list(self.foods):
            if food.x_position == x_position and food.y_position == y_position:
                value = food.value
                print("Collected! " + str(value))
                self.foods.remove(food)
                return value
        return 0

    def inside_queen_chamber(self, x_position, y_position):
        return abs((self.size / 2) - x_position) <= 2 and abs((self.size / 2) - y_position) <= 2
```

```
    def left_reserves(self, food):
        print("More fooood! " + str(food))
        self.collected_food = food

    def use_reserves(self):
        used_reserves = self.collected_food
        self.collected_food = 0
        return used_reserves

    def fields_map_to_list_around(self, x_position, y_position):
        result = []

        for row in self.fields.keys():
            for col in self.fields[row]:
                if abs(row - x_position) <= vision_render and abs(col - y_position) <=
vision_render:
                    result.append([row - x_position, col - y_position, self.fields[row][col]])

        return result

    def foods_to_list_around(self, x_position, y_position):
        result = []

        for food in self.foods:
            if abs(food.x_position - x_position) <= audition_render and abs(food.y_position -
y_position) <= smell_render:
                result.append([food.x_position - x_position, food.y_position - y_position,
food.value])

        return result

    def sounds_to_list_around(self, x_position, y_position):
        result = []

        for sound in self.sounds:
            if abs(sound.x_position - x_position) <= audition_render and abs(sound.y_position -
y_position) <= audition_render:
                result.append([sound.x_position - x_position, sound.y_position - y_position,
sound.type])

        return result

    def pheromones_to_list_around(self, x_position, y_position):
        result = []

        for pheromone in self.pheromones:
            if abs(pheromone.x_position - x_position) <= audition_render and
abs(pheromone.y_position - y_position) <= smell_render:
```

```
            result.append([pheromone.x_position - x_position, pheromone.y_position -
y_position, pheromone.nick])

        return result

class Game:
    def __init__(self):
        self.reset()

    def reset(self):
        self.board = GameBoard(board_size, food_probability)
        self.running = False
        self.round = 0
        self.score = 0
        self.players = dict()
        self.reserves = initial_reserves

    def add_player(self, addr, player):
        self.players[addr] = player

    def remove_player(self, addr):
        try:
            del self.players[addr]
        except KeyError:
            pass

    def render_lobby(self):
        lobby = "Lobby:"
        for p in game.players.values():
            lobby = lobby + " " + p.nick + " "
        return lobby

    def update_game(self):
        if self.reserves > 0:
            self.board.generate_food()
            self.board.update_sounds()
            self.board.update_pheromones()
            for player in self.players.values():
                player.execute_command(self.board)
                player.current_command = NoopCommand()
            self.reserves = self.reserves + self.board.use_reserves()
            self.round = self.round + 1
            self.reserves = self.reserves - 1
        else:
            self.running = False
            self.score = "Game finished, score = " + str(self.round)
```

```
    def render_for_player(self, addr):
        x_position = self.players[addr].x_position
        y_position = self.players[addr].y_position
        rendered = dict()
        rendered["fields"] = self.board.fields_map_to_list_around(x_position, y_position)
        rendered["foods"] = self.board.foods_to_list_around(x_position, y_position)
        rendered["sounds"] = self.board.sounds_to_list_around(x_position, y_position)
        rendered["pheromones"] = self.board.pheromones_to_list_around(x_position,
y_position)
        rendered["reserves"] = self.reserves
        if self.board.inside_queen_chamber(x_position, y_position):
            rendered["queen_chamber"] = True
        return json.dumps(rendered)

    def finished(self):
        return not self.running and self.reserves <= 0 and not self.players

def parse_input(raw_input):
    if raw_input == "north":
        return MoveCommand("north")
    elif raw_input == "east":
        return MoveCommand("east")
    elif raw_input == "south":
        return MoveCommand("south")
    elif raw_input == "west":
        return MoveCommand("west")
    elif raw_input == "collect":
        return CollectCommand()
    elif raw_input == "snarl":
        return SoundCommand("snarl")
    elif raw_input == "scrape":
        return SoundCommand("scrape")
    elif raw_input == "squeak":
        return SoundCommand("squeak")
    else:
        return NoopCommand()

def handle_client(conn, addr, game):
    with conn as client_socket:
        nick = client_socket.recv(1024).decode()
        player = Player(addr, nick)
        game.add_player(addr, player)
        client_socket.send(nick.encode())

        while True:
            try:
                msg = client_socket.recv(1024)
```

```
            if msg.decode() == "start":
                msg = "Game started by " + str(player.nick)
                game.running = True

            if game.running:
                msg = b'Game starts in seconds...'
                client_socket.send(msg)
                break

            msg = game.render_lobby().encode()
            client_socket.send(msg)
        except ConnectionResetError:
            game.remove_player(addr)

    while True:
        try:
            if game.running:
                player.current_command = parse_input(client_socket.recv(1024).decode())
                rendered_game = game.render_for_player(addr)
                client_socket.sendall(bytes(rendered_game,encoding="utf-8"))
            else:
                client_socket.recv(1024)
                client_socket.sendall(bytes(game.score,encoding="utf-8"))
                break
        except (ConnectionResetError, ConnectionAbortedError):
            game.remove_player(addr)
            break

def game_loop(game):
    while True:
        try:
            if game.running:
                game.update_game()
                print('Game updated! Round ' + str(game.round))
                time.sleep(0.5)
            elif game.finished():
                print('Game finished! ' + str(game.score))
                game.reset()
                time.sleep(0.5)
            else:
                print(game.render_lobby())
                time.sleep(2)
        except KeyboardInterrupt:
            break


with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as server_socket:
```

```
    print('Server started!')
    game = Game()
    _thread.start_new_thread(game_loop,(game,))
    print('Waiting for clients...')
    server_socket.bind(service_address)
    server_socket.settimeout(1.0)
    server_socket.listen(5)

    while True:
        try:
            conn, addr = server_socket.accept()
            print('New client! Address: ', addr)
            _thread.start_new_thread(handle_client,(conn, addr,game))
        except IOError:
            continue
        except KeyboardInterrupt:
            break
```

molerat_client.py

```
import socket
import _thread
import time
import json

class GameCondition:
    def __init__(self):
        self.finished = False
        self.message = "noop"

    def reset(self):
        self.message = "noop"

def recvall(socket):
    BUFF_SIZE = 1024
    data = b''
    while True:
        part = socket.recv(BUFF_SIZE)
        data += part
        if len(part) < BUFF_SIZE:
            break
    return data

def communicate(addr, game_condition):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.connect(addr)

        s.sendall(game_condition.message.encode())
```

```
        data = recvall(s).decode("utf-8")
        print(repr(data))

        while True:
            try:
                s.sendall(game_condition.message.encode())
                data = recvall(s).decode("utf-8")
                try:
                    print(json.loads(data))
                except json.decoder.JSONDecodeError as e:
                    print(data)
                time.sleep(1)
            except (ConnectionResetError, ConnectionAbortedError):
                game_condition.finished = True
                break

print("Welcome in molerats!")

addr = ('localhost', 65420)
game_condition = GameCondition()

print("We will try to connect to server at host ", addr[0], " on port ", addr[1])
print("First send your nick, „start" will finish the lobby and start the game on server")
print("After each iteration you will be informed about game status")
print("Move using wasd keys, 1,2 and 3 will allow you to make noise, use r to pick a food")
print("Remember! It is easy to get lost in molerats tunnels...")

game_condition.message = input("I am ")
_thread.start_new_thread(communicate,(addr, game_condition))

while True:
    if game_condition.finished:
        break

    raw_input = input("To server start to start, other options w a s d r 1 2 3 << ")

    if raw_input == "start":
        game_condition.message = "start"
    elif raw_input == "w":
        game_condition.message = "north"
    elif raw_input == "a":
        game_condition.message = "east"
    elif raw_input == "s":
        game_condition.message = "south"
    elif raw_input == "d":
        game_condition.message = "west"
    elif raw_input == "r":
        game_condition.message = "collect"
```

```
    elif raw_input == "1":
        game_condition.message = "snarl"
    elif raw_input == "2":
        game_condition.message = "scrape"
    elif raw_input == "3":
        game_condition.message = "squeak"
    else:
        time.sleep(0.1)
```

molerat_gui_client.py

```
import socket
import _thread
import time
import json
import pygame

class GameCondition:
    def __init__(self):
        self.finished = False
        self.message = "noop"
        self.state = {}

    def reset(self):
        self.message = "noop"

def recvall(socket):
    BUFF_SIZE = 1024
    data = b''
    while True:
        part = socket.recv(BUFF_SIZE)
        data += part
        if len(part) < BUFF_SIZE:
            break
    return data

def communicate(addr, game_condition):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.connect(addr)

        s.sendall(game_condition.message.encode())
        data = recvall(s).decode("utf-8")
        print(repr(data))

        while True:
            try:
                s.sendall(game_condition.message.encode())
                data = recvall(s).decode("utf-8")
```

```
            try:
                game_condition.state = json.loads(data)
            except json.decoder.JSONDecodeError as e:
                game_condition.state = json.loads('{"lobby" : "' + data + '"}')
            time.sleep(0.01)
        except (ConnectionResetError, ConnectionAbortedError):
            game_condition.finished = True
            break

print("Welcome in molerats!")

def draw_game(pygame, state):

    if "fields" in state.keys():
        for field in state["fields"]:
            if field[2] == 'F':
                pygame.draw.rect(screen, (0, 128, 255), pygame.Rect(field[0] * 20 + 15 * 20,
field[1] * 20 + 15 * 20, 20, 20))
            else:
                pygame.draw.rect(screen, (128, 128, 128), pygame.Rect(field[0] * 20 + 15 * 20,
field[1] * 20 + 15 * 20, 20, 20))
    if "pheromones" in state.keys():
        for pheromone in state["pheromones"]:
            pygame.draw.rect(screen, (128, 0, 0), pygame.Rect(pheromone[0] * 20 + 15 * 20,
pheromone[1] * 20 + 15 * 20, 20, 20))

    if "sounds" in state.keys():
        for sound in state["sounds"]:
            if field[2] == 'snarl':
                pygame.draw.rect(screen, (0, 108, 0), pygame.Rect(sound[0] * 20 + 15 * 20,
sound[1] * 20 + 15 * 20, 20, 20))
            elif field[2] == 'scrape':
                pygame.draw.rect(screen, (0, 128, 0), pygame.Rect(sound[0] * 20 + 15 * 20,
sound[1] * 20 + 15 * 20, 20, 20))
            elif field[2] == 'squeak':
                pygame.draw.rect(screen, (0, 148, 0), pygame.Rect(sound[0] * 20 + 15 * 20,
sound[1] * 20 + 15 * 20, 20, 20))
    if "foods" in state.keys():
        for food in state["foods"]:
            pygame.draw.rect(screen, (255, 0, 0), pygame.Rect(food[0] * 20 + 15 * 20, food[1] *
20 + 15 * 20, 20, 20))
        pygame.draw.rect(screen, (255, 255, 255), pygame.Rect(15 * 20, 15 * 20, 20, 20))

        text = "Reserves = " + str(state["reserves"])
        if "queen_chamber" in state.keys():
            text = text + ' in queen chamber'
        myfont = pygame.font.SysFont('Comic Sans MS', 30)
        textsurface = myfont.render(text, False, (50, 50, 50))
```

```
        screen.blit(textsurface,(20,20))
    else:
        myfont = pygame.font.SysFont('Comic Sans MS', 30)
        if 'score' in state['lobby']:
            textsurface = myfont.render(state['lobby'], False, (50, 50, 50))
            screen.blit(textsurface,(20,20))
        else:
            textsurface1 = myfont.render('Push space to start!', False, (50, 50, 50))
            textsurface2 = myfont.render('w a s d - move, r - pick food, 1,2,3 - noise', False, (50,
50, 50))
            textsurface3 = myfont.render('player is white rectangle, food is red', False, (50, 50,
50))
            textsurface4 = myfont.render('collected food left in quenn chamber', False, (50, 50,
50))
            textsurface5 = myfont.render(state['lobby'], False, (50, 50, 50))
            screen.blit(textsurface1,(20,140))
            screen.blit(textsurface2,(20,20))
            screen.blit(textsurface3,(20,60))
            screen.blit(textsurface4,(20,100))
            screen.blit(textsurface5,(20,180))

addr = ('localhost', 65420)
game_condition = GameCondition()

print("We will try to connect to server at host ", addr[0], " on port ", addr[1])
print("First send your nick, ,,start" will finish the lobby and start the game on server")
print("After each iteration you will be informed about game status")
print("Move using wasd keys, 1,2 and 3 will allow you to make noise, use r to pick a food")
print("Remember! It is easy to get lost in molerats tunnels...")
print("Space to start, then w a s d 1 2 3 and r")

game_condition.message = input("I am ")
_thread.start_new_thread(communicate,(addr, game_condition))

pygame.init()
screen = pygame.display.set_mode((600, 600))
clock = pygame.time.Clock()

while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
        if event.type == pygame.KEYDOWN and event.key == pygame.K_SPACE:
            game_condition.message = "start"
        elif event.type == pygame.KEYDOWN and event.key == pygame.K_w:
            game_condition.message = "south"
        elif event.type == pygame.KEYDOWN and event.key == pygame.K_a:
            game_condition.message = "east"
```

```
    elif event.type == pygame.KEYDOWN and event.key == pygame.K_s:
        game_condition.message = "north"
    elif event.type == pygame.KEYDOWN and event.key == pygame.K_d:
        game_condition.message = "west"
    elif event.type == pygame.KEYDOWN and event.key == pygame.K_r:
        game_condition.message = "collect"
    elif event.type == pygame.KEYDOWN and event.key == pygame.K_1:
        game_condition.message = "snarl"
    elif event.type == pygame.KEYDOWN and event.key == pygame.K_2:
        game_condition.message = "scrape"
    elif event.type == pygame.KEYDOWN and event.key == pygame.K_3:
        game_condition.message = "squeak"

screen.fill((0, 0, 0))
draw_game(pygame, game_condition.state)
pygame.event.pump()
pygame.display.flip()
clock.tick(60)
```