

Fila 1



Universitatea Tehnică de Construcții București

Facultatea de Hidrotehnica

Proiect la Programarea Calculatoarelor si Limbaje de Programare II

Profesor Coordonator
Olteanu Gabriela

Student
Condrat Felix-Ioan
Grigore Ana-Raluca

Simulare de trafic rutier: un program care simulează traficul rutier într-un oraș.

Cuprins

1.Introducere.....	4
2.Simulare.cpp.....	6
3.Interfata_grafica.py.....	12
4.Sim.json.....	17
5.run.bat.....	21
6.json.hpp.....	22

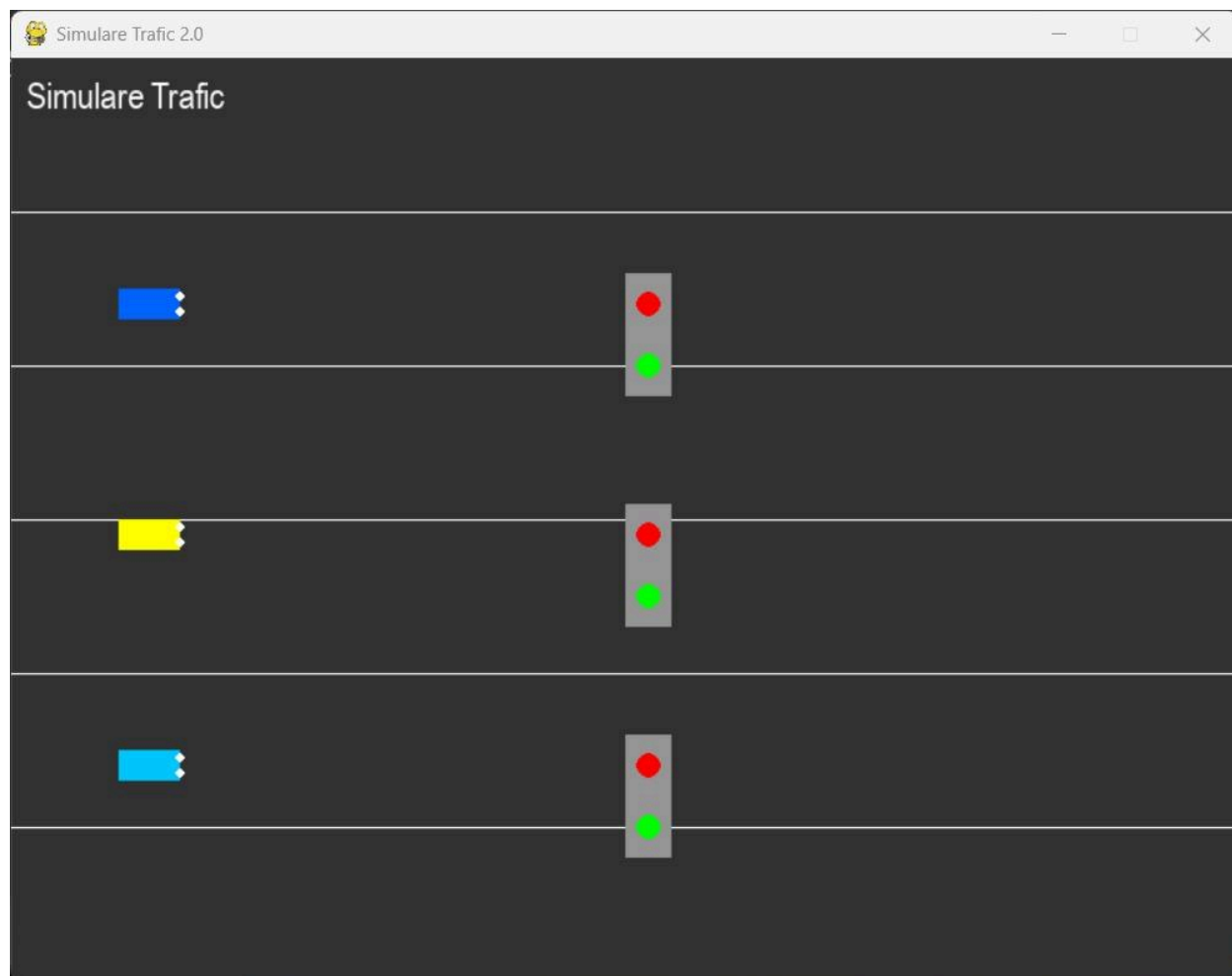
Introducere

Am ales acest proiect pentru că traficul este o parte importantă a vieții noastre de zi cu zi, iar simularea sa poate ajuta la înțelegerea modului în care funcționează semafoarele și mișcarea mașinilor pe stradă. Scopul proiectului este crearea unei aplicații simple care să arate cum reacționează mașinile în funcție de semafoare, într-un mod ușor de înțeles.

Proiectul este o simulare a unui trafic simplu, unde câteva mașini pornesc de pe drum și se opresc sau pornesc în funcție de culoarea semaforului. În aplicație, semafoarele pot fi fie roșii, fie verzi, iar mașinile se mișcă doar când este verde.

Proiectul este format din mai multe fișiere care lucrează împreună pentru a simula traficul rutier. Fișierul **simulare.cpp** conține logica principală a simulării: citește pozițiile mașinilor și starea semafoarelor din fișierul **sim.json**, actualizează pozițiile mașinilor în funcție de culoarea semafoarelor și salvează rezultatele înapoi în fișierul JSON. Fișierul **simulator.exe** este executabilul generat după compilarea codului C++ și rulează efectiv simularea. Fișierul **interfata_grafica.py** este programul care desenează pe ecran drumul, mașinile și semafoarele, citind datele din **sim.json** și actualizând imaginea la fiecare pas. Fișierul **sim.json** este fișierul de legătură dintre codul C++ și interfața grafică, care conține pozițiile actuale ale mașinilor și starea semafoarelor. Fișierul **json.hpp**, aflat în folderul **nlohmann**, este o bibliotecă care permite programului C++ să lucreze cu fișiere JSON. În final, fișierul **run.bat** este un script care compilează automat programul C++ și pornește atât simularea, cât și interfața grafică pentru o rulare completă a proiectului.

Am folosit cunoștințele de programare pentru a face ca totul să fie simplu, funcțional și ușor de înțeles. Proiectul arată cum se pot combina mai multe limbaje de programare (C++ și Python) pentru a crea o aplicație interesantă care poate avea aplicabilitate și în alte simulări.



Fereastra unde se afișează simulatorul de trafic

Simulare.cpp

simulare.cpp este programul care calculează poziția mașinilor în funcție de semafoare. Citește datele din fișierul JSON, verifică dacă semaforul e verde sau roșu, mută mașinile dacă este verde, actualizează pozițiile în JSON și oprește simularea când toate mașinile au trecut.

```

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37
simulare.cpp > main()
1 #include <iostream>
2 #include <fstream>
3 #include <vector>
4 #include "json.hpp" // asigură-te că json.hpp e în același folder
5
6 using json = nlohmann::json;
7 using namespace std;
8
9
10 struct Car {
11     int id;
12     int x, y;
13 };
14
15 struct TrafficLight {
16     int x, y;
17     bool green_ns;
18 };
19
20 int main() {
21     ifstream f("sim.json");
22     if (!f) {
23         cerr << "Nu s-a putut deschide sim.json" << endl;
24         return 1;
25     }
26
27     json data;
28     f >> data;
29
30     vector<Car> cars;
31     for (const auto& car : data["cars"]) {
32         Car c;
33         c.id = car["id"].get<int>();
34         c.x = car["x"].get<int>();
35         c.y = car["y"].get<int>();
36         cars.push_back(c);
37     }
38 }

```

```
38
39     vector<TrafficLight> lights;
40     for (const auto& light : data["traffic_lights"]) {
41         TrafficLight tl;
42         tl.x = light["x"].get<int>();
43         tl.y = light["y"].get<int>();
44         tl.green_ns = light["green_ns"].get<bool>();
45         lights.push_back(tl);
46     }
47
48     for (auto& car : cars) {
49         for (const auto& light : lights) {
50             if (abs(car.y - light.y) < 20) {
51                 if (light.green_ns || car.x + 40 < light.x) {
52                     car.x += 50;
53                 }
54                 break;
55             }
56         }
57     }
58
59     json out;
60     for (const auto& car : cars) {
61         out["cars"].push_back({{"id", car.id}, {"x", car.x}, {"y", car.y}});
62     }
63     for (const auto& light : lights) {
64         out["traffic_lights"].push_back({
65             {"x", light.x}, {"y", light.y}, {"green_ns", light.green_ns}
66         });
67     }
68
69     ofstream o("sim.json");
70     o << out.dump(4);
71
72     return 0;
73 }
74
```

```
#include <iostream>
#include <fstream>
#include <vector>
#include "json.hpp" // asigură-te că json.hpp e în același folder

using json = nlohmann::json;
using namespace std;
```


Acest cod pregătește programul pentru a lucra cu diverse funcții necesare simulării traficului. Sunt incluse biblioteci standard precum **iostream** pentru afișarea mesajelor în consolă, **fstream** pentru lucrul cu fișiere, și **vector** pentru a folosi liste dinamice. De asemenea, este inclusă biblioteca **json.hpp**, care permite programului să citească și să scrie date în format **JSON**, fiind important ca fișierul **json.hpp** să fie în același folder cu programul.

Prin instrucțiunea **using json = nlohmann::json;**, se creează o scurtătură, astfel încât să putem folosi cuvântul **json** în loc de a scrie mereu **nlohmann::json**, ceea ce face codul mai simplu și mai ușor de citit. În plus, **using namespace std;** ne ajută să evităm să scriem **std::** de fiecare dată când folosim funcții sau obiecte standard, cum ar fi **cout**, **cin** sau **vector**.

```
✓ struct Car {  
    |     int id;  
    |     int x, y;  
    | };  
  
✓ struct TrafficLight {  
    |     int x, y;  
    |     bool green_ns;  
    | };
```

În acest cod sunt definite două structuri care ajută la organizarea datelor despre mașini și semafoare. Structura **Car** este folosită pentru a descrie o mașină, având trei atribute: **id**, care identifică fiecare mașină, și **x** și **y**, care reprezintă poziția mașinii pe ecran. Structura **TrafficLight** este folosită pentru a reprezenta un semafor, având tot două coordonate (**x** și **y**) care arată poziția semaforului pe ecran și un câmp **green_ns**, de tip boolean, care indică dacă semaforul este verde (**true**) sau roșu (**false**). Aceste structuri simplifică gestionarea datelor în program, oferind o formă organizată pentru a stoca informații despre mașini și semafoare.

```
int main() {
    ifstream f("sim.json");
    if (!f) {
        cerr << "Nu s-a putut deschide sim.json" << endl;
        return 1;
    }
}
```

Se folosește **ifstream f("sim.json");** pentru a crea un flux de intrare care încearcă să deschidă fișierul **sim.json**. După aceea, se verifică dacă fișierul a fost deschis cu succes: dacă **f** nu este valid (adică fișierul nu a putut fi deschis), programul afișează un mesaj de eroare în consolă – „Nu s-a putut deschide sim.json” – și oprește rularea prin comanda **return 1;**.

```
json data;
f >> data;
```

Acest cod creează o variabilă de tip **JSON** numită **data** și încarcă în ea datele din fișierul **sim.json**. Mai exact, **json data;** definește o variabilă care poate stoca informații în format JSON, iar linia **f >> data;** citește toate datele din fișierul deschis anterior (**f**) și le salvează în variabila **data**. Practic, această parte de cod „traduce” fișierul **sim.json** într-o formă pe care programul o poate folosi în continuare.

```
vector<Car> cars;
for (const auto& car : data["cars"]) {
    Car c;
    c.id = car["id"].get<int>();
    c.x = car["x"].get<int>();
    c.y = car["y"].get<int>();
    cars.push_back(c);
}

vector<TrafficLight> lights;
for (const auto& light : data["traffic_lights"]) {
    TrafficLight tl;
    tl.x = light["x"].get<int>();
    tl.y = light["y"].get<int>();
    tl.green_ns = light["green_ns"].get<bool>();
    lights.push_back(tl);
}
```

Acest cod creează două liste: una pentru mașini și alta pentru semafoare. Lista de mașini, **vector<Car> cars;**, este umplută prin parcurgerea fiecărei intrări din **data["cars"]** (datele citite din fișierul JSON). Pentru fiecare mașină, se creează un obiect **Car**, se preiau valorile **id**, **x** și **y** din JSON folosind **.get<int>()** pentru a le converti în numere întregi, iar apoi obiectul **Car** este adăugat în lista **cars** folosind **cars.push_back(c)**. La fel se procedează și cu lista de semafoare.

```
for (auto& car : cars) {
    for (const auto& light : lights) {
        if (abs(car.y - light.y) < 20) {
            if (light.green_ns || car.x + 40 < light.x) {
                car.x += 50;
            }
            break;
        }
    }
}
```

Acest cod verifică fiecare mașină din lista **cars** și o compară cu fiecare semafor din lista **lights**. Pentru fiecare mașină, verifică dacă este aproape de un semafor, folosind expresia **abs(car.y - light.y) < 20**, adică dacă diferența dintre poziția y a mașinii și a semaforului este mai mică de 20 de pixeli.

Dacă mașina este aproape de semafor, programul verifică dacă semaforul este verde (**light.green_ns** este **true**) sau dacă mașina nu a ajuns încă în dreptul semaforului (**car.x + 40 < light.x**). Dacă una dintre aceste condiții este îndeplinită, mașina avansează pe axa **x** cu 50 de pixeli (**car.x += 50**;). După verificarea unui semafor, bucla interioară se oprește (**break**;) pentru că o mașină este afectată doar de primul semafor pe care îl întâlnește.

```
json out;
for (const auto& car : cars) {
    out["cars"].push_back({{"id", car.id}, {"x", car.x}, {"y", car.y}});
}
for (const auto& light : lights) {
    out["traffic_lights"].push_back({
        {"x", light.x}, {"y", light.y}, {"green_ns", light.green_ns}
    });
}
```

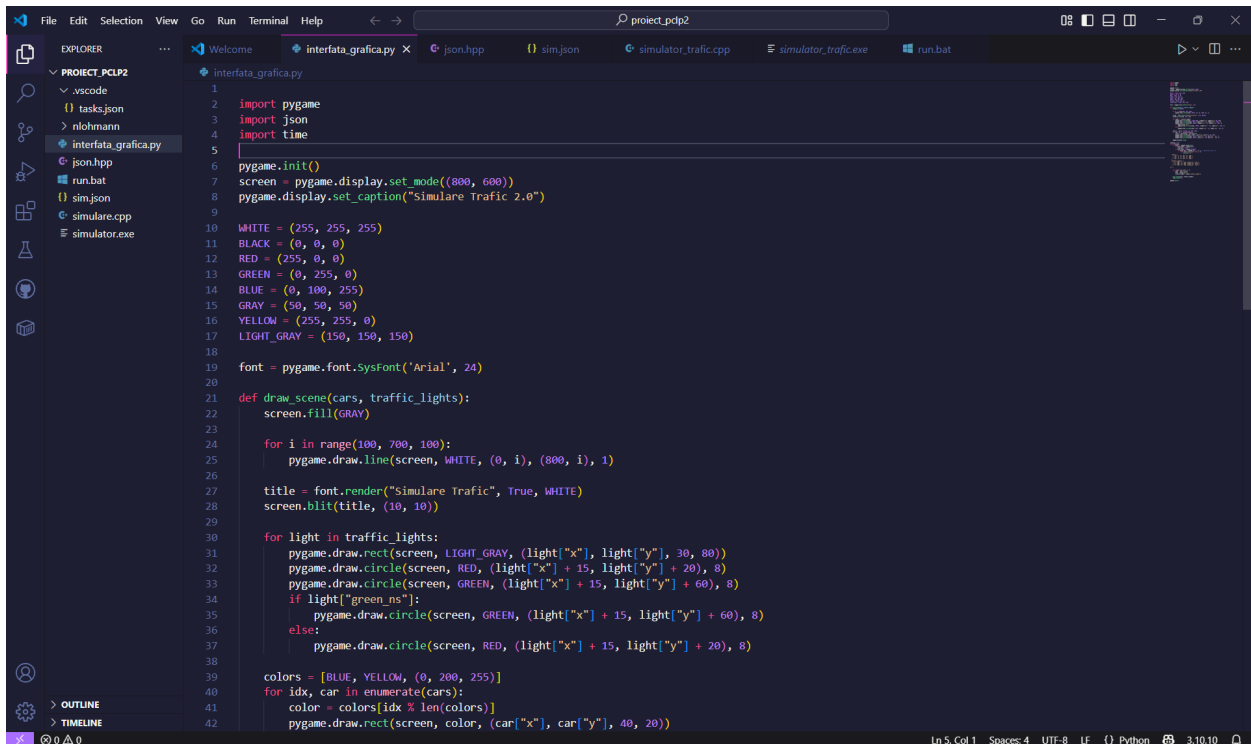
Această parte de cod creează un obiect JSON numit **out** și îl umple cu datele actualizate ale mașinilor și semafoarelor. Primul **for** parcurge lista de mașini **cars**, iar pentru fiecare mașină adaugă în **out["cars"]** un obiect cu valorile **id**, **x** și **y**. Aceste date sunt adăugate folosind **push_back** ca o listă de perechi **cheie-valoare**. Al doilea **for** face același lucru pentru semafoare.

```
ofstream o("sim.json");
o << out.dump(4);

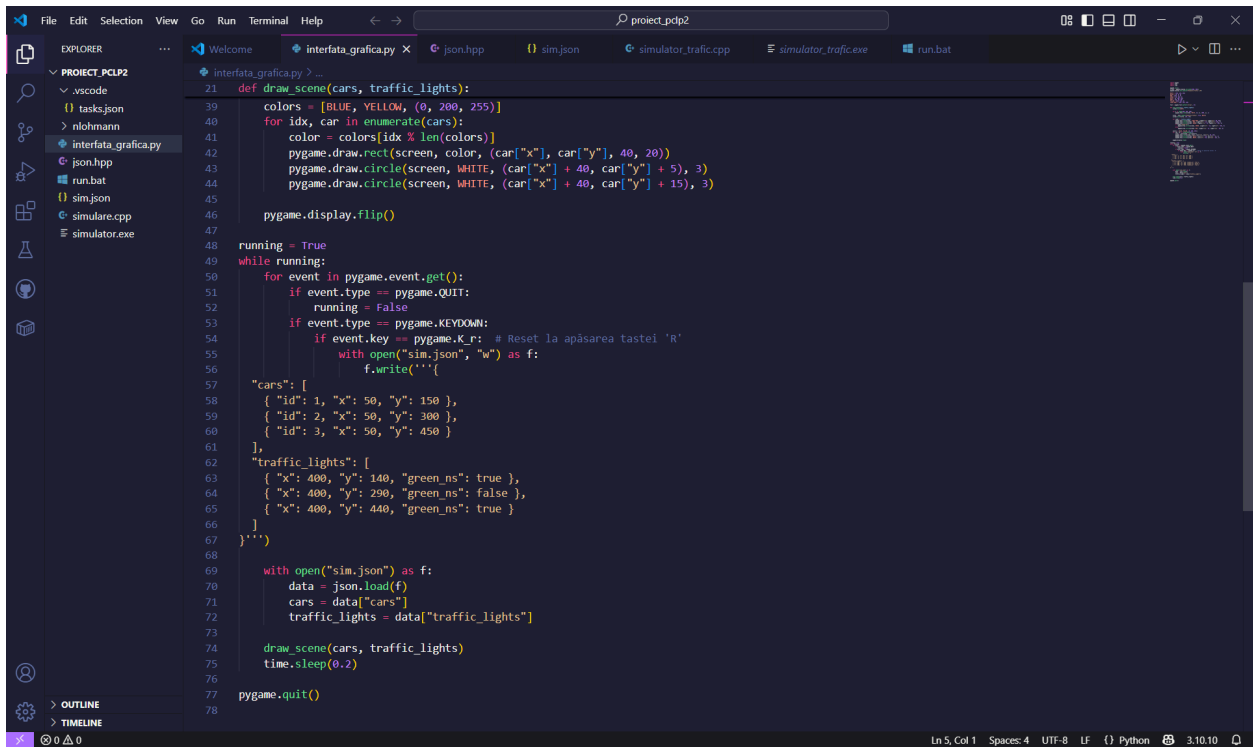
return 0;
}
```

Această parte a codului deschide un fișier numit **sim.json** folosind **ofstream o("sim.json")**;, pregătindu-l pentru a scrie în el. Linia **o << out.dump(4)**; scrie în fișier datele din obiectul JSON **out**, create anterior, iar **.dump(4)** asigură că fișierul va avea un format frumos și ușor de citit, cu indentare de 4 spații. La final, **return 0**; încheie funcția **main()**.

Interfata_grafica.py



```
1 interfata_grafica.py
2 import pygame
3 import json
4 import time
5
6 pygame.init()
7 screen = pygame.display.set_mode((800, 600))
8 pygame.display.set_caption("Simulare Trafic 2.0")
9
10 WHITE = (255, 255, 255)
11 BLACK = (0, 0, 0)
12 RED = (255, 0, 0)
13 GREEN = (0, 255, 0)
14 BLUE = (0, 100, 255)
15 GRAY = (50, 50, 50)
16 YELLOW = (255, 255, 0)
17 LIGHT_GRAY = (150, 150, 150)
18
19 font = pygame.font.SysFont('Arial', 24)
20
21 def draw_scene(cars, traffic_lights):
22     screen.fill(GRAY)
23
24     for i in range(100, 700, 100):
25         pygame.draw.line(screen, WHITE, (0, i), (800, i), 1)
26
27     title = font.render("Simulare Trafic", True, WHITE)
28     screen.blit(title, (10, 10))
29
30     for light in traffic_lights:
31         pygame.draw.rect(screen, LIGHT_GRAY, (light["x"], light["y"], 30, 80))
32         pygame.draw.circle(screen, RED, (light["x"] + 15, light["y"] + 20), 8)
33         pygame.draw.circle(screen, GREEN, (light["x"] + 15, light["y"] + 60), 8)
34         if light["green ns"]:
35             pygame.draw.circle(screen, GREEN, (light["x"] + 15, light["y"] + 60), 8)
36         else:
37             pygame.draw.circle(screen, RED, (light["x"] + 15, light["y"] + 20), 8)
38
39     colors = [BLUE, YELLOW, (0, 200, 255)]
40     for idx, car in enumerate(cars):
41         color = colors[idx % len(colors)]
42         pygame.draw.rect(screen, color, (car["x"], car["y"], 40, 20))
```



```
21 def draw_scene(cars, traffic_lights):
22     colors = [BLUE, YELLOW, (0, 200, 255)]
23     for idx, car in enumerate(cars):
24         color = colors[idx % len(colors)]
25         pygame.draw.rect(screen, color, (car["x"], car["y"], 40, 20))
26         pygame.draw.circle(screen, WHITE, (car["x"] + 40, car["y"] + 5), 3)
27         pygame.draw.circle(screen, WHITE, (car["x"] + 40, car["y"] + 15), 3)
28     pygame.display.flip()
29
30 running = True
31 while running:
32     for event in pygame.event.get():
33         if event.type == pygame.QUIT:
34             running = False
35         if event.type == pygame.KEYDOWN:
36             if event.key == pygame.K_r: # Reset la apăsarea tastei 'R'
37                 with open("sim.json", "w") as f:
38                     f.write('''
39 {
40   "cars": [
41     { "id": 1, "x": 50, "y": 150 },
42     { "id": 2, "x": 50, "y": 300 },
43     { "id": 3, "x": 50, "y": 450 }
44   ],
45   "traffic_lights": [
46     { "x": 400, "y": 140, "green_ns": true },
47     { "x": 400, "y": 290, "green_ns": false },
48     { "x": 400, "y": 440, "green_ns": true }
49   ]
50 }
51 ''')
52
53     with open("sim.json") as f:
54         data = json.load(f)
55         cars = data["cars"]
56         traffic_lights = data["traffic_lights"]
57
58     draw_scene(cars, traffic_lights)
59     time.sleep(0.2)
60
61 pygame.quit()
```

import pygame – importă biblioteca pentru grafică 2D, utilă în jocuri și simulări.

import json – pentru a citi și scrie fișiere .json, format folosit aici pentru a stoca pozițiile mașinilor și semafoarelor.

import time – pentru a introduce pauze (întârzieri) în execuție (cu time.sleep()).

```
pygame.init()
```

-----inițializează toate modulele pygame.

```
screen = pygame.display.set_mode((800, 600))
pygame.display.set_caption("Simulare Trafic 2.0")
```

---crează o

fereastră de 800x600 pixeli pentru afișarea grafică și setează titlul ferestrei.

```
def draw_scene(cars, traffic_lights):
    screen.fill(GRAY)

    for i in range(100, 700, 100):
        pygame.draw.line(screen, WHITE, (0, i), (800, i), 1)
```

---Funcția principală care desenează toate elementele: fundalul, liniile drumului, semafoarele și mașinile, umple fereastra cu culoarea gri (fundalul) și desenează linii orizontale albe (reprezintă drumurile/benzi) la fiecare 100 de pixeli pe verticală.

```
title = font.render("Simulare Trafic", True, WHITE)
screen.blit(title, (10, 10))
```

Afișează titlul "Simulare Trafic" în colțul din stânga sus.

```
for light in traffic_lights:
    pygame.draw.rect(screen, LIGHT_GRAY, (light["x"], light["y"], 30, 80))
    pygame.draw.circle(screen, RED, (light["x"] + 15, light["y"] + 20), 8)
    pygame.draw.circle(screen, GREEN, (light["x"] + 15, light["y"] + 60), 8)
    if light["green_ns"]:
        pygame.draw.circle(screen, GREEN, (light["x"] + 15, light["y"] + 60), 8)
    else:
        pygame.draw.circle(screen, RED, (light["x"] + 15, light["y"] + 20), 8)
```

- Desenează carcasa semaforului.
- Desenează ambele lumini (roșie și verde), inițial inactive.
- Activează una dintre lumini în funcție de starea semaforului (**green_ns**):

True → lumina **verde** pentru direcția nord-sud

False → lumina **roșie**

```
colors = [BLUE, YELLOW, (0, 200, 255)]
for idx, car in enumerate(cars):
    color = colors[idx % len(colors)]
    pygame.draw.rect(screen, color, (car["x"], car["y"], 40, 20))
    pygame.draw.circle(screen, WHITE, (car["x"] + 40, car["y"] + 5), 3)
    pygame.draw.circle(screen, WHITE, (car["x"] + 40, car["y"] + 15), 3)

pygame.display.flip()
```

- Alege culori diferite pentru mașini (alternativ, în funcție de index)
- Desenează corpul mașinii (un dreptunghi).
- Desenează două roți albe în partea dreaptă a mașinii.
- Actualizează fereastra pentru a afișa toate modificările.

```
running = True
while running: -----bucla principala ruleaza cat timp running este True
```



```
for event in pygame.event.get():
    if event.type == pygame.QUIT:
        running = False
```

----inchide fereastra daca utilizatorul

apasa pe butonul de inchidere.

```
if event.type == pygame.KEYDOWN:
    if event.key == pygame.K_r: # Reset la apăsarea tastei 'R'
```

---verifica daca a fost apasata tasta "R"

```
with open("sim.json", "w") as f:
    f.write('''{
    "cars": [
        { "id": 1, "x": 50, "y": 150 },
        { "id": 2, "x": 50, "y": 300 },
        { "id": 3, "x": 50, "y": 450 }
    ],
    "traffic_lights": [
        { "x": 400, "y": 140, "green_ns": true },
        { "x": 400, "y": 290, "green_ns": false },
        { "x": 400, "y": 440, "green_ns": true }
    ]
}''')
```

---scire în fișierul sim.json datele inițiale (resetarea de la inceput)

```
with open("sim.json") as f:
    data = json.load(f)
    cars = data["cars"]
    traffic_lights = data["traffic_lights"]
```

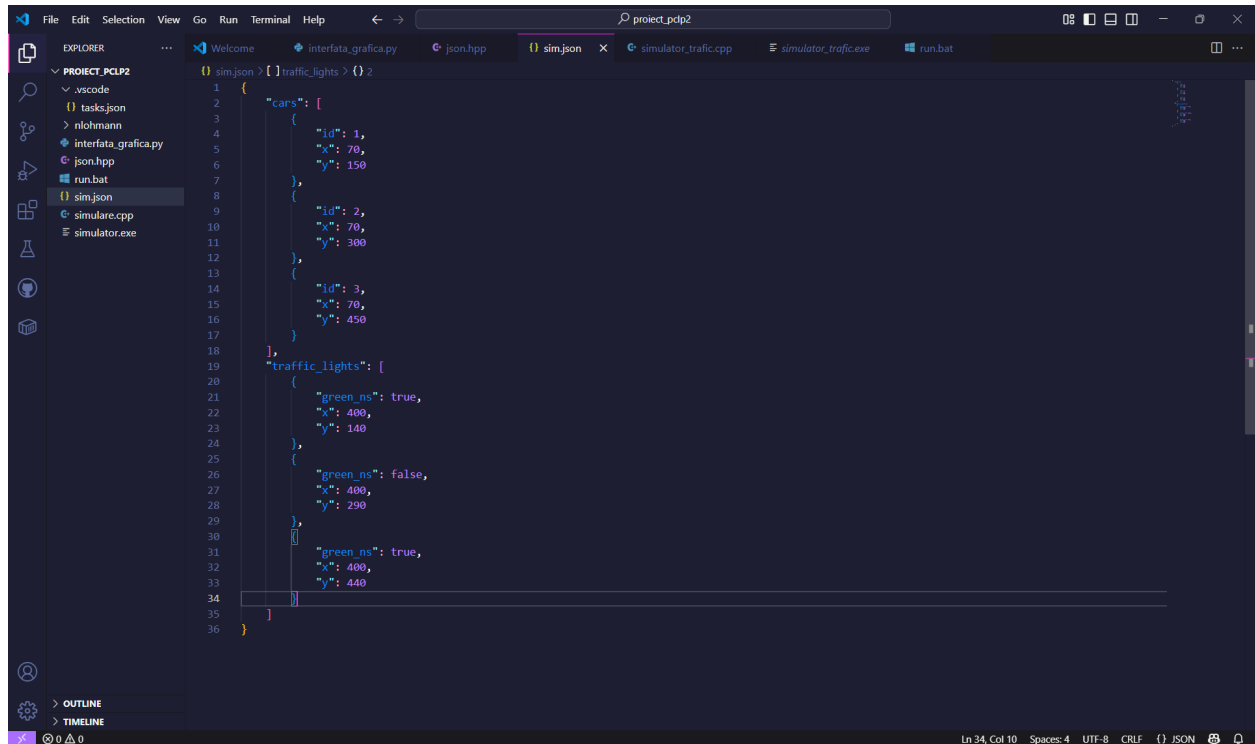
---deschide și

citește fișierul sim.json, extrage datele despre masini si semafoare.

```
draw_scene(cars, traffic_lights)
time.sleep(0.2)
```

Redesenare la fiecare 0.2 secunde (pentru a lăsa timp să se vadă schimbările).

Sim.json



```

1  {
2    "cars": [
3      {
4        "id": 1,
5        "x": 70,
6        "y": 150
7      },
8      {
9        "id": 2,
10       "x": 70,
11       "y": 300
12     },
13     {
14       "id": 3,
15       "x": 70,
16       "y": 450
17     }
18   ],
19   "traffic_lights": [
20     {
21       "green_ns": true,
22       "x": 400,
23       "y": 140
24     },
25     {
26       "green_ns": false,
27       "x": 400,
28       "y": 290
29     },
30     {
31       "green_ns": true,
32       "x": 400,
33       "y": 440
34     }
35   ]
36 }

```

Contine datele despre masinile si semafoarele prezente in “Simulatorul de trafic”.

```
1  {
2      "cars": [
3          {
4              "id": 1,
5              "x": 70,
6              "y": 150
7          },
8          {
9              "id": 2,
10             "x": 70,
11             "y": 300
12         },
13         {
14             "id": 3,
15             "x": 70,
16             "y": 450
17         }
18     ],
```

Fiecare obiect reprezintă o **mașină** cu următoarele informații:

- "id" – un identificator unic (1, 2, 3).
- "x" – poziția pe axa **orizontală** (70 pixeli de la marginea stângă).
- "y" – poziția pe axa **verticală** (150, 300, 450), adică pe ce bandă se află.

Deci sunt 3 mașini, toate aliniate la stânga ($x = 70$), fiecare pe o bandă diferită.

```
    ],  
    "traffic_lights": [  
      {  
        "green_ns": true,  
        "x": 400,  
        "y": 140  
      },  
      {  
        "green_ns": false,  
        "x": 400,  
        "y": 290  
      },  
      {  
        "green_ns": true,  
        "x": 400,  
        "y": 440  
      }  
    ]  
  ]  
}
```

Fiecare obiect este un **semafor**:

- "green_ns" – true sau false în funcție de culoarea afișată:
 - true → **verde** pentru direcția Nord–Sud (NS)
 - false → **roșu** pentru NS
- "x" – poziția semaforului pe axa orizontală (toate sunt la $x = 400$)
- "y" – poziția pe verticală, aliniată cu banda unde e plasat

Rezumat:

- Semaforul de pe banda 1 ($y = 140$) este **verde**.
- Semaforul de pe banda 2 ($y = 290$) este **roșu**.
- Semaforul de pe banda 3 ($y = 440$) este **verde**.

Cu ce ne ajuta pentru aplicatia noastra?

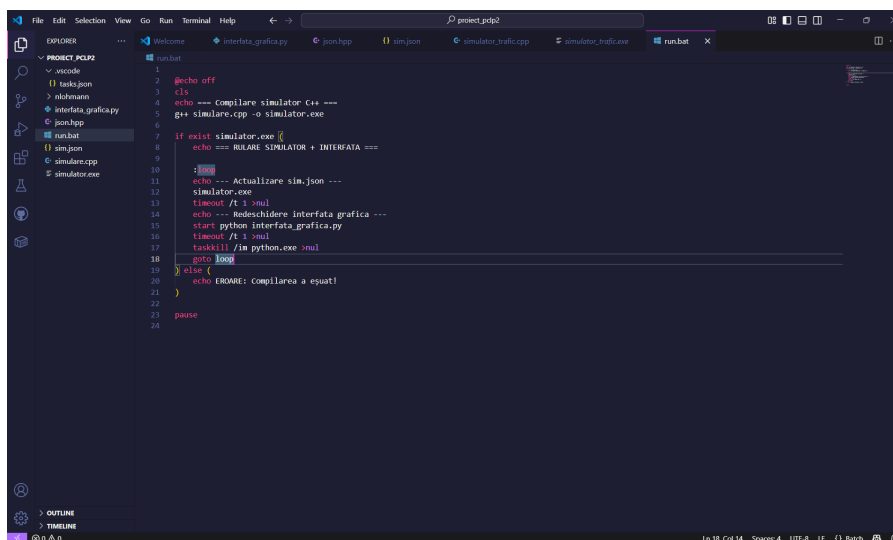
- Citește acest fișier și folosește informațiile pentru a desena:
 - Mașinile în pozițiile lor inițiale
 - Semafoarele și starea lor (verde sau roșu)
- Dacă apeși tasta **R**, fișierul este rescris cu aceste valori inițiale (resetare).

Run.bat

Fișierul **run.bat** este un script care automatizează tot procesul de rulare a proiectului. Acesta începe prin a ascunde comenzile din terminal cu **@echo off** și curăță ecranul cu **cls** pentru a avea un aspect curat. Afișează un mesaj care anunță că urmează să compileze programul scris în C++ și folosește comanda **g++ simulare.cpp -o simulator.exe** pentru a transforma fișierul **simulare.cpp** într-un executabil numit **simulator.exe**.

După compilare, scriptul verifică dacă executabilul a fost creat cu succes. Dacă da, afișează un mesaj că urmează să ruleze simulatorul și interfața, apoi intră într-un ciclu care face următoarele: rulează **simulator.exe** pentru a actualiza pozițiile mașinilor, așteaptă o secundă, pornește interfața grafică scrisă în Python pentru a desena traficul, așteaptă din nou o secundă, închide automat interfața grafică, și reia procesul de la început.

Dacă programul nu a fost compilat cu succes, scriptul afișează un mesaj de eroare care anunță că a eșuat compilarea. La final, comanda **pause** oprește scriptul pentru a putea citi mesajele afișate. Pe scurt, **run.bat** este ca un buton de start pentru proiect.



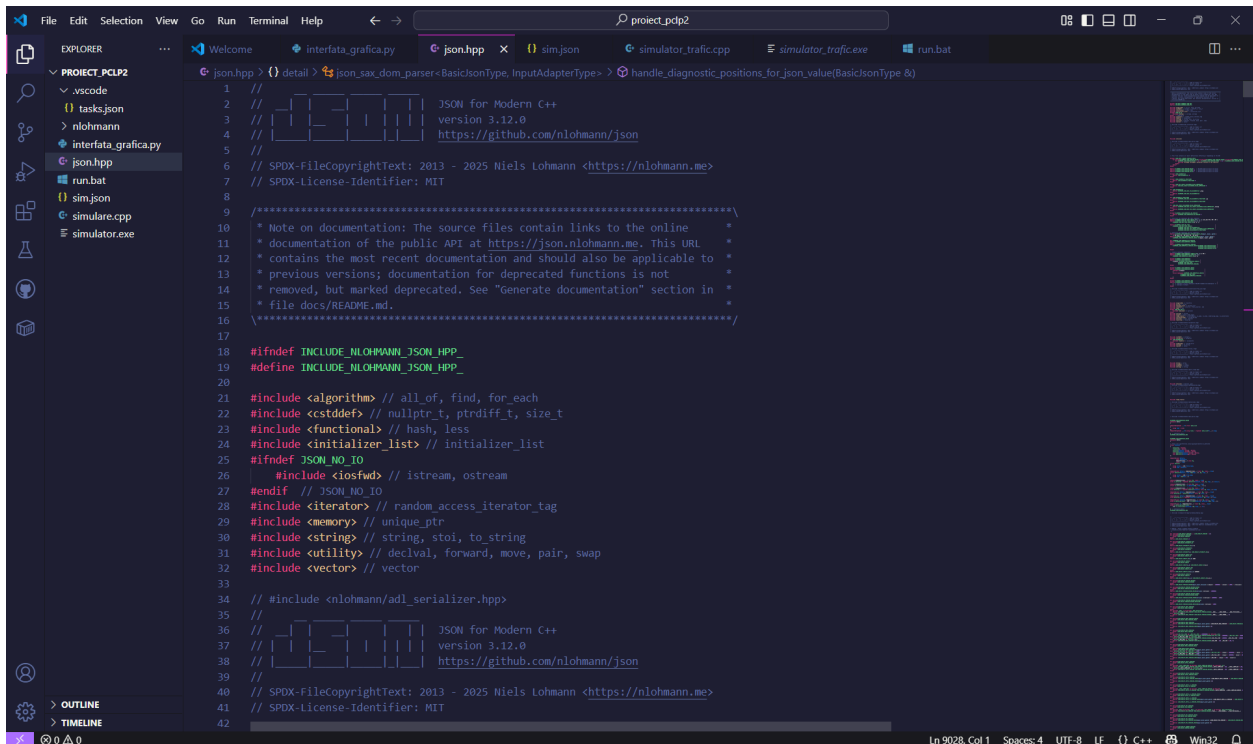
```

1 @echo off
2 cls
3
4 echo --- Compile simulator C++ ---
5 g++ simulare.cpp -o simulator.exe
6
7 if exist simulator.exe (
8     echo --- RULEAZA SIMULATOR + INTERFAȚA ---
9
10    :loop
11    echo --- Actualizare sim.json ---
12    simulator.exe
13    timeout /t 1 /nul
14    echo --- Redeschidere interfața grafică ---
15    start python interfața_grafică.py
16    timeout /t 1 /nul
17    taskkill /im python.exe /nul
18    goto loop
19 ) else (
20    echo ERORARE: Compilarea a eșuat!
21 )
22
23 pause
24
  
```

Json.hpp

Fișierul **json.hpp** este o bibliotecă externă care permite programului C++ să lucreze cu fișiere în format JSON. Acesta conține definiții și funcții pregătite care transformă un text JSON (un format folosit pentru stocarea datelor) într-un obiect pe care programul îl poate înțelege și folosi.

În proiectul nostru, **json.hpp** este inclus în fișierul **simulare.cpp** prin comanda **#include "json.hpp"**. Acest lucru înseamnă că toate funcțiile și clasele din **json.hpp** pot fi folosite direct în program. De exemplu, în **simulare.cpp**, folosim **json data;** pentru a crea o variabilă JSON și **f >> data;** pentru a citi datele din fișierul **sim.json**.



```

1 //
2 // [nlohmann] JSON for Modern C++
3 // [nlohmann] version 3.12.0
4 // [nlohmann] https://github.com/nlohmann/json
5 //
6 // SPDX-FileCopyrightText: 2013 - 2025 Niels Lohmann <https://nlohmann.me>
7 // SPDX-License-Identifier: MIT
8
9 /*****
10  * Note on documentation: The source files contain links to the online
11  * documentation of the public API at https://json.nlohmann.me. This URL
12  * contains the most recent documentation and should also be applicable to
13  * previous versions; documentation for deprecated functions is not
14  * removed, but marked deprecated. See "Generate documentation" section in
15  * file docs/README.md.
16  */
17
18 #ifndef INCLUDE_NLOHMANN_JSON_HPP_
19 #define INCLUDE_NLOHMANN_JSON_HPP_
20
21 #include <algorithm> // all_of, find, for_each
22 #include <cstdint> // nullptr_t, ptrdiff_t, size_t
23 #include <functional> // hash, less
24 #include <initializer_list> // initializer_list
25 #ifndef JSON_NO_IO
26 #include <iosfwd> // istream, ostream
27 #endif // JSON_NO_IO
28 #include <iterator> // random_access_iterator_tag
29 #include <memory> // unique_ptr
30 #include <string> // string, stoi, to_string
31 #include <utility> // declval, forward, move, pair, swap
32 #include <vector> // vector
33
34 // #include <nlohmann/adl_serializer.hpp>
35
36 // [nlohmann] JSON for Modern C++
37 // [nlohmann] version 3.12.0
38 // [nlohmann] https://github.com/nlohmann/json
39 //
40 // SPDX-FileCopyrightText: 2013 - 2025 Niels Lohmann <https://nlohmann.me>
41 // SPDX-License-Identifier: MIT
42

```

Sfârșit!

Vă mulțumim pentru atenția acordată!

Fila 2

