3D Ray Tracing

Mihai Gheoace 343C1 Cezar Crăciunoiu 343C1

Specificația cerințelor

Introducere

Proiectul 3D Ray Tracing are ca scop rularea algoritmului de ray tracing într-un spaţiu tridimensional pe un număr limitat de obiecte. Deoarece acest algoritm este unul neperformant atunci cand nu există suport hardware, am dorit să facem o comparaţie a rulării acestuia, atât pe CPU cât și pe GPU.
Pentru rasterizarea imaginilor se pune în evidenţă arhitectura SISD (la rulare s-a folosit OpenMP pentru a paraleliza uşor pe CPU, deoarece timpii de rulare erau foarte mari. Se poate alege totuşi să se ruleze SISD în loc de SIMD) a procesorului în raport cu arhitectura SIMD a GPU-ului. La final se va face o comparație a performanţei de rulare între cele două versiuni.

Descriere generală

Pentru rularea proiectului se utilizează un procesor Intel 7700k cu o placă grafică Nvidia GTX 1060. La nivel software am utilizat Visual Studio Enterprise 2019 și Framework-ul de laborator oferit de echipa de SPG. Am folosit versiunea 440 de OpenGL pentru rularea shader-erelor.
Am ales acest proiect pentru a sublinia rolul crucial al plăcii grafice în execuţia unor proiecte ce folosesc intensiv resursele calculatorului într-o paradigmă puternic paralelizabilă.

Sistemul propus

Algoritmul poate fi asimilat în alte proiecte care sunt construite în Framework-ul de laborator SPG. Cerinţele de sistem ale proiectului sunt un procesor ce suporta versiunea 440 de OpenGL cât şi Framework-ul de laborator care sa fie configurat corespunzător rulării. De asemenea, pentru rularea Framework-ului cu Visual Studio este nevoie ca sistemul de operare să fie Windows.
La general, proiectul va afişa trei scene de complexitate diferită, ce vor fi procesate atât pe GPU cât şi pe CPU. La comanda utilizatorului se va face trecere de pe CPU pe GPU şi invers. Totodată, acesta va putea să treacă prin scene folosind tastele numerice.

Caz de utilizare

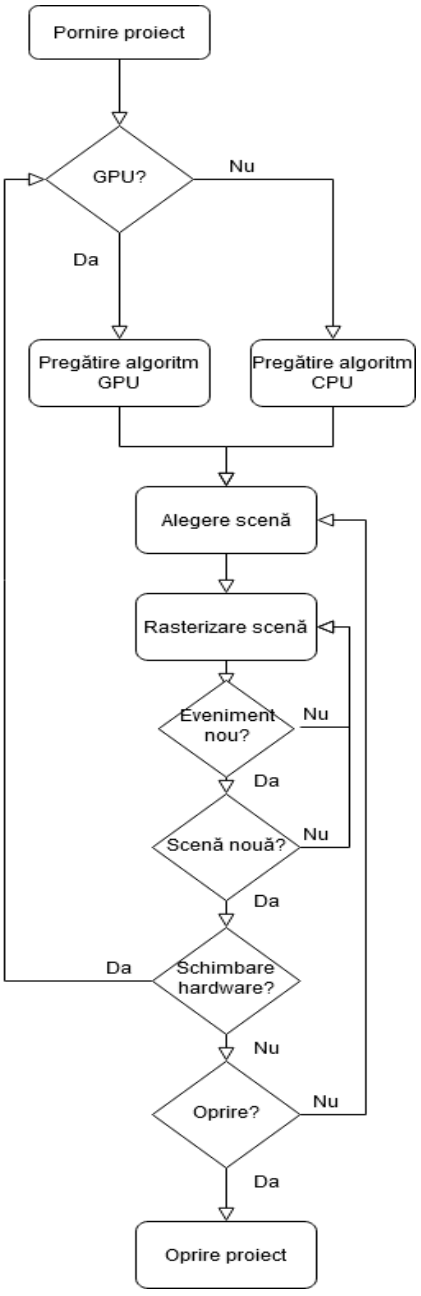
Precondiţii:

- Visual Studio 2019 (minim) instalat pe Windows (> 7)
- Framework de laborator bine configurat
- Proiect construit

Postcondiţii:

- Se observă în timp real diferenţa dintre CPU şi GPU

În organigrama alăturată se observă fluxul proiectului. Utilizatorul porneşte proiectul şi alege dacă să ruleze pe GPU sau CPU. Programul realizează în spate pregătirea algoritmului şi apoi utilizatorul trebuie să aleagă una din cele trei scene pentru a le reda. Scena este afişată în timp real, până când utilizatorul apasă o tastă numerică pentru a alege o scenă nouă sau una din tastele '-' şi '+' pentru a alege CPU, respectiv GPU. Dacă se apasă pe 'X' programul se opreşte.



Arhitectura

Prezentare generala sumară

Proiectul este făcut să meargă pe orice calculator ce poate rula framework-ul de laborator SPG, implicit Visual studio 2019 (cu OpenGL 440). De asemenea, o placă video este puternic recomandată pentru rularea decentă a proiectului. Nu sunt necesare alte elemente adiţionale pentru execuţia proiectului.

Decompoziţia în subsisteme

După cum s-a precizat, proiectul este structurat în două versiuni, una ce rulează pe CPU şi una ce rulează pe GPU. Astfel, şi implementările sunt diferite. Cele două încep comun, cu iniţializarea programului folosind framework-ul de laborator. Tot în mod comun se realizează cronometrarea şi contorizarea cadrelor pe secundă, respectiv interacţionarea cu utilizatorul. Sistemele se despart în partea de implementare propriu-zisă, o variantă fiind scrisă în limbajul GLSL, iar alta direct în CPP. Putem vorbi astfel de un subsistem pentru GPU şi unul pentru CPU.

Cazuri de utilizare limită

În anumite cazuri, cum ar fi atunci când camera este poziţionată incorect în interiorul obiectelor, sistemul nu mai poate trasa corespunzător razele, deoarece acestea nu ajung niciodată în interior. Astfel, se ajunge la o rulare la capacitate maximă, chit că nu ar fi nevoie.

Detalii de implementare

Discuţia în legătură cu implementarea va fi împărţită în două. Chit că ambele implementări fac acelaşi lucru, acestea se aseamănă doar conceptual. Atât limbajul, cât şi abordarea sunt diferite. De asemenea, deoarece generarea pe procesor este foarte lentă, masurarea pe acesta se va face în cadre pe oră, pentru a obţine valori apropiate de cele de pe GPU (chit că unitatea de măsură este diferită).

Implementare GPU

Pentru această implementare, s-a trimis către rasterizare de pe CPU un singur quad. Acesta a fost amplasat folosind coordonate dispozitiv normalizate în faţa camerei în Vertex Shader. Astfel încât, putem spune că, algoritmul Ray Tracing generează tot imagini şi le aplică pe quad, dar, deoarece acest lucru se face foarte repede, se obţine ideea de video. Tot în Vertex Shader se calculează 2 valori *eye_pos* şi *ray_direction_from_vertex,* ce vor fi folosite pentru trasarea corectă a razelor.
Toate calculele sunt realizate în Fragment Shader. Pentru a se realiza obiectele, s-au folosit mai multe structuri, prin intermediul cărora s-au definit următoarele forme geometrice: paralelipiped/cub, sferă, cilindru, con. Obiectele nu urmează pipeline-ul clasic OpenGL, fiind generate direct în Fragment Shader. Acestea reprezintă zone de intersecţie cu razele şi absorb diferite lungimi de undă (culori) din lumină. Ce rămâne este culoarea obiectelor.
În rest, se urmează algoritmul clasic de trasare a razelor. Se calculează intersecţiile, se verifică dacă un obiect este în umbra altui obiect, se calculează lumina Phong de tipul ambientală/difuză/speculară, şi, în final, se verifică dacă obiectul este transparent sau reflectorizant pentru a se modifica razele.
Mai jos sunt discutate funcţiile ce se ocupă cu intersecţia obiectelor cu razele. Aceasta este partea cea mai intensivă a codului. Toate aceste funcţii întorc cel mai apropiat punct de intersecţie cu raza.

Intersecţie Paralelipiped

Se foloseşte algoritmul Axis-aligned Bounding Boxes (AABB) pentru a genera uşor şi rapid un paraleliped (inclusiv Cub). Aceasta este, de departe, cea mai rapidă şi uşoară metodă de intersecţie a obiectelor.

Intersecţie Sferă

Pentru a calcula intersecţia cu sfera s-a utilizat ecuaţia de gradul 2 prin care s-au obţinut 2 rădăcini. Dintre acestea se alege cea mai mare. Această intersecţie este ceva mai înceată, dar, recuperează prin faptul ca este succintă.

Intersecţie Cilindru

Pentru generarea cilindrului s-a urmat modelul sferei. Mai întâi se generează capacele cilindrelor sub formă de cercuri şi apoi se foloseşte ecuaţia de gradul 2 în funcţie de X şi Y, pentru a se genera un cilindru infinit, care apoi este crestat la capete în funcţie de dimensiuni.

Intersecţie Con

Se urmează modelul cilindrului, dar ecuaţia de gradul 2 este ridicată la pătrat pentru a genera planurile în formă de X. Apoi se decupeaza forma până la dimensiunea dorită şi se pune capacul conului. De asemenea se elimină jumătatea mai mare decât centrul din figură, pentru a se înjumătăţi.

Implementare CPU

Codul de pe CPU este împărţit în mai multe fişiere, întrucât are o structură OOP, plecând de la propunerile din cartea Ray Tracing in One Weekend de Peter Shirley. Funcţiile de calculare a unui hit între o rază şi un obiect sunt similare cu abordarea de pe GPU(deci intersectiile), însă observăm câteva diferenţe de abordare:

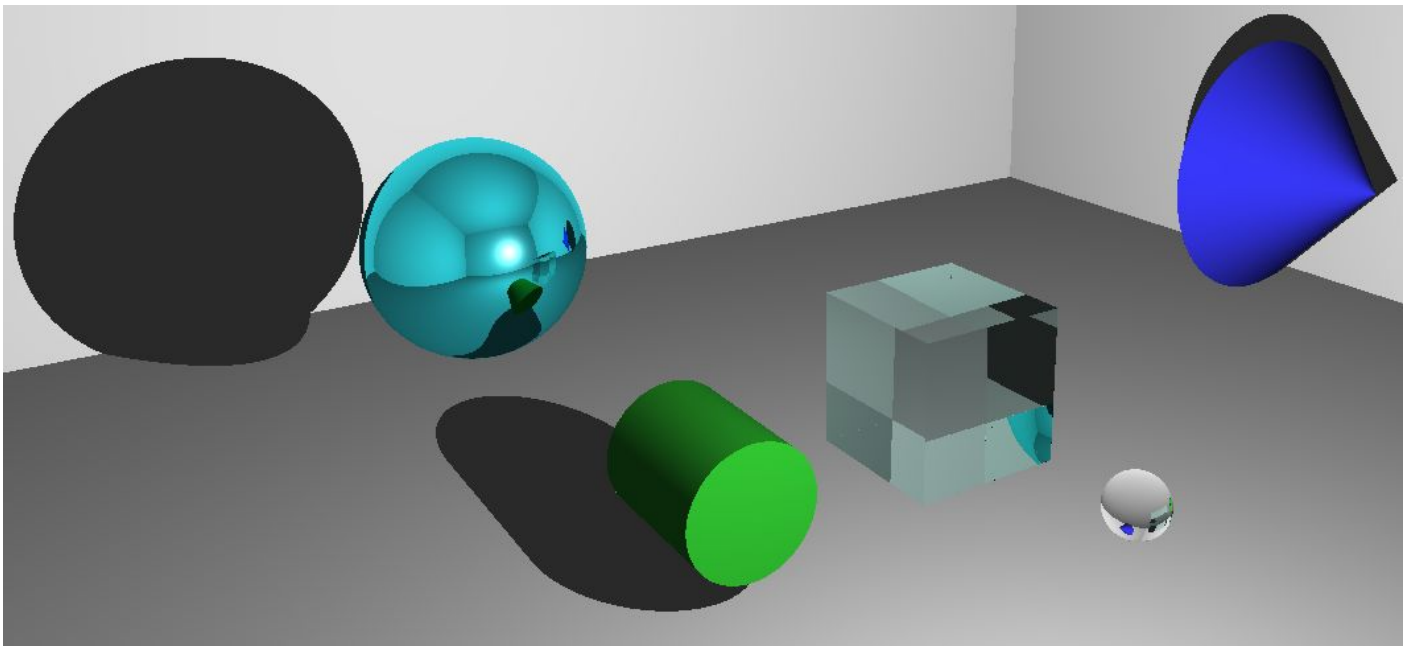
- toate obiectele sunt parcurse în aceeaşi buclă, în fiecare pas calculându-se normala, care este orientată invers faţă de sensul razei incidente (pe GPU este întotdeauna spre exteriorul obiectului).
- normala este calculată în acelaşi timp în care se descoperă cel mai apropiat obiect de originea razei.
- se realizează antialiasing prin realizarea de multiple calcule per pixel prin for-ul cel mai intern din funcţia Ray.
- codul este paralelizat pe CPU prin openMP(versiunea 2.0 de openMP pentru Visual Studio 2019) - se obţine speedup de ~3.5 pentru 4 core-uri (8 logice) faţă de versiunea serială ceea ce foloseşte la maxim procesorul. Acest pas este opţional, dar, în cele mai multe cazuri timpii de rulare sunt foarte mari, deci se recomandă paralelizarea.
- se declară adâncimea de vizualizare a camerei explicit.

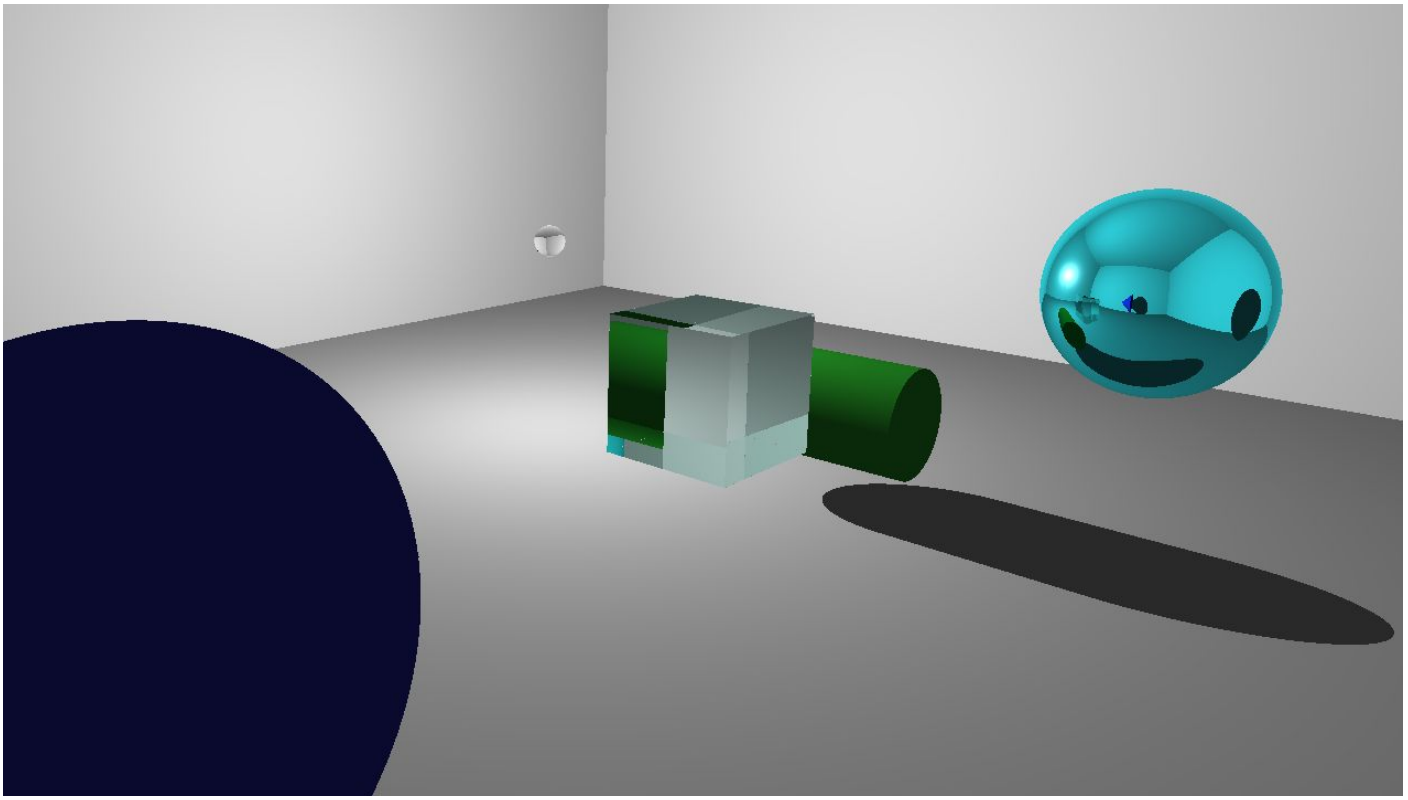
Screenshots

Scena 1

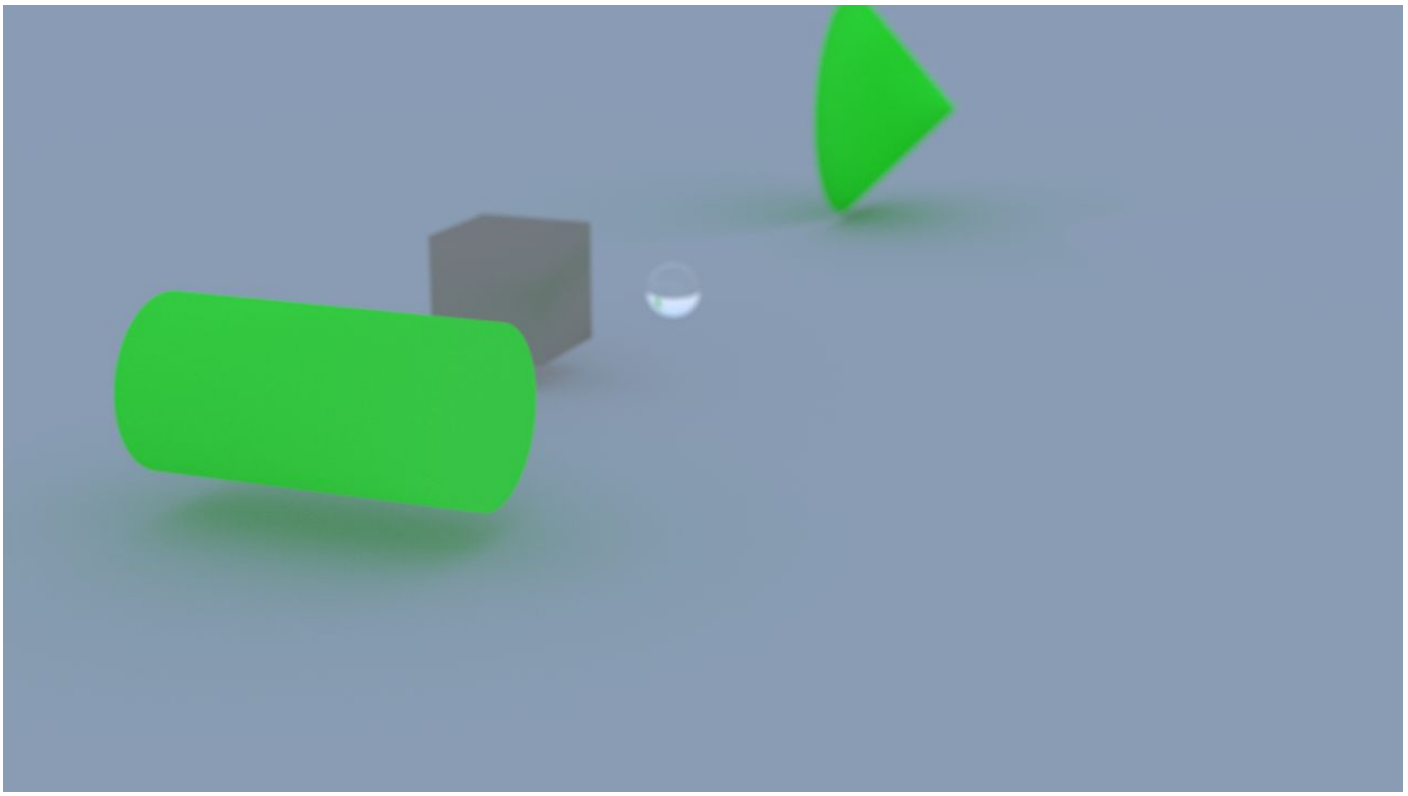
Scena 1 este folosită pentru a exemplifica obiectele şi proprietăţile acestora într-o lume ce foloseşte RayTracing.

GPU





CPU

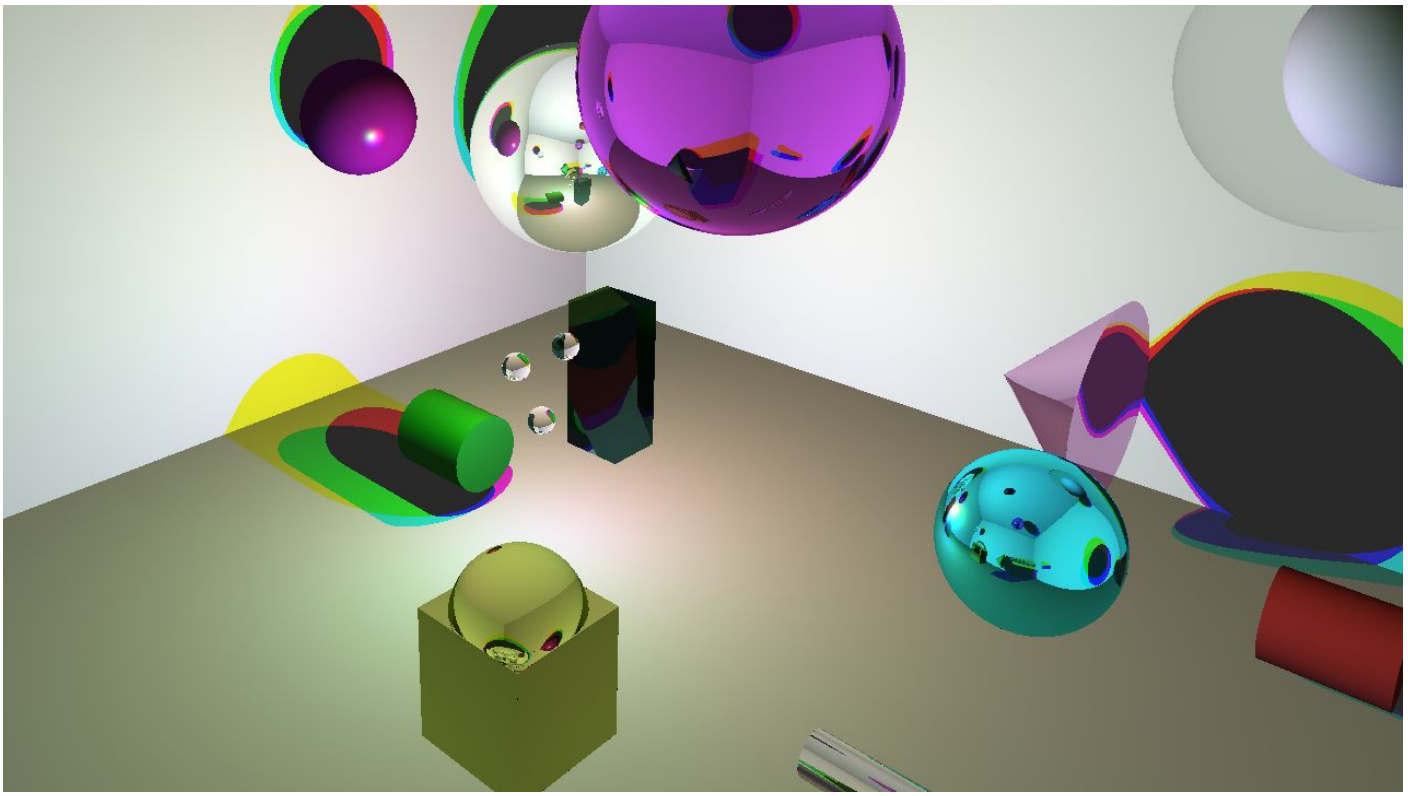


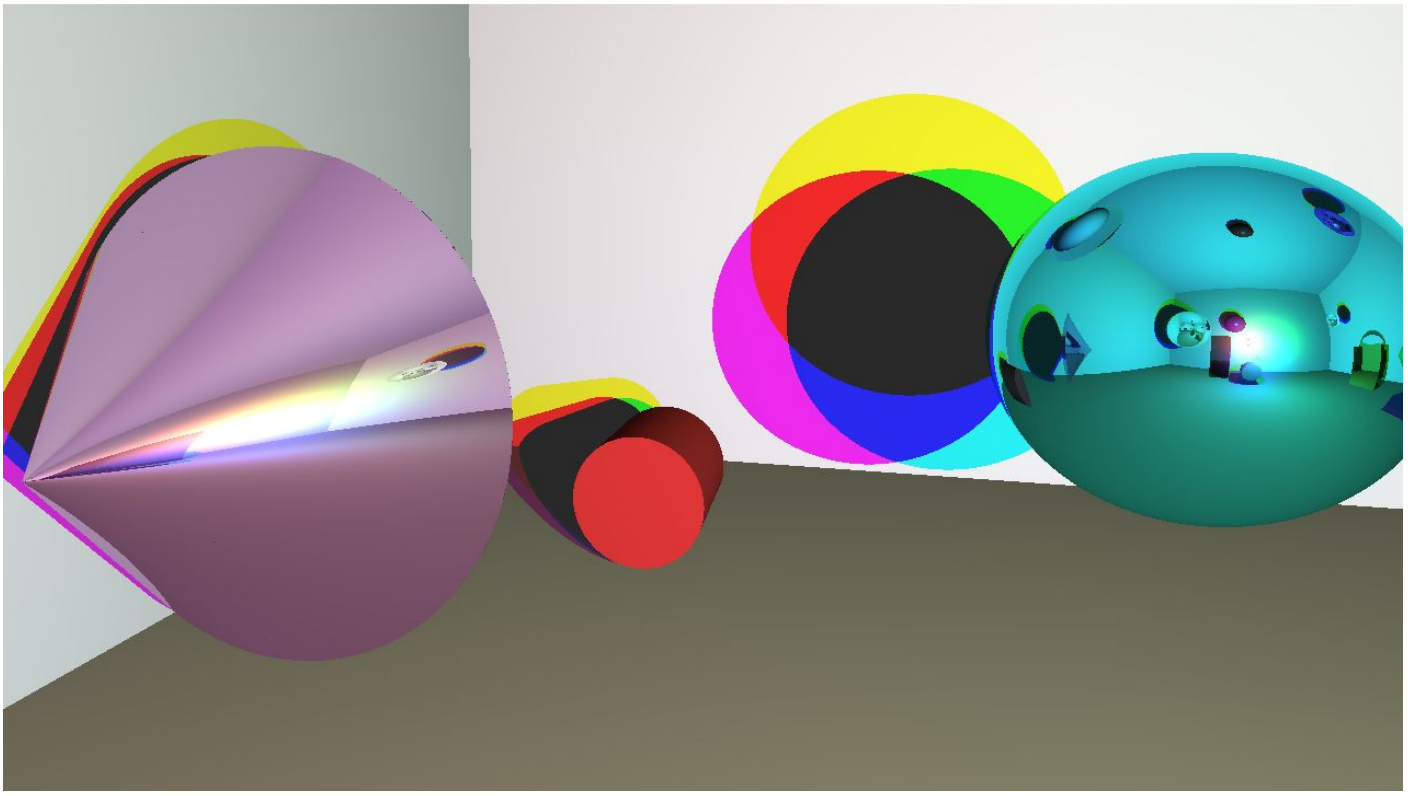
Scena 2

Această scenă încarcă majoritatea combinaţiilor de obiecte şi 3 surse de lumină, pentru a genera o lume standard ca dificultate de randare.

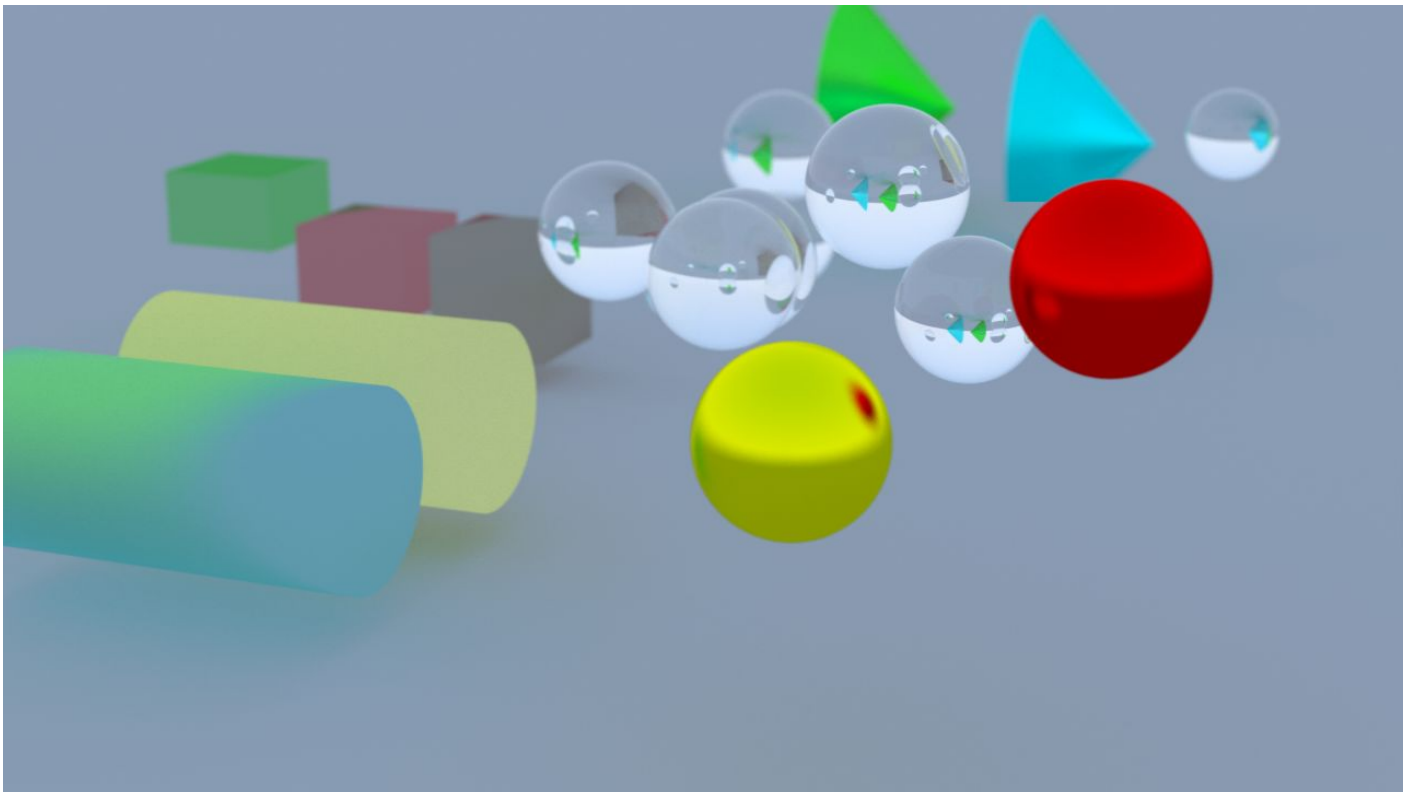
GPU

Sursele de lumină sunt de tip R, G şi B. Astfel că, deoarece umbra a fost calculată ca opusul luminii, se obţin umbre în cercul de culori CMYK. Deşi nu foarte corect, acest lucru a fost păstrat pentru aspectul vizual plăcut şi rezultatele interesante.





CPU

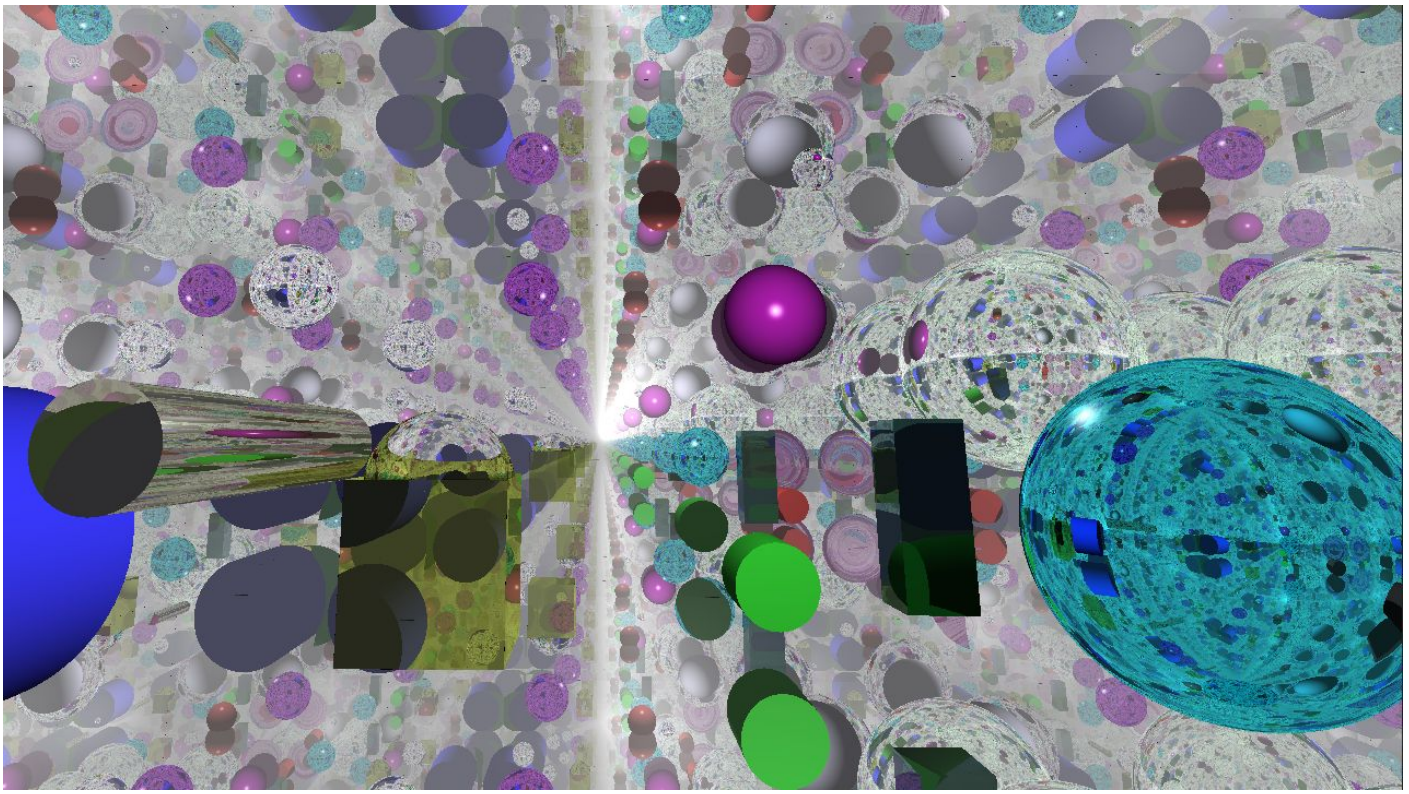


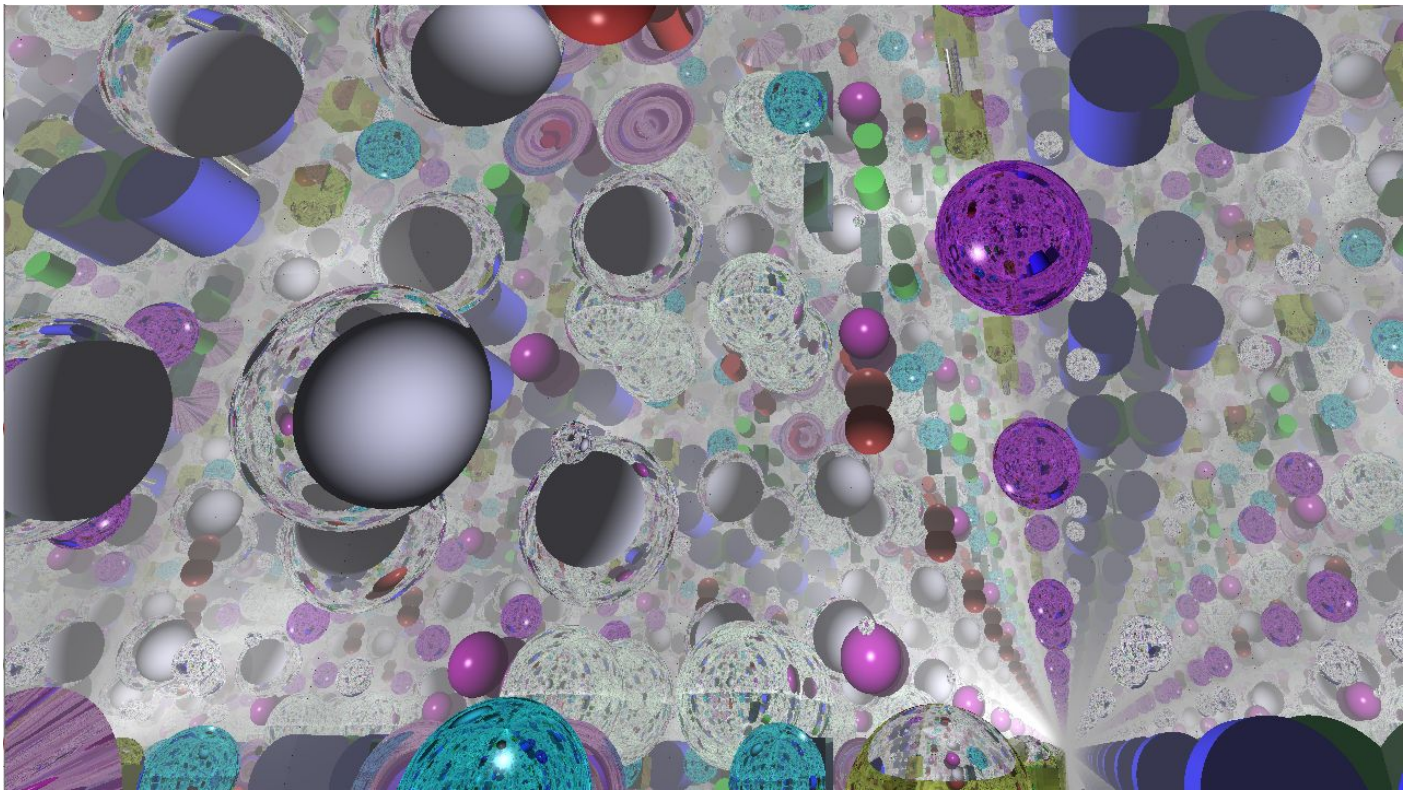
Pe CPU este pus accentul pe numărul mare de obiecte cu reflexii sub ele realiste şi uşor blur în fundal pentru efectul de depth-of-field.

Scena 3

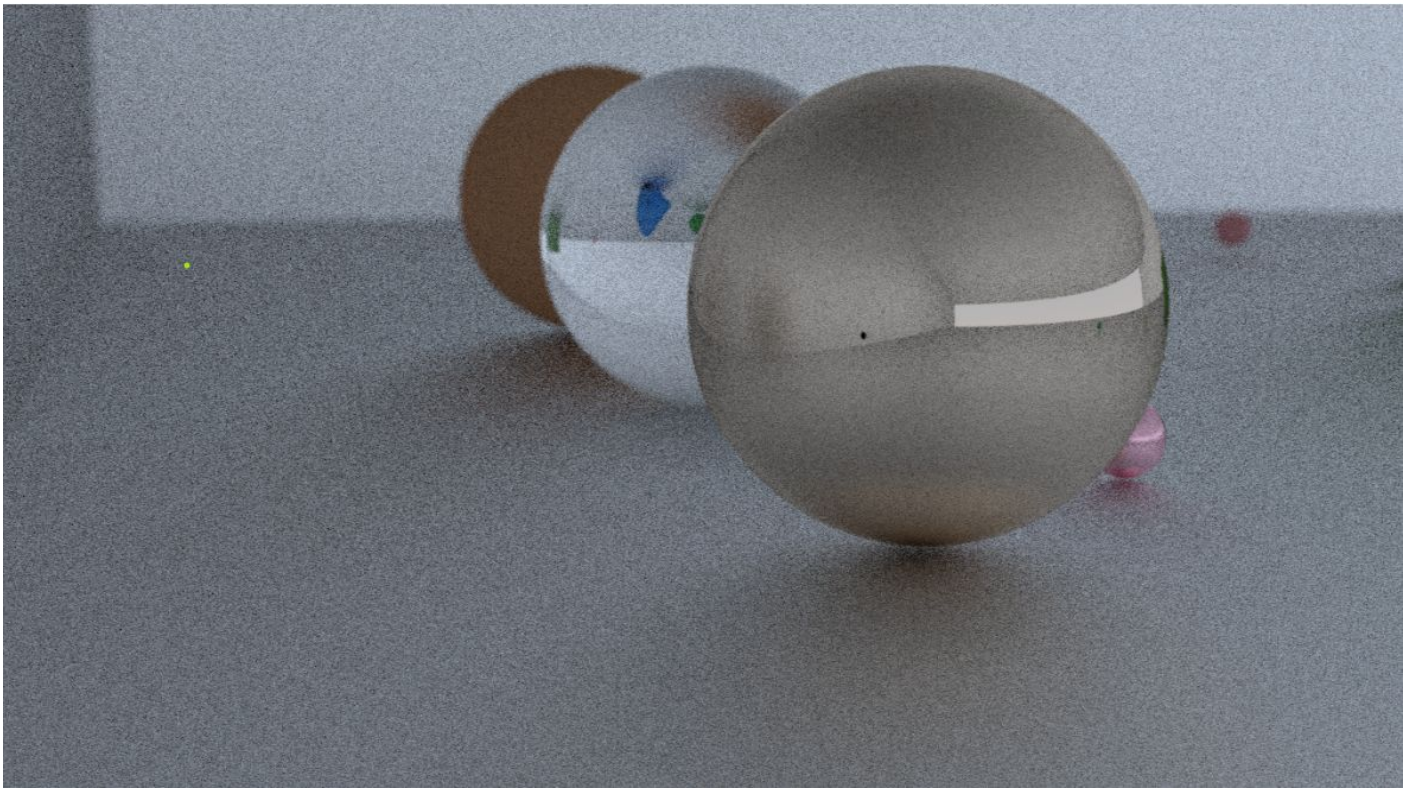
La fel de multe obiecte ca în a doua scenă, dar, mediul înconjurător reflectă lumina aproape perfect. Se obţine astfel un efect interesant şi foarte consumator de resurse.

GPU





CPU

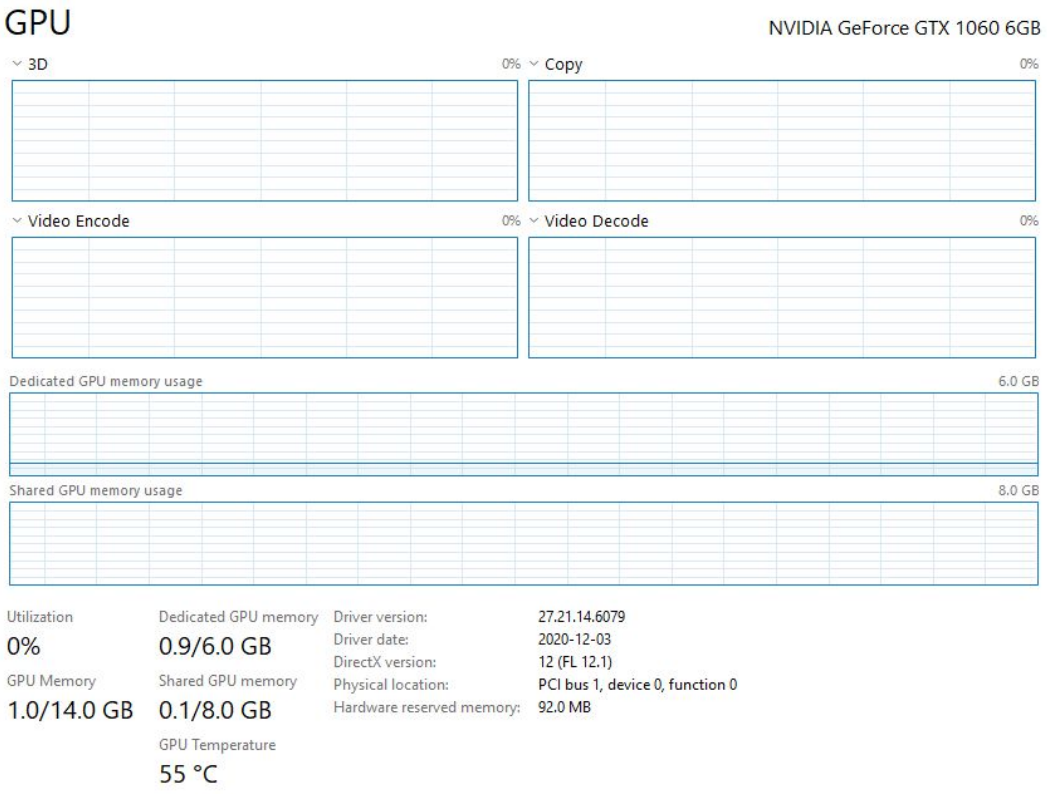


Testare

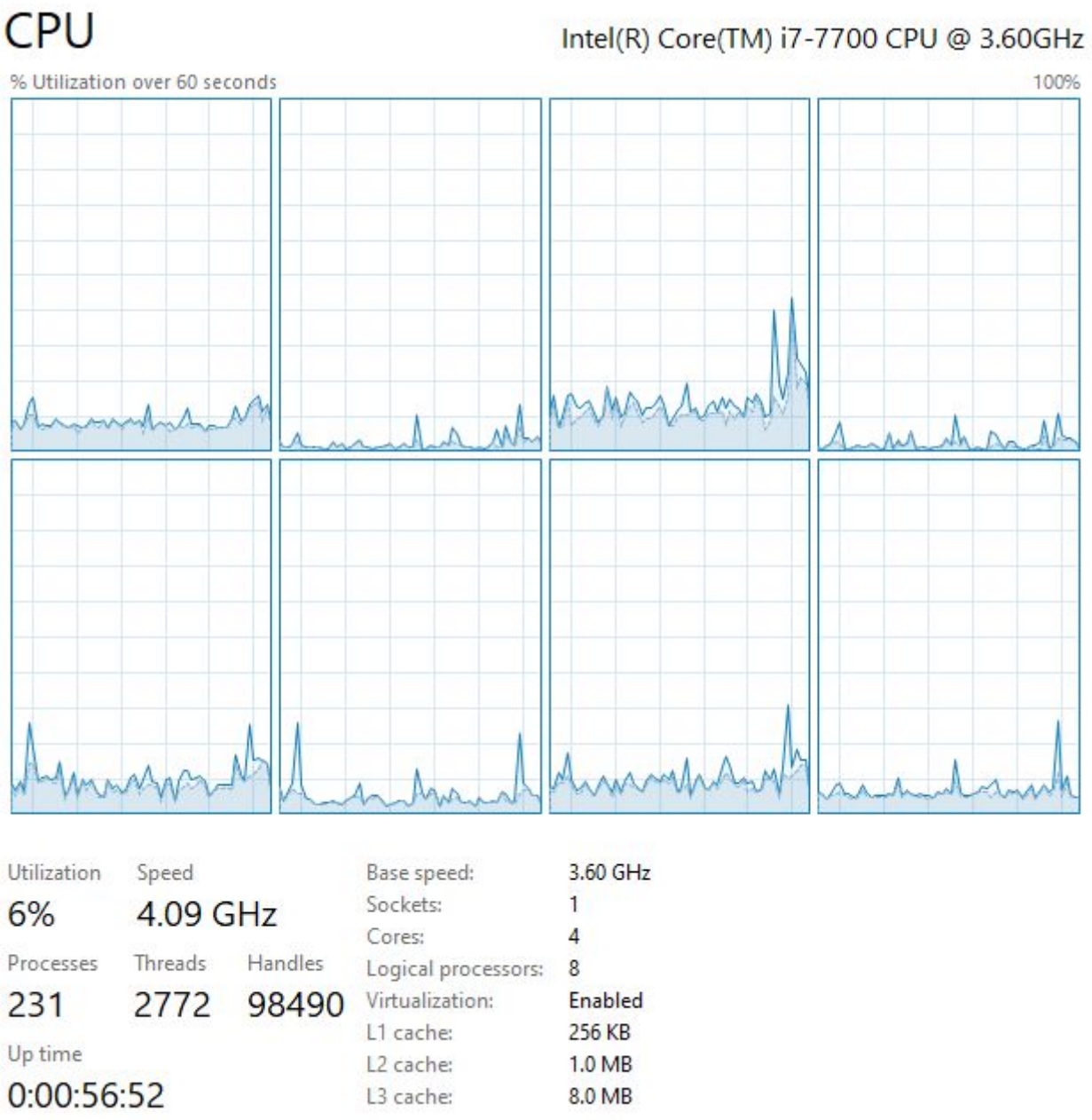
Pentru a testa rularea programului am verificat rularea celor 3 scene și am comparat încărcarea CPU/GPU-ului, respectiv FPH/FPS-ul în timpul rulării. Pe GPU s-au folosit 40 de bounce-uri maxime.

Se obțin următoarele măsurători instantanee pentru cele 3 scene.

Idle GPU

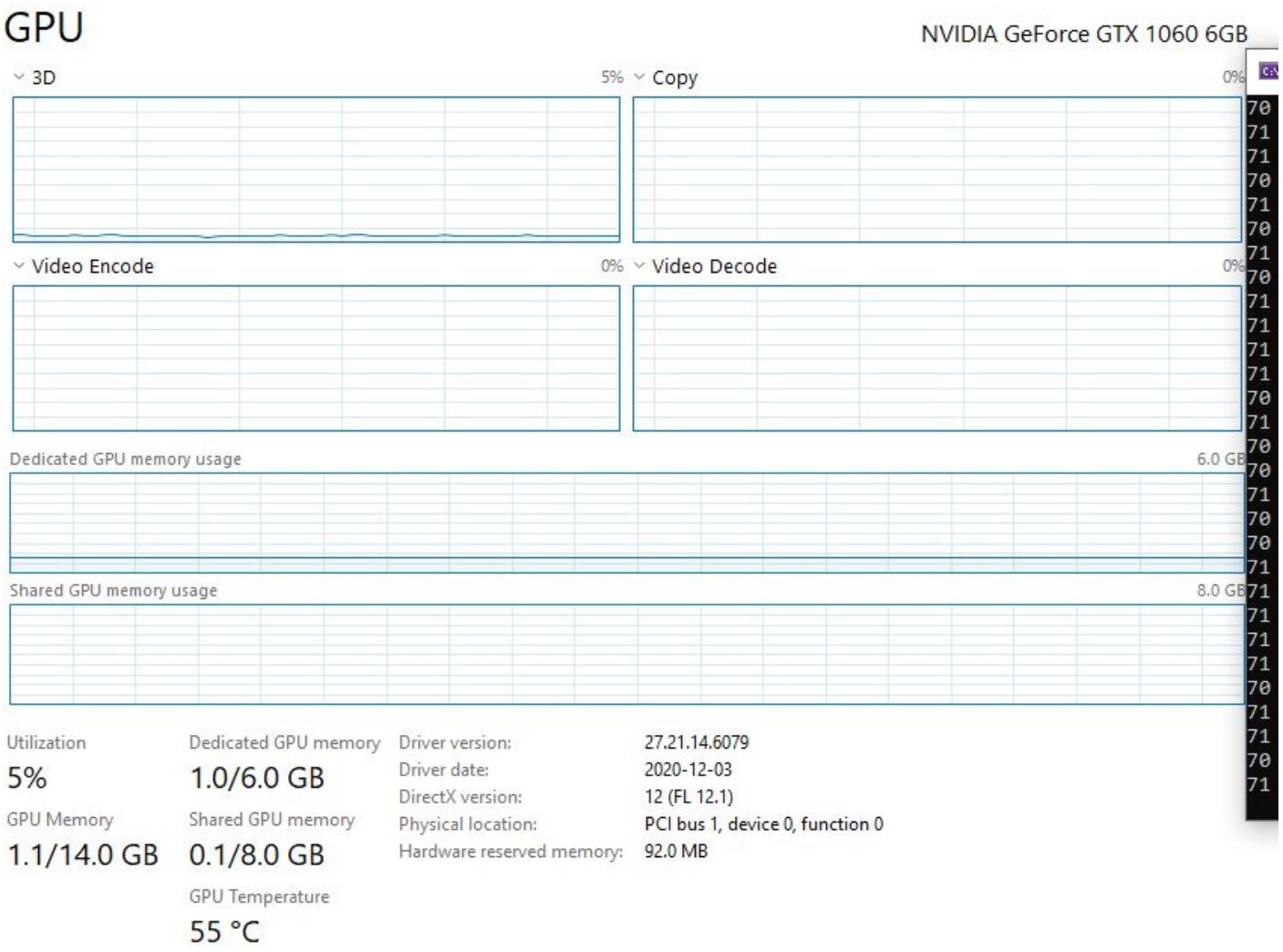

GPU
NVIDIA GeForce GTX 1060 6GB
3D
0%
Copy
0%
Video Encode
0%
Video Decode
0%
Dedicated GPU memory usage
6.0 GB
Shared GPU memory usage
8.0 GB
Utilization
0%
Dedicated GPU memory
0.9/6.0 GB
Driver version:
27.21.14.6079
Driver date:
2020-12-03
DirectX version:
12 (FL 12.1)
Physical location:
PCI bus 1, device 0, function 0
Hardware reserved memory:
92.0 MB
GPU Memory
1.0/14.0 GB
Shared GPU memory
0.1/8.0 GB
GPU Temperature
55 °C

Idle CPU


CPU
Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz
% Utilization over 60 seconds
100%
Utilization
6%
Speed
4.09 GHz
Processes
231
Threads
2772
Handles
98490
Up time
0:00:56:52
Base speed: 3.60 GHz
Sockets: 1
Cores: 4
Logical processors: 8
Virtualization: Enabled
L1 cache: 256 KB
L2 cache: 1.0 MB
L3 cache: 8.0 MB

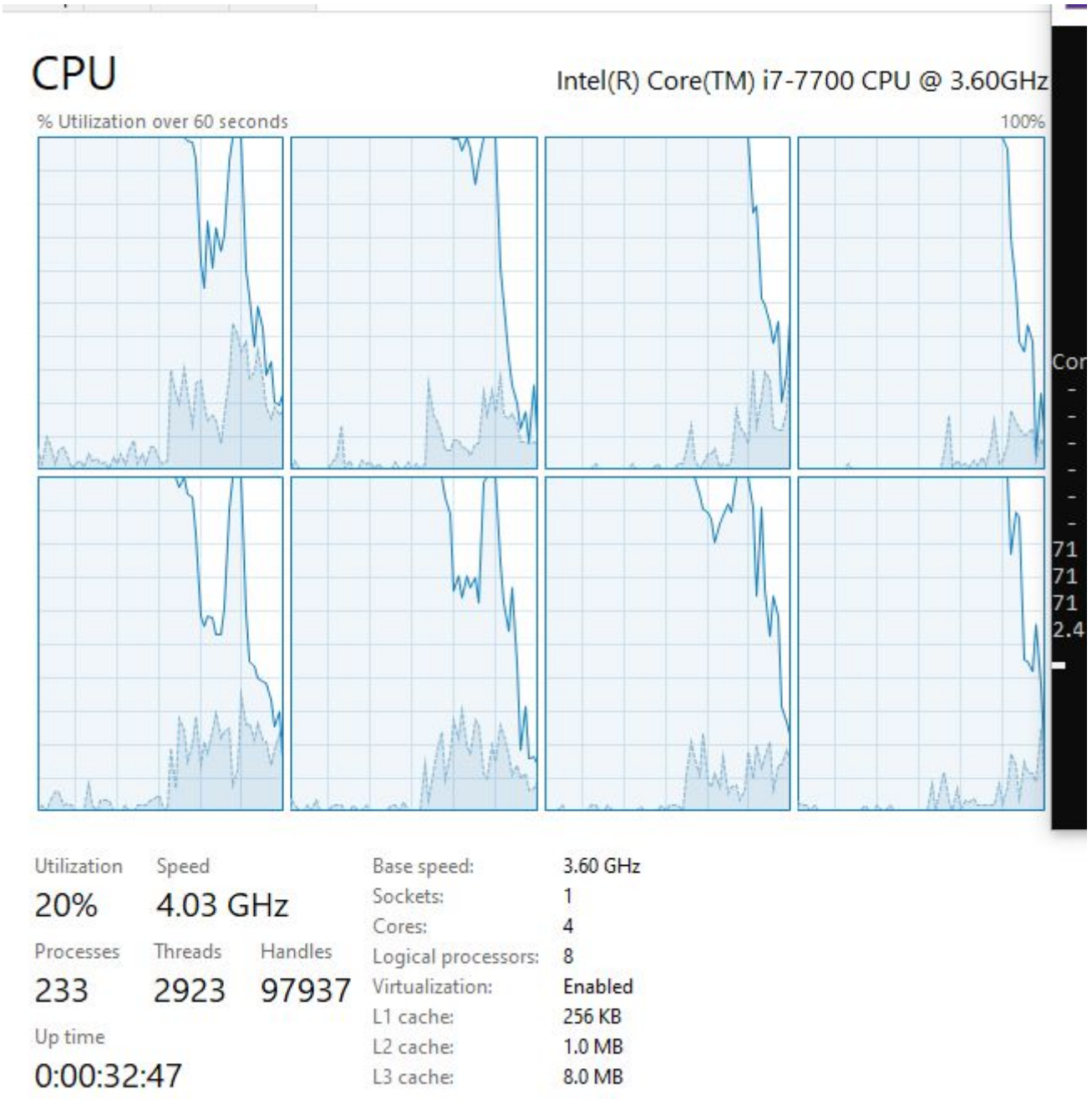
Scena 1

GPU



Se observă o încărcare de 5% şi FPS-uri maxime (monitor de 71 hz).

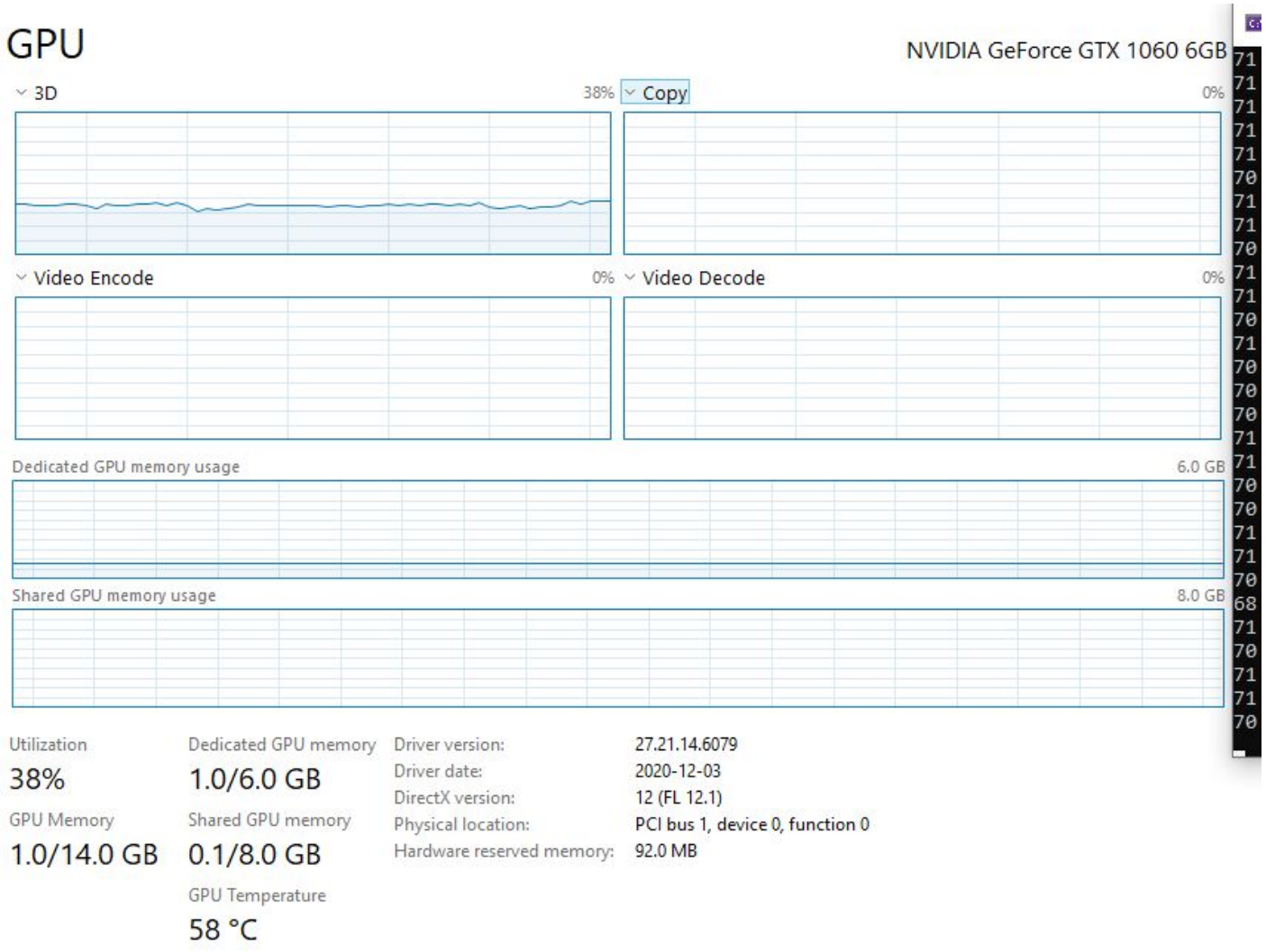
CPU



Se observă o rată de generare de 2.4 FPH. Sample rate-ul a fost de 600 și o adâncime maximă de 100.

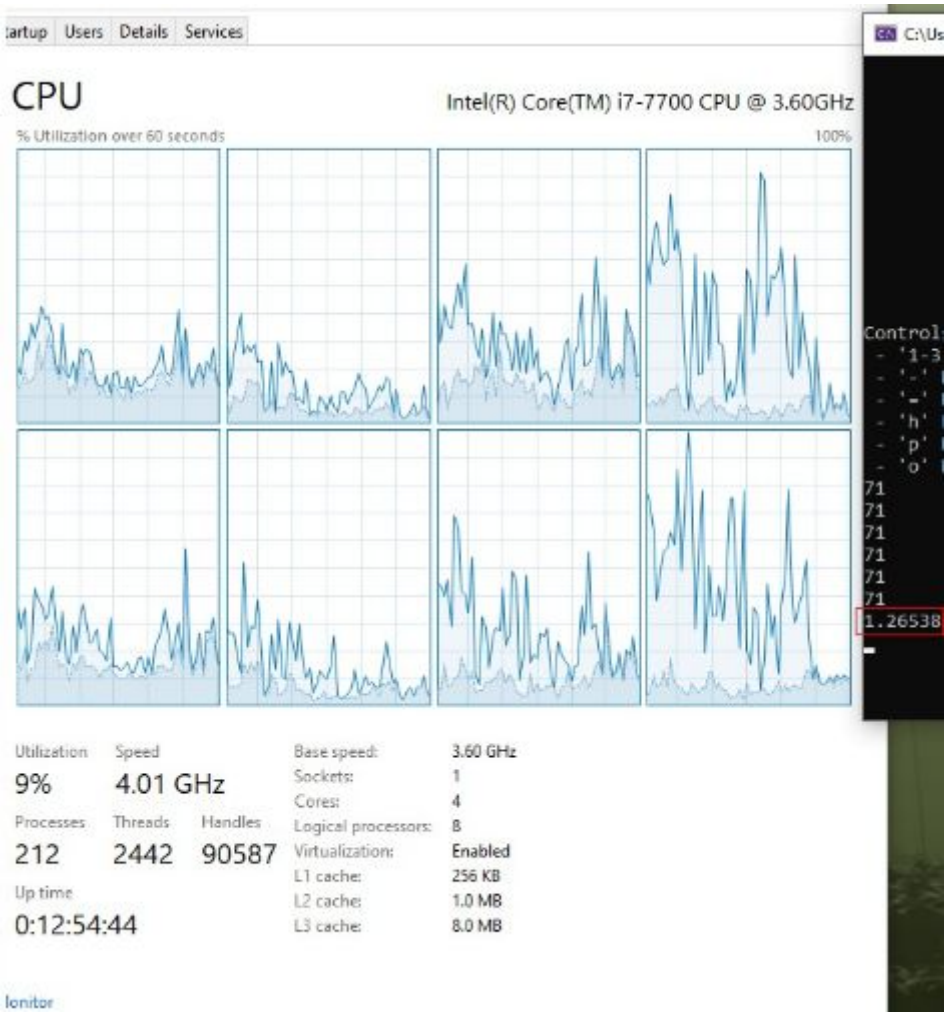
Scena 2

GPU



Se observă o încărcare de ~40% şi FPS-uri în continuare maxime.

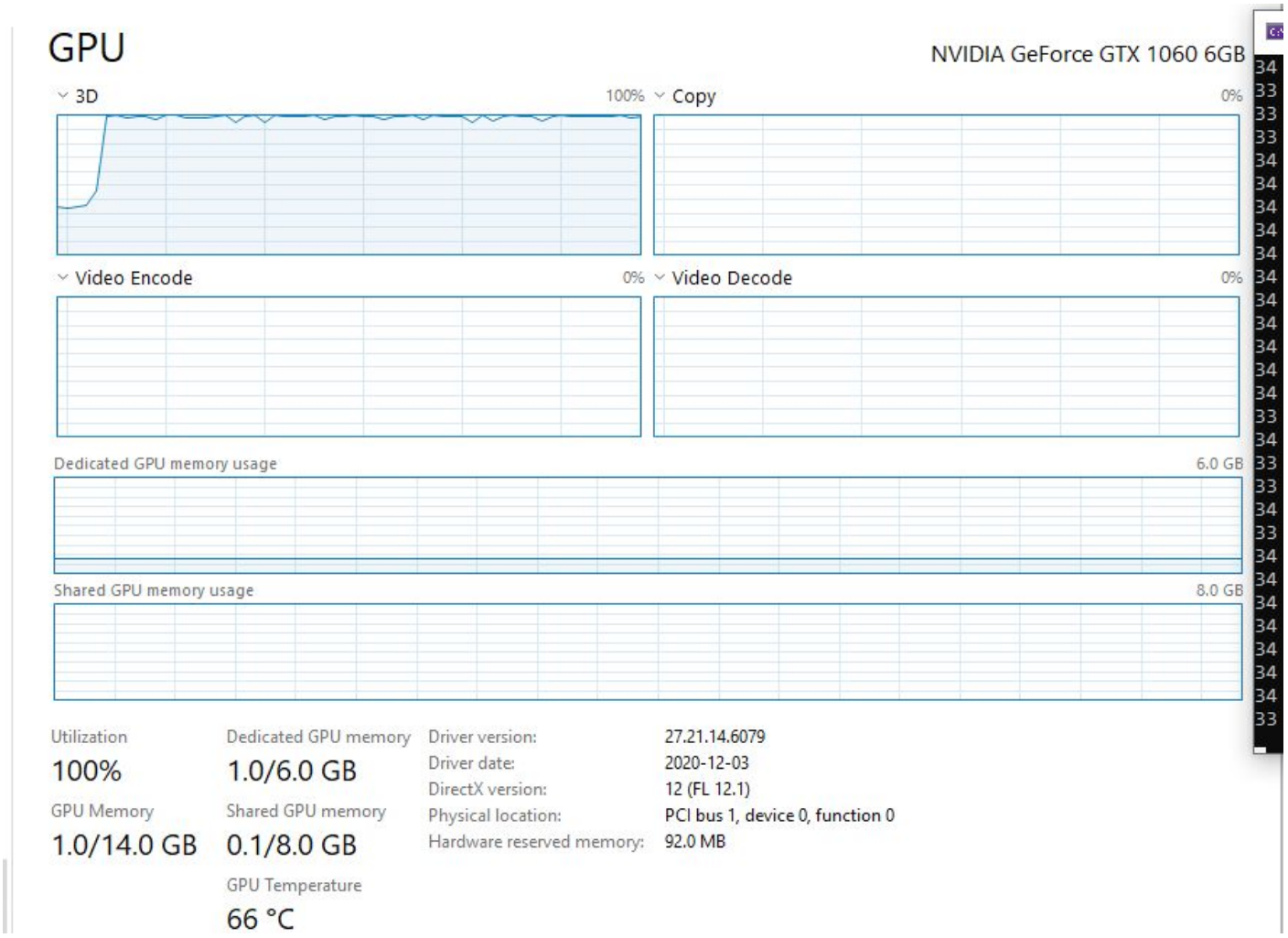
CPU



Încărcarea procesoarelor nu este mare deoarece captura a fost realizată ulterior. Totuşi, se observă numărul mic de FPH. Sample rate-ul folosit a fost de 400 şi distanţa maximă de 100.

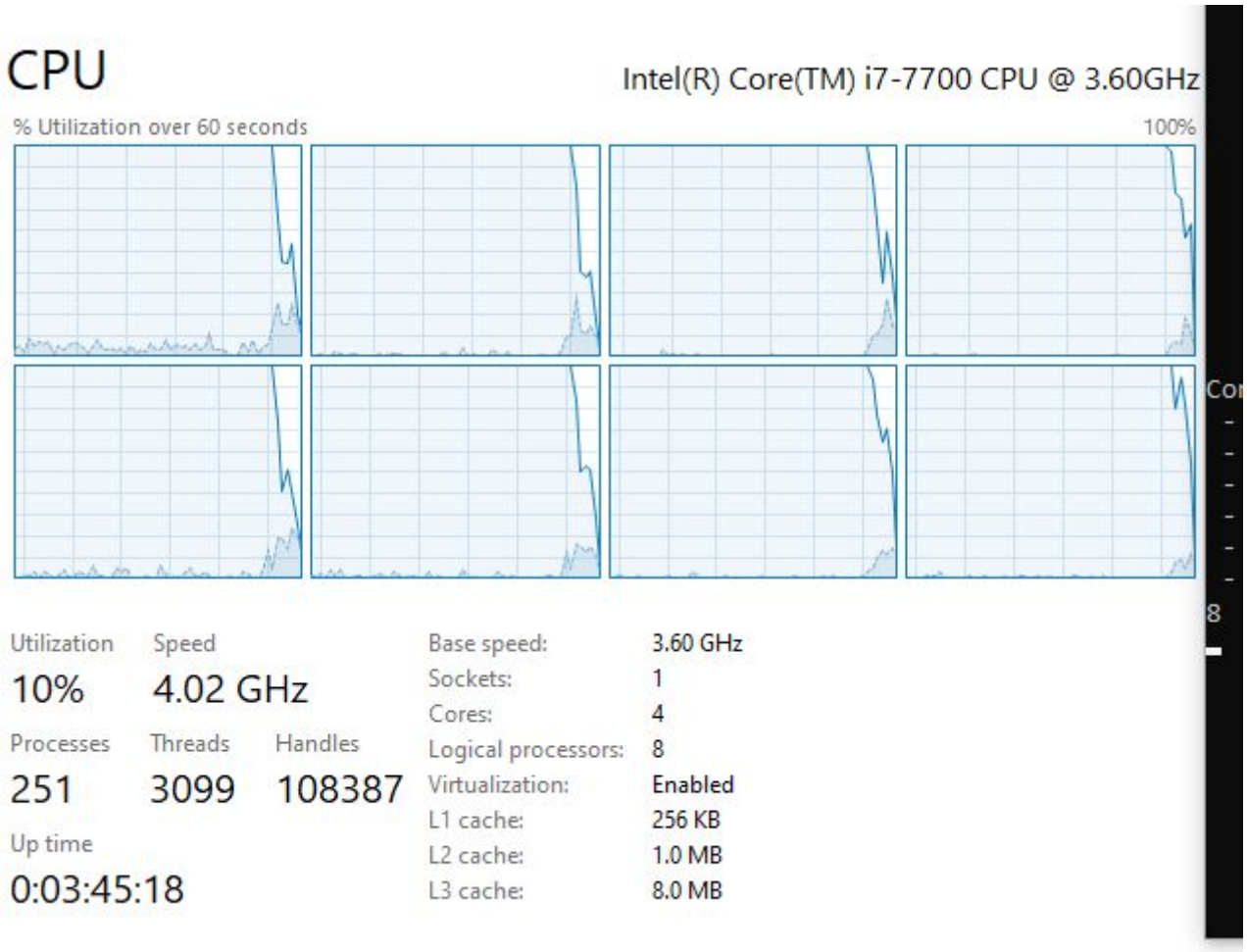
Scena 3

GPU



Se observă o încărcare maximă și FPS-uri precare ~34 (s-a rulat cu 40 de reflexii maxim a razelor).

CPU



S-a rulat cu 15 sample per pixel si 10 pentru max_depth. Aceste valori s-au ales deoarece, pentru configurația clasică de 400 cu 100 s-ar fi obţinut 8 / 1333.33 FPH.

Bibliografie

1. Ray Tracing in One Weekend , Peter Shirley, *Version 3.2.3, 2020-12-07,* https://github.com/RayTracing/raytracing.github.io