

# Proiect MGTR Vlkrt

Alex-Andrei Cioc

February 11, 2026

## 1 Introducere

Proiectul **Vlkrt** reprezintă o implementare de bază a unui motor grafic ce folosește Vulkan SDK pentru interfațarea cu GPU. Acesta construiește un pipeline de Ray tracing, cu funcționalități de bază precum încărcarea și editarea scenelor 3D și o arhitectură client-server ce permite mai multor clienți să interacționeze în aceeași scenă.

## 2 Arhitectura

Proiectul adoptă o arhitectură modulară, decuplată, construită pe un model client-server. Această structură permite separarea logicii de procesare a datelor și a stării globale (server headless) de logica de prezentare grafică și interacțiune cu utilizatorul (client).

### 2.1 Walnut Framework

La baza aplicației se află framework-ul **Walnut**, care oferă un strat subțire de abstractizare peste Vulkan și rulează aplicația **Vlkrt**. Acesta se ocupă de:

- **Gestionarea ferestrelor și a contextului:** Walnut utilizează **GLFW** sub capotă pentru crearea ferestrelor și gestionarea contextului de execuție.
- **Gestiunea resurselor Vulkan:** Framework-ul abstractizează concepte precum *Command Pool* și *Swapchain*. Clasa **Renderer** obține un **VkCommandBuffer** valid pentru fiecare frame prin **Walnut::Application::GetCommandBuffer()**. De asemenea, pentru a gestiona ștergerea resurselor utilizate de GPU în mod asincron (și a evita situații precum buffere distruse în timp ce sunt încă în uz într-un frame anterior), se folosește coada de eliberare a resurselor prin **SubmitResourceFree()**.
- **Layer System:** Arhitectura este bazată pe un sistem de *Layer*, similar cu cel din motoarele grafice moderne. Orice funcționalitate nouă este implementată ca un **Layer** (ex. **ClientLayer**, **ServerLayer**), care primește evenimente de **OnUpdate**, **OnRender** și **OnUIRender**.
- **Networking:** Walnut oferă suport pentru comunicații de rețea reliable prin UDP, folosind biblioteca **GameNetworkingSockets** de la Valve.

- **Imagini:** Clasa `Walnut::Image` încapsulează crearea `VkImage`, `VkImageView` și a descriptorilor necesari pentru a afișa rezultatul randării în **ImGui**.
- **Integrare ImGui:** Framework-ul oferă o integrare *out-of-the-box* cu **ImGui** pentru crearea rapidă de interfețe de utilizator reactive, optimizând procesul de dezvoltare. Pipeline-ul de randare 3D este de asemenea derivat din implementarea de Vulkan a **ImGui**.

## 2.2 Vlkrt-Client

Clientul reprezintă componenta principală de interfațare cu utilizatorul și motorul de vizualizare. Acesta conține:

- **Motor de randare:** Utilizează Vulkan SDK pentru un control granular asupra pipeline-ului de randare. Acesta este configurat pentru a utiliza extensiile de Ray tracing hardware (`VK_KHR_ray_tracing_pipeline`).
- **Procesare de input:** Gestionează camera și interacțiunile de mouse/tastatură prin intermediul framework-ului **Walnut**. În spate, **Walnut** folosește de fapt API-ul **GLFW**.
- **Motor de scripting:** Include un motor de scripting bazat pe **Lua** (via **Sol2** și **LuaJIT**), permițând definirea de logică custom pentru obiectele din scenă, fără re-compilarea codului C++.

## 2.3 Vlkrt-Server

Serverul este o aplicație *headless* responsabilă pentru sincronizarea stării jocului. Acesta nu inițializează contextul Vulkan pentru randare, utilizând în schimb o variantă minimală a **Walnut** (**Walnut-Headless**). Atribuții:

- **State Management:** Menține o bază de date in-memory cu toți utilizatorii conectați.
- **Packet Handling:** Procesează cererile primite de la clienți și replică modificările către toți ceilalți participanți în timp real.

## 2.4 Vlkrt-Common

Această bibliotecă statică conține simboluri partajate între binarele de client și server pentru a asigura consistența datelor. Aici se definesc structurile pachetelor de rețea (`ServerPacket.h` și `UserInfo.h`).

## 2.5 Sistemul de building

Pentru gestionarea complexității proiectului și a multiplelor dependențe, se utilizează un sistem de meta-build bazat pe **Premake5**. Caracteristici:

- **Modularizare:** Fiecare sub-componentă are propriul fișier de configurare scris în Lua (`Build-Vlkrt-Client.lua`, etc.). De asemenea, configurări de bază ce ar putea fi shared între diferite aplicații sunt definite în fișierele Lua `Build-Vlkrt-Common.lua`, respectiv `Build-Vlkrt-Common-Headless.lua`.

- **Portabilitate:** Soluția generează automat soluții de Visual Studio sau Makefile-uri de Linux, facilitând dezvoltarea cross-platform. Pentru testare, am rulat serverul într-un mediu de Linux, iar client-ul în Windows.

### 3 Networking

Comunicarea între clienți și server este gestionată de modulul **Walnut-Networking**, care utilizează biblioteca **GameNetworkingSockets** de la Valve pentru o comunicare bazată pe UDP, optimizată pentru latență redusă. Caracteristici:

- **Modelul de date:** Schimbul de informații se bazează pe pachete binare (structura **PacketType**). Mesajele sunt serializate într-un format compact pentru a minimiza utilizarea lății de bandă.
- **Sincronizarea stării:** Clientul trimite periodic actualizări ale poziției și stării proprii, în timp ce serverul transmite aceste date către ceilalți clienți conectați pentru a reconstrui scena globală.
- **Gestiunea conexiunilor:** Serverul implementează logica de acceptare/respingere a conexiunilor și de gestionare a deconectărilor.
- **Chat:** Clienții conectați pot comunica printr-un chat comun.
- **Server tickrate:** Pentru eficiență, serverul limitează procesarea pachetelor la o frecvență de 50Hz. Aceasta se poate crește cu ușurință.

```

1 // Exemplu: Trimitere update de la client la server
2 void ClientLayer::OnUpdate(float ts)
3 {
4     // ...
5     if (m_Client.GetConnectionStatus() ==
6         Walnut::Client::ConnectionStatus::Connected)
7     {
8         Walnut::BufferStreamWriter stream(s_ScratchBuffer);
9
10        // Scrie tip pachet si date player
11        stream.WriteRaw(PacketType::ClientUpdate);
12        stream.WriteRaw<glm::vec3>(m_PlayerPosition);
13        stream.WriteRaw<glm::vec3>(m_PlayerVelocity);
14
15        m_Client.SendBuffer(stream.GetBuffer());
16    }
17 }

```

La pornirea clientului:

- Utilizatorul este solicitat să introducă adresa serverului și portul pentru conexiune.
- După stabilirea conexiunii, se randează scena *default*.
- Toți jucătorii conectați sunt vizualizați drept cuburi mobile în scenă.

## 4 Input handling

Gestionarea evenimentelor de intrare este realizată prin integrarea cu sistemul de polling al **GLFW**, expus prin **Walnut**.

- **Polling: GLFW** procesează evenimentele de input prin polling. La fiecare frame, clientul verifică ultima stare a tastelor și a mouse-ului prin intermediul funcțiilor `glfwGetKey()` și `glfwGetMouseButton()`.
- **Interacțiune UI:** Sistemul de input este partajat cu **ImGui**; evenimentele sunt consumate prioritar de interfața grafică dacă este focusată o fereastră de UI.

## 5 Gestiunea resurselor

Un aspect critic al proiectului este încărcarea și organizarea eficientă a resurselor digitale (mesh-uri, scene, texturi).

### 5.1 Mesh-uri

Clasa `MeshLoader` utilizează biblioteca **tinyobjloader** pentru a încărca fișiere de tip `.obj`. După încărcare, se calculează automat normalele (dacă lipsesc) și coordonatele de textură.

Se folosește un format custom (structura `Mesh`) pentru a reține informații precum buffer-ele de vertexi și indecși, sau matricea de model. Se pot crea și obiecte în mod procedural, spre exemplu se pot folosi funcțiile `GenerateQuad` și `GenerateCube`.

```
1 struct Vertex
2 {
3     glm::vec3 Position{};
4     glm::vec3 Normal{};
5     glm::vec2 TexCoord{};
6 };
7
8 struct Mesh
9 {
10     std::string Filename;
11     std::string Name;
12     std::vector<Vertex> Vertices;
13     std::vector<uint32_t> Indices;
14
15     int MaterialIndex{ 0 };
16
17     glm::mat4 Transform = glm::mat4(1.0f);
18 };
```

```
1 // Exemplu: Create mesh procedural
2 Mesh cube = MeshLoader::GenerateCube();
3 cube.Transform = glm::translate(glm::mat4(1.0f),
4     glm::vec3(-3.0f, 0.0f, 0.0f));
5 cube.MaterialIndex = 0; // Referinta la un material din scena
6 m_Scene.StaticMeshes.push_back(cube);
```

## 5.2 Texturi

**Vlkr**t suportă încărcarea de texturi prin intermediul bibliotecii **stb\_image**. Resursele de tip imagini sunt gestionate prin clasa **Walnut::Image**.

- **Diferite formate:** Sunt suportate JPEG și PNG (teoretic și HDR, dar nu este suport în shadere).
- **Mipmap:** Texturile sunt mipmapped de către Vulkan (vezi Secțiunea 6).
- **Customizare:** Utilizatorul poate schimba texturile materialelor și aplica factori diferiți de tiling prin intermediul GUI-ului.

```
1 struct Material
2 {
3     std::string Name;
4     glm::vec3 Albedo{ 1.0f };
5     glm::vec3 Specular{ 1.0f };
6     float Shininess{ 32.0f };
7
8     std::string TextureFilename; // Daca este gol, se foloseste
9     doar Albedo
10    float Tiling{ 1.0f };
11 };
12 // Exemplu incarcare textura
13 auto newImg = std::make_shared<Walnut::Image>("resources/textures
14 /tiles.jpg");
15 m_TextureCache["tiles.jpg"] = texture;
```

## 5.3 Shadere

Shader-ele sunt scrise în GLSL și compilate în format intermediar SPIRV folosind **glslc**. Pasul de compilare trebuie gestionat separat de compilarea aplicației. Aceasta încarcă direct codul SPIRV la runtime. Proiectul include script-uri de compilare pentru Windows (**compile\_shaders.bat**) și Linux (**compile\_shaders.sh**).

Pentru mai multe detalii vezi Secțiunea 6.1.

## 5.4 Scripturi

Scripturile Lua sunt încărcate on-demand la primul update al fiecărui obiect scriptat. Mai multe detalii în Secțiunea 7.

## 5.5 Scene

### 5.5.1 Sistemul ierarhic

**Vlkr**t utilizează un sistem ierarhic de entități (**SceneEntity**), în care fiecare obiect din lume are o transformare relativă față de părintele său. Caracteristici:

- **Matrice separate:** Fiecare entitate deține o structură **Transform** care stochează separat poziția (**vec3**), rotația (**quat** - cuaternioni pentru evitarea *Gimbal Lock*) și scalarea (**vec3**).
- **Tipuri de entități:** Sistemul suportă multiple tipuri de entități: **Mesh** (geometrie), **Light** (surse de lumină), și **Empty** (folosite pentru gruparea logică a obiectelor).
- **Ierarhia transformărilor:** Matricea de transformare globală (**WorldTransform**) este recalculată recursiv:  $M_{world} = M_{parent\_world} \times M_{local}$ .

### 5.5.2 Serializare

Scenele sunt salvate și încărcate utilizând formatul YAML. **yaml-cpp** este biblioteca utilizată pentru parsarea și generarea fișierelor YAML.

- **Structură ierarhică:** YAML-ul reflectă ierarhia entităților din scenă, fiecare entitate având un câmp **children** care conține sub-entitățile sale, facilitând organizarea logică a scenei.
- **Persistență:** Toate proprietățile entităților (poziție, rotație, scalare, materiale etc.) sunt serializate, permițând salvarea progresului într-un fișier ușor de citit de către om (ex. **resources/scenes/default.yaml**). Modificările se pot salva și în timp real, din cadrul GUI-ului de editare a scenei.
- **Dynamic Loading:** Sistemul permite reîncărcarea la runtime a scenelor fără repornirea aplicației, facilitând iterarea rapidă asupra design-ului.

```

1 materials:
2   - name: stone
3     albedo: [ 0.5, 0.5, 0.5 ]
4     shininess: 25.6
5     specular: [ 0.1, 0.1, 0.1 ]
6     texture: tiles.jpg
7     tiling: 5
8   - name: metal
9     albedo: [ 0.8, 0.8, 0.8 ]
10    shininess: 102.4
11    specular: [ 1.0, 1.0, 1.0 ]
12    texture: wood.jpg
13    tiling: 0.5
14   - name: bricks
15     albedo: [ 0.9, 0.9, 0.9 ]
16     shininess: 8.0
17     specular: [ 0.1, 0.1, 0.1 ]
18     texture: brick.jpg
19     tiling: 2.0
20
21 entities:
22   - name: scene_root
23     type: empty
24     transform:
25       position: [ 0, -0.408099, 0 ]

```

```

26     rotation: [ 0, 0, 0, 0 ]
27     scale: [ 1, 1, 1 ]
28 children:
29 - name: grounded
30   type: mesh
31   transform:
32     position: [ 0, -1, 0 ]
33     rotation: [ 0, 0, 0, 1 ]
34     scale: [ 10, 1, 10 ]
35   mesh: ground.obj
36   material: 0
37 - name: cube
38   script: cube_animation.lua
39   type: mesh
40   transform:
41     position: [ 2, 2.3196, 0 ]
42     rotation: [ 0, 0.580628, 0, 0.814169 ]
43     scale: [ 1, 1, 1 ]
44   mesh: cube.obj
45   material: 1
46 - name: lighting_group
47   type: empty
48   transform:
49     position: [ -0.909, 5, 5 ]
50     rotation: [ -0.643505, 0.0327726, 0, 0.76474 ]
51     scale: [ 3.61, 1, 1 ]
52 children:
53 - name: sun
54   type: light
55   transform:
56     position: [ 0, 0, 0 ]
57     rotation: [ 0, 0, 0, 1 ]
58     scale: [ 1, 1, 1 ]
59   light_color: [ 1, 1, 1 ]
60   light_intensity: 1.5
61   light_type: 0
62   light_direction: [ 0, 0, -1 ]
63 - name: point_light
64   type: light
65   transform:
66     position: [ 0, 0, 0 ]
67     rotation: [ 0, 0, 0, 1 ]
68     scale: [ 1, 1, 1 ]
69   light_color: [ 0.729107, 0.518529, 0.275254 ]
70   light_intensity: 1.798
71   light_type: 1
72   light_radius: 19.074

```

## 5.6 Editor de Scene

Clientul include un editor vizual cu următoarele funcționalități:

- **Arborele de entități:** Afășează o reprezentare ierarhică a tuturor obiectelor din scenă. Utilizatorul poate:
  - Selecta entități în arbore.
  - Extinde/restrânge noduri pentru a vizualiza relațiile părinte-copil.
  - Modifica, salva și încărca scene.
- **Inspector de proprietăți:** Afășează detaliile entității selectate:
  - **Transform Controls:** Câmpuri pentru poziție (X, Y, Z), pentru rotație Euler (pitch, yaw, roll) și pentru scalare. Schimbările sunt aplicate imediat.
  - **Material Properties:** Selectare de material dintr-o listă, ajustare a albedoului, specular-ului, și shininess-ului. Texturile pot fi selectate dintr-o listă a fișierelor disponibile.
  - **Light Properties:** Pentru entități light, oferă controale pentru culoare, direcție (la lumini direcționale), intensitate, și raza de influență (la lumini de tip punct).
  - **Script Path:** Permite atribuirea unui script Lua unei entități.
- **Chat:** Acceptă input de la utilizator și afășează toate mesajele stocate pe server
- **Statistics Panel:** Afășează:
  - Durata buclei de randare în milisecunde.
  - ID-ul jucătorului.
  - Numărul de jucători conectați.

```

1 // Exemplu: Editare proprietati material din GUI
2 void ClientLayer::ImGuiRenderMaterialEditor(int matIdx)
3 {
4     Material& mat = m_Scene.Materials[matIdx];
5
6     // Albedo color picker
7     if (ImGui::ColorEdit3("Albedo##mat",
8         glm::value_ptr(mat.Albedo)))
9         m_Renderer.InvalidateScene();
10
11     // Shininess slider (1-512)
12     if (ImGui::DragFloat("Shininess##mat",
13         &mat.Shininess, 1.0f, 1.0f, 512.0f))
14         m_Renderer.InvalidateScene();
15
16     // Specular intensity
17     if (ImGui::ColorEdit3("Specular##mat",
18         glm::value_ptr(mat.Specular)))
19         m_Renderer.InvalidateScene();
20
21     // Texture tiling
22     if (ImGui::DragFloat("Tiling##mat",
23         &mat.Tiling, 0.1f, 0.1f, 100.0f))

```

```

24     m_Renderer.InvalidateScene();
25 }

```

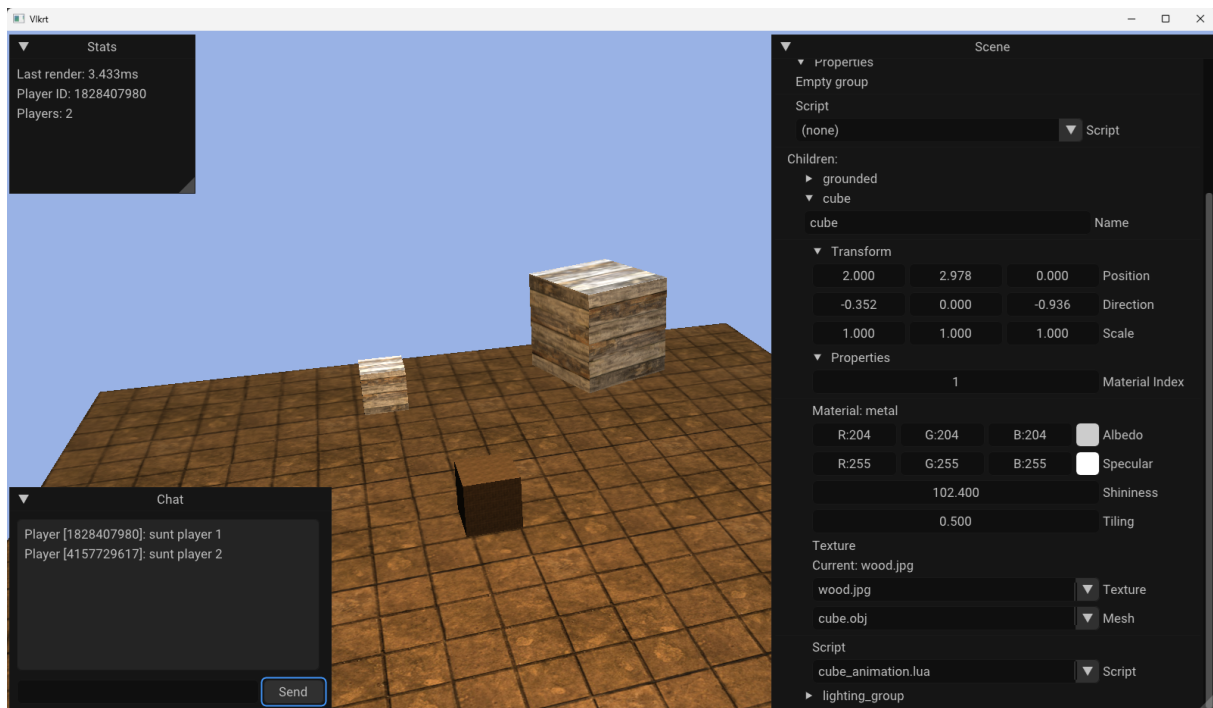


Figure 1: Editor de scene **Vlkrt** - vizualizare 3D cu panourile de proprietăți și arborele de entități

## 6 Rendering

Nucleul grafic al proiectului este un motor de **Ray tracing** bazat pe API-ul Vulkan, implementat în clasa **Renderer**. Acesta gestionează complet proprietățile pipeline-ului de Ray Tracing, inclusiv descriptor set-urile, shader binding table-ul (SBT) și bufferele de scenă.

### 6.1 Pipeline

Spre deosebire de rasterizarea tradițională, **Vlkrt** folosește extensia de pipeline Vulkan `VK_KHR_ray_tracing_pipeline` pentru a implementa un pipeline de Ray tracing.

- **Acceleration Structures (AS):** Geometria este organizată în două niveluri:
  1. **Bottom-Level AS (BLAS):** Conține geometria brută a fiecărui mesh (buffere de vertex și index).
  2. **Top-Level AS (TLAS):** Conține instanțe ale BLAS-urilor, fiecare cu propria matrice de transformare, permițând instanțierea eficientă a obiectelor.
- **Shader Binding Table (SBT):** O structură de date critică în Vulkan RT care face legătura între razele lansate și grupurile de shadere ce trebuie executate la impact. De altfel, se definesc 3 diferite tipuri de shadere: `raygen`, `closesthit`, `miss`.

Un caveat al implementării este folosirea a 1 BLAS și 1 TLAS, ceea ce este în regulă pentru scene mici și statice, însă ar putea fi partiționat mai bine pe mesh-uri statice și dinamice. De altfel, implementarea de față reconstruiește aceste structuri cam la fiecare frame.

## 6.2 Arhitectura Renderer-ului

Clasa `Renderer` implementează logica de randare pas cu pas în funcția `Render()`:

1. **Validarea Scenei:** La fiecare apel, se verifică dacă structura scenei s-a modificat (număr de mesh-uri, vertecși, materiale, lumini). Dacă scena este invalidată (ex. s-a schimbat scena), **Renderer**-ul distruge bufferele vechi și realocă memorie GPU pentru noile date. Nu este cel mai eficient proces, dar pentru o scenă mică sau cu modificări rare, este suficient.
2. **Managementul Bufferelor:** Datele scenei sunt încărcate în buffere dedicate Vulkan (`VkBuffer`):
  - `m_VertexBuffer` / `m_IndexBuffer`: Geometria scenei.
  - `m_MaterialBuffer`: Proprietățile materialelor (albedo, specular, shininess).
  - `m_MaterialIndexBuffer`: Mapează fiecare triunghi la un index de material.
  - `m_LightBuffer`: Structuri `GPULight` pentru iluminare.
3. **Descriptor Sets:** Imaginea finală (`m_FinalImage`) este legată ca un `STORAGE_IMAGE` în descriptor set, permițând shader-ului de Ray Generation să scrie direct pixelii rezultați. La redimensionarea ferestrei (`OnResize()`), imaginea este recreată, iar descriptor set-ul este actualizat cu noul `VkImageView`.
4. **Corectitudine și Sincronizare:** Înainte de dispatch-ul razelor, imaginea de output trece printr-o barieră de memorie (`vkCmdPipelineBarrier`) pentru a tranziționa layout-ul de la `SHADER_READ_ONLY_OPTIMAL` la `GENERAL`.

```
1 // Exemplu: Create buffer GPU si incarcare date
2 void Renderer::CreateSceneBuffers()
3 {
4     // Calculeaza numarul total de vertecsi si indecsi (static +
5     // dinamic)
6     size_t totalVertices = 0;
7     size_t totalIndices = 0;
8     for (const auto& mesh : scene.StaticMeshes) {
9         totalVertices += mesh.Vertices.size();
10        totalIndices += mesh.Indices.size();
11    }
12    for (const auto& mesh : scene.DynamicMeshes) {
13        totalVertices += mesh.Vertices.size();
14        totalIndices += mesh.Indices.size();
15    }
16
17    // Creeaza vertex buffer
18    size_t vertexCount = std::max(totalVertices, (size_t) 1);
```

```

18     m_VertexBufferSize = sizeof(GPUVertex) * vertexCount;
19     m_VertexBuffer      = CreateBuffer(m_VertexBufferSize,
VK_BUFFER_USAGE_STORAGE_BUFFER_BIT |
VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT |
VK_BUFFER_USAGE_ACCELERATION_STRUCTURE_BUILD_INPUT_READ_ONLY_BIT_KHR
, VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
VK_MEMORY_PROPERTY_HOST_COHERENT_BIT, m_VertexMemory);

20
21     // Create index buffer
22     size_t indexCount = std::max(totalIndices, (size_t) 1);
23     m_IndexBufferSize = sizeof(uint32_t) * indexCount;
24     m_IndexBuffer      = CreateBuffer(m_IndexBufferSize,
VK_BUFFER_USAGE_STORAGE_BUFFER_BIT |
VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT |
VK_BUFFER_USAGE_ACCELERATION_STRUCTURE_BUILD_INPUT_READ_ONLY_BIT_KHR
, VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
VK_MEMORY_PROPERTY_HOST_COHERENT_BIT, m_IndexMemory);

25
26     // Creeaza material buffer
27     size_t materialCount = std::max(scene.Materials.size(), (
size_t) 1);
28     m_MaterialBufferSize = sizeof(GPUMaterial) * materialCount;
29     m_MaterialBuffer      = CreateBuffer(m_MaterialBufferSize,
VK_BUFFER_USAGE_STORAGE_BUFFER_BIT,
VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
VK_MEMORY_PROPERTY_HOST_COHERENT_BIT, m_MaterialMemory);

30
31     // Creeaza material index buffer (1 uint32 per triunghi)
32     size_t triangleCount = std::max(totalIndices / 3, (
size_t) 1);
33     m_MaterialIndexBufferSize = sizeof(uint32_t) * triangleCount;
34     m_MaterialIndexBuffer      = CreateBuffer(
m_MaterialIndexBufferSize, VK_BUFFER_USAGE_STORAGE_BUFFER_BIT,
VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
VK_MEMORY_PROPERTY_HOST_COHERENT_BIT, m_MaterialIndexMemory);

35
36     // Creeaza light buffer
37     size_t lightCount = std::max(scene.Lights.size(), (size_t) 1)
;
38     m_LightBufferSize = sizeof(GPULight) * lightCount;
39     m_LightBuffer      = CreateBuffer(m_LightBufferSize,
VK_BUFFER_USAGE_STORAGE_BUFFER_BIT,
VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
VK_MEMORY_PROPERTY_HOST_COHERENT_BIT, m_LightMemory);
40 }

```

Procesul de randare (în shadeare) urmează fluxul standard de Ray tracing:

1. **Ray Generation (raygen.rgen):** Calculează direcția razei primare pentru fiecare pixel de pe ecran, plecând din poziția camerei.
2. **Acceleration Structure Traversal:** GPU-ul parcurge TLAS și BLAS pentru a

găsi cel mai apropiat punct de intersecție.

3. **Closest Hit** (`closesthit.rchit`): Se execută la intersecția cu geometria. Aici se calculează modelul de iluminare Phong și se eșantionează texturile. Pentru acestea din urmă, am folosit mipmap-uri cu calcul de LoD bazat pe Ray Cones.
4. **Miss** (`miss.rmiss`): Se execută dacă raza nu întâlnește niciun obiect, returnând culoarea *skybox*-ului.

## 6.3 Implementarea Shaderelor

Deoarece Ray Tracing-ul nu beneficiază de hardware-ul de rasterizare pentru interpolarea atributelor și calculul LOD-ului (Level of Detail) pentru texturi, aceste funcționalități au fost implementate manual în shaderele GLSL.

### 6.3.1 Ray Generation (`raygen.rgen`)

Acesta este punctul de intrare. Shader-ul:

- Calculează coordonatele normale de ecran (NDC) pentru pixelul curent.
- Reconstruiește direcția razei folosind inversul matricelor de Projection și View primite prin *Push Constants* (`CameraData`).
- Inițializează structura `RayPayload`. Aceasta conține:
  - `hitValue`: Culoarea acumulată (output).
  - `coneWidth`: Lățimea conului razei la punctul curent (inițial 0).
  - `spreadAngle`: Unghiul de dispersie a pixelului, aproximat pe baza FOV-ului vertical și a înălțimii viewport-ului.
- Apelează `traceRayEXT` pentru a lansa raza în scenă.

### 6.3.2 Closest Hit (`closesthit.rchit`)

Executat la intersecția cu geometria, acest shader realizează partea cea mai complexă a randării.

**Interpolarea Atributelor** Spre deosebire de pipeline-ul grafic clasic, aici avem acces doar la indecșii triunghiului lovit și coordonatele baricentrice. Shaderul accesează manual Buffer-ele (*Structured Buffers*) de vertexi și indecși pentru a încărca datele celor 3 vârfuri și a interpola poziția, normala și coordonatele UV.

**Texture Sampling (Ray Cones)** Pentru a evita efectul de *aliasing* pe texturi aflate la distanță sau la unghiuri mici, s-a implementat tehnica **Ray Cones**. Formula utilizată pentru calculul nivelului de detaliu (LOD) este adaptată din *Ray Tracing Gems, Chapter 20 (EA Seed)*. Aceasta ia în considerare:

1. Raportul dintre aria triunghiului în spațiul lumii și aria în spațiul UV.
2. Rezoluția texturii.

3. Lățimea conului razei la intersecție (`currentConeWidth = payload.coneWidth + payload.spreadAngle * gl_HitTEXT`).
4. Unghiul de incidență al razei cu suprafața.

Rezultatul este folosit în funcția `textureLod` pentru a eșantiona nivelul corect de mipmap.

**Modelul de Iluminare** Se utilizează un model clasic Blinn-Phong, fără componenta de emisie:

- **Ambient:** Termen constant ( $0.2 \times Albedo$ ).
- **Difuz:** Lambertian standard ( $N \cdot L$ ).
- **Specular:** Blinn-Phong cu exponent controlabil (*shininess*).

### 6.3.3 Miss Shader (`miss.rmiss`)

Un shader minimal care returnează o culoare constantă (`vec3(0.6, 0.7, 0.9)`) reprezentând *skybox*-ul, atunci când raza nu intersectează nicio geometrie.

```

1 #version 460
2 #extension GL_EXT_ray_tracing : require
3
4 struct RayPayload {
5     vec3 hitValue;
6     float coneWidth;
7     float spreadAngle;
8 };
9
10 layout(location = 0) rayPayloadInEXT RayPayload payload;
11
12 void main()
13 {
14     // Sky color
15     payload.hitValue = vec3(0.6, 0.7, 0.9);
16 }

```

### 6.3.4 Ray Cones

Este un algoritm pentru calculul LoD (Level of Detail) la eșantionarea texturilor în pipeline-ul Ray tracing. Acesta ia în calcul ariile triunghiului în spațiu lume și UV, pe unghiul de incidență al razei și pe rezoluția texturii de eșantionat.

```

1 // 1. Aria triunghiului (world + uv)
2 vec3 edge1 = v1.position - v0.position;
3 vec3 edge2 = v2.position - v0.position;
4 float triArea = length(cross(edge1, edge2)); // 2 * Area_world
5
6 vec2 texEdge1 = v1.texCoord - v0.texCoord;
7 vec2 texEdge2 = v2.texCoord - v0.texCoord;
8 float uvArea = abs(texEdge1.x * texEdge2.y - texEdge1.y *
    texEdge2.x); // 2 * Area_uv

```

```

9 uvArea *= mat.tiling * mat.tiling;
10
11 // 2. Constanta de LoD in functie de raportul celor 2 arii
12 float lodConstant = 0.5 * log2(uvArea / max(triArea, 1e-10));
13
14 // 3. Termen de rezolutie a texturii
15 ivec2 texSize = textureSize(textures[mat.textureIndex], 0);
16 float texScale = 0.5 * log2(float(texSize.x * texSize.y));
17
18 // 4. Termen al unghiului de incidenta
19 float cosTheta = abs(dot(normal, gl_WorldRayDirectionEXT));
20 float rayTerm = log2(max(currentConeWidth, 1e-10) / max(cosTheta,
    1e-10));
21
22 // Calcul final
23 // Bias cu -0.5 pentru sharpness
24 float lod = lodConstant + rayTerm + texScale - 0.5;
25 lod = max(0.0, lod);
26
27 albedo *= textureLod(textures[mat.textureIndex], uv, lod).rgb;

```

## 6.4 Camera

Sistemul de cameră este implementat în clasa **Camera**, oferind o proiecție perspectivă și control de tip *first-person*. Utilizatorul poate naviga folosind tastele WASD pentru translație și mouse-ul pentru rotație (pitch/yaw). Rotația se realizează numai la apăsarea butonului de click dreapta.

```

1 auto Camera::OnUpdate(float ts) -> bool
2 {
3     glm::vec2 mousePos = Input::GetMousePosition();
4     glm::vec2 delta = (mousePos - m_LastMousePosition) *
    0.002f;
5     m_LastMousePosition = mousePos;
6
7     // Miscam doar cand tinem apasat click dreapta
8     if (!Input::IsMouseButtonDown(MouseButton::Right)) {
9         Input::SetCursorMode(CursorMode::Normal);
10        return false;
11    }
12
13    Input::SetCursorMode(CursorMode::Locked);
14
15    bool moved = false;
16
17    constexpr glm::vec3 upDirection(0.0f, 1.0f, 0.0f);
18    glm::vec3 rightDirection = glm::cross(m_ForwardDirection,
    upDirection);
19
20    // Miscare
21    if (Input::IsKeyDown(KeyCode::W)) {

```

```

22     m_Position += m_ForwardDirection * m_MovementSpeed * ts;
23     moved = true;
24 }
25 else if (Input::IsKeyDown(KeyCode::S)) {
26     m_Position -= m_ForwardDirection * m_MovementSpeed * ts;
27     moved = true;
28 }
29 if (Input::IsKeyDown(KeyCode::A)) {
30     m_Position -= rightDirection * m_MovementSpeed * ts;
31     moved = true;
32 }
33 else if (Input::IsKeyDown(KeyCode::D)) {
34     m_Position += rightDirection * m_MovementSpeed * ts;
35     moved = true;
36 }
37 if (Input::IsKeyDown(KeyCode::Q)) {
38     m_Position -= upDirection * m_MovementSpeed * ts;
39     moved = true;
40 }
41 else if (Input::IsKeyDown(KeyCode::E)) {
42     m_Position += upDirection * m_MovementSpeed * ts;
43     moved = true;
44 }
45
46 // Rotatie
47 if (delta.x != 0.0f || delta.y != 0.0f) {
48     float pitchDelta = delta.y * m_RotationSpeed;
49     float yawDelta   = delta.x * m_RotationSpeed;
50
51     glm::quat q = glm::normalize(glm::cross(glm::angleAxis(-
pitchDelta, rightDirection), glm::angleAxis(-yawDelta, glm::
vec3(0.f, 1.0f, 0.0f))));
52
53     m_ForwardDirection = glm::rotate(q, m_ForwardDirection);
54     moved               = true;
55 }
56
57 if (moved) { RecalculateView(); }
58
59 return moved;
60 }

```

## 7 Scripting

Pentru a oferi flexibilitate în simulare, a fost integrat motorul de scripting **Lua**.

- **Sol2 + LuaJIT:** S-a utilizat biblioteca sol2 ca wrapper C++ peste LuaJIT, oferind o performanță ridicată și o sintaxă clean pentru bind-uri.
- **Expunerea API-ului:** Entitățile din scenă sunt expuse către scripturi, permițând

modificarea transformărilor sau a proprietăților materialelor la runtime (ex. animația `cube_animation.lua`). Proprietățile expuse includ:

- **Transform:** Conține poziția (`vec3`), rotația (`quat`), și scalarea (`vec3`) a entității. Modificările sunt automat aplicate la entitate.
- **Operatori glm:** Bindings pentru calcule cu vectori (operații de adunare, scădere, normalizare, produs scalar) și cuaternioni (rotații, interpolare).
- **Input:** Funcții pentru polling-ul input-urilor utilizatorului.
- **Funcții de bibliotecă standard:** Accesul la bibliotecile Lua standard (`math`, `os`, `table`) pentru logică complexă.
- **Logging:** Binding către metoda de logging din **Walnut**, bazată pe **spdlog**.

- **Ciclul de viață al scriptului:**

1. La prima actualizare a unei entități cu script, scriptul este încărcat din fișierul `.lua` și o funcție `OnUpdate(entity, dt)` este căutată.
2. La fiecare frame, această funcție este apelată cu entitatea și intervalul de timp `ts` ca parametri.
3. Scriptul poate modifica transformările sau alte proprietăți ale entității în mod direct.

```
1 -- cube_animation.lua - Exemplu script de animatie
2 -- Roteste cubul pe axa Y si aplica oscilatie verticala
3
4 local startTime = os.clock()
5 local initialPos = nil
6
7 function OnUpdate(entity, dt)
8     local time = os.clock() - startTime
9
10    if initialPos == nil then
11        initialPos = vec3.new(entity.Transform.Position.x, entity
12        .Transform.Position.y, entity.Transform.Position.z)
13    end
14
15    -- Rotation: rotate around Y axis (0, 1, 0)
16    local rotationSpeed = 1.0
17    entity.Transform.Rotation = AngleAxis(time * rotationSpeed,
18    vec3.new(0, 1, 0))
19
20    -- Up and down movement: sine wave on Y axis
21    local bounceHeight = 1.0
22    local bounceSpeed = 2.0
23    entity.Transform.Position.y = initialPos.y + math.sin(time *
24    bounceSpeed) * bounceHeight
25 end
```

## 8 Deploy

Se pot urmări instrucțiunile de aici. Clientul se poate compila și rula pe local, în timp ce serverul se poate deploya ori pe local, ori direct în cloud. Instrucțiunile de mai sus arată cum se poate deploya pe Unikraft Cloud. Astfel, clienți de pe mașini diferite pot interacționa cu serverul accesibil public.

## 9 Next steps

- Sincronizarea scenelor pe server.
- Rularea scripturilor server-side.
- Interpolarea update-urilor de transformare de la server.
- Customizarea caracterului.
- Pipeline îmbunătățit de Ray tracing (umbre, reflexii/refracții, Path tracing).
- Sistem PBR pentru materiale + texturi PBR.