Cyberpunk Alley — Proiect OpenGL (C++ + GLSL)

Echipa 2

Horneț Alex-Andrei & Damian Alexandru

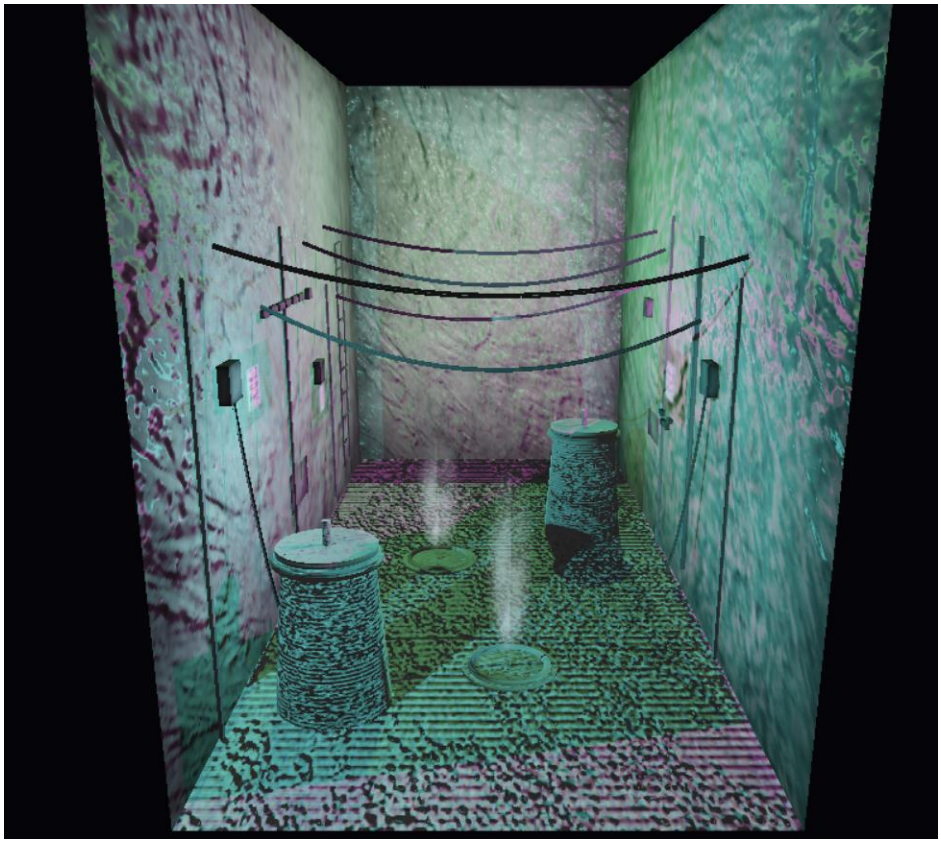
Grupa 342

CUPRINS

Conceptul proiectului

Cyberpunk Alley randează o scenă 3D de tip „alee urbană” cu asfalt și pereți texturati, panouri luminoase, obiecte 3D și efect de abur. Camera se rotește ușor în jurul scenei, iar iluminarea este realizată cu 3 surse de lumină colorate. Programul include **normal mapping** pentru detalii fine pe suprafețe, **ceață (fog)** pentru atmosferă, și **shadow mapping** pentru umbre reale (cu hărți de adâncime).

Scopul proiectului este să prezinte rapid o scenă 3D „cyberpunk”, prin combinarea mai multor idei din curs: **shadere (vertex + fragment)**, **texturi**, **normal mapping**, **ceață**, **tone mapping + gamma**, **camera cu cuaternioni** și **shadow mapping**. Scenei i se adaugă elemente decorative (țevi, scări, cabluri) și un efect procedural de abur, pentru atmosferă.



Elemente incluse

Randare și shading

- Randare OpenGL cu **VAO/VBO** și desenare cu glDrawArrays(GL_TRIANGLES, ...).
- **2 shadere principale**:
    - Shader de scenă: alley.vert + alley.frag
    - Shader pentru umbre: shadow_depth.vert + shadow_depth.frag (depth-only)
- Iluminare cu **3 lumini** (lightPos[3], lightColor[3]).

Texturi și materiale

- Texturi albedo: asphalt.jpg, wall.jpg, sign.png.
- Normal maps: asphalt_n.jpg, wall_n.jpg.
- Selecția texturii se face printr-un ID per-vertex (vTexId).

Efecte vizuale

- **Normal mapping** (toggle cu tasta n).
- **Fog / ceață** (toggle cu tasta f).
- **Shadow mapping** real (toggle cu tasta m) folosind 3 shadow maps.
- **Steam procedural** (abur) cu transparență și zgomot (noise), fără textură dedicată.
- **Tone mapping + gamma correction** pentru culori mai plăcute pe ecran.

Camera și control

- Camera orbit în jurul scenei cu **cuaternioni** (săgeți pentru yaw/pitch; +/- zoom).
- Mutare lumină principală cu i/j/k/l.

Originalitate

Proiectul nu este doar „o scenă cu texturi”, ci combină mai multe componente într-un rezultat coerent:

- **Abur procedural** (steam) generat din noise + măști + rim lighting, fără mesh complex și fără textură.
- **Shadow mapping cu 3 lumini**, cu filtrare PCF în shader.
- **Scenă generată procedural** (podea, pereți, detalii: țevi, cabluri, scări, cutii), plus obiecte încărcate din OBJ.
- **Camera orbit stabilă** implementată cu **cuaternioni**, evitând problemele clasice ale rotațiilor Euler.

Structura proiectului (fișiere importante)

- **main.cpp**
  - construiește scena (vertecși), încarcă texturi, setează shadere, face render loop
  - inițializează shadow maps (FBO + depth textures)
  - controlează camera și toggle-urile (fog, normal map, shadow map)
- **alley.vert**
  - primește atributele pe vertex (poziție, normal, UV, tangente)
  - calculează poziția în lume + matricea TBN
  - trimite datele către fragment shader
- **alley.frag**
  - alege textura pe baza vTexId
  - calculează iluminarea (difuz + specular), normal mapping, umbre, fog
  - randare specială pentru vTexId == 3 (steam)
- **shadow_depth.vert/.frag**
  - randare doar în adâncime pentru shadow maps
- **objloader.cpp/.hpp**
  - loader OBJ mai robust (triangulare, indici negativi etc.)

Construirea scenei (main.cpp)

Scena este construită în BuildAlley() prin adăugarea de triunghiuri într-un vector gVertices.

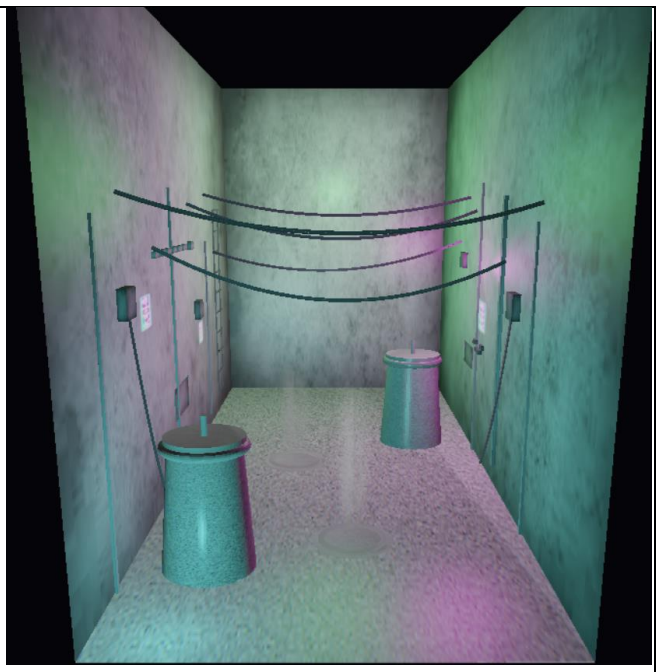
6.1. Tipul de vertex (structura Vtx)

Fiecare vertex conține:

- pos (vec4): poziția
- col (vec3): culoare/tint (folosită și ca intensitate pentru abur)
- nrm (vec3): normal
- uv (vec2): coordonate textură
- texId (float): ce material e (0 asfalt, 1 perete, 2 sign, 3 steam)
- tan, bit (vec3): tangentă și bitangentă pentru normal mapping

6.2. Suprafețe + detalii

- Podea: un quad (2 triunghiuri) cu TEX_ASPHALT.
- Pereți: quad-uri cu TEX_WALL.
- Sign-uri: quad-uri cu TEX_SIGN.
- Detalii (țevi, cutii, vent, scară, cabluri): generate procedural ca prism-uri/quad-uri.
- Obiecte OBJ: trashcan.obj, manhole.obj încărcate cu loadOBJ2().



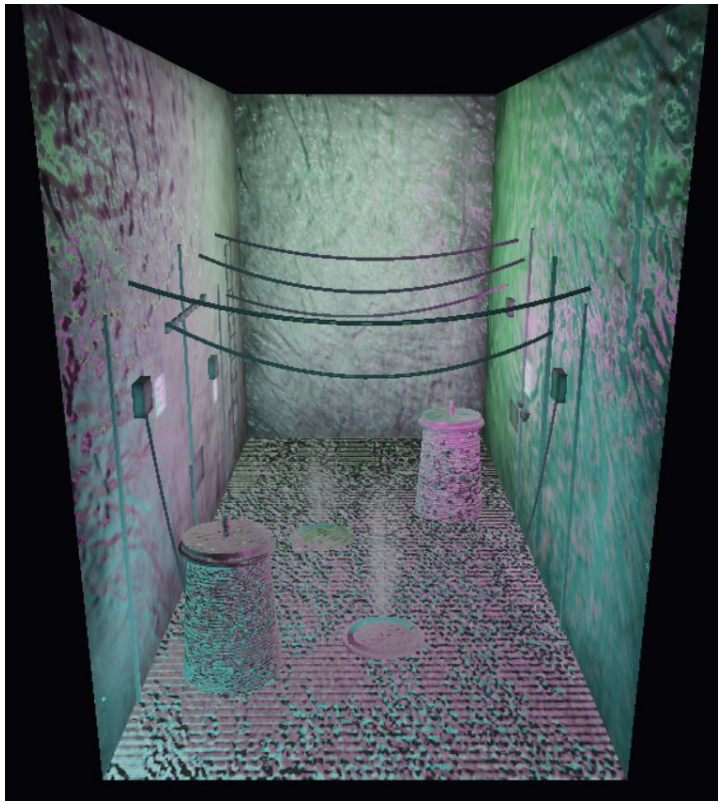
```
static void BuildAlley()
{
   gVertices.clear();
   gVertices.reserve(200000);

   float halfW = 2.2f;
   float len = 10.0f;
   float wallH = 6.0f;

   glm::vec3 tint(1.0f, 1.0f, 1.0f);

   const float TEX_ASPHALT = 0.0f;
   const float TEX_WALL    = 1.0f;
   const float TEX_SIGN    = 2.0f;
   const float TEX_STEAM   = 3.0f;

   // Ground quad (2 triangles)
   groundFirst = (GLint)gVertices.size();
   {
      glm::vec3 p0(-halfW, -len * 0.5f, 0.0f);
      glm::vec3 p1( halfW, -len * 0.5f, 0.0f);
      glm::vec3 p2( halfW,  len * 0.5f, 0.0f);
      glm::vec3 p3(-halfW,  len * 0.5f, 0.0f);

      glm::vec2 uv0(0,0), uv1(1,0), uv2(1,1), uv3(0,1);
      appendQuad(p0,p1,p2,p3, glm::vec3(0,0,1), tint, uv0,uv1,uv2,uv3, TEX_ASPHALT);
   }
   groundCount = (GLsizei)gVertices.size() - groundFirst;

   // Everything after this index is treated as a shadow caster candidate
   castersFirst = (GLint)gVertices.size();

   // Walls
   {
      glm::vec3 p0(-halfW, -len * 0.5f, 0.0f);
      glm::vec3 p1(-halfW,  len * 0.5f, 0.0f);
      glm::vec3 p2(-halfW,  len * 0.5f, wallH);
      glm::vec3 p3(-halfW, -len * 0.5f, wallH);
      glm::vec2 uv0(0,0), uv1(1,0), uv2(1,1), uv3(0,1);
      appendQuad(p0,p1,p2,p3, glm::vec3(1,0,0), tint, uv0,uv1,uv2,uv3, TEX_WALL);
   }

   // ... (other geometry that should cast shadows: pipes, boxes, cables, props)

   // Close the shadow-caster range (everything so far casts shadows)
   shadowCastersCount = (GLsizei)gVertices.size() - castersFirst;
```

```
    // Steam quads are added after and excluded from shadow casting
    steamFirst = (GLint)gVertices.size();
    appendSteamPuff(glm::vec3(0.2f, -2.6f, 0.03f), 2.2f, 0.28f, TEX_STEAM, 1.0f);
    appendSteamPuff(glm::vec3(-0.6f, 0.2f, 0.03f), 1.8f, 0.34f, TEX_STEAM, 0.9f);
    steamCount = (GLsizei)gVertices.size() - steamFirst;
}
```

Texturi (albedo & normal maps)

Texturile sunt încărcate cu SOIL (LoadTexture2D_SOIL), apoi sunt legate pe unități de textură:

- unit 0: texAsphalt
- unit 1: texWall
- unit 2: texSign
- unit 3: texAsphaltN
- unit 4: texWallN



Ce este „albedo” vs „normal map”

- **Albedo** = culoarea de bază (cum arată materialul).
- **Normal map** = o hartă care păcălește lumina să creadă că suprafața are mici denivelări (fisuri, pietriș), fără a crește numărul de poligoane.

```
// Load textures (albedo + normal maps)
texAsphalt = LoadTexture2D_SOIL("asphalt.jpg");
texWall    = LoadTexture2D_SOIL("wall.jpg");
texSign    = LoadTexture2D_SOIL("sign3.png");

texAsphaltN = LoadTexture2D_SOIL("asphalt_n.jpg");
texWallN    = LoadTexture2D_SOIL("wall_n.jpg");

glUseProgram(ProgramId);

// Albedo samplers
glUniform1i(texAsphaltLoc, 0);
glUniform1i(texWallLoc,    1);
glUniform1i(texSignLoc,    2);

// Normal map samplers
glUniform1i(texAsphaltNLoc, 3);
glUniform1i(texWallNLoc,    4);

// Shadow map sampler array -> texture units 5, 6, 7
GLint shadowUnits[LIGHT_COUNT] = {
   SHADOW_TEX_UNIT_BASE + 0,
   SHADOW_TEX_UNIT_BASE + 1,
   SHADOW_TEX_UNIT_BASE + 2
};
glUniform1iv(shadowMapLocation_Main, LIGHT_COUNT, shadowUnits);
```

Vertex Shader (alley.vert)

Ce primește (atribute)

- in_Position (vec4) — poziția vertexului
- in_Color (vec3) — culoare/tint
- in_Normal (vec3) — normal
- in_TexCoord (vec2) — UV
- in_TexId (float) — ID material (0/1/2/3)
- in_Tangent, in_Bitangent (vec3) — pentru normal mapping

Ce calculează

- worldPos = myMatrix * in_Position (poziția în lume)

- normalMat = transpose(inverse(mat3(myMatrix))) (transformare corectă pentru normale)
- Re-orthonormalizare pentru TBN (ca să fie stabil)

Ce trimite la fragment shader

- vColor, vFragPos, vUV, vTexId
- vTBN - matrice care transformă un normal din „spațiul texturii” (normal map) în „spațiul lumii”, ca lumina să se calculeze corect
- vLightPosLS[3] = poziția fragmentului în spațiul fiecărei lumini (folosită pentru shadow mapping)

```
vec4 sampleSurface()
{
   if (useTextures == 0) return vec4(vColor, 1.0);

   if (vTexId == 0) {
      vec2 uv = vUV * texTiling.x;
      return vec4(texture(texAsphalt, uv).rgb * vColor, 1.0);
   }
   if (vTexId == 1) {
      vec2 uv = vUV * texTiling.y;
      return vec4(texture(texWall, uv).rgb * vColor, 1.0);
   }

   // IMPORTANT: steam is not an albedo-textured surface
   if (vTexId == 3) {
      return vec4(vColor, 1.0);
   }

   // sign
   vec2 uv = vUV * texTiling.z;
   vec4 s = texture(texSign, uv);
   s.rgb *= vColor;

   if (signBlackKey == 1) {
      float lum = dot(s.rgb, vec3(0.299, 0.587, 0.114));
      if (lum < 0.05) s.a = 0.0;
   }
   return s;
}
```

Fragment shader (alley.frag)

Alegerea texturii cu vTexId

Funcția sampleSurface():

- dacă useTextures == 0 → folosește doar vColor
- dacă vTexId == 0 → asfalt (texAsphalt) cu tiling texTiling.x
- dacă vTexId == 1 → perete (texWall) cu tiling texTiling.y
- dacă vTexId == 2 → sign (texSign) cu tiling texTiling.z + opțional „black key” (alpha 0 la negru)
- dacă vTexId == 3 → steam (nu folosește albedo texture)

Iluminare (pe scurt)

Pentru fiecare lumină:

- componentă **difuză** (cât de mult „bate” lumina în suprafață)
- componentă **specular**ă (highlight lucios)
- **atenuare** cu distanța (lumina scade când obiectul e departe)

```
vec4 sampleSurface()
{
if (useTextures == 0) return vec4(vColor, 1.0);

if (vTexId == 0) {
vec2 uv = vUV * texTiling.x;
return vec4(texture(texAsphalt, uv).rgb * vColor, 1.0);
}
if (vTexId == 1) {
vec2 uv = vUV * texTiling.y;
```

```
return vec4(texture(texWall, uv).rgb * vColor, 1.0);
}

// IMPORTANT: steam is not an albedo-textured surface
if (vTexId == 3) {
return vec4(vColor, 1.0);
}

// sign
vec2 uv = vUV * texTiling.z;
vec4 s = texture(texSign, uv);
s.rgb *= vColor;

if (signBlackKey == 1) {
float lum = dot(s.rgb, vec3(0.299, 0.587, 0.114));
if (lum < 0.05) s.a = 0.0;
}
return s;
}
```

Normal mapping

Normal mapping este o tehnică prin care schimbăm **normalul folosit la iluminare pentru fiecare pixel**, folosind o textură specială numită **normal map**. Astfel, suprafețele (asfaltul și pereții) par să aibă relief fin (crăpături, asperități) fără să creștem numărul de poligoane. În proiect, avem două normal map-uri: asphalt_n.jpg și wall_n.jpg.

Ca să aplicăm corect normalul din textură, avem nevoie de o bază locală numită **TBN (Tangent–Bitangent–Normal)**. Ideea este: normal map-ul descrie normale în „spațiul texturii” (tangent space), iar noi trebuie să le transformăm în **world space** ca luminile să fie calculate corect.

Pasul 1 — Construirea TBN în vertex shader (alley.vert)

În vertex shader primim, pe lângă poziție/normal/UV, și **tangent** și **bitangent** ca atribute. Le transformăm în world space și construim matricea vTBN, pe care o trimitem la fragment shader. În plus, baza este re-ortogonalizată pentru stabilitate.

```
// alley.vert (fragments)
layout(location=2) in vec3 in_Normal;
layout(location=5) in vec3 in_Tangent;
```

```
layout(location=6) in vec3 in_Bitangent;
uniform mat4 myMatrix;
out mat3 vTBN;
void main()
{
mat3 normalMat = transpose(inverse(mat3(myMatrix)));
vec3 N = normalize(normalMat * in_Normal);
vec3 T = normalize(normalMat * in_Tangent);
vec3 B = normalize(normalMat * in_Bitangent);
// Re-orthonormalize T and B for a stable TBN basis
T = normalize(T - dot(T, N) * N);
B = normalize(cross(N, T));
vTBN = mat3(T, B, N);
// ... rest of the vertex shader (positions, varyings)
}
```

Pasul 2 — Citirea normal map-ului și calculul normalului final în fragment shader (alley.frag)

În fragment shader, funcția sampleNormalWS() decide ce normal folosim:

- dacă useNormalMap == 0, folosim normalul geometric (din vTBN[2])
- nu aplicăm normal mapping pentru **sign** (vTexId == 2) și **steam** (vTexId == 3)
- altfel, citim normal map-ul corespunzător (asfalt/perete), convertim valorile din **0..1** în **-1..1** și transformăm normalul în world space cu vTBN.

```
// alley.frag (fragments)
uniform sampler2D texAsphaltN;
uniform sampler2D texWallN;
uniform int useNormalMap;
uniform vec3 texTiling;
flat in int vTexId;
in vec2 vUV;
in mat3 vTBN;
vec3 sampleNormalWS()
{
   // Base world-space normal (geometric)
   vec3 N = normalize(vTBN[2]);
   // Toggle: normal mapping OFF
   if (useNormalMap == 0) return N;
   // Skip normal mapping for sign and steam materials
   if (vTexId == 2) return N; // Sign
   if (vTexId == 3) return N; // Steam
   // UV tiling differs per material
```

```
    vec2 uv = vUV * (vTexId == 0 ? texTiling.x : texTiling.y);

    // Read tangent-space normal from normal map (0..1 -> -1..1)
    vec3 nTS;
    if (vTexId == 0)
       nTS = texture(texAsphaltN, uv).xyz * 2.0 - 1.0;
    else
       nTS = texture(texWallN, uv).xyz * 2.0 - 1.0;
    // Convert tangent-space normal to world-space using TBN
    return normalize(vTBN * nTS);
}
```

Pasul 3 — Folosirea normalului (modificat) în iluminare

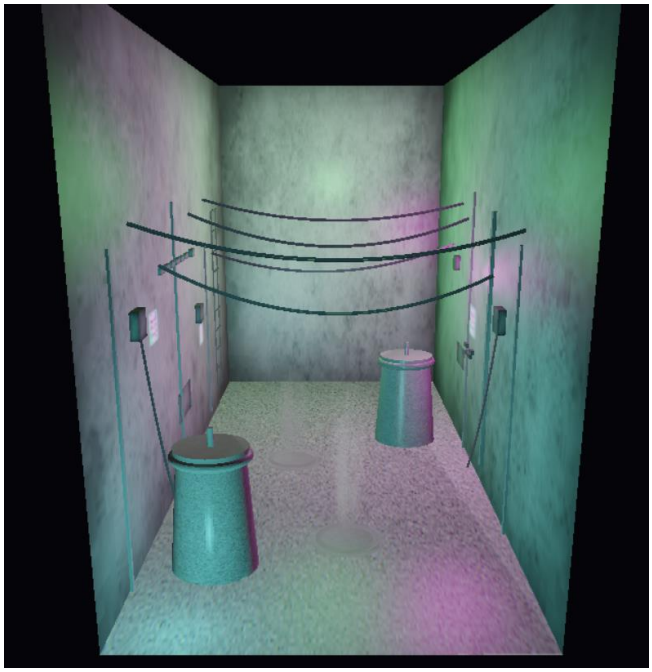
Normalul rezultat (N) intră direct în calculele de lumină difuză și speculară. Asta face ca highlight-urile și umbrele de iluminare să „urmeze” micro-relieful descris de normal map.

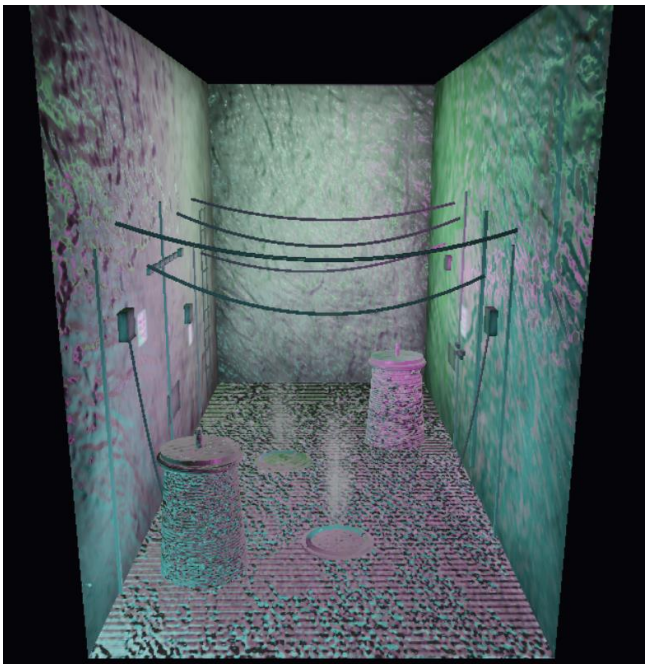
```
// alley.frag (fragments)
vec3 N = sampleNormalWS();
float diff = max(dot(N, L), 0.0);
vec3 R = reflect(-L, N);
float spec = pow(max(dot(V, R), 0.0), shininess);
```

Pasul 4 — Pornire/Oprire normal mapping din tastatură (main.cpp)

În main.cpp, tasta n inversează starea normal mapping-ului și trimite uniforma useNormalMap la shader:

```
C++
// main.cpp (fragment)


case 'n':
{
static int enabled = 1;
enabled = 1 - enabled;
glUseProgram(ProgramId);
glUniform1i(useNormalMapLocation, enabled);
printf("Normal mapping: %s\n", enabled ? "ON" : "OFF");
}
break;
```



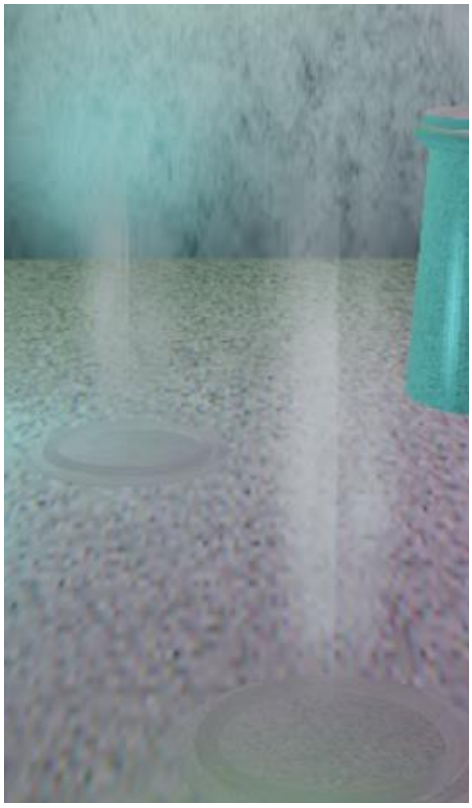


Steam

Când vTexId == 3, shaderul intră pe o ramură separată (nu folosește texturi albedo).

Ideea pe scurt:

- Pleacă de la UV (0..1) și construiește o „coloană” de abur.
- Folosește un **noise** simplu (steamNoise) ca să facă neregularități: goluri, dungi, turbulență.
- Calculează alpha din mai multe măști:
    - mască de formă (mai dens în centru, mai transparent la margini)
    - fade pe verticală (aburul dispare sus)
    - efecte de „holes/streaks” din noise
- Dacă alpha este foarte mic → discard (ca să nu deseneze pixeli aproape invizibili)



Culoarea aburului:

- pornește de la un baseCol (gri-albăstrui)
- este luminată de cele 3 lumini + un mic efect de margine (rim)

```
if (vTexId == 3)
{
vec2 uv = vUV;              // 0..1
vec2 p = uv * 2.0 - 1.0;    // -1..1

float r2 = dot(p, p);
float baseMask = smoothstep(1.35, 0.10, r2);
float edgeSoft = smoothstep(1.0, 0.65, sqrt(r2));

float t = timeSec;
vec2 drift = vec2(0.18 * sin(t * 0.6), 1.10) * t;

float n1 = steamNoise(uv * 5.5  + drift * vec2(0.25, 0.55));
float n2 = steamNoise(uv * 11.0 + drift * vec2(0.35, 0.85) + vec2(2.3, 1.7));
float n3 = steamNoise(uv * 22.0 + drift * vec2(0.55, 1.15) + vec2(5.1, 3.9));

float turb = 0.55 * n1 + 0.30 * n2 + 0.15 * n3;

float bottom = smoothstep(0.00, 0.12, uv.y);
float topFade = smoothstep(1.0, 0.20, uv.y);
float column = bottom * topFade;
```

```
float holes = smoothstep(0.25, 0.8, turb);
float streaks = smoothstep(0.15, 0.95,
steamNoise(vec2(uv.x * 3.0 + turb, uv.y * 14.0 - t * 0.9)));

float intensity = clamp(vColor.r, 0.0, 1.0);

float alpha = baseMask * edgeSoft * column;
alpha *= (0.70 + 0.50 * holes);
alpha *= (0.75 + 0.25 * streaks);
alpha *= (0.70 * intensity);

if (alpha < 0.012) discard;

out_Color = vec4(/*col*/, alpha);
return;
}
```

Fog

Dacă useFog == 1:

- se calculează distanța de la cameră la fragment
- se calculează un factor (0..1) care scade cu distanța
- culoarea finală = amestec între fogColor și result

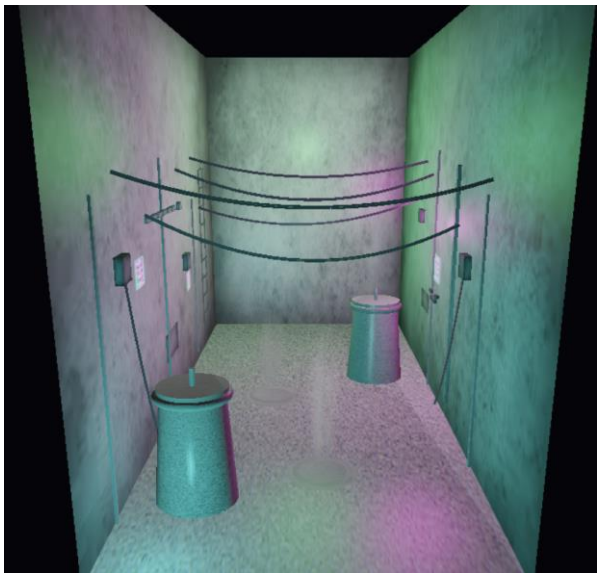
Pe înțeles: obiectele îndepărtate sunt „înghițite” treptat de ceață, ca într-o scenă nocturnă.

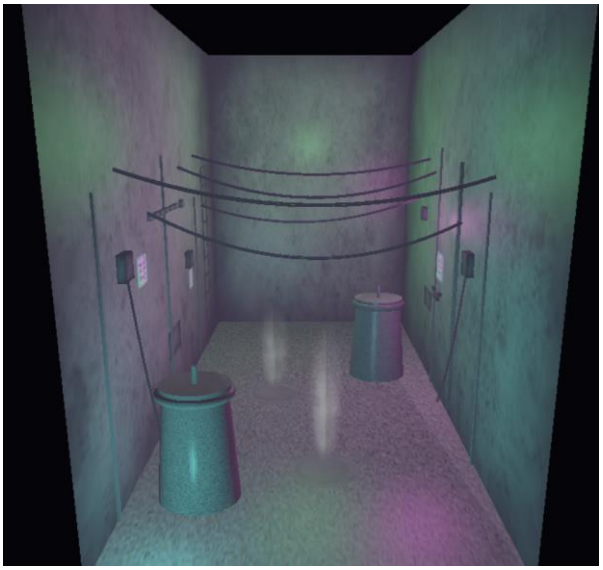
Shadow mapping (3 lumini) — ideea fără detalii grele

- Pentru fiecare lumină există o textură de adâncime shadowMap[i].
- În fragment shader, coordonatele vLightPosLS[i] sunt proiectate în spațiul shadow map.
- Se compară adâncimea curentă cu adâncimea din hartă:
    - dacă fragmentul e „mai în spate” → e în umbră
- Se folosește PCF (o mică medie pe 3x3) ca umbra să fie mai moale.

```
if (useFog == 1) {
   float fogDensity = 0.075;
   vec3 fogColor = vec3(0.045, 0.03, 0.07);
```

```
    float dFog = length(viewPos - vFragPos);

    // Exponential fog factor
    float fogFactor = exp(-fogDensity * dFog);
    fogFactor = clamp(fogFactor, 0.0, 1.0);

    // Artistic tweak (curve + minimum visibility)
    fogFactor = pow(fogFactor, 1.35);
    fogFactor = max(fogFactor, 0.22);

    result = mix(fogColor, result, fogFactor);
}
```





Tone mapping + gamma correction

În shader:

- toneMapReinhard(c) comprimă valorile mari de lumină ca să nu „ardă” imaginea (prea alb).
- applyGamma(c) ajustează imaginea ca să arate corect pe monitoare (care nu sunt liniare).

Practic:

- fără acestea, culorile pot părea fie prea întunecate, fie prea „spălate”.

```
vec3 toneMapReinhard(vec3 c)
{
   c *= max(exposure, 0.0);
   return c / (vec3(1.0) + c);
}

vec3 applyGamma(vec3 c)
{
   float g = max(gammaValue, 1e-4);
   return pow(max(c, vec3(0.0)), vec3(1.0 / g));
}

// Final color output
result = applyGamma(toneMapReinhard(result));
out_Color = vec4(result, alphaOut);
```

Shadow mapping

Shadow mapping este o tehnică prin care obținem **umbre reale** folosind o „hartă de adâncim e” (depth texture) văzută din perspectiva luminii. În proiect, sunt **3 lumini** și pentru fiecare lumin ă se generează o shadow map separată. Programul face randarea în **doi pași**:

1. **Shadow pass (depth-only):** randăm doar adâncimea scenei din perspectiva fiecărei lumini într-o textură (shadow map).
2. **Main pass:** randăm scena normal, iar în fragment shader verificăm dacă un pixel este „în spatele” unui obiect față de lumină (adică în umbră). Pentru margini mai moi, folosim **PCF** (o medie pe un mic vecinaj 3×3).

Pasul 1 — Inițializarea shadow map-urilor (FBO + depth textures) în main.cpp

Pentru fiecare lumină:

- creăm un **FBO** (framebuffer)
- atașăm o textură **GL_DEPTH_COMPONENT** ca „depth attachment”
- setăm CLAMP_TO_BORDER și border color alb ca să tratăm zona din afara hărții ca „lit” (fără umbră)

```
// main.cpp (fragments)
static const int LIGHT_COUNT = 3;
static const int SHADOW_RES = 2048;
GLuint ShadowFBO[LIGHT_COUNT] = { 0, 0, 0 };
GLuint ShadowDepthTex[LIGHT_COUNT] = { 0, 0, 0 };
static void CreateShadowMaps()
{
for (int i = 0; i < LIGHT_COUNT; i++) {
glGenFramebuffers(1, &ShadowFBO[i]);
glBindFramebuffer(GL_FRAMEBUFFER, ShadowFBO[i]);
glGenTextures(1, &ShadowDepthTex[i]);
glBindTexture(GL_TEXTURE_2D, ShadowDepthTex[i]);
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT24,
                SHADOW_RES, SHADOW_RES, 0, GL_DEPTH_COMPONENT,
                GL_FLOAT, NULL);
// Keep filtering simple; PCF is done manually in the fragment shader
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
// Outside shadow map -> treat as fully lit
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
GL_CLAMP_TO_BORDER);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
GL_CLAMP_TO_BORDER);
float borderCol[4] = { 1.f, 1.f, 1.f, 1.f };
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderCol);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
                GL_TEXTURE_2D, ShadowDepthTex[i], 0);
// Depth-only framebuffer
glDrawBuffer(GL_NONE);
glReadBuffer(GL_NONE);
GLenum status = glCheckFramebufferStatus(GL_FRAMEBUFFER);
if (status != GL_FRAMEBUFFER_COMPLETE) {
printf("ERROR: ShadowFBO[%d] incomplete, status=0x%x\n", i, status);
}
}
glBindFramebuffer(GL_FRAMEBUFFER, 0);
glBindTexture(GL_TEXTURE_2D, 0);
}
```

Pasul 2 — Calculul matricii light-space (proiecție + view pentru lumină) în main.cpp

Pentru fiecare lumină construim o matrice lightSpace = lightProj * lightView. Aici folosim o proiecție **ortografică** (bună pentru umbre stabile pe o zonă fixă) și un lookAt din poziția luminii către o țintă fixă.

```
// main.cpp (fragments)
glm::vec3 lightPos[LIGHT_COUNT] = {
glm::vec3(-1.5f, -4.7f, 0.3f),
glm::vec3( 2.0f,  0.5f, 2.9f),
glm::vec3( 0.0f,  2.5f, 4.1f)
};
static glm::mat4 ComputeLightSpace(int li)
{
// Stable target for the shadow camera
glm::vec3 target(0.0f, 0.0f, 1.6f);
glm::vec3 up(0, 0, 1);
glm::vec3 L = lightPos[li];
// Avoid degenerate up vector when light direction is near vertical
if (fabs(glm::dot(glm::normalize(target - L), up)) > 0.98f) {
up = glm::vec3(0, 1, 0);
}
glm::mat4 lightView = glm::lookAt(L, target, up);
// Orthographic volume (tuned for the alley)
float orthoHalfX = 4.0f;
float orthoHalfY = 7.0f;
float nearZ = 0.1f;
float farZ  = 25.0f;
glm::mat4 lightProj = glm::ortho(-orthoHalfX, orthoHalfX,
                                 -orthoHalfY, orthoHalfY,
                                 nearZ, farZ);
return lightProj * lightView;}
```

Pasul 3 — Shadow pass (depth-only render) în main.cpp

În acest pas randăm scena în fiecare ShadowFBO[i]. Folosim un shader foarte simplu care scrie doar gl_Position; adâncimea se scrie automat.

Observație importantă în proiect: **aburul (steam) este exclus din umbre**, deci desenăm doar intervalul castersFirst ... shadowCastersCount.

```
// main.cpp (fragments)
static void RenderShadowPass(const glm::mat4& model,
                             const glm::mat4& lightSpace,
                             int li)
{
glViewport(0, 0, SHADOW_RES, SHADOW_RES);
glBindFramebuffer(GL_FRAMEBUFFER, ShadowFBO[li]);
glClear(GL_DEPTH_BUFFER_BIT);
// Reduce shadow acne using polygon offset during depth pass
glEnable(GL_POLYGON_OFFSET_FILL);
glPolygonOffset(2.0f, 4.0f);
```

```
glUseProgram(ShadowProgramId);
glUniformMatrix4fv(myMatrixLocation_Shadow, 1, GL_FALSE, glm::value_ptr(model));
glUniformMatrix4fv(lightSpaceLocation_Shadow, 1, GL_FALSE,
glm::value_ptr(lightSpace));
glBindVertexArray(SceneVaoId);
// Draw ONLY shadow casters (exclude steam)
glDrawArrays(GL_TRIANGLES, castersFirst, shadowCastersCount);
glBindVertexArray(0);
glUseProgram(0);
glDisable(GL_POLYGON_OFFSET_FILL);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
}
```

Shader-ul de adâncime (shadow_depth.vert / shadow_depth.frag):

```
GLSL
// shadow_depth.vert
#version 330 core
layout(location=0) in vec4 in_Position;
uniform mat4 myMatrix;
uniform mat4 lampLightSpace;
void main()
{
vec4 worldPos = myMatrix * in_Position;
gl_Position = lampLightSpace * worldPos;
}
GLSL
// shadow_depth.frag
#version 330 core
void main()
{
// Depth is written automatically
}
```

Pasul 4 — Trimiterea matricilor și a shadow map-urilor către shader-ul principal (main.cpp)

În main pass:

- calculăm lightSpace[3] și le trimitem ca uniformă array
- legăm shadow map-urile pe unitățile 5,6,7 și setăm array-ul de samplere shadowMap[3]

```
// main.cpp (fragments)

// In the render loop:
glm::mat4 lightSpace[LIGHT_COUNT];
```

```
for (int i = 0; i < LIGHT_COUNT; i++) {
   lightSpace[i] = ComputeLightSpace(i);
}

if (gUseShadowMap) {
   for (int i = 0; i < LIGHT_COUNT; i++) {
      RenderShadowPass(model, lightSpace[i], i);
   }
}
// Bind shadow maps to units 5,6,7
glActiveTexture(GL_TEXTURE5); glBindTexture(GL_TEXTURE_2D,
ShadowDepthTex[0]);
glActiveTexture(GL_TEXTURE6); glBindTexture(GL_TEXTURE_2D,
ShadowDepthTex[1]);
glActiveTexture(GL_TEXTURE7); glBindTexture(GL_TEXTURE_2D,
ShadowDepthTex[2]);

// Send matrices to main shader (mat4 lightSpace[3])
glUniformMatrix4fv(lightSpaceLocation_Main, LIGHT_COUNT, GL_FALSE,
glm::value_ptr(lightSpace[0]));
glUniform1i(useShadowMapLocation, gUseShadowMap);
```

Pasul 5 — Pregătirea coordonatelor pentru testul de umbră (vertex shader alley.vert)

În vertex shader, calculăm poziția fragmentului în spațiul fiecărei lumini și o trimitem la fragment shader:

```
// alley.vert (fragments)
uniform mat4 lightSpace[3];
out vec4 vLightPosLS[3];
void main()
{
vec4 worldPos = myMatrix * in_Position;
// Light-space positions (one per light)
vLightPosLS[0] = lightSpace[0] * worldPos;
vLightPosLS[1] = lightSpace[1] * worldPos;
vLightPosLS[2] = lightSpace[2] * worldPos;
// ... rest of the vertex shader
}
```

Pasul 6 — Testul de umbră + PCF în fragment shader (alley.frag)

În fragment shader, funcția shadowFactorPCF():

- proiectează coordonata în [0..1]

- aplică un **bias** (ca să reducem acne)
- face **PCF 3×3**: verifică 9 eșantioane din shadow map și face media
- returnează 0.0 = luminat, 1.0 = în umbră

```
// alley.frag (fragments)
in vec4 vLightPosLS[3];
uniform sampler2D shadowMap[3];
float shadowFactorPCF(int li, vec3 N, vec3 L)
{
   vec3 proj = vLightPosLS[li].xyz / max(vLightPosLS[li].w, 1e-6);
   proj = proj * 0.5 + 0.5;
   // Outside the shadow map -> lit
   if (proj.x < 0.0 || proj.x > 1.0 ||
      proj.y < 0.0 || proj.y > 1.0 ||
      proj.z < 0.0 || proj.z > 1.0)
      return 0.0;
   // Bias reduces self-shadowing artifacts (shadow acne)
   float bias = max(0.0015 * (1.0 - dot(N, L)), 0.0006);
   vec2 texel = 1.0 / vec2(textureSize(shadowMap[li], 0));
   float shadow = 0.0;
   // 3x3 PCF filter
   for (int y = -1; y <= 1; y++)
   for (int x = -1; x <= 1; x++)
   {
      float closest = texture(shadowMap[li], proj.xy + vec2(x, y) * texel).r;
      float current = proj.z - bias;
      shadow += (current > closest) ? 1.0 : 0.0;
   }
   shadow /= 9.0;
   return shadow;
}
```

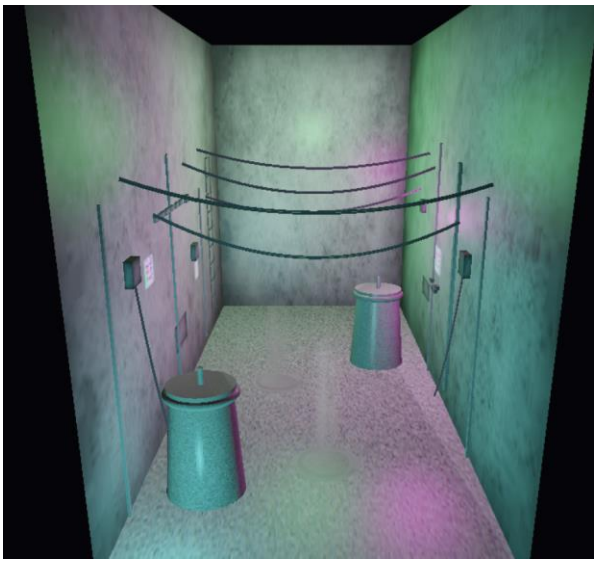
Apoi, în iluminarea principală, factorul de umbră scade contribuția luminii:

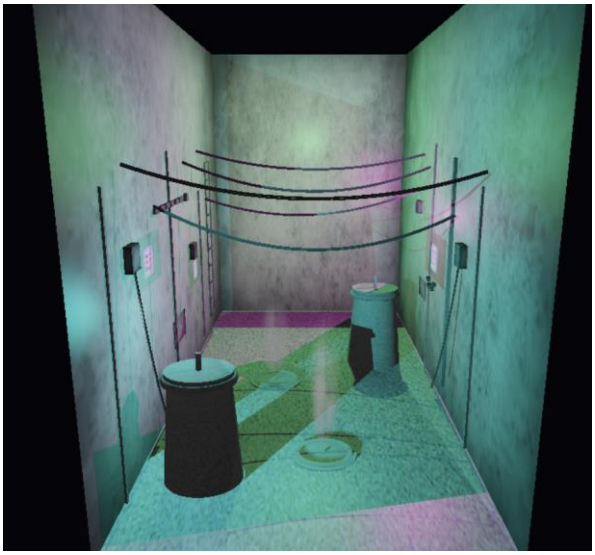
```
// alley.frag (fragments)
float shadow = 0.0;
if (useShadowMap == 1)
shadow = shadowFactorPCF(i, N, L);
result += attenuation * ((1.0 - shadow) * (diffuse + specular));
```

Pasul 7 — Toggle din tastatură (main.cpp)

Umbrele pot fi pornite/oprite cu m, trimițând useShadowMap în shader:

```
case 'm': // toggle shadow mapping
{
gUseShadowMap = 1 - gUseShadowMap;
glUseProgram(ProgramId);
glUniform1i(useShadowMapLocation, gUseShadowMap);
printf("Shadow mapping: %s\n", gUseShadowMap ? "ON" : "OFF");
}
break;
```





Controale

- + / - : zoom (distanța camerei)
- Săgeți: rotire cameră (yaw/pitch) cu cuaternioni
- i / k : mută lumina 0 pe axa Z
- j / l : mută lumina 0 pe axa Y
- n : toggle normal mapping (ON/OFF)
- f : toggle fog (ON/OFF)
- m : toggle shadow mapping (ON/OFF)
- Esc : exit

Contribuții individuale

- Horneț Alex – contruire și incărcare OBJ-uri, camera orbit + zoom, tone mapping +gamma, shadow mapping
- Damian Alexandru – scenă și geometrie (funcțiile helper), texturi și normal maps, fog

Utilizare AI

În implementarea proiectului am folosit un asistent AI (ChatGPT / GitHub Copilot), în special pentru clarificări și verificări rapide, nu pentru generarea integrală a aplicației. Utilizarea AI a fost limitată la:

- **Revizuire/validare de fragmente:** verificarea logicii pentru unele bucăți de shader (de exemplu: fluxul normal mapping → TBN → iluminare, pașii generali ai shadow mapping-ului și rolul bias/PCF)
- **Tuning orientativ (parametri):** sugestii inițiale de valori pentru câțiva parametri (ex.: fogDensity, shininess, bias la umbre, exposure/gamma), urmate de reglaj manual în funcție de rezultatul vizual obținut în aplicație.
- **Debugging și explicarea unor concepte**

Bibliografie

1. **Khronos Group** — *OpenGL 3.3 Core Profile Specification*. https://www.khronos.org/registry/OpenGL/specs/gl/glspec33.core.pdf
2. **Khronos Group** — *The OpenGL Shading Language (GLSL) 3.30 Specification*. https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.3.30.pdf
3. **LearnOpenGL** — *Normal Mapping (TBN, tangent space)*. https://learnopengl.com/Advanced-Lighting/Normal-Mapping
4. **LearnOpenGL** — *Shadow Mapping (depth pass, bias, PCF)*. https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping

Coduri sursă

Fișierele pot fi găsite aici: https://github.com/AlexHornet76/3D-Graphics.git