Documentatie Proiect-2 Grafica pe Calculator

Link cod repozitoriu:

Razvan48/Proiect-2-Grafica-pe-Calculator-GC: Proiect 2 Grafica pe Calculator (GC) Anul 3, Semestrul 1, Facultatea de Matematica si Informatica, Universitatea din Bucuresti

Membri echipa:

- Capatina Razvan Nicolae (352)
- Mihalache Sebastian Stefan (352)
- Luculescu Teodor (351)
- Petrovici Ricardo (351)

Tema aleasa:

Natura

Concept proiect:

Proiectul afiseaza un peisaj din natura. Terenul are un aspect de arhipelag infinit, apa separand insulele pe care se gasesc palmieri si iarba. Apar efecte de reflexie si refractie pentru mediul inconjurator. Pe apa plutesc barci in mod aleatoriu. Camera este libera sa traverseze scena si poate privi de oriunde.

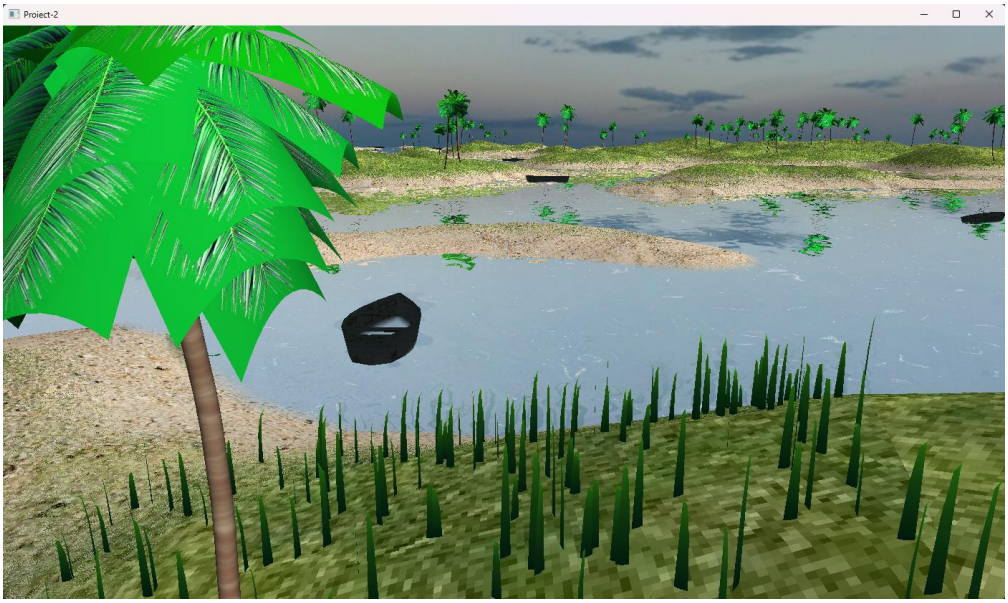
Elemente incluse in proiect:

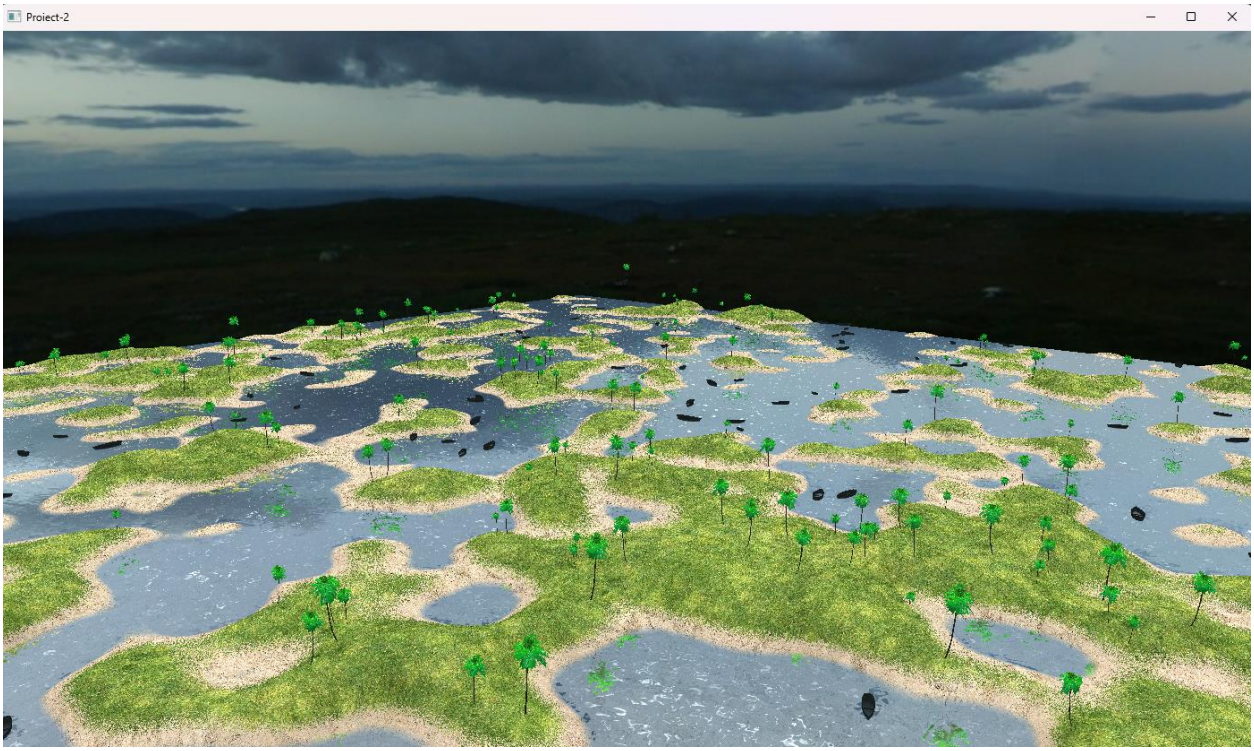
- Perlin Noise 2D
- Teren infinit, separat in chunk-uri
- Skybox
- Multithreading pentru generarea chunk-urilor
- Texturi
- Modelul de iluminare Phong
- Incarcare de obiecte obj folosind biblioteca Tiny Obj Loader
- Apa (se foloseste Normal Mapping si DuDv Mapping + frame buffer objects)
- Iarba (implementarea utilizeaza Compute Shader si Tessellation Shaders)
- Shadow Mapping
- Camera pentru navigarea scenei

Originalitatea proiectului:

Proiectul este o aplicatie multithreaded, acest lucru fiind necesar pentru a nu ingreuna main thread-ul cand geometria chunk-urilor noi este construita. Proiectul implementeaza de asemenea Perlin Noise 2D, iar harta este una infinita, fiind limitata doar de range-ul si precizia variabilei de tip float. Desenarea firelor de iarba se foloseste de curbele Bezier si are loc in Compute Shader si Tessellation Shaders. Apa din scena are un aspect realistic, prin redarea reflexiei si a refractiei celorlalte elemente, cu distorsionare folosind DuDv mapping si iluminare folosind normal mapping.

Imagini ale scenei construite de aplicatie:

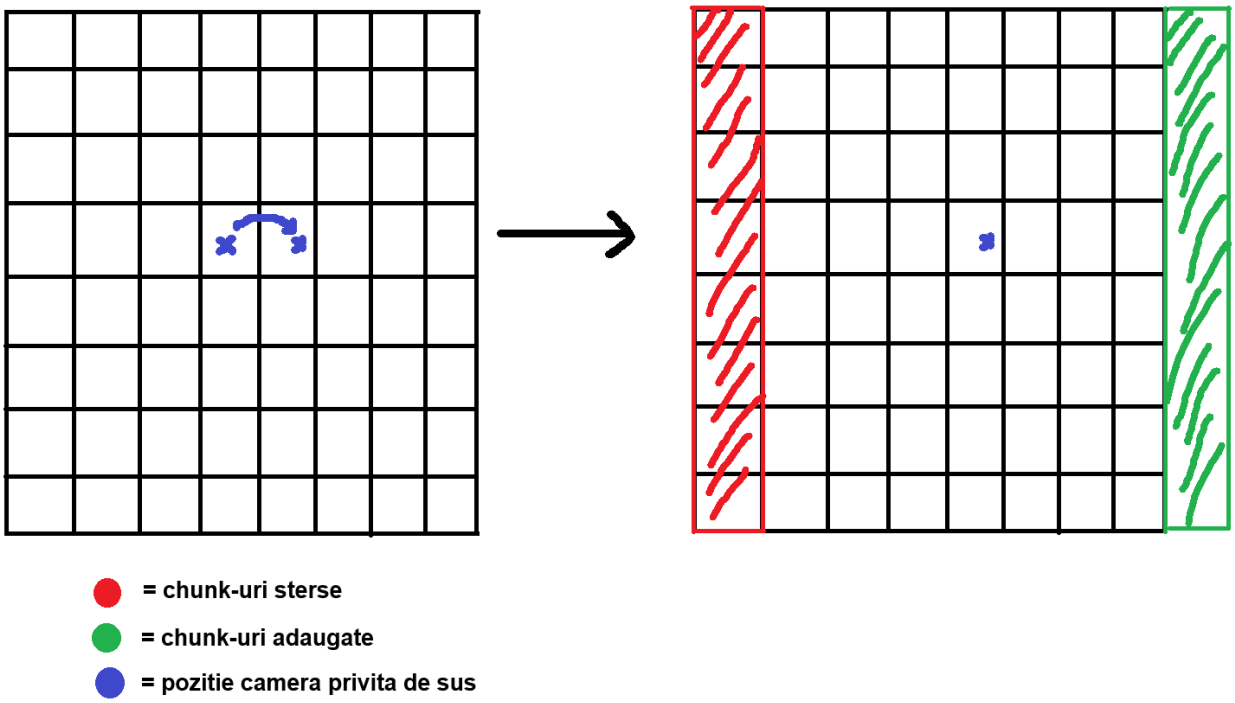




Aspecte interesante ale proiectului:

1. **Multithreading-ul aplicatiei:**

Atunci cand camera paraseste chunk-ul curent si trece intr-un nou chunk trebuie generata o complet noua linie de chunk-uri in departare.



Acest lucru ingreuneaza main thread-ul, deoarece trebuie construita intreaga geometrie pe RAM (incluzand si apelurile catre Perlin Noise, care aduc un overhead notabil), dupa care trebuie alocata memorie pe GPU si mutata aceasta informatie folosind glBufferData.

O solutie este sa pastram cat mai putin din calculele intense computational in main thread si sa creem un thread nou pentru fiecare chunk ce trebuie construit. Un impediment pe care l-am intalnit aici este ca metodele de OpenGL (glBindVertexArray, glBindBuffer, glBufferData, etc.) nu au suport de multithreading, ceea ce inseamna ca avem nevoie de un mecanism de lock/unlock cu mutexi de fiecare data cand vrem sa alocam memorie pe GPU si sa copiem informatii din RAM. Pentru a nu avea un mutex global ce trebuia folosit de fiecare data cand apelam metode de OpenGL am decis sa lasam aceste apeluri in main thread si doar constructia in RAM a geometriei chunk-ului sa o realizam pe un thread separat, urmand apoi ca main thread-ul sa aloce memoria pe GPU si sa copieze datele.

Astfel avem:

```
void Map::draw(GLuint modelProgramID)
{
    this->mapChunksMutex.lock();
    for (int i = 0; i < this->mapChunks.size(); ++i)
        if (this->mapChunks[i].getOpenGLSetupDone())
            this->mapChunks[i].draw(modelProgramID);
    this->mapChunksMutex.unlock();



    this->mapChunksMutex.lock();
    for (int i = 0; i < this->mapChunks.size(); ++i)
    {
        if (!this->mapChunks[i].getOpenGLSetupDone())
        {
            if (GlobalClock::get().getCurrentTime() - this->lastTimeLoadedOpenGL > this->TIME_BETWEEN_OPENGL_LOADS)
            {
                this->mapChunks[i].setupOpenGL();
                this->lastTimeLoadedOpenGL = GlobalClock::get().getCurrentTime();
            }
        }
    }
    this->mapChunksMutex.unlock();
}
```

Clasa singleton Map deseneaza doar chunk-urile carora li s-au apelat deja metodele de OpenGL (pe langa faptul ca au geometria incarcata in RAM). Apoi mai parcurge inca o data chunk-urile unde nu s-au apelat metodele de OpenGL si rezolva acest lucru pentru maxim un chunk intr-un interval de timp setat (TIME_BETWEEN_OPENGL_LOADS). Se observa mecanismele ce implica mutexi si lock/unlock pentru a evita concurenta.

```
68  void Map::update()
69  {
70      MapChunk::commonUpdate();
71
72      this->mapChunksMutex.lock();
73      for (int i = 0; i < this->mapChunks.size(); ++i)
74          this->mapChunks[i].update();
75      this->mapChunksMutex.unlock();
76
77
78
79      std::vector<std::vector<bool>> chunksAlreadyLoaded;
80      chunksAlreadyLoaded.resize(this->NUM_CHUNKS_AHEAD * 2 + 1);
81      for (int i = 0; i < chunksAlreadyLoaded.size(); ++i)
82          chunksAlreadyLoaded[i].resize(this->NUM_CHUNKS_AHEAD * 2 + 1, false);
83
84      int cameraChunkX = MapChunk::calculateChunkX(Camera::get().getPosition().x);
85      int cameraChunkY = MapChunk::calculateChunkY(Camera::get().getPosition().z);
86
87      this->mapChunksMutex.lock();
88      for (int i = 0; i < this->mapChunks.size(); ++i)
89      {
90          int chunkX = this->mapChunks[i].getX();
91          int chunkY = this->mapChunks[i].getY();
92
93          if (abs(chunkX - cameraChunkX) > this->NUM_CHUNKS_AHEAD || abs(chunkY - cameraChunkY) > this->NUM_CHUNKS_AHEAD)
94          {
95              std::swap(this->mapChunks[i], this->mapChunks.back());
96              this->mapChunks.pop_back();
97              --i;
98          }
99          else
100             chunksAlreadyLoaded[chunkX - cameraChunkX + this->NUM_CHUNKS_AHEAD][chunkY - cameraChunkY + this->NUM_CHUNKS_AHEAD] = true;
101     }
102     this->mapChunksMutex.unlock();
103
104     for (int i = -this->NUM_CHUNKS_AHEAD; i <= this->NUM_CHUNKS_AHEAD; ++i)
105     {
106         for (int j = -this->NUM_CHUNKS_AHEAD; j <= this->NUM_CHUNKS_AHEAD; ++j)
107         {
108             if (!chunksAlreadyLoaded[i + this->NUM_CHUNKS_AHEAD][j + this->NUM_CHUNKS_AHEAD] && cameraChunkX + i >= 0 && cameraChunkY + j >= 0
109                 && !this->isChunkBeingLoaded(cameraChunkX + i, cameraChunkY + j))
110             {
111                 this->addChunkToBeingLoaded(cameraChunkX + i, cameraChunkY + j);
112
113                 std::thread mapChunkThread([this, cameraChunkX, cameraChunkY, i, j]()
114                     {
115                         MapChunk mapChunk(cameraChunkX + i, cameraChunkY + j);
116                         this->addMapChunk(mapChunk);
117                         this->removeChunkFromBeingLoaded(cameraChunkX + i, cameraChunkY + j);
118                     });
119                 mapChunkThread.detach();
120             }
121         }
122     }
123 }
```

In metoda Map::update() verificam daca trebuie sa adaugam noi chunk-uri la harta (camera s-a deplasat dintr-un chunk in altul), iar pe linia 113 se creeaza un thread nou care va construi geometria in RAM.

```
125  void Map::addMapChunk(MapChunk& mapChunk)
126  {
127      this->mapChunksMutex.lock();
128      this->mapChunks.push_back(std::move(mapChunk));
129      this->mapChunksMutex.unlock();
130  }
131
132  void Map::addChunkToBeingLoaded(int x, int y)
133  {
134      this->chunksAlreadyBeingLoadedMutex.lock();
135      this->chunksAlreadyBeingLoaded.insert(std::make_pair(x, y));
136      this->chunksAlreadyBeingLoadedMutex.unlock();
137  }
138
139  void Map::removeChunkFromBeingLoaded(int x, int y)
140  {
141      this->chunksAlreadyBeingLoadedMutex.lock();
142      this->chunksAlreadyBeingLoaded.erase(std::make_pair(x, y));
143      this->chunksAlreadyBeingLoadedMutex.unlock();
144  }
145
146  bool Map::isChunkBeingLoaded(int x, int y)
147  {
148      this->chunksAlreadyBeingLoadedMutex.lock();
149      bool isChunkInSet = (this->chunksAlreadyBeingLoaded.find(std::make_pair(x, y)) != this->chunksAlreadyBeingLoaded.end());
150      this->chunksAlreadyBeingLoadedMutex.unlock();
151
152      return isChunkInSet;
153  }
```

Avem si structuri de date precum std::set, care memoreaza care sunt chunk-urile in proces de construire, ca sa nu exista situatia in care se aloca 2 thread-uri diferite pentru construirea aceluiasi chunk. Thread-urile au responsabilitatea sa se elimine din aceasta structura de date o data ce au terminat, asa cum se poate observa pe linia 117 din imaginea anterioara.

Rezultatul acestei implementari este ca nu exista frame drop-uri atunci cand se trece dintr-un chunk in altul si aplicatia suporta sa deseneze o suprafata mare in jurul camerei fara pierdere de performanta.

2. Iarba care tine cont de forta vantului

Implementarea foloseste curbe Bezier pentru a reprezenta firele individuale de iarba. Utilizeaza un Compute Shader pentru a simula fortele folosind metoda Euler si pentru a aplica diverse metode de culling. Ulterior, bufferul de curbe este transmis unui Tessellation Shader pentru a genera dinamic geometria de triunghiuri pentru firele de iarba.

Compute Shader

```
#version 450

#define WORKGROUP_SIZE 32
layout(local_size_x = WORKGROUP_SIZE,
local_size_y = 1,
local_size_z = 1) in;

layout(binding = 0) uniform CameraBufferObject {
    mat4 view;
    mat4 proj;
} camera;

uniform float current_time;
uniform float delta_time;

uniform float wind_magnitude;
uniform float wind_wave_length;
uniform float wind_wave_period;

struct Blade {
    vec4 v0;
    vec4 v1;
    vec4 v2;
    vec4 up;
};

layout(binding = 1, std140) buffer inputBuffer {
    Blade inputBlades[];
};

layout(binding = 2, std140) buffer outputBuffer {
    Blade outputBlades[];
};

// Indirect drawing count
layout(binding = 3) buffer NumBlades {
    uint vertexCount;
    uint instanceCount;// = 1
    uint firstVertex;// = 0
    uint firstInstance;// = 0
} numBlades;

bool inBounds(float value, float bounds) {
    return (value >= -bounds) && (value <= bounds);
}

```

```
47      float rand(float seed) {
48          return fract(sin(seed)*100000.0);
49      }
50
51      void main() {
52          // Reset the number of blades to 0
53          if (gl_GlobalInvocationID.x == 0) {
54              numBlades.vertexCount = 0;
55          }
56          barrier();// Wait till all threads reach this point
57
58          uint index = gl_GlobalInvocationID.x;
59          vec3 v0 = inputBlades[index].v0.xyz;
60          vec3 v1 = inputBlades[index].v1.xyz;
61          vec3 v2 = inputBlades[index].v2.xyz;
62          vec3 up = normalize(inputBlades[index].up.xyz);
63          float orientation = inputBlades[index].v0.w;
64          float height = inputBlades[index].v1.w;
65          float width = inputBlades[index].v2.w;
66          float stiffness = inputBlades[index].up.w;
67
68          // Frustum culling
69          vec4 v0ClipSpace = camera.proj * camera.view * vec4(v0, 1);
70          vec4 v1ClipSpace = camera.proj * camera.view * vec4(v1, 1);
71          v0ClipSpace /= v0ClipSpace.w;
72          v1ClipSpace /= v1ClipSpace.w;
73
74          bool v0OutFrustum =
75          v0ClipSpace.x < -1 || v0ClipSpace.x > 1
76          || v0ClipSpace.y < -1 || v0ClipSpace.y > 1;
77
78          bool v1OutFrustum =
79          v1ClipSpace.x < -1 || v1ClipSpace.x > 1
80          || v1ClipSpace.y < -1 || v1ClipSpace.y > 1;
81          if (v0OutFrustum && v1OutFrustum) return;
82
```

```
        // Distance culling
        const float far1 = 0.95;
        if (v0ClipSpace.z > far1 && v1ClipSpace.z > far1 && rand(index) > 0.5) {
            return;
        }
        const float far2 = 0.98;
        if (v0ClipSpace.z > far2 && v1ClipSpace.z > far2 && rand(index) > 0.2) {
            return;
        }
        const float far3 = 0.99;
        if (v0ClipSpace.z > far3 && v1ClipSpace.z > far3 && rand(index) > 0.1) {
            return;
        }

        // Apply forces {
            //  Gravities
            vec3 gE = vec3(0, -0.98, 0);
            vec3 widthDir = vec3(cos(orientation), 0, sin(orientation));
            vec3 bladeFace = normalize(cross(up, widthDir));
            vec3 gF = 0.25*length(gE)*bladeFace;
            vec3 g = gE + gF;

            //  Recovery
            vec3 r = (v0 + up * height - v2) * stiffness;

            //  Wind
            vec3 windForce = 0.25 * wind_magnitude *
            vec3(
            sin(current_time * 3. / wind_wave_period + v0.x * 0.1 * 11 / wind_wave_length),
            0,
            sin(current_time * 3. / wind_wave_period + v0.z * 0.2 * 11 / wind_wave_length) * 0.1
            );
            float fd = 1 - abs(dot(normalize(windForce), normalize(v2 - v0)));
            float fr = dot((v2 - v0), up) / height;
            vec3 w = windForce * fd * fr;

            v2 += (0.1 * g + r + w) * delta_time;

            float lproj = length(v2 - v0 - up * dot((v2-v0), up));
            v1 = v0 + height*up*max(1-lproj/height, 0.05*max(lproj/height, 1));

            inputBlades[index].v1.xyz = v1;
            inputBlades[index].v2.xyz = v2;
        // }

        outputBlades[atomicAdd(numBlades.vertexCount, 1)] = inputBlades[index];
    }
```

Vertex Shader

```
#version 450

layout(location = 0) in vec4 v0;
layout(location = 1) in vec4 v1;
layout(location = 2) in vec4 v2;
layout(location = 3) in vec4 up;

out VS_OUT
{
  vec4 v1;
  vec4 v2;
  vec4 up;
  vec4 dir;
} vs_out;

void main() {
  vs_out.v1 = v1;
  vs_out.v2 = v2;
  vs_out.up = vec4(normalize(up.xyz), up.w);

  float angle = v0.w;

  vec3 dir = normalize(cross(vs_out.up.xyz,
                             vec3(sin(angle),
                                  0,
                                  cos(angle))));

  vs_out.dir = vec4(dir, 0.0);

  gl_Position = v0;
}
```

Tesselation Control Shader

```
#version 450

layout(binding = 0) uniform CameraBufferObject {
    mat4 view;
    mat4 proj;
} camera;

in VS_OUT
{
  vec4 v1;
  vec4 v2;
  vec4 up;
  vec4 dir;
} tesc_in[];

patch out TESC_OUT
{
  vec4 v1;
  vec4 v2;
  vec4 up;
  vec4 dir;
} tesc_out;

layout(vertices = 1) out;

void main() {
  gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;

  tesc_out.v1 = tesc_in[gl_InvocationID].v1;
  tesc_out.v2 = tesc_in[gl_InvocationID].v2;
  tesc_out.up = tesc_in[gl_InvocationID].up;
  tesc_out.dir = tesc_in[gl_InvocationID].dir;

  float z1 = (camera.proj * camera.view * vec4(tesc_out.v1.xyz, 1)).z;
  float z2 = (camera.proj * camera.view * vec4(tesc_out.v2.xyz, 1)).z;


  if (z1 < 0.8 && z2 < 0.8) {
    gl_TessLevelInner[0] = 2.0;
    gl_TessLevelInner[1] = 7.0;
    gl_TessLevelOuter[0] = 7.0;
    gl_TessLevelOuter[1] = 2.0;
    gl_TessLevelOuter[2] = 7.0;
    gl_TessLevelOuter[3] = 2.0;
  } else {
    gl_TessLevelInner[0] = 1.0;
    gl_TessLevelInner[1] = 3.0;
    gl_TessLevelOuter[0] = 3.0;
    gl_TessLevelOuter[1] = 1.0;
    gl_TessLevelOuter[2] = 3.0;
    gl_TessLevelOuter[3] = 1.0;
  }

}
```

Tesselation Evaluation Shader

```
#version 450

layout(quads, equal_spacing, ccw) in;

layout(binding = 0) uniform CameraBufferObject {
    mat4 view;
    mat4 proj;
} camera;

patch in TESC_OUT
{
  vec4 v1;
  vec4 v2;
  vec4 up;
  vec4 dir;
} tese_in;

out TESE_OUT
{
  vec3 normal;
  vec2 uv;
} tese_out;

void main() {
  const float u = gl_TessCoord.x;
  const float v = gl_TessCoord.y;

  const vec3 v0 = gl_in[0].gl_Position.xyz;
  const vec3 v1 = tese_in.v1.xyz;
  const vec3 v2 = tese_in.v2.xyz;
  const vec3 dir = tese_in.dir.xyz;
  const float width = tese_in.v2.w;

  // interpolarea punctelor pe patch => alungirea folosind width si height
  const vec3 a = v0 + v * (v1 - v0);
  const vec3 b = v1 + v * (v2 - v1);
  const vec3 c = a + v * (b - a);
  const vec3 t1 = dir;

  const vec3 c0 = c - t1 * width * 0.5;
  const vec3 c1 = c + t1 * width * 0.5;
  const vec3 t0 = normalize(b - a);

  const float t = u + 0.5 * v - u * v;
  const vec3 p = (1.0 - t) * c0 + t * c1;

  tese_out.uv = vec2(u, v);
  tese_out.normal = normalize(cross(t0, t1));
  gl_Position = camera.proj * camera.view * vec4(p, 1.0);
}
```

Fragment Shader

```
#version 450

layout(binding = 0) uniform CameraBufferObject {
    mat4 view;
    mat4 proj;
} camera;

in TESE_OUT
{
  vec3 normal;
  vec2 uv;
} frag_in;

layout(location = 0) out vec4 outColor;

void main() {
  vec3 normal = frag_in.normal;
  vec2 uv = frag_in.uv;

  vec3 upperColor = vec3(0.4,1,0.1);
  vec3 lowerColor = vec3(0.0,0.2,0.1);

  vec3 sunDirection = normalize(vec3(-1.0, 5.0, -3.0));

  vec3 upperDarkColor = vec3(0.2,0.75,0.05);
  vec3 lowerDarkColor = vec3(0.0,0.5,0.05);

  float NoL = clamp(dot(normal, sunDirection), 0.5, 1.0);

  vec3 mixedColor = mix(lowerColor, upperColor, uv.y);

  outColor = vec4(mixedColor*NoL, 1.0);
}
```

3. Apa

Ideea generala de redare a apei:

a) Apa este un singur dreptunghi mare cat intreaga harta, asezat intr-un plan orizontal, paralel cu Oxz, in felul acesta calculele nu sunt prea intense (avem numai 6 varfuri – 2 triunghiuri)
b) Pe suprafata apei trebuie desenate 2 imagini (texturi): obiectele reflectate (aflate deasupra planului apei) si obiectele refractate (aflate sub planul apei – zonele din harta cu inaltimea (coordonata y) mai mica decat planul apei), cu un anumit factor de mixare intre ele
c) Cele 2 texturi trebuie distorsionate, altfel apa ar fi ca o oglinda (efect nerealist)
d) Efect de iluminare a apei, in special sclipirile valurilor (componenta *speculara*)
e) Efecte de adancime si corectare a diverselor probleme care pot aparea in randare

Detalii despre concepte si implementare :

a) Geometria apei

Apa este initial un patrat centrat in origine (0.0, 0.0, 0.0) si normalizat (latura de dimensiune 1.0). Este stabilita de la inceput o inaltime convenabila a planului apei (o constanta: *static const GLfloat WATER_HEIGHT;*), astfel apa va inunda orice zona din harta mai joasa decat *WATER_HEIGHT*. Cum harta este generata dinamic, in functie de cum se deplaseaza camera, dorim sa scalam si sa translatam dreptunghiul apei astfel incat sa se suprapuna complet cu harta. Aflam coordonatele coltului hartii cu *x* si *z* minime, respectiv maxime, furnizate de harta, de unde putem calcula atat scalarea de-a lungul fiecarei axe, cat si pozitia unde trebuie sa translatam centrul apei:

```
// model
glm::vec2 Map_minXZ = Map::get().getTopLeftCornerOfVisibleMap();
glm::vec2 Map_maxXZ =
Map::get().getBottomRightCornerOfVisibleMap();
GLfloat scaleFactorX = abs(Map_maxXZ.x - Map_minXZ.x);
GLfloat scaleFactorZ = abs(Map_maxXZ.y - Map_minXZ.y);
GLfloat posWaterX = (Map_maxXZ.x + Map_minXZ.x) / 2.0;
GLfloat posWaterZ = (Map_maxXZ.y + Map_minXZ.y) / 2.0;

glm::mat4 scale = glm::scale(glm::mat4(1.0f),
glm::vec3(scaleFactorX, 1.0f, scaleFactorZ));
glm::mat4 model = glm::translate(glm::mat4(1.0f),
glm::vec3(posWaterX, this->height, posWaterZ)) * scale;
```

Acum apa acopera toata harta, rezultand aspectul de arhipelag.

b) Imaginea apei (texturi)

Avem nevoie sa redam pe suprafata apei 2 imagini: reflexia si refractia. Pentru a obtine aceste 2 imagini, trebuie sa redam scena din perspectiva curenta a camerei (pentru refractie) si din perspectiva simetrica fata de planul apei, adica din pozitia de unde s-ar "vedea" reflexia, apoi sa salvam aceste imagini sub forma de texturi si sa le mapam corespunzator pe suprafata apei.

Pentru a obtine cele 2 texturi, folosim *frame buffer objects (FBO)*. Am creat o clasa *class WaterFrameBuffers{...};* unde tinem 2 *FBO*-uri, unul pentru reflexie (cu atasament de culoare: *glFramebufferTexture(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, texture, 0);*) si unul pentru refractie (cu atasament de culoare si de adancime: *glFramebufferTexture(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, texture, 0);*). Astfel, in programul principal am creat o functie *drawAllObjectsExceptWater()* unde se apeleaza functiile de desenare pentru toate elementele din scena cu exceptia apei, iar pentru a calcula texturile apei se leaga *FBO*-ul pentru reflexie, se pozitioneaza camera simetric fata de apa, se deseneaza toate obiectele, se aduce camera in pozitia initiala, apoi se leaga *FBO*-ul pentru refractie si se deseneaza din nou toate obiectele.

```
void computeWater()
{
glEnable(GL_CLIP_DISTANCE0);
// create the reflection of all objects
fbos->bindReflectionFrameBuffer();

// Repositioning of the camera for the reflection
GLfloat distance = 2 * (Camera::get().getPosition().y - water-
>getHeight());
glm::vec3 cameraPos = Camera::get().getPosition();
cameraPos.y -= distance;
Camera::get().setPosition(cameraPos);
Camera::get().invertPitch();
setClippingPlane(0, 1, 0, -water->getHeight() + 0.02f);

// Draw objects
drawAllObjectsExceptWater();

// Bring camera to the initial position
cameraPos.y += distance;
Camera::get().setPosition(cameraPos);
Camera::get().invertPitch();

fbos->unbindCurrentFrameBuffer();

// end create reflection


// create the refraction of all objects
fbos->bindRefractionFrameBuffer();
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

```
setClippingPlane(0, -1, 0, water->getHeight() + 0.1f);

// Draw objects
drawAllObjectsExceptWater();

fbos->unbindCurrentFrameBuffer();
// end create refraction

glDisable(GL_CLIP_PLANE0);
}
```

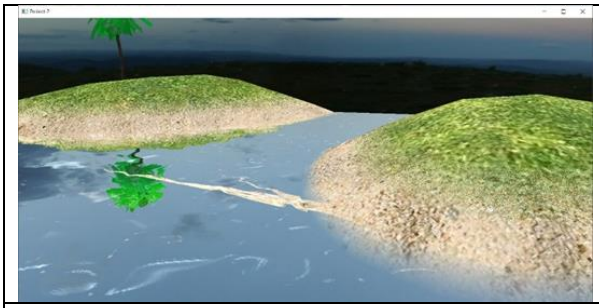
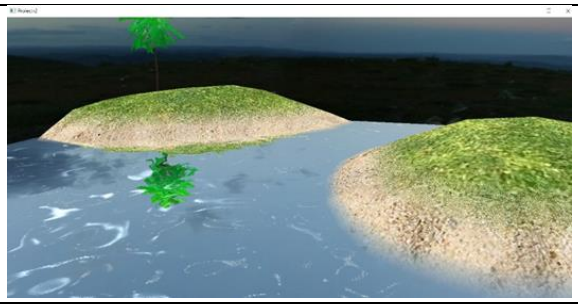
Un lucru important este functia *setClippingPlane()*:

```
void setClippingPlane(GLfloat x, GLfloat y, GLfloat z, GLfloat w)
{
GLint programInUse;
glGetIntegerv(GL_CURRENT_PROGRAM, &programInUse);

glUseProgram(Map::get().getProgramId());
glUniform4f(glGetUniformLocation(Map::get().getProgramId(), "plane"),
x, y, z, w);

glUseProgram(modelProgramID);
glUniform4f(glGetUniformLocation(modelProgramID, "plane"), x, y, z,
w);

glUseProgram(programInUse);
}
```

, asociata cu urmatoarele linii din vertex shader-ele obiectelor care fac parte din reflexie si refractie:

```
uniform vec4 plane;

// in void main()
    vec4 worldPosition = model * vec4(inPosition, 1.0);
    gl_ClipDistance[0] = dot(worldPosition, plane);
```

si care are urmatorul rol: cand cream imaginea reflectata, nu trebuie sa includem in ea si ce se afla sub apa (in zonele in care apa e putin adanca, camera translatata simetric sub apa va vedea si fata inferioara a solului de sub apa, in loc sa vada direct spre cer). Un efect asemanator s-ar intampla si pentru refractie.

	
Doar textura de reflexie **fara** clipping	Doar textura de reflexie **cu** clipping

Maparea celor doua texturi:

Cand asezam texturile pe apa, vrem ca pe suprafata apei sa avem ceea ce „incape” in patrulaterul (trapezul) de pe ecran care reprezinta apa (nu vrem colturile apei sa fie mapate exact pe colturile texturii), pentru ca se obtine un efect anormal:

In imagine este doar textura de refractie, dar se vede comportamentul anormal: in trapezul apei a ajuns exact ceea ce se vede in intrega fereastra (pentru ca textura contine tot continutul ferestrei).

Dorim deci ca in trapezul apei sa avem ca imagine exact ceea ce se reda in interiorul acestui trapez. Prin urmare coordonatele de texturare trebuie supuse acelorasi transformari pe care le sufera si varfurile (modelare, vizualizare, proiectie).

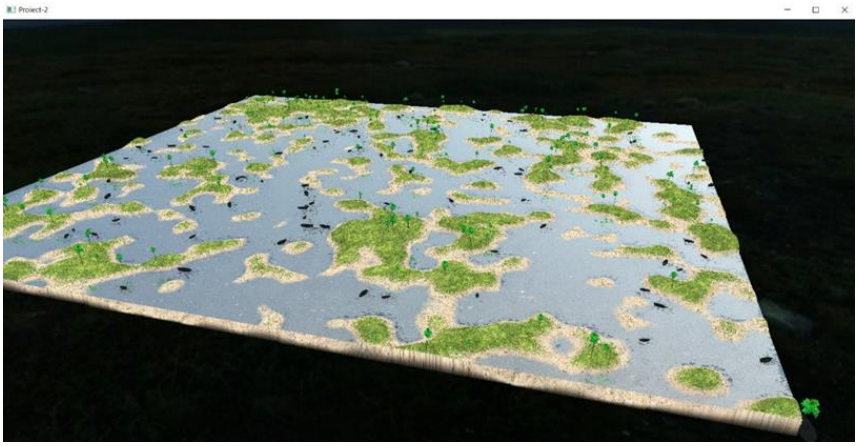
Astfel, coordonatele de texturare pe care le trimitem din vertex shader sunt coordonatele varfurilor dupa aplicarea transformarilor de modelare, de vizualizare si de proiectie (variabila de iesire `clipSpace`).

```
vec4 worldPosition = model * vec4(inPosition, 1.0);
clipSpace = projection * view * worldPosition;
```

In fragment shader executam *perspective division* (impartirea coordonatelor prin ultima componenta, *w*), ceea ce OpenGL aplica asupra varfurilor la finalul shader-ului de varfuri:

```
vec2 ndc = (clipSpace.xy/clipSpace.w)/2.0 + 0.5;

vec2 refractTexCoords = vec2(ndc.x, ndc.y);
vec2 reflectTexCoords = vec2(ndc.x, 1.0 - ndc.y);
```

Astfel obtinem pe suprafata apei imaginea corespunzatoare:



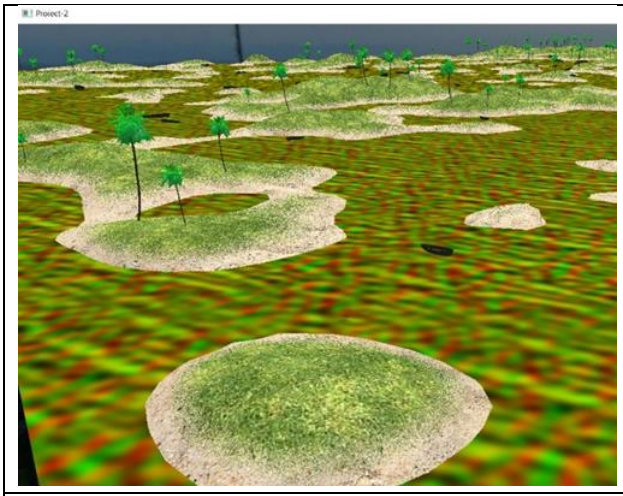
c) Distorsionarea texturilor (efect de valuri)

Cum apa este complet plana, daca doar redam texturile pe suprafata ei, ar parea ca este o oglinda :

Pentru a nu avea acest aspect, folosim o tehnica numita *dudv mapping*. Ideea este ca, in loc sa texturam imaginea reflectata sau refractata de la pozitia *(x, y)*, sa texturam cu o anumita deviere, de la pozitia *(x + du, y + dv)*, unde offset-urile *du* si *dv* sunt valorile de rosu, respectiv verde dintr-o textura speciala, numita *dudv map*. Din cod se incarca textura *dudv*, iar in fragment shader se calculeaza deviatiile de-a lungul celor 2 axe:

```
vec2 distortedTexCoords = texture(dudvMap, vec2(textureCoords.x +
moveFactor, textureCoords.y)).rg * 0.1;

distortedTexCoords = textureCoords + vec2(distortedTexCoords.x,
distortedTexCoords.y + moveFactor);

vec2 totalDistortion = (texture(dudvMap, distortedTexCoords).rg * 2.0
- 1.0) * waveStrength;

refractTexCoords += totalDistortion;
refractTexCoords = clamp(refractTexCoords, 0.001, 0.999);

reflectTexCoords += totalDistortion;
reflectTexCoords = clamp(reflectTexCoords, 0.001, 0.999);

vec4 reflectionColor = texture(reflectionTexture, reflectTexCoords);
vec4 refractionColor = texture(refractionTexture, refractTexCoords);
```

In plus, pentru a nu parea ca imaginea este prea statica, ne deplasam pe suprafata texturii *dudv* cu o anumita viteza (`moveFactor`), astfel ca valurile par ca se misca pe suprafata apei.
Efectul rezultat este:



Textura *dudv* pe suprafata apei

Imaginea apei deformata de *dudv mapping*

d) Efect de iluminare a apei

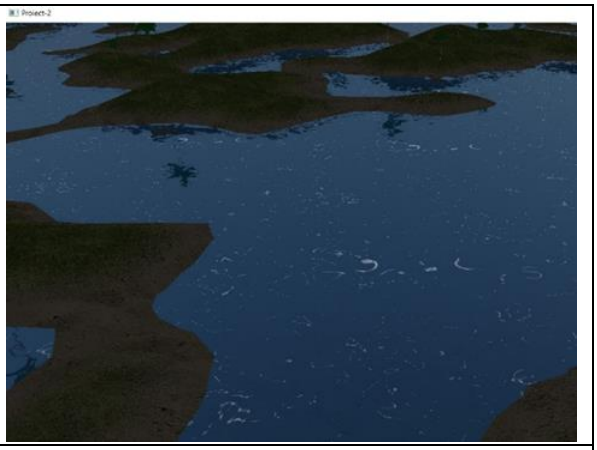
Doua componente interesante din modelul de iluminare: componenta *speculara* si componenta *difuza*.

- Componenta *speculara* este realizata cu *normal mapping* (o textura care indica devierea normalei de la verticala (normala la planul apei), de preferat sa fie in corespondenta cu *dudv map*, pentru ca sclipirile de pe apa sa fie sincronizate cu valurile (deformarea imaginii conform *dudv mapping*)) – mecanismul este foarte asemanator cu *dudv mapping*

```
// normal mapping and lighting
vec4 normalMapColor = texture(normalMap, distortedTexCoords);
vec3 normal = vec3(normalMapColor.r * 2.0 - 1.0, normalMapColor.b,
normalMapColor.g * 2.0 - 1.0);

normal = normalize(normal);

// specular
vec3 reflectedLight = reflect(normalize(fromLightVector), normal);
float specular = max(dot(reflectedLight, viewVector), 0.0);
specular = pow(specular, shineDamper);
vec3 specularHighlights = lightColor * specular * reflectivity;
```

Efect rezultat:

- Componenta *difuza* – implementata cu normala “naturala” *(0.0, 1.0, 0.0)* a apei, nu cu normalele din *normal map*; am ales sa procedam in acest fel deoarece “cat de intunecata sau de transparenta este apa” nu tine de deformatiile de la suprafata (valurile simulate), ci de cantitatea de lumina din momentul respectiv al zilei. Astfel, cand directia luminii este perpendiculara pe planul apei, apa va fi foarte transparenta, iar noaptea, apa va avea o culoare albastru inchis. Astfel, apa respecta ciclul zi-noapte din scena:

	
Zi	Noapte

e) Efecte de adancime si corectare defecte

Avem nevoie sa cunoastem adancimea apei din urmatorul motiv: distorsionarea texturii de pe apa (cu *dudv mapping*) este destul de mare pentru a crea un efect realist de valuri; prin urmare, in apropierea malurilor, pe suprafata apei se va aduce culoare de pe uscat (conform offset-ului dat de *dudv map*) :

Efectul este neplacut, mai ales deoarece culoarea adusa de pe mal este de fapt o culoare “din interiorul insulei”, pentru ca folosim decuparea tuturor elementelor aflate deasupra planului apei, deci se va textura dintr-o zona in care nu ajunge lunima, rezultand o culoare aproape neagra.

Vrem sa evitam acest efect nedorit, prin micsorarea valurilor langa tarm (adica micsoram distorsionarea cu un factor care depinde de adancime):

Calculul adancimii:
Valoarea adancimii este stocata in textura de adancime, dar nu este liniara cu distanta din spatiul de vizualizare. Conform inmultirii matricei de proiectie cu varful de coordonate (x, y, z, 1.0), avem rezultatul:

$$M_{proiectie} \cdot (x,\ y,\ z,\ 1.0) = (x_n,\ y_n,\ z_n,\ w)$$

Ne intereseaza valoarea pentru componenta *z*, dar mai intai pentru *w* (pentru a aplica *perspective division*).
Calculam *w*:

$$\langle (0.0,\ 0.0,\ -1.0,\ 0.0),\ (x,\ y,\ z,\ w) \rangle = -z$$

Calculam inmultirea liniei corespunzatoare lui *z* din matrice:

$$\left\langle \left(0.0,\ 0.0,\ -\frac{d_{far}+d_{near}}{d_{far}-d_{near}},\ -\frac{2\,d_{far}d_{near}}{d_{far}-d_{near}}\right),\ (x,\ y,\ z,\ 1.0) \right\rangle = -z\,\frac{d_{far}+d_{near}}{d_{far}-d_{near}} - \frac{2\,d_{far}d_{near}}{d_{far}-d_{near}}$$

Dar z_n este in coordonate normalizate [-1.0, 1.0], iar in buffer avem z_b in [0.0, 1.0].
Scriem z_n in functie de z_b: $z_n = 2\,z_b - 1$
si obtinem relatia, cuprinzand si *perspective division*:

$$2\,z_b - 1 = \frac{-z\frac{d_{far}+d_{near}}{d_{far}-d_{near}} - \frac{2\,d_{far}\,d_{near}}{d_{far}-d_{near}}}{-z}$$

De aici scoatem *z*:

$$z = -\frac{2\,d_{far}\,d_{near}}{d_{far}+d_{near}-(d_{far}-d_{near})(2\,z_b-1)}$$

, formula pe care o folosim in fragment shader:

```
float depth = texture(depthMap, refractTexCoords).r;
float floorDistance = 2.0 * near * far / (far + near - (2.0 * depth -
1.0) * (far - near));

depth = gl_FragCoord.z;
float waterDistance = 2.0 * near * far / (far + near - (2.0 * depth -
1.0) * (far - near));

float waterDepth = floorDistance - waterDistance;
```

Calculam aceasta distanta de 2 ori, o data pentru distanta de la camera la harta si o data de la camera la suprafata apei, iar prin scadere obtinem adancimea apei.
Folosim `waterDepth` pentru a atenua valurile langa mal:

```
vec2 totalDistortion = (texture(dudvMap, distortedTexCoords).rg * 2.0
- 1.0) * waveStrength * clamp(pow(waterDepth, 64) / 40.0, 0.0, 1.0);
```

Contributii individuale:

Capatina Razvan Nicolae:

- Perlin Noise 2D
- Impartirea hartii in chunk-uri, multithreading pentru generarea terenului
- Gasire de texturi pentru suprafata
- Incarcarea obj-urilor pentru palmier si barca
- Implementarea modelului Phong pentru iluminarea chunk-urilor (lumina directionala, ciclu zi-noapte)
- Texture Manager

Mihalache Sebastian Stefan:

- Skybox
- Iarba, folosind curbele Bezier, Compute Shader si Tessellation Shaders
- Integrarea bibliotecii Tiny Obj Loader pentru incarcare de obiecte
- Camera pentru navigarea scenei
- Sistemul de input al camerei, incluzand functionalitati de miscare, control al privirii si zoom

Luculescu Teodor:

- Apa, reflexie, refractie, valuri simulate, normal mapping pentru a reda efecte realiste
- Implementare iluminare (ciclu zi-noapte) pentru apa

Petrovici Ricardo:

- Shadow Mapping
- Implementarea clasei Model pentru o usoara adaugare a modelelor obj

Anexa:

Resurse folosite:

lect13-2d-perlin.pdf (implementare Perlin Noise 2D)

Proceedings Template - WORD (imbunatatiri Perlin Noise 2D)

LearnOpenGL - Shadow Mapping (implementare Shadow Mapping)

https://www.cg.tuwien.ac.at/research/publications/2017/JAHRMANN-2017-RRTG/JAHRMANN-2017-RRTG-draft.pdf (implementare iarba)

https://github.com/LesleyLai/GLGrassRenderer (implementare iarba)

https://learnopengl.com/Getting-started/Camera (implementare camera)

https://github.com/tinyobjloader/tinyobjloader (Tiny Obj Loader, biblioteca pentru incarcarea modelelor)

https://github.com/nothings/stb/blob/master/stb_image.h (biblioteca pentru incarcarea imaginilor)

Boat Free 3D Model - .3ds .obj .dae .blend .fbx .mtl - Free3D (obj barca)

https://www.youtube.com/watch?v=HusvGeEDU_U&list=PLRlWtICgwaX23jiqVByUs0bqhnalNTNZh (tutorial apa, detaliat)

https://learnopengl.com/Getting-started/Coordinate-Systems (pentru apa, mai ales *perspective division*)

http://web.archive.org/web/20130416194336/http://olivers.posterous.com/linear-depth-in-glsl-for-real (pentru apa, pentru calculul adancimii)