Arthur Chateauneuf **Master Informatique, Imagine**

TP1&2 Cartes d'Altitude

Rendu de terrain utilisant la carte d'ombre dans le moteur Vulpine

Notes

Le code de ce TP est disponible sur son répertoire GitHub , il est conseillé de pull ce dernier avec les options « --recursive –-depth 1 » afin d'obtenir également le code des sous modules.

La liste des dépendances peut être trouvée sur la page du moteur Vulpine.

Génération de la grille de départ

Afin d'obtenir une grille de départ, nous chargeons simplement un modèle créer sur blender. Nous lui appliquons ensuite l'ensemble des textures qui seront utilisées.

```
ModelRef floor = newModel(GameGlobals::PBRHeightMap);
floor->setVao(readOBJ("ressources/models/ground/model.obj"));
floor->state.scaleScalar(1e2f).setPosition(vec3(0, 30, 0));

Texture2D HeightMap = Texture2D().loadFromFileHDR("ressources/maps/RuggedTerrain.hdr")
    .setFormat(GL_RGB)
    .setInternalFormat(GL_RGB32F)
    .setPixelType(GL_FLOAT)
    .setWrapMode(GL_REPEAT)
    .setFilter(GL_LINEAR)
    .generate();
floor->setMap(HeightMap, 2);

const std::string terrainTextures[] = {
    "snowdrift1_ue", "limestone5-bl", "leafy-grass2-bl", "forest-floor-bl-1"};

int base = 4;
int i = 0;
for(auto str : terrainTextures)
{
    floor->setMap(Texture2D().loadFromFileKTX(
        ("ressources/models/terrain/"+str+"CE.ktx2").c_str()),
        base + i);
    floor->setMap(Texture2D().loadFromFileKTX(
        ("ressources/models/terrain/"+str+"NRM.ktx2").c_str()),
        base + 4 + i);
    i++;
}
```

Matériaux et tesselation

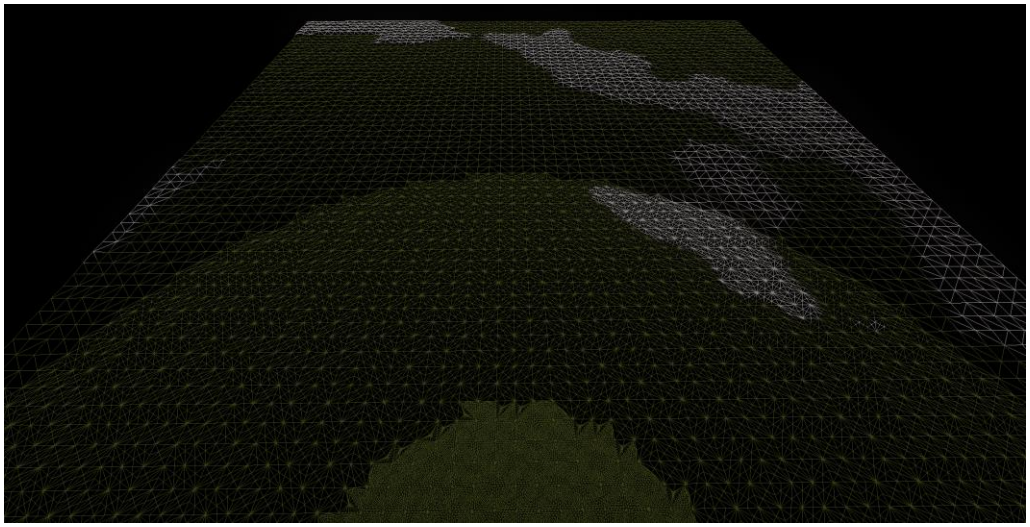
L'élément GameGlobals::PBRHeightMap est un objet de type MeshMaterial que nous chargeons à partir de 4 fichiers sources :

- Le fragment shader : build/shader/foward/PBR.frag
- Le vertex shader : build/shader/special/lod.vert
- Le tessellation control shader : build/shader/special/lod.vert
- Le tessellation evaluation shader : build/shader/special/lod.vert

Afin de rendre notre géométrie avec un niveau de détail adaptatif, nous décidons d'utiliser un tessellation shader. Ce dernier est composé de 2 parties : La partie de contrôle, qui défini le

niveau de tessellation de chaque triangle, et la partie d'évaluation, qui appliquera les divers effets que nous souhaitons implémenter comme la carte de hauteur.

Le shader de contrôle subdivisera ainsi les triangles de la mesh en fonction de la distance à la caméra. Nous ne considérons que la distance sur les axes x et z dans le cadre d'une carte de hauteur. Voici le résultat de ce passage de shader :

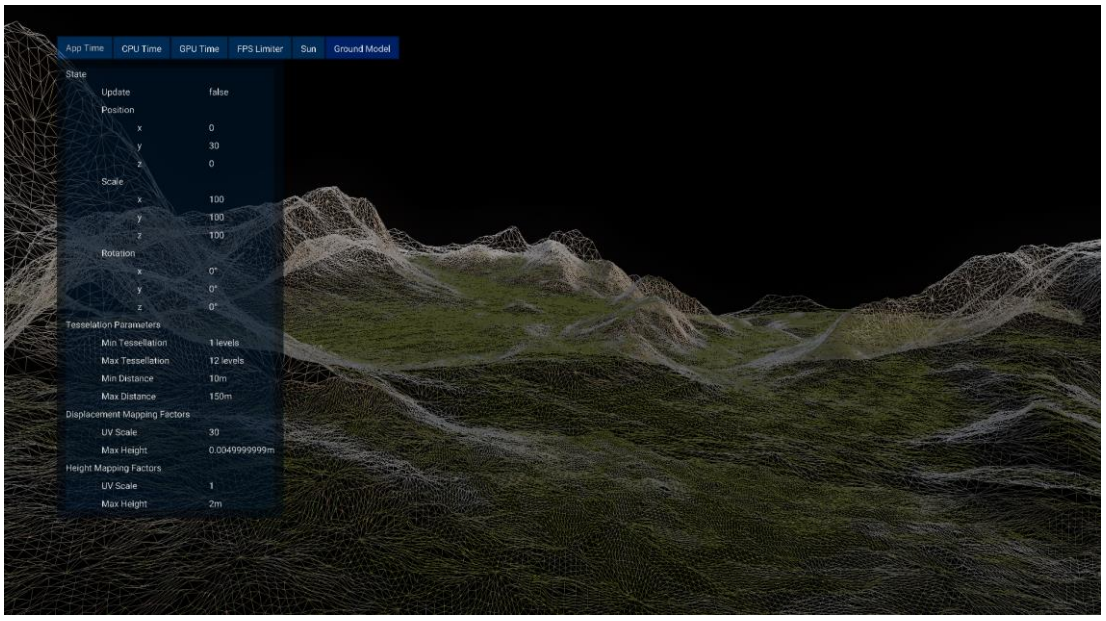


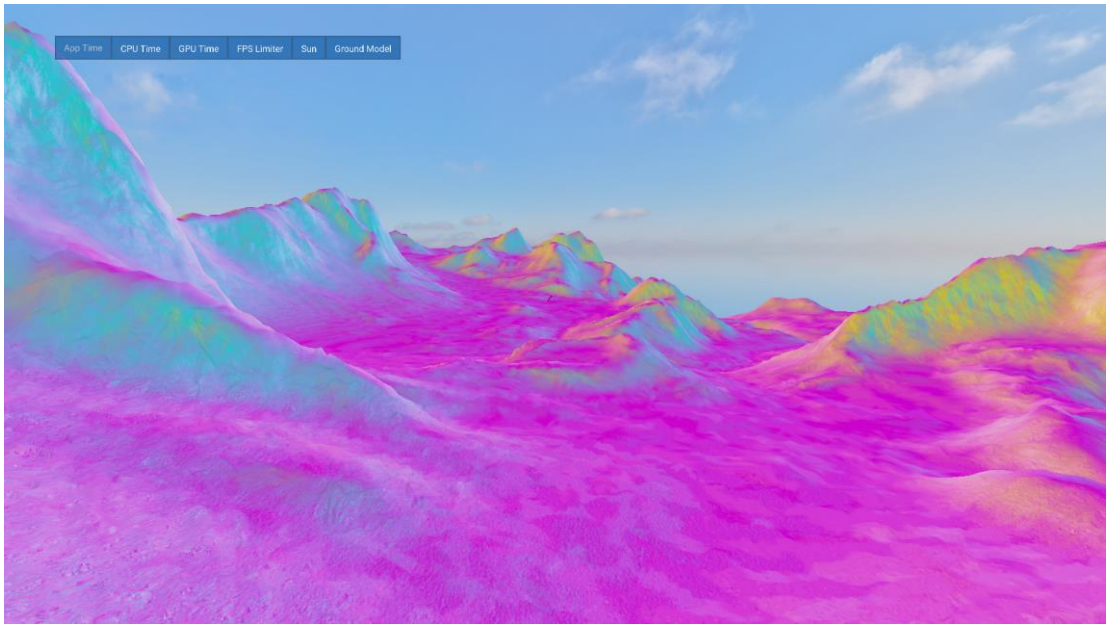
L'étape de shader suivante est l'évaluation de la tesselation. Nous y appliquons l'effet de la carte de hauteur ainsi qu'une carte de déplacement pour la texture du matériau. Nous ne nous concentrons ici que sur la carte de hauteur :

```
    if(lodHeightDispFactors.w > zero)
    {
        vec2 hUv = uv*lodHeightDispFactors.z;
        float h = texture(bHeight, clamp(hUv, 0.001, 0.999)).r;
        const float bias = 0.01*lodHeightDispFactors.z;
        float h1 = texture(bHeight, clamp(hUv+vec2(bias, 0), 0.001, 0.999)).r;
        float h2 = texture(bHeight, clamp(hUv-vec2(bias, 0), 0.001, 0.999)).r;
        float h3 = texture(bHeight, clamp(hUv+vec2(0, bias), 0.001, 0.999)).r;
        float h4 = texture(bHeight, clamp(hUv-vec2(0, bias), 0.001, 0.999)).r;

        float dist = bias/lodHeightDispFactors.w;
        vec3 nP1 = normal*h;
        vec3 nP2 = normal*h3 + vec3(0.0, 0.0, dist);
        vec3 nP3 = normal*h1 + vec3(dist, 0.0, 0.0);
        vec3 nP4 = normal*h4 - vec3(0.0, 0.0, dist);
        vec3 nP5 = normal*h2 - vec3(dist, 0.0, 0.0);

        vec3 n1 = normalize(cross(nP2-nP1, nP3-nP1));
        vec3 n2 = normalize(cross(nP4-nP1, nP5-nP1));
        vec3 n3 = -normalize(cross(nP2-nP1, nP5-nP1));
        vec3 n4 = -normalize(cross(nP4-nP1, nP3-nP1));
        normal = normalize(n1+n2+n3+n4);

    #ifdef USING_TERRAIN_RENDERING
```

```
        terrainHeight = h;
        terrainUv = hUv;
    #endif
        const float hfact = 2.0;
        h = (hfact*h+h1+h2+h3+h4)/(hfact+4.0);
        positionInModel += normalG*(h-0.5)*lodHeightDispFactors.w;
    }
```

Nous calculons la hauteur et la normale à partir de 4 échantillons de hauteur. Les éléments lodHeightFactor.z et w sont des entrées uniformes représentants respectivement l'échelle de répétition de la carte de hauteur et l'échelle de hauteur appliquée. Ces paramètres sont modifiables dans l'onglet « Ground Model ». Voici le résultat sur notre modèle :





Normales générées

Application de textures

Pour finir, nous utilisons 4 textures pour notre objet que nous appliquons avec une opacité différente en fonction de plusieurs arguments tels que la hauteur ou l'orientation du terrain. Nous avons également mis en place en system permettant d'obtenir l'état en fonction d'une autre texture (chaque canaux RGBA représentant l'état de l'opacité pour une des 4 textures). Voici le code de notre shader utilitaire gérant les textures, qui est importé lorsque USING_TERRAIN_RENDERING est défini.

```
layout (binding = 3) uniform sampler2D bTerrainMap;
layout (binding = 4) uniform sampler2D bTerrainCE[4];
layout (binding = 8) uniform sampler2D bTerrainNRM[4];

vec4 getTerrainTexture(vec4 factors, vec2 uv, sampler2D textures[4])
{
    vec4 res = vec4(0.0);
    res = mix(res, texture(textures[2], uv), factors[2]);
    res = mix(res, texture(textures[3], uv), factors[3]);
    res = mix(res, texture(textures[1], uv), factors[1]);
    res = mix(res, texture(textures[0], uv), factors[0]);
    return res;
}

vec4 getTerrainFactorFromState(vec3 tNormal, float tH)
{
    vec4 factors = vec4(1.0);
    const float steep = abs(tNormal.y);
    factors.g = 1.0 - pow(smoothstep(0.0, 0.75, steep), 2.0);
    factors.a = 1.0 - pow(smoothstep(0.6, 1.0, steep), 3.0);
    factors.r = smoothstep(0.275 + 0.05*steep, 0.330, tH);
    factors.r = factors.r * (1.0-factors.g);
    return factors;
}
```