TP 2 Programmation 3D : Transformations

Mateusz Birembaut

October 9, 2024

Contents

("1","2","3" pour afficher les exercices, "z" "q" "s" "d" pour bouger la caméra,"a","e" tourner la chaise, "+"/"-" modifier multiplicateur de vitesse système solaire)

1 Transformations 2D

1.1 Modification du shader

Soit la matrice "u model" décrivant la transformation du mesh, le shader va appliquer à chaque point de notre mesh la transformation, la nouvelle position = A * position.

```
#version 330 core

// Input vertex data, different for all executions of this shader.
layout(location = 0) in vec3 vertices_position_modelspace;

//TODO create uniform transformations matrices Model View Projection
uniform mat4 u_model;
//uniform mat4 u_view;
//uniform mat4 u_projection;

// Values that stay constant for the whole mesh.

void main(){
        //mat4 mvp = u_projection * u_view * u_model;
        // TODO : Output position of the vertex, in clip space : MVP * position
        gl_Position = u_model * vec4(vertices_position_modelspace,1)  ;

}
```

Figure 1: Ajout de la matrice "model" dans le vertex shader

1.2 Matrices de transformation d'un mesh

(De l'exercice 1 à "suzanne", j'utiliserai des matrices créées à la main ensuite avec glm::...)

Scale

$$\mathbf{S}(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Translation

$$\mathbf{T}(t_x, t_y, t_z) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 2: Matrices scale et translation

On peut créer à la main une matrice qui combine les deux. Attention les matrices sont en column-major. (ex : transformations[3][1] = transformations[1][3] normalement, le premier indice en column-major = la colonne)

```
// 1.A
Mat4 transformations = Mat4(1.0);

transformations[3][1] = -1.0; // sur le sol
transformations[3][0] = -0.5; // un peu a gauche
transformations[0][0] = 0.5; // scale x = 0.5
transformations[1][1] = 0.5; // scale y = 0.5

glUniformMatrix4fv(loc_transformations, 1,GL_FALSE, &transformations[0][0]);

drawChaise();
```

Figure 3: Ajout de la chaise en bas à droite

Pour créer la deuxième chaise qui est dans l'autre sens, on change la scale en x pour qu'il soit négatif, on garde la même taille, mais on inverse le sens de la chaise. Attention pour voir la chaise avec sens inversé bien penser à mettre "glDisable (GL CULL FACE)".

```
//ex1.b
transformations = Mat4(1.0);

transformations[3][1] = -1.0; // sur le sol
transformations[3][0] = 0.5; // un peu a droite
transformations[0][0] = -0.5; // scale x = -0.5 pour la retourner
transformations[1][1] = 0.5; // scale y = 0.5

glUniformMatrix4fv(loc_transformations, 1,GL_FALSE, &transformations[0][0]);

drawChaise();
```

Figure 4: Ajout de la chaise en bas à gauche

$$\mathbf{R}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R}_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 5: Matrices de rotations

Pour avoir une chaise qui tourne sur elle-même au niveau de son centre de gravité, on commence par là translater pour avoir son centre de gravité à l'origine du monde (0,0,0), ensuite la rotate avec une matrice de rotation sur z. Enfin, on la replace à sa position d'origine avec la matrice de translation inversée.

```
//EX 1.3.C / D
//matrice de translation pour avoir le centre de la chaise  en 0,0,0
Mat4 translation = Mat4(1.0);
translation[3][0] = 0;
translation[3][1] = -0.5;

//matrice de rotation sur z (tourne sur elle meme au niveau du milieu de la chaise)
Mat4 mat_rotation = Mat4(1.0);
mat_rotation[0][0] = cos(glm::radians(ex1_rotation));
mat_rotation[1][0] = -sin(glm::radians(ex1_rotation));
mat_rotation[0][1] = sin(glm::radians(ex1_rotation));
mat_rotation[1][1] = cos(glm::radians(ex1_rotation));

//matrice de translation retour à la position initiale
Mat4 translation_retour = Mat4(1.0);
translation_retour[3][0] = 0;
translation_retour[3][1] = 0.5;

// on met la chaise avec son centre de gravité au centre on rotationne et on remet la chaise à sa place
transformations = translation_retour * mat_rotation * translation;

//envoie les transformations au shader
glUniformMatrix4fv(loc_transformations, 1,GL_FALSE, &transformations[0][0]);
//dessine la chaise
drawChaise();
```

Figure 6: Ajout de la chaise en bas à gauche

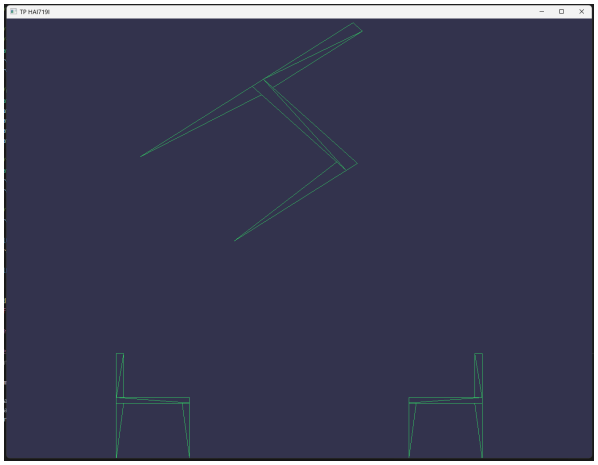

Figure 7: Résultat, "a" et "e" pour faire tourner la chaise

1.3 Calcul d'une rotation

On veut calculer et appliquez une rotation de telle sorte que l'axe vertical du personnage (0,1,0) soit aligné avec le vecteur (1,1,1) du repère monde. On utilise la formule du produit scalaire pour retrouver l'angle. Ensuite, pour faire la rotation, on utilise l'axe donné par le produit vectoriel entre ces deux vecteurs.

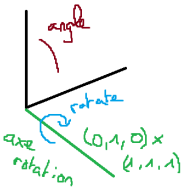

Figure 8: Croquis pour voir ce qui est a faire.

```
// pour calculer l'angle : on utilise la formule du produit scalaire pour calculer l'angle entre deux vecteurs
// u . v / norme(u) norme(v) = cos(theta)
float produit_saclaire = glm::dot(glm::vec3(0., 1., 0.), glm::vec3(1., 1., 1.));
float norme_u = glm::length(glm::vec3(0., 1., 0.));
float norme_v = glm::length(glm::vec3(1., 1., 1.));

float angle = acos(produit_saclaire / (norme_u * norme_v));

glm::vec3 axe = glm::normalize(glm::cross(glm::vec3(0., 1., 0.), glm::vec3(1., 1., 1.)));

transformations =  glm::rotate(Mat4(1.0), angle, axe);

glUniformMatrix4fv(loc_transformations, 1,GL_FALSE, &transformations[0][0]);
```

Figure 9: Calcul de la rotation

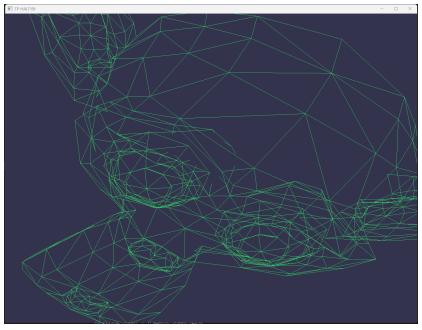

Figure 10: Résultat

2 Passage en 3D

2.1 Création des matrices View et Projection

```
glUseProgram(programID);

// View matrix : camera/view transformation lookat() utiliser camera_position camera_target camera_up
ViewMatrix = glm::lookAt(camera_position, camera_target, camera_up);

// Projection matrix : 45 Field of View, 4:3 ratio, display range : 0.1 unit <-> 500 units
ProjectionMatrix = glm::perspective(45.0f, 4.f / 3.f, 0.1f, 500.0f);

//recup les loc des variables dans le shader
GLint loc_transformations = glGetUniformLocation(programID, "u_model");
GLint loc_ViewMatrix = glGetUniformLocation(programID, "u_view");
GLint lov_ProjectionMatrix = glGetUniformLocation(programID, "u_projection");

// on envoie les matrices view et projection au shader
glUniformMatrix4fv(loc_ViewMatrix, 1, GL_FALSE, &ViewMatrix[0][0]);
glUniformMatrix4fv(lov_ProjectionMatrix, 1, GL_FALSE, &ProjectionMatrix[0][0]);
```

Figure 11: Création des matrices Projection et View

2.2 Modification des shaders

```
//TODO create uniform transformations matrices Model View Projection
uniform mat4 u_model;
uniform mat4 u_view;
uniform mat4 u_projection;

// Values that stay constant for the whole mesh.

void main(){
        mat4 mvp = u_projection * u_view * u_model;
        // TODO : Output position of the vertex, in clip space : MVP * position
        gl_Position = mvp * vec4(vertices_position_modelspace,1) ;

}
```

Figure 12: VertexShader mis à jour pour prendre en compte la caméra et transformations

2.3 Vue de 3/4

Pour créer une vue 3/4 de notre singe, on déplace la caméra vers -1.5, 0, 3.0 cela nous donnera notre vue de 3/4.

```
// camera
glm::vec3 camera_position   = glm::vec3(1.5f, 0.0f,  3.0f);
glm::vec3 camera_target = glm::vec3(0.0f, 0.0f, -1.0f);
glm::vec3 camera_up     = glm::vec3(0.0f, 1.0f,  0.0f);
```

Figure 13: Position de la caméra pour une vue de 3/4

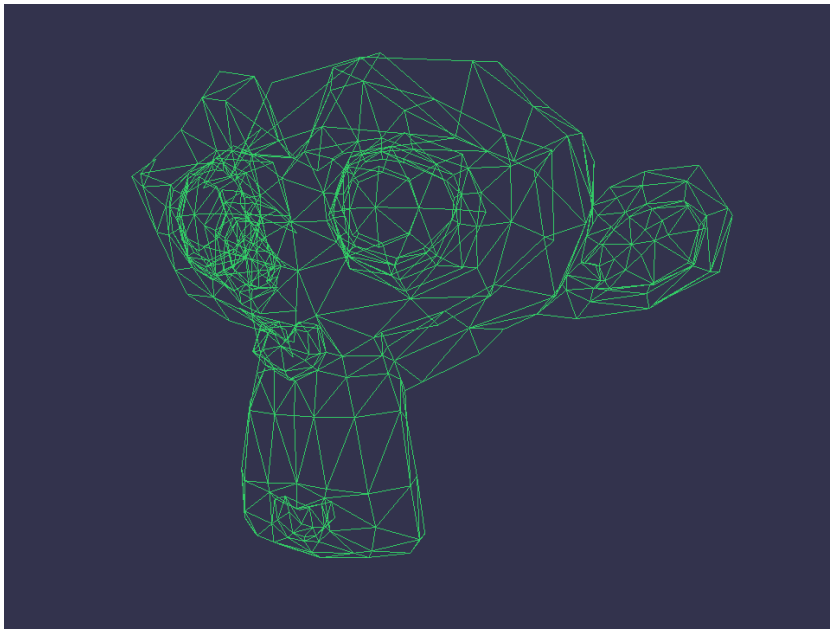

Figure 14: Suzanne vue de 3/4

2.4 Contrôle caméra : déplacement latéral

Pour le déplacement latéral, comme on a caméra up, on a juste besoin du vecteur entre la caméra target et caméra position, on calcule le produit vectoriel pour avoir le vecteur perpendiculaire aux deux. Enfin, on ajoute ou soustrait à notre position le vecteur camera droite. On met à jour le target aussi, sinon on ferait des tours autour de la target.

```
glm::vec3 camera_direction = glm::normalize(camera_target - camera_position);
glm::vec3 camera_right = glm::normalize(glm::cross(camera_direction, camera_up));
```

Figure 15: Calcul des vecteurs

```
case 'q':
    camera_position -= cameraSpeed * camera_right;
    camera_target -= cameraSpeed * camera_right;
    break;
case 'd':
    camera_position += cameraSpeed * camera_right;
    camera_target += cameraSpeed * camera_right;
    break;
```

Figure 16: Ajout des positions

3 Système solaire animé

Pour faciliter la création d'un système stellaire avec plus de planètes, j'ai créé deux struct afin de stocker l'étoile, les planètes et les lunes d'un système et permettre l'affichage des planètes.

```
struct CorpsCeleste{
    float vitesseRotationOrbitale;    // vitesse de rotation autour du soleil
    float vitesseRotation;          // vitesse de rotation sur lui meme
    float taille;                // taille du corps
    float angleRotation;         // angle de rotation sur lui meme
    float angleRotationOrbital;      // angle de rotation autour du soleil
    float inclinaisonAxiale;  // angle d'inclinaison de l'axe de rotation
    float distanceOrbite;   // distance par rapport au soleil
    float degresOrbite;     // degres de l'orbite (0 = orbite qui reste dans le plan xz)
    std::vector<CorpsCeleste> lunes; // liste des lunes
```

Figure 17: Attributs de la structure "CorpsCeleste"

```
float getAngleRotation() {
    angleRotation = fmod(angleRotation + vitesseRotation * multiplicateur_vitesse, 360.0f);
    return angleRotation;
}

float getAngleRotationOrbitale() {
    angleRotationOrbital = fmod(angleRotationOrbital + vitesseRotationOrbitale * multiplicateur_vitesse, 360);
    return angleRotationOrbital;
}

void addLune(const CorpsCeleste& lune) {
    lunes.push_back(lune);
}
```

Figure 18: Fonctions de la structure "CorpsCeleste"

```
struct SystemeStellaire{
    std::vector<CorpsCeleste> planetes; // planetes du systeme
    CorpsCeleste etoile; // l'etoile du systeme

    SystemeStellaire(const CorpsCeleste& etoile) :
        etoile(etoile), planetes() {
    }

    void addPlanete(const CorpsCeleste& planete) {
        planetes.push_back(planete);
    }
```

Figure 19: Attributs de la structure "SystemeStellaire"

```
void drawSystem() {
    glUseProgram(programID);

    // View matrix : camera/view transformation lookat() utiliser camera_position camera_target camera_up
    ViewMatrix = glm::lookAt(camera_position, camera_target, camera_up);

    // Projection matrix : 45 Field of View, 4:3 ratio, display range : 0.1 unit <-> 500 units
    ProjectionMatrix = glm::perspective(45.0f, 4.f / 3.f, 0.1f, 500.0f);

    //recup les loc des variables dans le shader
    GLint loc_transformations = glGetUniformLocation(programID, "u_model");
    GLint loc_ViewMatrix = glGetUniformLocation(programID, "u_view");
    GLint lov_ProjectionMatrix = glGetUniformLocation(programID, "u_projection");

    // on envoie les matrices view et projection au shader
    glUniformMatrix4fv(loc_ViewMatrix, 1, GL_FALSE, &ViewMatrix[0][0]);
    glUniformMatrix4fv(lov_ProjectionMatrix, 1, GL_FALSE, &ProjectionMatrix[0][0]);

    //matrice model pour l'étoile
    Mat4 model = Mat4(1.0);

    //rotation sur elle meme et angle si present
    model = glm::scale(model, glm::vec3(etoile.taille, etoile.taille, etoile.taille)); // changement taille
    model = glm::rotate(model, glm::radians(etoile.getAngleRotation()), glm::vec3(0, 1, 0)); // rotation sur y

    glUniformMatrix4fv(loc_transformations, 1, GL_FALSE, &model[0][0]);
    DrawSphere(); // dessine étoile
```

Figure 20: DrawSystem Part 1, dessin de l'étoile

Comme la position des lunes est lié à la position de sa planète, "modelLune" est initialisé à "model", on fait les transformations nécessaires, puis on dessine la lune, après avoir dessiné toutes les lunes, on scale la planete, on l'incline et on la fait tourner. On fait les transformations comme ça pour pas que la vitesse de rotation, le scale et l'inclinaison de la planète influe sur la lune.

```
// pour chaque planete
for (CorpsCeleste& planete : planetes) {

    model = Mat4(1.0);//matrice model pour la planete actuelle

    model = glm::rotate(model, glm::radians(planete.degresOrbite), glm::vec3(0, 0, 1)); // degres entre plan xz et orbite planete
    model = glm::rotate(model, glm::radians(planete.getAngleRotationOrbitale()), glm::vec3(0, 1, 0)); // rotation autour du soleil
    model = glm::translate(model, glm::vec3(planete.distanceOrbite, 0, 0)); // translation selon distance avec le soleil

    for (CorpsCeleste& lune : planete.lunes) {
        Mat4 modelLune = model;

        modelLune = glm::rotate(modelLune, glm::radians(lune.degresOrbite), glm::vec3(0, 0, 1)); // degres entre orbit lune et orbite planete
        modelLune = glm::rotate(modelLune, glm::radians(lune.getAngleRotationOrbitale()), glm::vec3(0, 1, 0)); // orbite
        modelLune = glm::translate(modelLune, glm::vec3(  lune.distanceOrbite, 0, 0));  // déplacement selon distance avec la planete
        modelLune = glm::scale(modelLune, glm::vec3(lune.taille, lune.taille, lune.taille));  // changement taille
        modelLune = glm::rotate(modelLune, glm::radians(lune.getAngleRotation()), glm::vec3(0, 1 , 0)); // rotation sur elle meme

        glUniformMatrix4fv(loc_transformations, 1, GL_FALSE, &modelLune[0][0]); // envoie transfo pour placer la lune au shader
        DrawSphere(); // draw lune
    }

    //changement de la taille (apres la translation pour garder l'unité de distance cohérente entre toutes les planetes)
    model = glm::scale(model, glm::vec3(planete.taille, planete.taille, planete.taille));
    model = glm::rotate(model, glm::radians(planete.inclinaisonAxiale), glm::vec3(1, 0, 0));  // inclinaison axiale planete
    model = glm::rotate(model, glm::radians(planete.getAngleRotation()), glm::vec3(0, 1 , 0)); // rotation sur elle meme planete

    glUniformMatrix4fv(loc_transformations, 1, GL_FALSE, &model[0][0]);
    DrawSphere(); //planete

}
```

Figure 21: DrawSystem Part 2, dessin des planètes et des lunes

3.1 Résultats :

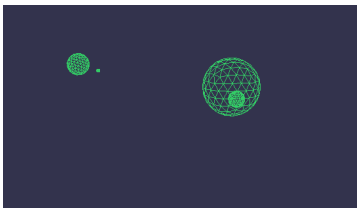

Figure 22: Soleil, terre + lune et une autre planète

```
//CorpsCeleste(float taille, float vitesseRotationOrbitale, float vitesseRotation, float inclinaisonAxiale, float distanceOrbite, float degresOrbite) :

CorpsCeleste soleil = CorpsCeleste(5.f, 0.f, 0.000154f, 0.f, 0.0f, 0.0f); // Soleil, taille maximale

CorpsCeleste mercure = CorpsCeleste(1.f, 0.0000017f, 0.000017f, 0.03f, 4.f, 7.0f);    // Mercure
CorpsCeleste venus   = CorpsCeleste(1.8f, 0.0000011f, 0.000004f, 177.36f, 10.f, 3.39f);  // Vénus
CorpsCeleste terre   = CorpsCeleste(2.f, 0.0000114f, 0.0041667f, 23.44f, 16.f, 0.0f);    // Terre
CorpsCeleste lune    = CorpsCeleste(0.5f, 0.0000152f, 0.0000152f, 6.68f, 2.5f, 5.14f);   // Lune
CorpsCeleste mars    = CorpsCeleste(1.2f, 0.0000086f, 0.003521f, 25.19f, 20.f, 5.85f);   // Mars
CorpsCeleste jupiter = CorpsCeleste(4.f, 0.0000045f, 0.004545f, 3.13f, 26.f, 1.31f);  // Jupiter
CorpsCeleste saturne = CorpsCeleste(3.5f, 0.0000029f, 0.003684f, 26.73f, 32.f, 3.49f); // Saturne
CorpsCeleste uranus  = CorpsCeleste(2.5f, 0.0000015f, 0.001479f, 97.77f, 38.f, 0.77f); // Uranus
CorpsCeleste neptune = CorpsCeleste(2.4f, 0.0000011f, 0.001588f, 28.32f, 44.f, 5.77f);  // Neptune

// Mars
CorpsCeleste luneMarsPhobos = CorpsCeleste(0.22f*4, 0.0000011f, 0.0000011f, 9.4f, 5.0f, 0.3f); // Phobos
CorpsCeleste luneMarsDeimos = CorpsCeleste(0.12f*4, 0.000006f, 0.0000006f, 15.0f, 3.0f, 1.3f); // Deimos

// Jupiter
CorpsCeleste luneJupiterIo = CorpsCeleste(0.18f*4, 0.000092f, 0.0000892f, 10.0f, 1.8f*3, 1.8f); // Io
CorpsCeleste luneJupiterEurope = CorpsCeleste(0.16f*4, 0.0000489f, 0.0000489f, 18.5f, 3.5f*2, 3.5f); // Europe
CorpsCeleste luneJupiterGanymede = CorpsCeleste(0.26f*4, 0.0000148f, 0.0000148f, 7.0f, 0.9f*6, 0.9f); // Ganymède
CorpsCeleste luneJupiterCallisto = CorpsCeleste(0.24f*4, 0.0000158f, 0.0000158f, 16.7f, 1.0f*5, 1.0f); // Callisto

// Saturne
CorpsCeleste luneSaturneTitan = CorpsCeleste(0.23f*4, 0.0000134f, 0.0000134f, 10.5f, 5.0f, 5.0f); // Titan
CorpsCeleste luneSaturneRhea = CorpsCeleste(0.18f*4, 0.000041f, 0.000041f, 15.3f, 4.0f, 4.0f); // Rhea
CorpsCeleste luneSaturneIapetus = CorpsCeleste(0.14f*4, 0.000017f, 0.0000017f, 20.0f, 6.0f, 6.0f); // Iapetus
CorpsCeleste luneSaturneDione = CorpsCeleste(0.12f*4, 0.000025f, 0.0000025f, 25.0f, 7.0f, 7.0f); // Dione

// Uranus
CorpsCeleste luneUranusTitania = CorpsCeleste(0.18f*4, 0.000008f, 0.000008f, 20.0f, 8.0f, 8.0f); // Titania
CorpsCeleste luneUranusOberon = CorpsCeleste(0.16f*4, 0.000007f, 0.000007f, 25.0f, 9.0f, 9.0f); // Oberon
CorpsCeleste luneUranusAriel = CorpsCeleste(0.14f*4, 0.000005f, 0.000005f, 30.0f, 10.0f, 10.0f); // Ariel
```

Figure 23: Listes des planètes avec leurs attributs

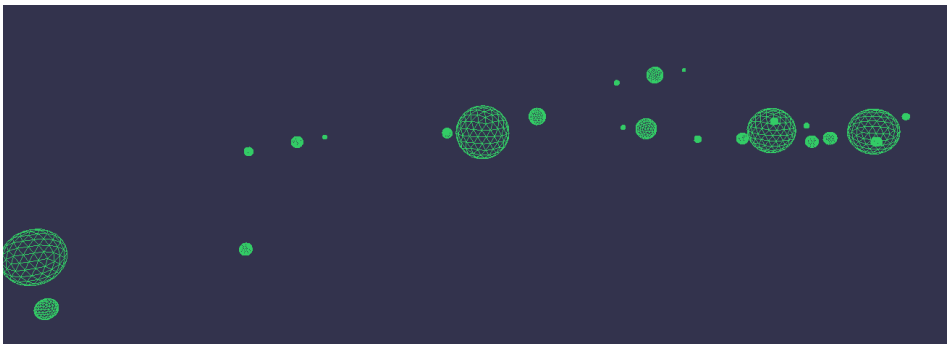

Figure 24: Système solaire avec quelques lunes