Moteur de jeux

Compte-rendu TP6

Caméras

Louis Jean
Master 1 IMAGINE
Université de Montpellier
N° étudiant : 21914083

11 avril 2024

Table des matières

1 Introduction

Dans le cadre du cours de moteur de jeux, ce travail pratique nous a permis de nous immerger dans la complexité et l'utilité des systèmes de caméras en 3D, en utilisant `ImGui` couplé à `OpenGL` pour manipuler et observer les effets de nos interactions en temps réel. Le projet consistait à développer diverses fonctionnalités pour contrôler une caméra dans un environnement 3D, en explorant des techniques telles que la modification de paramètres en direct, les transitions douces entre les états de caméra, et la simulation de phénomènes comme le tremblement de la caméra. Cette expérience pratique avait pour objectif de nous donner une compréhension approfondie des mécanismes derrière la gestion des caméras dans les jeux vidéo.

N.B : Pour lancer le projet, veuillez exécuter les commandes suivantes.

```
cd out/build
cmake ../..
make -j 24
sh launch-TP.sh
```

2 ImGui

Dans un premier temps, nous avons pris en main `ImGui`, en permettant le paramétrage en temps réel de plusieurs aspects de la caméra.

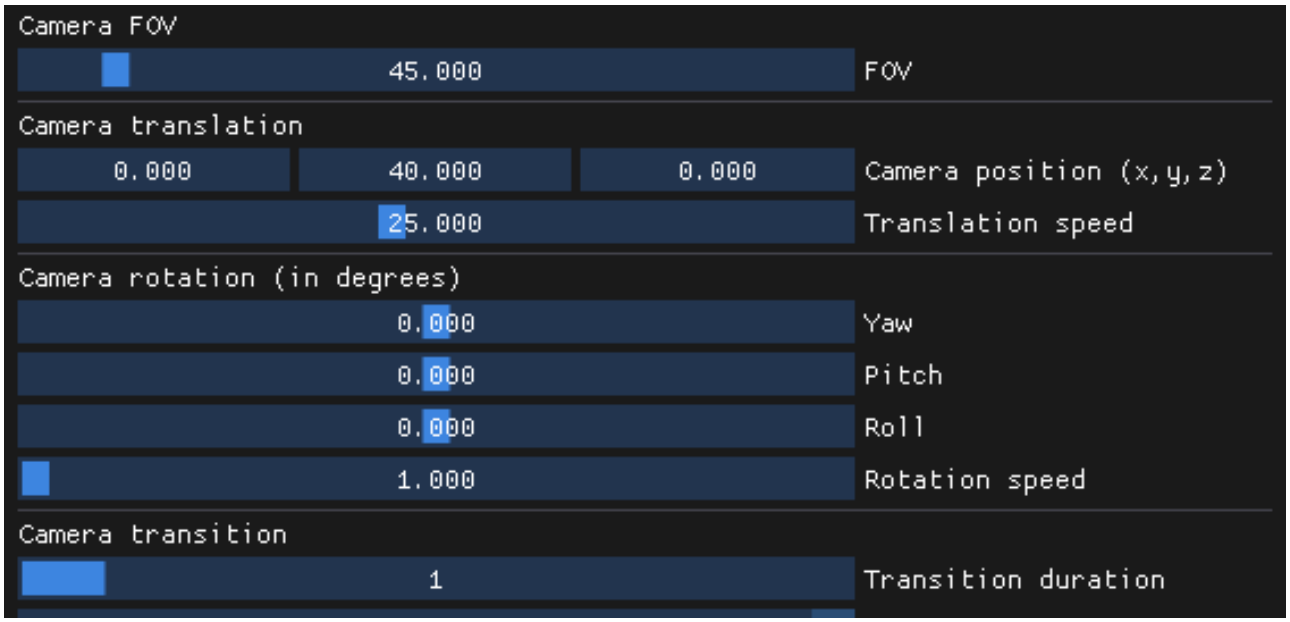


Figure 1: Première interface pour contrôler les paramètres de la caméra

Pour la position de la caméra, j'ai utilisé des `DragFloat` car théoriquement la position n'a pas de limite dans l'espace. Pour les rotations, j'ai utilisé des `SliderFloat`, en limitant le yaw à $[-180, 180]$, et le pitch et le roll à $[-90, 90]$. Pour permettre la manipulation par l'utilisateur de ces valeurs en degrés, j'ai créé une nouvelle variable `m_eulerAngleInDegrees` dans `Camera.hpp`, que j'ai ensuite convertie en radians juste avant de calculer la rotation à l'aide des quaternions dans la fonction `update`.

J'ai aussi créé un bouton "`Reset values`" qui réinitialise tous les paramètres de l'application, en appelant simplement la fonction `init`.

3 Contrôle de la caméra

Pour contrôler la caméra, il était proposé de permettre le choix entre deux modes distincts. Il est possible d'alterner entre ces deux modes en appuyant sur la touche `TAB`. Le premier mode permet le contrôle de la position de la caméra à l'aide des touches `Z,Q,S,D` et de sa rotation grâce à la souris. Le deuxième mode permet le contrôle de la position de la caméra de la même manière, mais le contrôle de la rotation se fait avec les flèches directionnelles. **La grande différence entre ces deux modes réside dans le fait que dans le premier mode, on se déplace en suivant le front de la caméra (il y a donc une composante de hauteur), alors que dans le second mode, on se déplace toujours de manière parallèle au plan (mode drone, $y = 0$).**

3.1 Distinction des deux modes de la caméra

Dans un premier temps, j'ai dû trouver un moyen de différencier ces modes au sein du code. Pour cela, j'ai créé une énumération.

```
enum CameraMode {
  FIRST_MODE = 1,
  SECOND_MODE,
  MODE_COUNT
};
```

Cela m'a permis de pouvoir identifier le mode courant et donc d'en changer facilement. De plus, cette énumération rend le code flexible car elle permet sans soucis d'ajouter des modes sans modifier trop de code. **N.B :** Veuillez noter la présence de `MODE_COUNT`, qui permet de garder une trace du nombre de modes différents, pour mieux naviguer entre eux.

J'ai par la suite créé une variable `m_cameraMode` de type `CameraMode` dans `Camera.hpp`, et ai permis la navigation entre les modes comme ceci :

```
if(ImGui::IsKeyPressed(ImGuiKey_Tab)) {
    m_cameraMode = (CameraMode)(m_cameraMode + 1);
  if(m_cameraMode >= MODE_COUNT) {
    m_cameraMode = FIRST_MODE;
  }
  init(); // Reset values when changing mode
}
```

À partir de là, un simple switch sur le mode courant permet de contrôler les inputs en fonction du mode.

```
switch(m_cameraMode) {
    case FIRST_MODE:
        ...
    break;

    case SECOND_MODE:
        ...
    break;

    default:
    break;
}
```

3.2 Premier mode

Pour le premier mode, j'ai dû apprendre à gérer les inputs de la souris avec `GLFW`. Pour cela, j'ai stocké les dernières positions en x et en y du curseur au sein de la fenêtre à chaque passe de rendu. En le comparant avec les positions actuelles (récupérées grâce à `glfwGetCursorPos`), on peut en déduire la variation, et ainsi savoir de quel côté la souris s'est déplacée.

```
case FIRST_MODE:
    if(!m_showMouse) {
      double cursorXPos,cursorYPos;
      glfwGetCursorPos(_window, &cursorXPos, &cursorYPos);
      if(firstPass) {
        firstPass = false;
        lastCursorXPos = cursorXPos;
        lastCursorYPos = cursorYPos;
      }
      double xDiff = cursorXPos - lastCursorXPos;
      double yDiff = cursorYPos - lastCursorYPos;
      lastCursorXPos = cursorXPos;
      lastCursorYPos = cursorYPos;
      if(m_xAxisReversed) {
      m_eulerAngleInDegrees.y += xDiff * m_rotationSpeed * _deltaTime; //
   Reversed yaw
      }
      else {
        m_eulerAngleInDegrees.y -= xDiff * m_rotationSpeed * _deltaTime; //
   Yaw
      }
      if(m_yAxisReversed) {
        m_eulerAngleInDegrees.x -= yDiff * m_rotationSpeed * _deltaTime; //
   Reversed pitch
      }
      else {
        m_eulerAngleInDegrees.x += yDiff * m_rotationSpeed * _deltaTime; //
   Pitch
      }
      glm::vec3 CFront = glm::normalize(getCFront());
      glm::vec3 CRight = glm::normalize(getCRight());
      if(glfwGetKey(_window, GLFW_KEY_W) == GLFW_PRESS) {
        m_position += CFront * m_translationSpeed * _deltaTime;
      }
      if(glfwGetKey(_window, GLFW_KEY_S) == GLFW_PRESS) {
        m_position -= CFront * m_translationSpeed * _deltaTime;
      }
      if(glfwGetKey(_window, GLFW_KEY_A) == GLFW_PRESS) {
        m_position += CRight * m_translationSpeed * _deltaTime;
      }
      if(glfwGetKey(_window, GLFW_KEY_D) == GLFW_PRESS) {
        m_position -= CRight * m_translationSpeed * _deltaTime;
      }
    }
break;
```

Pour les rotations, rien de bien compliqué. Cela se complique pour les translations, car elles doivent se faire dans le front de la caméra pour avancer/reculer et dans le right de la caméra pour aller à gauche/droite. Heureusement, grâce à l'implémentation des quaternions déjà présente et au récapitulatif de cours fourni par M^{me} Cardin, j'ai pu écrire les fonctions `getFront` et `getRight`.

$$C.Front = q_{cam} * Front$$
$$C.Up = q_{cam} * Up$$
$$C.Right = q_{cam} * Right$$
$$q_{cam} = q_{yaw} * q_{pitch} * q_{roll}$$

Figure 2: Rappel des formules dans le récapitulatif du cours

```
glm::vec3 Camera::getCFront() const {
  return m_rotation * VEC_FRONT;
}

glm::vec3 Camera::getCRight() const {
  return m_rotation * VEC_RIGHT;
}
```

N.B : Veuillez noter que ce mode s'utilise uniquement lorsque le curseur n'est pas affiché, pour des raisons d'accessibilité. Pour activer/désactiver l'affichage du curseur, il faut appuyer sur la touche `T`.

3.3 Second mode

Le second mode, plus simple, permet de contrôler la caméra en mode drone. Dans ce mode, on suit le front de la caméra, c'est-à-dire que lorsqu'on appuie sur `Z`, on se déplace toujours en avant, à la seule différence que cette fois-ci, si la caméra subit un pitch, on ne suit pas ce pitch dans le déplacement (on reste toujours à la même hauteur).

```
case SECOND_MODE:
    {
      m_rotationSpeed = 50.0f; // Adapt rotation speed for this mode
      glm::vec3 CFront = getCFront();
      glm::vec3 CRight = getCRight();
      CFront.y = 0; // Here, since we don't want to follow CFront height, we
    define its y component to 0 (little trick so we don't have to project on
    a plane)
      CRight.y = 0;
      if(glfwGetKey(_window, GLFW_KEY_W) == GLFW_PRESS) {
         m_position += CFront * m_translationSpeed * _deltaTime;
      }
      if(glfwGetKey(_window, GLFW_KEY_S) == GLFW_PRESS) {
         m_position -= CFront * m_translationSpeed * _deltaTime;
      }
      if(glfwGetKey(_window, GLFW_KEY_A) == GLFW_PRESS) {
         m_position += CRight * m_translationSpeed * _deltaTime;
      }
      if(glfwGetKey(_window, GLFW_KEY_D) == GLFW_PRESS) {
         m_position -= CRight * m_translationSpeed * _deltaTime;
      }
        ...
    }
break;
```

N.B : Ici, je n'ai pas mis la partie du code qui effectue la rotation, car elle est similaire au premier mode, à la différence près qu'ici on utilise les touches directionnelles du clavier.

Lors du cours, j'ai noté qu'il était de bon usage de proposer des options d'accessibilité utilisateur quand on travaille avec des caméras. C'est pourquoi j'ai permis d'inverser les axes de rotation dans mon programme (cela se reflète dans mon code plus haut).

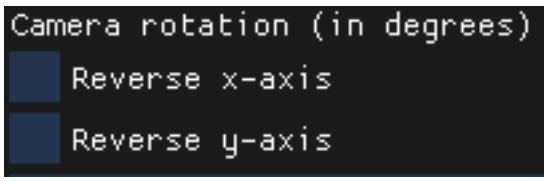


Figure 3: Boutons permettant l'inversion des axes de rotation

4 Les helpers

Dans cette partie, nous avons écrit des fonctions dans `CameraHelper.hpp` pour aider au développement de notre programme.

4.1 Projection d'un vecteur sur un plan

Pour projeter un vecteur sur un plan, on utilise le schéma suivant, encore une fois donné dans le récapitulatif du cours.

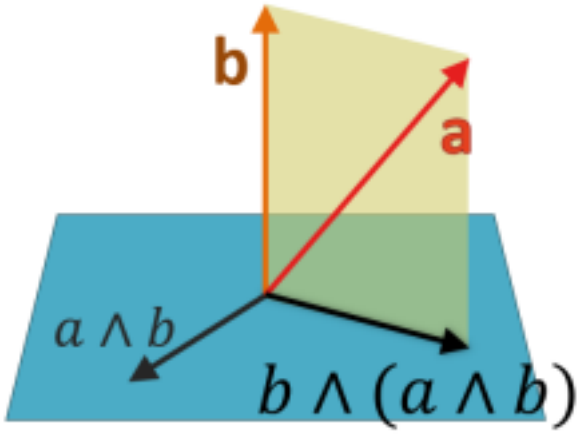


Figure 4: Projection d'un vecteur sur un plan

```
glm::vec3 Camera_Helper::projectVectorOnPlane(glm::vec3 &vector, glm::vec3 &
    planeNormal) {
  return cross(planeNormal, cross(vector, planeNormal));
}
```

4.2 Interpolation

Pour interpoler un ratio entre 0 et 1, je n'ai pas procédé comme demandé dans le sujet du TP. J'ai préféré directement interpoler un chemin entre un point A et un point B. Je détaille cela dans la section 6.

4.3 Clipper un angle

Pour clipper un angle à une valeur, j'ai trouvé deux façons, qui produisent un résultat différent. En fait, pour le yaw, on veut clipper de manière à ce que les valeurs soient toujours dans $[-180, 180]$, mais on veut tout de même permettre de passer de -180 à 180 de manière continue. Par exemple si l'on est à $-180°$ pour le yaw et que l'on soustrait 1°, on veut se retrouver à 179°. Cela assure un mouvement fluide, et permet de faire des rotations à 360° sur cet axe. Pour le pitch, ce n'est pas la même chose, car on ne veut pas se retrouver la tête à l'envers. Si on arrive à $-90°$ ou 90°, alors on ne veut pas pouvoir les dépasser de quelque manière que ce soit. Il faut donc faire un clamp.

```
// Here, the angle will be clipped to value, that means that if you are at 180 and you want to go further, you won't be able
// This is satisfying for the pitch since we don't want to be upside-down, but not for the yaw (you want to be able to do a full 360 turn)
void Camera_Helper::clampAngleToValue(float &angle, float value) {
  if(angle > value) {
    angle = value;
  }
  if(angle < -value) {
    angle = -value;
  }
}

// This is more appropriate for the yaw
void Camera_Helper::clipAngleToBounds(float &angle, float value) {
  if(angle > value) {
    angle -= 360;
  }
  if(angle < -value) {
    angle += 360;
  }
}
```

5 Contrôle affiné de la rotation

Grâce aux fonctions écrites dans la section précédente (4.3), on peut désormais clipper le yaw et clamper le pitch.

Dans la fonction `update`, on s'assure que les angles soient dans les bonnes valeurs.

```
Camera_Helper::clipAngleToBounds(m_eulerAngleInDegrees.y,180); // Clip yaw to [-180;180] (but still be able to do a 360 turn)
Camera_Helper::clampAngleToValue(m_eulerAngleInDegrees.x,90); // Clamp pitch to [-90;90]
```

6 Transition

Pour jouer une transition de caméra, j'ai choisi d'écrire une fonction `transition` dans `Camera`, et de créer les variables nécessaires dans cette classe.

Dans un premier temps, j'ai créé une nouvelle énumération, qui vise à regrouper les différents types d'interpolation pris en compte dans mon programme.

```
enum InterpolationMode {
  LINEAR,
  SMOOTHSTEP,
  SMOOTHSTEP2,
  SMOOTHSTEP3,
  SMOOTHERSTEP,
  SQUARED,
  INV_SQUARED,
  CUBED,
  INV_CUBED,
  SIN
};
```

Ensuite, j'ai mis au point les variables nécessaires à ma fonction.

```
// Transition
InterpolationMode m_interpolationMode{SMOOTHSTEP};
bool m_isTransitioning{false};
int m_transitionDuration{3};
float m_transitionProgress{0.0f};
glm::vec3 m_transitionStartPosition{m_position};
glm::vec3 m_transitionStopPosition{m_position + glm::normalize(getCFront())
    * 30.0f};
```

Enfin, j'ai écrit la fonction de transition.

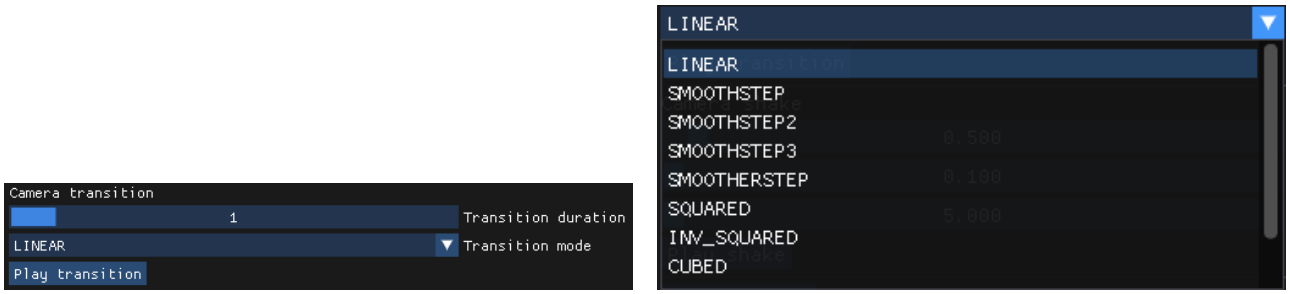
```
void Camera::transition(float delta_time) {
  if(m_transitionProgress < m_transitionDuration) { // Correct because
   glfwGetTime() returns time in seconds
    m_transitionProgress += delta_time;
    float v = m_transitionProgress / m_transitionDuration;
    switch(m_interpolationMode) {
      case LINEAR:
        m_position = glm::mix(m_transitionStartPosition,
   m_transitionStopPosition,v);
      break;

      case SMOOTHSTEP:
        v = v * v * (3 - 2 * v);
        m_position = glm::mix(m_transitionStartPosition,
   m_transitionStopPosition,v);
      break;
      case SMOOTHSTEP2:
        v = v * v * (3 - 2 * v);
        v = v * v * (3 - 2 * v);
        m_position = glm::mix(m_transitionStartPosition,
   m_transitionStopPosition,v);
      break;
      ... // Similaire pour les autres modes (voir le vrai code)
  }
  else {
    m_isTransitioning = false;
  }
}
```

Grâce à ce site, j'ai pu trouver multiples types d'interpolation et leurs implémentations.

Une fois cette fonction en main, j'ai implémenté la fonctionnalité dans `ImGui`, en permettant le contrôle de la durée de la transition, ainsi que le type d'interpolation, à choisir dans un menu déroulant.

```
ImGui::Text("Camera transition");
ImGui::SliderInt("Transition duration",&m_transitionDuration,1,10);
const char* transitions[] = {"LINEAR","SMOOTHSTEP","SMOOTHSTEP2","
    SMOOTHSTEP3","SMOOTHERSTEP","SQUARED","INV_SQUARED","CUBED","INV_CUBED","
    SIN"};
int currentTransition = (int)m_interpolationMode;
if(ImGui::BeginCombo("Transition mode", transitions[currentTransition])) {
  for(int n = 0; n < IM_ARRAYSIZE(transitions); n++) {
    bool isSelected = (currentTransition == n);
    if(ImGui::Selectable(transitions[n], isSelected)) {
      currentTransition = n;
      m_interpolationMode = (InterpolationMode)currentTransition;
    }
    if(isSelected) {
      ImGui::SetItemDefaultFocus();
    }
  }
  ImGui::EndCombo();
}
if(ImGui::Button("Play transition")) {
  m_transitionProgress = 0.0f;
  m_transitionStartPosition = m_position;
  m_transitionStopPosition = m_position + glm::normalize(getCFront()) * 30.0
    f;
  m_isTransitioning = true;
}
```



(a) Paramètres de la transition

(b) Menu déroulant pour choisir le type d'interpolation

Figure 5: Transition dans `ImGui`

N.B : Remarquez que la transition se fait toujours depuis la position actuelle de la caméra, jusqu'à 30 unités plus loin dans la direction du front de la caméra. Aussi, on ne peut pas interagir avec l'application durant toute la durée de la transition (condition `if(!m_isTransitioning)` dans `updateFreeInput`).

Pour des raisons évidentes, je ne peux pas mettre d'exemple de transitions ici, mais je vous laisse aller expérimenter par vous-même directement dans le programme !

7 Motion effect

Le but de cette partie était de rajouter un effet de shake à la caméra. Pour ce faire, j'ai créé une classe `Camera_Shake`.

```
class Camera_Shake {
    public:
        Camera &camera;
        glm::vec3 originalPosition;
        glm::vec3 originalRotation;
        float duration;
        float amplitude;
        float frequency;
        float timeElapsed;
        bool isShaking;

        Camera_Shake(Camera &camera, float duration, float amplitude, float
    frequency) :
        camera(camera), duration(duration), amplitude(amplitude), frequency(
    frequency), isShaking(false), timeElapsed(0) {}

        void update(float _deltaTime);
        void playShake();
};
```

Je ne sais pas si c'est la meilleure chose à faire, mais cette classe possède une référence vers un objet `Camera` déjà existant, permettant de le modifier.

Voici l'implémentation des fonctions de cette classe.

```
void Camera_Shake::update(float _deltaTime) {
    if(isShaking) {
        if(timeElapsed < duration) {
            float shake = amplitude * cos(2 * M_PI * frequency * timeElapsed
    );
            glm::vec3 currentPosition = camera.getPosition();
            glm::vec3 currentRotation = camera.getRotationDegrees();
            glm::vec3 shakePosition = currentPosition + glm::vec3(0,shake,0)
    ;
            glm::vec3 shakeRotation = currentRotation + glm::vec3(shake);
            camera.setPosition(shakePosition);
            camera.setRotationDegrees(shakeRotation);
            timeElapsed += _deltaTime;
        }
        else {
            camera.setPosition(originalPosition);
            camera.setRotationDegrees(originalRotation);
            isShaking = false;
        }
    }
}

void Camera_Shake::playShake() {
    if(!isShaking) {
        originalPosition = camera.getPosition();
        originalRotation = camera.getRotationDegrees();
        timeElapsed = 0;
        isShaking = true;
    }
}
```

L'implémentation que j'ai choisi nécessite un attribut de type `Camera_Shake` présent dans `Camera`. Dans la fonction `update`, on appelle `m_shake->update(_deltaTime)` (qui ne fait quelque chose que si `isShaking` vaut `true`).

Dans `ImGui`, j'ai fait en sorte qu'on puisse changer la durée du shake, sa fréquence et son amplitude. Quand on appuie sur le bouton "`Play shake`", on fait appel à `m_shake->playShake()`, ce qui déclenche le mouvement de caméra.

```
ImGui::Text("Camera shake");
ImGui::SliderFloat("Shake duration",&m_shake->duration,0.1f,10.0f);
ImGui::SliderFloat("Shake amplitude",&m_shake->amplitude,0.1f,2.0f);
ImGui::SliderFloat("Shake frequency",&m_shake->frequency,5.0f,10.0f);
if(ImGui::Button("Play shake")) {
  m_shake->playShake();
}
```

N.B : J'ai fait en sorte qu'à la fin du shake, la caméra revienne à sa position et rotation d'origine, mais je ne suis pas certain que ce soit une bonne pratique. Pour certaines amplitudes, fréquences et durées, cela provoque des sauts de caméra.

8 Bonus

J'aurai adoré faire les questions bonus, malheureusement par manque de temps je ne me suis pas penché dessus, croulant sous les divers projets simultanés.

Cependant, j'ai tout de même pris le temps de créer une petite fenêtre `ImGui` d'aide à l'utilisation de l'application.

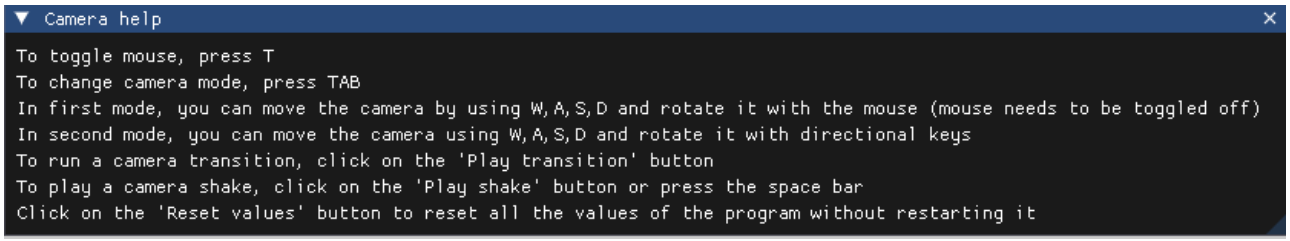


Figure 6: Aide à l'utilisation de l'application dans `ImGui`

9 Rendu final de l'application

Voici un aperçu final de l'application, avec la fenêtre `ImGui` complète.

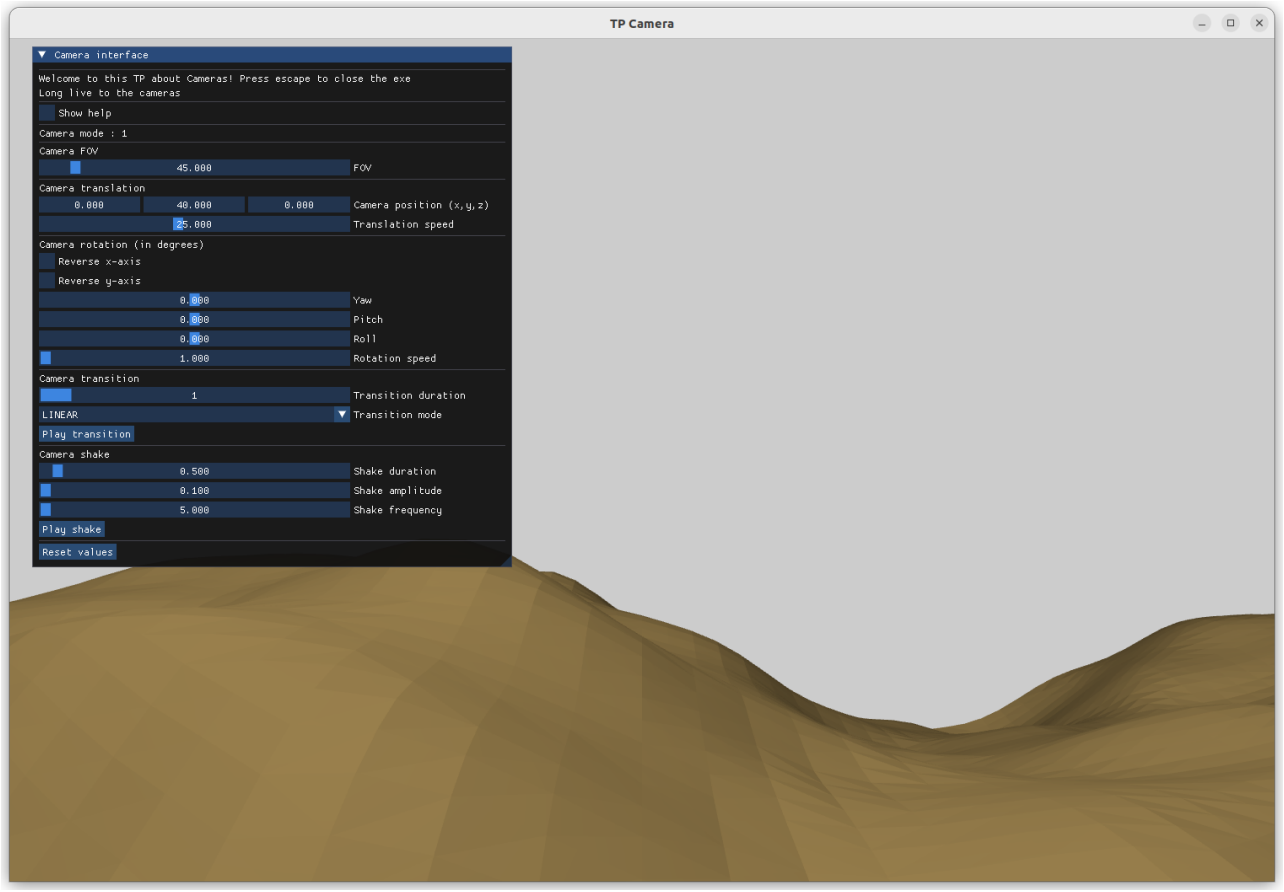


Figure 7: Rendu final de l'application

10 Conclusion

Le travail réalisé lors de ce TP a grandement contribué à ma compréhension des systèmes de caméra dans les moteurs de jeu. Bien que la section bonus n'ait pas été abordée, les fonctionnalités implémentées offrent déjà une base solide pour de futurs développements et expérimentations. J'ai pris un plaisir insoupçonné à réaliser ce TP, à vrai dire au premier abord je ne pensais pas que programmer des caméras pouvait être aussi fun. Le cours et le TP était tous deux bien structurés, ce qui m'a motivé.

Merci pour le temps et l'attention que vous avez consacrés à la lecture de ce compte-rendu.