

Gervais Alex
111 010 475
Levasseur Pierre-Marc
111 080 897
Perreault Keven
111 054 716

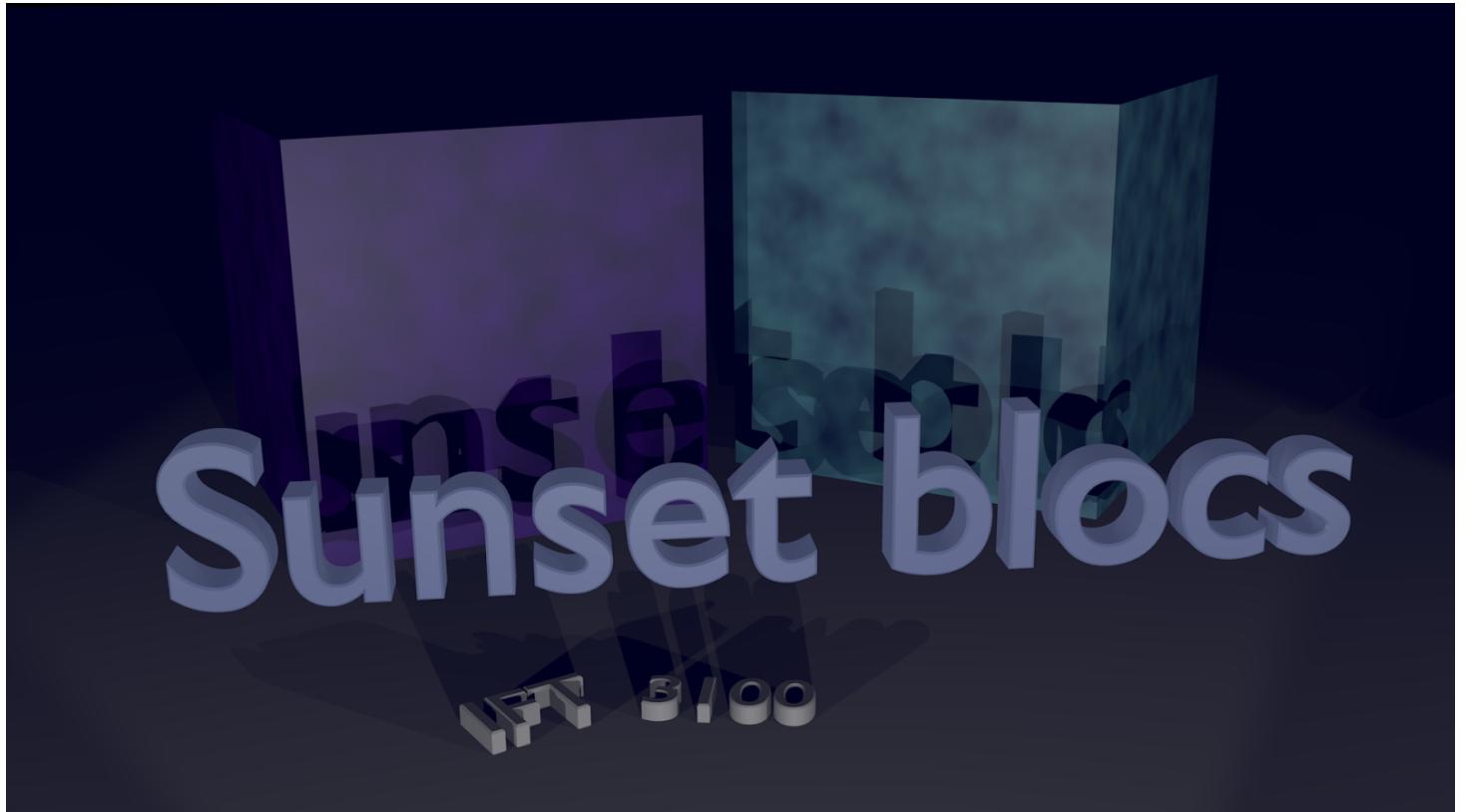
Infographie
IFT-3100

Document de design
Travail Pratique 2

Travail présenté à
Philippe Voyer

Département d'informatique
Université Laval
Hiver 2015

Projet Sunset Blocs



(Image créée dans *Blender*)

[Sommaire](#)

[Utilisation](#)

[Utilisation de l'application](#)

[Compilation et démarrage](#)

[Technologie](#)

[Architecture](#)

[Fonctionnalités](#)

[1.2 Graphe de scène](#)

[1.4 Occlusion](#)

[1.5 Sélection](#)

[2.2 Primitives géométriques](#)

[2.3 Modèle](#)

[2.5 Texture](#)

[4.2 Shader de fragment](#)

[5.2 Lumière directionnelle](#)

[5.3 Lumière ponctuelle](#)

[5.4 Projecteur](#)

[5.5 Lumières éditable](#)

[6.2 Matériaux avec effet de relief](#)

[6.3 Matériaux à textures multiples](#)

[7.5 Effets plein écran](#)

[8.5 Interactivité par pointage](#)

[Extra: Anti Aliasing](#)

[Ressources](#)

[Présentation](#)

Sommaire

L'application que nous avons choisi de développer est un petit jeu de construction. À l'aide d'une caméra à la première personne, il est possible de créer et de déplacer des primitives géométriques pour faire des constructions originales. Pour inspirer l'utilisateur, nous avons créé une scène de base qui contient plusieurs éléments. Donc, avec un « cubemap », quelques surfaces rocheuses et une surface représentant un étendue d'eau, nous avons créé une ambiance relaxante et propice à la création de construction en trois dimensions. De plus, pour créer une impression de réalisme, nous avons développer un ensemble d'algorithmes pour illuminer cette scène de différentes façons. Ainsi, les primitives que l'utilisateur créera dans la scène auront différentes aspects et différentes couleurs selon le choix de matériau et selon la position dans la scène. L'utilisateur aillant complété une construction pourra prendre une capture d'écran et la partager avec tous ces amis. Selon son inspiration, il pourra appliquer un filtre à l'écran avant de faire la capture et ainsi obtenir une photo unique.

La prise en main de l'application sera assez intuitive pour les habitués de jeu avec des caméras en première personne. Nous avons respecté les standards actuels en ce qui concerne les contrôles de déplacement dans l'univers 3D: la touche W permet d'avancer, A pour se déplacer vers la gauche, S pour reculer, D pour se déplacer vers la droite ainsi que l'orientation de la caméra à l'aide de la souris. Un menu à gauche de l'écran permet à l'utilisateur de configurer plusieurs options. Ce menu, accessible avec la touche P, permet de changer de type de primitive, changer de type de matériau, changer la vitesse de déplacement et beaucoup d'autres options.

Cette application est développée dans l'environnement de développement « Microsoft Visual Studio 2012 » et elle utilise la librairie « Openframeworks », qui elle utilise une multitude d'autres librairies. Cependant, puisque nous voulions avoir un plein contrôle sur tous les aspects de notre projet, nous avons choisi d'utiliser « Openframeworks » seulement pour nous donner un contexte de rendu et pour quelques classes utilitaires. Nous avons donc développé nous même un ensemble de classes originales qui utilise directement « OpenGL » pour faire le rendu. Nous avons aussi développé plusieurs groupes de shaders pour faire les calculs d'illumination et les effets de filtre. Le projet maintenant terminé aurait pu être développé davantage pour devenir un éditeur plus complet. Le système est très modulaire et avec les fonctionnalités et les modifications que nous avons fait depuis la première version, nous aurions pu l'étendre beaucoup plus. Le projet est disponible sur GitHub à l'adresse suivante:

<https://github.com/PMarcL/TP-IFT3100-H15>

Utilisation

Utilisation de l'application

L'application est très conviviale et son utilisation ressemble beaucoup à un jeu d'exploration à la première personne. En utilisant des touches qui sont similaires aux jeux vidéo, l'intuition de l'utilisateur lui permet d'évoluer dans l'application sans avoir une courbe d'apprentissage élevée.

Tout d'abord, l'utilisateur peut se déplacer dans l'environnement en utilisant les touches suivantes:

- w - se déplacer vers l'avant;
- s - se déplacer vers l'arrière;
- a - se déplacer vers le côté gauche;
- d - se déplacer vers le côté droit.

L'utilisateur peut regarder dans tous les angles en déplaçant la souris.

Le menu est présenté à l'aide de la touche 'm'. À l'intérieur de celui-ci se trouve plusieurs boutons permettant de modifier des paramètres affectant le rendu de la scène.

Dans l'ordre apparaissant dans le menu, les fonctionnalités sont:

- p - mettre en mode pause (et ainsi pouvoir accéder au menu avec la souris);
- v - activer/désactiver l'effet vertigo (visible en se déplaçant);
- q - activer/désactiver la lampe de poche;
- t - activer le mode double primitive. Dans ce mode, l'ajout de primitives se fait en double et les deux instances sont synchronisées au niveau de leur déplacements et autres changements;
- 1 - activer/désactiver l'effet plein écran de brouillard;
- 2 - activer/désactiver l'effet plein écran noir et blanc;
- 3 - activer/désactiver l'effet plein écran de détection de lignes;
- barre de sélection permettant de modifier la vitesse de déplacement de la caméra;
- barre de sélection permettant de changer la primitive à créer. Il y a 5 types de primitives différentes à choisir:
 - ◆ 1. Cube
 - ◆ 2. Tetraèdre
 - ◆ 3. Octaèdre
 - ◆ 4. Cône
 - ◆ 5. Sphère
- barre de sélection permettant de changer le matériau de la primitive à créer. Il y a 21 types de matériaux différents à choisir;
 - ◆ Plastique: (1) Noir, (2) Bleu, (3) Vert, (4) Rouge, (5) Blanc, (6) Jaune;
 - ◆ Caoutchouc: (7) Noir, (8) Rouge, (9) Bleu;
 - ◆ Pierres: (10) Émeraude, (11) Jade, (12) Obsidienne, (13) Perle, (14) Ruby (par défaut), (15) Turquoise;
 - ◆ Métaux: (16) Laiton, (17) Bronze, (18) Chrome, (19) Cuivre, (20) or, (21) argent

- barre de sélection permettant de changer la précision de la grille. Lorsqu'on déplace une primitive, sa position sera arrondie selon la grandeur de la grille;
- barres de sélection permettant de choisir la couleur de la lampe de poche et la grosseur du cone de projection de la lampe de poche.

Les touche suivantes sont utilisées pour modifier les paramètres de la caméra:

- la flèche vers le haut permet d'élargir le champ de vision;
- la flèche vers le bas permet de rétrécir le champ de vision.

De plus il est possible de:

- prendre une capture d'écran avec la touche 'i';
- entrer en mode plein écran avec la touche 'f';
- créer une primitive en cliquant sur le bouton droit de la souris;
- sélectionner une primitive et la déplacer en cliquant le bouton de gauche de la souris.

Compilation et démarrage

Le livrable 2 est accompagné d'une version exécutable de l'application. Pour l'exécuter, il ne suffit que de cliquer sur le fichier .exe fourni.

- ★ Attention! L'application charge plusieurs fichiers au démarrage et il est possible qu'il y ait un petit délai au lancement. Il faut simplement patienter devant l'écran blanc quelques secondes et l'application va démarrer comme il se doit.
- ★ La création de la première sphère sera un peu longue, car l'algorithme génère environ 4 millions de sommets. Par la suite, les sphères seront créées instannément.

Si vous voulez compiler le projet, il est nécessaire de placer le code source sous le path <librairie openframeworks>\apps\myApps. Le code a été testé sous VisualStudio 2012 avec la version 0.8.4 de la librairie OpenFrameworks.

Technologie

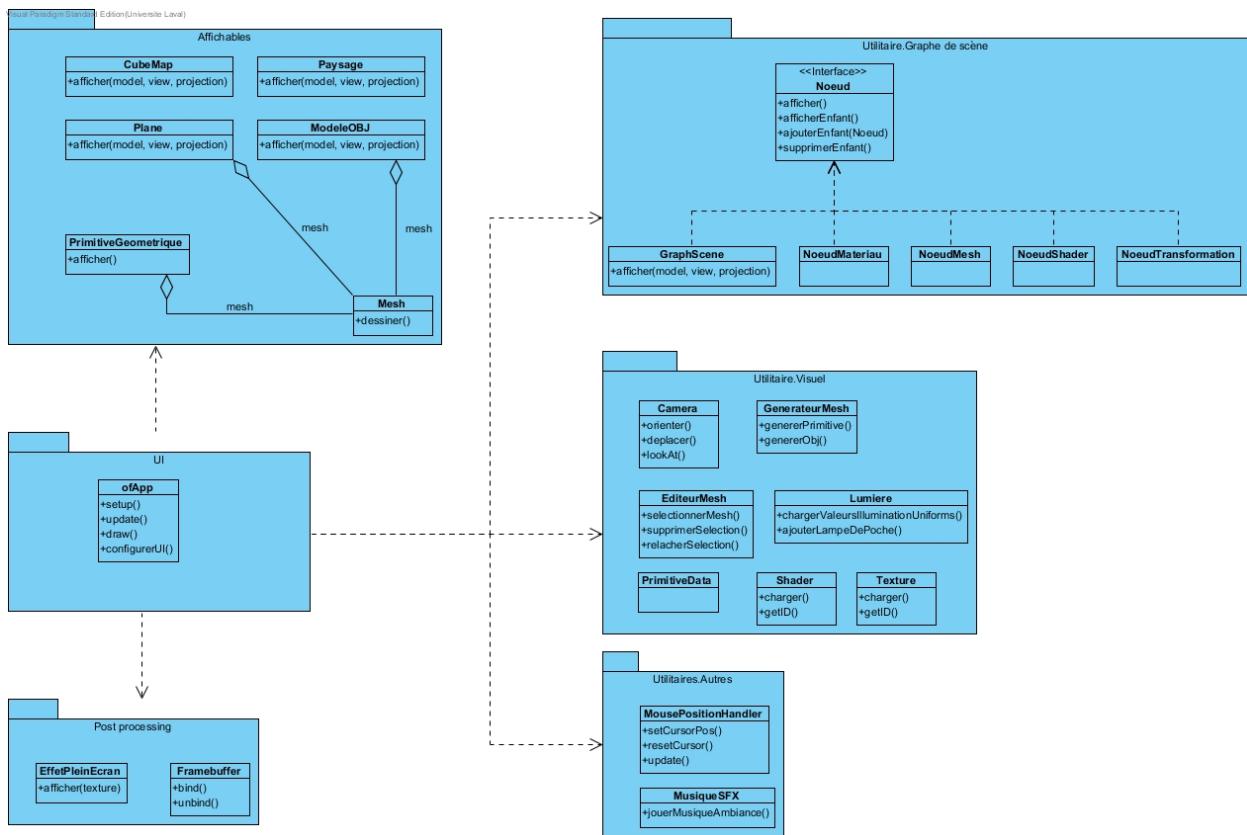
Tel que mentionné dans le sommaire, nous avons décidé d'utiliser la librairie openframeworks sous Windows pour réaliser le projet. Cette librairie nous offre déjà beaucoup de solution pour plusieurs fonctionnalités demandées dans ce travail, mais nous avons décidé de faire abstraction de la majorité de ce qu'offre openframeworks et faire le rendu graphique nous même. Alors, nous l'avons utilisé pour nous donner un contexte de rendu, pour tout ce qui entoure la manipulation des matrices et pour la gestion des entrées (clavier, souris). Nous avons fait le reste nous-même (shaders, textures, primitives géométriques, etc). Nous utilisons tout de même quelques fonctions et classes utilitaires de la librairie pour ce qui touche les fonctionnalités qui ne sont pas évaluées dans le cadre de ce cours comme par exemple la classe ofSoundPlayer pour la musique et les effets sonores.

Pour ce qui est de l'environnement de développement, openframeworks nous imposait deux choix sous windows: code::blocks et Visual Studio 2012. Nous avons donc décidé de travailler dans Visual Studio puisque nous avions déjà un peu d'expérience avec cet IDE.

Finalement, les exécutables générés par notre projet sont seulement fonctionnels dans un environnement Windows avec une carte graphique compatible avec la version 4.3 de OpenGL.

Architecture

Notre projet est composé de quatres packages distincts qui contiennent toutes les classes que nous avons créé pour le projet. Nous avons premièrement le package «UI» qui contient l'interface graphique et la boucle de rendu. Ensuite, nous avons rassemblé les éléments affichables ensemble dans un même package. Il contient donc le «CubeMap», le paysage, etc. Nous avons aussi rassemblé les classes qui font le «post-processing» ensemble. Finalement, nous avons l'ensemble des classes utilitaires qui sont divisées en trois sous-ensembles. Voici un diagramme de classe simplifié ainsi qu'une courte description de chacune des classes du projet:



Interface utilisateur

La classe principale est ofApp qui s'occupe d'encapsuler la boucle de rendu. Elle gère aussi les entrées au clavier et à la souris. Le fichier contenant la fonction «main» (main.cpp) s'occupe de générer le contexte de rendu et de lancer l'application. Cette classe et le fichier main sont fournis lorsqu'on génère un projet avec openframeworks. Elle a toutefois été grandement modifiée pour nous permettre d'implémenter les fonctionnalités décrites dans ce document.

Éléments affichable:

Dans les éléments affichables, on retrouve les classes suivantes: CubeMap, ModeleOBJ, Paysage, Plane, PrimitiveGeometrique et Mesh.

La classe PrimitiveGeometrique permet de créer toutes les primitives géométriques que nous avons choisis d'offrir à l'utilisateur. Pour y arriver, elle utilise le module PrimitiveData qui contient toutes les informations et les algorithmes pour générer les sommets, les normales et les coordonnées de texture des différentes primitives géométrique. La classe CubeMap est simplement un cube qui charge une texture par face et fait un rendu spécial pour créer un ciel enveloppant autour de la caméra. Ensuite, la classe Plane permet de rendre une «heightmap» selon un certain nombre de paramètres passé à son constructeur. Cette classe peut être rendu avec une texture et elle peut être illuminée selon le shader de fragment utilisée. La classe ModeleOBJ, permet de charger et de rendre un modèle. Ensuite, la classe Paysage est une classe spéciale qui utilise des instances de la classe Plane et ModeleOBJ pour rendre un environnement explorable par l'utilisateur. Finalement, plusieurs de ces classes dépendent de la classe Mesh qui contient les éléments pour créer le maillage géométrique et transférer les données vers le GPU (VBO, VAO).

Post-processing:

Ce package comprend seulement les deux classes suivantes: EffetPleinEcran et Framebuffer. Donc, la classe EffetPleinEcran s'occupe de faire le rendu final de l'image à l'écran. Par défaut, elle ne fait aucun traitement sur celle-ci. Par contre, elle l'offre la possibilité de traiter l'image de différentes façons avant de faire le rendu. La classe Framebuffer s'occupe de gérer dans quel framebuffer est utilisé et comment celui-ci est configuré. Cette classe permet aussi de faire de l'anti-aliasing sur le framebuffer dans lequel est rendu la scène avant le post-processing.

Utilitaires:

Graphe de scène:

Ce premier sous-ensemble des utilitaires contient les classes suivantes: Noeud, GraphScene, NoeudMesh, NoeudTransformation, NoeudShader, NoeudMateriau.

La classe Noeud est une classe générique utilisé pour construire tous les noeuds nécessaire à notre graphe de scène. Le graphe de scène est par ailleurs lui-même un noeud. Les classes enfants de la classe Noeud ont tous un rôle bien défini. GraphScene sert de point de départ pour le rendu du graphe de scène, il initialise tous les paramètres. NoeudMesh contient un pointeur vers un mesh et l'affiche lors du rendu d'une image si celui-ci se trouve devant la caméra. NoeudTransformation quant à lui applique toutes sortes de transformations (rotations, redimensions, translations, etc.) aux noeuds le suivant dans le graphe de scène. NoeudShader détermine le programme OpenGL qui sera utilisé lors du rendu de ses enfants. Pour terminer, NoeudMateriau sert a déterminer le matériau qui sera appliqué sur les mesh de ces enfants.

Visuel:

Les utilitaires visuels comprennent les classes suivantes: Camera, GenerateurMesh, EditeurMesh, Lumiere, PrimitiveData, Shader et Texture.

Premièrement, la classe Camera fait la gestion de ce qui entoure la caméra de la scène (déplacement, orientation, etc.). Ensuite, les classes Shader et Texture sont des classes qui nous permettent de charger des programmes en GLSL et des images dans le GPU. Ce sont des classes très générales utilisées un peu partout dans notre projet. La classe Lumière s'occupe de gérer toutes les lumières de la scène et d'envoyer l'information sur celles-ci vers les shaders. Elle s'occupe aussi de gérer la lampe de poche. Nous avons aussi le module PrimitiveData qui n'est pas une classe. Ce module contient des fonctions pour récupérer ou générer les sommets et les normales des primitives géométriques offertes dans notre projet. Il contient aussi les informations sur les différents matériaux que nous pouvons appliquer sur les primitives. La classe GenerateurMesh comme son nom l'indique s'occupe de générer des instances de la classes Mesh avec les informations voulues. Cette classe s'occupe d'ailleurs de garder en mémoire les sommets déjà générés pour une primitive donnée. Ainsi, le programme génère une seule fois les sommets des primitives complexes comme le cône et la sphère. Finalement, la classe EditeurMesh permet de sélectionner, déplacer et supprimer les primitives présentes dans la scène.

Autre:

Finalement, ce dernier sous ensemble comprend seulement les utilitaires suivants: MousePositionHandler et MusiqueSFX.

La classe MousePositionHandler permet de récupérer et de déplacer le curseur selon les limites de l'écran et la classe MusiqueSFX s'occupe de jouer de la musique et des effets sonores.

Fonctionnalités

1.2 Graphe de scène

Notre projet contient un graphe de scène qui sert principalement à organiser les primitives créées par l'utilisateur. Cependant, ces capacités sont loin d'être exploitées (dû au manque de temps pour ajouter des fonctionnalités). Notre graphe de scène est en fait un nœud; nous avons créé une classe très générique nommée Nœud dont plusieurs autres héritent : NoeudMateriau, NoeudMesh, NoeudShader, NoeudTransformation et bien entendu GraphScene. Elles ont chacun leurs particularités, comme leur nom l'indique. Un nœud peut avoir un parent et des enfants. Le graphe de scène a donc une structure d'arbre. Durant le rendu du graphe de scène, chaque nœud peut afficher quelque chose ou mettre à jour un paramètre, comme par exemple un matériau ou bien un shader qui sera utilisé par ses enfants. Si un nœud change un paramètre avant d'afficher ses enfants, il rétablit immédiatement le paramètre précédent une fois terminé.

Dans la boucle de mise à jour de la scène, la fonction « afficher » est appelée sur une instance de la classe GraphScene. C'est alors que le graphe de scène est parcouru en profondeur pour rendre les mesh selon les caractéristiques spécifiées par leurs parents.

Voici la méthode afficher de la classe GraphScene en détail :

```
void GraphScene::afficher(const ofMatrix4x4* projection, const ofMatrix4x4* vue, const Lumiere* lumiere, ofVec3f orientationCamera, ofVec3f positionCamera){  
    GLint precedentGlProgram;  
    glGetIntegerv(GL_CURRENT_PROGRAM, &precedentGlProgram);  
  
    preparerContextRendu(projection, vue, lumiere, this->shaderIdOrigine);  
  
    ParametresAffichage* paramsAff = new ParametresAffichage();  
    paramsAff->projection = projection;  
    paramsAff->vue = vue;  
    paramsAff->lumiere = lumiere;  
    paramsAff->modeles = new vector<ofMatrix4x4*>;  
    paramsAff->normalPlan = new ofVec3f(orientationCamera.getNormalized());  
    paramsAff->positionPlan = new ofVec3f(positionCamera);  
  
    this->afficherEnfants(paramsAff, this->shaderIdOrigine, MATERIAUX::AUCUN);  
  
    glUseProgram(precedentGlProgram);  
}
```

1. Récupère l'identifiant du programme présentement utilisé par OpenGL.
2. Prépare le contexte de rendu « par défaut » du graphe de scène.
3. Prépare une variable ParametresAffichage contenant plusieurs informations utiles au rendu.
4. Appel la fonction afficherEnfants avec tous les paramètres nécessaires qui elle appellera la fonction afficher de tous les enfants du graphe de scène. Ce qui va rendre la scène.
5. Rétabli le programme utilisé par OpenGL avant l'appel de la méthode.

La structure du graphe de scène se met à jour de manière interactive selon les manipulations effectuées lorsque l'utilisateur s'amuse à construire sa scène. La gestion de l'espace mémoire est bien implantée : chaque shader et mesh n'est chargé en mémoire GPU qu'une seule fois, peu importe le nombre de fois qu'ils sont utilisés dans le graphe, et ce de manière dynamique. Lors du rendu, le graphe de scène passe à travers chaque nœud qu'une seule

fois et un algorithme d'occlusion détermine si un mesh doit être rendu par le GPU ou non, ce qui rend son exécution très rapide.

1.4 Occlusion

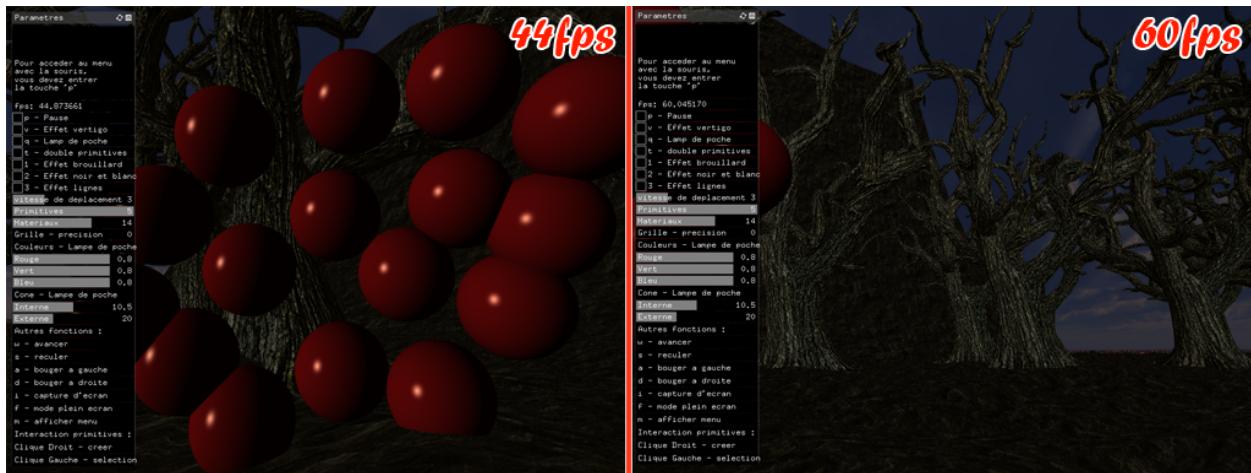
Un algorithme d'occlusion est présent dans notre graphe de scène. Il vérifie si un mesh se trouve devant la caméra et si c'est le cas, le mesh est rendu par le GPU. Le graphe de scène prend en paramètre, lors de la génération d'une image, la position ainsi que l'orientation de la caméra. Par la suite, la normal de l'orientation est calculé. Ces valeurs sont alors utilisées lorsque vient le moment de vérifier si la position absolue, d'un mesh dans la scène, se trouve devant ou derrière la caméra.

```
bool NoeudMesh::positionneDevantPlan(ofVec3f normalPlan, ofVec3f positionPlan){  
    ofVec3f temp = this->positionAbsolue - positionPlan;  
    float res = temp.dot(normalPlan);  
    if(res > 0){  
        return true;  
    }  
    return false;  
}
```

(on parle ici du “plan” formé par la position et la direction de la caméra)

Démonstration de l'efficacité de l'algorithme

Dans cette scène les sphères ont tous 3 898 800 sommets. Lorsque la caméra regarde directement un grand ensemble de ces sphères, on constate que le nombre d'images par seconde diminue à 44 et lorsque la caméra regarde plus vers la droite, là où il y a moins de sphères visibles, le nombre d'images par seconde est augmenté à 60 (notre cap).



1.5 Sélection

Dans notre application, il est possible d'effectuer des sélections sur les mesh générés dans la scène. La sélection se fait avec le clique de la souris et choisi l'élément se trouvant à 75 unités devant la caméra ou bien l'élément le plus proche des 75 unités devant la caméra allant jusqu'à un écart de 50 unités.

Pour ce faire, l'appel de la fonction trouverMesh sera fait sur une instance du graphe de scène. Cette fonction prépare la recherche et appelle la fonction chercherMesh qui cherche parmi tous les enfants du graphe, le mesh se trouvant le plus près de la position demandée.

```
Noeud* Noeud::trouverMesh(ofVec3f position, float rayon){
    float d = FLT_MAX;
    float* distanceMinimum = &d;

    return this->chercherMesh(NULL, position, rayon, distanceMinimum, new vector<ofMatrix4x4*>);
}

Noeud* Noeud::chercherMesh(Noeud* meshPlusProche, ofVec3f position, float rayon, float* distanceMinimum, vector<ofMatrix4x4*>* transformations){
    for(list<Noeud*>::iterator i = this->listeEnfants.begin(); i != this->listeEnfants.end(); i++)
        meshPlusProche = (*i)->chercherMesh(meshPlusProche, position, rayon, distanceMinimum, transformations);
    return meshPlusProche;
}
```

Lorsque la fonction chercherMesh est appelée sur une instance de NoeudMesh, il y a alors un calcul de la distance entre le point visé par la caméra et la position du mesh dans la scène. Si celui-ci se trouve dans le rayon voulu et que sa distance est plus petite que le dernier trouvé, il devient alors le mesh le plus proche et la recherche se poursuit. Dans le cas où aucun mesh n'a été trouvé, la valeur NULL est retournée.

```
Noeud* NoeudMesh::chercherMesh(Noeud* meshPlusProche, ofVec3f position, float rayon, float* distanceMinimum, vector<ofMatrix4x4*>* transformations){
    this->ajouterModele(transformations);

    float distance = transformations->back()->getTranslation().distance(position);
    if(distance <= rayon && distance < *distanceMinimum){
        *distanceMinimum = distance;
        meshPlusProche = this;
    }

    meshPlusProche = Noeud::chercherMesh(meshPlusProche, position, rayon, distanceMinimum, transformations);

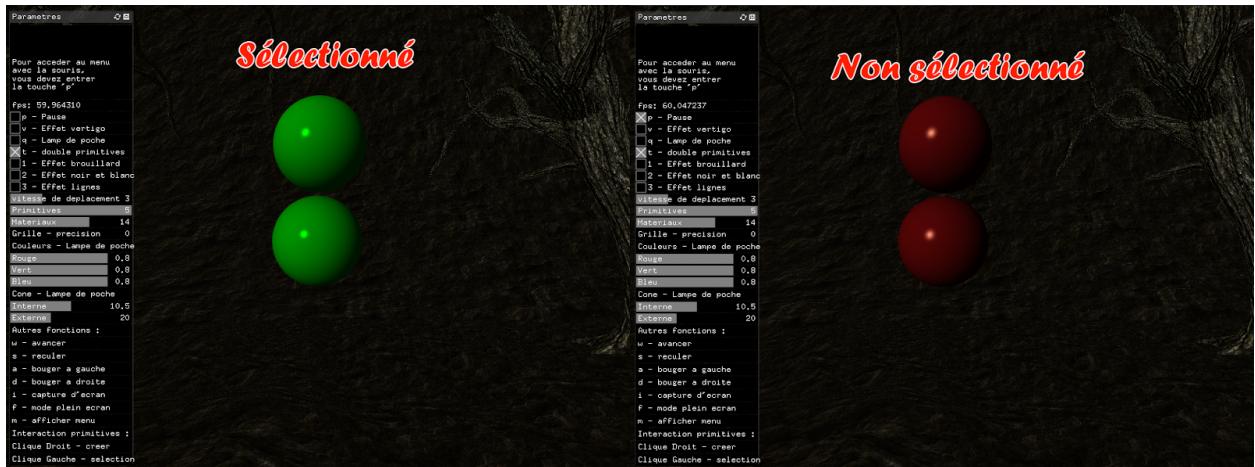
    this->retirerModele(transformations);
}

return meshPlusProche;
}
```

Le résultat de cette recherche est retourné à l'éditeur de mesh, c'est-à-dire une instance de la classe EditeurMesh. Le mesh sélectionné par cet objet se voit appliqué un matériau de sélection (de couleur verte) et peut être édité. Dans cette version, seule la position peut être changée par le déplacement de la souris.



Si un mesh a des enfants, ces derniers subissent également le changement de matériau ainsi que les déplacements, pendant que la sélection est active.



Pour désélectionner un mesh, il suffit de relâcher le bouton gauche de la souris.

2.2 Primitives géométriques

Puisque notre application sert à faire des constructions géométriques, nous avons beaucoup changé la manière dont les primitives sont gérées et créées. Nous avons aussi ajouté deux primitives plus complexes qui sont générées dynamiquement par des algorithmes.

Premièrement, nous avons ajouté un cône. L'algorithme qui génère les cônes est totalement original. Nous nous sommes basés sur aucun tutoriel pour le faire et nous l'avons fait pour qu'il soit configurable par des constantes. Il commence tout d'abord par calculer l'angle qu'il y

aura entre chacun des triangles qui compose sa base selon une précision. Cette précision est une constante que nous avons décidé de fixer à 500. Bien sûr, elle est facilement modifiable par programmation pour obtenir des cônes plus ou moins précis. Ensuite, nous faisons un nombre d'itération égal à la précision pour générer les triangles qui composent le cône. Le premier triangle est fait partie de la base. Il utilise le rayon (une autre constante) et les fonctions trigonométriques pour trouver l'emplacement des sommets. Nous avons décidé de fixer le centre de la base du cône au point (0,0,0) dans l'espace de l'objet. Le deuxième triangle généré utilise les deux sommets à l'extrémité du cercle de sa

```
vector<float> genererSommetsCone() {
    vector<float> sommets;
    float angleEntreSommets = (2*PI) / CONE_PRECISION;
    for(int i = 0; i < CONE_PRECISION; i++) {
        // base du cone
        sommets.push_back(0);
        sommets.push_back(0);
        sommets.push_back(0);

        sommets.push_back(CONE_RAYON * cos(i * angleEntreSommets));
        sommets.push_back(0);
        sommets.push_back(CONE_RAYON * sin(i * angleEntreSommets));

        sommets.push_back(CONE_RAYON * cos((i + 1) * angleEntreSommets));
        sommets.push_back(0);
        sommets.push_back(CONE_RAYON * sin((i + 1) * angleEntreSommets));

        // Côté du cone
        sommets.push_back(0);
        sommets.push_back(CONE_HAUTEUR);
        sommets.push_back(0);

        sommets.push_back(CONE_RAYON * cos((i + 1) * angleEntreSommets));
        sommets.push_back(0);
        sommets.push_back(CONE_RAYON * sin((i + 1) * angleEntreSommets));

        sommets.push_back(CONE_RAYON * cos(i * angleEntreSommets));
        sommets.push_back(0);
        sommets.push_back(CONE_RAYON * sin(i * angleEntreSommets));
    }

    return sommets;
}
```

base un le point (0, hauteur, 0) qui représentera la pointe du cône. Voici un exemple d'un cône généré avec cet algorithme:



Nous avons aussi ajouté un algorithme qui génère une sphère. Cet algorithme est plus complexe et moins efficace que celui du cône, mais il donne un résultat assez intéressant. Il est divisé en deux parties. Il génère tout d'abord l'ensemble des sommets qui composent la sphère. L'algorithme que nous avons choisis travaille avec une précision en latitude et en longitude. Donc, pour générer les sommets, on fait deux boucles imbriquées qui génère une série de cercles qui forment en quelque

```
void genererSommetsSphere(vector<float>& sommets) {
    vector<ofVec3f> sommetsInit;
    generationDesSommetsInitiauxSphere(sommetsInit);

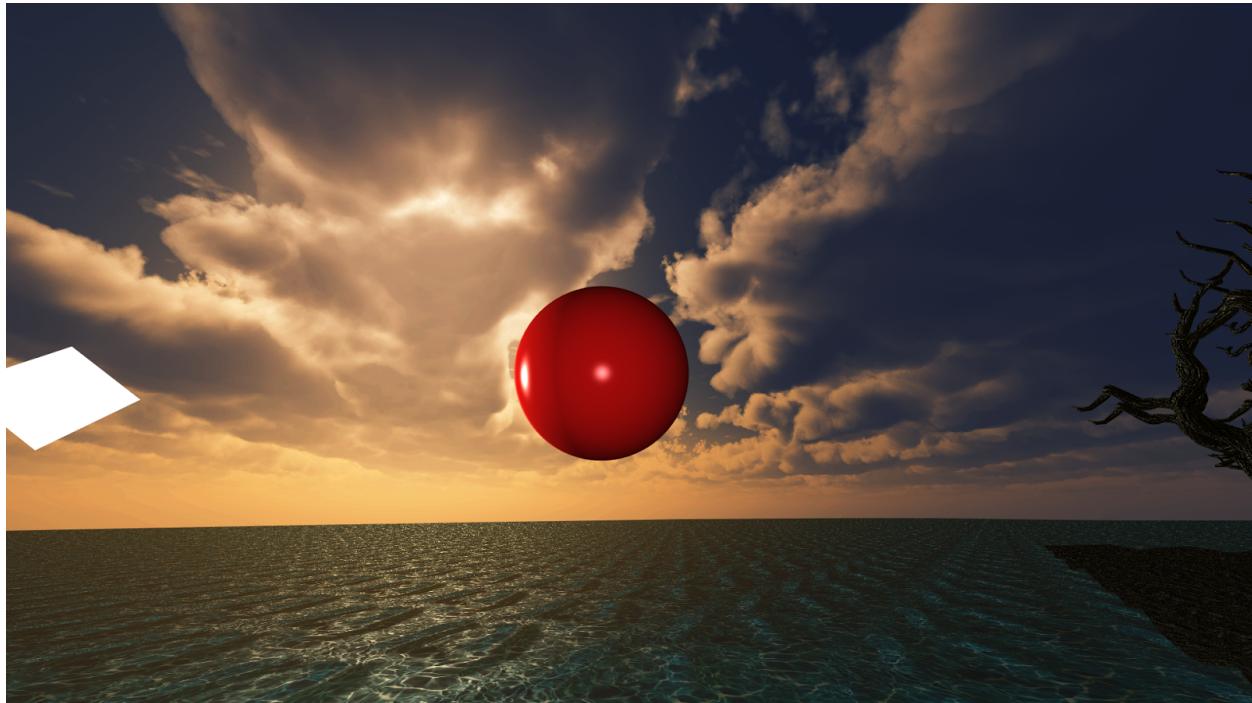
    triangulationDessusSphere(sommetsInit, sommets);
    triangulationPartieCentraleSphere(sommetsInit, sommets);
    triangulationDessousSphere(sommetsInit, sommets);
}
```

```
void generationDesSommetsInitiauxSphere(vector<ofVec3f>& sommets) {
    float deuxPI = 2 * PI;
    sommets.push_back(ofVec3f(0, SPHERE_RAYON, 0));
    for( int lat = 0; lat < SPHERE_PRECISION_LAT; lat++ )
    {
        float a1 = PI * (float)(lat+1) / (SPHERE_PRECISION_LAT+1);
        float sin1 = sin(a1);
        float cos1 = cos(a1);

        for( int lon = 0; lon <= SPHERE_PRECISION_LON; lon++ )
        {
            float a2 = deuxPI * (float)(lon == SPHERE_PRECISION_LON ? 0 : lon) / SPHERE_PRECISION_LON;
            float sin2 = sin(a2);
            float cos2 = cos(a2);

            sommets.push_back(ofVec3f(sin1 * cos2, cos1, sin1 * sin2) * SPHERE_RAYON);
        }
        sommets.push_back(ofVec3f(0, -SPHERE_RAYON, 0));
    }
}
```

sorte des tranches de la sphère. On triangularise ensuite toute la surface de la sphère pour créer un maillage géométrique affichable à l'écran. Voici un exemple de sphère générée avec cet algorithme dans notre projet:



2.3 Modèle

Nous avons décidé d'améliorer notre chargeur de modèle OBJ afin qu'il soit en mesure de récupérer les coordonnées de texture et les normales. Pour y arriver, nous avons ajouter l'interprétation des lignes commençant par les indices «vn», «vt» et «t». Ainsi, nous accumulons les sommets, les coordonnées de textures et les normales dans des listes et en utilisant le système d'indice (ligne commençant pas «t») nous sommes en mesure de récupérer l'ensemble des attributs de sommets pour chacun des sommets. C'est une grande

amélioration par rapport à notre première version, car nous évitons de calculer nous même les normales et nous avons la possibilité d'appliquer une texture au modèle que nous chargeons. La photo qui suit montre les modèles que nous avons chargé dans le paysage de base de notre projet. Il y a des champignons et des arbres que nous avons modifié avec le logiciel Blender.

```
while(getline(fichier, ligne))
{
    string enTete = ligne.substr(0,2);
    istringstream s(ligne.substr(2));
    if(enTete == "v ")
    {
        ofVec3f vertex;
        s >> vertex.x;
        s >> vertex.y;
        s >> vertex.z;
        tempVertices.push_back(vertex);
    }
    else if(enTete == "vt")
    {
        ofVec2f texCoord;
        s >> texCoord.x;
        s >> texCoord.y;
        tempTexCoords.push_back(texCoord);
    }
    else if(enTete == "vn")
    {
        ofVec3f normal;
        s >> normal.x;
        s >> normal.y;
        s >> normal.z;
        tempNormals.push_back(normal);
    }
    else if(enTete == "f ")
    {
        GLuint vertexIndex[3], texCoordIndex[3], normalIndex[3];
        s >> vertexIndex[0];
```



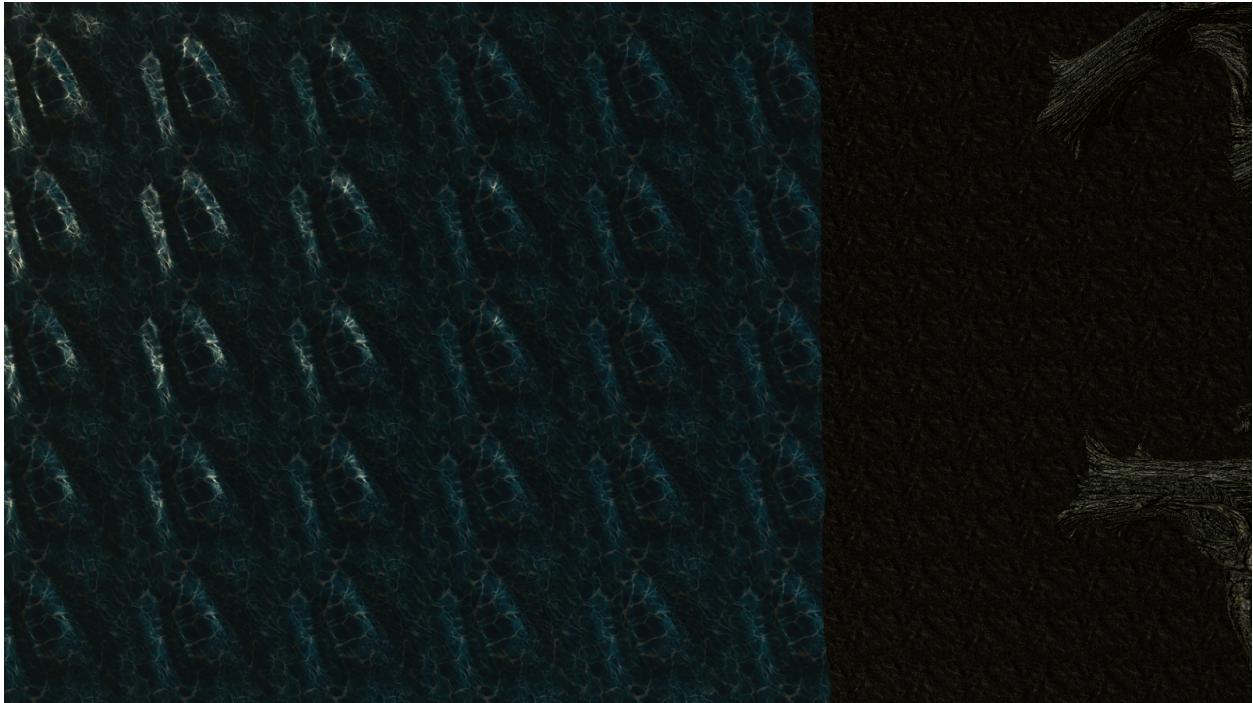
2.5 Texture

Dans notre projet, nous utilisons les textures de plusieurs manières différentes et sur plusieurs types d'objet. L'utilisation la plus notable des textures est sur nos grandes surfaces qui servent à créer la surface de l'eau et le sol rocheux. La triangulation de ces surfaces est paramétrisable. Donc, on peut créer un maillage assez serré ou on peut faire une surface carré contenant seulement deux triangles. Alors, pour appliquer une texture sur la

surface, nous avons décidé de distribuer les coordonnées de texture selon un ratio modifiable selon les besoins et les textures utilisées. En utilisant cette technique, nous évitons de distribuer les coordonnées de texture toujours de la même façon à l'intérieur d'un des carré de la surface. Ensuite, pour charger les images et créer les textures dans OpenGL, nous avons créé

```
void Plane::ajouterTexCoordPourChaqueSommet()
{
    for(int i = 0; i < nbColonnes + 1; i++)
    {
        for(int j = 0; j < nbLignes + 1; j++)
        {
            texCoords.push_back(j * ratioTextureParCarre);
            texCoords.push_back(i * ratioTextureParCarre);
        }
    }
}
```

une classe nommée Texture. Cette classe utilise la classe ofImage de OpenFrameworks pour charger les images et elle s'occupe de générer la texture avec les commandes de OpenGL. Cette classe est utilisée partout où nous avons utilisé une ou plusieurs textures. On peut voir un exemple de la distribution des textures sur une surface dans la photo suivante:



Suite au modification que nous avons fait dans notre chargeur de modèle OBJ, nous nous sommes donné la possibilité d'appliquer des textures sur la surface de ces modèles. Puisque les coordonnées de textures sont lues directement dans le fichier OBJ, le reste du travail est extrêmement simple pour appliquer une texture sur le modèle. Nous ajoutons les coordonnées de texture au maillage géométrique (classe Mesh) et nous utilisons la classe Texture pour

charger les images voulues comme pour les surfaces. Nous avons utilisé cette technique pour les arbres présent dans la scène de notre projet. Plusieurs photos de ces arbres sont déjà présentes dans le document dans les fonctionnalités entourant les techniques d'illumination.

4.2 Shader de fragment

Nous avons développé plusieurs familles de shader de fragment pour permettre de faire le rendu de plusieurs effets supplémentaire par rapport au premier travail pratique. Il y a, tout d'abord, trois shaders de fragment qui sont utilisés pour faire l'illumination des divers objets

```
in vec3 fragColor;
in vec2 fragTexCoord;
in vec3 fragNormal;
in vec3 fragPos;
in vec3 fragTangente;

struct LumiereDirectionnelle {
    vec3 direction;
    vec3 ambiane;
    vec3 diffuse;
    vec3 speculaire;
};

struct Projecteur {
    vec3 position;
    vec3 direction;
    vec3 diffuse;
    vec3 speculaire;

    float coneExterne;
    float coneInterne;
};

struct Ponctuelle {
    vec3 position;
    vec3 ambiane;
    vec3 diffuse;
    vec3 speculaire;

    float constante;
    float lineaire;
    float quadratique;
};

#define NB_PROJECTEURS_MAX 10
#define NB_LUMIERES_PONCTUELLES_MAX 10

uniform LumiereDirectionnelle lumDirectionnelle;
uniform int nbProjecteurs;
uniform Projecteur projecteur[NB_PROJECTEURS_MAX];
uniform int nbPonctuelles;
uniform Ponctuelle ponctuelles[NB_LUMIERES_PONCTUELLES_MAX];
uniform sampler2D diffuseMap;
uniform sampler2D specularMap;
uniform sampler2D normalMap;
uniform vec3 positionCamera;
```

de la scène. Ils commencent tous par une série de structures qui servent à définir les propriétés de tous les types de lumière que ces shaders sont capable de calculer. On envoie en variable uniforme des tableaux contenant toutes les informations sur ces structures. La quantité de lumières ponctuelles et de projecteurs qui seront utilisés pour illuminer les fragments sera envoyée par deux autres variables uniformes (nbProjecteurs et nbPonctuelles). Ainsi, il est possible de faire varier dynamiquement le nombre de lumière dans la scène. Nous avons choisis de toujours utiliser une lumière directionnelle pour donner l'impression qu'il y a un soleil puisque la scène est à l'extérieur.

Pour illuminer les fragments, nous avons définis trois fonctions différentes qui prennent un type de lumière en entrée et renvoie un vecteur à trois dimensions contenant le fragment illuminé.

Selon la méthode choisis pour donner une couleur ou une texture à un objet, nous utilisons deux techniques différentes dans

les shaders de fragment. Dans l'exemple ci-dessus, nous utilisons trois textures qui détermineront la couleur des différents types de réflexion. Ces textures sont envoyées en variable uniform ce qui permet de les faire varier selon l'objet à rendre. Nous expliquerons l'utilité de ces différentes textures dans les prochaines sections. Ensuite, il est aussi possible d'envoyer les caractéristiques de réflexion du materiau à rendre à travers une structure envoyée en variable uniforme. Donc, au lieu d'envoyer une texture, on envoie la couleur des réflexions diffuses, spéculaires et la brillance du matériau. Ainsi, en faisant varier ces caractéristiques, il est possible de donné un aspect particulier (or, argent, plastique, etc.) à chacun des objets de la scène.

Voici un exemple du programme principal de ces shaders:

```
vec3 calculerIlluminationDirectionnelle(LumiereDirectionnelle lumiere, vec3 normal, vec3 directionCamera);
vec3 calculerProjecteur(Projecteur proj, vec3 normal, vec3 directionCamera);
vec3 calculerPonctuelle(Ponctuelle lumiere, vec3 normal, vec3 directionCamera);
vec3 calculerNormal();

void main()
{
    vec3 normal = calculerNormal();
    vec3 directionCamera = normalize(positionCamera - fragPos);

    vec3 resultatDir = calculerIlluminationDirectionnelle(lumiere, normal, directionCamera);
    vec3 resultatProj;
    for(int i = 0; i < nbProjecteurs; i++) {
        resultatProj += calculerProjecteur(projecteur[i], normal, directionCamera);
    }
    vec3 resultatPonctuelles;
    for(int i = 0; i < nbPonctuelles; i++) {
        resultatPonctuelles += calculerPonctuelle(ponctuelles[i], normal, directionCamera);
    }

    color = vec4(resultatDir + resultatProj + resultatPonctuelles, 1.0f);
}
```

Nous avons ensuite une série de shaders de fragments qui servent à faire du «post processing» sur une texture. Nous les utilisons pour faire des effets de filtres sur la fenêtre principale. Le fonctionnement de ces shaders sera expliqué dans la section **7.5 effets plein écran**.

Finalement, nous avons quelques shaders très simples qui servent à donner une couleur particulière à un fragment ou à simplement appliquer la couleur du sommet au fragment.

5.2 Lumière directionnelle

Pour donner une illumination de base à notre scène, nous avons implémenté une lumière directionnelle. Elle est définie dans la classe Lumière avec des propriétés encapsulées dans une structure. Celle-ci contient la direction de la lumière (par défaut alignée avec le soleil du cubemap), la couleur ambiante, diffuse et spéculaire.

```
struct LumiereDirectionnelle {
    ofVec3f direction;
    ofVec3f ambiante;
    ofVec3f diffuse;
    ofVec3f speculaire;
};
```

Ces valeurs sont envoyées au différents shaders de fragment par variables uniformes lors de l'appel de la méthode chargerValeursIlluminationUniforms(programId) de la classe Lumière. Contrairement aux lumières ponctuelles et aux projecteurs, nous avons décidé de permettre d'avoir seulement une lumière directionnelle

dans la scène. Donc, les propriétés ne sont envoyées qu'une seule fois au shader de fragment.

Pour calculer l'illumination directionnelle à l'intérieur du shader, on appelle la fonction

```
vec3 calculerIlluminationDirectionnelle(LumiereDirectionnelle lumiere, vec3 normal, vec3 directionCamera)
{
    vec3 directionLumiere = normalize(lumiere.direction);

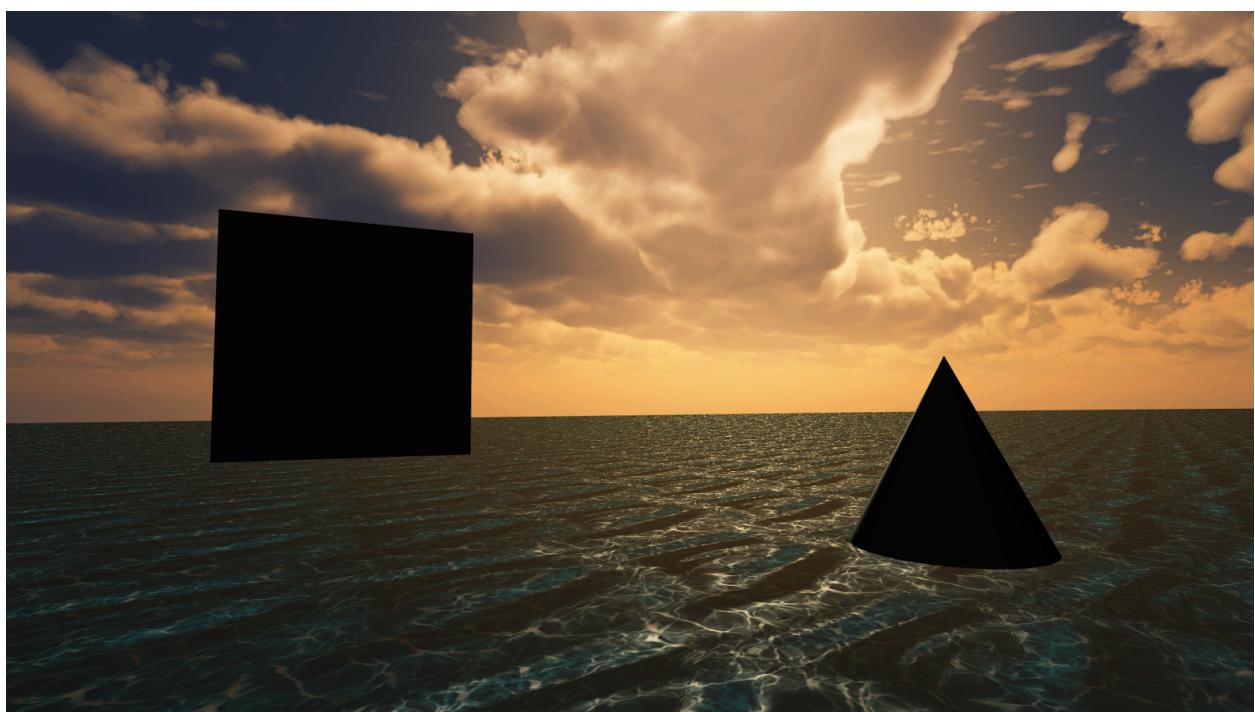
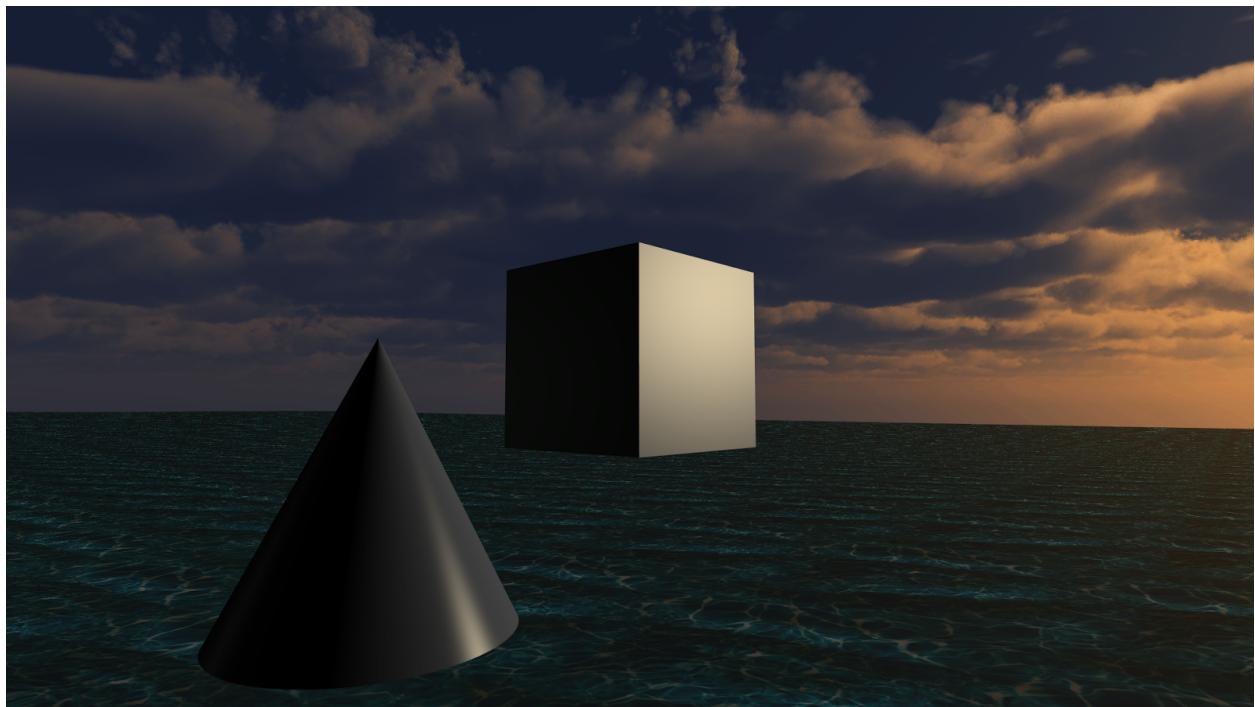
    // Lumière ambiante
    vec3 ambient = lumiere.ambiante * vec3(texture(diffuseMap, fragTexCoord));

    // Reflection diffuse
    float diff = max(dot(normal, directionLumiere), 0.0);
    vec3 diffuse = lumiere.diffuse * diff * vec3(texture(diffuseMap, fragTexCoord));

    // Reflection spéculaire
    vec3 directionReflection = reflect(-directionLumiere, normal);
    float spec = pow(max(dot(directionCamera, directionReflection), 0.0), 2);
    vec3 specular = lumiere.speculaire * spec * vec3(texture(specularMap, fragTexCoord));

    return (ambient + diffuse + specular);
}
```

calculerIlluminationDirectionnelle(...) avec la structure représentant la lumière directionnelle, la normale du fragment et la position de la caméra. Avec ces informations, la fonction calcul la couleur ambiante, diffuse et les réflexions spéculaires sur le fragment et renvoie la somme des couleurs obtenues. À la page suivante, vous verrez un exemple d'illumination directionnelle dans notre projet avec une vue sur le côté et derrière des primitives illuminées:



5.3 Lumière ponctuelle

Pour cette fonctionnalité, nous avons définis une structure dans la classe Lumiere qui contient toutes les caractéristiques d'une instance d'une lumière ponctuelle dont les constantes

```
struct LumierePonctuelle {
    ofVec3f position;
    ofVec3f ambiante;
    ofVec3f diffuse;
    ofVec3f speculaire;

    float constante;
    float lineaire;
    float quadratique;
};
```

d'atténuation et les couleurs des réflexions. Cette classe contient une liste de LumierePonctuelle dont les caractéristiques seront envoyées au shader de fragment actif en variables uniformes lors de l'appel de la méthode chargerValeursIlluminationUniforms(programId). On envoie tout d'abord le nombre de lumière ponctuelle. Le shader se servira de cette information pour savoir combien d'instances de lumière ponctuelle il y a dans la scène et donc combien de structures il y a dans son tableau de lumière ponctuelle. À l'intérieur du shader, pour chacune des lumières, on appelle la fonction calculerPonctuelle qui prend en argument la lumière à

```
vec3 calculerPonctuelle(Ponctuelle lumiere, vec3 normal, vec3 directionCamera)
{
    vec3 directionLumiere = normalize(lumiere.position - fragPos);

    float diff = max(dot(normal, directionLumiere), 0.0);

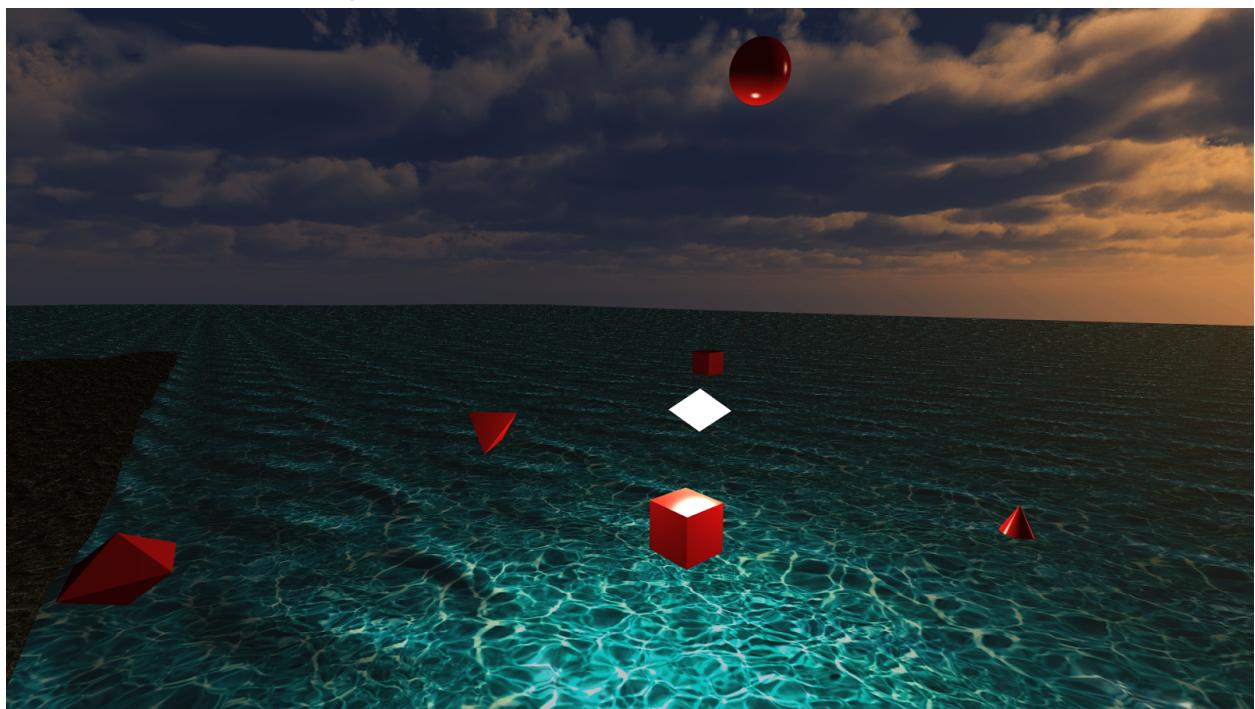
    vec3 directionReflection = reflect(-directionLumiere, normal);
    float spec = pow(max(dot(directionCamera, directionReflection), 0.0), 2);

    // Atténuation
    float distance      = length(lumiere.position - fragPos);
    float attenuation = 1.0f / (lumiere.constante + lumiere.lineaire * distance +
                                lumiere.quadratique * (distance * distance));

    // Résultat combiné
    vec3 ambiante = lumiere.ambiante * vec3(texture(diffuseMap, fragTexCoord));
    vec3 diffuse  = lumiere.diffuse * diff * vec3(texture(diffuseMap, fragTexCoord));
    vec3 speculaire = lumiere.speculaire * spec * vec3(texture(specularMap, fragTexCoord));
    ambiante *= attenuation;
    diffuse *= attenuation;
    speculaire *= attenuation;
    return (ambiante + diffuse + speculaire);
}
```

évaluer, la normale du fragment et la direction de la caméra. On fait ensuite une série de calculs pour trouver la position du fragment par rapport à la lumière, la direction des réflexions et le facteurs d'atténuation selon la distance. On combine ensuite la couleur de la lumière avec la couleur du fragment en utilisant la bonne texture et en utilisant les facteurs calculés (l'utilité des différentes texture sera expliquée à la fonctionnalité 6.3). Avant, de combiner le tout, on multiplie tous les types d'illumination par le facteur d'atténuation. Le vecteur retourné correspond à la couleur du fragment illuminé par l'instance de la lumière passée en argument.

On peut voir le résultat dans la photo suivante. L'octaèdre blanc correspond à la source de la lumière et les autres primitives géométriques sont disposés pour qu'on voit l'atténuation de la lumière et les réflexions spéculaires.



5.4 Projecteur

Nous avons implanté les projecteurs presque de la même manière que les lumières ponctuelles et la lumière directionnelle. Nous avons tout d'abord créer une structure Projecteur dans la classe Lumière. Cette structure contient la grandeur des cônes de lumière

```
struct Projecteur {
    ofVec3f direction;
    ofVec3f position;
    ofVec3f diffuse;
    ofVec3f speculaire;

    float coneInterne;
    float coneExterne;
};
```

(interne et externe), la position, la direction et les couleurs des réflexions d'un instance de projecteur. Comme pour les lumières ponctuelles, la classe Lumière contient une liste des tous les projecteurs actif de la scène et elle envoie les caractéristiques de chacun d'entre eux au shader de fragment actif lors de l'appel de la méthode chargerValeursIlluminationUniforms(programId). On envoie encore une fois le nombre de projecteurs et les caractéristiques de chacun d'entre eux pour que le shader sache le nombre de calcules qu'il devra faire.

À l'intérieur du shader de fragment, on a un autre tableau dont la grandeur est déterminée par le nombre de projecteur envoyé en valeur uniforme. Pour chacun des éléments du tableau, on appelle la fonction calculerProjecteur qui prend comme argument une instance de projecteur, la normale du fragment et la direction de la caméra. Au tout début de cette

```
vec3 calculerProjecteur(Projecteur proj, vec3 normal, vec3 directionCamera)
{
    vec3 directionLumiere = normalize(proj.position - fragPos);
    float theta = dot(directionLumiere, normalize(-proj.direction));
    float epsilon = proj.coneInterne - proj.coneExterne;
    float intensite = clamp((theta - proj.coneExterne) / epsilon, 0.0, 1.0);

    float diff = max(dot(normal, directionLumiere), 0.0f);
    vec3 diffuse = proj.diffuse * diff * vec3(texture(diffuseMap, fragTexCoord));

    vec3 directionVue = normalize(positionCamera - fragPos);
    vec3 directionReflection = reflect(-directionLumiere, normal);
    float spec = pow(max(dot(directionVue, directionReflection), 0.0), 2);
    vec3 specular = proj.speculaire * (spec * vec3(texture(specularMap, fragTexCoord)));

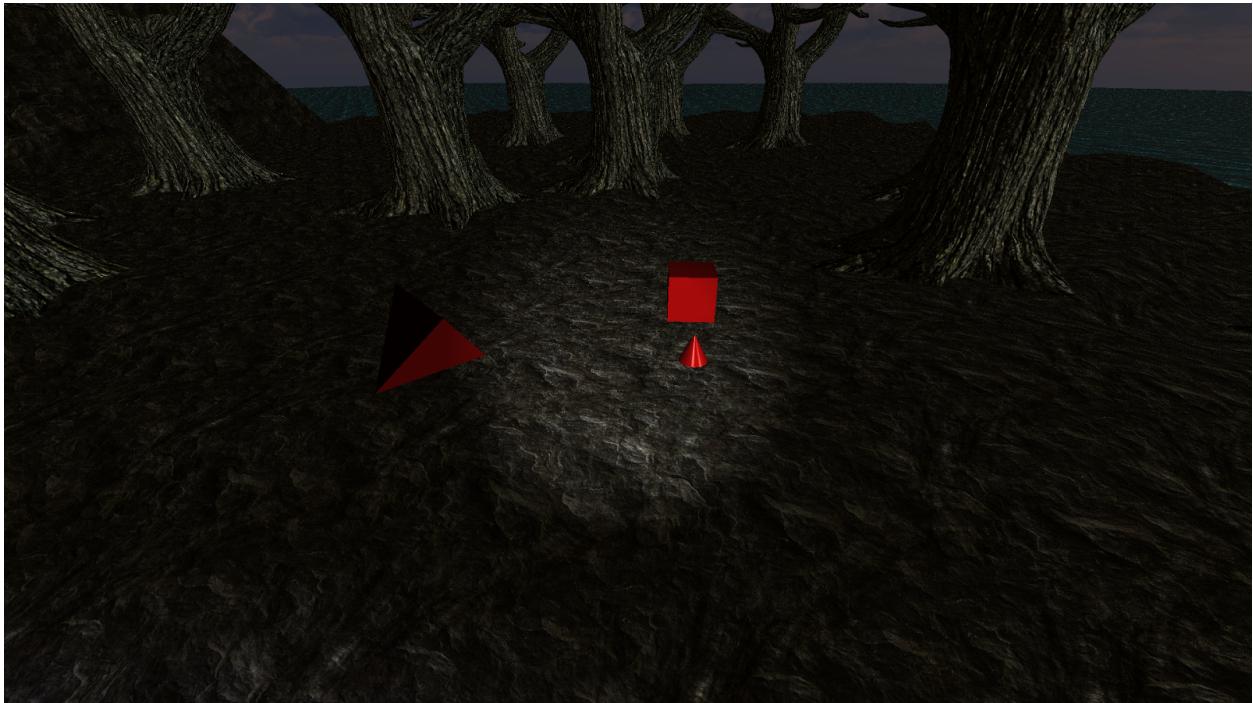
    diffuse *= intensite;
    specular *= intensite;

    return (diffuse + specular);
}
```

fonction, on fait une série de calcules pour déterminer la position du fragment par rapport à la position et la direction du projecteur. On calcule aussi l'intensité de la lumière selon la position du fragment par rapport au cône interne et externe du projecteur. Si le fragment est en face du cône interne, il sera illuminé à pleine puissance. S'il se trouve entre la limite du cône interne et la limite du cône externe, il sera partiellement illuminé dépendant de sa distance par rapport à la limite du cône interne. Finalement, s'il est à l'extérieur des deux cônes, il ne sera pas illuminé. Le reste des calcules d'illumination est très similaire à ceux des lumières

ponctuelles, mais au lieu de multiplier les couleurs de réflections par un facteur d'atténuation, on les multiplie par l'intensité calculée au début de la fonction. Le vecteur retourné correspond à la couleur du fragment illuminé par le projecteur passé en argument.

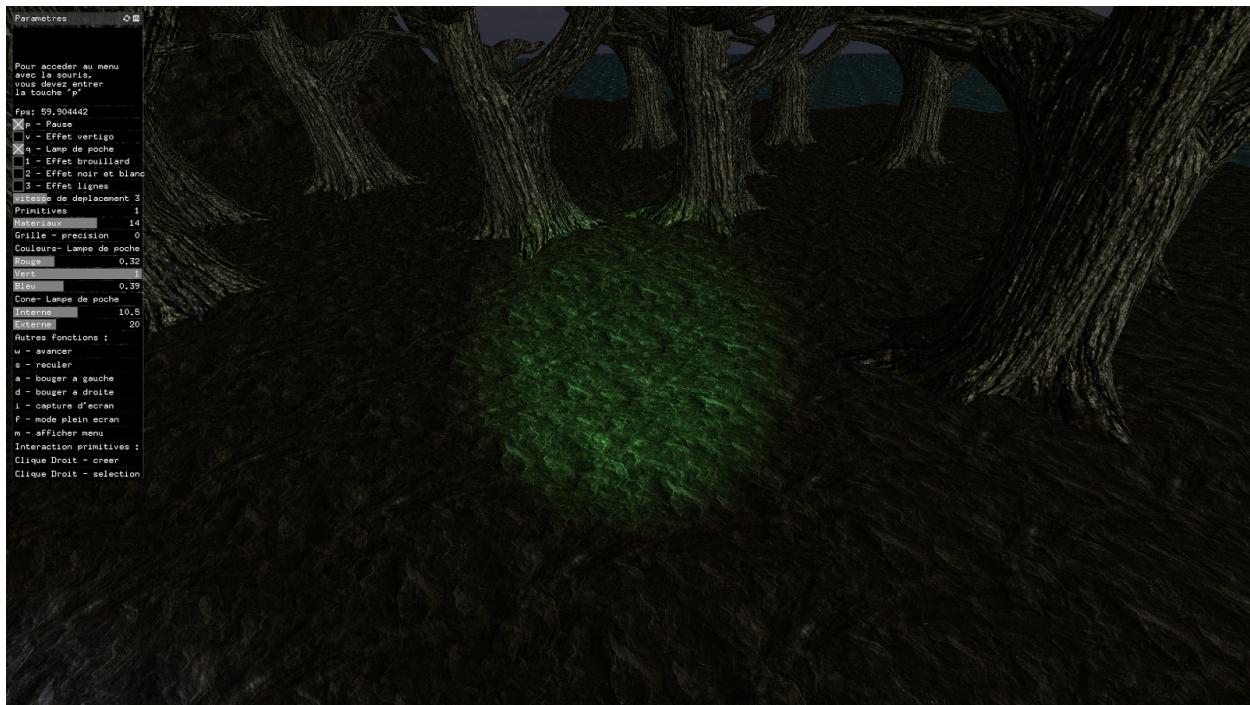
La photo suivante montre un exemple de projecteur dans notre projet. Le projecteur est positionné à la même position que la caméra et projette dans la même direction que celle-ci. On voit bien la quantité d'illumination dépendant de la position de l'objet par rapport aux cônes du projecteur.



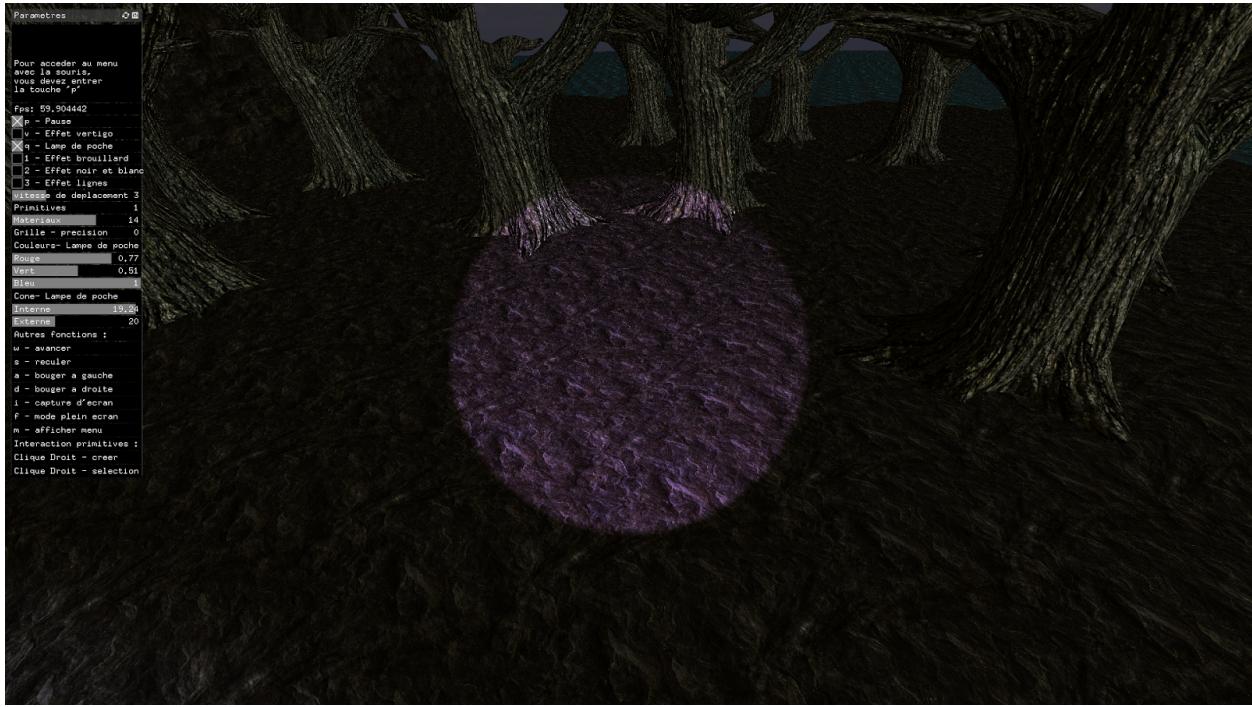
5.5 Lumières éditables

Pour permettre à l'utilisateur d'éditer une lumière, nous avons mis une série de curseurs dans l'interface graphique qui permettent de modifier toutes les propriétés de la lampe de poche (le projecteur à la position de la caméra). Premièrement, cette lumière est activable avec la touche «q» sur le clavier. Sa position et sa direction est mise à jour continuellement pour que la lumière suive les mouvements de la caméra. Ensuite, nous avons mis une première série de curseurs qui permettent de modifier la couleur de la lumière en RGB. Le changement de couleur se fait en temps réel pour que l'utilisateur puisse facilement ajuster la couleur selon ses préférences. La photo suivante montre l'effet d'un changement de couleur.

Couleurs- Lampe de poche	
Rouge	0.8
Vert	0.8
Bleu	0.8
Cone- Lampe de poche	
Interne	10.5
Externe	20



Nous avons aussi ajouté deux autres curseur pour que l'utilisateur puisse modifier la grandeur des cônes de la lampe de poche. Il peut donc choisir la grandeur du cône et l'importance du cône externe par rapport au cône interne. Si on met les cônes à la même grandeur, on obtient un projecteur dont les limites sont très claires (un peu surréaliste comme effet!). Voici un exemple:



En ajoutant ces curseurs à l'interface graphique, nous avons fait une découverte intéressante. Si l'utilisateur choisit de mettre le cône externe plus petit que le cône interne, les calculs d'illumination font en sorte que toute la scène est illuminée par le projecteur sauf le cône externe. Ce qui donne un effet assez impressionnant! Nous avons choisis de laisser cette possibilité à l'utilisateur puisque l'effet fonctionne bien et que l'idée d'un anti-projecteur est assez intéressante! Voici un exemple:



Finalement, toutes les lumières de la scène sont facilement éditables par programmation. Les options disponibles dans l'interface graphiques ne sont qu'une petite démonstration des possibilités qu'offre notre projet.

6.2 Matériaux avec effet de relief

Afin d'ajouter un effet de profondeur et de relief à certains objets de la scène, nous avons décidé de faire du «normal mapping». Pour réussir cet effet, nous avons ajouter à nos attributs de sommets la tangente de chacun d'entre eux. Alors, lorsqu'on crée un maillage géométrique avec notre classe Mesh, nous avons ajouté une

```
ofVec3f Mesh::calculerTangentePourSommets(Vertex& v0, Vertex& v1, Vertex& v2)
{
    ofVec3f arete1 = ofVec3f(v1.x, v1.y, v1.z) - ofVec3f(v0.x, v0.y, v0.z);
    ofVec3f arete2 = ofVec3f(v2.x, v2.y, v2.z) - ofVec3f(v0.x, v0.y, v0.z);

    float deltaU1 = v1.texCoordX - v0.texCoordX;
    float deltaV1 = v1.texCoordY - v0.texCoordY;
    float deltaU2 = v2.texCoordX - v0.texCoordX;
    float deltaV2 = v2.texCoordY - v0.texCoordY;

    float f = 1.0 / (deltaU1 * deltaV2 - deltaU2 * deltaV1);

    ofVec3f tangente;

    tangente.x = f * (deltaV2 * arete1.x - deltaV1 * arete2.x);
    tangente.y = f * (deltaV2 * arete1.y - deltaV1 * arete2.y);
    tangente.z = f * (deltaV2 * arete1.z - deltaV1 * arete2.z);

    return tangente.normalize();
}
```

```
struct Vertex {
    Vertex()
        :x(0), y(0), z(0),
         colorR(0), colorG(0), colorB(0),
         texCoordX(0), texCoordY(0),
         normalX(0), normalY(0), normalZ(0)
    {}

    GLfloat x;
    GLfloat y;
    GLfloat z;

    GLfloat colorR;
    GLfloat colorG;
    GLfloat colorB;

    GLfloat texCoordX;
    GLfloat texCoordY;

    GLfloat normalX;
    GLfloat normalY;
    GLfloat normalZ;

    GLfloat tangenteX;
    GLfloat tangenteY;
    GLfloat tangenteZ;
};
```

fonction qui permet de calculer la tangente pour chacun des triangles. Cette fonction est appellée lors de l'initialisation des attributs de sommets du maillage. Nous avons alors ajouté cet attribut au VBO ce qui nous permet de récupérer et de manipuler la tangente dans nos shaders.

Dans le shader de fragment, nous avons ajouter une fonction qui calcule la matrice de l'espace tangent pour la normale et la tangente du fragment. Avec cette matrice et une texture de normal mapping passée en variable uniforme, on fait varier la direction de la normale et on réussi à donner un effet de profondeur assez intéressant à certaines surfaces.

```
vec3 calculerNormal()
{
    vec3 normal = normalize(fragNormal);
    vec3 tangente = normalize(fragTangente);
    tangente = normalize(tangente - dot(tangente, normal) * normal);
    vec3 biTangente = cross(tangente, normal);

    vec3 normalTexture = vec3(texture(normalMap, fragTexCoord));
    normalTexture = 2.0 * normalTexture - vec3(1.0);

    mat3 espaceTangent = mat3(tangente, biTangente, normal);
    vec3 normalAjustee = espaceTangent * normalTexture;

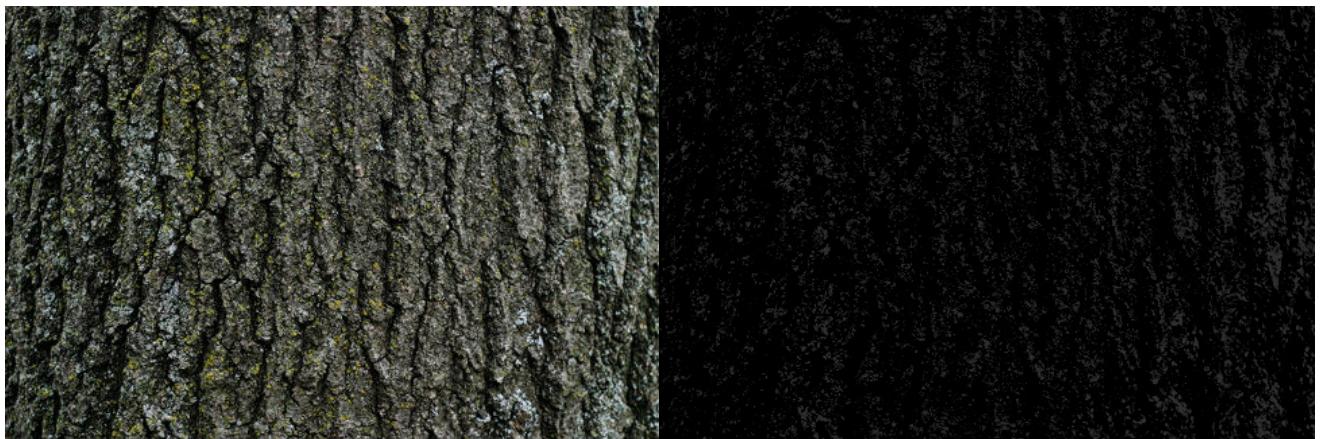
    return normalize(normalAjustee);
}
```

Nous avons utilisé cette techniques sur toutes les surfaces et sur les arbres qui sont présent dans la scène. L'effet est assez remarquable sur la surface de l'eau puisque nous faisons varier les coordonnées de textures de celle-ci ce qui fait un mouvement intéressant dans la scène. Voici un exemple:



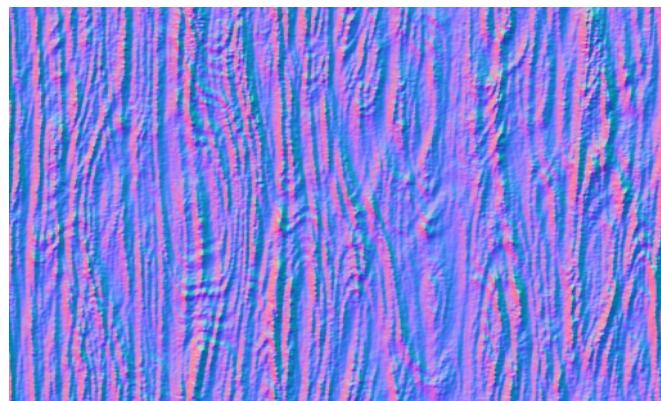
6.3 Matériaux à textures multiples

Pour réussir à faire des effets intéressants sur certaines surfaces, nous avons dû utiliser plusieurs couches de textures. Dans les fonctionnalités portant sur l'éclairage, on remarque que les shaders de fragment utilisent souvent une texture nommée «diffuseMap» et une autre nommée «specularMap». Ces deux textures permettent de donner un aspect un peu plus réaliste à certaines surfaces puisque nous avons la possibilité de déterminer quelle partie de celles-ci sont réfléchissante. Donc, la «diffuseMap» est une texture qui détermine la couleur de base de l'objet. Elle est utilisée dans les calculs d'éclairage ambiant et diffus pour être combinée avec la couleur ambiant et diffuse de la lumière. La «specularMap» quant à elle est une texture en ton de gris qui détermine quelle partie de l'objet est réfléchissante. Donc, en combinant la couleur d'une réflexion spéculaire d'une lumière et un fragment de la «specularMap», on obtient une réflexion qui dépend de la quantité de blanc présent dans l'échantillon de texture. Par exemple, voici les textures utilisées pour les arbres de la scène:



Puisque le bois n'est pas très réfléchissant, la texture en ton de gris est assez foncé. Les réflexions spéculaires sur les arbres sont donc assez subtiles.

En plus de ces deux textures, nous ajoutons la «normalMap» pour donner un effet de profondeur supplémentaire. Voici la texture utilisée sur les arbres:



En combinant ces trois textures, nous obtenons des effets assez intéressant. Nous avons déjà présenté la surface de l'eau dans la fonctionnalité précédante. Elle utilise elle aussi la combinaisons de ces trois types de textures. Voici le rendu final d'un arbre dans notre projet:



7.5 Effets plein écran

La technique utilisée afin de réaliser des effets plein écran consiste à rendre la scène dans un framebuffer ‘hors écran’ et d’attacher une texture à celui ci. Après avoir rendu ‘hors écran’, un quadrilatère de la grandeur de l’écran est rendu dans le framebuffer effectif pour le rendu à l’écran auquel on applique la texture issue de la scène.

L’avantage de cette méthode est que le rendu de la scène n’est aucunement affecté. On continue de rendre la scène de la même manière qu’avant, avec les mêmes shaders de fragments. Une fois rendue sous forme de texture, on peut donc appliquer un ‘post-processing’ directement sur les pixels dans une seconde passe de rendu.

Concrètement, nous avons créé une classe qui permet de gérer le framebuffer ‘hors écran’ et la texture associée à ce framebuffer. À la création de l’application, ce framebuffer est créé.

Puis, à chaque itération de dessin nous effectuons les étapes suivantes:

1. Activation du framebuffer ‘hors-écran’
2. Rendu de la scène
3. Désactivation du framebuffer ‘hors-écran’ et activation du framebuffer par défaut.
4. Rendu du quadrilatère avec la texture de la scène.

L’image suivante montre le code relié à la boucle de dessin de notre application. Les étapes 1, 3 et 4 sont surlignées en jaune.

```

void ofApp::draw(){
    fbo.bind();
    {
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glEnable(GL_DEPTH_TEST);
        glEnable(GL_LIGHTING);
        camera.lookAt(view);
        lumiere.setPositionVue(camera.getPosition());

        pushMatrix();
        model.glTranslate(camera.getPosition().x, camera.getPosition().y, camera.getPosition().z);
        cubeMap.afficher(projection, model, view);
        popMatrix();

        graphScene->afficher(&projection, &view, &lumiere, camera.getOrientation(), camera.getPosition());

        origineDuMonde.afficher(projection, model, view);
        paysage.afficher(projection, model, view, lumiere);

        pushMatrix();
        glUseProgram(shaderLampe.getProgramID());
        model.glTranslate(positionLampe.getPosition());
        glUniformMatrix4fv(glGetUniformLocation(shaderLampe.getProgramID(), "model"), 1, GL_FALSE, model.getPtr());
        glUniformMatrix4fv(glGetUniformLocation(shaderLampe.getProgramID(), "view"), 1, GL_FALSE, view.getPtr());
        glUniformMatrix4fv(glGetUniformLocation(shaderLampe.getProgramID(), "projection"), 1, GL_FALSE, projection.getPtr());
        positionLampe.afficher();
        popMatrix();
        pushMatrix();
        model.glTranslate(positionPonctuelle.getPosition());
        glUniformMatrix4fv(glGetUniformLocation(shaderLampe.getProgramID(), "model"), 1, GL_FALSE, model.getPtr());
        positionPonctuelle.afficher();
        glUseProgram(0);
        popMatrix();
    }
    fbo.unbind();

    if(effetBrouillard) {
        effetPleinEcran.activerEffetBrouillard();
    } else if(effetNoirEtBlanc) {
        effetPleinEcran.activerEffetNoirEtBlanc();
    } else if(effetLignes) {
        effetPleinEcran.activerEffetLignes();
    } else {
        effetPleinEcran.desactiverEffet();
    }

    effetPleinEcran.afficher(fbo.getColorTexture());

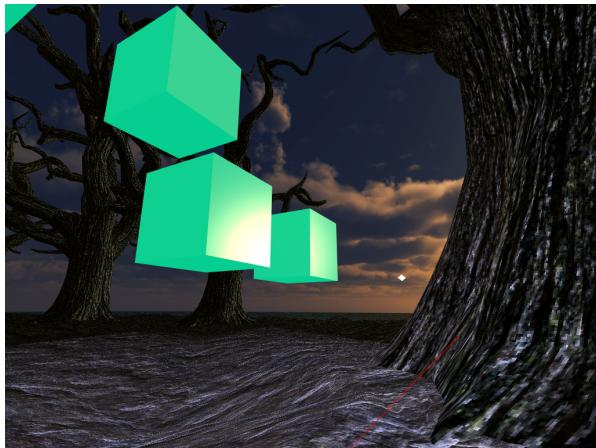
    glDisable(GL_LIGHTING);
    glDisable(GL_DEPTH_TEST);
    if (showMenu) {
        gui.draw();
    }
}

```

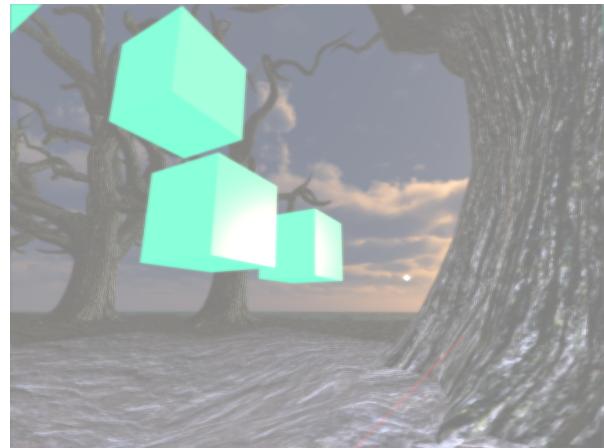
L'effet désiré est appliqué à l'étape 4 à l'aide de shader de fragments appropriés. Dans notre cas, nous avons créé une classe qui s'occupe d'activer le bon shader de fragment par rapport à l'effet choisi.

Les effets de ‘post-processing’ que nous avons décidé de réaliser sont (voir les trois images qui suivent):

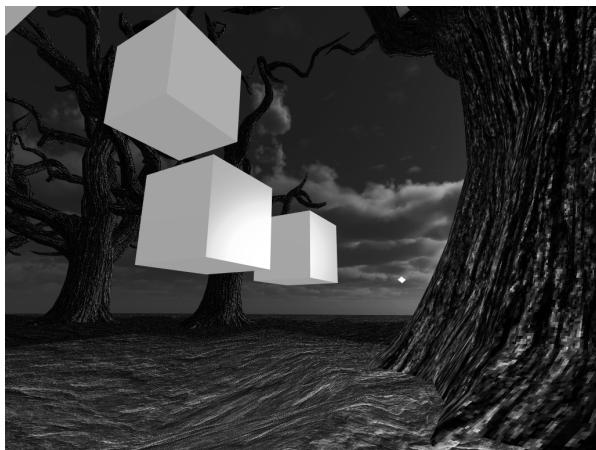
- Effet brouillard
- Effet noir et blanc
- Effet contour de lignes



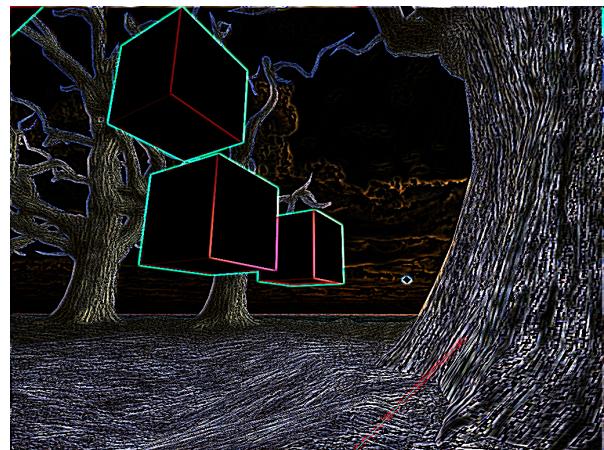
1. Scène normale



2. Effet brouillard



3. Effet noir et blanc



4. Effet contour de lignes

Comme nous avons 3 effets possibles, nous avons 3 shaders de fragments associés. Le plus complexe de ceux-ci est le shader pour l'effet contour de lignes. La méthode n'est pas très compliquée, il s'agit d'utiliser un noyau 3 par 3 pour traiter chaque pixel. En regardant les pixels environnant, nous allons échantillonner les couleurs avoisinantes et appliquons une transformation à celles-ci dépendamment de leur valeur. Le shader complet est montré ci-après. On peut y voir notamment comment on échantillonne les valeurs de pixels par rapport au noyau.

```

1 #version 430 core
2
3 layout (location = 0) out vec4 color;
4 uniform sampler2D screenTexture;
5
6 in vec2 texcoords;
7
8 const float offset = 1.0 / 300;
9
10 void main()
11 {
12     vec2 offsets[9] = vec2[] (
13         vec2(-offset, offset), // top-left
14         vec2(0.0f, offset), // top-center
15         vec2(offset, offset), // top-right
16         vec2(-offset, 0.0f), // center-left
17         vec2(0.0f, 0.0f), // center-center
18         vec2(offset, 0.0f), // center-right
19         vec2(-offset, -offset), // bottom-left
20         vec2(0.0f, -offset), // bottom-center
21         vec2(offset, -offset) // bottom-right
22     );
23
24     float kernel[9] = float[] (
25         1.0, 1.0, 1.0,
26         1.0, -8.0, 1.0,
27         1.0, 1.0, 1.0
28     );
29
30     vec3 col;
31     for(int i = 0; i < 9; i++) {
32         vec3 sampling = vec3(texture(screenTexture, texcoords.st + offsets[i]));
33         col += sampling * kernel[i];
34     }
35
36     color = vec4(col, 1.0);
37 }

```

8.5 Interactivité par pointage

L'interactivité par pointage la plus présente dans notre application est bien entendu la possibilité de changer l'orientation de la caméra à l'aide des mouvements de la souris. À moins d'être en mode pause, qui permet l'interaction avec le menu par clics de souris, chaque déplacement de la souris met à jour l'orientation de la caméra.

Nous avons programmé la classe MousePositionHandler qui sert à récupérer les déplacements de la souris. Cette classe recueille les informations sur la position de la souris avant chaque rendu par la fonction update. On peut par la suite appeler les fonctions getRelPosX et getRelPosY pour obtenir la position relative au dernier « update » en X et en Y, c'est-à-dire le déplacement de la souris depuis le dernier rendu. La caméra appelle également une fonction update qui met à jour sa position et son orientation. Sa nouvelle orientation se sert des positions relatives obtenues par une instance de la classe MousePositionHandler.

```
void Camera::orienter()
{
    // Récupération des angles
    m_phi += -mouseHandler->getRelPosY() * m_sensibilite;
    m_theta += -mouseHandler->getRelPosX() * m_sensibilite;

    // Limitation de l'angle phi
    if(m_phi > 89.0)
        m_phi = 89.0;

    else if(m_phi < -89.0)
        m_phi = -89.0;

    // Conversion des angles en radian
    float phiRadian = m_phi * PI / 180;
    float thetaRadian = m_theta * PI / 180;

    // Calcul des coordonnées sphériques
    if(m_axeVertical.x == 1.0)// Si l'axe vertical est l'axe X
    {
        m_orientation.x = sin(phiRadian);
        m_orientation.y = cos(phiRadian) * cos(thetaRadian);
        m_orientation.z = cos(phiRadian) * sin(thetaRadian);
    }else if(m_axeVertical.y == 1.0)// Si c'est l'axe Y
    {
        m_orientation.x = cos(phiRadian) * sin(thetaRadian);
        m_orientation.y = sin(phiRadian);
        m_orientation.z = cos(phiRadian) * cos(thetaRadian);
    }else// Sinon c'est l'axe Z
    {
        m_orientation.x = cos(phiRadian) * cos(thetaRadian);
        m_orientation.y = cos(phiRadian) * sin(thetaRadian);
        m_orientation.z = sin(phiRadian);
    }

    //Calcul de la normale, retour le vecteur orthogonal et le normalise (le rend de longueur 1)
    m_deplacementLateral = m_axeVertical.getCrossed(m_orientation);
    m_deplacementLateral.normalize();

    //Calcul du point ciblé
    m_pointCible = m_position + m_orientation;
}
```

Avec un clic de souris, l'utilisateur peut sélectionner des primitives qu'il a créées dans sa scène et même les déplacer en bougeant sa souris durant sa sélection. Les détails sur la sélection sont expliqués au point 1.5. Il peut également, lorsqu'il est en mode pause, interagir avec le menu de l'application pour changer une multitude de paramètre. L'un des paramètres est « lampe de poche ». Lorsqu'activé, l'utilisateur peut éclairer les zones voulu se déplaçant

à l'aide des touches clavier et en orientant la lumière projeté avec les mouvements de la souris.

Extra: Anti Aliasing

Afin d'améliorer l'apparence général du rendu, nous avons ajouté de l'anti aliasing à notre projet. Puisque nous faisons le rendu de l'image en deux passes, nous avons ajusté le framebuffer hors écran (off screen) pour qu'il rende la scène dans une texture à multi-échantillons. Ainsi, nous pouvons rendre la scène avec 8 échantillons par pixel au lieu

```
void Framebuffer::generateColorTexture(){
    glGenTextures(1, &texture_color);
    glBindTexture(GL_TEXTURE_2D_MULTISAMPLE, texture_color);
    glTexImage2DMultisample(GL_TEXTURE_2D_MULTISAMPLE, NUM_SAMPLES, GL_RGB, width, height, GL_TRUE);
    glBindTexture(GL_TEXTURE_2D_MULTISAMPLE, 0);
}

void Framebuffer::generateFBO(unsigned int width, unsigned int height){
    this->width = width;
    this->height = height;

    glGenFramebuffers(1, &FBO);
    glBindFramebuffer(GL_FRAMEBUFFER, FBO);

    generateColorTexture();

    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D_MULTISAMPLE, texture_color, 0);

    glGenRenderbuffers(1, &rbo);
    glBindRenderbuffer(GL_RENDERBUFFER, rbo);
    glRenderbufferStorageMultisample(GL_RENDERBUFFER, NUM_SAMPLES, GL_DEPTH24_STENCIL8, width, height);
    glBindRenderbuffer(GL_RENDERBUFFER, 0);

    glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_STENCIL_ATTACHMENT, GL_RENDERBUFFER, rbo);

    if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE){
        std::cout << "Erreur: Le framebuffer n'est pas complete." << std::endl;
        std::cin.get();
        std::terminate();
    }
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
    generateIntermediateMultiSampleFBO();
}
```

de un seul. L'essentiel du code pour créer et attacher la texture multi-échantillons est présenté dans l'exemple ci-dessus.

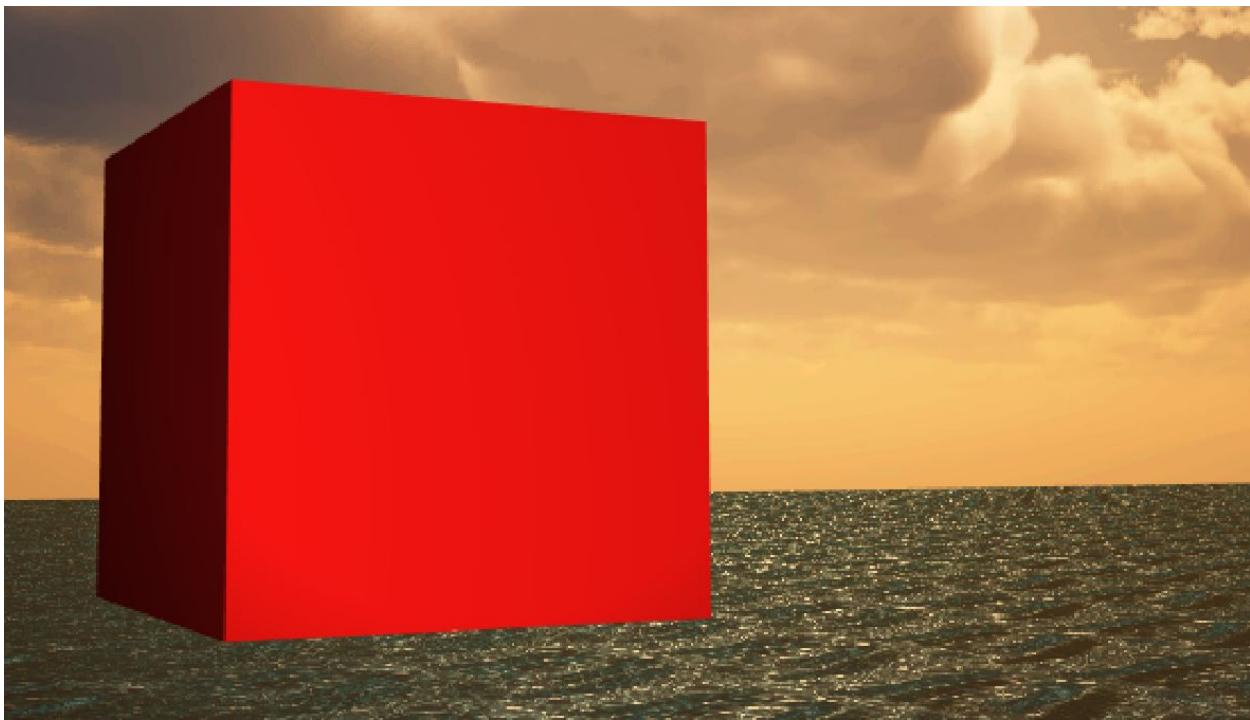
Par contre, pour réussir à rendre nos effet plein écran, nous avons dû ajouter un framebuffer supplémentaire pour faire la conversion entre le framebuffer multi-échantillonné et le framebuffer par défaut dans lequel l'image finale est rendu. Donc, nous attachons le

```
void Framebuffer::unbind(){
    glBindFramebuffer(GL_READ_FRAMEBUFFER, FBO);
    glBindFramebuffer(GL_DRAW_FRAMEBUFFER, intermediateFBO);
    glBlitFramebuffer(0, 0, width, height, 0, 0, width, height, GL_COLOR_BUFFER_BIT, GL_NEAREST);

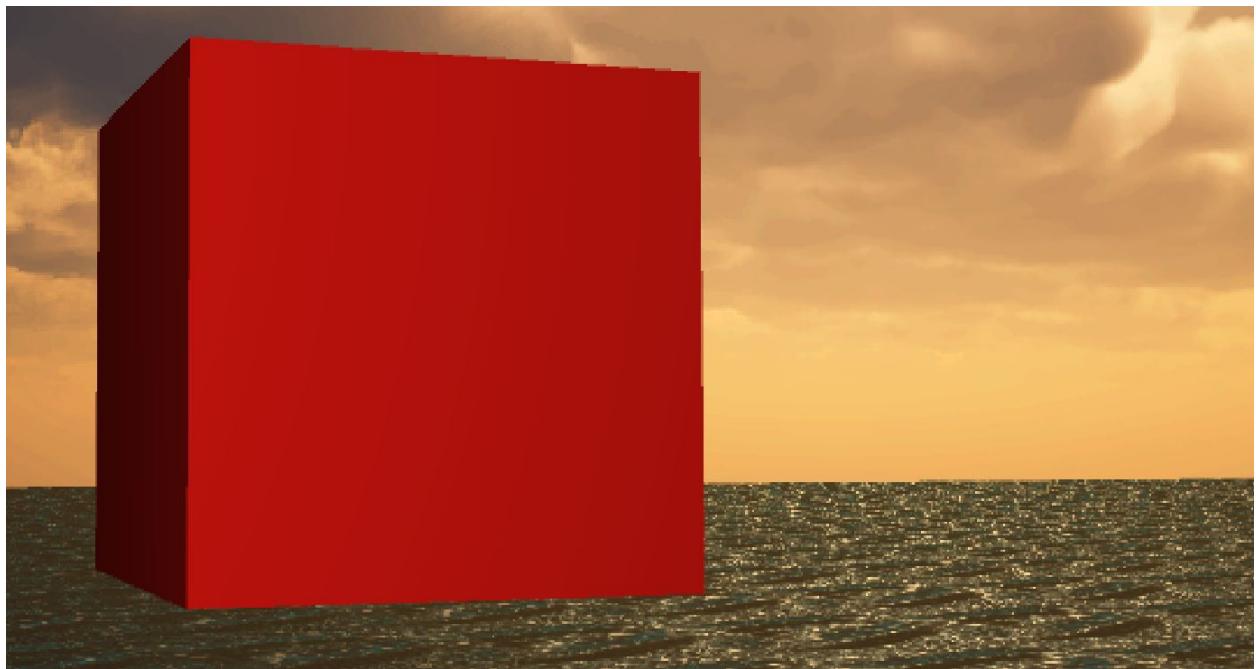
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
}
```

framebuffer multi-échantilloné au buffer en lecture et nous attachons le framebuffer intermédiaire au framebuffer en écriture. Ensuite, nous utilisons la commande glBlitFramebuffer qui s'occupe d'adapter la texture multi-échantilloné en texture standard

dans laquelle on peut facilement lire et traiter les couleurs. Ainsi, le framebuffer par défaut ira chercher la texture du framebuffer intermédiaire pour rendre la scène finale. Voici l'exemple d'un cube rendu avec et sans anti aliasing:



(avec Anti Aliasing)



(sans Anti Aliasing)

L'effet est notable sur les côtés du cube et sur la ligne d'horizon créé avec la surface de l'eau.

Ressources

- Pièce musicale « November », fichier november.mp3 musique libre de droit, disponible sur le site bensound.com.
- Ambiance sonore de vent, fichier wind.mp3, banque d'effets sonore Sound Ideas.
- Texture de surface d'eau, fichier water1.jpg, texture gratuite disponible sur le site thepixellab.net.
- Texture de l'écorce de l'abre, fichier wood.jpg, texture gratuite disponible sur le site bittbox.com.
- Texture de la normal map des arbres, fichier woodNormalMap.jpg, texture disponible sur le site filterforge.com.
- Texture de la normal map des champignons, fichier champignonNormalMap.png, texture disponible sur le site keithlantz.net.
- Texture de la normal map de l'eau, fichier waterNormalMap.jpg, texture disponible sur le site filterforge.com.
- Tous les modèles obj ont été téléchargés sur le site <http://tf3dm.com>.
- Les textures du cubemap, fichiers dans le répertoire bon/data/Textures/ciel, ont été créé par begla et assombri avec Photoshop. Elles sont permises pour toutes usages. License: Creative Commons Attributionl 3.0 Unported License.
- Modèle d'illumination inspiré des tutoriels disponible sur le site learnopengl.com.
- Tutoriel pour les effets de 'post-processing'. <http://www.gbzogany.com/?p=58> et <http://www.learnopengl.com/#Advanced-OpenGL/Framebuffers>

Présentation

Pierre-Marc Levasseur, Bac en informatique

Alex Gervais, Bac en Génie Logiciel

Keven Perreault, Bac en informatique