Document de design

Sommaire

Lors du cours d'infographie IFT-3100 de l'hiver 2018, notre équipe a réalisé une application visant à simuler une version amateur d'un moteur graphique comme Unreal Engin ou Unity.

L'objectif principal du projet de session étant de développer une application qui permet de construire, éditer et rendre des scènes, nous avons choisi de prendre comme ligne directrice une approche orientée vers le jeu vidéo.

Tout au cours du projet, nous avons dû choisir des thèmes à développer parmi 50 objectifs fonctionnels, soit 25 par livrable.

Chaque tranche de 5 objectifs fonctionnels cible un des sujets visités lors du cours.

Lors de l'évaluation des objectifs, ceux choisis seront évalués avec un système d'étoile ayant cette légende:
*** au-delà des attentes
** au niveau des attentes
* en bas des attentes

Les points du projet sont accordés en conséquence, soit un point par étoile pour un cumulatif maximum de 40 points.

Afin de profiter au maximum des connaissances théoriques du cours, nous avons choisi des technologies et designs nous permettant d'étendre au maximum nos apprentissages sans pour autant se couper les ailes.

Interactivité

Dans le logiciel conçu, il existe plusieurs formes d'interactivité afin de mettre l'utilisateur à l'aise.

Afin de permettre à l'utilisateur d'admirer son monde dans toute sa splendeur, nous avons implémenté une vision par caméra en première personne. La caméra est accessible à l'aide d'un clic droit soutenu et le mouvement de la souris sur la fenêtre.

Pour accentuer la caméra, nous permettons aussi le mouvement de la caméra accessible à l'aide d'un clic droit soutenu et l'utilisation des touches flèches.

L'importation de modèles 3D de plusieurs types est possible directement dans le programme à l'aide de l'interface graphique utilisateur.

Il est possible d'effectuer une capture d'écran avec l'interface utilisateur, ainsi qu'à l'aide de la touche F11 du clavier, ce qui fournit un raccourci intéressant pour accélérer la manoeuvre.

Il est aussi possible de faire l'importation d'une texture dans le monde directement sur une forme de type quadrilatère.

Afin de bien gérer les objets du monde, il existe une liste représentant le graphe de scène interne à l'application. Elle permet ainsi de voir tous les objets entourant la caméra, leurs regroupements et des outils pour les modifier. Voici quelques modifications possibles grâce aux outils fournis:
-translation, la rotation et l'échelle d'un objet quelconque.
-choix de couleur gérée par un sélecteur de couleur.
-possibilité de dessin de primitives vectorielles non affectées par le monde 3D.
-une case peut être cochée afin d'activer le *skybox*.
-possibilité de changer le curseur pour mieux plaire à l'oeil de l'utilisateur.
-etc.

Technologie

Les outils utilisés ont été choisis afin de permettre une certaine liberté dans la conception en offrant l'accès direct à Opengl.

Logiciel système (*Framework*): Nous avons décidé d'utiliser SDL et SDL_Image, un framework qui semble bien aimé des professionnels afin de gérer les évènements et les fenêtres. Toutefois son utilisation dans le projet se limite sensiblement à ceci, car nous souhaitions utiliser l'essence même du OpenGl et non passer par des fonction préconçues comme celles de Openframeworks. Heureusement, SDL offre la possibilité d'utiliser directement un contexte OpenGL dans les fenêtre.

Gestion de l'interface graphique utilisateur: Pour le GUI, nous avons décidé d'utiliser un projet disponible sur github créé par Omar Cornut nommé ImGui. Une fois installé, cet outil offre l'accès à plusieurs fichiers c++ qui contiennent parfois des défauts, mais de par leur disponibilité même permettent la correction de ces problèmes.

Activation des fonctionnalités du OpenGL moderne: Pour l'activation de ces fonctionnalités sur Windows, nous avons utilisé glew de nigels-com. Bref, l'utilisation de cette librairie s'effectue en deux lignes, soit activer le mode expérimental et initialiser glew.

Mathématiques: Pour la convivialité du côté des mathématiques nécessaires à l'infographie, nous avons utilisé la librairie OpenGL Mathematics (GLM) qui permet l'utilisation de vecteur et matrices rapidement compatibles avec les shaders d'OpenGL.

Gestion de la compilation: Pour la gestion des liens, de comment construire le projet et aider l'installation sur les ordinateurs à l'externe, nous avons choisi d'adopter la solution CMAKE qui est compatible avec les différents systèmes d'exploitation, ce qui permet alors le développement facile pour tous.

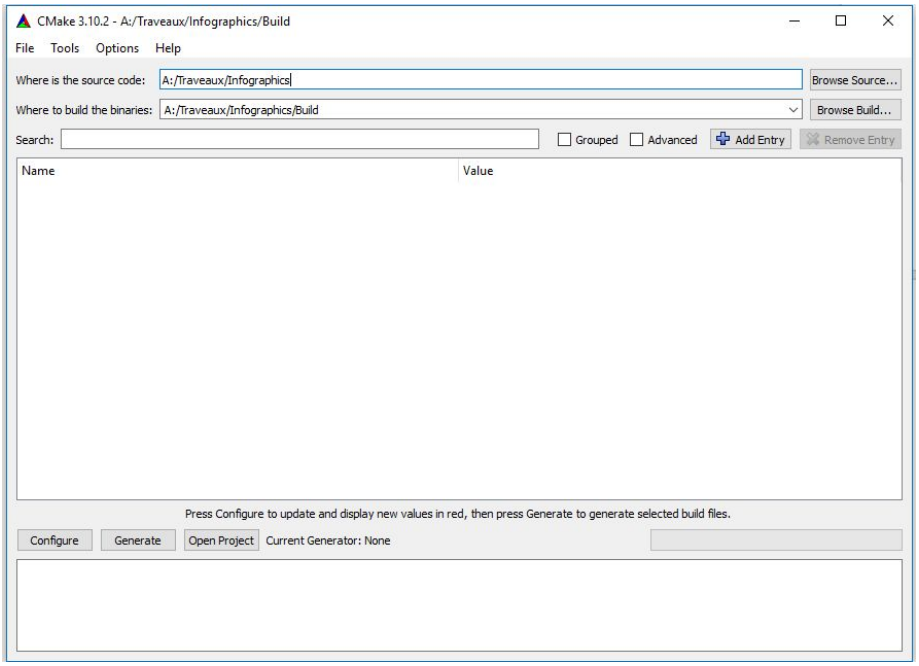
Gestion des versions: Pour la gestion des versions, nous avons profité de la gratuité de github pour les étudiants afin de nous créer un dépôt. Ceci permet donc de gérer la fusion automatique de différentes parties du projet réalisé par des étudiants différents. Github assure aussi la sécurité du projet en fournissant une façon de récupérer une version précédente en cas de catastrophe.

Chargement des modèles 3D externes: Pour le chargement des modèles, nous avons adopté une librairie cross-platform dénommée open asset import library (Assimp). Comme le nom le dit, elle sert à lire les vertices et indices de texture d'un modèle particulier. Toutefois, nous devons quand même gérer l'ajout de ces éléments dans des tableaux opengl et les envoyer au shader.

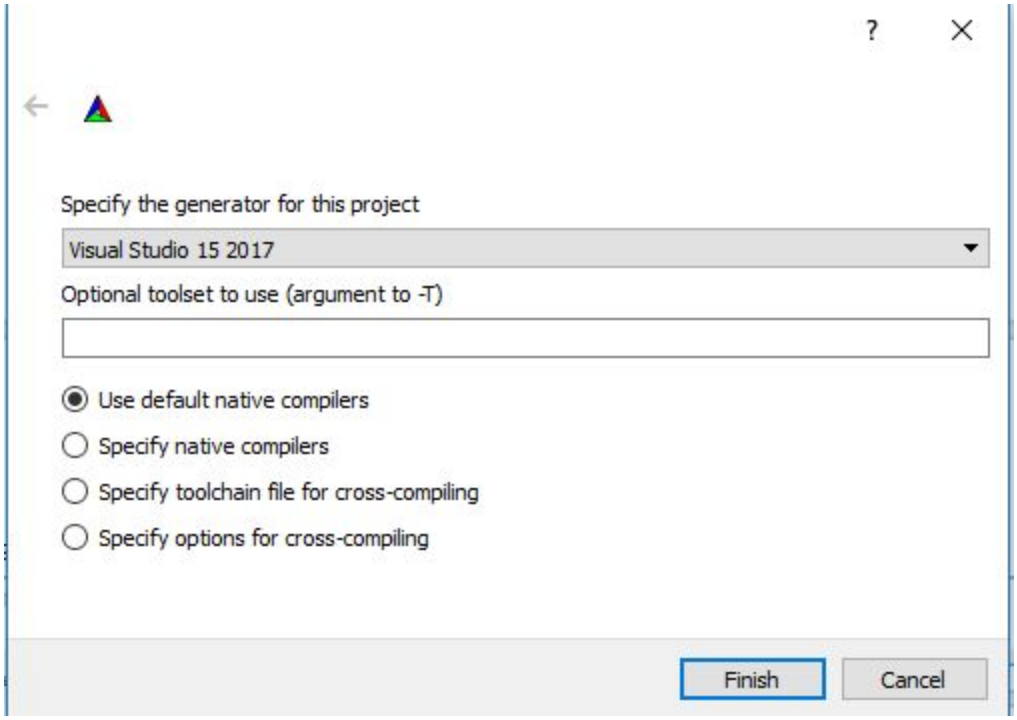
Compilation

Voici le processus pour avoir accès à l'outil de compilation sur Windows:

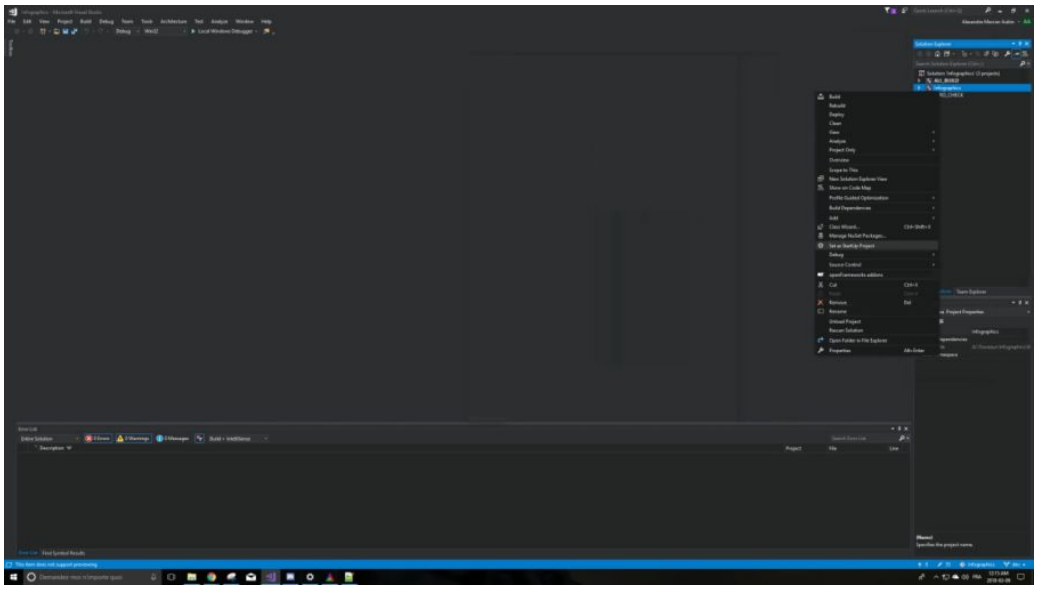
1. Installez Cmake selon l'architecture de l'ordinateur au lien suivant: https://cmake.org/download/
2. Dans le champ source code, entrer le chemin vers le dossier Infographics et dans le champ binaries, entrez le même chemin en ajoutant /Build. Voici ce à quoi la fenêtre devrait ressembler:



3. Cliquez sur configure et choisir l'éditeur que vous utilisez, pour les étapes, nous assumons que vous utiliserez Visual Studio 15. Voici ce à quoi la fenêtre devrait ressembler:



4. Cliquez sur Finish
5. Une fois la configuration terminée, appuyez sur Generate.
6. Une fois la génération terminée, appuyez sur Open Project.
7. Dans Visual Studio, aller dans l'explorateur de solution, faire un clic droit sur le projet Infographics et choisir set as StartUp Project.



8. Une fois que le projet est choisi comme projet de démarrage, cliquer sur build.
9. Rouler le programme et apprécier la vue.

Architecture

L’architecture du projet se divise en plusieurs parties comme suit:

Main: Se charge uniquement de préparer l’application, de partir la boucle principale et de fermer l’application.

Application: Se charge principalement de l’initialisation de la fenêtre utilisée par Renderer, la boucle principale qui permet la capture des événements système et de l’appel de la fonction de dessin du Renderer.

Renderer: Se charge de la gestion de l’interface graphique et des dessins au bon moment des objets graphiques de Scene. Il est aussi responsable des dessins de l’interface graphique utilisateur supportée par ImGui. Elle délègue les événements en lien avec la scène à son instance de l’objet Scene.

Scene: Cette classe contient tous les objets d’une scène dans un GroupObject et comporte l’essentiel à leur rendu, soit des matrices de vue et de perception. Cette classe gère la caméra et l'interaction avec les objets du monde virtuel.

AbstractObject: Classe polymorphe qui permet de classer les objets du monde. Cette classe comporte plusieurs fonctions utiles comme uniformColor qui prépare une couleur uniforme ou les fonctions permettant de faire des matrices de translation,rotation et échelle.

GroupObject: Enfant de AbstractObject qui contient une liste d’AbstractObject afin de permettre un classement des objets monde sous forme d’arborescence.

QuadObject: Enfant de AbstractObject permettant de dessiner un quadrilatère dans le monde 3D

CubeObject: Enfant de AbstractObject permettant de dessiner un cube

SkyboxObject: Enfant de AbstractObject permettant de dessiner un skybox.

PrimitiveObject: Enfant de AbstractObject permettant de dessiner les différents types de primitives vectorielles. Elle est aussi utilisée pour faire le dessin du curseur de la souris.

ModelObject: Enfant de AbstractObject qui permet de dessiner un objet préalablement importé par la classe Model.

AutresObjects: Ceci n'est pas vraiment une classe, mais c'est ici pour montrer le polymorphisme des classes Objects. Il existe MirrorObject pour les miroirs, SBPyramidObject pour les pyramides, etc.

AbstractShader: Classe polymorphe qui contient des shaders de fragment et de vertex sous forme de raw string.

ShaderLoader: Classe prenant un enfant de AbstractShader et crée un programme shader retourné sous forme de GLuint.

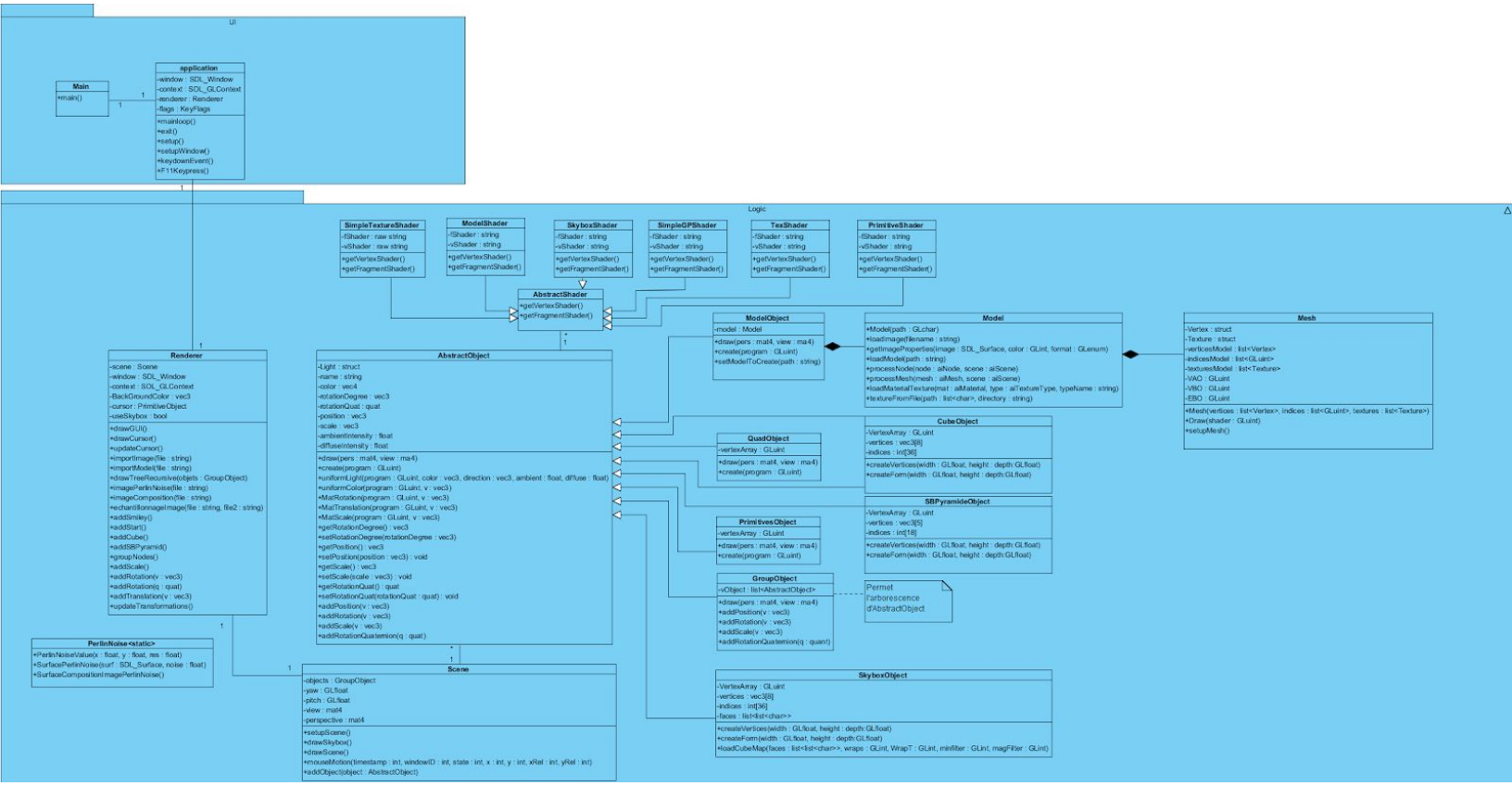
Model: Cette classe permet de gérer l'importation d'un modèle avec Assimp et l'importation de textures avec SDL_Surface. Une fois chargé, il lie la texture à un identifiant et transfert toutes les information du model à Mesh.

Mesh: Gère le dessin du modèle importé et lie ses vertices, ses normales et ses textures aux entrées de shader.

PerlinNoise: Fonction qui génère une texture procédurale de type bruit de Perlin.

Nous avons aussi un diagramme de classe rapide qui représente bien l'application, il est disponible en meilleur qualité dans le vpp du dossier doc (Ouvrable avec *Visual Paradigm*).

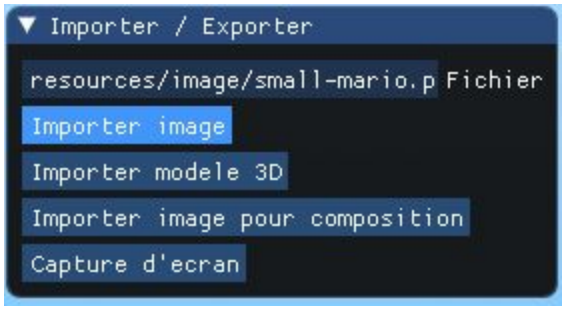
TessellationQuad: un objet compatible avec le shader de tesselation.


UI
Main
+main()
application
-window : SDL_Window
-context : SDL_GLContext
-renderer : Renderer
-flags : KeyFlags
+mainloop()
+exit()
+setup()
+setupWindow()
+keydownEvent()
+F11Keypress()
Logic
SimpleTextureShader
-fShader : raw string
-vShader : raw string
+getVertexShader()
+getFragmentShader()
ModelShader
-fShader : string
-vShader : string
+getVertexShader()
+getFragmentShader()
SkyboxShader
-fShader : string
-vShader : string
+getVertexShader()
+getFragmentShader()
SimpleGPShader
-fShader : string
-vShader : string
+getVertexShader()
+getFragmentShader()
TexShader
-fShader : string
-vShader : string
+getVertexShader()
+getFragmentShader()
PrimitiveShader
-fShader : string
-vShader : string
+getVertexShader()
+getFragmentShader()
AbstractShader
+getVertexShader()
+getFragmentShader()
ModelObject
-model : Model
+draw(pers : mat4, view : mat4)
+create(program : GLuint)
+setModelToCreate(path : string)
Model
+Model(path : GLchar)
+loadImage(filename : string)
+getImageProperties(image : SDL_Surface, color : GLint, format : GLenum)
+loadModel(path : string)
+processNode(node : aiNode, scene : aiScene)
+processMesh(mesh : aiMesh, scene : aiScene)
+loadMaterialTexture(mat : aiMaterial, type : aiTextureType, typeName : string)
+textureFromFile(path : list<char>, directory : string)
Mesh
-Vertex : struct
-Texture : struct
-verticesModel : list<Vertex>
-indicesModel : list<GLuint>
-texturesModel : list<Texture>
-VAO : GLuint
-VBO : GLuint
-EBO : GLuint
+Mesh(vertices : list<Vertex>, indices : list<GLuint>, textures : list<Texture>)
+Draw(shader : GLuint)
+setupMesh()
Renderer
-scene : Scene
-window : SDL_Window
-context : SDL_GLContext
-BackGroundColor : vec3
-cursor : PrimitiveObject
-useSkybox : bool
+drawGUI()
+drawCursor()
+updateCursor()
+importImage(file : string)
+importModel(file : string)
+drawTreeRecursive(objets : GroupObject)
+imagePerlinNoise(file : string)
+imageComposition(file : string)
+echanillonnageImage(file : string, file2 : string)
+addSmiley()
+addStart()
+addCube()
+addSBPyramid()
+groupNodes()
+addScale()
+addRotation(v : vec3)
+addRotation(q : quat)
+addTranslation(v : vec3)
+updateTransformations()
AbstractObject
-Light : struct
-name : string
-color : vec4
-rotationDegree : vec3
-rotationQuat : quat
-position : vec3
-scale : vec3
-ambientIntensity : float
-diffuseIntensity : float
+draw(pers : mat4, view : ma4)
+create(program : GLuint)
+uniformLight(program : GLuint, color : vec3, direction : vec3, ambient : float, diffuse : float)
+uniformColor(program : GLuint, v : vec3)
+MatRotation(program : GLuint, v : vec3)
+MatTranslation(program : GLuint, v : vec3)
+MatScale(program : GLuint, v : vec3)
+getRotationDegree() : vec3
+setRotationDegree(rotationDegree : vec3)
+getPosition() : vec3
+setPosition(position : vec3) : void
+getScale() : vec3
+setScale(scale : vec3) : void
+getRotationQuat() : quat
+setRotationQuat(rotationQuat : quat) : void
+addPosition(v : vec3)
+addRotation(v : vec3)
+addScale(v : vec3)
+addRotationQuaternion(q : quat)
QuadObject
-vertexArray : GLuint
+draw(pers : mat4, view : ma4)
+create(program : GLuint)
CubeObject
-VertexArray : GLuint
-vertices : vec3[8]
-indices : int[36]
+createVertices(width : GLfloat, height : depth:GLfloat)
+createForm(width : GLfloat, height : depth:GLfloat)
PrimitivesObject
-vertexArray : GLuint
+draw(pers : mat4, view : ma4)
+create(program : GLuint)
SBPyramideObject
-VertexArray : GLuint
-vertices : vec3[5]
-indices : int[18]
+createVertices(width : GLfloat, height : depth:GLfloat)
+createForm(width : GLfloat, height : depth:GLfloat)
GroupObject
-vObject : list<AbstractObject>
+draw(pers : mat4, view : ma4)
+addPosition(v : vec3)
+addRotation(v : vec3)
+addScale(v : vec3)
+addRotationQuaternion(q : quat)
Permet
l'arborescence
d'AbstractObject
PerlinNoise<static>
+PerlinNoiseValue(x : float, y : float, res : float)
+SurfacePerlinNoise(surf : SDL_Surface, noise : float)
+SurfaceCompositionImagePerlinNoise()
Scene
-objects : GroupObject
-yaw : GLfloat
-pitch : GLfloat
-view : mat4
-perspective : mat4
+setupScene()
+drawSkybox()
+drawScene()
+mouseMotion(timestamp : int, windowID : int, state : int, x : int, y : int, xRel : int, yRel : int)
+addObject(object : AbstractObject)
SkyboxObject
-VertexArray : GLuint
-vertices : vec3[8]
-indices : int[36]
-faces : list<list<char>>
+createVertices(width : GLfloat, height : depth:GLfloat)
+createForm(width : GLfloat, height : depth:GLfloat)
+loadCubeMap(faces : list<list<char>>, wraps : GLint, WrapT : GLint, minfilter : GLint, magFilter : GLint)

Fonctionnalités

1.1 Importation d'images

Pour importer une image, il suffit d'entrer le chemin de l'image à importer dans le champ *Fichier* de la fenêtre *Importer / Exporter*, puis cliquer sur le bouton *Importer image*, tel qu'illustré sur l'image suivante, suivie du résultat de l'importation:



Ce qui se produit ici dans le code est qu'un objet de type *QuadObject* est ajouté à la scène, en utilisant l'image importée comme texture.

Note 1: Quelques images sont disponibles dans le dossier *Resources* du projet pour simplifier les tests.
Note 2: Les images au format JPEG (ou JPG) ne sont pas supportées.

1.2 Exportation d'images

L'exportation d'images est l'équivalent d'une capture d'écran de la fenêtre de l'application. L'image est cherchée dans le buffer de dessin et ensuite convertie en *SDL_Surface* pour ensuite être enregistrée en BMP grâce à *SDL_SaveBMP*. La raison de la nécessité de l'inversion en y semblerait être que sous Windows, les fichiers BMP sont inversés nativement. Pour expérimenter la fonction, il suffit d'appuyer sur F11 pour prendre la capture ou de cliquer sur le bouton *Capture d'écran* dans la fenêtre *Importer / Exporter*. Lorsqu'on utilise le bouton de capture, on peut également sélectionner l'endroit où l'image sera enregistrée en écrivant le chemin dans le champ *Fichier* un peu plus haut.

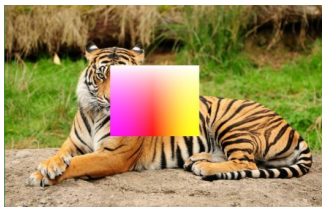
Voici la partie principale du code:

```
void Renderer::screenShot(int x, int y, int w, int h, const char * filename)
{
    unsigned int size = w * h * 4;
    unsigned char *pixels = new unsigned char[size]; // 4 bytes for RGBA
    glReadBuffer(GL_FRONT);
    glReadPixels(x, y, w, h, GL_BGRA, GL_UNSIGNED_BYTE, pixels);

    //vertical flip cause glRead goes backwards for some reason
    unsigned char *flipPixels=new unsigned char[size];
    for (int i = 0; i < w; ++i) {
        for (int j = 0; j < h; ++j) {
            for (int k = 0; k < 4; ++k) {
                flipPixels[(i + j * w) * 4 + k] = pixels[(i + (h - 1 - j) * w) * 4 + k];
            }
        }
    }

    SDL_Surface * surf = SDL_CreateRGBSurfaceFrom(flipPixels, w, h, 8 * 4, w * 4, 0, 0, 0, 0);
    SDL_SaveBMP(surf, filename);

    SDL_FreeSurface(surf);
    delete[] pixels;
    delete[] flipPixels;

}
```

1.3 Échantillonnage d'image

Pour l'échantillonnage d'image, il est possible de générer une image à partir d'échantillons de pixels en provenance de deux images. Il suffit simplement de sélectionner une image de base et une image qui va servir d'échantillon de pixels, puis de choisir le pourcentage de cette image qui va être échantillonnée et de sélectionner où nous voulons faire afficher cet échantillon sur l'image de base pour finalement générer une nouvelle image. Tout ceci est géré à l'aide de l'interface graphique dans la fenêtre d'échantillonnage d'image. Voici un exemple où les deux images de gauche ensemble deviennent l'image de droite.





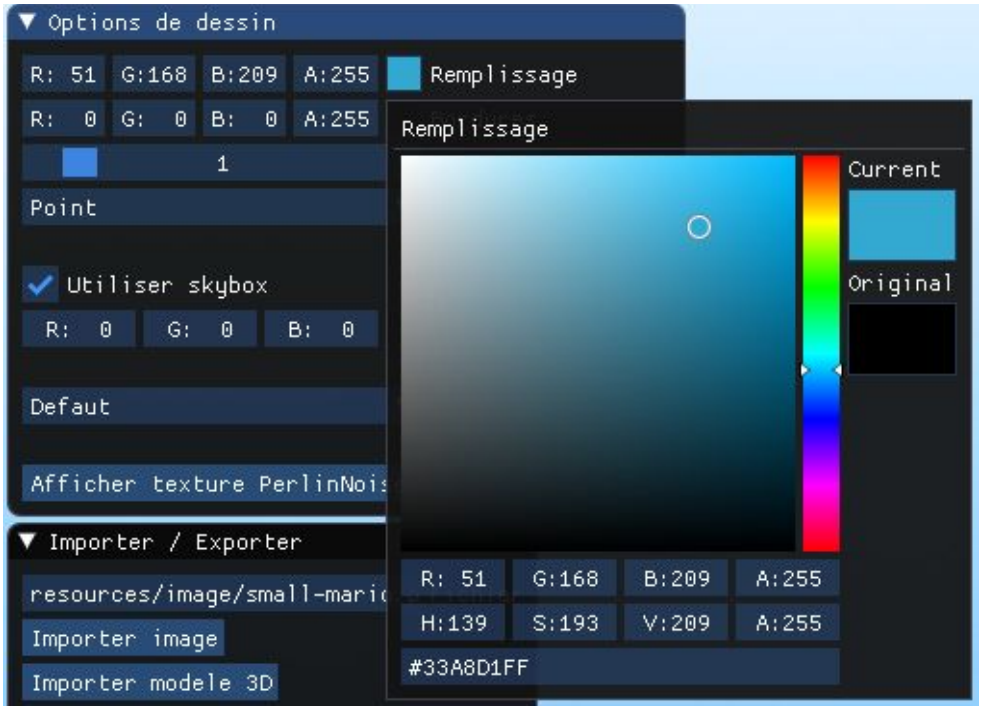
1.4 Sélecteur de couleur

Des sélecteurs de couleur sont disponibles à quatre emplacements différents dans l'application.

1. Dans la fenêtre *Options de dessin*, on peut sélectionner la couleur de remplissage utilisée pour le prochain dessin de primitive.
2. Dans la même fenêtre, on peut sélectionner la couleur des bordures pour le prochain dessin de primitive.
3. Toujours dans la fenêtre *Options de dessin*, il est possible de sélectionner la couleur de l'arrière-plan de la scène lorsque le skybox est désactivé. Ce sélecteur n'inclut pas de composante alpha, contrairement aux trois autres, puisque ce serait inutile dans le contexte d'une couleur d'arrière-plan.
4. Dans le fenêtre *Transformations* (qui apparaît lorsqu'un élément est sélectionné dans le graphe de scène), il est possible d'aller modifier la couleur des éléments sélectionnés.

Ces sélecteurs de couleur sont entièrement gérés par *ImGui*. Il y a trois façons d'éditer la couleur à l'aide de ces sélecteurs. On peut soit:

- Double-cliquer sur l'une des composantes de couleur et modifier manuellement sa valeur à l'aide du clavier.
- Cliquer sur l'une des composantes puis, tout en maintenant le bouton enfoncé, glisser vers la droite pour augmenter la valeur ou glisser vers la gauche pour la réduire.
- Cliquer sur le carré d'aperçu de couleur à droite des champs d'édition pour ouvrir une petite fenêtre supplémentaire permettant de sélectionner la couleur intuitivement, puis cliquer à côté pour fermer la fenêtre.



1.5 Espace de couleur

Les différents espaces de couleurs sont gérés par exemple lors de l'importation des textures.
Le tout fonctionne grâce à une petite fonction qui permet de classer quel est le type d'image selon ses caractéristiques, soit le nombre d'octets par pixel et son résultat de l'application d'un masque de rouge. Autrement dit, nous gérons RGB, BGR, RGBA et BGRA.
Voici cette fonction:

```
void  Model::getImageProperties(SDL_Surface *image, GLint &nOfColors,GLenum &texture_format)
{
    if (image == NULL)
    {
        cout << "can't get properties of a null image" << endl;
        return;
    }

    nOfColors = image->format->BytesPerPixel;
    if (nOfColors == 4)     // contains an alpha channel
    {
        if (image->format->Rmask == 0x000000ff) texture_format = GL_RGBA;
        else texture_format = GL_BGRA;
    }
    else if (nOfColors == 3)     // no alpha channel
    {
        if (image->format->Rmask == 0x000000ff) texture_format = GL_RGB;
        else texture_format = GL_BGR;
    }
    else
    {
        cout << "weird texture detected" << endl;
    }
}
```

De plus, les sélecteurs de couleur de *ImGui* permettent si on le souhaite de sélectionner la couleur en format HSV lorsqu'on ouvre l'éditeur de l'un des sélecteurs (se référer à l'image de la fonctionnalité 1.4).

2.1 Curseur dynamique

L'application offre la possibilité de modifier le curseur de la souris pour cinq différents dessins vectoriels au choix. Pour sélectionner le curseur souhaité, il suffit de sélectionner son préféré dans la liste déroulante *Curseur* de la fenêtre *Options de dessin*. Voici à quoi ressemble ces différents curseurs:

Point Points Croix Triangle Quad

L'essentiel du code qui gère les curseurs se trouve dans Renderer::updateCursor() et *Renderer::drawCursor()*. Le code n'est pas inclus ici car il est un peu trop long.

2.2 Outils de dessin

La fenêtre *Options de dessin* offre la possibilité à l'utilisateur de modifier la valeur des différents outils de dessin tels que:

- La couleur de remplissage de la prochaine forme qui sera dessinée
- La couleur des bordures de la prochaine forme qui sera dessinée
- L'épaisseur des bordures de la prochaine forme qui sera dessinée (si la valeur est 0, la forme n'aura pas de bordures)
- La couleur de l'arrière-plan (prend seulement effet lorsque le skybox est désactivé)

Ces valeurs sont stockées dans des variables membres de la classe *Renderer* et utilisées lors du dessin.

2.3 Primitives vectorielles

L'utilisateur peut dessiner cinq types de primitives vectorielles. Pour ce faire, on commence par sélectionner la forme voulue dans la liste déroulante *Forme à dessiner* de la fenêtre *Options de dessin*. Ensuite, on ajuste les différents paramètres décrits dans la fonctionnalité 2.2. Finalement, on clique à des endroits libres de la fenêtre où on veut placer un des sommets de la primitive à dessiner. Tous les exemples suivants sont faits avec remplissage jaune, bordure noire, bordure de taille 3:

- Lorsque *Point* est sélectionné, on clique à un emplacement et un point y sera dessiné. Sa taille et sa couleur sont déterminées par la couleur et l'épaisseur des bordures. Dessiné à l'aide de GL_POINTS.

- Lorsque *Ligne* est sélectionné, on clique à deux emplacements pour dessiner une ligne entre ces deux points. Les lignes ne sont pas affectées par la couleur de bordure. Dessiné à l'aide de GL_LINES

- Lorsque *Triangle* est sélectionné, on clique à trois emplacements pour indiquer les emplacements des trois sommets. Dessiné à l'aide de GL_TRIANGLES.

- Lorsque *Rectangle* est sélectionné, on clique à deux emplacements pour indiquer deux sommets opposés du rectangle. Les deux autres sommets sont extrapolés. Dessiné à l'aide de GL_TRIANGLE_FAN.

- Lorsque *Quad* est sélectionné, on clique à quatre emplacements, en sens horaire ou anti-horaire (au choix) pour indiquer les quatre sommets du quadrilatère à dessiner. Dessiné à l'aide de GL_TRIANGLE_FAN.

L'essentiel du code servant à dessiner ces primitives peut être consulté dans les méthodes *Application::mainLoop()* (pour la réception des événements de clic) et *Renderer:: ajouterPtDessin()*.

2.4 Formes vectorielles

L'application permet à l'utilisateur de dessiner deux formes vectorielles différentes: le bonhomme sourire (*Smiley*) et l'astérisque (*Étoile*). La marche à suivre pour les dessiner est la même que pour les primitives vectorielles. Les exemples suivants utilisent les même paramètres de dessin que ceux de la fonctionnalité précédente (remplissage jaune, bordures, noires, bordures de taille 3):

- Lorsque *Smiley* est sélectionné, on clique à deux emplacements pour indiquer deux sommets opposés du visage du bonhomme. Les yeux et la bouche sont de la couleur des bordures. Chacune des quatre primitives composant cette forme (tête, oeil gauche, oeil droit, bouche) sont dessinées à l'aide de GL_TRIANGLE_FAN.

- Lorsque *Étoile* est sélectionné, on clique à deux emplacements pour donner les coordonnées de l'une des deux lignes diagonales. Les coordonnées des autres lignes sont extrapolés. La couleur de remplissage n'a aucun effet. Chacune des quatre lignes sont dessinées à l'aide de GL_LINES.

L'essentiel du code de dessin de ces formes se retrouve dans *Renderer::ajouterSmiley()* et *Renderer::ajouterEtoile()*, en plus des deux mêmes méthodes que celles utilisées pour le dessin de primitives.

2.5 Interface

L'interface graphique gérée par ImGui est utilisée pour accéder à presque toutes les autres fonctionnalités de l'application. Elle est composée de cinq fenêtres flottantes ancrées aux bordures de la fenêtre principale de l'application. De plus, chacune des fenêtres peut être réduite en cliquant sur la petite flèche en haut à gauche de la fenêtre en question. Ces cinq fenêtres sont:

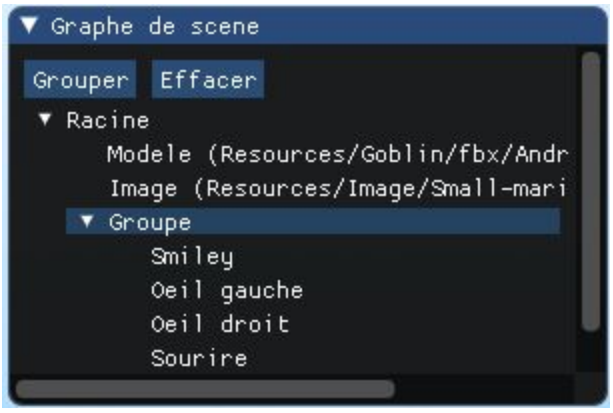
- *Options de dessin* qui permet de régler les paramètres de dessin vectoriel, de modifier le curseur à utiliser, de choisir si on veut utiliser le skybox ou une couleur fixe d'arrière-plan, ainsi qu'un bouton permettant d'accéder à la fonction de génération de texture procédurale.
- *Importer / Exporter* qui permet d'importer des images et des modèles, exporter des captures d'écran, ainsi qu'accéder à la fonction de composition d'image.
- *Échantillonnage d'image* qui permet d'accéder à la fonction du même nom.

- *Graphe de scène* qui affiche les différents éléments présents dans la scène en se moment, sous forme d'arborescence.
- *Transformations* (seulement visible lorsqu'un élément est sélectionné dans le graphe de scène) qui permet d'appliquer dynamiquement des transformations aux différents éléments de la scène qui sont sélectionnés dans le graphe de scène.

Les captures d'écran sont ici intentionnellement omises, puisque celles-ci sont déjà disponibles dans les descriptions des autres fonctionnalités. L'essentiel du code de l'affichage du GUI est consultable dans *Renderer::drawGUI()*.

3.1 Graphe de scène

Le graphe de scène côté architecture est composé d'un GroupObject qui permet de conserver des pointeurs intelligents d'AbstractObject. Cette classe étant elle-même un AbstractObject, elle peut contenir des références vers d'autres GroupObjects afin de former une structure de type arbre n-aire. Lors de l'appel de la fonction draw, l'évènement est délégué à tous les objets de sa liste interne. Il est aussi possible d'effacer, ajouter ou obtenir un objet du tableau avec les fonctions implémentées dans cette classe.



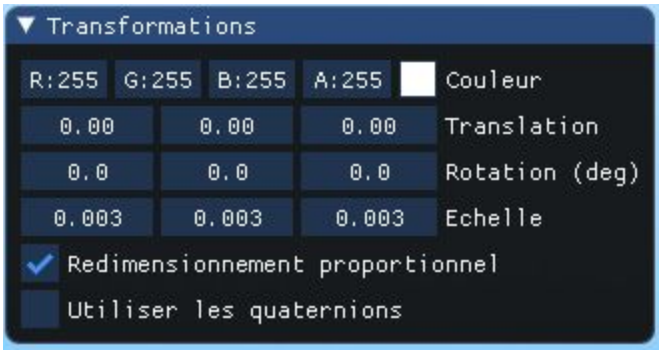
Du côté GUI, le graphe de scène est dessiné dans la fenêtre *Graphe de scène* à l'aide de la méthode récursive *Renderer:: drawTreeRecursive()*. Au départ, les groupes sont fermés. Pour afficher leur contenu, il suffit de cliquer sur la petite flèche à gauche du nom du groupe. Lorsqu'on clique sur l'un des éléments, il devient sélectionné. Lorsqu'un ou plusieurs éléments sont sélectionnés, on peut appuyer sur la touche Suppr. du clavier ou le bouton *Effacer* du GUI pour retirer ces éléments de la scène. On peut également mettre plusieurs éléments sélectionnés dans un même groupe avec le bouton *Grouper*. Finalement, pour ajouter des nouveaux éléments, il faut procéder avec les diverses façon de créer des nouveaux objets (par exemple, importer une image, importer un modèle ou dessiner une primitive vectorielle.

3.2 Sélection multiple

Dans le graphe de scène, il est possible de sélectionner plusieurs éléments simultanément en maintenant appuyée la touche Ctrl du clavier avant de cliquer sur les éléments à sélectionner. Cette opération peut servir à effacer plusieurs éléments en même temps ou encore à les grouper. De plus, lorsque plusieurs éléments sont sélectionnés, les transformations interactives (décrites dans la prochaine fonctionnalité) sont appliquées à chacun des éléments sélectionnés.

3.3 Transformations interactives

Les transformations interactives sont effectuées à l'aide de l'interface graphique, plus précisément, la fenêtre *Transformations*. Afin d'accéder à cette fenêtre, assurez-vous qu'au moins un élément est sélectionné dans le graphe de scène. Une fois qu'un objet est sélectionné, il suffit de modifier ses paramètres. Pour ce faire, on peut double cliquer sur un champ d'édition et entrer manuellement la valeur souhaitée au clavier, ou encore cliquer sur un des champs et, tout en maintenant le bouton appuyé, déplacer la souris vers la droite pour augmenter la valeur ou vers la gauche pour la réduire.



Il est aussi possible d'appliquer les transformations à plusieurs objets en utilisant la sélection multiple ou en sélectionnant un groupe (tous les éléments du groupe seront alors affectés par les transformations). Dans le code, cette partie est implémentée de façon très simple en profitant des propriétées du polymorphisme. Un seul appel aux méthodes de *AbstractObject setPosition()* (ou *addPosition()*), *setRotation()* (ou *addRotation()*) et *setScale()* (ou *addScale()*) permettent de modifier les transformations des objets dans la scène.

De plus, deux cases à cocher sont disponibles. L'une sert à sélectionner si on veut que les modifications à l'échelle de l'objet soient appliquées de façon proportionnelle en xyz ou individuellement. L'autre sert à activer le mode d'édition de la rotation par quaternions.

3.4 Historique de transformation

Non implémenté.

3.5 Système de coordonnées

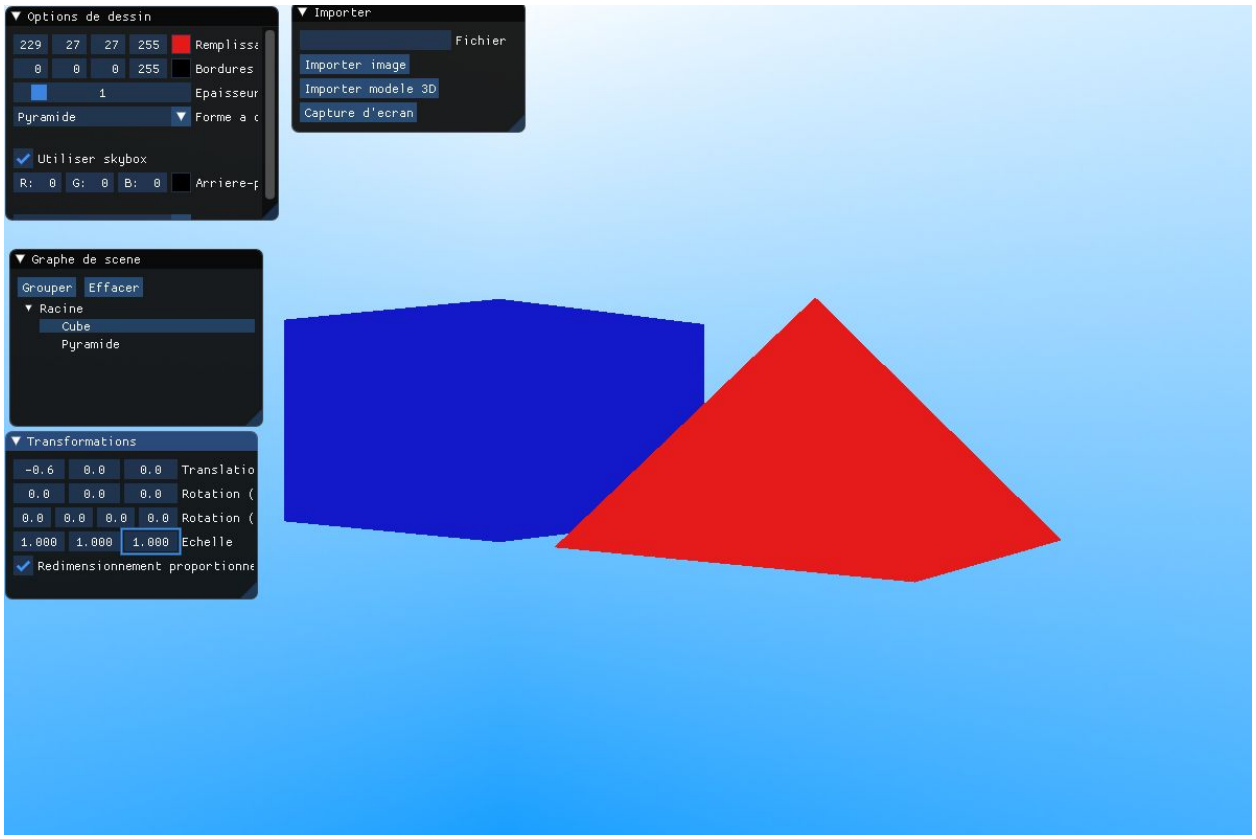
Non implémenté.

4.1 Boîte de délimitation

Non implémenté.

4.2 Primitives géométriques

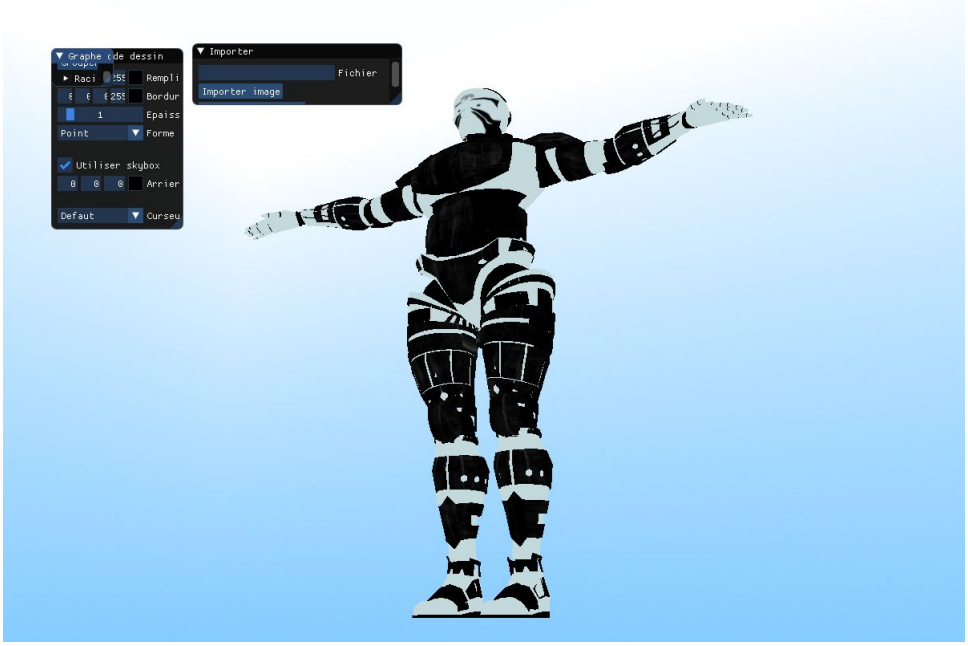
Pour les deux primitives géométriques demandées, nous avons choisi d'offrir le cube et la pyramide à base carrée à l'utilisateur. Comme pour les primitives vectorielles, il est possible de choisir la couleur de la forme avec le sélecteur de couleur de remplissage. On clique ensuite à un emplacement libre de l'écran pour que la primitive soit générée aux coordonnées (0,0,0). Ces formes géométriques peuvent être déplacées, tournées et modifiées en taille avec le menu de transformation. Le code permettant d'effectuer le dessin de ces formes se situe respectivement dans les classes CubeObject et SBPyramidObject. Le principe derrière leur affichage est de dessiner des triangles en fonction des points de coordonnées des sommets.



4.3 Modèle 3D

Pour importer un modèle, il suffit d'entrer le chemin du fichier dans le champ *Fichier* de la fenêtre *Importer / Exporter*. Puis, dans la même fenêtre, on appuie sur le bouton *Importer modèle 3D*.

Côté code, le chargement de modèles 3D est fait à partir d'*Assimp*. Le code de cette fonctionnalité se situe principalement dans *ModelObject*, *Model* et *Mesh*. *ModelObject* représente l'objet du monde virtuel, cette classe contients des informations telles que l'échelle, la rotation et la position de l'objet. *Model* se charge de la lecture des informations du fichier modèle et de transférer ces informations à Mesh. Mesh permets le rendu de cet objet grâce aux pointeurs qu'il initialise lors de l'appel de setupMesh() qui lui permettront de passer les vertices, normales et textures au shaders lors de l'appel de Draw(). Les modèles sont dessinés avec des GL_TRIANGLES.



4.4 Animation

Non implémenté.

4.5 Instanciation

Non implémenté.

5.1 Mapping

Le mapping autre que les quadrilatères et le skybox est principalement fait sur les modèles 3D importés. La partie qui lie la texture à un buffer est située dans la classe *Model*, tandis que le pointeur vers les coordonnées de textures est situé dans *Mesh*. Lors d'un appel à la fonction de dessin, *Model* ne fait que transférer l'appel au *Mesh*. Un bel exemple représentant le mapping est le requin disponible dans les ressources. Comme *Assimp* se charge de lire les indices, il nous sauve énormément de temps pour cet objectif fonctionnel.

5.2 Composition

Cet objectif fonctionnel est représenté par la composition d'une image choisie par l'utilisateur et un bruit de perlin. Celle-ci est instanciée sur un quadrilatère dans la scène lorsque que l'on appuie sur le bouton *Importer image pour composition* dans la section importer. Pour ce qui est du code, l'algorithme multiplie chacun des pixels de l'image par un bruit de Perlin pour donner une nouvelle texture. Voici un petit exemple où l'image de gauche devient celle de droite lorsque composé avec un algorithme de bruit de Perlin:

5.3 Traitement

Non implémenté.

5.4 Texture procédurale

La texture procédurale est représentée par un bruit de Perlin. Celle-ci est instanciée sur un quadrilatère dans la scène lorsque l'on appuie sur le bouton *Afficher texture PerlinNoise* dans la fenêtre *Options de dessin*. La texture est définie sur une surface où l'algorithme va modifier chacun des pixels de cette surface par un algorithme de bruit de Perlin multiplié par une couleur définie.

```
void SurfacePerlinNoise(SDL_Surface *surf, float noise)
{
    float red, green, blue;
    float  r = 53.0f;
    float  g = 122.0f;
    float  b = 183.0f;
    for (float i = 0; i < surf->w; i++)
    {
        for (float j = 0; j < surf->h; j++)
        {
            red = (r * PerlinNoiseValue(i, j, noise)) + r;
            green = (g * PerlinNoiseValue(i, j, noise)) + g;
            blue = (b * PerlinNoiseValue(i, j, noise)) + b;
            putpixel(surf, (int)i, (int)j, SDL_MapRGB(surf->format, (red <= 255.0f) ? (Uint8)red : 255,
                (green <= 255.0f) ? (Uint8)green : 255, (blue <= 255.0f) ? (Uint8)blue : 255));
        }
    }
};
```

5.5 Cubemap

Le cubemap est représenté par le skybox présent dans la scène, celui-ci est instancié à partir d'images dégradées faites à la main. Il est entièrement possible de le désactiver à l'aide de la case à cocher *Utiliser skybox* dans la fenêtre *Options de dessin*. Pour le code, nous avons utilisé GL_TEXTURE_CUBEMAP avec 6 images que nous envoyons dans la texture (en réalité c'est 3, car tous les côtés sont pareils). Pour le reste, la majorité du code ressemble au dessin d'un cube normal. Voir à la page suivante pour le code.

```cpp
GLuint SkyboxObject::loadCubemap(std::vector<char*> faces, GLint wrapS, GLint wrapT, GLint minFilter, GLint magFilter)
{
    GLuint textureID;
    SDL_Surface* image;
    GLint channels;
    GLenum type;

    //Générer la texture
    glGenTextures(1, &textureID);

    //Lier la texture à la cible texture 2D
    glBindTexture(GL_TEXTURE_CUBE_MAP, textureID);

    //wrapping
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
    //filtering
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);

    for (GLuint i = 0; i < faces.size(); i++)
    {

        image = Model::loadImage(string(faces[i]));
        Model::getImageProperties(image, channels, type);

        //link the texture
        glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0, channels, image->w, image->h, 0, type, GL_UNSIGNED_BYTE, image->pixels);
        //Free image
        SDL_FreeSurface(image);
    }

    glBindTexture(GL_TEXTURE_CUBE_MAP, 0); //Délie la texture pour éviter d'être corrompue avec une autre

    return textureID; //retourne l'objet texture créé
}
```

6.1 Point de vue

```cpp
void Scene::MatView(glm::mat4 &matView, bool staticPos)
{
    glm::vec3 front;
    glm::vec3 tempPos;

    position.y = 0;

    front.x = cosf(glm::radians(yaw)) * cosf(glm::radians(pitch));
    front.y = sinf(glm::radians(pitch));
    front.z = sinf(glm::radians(yaw)) * cosf(glm::radians(pitch));

    direction = glm::normalize(front);

    if (staticPos)
    {
        tempPos = glm::vec3(0.0, 0.0, 1.0);
    }
    else
    {
        tempPos = position;
    }

    matView = glm::lookAt(tempPos, direction + tempPos, orientation);
}
```

Le point de vue permet à l'utilisateur de se déplacer et de regarder vers différentes directions dans la scène. Afin de conserver une certaine convivialité avec les menus, ce déplacement doit être activé avec un clic droit soutenu, comme dans Unreal Engine.

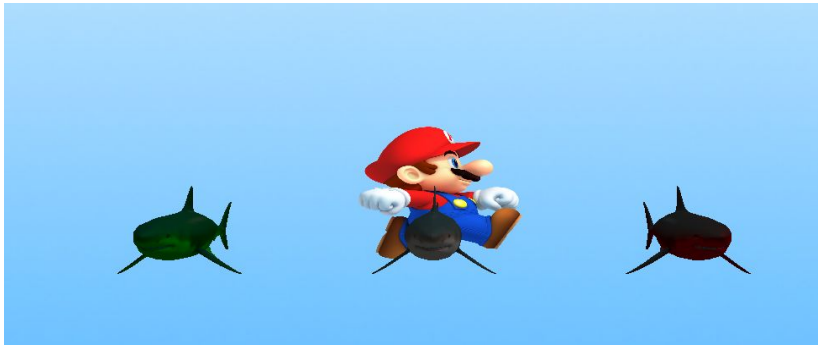
Lorsque le clic est soutenu, si la souris est déplacée, alors le roulis, tangle et lacet sont modifiés afin de représenter ce dit déplacement.

Si ce sont les flèches qui sont appuyées, c'est la position du personnage dans la scène qui change. Pour chaque modification du point de vue, la matrice correspondante doit être modifiée afin d'apporter le changement aux multiples shaders du programme.

Dans le code, Il est possible de retrouver les manipulation dans le fichier Scene.cpp.

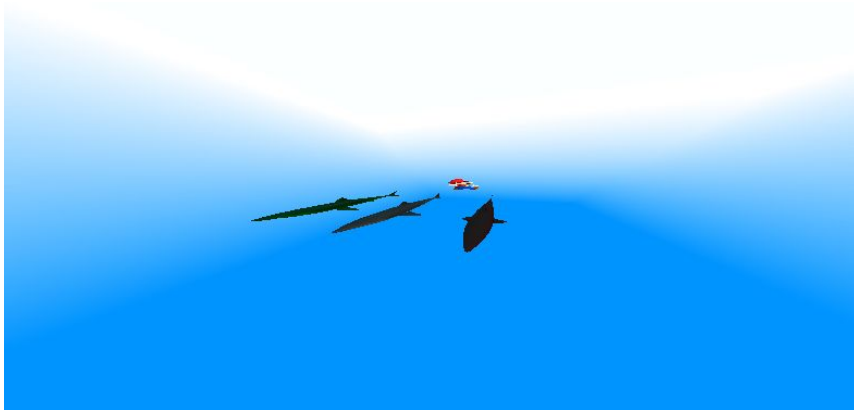
6.2 Mode de projection

Perspective



Orthographique

Perspective inverse



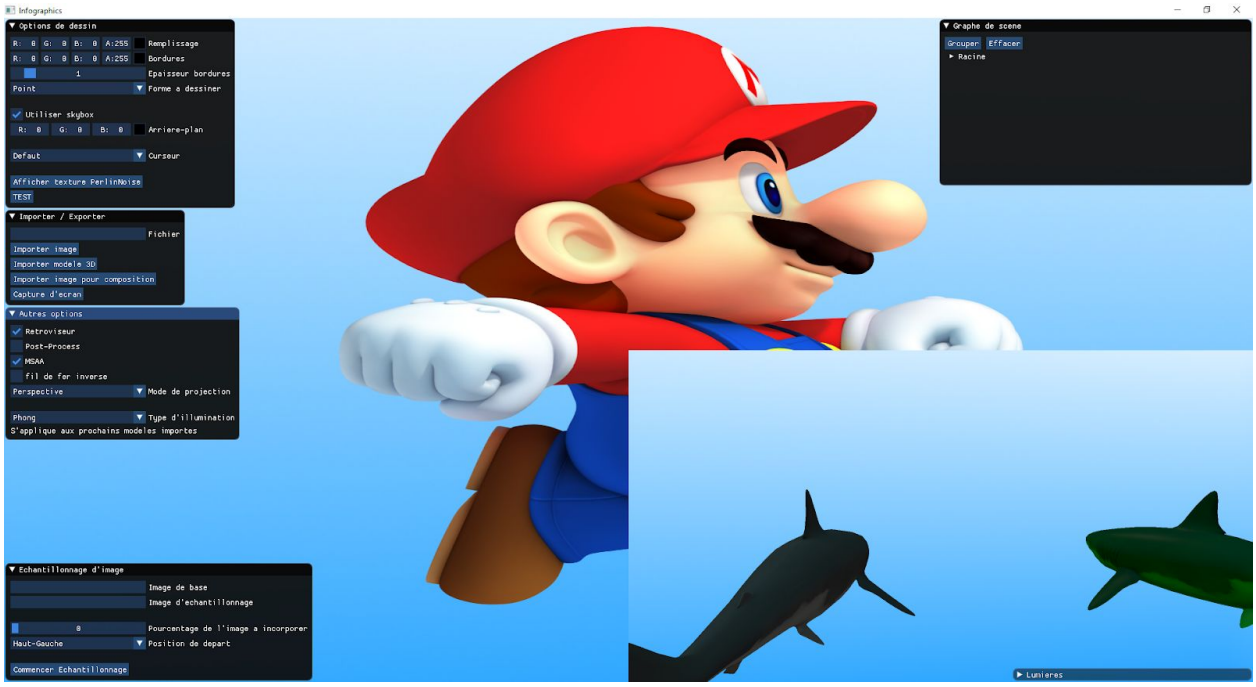
Trois types de perspectives sont disponibles et interchangeables à partir du menu autres options dans le menu déroulant mode de projection. Les deux premiers étant typiques, soit la perspective qui simule l'oeil humain et la projection orthographique qui montre une approche plus architecturale.

Il est à noter que lors d'un changement de perspective vers orthographique, les objets semblent plus gros, car la distance n'as pas d'impacte sur la taille de l'affichage. Ce changement peut alors apporter des confusions lorsque les objets affichés sont volumineux ou trop nombreux.

La troisième option est plus particulière, c'est un projection qui va à l'encontre de comment l'oeil humain fonctionne, offrant ainsi un effet intéressant sur la scène et une impression de vitesse. Une fois le mode choisi, une matrice correspondante à cette projection est envoyée aux divers shaders du programme afin d'ajuster le rendu de la scène.

Dans le code, il est possible de voir les manipulations dans Scene.cpp.

6.3 Agencement



La caméra de recul peut être activée à l'aide de la case à cocher se situant à la gauche dans le menu autres options.

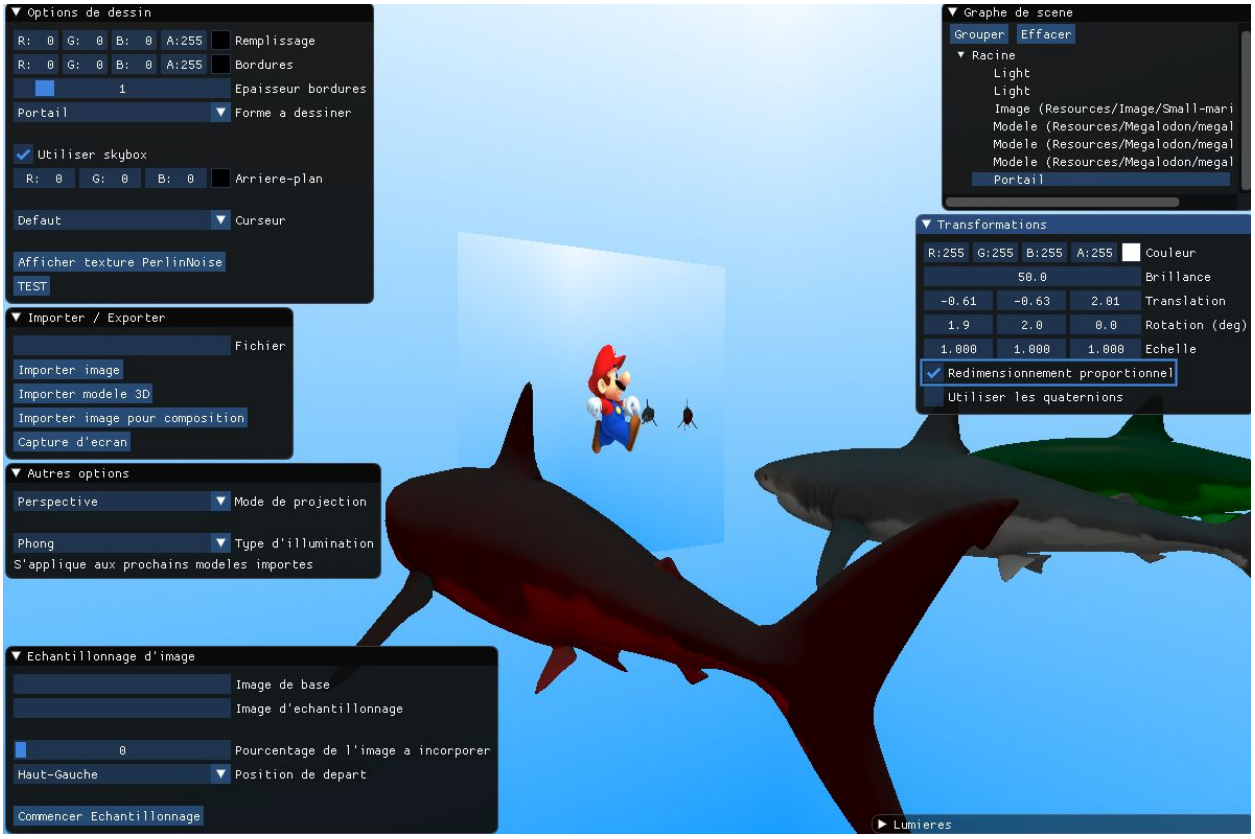
Le principe derrière son rendu est d'effectuer une rotation de 180 degrés en lacet (yaw), faire un rendu sur un framebuffer qui sera enregistré dans une texture. Une fois la texture prête, il suffit de la dessiner sur un carré affiché directement devant la caméra. Afin de ne pas influencer la scène de base, il faut ramener le lacet à son état initial en enlever le 180 degré lorsque la procédure est finie.

Dans le code, il est possible d'observer les manipulations dans Renderer.cpp.

6.4 Occlusion

L'occlusion de base de OpenGL a été activée à l'aide de glEnable(GL_CULL_FACE) pour diminuer le nombre d'éléments à rendre dans le graphe de scène. Son effet est simplement de ne pas rendre les faces dont la normale pointe dans la direction inverse de la caméra. Ainsi, on diminue théoriquement de moitié le nombre de faces à rendre pour un modèle 3D. Cependant, une conséquence est que les objets plats comme les images importées ne sont maintenant visible que d'un des deux côtés.

6.5 Portail



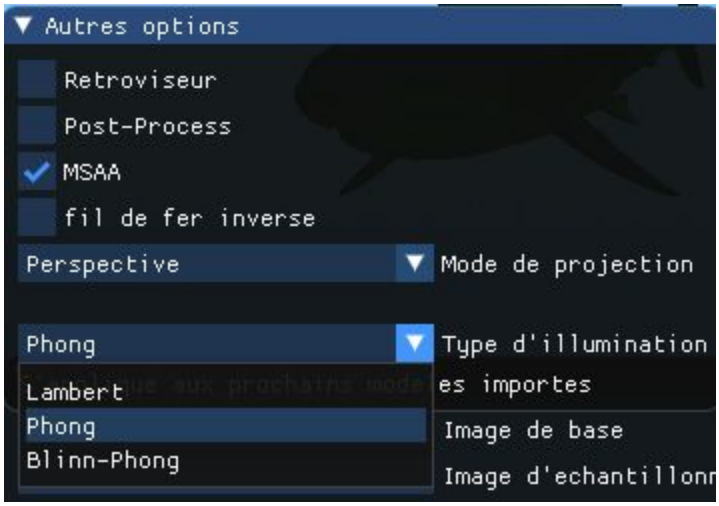
Le portail utilise un principe très similaire au miroir de recul agencé avec la technique utilisée pour rendre les lumières en dernier dans la scène.

Un vecteur des pointeurs vers chaque miroir est fait afin de faire leur rendu à la fin seulement. Pour les dessiner, un rendu est fait dans un framebuffer selon la position x,y,z du portail par rapport au personnage. Par choix de design, le portail se déplace avec le personnage afin de l'empêcher d'y entrer.

Un ajout intéressant serait de permettre aux autres portail de s'afficher lorsqu'ils sont visible dans un portail, ce qui ne devrait pas toujours être le cas avec cette implémentation.

Il est possible de voir les manipulation dans le fichier Scene.cpp et MirrorObject.cpp.

7.1 Modèles d’illumination



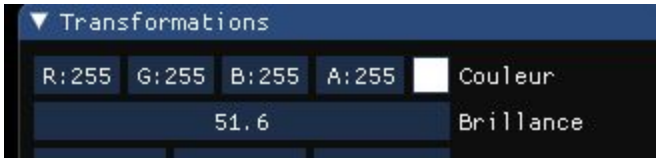
Trois modèles d’illumination sont disponibles. Lors de l’importation d’un modèle 3D, le choix du modèle d’illumination est choisi en fonction de celui sélectionné sur le menu déroulant type d’illumination dans le menu autres options. Une fois importé, le modèle 3D conservera le modèle d’illumination pendant toute sa durée de vie.

Ce choix de design permet d’avoir plusieurs objets ayant des modèles d’illumination différents de manière simultanée. Par exemple, dans la capture d’écran plus haut, nous pouvons voir que le modèle de gauche utilise le modèle de Lambert, celui du centre utilise Phong et lui de droite utilise Blinn-Phong.

Tous les objets de type ModelObject sont compatibles avec ces choix de type d'illumination. L'implémentation au niveau du programme est fait en utilisant 3 shaders différents chacun représentant son modèle d'illumination respectif.

Il est possible de voir les manipulations dans les shaders ModelShader, ModelShaderBlinnPhong, ModelShaderLambert.

7.2 Matériaux



Dans le programme, il est possible de changer le matériel d'un objet à partir du menu transformations. En changeant la valeur de brillance, plusieurs effets s'offrent à l'utilisateur, permettant ainsi une panoplie de matériaux commençant à métallique jusqu'à plastique.

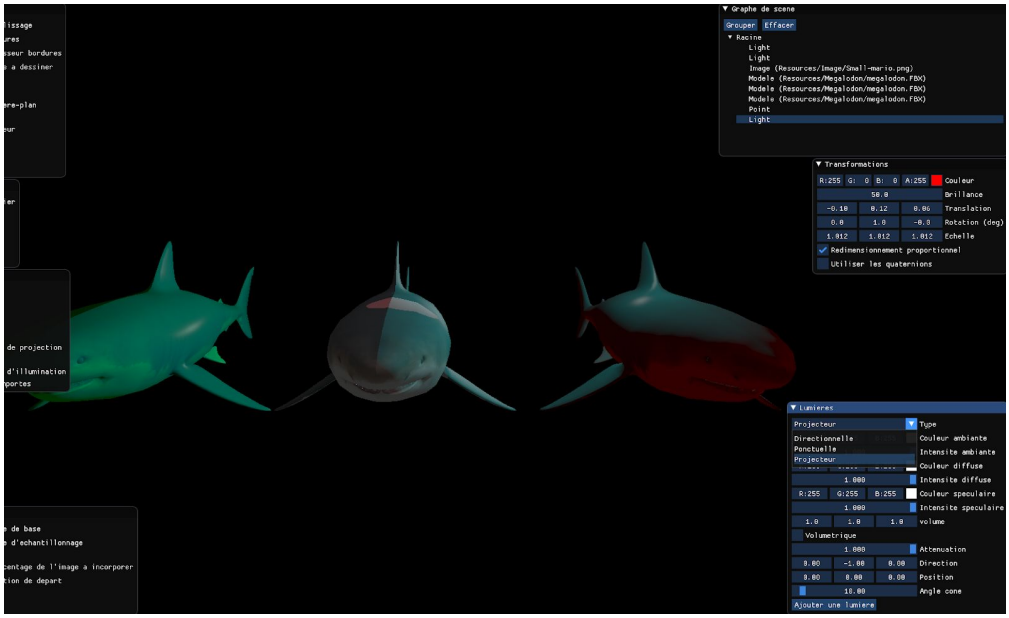
Il est aussi possible de modifier la couleur initiale de l'objet avec le les champs de couleur du même menu. La brillance est disponible pour tous les modèles de type ModelObject et la couleur est disponible pour presque tous les autres types.

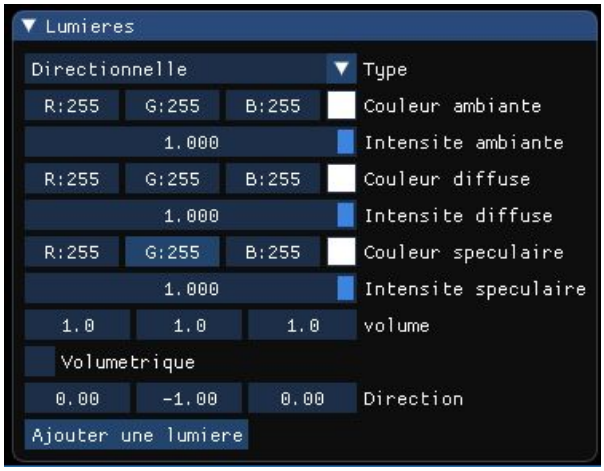
L'implémentation des matériaux est fait grâce à une valeur uniforme modulant l'impacte des lumières environnantes sur la réflexion spéculaire d'un objet et d'un vecteur uniforme qui offre une multiplication de base à la couleur de la texture de l'objet.

Il est possible, dans l'image plus haut, de voir que la même lumière affecte différemment les deux requins ayant chacun un matériel particulier.

Le code de cet objectif est dans les shaders et AbstractObject.cpp.

7.3 Types de lumière





Trois types de lumières sont accessibles à l'utilisateur, soit directionelle, poncutelle et projecteur.

Afin d'en ajouter à la scène, il suffit d'accéder au menu Lumières en bas à droite et de cliquer sur le bouton ajouter. Plusieurs paramètres sont présent afin de créer la lumière que désire l'utilisateur.

Dans ces paramètres, un menu déroulant type permet de choisir le type de la nouvelle lumière. Les trois types de lumières sont gérées par le même shader selon un paramètre uniforme représentant son type.

Il est à noter que les types de lumières sont implémentées sur les objets de type ModelObject.

Pour visualiser le code de cet objectif, il suffit de regarder dans les shaders de type ModelShader, dans LightObject, dans AbstractObject et dans Scene.

7.4 Volume de lumière

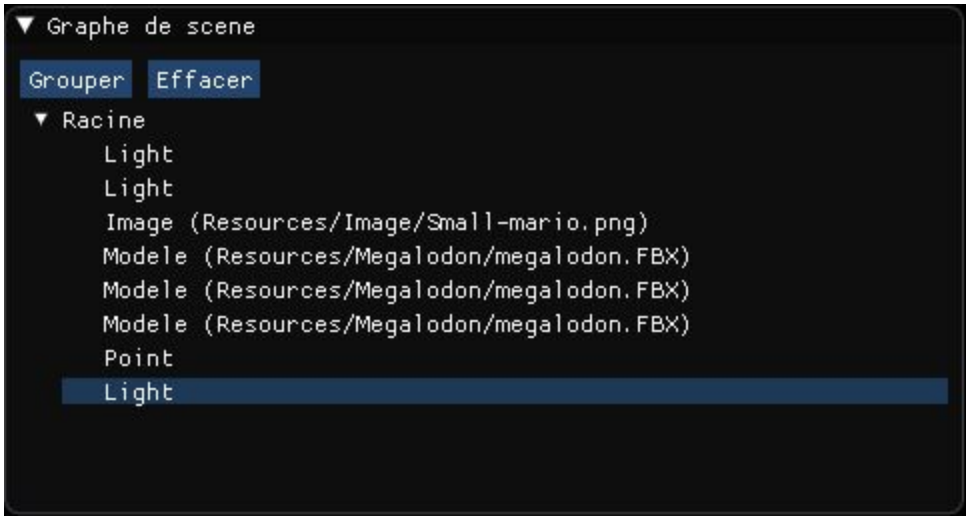
Le volume de lumière peut être activé lors de la création selon un vecteur volume qui représente un prisme où chaque valeur est la distance du centre du prisme vers un extremité en fonction d'un axe.

Autrement dit, chaque valeur du champ volume représente la longueur d'une demi arête. Cette fonctionnalité peut être facilement expérimentée avec une lumière ponctuelle à haute intensité.

Les volumes sont simplement une vérification de la distance de la lumière par rapport au pixel effectué dans le shader. Dans l'image plus haut, nous pouvons constater la démarcation laissée par le volume sur les requins du milieu et de gauche.

Le code est directement accessible dans les shaders de type ModelShader.

7.5 Lumières multiples



Comme représenté dans les précédentes captures, plusieurs lumières peuvent coexister de manière indépendantes. Cependant, lorsqu'aucune lumière n'est présente, les objets disparaissent, ce qui est un effet désiré pour représenter un effet de néant. Si il est désiré, les paramètres des lumières permettent de simuler tous les effets d'un cycles jour/nuit grâce aux menus d'intensité et de couleur de chaque effet lumineux. Afin que les lumières aient un impact sur tous les objets de type ModelObject, un vecteur dans scene conserve un pointeur vers chaque objet lumière, ce qui permet ainsi de transmettre ces lumières à chaque objet lorsqu'il est dessiné. Toutes les lumières sont donc passées dans une gros tableau uniforme de structure lumière au shader afin d'être considérée lors du rendu. Une fois que les couleurs de toutes les lumières sont ajoutées, une réduction est fait pour réajuster à un niveau respectable la couleur finale.

Pour visualiser le code de cet objectif, il suffit de regarder dans les shaders de type ModelShader, dans LightObject, dans AbstractObject et dans Scene.

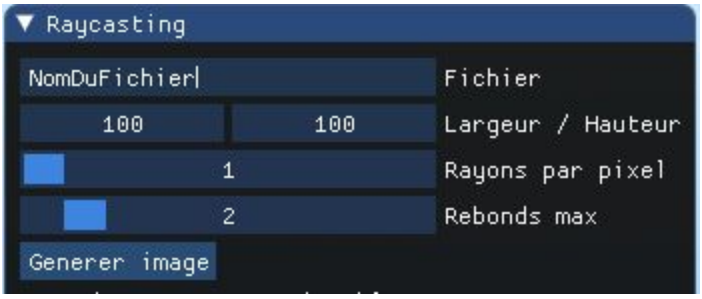
8.1 Intersection

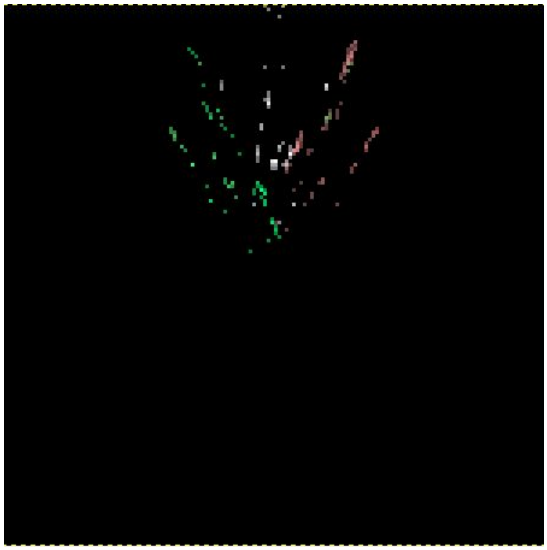
Un algorithme de raytracing de base a été implémenté en s'inspirant de l'exemple fournie par le professeur (Plus de détails à ce sujet à la section 8.5). Cet algorithme, bien qu'incomplet, est fonctionnel pour ce qui a trait à la détection d'intersection entre un rayon et certains des objets pouvant être instanciés dans notre graphe de scène. Plus spécifiquement, la détection d'intersection est implémentée pour les *cubes* (pouvant être instanciés à l'aide de la fenêtre *Options de dessin*), ainsi les objets de type *Modèle 3D* (pouvant être importés).

Pour les cubes, nous avons utilisé l'algorithme de calcul d'intersection entre un rayon et une boîte, tel que décrit à la section 8.3.4 des notes de cours. Son implémentation est visible dans la méthode CubeObject::raycast(). Finalement, pour ce qui est des modèles 3D, notre manière de procéder est d'itérer à travers chacun des triangles composant les différents *mesh* du modèle, puis en utilisant l'algorithme de calcul d'intersection entre un rayon et un triangle, tel

que décrit à la section 8.3.6 des notes de cours. Celui-ci a été implémenté dans ModelObject::raycast().

8.2 Réflexion
8.3 Réfraction
8.4 Ombrage
8.5 Illumination globale



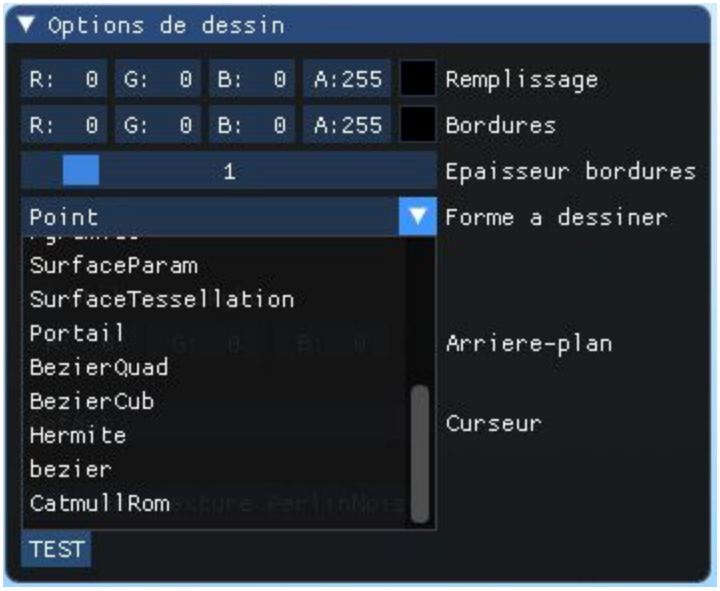


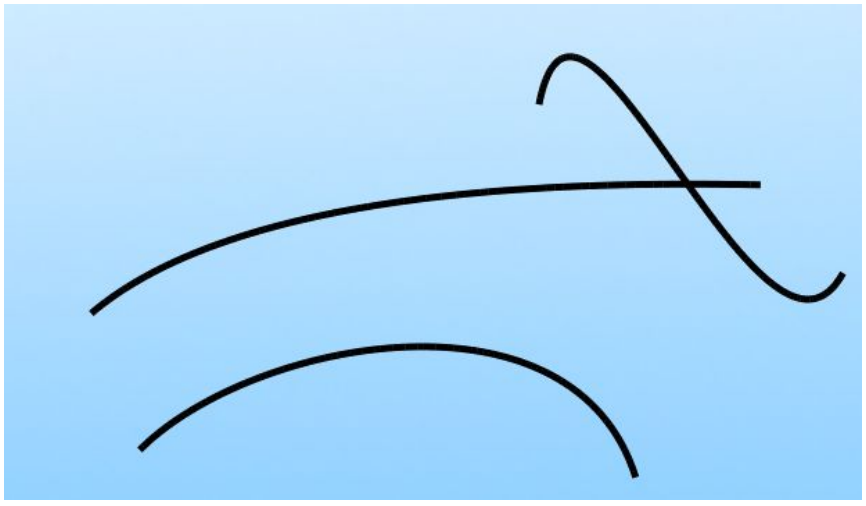
L'algorithme de raytracing mentionné à la section 8.1 utilise certaines notions d'illumination globale pour effectuer son calcul. En effet, puisque l'algorithme supporte des rayons qui effectuent plusieurs rebonds, les couleurs de l'image résultante prennent en compte l'impact des reflets de lumière sur les surfaces avoisinantes. Cette fonctionnalité est accessible dans la fenêtre *Raycasting* du GUI. Elle offre de sélectionner le nom du fichier, la taille de l'image à générer, le nombre de rayons à être envoyés par pixel, ainsi que le nombre de rebonds maximum qu'un rayon peut effectuer. Lorsqu'on clique sur *Generer image*, le processus commence à générer une image de la scène actuelle à partir du point de vue actuel de la caméra. On peut suivre le progrès actuel dans la console de l'application.

Malheureusement, cette fonctionnalité ne fonctionne pas parfaitement dans la version actuelle du logiciel, dû principalement au manque de temps. L'image ci-haut a été générée avec le point de vue par défaut et les trois requins présents dans la scène au démarrage de l'application, au format 150x150, avec 2 rayons par pixel et 2 rebonds maximum. Comme on peut le constater,

les bonnes couleurs sont présentes dans le bon ordre (vert à gauche, gris au centre, rouge à droite), mais leurs positions dans l'image ne sont pas exactes, probablement à cause d'une erreur dans le calcul des directions des rayons, et il manque de grosses portions à l'image. L'implémentation de cette fonctionnalité peut être consultée dans les méthodes Scene::renderRaycast(), ainsi que toutes les autres méthodes appelées par cette dernière.

9.1 Courbe cubique



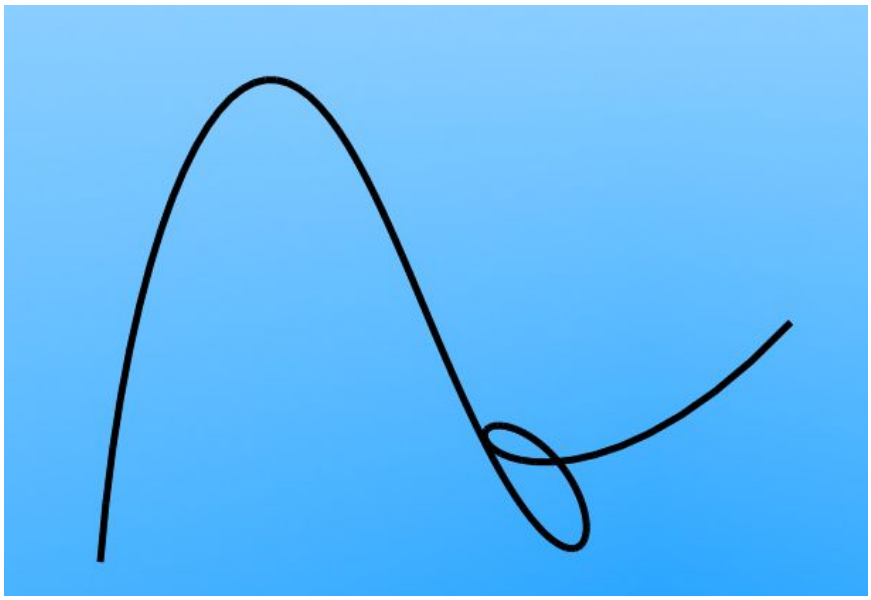


Les courbes paramétriques cubiques sont dessinées commes les lignes, il faut sélectionner le type de courbe dans le menu d'option de dessin et cliquer sur l'écran aux emplacement correspondant aux paramètres de la courbe.

En réalité, les courbes sont plusieurs instances de l'objet PrimitiveObject dessinées selon un nombre de segments et où devrait se situer ce segment dans la formule respective de la courbe paramétrique. Le cumulatif de toutes ces lignes forme la courbe qui sera ensuite affichée à l'écran et suivra le personnage dans la scène.

Il est possible d'observer le code de ces courbes dans ParametricCurveObject.cpp.

9.2 Courbe paramétrique



```
glm::vec3 ParametricCurveObject::bezier(std::vector<glm::vec3> pt, float t)// would work approximately up to 20 before running out of memory
{
    unsigned int n = pt.size();
    glm::vec3 point=glm::vec3(0,0,0);
    for (unsigned int i = 0; i < n; ++i)
    {
        double a = (factorial(n) / (factorial(i)*factorial(n - i)))*pow(t, i)*pow(1 - t, n - i);
        point.x += (float)(a * pt[i].x);
        point.y += (float)(a * pt[i].y);
        point.z += (float)(a * pt[i].z);
    }
    return point;
}
```

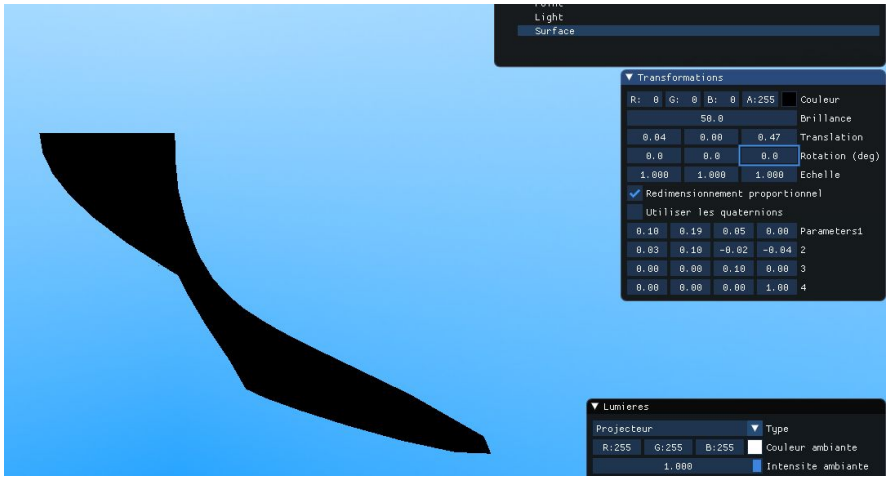
Comme pour les courbes paramétriques cubiques, ces autres courbes paramétriques doivent être sélectionnées dans le menu d'option de dessin et sont plusieurs objets PrimitiveObjects dessinés bout à bout.

La différence est que par exemple la courbe de Bézier peut accumuler les points de paramètres jusqu'à manquer de mémoire pour son calcul, ce qui correspond à environ 20 points paramétriques.

Bref, une implémentation directe et non approximée de la courbe de Bézier est fait, ce qui apporte une plus grande précision, mais apporte en contrepartie ce problème de mémoire lorsque de grosse expressions factorielles sont en jeu.

Il est aussi possible d'observer le code de ces courbes dans ParametricCurveObject.cpp.

9.3 Surface paramétrique




```
glm::vec3 ParametricSurfaceObject::bezierBicubic( glm::mat4 G,float u, float v)
{

    glm::vec4 U = glm::vec4(1.0, u, pow(u, 2), pow(u, 3));

    glm::mat4 matrix;
    matrix[0][0] = 1.0f;        matrix[0][1] = 0.0f;          matrix[0][2] = 0.0f;        matrix[0][3] = 0.0f;
    matrix[1][0] = -3.0f;       matrix[1][1] = 3.0f;          matrix[1][2] = 0.0f;        matrix[1][3] = 0.0f;
    matrix[2][0] = 3.0f;        matrix[2][1] = -6.0f;         matrix[2][2] = 3.0f;        matrix[2][3] = 0.0f;
    matrix[3][0] = -1.0f;       matrix[3][1] = 3.0f;          matrix[3][2] = -3.0f;       matrix[3][3] = 1.0f;


    glm::vec4 V = glm::vec4(1.0, v, pow(v, 2), pow(v, 3));

    return U* glm::transpose(matrix)*G*matrix*V;
}
```

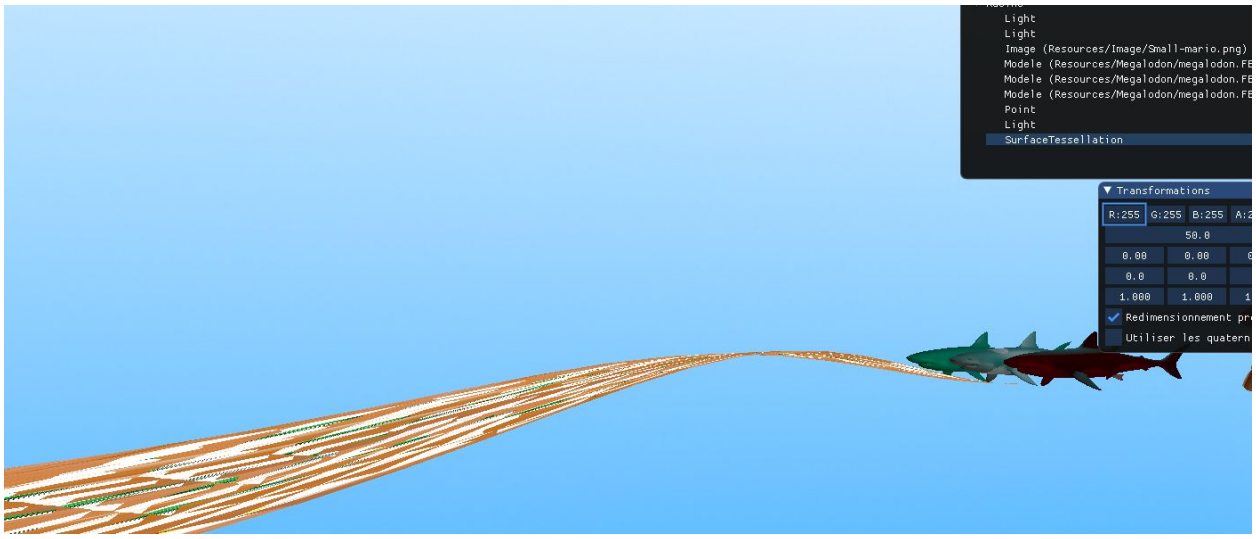
Pour la surface paramétrique, une version modifiée de la formule de Bézier bicubique mise sous forme de matrice est utilisée afin de rendre les paramètres disponibles à l'utilisateur de façon plus conviviale.

Autrement dit, lorsqu'un objet surface paramétrique est ajouté avec le menu déroulant pour les formes à ajouter et qu'il est par la suite sélectionné dans le graphe de scène, une menu personnalité représentant sa matrice de paramètres apparaît.

Il est alors possible de contorsionner la surface selon les besoins de la cause.

Le code de cet objectif est disponible dans ParametricSurfaceObject.cpp.

9.4 Shader de tesselation



La tessellation s'effectue sur deux nouveaux shaders qui doivent être attachés en plus de ceux de base, soit le shader d'évaluation et le shader de contrôle. Grâces à ces deux entitées, de nouvelles coordonnées peuvent être créées pour une surface donnée.

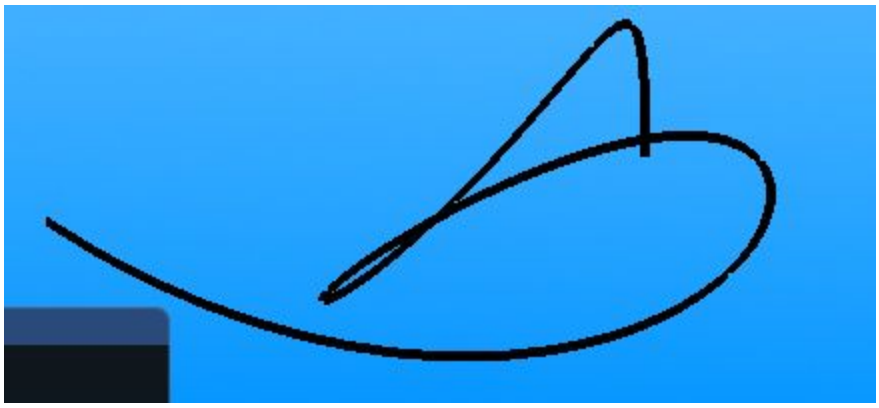
Dans la capture plus haut, nous partons d'un QuadObject que nous passons dans le shader de tessellation afin d'obtenir un résultat assez surprenant. Pour créer cette même surface dans le projet, il suffit d'accéder au menu d'option de dessin, choisir SurfaceTessellation dans la liste et cliquer à un endroit quelconque sur l'écran.

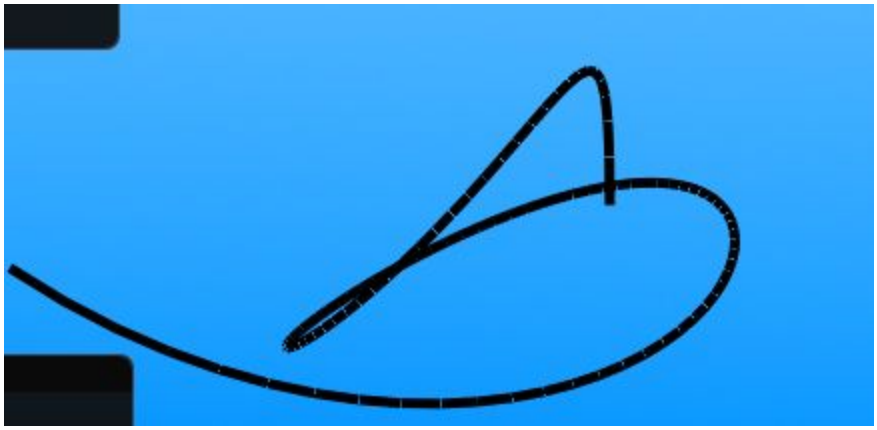
Il est alors possible d'expérimenter différents effets intéressants comme lui implémenté en modifiant les shaders de tessellation (TessellationShader et TessellationCEShader).

Pour voir le code de ce livrable, il suffit de se diriger vers les shaders de tesselation, tessellationQuad.cpp et faire connaissance de la modification fait sur ShaderLoader.cpp.

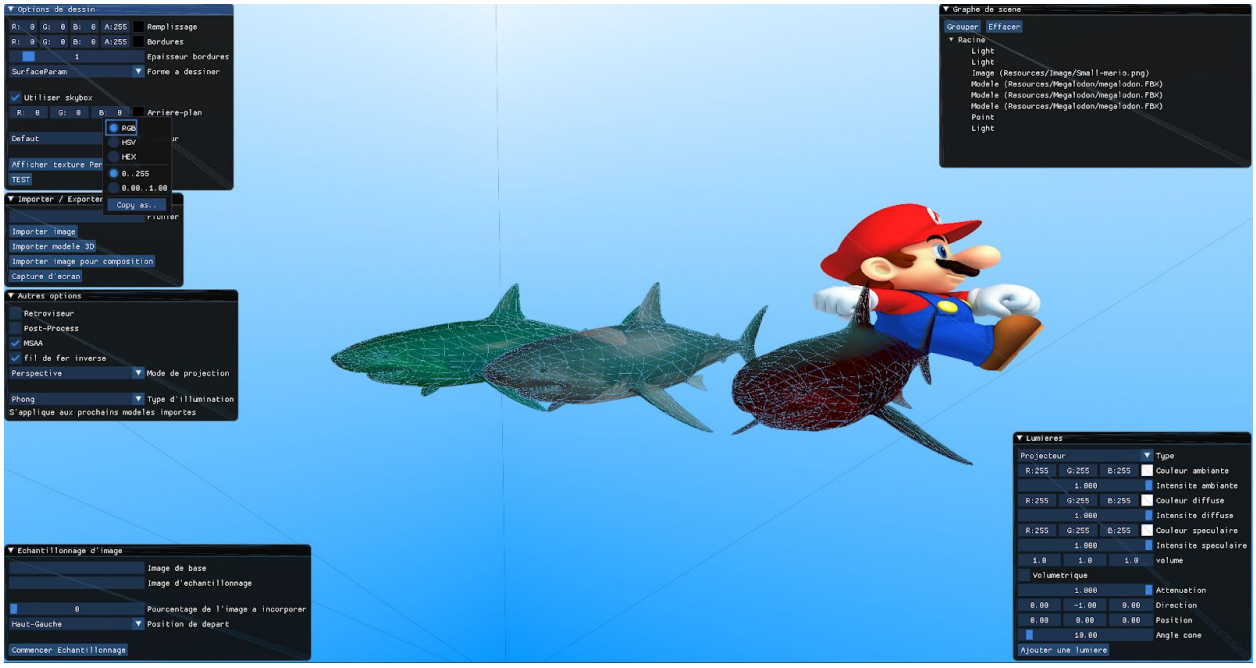
9.5 Triangulation

10.1 Effet en pleine fenêtre





Il est possible d'activer l'anti-aliasing de OpenGL en cochant l'option MSAA du menu autres options. Il est particulièrement facile de voir une différence dans les lignes dessinées qui deviennent moins pixelisées.



L'option des lignes de fer inversées(wireframe) est une découverte accidentelle d'une particularité d'OpenGL qui ne semble pas volontaire de leur part. En lissant les surfaces des polygones, nous pouvons obtenir une démarcation entre chaque polygone des modèles. Ceci offre alors une outil intéressant pour les tests lors de l'implémentation de nouvelles fonctionnalitées. Il est possible d'activer ce mode en cochant l'option fil de fer inversé du menu autres options.

Un post-process intéressant peut être activé à l'aide de l'option post-process du menu autres options. Ce processus consiste à faire une rendu de la scène dans un framebuffer, et passer le résultat dans une texture qui sera affiché sur un rectangle de la taille de la fenêtre à l'aide d'un shader. Ce post-process affiche l'image reproduite 4 fois et ajoute une teinte rouge à toute la texture.

10.2 Effet de relief

10.3 BRDF

10.4 Rendu en différé

10.5 Style libre

L'élément additionnel que nous avons choisi d'implémenter est un effet de brouillard ou de brume. Pour l'activer, il suffit de cocher la case *Activer le brouillard* dans la fenêtre *Autres options* du GUI. L'effet résultant est que plus les objets rendus sont loins de la caméra, plus leur couleur se confond avec l'arrière-plan, qui donne l'impression d'un épais brouillard. Pour l'implémentation, il a fallu ajouter une variable booléenne dans chacun de nos shaders, qui, lorsqu'activée, calcule la distance du fragment par rapport à la caméra et mélange sa couleur à celle de l'arrière-plan en utilisant la distance calculée comme facteur.

Ressources

La règle générale est que tous les fichiers du dossier Src sont **faits par nous** sauf pour ceux provenant de ImGui.
Voici quand même une liste non exhaustive:

- Main
- Application
- Renderer
- Scene
- AbstractObject
- GroupObject
- QuadObject
- CubeObject
- SkyboxObject
- PrimitiveObject
- ModelObject
- AbstractShader
- ShaderLoader
- Shaders du fichier Src
- CMakeList.txt
- PerlinNoise

Fait par nous, mais **grandement inspiré** de: https://learnopengl.com/

- Model
- Mesh

Matrice de perspective :
http://www.geeks3d.com/20090729/howto-perspective-projection-matrix-in-opengl/
Matrice Bezier bicubique: https://www.cs.uky.edu/~cheng/cs535/Notes/CS535-Curves-3.pdf

Ressources externes:

ImGui : https://github.com/ocornut/imgui
SDL: https://www.libsdl.org/
Glew: https://github.com/nigels-com/glew
Cmake : https://cmake.org/
Assimp:https://github.com/assimp/assimp
GLM:https://glm.g-truc.net/0.9.8/index.html

Tutoriels:
post-process: https://en.wikibooks.org/wiki/OpenGL_Programming/Post-Processing

Présentation

Mathieu Dubreuil fait un bac en informatique. Il adore l'intelligence artificielle et aimerait éventuellement aller faire de la recherche dans ce domaine. Il aime bien apprendre de nouvelles choses. Prêt à tester les éventuels livrables qui lui sont proposés par son équipe, il s'adapte assez rapidement au domaine assez complexe qu'est l'infographie sans expérience préalable dans le milieu.

Samuel Baillargeon est à sa 2e session du bac en informatique (IFT). Il a suivi le parcours DEC-BAC offert en collaboration entre le Cégep de Lévis-Lauzon et l'Université Laval. Au Cégep, il a eu l'occasion de faire beaucoup de C++ et a également suivi un cours d'infographie utilisant C++ et OpenGL durant son DEC en informatique industrielle. Il a eu la chance de travailler chez Creaform (scanneurs 3D) lors d'un de ses stages et apprécie grandement le domaine, ainsi que les mondes virtuels. En accomplissant ce cours, il espère poursuivre avec le cours programmation de jeux vidéo, puisque c'est un domaine qui le passionne et dans lequel il aimerait un jour travailler.

Alexandre Mercier-Aubin est diplômé du DEC technique en informatique industrielle et poursuit son parcours scolaire à l'Université en IFT. Il souhaite principalement bâtir sur les connaissances techniques du cégep afin de lui donner une base théorique forte pour le préparer au marché du travail. Il aura la chance de faire un stage chez Activision dans le département Engine l'été prochain et considère le cours comme une bonne préparation aux éventuels défis du domaine.