



Visualiseur interactif de scènes 3D

Document de design – Remise TP2

présenté à
Philippe Voyer

par
Équipe 4

<i>matricule</i>	<i>nom</i>	<i>signature</i>
111 127 868	Jérémie Bolduc	
111 126 561	Gabriel Chantal	
111 152 662	Alex Gilbert	
111 130 693	Alexandre McCune	
111 133 069	Tania Toloza	

Université Laval
12 mars 2017

Historique des versions		
<i>version</i>	<i>date</i>	<i>description</i>
1.0	16 février 2017	Création du document
1.1	12 mars 2017	Mise à jour du document pour la remise

Table des matières

Table des figures	iii
1 Sommaire	1
2 Interactivité	2
3 Technologie	3
4 Architecture	4
5 Fonctionnalités	6
5.1 Image	6
5.1.1 Importation	6
5.1.2 Exportation	6
5.1.3 Espace de couleur	6
5.1.4 Traitement d'image	7
5.1.5 Image procédurale	8
5.2 Dessin vectoriel	9
5.2.1 Curseur dynamique	9
5.2.2 Primitives vectorielles	10
5.2.3 Formes vectorielles	11
5.2.4 Outils de dessin	12
5.2.5 Interface	13
5.3 Transformation	16
5.3.1 Transformation interactive	16
5.3.2 Structure de scène	16
5.3.3 Sélection multiple	20
5.3.4 Coordonnées non cartésiennes	23
5.3.5 Historique	23
5.4 Géométrie	24
5.4.1 Particules	24
5.4.2 Primitives	25
5.4.3 Modèle	28
5.4.4 Texture	31
5.4.5 Géométrie procédurale	31
5.5 Caméra	32
5.5.1 Propriétés de caméra	32
5.5.2 Mode de projection	32
5.5.3 Caméra interactive	32
5.5.4 Caméra multiple	33

5.5.5	Caméra animée	33
5.6	Pipeline de rendu	34
5.6.1	Portabilité	34
5.6.2	Shader de géométrie	34
5.6.3	Rétroaction de transformation	34
5.6.4	Passes de rendu	34
5.6.5	Techniques d'occlusion	34
5.7	Illumination	35
5.7.1	Types de lumière	35
5.7.2	Lumières multiples	35
5.7.3	Matériaux	35
5.7.4	Modèle d'illumination	35
5.7.5	Volume de lumière	35
5.8	Lancer de rayon	36
5.8.1	Intersection	36
5.8.2	Réflexion	39
5.8.3	Réfraction	41
5.8.4	Ombrage	41
5.8.5	Rendu graphique	41
5.9	Topologie	42
5.9.1	Courbe cubique	42
5.9.2	Courbe paramétrique	42
5.9.3	Surface paramétrique	43
5.9.4	Shader de tessellation	43
5.9.5	Triangulation	43
5.10	Techniques de rendu	44
5.10.1	Effet de relief	44
5.10.2	Cube de réflexion	44
5.10.3	BRDF	44
5.10.4	Effet en pleine fenêtre	44
5.10.5	Style libre	44
6	Ressources	45
7	Présentation	46
7.1	Liste des membres de l'équipe	46
7.2	Présentation	46

Table des figures

4.1	Diagramme de package de l'application	5
5.1	Une scène avec filtres.	8
5.2	Curseur normal de type "flèche"	9
5.3	Curseur pour bouton de type "main"	9
5.4	Curseur slider de type "slider"	9
5.5	Une scène avec primitives 2D.	11
5.6	Cornet de crème glacée !	12
5.7	Exemple d'interface	15
5.8	Menu des transformations.	16
5.9	Menu de sélection	20
5.10	Plusieurs entités sélectionnées	20
5.11	La sélection est en «wireframe»	21
5.12	Chaque entité de la sélection reçoit la transformation	22
5.13	Options de création d'une primitive	25
5.14	Bouton pour importer un modèle	29
5.15	Propriété de la caméra dans l'interface	32
5.16	Cube de type miroir	39
5.17	Les miroirs prennent la caméra en compte	39
5.18	On voit que le fond et chaque cube sont plus pâles que les vrais	40
5.19	Évidemment, la couleur de fond choisie par l'utilisateur est parfaitement compatible avec les miroirs	40
5.20	Après un mouvement de caméra, le miroir ne s'est pas mis à jour à chaque frame, pour éviter des délais de plusieurs minutes	41
5.21	L'utilisateur appuyé sur une touche, attend la génération de la frame, et le miroir est maintenant à jour !	41

Chapitre 1

Sommaire

Cette application permet de construire, éditer et rendre des scènes 3D. Notre équipe a développé plusieurs fonctionnalités en lien avec le traitement d'image, le dessin vectoriel, les transformations géométriques, la géométrie et le concept de caméra.

Ce projet vise à développer un visualiseur interactif basé sur les notions de l'infographie. Nous avons, donc implémenté des fonctionnalités qui permettent à l'utilisateur de construire des scènes avec différentes entités géométriques. Des entités telles que des cubes, triangles, lignes, points, cercle et des carrés peuvent être construits selon des paramètres choisis par l'utilisateur.

L'utilisateur peut construire des scènes interactivement et exporter des images.

Dans ce document, les différentes fonctionnalités seront détaillées et des parties du code seront présentées. Un diagramme présentera l'ensemble des fonctions de l'application. De cette façon, une vue d'ensemble pourra bien d'écrire l'architecture de l'application.

Nous présentons également les technologies que nous avons utilisées pour développer le visualiseur interactif de scène 3D.

Chapitre 2

Interactivité

L'option «Dessiner Primitive» permet d'ajouter une primitive dans la scène. L'utilisateur peut choisir les valeurs pour la couleur du fond de la scène.

L'application permet d'importer des modèles et de les représenter en 3D. Il est aussi possible d'exporter à l'aide de l'option Exportation en image.

L'option «Type» permet à l'utilisateur de choisir le type de primitive voulu soit en 2D, soit en 3D.

Pour les primitives en 3D il est possible de choisir la position en X, Y et Z. La taille est définie par la hauteur, largeur et profondeur.

Pour les primitives en 2D il est possible de choisir la position en X et Y. La taille est définie par la hauteur et largeur.

L'utilisateur peut également modifier la couleur de remplissage des primitives choisi, la couleur de la bordure et l'épaisseur des traits.

L'option «Parametre de la camera» permet de définir tous les paramètres de la caméra de l'application.

L'option «Traitement d'image» permet de soit brouiller, inverser ou dilater les éléments de la scène.

Finalement sur le gauche de l'application il est possible de modifier les valeurs de la translation, de la rotation et de la proportion.

Chapitre 3

Technologie

L'application a été développée en C++ avec Visual Studio. Nous avons choisi de travailler avec OpenFrameworks, une boîte à outil idéal pour les rendus 3D. Nous avons développé l'interface avec la librairie ofxGui. L'application utilise aussi AssimpModelLoader pour charger les modèles 3D, ainsi que OpenCv pour les filtres des images. Des extensions comme ofxBezierSurface ou ofxCvImage ont été ajoutées dans le but de répondre à tous les critères.

Chapitre 4

Architecture

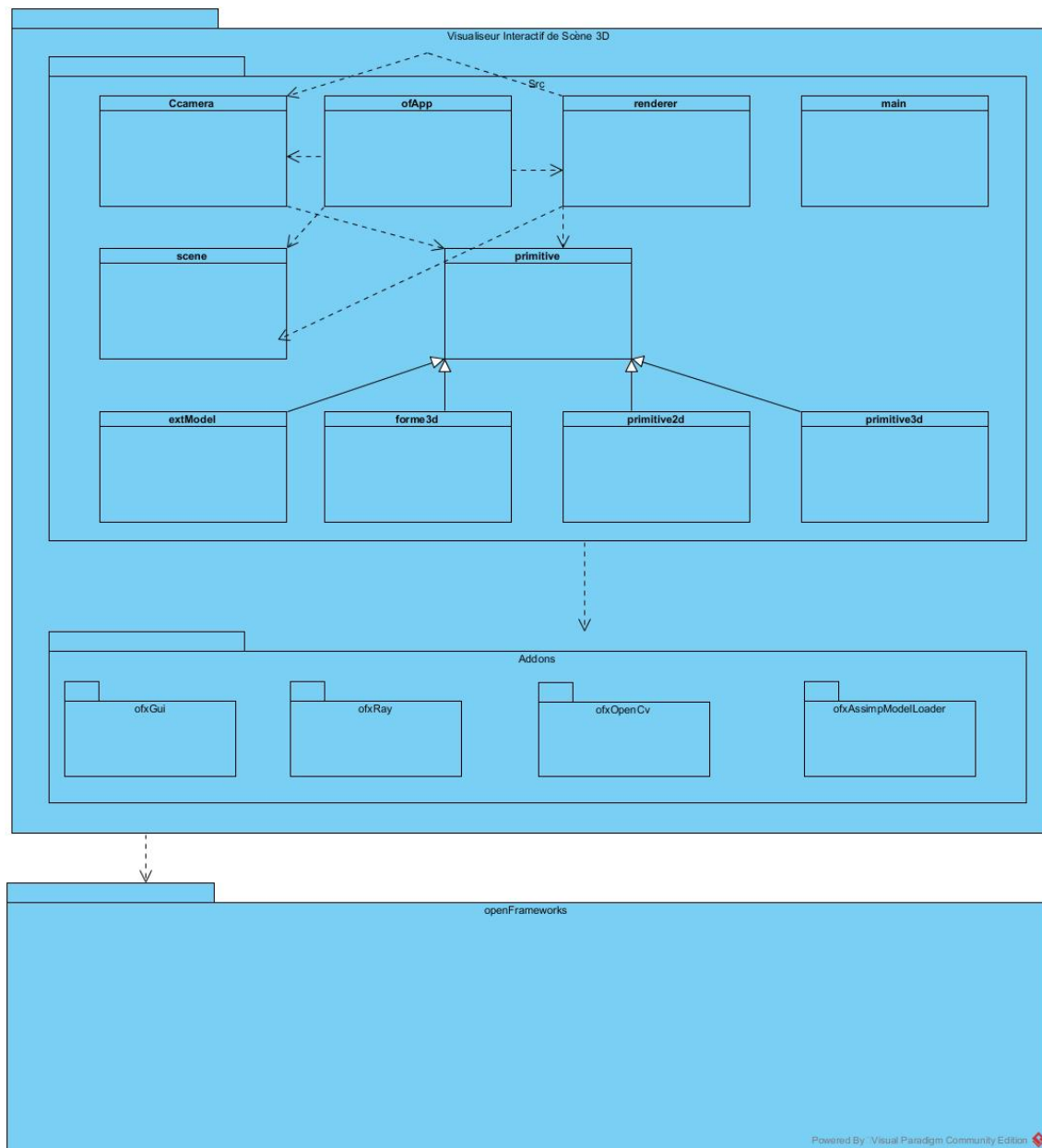


FIGURE 4.1 – Diagramme de package de l'application

Chapitre 5

Fonctionnalités

5.1 Image

5.1.1 Importation

Non implémenté

5.1.2 Exportation

L'exportation en image est implémentée grâce à la méthode `imageExport` de la classe `render`. Tout d'abord, on dessine la scène, sans l'interface graphique. Ensuite, on sauvegarde une capture de l'écran avec le nom, un «time stamp», pour que le fichier soit unique, et l'extension reçue en paramètre.

```
void renderer::imageExport(const string name, const string extension)
{
    draw();

    ofImage imageTemp;

    string timestamp = ofGetTimestampString("-%y%m%d-%H%M%S-%i");

    string fileName = name + timestamp + "." + extension;

    imageTemp.grabScreen(0, 0, ofGetWidth(), ofGetHeight());

    imageTemp.save(fileName);

    ofLog() << "<export image: " << fileName << ">";
}
```

5.1.3 Espace de couleur

L'espace de couleur utilisé par l'application est le HSB. Ainsi, pour chaque couleur à saisir, trois «sliders» sont disposés dans l'interface pour saisir la teinte (Hue), la saturation (Saturation) et la valeur (Brightness).

En soi, à chaque mise à jour de l'application, l'application met à jour les valeurs des attributs de couleur dans la classe `render`. Ainsi, à chaque instant, le `render` a les couleurs entrées par l'utilisateur.

Voici un exemple d'initiation des valeurs pour la couleur de remplissage :

```
fillHue.setName("Teinte");
fillHue.setMin(0);
fillHue.setMax(255);
fillHue.set(0);

fillSaturation.setName("Saturation");
fillSaturation.setMin(0);
fillSaturation.setMax(255);
fillSaturation.set(100);

fillBrihtess.setName("Valeur");
fillBrihtess.setMin(0);
fillBrihtess.setMax(255);
fillBrihtess.set(255);

fillAlpha.setName("Transparence");
fillAlpha.setMin(0);
fillAlpha.setMax(255);
fillAlpha.set(255);
```

Voici la méthode qui génère les couleurs à partir de l'espace HSB :

```
void ofApp::setColors()
{
    stroke = ofColor::fromHsb(strokeHue, strokeSaturation, strokeBrihtess, strokeAlpha);
    fill = ofColor::fromHsb(fillHue, fillSaturation, fillBrihtess, fillAlpha);
    background = ofColor::fromHsb(bgHue, bgSaturation, bgBrihtess);
}
```

Voici la méthode qui met à jour les couleurs du renderer (et l'épaisseur de lignes) :

```
void ofApp::setRendererParameter() {

    rend->stroke = stroke;
    rend->fill = fill;
    rend->background = background;

    rend->strokeThickness = strokeThickness;
}
```

5.1.4 Traitement d'image

Une catégorie située au coin droit de l'écran permet l'utilisation de plusieurs filtres s'affichant sur l'entièreté de la scène. Elle regroupe trois types de filtres : le brouillage, l'inversion des couleurs et la dilatation. Les trois filtres peuvent être appliqués tous en même temps ou un à la fois.

Un objet `ofxCvColorImage` est utilisé pour la réalisation des filtres. On capture les pixels de la scène et on l'ajoute dans cet objet, pour ensuite lui faire subir les filtres sélectionnés par l'utilisateur. Les fonctions `blur()`, `invert()` et `dilate()` ont été utilisées pour construire le filtre. Ces fonctions utilisent différentes opérations sur chaque pixel de l'image pour les modifier.

Voici une image d'une scène avec des filtres d'inversement de couleurs et de brouillage :



FIGURE 5.1 – Une scène avec filtres.

5.1.5 Image procédurale

Non implémenté

5.2 Dessin vectoriel

5.2.1 Curseur dynamique

Dans notre application il y a une interface graphique créée avec ofxGui. Cette technologie, malgré le fait qu'elle soit très pratique pour nos besoins, contient des boutons, des cases à cocher, des "sliders", des groupes, mais le curseur était toujours identique (en forme de flèche), ce qui ne rendait pas évident avec quoi on peut ou ne peut pas interagir.

Nous avons donc ajouté, en utilisant les événements `mouseMoved()` de notre fenêtre et des différents contrôles, une modification dynamique du curseur selon au-dessus de quoi il se trouve.

Au a donc trois curseurs différents.

Le curseur normal, qui est présent quand la souris est dans l'espace de dessin

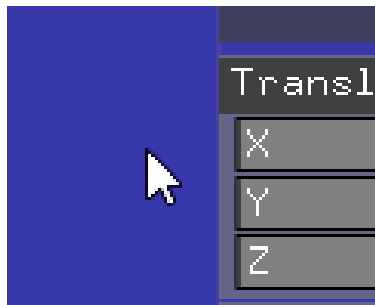


FIGURE 5.2 – Curseur normal de type "flèche"

Le curseur de type "main", qui est présent quand la souris est au-dessus d'un bouton ou d'une case à cocher

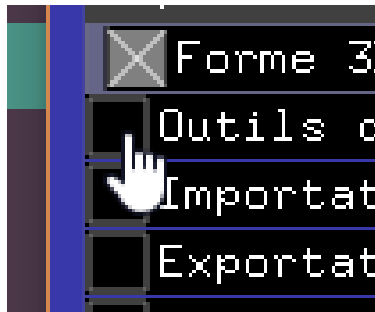


FIGURE 5.3 – Curseur pour bouton de type "main"

Et le curseur de type "slider", qui est présent quand la souris est au-dessus d'un contrôle du même nom.

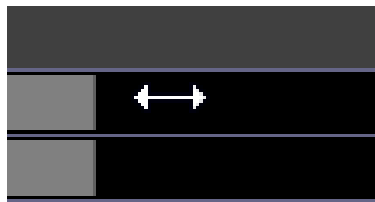


FIGURE 5.4 – Curseur slider de type "slider"

```

void ofApp::mouseMoved(int x, int y) {
    HCURSOR curs;
    if (!cursorIsInControl(x, y))
    {
        curs = LoadCursor(NULL, IDC_ARROW);
        SetCursor(curs);
    }
}

bool ofAppButton::mouseMoved(ofMouseEventArgs & args){

    HCURSOR curs;
    if (cursorIsInControl(args.x, args.y))
    {
        curs = LoadCursor(NULL, IDC_HAND);
        SetCursor(curs);
    }
    return ofAppToggle::mouseMoved(args);
}

template<typename Type>
bool ofAppSlider<Type>::mouseMoved(ofMouseEventArgs & args){
    mouseInside = isGuiDrawing() && b.inside(ofPoint(args.x,args.y));

    HCURSOR curs;
    if (mouseInside)
    {
        curs = LoadCursor(NULL, IDC_SIZEWE);
        SetCursor(curs);
    }

    return mouseInside;
}

```

5.2.2 Primitives vectorielles

Dans les types de primitives disponibles, on trouve la catégorie 2D. En sélectionnant cette catégorie, on peut ainsi dessiner des carrés, des cercles (ellipses), des triangles, des lignes et des points. Toutes ces primitives sont affectées par la position, la taille, l'épaisseur de traits, la couleur de remplissage et de bordure ainsi que la texture passée par l'utilisateur.

Des objets `ofPath` sont utilisés pour tous les types de primitives 2D créés. Afin d'avoir une meilleure intégration à la scène, une casse générique de primitives 2D a été créée. Pour les différents types, différentes méthodes ont été conçues pour créer les primitives. Ces méthodes utilisent les méthodes `ellipse()`, `circle()`, `rect()`, `triangle()` et `line()`. Elles appliquent chaque propriété spécifiée par l'utilisateur. Après chaque ajout, on ajoute ensuite les primitives dans la scène.

Voici à quoi ressemblent les différentes primitives 2D :

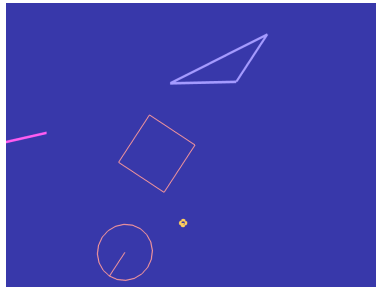


FIGURE 5.5 – Une scène avec primitives 2D.

5.2.3 Formes vectorielles

La forme qui a été implémentée est un cornet de crème glacée, formé d'un cône et d'une sphère. Le code se trouve dans «renderer : :createIceCream()»

```
ofParameter<bool> renderer::createIcecream(int x, int y, int z, int sizeX, int sizeY, int sizeZ,
    ofColor color) {
    ofSpherePrimitive* ball = new ofSpherePrimitive();
    ball->setPosition(x, y + sizeY / 3, z);

    ofConePrimitive* cone = new ofConePrimitive();
    cone->setPosition(x, y - sizeY / 3, z);

    float smallestSphere = min(sizeX, min(sizeY, sizeZ));
    ball->setRadius(smallestSphere / 2);

    float smallestCone = min(sizeX, sizeZ);
    cone->setRadius(smallestCone / 2);
    cone->setHeight(sizeY);

    float newX = (float)sizeX / smallestSphere;
    float newY = (float)sizeY / smallestSphere;
    float newZ = (float)sizeZ / smallestSphere;

    ofMatrix4x4 matrix = ofMatrix4x4();
    matrix.scale(newX, newY, newZ);
    matrix.setTranslation(x, y, z);

    forme3d forme{ ball, color, matrix };
    forme.addPrimitive(cone);
    forme.setName("IceCream " + to_string(scn->nbElements() + 1));
    scn->addElement(forme);
    return forme.selected;
}
```

On aurait bien voulu le faire avec des primitives 2D, mais on s'est dit qu'en 3D ça amenait un peu plus de défi !

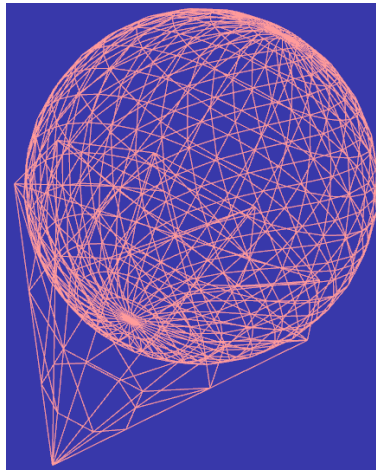


FIGURE 5.6 – Cornet de crème glacée!

5.2.4 Outils de dessin

L'outil de dessin a été implémenté en ajoutant des «sliders» dans l'interface afin de pouvoir modifier la couleur de remplissage, la couleur du contour, la couleur d'arrière-plan et l'épaisseur des lignes de contour. Ainsi, il est possible de modifier en tout temps la couleur d'arrière-plan ou bien de modifier la couleur de remplissage, la couleur de contour et l'épaisseur des lignes de contour pour la création de primitive.

En pratique, à chaque mise à jour de l'application, les valeurs des attributs de la classe `render` sont mises à jour. Ainsi, les valeurs de couleurs ou d'épaisseur des lignes sont à jour lors de la création d'une primitive, tout comme la couleur d'arrière-plan est à jour lorsque l'arrière-plan est redessiné.

Voici la méthode qui met à jour les couleurs et l'épaisseur de lignes du `render` :

```
void ofApp::setRendererParameter() {
    rend->stroke = stroke;
    rend->fill = fill;
    rend->background = background;

    rend->strokeThickness = strokeThickness;
}
```

Voici la méthode qui affiche à l'écran le contenu du `render`. La première ligne consiste à dessiner l'arrière-plan :

```
void render::draw()
{
    ofClear(background);

    [...]
}
```

Voici une des méthodes qui ajoute une primitive à la scène. Les paramètres «fill» et «stroke» utilisés ici sont les attributs de couleurs de remplissage et de contour de la classe `render` :

```

ofParameter<bool> renderer::createSquare(float x, float y, float w, float h) {
    return createSquare(x,y,w,h,fill, stroke);
}

```

5.2.5 Interface

L'application est dotée d'une interface utilisateur qui permet de modifier les paramètres d'une primitive à ajouter, ajouter une primitive à la scène, vider la scène, modifier les paramètres de la caméra, appliquer des transformations sur les primitives ou sur la scène, appliquer des effets de traitement d'image, sélectionner des primitives à l'aide d'une liste à cocher, importer un modèle 3D, exporter en image et quitter.

L'interface a été développée avec l'extension ofxGui. Ainsi, nous avons créé des «panels» (ofxPanel) dans lesquels les différents paramètres de l'application ont été ajoutés et l'extension se charge de générer les «sliders» et les cases à cocher associées aux paramètres. Nous avons aussi pu ajouter des boutons (ofxButton).

Comme l'essentiel du code de la classe qui gère l'interface représente plus de 1000 lignes, seulement quelques exemples seront montrés afin d'alléger ce document.

Voici l'initialisation de paramètres :

```

void ofApp::initOfParameters()
{
    [...]

    primPosX.setName("X");
    primPosX.setMin(MinX);
    primPosX.setMax(MaxX);
    primPosX.set((MinX + MaxX) / 2);

    primPosY.setName("Y");
    primPosY.setMin(MinY);
    primPosY.setMax(MaxY);
    primPosY.set((MinY + MaxY) / 2);

    primPosZ.setName("Z");
    primPosZ.setMin(MinZ);
    primPosZ.setMax(MaxZ);
    primPosZ.set((MinZ + MaxZ) / 2);

    [...]
}

```

Voici l'initialisation de groupes de paramètres (ofParameterGroup) :

```

void ofApp::initGroups()
{
    [...]

    groupPrimitivePosition2D.setName("Position");
    groupPrimitivePosition2D.add(primPosX.set(primPosX));
    groupPrimitivePosition2D.add(primPosY.set(primPosY));

```

```

    groupPrimitivePosition3D.setName("Position");
    groupPrimitivePosition3D.add(primPosX.set(primPosX));
    groupPrimitivePosition3D.add(primPosY.set(primPosY));
    groupPrimitivePosition3D.add(primPosZ.set(primPosZ));

    [...]
}

```

Voici la configuration de menu :

```

void ofApp::setupMenu2D() {

    menu2D.setDefaultWidth(200);

    menu2D.setup();

    menu2D.add(groupPrimitiveType2D);
    menu2D.add(groupPrimitivePosition2D);
    menu2D.add(groupPrimitiveSize2D);

    menu2D.add(groupThick);

    menu2D.add(groupFill);

    menu2D.add(groupStroke);

    menu2D.add(groupTexture);

    menu2D.minimizeAll();

    menu2D.registerMouseEvents();
}

```

Voici la configuration des écouteurs sur les différents boutons de l'application :

```

void ofApp::initButtonListener() {
    btnDrawPrimitive.addListener(this, &ofApp::btnDrawPrimitiveClicked);
    btnClear.addListener(this, &ofApp::btnClearClicked);
    btnExit.addListener(this, &ofApp::btnExitClicked);

    btnExport.addListener(this, &ofApp::btnExportClicked);
    btnImport.addListener(this, &ofApp::btnImportClicked);

    btnApplySelect.addListener(this, &ofApp::btnApplySelectClicked);
}

```

Voici la méthode setup de la classe ofApp :

```

void ofApp::setup()
{
    [...]

    initOfParameters();
    initGroups();

    setupMenu2D();
}

```

```

    setupMenu3D();
    setupCameraMenu();
    setupTransformationMenu();
    setupOptionsMenu();
    setupSelectionMenu();

    initButtonListener();

    [...]
}

```

Voici une capture d'écran pour montrer l'interface :

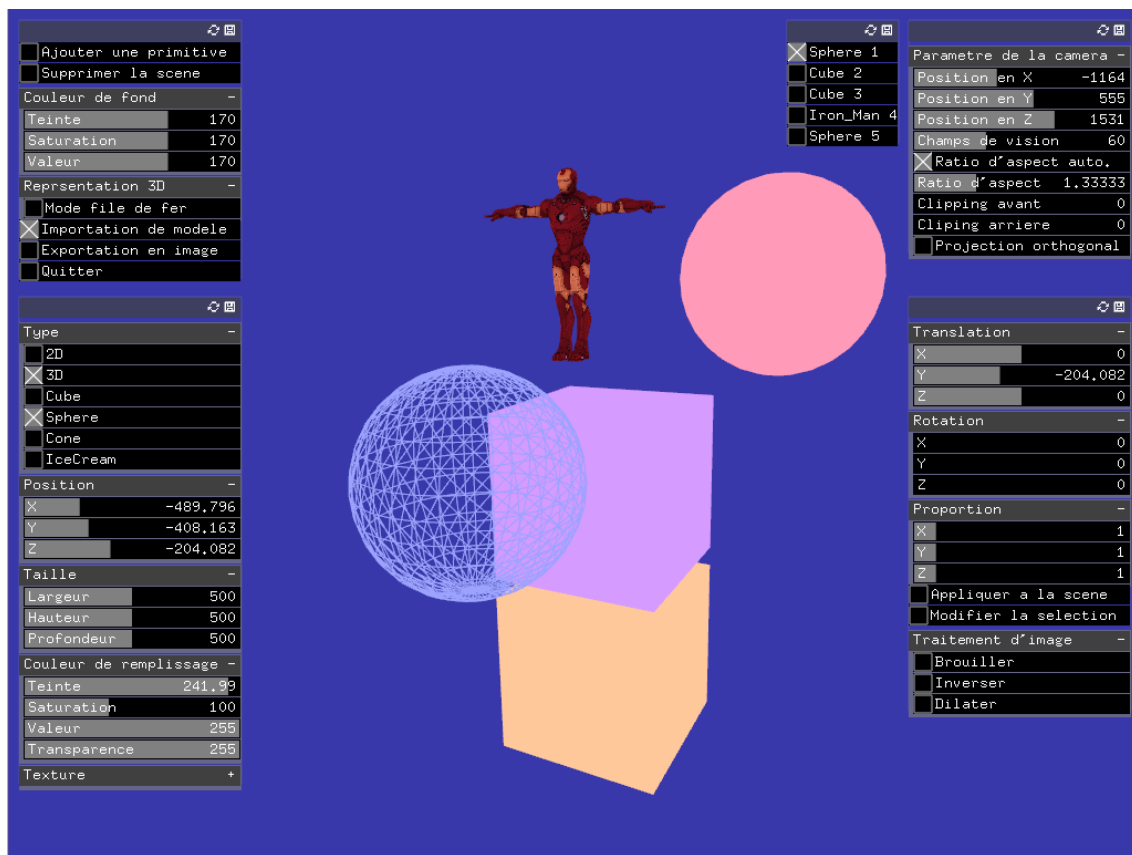


FIGURE 5.7 – Exemple d'interface

5.3 Transformation

5.3.1 Transformation interactive

Une catégorie à droite de l'application permet de transformer le système de coordonnées des entités géométriques de la scène. On peut exercer des transformations telles que la translation en X, Y et Z, la rotation en X, Y et Z ainsi que la proportion en X, Y et Z. Elle peut se faire autant en temps réel qu'en différé.

Les matrices de transformation d'openframeworks sont utilisées pour la réalisation des transformations. On utilise les méthodes `ofPushMatrix()` et `ofPopMatrix()`. Une fois la matrice empilée, on ajoute les transformations à l'aide des méthodes `ofTranslate()`, `ofRotate()` et `ofScale()` selon les paramètres inscrits par l'utilisateur. Enfin, on dépile la matrice pour permettre la transformation.

Voici le menu des transformations et ces différentes options :

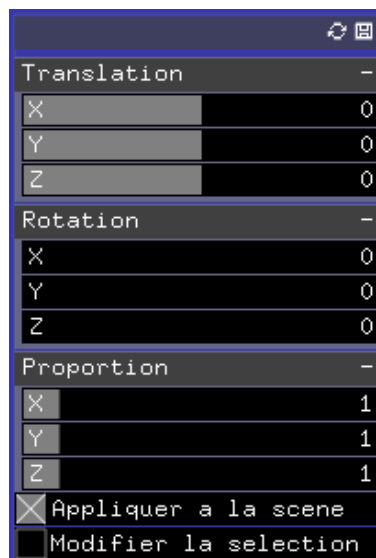


FIGURE 5.8 – Menu des transformations.

5.3.2 Structure de scène

La structure de scène a été implémentée sous la forme d'un arbre ordonné, dans lequel les feuilles sont les éléments de la scène. La classe scène comprend 4 classes interne, soit «element», «group», «node» et «scene_iterator». Les classes «group» et «node» héritent d'«element» et servent à stocker tout ce qui se trouve dans la scène. La classe «scene_iterator» permet, quant à elle, de parcourir les éléments pour les dessiner.

On utilise les «shared_ptr» au lieu des pointeurs normaux pour conserver les éléments qui sont ajoutés à la scène pour ne pas avoir à trop gérer la mémoire.

L'essentiel du code de la scène se trouve dans les méthodes `addElement` de la scène, des sous-classes de stockage et l'opérateur++ de l'itérateur.

```

void scene::addElement(size_t index, primitive_ptr& p, bool insertFirstChild) {
    if (index == 0 && !insertFirstChild) {
        throw invalid_argument("root don't have parent...");
    }
    root->addElement(index, p, insertFirstChild);
}

//Retourne la quantite d'element ajoute
size_t scene::node::addElement(size_t index, primitive_ptr& p, bool insertFirstChild) {
    if (index != this->index) {
        throw invalid_argument("index need to be equals to the index of the node");
    }
    if (insertFirstChild) {
        throw invalid_argument("node need to be wrapped in a group");
    }
    this->content = p;
    contentType = "primitive";
    return 1;
}

//Retourne la quantite d'element ajoute
size_t scene::group::addElement(size_t index, primitive_ptr& p, bool insertFirstChild) {
    size_t addedSize = 0;

    if (this->index == index) {
        if (insertFirstChild) {
            //Insérer comme premier element
            childrens.insert(childrens.begin(), element_ptr{ new node{ index + 1, height + 1, p } });
            for (auto& it = childrens.begin() + 1; it < childrens.end(); ++it) {
                it->get()->setIndex(it->getIndex() + 1);
            }
            addedSize++;
        } else {
            throw invalid_argument("element must to be add in the parent");
        }
    } else {
        size_t ubound = childrens.size();
        size_t lbound = 0;
        size_t i;

        //Recherche binaire
        while (lbound <= ubound) {
            i = lbound + (ubound - lbound) / 2;
            if (childrens[i]->getIndex() == index) {
                if (insertFirstChild) {
                    if (childrens[i]->getType() != "group") {
                        group_ptr temp = group_ptr{ new group{ index, height + 1 } };
                        temp->childrens.push_back(childrens[i]);
                        temp->childrens[0]->setIndex(index + 1);
                        temp->childrens[0]->setHeight(height + 2);
                        temp->size = temp->childrens[0]->getSize() + 1;
                        childrens[i] = temp;
                        addedSize++;
                    }
                }
            }
        }
    }
}

```

```

        addedSize += childrens[i]->addElement(index, p, insertFirstChild);
    } else {
        childrens.insert(childrens.begin() + i + 1, element_ptr{ new node{ index +
            childrens[i]->getSize(), height + 1, p } });
        i++;
        addedSize++;
    }
    i++;
    break;
} else if (childrens[i]->getIndex() < index) {
    lbound = i + 1;
    if (ubound < lbound) {
        //Ajoute l'element dans le groupe sous-jacent
        addedSize += childrens[i]->addElement(index, p, insertFirstChild);
        i++;
        break;
    }
} else {
    ubound = i - 1;
    if (ubound < lbound) {
        //Ajoute l'element dans le groupe sous-jacent
        addedSize += childrens[i - 1]->addElement(index, p, insertFirstChild);
        break;
    }
}
}
}
for (auto& it = childrens.begin() + i; it < childrens.end(); ++it) {
    it->get()->setIndex(it->get()->getIndex() + addedSize);
}
}
size += addedSize;
return addedSize;
}

//Avance jusqu'au prochain node
void scene::scene_iterator::operator++() {
    for (rootIndex; rootIndex <= root->getSize(); ++rootIndex) {
        element* elem = root->getElement(rootIndex);
        if (elem->getType() != "group" && elem->getType() != "root") {
            primitive_ptr ptr = (dynamic_cast<node*>(elem))->content;
            if (p != ptr) {
                p = ptr;
                break;
            }
        }
    }
    if (rootIndex > root->getSize()) {
        p = primitive_ptr{ nullptr };
    }
}

```

Comme vous avez sans doute remarqué, l'ajout d'élément à la scène se fait récursivement, à l'index en paramètre. Le paramètre «insertFirstChild» indique s'il faut insérer l'élément comme le premier enfant de l'élément à l'index en paramètre. S'il est faux, on insère simplement le nouvel élément après l'index. Dans `group : addElement`, on utilise un algorithme de recherche binaire pour trouver dans ou après quel élément

il faut ajouter le nouvel élément. L'operator++, quant à lui, parcourt la scène en s'arrêtant seulement sur les classes node.

Malheureusement, par manque de temps la structure de scène n'est pas utilisée à son plein potentiel et tous les éléments sont stockés dans le groupe à la racine de la scène. Il est tout de même possible de voir le résultat en changeant la ligne «#define test 0» pour «#define test 1» dans le fichier «main.cpp». Vous verrez alors le résultat de l'exécution des tests de la classe «scene» (principalement de l'ajout et de la suppression d'élément), se trouvant à la fin de «scene.cpp».

5.3.3 Sélection multiple

Dans notre application, toutes les entités géométriques, soit les primitives en 2D, les primitives en 3D et les modèles 3D importés du disque, apparaissent dans un menu avec un nom unique.

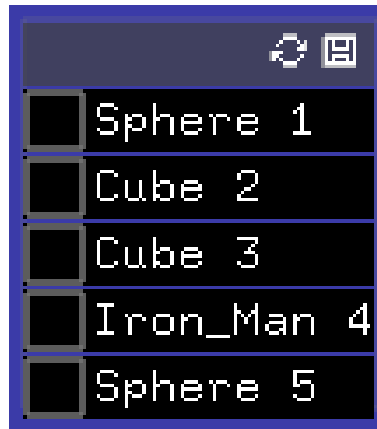


FIGURE 5.9 – Menu de sélection

À partir de là, il est possible d'en sélectionner un ou plusieurs.

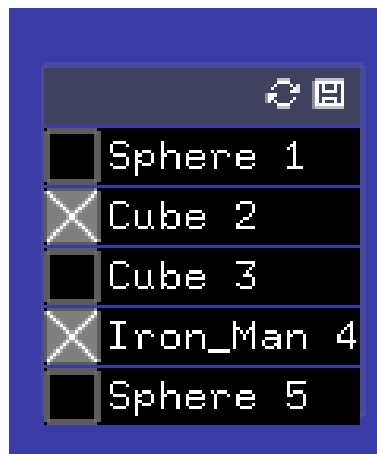


FIGURE 5.10 – Plusieurs entités sélectionnées

Lorsqu'on n'est pas en mode «wireframe», les entités sélectionnées seront affichées en «wireframe» quand même, de façon à les identifier. Comme le mode «wireframe» existe surtout à des fins de débogage et pour des opérations précises, on n'est pas censé l'utiliser en permanence, c'est pourquoi ce n'est pas grave si dans ce mode on ne peut pas voir aussi bien quelles entités sont sélectionnées.

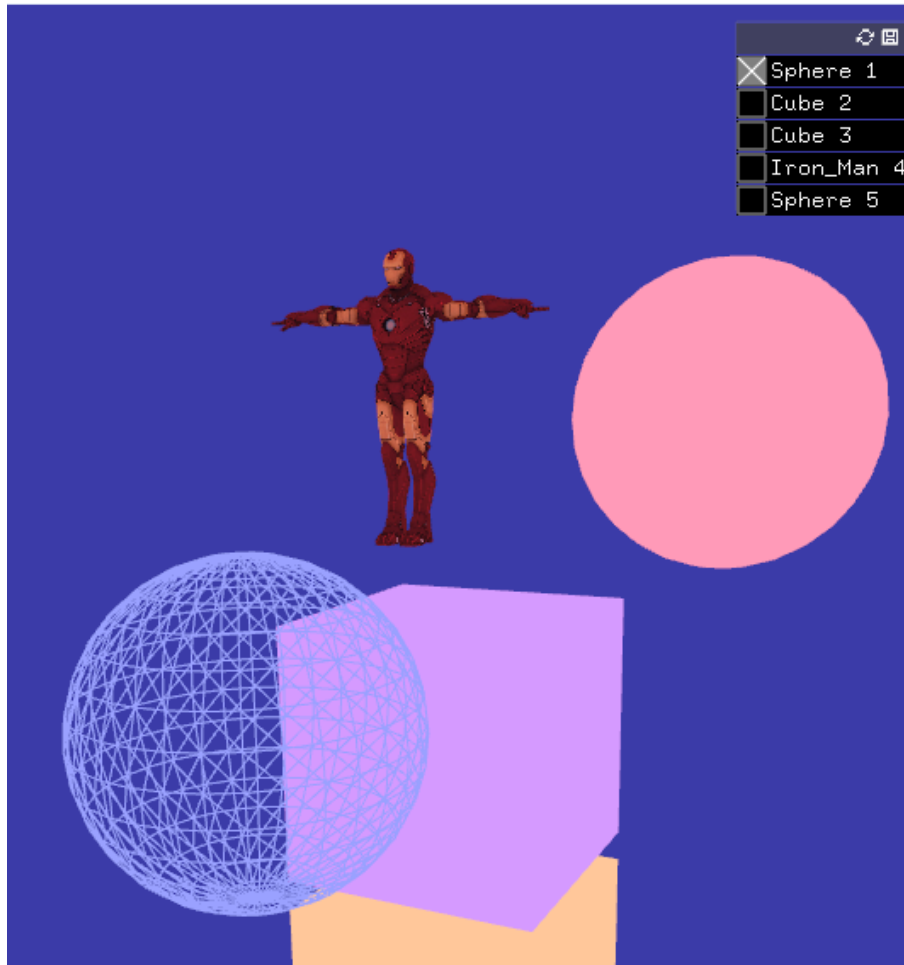


FIGURE 5.11 – La sélection est en «wireframe»

Les transformations géométriques peuvent être appliquées en même temps à toutes les entités sélectionnées, en appliquant une matrice de transformation à chacun d'entre eux en même temps. Cette matrice est créée à partir de "sliders", de translation, de rotation et de taille.

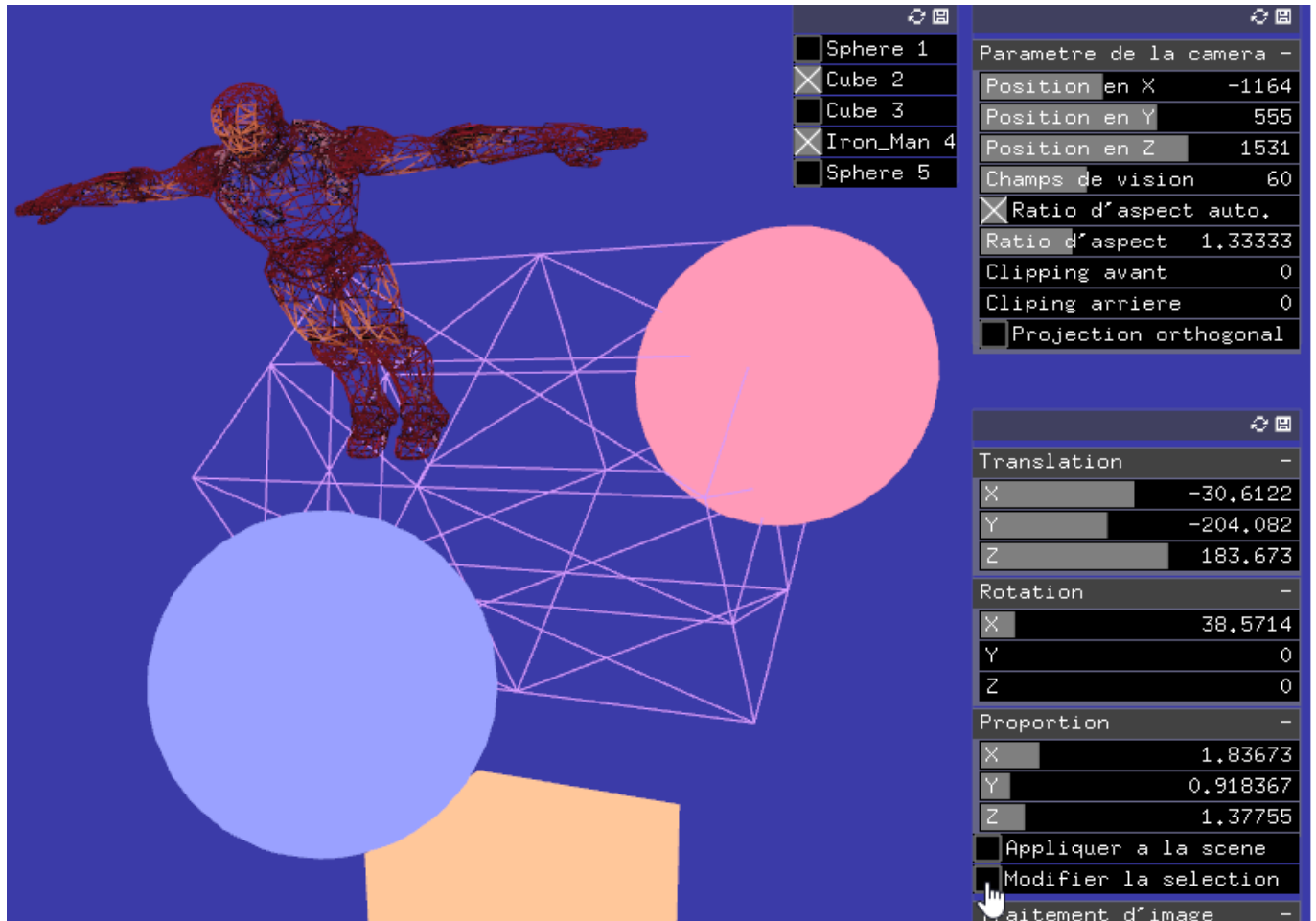


FIGURE 5.12 – Chaque entité de la sélection reçoit la transformation

```
void renderer::applySelection(ofMatrix4x4 matrix)
{
    for (auto& p : *scn)
    {
        if (p.selected.get())
        {
            ofMatrix4x4 oldMat = p.getTransfo();
            p.setTransfo(oldMat * matrix);
        }
    }
    std::list<extModel>::iterator iterator4;
    for (iterator4 = externalModels.begin(); iterator4 != externalModels.end(); ++iterator4)
    {

```

```
        if (iterator4->selected.get())
        {
            ofMatrix4x4 oldMat = iterator4->getTransfo();
            iterator4->setTransfo(oldMat * matrix);
        }
    }
}
```

```
void primitive3d::draw(bool wireframe) {

    ofPushMatrix();
    ofTranslate(transfoMatrix.getTranslation());

    ofSetColor(fillCol);

    ofQuaternion rotation = transfoMatrix.getRotate();
    float rotationAmount;
    ofVec3f rotationAngle;
    rotation.getRotate(rotationAmount, rotationAngle);

    ofRotate(rotationAmount, rotationAngle.x, rotationAngle.y, rotationAngle.z);

    ofScale(transfoMatrix.getScale());

    if (wireframe || selected.get())
        prim->drawWireframe();
    else
        prim->drawFaces();

    ofPopMatrix();
}
```

5.3.4 Coordonnées non cartésiennes

Non-implémenté

5.3.5 Historique

Non-implémenté

5.4 Géométrie

5.4.1 Particules

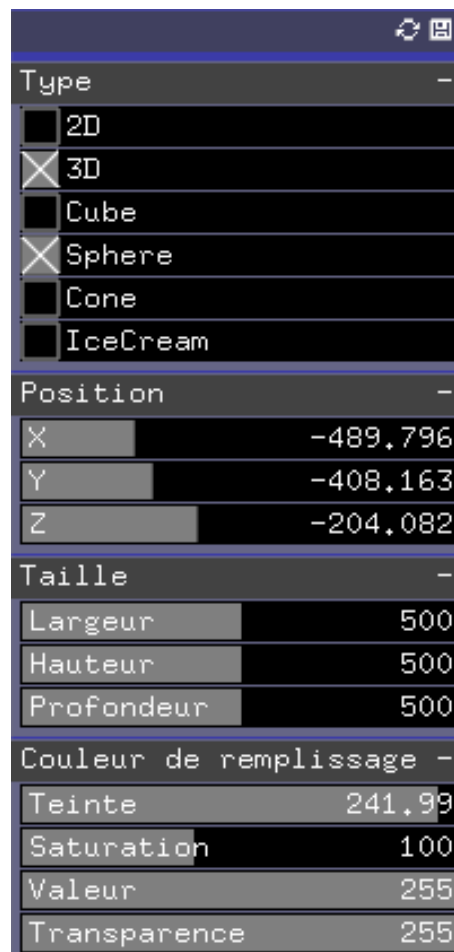
Non-implémenté

5.4.2 Primitives

Dans notre application, il est possible de créer à partir d'algorithmes seulement plusieurs primitives en 3D. Ces primitives sont le cube, la sphère et le cône. Chaque primitive peut être créée directement avec une position et taille choisie par l'utilisateur, mais ces attributs pourront bien évidemment être modifiés par la suite (voir la section 4.3.3 sur la sélection multiple.).

On ne peut pas créer les primitives avec une rotation dès le départ, car ce n'est pas pertinent de donner une rotation à un objet comme un cône, quand il n'y a aucun moyen de savoir quelle est son orientation "sans rotation" avant d'en avoir créé un de toute façon. Il faut leur donner la rotation voulue après les avoir créées.

On peut aussi choisir une couleur par primitive, dans l'espace de couleur HSB.



Type	
<input type="checkbox"/>	2D
<input checked="" type="checkbox"/>	3D
<input type="checkbox"/>	Cube
<input checked="" type="checkbox"/>	Sphere
<input type="checkbox"/>	Cone
<input type="checkbox"/>	IceCream

Position	
X	-489.796
Y	-408.163
Z	-204.082

Taille	
Largeur	500
Hauteur	500
Profondeur	500

Couleur de remplissage	
Teinte	241.99
Saturation	100
Valeur	255
Transparence	255

FIGURE 5.13 – Options de création d'une primitive

```

void ofApp::btnDrawPrimitiveClicked()
{
ofLog() << "<app::btnDrawPrimitiveClicked>";

if (primType2D.get()) {
if (primTypeCube.get()) {
selectionMenu.add(rend->createSquare(primPosX, primPosY, primSizeWidth, primSizeHeight));
}
else if (primTypeSphere.get()) {
selectionMenu.add(rend->createCircle(primPosX, primPosY, primSizeWidth, primSizeHeight));
}
else if (primTypeTriangle.get()) {
selectionMenu.add(rend->createTriangle(primPosX, primPosY, primPosX + primSizeWidth, primPosY,
(primPosX + primSizeWidth) / 2, primPosY + primSizeHeight));
}
else if (primTypeLine.get()) {
selectionMenu.add(rend->createLine(primPosX, primPosY, primSizeWidth, primSizeHeight));
}
else if (primTypePoint.get()) {
selectionMenu.add(rend->createPoint(primPosX, primPosY, strokeThickness));
}
}
else {
if (primTypeCube.get()) {
selectionMenu.add(rend->createCube(primPosX, primPosY, primPosZ, primSizeWidth, primSizeHeight,
primSizeDepth));
}
else if (primTypeSphere.get()) {
selectionMenu.add(rend->createSphere(primPosX, primPosY, primPosZ, primSizeWidth, primSizeHeight,
primSizeDepth));
}
else if (primTypeTriangle.get()) {
selectionMenu.add(rend->createCone(primPosX, primPosY, primPosZ, primSizeWidth, primSizeHeight,
primSizeDepth));
}
else {
selectionMenu.add(rend->createIcecream(primPosX, primPosY, primPosZ, primSizeWidth,
primSizeHeight, primSizeDepth));
}
}
}
}

```

```
//-----3D primitives-----
ofParameter<bool> renderer::createCube(int x, int y, int z, int w, int h, int d)
{
    return createCube(x, y, z, w, h, d, fill);
}

ofParameter<bool> renderer::createCube(int x, int y, int z, int w, int h, int d, ofColor fillCol)
{
    ofBoxPrimitive* box = new ofBoxPrimitive();

    float smallest = min(w, min(h, d));

    box->setWidth(smallest);
    box->setHeight(smallest);
    box->setDepth(smallest);

    float newX = (float)w / smallest;
    float newY = (float)h / smallest;
    float newZ = (float)d / smallest;

    ofMatrix4x4 matrix = ofMatrix4x4();
    matrix.scale(newX, newY, newZ);
    matrix.setTranslation(x, y, z);

    for (int i = 0; i < 6; i++)
    {
        box->setSideColor(i, fillCol);
    }

    primitive3d prim = primitive3d{ box, fillCol, matrix };
    prim.setName("Cube " + to_string(scn->nbElements() + 1));
    scn->addElement(prim);
    return prim.selected;
}

ofParameter<bool> renderer::createSphere(int x, int y, int z, int sizeX, int sizeY, int sizeZ)
{
    return createSphere(x, y, z, sizeX, sizeY, sizeZ, fill);
}

ofParameter<bool> renderer::createSphere(int x, int y, int z, int sizeX, int sizeY, int sizeZ,
    ofColor color)
{
    ofSpherePrimitive* ball = new ofSpherePrimitive();
    ball->setPosition(0, 0, 0);

    float smallest = min(sizeX, min(sizeY, sizeZ));

    ball->setRadius(smallest/2);

    float newX = (float)sizeX / smallest;
    float newY = (float)sizeY / smallest;
    float newZ = (float)sizeZ / smallest;
}
```



```

ofMatrix4x4 matrix = ofMatrix4x4();
matrix.scale(newX, newY, newZ);
matrix.setTranslation(x, y, z);

primitive3d prim = primitive3d{ ball, color, matrix };
prim.setName("Sphere " + to_string(scen->nbElements() + 1));
scn->addElement(prim);
return prim.selected;
}

ofParameter<bool> renderer::createCone(int x, int y, int z, int sizeX, int sizeY, int sizeZ)
{
return createCone(x, y, z, sizeX, sizeY, sizeZ, fill);
}

ofParameter<bool> renderer::createCone(int x, int y, int z, int sizeX, int sizeY, int sizeZ,
ofColor color)
{
ofConePrimitive* cone = new ofConePrimitive();
cone->setPosition(0, 0, 0);

float smallest = min(sizeX, sizeZ);
cone->setRadius(smallest / 2);
cone->setHeight(sizeY);

float newX = (float)sizeX / smallest;
float newY = 1.0f;
float newZ = (float)sizeZ / smallest;

ofMatrix4x4 matrix = ofMatrix4x4();
matrix.scale(newX, newY, newZ);
matrix.setTranslation(x, y, z);

primitive3d prim = primitive3d{ cone, color, matrix };
prim.setName("Cone " + to_string(scen->nbElements() + 1));
scn->addElement(prim);
return prim.selected;
}

```

5.4.3 Modèle

Il est possible pour un utilisateur d'importer un modèle choisi sur son ordinateur. Les modèles supportés sont ceux qui ont un des formats suivants :

3DS	ASE	DXF	HMP	MD2	MD3
MD5	MDC	MDL	NFF	PLY	STL
X	LWO	OBJ	SMD	Collada	LWO
Ogre XML	partly LWS				

Si le modèle est dans un répertoire avec ses textures dans le bon chemin relatif, celles-ci seront automatiquement chargées et appliquées. De plus, si l'importation du modèle échoue pour une quelconque raison, un message sera affiché à l'utilisateur pour l'informer.

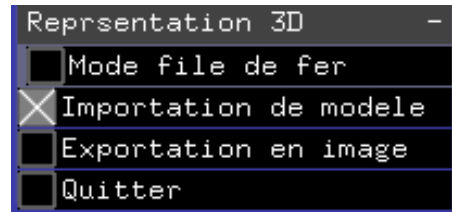


FIGURE 5.14 – Bouton pour importer un modèle

Un modèle peut être sélectionné comme une primitive, sera affecté par le mode "Wireframe", et sera modifié si on applique une matrice de transformation pendant qu'il est sélectionné.

Une partie du code qui suit vient de MSDN (Microsoft), dont le lien peut-être trouvé dans la section Références. Nous avons modifié le code pour qu'il convienne à nos besoins.

```
void ofApp::btnImportClicked()
{
    HRESULT hr = CoInitializeEx(NULL, COINIT_APARTMENTTHREADED |
    COINIT_DISABLE_OLE1DDE);
    if (SUCCEEDED(hr))
    {
        IFileOpenDialog *pFileOpen;

        // Create the FileOpenDialog object.
        hr = CoCreateInstance(CLSID_FileOpenDialog, NULL, CLSCTX_ALL,
        IID_IFileOpenDialog, reinterpret_cast<void*>(&pFileOpen));

        if (SUCCEEDED(hr))
        {
            // Show the Open dialog box.
            hr = pFileOpen->Show(NULL);

            // Get the file name from the dialog box.
            if (SUCCEEDED(hr))
            {
                IShellItem *pItem;
                hr = pFileOpen->GetResult(&pItem);
                if (SUCCEEDED(hr))
                {
                    LPWSTR pszFilePath;
                    hr = pItem->GetDisplayName(SIGDN_FILESYSPATH, &pszFilePath);

                    // Display the file name to the user.
                    if (SUCCEEDED(hr))
                    {
                        std::wstring path = wstring(pszFilePath);
                        std::string strPath(path.begin(), path.end());
                        ofParameter<bool> param = rend->importModel(strPath);

                        if (param.get() == false)
                        {
                            selectionMenu.add(param);
                            strPath = "Le modele " + strPath + " a ete importe avec succes!";
                        }
                    }
                }
            }
        }
    }
}
```

```

path = std::wstring(strPath.begin(), strPath.end());
LPCWSTR title = (LPCWSTR)path.c_str();
MessageBox(NULL, title, L"Success", MB_OK);
}
else
{
    strPath = "Le modele n'a pas pu etre importe.";
    path = std::wstring(strPath.begin(), strPath.end());
    LPCWSTR title = (LPCWSTR)path.c_str();
    MessageBox(NULL, title, L"Erreur", MB_OK);
}

CoTaskMemFree(pszFilePath);
}
pItem->Release();
}
}
pFileOpen->Release();
}
CoUninitialize();
}
}

```

```

ofParameter<bool> renderer::importModel(string path) {
    ofxAssimpModelLoader* model = new ofxAssimpModelLoader();
    bool ret = model->loadModel(path, false);
    if (ret)
    {
        model->enableTextures();
        ofTexture tex = ofTexture();
        extModel mod = extModel(model);

        string fName(path);
        size_t pos = fName.rfind(".");
        if (pos != string::npos)
        {
            if (pos != 0)
            {
                fName = fName.substr(0, pos);
            }
        }
        pos = fName.rfind("\\");
        if (pos != string::npos)
        {
            if (pos != 0)
            {
                fName = fName.substr(pos + 1);
            }
        }

        mod.setName(fName + " " + to_string(externalModels.size() + 1));
        scn->addElement(mod);
        return mod.selected;
    }
}

```

```
return ofParameter<bool>(true);  
}
```

5.4.4 Texture

Non-implémenté

5.4.5 Géométrie procédurale

Non-implémenté

5.5 Caméra

5.5.1 Propriétés de caméra

Les propriétés de la caméra telle que le champ de vision, le ratio d'aspect ainsi que la distance du plan de clipping avant et arrière. Ils peuvent être modifiés dans l'interface graphique à l'aide de «slider». L'essentiel du code se trouve dans «ccamera : :update()».

```
cam->setFov(fov.get());
if (autoRatio.get()) {
cam->setForceAspectRatio(false);
} else {
cam->setAspectRatio(ratio.get());
}
cam->setNearClip(nearClip.get());
cam->setFarClip(farClip.get());
```

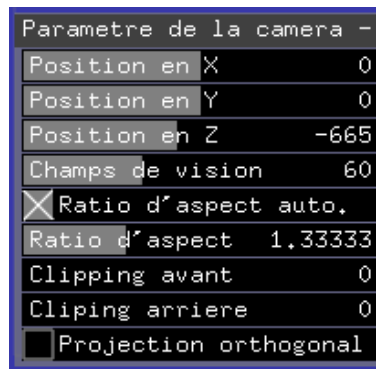


FIGURE 5.15 – Propriété de la caméra dans l'interface

5.5.2 Mode de projection

Le changement de mode de projection de perspective à orthogonale a été implémenté dans l'application. L'essentiel du travail se fait dans la méthode «ccamera : :changeMode()». Elle est appelée lorsqu'on appuie sur le bouton à cet effet dans l'interface graphique.

```
if (ortho.get()) {
cam->enableOrtho();
} else {
cam->disableOrtho();
}
```

5.5.3 Caméra interactive

La caméra interactive est implémentée dans l'application principalement à l'aide de la classe ofEasyCam de openFrameworks. Nous avons tout de même ajouté la possibilité de déplacer à l'aide des flèches du clavier, pour permettre de repositionner facilement la caméra. Il est aussi possible d'avancer à caméra à l'aide de pageUp/Down. L'essentiel du code se trouve au début de «ccamera : :update()».

```
float dist = speed * dt;
float dx = 0;
float dy = 0;
float dz = 0;

dx = 0;
if (isCameraMoveLeft)
dx += dist;
if (isCameraMoveRight)
dx -= dist;
cam->truck(-dx);
posX.set(round(-cam->getX()));

dy = 0;
if (isCameraMoveUp)
dy -= dist;
if (isCameraMoveDown)
dy += dist;
cam->boom(-dy);
posY.set(round(cam->getY()));

dz = 0;
if (isCameraMoveForward)
dz -= dist;
if (isCameraMoveBackward)
dz += dist;
cam->dolly(dz);
posZ.set(round(cam->getZ()));
```

5.5.4 Caméra multiple

Non-implémenté

5.5.5 Caméra animée

Non-implémenté

5.6 Pipeline de rendu

5.6.1 Portabilité

Non-implémenté

5.6.2 Shader de géométrie

Non-implémenté

5.6.3 Rétroaction de transformation

Non-implémenté

5.6.4 Passes de rendu

Non-implémenté

5.6.5 Techniques d’occlusion

Non-implémenté

5.7 Illumination

5.7.1 Types de lumière

Non-implémenté

5.7.2 Lumières multiples

Non-implémenté

5.7.3 Matériaux

Non-implémenté

5.7.4 Modèle d'illumination

Non-implémenté

5.7.5 Volume de lumière

Non-implémenté

5.8 Lancer de rayon

5.8.1 Intersection

Dans l'application, il y a des fonctionnalités de lancer de rayon. Celles-ci sont utilisées pour faciliter la sélection d'entités dans la scène. Cette fonctionnalité n'est implémenté que pour 2 types de primitives géométriques, soit les cubes et les sphères.

Le fonctionnement est très simple. Comme dans la section 5.3.3 (Sélection multiple), on pouvait ajouter ou retirer des objets de la sélection. Avec le lancer de rayon, tout clic dans la fenêtre lancera un rayon, et si celui-ci intersecte avec une primitive de type cube ou sphère (les seuls supportés), ce agira comme si on avait cliqué sur sa case à coché. Cela l'ajoutera ou le retirera de la sélection courante, selon si cet objet était déjà sélectionné ou pas.

Le rayon n'intersecte pas les objets qui sont derrière la caméras (qui sont pourtant touchés par le rayon qui est infini dans les deux sens) car on ne conserve que les intersections à distance positive vers l'avant. De plus, si plusieurs primitives valides sont en intersection avec le rayon, la selection n'affectera que celle qui est la plus proche de la caméra, puisque logiquement c'est sur celle-ci que l'utilisateur a cliqué, car la partie sous le clic de la seconde primitive est forcément cachée par la première du point de vue de la caméra.

```

struct hit {
    int faceIndex;
    float distance;
};

struct by_distance {
    bool operator()(hit const &a, hit const &b) {
        return a.distance < b.distance;
    }
};

```

```

bool primitive::intersectsMesh(ofRay ray, const ofMesh &mesh, const ofMatrix4x4 &toWorldSpace,
    vector<hit> *meshHit) {
    const vector<ofMeshFace>& faces = mesh.getUniqueFaces();
    bool intersection = false;
    bool intersectedOnce = false;
    vector<hit> distances = vector<hit>();
    for (int i = 0; i < faces.size(); i++) {
        const ofMeshFace &face = faces[i];
        // intersections are done worldSpace
        ofVec3f one = face.getVertex(0) * toWorldSpace;
        ofVec3f two = face.getVertex(1) * toWorldSpace;
        ofVec3f three = face.getVertex(2) * toWorldSpace;
        one = one * transfoMatrix;
        two = two * transfoMatrix;
        three = three * transfoMatrix;

        float t;
        intersection = calcTriangleIntersection(one, two, three, ray, &t);

        if (intersection && t > 0) {
            hit newHit = hit();
            newHit.distance = t;
            newHit.faceIndex = i;
            distances.push_back(newHit);
        }
    }
}

```

```

        intersectedOnce = true;
        //break;
    }
}

if (intersectedOnce)
{
    std::sort(distances.begin(), distances.end(), by_distance());
    for (int i = 0; i < distances.size(); i++)
    {
        meshHit->push_back(distances[i]);
    }
}

return intersectedOnce;
}

bool primitive::calcTriangleIntersection(const ofVec3f &vert0, const ofVec3f &vert1, const ofVec3f
    &vert2, ofRay ray, float *result) const
{
    ofVec3f edge1, edge2, tvec, pvec, qvec;
    float det;
    float u, v;
    const float EPSILON = 0.000001f;

    edge1 = vert1 - vert0;
    edge2 = vert2 - vert0;

    pvec = ray.getTransmissionVector().getNormalized().getCrossed(edge2);
    det = edge1.dot(pvec);

    #if 0 // we don't want to backface cull
        if (det < EPSILON)
            return false;
        tvec = getOrigin() - vert0;

        u = tvec.dot(pvec);
        if ((u < 0.0f) || (u > det))
            return false;

        qvec = tvec.getCrossed(edge1);
        v = getDirection().dot(qvec);
        if (v < 0.0f || u + v > det)
            return false;

        *result = edge2.dot(qvec) / det;
        return true;
    #else
        if (det > -EPSILON && det < EPSILON)
            return false;

        float inv_det = 1.0f / det;
        tvec = ray.getStart() - vert0;
        u = tvec.dot(pvec) * inv_det;

```

```

    if (u < 0.0f || u > 1.0f)
        return false;

    qvec = tvec.getCrossed(edge1);

    v = ray.getTransmissionVector().getNormalized().dot(qvec) * inv_det;
    if (v < 0.0f || u + v > 1.0f)
        return 0;

    *result = edge2.dot(qvec) * inv_det;
    return true;
#endif
}

vector<hit> primitive3d::intersectsMeshInstance(const ofVec2f &screenCoordinates, const ofCamera
    &cam) {

    ofMatrix4x4 toWorldSpace = prim->getGlobalTransformMatrix();
    ofMesh mesh = prim->getMesh();

    vector<hit>* hits = new vector<hit>();

    ofVec3f screenToWorld = cam.screenToWorld(ofVec3f(screenCoordinates.x, screenCoordinates.y,
        0.0));
    ofRay ray(cam.getPosition(), screenToWorld - cam.getPosition());

    intersectsMesh(ray, mesh, toWorldSpace, hits);

    return *hits;
}

void renderer::selectPrimitive(int x, int y, bool shiftHeld)
{
    primitive* toSelect;
    float distance = -1;
    for (primitive& p : *scn)
    {
        vector<hit> hits = p.intersectsMeshInstance(ofVec2f(x, y), (**cam));
        if (hits.size() > 0 && (distance == -1 || hits[0].distance < distance))
        {
            distance = hits[0].distance;
            toSelect = &p;
        }
    }

    if (distance > -0.9)
    {
        toSelect->changeSelected();
    }
}

void ofApp::mouseReleased(int x, int y, int button) {

```

```
rend->selectPrimitive(x, y, GetKeyState(VK_SHIFT));  
}
```

5.8.2 Réflexion

Dans notre application, il est possible de créer des cubes de type "miroir".

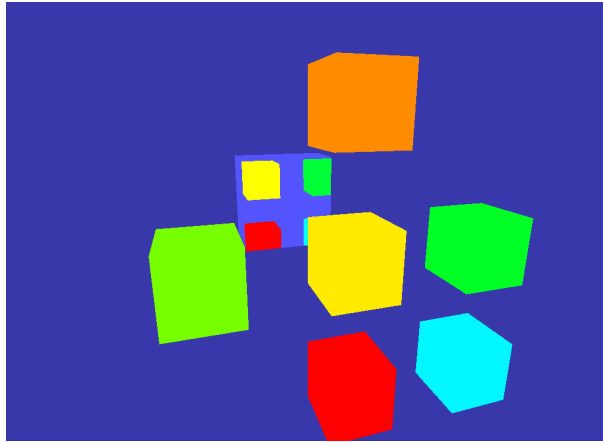


FIGURE 5.16 – Cube de type miroir

Chacune des faces d'un tel cube reflète son entourage. À cause des limitations de temps (et de vitesse de calcul), seuls les sphères et les autres cubes s'affichent dans les miroirs, car, de par le point précédent, ce sont les seuls objets qui incorporent le lancer de rayon.

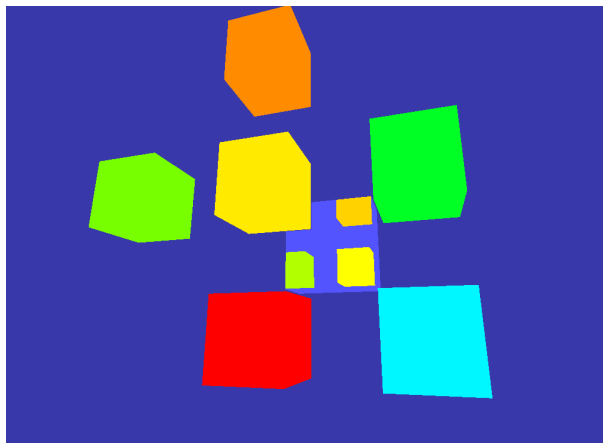


FIGURE 5.17 – Les miroirs prennent la caméra en compte

Tout ce qui est reflété dans un miroir, même le fond de la scène, est rendu un peu plus pâle (à l'exception des pixels déjà blancs, évidemment). Cela est fait pour que les délimitations de la surface miroir soient facile à voir, car sinon elles seraient identiques au fond derrière, et donc partiellement invisible.

Il est techniquement aussi possible de créer des sphères miroir, mais nous n'avons pas mis d'option pour le faire dans l'interface.

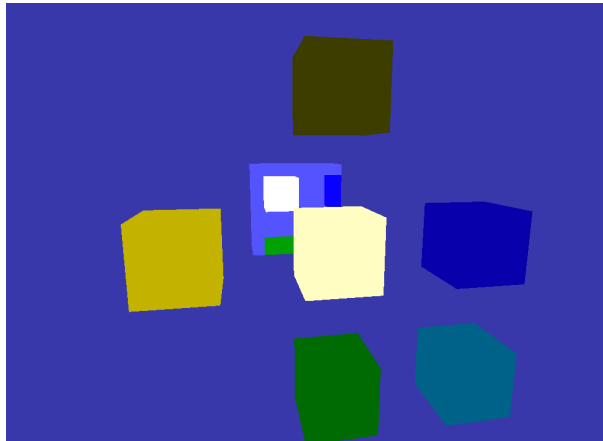


FIGURE 5.18 – On voit que le fond et chaque cube sont plus pâles que les vrais

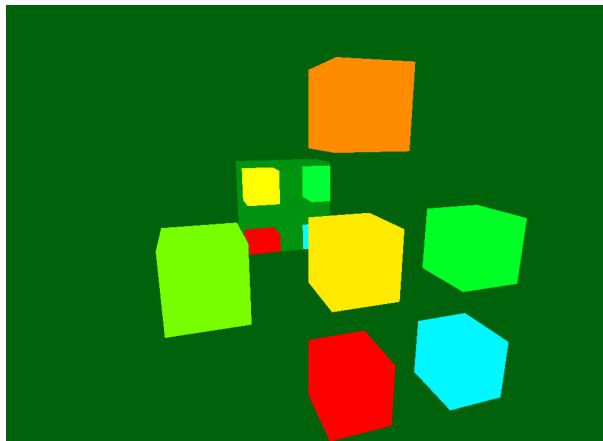


FIGURE 5.19 – Évidemment, la couleur de fond choisie par l'utilisateur est parfaitement compatible avec les miroirs

En effet, la charge de calcul pour les cubes est déjà terriblement élevée, et celle-ci ne fait que grandir très rapidement avec le nombre de triangles du miroir et des entités reflétées. Cela fait que, pour un cube ne reflétant que d'autres cubes, cela peut prendre jusqu'à 1 minute pour générer une seule frame. Si le cube doit refléter des sphères, ce temps augmente à plusieurs minutes, et si le miroir est lui-même une sphère, on peut avoir jusqu'à des heures d'attente si le miroir occupe une assez grande partie de l'écran (on envoie un rayon par pixel qui est au dessus d'un miroir après tout). C'est la raison pourquoi nous avons jugé inutile d'offrir l'option des sphères miroirs dans le menu, et nous recommandons fortement de n'utiliser que des cubes pour tester les miroirs.

Pour évidemment de complètement geler la fenêtre en permanence parce que chaque frame prend beaucoup de temps à se générer, les miroirs sont configurés en mode "manuel". Cela signifie que l'utilisateur doit spécifier au miroir qu'il veut le rafraîchir, et que sinon, le miroir ne se met pas à jour en temps réel.

Nous avons simplement, pour les besoins du travail, configuré l'événement qui met à jour les miroirs sur le relâchement d'une touche du clavier. L'utilisateur se positionne donc de façon fluide comme il veut, et ensuite il peut appuyer sur la touche de son choix pour démarrer l'attente de 30-60sec pour rendre le miroir avec le nouvel angle.

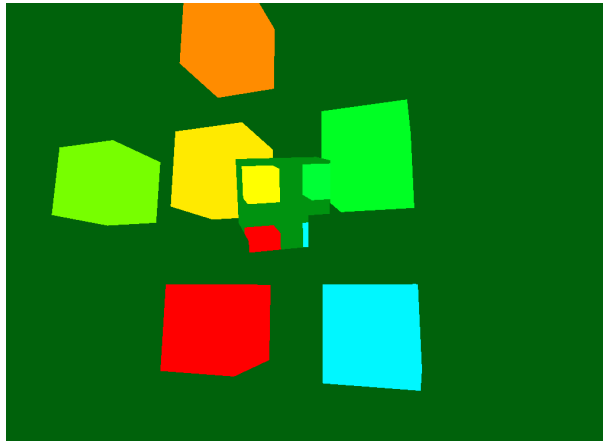


FIGURE 5.20 – Après un mouvement de caméra, le miroir ne s’est pas mis à jour à chaque frame, pour éviter des délais de plusieurs minutes

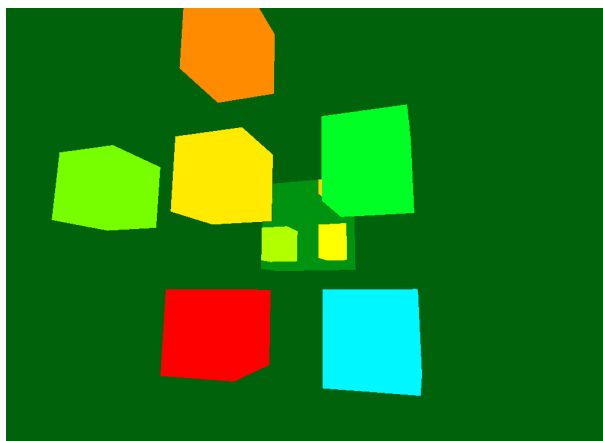


FIGURE 5.21 – L’utilisateur appuis sur une touche, attend la génération de la frame, et le miroir est maintenant à jour !

5.8.3 Réfraction

Non-implémenté

5.8.4 Ombrage

Non-implémenté

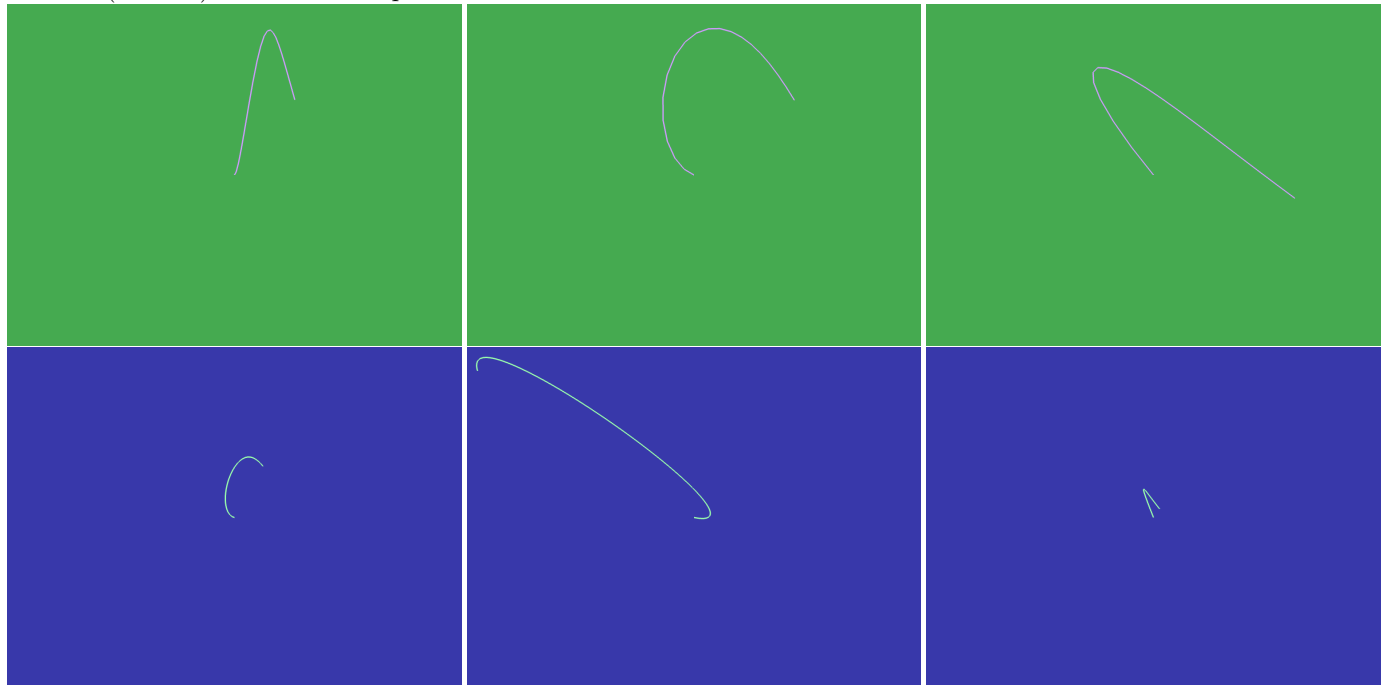
5.8.5 Rendu graphique

Non-implémenté

5.9 Topologie

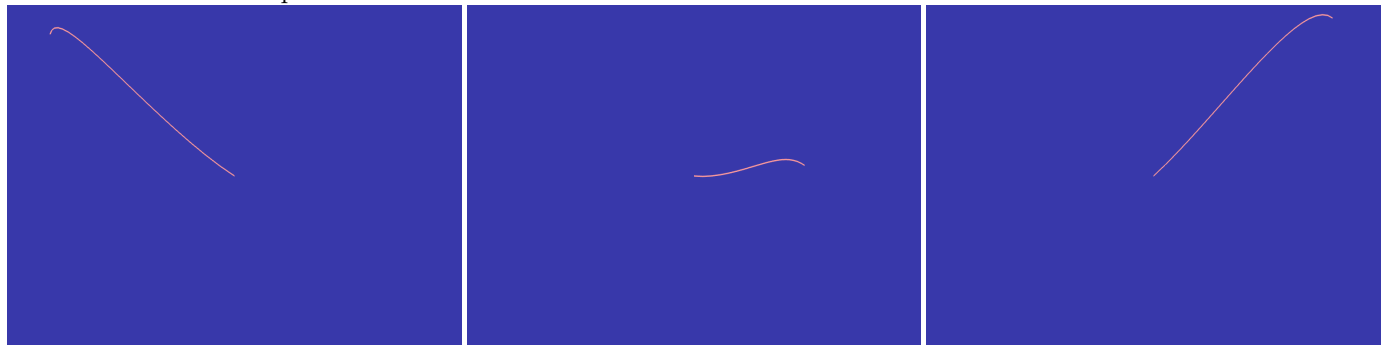
5.9.1 Courbe cubique

Dans le menu Types se trouve une catégorie Topologie. Celle-ci regroupe l'ensemble des fonctionnalités pour cette section. L'option Bezier et Hermite permettent respectivement de rendre les courbes associées à leur nom. Lorsque cette option sera choisie, le menu de position se mettra à jour pour permettre d'ajouter les positions initiales et finales (Largeur, hauteur et profondeur). Ensuite, dans la section Controle du menu, on retrouve les coordonnées des deux points de contrôle nécessaires au rendu. Les autres paramètres du menu sont également applicables. Les images suivantes donne en exemple une courbe de bezier et une courbe de hermite (en bleu) sous différents points de vue.



5.9.2 Courbe paramétrique

Dans le menu Types dans la catégorie Topologie se trouve également l'option CatmullRom. Celle-ci permet de rendre une courbe de Catmull-Rom en lui donnant un point initial et final et deux autres points de contrôle. Voici un exemple d'une courbe CatmullRom.



5.9.3 Surface paramétrique

Le dernier élément de la catégorie Topologie est la surface de Bézier. En sélectionnant cette option, on peut rendre une surface de Bézier cubique en donnant le point initial ainsi que la largeur et la hauteur. Ceci doit être fait dans le menu Position. Ensuite, on indique dans le menu Controle les 4 points de contrôle influençant la surface à rendre. En ajoutant la primitive, on peut alors voir l'influence de ceux-ci sur la surface. Voici un exemple d'une surface de bézier cubique :



5.9.4 Shader de tessellation

Non-implémenté

5.9.5 Triangulation

Non-implémenté

5.10 Techniques de rendu

5.10.1 Effet de relief

Non-implémenté

5.10.2 Cube de réflexion

Non-implémenté

5.10.3 BRDF

Non-implémenté

5.10.4 Effet en pleine fenêtre

Non-implémenté

5.10.5 Style libre

Non-implémenté

Chapitre 6

Ressources

Voici la liste des ressources externes que nous avons utilisées dans le cadre du projet

1. Notes de cours et exemple d'infographie
2. Modèle 3D d'Iron man téléchargé sur le site TF3DM à l'adresse : <http://tf3dm.com/3d-model/iron-man-36822.html>
3. Modèle d'Alduin téléchargé sur le site TF3M à l'adresse : <http://tf3dm.com/3d-model/alduin-15997.html>
4. Code pour produire une boîte de dialogue trouvé sur le site Microsoft Developer Network à l'adresse : [https://msdn.microsoft.com/en-us/library/windows/desktop/bb776913\(v=vs.85\).aspx?f=255#&MSPPErrors=-2147217396](https://msdn.microsoft.com/en-us/library/windows/desktop/bb776913(v=vs.85).aspx?f=255#&MSPPErrors=-2147217396)
5. ofxGui
6. ofxOpenCv
7. ofxAssimpModelLoader
8. ofxBezierSurface

Chapitre 7

Présentation

7.1 Liste des membres de l'équipe

Nom du membre	Programme d'étude
Jérémie Bolduc	Baccalauréat en Génie logiciel
Gabriel Chantal	Baccalauréat en Génie logiciel
Alex Gilbert	Baccalauréat en Informatique
Alexandre Mc Cune	Baccalauréat en Génie logiciel
Tania Toloza	Baccalauréat intégré en informatique et gestion

7.2 Présentation

L'équipe 04 a été formée dans une conversation sur messenger. L'équipe s'est formée à partir de deux duos, Alex et Jérémie, ainsi que Tania et Alexandre. Gabriel s'est par la suite ajouté à l'équipe, puisque les personnes avec qui il prévoyait de faire le travail ont abandonné le cours. La plupart des membres de l'équipe se connaissaient depuis le cégep, ce qui leur a permis de travailler efficacement en se basant sur les forces et les faiblesses des autres.