



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 xxx xxx	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 Fonctionnalités	4
4.1 Image	4
4.1.1 Importation	4
4.1.2 Exportation	4
4.1.3 Espace de couleur	4
4.1.4 Traitement d'image	4
4.1.5 Image procédurale	5
4.2 Dessin vectoriel	6
4.2.1 Curseur dynamique	6
4.2.2 Primitives vectorielles	7
4.2.3 Formes vectorielles	8
4.2.4 Outils de dessin	8
4.2.5 Interface	9
4.3 Transformation	10
4.3.1 Transformation interactive	10
4.3.2 Structure de scène	10
4.3.3 Sélection multiple	14
4.3.4 Coordonnées non-cartésiennes	17
4.3.5 Historique	17
4.4 Géométrie	18
4.4.1 Particules	18
4.4.2 Primitives	19
4.4.3 Modèle	22
4.4.4 Texture	25
4.4.5 Géométrie procédurale	25
4.5 Caméra	26
4.5.1 Propriétés de caméra	26
4.5.2 Mode de projection	26
4.5.3 Caméra interactive	26
4.5.4 Caméra multiple	27
4.5.5 Caméra animée	27

5 Ressources	28
---------------------	-----------

6 Présentation	29
-----------------------	-----------

Table des figures

4.1	Une scène avec filtres.	4
4.2	Curseur normal de type "flèche"	6
4.3	Curseur pour bouton de type "main"	6
4.4	Curseur slider de type "slider"	6
4.5	Une scène avec primitives 2d.	8
4.6	Cornet de crème glacé!	9
4.7	Menu des transformations.	10
4.8	Menu de selection	14
4.9	Plusieurs entités sélectionnées	14
4.10	La selection est en wireframe	15
4.11	Chaque entité de la selection reçoit la transformation	16
4.12	Options de création d'une primitive	19
4.13	Bouton pour importer un modèle	23
4.14	Propriété de la caméra dans l'interface	26

Chapitre 1

Sommaire

Ce document présente l'évolution du projet de session réalisé par l'équipe 4 dans le cadre du cours d'infographie IFT 3100. Le corps du document présente le projet dans l'état actuel. En annexe, vous retrouverez une copie complète des anciens rapports afin de pouvoir voir l'évolution du projet au fil de la session.

Chapitre 2

Interactivité

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Curabitur odio nisl, feugiat quis quam non, consectetur tempus leo. Etiam nec enim lacus. In porta tempor nisi. Aenean fermentum, sapien at tincidunt pharetra, nibh nunc vehicula urna, sed scelerisque elit risus at ex. Donec egestas, turpis a pellentesque posuere, nibh tellus malesuada elit, sit amet porttitor enim eros a felis. Integer non congue enim. Donec bibendum ex id elementum rutrum. Donec porta nunc et odio gravida, vel vehicula orci aliquet.

Pellentesque gravida fermentum lectus, in laoreet sapien facilisis laoreet. Nunc sit amet leo volutpat, ornare lorem ut, hendrerit ex. Donec lectus augue, interdum in placerat in, dignissim dictum diam. Mauris tincidunt leo nisl, eu convallis odio consectetur id. Vestibulum placerat sem non mattis convallis. Etiam quis lorem imperdiet, gravida felis a, venenatis justo. Praesent eu lorem diam. Phasellus purus mi, tincidunt quis sapien iaculis, eleifend hendrerit est. Sed in justo efficitur, vulputate massa nec, rhoncus tortor. Cras risus nisl, finibus non felis vitae, malesuada sollicitudin ex. Donec finibus sit amet nisi at condimentum. Vivamus vitae libero semper, iaculis orci eget, porttitor sem. Phasellus eget hendrerit mi. Ut feugiat, nulla eu pretium egestas, dui est pretium eros, et tristique ligula magna sed purus

Chapitre 3

Technologie

Chapitre 4

Fonctionnalités

4.1 Image

4.1.1 Importation

Non implémenté

4.1.2 Exportation

Implémenté

4.1.3 Espace de couleur

Implémenté

4.1.4 Traitement d'image

Une catégorie situé au coin droit de l'écran permet l'utilisation de plusieurs filtres s'affichant sur l'entièreté de la scène. Elle regroupe trois type de filtres : le brouillage (Blur), l'inversement 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 4.1 – Une scène avec filtres.

4.1.5 Image procédurale

Non implémenté

4.2 Dessin vectoriel

4.2.1 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

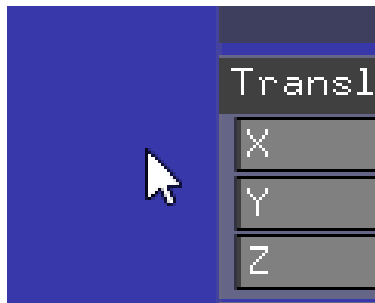


FIGURE 4.2 – Curseur normal de type "flèche"

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



FIGURE 4.3 – 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 4.4 – 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 ofxButton::mouseMoved(ofMouseEventArgs & args){

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

template<typename Type>
bool ofxSlider<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;
}

```

4.2.2 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 affectés 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 créer. 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 ressemble les différentes primitives 2D :

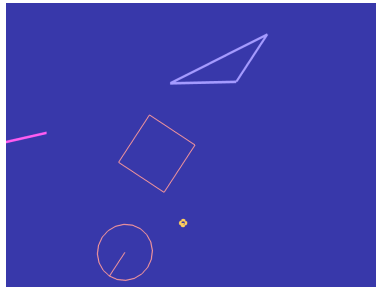


FIGURE 4.5 – Une scène avec primitives 2d.

4.2.3 Formes vectorielles

La forme qui a été implémenté est un cornet de crème glacé, 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 !

4.2.4 Outils de dessin

Implémenté

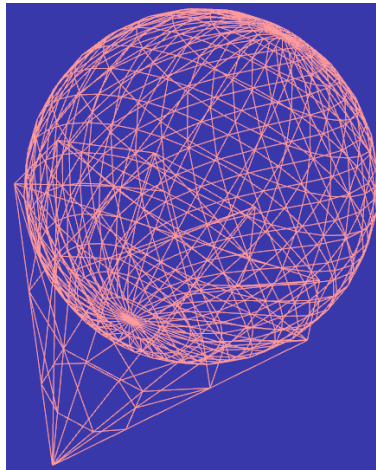


FIGURE 4.6 – Cornet de crème glacé!

4.2.5 Interface

Implémenté

4.3 Transformation

4.3.1 Transformation interactive

Une catégorie à droite de l'application permet de transformer le systèmes de coordonnées des entités géométriques de la scène. On peut exercer des transformations tel 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 utilisés pour la réalisation des transformations. On utilise les méthodes `ofPushMatrix()` et `ofPopMatrix()`. Une fois la matrice empilé, on ajoute les transformation à l'aide des méthodes `ofTranslate()`, `ofRotate()` et `ofScale()` selon les paramètres inscrit par l'utilisateur. Enfin, on dépile la matrice pour permettre la transformation.

Voici le menu des transformations et ces différentes options :

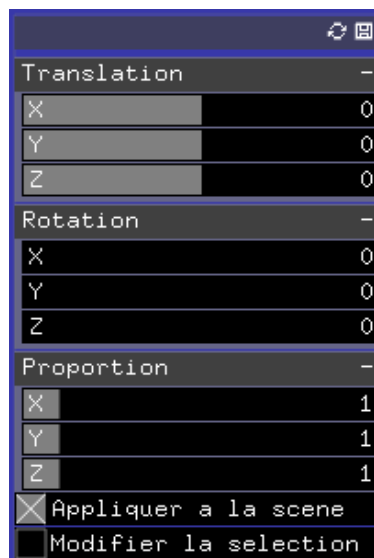


FIGURE 4.7 – Menu des transformations.

4.3.2 Structure de scène

La structure de scène a été implémenté 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 lieux des simples pointeurs pour conserver les éléments qui sont ajouté 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->get()->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 utilisé à son plein potentiel et tous les éléments sont stocké 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`».

4.3.3 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.

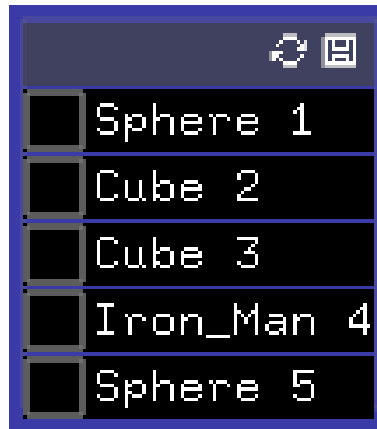


FIGURE 4.8 – Menu de selection

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

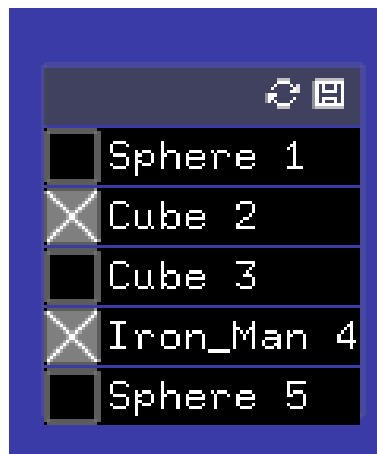


FIGURE 4.9 – Plusieurs entités sélectionnées

Lorsqu'on 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.

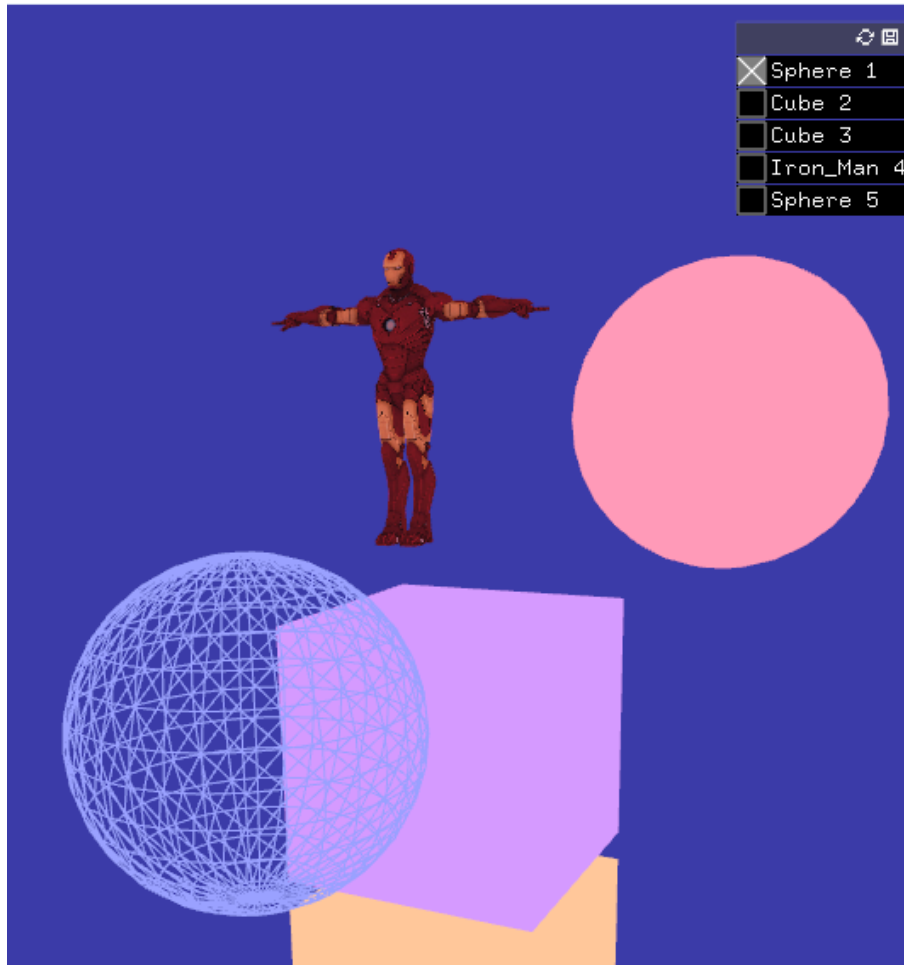


FIGURE 4.10 – La selection est en wireframe

Les transformation 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.

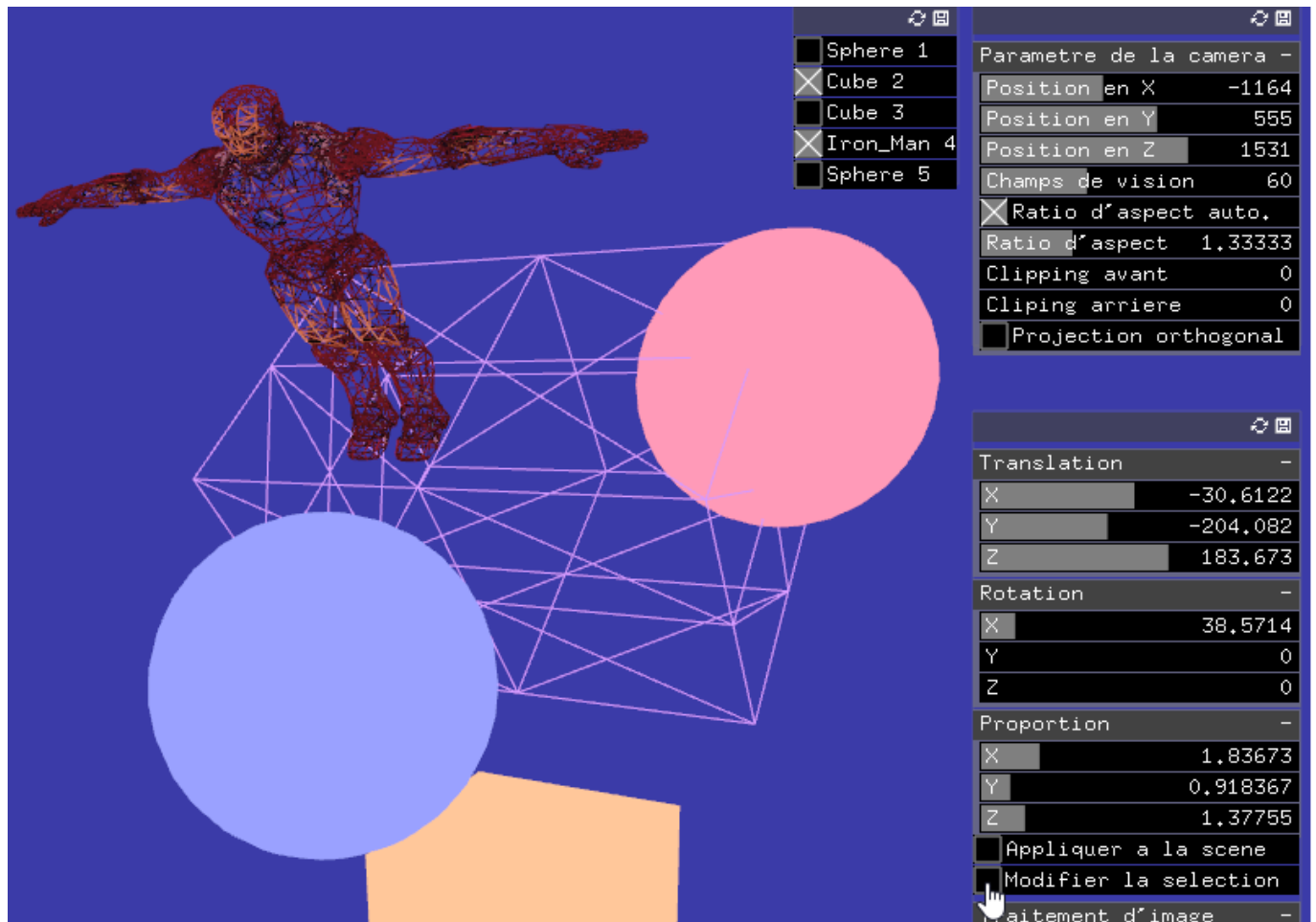


FIGURE 4.11 – Chaque entité de la selection reçoit la transformation

```
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();
}
```

4.3.4 Coordonnées non-cartésiennes

Non-implémenté

4.3.5 Historique

Non-implémenté

4.4 Géométrie

4.4.1 Particules

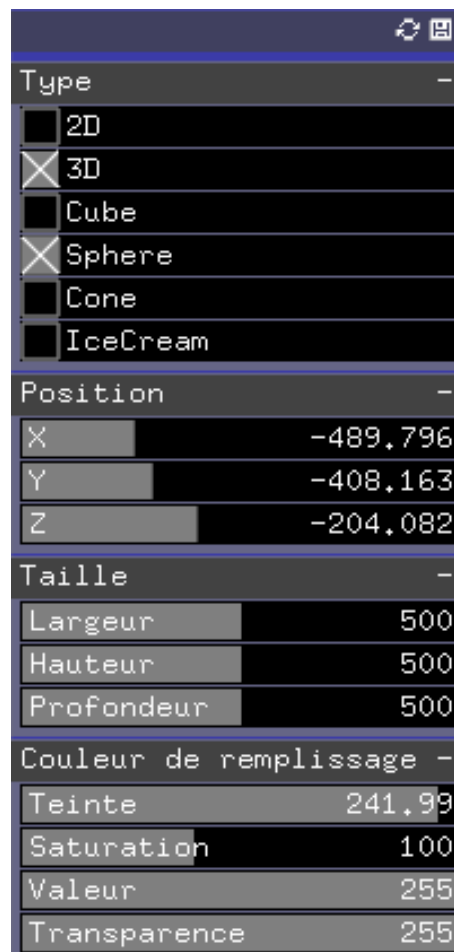
Non-implémenté

4.4.2 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 modifiées 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 quel 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.



The image shows a software interface for creating 3D primitives. It features several sections with labels and input fields. The 'Type' section has checkboxes for 2D, 3D (checked), Cube, Sphere (checked), Cone, and IceCream. The 'Position' section has input fields for X (-489.796), Y (-408.163), and Z (-204.082). The 'Taille' section has input fields for Largeur (500), Hauteur (500), and Profondeur (500). The 'Couleur de remplissage' section has input fields for Teinte (241.99), Saturation (100), Valeur (255), and Transparence (255). The interface has a dark theme with light text and a blue header bar.

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 4.12 – 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(scen->nbElements() + 1));
    scen->addElement(prim);
    return prim.selected;

    //cout << *scn;
}

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;
}

```

4.4.3 Modèle

Il est possible pour un utilisateur d'importer un modèle choisit 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

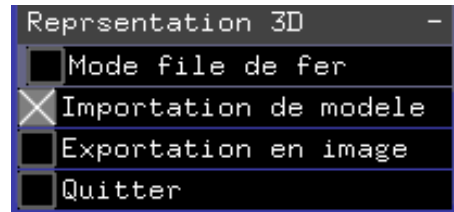


FIGURE 4.13 – Bouton pour importer un modèle

message sera affiché à l'utilisateur pour l'informer.

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"Succes", 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"Echec", 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);
}
```

4.4.4 Texture

Implémenté

4.4.5 Géométrie procédurale

Non-implémenté

4.5 Caméra

4.5.1 Propriétés de caméra

Les propriétés de la caméra tel que le champ de vision, le ratio d'aspect ainsi que la distance du plan de clipping avant et arrière. Ils peuvent être modifier 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());
```

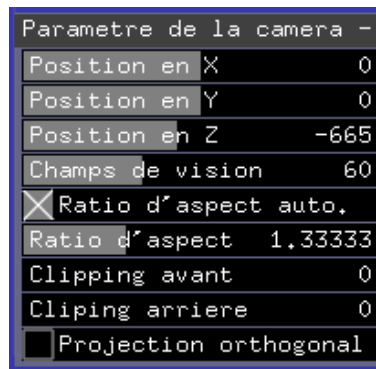


FIGURE 4.14 – Propriété de la caméra dans l'interface

4.5.2 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();
}
```

4.5.3 Caméra interactive

La caméra interactive est implémenter dans l'application principalement à l'aide de la classe ofEasyCam de openFrameworks. Nous avons tout de même ajouter 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()));
```

4.5.4 Caméra multiple

Non-implémenté

4.5.5 Caméra animée

Non-implémenté

Chapitre 5

Ressources

Chapitre 6

Présentation