



## Travail Pratique #2

Application graphique Paint3D+

présenté à

**Philippe Voyer**

par

Équipe 23

Université Laval  
13 Mars 2016

# Sommaire

L’application Paint3D+ que nous développons est un programme d’édition graphique interactif dans le style de PaintdotNet. Elle comporte 2 modes : mode 2D et mode 3D.

Concrètement, l’application permet de construire des primitives vectorielles en 2D (ligne, triangle, rectangle et cercle) et d’afficher un modèle 3D pour ensuite modifier ses propriétés. Il est aussi possible d’importer des images et d’exporter le contenu de la scène en image.

Une interface intuitive est affichée lors du démarrage du programme et l’utilisateur peut interagir à l’aide des menus et des panneaux graphiques. D’autres possibilités de manipulation sont aussi décrites plus loin dans le rapport.

Toutes les entités géométriques sont organisées dans une hiérarchie de classes et elles peuvent être manipulées par différentes méthodes, telles que l’application de translations, rotations, changement d’échelle, etc. Les primitives peuvent aussi être composées dans une seule entité.

# Interactivité

## 2.1 Aperçus de l'interface graphique

Voici quelques aperçus de l'interface graphique aux figures 2.1, 2.2, 2.3, 2.4.

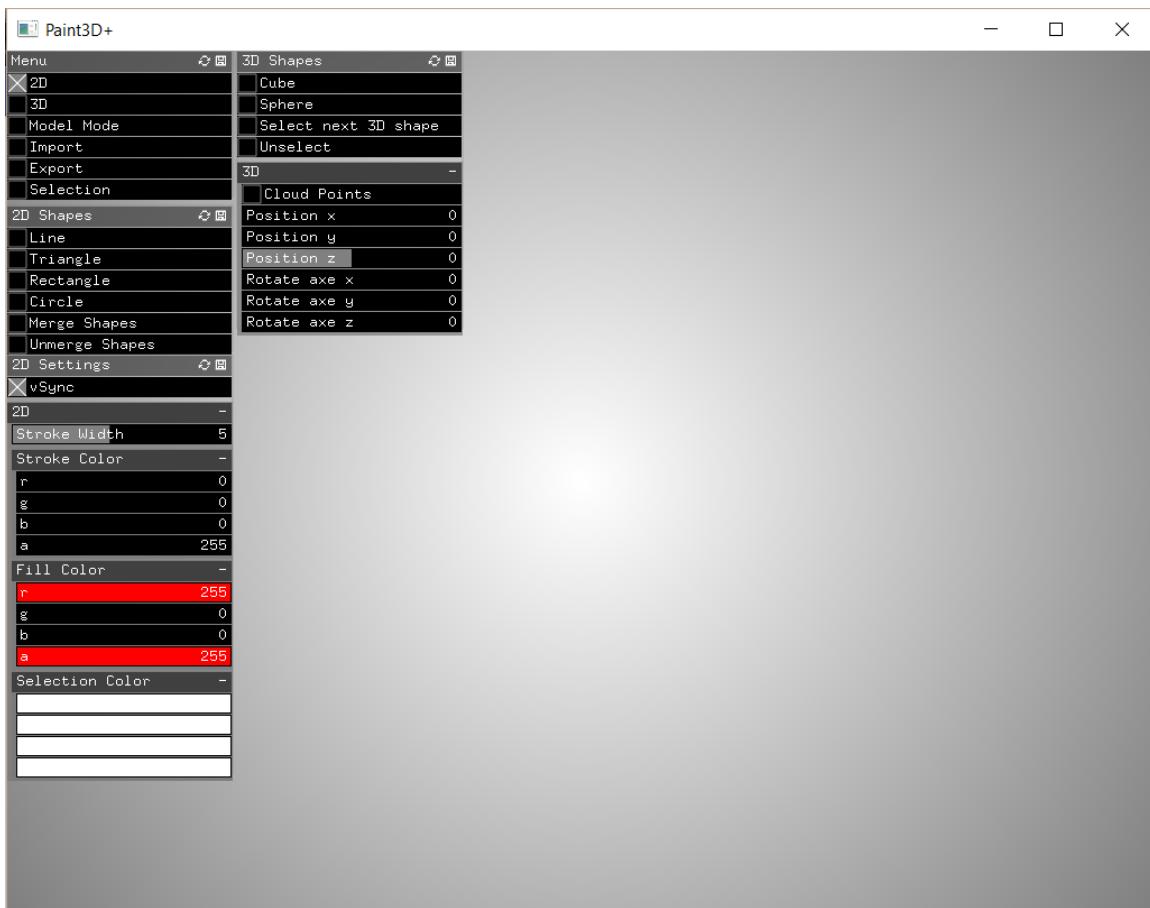


FIGURE 2.1 – Interface graphique à l'ouverture

Voici un aperçu de l'interface graphique lors de l'ajout de formes 2D.

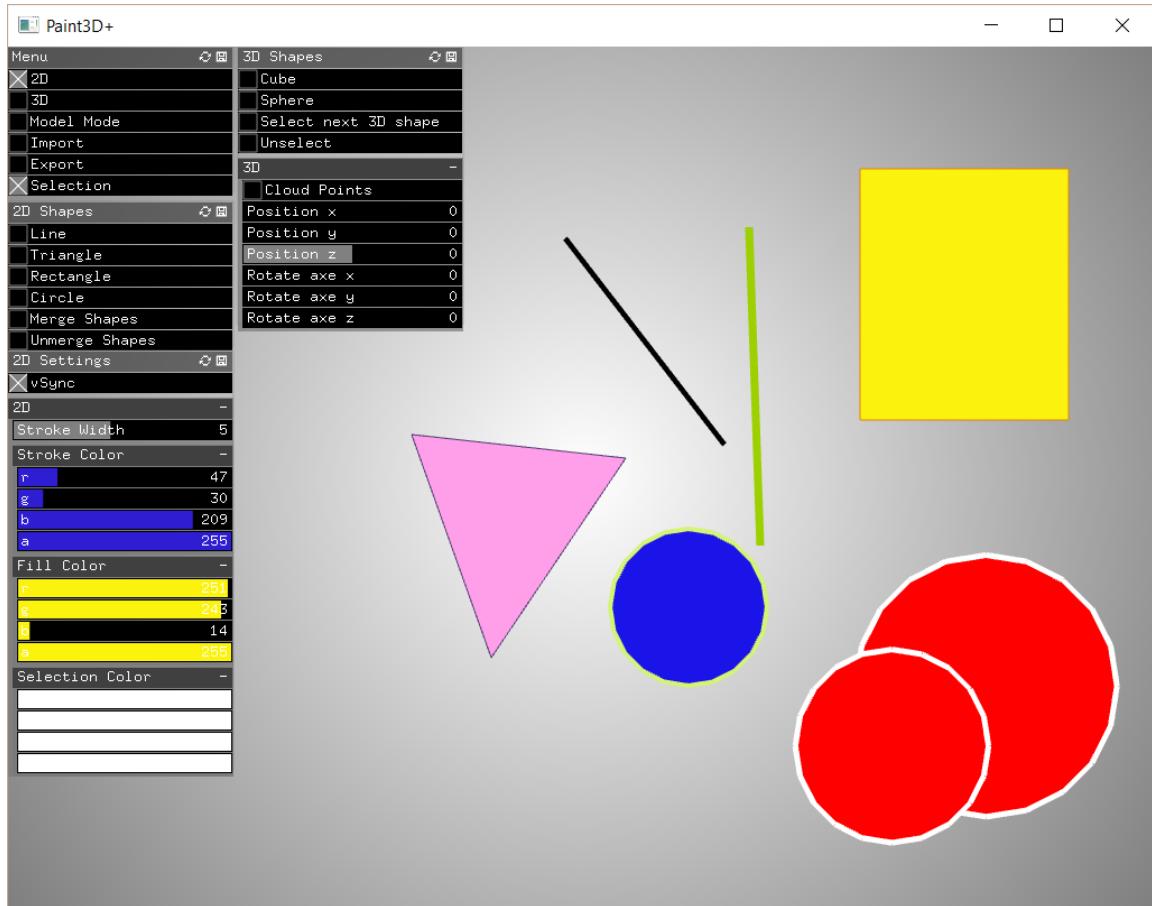


FIGURE 2.2 – Interface graphique en mode 2D

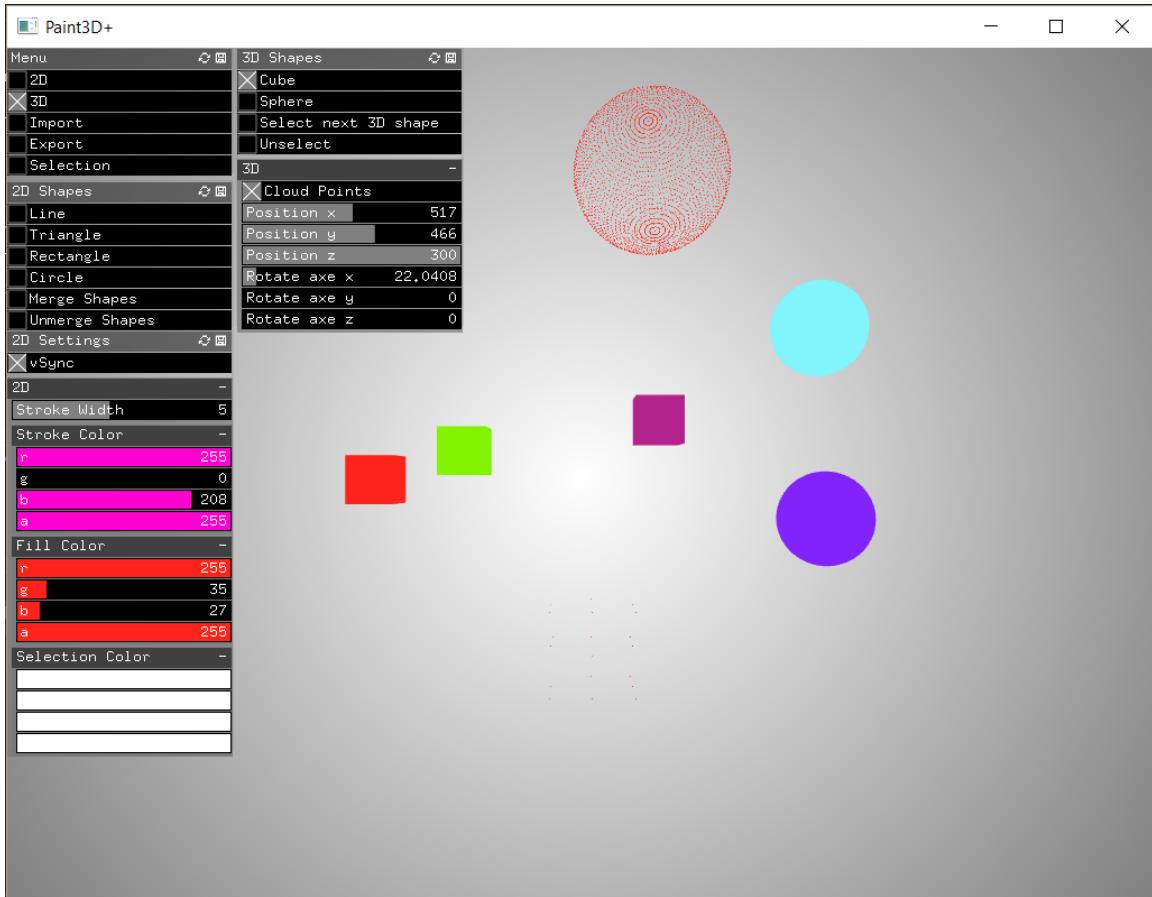


FIGURE 2.3 – Interface graphique en mode 3D

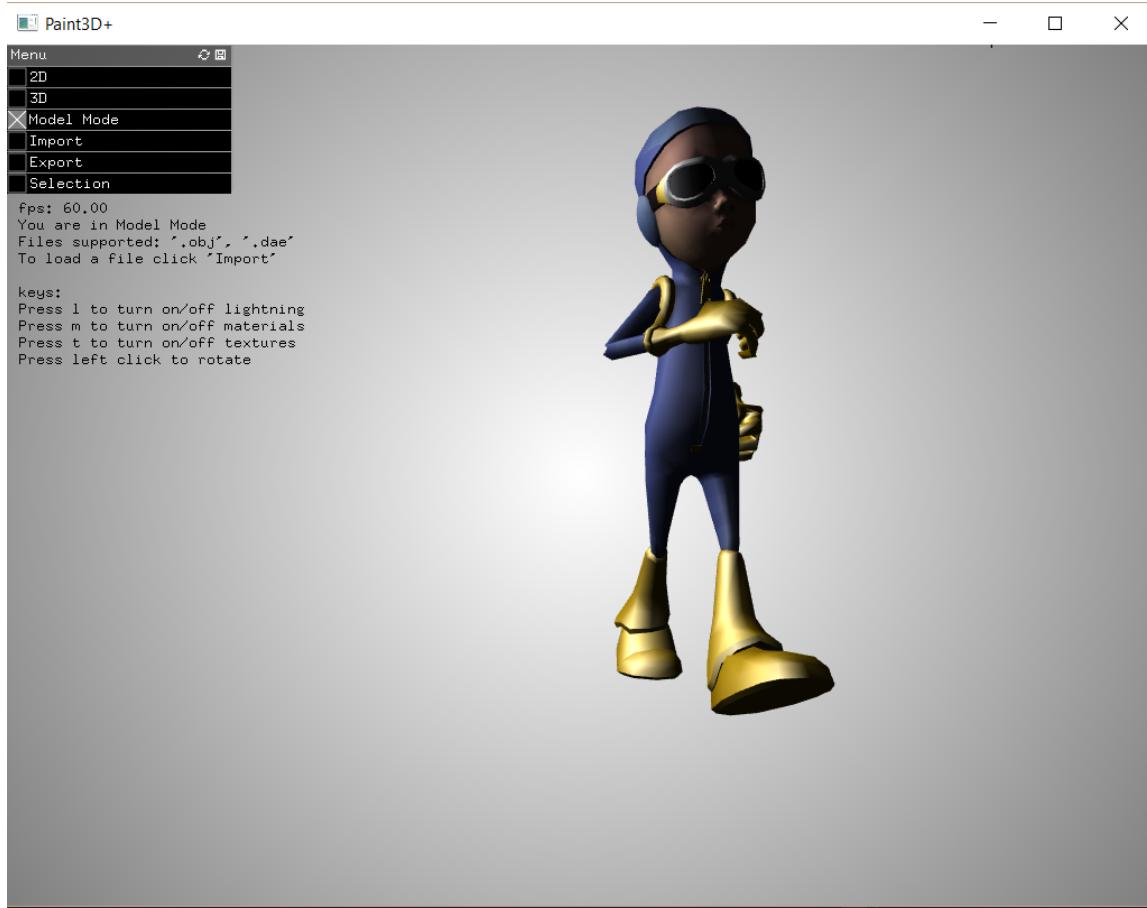


FIGURE 2.4 – Interface graphique en mode modèle

## 2.2 Détails de l’interface graphique

L’interface graphique permet de choisir une scène de rendu 2D, 3D ou de modèle selon le choix de l’utilisateur. De plus, il est possible d’importer des images en mode 2D et d’exporter la scène sous forme d’image de format png à l’aide d’un dialogue de d’importation ou d’exportation. Le dernier item de la section Menu permet la sélection des différents objets de la scène 2D.

- **2D** : Permet l’édition d’une scène de rendu 2D.
- **3D** : Permet l’édition d’une scène de rendu 3D.
- **Model Mode** : Permet l’importation d’un modèle 3D.
- **Import** : Permet l’importation d’une image dans la scène.
- **Export** : Permet l’expotation de la scène de rendu, sans les différents menus, sous forme d’une image de format png.
- **Selection** : Permet la sélection d’un ou de plusieurs objets 2D.

### 2.2.1 2D Shapes

Il est possible de dessiner quatre sortes de primitives vectorielles, en plus d'offrir la possibilité de fusioner ces primitives, une fois qu'elles sont créées, ou de les défusionner avec les boutons Merge Shapes et Unmerge Shapes.

Dans le cas de la ligne, du rectangle et du cercle, l'utilisateur clique en un premier point et se déplace dans l'écran afin de cliquer une seconde fois et former la primitive.

Pour ce qui est du triangle, l'utilisateur doit cliquer en trois points différents.

- **Line** : Permet la création d'une ligne.
- **Triangle** : Permet la création d'un triangle.
- **Rectangle** : Permet la création d'un rectangle.
- **Circle** : Permet la création d'un cercle.
- **Merge Shapes** : Permet la création d'un objet 2D regroupant les formes sélectionnées.
- **Unmerge Shapes** : Permet de détruire l'objet 2D issu d'une fusion de formes. Les primitives vectorielles peuvent donc être manipulées individuellement.

### 2.2.2 2D

Cette section de l'interface graphique regroupe les options de lignes de contour et de remplissage.

- **Stroke Width** : Permet l'ajustement de la largeur de la ligne de contour entre 1 et 10.
- **Stroke Color** : Permet l'ajustement de la couleur de la ligne de contour sous forme RGBA.
- **Fill Color** : Permet l'ajustement de la couleur de remplissage sous forme RGBA.
- **Selection Color** : Permet l'ajustement de la couleur de sélection sous forme RGBA. Cette couleur permet de déterminer quel ou quels objets 2D sont sélectionnés en modifiant la couleur de la ligne de contour lors d'une sélection.

### 2.2.3 3D Shapes

Cette section de l'interface graphique permet de faire l'ajout de formes 3D lorsque le mode 3D est activé.

- **Cube** : Permet d'ajouter un cube.
- **Sphere** : Permet d'ajouter une sphère.
- **Select next 3D shape** : Permet de parcourir et sélectionner les formes 3D une à une.
- **Unselect** : Permet de désélectionner une objet 3D.

## 2.2.4 3D

Cette section de l'interface graphique permet de faire l'ajout de formes 3D lorsque le mode 3D est activé.

- **Cloud points** : Permet d'ajouter un cube.
- **Position x/y/z** : Permet de changer la position de l'objet 3D selon l'axe désiré.
- **Rotate x/y/z** : Permet d'effectuer une rotation de l'objet 3D autour de l'axe désiré.

## 2.3 Utilisation de la souris et raccourcis clavier

Lorsque l'utilisateur sélectionne l'une ou l'autre des primitives vectorielles, le curseur change de forme afin de permettre à ce dernier de visualiser dans quel mode il se situe. Pour la sélection il s'agit d'un curseur normal, tandis que pour les différentes formes, le curseur prend l'apparence de la forme sélectionnée. Lorsque la souris se retrouve au dessus des différents panneaux de configuration le curseur reprend sa forme normale.

Voici les différentes fonctions de la souris selon les modes.

### 2.3.1 2D

- **Clic gauche** : Permet la sélection d'un objet 2D.
- **Clic gauche et déplacement** : Permet la sélection d'objets dont au moins un coin touche à la zone de sélection tracée.
- **Clic droit et ctrl** : Permet la sélection multiple.
- **Clic droit et déplacement** : Permet le déplacement d'un ou de plusieurs objet 2D.
- **Clic droit et gauche** : Permet la rotation d'un ou de plusieurs objets 2D.
- **Clic gauche et déplacement** : Permet d'ajuster la taille d'un ou de plusieurs objets 2D.
- **Touche supprimer** : Permet de supprimer un objet 2D.

### 2.3.2 3D

- **Touche supprimer** : Permet la suppression de l'objet sélectionné.

### 2.3.3 Model Mode

Lorsqu'on entre en mode modèle l'interface graphique change quelque peu afin d'afficher les raccourcis clavier à l'écran.

- **Clic gauche et déplacement** : Permet une rotation du modèle.
- **Touche l** : Permet d'appliquer ou non une lumière.
- **Touche m** : Permet d'appliquer ou non un matériau.
- **Touche t** : Permet d'appliquer ou non une texture.

# Technologie

Voici les principaux outils utilisés pour la réalisation du projet.

Tout d'abord, OpenFrameworks est utilisé en tant que framework de programmation graphique. Pour ce qui est de l'environnement de développement, Visual Studio est celui employé.

# Architecture

Paint3D+ est organisée en plusieurs classes qui supportent ces fonctionnalités. La structure hiérarchique permet de minimiser le volume du code nécessaire et de faciliter l'implémentation de plusieurs fonctions (comme les transformations interactives, la sélection multiple, etc.). Comme le programme fonctionne en 3 modes différents, on peut grouper les classes en 3 parties respectivement : les classes qui supportent le mode 2D, le mode 3D et la partie du rendu qui est construite autour de la classe "OfApp" (fournie initialement par OpenFrameworks).

La vue d'ensemble est représentée dans la figure suivante :

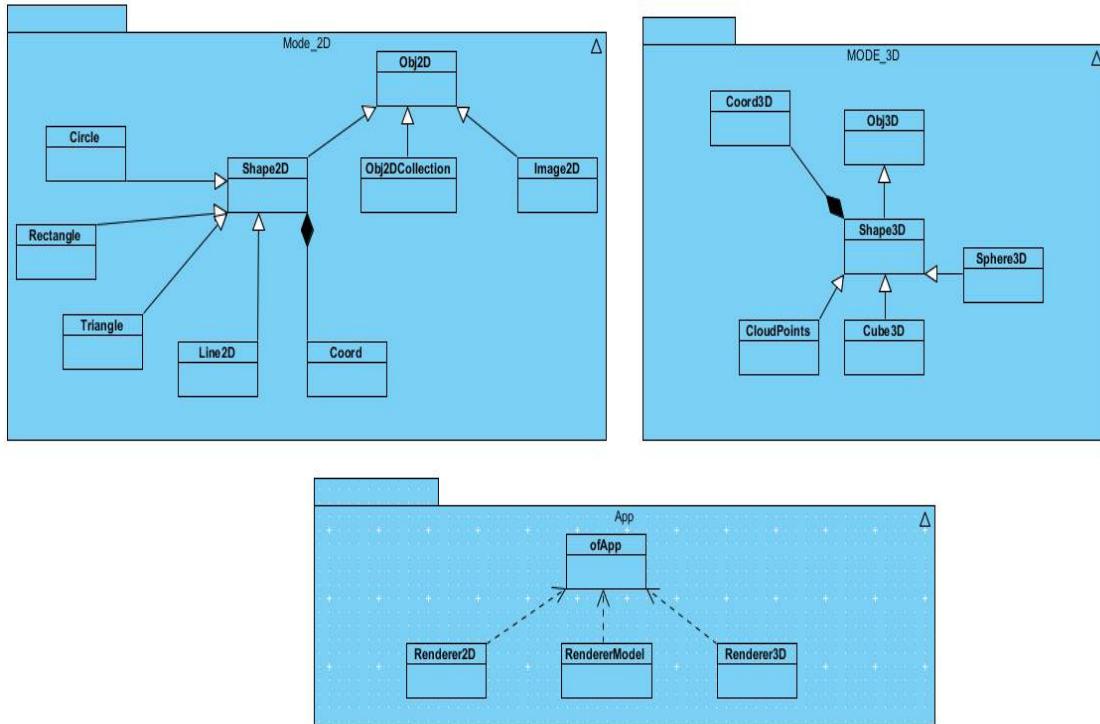
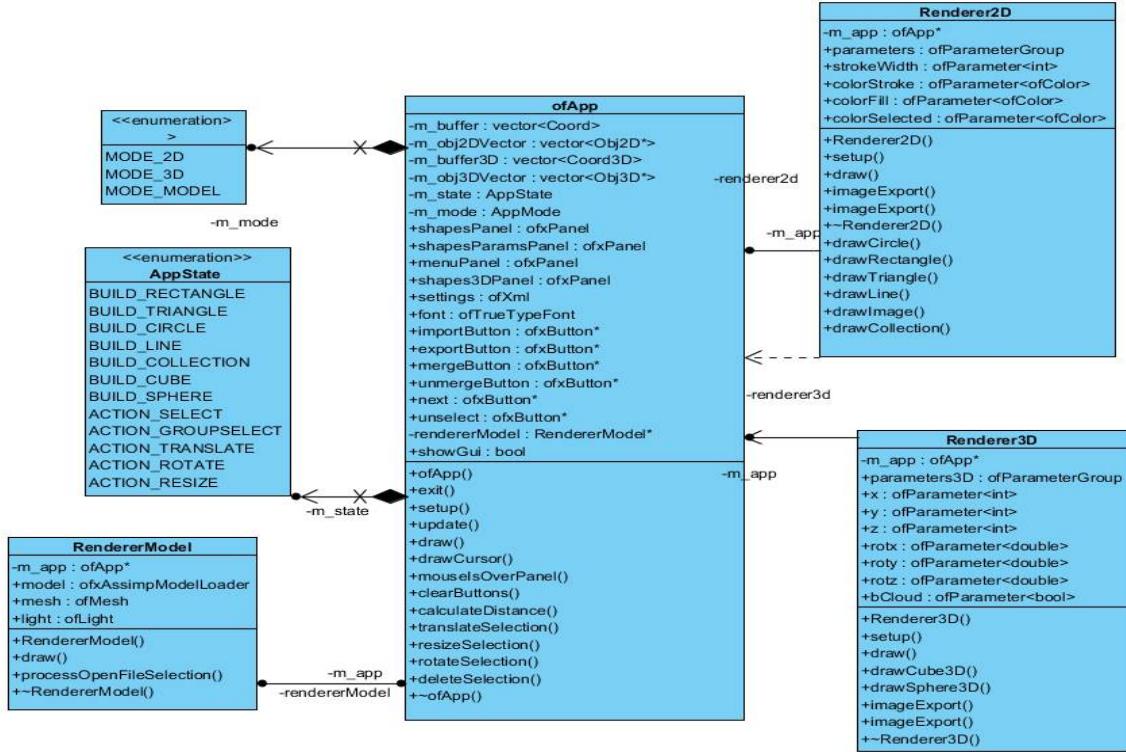


FIGURE 4.1 – Diagramme UML vue d'ensemble.

Les détails pertinents de chaque classe sont présents dans les diagrammes suivants :

FIGURE 4.2 – Diagramme UML de `ofApp`.

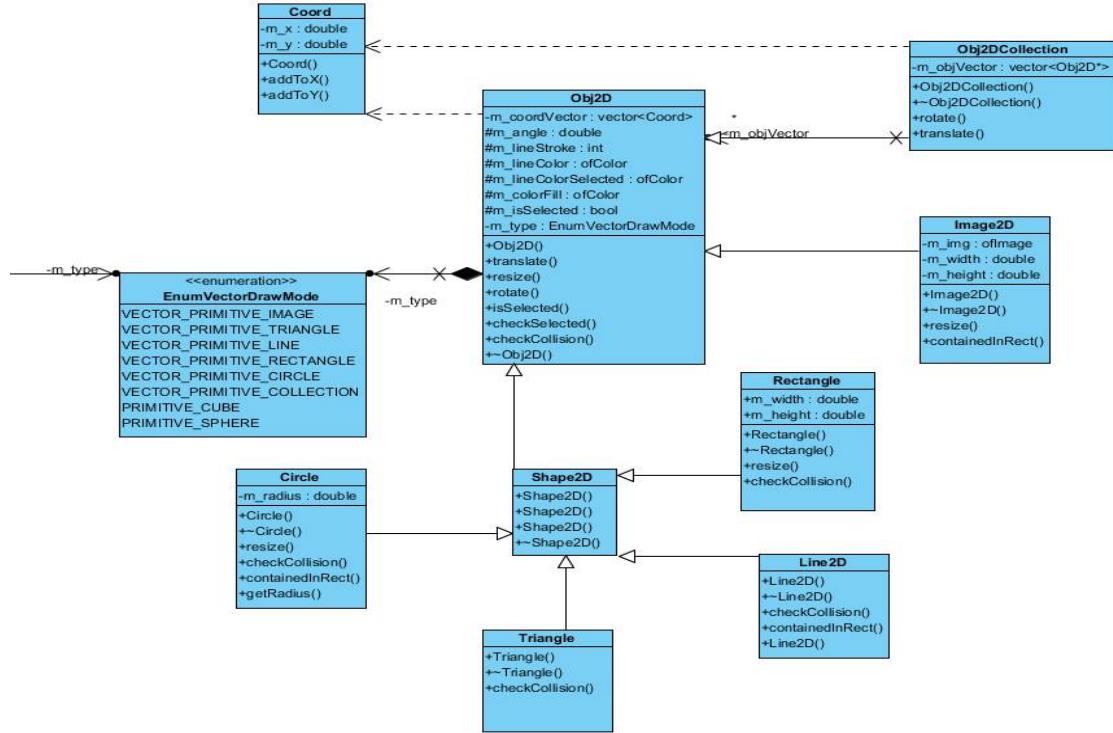


FIGURE 4.3 – Diagramme UML du 2D.

Il faut tenir compte que le rôle de ces diagrammes est de montrer l'architecteur de l'application et la logique de fonctionnement (par exemple les méthodes "set" et "get") ont été omises pour simplifier la complexité des diagrammes. Pour plus de détails, il faut consulter leur implémentation (le code dans les fichiers .h et .cpp respectivement).

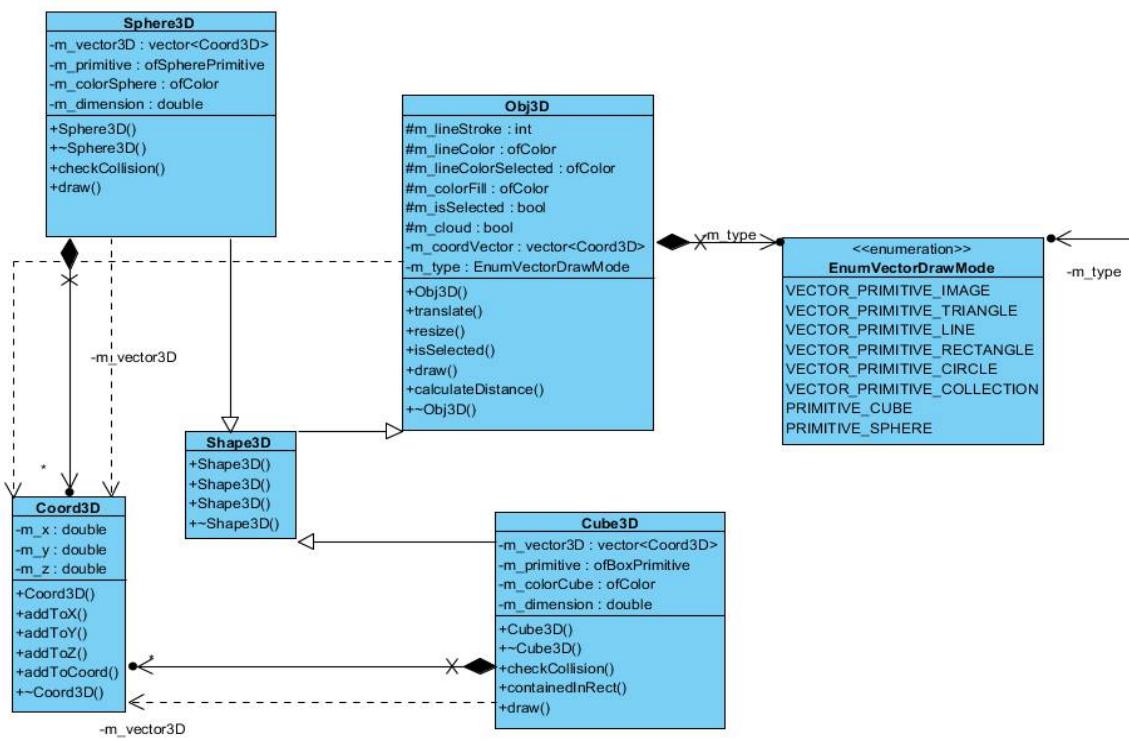


FIGURE 4.4 – Diagramme UML du 3D.

# Fonctionnalités

## 5.1 Image

### 5.1.1 Importation

Pour satisfaire ce critère, l'application doit pouvoir importer des fichiers images (.jpg, .png, etc.) pour les utiliser à des fins de rendu graphique.

L'application permet cette fonctionnalité grâce à l'utilisation du bouton "Import". Celui-ci ouvre un navigateur de fichier qui permet à l'utilisateur de sélectionner facilement le fichier image voulu. Cette image est placée dans le renderer 2D, initialement avec sa taille d'origine et à la position (0,0).

Les figures suivantes démontrent cette fonctionnalité.

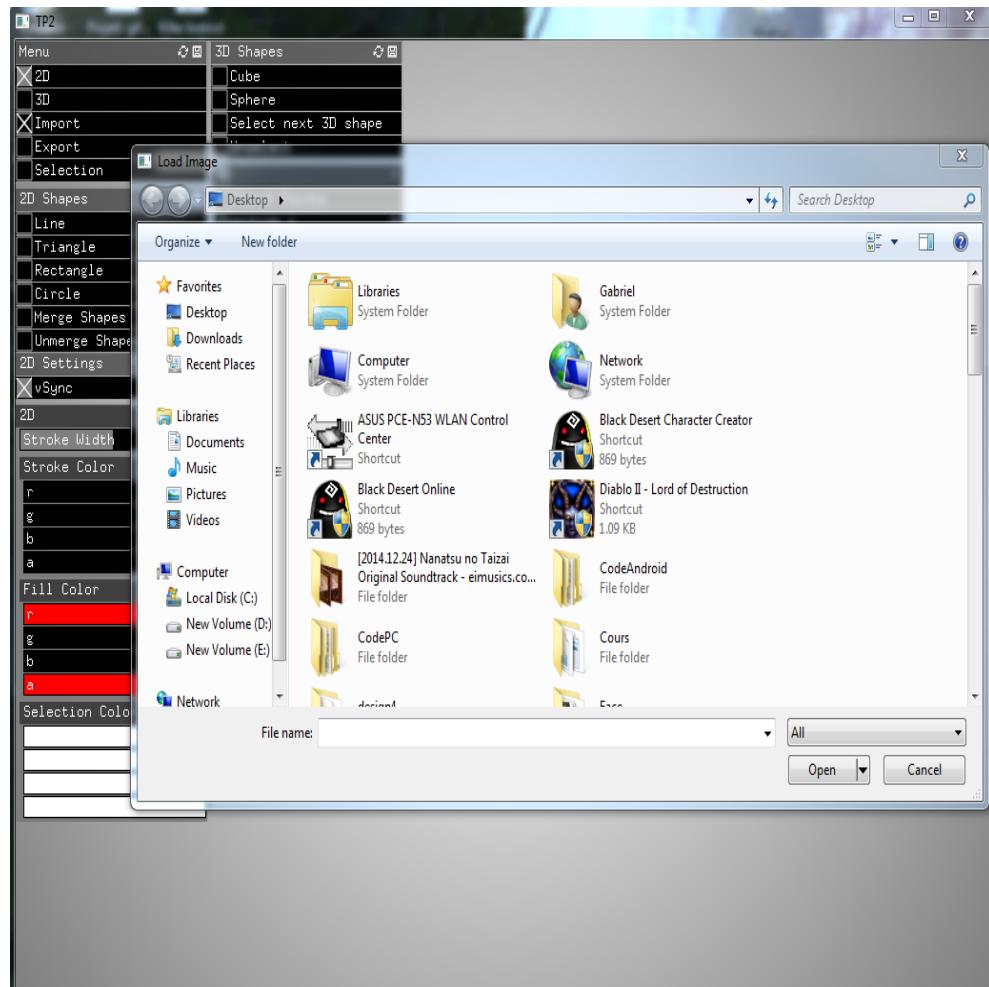


FIGURE 5.1 – Le bouton Import et son navigateur.

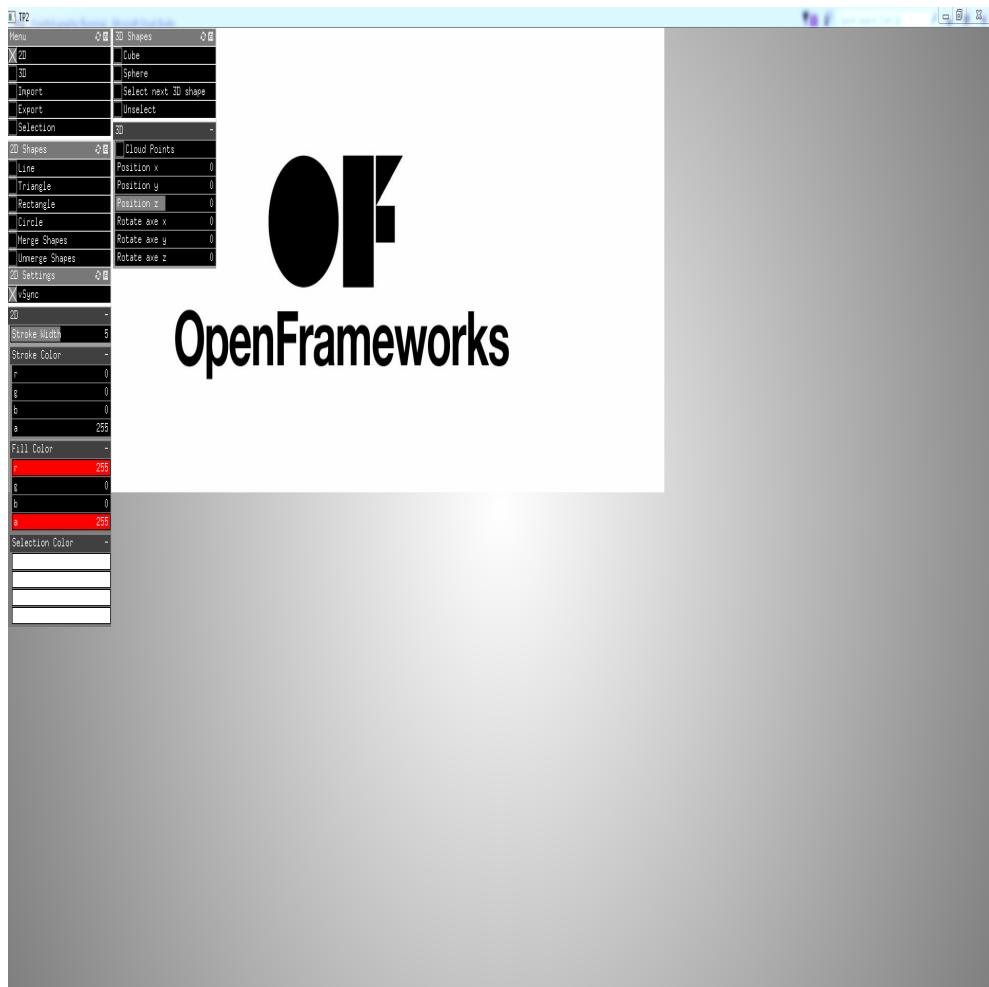


FIGURE 5.2 – Une image importée dans le renderer 2D.

```

if (_mode == MODE_2D) {
    if (btnName == "Import") {
        ofFileDialogResult file = ofSystemLoadDialog("Load Image", false);
        if (file.getPath() != "") {
            app::Image2D *newImage = new app::Image2D(file.getPath(), 0, renderer2d->strokeWidth.get(),
                renderer2d->colorStroke.get(), renderer2d->colorSelected.get(), renderer2d->colorFill.get());
            if (newImage->getImage().getTextureReference().isAllocated()) {
                _obj2DVector.push_back(newImage);
            }
        }
    }
}

```

FIGURE 5.3 – Code important de la fonctionnalité Import.

### 5.1.2 Exportation

Pour satisfaire ce critère, l'application doit pouvoir exporter des fichiers images (dans notre cas une image .png) et de la sauvegarder sur l'ordinateur utilisé.

L'application permet cette fonctionnalité grâce à l'utilisation du bouton "Export". Celui-ci ouvre un navigateur de fichier qui permet à l'utilisateur de choisir facilement l'emplacement de sauvegarde ainsi que le nom du fichier sauvegardé. Cette image est celle rendue sur le renderer 2D.

Les figures suivantes démontrent cette fonctionnalité.

endue sur le renderer 2D.

Les figures suivantes démontrent cette fonctionnalité.

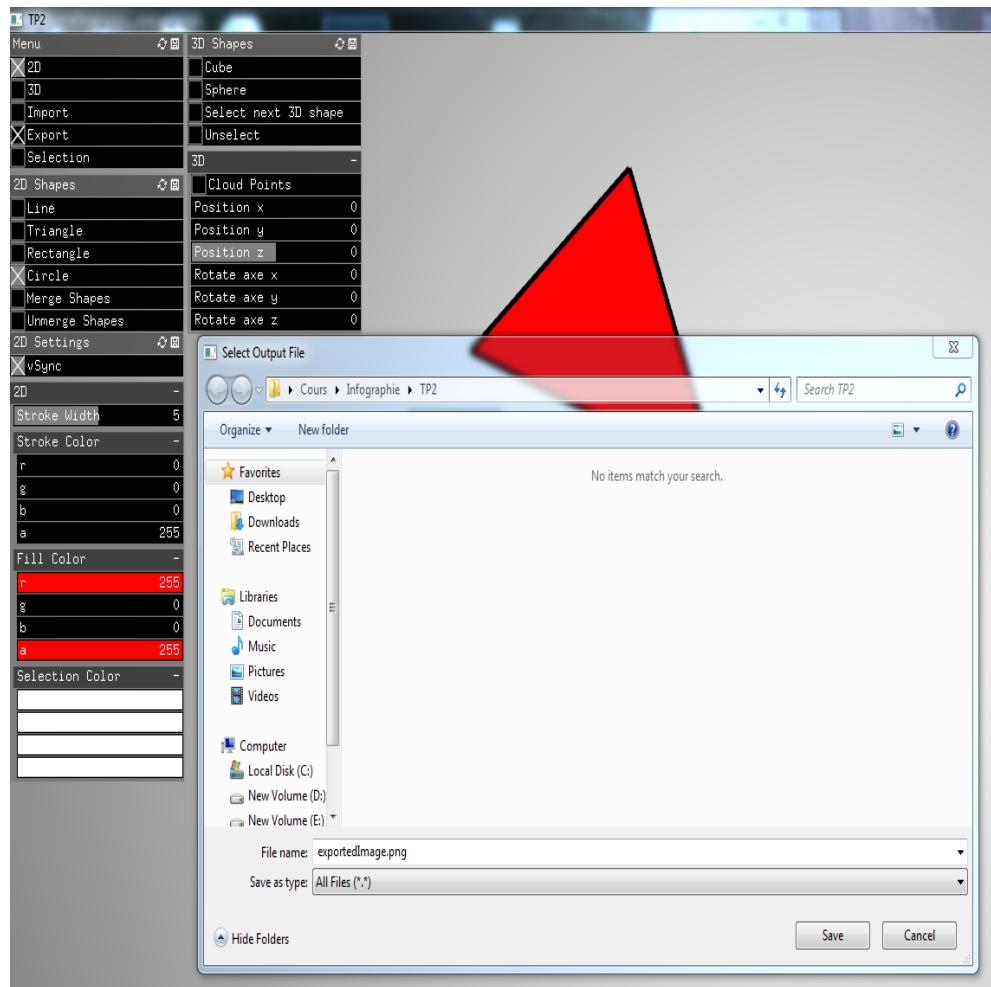


FIGURE 5.4 – Le bouton Export et son navigateur.

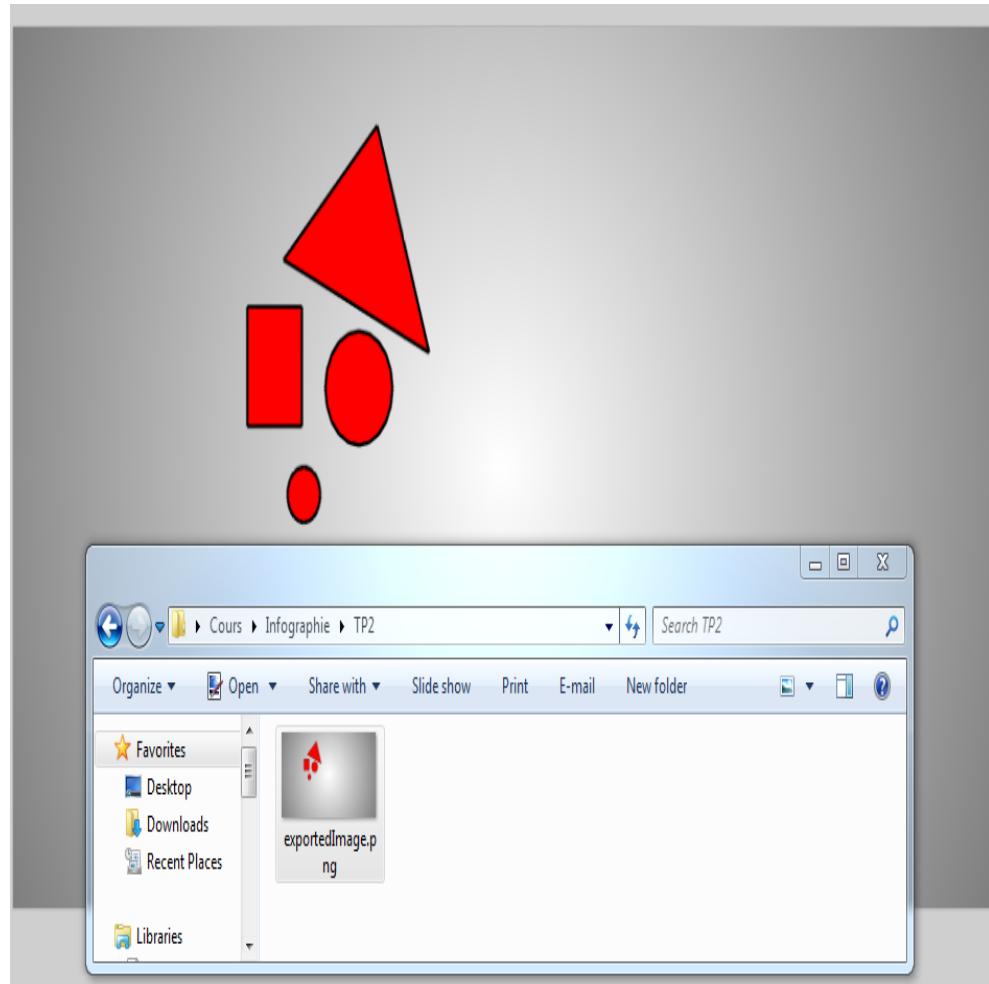


FIGURE 5.5 – Une image exportée à partir de l’application Paint3D+.

```

else if (btnName == "Export") {
    isTakingScreenshot = true;
    draw();
    isTakingScreenshot = false;

ofFileDialogResult file = ofSystemSaveDialog("Save", "Export");

if (file.getPath() != "") {
    renderer2d->imageExport(file.getPath(), "png");
}
}

```

FIGURE 5.6 – Code important de la fonctionnalité Export.

## 5.2 Dessin vectoriel

### 5.2.1 Curseur dynamique

L’application permet un affichage dynamique du curseur de façon à ce que l’utilisateur obtienne une rétroaction visuelle lors de ses actions. Le curseur est représenté différemment que ce soit pour chacune des formes 2D que l’utilisateur ajoute ou pour l’ensemble des opérations de transformation. De plus, le curseur redévient normal lorsque situé au dessus des panneaux de configuration de l’interface graphique. Pour ce faire, la fonction dessinant le curseur vérifie quel mode de dessin est sélectionné ou quelle action est effectuée.

Les figures ci-dessous démontrent la présence de la fonctionnalité.

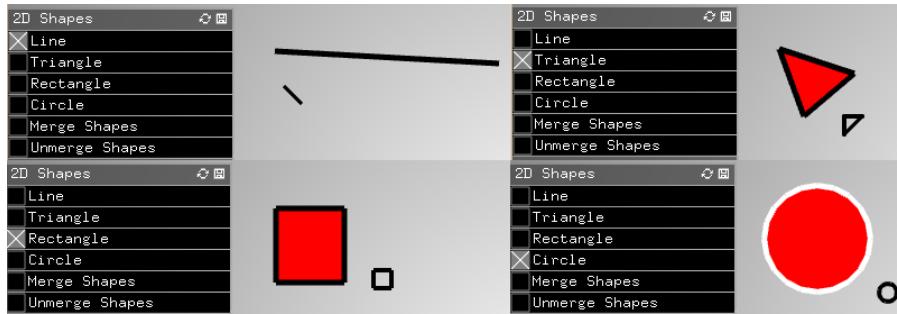


FIGURE 5.7 – Curseur dynamique - Ligne, triangle, rectangle et cercle

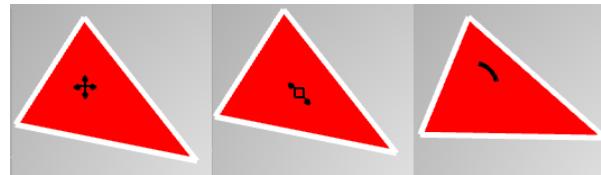


FIGURE 5.8 – Curseur dynamique - Translation, redimensionnement et rotation

```
void ofApp::drawCursor() {
    int mouseX = ofGetMouseX();
    int mouseY = ofGetMouseY();

    if (mouseIsOverPanel()) {
        ofShowCursor();
        return;
    }

    ofHideCursor();
    ofSetColor(0);
    ofSetLineWidth(4);
    ofNoFill();
    switch (m_state) {
        case AppState::ACTION_SELECT:
        case AppState::ACTION_GROUPSELECT:
            ofShowCursor();
            break;
        case AppState::BUILD_RECTANGLE:
            ofDrawRectangle(mouseX, mouseY, 16, 16);
            break;
        case AppState::BUILD_TRIANGLE:
            ofDrawTriangle(mouseX, mouseY, mouseX + 16, mouseY, mouseX + 16);
            ofFill();
            ofDrawCircle(mouseX, mouseY, 2);
            ofDrawCircle(mouseX + 16, mouseY, 2);
            ofDrawCircle(mouseX, mouseY + 16, 2);
            break;
        case AppState::BUILD_CIRCLE:
            ofCircle(mouseX + 6, mouseY + 6, 8);
            break;
        case AppState::BUILD_LINE:
            ofDrawLine(mouseX, mouseY, mouseX + 16, mouseY + 16);
            break;
        case AppState::ACTION_TRANSLATE:
            ofSetLineWidth(2);
            ofDrawArrow(ofVec3f(mouseX + 8, mouseY + 8), ofVec3f(mouseX, mouseY + 8), 2);
            break;
    }
}
```

FIGURE 5.9 – Exemple de code pour la fonctionnalité

### 5.2.2 Formes vectorielles

L'application Paint3D+ permet de créer des quatre instances de formes, soit des lignes, des triangles, des rectangles ou bien des cercles. Les paramètres sont établis par l'utilisateur et envoyés à l'application qui appelle au final les méthodes de base de OpenFrameworks. De plus, il est possible de créer un ensemble de plusieurs primitives vectorielles en sélectionnant les primitives désirées et en appuyant sur le bouton Merge Shapes. La manière de procéder consiste à regrouper les différentes formes sélectionnées dans un vecteur et créer un nouvel objet.

La figure ci-dessous démontre le code afin de dessiner un objet 2D regroupant plusieurs primitives.

```
void Renderer2D::drawCollection(app::Obj2DCollection *p_coll) {
    for (Obj2D* o : p_coll->getObjVector()) {
        switch (o->getType()) {
            case EnumVectorDrawMode::VECTOR_PRIMITIVE_CIRCLE: {
                app::Circle* c = dynamic_cast<app::Circle*>(o);
                drawCircle(c);
            }
            break;
            case EnumVectorDrawMode::VECTOR_PRIMITIVE_RECTANGLE: {
                app::Rectangle* r = dynamic_cast<app::Rectangle*>(o);
                drawRectangle(r);
            }
            break;
            case EnumVectorDrawMode::VECTOR_PRIMITIVE_TRIANGLE: {
                app::Triangle* t = dynamic_cast<app::Triangle*>(o);
                drawTriangle(t);
            }
            break;
            case EnumVectorDrawMode::VECTOR_PRIMITIVE_LINE: {
                app::Line2D* l = dynamic_cast<app::Line2D*>(o);
                drawLine(l);
            }
            break;
            case EnumVectorDrawMode::VECTOR_PRIMITIVE_IMAGE: {
                app::Image2D* i = dynamic_cast<app::Image2D*>(o);
                drawImage(i);
            }
            break;
            case EnumVectorDrawMode::VECTOR_PRIMITIVE_COLLECTION: {
                app::Obj2DCollection* i = dynamic_cast<app::Obj2DCollection*>(o);
                drawCollection(i);
            }
            break;
        }
    }
}
```

FIGURE 5.10 – Exemple de code pour le dessin de collection d'objets

### 5.2.3 Interface

Au niveau de l'interface graphique, l'application permet d'offrir des contrôles interactifs en plus de donner de la rétroaction à l'utilisateur. En effet, comme il est possible de voir sur les différentes figures présentes dans ce rapport, des panneaux de configuration des différentes instances de formes permettent d'ajuster chacun des paramètres de celles-ci. Lors de la fonction setup, des instances (boutons, panneaux, etc.) dérivant de la classe OfxGui sont ajoutés à la fenêtre de l'application.

## 5.3 Transformation

### 5.3.1 Structure de la scène

Toutes les entités géométriques présentes dans une scène sont organisées dans une ou des structures de données qui permettent de gérer les transformations et le rendu graphique de chaque élément.

Dans notre application, la structuration des entités géométriques est faite sous forme de vecteurs (un vecteur pour le 2D et un pour le 3D). La recherche d'une entité particulière est donc faite en parcourant les vecteurs au complet.

```
class ofApp : public ofBaseApp{
public:
    std::vector<Coord> m_buffer;
    std::vector<Obj2D*> m_obj2DVector;
    std::vector<Coord3D> m_buffer3D;
    std::vector<Obj3D*> m_obj3DVector;
    AppState m_state;
    AppMode m_mode;
```

FIGURE 5.11 – Code important de la fonctionnalité Structure de scène.

Pour le mode 3D de l'application, il est possible de sélectionner chaque Primitive 3D en appuyant sur le bouton "Select next 3d shape". Une fois sélectionnée, il est possible de sa

couleur de remplissage, sa position en x, y, z, son orientation en x, y, z, de la supprimer et de la transformer en nuage de points.

Cela est possible grâce à une structure `vector<Obj3D*>` qui contient toutes les classes dérivées de `Obj3D` comme le cube ou la sphère. Voici un exemple de code le montrant :

```
27 class Renderer2D;
28 class Obj3D;
29 class Renderer3D;
30 class RendererModel;
31
32 class ofApp : public ofBaseApp{
33     public:
34
35     std::vector<Coord> m_buffer;
36     std::vector<Obj2D*> m_obj2DVector;
37     std::vector<Coord3D> m_buffer3D;
38     std::vector<Obj3D*> m_obj3DVector;
39     AppState m_state;
40     AppMode m_mode;
41     int m_clickRadius;
42     bool isTakingScreenshot;
43     bool isClearingButtonsShapes, isClearingButtonsModes;
44     bool m_firstTimeSelection;
45     int m_selectionIndex;
46
47     ofApp();
48     ~ofApp();
49     void exit();
50
```

FIGURE 5.12 – Structure de données stockant les entités 3D.

```

785 ▼ void ofApp::buildTriangle() {
786     m_obj2DVector.push_back(new app::Triangle(m_buffer, 0, renderer2d->strokeWidth.get(), renderer2d->colorStroke.get(), renderer2d->fillColor.get()));
787 }
788
789 ▼ void ofApp::buildCircle() {
790     double radius = calculateDistance(m_buffer[0], m_buffer[1]);
791     m_obj2DVector.push_back(new app::Circle(m_buffer[0], radius, 0, renderer2d->strokeWidth.get(), renderer2d->colorStroke.get(), renderer2d->fillColor.get()));
792 }
793
794 ▼ void ofApp::buildLine() {
795     m_obj2DVector.push_back(new app::Line2D(m_buffer, 0, renderer2d->strokeWidth.get(), renderer2d->colorStroke.get(), renderer2d->fillColor.get()));
796 }
797
798 ▼ void ofApp::buildCube() {
799     m_obj3DVector.push_back(new app::Cube3D(m_buffer3D, renderer2d->strokeWidth.get(), renderer2d->colorStroke.get(), renderer2d->fillColor.get()));
800 }
801
802 ▼ void ofApp::buildSphere() {
803     m_obj3DVector.push_back(new app::Sphere3D(m_buffer3D, renderer2d->strokeWidth.get(), renderer2d->colorStroke.get(), renderer2d->fillColor.get()));
804 }
805
806 ▼ double ofApp::calculateDistance(Coord p_coord1, Coord p_coord2) {
807     double x2 = pow((p_coord1.getX() - p_coord2.getX()), 2);
808     ...
809 }

```

FIGURE 5.13 – Méthode montrant comment on stock une nouvelle entité 3D dans vector<Obj3D\*>.

### 5.3.2 Transformation interactive

Il est possible de modifier interactivement la translation, la rotation et le redimensionnement de tous les objets présents dans la scène.

Pour les transformations 2D, le code de ceux-ci a été fait au complet. L'utilisation des fonctions prédéfinies de openFrameworks est seulement présente pour les images importées. Dans les autres cas, soit des paramètres de taille sont changés (width, height et radius) ou soit les coordonnées des coins et/ou du point milieu. Les transformations de redimensionnement et de rotation sont faites par rapport au point milieu de la structure (moyenne des coins/points critiques). Pour ce qui est des objets 3D, la translation est faite par position de openFrameworks et l'orientation par quaternion, une fonctionnalité de openFrameworks.

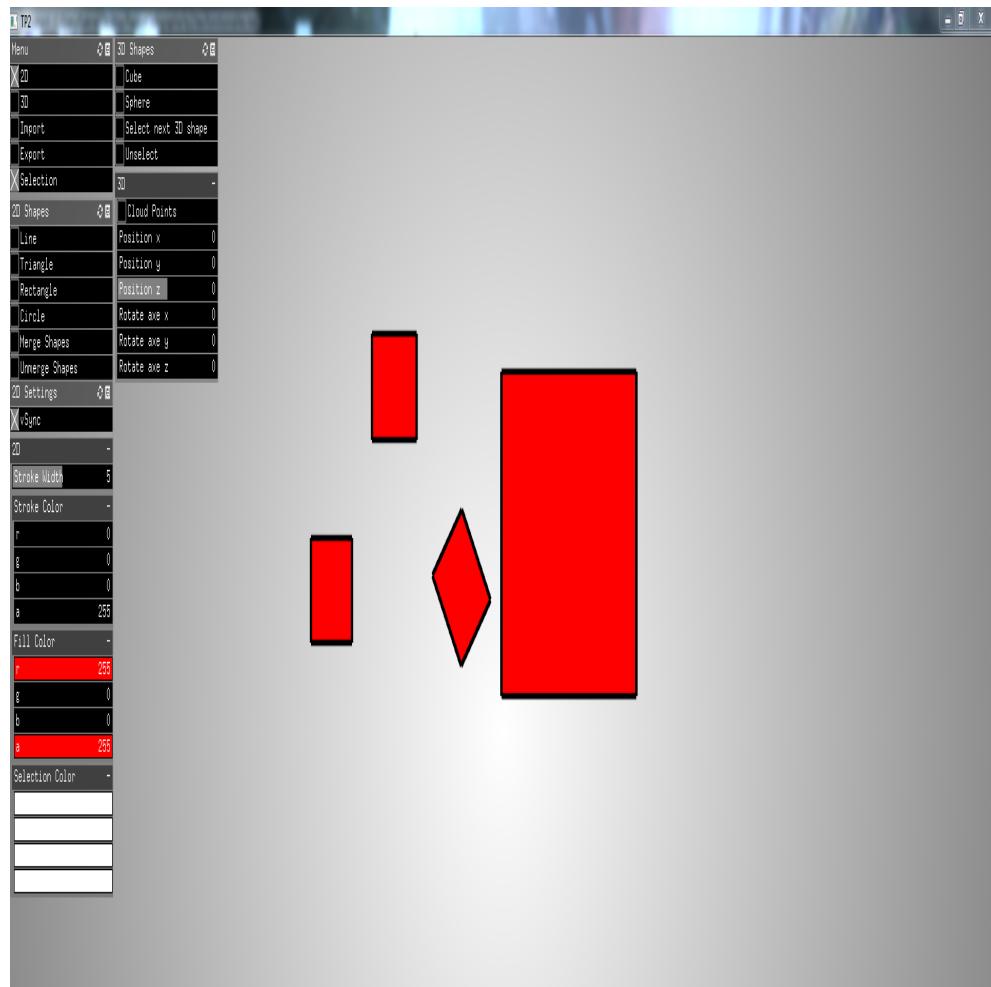


FIGURE 5.14 – Translation, rotation et redimensionnement d'un rectangle.

```
void ofApp::translateSelection(double p_x, double p_y) {
    for (Obj2D* o : m_obj2DVector) {
        if (o->isSelected()) {
            o->translate(p_x, p_y);
        }
    }
}

void ofApp::rotateSelection() {
    cout << "rotate" << endl;
    for (Obj2D* o : m_obj2DVector) {
        if (o->isSelected()) {
            o->rotate(m_buffer[0], m_buffer[1]);
        }
    }
}

void ofApp::resizeSelection() {
    for (Obj2D* o : m_obj2DVector) {
        if (o->isSelected()) {
            o->resize(calculateDistance(m_buffer[1], o->getRotationCenter()) / calculateDistance(m_buffer[0], o->getRotationCenter())));
        }
    }
}
```

FIGURE 5.15 – Code important des fonctionnalités de transformation.

```
void Obj2D::resize(double p_percent) {
    resize(getRotationCenter(), p_percent);
}

void Obj2D::resize(Coord p_coord, double p_percent) {
    std::vector<Coord> newCoordVector;
    double newDistanceX = 0;
    double newDistanceY = 0;
    for (Coord c : m_coordVector) {
        newDistanceX = p_percent*(c.getX() - p_coord.getX());
        newDistanceY = p_percent*(c.getY() - p_coord.getY());
        newCoordVector.push_back(Coord(p_coord.getX() + newDistanceX, p_coord.getY() + newDistanceY));
    }
    m_coordVector = newCoordVector;
}
```

FIGURE 5.16 – Code important de la fonctionnalité de redimensionnement.

```
█Coord Obj2D::getRotationCenter() {
    double xTot = 0;
    double yTot = 0;
    █for (Coord c : m_coordVector) {
        xTot += c.getX();
        yTot += c.getY();
    }
    return Coord(xTot / m_coordVector.size(), yTot / m_coordVector.size());
}

█void Obj2D::translate(double p_x, double p_y) {
    std::vector<Coord> newCoordVector;
    █for (Coord c : m_coordVector) {
        c.addToCoord(p_x, p_y);
        newCoordVector.push_back(c);
    }
    m_coordVector = newCoordVector;
}
```

FIGURE 5.17 – Code important de la fonctionnalité de translation.

```

void Obj2D::rotate(Coord p_first, Coord p_sec) {
    Coord rotCenter = getRotationCenter();
    double vector1[] { p_first.getX() - rotCenter.getX(), p_first.getY() - rotCenter.getY() };
    double vector2[] { p_sec.getX() - rotCenter.getX(), p_sec.getY() - rotCenter.getY() };
    double angle = calculateAngle(vector1, vector2);
    angle = angle * getAngleSign(p_first, p_sec, rotCenter);
    rotate(rotCenter, angle);
}

void Obj2D::rotate(double p_degree) {
}

void Obj2D::rotate(Coord p_coord, double p_degree) {
    std::vector<Coord> newCoordVector;
    double x = 0;
    double y = 0;
    for (Coord c : m_coordVector) {
        x = c.getX() - p_coord.getX();
        y = c.getY() - p_coord.getY();
        c.setX((x*cos(p_degree*PI / 180) - y*sin(p_degree*PI / 180)) + p_coord.getX());
        c.setY((x*sin(p_degree*PI / 180) + y*cos(p_degree*PI / 180)) + p_coord.getY());
        newCoordVector.push_back(c);
    }
    m_angle += p_degree;
    m_coordVector = newCoordVector;
}

```

FIGURE 5.18 – Code important de la fonctionnalité de rotation.

En mode 3D, il est possible de changer la position x, y, z, l'orientation en x, y, z et le redimensionnement grâce aux paramètres glissant de la fenêtres "3D Settings".

Voici les images le montrant :

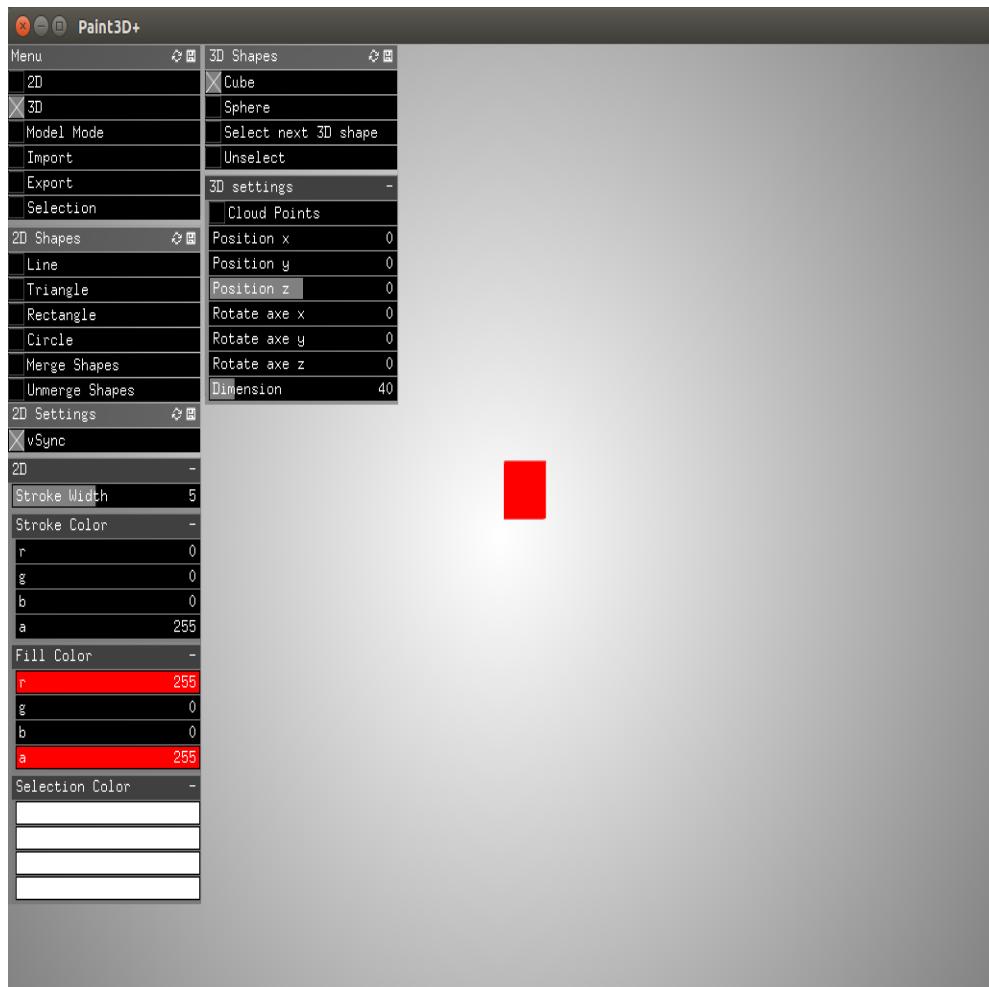


FIGURE 5.19 – Image de référence.

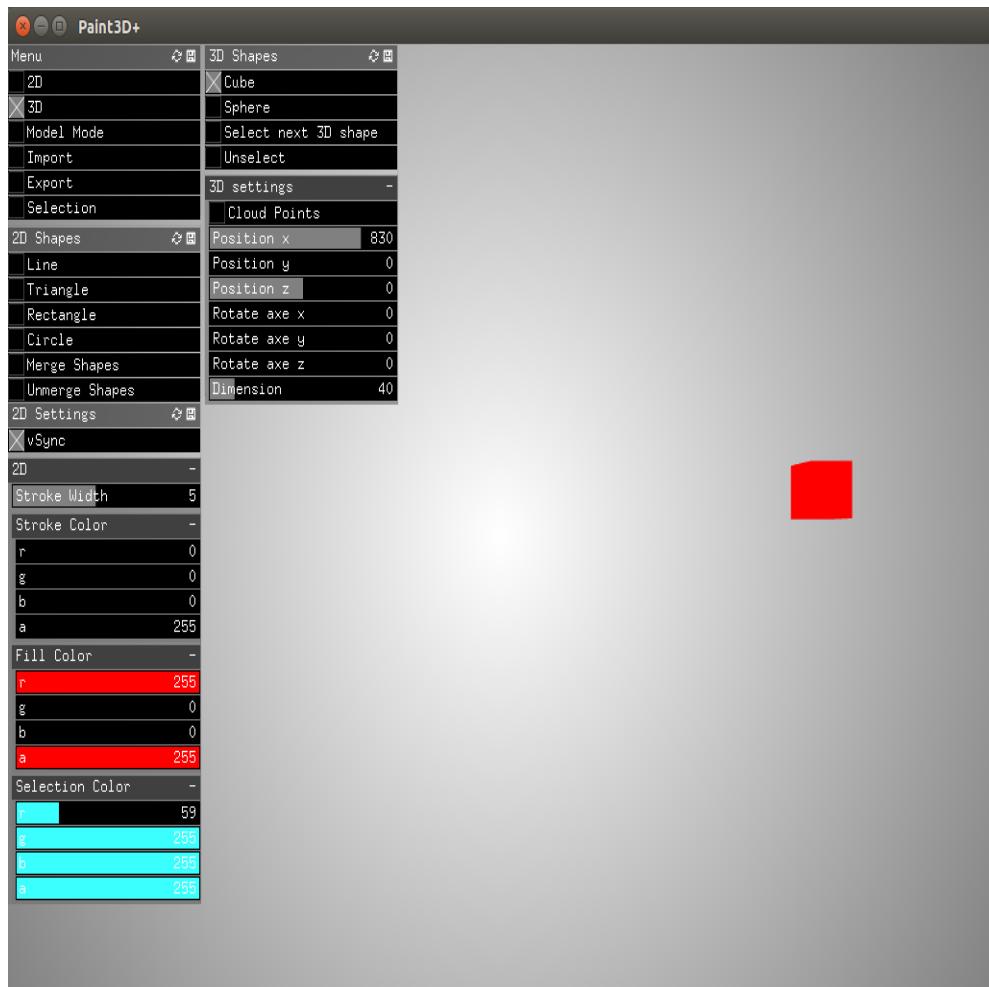


FIGURE 5.20 – Changement de Position.

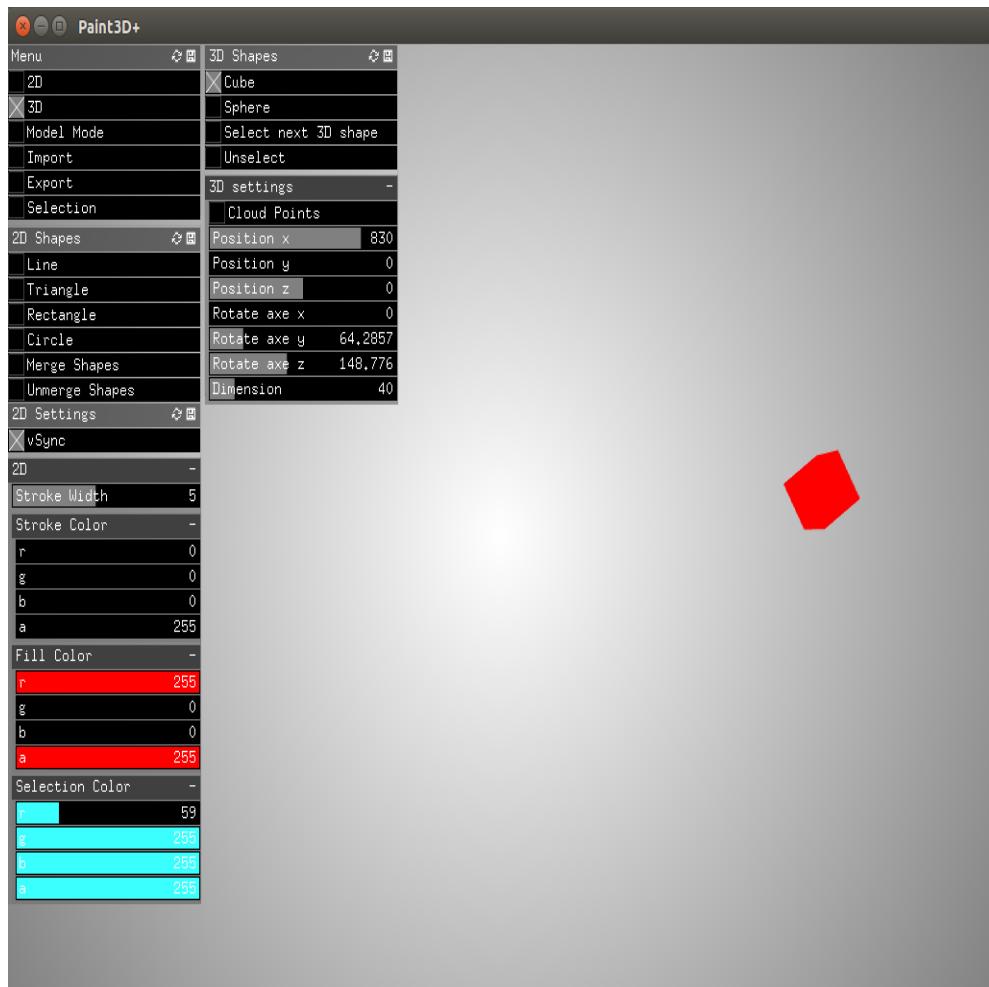


FIGURE 5.21 – Changement d’orientation.

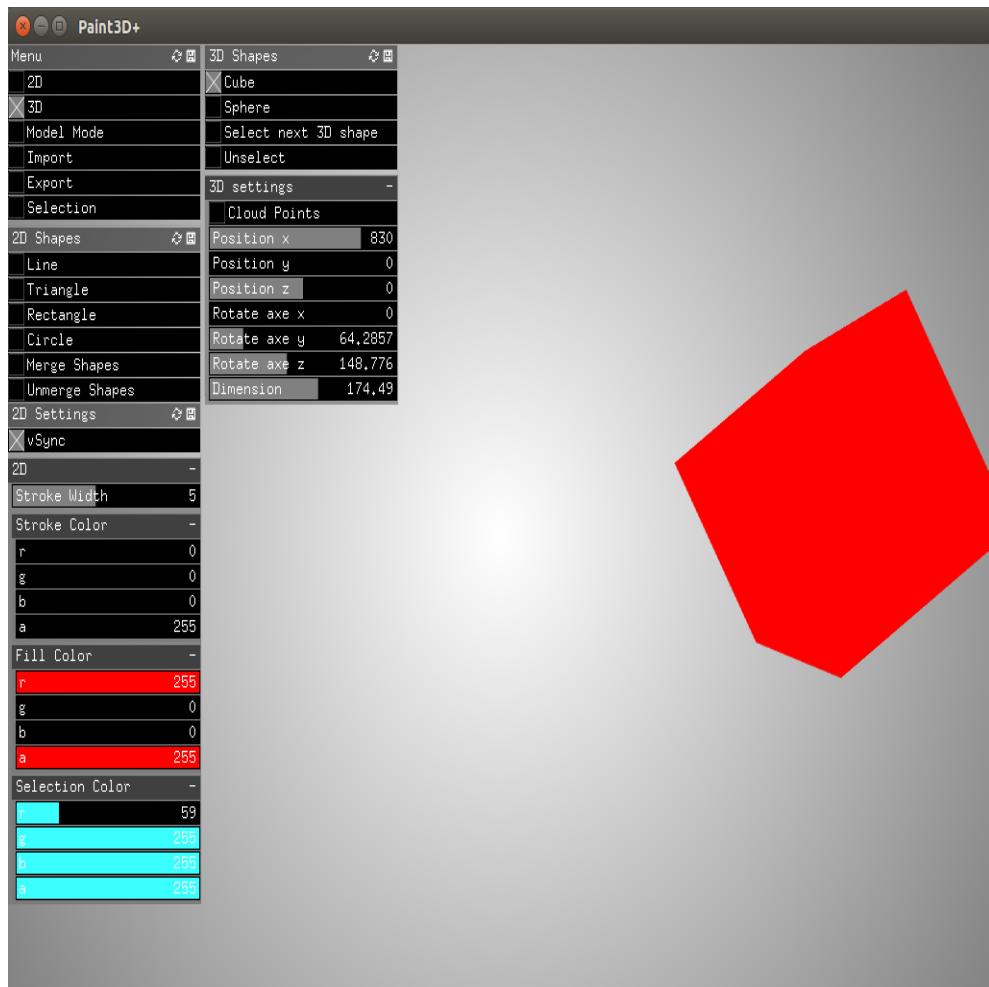
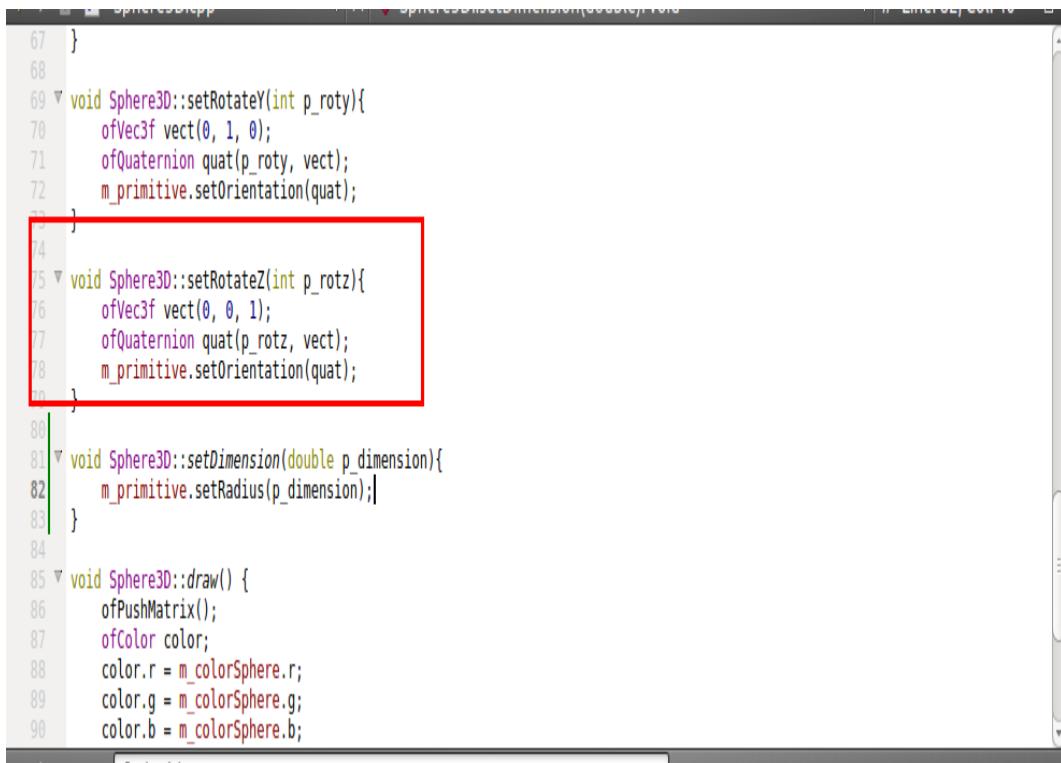


FIGURE 5.22 – Redimensionnement.

De plus, pour méthodes s'occupant de faire la rotation aux primitives 3D, les quaternions sont utilisés pour leur côté pratique et éviter tout blocage de cardan :



```

67 }
68
69 void Sphere3D::setRotateY(int p_rotY){
70     ofVec3f vect(0, 1, 0);
71     ofQuaternion quat(p_rotY, vect);
72     m_primitive.setOrientation(quat);
73 }
74
75 void Sphere3D::setRotateZ(int p_rotZ){
76     ofVec3f vect(0, 0, 1);
77     ofQuaternion quat(p_rotZ, vect);
78     m_primitive.setOrientation(quat);
79 }
80
81 void Sphere3D::setDimension(double p_dimension){
82     m_primitive.setRadius(p_dimension);
83 }
84
85 void Sphere3D::draw() {
86     ofPushMatrix();
87     ofColor color;
88     color.r = m_colorSphere.r;
89     color.g = m_colorSphere.g;
90     color.b = m_colorSphere.b;

```

FIGURE 5.23 – Exemple de code contenant des quaternions.

### 5.3.3 Sélection multiple

Il est possible de sélectionner une ou des instances d'entités visuelles présentes dans la scène et de modifier les attributs qu'elles ont en commun même si elles sont de différents types. Ces entités peuvent même être rassemblées pour créer un nouvel object appelé Obj2DCollection. Cette collection emmagasine les structures selectionnées dans un nouveau vecteur.

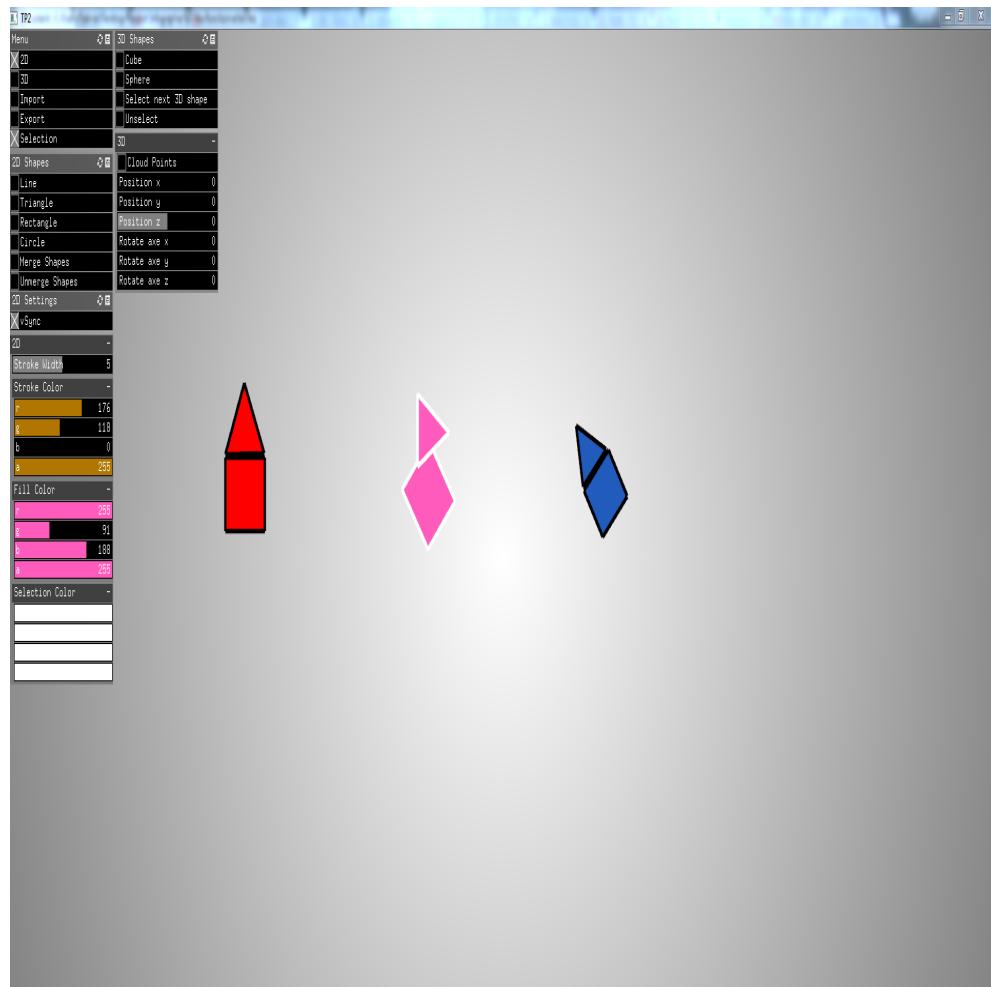


FIGURE 5.24 – Objet original, modifications à l'aide d'une sélection multiple et modifications après un collage en Obj2DCollection.

```
bool Rectangle::isContainedInRect(Coord p_topLeft, double p_width, double p_height) {
    double distanceX = 0;
    double distanceY = 0;
    for (Coord c : m_coordVector) {
        distanceX = c.getX() - p_topLeft.getX();
        distanceY = c.getY() - p_topLeft.getY();
        if (distanceX <= p_width && distanceY <= p_height && distanceX >= 0 && distanceY >= 0) return true;
    }
    return false;
}
```

FIGURE 5.25 – Code important de la fonctionnalité de sélection multiple.

```
void Renderer2D::bStrokeWidthChanged(int & p_strokeWidth) {
    for (Obj2D* o : m_app->m_obj2DVector) {
        if (o->isSelected()) {
            o->setLineStroke(p_strokeWidth);
        }
    }
}

void Renderer2D::bColorStrokeChanged(ofColor & p_colorStroke) {
    for (Obj2D* o : m_app->m_obj2DVector) {
        if (o->isSelected()) {
            o->setLineColor(p_colorStroke);
        }
    }
}

void Renderer2D::bColorFillChanged(ofColor & p_colorFill) {
    for (Obj2D* o : m_app->m_obj2DVector) {
        if (o->isSelected()) {
            o->setColorFill(p_colorFill);
        }
    }
    for (Obj3D* o : m_app->m_obj3DVector) {
        if (o->isSelected()) {
            o->setColorFill(p_colorFill);
        }
    }
}
```

FIGURE 5.26 – Code important de la fonctionnalité de sélection multiple.

```

void Renderer2D::bColorSelectedChanged(ofColor & p_colorSelected) {
    for (Obj2D* o : m_app->m_obj2DVector) {
        if (o->isSelected()) {
            o->setLineColorSelected(p_colorSelected);
        }
    }

    for (Obj3D* o : m_app->m_obj3DVector) {
        if (o->isSelected()) {
            o->setLineColorSelected(p_colorSelected);
        }
    }
}

```

FIGURE 5.27 – Code important de la fonctionnalité de selection multiple.

## 5.4 Géométrie

### 5.4.1 Particules

Pour respecter ce critère fonctionnel, l'application doit rendre un nuage de points en une seule commande d'affichage.

Dans l'application Paint3D+, il y a un bouton nommé "Cloud Points" qui permet de transformer un objet 3D sélectionné en un nuage de points où chaque point est un sommet d'un des faces qui compose l'objet en 3D.

Donc, pour pouvoir utiliser cette option, il faut créer un objet 3D avec le bouton "Cube"

ou "Sphere" en mode 3D, sélectionner un des objets 3D créé avec le bouton "Select next 3D Shape" et appuyer sur le bouton "Cloud Points" pour rendre l'objet 3D sélectionné en tant que nuage de points.

Les 2 figures suivantes montrent une sphère rendue sans nuage de points et une sphère rendue avec un nuage de points.

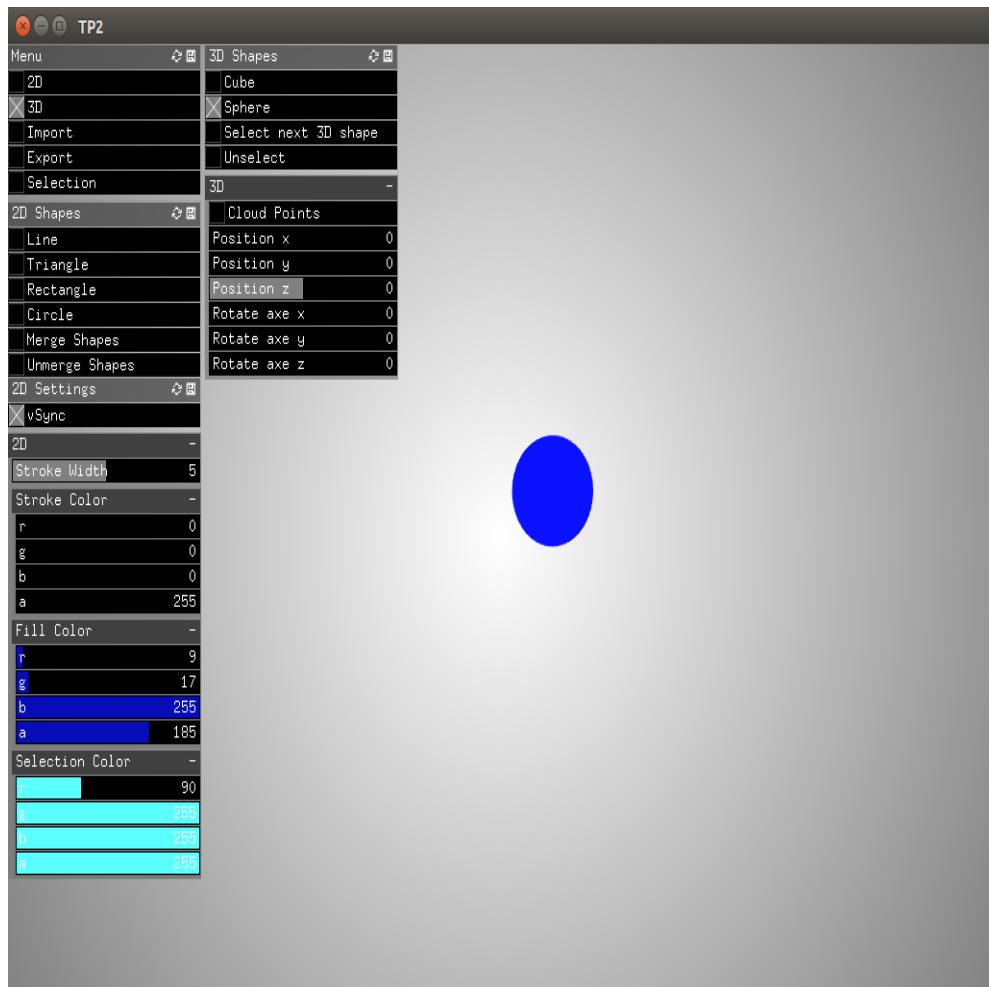


FIGURE 5.28 – Une sphère rendue sans nuage de points.

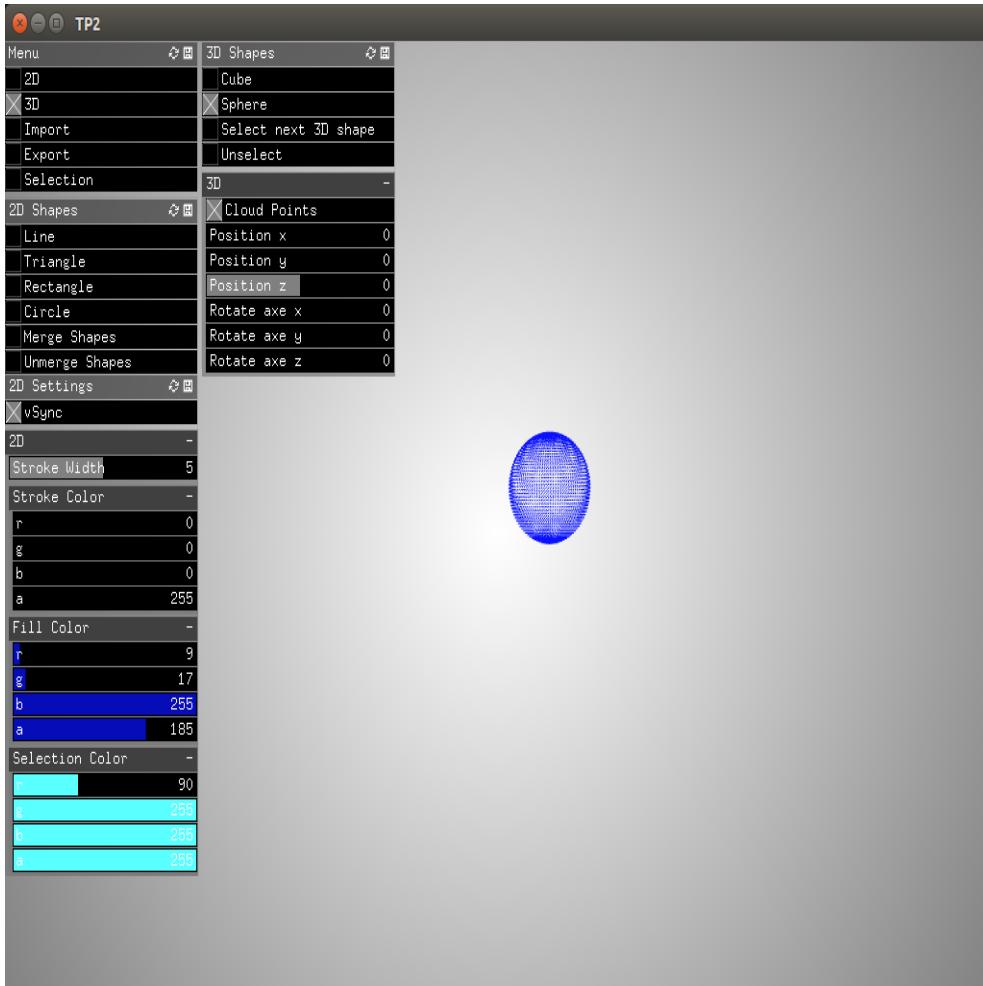


FIGURE 5.29 – Une sphère rendue avec un nuage de points.

#### 5.4.2 Primitives 3D

L’application doit permettre de créer au moins 2 types de primitives géométriques 3D à partir d’un algorithme sans aucune données externes à l’application.

L’application Paint3D+ permet de créer 2 primitives 3D localement sans données externes, soit le cube et la sphère. Ces 2 primitives 3D sont créées par composition à partir de 2 classes d’OpenFrameworks, soit OfSpherePrimitive et OfxBoxPrimitive.

Pour créer ces 2 primitives 3D, il faut d’abord sélectionner le mode 3D dans le menu. Ensuite, il faut cocher soit Cube ou Sphere, selon la primitive 3D à créer et cliquer gauche sur la souris à l’endroit où l’on veut que la primitive 3D apparaisse. Une fois créée, chaque primitive 3D a une profondeur par défaut de 30 unités. Cela permet de mettre en valeur l’ef-

fet de perspective et bien montrer qu'il s'agit d'un objet en 3D. Ensuite, pour modifier une primitive 3D créée, il faut la sélectionner avec l'option "Select next 3D shape" et modifier les paramètres modifiables, soit la position en x,y,z l'orientation en x,y,z ou la couleur de sélection ou de remplissage. De plus, lorsqu'une primitive 3D est sélectionnée, il est possible de l'effacer en appuyant sur la touche "delete".

Il est à noter que la rotation est faite à partir de quaternions dans le code.

Les 2 figures suivantes montrent les 2 primitives 3D que l'application peut créer sans données externes.

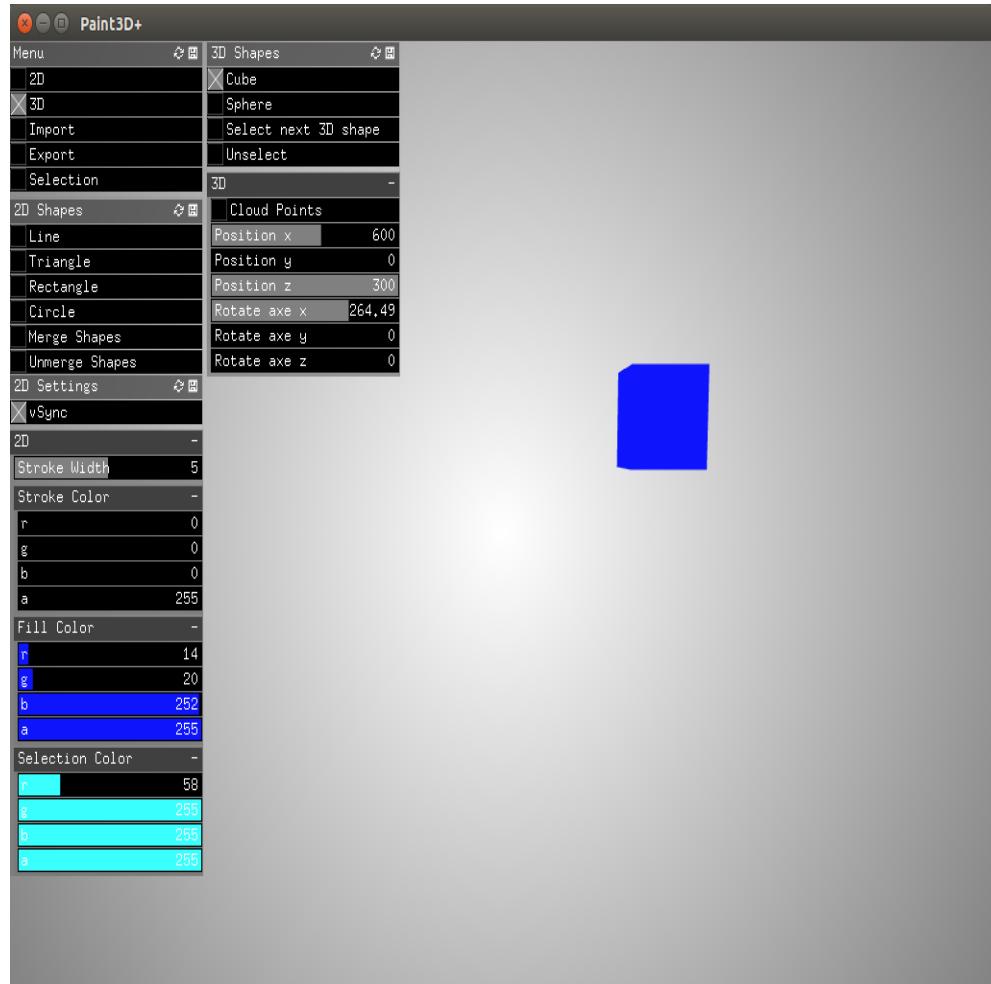


FIGURE 5.30 – Une primitive 3D d'un cube.

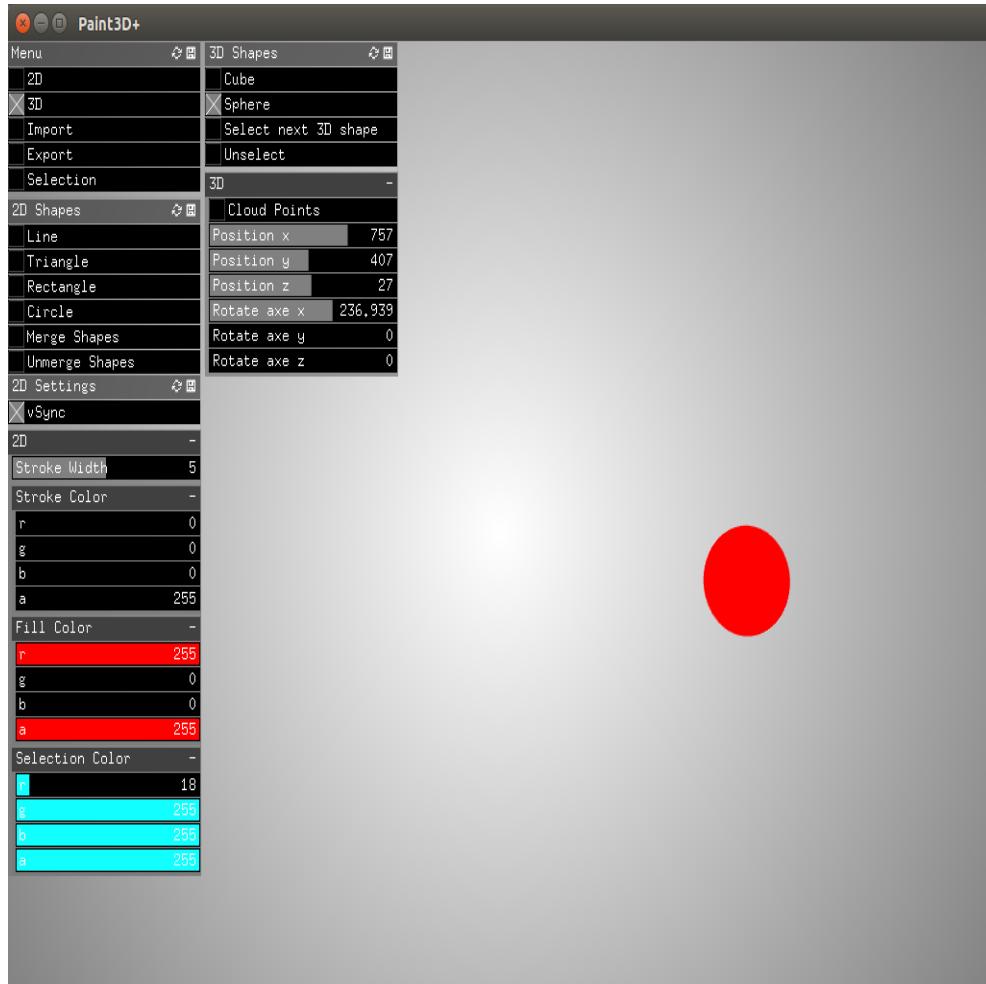


FIGURE 5.31 – Une primitive 3D d'une sphère.

### 5.4.3 Modèle

L'application permet de créer une ou des instances de modèles 3D à partir d'un maillage géométrique importé à partir d'un fichier externe.

Dans le mode "Model", l'application permet d'importer un maillage géométrique et d'afficher une instance de modèle3D. Les maillages peuvent être stockés dans les dossiers /bin/data. Pour cette version de l'application, l'utilisateur ne peut qu'activer ou désactiver la lumière, le matériau et la texture (si elle existe). Ce mode est à développer dans la prochaine version du projet.

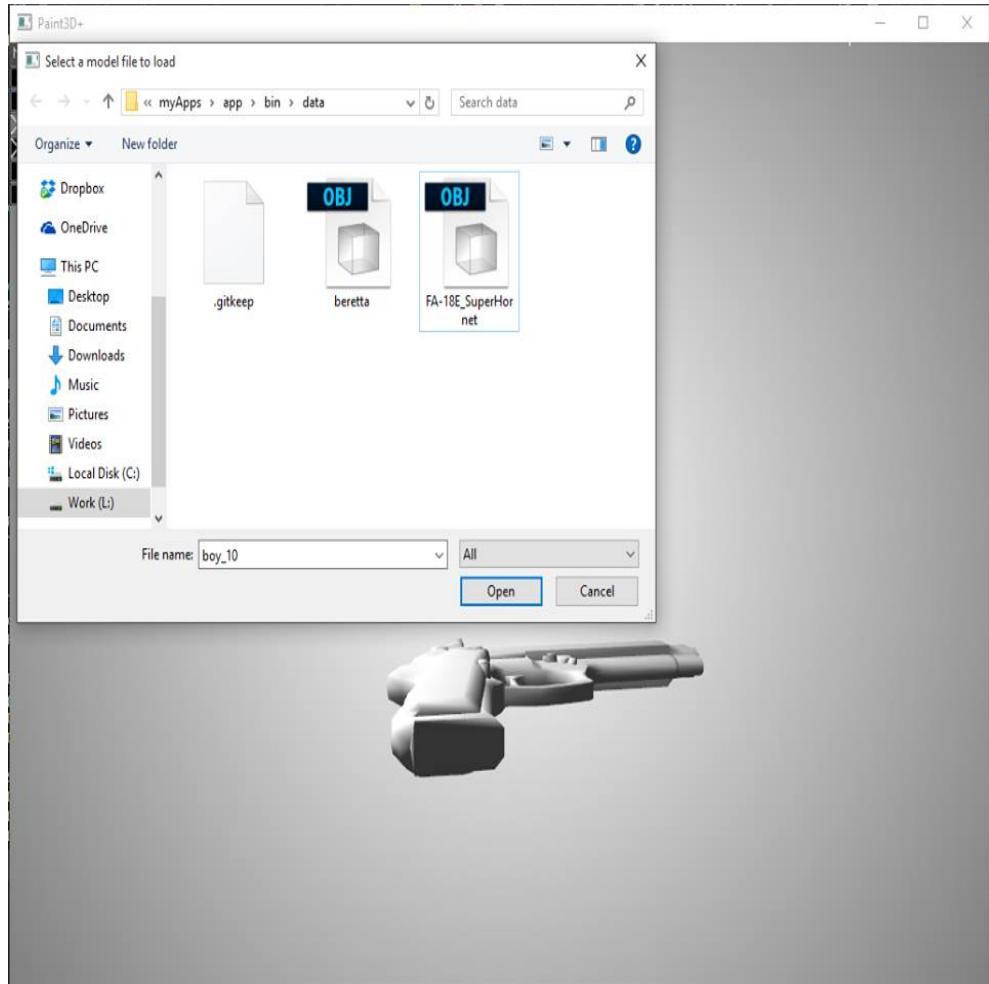


FIGURE 5.32 – Exemple de modèle par un fichier externe contenant un maillage.

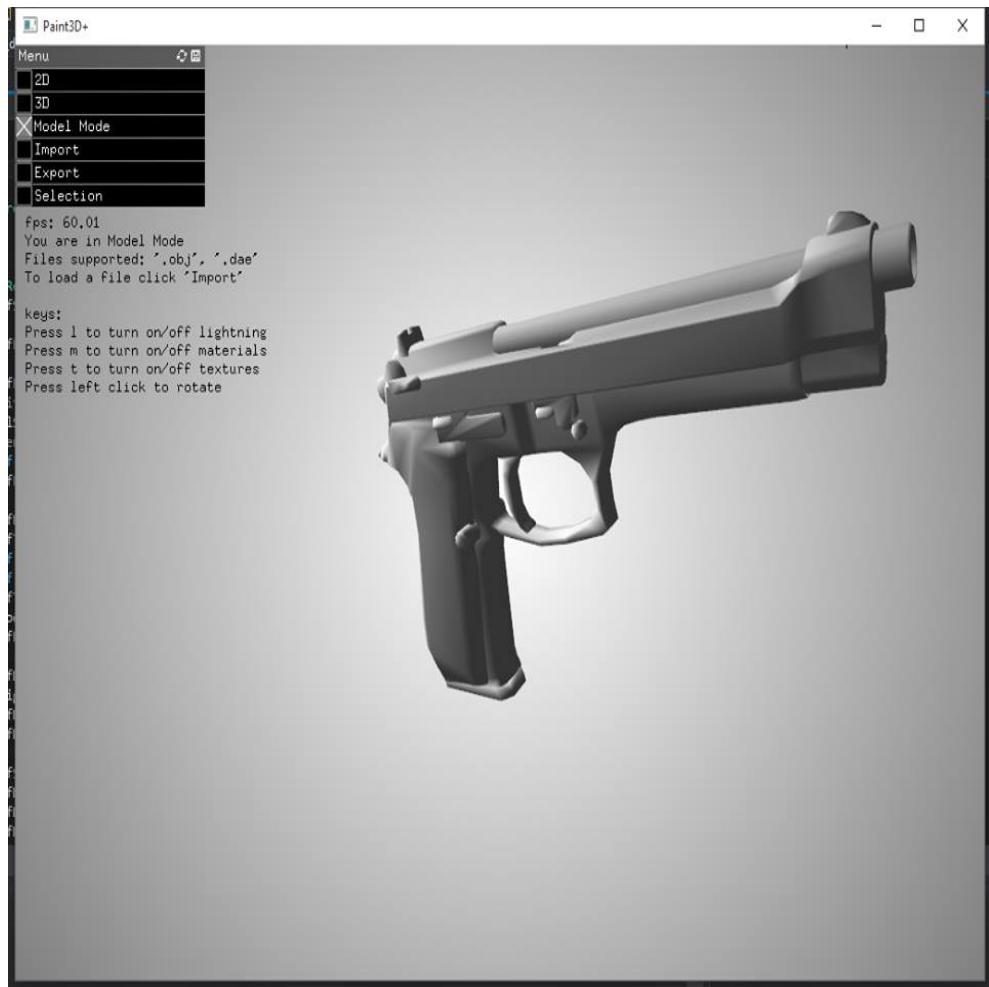


FIGURE 5.33 – Exemple de modèle par un fichier externe contenant un maillage.

## 5.5 Illumination

### 5.5.1 Modèle d'illumination

Le rendu des entités visuelles est fait à partir d'un modèle d'illumination qui supporte au moins une combinaison de lumière dynamique et de matériau.

Dans l'application Paint 3D+, on utilise, pour la partie du projet "Model 3D", la lumière standard (specularlight) et le modèle "shade" de OpenGL (GL\_SMOOTH) pour rendre les fichiers modèles en 3D. On peut enlever la lumière et les matériaux des objets avec les touches correspondantes.

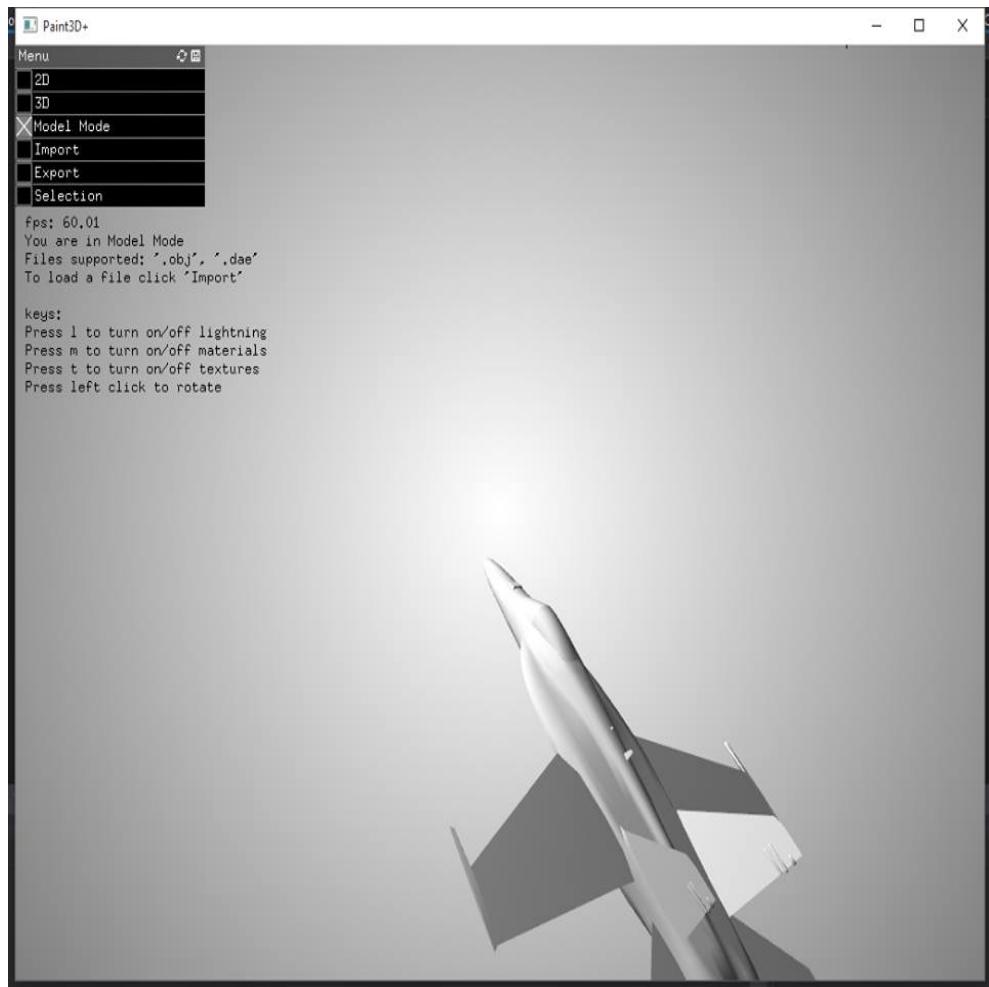


FIGURE 5.34 – Image représentant comme est traitée et réfléchie la lumière avec l'utilisation du shader "shade" de OpenGl (GL\_SMOOTH)

## 5.6 Caméra

Dans l'application Paint3d+, une caméra interractive peut être activée en appuyant sur le bouton "Camera" dans la fenêtre des paramètres 3D. Il faut toutefois être en mode "3D" pour pouvoir activer la caméra.

### 5.6.1 Caméra interactive

En appuyant sur le bouton "Camera" en mode 3D, la fenêtre du programme affiche la scène du point de vue de la caméra. Donc, tous les objets de la scène sont transformés pour être rendus du point de vue de la caméra.

Au départ, la caméra est initialisée à la position "(ofGetWidth()/2, ofGetHeight()/2, 1000)". Cela veut dire que la position par défaut de la caméra est au milieu de l'écran à une distance de 1000 pixels dans l'axe des z de l'espace de la scène. Visuellement, en mode Camera 3D, la scène devient comme celle décrit dans la figure 5.35.

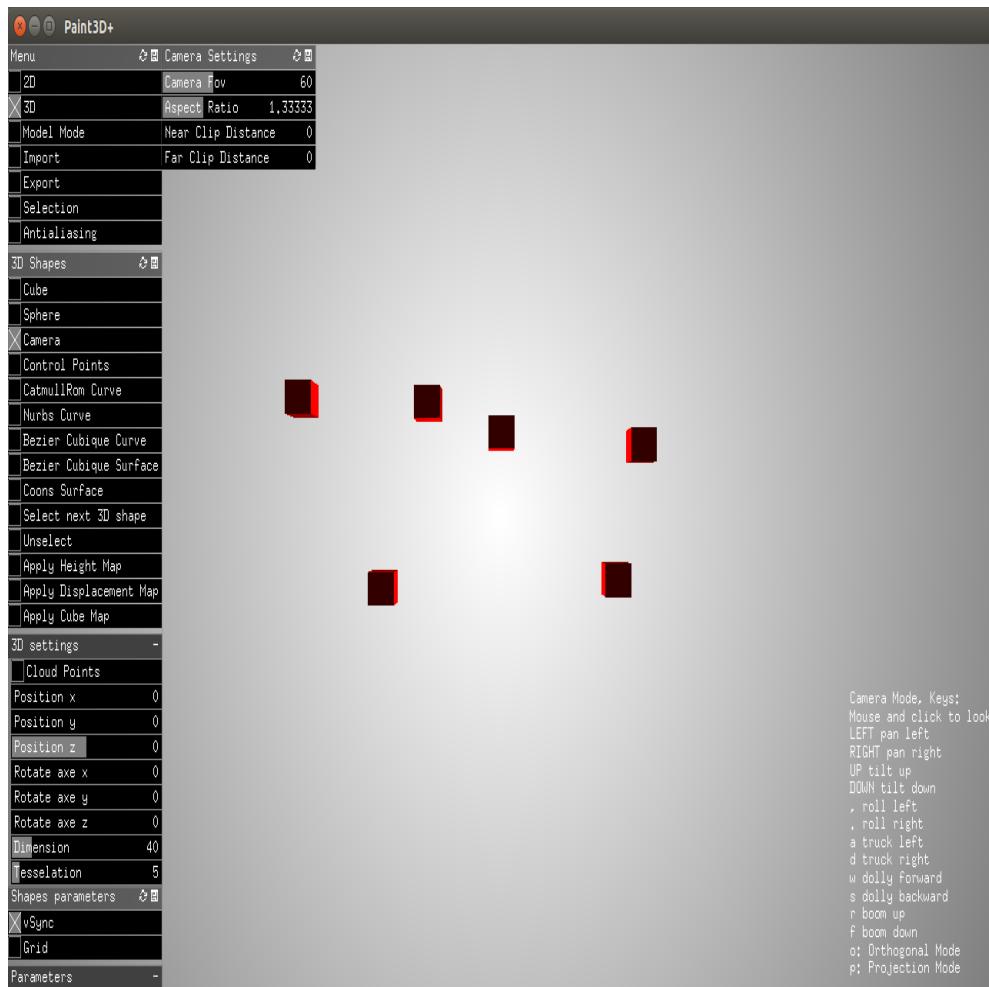


FIGURE 5.35 – Rendu du point de vue la caméra au moment où elle est initialisée.

Une fois en mode "Camera", il est possible d'ajuster l'orientation et la position de la camera avec les touches du clavier décrits dans la figure 5.36. Grossso modo, ce sont les flèches du clavier et les touches "w", "a", "s", "d", "r", "f", ";" et ":" qui permettent respectivement de se déplacer vers l'avant, la gauche, le bas, la droite, le haut et le bas. Les flèches du clavier, quant à elles, changent l'orientation de la vue da la caméra. Il est également possible d'orienter la vue de la caméra avec la souris et son firstButton. Les figures 5.37 et 5.38 montrent un peu les différentes navigations interractives qu'il est possible de faire avec la caméra et leurs effets sur le rendu final.

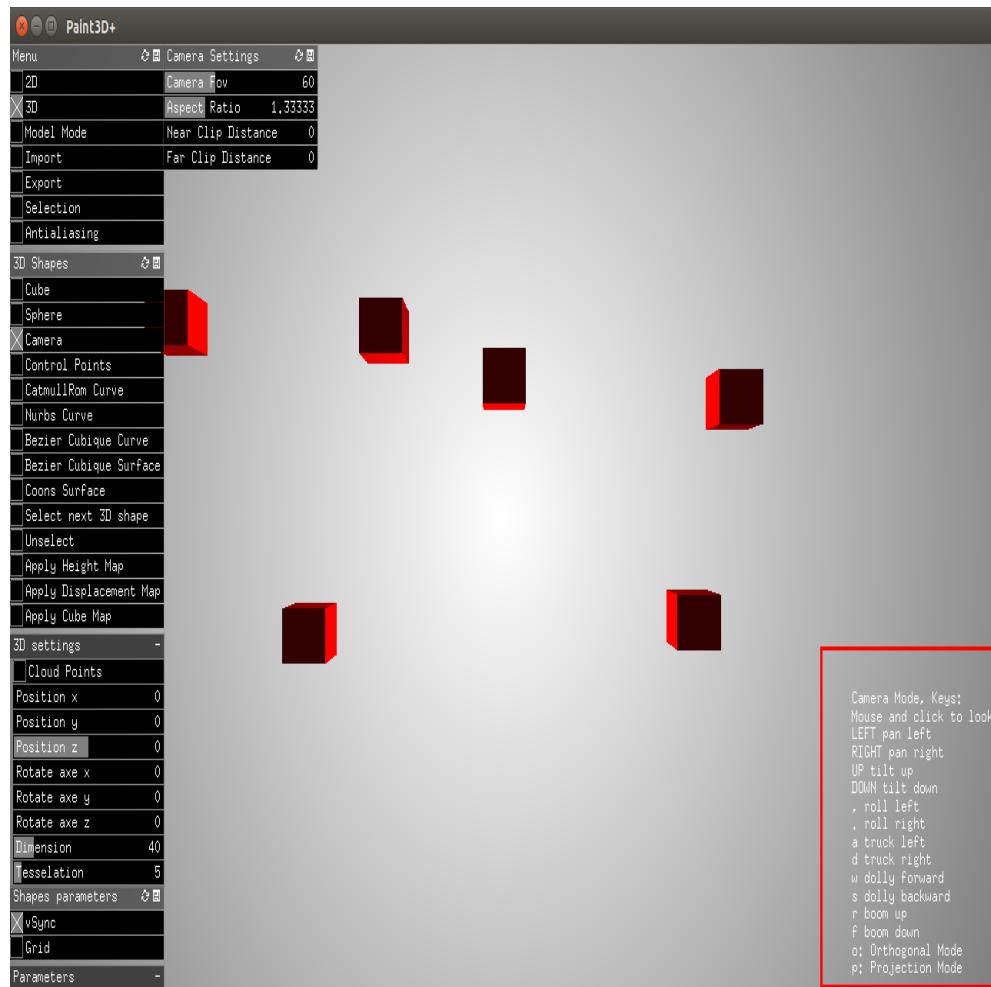


FIGURE 5.36 – Touches à utiliser pour modifier la caméra

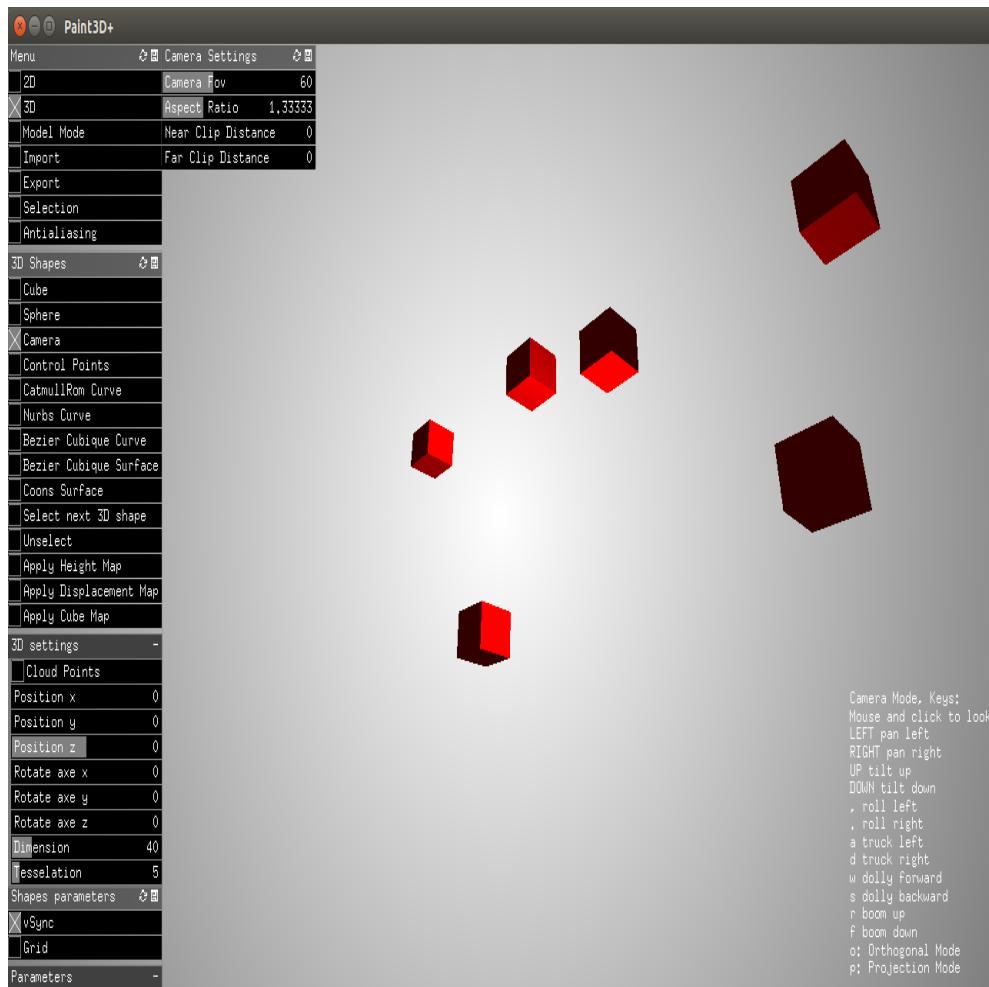


FIGURE 5.37 – Rendu du point de vue de la caméra après s'être déplacé.

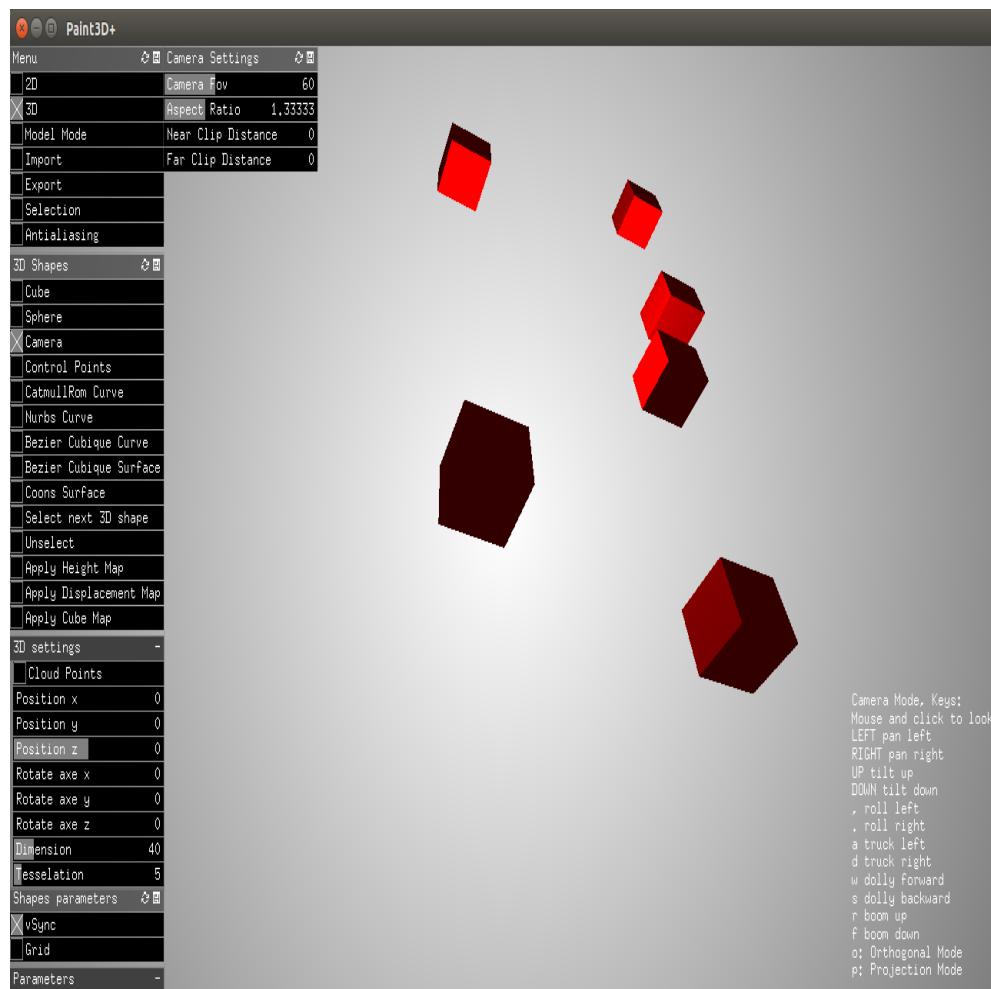


FIGURE 5.38 – Rendu du point de vue de la caméra après avoir changé interactivement l'orientation et la position de la caméra

### 5.6.2 Propriétés de caméra

Avec les paramètres décrits dans la figure 5.39, il est également possible de changer les paramètres de la caméra tels que le type de projection (orthogonal, projectif), l'angle de champ de vision (horizontal et vertical), le ratio d'aspect et la distance du plan de clipping avant et arrière.

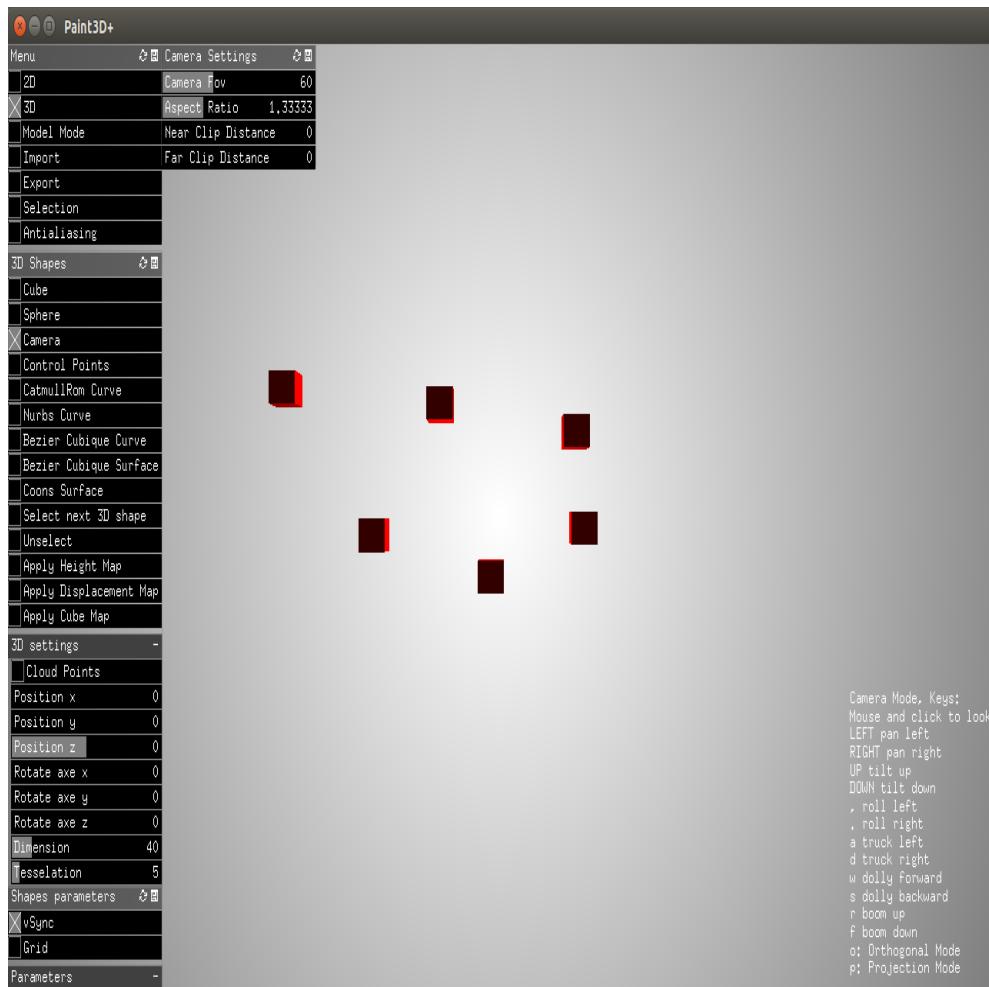


FIGURE 5.39 – Description des différents paramètres modifiables sur la caméra.

Les figures qui suivent montrent l'effet de changer les paramètres décrits précédemment.

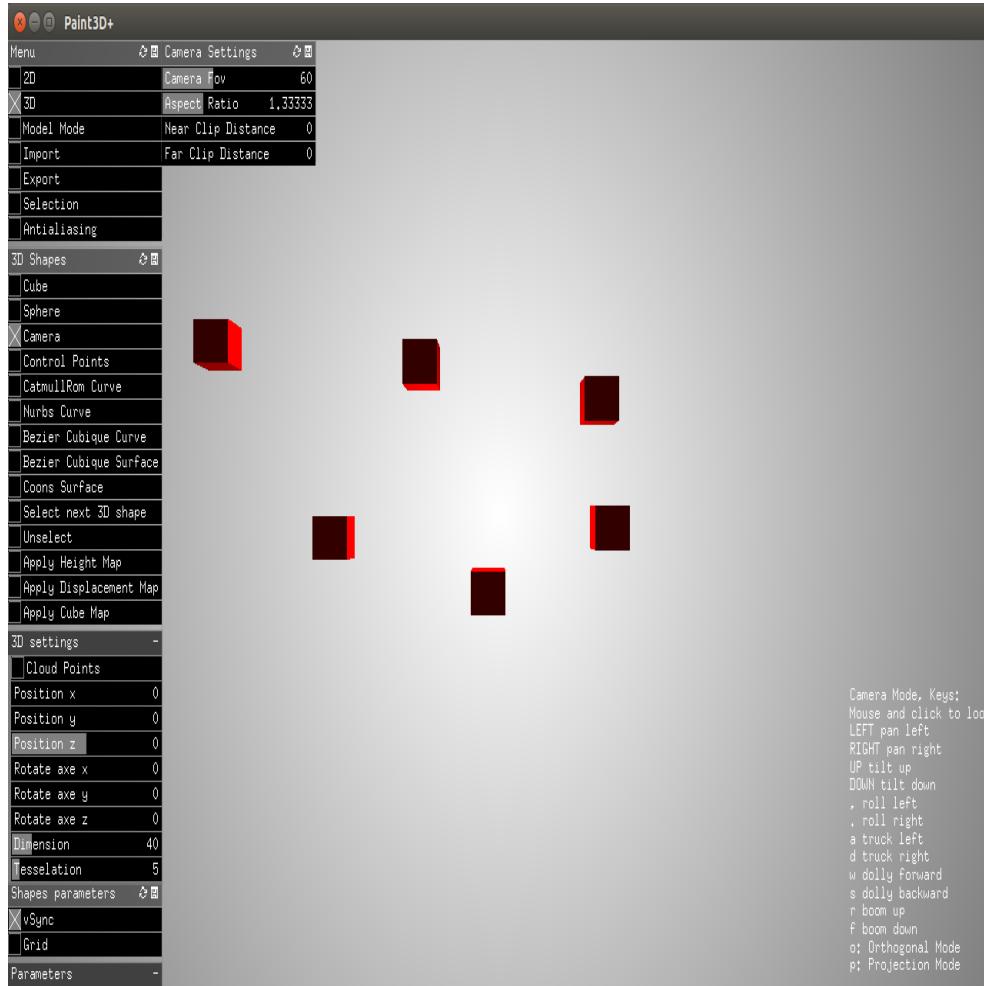


FIGURE 5.40 – Rendu du point de vue de la caméra après initial avec tous les paramètres mis à leur valeur par défaut

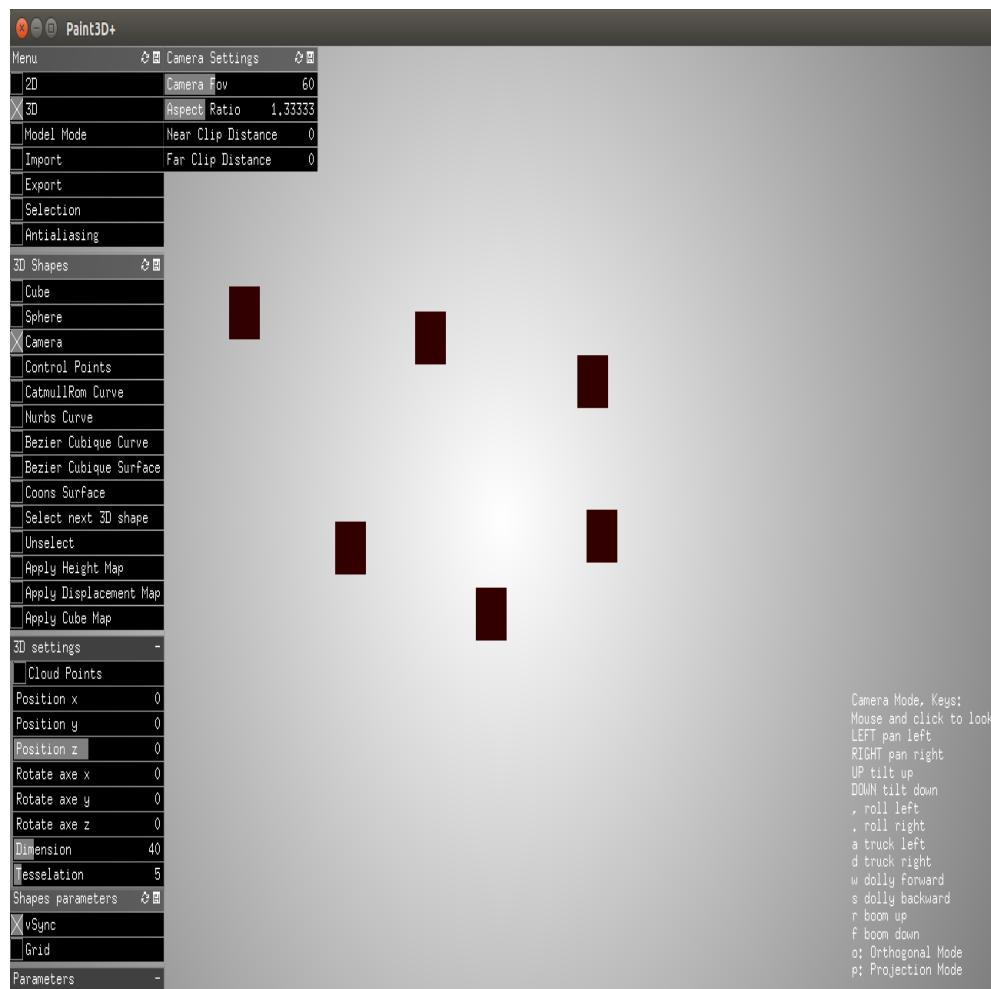


FIGURE 5.41 – Rendu du point de vue de la caméra après avoir mis le type de projection à orthogonal

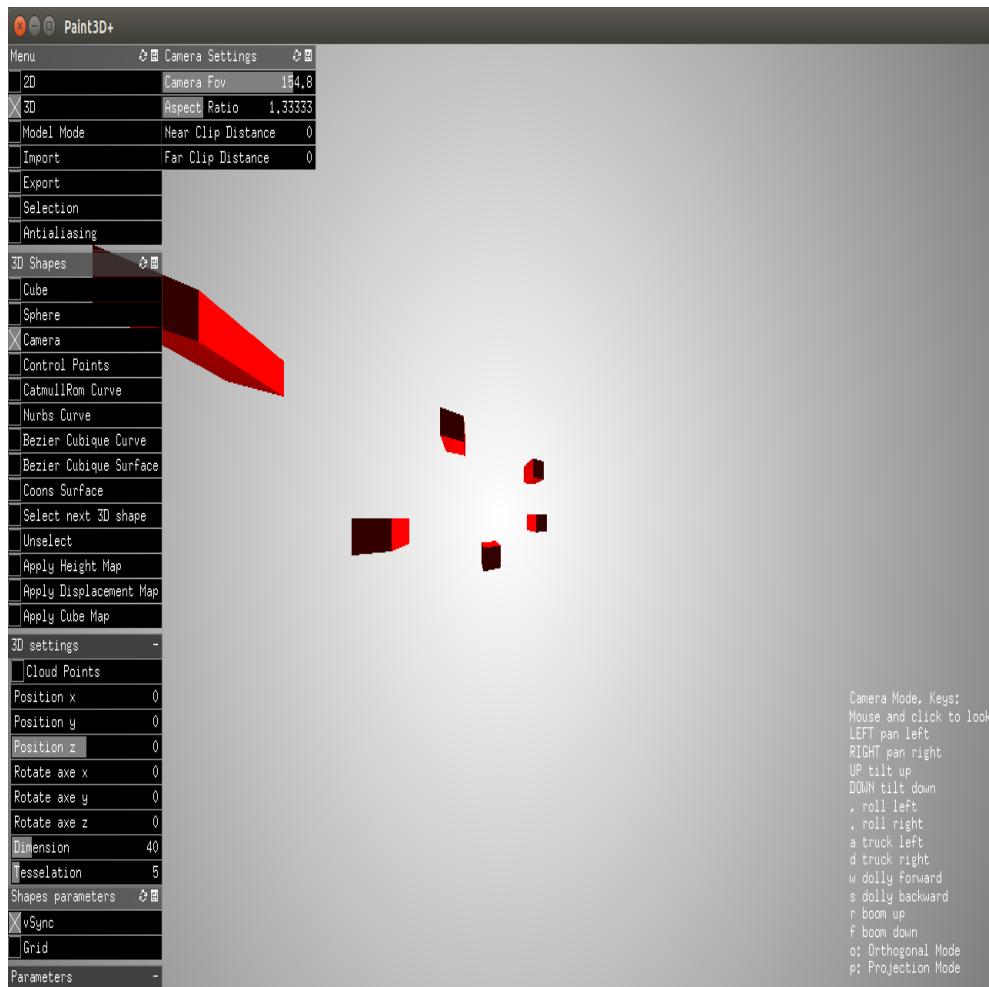


FIGURE 5.42 – Changement du champ de vision de la caméra (fov)

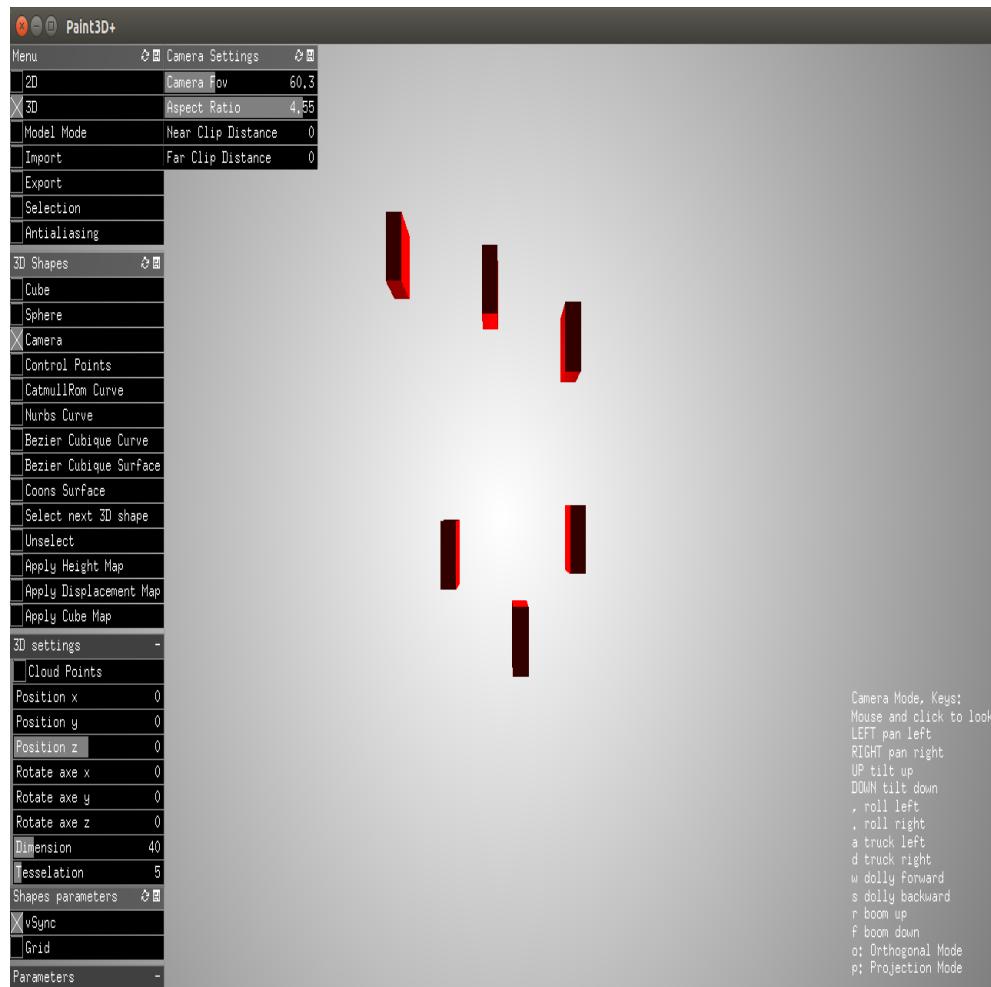


FIGURE 5.43 – Rendu du point de vue de la caméra après avoir changé interactivement l’aspect ratio

### 5.6.3 Caméras multiples

To add

## 5.7 Rastérisation

### 5.7.1 Texture d'élévation (height map)

Tel que le démontre la figure 5.44, il est possible d'appliquer une texture d'élévation à partir du logiciel Paint3D+. L'image utilisée pour cette texture est représentée à la figure 5.45. Un maillage géométrique (« Mesh ») est créé en ajoutant des faces à ce dernier. Les faces sont elles-même créées en fonction de quatre sommets définis selon chaque pixel x,y de l'image et de l'indice de luminosité à cet endroit. On procède donc à un surélèvement de l'image selon l'indice de luminosité de chaque pixel. Il est possible de voir une partie de code qui implémente cette fonction à la figure 5.46

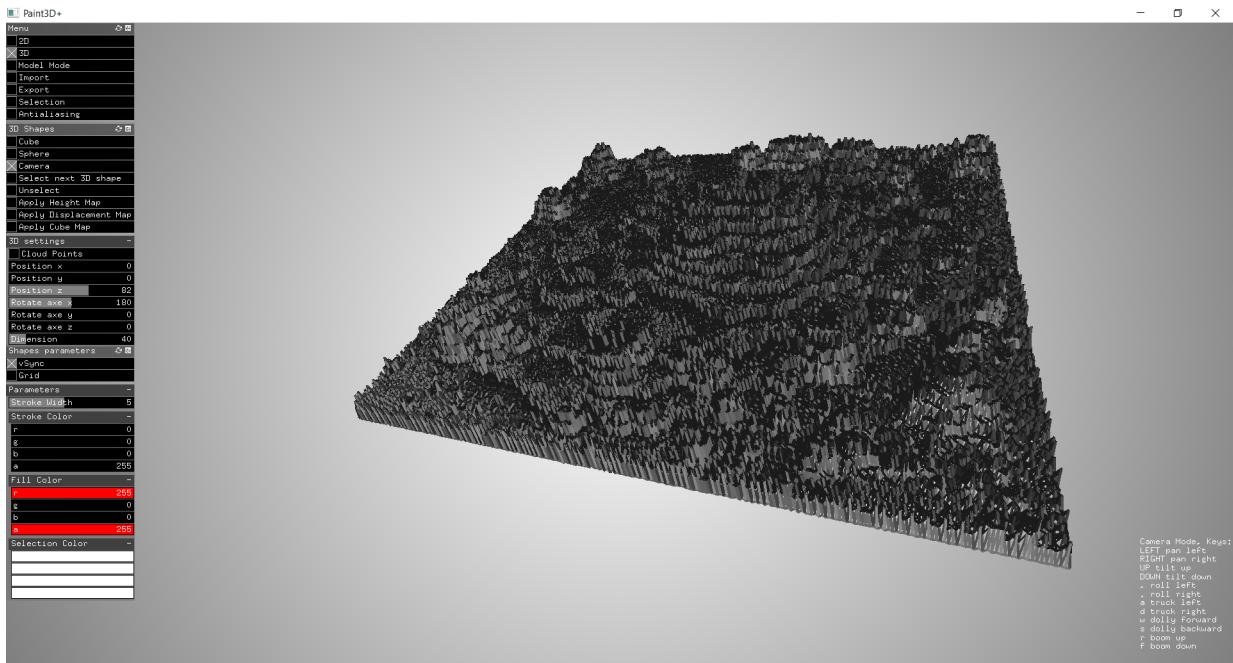


FIGURE 5.44 – Résultat de texture d'élévation

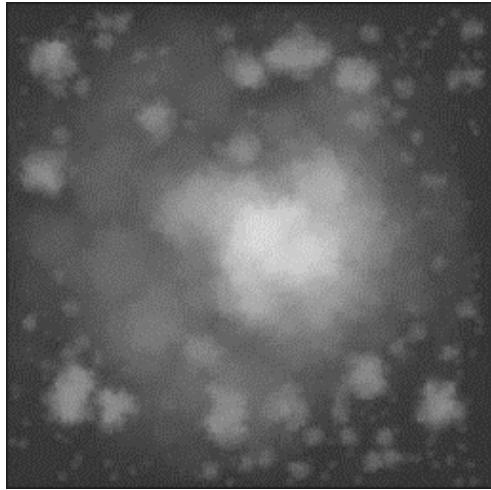


FIGURE 5.45 – Image source pour la texture d’élévation

```

void Terrain3D::prepareToDraw() {
    m_mesh.setMode(OF_PRIMITIVE_TRIANGLES);
    int skip = 1;
    int width = m_img.getWidth();
    int height = m_img.getHeight();
    for (int y = 0; y < height - skip; y += skip) {
        for (int x = 0; x < width - skip; x += skip) {
            ofVec3f nw = getVertexFromImg(m_img, x, y);
            ofVec3f ne = getVertexFromImg(m_img, x + skip, y);
            ofVec3f sw = getVertexFromImg(m_img, x, y + skip);
            ofVec3f se = getVertexFromImg(m_img, x + skip, y + skip);

            addFace(m_mesh, nw, ne, se, sw);
        }
    }
}

//-----
void Terrain3D::addFace(ofMesh& mesh, ofVec3f a, ofVec3f b, ofVec3f c, ofVec3f d) {
    addFace(mesh, a, b, c);
    addFace(mesh, a, c, d);
}

//-----
ofVec3f Terrain3D::getVertexFromImg(ofFloatImage& img, int x, int y) {
    return ofVec3f(x, y, 64 * img.getColor(x, y).getBrightness());
}
  
```

FIGURE 5.46 – Code pour la texture d’élévation

De plus, une fonction de displacement mapping a été ajouté pour répliquer le heightmap sur les formes géométriques.

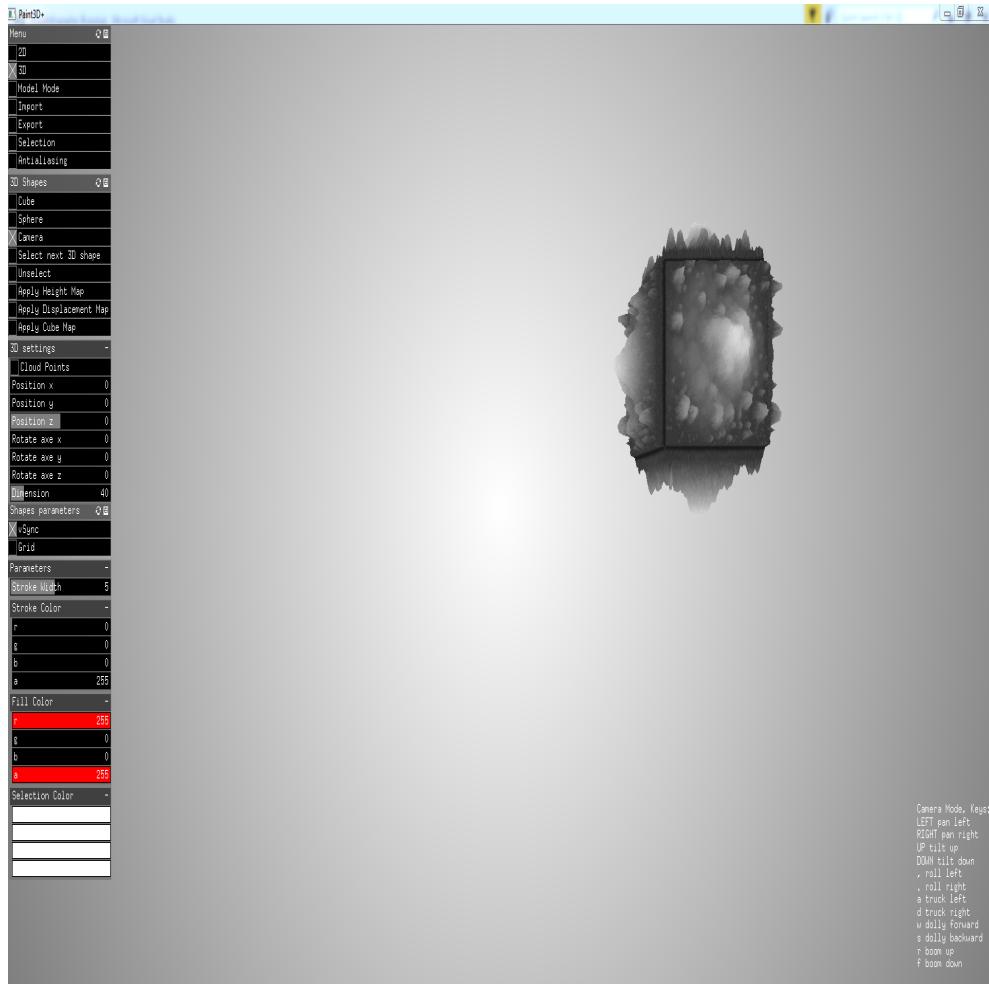


FIGURE 5.47 – Résultat du displacement mapping sur un cube

### 5.7.2 Texture de variation de normale (normal map)

### 5.7.3 Cube de réflexion (cube map)

Comme on peut le voir sur la figure 5.49, une option pour appliquer une texture cubeMap est disponible. L'image à la figure 5.48 est séparée en 6(haut, bas, gauche, droite, avant, arrière) et est ensuite appliquée à une forme géométrique grâce à un shader.



FIGURE 5.48 – Image de base du cubemap

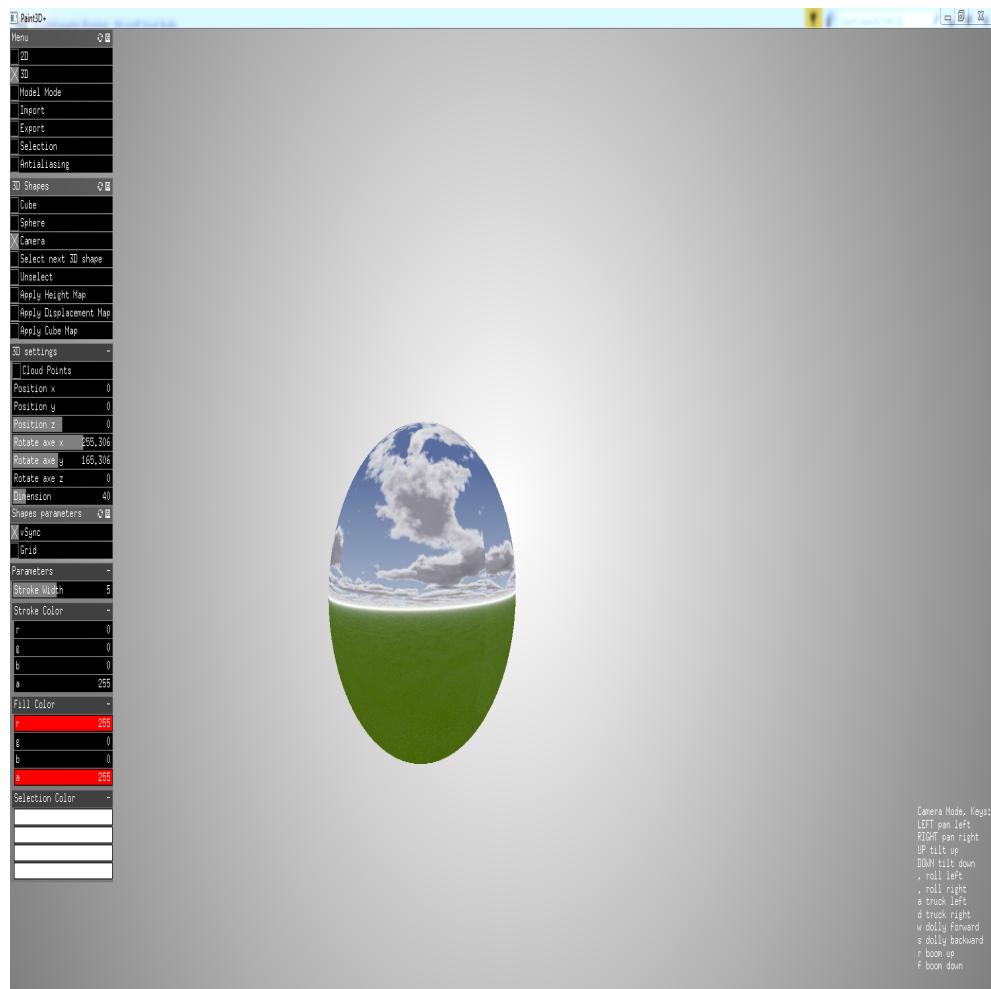


FIGURE 5.49 – Résultat du cube mapping sur une sphère

## 5.8 Courbes et surfaces

5.8.1 Courbe paramétrique

5.8.2 Surface paramétrique

5.8.3 Shader de tessellation

## 5.9 Lancer de rayon

5.9.1 Ombrage

5.9.2 Réfraction

5.9.3 Rendu graphique

## 5.10 Techniques de rendu

### 5.10.1 Shader de géométrie

Des shaders sont utilisés pour rendre les formes affectées par le cube mapping<sup>5.49</sup> et le displacement mapping<sup>5.47</sup>.

Voici les shaders de vertex et de fragments utilisés.

```
varying vec3 texcoord;  
  
void main(void)  
{  
    vec4 texcoord0 = gl_ModelViewMatrix * gl_Vertex;  
    //texcoord = texcoord0.xyz;  
    texcoord = normalize(gl_Vertex.xyz);  
  
    gl_Position = ftransform();  
  
}
```

FIGURE 5.50 – Shader de vertex du cubemap

```
uniform samplerCube EnvMap;
uniform int selected;
varying vec3 texcoord;

void main (void)
{
    vec3 envColor = vec3 (textureCube(EnvMap, texcoord));
    //vec3 envColor = vec3 (textureCube(EnvMap, gl_TexCoord[0]));
    if(selected == 1) envColor = vec3(1,1,1);

    gl_FragColor = vec4 (envColor, 1.0);
}
```

FIGURE 5.51 – Shader de fragment du cubemap

```
uniform sampler2D colormap;
uniform sampler2D bumpmap;
uniform float min;
varying vec2 TexCoord;
uniform int maxHeight;

void main(void) {
    TexCoord = gl_MultiTexCoord0.st;
    float realMin = min;

    vec4 bumpColor = texture2D(bumpmap, TexCoord);
    float df = 0.30*bumpColor.x + 0.59*bumpColor.y + 0.11*bumpColor.z;
    if(df - min < 0.03) realMin=df;
    df -= realMin;
    vec4 newVertexPos = vec4(gl_Normal * df * float(maxHeight), 0.0) + gl_Vertex;

    gl_Position = gl_ModelViewProjectionMatrix * newVertexPos;
}
```

FIGURE 5.52 – Shader de vertex du displacement map

```

uniform sampler2D colormap;
uniform sampler2D bumpmap;
varying vec2 TexCoord;
uniform float min;
uniform int selected;
void main(void) {
    vec4 color = texture2D(colormap, TexCoord);
    if(selected == 1) color = vec4(1,1,1,1);
    gl_FragColor = color;
}

```

FIGURE 5.53 – Shader de fragment du displacement map

### 5.10.2 BRDF

### 5.10.3 Effet visuel en post-process

L’interface graphique propose à l’utilisateur une option afin de permettre l’activation ou la désactivation de l’anti-crénelage de l’ensemble de la surface de rendu et des objets qui s’y retrouve. Cela est possible pour n’importe quel mode, que ce soit 2D ou 3D.

Voici quelques captures d’écran afin de le démontrer. Toutefois, il est plus facile de le voir lors de l’utilisation du logiciel que sur des images.

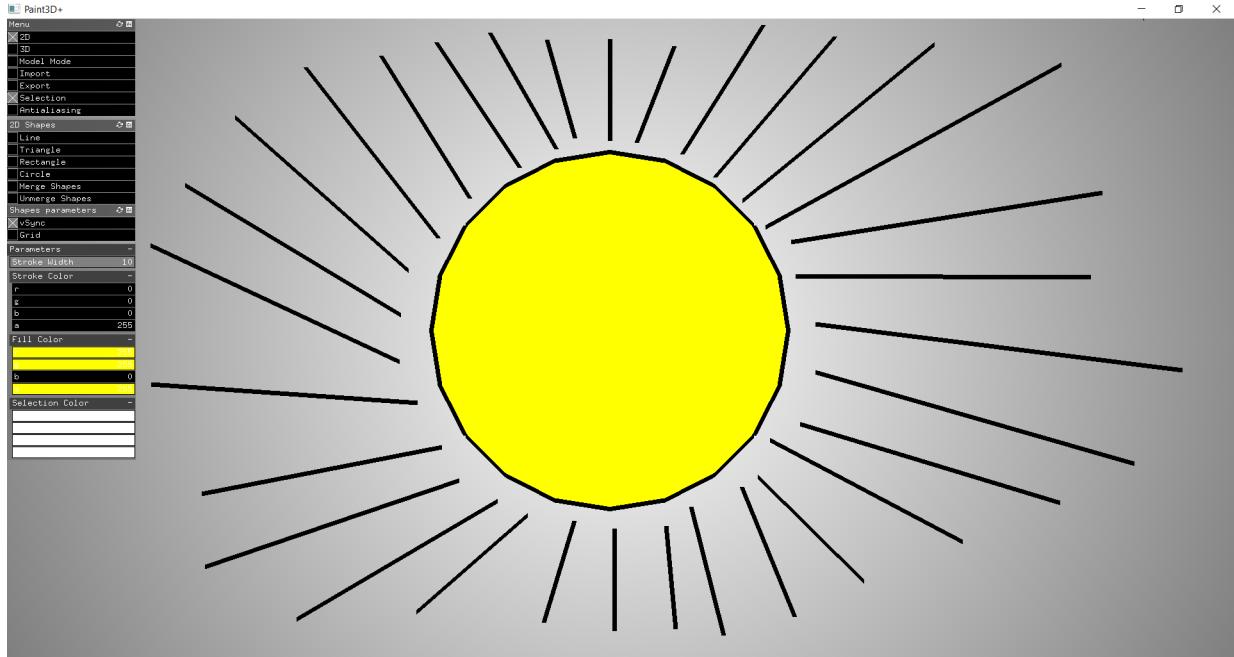


FIGURE 5.54 – 2D - Anti-crénelage désactivé

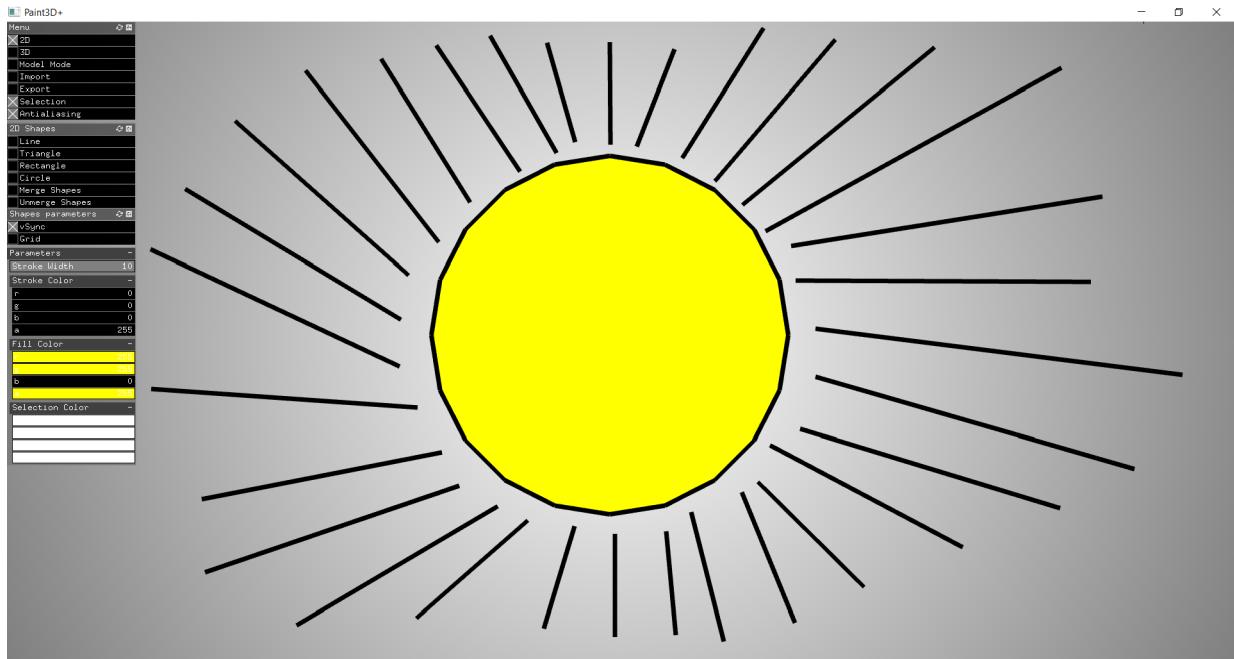


FIGURE 5.55 – 2D - Anti-crénelage activé

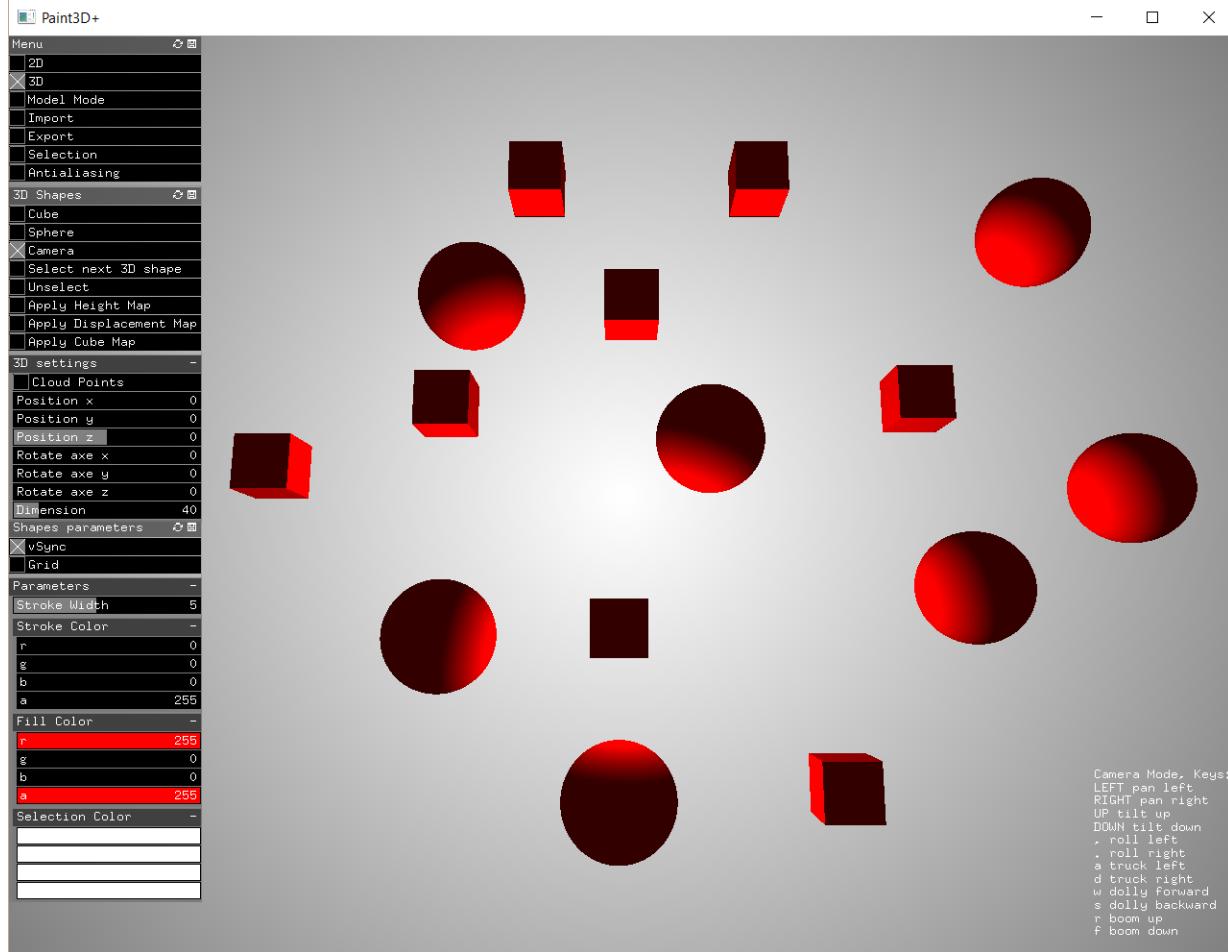


FIGURE 5.56 – 3D - Anti-crénelage désactivé

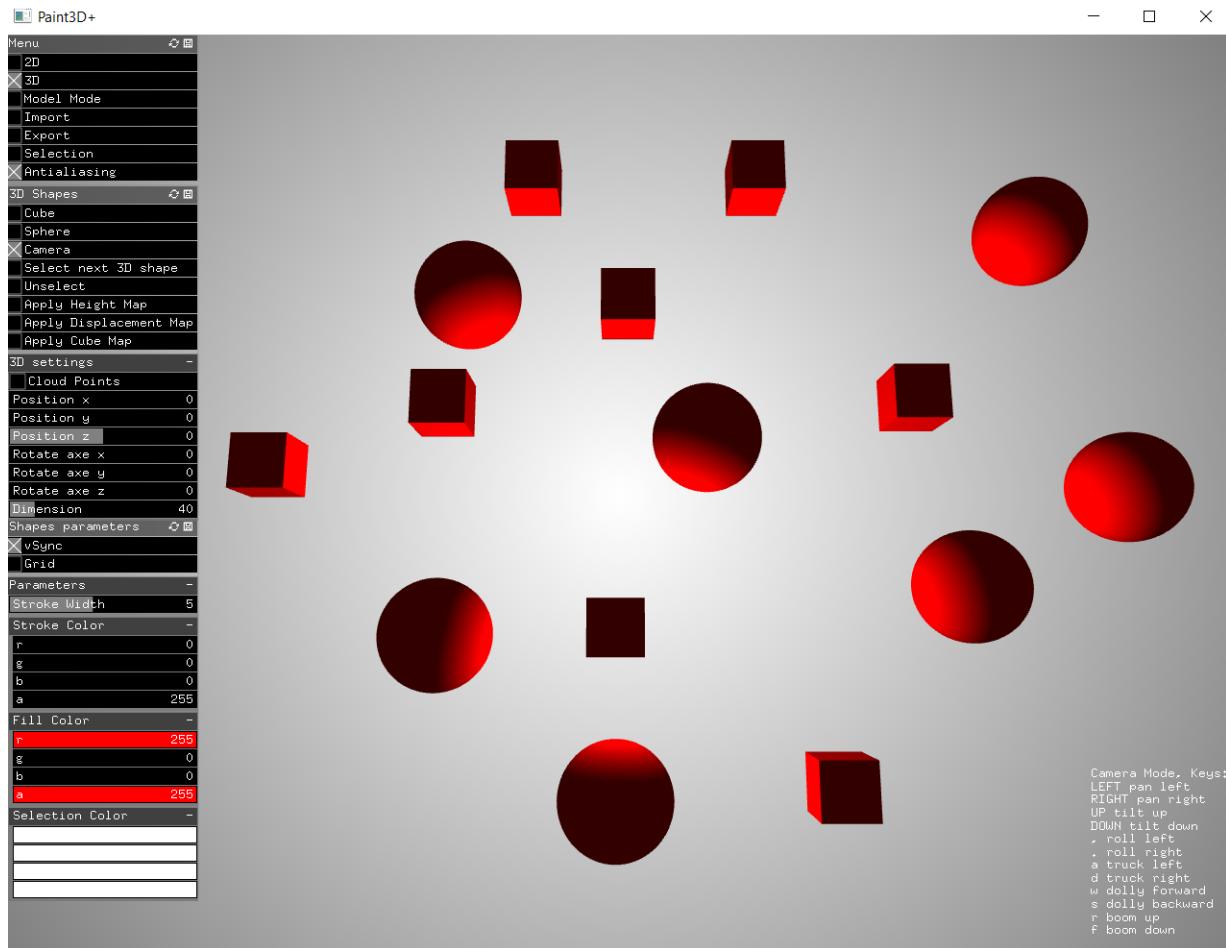


FIGURE 5.57 – 3D - Anti-crénelage activé

# Ressources

Comme notre application représente un outil de dessin, on n'a pas utilisé beaucoup de ressources externes (des différentes images, shaders, etc.). Les ressources utilisées pour accomplir le travail sont les "addons" et la documentation officielle de "Openframeworks" ([http ://openframeworks.cc/documentation/](http://openframeworks.cc/documentation/)). Pour construire l'interface graphique, on a utilisé le module "ofxGui". Cela nous a permis d'ajouter les différents boutons, le menu et les composantes pour modifier les paramètres des primitives dessinées. Pour lire et rendre les fichiers modèles, on a utilisé le module "ofxAximpModelLoader" (inclus en openframeworks aussi). La librairie "OpenGL" a été utilisée surtout pour représenter l'ombre sur les modèles (GL\_SMOOTH shading). Pour manipuler les entités 3D, on utilise "ofxOpenCv".

On a utilisé également les librairies standard de C++ (<cmath>, <vector>) pour faire les opérations mathématiques et pour utiliser des structures de données qui contiennent des références vers les objets instanciés.

# Présentation

L'équipe de développement est composé de 4 membres qui ont travaillé dans une symbiose fructueuse. Cela a permis d'effectuer un travail conforme aux exigences du cours. Plusieurs moyens de communication ont été utilisés pour l'organisation de travail et la répartition des tâches. Pour la planification de travail, nous avons priorisé les rencontres en personnes tandis que la phase de construction de l'application a été réalisée surtout à l'aide des rencontres virtuelles (skype). Également, la plateforme "Github" a été un outil indispensable pour le partage du code et la fusion des parties. À part le côté technique, la collaboration de tous les membres de l'équipe a été essentielle. On a essayé d'intégrer le plus souvent les fonctionnalités et GitHub a été indispensable pour accomplir cela. De cette façon, on a réduit le risque de conflits et on a détecté le plus vite possible les erreurs de développement (qui surviennent dans le cycle de vie du processus). L'intégration continue implique de faire de petits changements au logiciel en lui appliquant des tests de qualité (si les erreurs sont détectées plus tard dans le processus de développement, leurs corrections sont plus difficiles).

Les membres de l'équipe et leur programme d'étude :

Gabriel Lefrançois : génie informatique

Marcel Bernic : baccalauréat en informatique

Martin Richard Cerdà : génie informatique

William-José Simard-Touzet : génie informatique