Projet de session

Document de design

Visualiseur interactif de scènes 3d

(Partie 1/2)

Présenté à
Philippe Voyer

IFT-3100 — Infographie

Hiver 2017

UNIVERSITÉ LAVAL

Maxime Lavergne 111 110 783

maxime.lavergne.1@ulaval.ca

Benjamin Morency 906 195 150

benjamin.morency.1@ulaval.ca

Table des matières

Sommaire

Dans un monde où prime l'esthétique et le visuel et dans lequel nos écrans nous bombardent à coups de plus de 60 millions de pixels à la seconde, il est important de s'intéresser aux techniques et aux méthodes qui rendent possible ces rendus graphiques de haute voltige. Dans le but d'en savoir plus et d'acquérir de expertise en la matière, nous avons développé une application permettant à un utilisateur de générer des scènes avec rendu 3d en utilisant soit des fichiers de ressources externes à l'application (images, modèles), soit en créant interactivement certains objets ou figures géométriques.

Le présent projet consiste donc en la conception d'une application permettant la construction, l'édition et le rendu de scènes 3d. Pour ce faire, notre application met de l'avant différents concepts et notions liés à la manipulation et à la génération d'images, au dessin vectoriel, aux transformations géométriques dans l'espace, à la géométrie 2d et 3d et finalement aux caméras. Ce sont donc un total de 17 différentes fonctionnalités qui ont été implémentées en lien avec ces différentes thématiques pour permettre à l'utilisateur de créer des solides 3d tels le cube, la sphère et l'octaèdre, de créer des figures géométriques plus simples comme une ligne ou un rectangle, de charger des modèles 3d et d'appliquer sur tous ces objets des transformations dans l'espace, le tout, à l'aide d'une interface graphique minimaliste et intuitive.

Ce document présentera les différentes facettes de notre application, mettant de l'avant chacune des fonctionnalités offertes à l'utilisateur. Dans l'ordre, nous discuterons de l'interactivité possible avec l'interface graphique, des choix faits sur le plan technologique, de l'architecture logicielle, des différentes fonctionnalités de l'application, des ressources internes et externes utilisées pour rendre le projet et finalement une présentation des membres de l'équipe.

Interactivité

Le viewport peut être manipulé avec la souris. En effet, le bouton gauche de la souris permet un déplacement vers la gauche ou la droite. Le bouton de droite permet une rotation autour du point central, qui est p(0,0,0). En enfonçant la touche ‘shift’ et en utilisant la roulette de défilement de la souris, un zoom peut être fait.

Les interactions pour les transformations sont représentées à l'aide des trois curseurs suivants :

La translation :  La rotation :  Le zoom : 

La création d’objets se fait avec le menu sur la gauche. L’objet créé sera placé au centre de la scène, c’est-à-dire au point p(0,0,0) et pourra ensuite être manipulé et modifié s'il est sélectionné.

La sélection des objets se fait directement avec la souris. Une fois un ou plusieurs objets sélectionnés en cliquant dessus avec le bouton de gauche, les contrôles sont sensiblement les mêmes que lorsqu’aucun objet n'est sélectionné. Le bouton de gauche de la souris permet d’appliquer une translation. Le bouton de droite permet de modifier la rotation des objets. Le scale des objets est affecté avec la touche ‘shift’ enfoncée en addition d'un mouvement de la roulette de défilement de la souris. Un autre moyen d’affecter les transformations des objets est l’utilisation du menu de transformation qui apparaît à la gauche lorsqu’un ou plusieurs objets sont sélectionnés, le menu proposé étant différent s'il s'agit d'une sélection simple ou d'une sélection multiple. Pour annuler sa sélection, l'utilisateur peut appuyer sur la touche "u" du clavier.

Un bouton ‘Camera Options’ permet de basculer vers le menu où les options de la caméra peuvent être modifiées.

Il est possible d'annuler une transformation appliquée sur un ou plusieurs objets en utilisant la combinaison de touches clavier : CTRL + Z.

Finalement, la touche "x" du clavier permet la sauvegarde de l'image dans le dossier de l'application.

Technologie

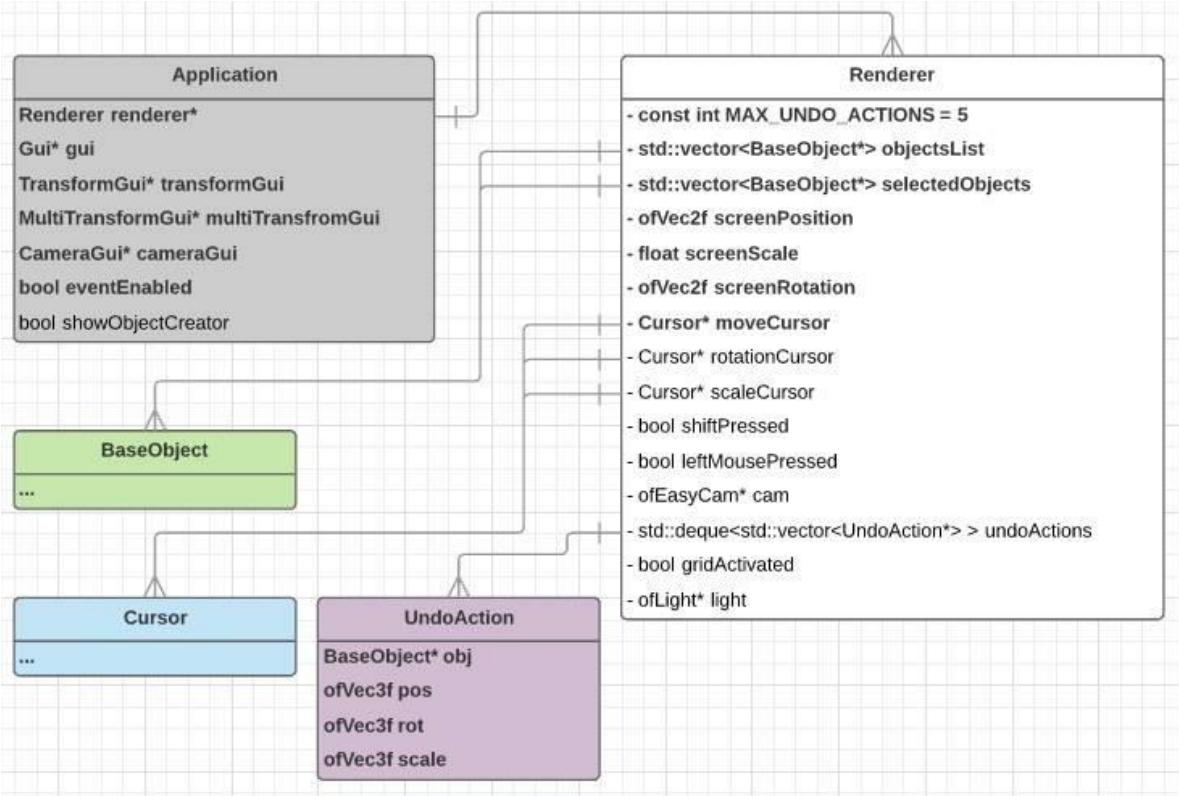
La technologie utilisée pour bâtir cette application est l'ensemble 'open source' OpenFramework 0.9.8 pour C++, travaillé via l'éditeur Visual Studio 2015 Community de Microsoft. La librairie proposée par OpenFramework a rendu possible la génération de plusieurs éléments graphiques tant 2d que 3d.

Les 'addons' OpenFramework utilisés sont ofxGui pour bâtir les différents menus et ofxAssimpModelLoader qui permet de faire le chargement de modèles 3d provenant de différents formats, notamment .obj et .3ds. De plus, ofxInputField a facilité la génération de boîtes de textes, permettant la saisie par l'utilisateur du nom de l'image à charger.

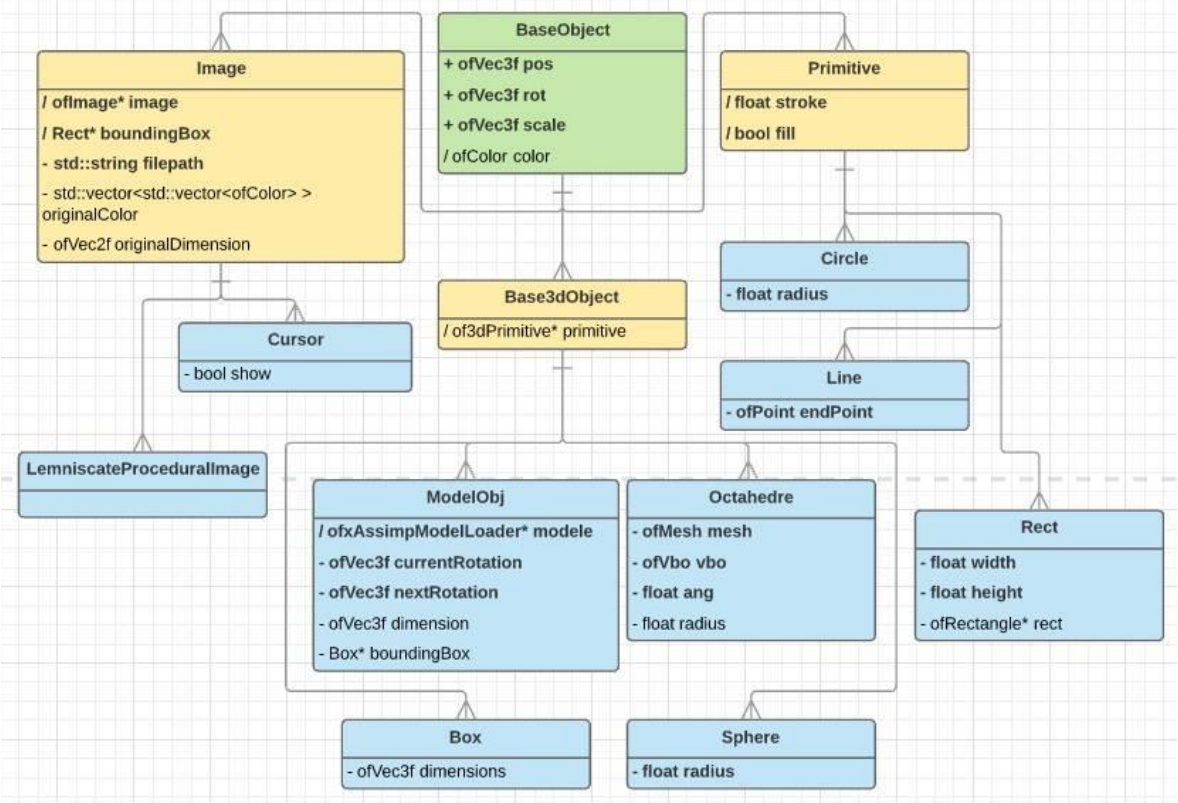
Finalement, le projet n'en serait probablement pas un si ce n'était de la librairie graphique libre OpenGL, permettant le rendu graphique de vecteurs 2d et 3d et favorisant la communication entre le CPU et le GPU (Graphics Processing Unit).

Architecture

L'architecture choisie pour la conception de notre application peut être résumée par les deux diagrammes suivants. Le premier présente une arborescence partant de l'application en elle-même et contenant les différentes interfaces ainsi que le renderer. Nous avons ensuite le renderer qui est responsable du rendu graphique de tous les éléments de la scène. Il contient les listes représentant les objets de la scène et les objets choisis, en plus de gérer les curseurs, les objets, l'affichage et la lumière. Viennent ensuite les BaseObject qui seront développés dans le prochain diagramme, tout comme les curseurs. Pour finir, nous avons un vecteur contenant les dernières modifications et une entité permettant de les annuler.



Ce second diagramme représente l'ensemble des entités pouvant être générées à partir de notre application. Le parent hiérarchique de tous les objets est le BaseObject. Viennent ensuite les images, les objets 3d et les primitives. Parmi les images, on retrouve l'image procédurale et les curseurs. Parmi les objets 3d on retrouve les modèles, l'octaèdre, la sphère et le cube. Finalement, pour les primitives, nous retrouvons le cercle, la line et le rectangle.



Fonctionnalités

1. Image

1.1 Importation

L'importation d'image se fait via l'interface à droite. Le champ texte 'Filepath' permet de spécifier le nom de l'image ou le chemin d'accès et le nom de l'image qui sera chargée après avoir appuyé sur le bouton 'Image'. Les images doivent se trouver dans le répertoire data ou dans un répertoire sous-jacent.

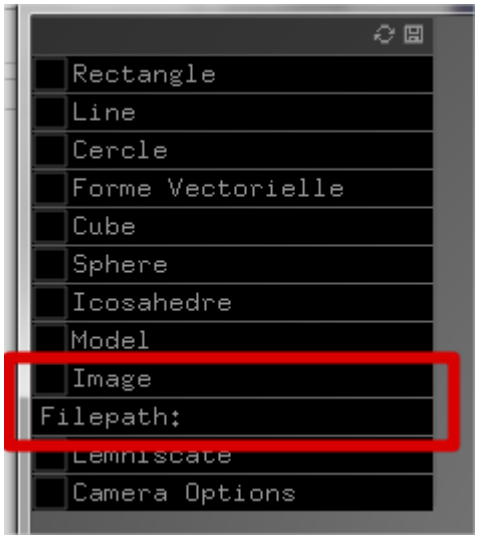


Figure 1: Interface graphique de création d'objets

Extrait du code :

Dans **image.h** :

```
protected:
	ofImage* image;
```

Dans **image.cpp** :

```
void Image::Load()
{
	// Allocation mémoire pour le chargement de l'image
	image = new ofImage();
	// Chargement de l'image située au chemin d'accès
	image->load(filepath);
	// Initialisation de l'image
	Setup();
}

void Image::Draw()
{
	ofPushMatrix();
	ofTranslate(pos.x + image->getWidth() * 0.5f, pos.y + image->getHeight() * 0.5f, pos.z);
	ofScale(scale);
	ofRotateX(rot.x);
	ofRotateY(rot.y);
	ofRotateZ(rot.z);
	image->setAnchorPoint(image->getWidth() * 0.5f, image->getHeight() * 0.5f);
	image->draw(0.0f, 0.0f, 0.0f, image->getWidth(), image->getHeight());
	ofPopMatrix();
}
```

1.2 Exportation

L'exportation permet de prendre une capture d'écran de la scène. Cette option se fait en appuyant sur 'X'. Le fichier résultant sera nommé 'Screenshot_x' où x est le 'timestamp' présent de l'ordinateur et placé dans le dossier /bin/data/.

Extrait du code :

Dans **application.cpp** :

```
void Application::keyPressed(int key)
{
	if (eventEnabled)
	{
		// La touche x permet de sauvegarder l'image de la scène actuelle sur le disque dur
		if (key == 'x')
		{
			time_t timev;
			time(&timev);

			ofImage img;
			img.grabScreen(0, 0, ofGetWidth(), ofGetHeight());
			img.save("screenshot_" + boost::lexical_cast<std::string>(timev) + ".png");

			return;
		}

	}
}
```

1.4 Traitement d'image

La teinte des objets peut être modifiée via des 'sliders' lorsqu'un ou des objets sont sélectionnés. L'opacité peut aussi être changée via le dernier 'slider', modifiant essentiellement la transparence de la sélection.

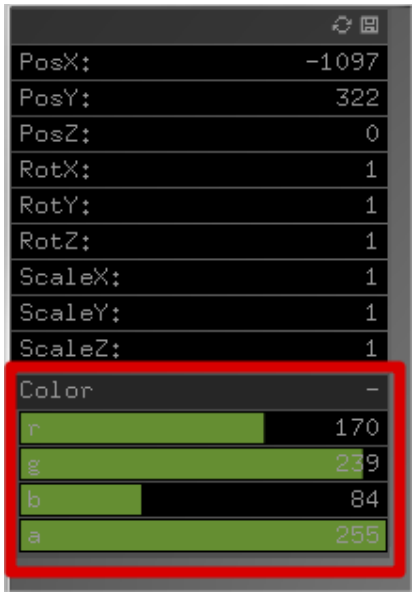


Figure 2: Interface graphique de transformation et de modification d'objets

Extrait du code :

Dans **image.cpp** (par exemple), où r,g,b et a sont les valeurs des sliders de couleur.

```
// Fonction permettant d'ajuster la teinte des pixels de l'image
void Image::SetColor(int r, int g, int b, int a)
{
	BaseObject::SetColor(r, g, b, a);
// On parcours tous les pixels de l'image
	for (int i = 0; i < image->getWidth(); i++)
	{
		for (int j = 0; j < image->getHeight(); j++)
		{
			// On récupère la couleur du pixel
			ofColor finalColor = image->getColor(i, j);
			// On ajuste la teinte de chaque couleur
			finalColor.r = (originalColor[i][j].r + color.r) / 2;
			finalColor.g = (originalColor[i][j].g + color.g) / 2;
			finalColor.b = (originalColor[i][j].b + color.b) / 2;
			finalColor.a = color.a;
			// On modifie le pixel
			image->setColor(i, j, finalColor);
		}
	[...]

// Fonction permettant d'ajuster la composante de transparence de l'image
void Image::SetAlpha(int a)
{
	// On parcours chaque pixel de l'image
	for (int i = 0; i < image->getWidth(); i++)
	{
		for (int j = 0; j < image->getHeight(); j++)
		{
			// On récupère la couleur du pixel
			ofColor color = image->getColor(i, j);
			// On ajuste sa composante de transparence
			color.a = a;
			// On modifie le pixel
			image->setColor(i, j, color);
		}
	}
}
```

// Une procédure similaire est appliquée sur les autres figures, utilisant la méthode ofSetColor et récupérant les valeurs des sliders de chaque couleur.

1.5 Image procédurale

La lemniscate est créée via un algorithme qui va procéduralement changer la couleur des pixels à un endroit précis dans les données de l'image.

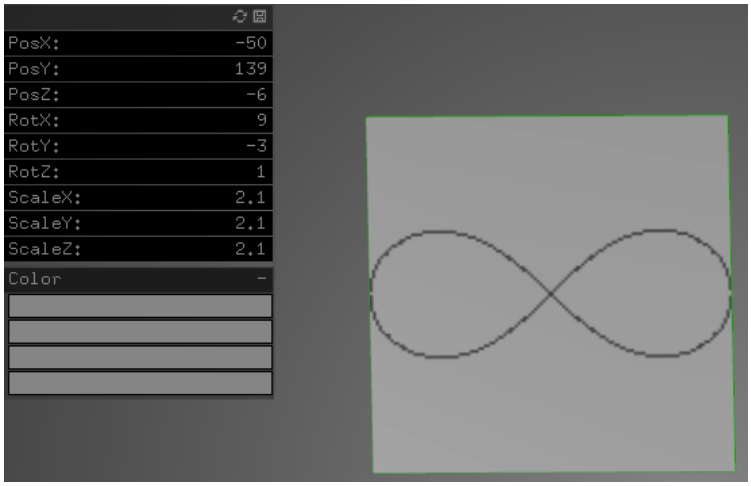


Figure 3: Interface de transformation et Lemniscate

Extrait du code :

Dans **lemniscateProceduralImage.cpp**

```
// Fonction permettant l'initialisation du lemniscate
void LemniscateProceduralImage::Setup()
{

	float length = image->getWidth() * 0.5f;
	// Dessin procédural du lemniscate
	for (float t = 0; t<360; t++)
	{
		float x = image->getWidth()*0.5f + (length * cos(ofDegToRad(t))) / (1 +
			pow(sin(ofDegToRad(t)), 2)); // x position
		float y = image->getHeight() * 0.5f + (length * sin(ofDegToRad(t)) *
			cos(ofDegToRad(t))) / (1 + pow(sin(ofDegToRad(t)), 2)); // y position
		image->setColor(x, y, ofColor(0, 0, 0));
	}
	// Mise à jour de l'image
	Image::Setup();
}
```

2. Dessin vectoriel

2.1 Curseur dynamique

Le curseur peut avoir 4 états distincts. La souris usuelle du système d'exploitation sera affichée lorsqu'aucune action ne sera effectuée. Un curseur spécifique est offert pour chacun des états suivant : translation, rotation et 'scale'

La translation :  La rotation :  Le zoom : 

Figures 4-5-6 : Curseurs de translation, de rotation et de zoom

Extrait du code :

Dans **cursor.cpp**

```
// Fonction permettant l'affichage du curseur à l'écran
void Cursor::Draw()
{
	// S'il est visible, on l'affiche à sa position actuelle
	if (show)
	{
		image->draw(pos.x - image->getWidth() * 0.5f, pos.y - image->getHeight() * 0.5f,
image->getWidth(), image->getHeight());
	}
}
```

Dans **renderer.cpp**

```
// Fonction permettant de cacher les curseurs spéciaux
void Renderer::HideAllCustomCursors()
{
	ofShowCursor();

	moveCursor->Hide();
	rotationCursor->Hide();
	scaleCursor->Hide();
}

// Si shift est enfoncé, on permet le zoom
	if (key == OF_KEY_SHIFT)
	{
		ofHideCursor();

		scaleCursor->pos = ofVec2f(ofGetMouseX(), ofGetMouseY());
		// On affiche le curseur de redimensionnement
		scaleCursor->Show();

		shiftPressed = true;
}
```

2.2 Primitives vectorielles

Trois types de primitives vectorielles ont été intégrés dans l'application. Le rectangle, le cercle ainsi que la ligne. Elles peuvent être créées via l'interface de création d'objets et ensuite transformées et modifiées à l'aide de l'interface de transformation des objets.

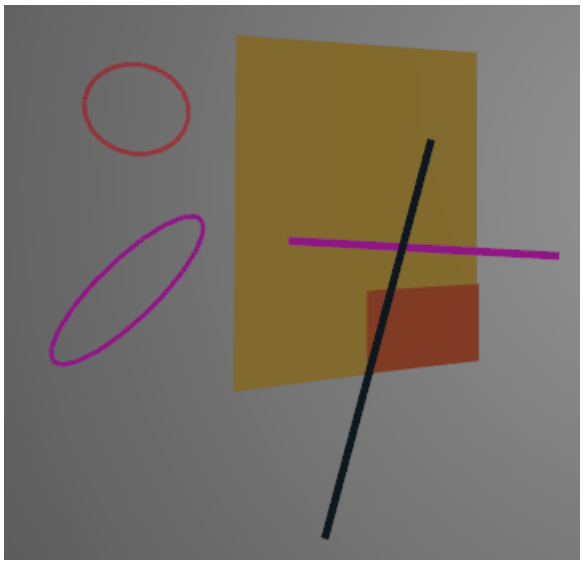


Figure 7: Rendu de lignes, de rectangles et de cercles après transformations

Extrait du code :

Dans **circle.cpp**

```
// Fonction permettant le rendu graphique d'un rectangle
void Rect::Draw()
{
	PreDraw();

	ofPushMatrix();
	// Transformations sur le rectangle
	[...]
	// Rendu du rectangle
	ofDrawRectangle(-width * 0.5f, -height * 0.5f, width, height);
	ofPopMatrix();

	PostDraw();
}
```

Dans **renderer.cpp**

```
// Fonction permettant de générer un cercle
void Renderer::CreateCercle()
{
	objectsList.push_back(new Circle(ofPoint(0, 0, 0), 50.0f, 3.0f, false, ofColor::white));
}
```

2.3 Formes vectorielles

Une image procédurale est créée à partir de plusieurs cercles concentriques de grosseurs différentes.

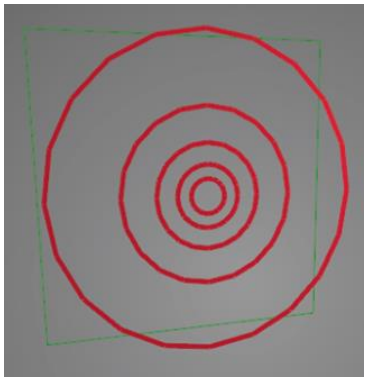


Figure 8: Rendu graphique d'une image formée de plusieurs primitives vectorielles.

Extrait du code :

Dans **FormeVectorielle.cpp**

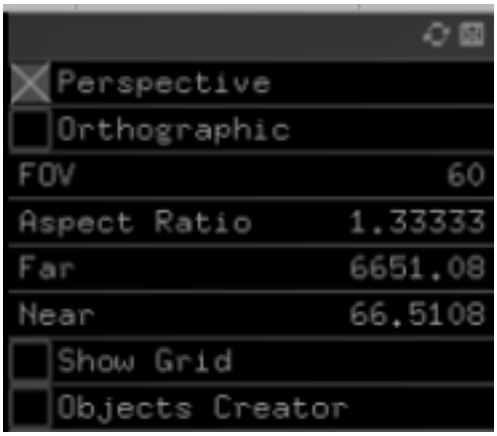
```
// Fonction permettant le rendu d'un cercle
void FormeVectorielle::Draw()
{
	PreDraw();
	// Dessin de 5 cercles concentriques
	ofDrawCircle(pos, radius);

	ofDrawCircle(pos, radius + 20);

	ofDrawCircle(pos, radius + 50);

	ofDrawCircle(pos, radius + 100);

	ofDrawCircle(pos, radius + 200);

	PostDraw();
}
```

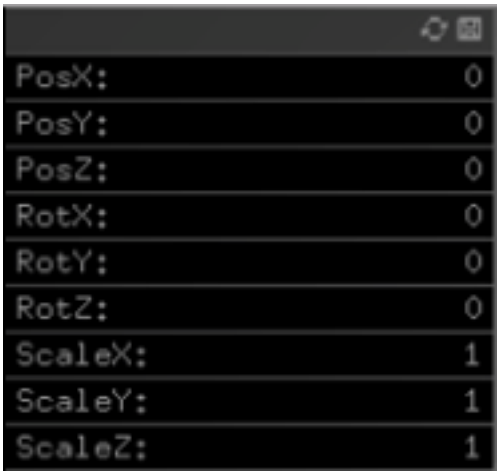
Dans **renderer.cpp**

```
// Fonction permettant de générer une Forme vectorielle
void Renderer::CreateFV()
{
	ofPoint center; center.x = 0; center.y = 0; float radius = 25; float stroke = 5; bool fill = false;
ofColor c; c.r = 255; c.g = 255; c.b = 0;
	objectsList.push_back(new FormeVectorielle(center,radius,stroke,fill,c));
}
```

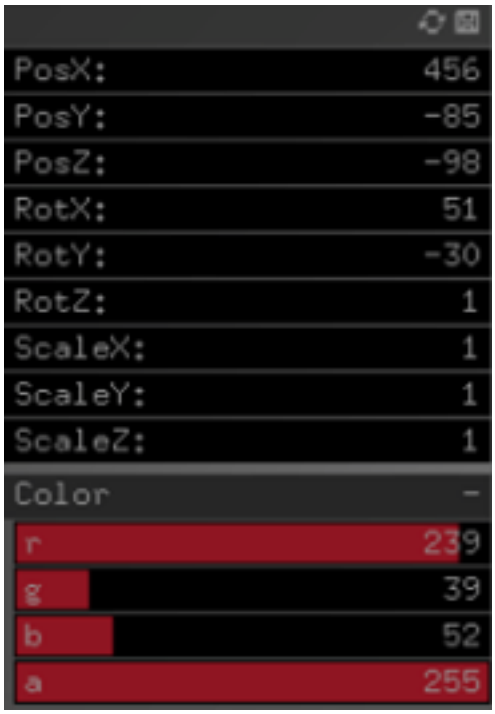
2.5 Interface

Une interface visuel est mis à la disposition de l'utilisateur, lui permettant de créer des objets, modifier les attributs des objets et de modifier les attributs relatifs à la caméra.





Figures 9 et 10: Interface de la caméra et interface de transformation sur sélection multiple



Figures 11 et 12: Interface de transformation sur sélection simple et interface de création d'objets

Extrait du code :

Dans **gui.cpp**

```
// Fonction permettant l'initialisation de l'interface graphique
void Gui::Setup()
{
	BaseGui::Setup();
	// Initialisation des options de l'interface graphique
	gui.setup();
	// Options permettant la génération de primitives 2d
	gui.add(createRectButton.setup("Rectangle"));
	gui.add(createLineButton.setup("Line"));
	gui.add(createCercleButton.setup("Cercle"));
	gui.add(createFVButton.setup("Forme Vectorielle"));
	// Options permettant la génération de primitives 3d
	gui.add(createCubeButton.setup("Cube"));
	gui.add(createSphereButton.setup("Sphere"));
	gui.add(createOctahedreButton.setup("Octahedre"));
	// Options permettant le chargement de modèles et d'images
	gui.add(createModelButton.setup("Model"));
	gui.add(createImageButton.setup("Image"));
	gui.add(imageInputField.setup("Filepath:", ""));
	// Option permettant le rendu d'une image procédurale
	gui.add(createLemniscateButton.setup("Lemniscate"));
	// Option permettant de passer à l'interface de la caméra
	gui.add(gotoCamOptions.setup("Camera Options"));
}
```

Dans **gui.h**

```
ofxButton& GetCreateRectangleButton() { return createRectButton; }
ofxButton& GetCreateLineButton() { return createLineButton; }
ofxButton& GetCreateCercleButton() { return createCercleButton; }

ofxButton createRectButton;
ofxButton createLineButton;
ofxButton createCercleButton;
```

Dans **application.cpp**

```
// Ajoute des listeners pour valider les clics des boutons de l'interface Création d'objets
gui->GetCreateRectangleButton().addListener(this, &Application::CreateRectangle);
gui->GetCreateLineButton().addListener(this, &Application::CreateLine);
gui->GetCreateCercleButton().addListener(this, &Application::CreateCercle);
```

[...]

3. Transformation

3.1 Transformation interactive

Chaque objet de la scène est modifiable suivant différents paramètres, soit la translation, la rotation, la proportion et la couleur. Si plusieurs entités sont sélectionnées, il est possible de leur appliquer une même transformation à tous en même temps. Si un seul objet est sélectionné, il sera aussi possible de modifier sa couleur avec les sliders.

Extrait du code :

Dans **Base3DObject.cpp**

```
// Fonction permettant le rendu graphique d'instances de la classe Base3DObject
void Base3DObject::Draw()
{
	// Détermination de certains paramètres et transformations
	ofSetColor(color);

	primitive->setScale(scale);
	primitive->setPosition(pos);
	primitive->setOrientation(rot);
	// Rendu graphique de la primitive associée au Base3DObject
	primitive->draw();

	ofSetColor(ofColor::white);
}
```

Dans **renderer.cpp**

```
// On déplace par translation tous les éléments sélectionnés
if (selectedObjects.size() > 0)
{
	for (int i = 0, count = selectedObjects.size(); i < count; i++)
	{
	ofVec3f mouseWorld = cam->screenToWorld(ofVec3f(x, y));
	ofVec3f mouseWorldPrev = cam->screenToWorld(ofVec3f(ofGetPreviousMouseX(),
ofGetPreviousMouseY()));
	selectedObjects[i]->pos.x += (mouseWorld.x - mouseWorldPrev.x) * 10;
	selectedObjects[i]->pos.y += (mouseWorld.y - mouseWorldPrev.y) * 10;
	selectedObjects[i]->pos.z += (mouseWorld.z - mouseWorldPrev.z) * 10;
	}
}
```

3.2 Structure de scène

Les objets créés par l'application sont contenus dans une structure de vecteur qui assure le rendu graphique de chacun des éléments qu'il contient à chaque frame. Ces éléments peuvent être sélectionnés par clic dans l'application et entrer dans une structure vecteur permettant la gestion des transformations, qui ne seront appliquées que sur les éléments du vecteur. Il est donc possible de ne modifier qu'une partie de la scène plutôt que d'affecter tous les éléments à la fois. Le vecteur peut être vidé en appuyant sur la touche "u" du clavier.

Extrait du code :

Dans **renderer.h**

```
std::vector<BaseObject*> objectsList;

std::vector<BaseObject*> selectedObjects;
```

Dans **renderer.cpp**

```
// Affichage de tous les objets de la scène
for (int i = 0; i < objectsList.size(); i++)
{
	objectsList[i]->Draw();
}

// Affichage d'une boîte autour des objects sélectionnés
for (int i = 0; i < selectedObjects.size(); i++)
{
	selectedObjects[i]->DrawBoundingBox();
}
```

3.3 Sélection multiple

Plusieurs objets peuvent être sélectionnés simultanément. Si les propriétés sont changé, le changement de la dite propriété sera appliqué sur toutes les objets sélectionnées.

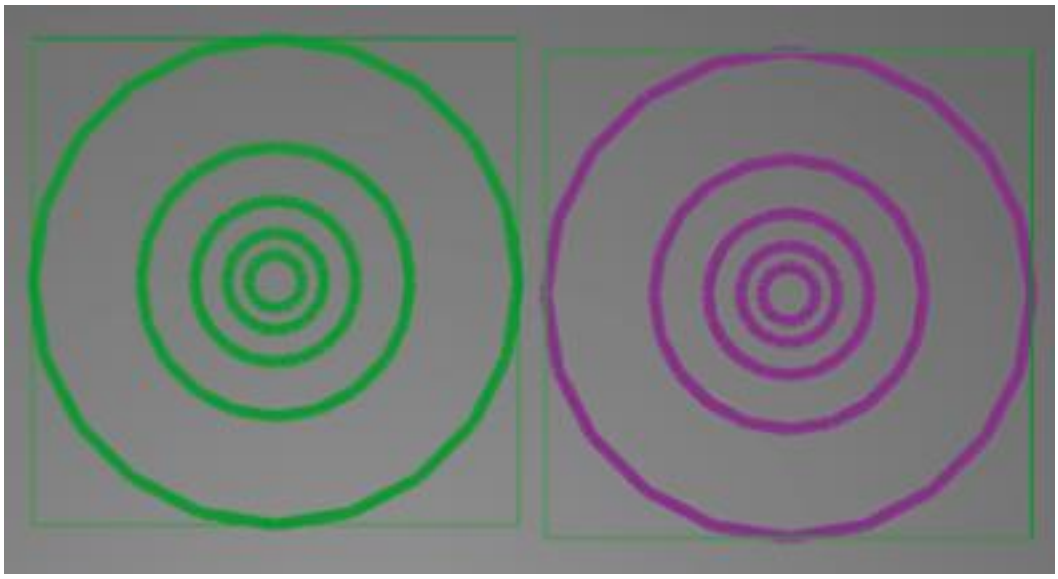


Figure 13: Boîtes de contour visibles sur une sélection de deux objets distinct

Extrait du code :

Dans **FormeVectorielle.cpp**

```
// Fonction permettant de déterminer s'il y a collision entre la souris et la figure géométrique
bool FormeVectorielle::CheckPointCollision(const ofVec3f& mouse, const ofVec3f& objScreenPos)
{
	return mouse.x >= objScreenPos.x - ((radius + 200) * 0.5f) * scale.x && mouse.y >= objScreenPos.y - ((radius + 200) * 0.5f) * scale.y &&
		mouse.x <= objScreenPos.x + ((radius +200) * 0.5f) * scale.x && mouse.y <= objScreenPos.y + ((radius + 200) * 0.5f) * scale.y;
}
```

Dans **renderer.cpp**

```
// Si le bouton de gauche est enfoncé
if (button == 0)
{
	bool hit = false;
	for (int i = 0, count = objectsList.size(); i < count; i++)
	{
		ofVec3f sPos = cam->worldToScreen(objectsList[i]->pos);
		hit = objectsList[i]->CheckPointCollision(ofVec3f(ofGetMouseX(), ofGetMouseY(),
		0.0f), sPos);
		if (hit)
		{
			std::vector<BaseObject*>::iterator obj = std::find(selectedObjects.begin(),
			selectedObjects.end(), objectsList[i]);
			if (obj == selectedObjects.end())
			{
				selectedObjects.push_back(objectsList[i]);
			}
			break;
		}
	}
	if (!hit)
	{
		selectedObjects.clear();
	}
[...]
```

3.5 Historique

L'historique des dernières modifications de transformation est conservé en mémoire jusqu'à 5 itérations. L'historique est utilisé en enfonçant 'CTRL' et en appuyant sur la touche 'z'. Ceci permet d'annuler certaines transformations indésirables et offre du contrôle à l'utilisateur.

Extrait du code :

Dans **renderer.cpp**

```
if (selectedObjects.size() > 0)
	{
		// Effacer la première action
		if (undoActions.size() == MAX_UNDO_ACTIONS)
		{
			undoActions.pop_front();
		}

		// Ajouter une action pour chaque objet sélectionné
		std::vector<UndoAction*> actions;
		for (int i = 0, count = selectedObjects.size(); i < count; i++)
		{
			actions.push_back(new UndoAction(selectedObjects[i]));
		}
		undoActions.push_back(actions);
	}

// Si on relâche ctrl + z
	else if (key == 26)
	{
		// Si la liste d'actions à supprimer n'est pas vide, on annule la dernière action
		if (!undoActions.empty())
		{
			std::vector<UndoAction*> toApply = undoActions[undoActions.size() - 1];
			undoActions.pop_back();

			for (int i = 0, count = toApply.size(); i < count; i++)
			{
				toApply[i]->obj->pos = toApply[i]->pos;
				toApply[i]->obj->rot = toApply[i]->rot;
				toApply[i]->obj->scale = toApply[i]->scale;
			}
		}
	}
```

4. Géométrie

4.2 Primitives

Les primitives géométriques intégrées dans l'application sont la sphère, le cube et l'octaèdre. Ils peuvent être générés automatiquement en cliquant sur le bouton correspondant au solide désiré dans l'interface de création d'objets. La sphère et le cube sortent directement d'openframework alors que l'octaèdre est obtenu par création d'un mesh avec les valeurs des sommets ainsi qu'avec les indices des sommets de la génération du maillage triangulaire.

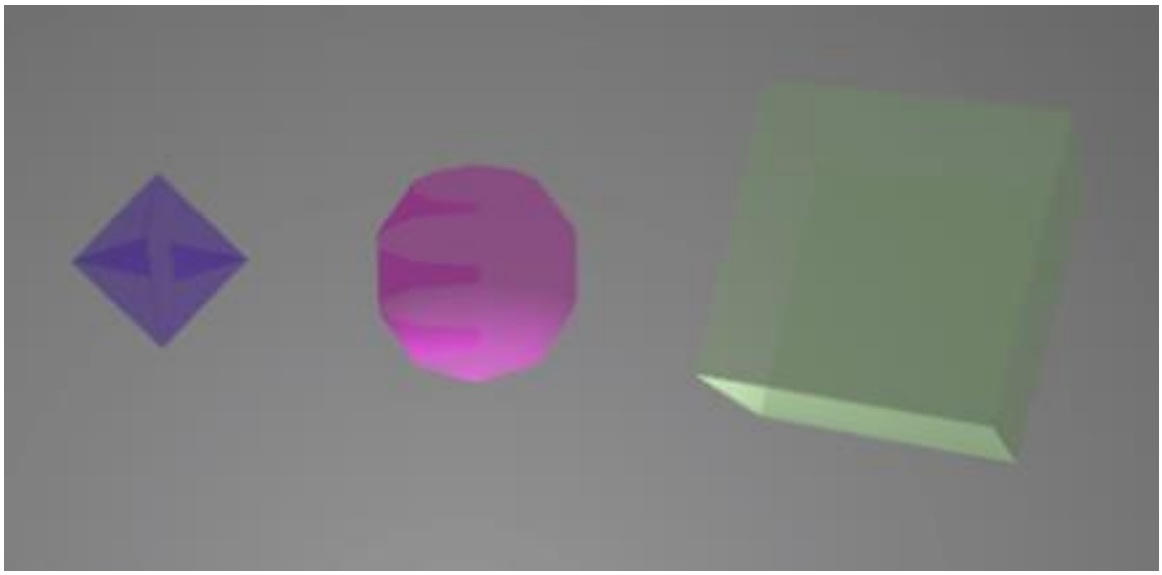


Figure 14 : De gauche à droite : L'octaèdre, la sphère et le cube.

Extrait du code :

Dans **sphere.cpp, application.cpp et renderer.cpp**

```
// Fonction permettant de charger certains paramètres de la sphère
void Sphere::Load()
{	primitive = new ofSpherePrimitive(radius, 5);}


// Fonction appelant la méthode CreateSphere du renderer
void Application::CreateSphere()
{	renderer->CreateSphere();}

// Fonction permettant de générer une sphère
void Renderer::CreateSphere()
{objectsList.push_back(new Sphere());}
```

4.3 Modèle

L'importation d'un modèle 3D est faite à partir d'un modèle .obj ou .3ds situé à l'intérieur du dossier /bin/data/. À l'aide d'un simple bouton se trouvant sur l'interface de création d'objet, il est possible de charger un modèle prédéterminé. Pour simplifier le travail, un seul modèle a été implémenté avec sa boîte de sélection individualisée. Il s'agit d'un modèle du populaire personnage "MegaMan" de la série de jeux vidéo du même nom.

Figure 15: Modèle de MegaMan (voir Ressources)

Extrait du code :

Dans **modelObj.h, modelObj.cpp, et renderer.cpp**

```
ofxAssimpModelLoader* modele;

// Fonction permettant le chargement d'un nouveau modèle format obj ou 3ds
void ModelObj::Load()
{
	// On génère un identifiant pour le modèle
	modele = new ofxAssimpModelLoader();
	// On récupère et initialise le modèle dont le nom de fichier est connu
	modele->loadModel(filepath);
	Setup();

	dimension = ofVec3f(100.0f, 300.0f, 100.0f);
	boundingBox = new Box(0.0f, 0.0f, 0.0f, dimension.x, dimension.y, dimension.z);
}

// Fonction permettant le rendu du modèle après transformations dans l'espace
void ModelObj::Draw() {
	// Initialisation de paramètres
	ofSetColor(color);

	ofPushMatrix();
	[...]
	modele->setPosition(0.0f, 0.0f, 0.0f);
	ofTranslate(0.0f, dimension.y * 0.5f, dimension.z * 0.5f);
	// Rendu graphique du mesh du modèle
	modele->draw(ofPolyRenderMode::OF_MESH_FILL);
	ofPopMatrix();

	ofSetColor(ofColor::white);
}


// Fonction permettant la génération d'un modèle à l'aide d'un chemin source
void Renderer::CreateModel(const std::string& filepath)
{
	objectsList.push_back(new ModelObj(filepath));
}
```

4.4 Texture

Le modèle de MegaMan présente une texture dont les coordonnées de mapping sont adéquatement distribuées sur la surface du maillage géométrique (voir Figure 15 à la page précédente).

5. Caméra

5.1 Propriétés de caméra

Les propriétés de la caméra peuvent être modifiées via le panneau de configuration de caméra. Il est possible pour l'utilisateur de modifier l'angle du champ de vision, le ratio d'aspect et la distance du plan de clipping avant et arrière.

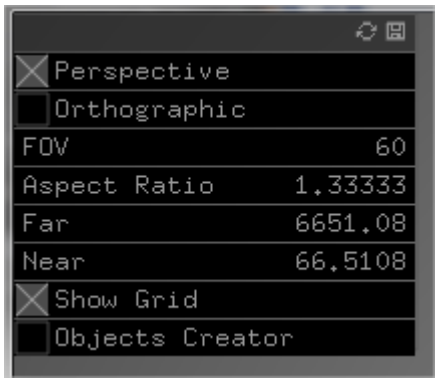


Figure 9 : Interface de la caméra

Extrait du code :

Dans **cameraGui.cpp**

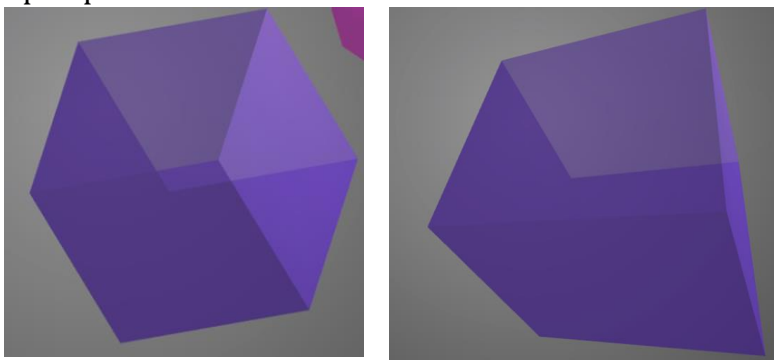
```
void CameraGui::Setup()
{
	BaseGui::Setup();
	// Initialisation de l'interface de la caméra
	gui.setup();
	// Paramètres liés à la projection
	gui.add(perspToggle.setup("Perspective", true));
	gui.add(orthoToggle.setup("Orthographic", false));
	// Paramètres liés au champ de vision et aux plans avant/arrière
	gui.add(fovField.setup("FOV", 60, 0, 300));
	gui.add(aspectRatio.setup("Aspect Ratio", 1.33f, 0, 2.0f));
	gui.add(farClipping.setup("Far", 2000, 0, 3000));
	gui.add(nearClipping.setup("Near", 0, 0, 3000));
	// Bouton permettant l'affichage d'un quadrillage
	gui.add(gridActivatorToggle.setup("Show Grid", true));
	// Bouton permettant de passer aux options de création d'objets
	gui.add(showObjectsCreator.setup("Objects Creator"));
}
```

Dans **application.cpp**

```
// Fonction appelant la méthode du renderer permettant de modifier l'angle d'ouverture du champ de vision
void Application::SetFOV(const void* sender, float& value)
{
	renderer->SetFOV(value);
}
// Fonction appelant la méthode du renderer permettant de modifier le ratio d'aspect
void Application::SetAspectRatio(const void* sender, float& value)
{
	renderer->SetAspectRatio(value);
}
// Fonction appelant la méthode du renderer permettant de modifier la valeur du plan de clipping arrière
void Application::SetFarClippingPlane(const void* sender, float& value)
{
	renderer->SetFarClippingPlane(value);
}
// Fonction appelant la méthode du renderer permettant de modifier la valeur du plan de clipping avancé
void Application::SetNearClippingPlane(const void* sender, float& value)
{
	renderer->SetNearClippingPlane(value);
}
```

5.2 Mode de projection

Le mode de projection peut être changé de la même manière que les propriétés de la caméra. Il existe deux modes de projection disponibles : la projection orthogonale et la projection en perspective.



Figures 16 et 17 : Représentation en projection orthogonale et en perspective d'un même cube.

Extrait du code :

Dans **application.cpp**

```
// Fonction permettant de mettre la vision de la caméra en projection en perspective
void Application::CamToPerspective(const void* sender, bool& pressed)
{
	// Si cliqué, Perspective est en fonction, sinon c'est le mode projection orthogonale
	if (pressed)
	{
		cameraGui->GetOrhtoToggle() = false;
		cameraGui->GetPerspToggle() = true;
		renderer->CamToPerspective();
	}
	else{
		cameraGui->GetOrhtoToggle() = true;
		cameraGui->GetPerspToggle() = false;
		renderer->CamToOrtho();
	}
}
// Fonction permettant de mettre la vision de la caméra en projection orthogonale
void Application::CamToOrtho(const void* sender, bool& pressed)
{
	// Si cliqué, orthogonale, sinon perspective
	if (pressed)
	{
		cameraGui->GetOrhtoToggle() = true;
		cameraGui->GetPerspToggle() = false;
		renderer->CamToOrtho();
	}
	else{
		cameraGui->GetOrhtoToggle() = false;
		cameraGui->GetPerspToggle() = true;
		renderer->CamToPerspective();
	}
}
```

Dans **renderer.h**

```
void CamToPerspective() { cam->disableOrtho(); }
void CamToOrtho() { cam->enableOrtho(); }
```

Ressources

Modèle 3d de Megaman :

https://www.models-resource.com/3ds/supersmashbrosfornintendo3ds/model/9087/

Image tableflip.jpg :

https://imgflip.com/memegenerator/Table-Flip-Guy

Image cave.jpg :

http://www.wondermondo.com/Countries/NA/LesserAntilles/Aruba/QuadirikiriCave.htm

Image darkness.jpg :

http://weknowyourdreams.com/darkness.html

Image nature.jpg :

http://www.planwallpaper.com/wallpaper-nature

Présentation

L’équipe est compose de deux membres: Maxime Lavergne et Benjamin Morency. Nous nous sommes connus auparavant dans des cours antérieurs et nous avons toujours bien travaillé ensemble.

Maxime Lavergne : Étudiant au baccalauréat en informatique et programmeur chez Frima Studio depuis 2011. Passionné de jeux vidéo, j’ai fait un AEC au Collège Bart, ce qui m’a permis d’obtenir un emploi chez Frima Studio. Après quelques années, j’ai décidé de faire mon baccalauréat pour combler certaines lacunes en termes de compréhension de l’informatique en général.

Benjamin Morency : Étudiant au baccalauréat en informatique depuis l'automne 2015 et enseignant en mathématiques au secondaire depuis 2010. Passionné des jeux vidéo, de la logique et de l'animation depuis mon très jeune âge, j'envisage suivre le cours de programmation de jeux vidéo à l'automne pour mettre à profit le cours d'infographie et mon amour du domaine. Je fais actuellement le baccalauréat pour acquérir de nouvelles notions dans un domaine qui me fascine et pour faire une réorientation de carrière, suivant les opportunités qui se présenteront.