UNIVERSITÉ DE TECHNOLOGIE DE BELFORT-MONTBÉLIARD

Animation d'un personnage 3D avec OpenGL

Rapport de projet IN55 – A2018

Vanson Guillaume
Ponnelle Remy

Département Informatique

Professeur
Lauri Fabrice

utbm
université de technologie
Belfort-Montbéliard

Sommaire

Sommaire 2
1. Présentation du projet 3
2. Modélisation et armature 4
3. Architecture du projet 5
   i. Choix technologiques 5
   ii. Structure de données 5
      a. Structure d'importation 5
      b. Structure d'affichage 5
4. Diagramme de classe 6
   i. Structure d'importation 6
   ii. Structure de l'animation 7
5. Fonctionnement de l'application 9
   i. Dérouler de l'importation 9
   ii. Dérouler de l'animation 11
      a. L'animator 11
      b. Le vertex shader 11
   iii. Mode d'emploi 12
6. Bilan 13
   i. Difficulté rencontrée 13
   ii. Limites du projet rendu et amélioration possible 13
   iii. Conclusion 13

1. Présentation du projet

Pour le semestre P18, et dans le cadre de l'UV IN55, nous avons choisi comme projet l'Animation d'un personnage 3D. Pour cela nous avons représenté un personnage plutôt simple facilitant la mise en place d'animations. Avec cette structure très simple nous avons pu tester plusieurs animations.

Nous développerons dans ce rapport la modélisation, l'architecture logiciel et le fonctionnement de notre projet.

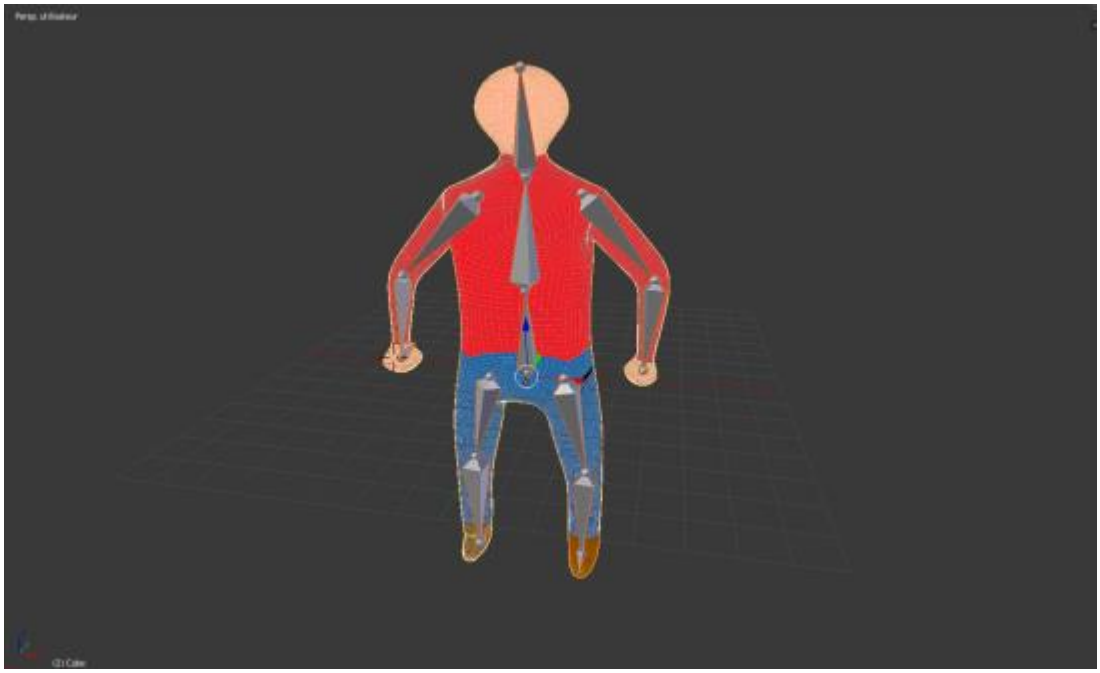
2. Modélisation et armature

Ce personnage a été réalisé à l'aide de Blender. Logiciel gratuit accessible à tous, il est également facile d'utilisation et se prend facilement en main. Ce personnage est donc composé d'une peau et d'une armature d'os.

Cette armature est donc composée de 13 os :

- 2 pour chaque bras
- 3 pour chaque jambe
- 2 pour le corps
- 1 pour le cou
- 1 pour la tête

Ces différents os sont reliés entre eux et ont une orientation permettant de reproduire les mouvements d'un être humain.



3. Architecture du projet

i. Choix technologiques

Le sujet nous imposait d'utiliser le langage orienté agent C++ ainsi que OpenGl pour le rendu graphique. Comme nous avions utilisé Qt pendant nos TD et TP nous avons choisi de continuer sur cet outil de développement afin de pas avoir à en apprivoiser d'autre, n'ayant ni l'un ni l'autre réaliser de projet en C++ avant celui-ci.
Notre modèles 3D seras modéliser sur le logiciel Blender car il est gratuit est permet de réaliser assez vite des modèles. Pour ce qui est du format de fichier pour le modèle 3D, nous stockons notre modèle dans un fichier colada (.dae), il s'agit d'un format xml simple pouvant contenir des animations. Malheureusement nous nous sommes rendons compte plus tard que blender exportai une seule animation dans ce fichier ainsi notre projet ne peut gérer qu'une animation.

ii. Structure de données

Notre projet se découpe en deux grandes parties la partie Importation du modèle depuis le fichier colada et la partie affichage et animation sur OpenGL

a. Structure d'importation

Pour l'importation on peut décomposer la structure en 4 parties principales, l'importateur de la géométrie du modèle, l'importateur des animations des différents os (le mouvement des os influence la géométrie), l'importateur du squelette (comment les os sont organiser et leur influence sur les ne diffèrent point du modèle) et enfin l'importateur de la texture. Ces importateurs utilisent des classes stockant les informations récupérées.

- **Importateur du squelette :**

Cet importateur est en réalité composé de deux importateurs :
Le SkinLoader retournant des SkinningData contenant le mapping entre le nom de os et leurs indices et une liste de VertexSkinData contenant le mapping entre les os et leurs poids sur les points.
Le SkeletonLoader retournant des SkeletonData, contenant le nombre d'os ainsi qu'une hiérarchie de BoneData. Les Bone Data contiennent le nom de l'os, son index sa transformation local par défaut et la liste des os dont il est le parent.

- **Importateur de la géométrie :**

Cet importateur se sert retourne un MeshData se mesh contient la liste des points, des normales, des coordonnées de textures, et se sert de l'importateur précédent pour stocker les index des os et les poids. Pour un indice i on a donc un point et toutes les infos le concernant. Afin de pouvoir construire le modèle un tableau d'indice permet de savoir quel point relier afin de former des faces triangulaires.

- **Importateur des animations :**

Cet importateur renvois un AnimationData, il contient l'ensemble des KeyFrame, temps dans l'animation ou on connait les orientation ainsi déplacement des os qui sont stocker dans des BoneTransformData.

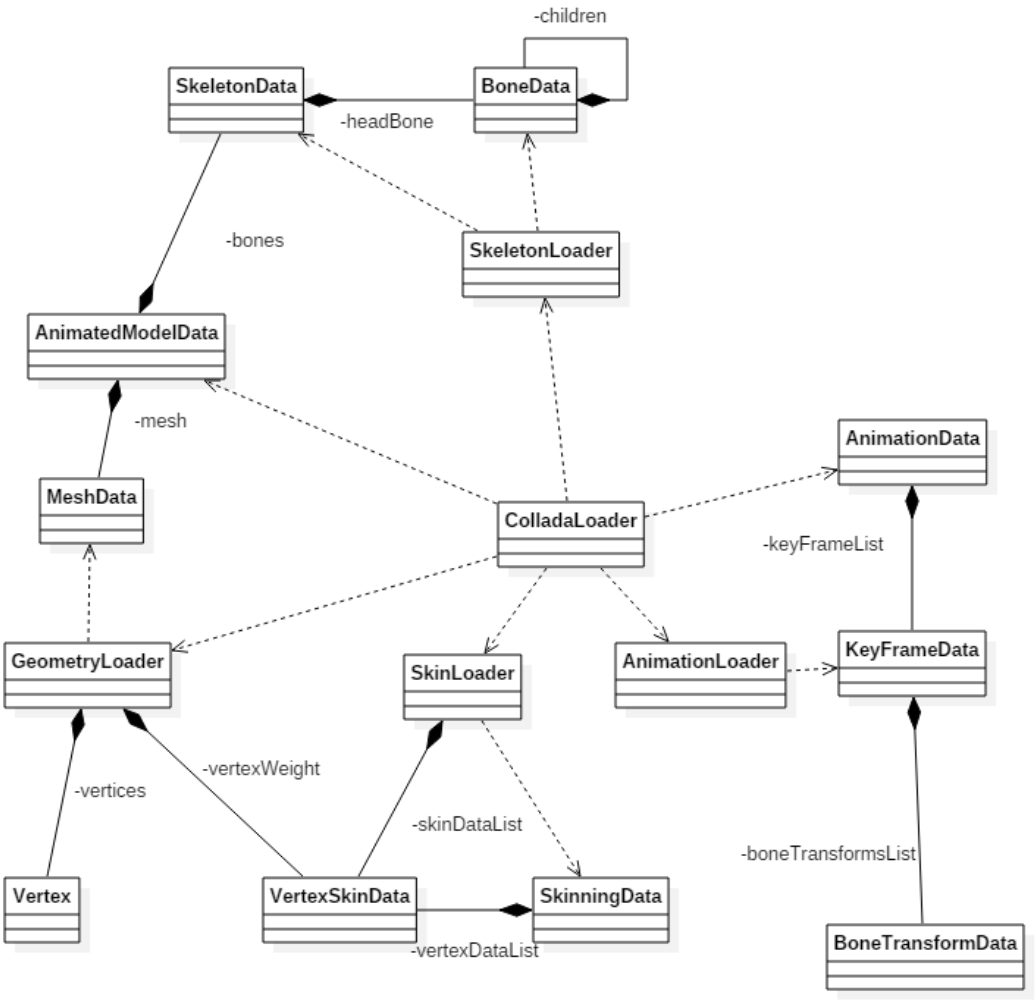
b. Structure d'affichage

Afin de stocker notre modèle après avoir récupérer les données, on les convertie en Object3DDynamic cette classe contiendra la texture, l'Animation, l'Animator, les os et enfin le maillage.
L'Animator sert à lire l'animation et mettre à jour les transformations des os afin d'animer les modèles.
Une Animation contient une liste de BoneAnimation il s'agit de la liste de tout les KeyFrame associer à un os donné afin de pouvoir avoir des Key Frames a des temps diffèrent pour nos os.
Le Mesh contient tous les points, normales, coordonnées de textures etc., afin de pouvoir dessiner l'Object3D

Notre Object3dDynamic et stocker dans une classe GLWidget c'est elle qui mets a jour tout les information, contient la caméra et demande l'affichage.

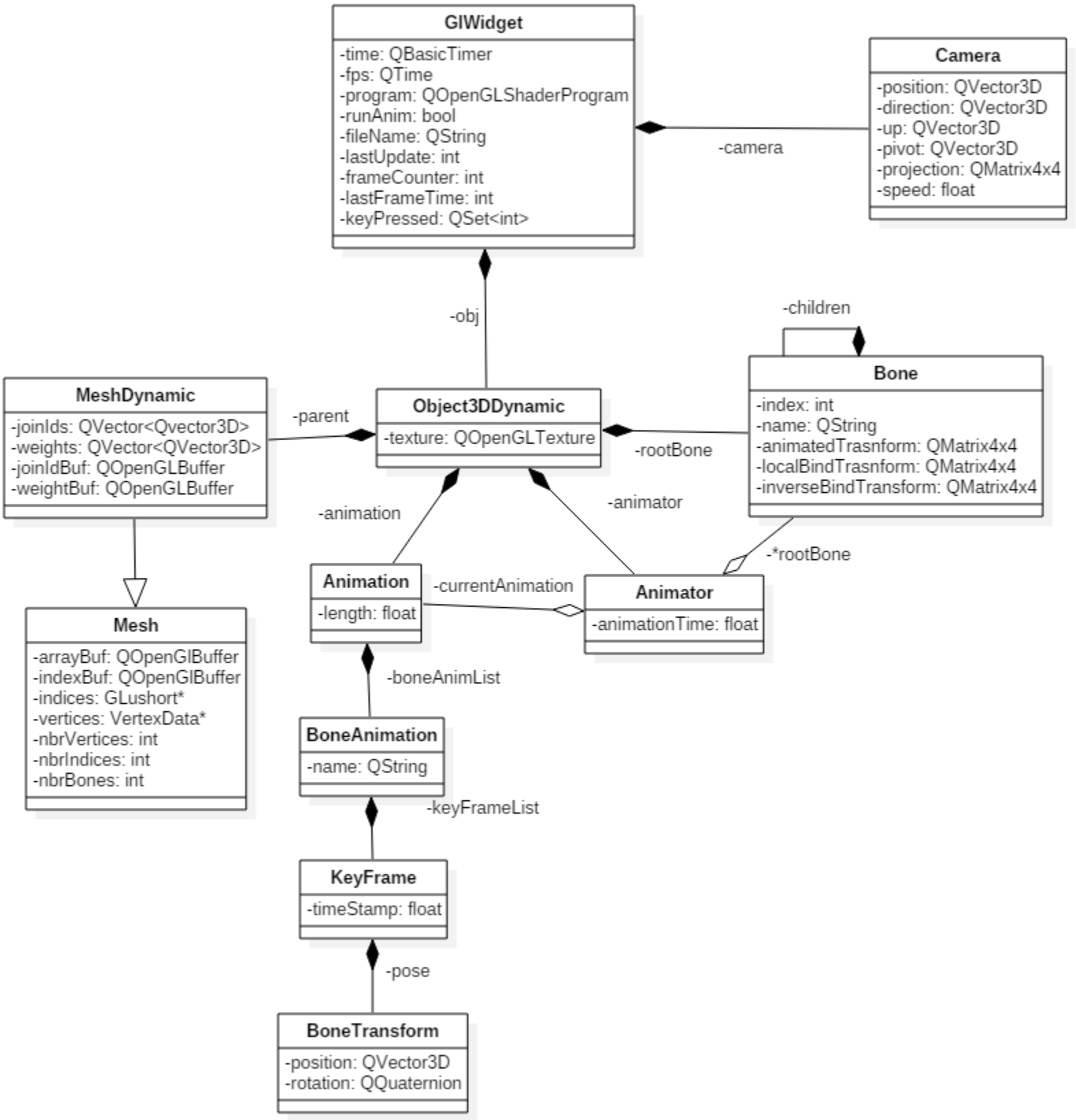
4. Diagramme de classe

i. Structure d'importation



Sur le diagramme ci-dessus on voit bien qu'il y a deux importateurs principaux, celui de la géométrie et du squelette qui retourne un AnimatedModelData et celui de l'animation retournant une AnimationData.

ii. Structure de l'animation



Nous avons choisi de stocker toutes les informations du modèle dans un seul object Object3DDynamic, se choix vient de l'observation faite dans blender ou unity ou un Object et

composer de plusieurs composantes et pas uniquement son mesh. Animation et animator sont séparer de façon a si nous continuons le projet pouvoir importer plusieurs animations, et de pouvoir changer facilement l'animation jouez et modifiant l'animation courante de l'animator.
Enfin la camera peut tourner autour d'un pivot, le centre de la scène par default, elle peut sans approcher et monter se points afin de le remplacer au centre de la géométrie si le modèle n'est pas centré

5. Fonctionnement de l'application

i. Dérouler de l'importation

L'importation se déroule de la façon suivante :

- On commence par importer le modèle, pour se faire on récupère déjà les informations de mapping entre les os et les poids associer, on appellera cela le skin.
  Ces informations sont contenues dans la balise library_controller du fichier colada.

```
<library_controllers>
  <controller id="Armature_Cube-skin" name="Armature">
    <skin source="#Cube-mesh">
      <bind_shape_matrix>1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1</bind_shape_matrix>
      <source id="Armature_Cube-skin-joints">
        <Name_array id="Armature_Cube-skin-joints-array" count="3">Bone1 Bone2 Bone3</Name_array>
        <technique_common>
          <accessor source="#Armature_Cube-skin-joints-array" count="3" stride="1">
            <param name="JOINT" type="name"/>
          </accessor>
        </technique_common>
      </source>
      <source id="Armature_Cube-skin-bind_poses">
        <float_array id="Armature_Cube-skin-bind_poses-array" count="48">1 0 0 0 0 0 1 0 0 -1 0 0 0 0 0 1 1 0 0 0 0 0 1 -1 0 -1 0 0 0 0 0 1 1 0 0 0 0 0 1 -2 0 -1 0 0 0 0 0 1</float_array>
        <technique_common>
          <accessor source="#Armature_Cube-skin-bind_poses-array" count="3" stride="16">
            <param name="TRANSFORM" type="float4x4"/>
          </accessor>
        </technique_common>
      </source>
      <source id="Armature_Cube-skin-weights">
        <float_array id="Armature_Cube-skin-weights-array" count="16">1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1</float_array>
        <technique_common>
          <accessor source="#Armature_Cube-skin-weights-array" count="16" stride="1">
            <param name="WEIGHT" type="float"/>
          </accessor>
        </technique_common>
      </source>
      <joints>
        <input semantic="JOINT" source="#Armature_Cube-skin-joints"/>
        <input semantic="INV_BIND_MATRIX" source="#Armature_Cube-skin-bind_poses"/>
      </joints>
      <vertex_weights count="16">
        <input semantic="JOINT" source="#Armature_Cube-skin-joints" offset="0"/>
        <input semantic="WEIGHT" source="#Armature_Cube-skin-weights" offset="1"/>
        <vcount>1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 </vcount>
        <v>0 0 0 1 0 2 0 3 0 4 0 5 0 6 0 7 1 8 1 9 1 10 1 11 2 12 2 13 2 14 2 15</v>
      </vertex_weights>
    </skin>
  </controller>
</library_controllers>
```

Ensuite on va récupérer le squelette, c'est-à-dire les os et leur hiérarchies ainsi que leurs positions et rotations de départ ceci est contenu dans la balise library_visual_scene, afin de connaitre l'indice de chaque os, on se sert des informations récupérer plus tôt.

```
<library_visual_scenes>
  <visual_scene id="Scene" name="Scene">
    <node id="Camera" name="Camera" type="NODE">
    <node id="Lamp" name="Lamp" type="NODE">
    <node id="Armature" name="Armature" type="NODE">
      <translate sid="location">0 0 0</translate>
      <rotate sid="rotationZ">0 0 1 0</rotate>
      <rotate sid="rotationY">0 1 0 0</rotate>
      <rotate sid="rotationX">1 0 0 0</rotate>
      <scale sid="scale">1 1 1</scale>
      <node id="Bone1" name="Bone1" sid="Bone1" type="JOINT">
        <matrix sid="transform">1 0 0 0 0 0 -1 0 0 1 0 0 0 0 0 1</matrix>
        <node id="Bone2" name="Bone2" sid="Bone2" type="JOINT">
          <matrix sid="transform">1 0 0 0 0 1 0 1 0 0 1 0 0 0 0 1</matrix>
          <node id="Bone3" name="Bone3" sid="Bone3" type="JOINT">
            <matrix sid="transform">1 0 0 0 0 1 0 1 0 0 1 0 0 0 0 1</matrix>
          </node>
        </node>
      </node>
    </node>
    <node id="Cube" name="Cube" type="NODE">
      <translate sid="location">0 0 0</translate>
      <rotate sid="rotationZ">0 0 1 0</rotate>
      <rotate sid="rotationY">0 1 0 0</rotate>
      <rotate sid="rotationX">1 0 0 0</rotate>
      <scale sid="scale">1 1 1</scale>
      <instance_controller url="#Armature_Cube-skin">
        <skeleton>#Bone1</skeleton>
        <bind_material>
          <technique_common>
            <instance_material symbol="Material-material" target="#Material-material">
              <bind_vertex_input semantic="UVMap" input_semantic="TEXCOORD" input_set="0"/>
            </instance_material>
          </technique_common>
        </bind_material>
      </instance_controller>
    </node>
  </visual_scene>
```

Avec toutes les informations actuelles, skin et squelette on va pouvoir créer le MeshData en récupérant également les données contenues dans library_geomerty. On y trouve la liste des points, la liste des normales et la liste des coordonnées de textures du mesh. Associé aux informations du skin et du squelette on va pouvoir créer une liste de Vertex où chaque vertex contient un point, une normal, une coordonnée de texture, des os qui l'influencent et les poids associés.

```
<library_geometries>
  <geometry id="Cube-mesh" name="Cube">
    <mesh>
      <source id="Cube-mesh-positions">
        <float_array id="Cube-mesh-positions-array" count="48">0.1999999 0.1999998 -1 0.1999999 -0.2
        <technique_common>
          <accessor source="#Cube-mesh-positions-array" count="16" stride="3">
            <param name="X" type="float"/>
            <param name="Y" type="float"/>
            <param name="Z" type="float"/>
          </accessor>
        </technique_common>
      </source>
      <source id="Cube-mesh-normals">
        <float_array id="Cube-mesh-normals-array" count="42">0 0 -1 2.98023e-7 1 0 1 0 0 0 -1 0 -1 1
        <technique_common>
          <accessor source="#Cube-mesh-normals-array" count="14" stride="3">
            <param name="X" type="float"/>
            <param name="Y" type="float"/>
            <param name="Z" type="float"/>
          </accessor>
        </technique_common>
      </source>
      <source id="Cube-mesh-map-0">
        <float_array id="Cube-mesh-map-0-array" count="168">0.6024198 0.4202655 0.5790901 0.3969358
        <technique_common>
          <accessor source="#Cube-mesh-map-0-array" count="84" stride="2">
            <param name="S" type="float"/>
            <param name="T" type="float"/>
          </accessor>
        </technique_common>
      </source>
      <vertices id="Cube-mesh-vertices">
        <input semantic="POSITION" source="#Cube-mesh-positions"/>
      </vertices>
      <polylist material="Material-material" count="28">
        <input semantic="VERTEX" source="#Cube-mesh-vertices" offset="0"/>
        <input semantic="NORMAL" source="#Cube-mesh-normals" offset="1"/>
        <input semantic="TEXCOORD" source="#Cube-mesh-map-0" offset="2" set="0"/>
        <vcount>3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 </vcount>
        <p>0 0 0 2 0 1 3 0 2 7 1 3 8 1 4 4 1 5 4 2 6 1 2 7 0 2 8 5 3 9 2 3 10 1 3 11 6 4 12 3 4 13 2
      </polylist>
    </mesh>
  </geometry>
```

Enfin on peut récupérer la texture dans la balise library_images afin de pouvoir afficher un modèle colorer.

```
<library_images>
  <image id="Color_png" name="Color_png">
    <init_from>Color.png</init_from>
  </image>
</library_images>
```

- Pour l'animation on va récupérer dans library_animation différents temps, des key frames, pour chaque os ainsi que la transformation de l'os à ces différents temps

```
<source id="Armature_Bone1_pose_matrix-input">
  <float_array id="Armature_Bone1_pose_matrix-input-array" count="6">0.04166662 0.8333333 1.666667 2.291667 2.5 2.916667</float_array>
  <technique_common>
    <accessor source="#Armature_Bone1_pose_matrix-input-array" count="6" stride="1">
      <param name="TIME" type="float"/>
    </accessor>
  </technique_common>
</source>
<source id="Armature_Bone1_pose_matrix-output">
  <float_array id="Armature_Bone1_pose_matrix-output-array" count="96">1 0 0 0 0 0 -1 0 0 1 0 0 0 0 0 1 1 0 0 0 0 0 -1 0 0 1 0 1 0 0 0
  <technique_common>
    <accessor source="#Armature_Bone1_pose_matrix-output-array" count="6" stride="16">
      <param name="TRANSFORM" type="float4x4"/>
    </accessor>
  </technique_common>
</source>
```

ii. Dérouler de l'animation

L'animation se déroule en deux étapes majeur la première est le fonctionnement de l'animator, la seconde se trouve dans le shader.

a. L'animator

Il s'agit de la classe qui va lire l'animation et mettre à jour le os. Ainsi afin d'animer notre modèle à chaque image, soit à peut prêt toutes les 16 ms, on va mettre à jour le temps de notre animator en ajoutant le temps écouler depuis le dernier appelle au temp actuel de l'animation. Si se temps dépasse la durée de l'animation on redémarre à 0 plus le trop afin de garder une animation fluide.

Une fois se temps mit à jour on va rechercher les deux frames entourant le temps actuel de l'animation, et calculer à qu'elle pourcentage entre les deux nous nous trouvons, 0 étant la première frames et 1 étant la deuxième. Grace à se pourcentage on va pouvoir interpoler la position et la rotation actuel entre les deux frames pour chaque os.

Une fois ces transformations calculer et stocker, il va falloir les appliquer au squelette.
Pour ce faire on démarre depuis l'os racine, en effet chaque os peut avoir des enfants, si le père bouge les enfants subissent la même transformation.
On calcul sa transformation actuelle depuis l'origine du monde qui est la transformation d'animation calculer plus tôt multiplié par la transformation de son parent, si c'est l'os de départ il s'agit de la matrice identité.
Puis on rappelle cette fonction récursive sur chacun de ces fils en envoyant la matrice calculer à l'instant. Enfin on multiplie cette matrice par la transformation inverse par default de l'os afin d'obtenir la matrice de transformation de l'os depuis sa position initial puis on passe cette matrice a l'os pour qu'il la sauvegarde.

```
void Animator::applyPoseToJoints(map<QString, QMatrix4x4> currentPose, Bone* bone, QMatrix4x4 parentTransform){
    QMatrix4x4 currentLocalTransform = currentPose[bone->getName()];
    QMatrix4x4 currentTransform = parentTransform * currentLocalTransform;
    for (Bone* childBone : bone->getChildren()) {
        applyPoseToJoints(currentPose, childBone, currentTransform);
    }
    currentTransform = currentTransform * bone->getInverseBindTransform();
    bone->setAnimationTransform(currentTransform);
}
```

Une fois tous les os à jour. On récupère leurs matrices d'animation pour les envoyer au vertex shader, on envoi également pour chaque vertices les position, normal, etc.

b. Le vertex shader

Dans le vertex shader on va calculer la nouvelle position du vertex pour se faire on récupère les trois indices des 3 os qui lui sont associés ainsi que leur matrice et pour chaque os on va calculer sa matrice fois la position du vertex fois le poids de l'os, un vertex peut être influence par maximum trois os et le total des poids doit faire un. On ajoute les trois vecteurs obtenus afin d'avoir la nouvelle position du vertex, on fait la même chose avec les normales.

```
const int MAX_JOINTS = 50;//max joints allowed in a skeleton
const int MAX_WEIGHTS = 3;//max number of joints that can affect a vertex

//VertexData
in vec3 in_position;
in vec3 in_boneIndices;
in vec3 in_weights;
in vec3 in_normal;
in vec2 in_textureCoords;

//what is pass to the fragment Shader
out vec2 pass_textureCoords;
out vec3 pass_normal;

//uniform data
uniform mat4 boneTransforms[MAX_JOINTS];
uniform mat4 mvp;

void main(void){

        vec4 totalLocalPos = vec4(0.0);
        vec4 totalNormal = vec4(0.0);

        for(int i=0;i<MAX_WEIGHTS;i++){
            mat4 boneTransform = boneTransforms[int(in_boneIndices[i])];
            vec4 posePosition = boneTransform * vec4(in_position, 1.0);
            totalLocalPos += posePosition * in_weights[i];

            vec4 worldNormal = boneTransform * vec4(in_normal.xyz, 1.0);
            totalNormal += worldNormal * in_weights[i];
        }

        gl_Position = mvp * vec4(totalLocalPos.xyz,1);
        pass_normal = totalNormal.xyz;
        pass_textureCoords = in_textureCoords;
}
```

iii. Mode d'emploi

Tourner autour de l'origine : flèche directionnelle
Monter : Shift gauche
Descendre : Ctrl Droit
Avancer : Z
Reculer : S
Lancer/pause l'animation : P

Les touches sont testées à chaque frames, ainsi parfois l'animation et lancer et s'arrête de suite car p est rester appuyer a la suivante, dans ce cas réappuyez sur p

6. Bilan

i. Difficulté rencontrée

Plusieurs difficultés se sont présentées à nous durant le projet :

- La prise en main de blender a demander un certains temps d'adaptation.
- L'importation à demander beaucoup de temps à réaliser par le nombre de donnée à récupérer
- Beaucoup de temps a été perdu pour tenter de débuguer l'animation que se soit coté shader ou c++.

ii. Limites du projet rendu et amélioration possible

Actuellement aucune sécurité et gestion d'erreur existe ainsi un modèle charger sans texture ne seras pas visible ou l'appli peut crasher sans dire pourquoi si on charge le mauvais modèle.

De plus l'animation existe mais reste boguer, nous savons que cela vient de nos matrices mais nous n'arrivons pas à trouver où.

Un bug existe et vient sans doute de la version de blender utiliser, en effet un node xml et appelé dans un cas polylist et dans l'autre triangles, se sont les même nodes mais nous gérons que le premier cas. Si l'application crash essayer d'exporter votre modèle avec blender 2.77.

iii. Conclusion

Ce projet nous auras permit de découvrir la programmation avec OpenGl, voir les modèles apparaitre et s'animer auras permit de rendre le projet motivant. Même si l'application final bug nous sommes content de notre résultat et essayeront de la corriger après le rendu.