INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

Informatique Graphique - TP 4

Animation procédurale

GRAINDORGE Amance

VERNET Hector

Université de Rennes

I. INTRODUCTION

Au cours de ce TP, nous nous attelerons à compléter et étendre le code d'animation présent dans la pipeline fournie. Ce code aura pour but de pouvoir définir des animations de modèles à partir de *keyframes* (ou images clés) représentant la transformation d'un objet à un point dans le temps. Une fois que les fonctions de base fonctionneront, nous pourrons les améliorer pour notre projet avec des fonctionnalités telles que l'importation depuis un logiciel d'animation, et d'autres méthodes d'interpolation.

II. EXERCICE 1

Dans cet exercice, nous allons tout d'abord finir la fonction `interpolateTransformation()` de la classe `KeyframeCollection` qui a pour but de donner la transformation d'un objet à tout instant t, en prenant en compte les deux keyframes l'entourant.

Voici le code de la fonction terminée :

```
1   // Handle the case where the time parameter is outside the keyframes time scope.
2   std::map<float, GeometricTransformation>::const_iterator itFirstFrame =
    m_keyframes.begin();
3   std::map<float, GeometricTransformation>::const_reverse_iterator itLastFrame =
    m_keyframes.rbegin();
4   float effective_time = std::fmod(time, itLastFrame->first);
5
6   // Get keyframes surrounding the time parameter
7   std::array<Keyframe, 2> result = getBoundingKeyframes(effective_time);
8
9   float factor = (time - result[0].first) / (result[1].first - result[0].first);
10
11  glm::vec3 interpTranslation = glm::lerp(result[0].second.getTranslation(),
    result[1].second.getTranslation(), factor);
12  glm::vec3 interpScale = glm::lerp(result[0].second.getScale(),
    result[1].second.getScale(), factor);
13  glm::quat interpOrientation =
    glm::slerp(glm::normalize(result[0].second.getOrientation()),
    glm::normalize(result[1].second.getOrientation()), factor);
14  glm::mat4 iMatrix(1.0f);
15
16  iMatrix = glm::translate(iMatrix, interpTranslation);
17  iMatrix *= glm::toMat4(interpOrientation);
18  iMatrix = glm::scale(iMatrix, interpScale);
19  iMatrix = glm::scale(iMatrix, interpScale);
20
21  return iMatrix;
```

On interpole chaque composante de la transformation séparément avant de les combiner, afin de conserver la forme du modèle.

Une fois que cette fonction marche, on peut s'en servir dans le fichier `practical4.cpp`. On commence par la fonction `movingCylinder` :

```
1   // Add shader
2   ShaderProgramPtr flatShader = std::make_shared<ShaderProgram>(
3       "../../sfmlGraphicsPipeline/shaders/flatVertex.glsl",
4       "../../sfmlGraphicsPipeline/shaders/flatFragment.glsl"
```

```
    );
    viewer.addShaderProgram(flatShader);

    // Frame
    FrameRenderablePtr frame = std::make_shared<FrameRenderable>(flatShader);
    viewer.addRenderable(frame);

    // Animated cylinder
    auto cylinder = std::make_shared<CylinderMeshRenderable>(
        flatShader,
        false,
        20,
        false
    );
    cylinder->setGlobalTransform(glm::mat4(1.0));

    cylinder->addGlobalTransformKeyframe(
        GeometricTransformation(
            glm::vec3(0.0f, 0.0f, 0.0f),
            glm::quat(),
            glm::vec3(1.0f, 1.0f, 1.0f)
        ),
        0.0f
    );
    cylinder->addGlobalTransformKeyframe(
        GeometricTransformation(
            glm::vec3(0.0f, 0.0f, 0.0f),
            glm::angleAxis(M_PIf * 1.5f,
            glm::vec3(0.0f, 1.0f, 0.0f)),
            glm::vec3(1.0f, 1.0f, 1.0f)
        ),
        5.0f
    );

    cylinder->addLocalTransformKeyframe(
        GeometricTransformation(
            glm::vec3(0.0f, 0.0f, -5.0f),
            glm::quat(),
            glm::vec3(1.0f, 1.0f, 1.0f)
        ),
        0.0f
    );
    cylinder->addLocalTransformKeyframe(
        GeometricTransformation(
            glm::vec3(0.0f, 0.0f, -5.0f),
            glm::angleAxis(M_PIf * 1.5f,
            glm::vec3(0.0f, 0.0f, 1.0f)),
            glm::vec3(1.0f, 1.0f, 1.0f)
        ),
        5.0f
    );

    viewer.addRenderable(cylinder);
    viewer.startAnimation();
```

En plaçant des keyframes à t=0 et 5 secondes, on définit un cylindre qui fait une rotation de 270° autour de l'axe Y global, et une rotation de 270° autour de son axe local Z en 5 secondes (pour donner l'impression qu'il roule).

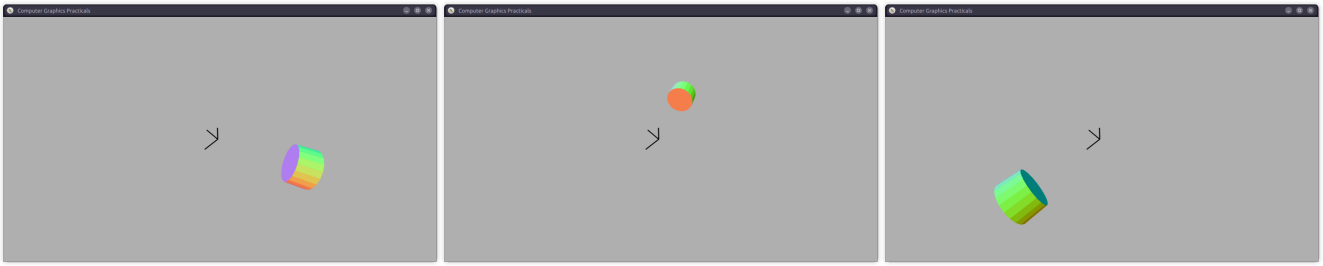

Figure 1 : Déplacement du cylindre entre les keyframes

Ensuite, on s'occupe de la fonction `movingTree`, qui doit générer un arbre animé. Pour cela, on modifie la fonction `creer_branche` que nous avions écrite pour le TP précédent :

```
1   void creer_branche(
2       int level,
3       HierarchicalRenderablePtr parent,
4       ShaderProgramPtr shaderProgram,
5       float lengthMultiplier,
6       float angle,
7       int branching_branches,
8       float thickness
9   )
10  {
11      if (level == 0) { return; }
12      for (int i = 0; i < branching_branches; ++i)
13      {
14          // Create child
15          auto child = std::make_shared<CylinderMeshRenderable>(
16              shaderProgram,
17              false,
18              20u,
19              false
20          );
21          // Set global transform
22          child->setGlobalTransform(glm::mat4(1.0));
23
24          glm::mat4 childGlobalTransform;
25          childGlobalTransform *= getTranslationMatrix(0.0, 1.0, 0.0);
26          childGlobalTransform *= getRotationMatrix(
27              glm::radians(angle + i * (360.0f / branching_branches)),
28              glm::vec3(0.0, 1.0, 0.0)
29          );
30          childGlobalTransform *= getRotationMatrix(
31              glm::radians(angle),
32              glm::vec3(1.0, 0.0, 0.0)
33          );
34          childGlobalTransform *= getScaleMatrix(
35              lengthMultiplier,
36              lengthMultiplier,
37              lengthMultiplier
38          );
39
```

```
40            childGlobalTransform *= getRotationMatrix(
41                glm::radians(10.0f),
42                glm::vec3(1.0, 0.0, 0.0)
43            );
44            child->addGlobalTransformKeyframe(childGlobalTransform, 0.0f);
45            child->addGlobalTransformKeyframe(childGlobalTransform, 5.0f);
46
47            childGlobalTransform *= getRotationMatrix(
48                glm::radians(-10.0f),
49                glm::vec3(1.0, 0.0, 0.0)
50            );
51            child->addGlobalTransformKeyframe(childGlobalTransform, 2.5f);
52
53            // Set local transform
54            glm::mat4 childLocalTransform;
55            childLocalTransform *= getScaleMatrix(thickness, 1.0, thickness);
56            childLocalTransform *= getRotationMatrix(
57                glm::radians(-90.0f),
58                glm::vec3(1.0, 0.0, 0.0)
59            );
60            child->setLocalTransform(childLocalTransform);
61            // Define parent / children relationships
62            HierarchicalRenderable::addChild(parent, child);
63            // Recursive call
64            creer_branche(
65                level - 1, child,
66                shaderProgram, lengthMultiplier,
67                angle, branching_branches,
68                thickness
69            );
70        }
71    }
72
73    void movingTree(Viewer &viewer)
74    {
75        // Add shader
76        ShaderProgramPtr flatShader = std::make_shared<ShaderProgram>(
77            "../../sfmlGraphicsPipeline/shaders/flatVertex.glsl",
78            "../../sfmlGraphicsPipeline/shaders/flatFragment.glsl"
79        );
80        viewer.addShaderProgram(flatShader);
81
82        // Frame
83        FrameRenderablePtr frame = std::make_shared<FrameRenderable>(flatShader);
84        viewer.addRenderable(frame);
85
86        std::shared_ptr<CylinderMeshRenderable> root =
87            std::make_shared<CylinderMeshRenderable>(
88                flatShader,
89                false,
90                20u,
91                false
92            );
93
94        glm::mat4 rootLocalTransform;
95        rootLocalTransform *= getScaleMatrix(0.15, 1.0, 0.15);
```

```
96        rootLocalTransform *= getRotationMatrix(
97            glm::radians(-90.0f),
98            glm::vec3(1.0, 0.0, 0.0)
99        );
100       root->setLocalTransform(rootLocalTransform);
101
102       creer_branche(5, root, flatShader, 0.75f, 35.0f, 4, 0.15);
103       viewer.addRenderable(root);
104       viewer.startAnimation();
105   }
```

Cela a pour résultat de faire bouger les branches de haut en bas (ce qui correspond à une rotation autour de l'axe X des branches). On peut jouer l'animation en boucle avec `viewer.setAnimationLoop(true, 5.0)`.

Figure 2 : Mouvement de l'arbre entre les keyframes

III. EXERCICE 2

À présent, nous allons voir des améliorations possibles à apporter au système d'animation au delà de ce qui est suggéré.

III.1. Interpolation

Actuellement, nous utilisons une interpolation linéaire pour les transformations. Cependant, il existe d'autres méthodes qui peuvent donner des animations plus fluides pour le même nombre de keyframes.

En se basant sur https://graphicscompendium.com/opengl/22-interpolation, nous allons donc implémenter l'interpolation cubique, pour remplacer la fonction `glm::lerp`. Nous allons aussi changer le modèle de données des keyframes, afin de stocker le type d'interpolation.

```
1   float cubicInterpolate(
2       float x0, float x1,
3       float x2, float x3,
4       float t)
5   {
6       float a = (3.0 * x1 - 3.0 * x2 + x3 - x0) / 2.0;
7       float b = (2.0 * x0 - 5.0 * x1 + 4.0 * x2 - x3) / 2.0;
8       float c = (x2 - x0) / 2.0;
9       float d = x1;
10
11      return a * t * t * t +
12             b * t * t +
13             c * t +
14             d;
15  }
```

Ensuite, pour appeler cette fonction, on fait un switch sur le type d'interpolation stocké dans la keyframe :

```
// Handle the case where the time parameter is outside the keyframes time scope.
std::map<float, Keyframe>::const_iterator first = m_keyframes.begin();
std::map<float, Keyframe>::const_iterator last = std::prev(m_keyframes.end());
if (time <= first->first) {
    return first->second.transform.toMatrix();
} else if (time >= last->first) {
    return last->second.transform.toMatrix();
}

glm::mat4 iMatrix(1.0f);

std::map<float, Keyframe>::const_iterator ki2 = m_keyframes.upper_bound(time);
std::map<float, Keyframe>::const_iterator ki1 = std::prev(ki2);

float factor = (time - ki1->first) / (ki2->first - ki1->first);

switch (ki1->second.interpolation)
{
    case CUBIC: {
        std::map<float, Keyframe>::const_iterator first = m_keyframes.begin();
        std::map<float, Keyframe>::const_iterator last = m_keyframes.end();

        std::map<float, Keyframe>::const_iterator ki0;
        std::map<float, Keyframe>::const_iterator ki3 = std::next(ki2);
        if (ki1 == first) {
            ki0 = ki1;
        } else {
            ki0 = std::prev(ki1);
        }
        if (ki3 == last) {
            ki3 = ki2;
        }
        GeometricTransformation g0 = ki0->second.transform;
        GeometricTransformation g1 = ki1->second.transform;
        GeometricTransformation g2 = ki2->second.transform;
        GeometricTransformation g3 = ki3->second.transform;
        glm::vec3 t0 = g0.getTranslation();
        glm::vec3 t1 = g1.getTranslation();
        glm::vec3 t2 = g2.getTranslation();
        glm::vec3 t3 = g3.getTranslation();
        glm::vec3 s0 = g0.getScale();
        glm::vec3 s1 = g1.getScale();
        glm::vec3 s2 = g2.getScale();
        glm::vec3 s3 = g3.getScale();
        glm::quat r0 = glm::normalize(g0.getOrientation());
        glm::quat r1 = glm::normalize(g1.getOrientation());
        glm::quat r2 = glm::normalize(g2.getOrientation());
        glm::quat r3 = glm::normalize(g3.getOrientation());
        glm::vec3 interpTranslation;
        glm::vec3 interpScale;
        glm::quat interpOrientation;
        for (int i = 0; i < 3; i++) {
            interpTranslation[i] = cubicInterpolate(
```

```
                    t0[i], t1[i], t2[i], t3[i],
                    factor
                );
                interpScale[i] = cubicInterpolate(
                    s0[i], s1[i], s2[i], s3[i],
                    factor
                );
            }
            for (int i = 0; i < 4; i++) {
                interpOrientation[i] = cubicInterpolate(
                    r0[i], r1[i], r2[i], r3[i],
                    factor);
            }
            interpOrientation = glm::normalize(interpOrientation);

            iMatrix = glm::translate(iMatrix, interpTranslation);
            iMatrix *= glm::toMat4(interpOrientation);
            iMatrix = glm::scale(iMatrix, interpScale);

            break;
        }
        case LINEAR: {
            Keyframe k1 = ki1->second;
            Keyframe k2 = ki2->second;

            glm::vec3 interpTranslation = glm::lerp(
                k1.transform.getTranslation(),
                k2.transform.getTranslation(),
                factor
            );
            glm::vec3 interpScale = glm::lerp(
                k1.transform.getScale(),
                k2.transform.getScale(),
                factor
            );
            glm::quat interpOrientation = glm::slerp(
                glm::normalize(k1.transform.getOrientation()),
                glm::normalize(k2.transform.getOrientation()),
                factor
            );

            iMatrix = glm::translate(iMatrix, interpTranslation);
            iMatrix *= glm::toMat4(interpOrientation);
            iMatrix = glm::scale(iMatrix, interpScale);

            break;
        }
        case CONSTANT: {
            iMatrix = ki1->second.transform.toMatrix();

            break;
        }
    }

    return iMatrix;
```

On a aussi ajouté le mode d'interpolation CONSTANT, qui ne correspond à aucune interpolation, et qui est donc trivial à implémenter.

III.2. Importation d'animation depuis Blender

Notre système d'animation a désormais le minimum syndical de fonctionnalités pour produire une animation correcte. Cependant, entrer les valeurs des keyframes à la main est une technique qui était utilisée dans les années 80, mais des outils bien plus faciles d'utilisations existent aujourd'hui, et nous permettraient de gagner beaucoup de temps.

Pour cela, nous utiliserons Blender, dont nous nous servons déjà pour la modélisation de nos objets. Nous exportons les modèles au format Wavefront, qui est assez basique et ne contient que des données sur le mesh. Pour l'animation, les formats la supportant sont plus complexes et donc plus difficiles à implémenter. Nous créerons donc notre propre format.

Pour gérer tout cela, vu que Blender a une API Python très complète (`bpy`), nous allons écrire un script qui pourra exporter tous les objets d'une scène ainsi que leur animation. Le voici :

```python
# pyright: reportOptionalMemberAccess=false

import bpy
import mathutils

abs_path = bpy.path.abspath("//")
export_path = f"{abs_path}/../sampleProject/ObjFiles"

def frame_to_time(frame: int):
    return (frame - bpy.context.scene.frame_start) / bpy.context.scene.render.fps

def export_mesh(obj):
    for ob in bpy.data.objects:
        ob.select_set(False)
    obj.select_set(True)
    backup_matrix = obj.matrix_world.copy()
    obj.matrix_world = mathutils.Matrix.Identity(4)
    bpy.ops.wm.obj_export(
        filepath=f"{export_path}/{obj.name}.obj",
        apply_modifiers=True,
        export_selected_objects=True,
        export_materials=False
    )
    y_up_rot = mathutils.Matrix((
        (1, 0, 0, 0),
        (0, 0, 1, 0),
        (0, -1, 0, 0),
        (0, 0, 0, 1)
    ))
    y_up_matrix = y_up_rot @ backup_matrix @ y_up_rot.inverted()
    with open(f"{export_path}/{obj.name}.obj", "a") as f:
        f.write("\nTRANSFORM")
        for row in y_up_matrix:
            for val in row:
                f.write(f" {val}")
        f.write("\n")
    obj.matrix_world = backup_matrix
    obj.select_set(False)
```

```python

def export_animation(obj):
    output = ""
    if obj.animation_data and obj.animation_data.action:
        frames = {}
        for fcurve in obj.animation_data.action.fcurves:
            for keyframe in fcurve.keyframe_points:
                if int(keyframe.co.x) not in frames:
                    frames[int(keyframe.co.x)] = [keyframe.interpolation]
                else:
                    frames[int(keyframe.co.x)].append(keyframe.interpolation)
        scene = bpy.context.scene
        depsgraph = bpy.context.evaluated_depsgraph_get()
        for fr in sorted(frames.keys()):
            scene.frame_set(fr)
            eval_obj = obj.evaluated_get(depsgraph)
            loc, rot, scale = eval_obj.matrix_world.decompose()
            t = frame_to_time(fr)
            most_frequent_interp = max(set(frames[fr]), key=frames[fr].count)
            output += f"{t},{most_frequent_interp},{loc.x},{loc.y},{loc.z},{rot.x},
{rot.y},{rot.z},{rot.w},{scale.x},{scale.y},{scale.z}\n"
        if output != "":
            with open(f"{export_path}/{obj.name}.animation", "w") as f:
                f.write(output)

if bpy.context.scene is not None:
    for obj in bpy.data.objects:
        export_mesh(obj)
        export_animation(obj)
```

Ce script rajoute également une propriété `TRANSFORM` au fichier `.obj`, qui permet de partager la transformation globale appliquée aux objets dans Blender. En effet, normalement, le format Wavefront n'inclut que la position absolue des points après transformation.

Après exécution, on se retrouve avec un fichier `.obj` et un fichier `.animation` par objet. Pour parser ces fichiers, nous écrivons la méthode suivante :

```cpp
void KeyframeCollection::addFromFile(
    const std::string &animation_filename,
    float time_shift
) {
    std::ifstream is(animation_filename);
    std::string str;
    while (getline(is, str))
    {
        std::regex regexz(",");
        std::vector<std::string> split(
            std::sregex_token_iterator(
                str.begin(),
                str.end(),
                regexz, -1
            ),
            std::sregex_token_iterator()
        );

```

```
19            // Don't forget to convert to Y-up !
20            glm::vec3 loc = glm::vec3(
21                std::stof(split[2]),
22                std::stof(split[4]),
23                -std::stof(split[3])
24            );
25            glm::vec3 size = glm::vec3(
26                std::stof(split[9]),
27                std::stof(split[11]),
28                std::stof(split[10])
29            );
30            glm::quat rot = glm::quat(
31                std::stof(split[8]),
32                std::stof(split[5]),
33                std::stof(split[7]),
34                -std::stof(split[6])
35            );
36
37            // Default interpolation is cubic
38            KeyframeInterpolationMode interp = CUBIC;
39            if (split[1] == "LINEAR") {
40                interp = LINEAR;
41            } else if (split[1] == "CONSTANT") {
42                interp = CONSTANT;
43            } else if (split[1] == "CUBIC") {
44                interp = CUBIC;
45            }
46
47            this->add(
48                GeometricTransformation(loc, glm::normalize(rot), size),
49                std::stof(split[0]) + time_shift, interp);
50        }
51    }
```

Nous avons du apporter des modifications à d'autres fichiers, notamment pour gérer les nouvelles données de transformation dans le `.obj`, et pour stocker une matrice inverse lorsqu'on rajoute un parent à un `HierarchicalRenderable`, afin d'avoir une hiérarchie de transformations fonctionnelle. Mais ce rapport est déjà assez long, nous n'irons pas dans les détails. Vous pouvez aller voir ces deux commits :

- https://github.com/pixup1/tpinfographique/commit/8d7807f03bac0b80625253ebc1c0289da87e0284
- https://github.com/pixup1/tpinfographique/commit/d42c34f690a0589d573c4d8046d1d9b112fae514

IV. CONCLUSION

En conclusion, lors de ce TP, nous avons implémenté un système d'animation procédurale basé sur des keyframes, avec différentes méthodes d'interpolation, et un outil d'exportation depuis Blender. Cela nous permettra de réaliser notre projet plus rapidement et avec des animations moins rudimentaires.