

# Rapport d'Informatique Graphique

*La Famille Tortue*

Loris Thibaud, Maxime Garibaldi

December 15, 2025

# Contents

<b>1</b>	<b>Préproduction</b>	<b>4</b>
1.1	Prémises . . . . .	4
1.2	Cover . . . . .	4
1.3	Concept . . . . .	4
1.4	Storyboard . . . . .	5
1.5	l'intrigue . . . . .	5
<b>2</b>	<b>Les acteurs</b>	<b>6</b>
2.1	Les tortues . . . . .	6
2.2	La souris . . . . .	6
2.3	Le pingouin . . . . .	7
<b>3</b>	<b>Un grand voyage</b>	<b>8</b>
3.1	Match cut . . . . .	8
3.2	Les scènes 3D . . . . .	8
3.3	La Plaine . . . . .	8
3.4	Le Puits . . . . .	8
3.5	La Forêt . . . . .	9
3.6	L'Arctique . . . . .	9
3.7	La Ville . . . . .	10
3.8	Le Volcan . . . . .	10
3.9	L'INSA . . . . .	11
3.10	La Weird Zone . . . . .	11
3.11	Les Egouts . . . . .	12
<b>4</b>	<b>Accepter les matériaux et les textures</b>	<b>12</b>
4.1	Problématique . . . . .	12
4.2	Code . . . . .	12
4.3	Application . . . . .	13
<b>5</b>	<b>Le cycle d'animation de la tortue</b>	<b>13</b>
5.1	Problématique . . . . .	13
5.2	Attentes . . . . .	13
5.3	Nos itérations . . . . .	14
<b>6</b>	<b>Defying Gravity</b>	<b>16</b>
<b>7</b>	<b>Nos fonds</b>	<b>17</b>
<b>8</b>	<b>Les Lumières</b>	<b>18</b>
<b>9</b>	<b>La caméra</b>	<b>21</b>
9.1	Plan fixe . . . . .	21
9.2	Plan en mouvement . . . . .	21
<b>10</b>	<b>Mission Vertigo</b>	<b>22</b>
10.1	Point de départ . . . . .	22
10.2	Théorie et histoire . . . . .	22
10.3	Application sur le projet . . . . .	23
10.4	Implémentation du système de keyframe pour notre focale . . . . .	23

10.5 Implémentation dans la caméra . . . . .	24
10.6 Mission terminée ! . . . . .	25
<b>11 Nos shaders</b>	<b>26</b>
11.1 Point de départ . . . . .	26
11.2 Le cel-shading . . . . .	27
11.3 L'effet papier (bruit) . . . . .	27
11.4 Les contours . . . . .	28
11.4.1 Contour noir . . . . .	28
11.4.2 Contour complémentaire . . . . .	28
11.5 Application de nos effets . . . . .	29
<b>12 Le montage</b>	<b>30</b>
12.1 Render . . . . .	30
12.2 Recadrage . . . . .	30
12.3 Sound effects . . . . .	30
12.4 Crédits . . . . .	30
<b>13 Conclusion</b>	<b>30</b>
<b>14 Bibliographie</b>	<b>30</b>

# 1 Préproduction

## 1.1 Prémisses

Lorsque nous avons entendu le thème de cette année : les tortues, nous avons été frappés par un souvenir, celui de notre enfance et de ses comptines. Plus précisément, nous n'arrivions pas à nous sortir de la tête la comptine de la famille tortue :

Jamais on a vu,  
Jamais on ne verra,  
La famille tortue courir après les rats.  
Le papa tortue et la maman tortue,  
Et les enfants tortue,  
Iront toujours au pas !

## 1.2 Cover

Nous voulions faire, non pas un court-métrage, mais un clip. L'accompagnement en image de la comptine. Mais encore fallait-il trouver une version de la musique à mettre en images... Notre projet avait comme limite minimale 60 secondes, or la comptine en faisait 15. Nous avions peur qu'une musique aussi courte, douce et rébarbative ne finisse par endormir nos spectateurs. Il nous fallait un twist, de quoi surprendre et donner envie aux spectateurs de nous suivre dans cette aventure. Et pour cela, quoi de mieux qu'une cover de la comptine dans un autre style ? Nous avons donc commandé une cover à un de nos amis compositeurs. Malheureusement, faute de temps, il a abandonné le projet très tôt dans la production. Nous nous sommes donc rabattus sur une cover rock réalisé par un youtuber[1] et qui rentrait parfaitement dans nos critères.

## 1.3 Concept

Pour trouver des idées de mise en image, nous avons étudié les paroles de la comptine attentivement. Dans un premier temps, une inspiration punk nous est venue. Nous voulions, en nous inspirant des paroles : "iront toujours au pas", "jamais on ne verra la famille tortue courir après les rats" et les tonalités rock de notre chanson, amener une interprétation cynique et antisystème. Les tortues seraient devenues l'image d'un régime autoritaire qui traquerait les souris, nos opprimés. Pour montrer ce monde, nous comptions nous inspirer des méthodes d'animation du "spider-punk" de Spider-Man : Across the Spider-Verse[2]. Jouer avec des contours, des perturbations de style, des techniques de texturing à base d'arts populaires (Graffiti, Collage, Art Brut). Malheureusement, le projet était trop ambitieux. Un tel objectif en un délai aussi court vu notre niveau préalable était inatteignable.



Figure 1: Inspirations punk pour notre projet

Nous nous sommes rabattus sur une histoire plus simple en se concentrant sur ce que nous pouvions faire sans risque et se concentrant sur l'idée d'offrir un rendu différent et esthétiquement réussi.

#### 1.4 Storyboard

Nous ne sommes pas passés par la case scénario pour ce court-métrage, car nous ne voulions pas distraire les spectateurs du cœur de notre histoire : la chanson. Pour raconter les images plus que l'intrigue, nous avons décidé de travailler à partir d'un storyboard.

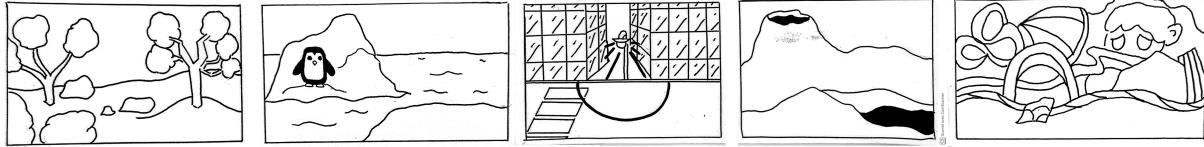


Figure 2: Section du storyboard

Le storyboard est une étape tout aussi primordiale que le scénario dans la préproduction, car ici, nous parlons avec les images.

Nous voulions rendre à la comptine toute son esthétique enfantine. Notre objectif était de donner une impression d'illustration pour enfant prenant vie. Pour cela, nous devions avoir des décors simplifiés, des personnages mignons et surtout un rendu qui donnait un effet papier.

#### 1.5 l'intrigue

Notre intrigue, elle aussi, se baserait sur la simplicité et l'inspiration enfantine. Pourquoi la famille tortue se trouverait-elle à courir après les rats ? Ils vont toujours au pas, mais où ces pas les mèneront ?

### Synopsis

Alors que la famille tortue gambade dans la forêt, elle tombe nez à nez avec une magnifique pomme, à la douceur et à la fraîcheur subtile, mais voilà que la pomme tombe et est emportée aussitôt par un rat. La famille tortue, ni une, ni de, par dans une aventure à travers le monde pour s'emparer du fruit défendu.

## 2 Les acteurs

Les principaux éléments 3D de notre projet sont bien évidemment les divers personnages qui apparaissent durant le court-métrage. Ils sont au nombre de trois.

### 2.1 Les tortues

Nos tortues, d'abord, ces 4 membres d'une même famille, seront présentes tout le long de notre court métrage. Nous avons choisi de créer un seul modèle unique pour gagner du temps dans la conception. Nous avons ensuite joué sur le scale pour différencier les adultes et les enfants.



Figure 3: modèle 3D tortue

### 2.2 La souris

Ensuite, nous avons notre souris qui possède deux modèles. Le premier est utilisé lorsque nous la retrouvons dans notre première scène en train de voler la pomme tombant de l'arbre. Le deuxième apparaît à la fin, lorsque les tortues la retrouvent endormie à côté de la pomme entamée.



(a) modèle 3D souris



(b) modèle 3D souris dodo

### 2.3 Le pingouin

Et pour finir, afin de faire honneur à la tradition en option IG, nous avons décidé de créer un pingouin. Nous le faisons apparaître sur un iceberg dans le fond de notre scène de l'Arctique. C'est une référence métà au projet d'informatique graphique, mais le pingouin rentre parfaitement dans l'univers de notre projet.

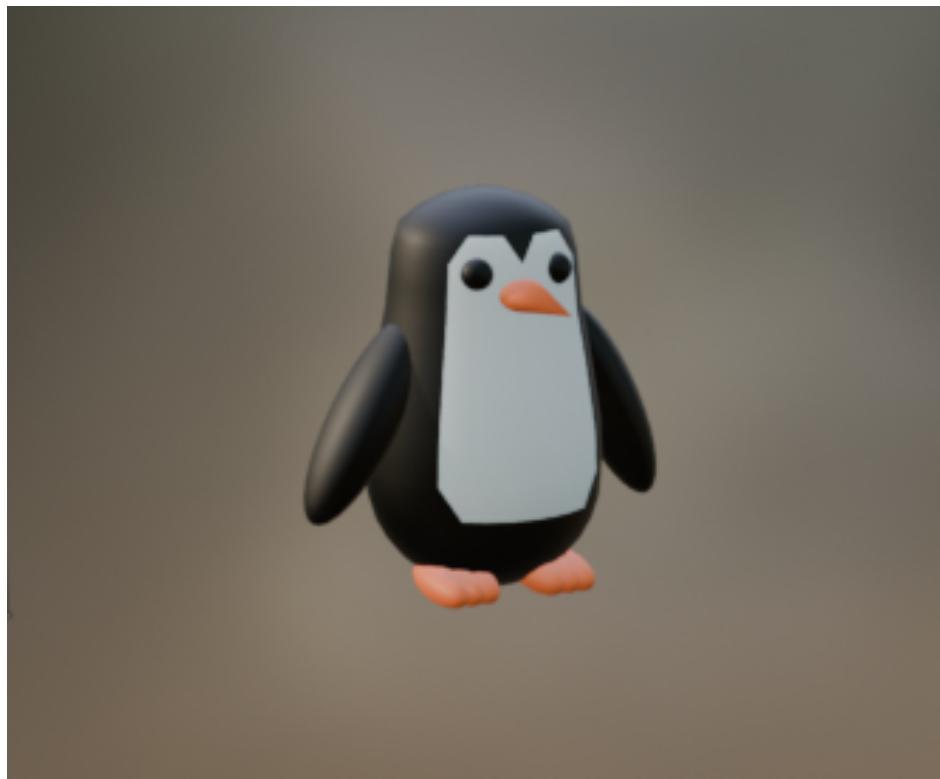


Figure 5: modèle 3D pingouin

## 3 Un grand voyage

### 3.1 Match cut

Au cœur de notre projet, se trouvait l'idée d'un grand voyage. Celui des tortues pour retrouver leur pomme volée. Pour représenter la diversité des espaces et garder le rythme de la musique, nous sommes partis vers un match cut. Le match cut est un effet de transition qui utilise un élément de liaison entre deux séquences. Ce raccord au montage permet une transition élégante entre des personnages, des lieux, des époques. Ici, on utilise le mouvement de marche continu des tortues comme élément de transition entre nos scènes 3D.

### 3.2 Les scènes 3D

Nous avons donc créé notre ensemble de 9 scènes, afin de simplifier le placement de nos objets. Nous avons fait le choix de créer des modèles 3D avec tous les éléments immobiles, préplacés et prêts à être mis dans le projet.

### 3.3 La Plaine

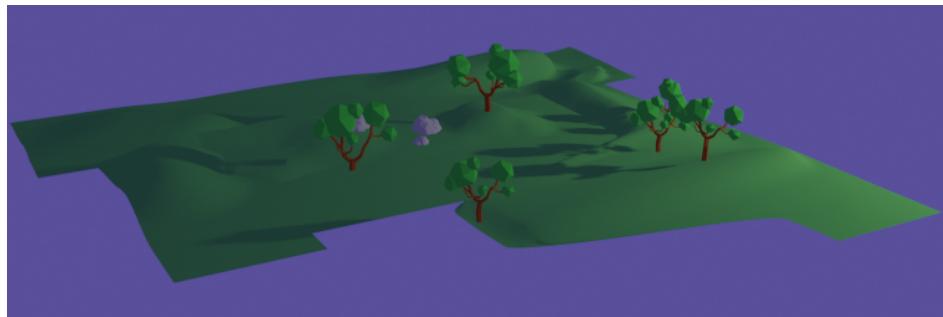


Figure 6: Notre modèle de plaine

### 3.4 Le Puits



Figure 7: Notre modèle de puits

On a réutilisé le puits que Maxime avait pu créer dans un précédent projet.

### 3.5 La Forêt

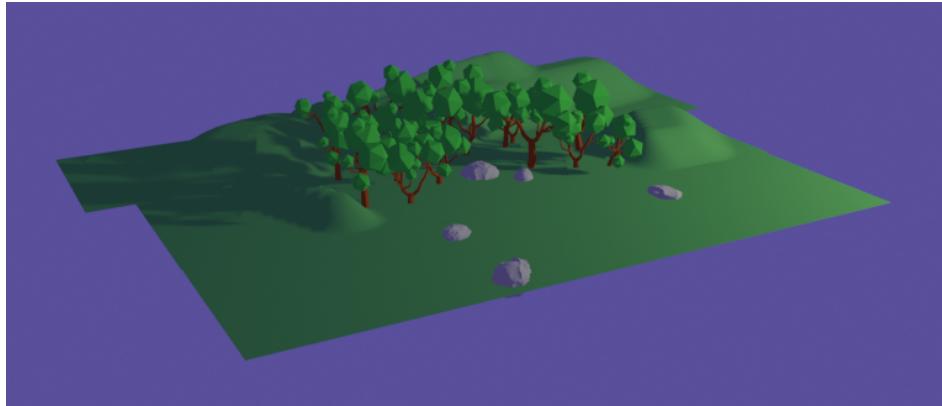


Figure 8: Notre modèle de forêt

### 3.6 L'Arctique

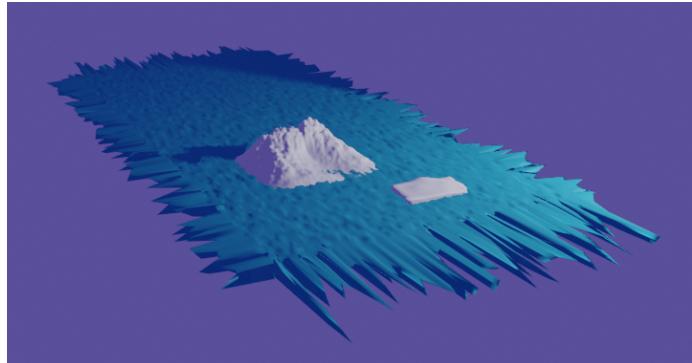


Figure 9: Notre modèle d'iceberg

D'une part, pour modéliser des icebergs réalistes, nous avons utilisé l'add-on Blender "A N T Landscape" qui permet de générer des formes et de jouer sur le niveau de bruits de chaque point sur les axes de translation.

D'autre part, pour modéliser la mer, nous aurions pu utiliser un tas d'outils de Blender pour créer des liquides réalistes, mais les résultats semblaient impossibles à notre niveau à adapter à notre code, nous avons donc simplement opéré une transformation randomize pour simuler le mouvement des vagues sur l'eau.

Dernier point, nous avons rajouté une plateforme séparée du reste du modèle qui avancera en décalé de nos tortues et sur laquelle elles se déplaceront.

### 3.7 La Ville

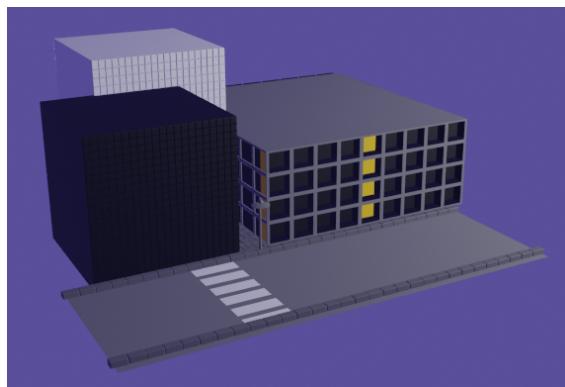


Figure 10: Notre modèle de ville

On a rajouté un lampadaire pour ajouter un effet de spotlight.

### 3.8 Le Volcan

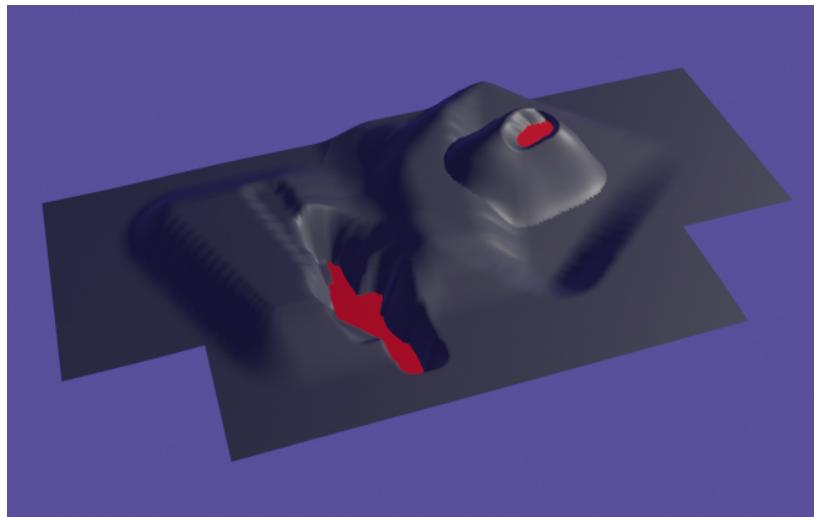


Figure 11: Notre modèle de volcan

### 3.9 L'INSA

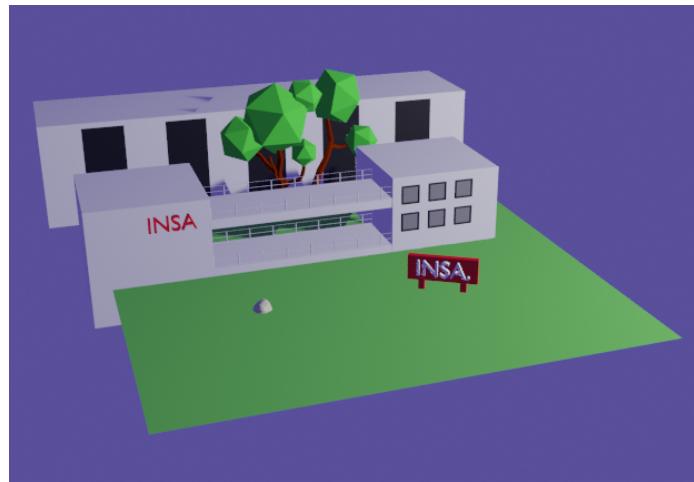


Figure 12: Notre modèle de l'INSA

On était obligé de voyager dans l'univers de l'INSA à un moment. L'objectif au début du projet, c'était qu'il apparaisse au moment du "oh que le monde est petit" de la musique, mais on a dû rajouter du temps sur les scènes pour laisser aux gens le temps de les voir.

### 3.10 La Weird Zone

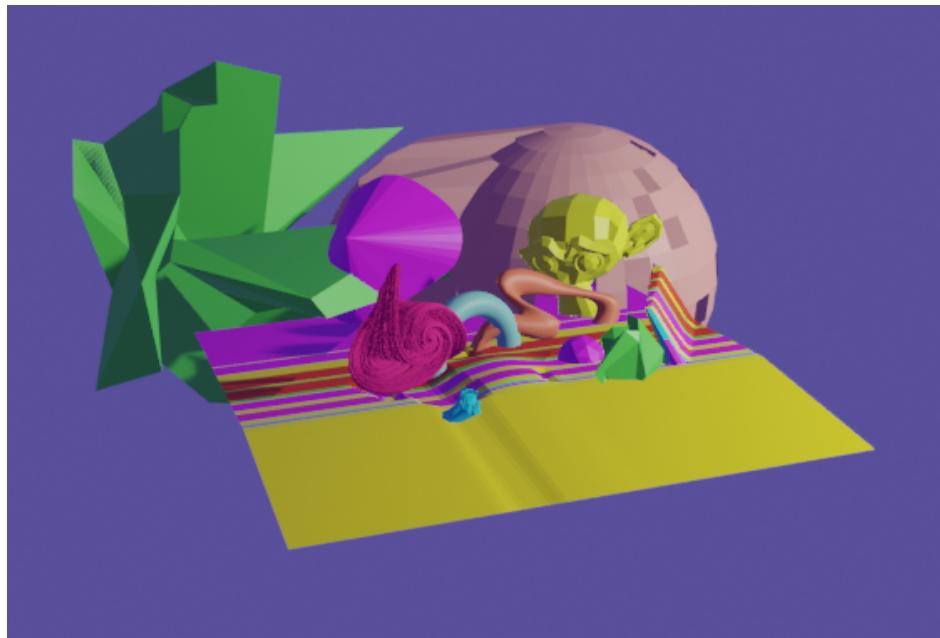


Figure 13: Notre modèle de weird zone

Ici, on s'est permis de jouer sur l'espace et sur les notions importantes de l'informatique graphique, comme le fameux modèle de Suzanne, par exemple. On a rajouté le travail en geometry node Blender sur les attractors de Halvorsen d'un ami.

### 3.11 Les Egouts

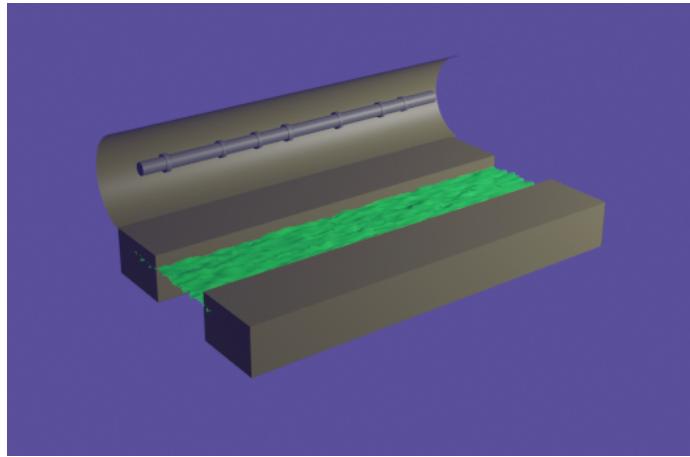


Figure 14: Notre modèle d'égouts

## 4 Accepter les matériaux et les textures

### 4.1 Problématique

On nous a donné une fonction qui permettait de lire les matériaux d'un objet. Comme nous avions beaucoup d'objets et beaucoup de matériaux, nous avons construit une fonction pour load un objet entier avec tous ses matériaux.

### 4.2 Code

```

1 void loadObjWithMaterials(
2     const std::string& obj_path,
3     const std::string& basepath,
4     ShaderProgramPtr shader,
5     std::vector<LightedMeshRenderablePtr>& result_parts)
6 {
7     std::vector<std::vector<glm::vec3>> positions, normals;
8     std::vector<std::vector<glm::vec2>> texcoords;
9     std::vector<MaterialPtr> materials;
10
11     //On lit l'objet avec ses matériaux.
12     bool ok = read_obj_with_materials(obj_path, basepath, positions, normals,
13                                         texcoords, materials);
14
15     size_t n_parts = materials.size();
16
17     positions.resize(n_parts);
18     normals.resize(n_parts);

```

```

18     texcoords.resize(n_parts);
19
20
21     for (size_t i = 0; i < n_parts; ++i)
22     {
23         //On construit notre lightedMeshRenderable
24         LightedMeshRenderablePtr part = std::make_shared<LightedMeshRenderable>(
25             shader, positions[i], normals[i],
26             std::vector<glm::vec4>(),
27             materials[i]
28         );
29         result_parts.push_back(part);
30     }
31 }
```

### 4.3 Application

On a ensuite mis en place nos objets de cette façon :

```

1 //on construit un vecteur de pointeur vers des LightedMeshRenderable
2 std::vector<LightedMeshRenderablePtr> plaine_parts;
3
4 //on utilise notre fonction
5 loadObjWithMaterials(basepath + "PlaineNT.obj", basepath, phongShader,
6 plaine_parts);
7
8 //on prends notre racine.
9 LightedMeshRenderablePtr plaine_root = plaine_parts[0];
10 viewer.addRenderable(plaine_root);
11
12 //puis on lui rajoute ses fils : toutes les autres parties de l'objet
13 for (int i = 1; i < plaine_parts.size(); ++i)
    HierarchicalRenderable::addChild(plaine_root, plaine_parts[i]);
```

## 5 Le cycle d'animation de la tortue

### 5.1 Problématique

Le cycle d'animation de la tortue est l'un des éléments qui nous ont donné le plus de fils à retordre. Nous avons pondu d'innombrables versions tout au long du projet, sans jamais être satisfaits.

### 5.2 Attentes

Nous voulions un cycle de marche réaliste, inspiré par ceux que nous pouvions trouver en ligne. Malheureusement, impossible de manipuler les pattes point par point dans notre configuration. Nous étions destinés à être déçus par notre marche.

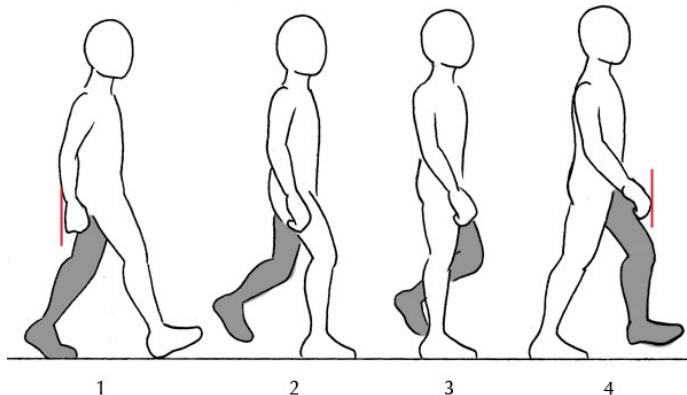


Figure 15: Notre cycle de marche attendu

### 5.3 Nos itérations

Dans notre première version, nous faisions simplement monter les pattes et les redescendre en rythme. Cela rendait, cependant, l'aspect de notre tortue trop mécanique. Nous avons donc essayé de construire une deuxième version en incluant des cycles de rotation. Malheureusement, lorsque l'on importe nos modèles Blender sur openGL le centre de rotation de nos pattes reste le même que l'objet. Ainsi, notre rotation était beaucoup trop basse. Le rendu final était peu convaincant. Il aurait fallu placer le centre de rotation au-dessus des pattes à la main.

Version 1:

```

1 piedsT_1->addGlobalTransformKeyframe(GeometricTransformation(glm::vec3(0,0,0),
2                                         glm::angleAxis(glm::radians(0.0f), glm::vec3(0,1,0)), glm::vec3(1,1,1)), 0.0f)
3 ;
4 piedsT_1->addGlobalTransformKeyframe(GeometricTransformation(glm::vec3
5 (0,0.07,0), glm::angleAxis(glm::radians(0.0f), glm::vec3(0,1,0)), glm::vec3
6 (1,1,1)), 0.5f);
7 piedsT_1->addGlobalTransformKeyframe(GeometricTransformation(glm::vec3(0,0,0),
8                                         glm::angleAxis(glm::radians(0.0f), glm::vec3(0,1,0)), glm::vec3(1,1,1)), 1.0f)
9 ;
10 piedsT_1->addGlobalTransformKeyframe(GeometricTransformation(glm::vec3(0,0,0),
11                                         glm::angleAxis(glm::radians(0.0f), glm::vec3(0,1,0)), glm::vec3(1,1,1)), 2.0f)
12 ;

```

Version 2:

```
1 // Jambes avant
2 float cycleDuration = 8.0f; // 4 secondes pour un cycle complet
3
4 // Amplitude réduite pour lever moins la patte
5 float lift = 0.015f; // moins haut
6 float forward = 0.01f; // translation avant/arrière
7 float rotationAngle = 5.0f; // plus grande rotation pour compenser le lift
8
9 // Jambes avant
10 tort_parts[1] -> addLocalTransformKeyframe(GeometricTransformation(glm::vec3(0,0,0),
    glm::angleAxis(glm::radians(rotationAngle), glm::vec3(1,0,0)), glm::vec3(1)),
```

```

    0.0f);
11 tort_parts[1]->addLocalTransformKeyframe(GeometricTransformation(glm::vec3(0, lift,
    forward), glm::angleAxis(glm::radians(-rotationAngle), glm::vec3(1,0,0)), glm::
    vec3(1)), cycleDuration*0.25f);
12 tort_parts[1]->addLocalTransformKeyframe(GeometricTransformation(glm::vec3(0,0,0),
    glm::angleAxis(glm::radians(rotationAngle), glm::vec3(1,0,0)), glm::vec3(1)),
    cycleDuration*0.5f);
13 tort_parts[1]->addLocalTransformKeyframe(GeometricTransformation(glm::vec3(0, lift,
    forward), glm::angleAxis(glm::radians(-rotationAngle), glm::vec3(1,0,0)), glm::
    vec3(1)), cycleDuration*0.75f);
14 tort_parts[1]->addLocalTransformKeyframe(GeometricTransformation(glm::vec3(0,0,0),
    glm::angleAxis(glm::radians(rotationAngle), glm::vec3(1,0,0)), glm::vec3(1)),
    cycleDuration);

```

Pour notre dernière version, nous avons fait le choix de revenir aux bases de notre première version. Cette fois-ci, nonobstant, lors de la montée de la patte, celle-ci fait un mouvement en avant et lors de la descente aussi. Puis, elle finit par revenir sur sa position initiale ce qui nous donne une animation déjà un peu plus naturel, même si un mouvement de rotation l'aurait été beaucoup plus.

```

1 tort_root->addGlobalTransformKeyframe(GeometricTransformation(glm::vec3(0,0.5,-16)
    , glm::angleAxis(glm::radians(180.0f), glm::vec3(0,1,0)), glm::vec3(1,1,1)),
    0.0f);
2 tort_root->addGlobalTransformKeyframe(GeometricTransformation(glm::vec3
    (0,0.5,-10.6) , glm::angleAxis(glm::radians(180.0f), glm::vec3(0,1,0)), glm::
    vec3(1,1,1)), 12.5f);
3 tort_root->addGlobalTransformKeyframe(GeometricTransformation(glm::vec3
    (0,0.5,-10.6) , glm::angleAxis(glm::radians(180.0f), glm::vec3(0,1,0)), glm::
    vec3(1,1,1)), 16.0f);
4 tort_root->addGlobalTransformKeyframe(GeometricTransformation(glm::vec3
    (0,0.5,-1.6) , glm::angleAxis(glm::radians(180.0f), glm::vec3(0,1,0)), glm::
    vec3(1,1,1)), 46.0f);
5 tort_root->addGlobalTransformKeyframe(GeometricTransformation(glm::vec3
    (0,0.5,16) , glm::angleAxis(glm::radians(180.0f), glm::vec3(0,1,0)), glm::
    vec3(1,1,1)), 70.0f);
6 tort_root->addGlobalTransformKeyframe(GeometricTransformation(glm::vec3
    (0,0.5,16) , glm::angleAxis(glm::radians(180.0f), glm::vec3(0,1,0)), glm::
    vec3(0,0,0)), 70.001f);
7 tort_root->addGlobalTransformKeyframe(GeometricTransformation(glm::vec3
    (0,0.5,16) , glm::angleAxis(glm::radians(180.0f), glm::vec3(0,1,0)), glm::
    vec3(0,0,0)), 83.0f);
8
9 tort_parts[1]->addLocalTransformKeyframe(GeometricTransformation(glm::vec3(0,0,0),
    glm::angleAxis(glm::radians(0.0f), glm::vec3(0,1,0)), glm::vec3(1,1,1)), 0.0f)
    ;
10 tort_parts[1]->addLocalTransformKeyframe(GeometricTransformation(glm::vec3
    (0,0.1,-0.025), glm::angleAxis(glm::radians(0.0f), glm::vec3(0,1,0)), glm::
    vec3(1,1,1)), 1.0f);
11 tort_parts[1]->addLocalTransformKeyframe(GeometricTransformation(glm::vec3
    (0,0,-0.05), glm::angleAxis(glm::radians(0.0f), glm::vec3(0,1,0)), glm::
    vec3(1,1,1)), 1.5f);
12 tort_parts[1]->addLocalTransformKeyframe(GeometricTransformation(glm::vec3(0,0,0),
    glm::angleAxis(glm::radians(0.0f), glm::vec3(0,1,0)), glm::vec3(1,1,1)), 2.0f)
    ;
13 tort_parts[1]->addLocalTransformKeyframe(GeometricTransformation(glm::vec3(0,0,0),
    glm::angleAxis(glm::radians(0.0f), glm::vec3(0,1,0)), glm::vec3(1,1,1)), 4.0f)

```

```
;
```

## 6 Defying Gravity

Vient ensuite le deuxième défi technique : l'animation d'une pomme qui chute et qui rebondit sur le sol (et pas sur la tête d'un certain physicien). Pour cela, nous avons en premier lieu adapté le code que nous avions dans particuleRenderable pour créer un objet pomme LightedMershRenderable. Tout cela afin de pouvoir transformer notre particule en pomme qui possède des matériaux et qui réagit à la lumière.

```
1 ParticleRenderable::ParticleRenderable(ShaderProgramPtr program, const ParticlePtr
2   & particle, const std::string & mesh_filename) :
3   LightedMeshRenderable(program, mesh_filename, Material::Pomme()),
4   m_particle(particle)
5 {
6   update_all_buffers();
}
```

Nous avons aussi dû créer un matériau pomme.

```
1 MaterialPtr Material::Pomme()
2 {
3   float openGLFactor=128.0;
4   glm::vec3 ambient(1);
5   glm::vec3 diffuse(0.800000,0.006386,0.020132);
6   glm::vec3 specular(0.5);
7   float shininess = openGLFactor*0.2;
8   return std::make_shared<Material>(ambient, diffuse, specular, shininess);
9 }
```

Ensuite, il fallait déclencher l'animation de chute à un moment précis, puis faire disparaître la pomme animée ainsi que son plan de support. Nous n'avons pas trouvé de méthode simple pour retarder directement cette animation. Cela aurait été facile à réaliser à l'aide de keyframes, si celles-ci avaient été implémentées pour ce type de tâche, ce qui n'était pas le cas. Nous avons donc mis en place une solution dans la boucle d'animation en utilisant le temps du système pour déclencher la chute. De la même manière, nous désactivons la collision et remplaçons progressivement la pomme par une autre, que nous pouvons cette fois contrôler avec des keyframes afin d'enchaîner la suite de l'animation.

```
1 int main()
2 {
3   glm::vec4 background_color(0.7,0.7,0.7,1);
4   Viewer viewer(1280,720,background_color);
5   DynamicSystemPtr system = std::make_shared<DynamicSystem>();
6   DynamicSystemRenderablePtr systemRenderable = std::make_shared<
7     DynamicSystemRenderable>(system);
8   ShaderProgramPtr flatShader = std::make_shared<ShaderProgram>(
9     "../../../
sfmlGraphicsPipeline/shaders/flatVertex.glsl",
"../../../
sfmlGraphicsPipeline/shaders/flatFragment.glsl");
  QuadMeshRenderablePtr planeRenderable = std::make_shared<QuadMeshRenderable>(
    flatShader, p1, p2, p3, p4);
```

```

10    float pr = 0.3;
11    initialize_scene(viewer, system, systemRenderable, flatShader);
12
13
14    while( viewer.isRunning() )
15    {
16        viewer.handleEvent();
17        viewer.animate();
18        if (!collisionLaunched && viewer.getTime() >= animcol) {
19            collisions(viewer, system, systemRenderable, planeRenderable);
20            collisionLaunched = true;
21        }
22
23
24        if (viewer.getTime() >= animcol + 3.5f){
25            system->setCollisionsDetection(false);
26            planeRenderable->setLocalTransform(getScaleMatrix(0,0,0));
27            viewer.getCamera();
28        }
29
30
31        viewer.draw();
32        viewer.display();
33    }
34
35
36    return EXIT_SUCCESS;
37 }
```

## 7 Nos fonds

Une fois que toutes nos scènes et personnages ont été mis en place, nous avons eu besoin de fonds pour créer notre paysage. Dans cet objectif, nous avons utilisé deux méthodes différentes. Dans la première scène, nous avons utilisé la cubeMap, car nous avions deux plans différents de caméra qui regardaient chacun sur un axe différent et donc nous avions besoin d'avoir un décor panoramique.

```

1 // cubemap
2
3     viewer.getCamera().setViewMatrix( glm::lookAt( glm::vec3(1, 1, 1), glm::
4         vec3(0, 0, 0), glm::vec3( 0, 1, 0 ) ) );
5     ShaderProgramPtr cubeMapShader = std::make_shared<ShaderProgram>( " ../../
6         sfmlGraphicsPipeline/shaders/cubeMapVertex.glsl",
7                                     " ../../
8         sfmlGraphicsPipeline/shaders/cubeMapFragment.glsl");
9     viewer.addShaderProgram(cubeMapShader);
10
11
12     std::string cubemap_dir = "../../sfmlGraphicsPipeline/textures/skybox3";
13     auto cubemap = std::make_shared<CubeMapRenderable>(cubeMapShader,
14     cubemap_dir);
15
16     viewer.addRenderable(cubemap);
17 }
```

```
14 }
```

Dans les plans suivants, nous avons utilisé une méthode plus simple. Pour ceux-ci, nous avons créé des plans sur Blender que nous avons placés à l'arrière de nos scènes et sur lesquels nous avons ajouté des images à l'aide des TexturedLightedMeshRenderable.

```
1 const std::string fond2_texture_path = "../../sfmlGraphicsPipeline/meshes/
2 fond2.png";
3 TexturedLightedMeshRenderablePtr Fond2 =
4     std::make_shared<TexturedLightedMeshRenderable>(
5         textureShader,
6         fond_mesh_path,
7         fond_material,
8         fond2_texture_path
9 );
10 Fond2->addGlobalTransformKeyframe(GeometricTransformation(glm::vec3(0,0,0), glm::
11     angleAxis(0.0f,glm::vec3(0,0,1)), glm::vec3(1)), 28.0f);
12 Fond2->addGlobalTransformKeyframe(GeometricTransformation(glm::vec3(0,0,0), glm::
13     angleAxis(0.0f,glm::vec3(0,0,1)), glm::vec3(1)), 34.0f);
14 Fond2->addGlobalTransformKeyframe(GeometricTransformation(glm::vec3(0,0,0), glm::
15     angleAxis(0.0f,glm::vec3(0,0,1)), glm::vec3(0)), 34.0001f);
16 Fond2->addGlobalTransformKeyframe(GeometricTransformation(glm::vec3(0,0,0), glm::
17     angleAxis(0.0f,glm::vec3(0,0,1)), glm::vec3(0)), 83.0f);
18 viewer.addRenderable(Fond2);
```

## 8 Les Lumières

Une fois que nous avons placé tous les éléments d'animation, il était temps de s'occuper de l'éclairage de nos scènes. La première lumière est une directionalLight qui éclaire le dessous de nos modèles pour éviter qu'ils soient trop sombres.

```
1 dir = glm::normalize(glm::vec3(0,1,0));
2 ambient = glm::vec3(0,0,0);
3 diffuse = glm::vec3(0.3);
4 specular = glm::vec3(0.5);
5 DirectionalLightPtr light3 = std::make_shared<DirectionalLight>(dir, ambient,
6     diffuse, specular);
    viewer.addDirectionalLight(light3);
```

Ensuite, la deuxième est une nouvelle directionalLight qui est censée représenter le soleil et donc qui évolue sur le temps durant les différentes scènes afin de montrer le temps qui passe.

```
1 { // Moving DirectionalLight
2     auto dir_light = std::make_shared<DirectionalLight>(glm::vec3(-1,-1,-1),
3         glm::vec3(0), glm::vec3(1), glm::vec3(1));
4     viewer.addDirectionalLight(dir_light);
5
6     dir_light->addGlobalTransformKeyframe(getRotationMatrix( 0.375 * 2 * M_PI ,
0, 1, 1)*getTranslationMatrix(0,3,60) , 0);
7     dir_light->addGlobalTransformKeyframe(getRotationMatrix( 0.375 * 2 * M_PI ,
0, 1, 1)*getTranslationMatrix(0,3,60) , 28);
```

```

7     dir_light->addGlobalTransformKeyframe(getRotationMatrix( 0.425 * 2 * M_PI ,
0, 1, 1) *getTranslationMatrix(0,3,60) , 52);
8     dir_light->addGlobalTransformKeyframe(getRotationMatrix( 0.375 * 2 * M_PI ,
0, 1, 1) *getTranslationMatrix(0,3,60) , 52.001);
9     dir_light->addGlobalTransformKeyframe(getRotationMatrix( 0.425 * 2 * M_PI ,
0, 1, 1)*getTranslationMatrix(0,3,60) , 70);
10    dir_light->addGlobalTransformKeyframe(getRotationMatrix( 0.975 * 2 * M_PI ,
0, 1, 1)*getTranslationMatrix(0,3,60) , 70.001);
11    dir_light->addGlobalTransformKeyframe(getRotationMatrix( 0.975 * 2 * M_PI ,
0, 1, 1) *getTranslationMatrix(0,3,60) , 83);
12
13    auto dir_light_renderable = std::make_shared<DirectionalLightRenderable>(
flatShader, dir_light);
14    dir_light_renderable->setLocalTransform(getTranslationMatrix(0, 1.5, 3) *
getScaleMatrix(0.5));
15    viewer.addRenderable(dir_light_renderable);
16 }
```

Sur la scène de la ville, nous avons ajouté une spotlight pour créer un lampadaire qui aura une lumière jaune à la façon des éclairages de ville la nuit. Il n'y a que cette lumière dans la scène pour la mettre en valeur.

```

1 {
2     glm::mat4 globalTransformation, localTransformation;
3     //Define a spot light
4     glm::vec3 s_position(0.0,5.0,-8.0), s_spotDirection = glm::normalize(glm::vec3
5     (0.0,-1.0,1.0));
6     //glm::vec3 s_ambient(0.0,0.0,0.0), s_diffuse(0.0,0.0,0.0), s_specular
7     (0.0,0.0,0.0);
8     glm::vec3 s_ambient(0.0,0.0,0.0), s_diffuse(1), s_specular(1.2);
9     float s_constant=1.0, s_linear=0.0, s_quadratic=0.0;
10    float s_innerCutOff=std::cos(glm::radians(20.0f)), s_outerCutOff=std::cos(glm::
11     radians(40.0f));
12    SpotLightPtr spotLight2 = std::make_shared<SpotLight>(s_position,
13     s_spotDirection,
14                                         s_ambient, s_diffuse,
15                                         s_specular,
16                                         s_constant, s_linear,
17                                         s_innerCutOff,
18                                         s_outerCutOff);
19    viewer.addSpotLight(spotLight2);
20    spotLight2->addGlobalTransformKeyframe(lookAtModel(glm::vec3(0,-100,0), glm::
21     vec3(0,-101,0), Light::base_forward), 0.0f);
22    spotLight2->addGlobalTransformKeyframe(lookAtModel(glm::vec3(0,-100,0), glm::
23     vec3(0,-101,0), Light::base_forward), 70.0f);
24    spotLight2->addGlobalTransformKeyframe(lookAtModel(glm::vec3(-2,10,-2), glm::
25     vec3(0,0,0), Light::base_forward), 70.001f);
26    spotLight2->addGlobalTransformKeyframe(lookAtModel(glm::vec3(-2,10,-2), glm::
27     vec3(0,0,0), Light::base_forward), 83.0f);
28    SpotLightRenderablePtr spotLightRenderable2 = std::make_shared<
29     SpotLightRenderable>(flatShader, spotLight2);
30    localTransformation = glm::scale(glm::mat4(1.0), glm::vec3(0,0,0));
31    spotLightRenderable2->setLocalTransform(localTransformation);
32    viewer.addRenderable(spotLightRenderable2);
```

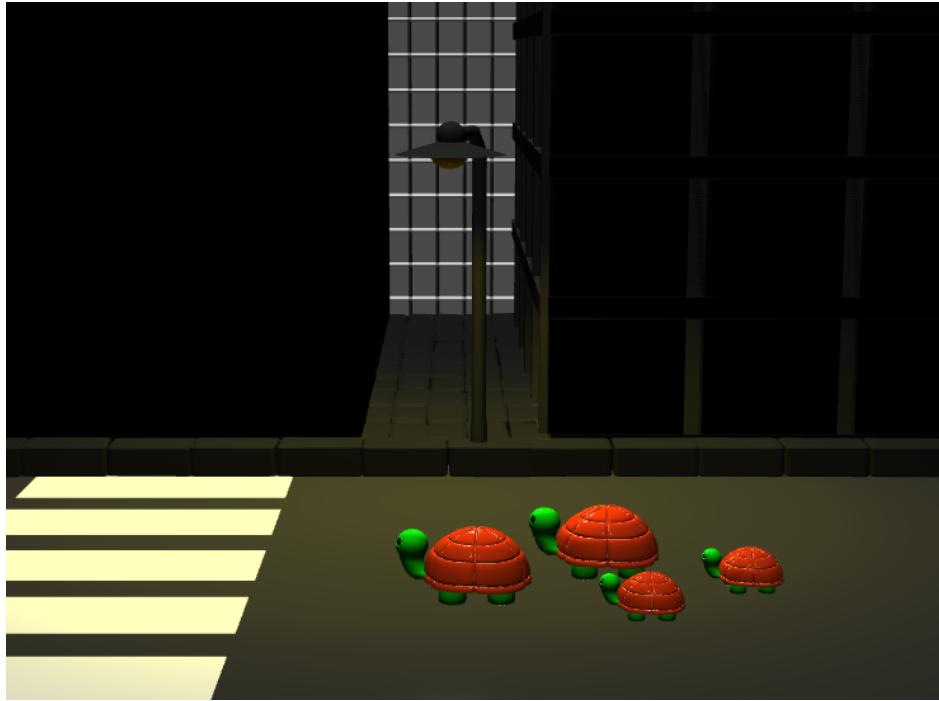


Figure 16: Le lampadaire en ville

Enfin, dans la scène de l'égout, nous avons disposé plusieurs projecteurs clignotants afin d'ajouter un effet visuel supplémentaire. Pour créer cet effet, nous avons mis en place un cycle durant lequel la lumière apparaît dans le champ de la scène, puis disparaît périodiquement.

```

1   spotLight3->addGlobalTransformKeyframe(lookAtModel(glm::vec3(0,-100,0), glm::
2     vec3(0,-101,0), Light::base_forward), 0.0f);
3   spotLight3->addGlobalTransformKeyframe(lookAtModel(glm::vec3(0,-100,0), glm::
4     vec3(0,-101,0), Light::base_forward), 70.0f);
5   spotLight3->addGlobalTransformKeyframe(lookAtModel(glm::vec3(2,10,2), glm::vec3
6     (2,-101,2), Light::base_forward), 70.001f);
7   spotLight3->addGlobalTransformKeyframe(lookAtModel(glm::vec3(2,10,2), glm::vec3
8     (2,-101,2), Light::base_forward), 71.000f);
9   spotLight3->addGlobalTransformKeyframe(lookAtModel(glm::vec3(0,-100,0), glm::
10    vec3(0,-101,0), Light::base_forward), 71.001f);
11  spotLight3->addGlobalTransformKeyframe(lookAtModel(glm::vec3(0,-100,0), glm::
12    vec3(0,-101,0), Light::base_forward), 71.20f);
13  spotLight3->addGlobalTransformKeyframe(lookAtModel(glm::vec3(2,10,2), glm::vec3
14    (2,-101,2), Light::base_forward), 71.201f);
15  spotLight3->addGlobalTransformKeyframe(lookAtModel(glm::vec3(2,10,2), glm::vec3
16    (2,-101,2), Light::base_forward), 72.201f);

```

## 9 La caméra

Pour obtenir une animation de bonne qualité, il est aussi important d'avoir de bons plans de caméra qu'il s'agisse d'un plan fixe ou en mouvement. Cela permet, en effet, d'apporter une fluidité à l'animation tandis que si nous avions bougé la caméra manuellement le rendu aurait potentiellement été saccadé. C'est le bénéfice de l'animation comparé au film live de pouvoir faire des courbes de déplacement linéaires, mais cela peut aussi donner un aspect trop mécanique au mouvement parfois.

### 9.1 Plan fixe

Notre animation utilise en grande partie un plan fixe, car dans l'objectif de réaliser un match cut, nous avions besoin d'un plan fixe.

### 9.2 Plan en mouvement

Nous avons tout de même réalisé 3 plans en mouvement durant notre animation. Le premier concerne le déplacement de la caméra vers l'arbre avec sa pomme. Ensuite, une transition s'effectue sur la tête de la tortue et mène au deuxième plan, qui est animé par un effet de vertigo. Pour finir, dans la dernière scène, nous effectuons un mouvement dans le but de nous rapprocher de la souris endormie.

```
1 //Premier plan fixe sur notre première scène
2     viewer.getCamera().addGlobalTransformKeyframe( lookAtUpModel(glm::vec3(
3         30, 10, 0), glm::vec3(0,5,0)), 0 );
4     viewer.getCamera().addGlobalTransformKeyframe( lookAtUpModel(glm::vec3(
5         30, 10, 0), glm::vec3(0,5,0)), 8 );
6     //Mouvement de caméra afin de se rapprocher de l'arbre avec la pomme dans
7     //la première scène
8     viewer.getCamera().addGlobalTransformKeyframe( lookAtUpModel(glm::vec3( 8,
9         10, 0), glm::vec3(-13,3.8,14.7)), 11 );
10    viewer.getCamera().addGlobalTransformKeyframe( lookAtUpModel(glm::vec3( 8,
11        10, 0), glm::vec3(-13,3.8,14.7)), 12.5 );
12    //Changement instantané de plan sur la tête de la tortue puis mouvement de
13    //recul pour l'effet vertigo
14    viewer.getCamera().addGlobalTransformKeyframe( lookAtUpModel(glm::vec3( 0,
15        0.75, -8), glm::vec3(0,0.75,-10.6)), 12.501 );
16    viewer.getCamera().addGlobalTransformKeyframe( lookAtUpModel(glm::vec3( 0,
17        0.75, -5), glm::vec3(0,0.75,-10.6)), 16.0 );
18    //Changement de plan de nouveau en regardant l'arbre met avec un distance
19    //plus grand à l'arbre afin de voir la pomme tombé
20    //et la souris la récupérer.
21    viewer.getCamera().addGlobalTransformKeyframe( lookAtUpModel(glm::vec3(
22        20, 10, 0), glm::vec3(-13,3.8,14.7)), 16.001 );
23    viewer.getCamera().addGlobalTransformKeyframe( lookAtUpModel(glm::vec3(
24        20, 10, 0), glm::vec3(-13,3.8,14.7)), 22.0 );
25    //Retour sur le plan fixe global qui dure jusqu'à la scène de l'égout.
26    viewer.getCamera().addGlobalTransformKeyframe( lookAtUpModel(glm::vec3(
27        30, 10, 0), glm::vec3(0,5,0)), 22.001 );
28    viewer.getCamera().addGlobalTransformKeyframe( lookAtUpModel(glm::vec3(
29        30, 10, 0), glm::vec3(0,5,0)), 70 );
30    //Mouvement de caméra pour se rapprocher de la souris endormie et finir l'
31    //animation avec les tortues qui rentre dans
32    //le champ de la caméra.
33    viewer.getCamera().addGlobalTransformKeyframe( lookAtUpModel(glm::vec3(
34        20, 9, 0), glm::vec3(0,5,0)), 70.0001 );
35    viewer.getCamera().addGlobalTransformKeyframe( lookAtUpModel(glm::vec3(
```

```

21     14, 6, 0), glm::vec3(-4.25, 0.5, 19.5)), 77.0 );
viewer.getCamera().addGlobalTransformKeyframe( lookAtUpModel(glm::vec3(
14, 6, 0), glm::vec3(-4.25, 0.5, 19.5)), 83 );

```

## 10 Mission Vertigo

### 10.1 Point de départ

Un des plans les plus importants de notre projet consistait en la réaction de la maman tortue face à la découverte de la pomme. Mais comment faire comprendre la surprise et l'envie de la tortue sans bruit ? Pour ça, nous avions besoin d'un plan vertigo.

Maxime, cinéphile et ingénieur de son état, a toujours envie de mettre des plans vertigés dans ses films. Ce sont des plans très difficiles techniquement, mais qui apportent une esthétique fabuleuse. Et, bien utilisé, ils apportent toujours un petit quelque chose au court-métrage.

### 10.2 Théorie et histoire

L'effet vertigo repose sur le principe de la parallaxe, c'est-à-dire le contraste de mouvement entre le premier plan et l'arrière-plan, qui ne se déplacent pas à la même vitesse lors d'un mouvement de caméra. Il consiste à combiner un travelling avant ou arrière (dans notre court métrage, un travelling arrière) avec un zoom dans la direction opposée. Le zoom ayant pour effet d'écraser les formes, notamment les visages, la combinaison de ces deux mouvements crée une forte impression de distorsion : l'arrière-plan semble se modifier très rapidement, tandis que le premier plan paraît s'écraser tout en restant visuellement à la même position.



Figure 17: Emplois célèbres de l'effet Vertigo

Cet effet a été utilisé pour la première fois par Hitchcock[3] dans son film *Vertigo* pour représenter les crises d'angoisse de son personnage principal lorsqu'il se rapprochait du vide. Puis, il a ensuite été utilisé par Spielberg pour la première fois dans son utilisation qu'on lui connaît : celle de représenter le changement d'état d'âme d'un personnage. Dans *Jaws*[4], cela représente l'effroi soudain que l'arrivée du requin procure au protagoniste. Ici, on l'utilise pour la stupide et instantanée surprise joyeuse de voir une pomme à manger.

### 10.3 Application sur le projet

Restait toutefois la question de l'implémentation d'un tel effet dans notre projet. Au départ, nous pensions ce plan impossible à réaliser, car nous ne disposions que de très peu d'informations sur l'objet caméra. Nous avons néanmoins constaté que la focale y était implémentée, ce qui nous permettait de la modifier à la demande. Cependant, un problème subsistait : il n'était pas possible de changer la focale plusieurs fois au cours du court-métrage, ni d'effectuer une variation progressive de celle-ci. Or, la réalisation de l'effet vertigo repose précisément sur une transition continue de la focale, sans laquelle le plan devient compliqué voire irréalisable.

Obstinés, nous avons pris le code à bras-le-corps. En nous inspirant de notre KeyFrameCollection implémenté en TP, nous avons recréé un système de keyframe semblable à celui des positions, mais adapté à un float.

### 10.4 Implémentation du système de keyframe pour notre focale

Dans notre KeyframeFloatCollection, nous disposons d'un unique attribut, à l'image de la KeyframeCollection d'origine : une map associant des temps à leurs valeurs de focale correspondantes. Nous y implémentons ensuite les principales fonctions nécessaires à son fonctionnement :

**Add** : Permet de rajouter un keyframe à notre map.

**Empty** : Permet de vider notre map.

**GetBoundingKeyframes** : Permet de récupérer les keyframes qui bornent le moment où la fonction est appelée.

```
1 std::array<KeyframeCollectionFloat::Keyframe, 2>
2 KeyframeCollectionFloat::getBoundingKeyframes(float time) const
3 {
4     std::array<Keyframe, 2> result{
5         std::make_pair(0.0f, 0.0f),
6         std::make_pair(0.0f, 0.0f)
7     }; // On initialise notre array avec des Keyframe.
8
9
10    std::map<float, float>::const_iterator upper = m_keyframes.upper_bound(
11        time); // on récupère la première keyframe donc le temps est sup à time.
12    std::map<float, float>::const_iterator lower = std::prev(upper); // on récup
13    ère la keyframe juste avant ou à time => borne inférieure
14    std::map<float, float>::const_iterator end = m_keyframes.end();
15
16
17    if (upper != end && lower != end) // on vérifie que tout est valide.
18    {
19        result[0] = *lower;
20        result[1] = *upper;
21    }
22
23
24    return result;
25 }
```

**Interpolate** : Permet d'avoir chaque valeur de focale calculée selon le temps.

```

1 float KeyframeCollectionFloat::interpolate(float time) const
2 {
3     if (!m_keyframes.empty())
4     {
5         std::map<float, float>::const_iterator itFirst = m_keyframes.begin();
6         std::map<float, float>::const_reverse_iterator itLast = m_keyframes.
7         rbegin();
8
9         float effective_time = std::fmod(time, itLast->first); // modulation
10        pour "boucler" entre itLast et itFirst
11
12        std::array<Keyframe, 2> k = getBoundingKeyframes(effective_time); // les
13        keyframes entre effective_time
14
15        float t0 = k[0].first;
16        float t1 = k[1].first;
17        float v0 = k[0].second;
18        float v1 = k[1].second;
19
20        if (t0 == t1) return v0; // eviter la division par 0
21
22
23        float factor = (effective_time - t0) / (t1 - t0);
24
25
26        return glm::mix(v0, v1, factor); // la valeur interpolée selon le temps
27        effectif
28    }
29    else
30    {
31        return 0.0f; // pas de keyframe
32    }
33}

```

## 10.5 Implémentation dans la caméra

Puis, nous sommes allés dans le code de la caméra, et avons implémenté ceci :

```

1 void Camera::addFovKeyframe(float fov, float time)
2 {
3     m_fovKeyframes.add(fov, time); // rajoute une keyframe de temps, un peu comme
4     les keyframes de position classiques.

```

Mais surtout, dans notre fonction doAnimate, nous avons ajouté une mise à jour du FOV à chaque instant :

```

1 void Camera::do_animate(float time)
2 {

```

```

3     KeyframedHierarchicalRenderable::do_animate(time);
4     updateModelMatrix();
5     m_view = glm::inverse(getModelMatrix());
6
7
8     //mon ajout pour gérer les keyframes de focale.
9     if (!m_fovKeyframes.empty())//Si on a rajouté des keyframes
10    {
11        m_fov = m_fovKeyframes.interpolate(time);// Cela trouve la focale en un temps
12        time
13        setFov(m_fov);//Puis la setup.
14    }//Et on remercie Alfred Hitchcock pour l'effet Vertigo qu'on va pouvoir faire
15    //Mais on ne le remercie pas pour sa façon de diriger ses actrices
16    //Aucune femme ne devrait être objectifiée et sacrifiée pour l'oeuvre d'un homme
17    // https://www.bbc.com/news/entertainment-arts-37821400
}

```

## 10.6 Mission terminée !

Il ne nous restait plus qu'à ajouter les positions des keyframes de FOV dans notre projet et le mouvement de travelling arrière, et c'était fait !

```

1     viewer.getCamera().addFovKeyframe(glm::radians(30.0f), 0.0f);
2     viewer.getCamera().addFovKeyframe(glm::radians(30.0f), 12.5f);
3     viewer.getCamera().addFovKeyframe(glm::radians(90.0f), 12.501f);
4     viewer.getCamera().addFovKeyframe(glm::radians(35.0f), 16.0f);
5     viewer.getCamera().addFovKeyframe(glm::radians(30.0f), 16.0001f);
6     viewer.getCamera().addFovKeyframe(glm::radians(30.0f), 83.0f);

```

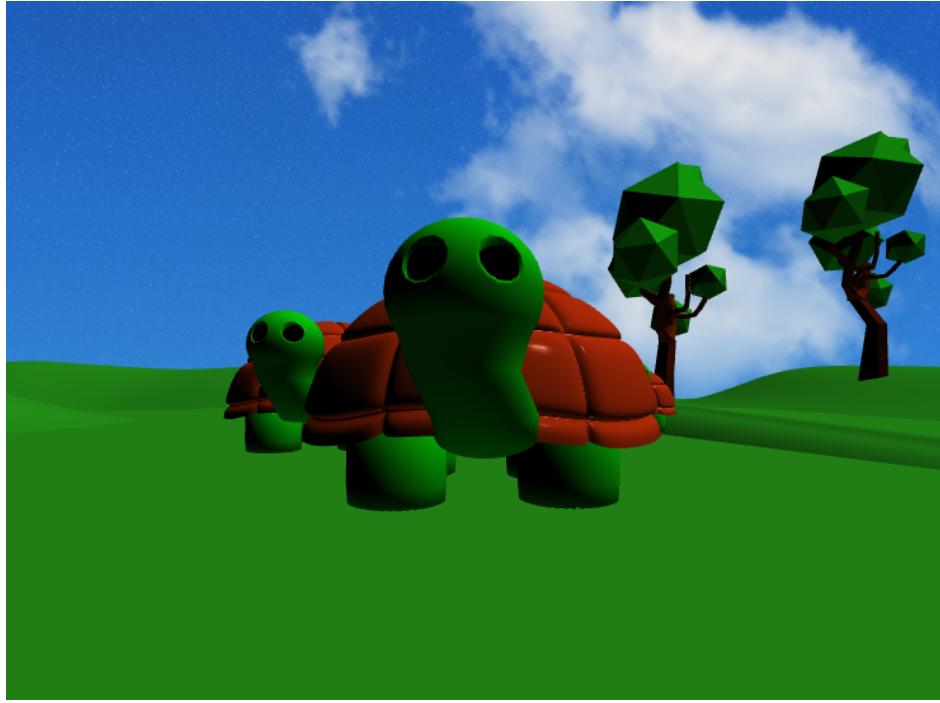


Figure 18: Rendu de notre FOV

## 11 Nos shaders

### 11.1 Point de départ

Nous voulions donner à notre court-métrage un style d'illustration pour enfant. Pour ça, nous avions besoin de trois points : limiter notre palette à un nombre de couleurs limitées (cel-shading), ajouter du bruit pour donner un effet papier à notre image, et rajouter à nos personnages des contours pour leur donner forme. Le rendu du jeu Okami, copiant le style des estampes japonaises, nous a servis d'inspiration pour notre approche visuelle.



Figure 19: Image du jeu Okami

## 11.2 Le cel-shading

Pour avoir un effet de cel-shading, il faut limiter les couleurs à des paliers d'intensité. Pour ça, on utilise la fonction `floor[5]` de glsl. Puis, avec une valeur `levels` qui correspond à notre nombre de pallier, on peut faire ceci :

```

1  vec3 cel(vec3 color)
2 {
3     float levels = 8.0;
4     return floor(color * levels) / levels;
5 }
```

## 11.3 L'effet papier (bruit)

Pour donner un effet papier à notre animation, on doit créer du bruit. Nous avons eu beaucoup de mal à trouver comment faire, car nous n'avions pas une bonne maîtrise de glsl. Nous avons fini par trouver ce code sur internet[6] :

```

1  float grain = 10;
2  float intensity = 0.05;
3  float n = fract(sin(dot(position.xy * grain, vec2(12.9898, 78.233))) *
4 43758.5453); //on utilise la position pour créer du bruit
```

Ensuite, pour contrôler l'amplitude de notre bruit et éviter des variations trop irrégulières, nous avons ajouté un mix [7].

```

1 vec3 papier(vec3 color, vec3 position)
{
3     float grain = 10;
4     float intensity = 0.05;
5     float n = fract(sin(dot(position.xy * grain, vec2(12.9898, 78.233))) *
6 43758.5453); //on utilise la position pour créer du bruit
7
8     float paper = mix(1.0 - intensity, 1.0 + intensity, n);
```

```

8     return color * paper;
9 }
10 }
```

## 11.4 Les contours

### 11.4.1 Contour noir

Les contours ont constitué notre principal défi technique sur les shaders. Il était particulièrement complexe de comprendre comment un rendu pouvait générer un contour noir autour d'un objet. Après quelques recherches sur Internet, nous avons finalement trouvé une thèse traitant spécifiquement de ce sujet :

#### Object-based edge detection[8]

It involves finding edges on an object based on two parameters. For every pixel on screen, whether that pixel is at the silhouette edge of an object or not is determined by the distance from the viewport of the vertex at that pixel and the value of the normal of the vertex at that pixel.

En nous basant sur la théorie, nous avons essayé de calculer le produit scalaire entre le vecteur direction de la caméra et la valeur de la normale du sommet de ce pixel.

```

1  vec3 contourNoir(vec3 color, vec3 N, vec3 V)
2 {
3     float ndotv = max(dot(N, V), 0.0); //N c'est la normale à l'objet, V c'est le
4                                         //vecteur direction de la caméra.
5
6     //Si c'est plus petit que 0.55, c'est le contour de notre objet => noir, sinon
7     //on laisse.
8
9     if(ndotv < 0.55)
10        return color*0.0;
11    else
12        return color;
13 }
```

### 11.4.2 Contour complémentaire

Ensuite, pour la zone « weird », nous avons eu l'idée, en ajustant les paramètres, d'ajouter un contour utilisant les couleurs complémentaires de chaque objet.

```

1  vec3 contourComplem(vec3 color, vec3 N, vec3 V)
2 {
3     float ndotv = max(dot(N, V), 0.0);
4
5     if(ndotv < 0.5)
6     {
7         return vec3(color.b, 1.0 - color.g, color.r);
8     }
9     else
10    {
```

```
11     return color;
12 }
13 }
```

## 11.5 Application de nos effets



Figure 20: Rendu de nos shaders

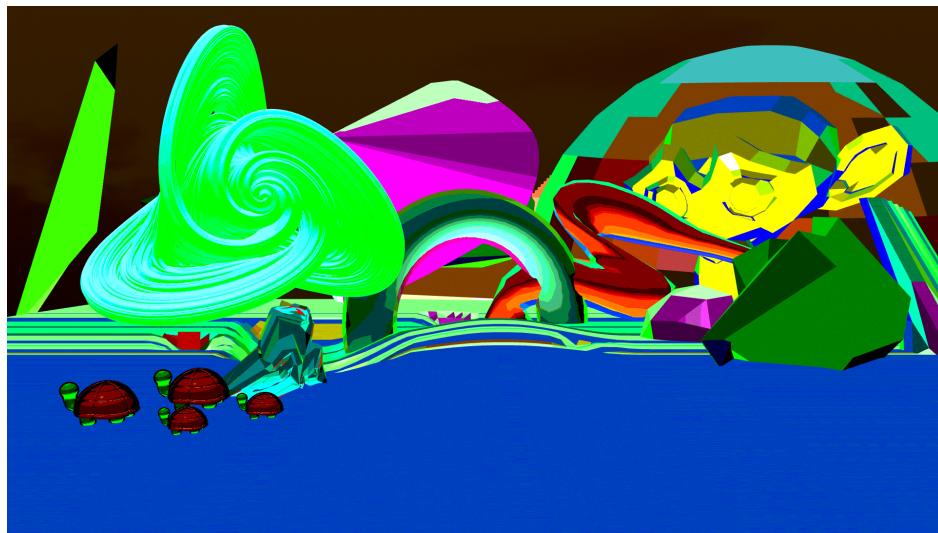


Figure 21: La Zone Weird version shaders

## 12 Le montage

### 12.1 Render

Nous avons d'abord tenté d'enregistrer image par image en utilisant la fonction TakeScreenshot, puis le module FFmpeg. Malheureusement, FFmpeg ne fonctionnait pas sur Windows. Nous avons donc opté pour un enregistrement via OBS de qualité légèrement inférieure, comme cela a été fait par une partie de la classe.

### 12.2 Recadrage

L'enregistrement via OBS n'étant pas le dispositif optimal pour un rendu, un contour blanc apparaissait autour de notre court-métrage. Nous avons donc recadré la vidéo dans DaVinci Resolve afin de lui donner un aspect plus professionnel.

### 12.3 Sound effects

Nous avons utilisé la librairie gratuite BBC Sound Effects[9] pour ajouter des légers bruits de fond pour donner plus de vie à notre monde. Nous conseillons cette base de données accessible à tous et fruit de plus de 100 ans de fiction sonore de la compagnie britannique.

### 12.4 Crédits

Nos crédits ont été réalisés à partir d'un rendu Blender relativement simple, complété par un travail sur un carton dans Photoshop. Nous avons ensuite animé ce carton en utilisant les keyframes de rotation de DaVinci Resolve. De même, le pingouin est issu d'un rendu Blender.

## 13 Conclusion

Ainsi se termine notre projet d'animation La Famille Tortue, désormais disponible sur YouTube : La Famille Tortue. Nous avons beaucoup apprécié ce projet, malgré le temps qu'il nous a demandé pour un résultat qui pourrait paraître anodin pour la plupart des spectateurs. Il nous a permis de mesurer le chemin parcouru dans le domaine de l'animation et, surtout, d'apprécier le travail acharné de tous les professionnels de l'informatique graphique, tant dans le domaine artistique que technique.

## 14 Bibliographie

### References

- [1] Alexandre Thepot Cover Rock de La Famille Tortue
- [2] Making-Off de Spider-Punk
- [3] Alfred Hitchcock, Dolly Zoom du film *Vertigo*.
- [4] Steven Spielberg, Dolly Zoom du film *Jaws*.
- [5] Documentation du `floor`.
- [6] Documentation pour le `noise`.
- [7] Documentation pour le `mix`.
- [8] Thèse sur les shaders (TCD, 2009).

[9] Base de données *BBC Sound Effects*.