INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

Informatique Graphique - Projet

Tortue Kaizen

GRAINDORGE Amance

VERNET Hector

Université de Rennes

Table des matières

Introduction . . . 1
Animation . . . 1
Modes d'interpolation . . . 1
Import depuis Blender . . . 2
Objets . . . 3
Fichiers `.animation` . . . 3
Import . . . 5
Matériaux avec transparence . . . 6
Post-processing . . . 7
Simulation . . . 7
Audio . . . 10
Conclusion . . . 11

Introduction

Dans ce rapport, nous allons détailler la réalisation de notre projet d'informatique graphique et, plus spécifiquement, les éléments techniques que nous avons ajouté au code fourni pour obtenir le résultat final.

L'animation ne comporte aucun montage, elle peut donc être lancée directement depuis la racine de ce repo avec `./make\&run.sh tortuekaizen`. Un enregistrement est également disponible sur Youtube : https://youtu.be/EZeGAYSm5xw.

Le fichier source principal de l'animation est `sampleProject/tortuekaizen.cpp`.

Animation

Pour ce projet, nous avons développé quelques outils à la fois pour améliorer la pipeline d'animation existante, mais aussi pour pouvoir animer plus efficacement. Grâce à ces outils, nous avons pu créer une animation assez complexe d'1min40s, ce qui aurait été impossible dans le laps de temps imparti sans eux.

Modes d'interpolation

La pipeline existante comporte une interpolation linéaire. Cela fonctionne pour des mouvements simples, mais dès qu'on veut avoir des personnages et une caméra qui bougent de façon organique, ça ne suffit clairement plus. L'interpolation la plus flexible que nous pourrions implémenter est celle de Bézier, qui permet de contrôler précisément la courbe d'interpolation en ajoutant des points de contrôle aux keyframes.

Cependant, cette méthode est assez lourde à mettre en place, et gérer les points de contrôle pour chaque keyframe aurait été fastidieux. Nous avons donc opté pour un compromis avec l'interpolation cubique. Cette interpolation n'utilise pas de points de contrôle, mais elle prend en compte les quatre keyframes les plus proches du point actuel, contrairement à l'interpolation linéaire qui n'en utilise que deux.

Nous avons également implémenté une interpolation constante, qui n'est pas vraiment une interpolation, mais qui permet de faire des cuts propres sans avoir quelques frames où la caméra vole entre deux positions.

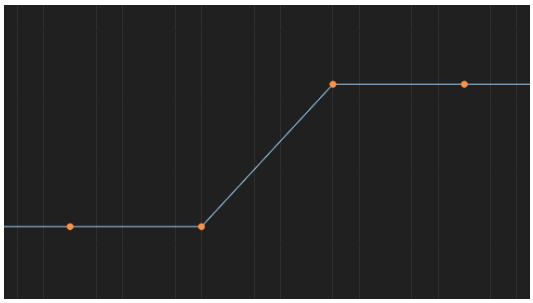

Figure 1 : Interpolation linéaire.

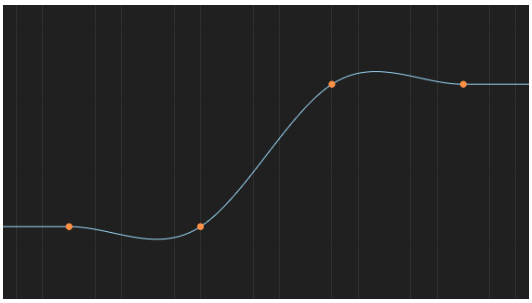

Figure 2 : Interpolation cubique.

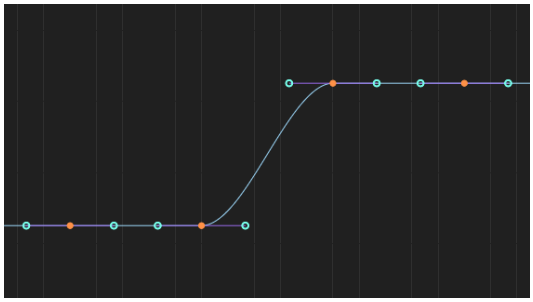

Figure 3 : Interpolation Bézier.

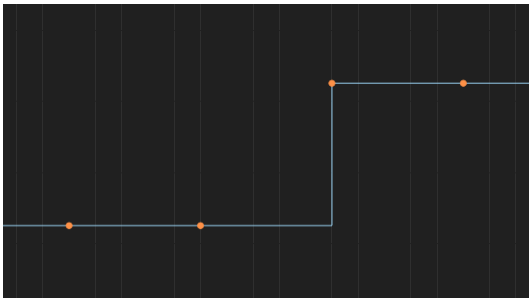

Figure 4 : Interpolation constante.

Pour spécifier le mode d'interpolation, on déclare la structure `Keyframe` dans `KeyframeCollection.h` comme suit :

```
1   typedef enum KeyframeInterpolationMode
2   {
3     LINEAR,
4     CUBIC,
5     CONSTANT
6   } KeyframeInterpolationMode;
7
8   struct Keyframe
9   {
10    float time;
11    GeometricTransformation transform;
12    KeyframeInterpolationMode interpolation;
13  };
```

Ensuite, dans `KeyframeCollection::interpolateTransformation`, on fait un switch sur le mode d'interpolation de la keyframe précédant l'instant actuel, en implémentant chaque mode d'interpolation. Cette méthode étant assez longue, nous vous invitons à aller la voir directement dans `KeyframeCollection.cpp`.

Import depuis Blender

Le scénario que nous avons écrit étant relativement ambitieux en terme de longueur, il serait intéressant de pouvoir animer nos objets dans un logiciel adapté comme Blender, ce qui permettrait d'aller infiniment plus vite que d'écrire des keyframes à la main. C'est donc la fonctionnalité que nous avons développée dans un script Python utilisant l'API de Blender `bpy`. Cette API permet d'accéder à toutes les données d'une scène Blender, et plus particulièrement les objets et leurs animations, ce qui est parfait pour notre application.

Ce script est disponible dans `Blender/Autres/export.py`.

Objets

On va commencer par exporter tous les objets de la scène en format Wavefront `.obj`. Attention toutefois, une complication de ce format est qu'il ne fait pas de séparation entre les transformations globales et locales. En d'autre termes, il ne permet pas d'avoir un origine différent de celui du monde. Pour rémédier à cela, avant d'exporter chaque objet, on annule sa tranformation globale, qu'on enregistre séparément des positions des sommets dans le même fichier.

Il y a une autre subtilité dûe au fait que dans Blender, le haut est l'axe Z+, tandis que dans notre pipeline, c'est l'axe Y+. On peut rémédier à cela en appliquant une matrice de rotation avant d'exporter chaque objet.

```python
def export_mesh(obj):
    for ob in bpy.data.objects: # Deselect all other objects just to be sure
        ob.select_set(False)
    obj.select_set(True)
    with open(f"{obj_path}/{obj.name}.obj", "w"):
        pass
    bpy.ops.wm.obj_export(
        filepath=f"{obj_path}/{obj.name}.obj",
        apply_modifiers=True,
        export_selected_objects=True,
        export_materials=False,
        apply_transform=False
    )
    y_up_rot = mathutils.Matrix((
        (1, 0, 0, 0),
        (0, 0, 1, 0),
        (0, -1, 0, 0),
        (0, 0, 0, 1)
    ))
    if obj.parent is not None: # Subtract parent transform, we will handle that ourselves
        parent_matrix = obj.parent.matrix_world
        inv_parent_matrix = parent_matrix.inverted()
        y_up_matrix = y_up_rot @ inv_parent_matrix @ obj.matrix_world @ y_up_rot.inverted()
    else:
        y_up_matrix = y_up_rot @ obj.matrix_world @ y_up_rot.inverted()
    with open(f"{obj_path}/{obj.name}.obj", "a") as f:
        f.write("\nTRANSFORM") # Write global transform directly at the end of the file
        for row in y_up_matrix:
            for val in row:
                f.write(f" {val}")
        f.write("\n")
    obj.select_set(False)
```

La fonction `bpy.ops.wm.obj_export` (dont la documentation est ici) effectue le changement vers Y+ elle même, mais il faut tout de même le faire pour la transformation globale.

Fichiers `.animation`

L'animation 3D est un vaste domaine, et en conséquence les formats ayant pour but de l'exporter sont très complexes. Nous avons donc opté pour créer notre propre format, comprenant simplement un repère temporel, le type de keyframe, et la matrice de transformation globale pour chaque keyframe.

La structure de données d'animation dans Blender est également très complexe, donc la fonction `export_animation` est plus imbuvable que celle d'export des objets. L'approche globale est celle-ci : pour un objet, on stocke les temps et les modes d'interpolation de toutes ses keyframes, puis on parcourt l'animation en enregistrant la transformation globale de l'objet à chaque temps de keyframe.

```
1   def export_animation(obj):
2       output = ""
3       anim_data = obj.animation_data
4       if anim_data is not None and anim_data.action is not None:
5           action = anim_data.action
6           slot = anim_data.action_slot
7           if slot is None:
8               slot = anim_data.action_suitable_slots[0]
9               anim_data.action_slot = slot
10          channelbag = anim_utils.action_get_channelbag_for_slot(action, slot)
11          frames = {}
12          # Gather all the interpolation modes for all the keyframes
13          for fcurve in channelbag.fcurves:
14              for keyframe in fcurve.keyframe_points:
15                  frame = int(keyframe.co.x)
16                  interp = keyframe.interpolation
17                  if frame not in frames:
18                      frames[frame] = [interp]
19                  else:
20                      frames[frame].append(interp)
21          scene = bpy.context.scene
22          depsgraph = bpy.context.evaluated_depsgraph_get()
23          prev_rot = None
24          for fr in sorted(frames.keys()):
25              # Go to the frame in question
26              scene.frame_set(fr)
27              eval_obj = obj.evaluated_get(depsgraph)
28              loc, rot, scale = eval_obj.matrix_world.decompose()
29              # Subtract parent transform if present
30              if obj.parent is not None:
31                  parent_eval = obj.parent.evaluated_get(depsgraph)
32                  parent_matrix = parent_eval.matrix_world
33                  inv_parent_matrix = parent_matrix.inverted()
34                  local_matrix = inv_parent_matrix @ eval_obj.matrix_world
35                  loc, rot, scale = local_matrix.decompose()
36              if obj.type == 'CAMERA': # Cameras don't point in the same direction in
    sfml and Blender
37                  rot = rot @ mathutils.Euler((math.radians(-90), 0, 0),
    'XYZ').to_quaternion()
38              # Ensure quaternion continuity to avoid flipping
39              if prev_rot is not None and rot.dot(prev_rot) < 0:
40                  rot = -rot
41              prev_rot = rot.copy()
42              t = frame_to_time(fr)
43              # For a keyframe with multiple interpolation modes, choose the most
    frequent one
44              most_frequent_interp = max(set(frames[fr]), key=frames[fr].count)
45              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"
46          if output != "":
```

```
47                 with open(f"{anim_path}/{obj.name}.animation", "w") as f:
48                     f.write(output)
```

De manière un peu arbitraire, pour l'animation, on applique la transformation en Y+ up non pas lors de l'export, mais lors de l'import dans notre pipeline.

Import

Nous n'irons pas dans le détail de l'import des fichiers `.obj` modifiés et `.animation` dans ce rapport, mais vous pouvez retrouver les modifications apportées dans la fonction `HierarchicalRenderable::applyObjTransform` du fichier `HierarchicalRenderable.cpp`, ainsi que dans la fonction `KeyframeCollection::addFromFile` dans `KeyframeCollection.cpp`.

Dans notre projet en particulier, nous avons défini des fonctions `add_object` et `add_textured_object` pour ajouter facilement un objet et son animation dans la scène en une ligne. Ces fonctions peuvent aussi ajouter un objet en tant que fils d'un autre.

```
1   LightedMeshRenderablePtr add_object(Viewer& viewer,
2                                       const std::string& name,
3                                       const MaterialPtr& material,
4                                       ShaderProgramPtr& shaderProgram,
5                                       HierarchicalRenderablePtr parent = nullptr)
6   {
7     std::string obj_path = "../ObjFiles/" + name + ".obj";
8     std::ifstream file(obj_path);
9     if (!file.good())
10    {
11      std::cerr << "Error: File " << obj_path << " does not exist." << std::endl;
12      return nullptr;
13    }
14    auto obj = std::make_shared<LightedMeshRenderable>(shaderProgram, obj_path,
    material);
15    if (parent != nullptr)
16    {
17      HierarchicalRenderable::addChild(parent, obj, true);
18    }
19    std::string anim_path = "../Animation/" + name + ".animation";
20    std::ifstream anim_file(anim_path);
21    if (anim_file.good())
22    {
23      obj->addKeyframesFromFile(anim_path, 0.0, false);
24    }
25    viewer.addRenderable(obj);
26
27    return obj;
28  }
29
30  TexturedLightedMeshRenderablePtr add_textured_object(Viewer& viewer,
31                                                       const std::string& name,
32                                                       const MaterialPtr& material,
33                                                       const std::string& texture_path,
34                                                       ShaderProgramPtr& shaderProgram,
35                                                       HierarchicalRenderablePtr parent
    = nullptr)
36  {
37    std::string obj_path = "../ObjFiles/" + name + ".obj";
```

```
38      std::ifstream file(obj_path);
39      if (!file.good())
40      {
41        std::cerr << "Error: File " << obj_path << " does not exist." << std::endl;
42        return nullptr;
43      }
44      auto obj = std::make_shared<TexturedLightedMeshRenderable>(shaderProgram, obj_path,
      material, texture_path);
45      if (parent != nullptr)
46      {
47        HierarchicalRenderable::addChild(parent, obj, true);
48      }
49      std::string anim_path = "../Animation/" + name + ".animation";
50      std::ifstream anim_file(anim_path);
51      if (anim_file.good())
52      {
53        obj->addKeyframesFromFile(anim_path, 0.0, false);
54      }
55      viewer.addRenderable(obj);
56
57      return obj;
58    }
```

Matériaux avec transparence

Pour avoir des objets transparents, ce qui était une nécessité pour une scène en particulier dans laquelle on voit des tortues sur la plage depuis le dessous de la mer, nous avons ajouté une propriété `alpha` à la classe `Material`. Une fois cette propriété passée aux shaders avec `program->getUniformLocation("material.alpha")` et `glUniform1f` comme pour les autres propriétés, on peut l'utiliser à la fin du fragment shader ainsi :

```
1   outColor = vec4(tmpColor,material.alpha);
```

Cependant, il reste un problème : si un objet transparent est dessiné avant un objet opaque qui se trouve derrière lui, l'objet opaque ne sera pas visible. La solution est de déterminer quels objets sont transparents et de les dessiner en dernier. On rajoute donc quelques lignes dans `Viewer::draw()` :

```
1   // Separate opaque and transparent objects
2   std::vector<RenderablePtr> opaque_renderables;
3   std::vector<RenderablePtr> transparent_renderables;
4
5   for (const RenderablePtr &r : m_renderables)
6   {
7     LightedMeshRenderablePtr lm = std::dynamic_pointer_cast<LightedMeshRenderable>(r);
8     if (lm != nullptr && lm->getMaterial()->alpha() < 1.0f)
9     {
10      transparent_renderables.push_back(r);
11    }
12    else
13    {
14      opaque_renderables.push_back(r);
15    }
16  }
17
```

```
18   // We draw the opaque objects first so that they can always appear behind the
     transparent ones
19   std::vector<RenderablePtr> sorted_renderables;
20   sorted_renderables.insert(sorted_renderables.end(), opaque_renderables.begin(),
     opaque_renderables.end());
21   sorted_renderables.insert(sorted_renderables.end(), transparent_renderables.begin(),
     transparent_renderables.end());
```

On peut maintenant créer le matériau `Water` avec un `alpha` de 0.7, et la scène s'affiche comme prévu.

Figure 5 : Tortues terrestres vues par dessous la surface de l'eau.

Post-processing

Depuis le début du projet, la question du post-processing se posait : la moitié de l'animation se passant sous l'eau, il semble assez important de le montrer avec, pourquoi pas, un effet de « brouillard » bleu.

Cependant, ajouter des shaders de post-processing avec SFML s'est avéré complexe et nous avons trouvé une solution suffisante au vu de la qualité visuelle globale de notre animation : utiliser la transparence précédemment détaillée pour placer un filtre bleu directement devant la caméra. On crée donc un plan auquel on assigne un matériau bleu transparent et on l'apparente à la caméra en copiant sa position à chaque image de l'animation où on veut appliquer le filtre :

```
1   if (viewer.getTime() >= 2.6666f && viewer.getTime() <= 66.7f) {
2     filter->setGlobalTransform(viewer.getCamera().getGlobalTransform());
3   } else {
4     filter->setGlobalTransform(getTranslationMatrix(0.0, -50000.0, 0.0)); // Hide filter
    by moving it very very far
5   }
```

Simulation

La scène de fin du projet nécessitait de montrer une explosion, que nous avons décidé de simuler avec des particules. On utilise donc les classes `Particle` et `ParticleListRenderable` pour gérer et afficher les particules. Les particules sont initialiées avec une position aléatoire dans une sphère, une vitesse nulle, un poids suivant une distribution gaussienne et un rayon aléatoire entre 0.05 et 0.1.

```
1   std::vector<ParticlePtr> particles;
2   const unsigned int count = 500u;
3   for (unsigned int i = 0; i < count; ++i)
4   {
5     glm::vec3 pos = glm::ballRand(0.35f) + glm::vec3(-14.5f, -1.0f, 16.0f);
6     pos.y += 0.35f;
```

```
    float const weight = std::max(glm::gaussRand(20.0f, 5.0f), 10.0f);
    float const radius = glm::linearRand(0.05f, 0.1f);
    ParticlePtr p = std::make_shared<Particle>(pos, glm::vec3(0.0f), weight, radius);
    particles.push_back(p);
    system->addParticle(p);
  }
```

L'intérêt de ces paramètres initiaux est que les particules se répartissent de manière naturelle sur le plan de la simulation.

La classe `DynamicSystemRenderable` a été modifiée pour pouvoir spécifier un temps de début de la simulation :

```
auto systemRenderable = std::make_shared<DynamicSystemRenderable>(system);
systemRenderable->setStartTime(92.87f);
viewer.addRenderable(systemRenderable);
```

L'explosion en elle-même est simulée avec deux champs de force, en plus de la gravité et du damping: un champ de force radial qui pousse les particules vers l'extérieur par rapport à un point, et un champ de force qui donne une forme de champignon à l'explosion. Les deux nouvelles classes `RadialImpulseForceField` et `MushroomForceField` héritent de `ForceField`, et possèdent leur propre implémentation de la méthode `do_addForce()`.

RadialImpulseForceField

```
void RadialImpulseForceField::do_addForce()
{
  if (!m_active || m_remainingTime <= 0.0f)
  {
    m_active = false;
    return;
  }

  // La force diminue linéairement avec le temps
  float timeFactor = (m_duration > 0.0f) ? (m_remainingTime / m_duration) : 1.0f;

  for (auto& p : m_particles)
  {
    glm::vec3 dir = p->getPosition() - m_center;
    float dist = glm::length(dir);
    if (dist < 1e-5f)
    {
      continue;
    }

    dir /= dist;
    // La force diminue de façon gaussienne avec la distance
    // pour éviter des effets trop brusques
    float spatialFactor = std::exp(-(dist * dist) / (m_radius * m_radius));
    glm::vec3 force = dir * (m_strength * spatialFactor * timeFactor);
    p->incrForce(force);
  }

  m_remainingTime -= m_dt;
  if (m_remainingTime <= 0.0f)
```

```
    {
      stop();
    }
  }
```

MushroomForceField

```
void MushroomForceField::do_addForce()
{
  if (!m_active)
    return;

  // On simule un vortex toroïdal centré en (m_center.x, m_center.y + m_ringHeight, m_center.z) avec un rayon majeur m_ringRadius.

  float coreY = m_center.y + m_ringHeight;
  // Sigma contrôle la "largeur" du vortex
  float sigma = m_ringRadius * 0.5f;
  float sigmaSq = sigma * sigma;

  for (auto& p : m_particles)
  {
    glm::vec3 pos = p->getPosition();
    glm::vec3 local = pos - m_center;

    // Distance radiale dans le plan horizontal (x, z)
    float r = glm::length(glm::vec2(local.x, local.z));

    // Vecteur depuis le centre du vortex dans le plan (r, y)
    float dr = r - m_ringRadius;
    float dy = pos.y - coreY;

    // Distance au carré depuis le centre du vortex
    float distSq = dr * dr + dy * dy;

    // Direction de la circulation : tangente au cercle autour du noyau
    // On veut que le flux monte au centre (r < R) -> dy positif
    // Vecteur depuis le noyau : (dr, dy)
    // Perpendiculaire (tangente) : (dy, -dr)
    glm::vec2 tangent(dy, -dr);

    // La force diminue de façon gaussienne avec la distance au centre du vortex
    float intensity = m_strength * std::exp(-distSq / sigmaSq);

    // Composantes radiale et verticale
    float fRadial = tangent.x * intensity;
    float fVertical = tangent.y * intensity;

    // Application d'un facteur de vitesse
    fRadial *= m_radialSpeed;
    fVertical *= m_riseSpeed;

    // Conversion en repère cartésien
    glm::vec3 force(0.0f);
    if (r > 1e-5f)
    {
```

```
49          force.x = (local.x / r) * fRadial;
50          force.z = (local.z / r) * fRadial;
51        }
52        force.y = fVertical;
53
54        p->incrForce(force);
55      }
56    }
```

Pour se représenter plus facilement l'effet de ce champ de force :

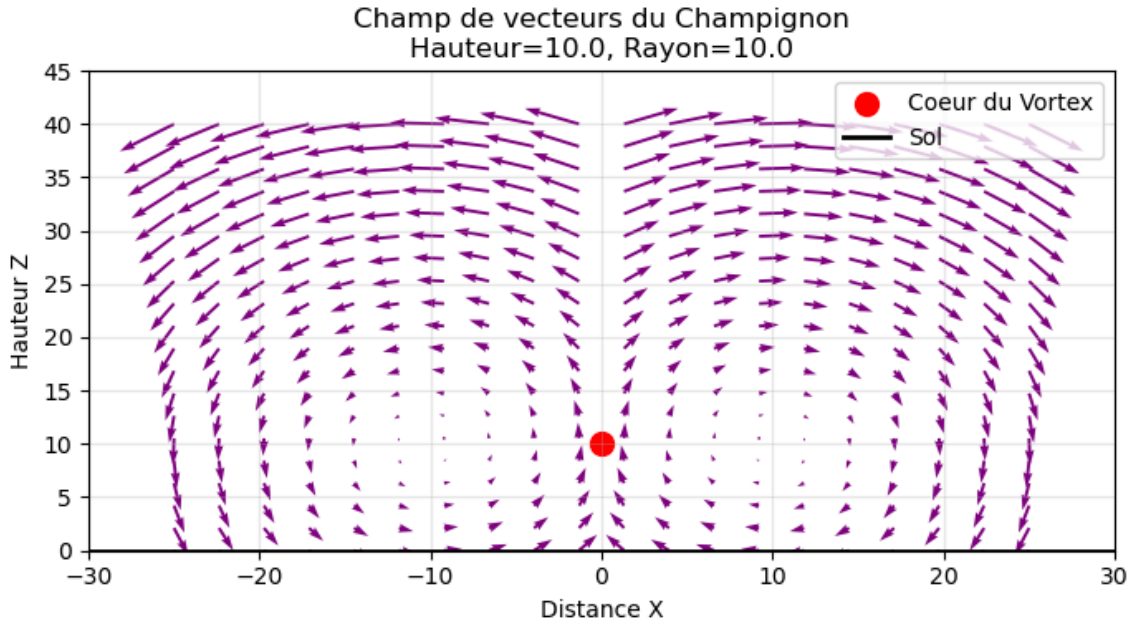

Figure 6 : Graphique illustrant l'influence du champ de force en forme de vortex toroïdal sur les particules.

Les deux champs de force possèdent également une méthode `trigger()` et une méthode `stop()` pour démarrer et arrêter leur effet. On peut ainsi déclencher l'explosion à un moment précis dans l'animation. Nous avons ensuite ajusté les paramètres des champs de force pour obtenir un rendu satisfaisant.

Nous nous sommes cependant heurtés à une limite de performance : avec un grand nombre de particules, la simulation devenait très lente à cause du calcul des interactions entre particules. Nous avons donc limité le nombre de particules à 500 pour maintenir une fluidité acceptable. Une amélioration possible serait d'utiliser le GPU pour effectuer les calculs de ces interactions, ou alors de trouver une méthode plus efficace pour répartir les particules de façon naturelle et désactiver les collisions.

Audio

Pour l'audio du projet, nous utilisons l'utilitaire linux `aplay` qui joue un fichier `.wav` contenant toute notre bande son. Cette commande est lancée automatiquement dans un nouveau processus au lancement de l'animation par `Viewer.cpp`.

```
1   void Viewer::startAnimation()
2   {
3     m_lastSimulationTimePoint = clock::now();
4     m_animationIsStarted = true;
5     if (!soundtrack_path.empty()) {
```

```
6       std::string cmd = "aplay \"" + soundtrack_path + "\" &";
7       std::system(cmd.c_str());
8     }
9   }
```

Cependant, après avoir ajouté notre bande son, nous avons remarqué que l'animation et l'audio se décalaient très légèrement au fil du temps. Nous n'avons pas vraiment trouvé la source de ce décalage minime. Nous avons simplement ajouté une propriété `m_timeFactor` dans la class `Viewer`. Ce facteur est simplement multiplié par le temps réel à chaque appel de `Viewer::getTime()`:

```
1   return m_simulationTime * m_timeFactor;
```

Ainsi, on peut modifier la vitesse de toute l'animation arbitrairement. Par l'expérimentation, nous avons déterminé que `1.004` est la meilleure valeur de ce facteur pour synchroniser l'animation et l'audio.

Conclusion

En conclusion, ce projet nous a permis de nous familiariser avec plusieurs aspects de l'informatique graphique, notamment la modélisation, l'animation et la simulation. Le résultat final est une animation qui raconte une histoire et qui est tout à fait regardable, avec même quelques gags visuels, ce projet est donc un succès.

INSA Rennes
20 avenue des Buttes de Coësmes
CS 70839
35708 Rennes cedex 7
Tél : + 33 (0)2 23 23 82 00
www.insa-rennes.fr

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR ET DE LA RECHERCHE
Liberté
Égalité
Fraternité