Documentation de Fin de Stage FARI

Swift et Enable3D

Github

Librairies utilisées

Swift: Base du simulateur
Enable3D: Utilisé pour ajouter les lois de la physique à la simulation
Gaussian Splatting Viewer by Mkkellogg: Utilisé pour visualiser une gaussian splatting dans le simulateur

Création de la corde

Il existe trois méthodes principales pour créer une corde :

- **Patch :** Cette méthode permet de créer une surface avec quatre points d'ancrage. En réduisant la largeur de cette surface, on obtient un effet visuel similaire à celui d'une corde.
- **Rope :** Cette méthode permet de créer directement une corde dans la simulation.
- **Addition de RigidBody :** Cette méthode consiste à lier plusieurs rigidBody (par exemple des sphères) entre eux à l'aide de contraintes. Les RigidBody ne traversant pas les autres rigidBody, cela permet de créer une corde qui ne passe pas au travers d'autres éléments de la simulation.

Cependant, chacune de ces méthodes présentent des inconvénients :

- **Patch :** Bien que cette méthode puisse donner l'effet visuel d'une corde, elle est moins réaliste qu'une corde créée directement.
- **Rope :** Le principal problème avec cette méthode est que la corde passe souvent à travers les RigidBody, ce qui complique la montée de la corde et réduit son réalisme.
- **Addition de RigidBody :** Malgré les contraintes appliquées (comme les contraintes de charnière (hinge) => interdiction de translation), les rigidBody ont tendance à s'éloigner significativement les uns des autres lorsque l'on applique une force à l'extrémité de la "corde". Cela affecte la cohésion de l'ensemble.

La solution retenue a été d'utiliser la méthode **Rope** tout en augmentant la masse des RigidBody avec lesquels la corde entre en collision. Cette approche semble réduire la fréquence à laquelle la corde traverse les RigidBody, probablement en raison de l'augmentation de la densité des RigidBody, ce qui rend plus difficile le passage de la corde à travers eux.

Activation de l'affichage du gaussian splatting

Pour activer la visualisation du gaussian splatting il faut mettre la variable `splat` a **true** dans le fichier `swift/swift/public/js/index.js`.

Modifications principales des librairies

Swift: Modification entière de la partie en JavaScript
Enable3D: Passage de la librairie en Z up pour correspondre à la convention et pour l'utiliser avec Swift (qui est en Z up par default)
Gaussian Splatting: Passage de la librairie en Standalone pour l'utilisation avec Swift

Prochaines étapes

Les deux améliorations possibles pour le projet sont :

- Rendre le rendu du gaussian splatting fixe. Pour l'instant quand on essaie de bouger dans la simulation, on ne fait que bouger le rendu du gaussian splatting. OrbitControl est comme désactivée.
- Mettre à jour régulièrement la librairie de Mkkellogg pour avoir les dernières modification au niveau des performances pour avoir un meilleur rendu (le rendu lag énormément actuellement)

Gaussian Splatting

Pour la réalisation d'une Gaussian splatting nous avons utilisé le modèle suivant : Github

Prérequis

Pour faire fonctionner le modèle il faut des versions spécifique de librairie / packages sinon le code ne fonctionnera pas.

```
Cuda == 11.6
    | Pytorch version compatible avec Cuda 11.6 (https://pytorch.org/get-started/previous-versions/)
    | Drivers Nvidia >= 510
Colmap == 3.8
```

Réalisation d'une Gaussian splatting

Le modèle utilisé utilise directement des images. La manière la plus simple de réaliser ces images est de prendre une vidéo de la même façon que lorsque l'on fait un scan d'un object 3D avec un Apple. Il faut tourner autour de notre object de façon assez lente pour ne pas avoir d'image flou et de sorte à avoir une plus de face possible (prendre plusieurs angles)

Ensuite, on peut séparer cette vidéo en images via le script suivant :

split.sh

```sh
#!/bin/bash

# Check if the correct number of arguments are passed
if [ -z "$1" ]; then
    read -p "Please provide the input file : "  input
else
    input="$1"
fi
if [ ! -f "$input" ]; then
  echo "file not found : $input"
  exit 1
fi

# Input video file
input_video=$1


# Convert the video to 1 FPS frames
ffmpeg -i "$input" -qscale:v 1 -qmin 1 -vf fps=1 %04d.jpg
```

```
if [ $? -eq 0 ]; then
    echo "video conversion successful"
else
    echo "video conversion failed"
fi
```

```
./split <videoFileName>
```

Ce script permet de convertir une vidéo en une suite d'image (à 1 Fps), il faut avoir FFMPEG d'installé sur l'ordinateur et donner le chemin d'accès vers la vidéo en input.

Ensuite il faut utiliser le code convert.py (qui se trouve sur le github précédent)

Ce script permet de 'preparer' les images pour l'entrainement du modèle. Il faut que les images respectent la position suivante :

```
python convert.py -s Fari

|---Fari
    |---input
        |---<image 0>
        |---<image 1>
        |---...
```

Après que la conversion a été réalisée :

```
python train.py -s input
```

En fonction du nombre d'images et de leurs qualité, cela peut prendre plus ou moins de temps (30 min à 1h)

Lorsque l'entrainement est terminé, on obtiens dans le dossier Output (au même niveau que les scripts convert.py et train.py).

Dans ce dossier se trouve un ou plusieurs dossiers (tout dépend combien de gaussian splatting ont été réalisé) qui a le nom du dossier où les images se trouvaient (Fari dans notre exemple).

Ce dossier contient les .ply (en fonction du nombre d'itérations), ce sont des nuages de point que l'on peut visionner via le code de Mkkellogg.

On peut aussi visionner le rendu via le viewer de l'INRIA (SIBR)

La commande est dans notre exemple la suivante :

```
sudo MESA_GL_VERSION_OVERRIDE=4.5
./SIBR_viewers/install/bin/SIBR_gaussianViewer_app -m
```

```
/home/fari/Documents/gaussian-splatting/output/Fari
```

`MESA_GL_VERSION_OVERRIDE=4.5` permet de ne pas avoir d'erreur lié à la version d'OpenGL

Visualisation dans enable3D

Pour cette partie, nous avons utilisé le code de Mkkellogg comme indiqué précédemment.

Plusieurs problèmes ont été rencontrés lors de l'utilisation du code :

- Pour que l'affichage fonctionne correctement en mode standalone (sans utiliser les commandes npm), il est nécessaire de définir le paramètre **sharedMemoryForWorkers** à **false**.
- Selon le navigateur utilisé, il peut être nécessaire de désactiver CORS (Cross-Origin Resource Sharing).

MuJoCo

Github

Librairies utilisées

- numpy==1.26
- mujoco==latest
- spatialmath==latest
- roboticstoolbox-python==this version

Selection du Robot

Modification de la variable `robot` dans le fichier `model/config.json`

`mycobot` pour le robot **mycobot** de Elephant Robotics

`lite6` pour le robot **lite6** de ufactory

Utilisation de MuJoCo avec le Lite6 et le MyCobot

MuJoCo utilise des fichiers .stl pour l'affichage des meshes et des fichiers XML/MJCF pour la contruction des robots (ce qui est fait généralement en URDF).

Lite6

Pour le robot Lite6, on a trouvé directement le fichier xml correspondant : MuJoCo Menagerie GitHub.

MyCobot

Pour le robot mycobot, il a fallu convertir le fichier URDF en fichier XML (MJCF). On réalise ceci en utilisant le script suivant :

```
cd bin
./compile [path to your file] [file path]
```

Créer une corde dans MuJoCo

Dans MuJoco il y a deux façons de créer une corde ou presque-corde:

Rope: Ceci permet de créer directement une corde mais cette technique a 2 inconvénients (pour ce que l'on désire faire):

La corde passe facilement au travers d'un rigidBody (comme sous enable3D).
Dans notre cas, elle ne peut pas être enroulé plus d'une fois, si la corde s'enroule sur elle même, elle fait

buger les collisions qui lui sont appliqués et peut passer au travers de la poulie.

Tendon: Cette solution est très bien quand l'on veut juste une corde pour attacher 2 objets entre eux car cela ne créer pas une corde via une multitude de cylindre mais cela créer directement un lien entre les deux objets. Cependant, cette "corde" passe au travers de tout objets qui n'est pas l'un deux deux objets où elle est attaché.

Au final, nous avons choisi la Rope car elle permet d'avoir quand même des collisions malgré certains soucis. Nous tirons la corde au lieu de l'enrouler (pour éviter le problème évoqué précédemment).

Les différents solvers de MuJoCo

Il y a 3 types de solvers différents sous MuJoCo: Newton, CG, and PGS.

Newton: L'avantage principal de ce solver est qu'il converge très rapidement (en 2 / 3 itérations). Cependant, observation personnelle, j'ai l'impression que l'on obtient plus d'erreur lié à la physique avec ce solver que les deux autres.

CG: On utilise actuellement le solver CG avec 1000 itérations. C'est le solver qui prend le moins de capacité du CPU.

PGS: Je n'ai pas vu de différences entre le solver CG et PGS au niveau de la physique.

Réalisation d'un simulateur temps réel

Pour créer un simulateur efficace, la performance en temps réel est essentielle, ce qui signifie qu'une seconde dans le monde réel doit correspondre à une seconde dans le simulateur. Pour y parvenir, il faut un traitement en temps réel.

Étant donné que notre principal langage de programmation est Python, un langage interprété plutôt que compilé, sa vitesse d'exécution est nettement plus lente que celle de C/C++.

Actuellement, nous utilisons une simple instruction de correspondance pour mettre en œuvre une machine à états. En plaçant cette machine à états dans une boucle while (qui s'exécute indéfiniment tant que la fenêtre du simulateur est ouverte), nous pouvons passer d'un état à l'autre de manière séquentielle.

À la fin de chaque boucle, nous nous synchronisons avec le visualisateur MuJoCo, nous mettons à jour l'état de la simulation (y compris le modèle et les données) et nous introduisons un délai relatif entre chaque itération de la boucle. Ce délai est calculé comme la différence entre le temps d'attente maximal et le temps d'exécution réel de la boucle.