Ecole Polytechnique Universitaire de Lille

Projet IMA3/4 2018/2020

Rapport Projet SimulPhys

Corto Callerisa
Alex Lagneau
Sébastien Dardenne
Quentin Normand

Soutenance le 5 mai 2020

Remerciements

Nous tenons à remercier dans un premier temps, toute l'équipe pédagogique de l'école universitaire Polytech Lille et les responsables de la formation Informatique, Microélectronique et Automatique.

Nous tenons également à remercier Monsieur Redon et Monsieur Conrard pour l'encadrement et les conseils donnés pendant le projet.

Enfin, nous remercions nos professeurs pour avoir mis à disposition leurs expériences et leurs compétences.

Introduction

Dans un contexte économique où pour chaque entreprise, réduire les coûts de conception et de production est nécessaire afin d'être compétitif sur le marché, la simulation physique sur ordinateur s'impose comme un outil extraordinaire pour faire des premiers prototypes sans avoir à dépenser une fortune en crash-test. C'est dans ce contexte que nous avons choisi de réaliser un simulateur configurable de processus physique destiné à la conception et au développement de systèmes automatisés, s'implémentant aussi dans le concept d'entreprise 4.0.

Ce simulateur a pour but de pouvoir créer et remplacer les maquettes de TP en rendant les simulations compatibles avec un automate industriel programmable. C'est pourquoi nous avons choisi de pouvoir reproduire dans un premier temps l'ascenseur avec les objets présent dans une bibliothèque.

Syntaxe du projet au semestre 6

Tout d'abord, les objectifs du semestre 6 étaient de reproduire un système de la salle de TP d'automatique, nous avons choisi de reproduire l'ascenseur. Pour modéliser au mieux notre simulation, nous avons choisi Godot dû à sa facilité de prise en main et sa portabilité sur différentes plateformes. Grâce à cet outils, nous avons reproduit l'ascenseur sur 5 étages avec des capteurs et des actionneurs.

Ensuite, nous avons implémenté une communication en UDP afin de pouvoir contrôler l'ascenseur. La communication se fait entre 3 parties indépendantes s'inspirant du modèle MVC (Modèle-vue-contrôleur). La commande s'effectue depuis un script python avec une invite de commande, la vue est la fenêtre de simulation de Godot qui héberge aussi le modèle.

Syntaxe du projet au semestre 7

Au commencement du semestre 7, nous avons choisi de créer le simulateur en le contrôlant avec un automate programmable. Pour la partie simulation, nous avons créé les bases du simulateur avec la création de l'application qui contient la bibliothèque d'objets, la scène 3D, l'arbre des objets créés, les propriétés de l'objet sélectionné et l'affichage des modifications.

Pour la partie communication, nous avons eu des difficultés à faire fonctionner l'automate en le connectant à notre simulation. Nous avons donc étudié la documentation du boitier USB-4750 et utilisé le logiciel du DAQ-Navi permettant d'extraire les valeurs des ports d'entrées et de sorties.

Table des matières

Introduction **2**

1 Simulation **5**

1.1 Arborescence de l'application Godot 5
1.2 Placement et déplacement des objets 7
1.3 Clic sur un objet 9
1.4 Simulation du projet 9

2 Communication **12**

2.1 Reprise du projet 12
2.2 Recherche de méthodes de connections alternatives 12
2.3 Résolution 13
2.4 Compilation de Godot 13
2.5 GodotNative et Module Personnalisé 15
2.6 Liaison Simulation-Automate 15
2.6.1 Écriture dans un fichier 15
2.6.2 Les pipes nommés 15
2.6.3 Autres méthodes considérées 16
2.6.4 Modbus TCP/IP 17
2.7 Le module Modbusconnect 17
2.8 La simulation SimPyLC 19

3 Intégration SimPyLC <−> Godot **23**

4 Conclusion **26**

1 Simulation

Au semestre dernier, nous avions établi une base pour notre application de création contenant :

— une vue pour interagir avec les objets de notre système
— la bibliothèque d'objets
— l'arbre des objets contenus dans le système
— une fenêtre pour afficher les propriétés de l'objet
— une fenêtre affichant les modifications apportées au système.

Les objectifs de ce semestre sont :

— implémenter l'ajout des objets
— le déplacement des objets avec la souris
— la personnalisation des capteurs
— la sauvegarde des projets dans des fichiers
— la communication avec l'automate.

1.1 Arborescence de l'application Godot

Les nœuds sont des éléments fondamentaux pour la création d'un projet. Un nœud peut exécuter une variété de fonctions spécialisées. Cependant, un nœud donné a toujours les attributs suivants :

— Il a un nom.
— Il a des propriétés éditables.
— Il peut recevoir une fonction de rappel à traiter à chaque trame.
— Il peut être étendu (pour avoir plus de fonctions).
— Il peut être ajouté à un autre nœud en tant qu'enfant.

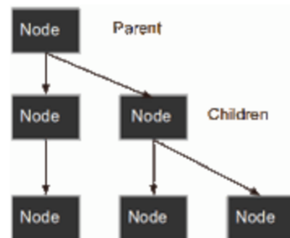

Figure 1 – Arbre des noeuds dans Godot

Le dernier est important. Les nœuds peuvent avoir d'autres nœuds comme enfants. Lorsqu'ils sont disposés de cette façon, les nœuds deviennent un

arbre.
Dans Godot, la capacité d'organiser les nœuds de cette façon en fait un outil puissant pour organiser les projets. Comme les différents nœuds ont des fonctions différentes, leur combinaison permet de créer des fonctions plus complexes.
Il est donc important pour la réalisation de l'application de s'attarder sur un arbre de noeud cohérent afin de lier les éléments correctement entre eux.

FIGURE 2 – Arbre des objets dans le logiciel GoDot

L'enjeu de la création des systèmes pour la simulation est donc de pouvoir également créer un tel arbre de noeuds grâce à notre application.

C'est grâce à cet arbre que lorsque nous supprimons un objet du projet, notre fonction de suppression va parcourir tous les objets fils et les supprimer aussi.
Nous avons aussi ajouté une fonction pour renommer des objets. En effet cela est nécessaire pour les capteurs et actionneurs qui, par défaut, ont le

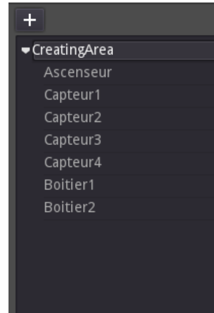

FIGURE 3 – Arbre des objets dans notre application

même nom lorsque nous les plaçons dans le projet. Il est donc important de les renommer pour ensuite distinguer les différents objets dans notre système.

Lorsque nous faisons un clic droit sur un objet dans l'arbre des objets, une fenêtre popup apparaît donnant 4 choix :
— Ajouter un objet
— Supprimer l'objet
— Renommer l'objet
— Cacher l'objet (ce dernier n'a malheureusement pu être réalisé)

1.2 Placement et déplacement des objets

Au semestre 7, nous avions commencé à mettre en place un système de glisser-déposer pour placer des objets dans le projet. Il suffisait de cliquer sur un objet dans notre bibliothèque et de l'amener sur notre scène d'édition. Cependant, ce système a été abandonné car la gestion de la distance dans l'environnement 3D générait beaucoup de bugs que nous ne parvenions pas à gérer. Nous avons donc préféré ne pas perdre de temps avec ceci et de remplacer le glisser-déposer par un simple bouton “ajout”. Lorsque nous cliquons sur le bouton + en haut ou choisissons d'ajouter un objet, une fenêtre apparaît nous affichant la liste des objets de notre bibliothèque. Si nous choisissons un de ces objets, celui-ci est ajouté au système et à l'arbre des objets. S'il est déclaré comme enfant d'un autre objet, il suivra les mouvements de cet objet ; cet objet est donc appelé parent de notre nouvel objet.

Ensuite, pour déplacer des objets, nous avons établi 2 types de déplacement :

à l'aide de fenêtre des propriétés de l'objet et le déplacement avec la souris. Lorsque nous cliquons sur un objet dans la scène 3D ou lorsque nous cliquons sur un objet dans l'arbre, la fenêtre des propriétés d'actualise en affichant le nom de l'objet sélectionné, sa position et sa rotation dans l'espace. Nous pouvons entrer une nouvelle valeur de position ou de rotation, ce qui va directement actualiser l'objet dans la scène.

FIGURE 4 – Fenêtre des propriétés de l'objet sélectionné

Afin d'avoir un rendu plus visuel, nous voulions aussi déplacer les objets à la souris. Tout d'abord, lorsque nous sélectionnons un objet, des flèches tridirectionnelles apparaissent au centre de l'objet. Il suffit ensuite de cliquer sur une flèche et de déplacer la souris pour faire déplacer l'objet dans l'axe de la flèche.

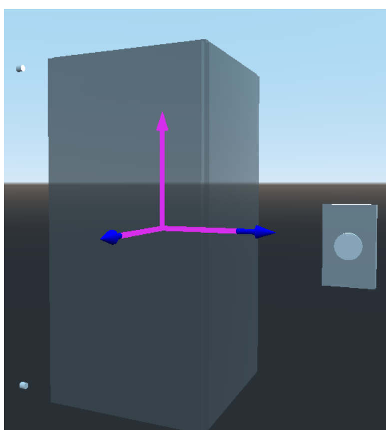

FIGURE 5 – Flèches permettant de déplacer un objet dans l'application

1.3 Clic sur un objet

L'implémentation du clic fut très importante. Pour ce faire, nous avons attaché un script sur l'objet nous permettant de nous déplacer dans l'espace 3D. Ce script regarde à chaque clic quel objet rencontre la vue de la caméra en premier.
Pour être plus précis, un objet appelé raycast part du centre de la caméra suivant la direction pointée par la souris. Dès que ce raycast va rencontrer un objet de type "Collider", elle va l'enregistrer et on va pouvoir le récupérer avec une fonction godot ; s'il ne rencontre aucun objet, alors la fonction retournera "null". On peut aussi utiliser un "masque de collision" qui va cibler certains objets Collider avec un masque correspondant. Ce masque va nous permettre de rejeter les noeuds "Collider" de l'objet qui est déjà sélectionné ; obligatoire car si les flèches sont cachées par l'objet sélectionné, nous ne pourrons jamais cliquer dessus.
Un script est aussi attaché à chaque flèche. Ainsi, lorsque nous cliquons sur une flèche nous lui transmettons la position de la caméra. Le script associé à la flèche va en déduire la direction du clic. On va créer un plan (invisible) de type Collider passant par la flèche avec pour normale la direction calculée. Ainsi, avec ce plan, nous pourrons suivre le déplacement de la souris même si celle-ci ne pointe plus sur la flèche. Au relâchement du clic nous faisons disparaître ce plan.

1.4 Simulation du projet

Maintenant que nous pouvons créer un projet et y incorporer des objets, il faut l'enregistrer et le lancer dans un environnement de simulation. Dans cet environnement les capteurs et actionneurs fonctionneront et les objets ne seront plus déplaçables. On s'est donc heurtés à la problématique de la sauvegarde du projet. En effet, il est aisé sur godot de sauvegarder des objets dans un dictionnaire, mais n'y sont alors conservées que les propriétés physiques de l'objet, et non sa position. De ce fait, nous avons préféré de sauvegarder le projet en sauvegardant ces différents objets dans un fichier avec l'intégralité de leurs propriétés (interactions, position, ...).

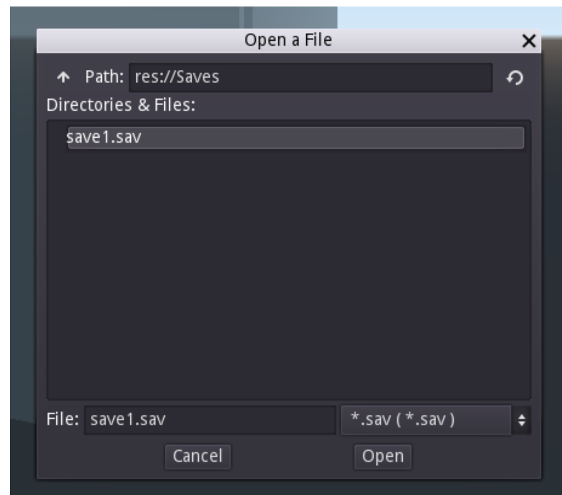

FIGURE 6 – Interface de sauvegarde d'un fichier

Pour sauvegarder les objets dans un fichier, il faut que chaque objet possède une fonction "save" renvoyant un dictionnaire avec ses attributs et donc il faut aussi qu'ils aient tous un script associé. Pour ce faire, nous utilisons des classes avec la hiérarchie suivante :

- Une classe mère "GlobalObject" contenant les attributs et les méthodes communes à tous les objets, dont la fonction save renvoyant un dictionnaire avec les attributs de l'objet.
- Une classe "Capteur" qui hérite des attributs et méthodes de la classe GlobalObject. Elle contient en plus des attributs spécifiques au capteur et on réécrit la méthode "save" qui appelle la méthode de la classe GlobalObject et y ajoute ses attributs spécifiques.
- Une classe "Actionneur" qui, comme la classe capteur, hérite de la classe GlobalObject en y ajoutant ses attributs et en réécrivant la méthode "save".

Pour charger un projet il suffit alors de lire le fichier ligne par ligne, une ligne correspondant à un objet. Ainsi on charge toute la scène en instanciant les objets avec leurs scripts et toutes le propriétés enregistrées.

Pour lancer la simulation, nous allons lancer une nouvelle scène et recréer le système à partir de l'arbre des objets. Nous allons en plus créer deux listes, une contenant tous les capteurs et l'autre les actionneurs. Ensuite nous ouvrons une fenêtre pour lier ces capteurs et actionneurs aux ports de

l'automate (dans notre cas les ports sont les bits du message circulant entre Godot et l'automate simulé).

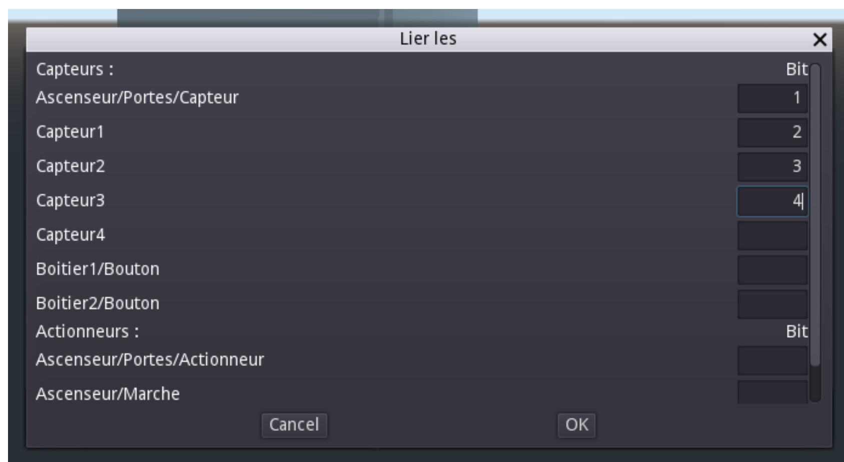

FIGURE 7 – Fenêtre pour lier les objets du simulateur à l'automate

Une fois que l'utilisateur a validé la connection avec l'automate, nous lançons la simulation. Pour communiquer avec l'automate, nous utilisons un entier pour l'état des capteurs et un entier pour les actionneurs. La partie simulation de Godot s'occupe de créer le message contenant l'information des capteurs. L'automate va créer, en fonction de ce qu'il reçoit, un message contenant l'information des actionneurs et l'envoyer à notre simulation. Nous décomposons ce message puis activons ou arrêtons les actionneurs grâce à cette information. Notre simulateur n'a donc aucun contrôle sur les actionneurs, l'automate est le seul à pouvoir contrôler les actionneurs.

2 Communication

2.1 Reprise du projet

A la reprise du projet au S8 notre objectif fut de résoudre les problèmes de communication du semestre précédent. En effet nous arrivions à lire et écrire l'état des I/O de l'automate mais cela semblait se dérégler sans que l'on ne comprenne pourquoi.

Nous avons alors demandé l'expertise de Mr. Conrard, notre tuteur de projet et un des responsables des installations en C006. Celui-ci nous a éclairé sur le fonctionnement de l'automate, le rôle des interrupteurs 3-états permettant de laisser passer le signal ou de forcer la valeur. La création du profil d'E/S du boitier USB4750 ainsi que de l'utilisation de l'interface de test du logiciel constructeur DaqNavi.

Nous avons alors étudié l'ensemble des fonctionnalités de l'USB4750 et les exemples de codes fournis dans la documentation. Deux exemples ont particulièrement retenu notre attention : Digital Input et Digital Output. Ces exemples sont implémentés dans la majorité des langages traditionnels : Java, C et C++.

Nous choisissons d'utiliser les versions C++ en vue d'une intégration au noyau Godot écrit également en C++. Seulement nous obtenons les mêmes problèmes que précédemment : Nous n'arrivons pas à maîtriser avec certitude la valeur des mots binaires enregistrés dans l'automate.

2.2 Recherche de méthodes de connections alternatives

Devant ce blocage nous avons exploré des méthodes alternatives de connections. Voulant nous assurer que l'erreur était indépendante de notre projet Visuel Studio nous essayons une implémentation python du driver de l'USB-4750 : https ://github.com/pohmelie/python-advantech-DAQNavi-bdaqctrl.

Cette implémentation utilise la librairie C Foreign Function Interface (CFFI) pour encapsuler le driver C d'Advantech. Afin d'utiliser le code sous notre machine nous devons créer une version du code C en supprimant une directive #include puis appelant le préprocesseur par la commande "c99 -E".

Nouvel obstacle : malgré nos essais il reste impossible d'ouvrir la connexion sur un appareil du type #USB4750.

Continuant notre recherche nous essayons alors d'installer le driver Advantech pour linux. Malheureusement celui-ci n'est pas compatible avec les systèmes basés sur Arch Linux tel que Manjaro que nous utilisons. Nous installons alors Ubuntu sur l'ordinateur portable de Sébastien car nous estimons qu'une machine virtuelle rajouterait un niveau de complexité et potentiellement des interférences. Nous lançons ensuite l'installation depuis les paquets officiels. Sur les 80 appareils Advantech supportés, l'USB4750 fait partie des 6 absents. Après double vérification nous sommes contraints d'abandonner également cette piste.

2.3 Résolution

Suite à ce double blocage nous reprenons rendez-vous avec Mr. Conrard pour vérifier notre compréhension du système. Ses explications renouvelées ainsi qu'une réinstallation de l'ensemble des logiciels permettant d'avoir un prototype de communication fonctionnel entre l'automate et les scripts C++. . Post-mortem : lors de notre découverte de DaqNavi nous avions modifié le profil d'E/S de l'USB4750 déterminant le mode des entrées sorties avec les paramètres suivant : l'activation, la direction et l'inversion des ports. Afin de déterminer l'impact des différents paramètres nous avons fait varier incrémentalement les valeurs pour nous avions modifié certains paramètres afin de déterminer leur impact et n'avions pas dû correctement rétablir le fichier ensuite. Ce profil était ensuite importé dans chacun de nos scripts.

La communication par le driver C++ mise en place et les défauts de compréhension du système du S7 résolus, nous pouvions passer à la suite : l'intégration du module de communication avec l'automate à Godot.

2.4 Compilation de Godot

C'est à ce moment que le confinement a commencé et que nous n'avons plus eu accès à l'automate.

Poursuivant notre objectif de création d'une liaison entre la simulation d'ascenseur et un potentiel automate nous avons suivi la documentation présente sur le site de GodotEngine. La première étape est de compiler Godot depuis les sources. Notre système d'exploitation est windows 10 à jour. Nous installons et configurons Scons (une alternative de CMake) pour utiliser le développement Godot. Les principaux avantages par rapport à CMAKE sont la possibilité d'utiliser un langage de programmation complet pour la

logique du build (Python) et la compilation pour différentes plateformes en simultané. Nous modifions également l'installation Visual Studio comme demandé.

Il n'y a pas d'erreurs à la compilation. On peut lancer l'éditeur et modifier le projet. Seulement des erreurs visuelles sont constatées lors de l'exécution du projet avec également des erreurs dans le terminal de type "ERROR : Condition " !windows.has(p_window)" is true. at : VulkanContext : :window_resize". Installer et dés-installer Vulkan n'eut pas d'effet.

Ce résultat n'étant pas satisfaisant nous essayons de diminuer progressivement le nombre de coeurs alloués à la compilation, passant de 4 à 1. Le temps nécessaire varie entre 40mn et 1h20 avec un Core i5-6200U mais cela n'a pas d'impact sur le problème. Nous utilisions premièrement Windows afin d'être compatible avec les machines de l'école. Suite à ce problème et confortés par la suggestion de Mr. Redon de simuler l'automate nous avons pris la liberté de continuer le projet sous Linux.

Même problème. La simple commande "scons -j8 platform=x11" de la documentation Godot ne produisait pas un binaire fonctionnel chez nous. La solution est venu du package Godot 3.2.1 de l'Arch User Repository.
Celui-ci fonctionnant nous en avons déduit qu'une solution était possible avec la bonne commande. Pour cela nous avons extrait la ligne du fichier PKBUILD.

```
→  godot-3.1.2-stable scons platform=x11 \
       tools=yes \
       target=release_debug \
       use_llvm=no \
       colored=yes \
       pulseaudio=no bits=64 -j $((`nproc`+1))
```

FIGURE 8 – Commande utilisée pour build Godot

Voici le détail des arguments :

- platform=x11 : Compiler pour linux
- tools=yes : inclut l'éditeur et non juste l'exécutable
- use_llvm=no : ne pas utiliser CLang
- pulseaudio=no : ne pas utiliser le serveur de son PulseAudio. (ALSA dans ce cas)
- bits=64 : architecture de destination

— -j $((nproc+1)) : nproc retourne la valeur du champ cpu de la commande lscpu soit Threads per core X cores per socket X sockets = 4 sur notre i5-6200U. C'est donc équivalent à -jobs 5, soit 5 thread dédiés.

La compilation maîtrisée, nous avons deux possibilités pour ajouter la capacité de communication à notre simulation.

2.5 GodotNative et Module Personnalisé

GodotNative : Permet d'écrire la logique de l'application en C++ au lieu de GDScript. Plus facilement partageable et supporte également les plateformes Android et Web

Module Personnalisé : Permet d'intégrer des bibliothèques C++ ou C au moteur Godot. Mais cela nécessite de recompiler à chaque modification et ce mode est moins adapté pour la réutilisation.

Comme l'objectif de notre projet et d'ajouter une interface de communication avec un automate, nous nous dirigeons vers un module personnalisé.

2.6 Liaison Simulation-Automate

Nous avons étudié plusieurs modes de communication pour relier la simulation à l'automate suite à l'impossibilité d'utiliser le matériel physique. Ces choix sont présentés dans l'ordre où nous les avons considérés.

2.6.1 Écriture dans un fichier

Nous avons commencé à expérimenter cette solution de par sa simplicité d'implémentation. Godot est devenu capable de lire et d'écrire dans un fichier que nous traitions à l'autre bout en python.

Les mécanismes de verrouillage d'accès aux fichiers rendent cependant cette solution inutilisable même en utilisant deux fichiers (un programme écrit toujours dans le même et lit dans un autre).

Néanmoins cela nous a permis de tester notre module.

2.6.2 Les pipes nommés

Les pipes sont un système IPC puissant sous linux. Un pipe nommé contrairement à un fichier ne prends pas de place sur le disque car le noyau

s'occupe de transmettre les informations directement entre le processus de lecture et le processus d'écriture. Un pipe nommé apparaît cependant comme un fichier dans lequel on peut lire et écrire pour les processus l'utilisant. Cette méthode est une amélioration plus propre de la méthode précédente.

Lors de l'implémentation sous C++ et python, nous avons constaté que la méthode la plus viable était d'envoyer les informations sans encodage (raw) précédées de leur taille. Un exemple est disponible sur notre dépôt git : INSERER GITHUB

2.6.3 Autres méthodes considérées

La sérialisation en JSON. le format JavaScript Object Notation se veut optimisé pour les traitements informatiques tout en étant facilement compréhensible par les humains. Notre simulation sauvegarde et charge les projets créés avec la bibliothèque d'objets en JSON. De plus il existe des modules de traitement de JSON en python et C++. Cette méthode nous semblait toutefois peu cohérente avec le système que nous cherchons à simuler : Des fichiers JSON sont relativement lourds comparés aux bits transmis par le boitier USB4750

Les files de messages : les files de messages sont une méthode de communication utilisée pour la communication inter-processus. Nous avons implémenté une version minimale afin de tester la faisabilité. Cette approche aurait pu être valable si nous n'avions pas trouvé une méthode plus proche de notre système physique.

Les sockets : les sockets sont un ensemble de fonctions de communication permettant de se brancher sur un réseau et de communiquer avec d'autres applications qui y sont branchées. De même qu'avec les files de message nous avons fait un bref essai avant d'opter pour notre méthode finale.

ZeroMQ - 0MQ est une bibliothèque de messagerie asynchrone haute performance open source dont l'interface est conçue pour ressembler à celle des sockets. Elle fut beaucoup mentionnée lors de nos recherches d'un mode de communication entre Godot et notre simulation d'automate. Bien que prometteuse, nous avons décidé d'opter pour une communication basée sur un réseau Modbus pour les raisons présentées à la section suivante.

2.6.4 Modbus TCP/IP

D'après Wikipédia : MODBUS est un protocole de communication non-propriétaire [...] utilisé pour des réseaux d'automates programmables, [...]. Ce protocole basé sur une structure hiérarchisée entre un client unique et plusieurs serveurs est dans le domaine public et sa spécification est publique.

Modbus peut être implémenté sur une liaison série, sur un réseau à passage de jetons ou sur un réseau Ethernet via TCP/IP. Notre projet utilise une implémentation TCP/IP.

Une liaison Modbus est directement intégrée aux automates des salles de TP d'automatique. Ce protocole étant adapté à notre besoin de transmissions de mots binaires, nous le choisissons pour coller avec l'objectif du projet avant le confinement.

Nous utilisons alors modbuspp et pyModbusTCP qui sont les implémentations open-source de ce protocole en C++ et Python, respectivement.

Les deux libraires offrent les 8 fonctions suivantes : read_coils(), read_input_bits(), read_holding_registers(), read_input_registers(), write_coil(), write_register(), write_coils(), write_registers() qui permettent d'écrire et de lire un ou plusieurs bits/mots.

2.7 Le module Modbusconnect

La méthode de communication déterminée, nous l'implémentons alors dans Godot et donc dans la simulation par un module personnalisé, Modbusconnect.

En plus des sources de modbuspp, notre module Godot est structuré de la façon suivante :

- /modbusconnect/config.py
- /modbusconnect/modbusconnect.h
- /modbusconnect/modbusconnect.cpp
- /modbusconnect/register_types.h
- /modbusconnect/register_types.cpp
- /modbusconnect/SCsub

Les fichiers register_types, SCsub et config.py regroupent les informations sur les plateformes supportées et les déclarations pour l'intégration dans le noyau Godot. Ces fichiers sont similaires pour la plupart des modules et sont fournis par Godot.

Modbusconnect.h définit la class Modbusconnect qui hérite de la classe

référence nécessaire à l'intégration dans Godot. Dans modbusconnect.cpp nous implémentons les fonctions intégrant la bibliothèque modbus.cpp : init_modbus(). Cette fonction ouvre une connexion modbus sur le port 502 (utilisé pour les applications TCP) en local. On configure les paramètres suivant les besoins de l'application.

read_capteur et read_actionneur() permettent de lire un mot dans le registre 0 ou le registre 1. Le mot reçu est de type uint16_t. Le mot est converti automatiquement et sans perte en int, ce qui est compatible avec le typage dynamique de GDScript ("var x = read_X()")

write_capteur()/write_actionneur(int état) permettent d'écrire un mot dans le registre 0 ou le registre 1. Nous appliquons un masque sur état afin d'extraire les 16 bits de poids faible de l'int dans un uint16_t compatible avec la fonction write_register() de mobduspp.

Les méthodes write_capteurs et write_actionneurs sont utilisés pour le débogage mais pas utilisé dans notre simulation.

Ces fonctions peuvent être paramétrées pour lire et écrire jusqu'à 2Î6 bits par appel. Cela permet de s'adapter à différents types d'applications sans être limités par le nombre de capteurs ou d'actionneurs. Ce module n'est donc pas lié à une application particulière mais peut servir à chaque simulation créée avec notre bibliothèque d'objets.

Lors de la compilation de ce module nous obtenions l'erreur suivante : "candidate : 'modbus : modbus(std : string, uint16_t)' expect 2 arguments, 0 provided". Alors que nous appelions explicitement la création d'un objet modbus par la ligne modbus("127.0.0.1", 502) dans le constructeur de class modbusconnect. N'arrivant pas à résoudre le problème nous avons contacté un des développeurs s'occupant du noyau. Ernest (iFire) Lee nous suggéra d'initialiser la variable modbus avant l'appel du constructeur ce qui se traduit par : Modbusconnect : :Modbusconnect() : mb(modbus("127.0.0.1", 502)) //....

Ce problème résolu, nous pouvions maintenant utiliser la bibliothèque modbuspp dans l'éditeur de script comme toute autre fonction native.

```
func demo():
    modbus.read
              read_actionneur()
func dic_toread_capteur()
    var val = 0
```

2.8 La simulation SimPyLC

Avec le coronavirus, Mr Redon a suggéré que nous simulions un automate comme les maquettes de la salle C006.

Nous avons initialement considéré l'écriture maison d'un automate en utilisant le code Grafcet de nos TP de CSPPE. Nous avons découvert peu après SimPyLC qui est plus adapté. SimPyLC est une implémentation à visée pédagogique d'un automate, faite par Garry Boyer, l'auteur du programme simPyLC utilisé industriellement.

Un PLC, ou automate programmable industriel (API) en français, est un dispositif électronique programmable destiné à la commande de processus industriels par un traitement séquentiel. Un API envoie des ordres vers une partie opératrice à partir de données d'entrées (capteurs), de consignes et d'un programme informatique.

Un API se doit d'être extrêmement stable et sécurisée. C'est pourquoi des règles strictes sont apparues :

— Pas d'autre boucle que la boucle principale, le "sweep".

— Pas de branchement conditionnel.

— Pas de pause, sleep et autres.

Ainsi la sortie de l'automate est calculée à chaque itération de la boucle en fonction de l'état instantané des capteurs. La seule façon de retenir l'état d'une itération précédente est d'utiliser des bascules, cependant il faut les limiter afin de garantir un fonctionnement fiable.

SimPyLC définit 6 classes d'objets prêtes à être utilisées pour réaliser la programmation de l'automate :

— Class Marker : Permet d'évaluer une expression booléenne et de changer la valeur booléenne du marker en fonction de l'expression.

— Class OneShot : Valeur booléenne ne pouvant être à vrai que pendant la durée d'un sweep. Notre automate n'en utilise pas.

— Class Latch : Bascule permettant de sauvegarder un état entre deux sweeps. Nous en utilisons pour le déplacement de l'ascenseur.

— Class Register : Permet de stocker un entier.

— Class Timer : Un simple timer pouvant être remis à 0.

— Class Runner : Permet de mettre la simulation en pause.

Outre ces 6 classes, simPyLC fournit un moteur permettant d'évaluer les interactions entre ces classes ainsi qu'une méthode d'affichage des valeurs de la simulation sur une fenêtre et l'évolution temporelle des valeurs sur une autre, ainsi que toute une partie pour la modélisation 3d qui ne nous servira

pas car nous utilisons Godot.

Après avoir étudié les simulations fournies en exemple (une led clignotante et un bras d'assemblage industriel) nous avons mis en place l'automate correspondant à notre application Godot.
Les caractéristiques sont les suivantes :

— 2 Registers : un pour les capteurs et un pour les actionneurs

— 10 Markers : 7 pour les capteurs de la simulation : Portes, appels, position de l'ascenseur ; 3 pour les actionneurs : mouvement des portes on/off, direction de l'ascenseur et déplacement de l'ascenseur.

— 5 Latchs : 2 mémorisant l'appel de l'ascenseur, et 3 nécessaires à la gestion des portes.

Notre boucle sweep (boucle principale) suit le format standard d'un API.
while True :

— readInputFromSensors ()

— calculateOutputFromInputAndPreviousState ()

— writeOutputToActuators ()

On va donc développer chaque fonction :

readInputFromSensors() :

Dans un premier temps, on lit la valeur des capteurs sur le serveur modbus avec la fonction read_holding_register dans capt_register. On actualise les markers correspondant à l'état de chaque actionneur. Pour cela nous avons directement pensé à faire une logique entre le registre et un masque de position. Seulement, bien que le registre supporte les opération arithmétiques +,-,*,/ et %, l'opérateur n'était pas défini. Cependant comme simPyLC est open source nous avons pu nous familiariser avec l'architecture et ajouter la "méthode magique" (nom des méthodes commençant et finissant par __ en python) __and__. de même que la méthode "≫", __rshift__. Il aurait également été possible d'utiliser la fonction interne evaluate() utilisée par le moteur, qui renvoie la valeur attribuée à l'objet, mais cela n'aurait pas été très homogène avec l'ensemble de la simulation, ni propre car cela expose l'architecture interne de simPyLC.

calculateOutputFromInputAndPreviousState() :

Ensuite, on exécute la logique commandant l'état des actionneurs qui est la suivante :

- mettre à jour la bascule indiquant un appel à l'étage 1.
- mettre a jour la la bascule porte_charge indiquant qu'il faudra ouvrir ou fermer les portes dès que cela est possible, c'est à dire à la fin du mouvement entre étage.
- mettre à jour une bascule intermédiaire indiquant qu'on est en train de se déplacer et qu'il n'y a pas eu d'ordre de mouvement dans la direction opposée
- désactiver la bascule porte charge si elle était précédemment activée et que nous sommes arrivés à un étage. Cela permet aux portes de ne pas s'ouvrir et se refermer en continu après un déplacement.
- désactiver la bascule de sauvegarde d'appel à l'étage 1 si le capteur d'arrivée est activé.

On effectue ensuite des opérations similaires avec les capteurs et bascules de l'étage 0. On active l'actionneur de mouvement de l'ascenseur s'il y a eu un appel. On détermine le sens de l'appel en testant l'état des capteurs.

writeOutputToActuators () :

Enfin, étant donnés le peu d'opérations autorisées par les paradigmes il n'y a pas d'autres méthodes que de "hardcoder" la table de vérité des actionneurs pour attribuer la valeur finale a act_register. La boucle se termine en envoyant la valeur résultante de act_register à l'automate avec la méthode write_single_register de la librairie PyModbusTCP.

L'automate produit deux fenêtre lors de son fonctionnement : une fenêtre de contrôle permettant d'afficher/forcer les valeurs de variable ainsi que la mise en pause de la simulation et une fenêtre représentant l'évolution temporelle des variables. Voici la fenêtre 1 :

Les premiers paramètres affichent l'horloge de la simulation. Counter compte le temps écoulé depuis le lancement. Ce contrôle correspond à l'état initial de la simulation d'ascenseur. On y lit que le registre des actionneurs vaut 0 car il n'y pas encore eu de commande. Le registre des capteurs vaut 2 car le capteur indiquant que les portes sont fermées est à 1, ce qui correspond au bit à la position 1 de registre_capteur.

Cette fenêtre est le résultat d'un clic sur le bouton d'appel de l'étage 1

FIGURE 9 – Fenêtre de contrôle de la simulation PyLC

dans la simulation. Les lignes de haut en bas sont les suivantes :

- La première ligne en rouge permet de constater que le simulateur est en marche.
- Capt_register (Gris) : Valeur numérique de Capt_register
- capt_fermeture_portes, ..., capt_appel_1. (Vert) : état des capteurs de la simulation Godot
- act_register (Gris) : Valeur numérique de Capt_register
- act_portes, act_mouvement, act_direction_mouvement (Rouge) : état des actionneurs de la simulation Godot.

De gauche à droite on voit que initialement capt_etage_0_bas est à 1. Peu après on constate un pic sur la dernière ligne verte : capt_appel_1 s'est activé. Les conséquences sont les suivantes : act_mouvement et act_direction_mouvement sont mis à 1, signifiant que l'ascenseur se déplace vers le haut. De plus, capt_etage_0_bas passe à 0 car l'ascenseur a commencé à monter.
On voit ensuite l'enchaînement des évènements suivants :

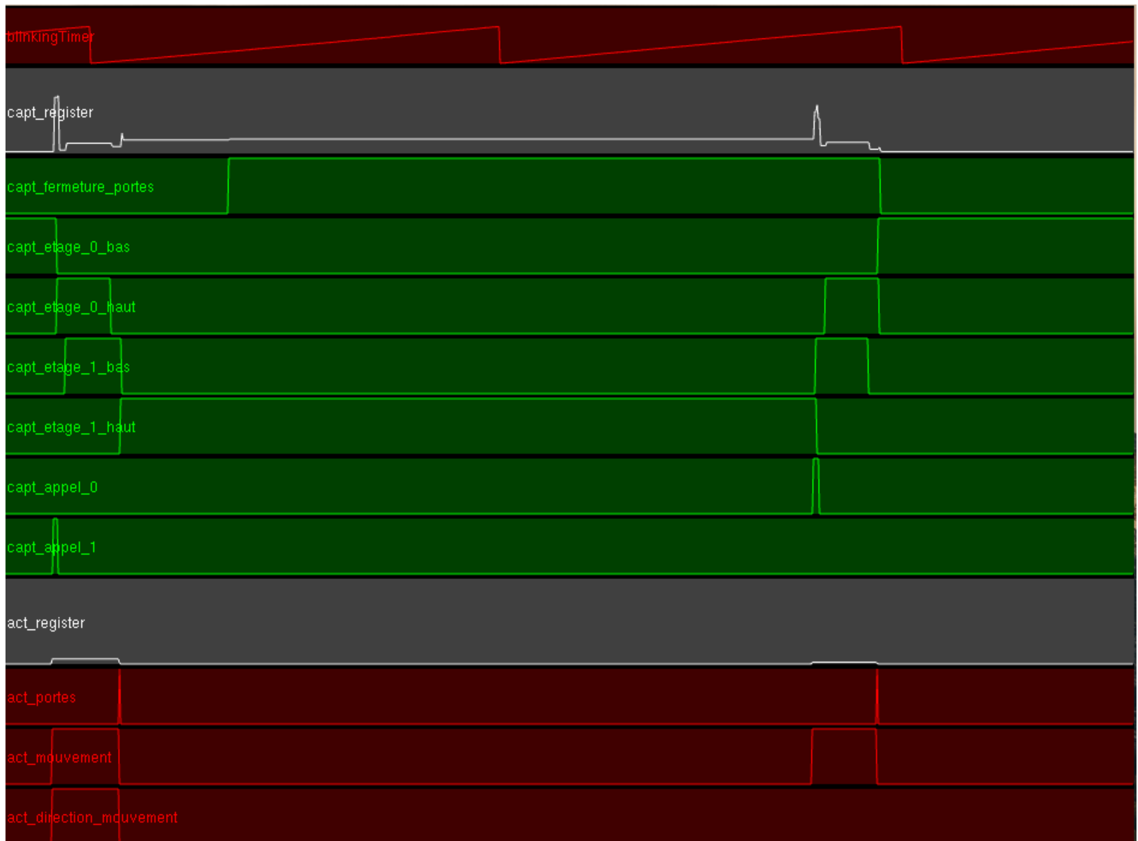

FIGURE 10 – Fenêtre de contrôle de la simulation PyLC

- capt_etage_0_haut passe à 1
- capt_etage_1_bas passe à 1
- capt_etage_0_haut passe à 0
- capt_etage_1_haut passe à 1
- act_portes passe très brièvement à 1

La suite du graphique présente l'évolution de la simulation lors de l'appel à l'étage 0. L'évolution est similaire à la montée mais les étapes sont inversées.

3 Intégration SimPyLC $<->$ Godot

Tous les capteurs et tous les actionneurs de la bibliothèque ajoutés au fur et à mesure de l'édition d'une simulation sont ajoutés automatiquement

à un dictionnaire pour faciliter la récupération et l'actualisation des états. Voici les fonctions que nous avons écrites pour lier l'état de la simulation Godot à la simulation d'Automate de commande SimPyLC, en passant par les bibliothèque Modbuspp et PyModBusTCP.
La première fonction permet de synthétiser l'état du dictionnaire des capteurs sous la forme d'un registre.

```
func dic_to_int(dic):
    var val = 0
    var ind = 0
    for key in dic.keys():
        if dic[key].get_etat() :
            val += 1<<ind
        ind+=1
    return val
```

FIGURE 11 – Commande extrayant les informations d'un dictionnaire dans un int

La seconde fonction permet de mettre à jour un dictionnaire en fonction d'un registre reçu.

```
func update_dict(dic, int_read):
    var ind = 0
    for key in dic.keys():
        if (int_read & 1<<ind) > 0:
            dic[key].change_etat(1)
        else:
            dic[key].change_etat(0)
        ind+=1
    return #void
```

FIGURE 12 – Commande mettant à jour le dictionnaire d'actionneur

Finalement nous avons ajouté une fonction _process dans le fichier global.gd qui est appelée à chaque frame, de façon similaire à la boucle sweep de l'automate.

```
func _process(delta):
	ticks += 1
	if ticks == 500 :
		for key in capteur_dict.keys() :
			print("Capteur :"+str(key)+" etat:"+str(capteur_dict[key].get_etat()))
			
		for key in actionneur_dict.keys() :
			print("Actionneur :"+str(key)+" etat:"+str(actionneur_dict[key].get_etat()))
		ticks = 0
		
	modbus.write_capteur(dic_to_int(capteur_dict))
	update_dict(actionneur_dict, modbus.read_actionneur())
```

Figure 13 – Fonction permettant de mettre à jour la simulation Godot

Elle affiche périodiquement l'état des capteurs et des actionneurs et communique avec l'automate pour envoyer l'état des capteurs et recevoir la commande des actionneurs.

Ces fonctions s'adaptent à n'importe quelle simulation créée avec notre éditeur.

4 Conclusion

Au cours du projet "Simulation de processus physiques" proposé par Mr Conrard et supervisé par Mr. Redon nous avons pu développer nos connaissances en gestion de projet et en en informatique.

En gestion de projet nous avons utilisé le critère SMART.

Au cours de ce projet nous avons développé nos connaissances en informatique, particulièrement en modélisation et réseau, en réalisant un simulateur de processus physique en collaboration.
Pour la suite notre objectif est d'implémenter un système plus complexe.