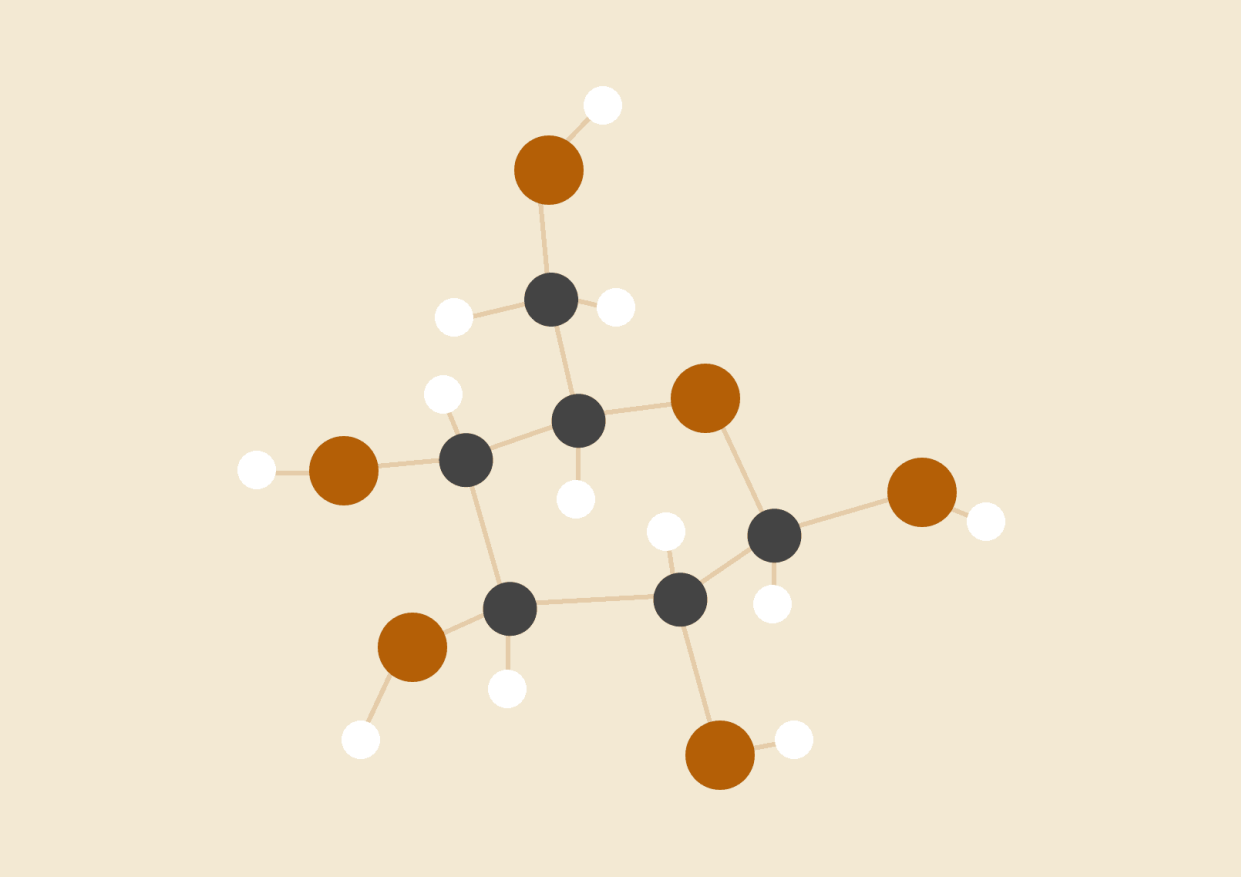
Projet TPI - Conventus

Application de conception et simulation de circuits logiques



Mathias Renoult
Étudiant
CPNV - SI-MI4B
Mathias.renoult@cpnv.ch

Jonathan Melly
Expert
ETML
Jonathan.melly@eduvaud.ch

Loïc Viret
Chef de projet
CPNV
Loic.viret@cpnv.ch

Romain Gehrig
Expert
Romain.gehrig@gmail.com

Glossaire

Aggrégation	C'est le fait de pouvoir stocker un objet dans un autre objet.
Algèbre de bool	Approche algébrique de la logique. Permet de représenter des schémas logique sous forme d'expressions.
ALU (Arithmetic Logic Unit)	Composant chargé d'effectuer des calculs. Souvent faisant partie d'un microprocesseur.
AND	Porte logique "ET" (∧) requérant toutes ses entrées étant allumée pour renvoyer un signal positif.
Asset	Entité Unity que l'on peut voir dans l'application ou dans le projet. Il peut venir de l'extérieur ou être créé par l'utilisateur.
Asset Store	Magasin en ligne permettant de télécharger et ajouter des *assets* créé par d'autres personne dans notre projet Unity.
Attribut	Variable en C# étant définie par un type, un niveau de protection et un nom. Les attributs composent les classes.
C#	Language de programmation orienté objet développé par Microsoft et proche du Java. C'est le language de *scripting* d'Unity
Classe	Moule permettant de servir de patrons lors de la création d'objets. Lorsqu'on utilise une classe pour créer un objet, on crée une *instance* de cette classe.
Component	*Asset* Unity ou classe ajoutable à un *GameObject* pour interagir avec lui.
Conventus	Nom de l'application. Signifie "Assemblée", "Assemblage" et plus rarement "Circuit" en latin.
CPNV (Centre Professionnel du Nord Vaudois)	École professionnelle dans laquelle se déroule le projet.
Encapsulation	On cache la structure de l'objet et on propose plutot des méhodes pour manipuler les propriétés de cet objet afin

	de s'assurer de la bonne manipulation de ceux-ci.
GameObject	Élément présent dans la hiérarchie du projet *Unity*. Chaque *GameObject* à un *Transform* qui lui est attaché et qui gère sa position, rotation et échelle. On peut lui ajouter des *components* au besoin.
Git Extensions	Interface visuelle pour *Git*.
GitHub	Service web d'hébergement et de gestion de développement de logiciels, utilisant le logiciel de *versioning Git*.
Héritage	Un des quatre éléments centraux de la *POO*. L'*héritage* permet de transmettre des *attributs* et *méthodes* aux classe qui lui sont enfant. Permet l'utilisation du *polymorphisme*.
Instance	Objet créé à partir d'une classe source.
LabView	Logiciel informatique développé par National Instruments et qui permet de créer des systèmes de mesure de contrôle via une interface graphique.
LogiSim	Simulateur de circuits logiques libre et gratuit écrit en Java.
MA-01	Module "Base du numérique" donné généralement en première année d'Informatique au CPNV. On y apprend notamment le binaire, l'hexadécimal et l'algèbre *booléenne*.
Méthode	Portion de code présent dans une classe pouvant être exécuté suite à un appel. Une méthode se caractérise par son niveau de protection, son type de retour, son nom et les arguments qu'elle demande.
NAND	Porte logique "NON-ET" ($\overline{\wedge}$) laissant passer le signal tant que les deux entrées ne sont pas activées. C'est l'équivalent théorique d'un transistor classique.
Nesting (nidification)	Procédé visant à encapsuler des composants pour en créer un nouveau.
NOR	Porte logique "NON-OU" ($\overline{\vee}$) qui est laisse passer le signal seulement si aucune entrée n'est activée.
NOT	Porte logique "NOT" ($\neg$) qui inverser le signal.

OOP (Object Oriented Programmation)	Se référer à *POO*.
OR	Porte logique "OU" qui est laisse passer le signal si au moins l'une des deux entrées est activée.
Polymorphisme	Un des quatre éléments centraux de la *POO*. Le polymorphisme permet d'exécuter un code spécifique en fonction du type de l'objet sur lequelle on appelle une *méthode*.
POO (Programmation Orientée Objet)	Paradigme de programmation centrée autour des objets. Ce paradigme à quatre piliers : *l'héritage,* le *polymorphisme, l'encapsulation* et *l'aggrégation*
Porte logique	Une porte logique est un élément physique ou logique qui prend généralement deux entrées binaires et renvoie un seul signal de sortie, également en binaire.
POS (Product Of Sum)	Expression booléenne faite à partir de la somme d'une ou plusieurs variables.
Scalable (échelonnable)	Augmenter ou réduire un ensemble de valeurs tout en gardant le rapport de grandeur entre elles.
Script	Fichier contenant du code. Désigne traditionnellement uniquement le code exécuté sur le CPU.
Shader	Fichier contenant du code. Désigne traditionnellement uniquement le code exécuté sur le GPU.
Singleton	Le *singleton* est un patron de conception dont l'objectif est de restreindre l'instantiation d'une classe à un seul et unique objet. Appelé parfois manager.
SMART	Acronyme signifiant Spécifique, Mesurable, Atteignable, Réalisable et Temporellement ancré. Cette méthode à pour but de forcer la création d'objectifs précis et cohérents.
SOC (Security Operations Center)	Acronyme désignant la partie sécurité d'un élément.
SOP (Sum Of Products)	Expression booléenne faisant la somme des produits d'un ensemble de variables étant liées entre elles par des portes *AND* et *OR*.

Static	Propriété du *C#* pouvant être appliqué à un *attribut* ou une *classe*. Cela permet d'interdire l'instanciation d'une classe ou d'un attribut.
Transistor	Élément électronique utilisé pour faire des processeurs notamment. C'est l'équivalent de la porte *NAND* logique.
Trigonométrie	Ensemble de méthodes de calcul permettant de calculer des angles et des distance dans des triangles, le plus souvent rectangles.
Unity	Moteur de jeux-vidéos 2D ou 3D utilisant le *C#* comme langage de scripting. Logiciel utilisé pour créer l'application de ce projet.
XNOR	Porte logique "NON-OU-EXCLUSIF" (=)
XOR	Porte logique "OU-EXCLUSIF" (⊻) qui laisse passer le signal si les deux entrées sont différentes.

Table des matières

Table des matières

Glossaire 1

Table des matières 4

Résumé 7

Analyse préliminaire 7

Introduction 7

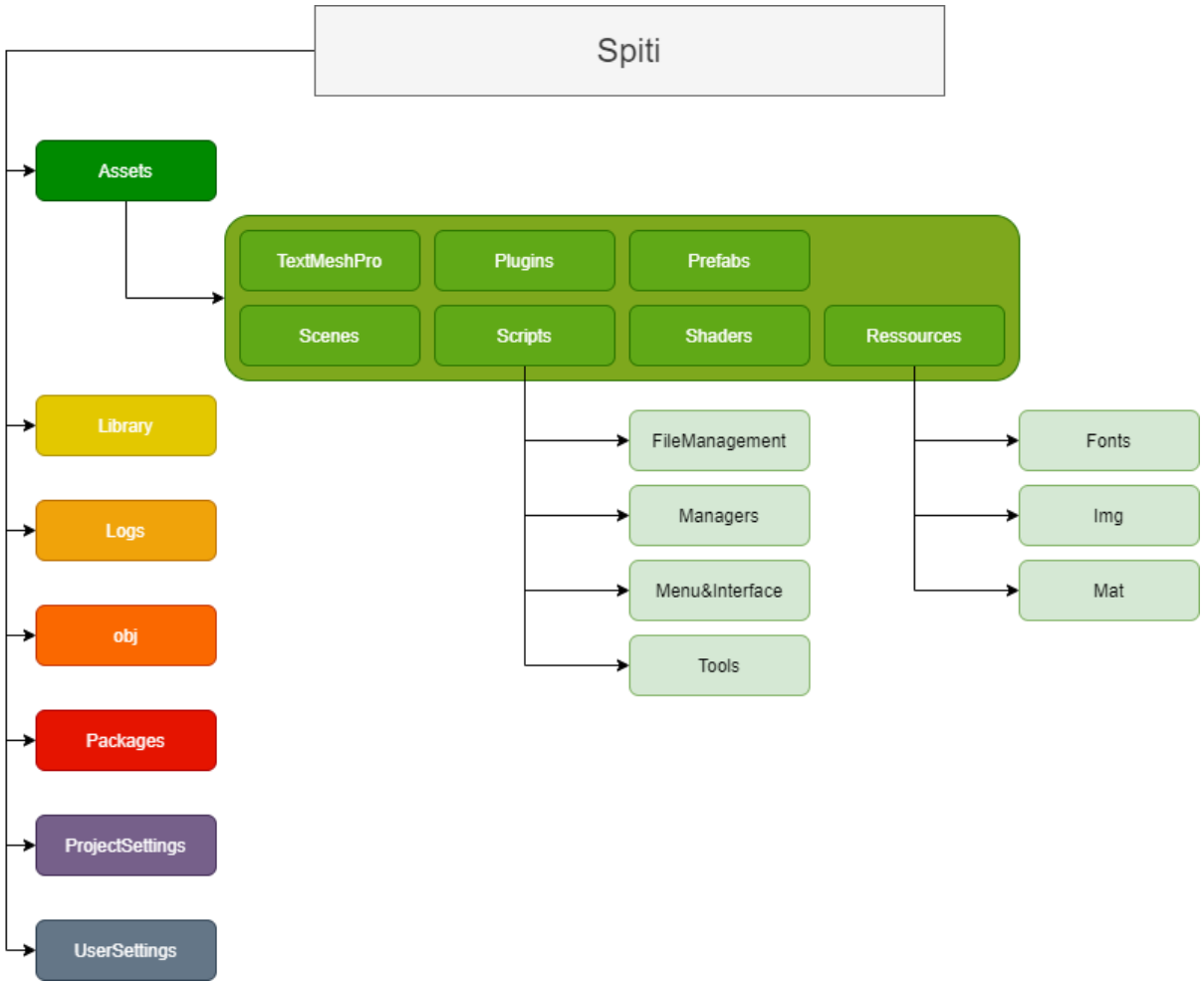
Buts 7

Objectifs 8

Analyse 9

Concept 9

Difficultés attendues 9
Planification initiale 11
Planification détaillée 12
Sprint 1 12
Sprint 2 13
Sprint 3 13
Dossier de conception 14
Matériel 14
Hardware 14
Système d'exploitation 14
Logiciels et sites Internet 14
Modèle conceptuel de données 15
Programmation 16
Versioning 17
Conception des maquettes 17
Présentation des maquettes 19
Palette de couleurs 21
Réalisation 22
Structure du projet 22



..... 22

Description de l'application 23

Les components 23

Déroulement du développement 24

Interface de base 24

Portes logiques 24

Câbles 24

Synergie câbles – components 26

Analyse du journal de travail 29

Problèmes rencontrés 29

Rétrospective 29
Erreurs restantes 29
Pour aller plus loin 29
Tests effectués 29
Documents fournis 29
Conclusions 29
Annexes 29
Manuel d'utilisation 30
Archives du projet 30

Résumé

Analyse préliminaire

Introduction

Conventus est une application de conception et de simulation de circuits logiques. Elle permet de placer des composants logiques (soit des portes logiques[1] basiques, soit des composants créés par l'utilisateur) sur un canevas, de les relier et de tester le circuit en choisissant le nombre d'entrées, de sorties, et leur état. L'un des objectifs non-officiel de ce projet est de servir de support pour le module *CPNV MA-OI*. Bien qu'il existe déjà des applications permettant de faire de genre de simulations, je pense que celle-ci est unique dans son concept de création quasi automatique de composants personnalisés.

Buts

Étant donné que je souhaite livrer une application contenant plus de choses que

demandé dans le cahier des charges, je vais séparer mes buts en deux : les buts impératifs et les buts circonstanciels

Impératifs :

- Se concentrer sur l'aspect ergonomique et « *easy-to-use* » de l'application
- Porter une attention particulière à la communication avec le chef de projet et les experts
- Livrer une application agréable à utiliser et conforme au cahier des charges
- Développer une interface responsive, interactive et *scalable*

Circonstanciels :

- Ajouter la fonctionnalité de *nesting* (création d'un nouveau composant à partir d'autres)
- Ajouter la fonctionnalité de simplification des circuits et de traduction en expressions algébriques booléennes
- Ajouter une section « Apprentissage » qui permettrait d'apprendre pas à pas comment fonctionne les circuits logiques et qu'est-ce qu'on peut faire avec

Objectifs

Les objectifs doivent respecter la méthode *SMART*. Ils servent à donner des cibles concrètes à viser lors de la réalisation du projet.

La grille d'évaluation reste le document d'autorité, néanmoins, la liste d'objectif ci-dessous ne perd pas sa valeur en production. La composante « T » du *SMART* dans les objectifs ci-dessous est implicitement « lors du rendu du projet ».

1. L'application est créée avec *Unity* et est en *C#*
2. L'utilisateur peut ajouter des portes logiques (*NOT, AND, OR, XOR, NAND, NOR, XNOR*) sur le canevas
3. L'utilisateur peut ajouter et supprimer les entrées et sorties du schéma. Il peut modifier l'état des entrées
4. L'application contient une partie « conception » qui lui permet de créer son circuit

5. L'application contient une partie « simulation » qui permet de tester le circuit créer par l'utilisateur. Il peut modifier l'état des entrées pendant la simulation
6. L'utilisateur peut enregistrer le schéma sur lequel il est en train de travailler et le rouvrir plus tard pour le modifier à sa guise
7. L'application supporte l'exportation des schémas sous forme de fichier image

Analyse

Concept

Avant toute chose, j'apprécie donner un nom aux application que je suis en train de développer. Cela me permet de mettre un mot sur le projet et de me motiver à faire quelque chose de réellement concret.

Pour mon projet Pré-TPI, j'avais choisis le nom *Spiti*. Cela signifie « maison » en grec et l'application devait me permettre de dessiner des plans de bâtiments et d'y placer des meubles en vue d'un déménagement. Pour rester dans la même logique, j'ai cherché un nom en fonction d'un élément central du projet. N'ayant rien trouvé de convaincant en grec, j'ai décidé de chercher du côté du latin. J'ai fini par trouvé un nom qui me plaisait bien : *Conventus*. Cela peut se traduire par « assemblage » ou plus rarement par « circuit ».

Il est clair que *Conventus* n'a pas la prétention de révolutionner le marché des simulations de circuits logiques. Il y a déjà beaucoup d'applications très complètes qui permettent de réaliser ce genre de choses et bien plus encore. On peut citer par exemple *MultiSim* ou *LabView*. En revanche, ce que j'espère fera la force de mon application, c'est son côté très simple à prendre en main et surtout pédagogique.

Difficultés attendues

Lors de ce projet, il y a plusieurs éléments nouveaux auxquels j'aurai à faire face. Afin de limiter au maximum les risques de blocage lors de l'implémentation des fonctionnalités, je vais lister ces potentiels points « à problème » et réfléchir

à comment je pourrais limiter leur risque dans la suite de l'analyse et conception du projet. Voici donc les points que j'ai relevés :

1. Interface efficace et *scalable*

Je n'ai jamais réalisé d'interfaces réellement dynamiques. Je ne sais pas vraiment s'il y a un moyen assez « simple » d'automatiser la mise à l'échelle de celle-ci.

2. Portes bouclant sur elles-mêmes

J'ai peur qu'il y ait des soucis de clignotement s'il y a des boucles qui se forment dans le circuit. Une des solutions serait de détecter les boucles et de renvoyer un état « indéfini » à la place de 0 ou 1. Bien que j'aie déjà un début de solution, j'ai peur que cela soit techniquement ardu à mettre en place.

3. Simulation « au ralenti » du circuit

La simulation devrait pouvoir être jouée à différentes vitesses. C'est important pour le côté pédagogique de l'application. Le problème que cela me pose c'est que je vais devoir synchroniser d'une manière ou d'une autre les signaux et je n'ai pas encore trouvé de manière simple de faire cela.

4. Exportation du circuit en PNG

L'exportation du schéma en image était déjà un des points de mon Pré-TPI. Je n'ai pas réussi à mettre en place cette solution dans les temps, ni même de commencer les recherches à vrai dire. C'est pour ça que cela me fait un peu peur de devoir faire cela de nouveau. Je vais essayer de trouver une librairie ou un *asset* qui fait ça de manière quasi automatique.

Planification initiale

Le Logiciel Gantt Project 2.8 a été utilisé pour mettre en place la planification initiale du projet. Les tâches sont regroupées par type (analyse, conception, réalisation, tests et documentation).

Nom	Date de début	Date de fin
Analyse	03.05.2021	06.05.2021
Création de la planification initiale	03.05.2021	03.05.2021
Créer le Git, le projet Unity, le journal de bord, le rapport	03.05.2021	03.05.2021
Recherche d'informations sur le sujet	04.05.2021	04.05.2021
Maquettes de l'app	04.05.2021	04.05.2021
Choix des conventions de nommage et architecteur du code	04.05.2021	04.05.2021
Début du rapport	06.05.2021	06.05.2021
Analyse du code d'applications similaires	06.05.2021	06.05.2021
Conception	10.05.2021	11.05.2021
Diagramme de flux sur les parties les plus critiques de l'app	10.05.2021	11.05.2021
Choix de la technologie de sauvegarde de fichier	11.05.2021	11.05.2021
Finalisation du dossier de l'analyse et du dossier de conception dans le rapport	11.05.2021	11.05.2021
Implémentation	17.05.2021	27.05.2021
Création de l'interface de base	17.05.2021	17.05.2021
Recherche des ressources (images et sons)	17.05.2021	17.05.2021
Création des managers et des classes "de base"	17.05.2021	17.05.2021
Création des managers et classes "de base"	18.05.2021	18.05.2021
Codage des "câbles" qui relient les éléments	18.05.2021	18.05.2021
Codage des portes logiques de base	18.05.2021	18.05.2021
Fonctionnalité "simulation" du circuit	20.05.2021	21.05.2021
Affichage des états des portes et des câbles	21.05.2021	21.05.2021
Enregistrement des fichiers	25.05.2021	25.05.2021
Exportation de fichiers image	25.05.2021	27.05.2021
Tests et Finalisation	28.05.2021	04.06.2021
Session de tests approfondie	28.05.2021	28.05.2021
Session de test approfondie	31.05.2021	31.05.2021
Terminer le rapport	31.05.2021	01.06.2021
Correction du rapport, enjolivage, manuel utilisateur	03.06.2021	03.06.2021
Rendu final TPI	04.06.2021	04.06.2021

Planification détaillée

Comme solution de gestion de projet, j'ai décidé de partir sur la solution intégrée à *GitHub* directement dans l'onglet « Projects » du dépôt sur les conseils de Mr. Viret. J'ai utilisé le modèle « Standard Kanban ».

C'est une méthode de gestion *agile* tout en restant simple et légère à utiliser. Cela signifie que j'ai des *sprints* qui segmentent mon projet et une liste de tâche dans chacune de celles-ci.

La durée des *sprints* n'est pas fixe, comme le demande normalement la norme agile. C'est parce qu'avec les fête de l'ascension et la pentecôte, un des *sprints* aurait été très court. J'ai préféré me concentrer sur des sprints fonctionnels.

Voici les sprints et les tâches :

Sprint 1

03.05.21 - 17.05.21 Analyse et conception. Création de l'interface de base, recherche des ressources nécessaire au projet, création des managers et des classes "de base".

- Création de tests
- Finaliser la partie analyse et conception du rapport
- Trouver comment exporter le projet en tant que PNG et expliquer le processus de fonctionnement
- Choisir la technologie de sauvegarde de fichier et réaliser un diagramme expliquant le processus de sauvegarde
- Diagramme de flux du calcul de la *table de vérité*
- Diagramme de flux du « *nesting* » des composants
- Diagrammes de flux de la simulation d'un circuit
- Choisir les conventions de nommage et l'architecture du code
- Maquettes de l'application
- Création du Git
- Créer le document du rapport et documenter l'analyse
- Analyser le code d'applications similaires

- Création de la planification initiale
- Création du journal de bord
- Recherche d'infos sur le sujet

Sprint 2

18.05.21 - 28.05.21 Création de l'application en soi. Placement des portes logiques, simulation, sauvegarde et export des fichiers.

- Exportation d'un projet en PNG
- Enregistrement du projet & gestion de projets
- Affichage de l'état des câbles et des composants
- Programmation de la simulation des circuits
- Programmation des portes logiques
- Programmation des « câbles » qui relieront les éléments entre eux
- Création des managers et des classes « de base »

Sprint 3

- 28.05.21 - 04.06.21 Déroulement des tests. Correction d'un maximum de bugs. Finalisation du rapport.
- Préparation de la défense TPI
- Envoyer le mail de rendu
- Impression du rapport et reliage
- Création du manuel utilisateur
- Corriger le rapport
- Terminer le rapport
- Session de test approfondie
- Session de test approfondie 2

Dossier de conception

Matériel

Le développement d'une application de si petite envergure ne nécessite pas de matériel très spécifique. J'utiliserai mon poste de travail au *CPNV*. En voici les spécifications :

Hardware

- Dell OptiPlex 7050
- ACPI x64 based PC
- Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz
- Intel(R) HD Graphics 530
- 16 GO de RAM
- Souris Microsoft HID
- Calvier Dell Standard

Système d'exploitation

Le système d'exploitation est Microsoft Windows 10 Education.

Logiciels et sites Internet

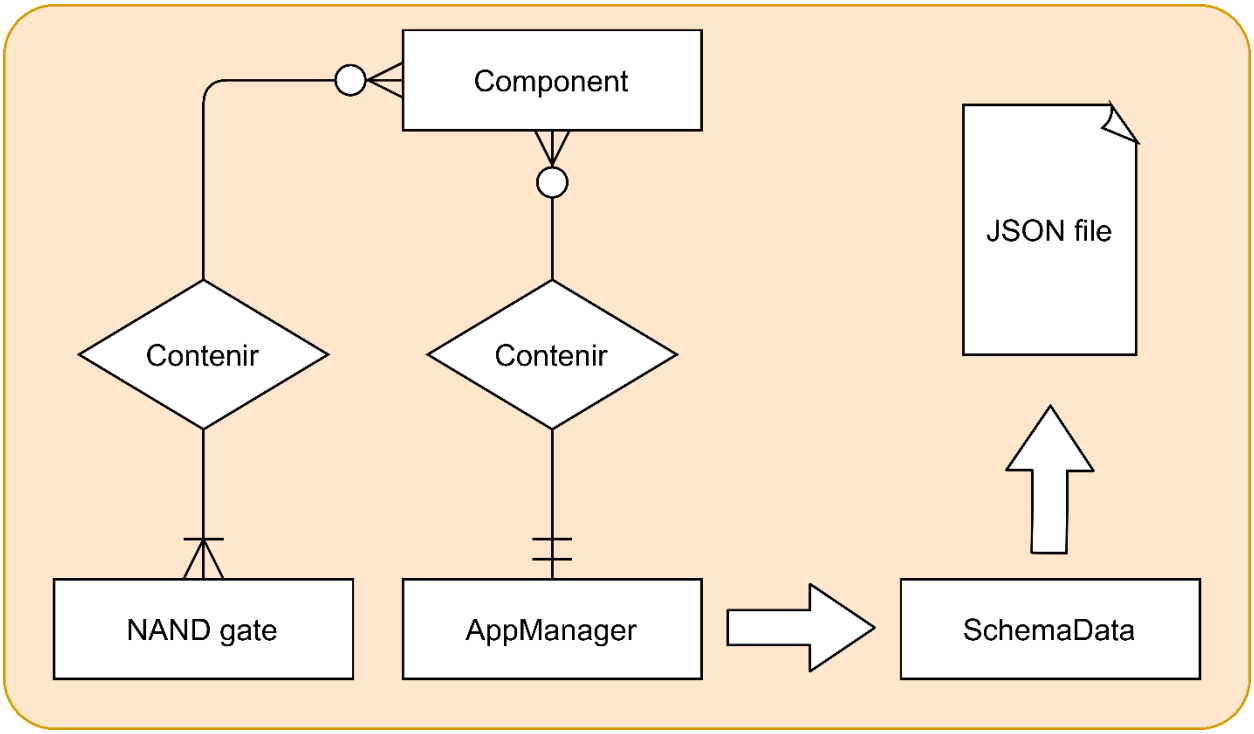
J'essaye au maximum de prendre le plus possible des logiciels gratuits et si possible open-source. J'aimerais que tout le monde puisse, à moindre coûts, suivre cette documentation et tester / réaliser les mêmes choses que moi. Voici les logiciels principaux que j'utiliserai :

- Unity 2020.2.2f1
- Visual Studio Code (version 1.53)
- Gantt Project Manager 2.8
- Git Extensions
- Draw.io

- Coolors.co
- NounProject.com
- Photopea.com
- Suite Office
- GitHub.com
- Jimdo.com

Modèle conceptuel de données

Pour ce projet, je souhaiterais faire deux types de fichiers de sauvegarde : un fichier complet contenant les informations sur tous les composants du schéma et un fichier ne contenant que l'expression algébrique du schéma. La logique de sauvegarde derrière restant la même, je ne vais présenter qu'un seul et unique schéma.



J'ai décidé d'utiliser du *JSON* comme format de fichier parce que c'est un format flexible et que je connais bien. Je l'ai utilisé pendant mon Pré-TPI et je n'ai pas eu de problème avec. De plus, *Unity* intègre directement dans ses

librairies un utilitaire *JSON* qui permet de le *parser/déparser* très simplement.

Programmation

Unity propose deux manières de programmer : soit via les *scripts* en *C#*, soit via les *blueprints*. Ces derniers permettent de faire théoriquement presque les mêmes choses que les *scripts C#* mais sans avoir à taper une seule ligne de code.

J'ai décidé d'utiliser la version textuelle pour plusieurs raisons :

1. Je connais beaucoup mieux le *C#*
2. La majorité des gens ne travaillent pas avec les *blueprints*, il y a donc moins d'aide en ligne
3. La technologie *blueprints* est plus difficile à optimiser (en terme de performances)

Ce n'est pas encore une certitude, mais j'aimerai bien toucher un petit peu au *shaders*. J'ai eu l'opportunité de m'y frotter un peu lors de mon stage où j'en ai créé un en *HLSL* pour *Unity*. La création de *shaders* est une discipline très spécifique et assez particulière. Si je n'ai pas le temps d'en faire ce ne sera pas un problème.

Pour le nommage j'utiliserai du *CamelCase* et du *pascalCase*. Ce sont les conventions de nommage que je préfère car elles sont rapides à écrire et élégante.

En *pascalCase* :

- Les *attributs*

En *CamelCase* :

- Les *classes*
- Les *méthodes*
- Les *dictionnaires*
- Les *énums*
- *GameObjects*

Lors de ce projet, j'aurai besoin de managers. N'appréciant pas travailler avec

les *classes statiques*, je vais préférer l'utilisation de *singletons* lors de ce projet. Ceux-ci sont pratique à utiliser, optimisés, et j'ai l'habitude de travailler avec lors de ce genre de projets.

Versioning

Il y a plusieurs solutions pour assurer le *versioning* d'un projet sur *Unity* : *Colaborate*, *GitHub*, SourceTree, *Git Kraken*, etc… Ayant travaillé lors de mon année de stage sur *Unity*, et ayant utilisé *Colaborate* également, j'ai pu me rendre compte de l'inefficacité de cette solution. Lors de mon pré-TPI j'ai utilisé un *asset* téléchargé sur l'*asset store* s'appelant *GitHub for Unity* ; c'était une catastrophe. J'ai eu énormément de problème de *merge* parce que j'avais travaillé sur deux ordinateurs différents.

Il me faut donc trouver une alternative gratuite, pratique et efficace. Suite aux conseils de Mr. Viret, je suis parti sur *Git Extensions*. C'est une interface graphique pour *Git* qui comporte beaucoup d'outils intéressant tout en restant relativement simple dans son fonctionnement.

Conception des maquettes

Étant à l'aise avec la manipulation du *webgiciel Draw.io*, j'ai décidé de l'utiliser pour réaliser les maquettes de l'application. J'aime cette application parce qu'elle est extrêmement complète tout en restant très simple dans sa mise en œuvre,

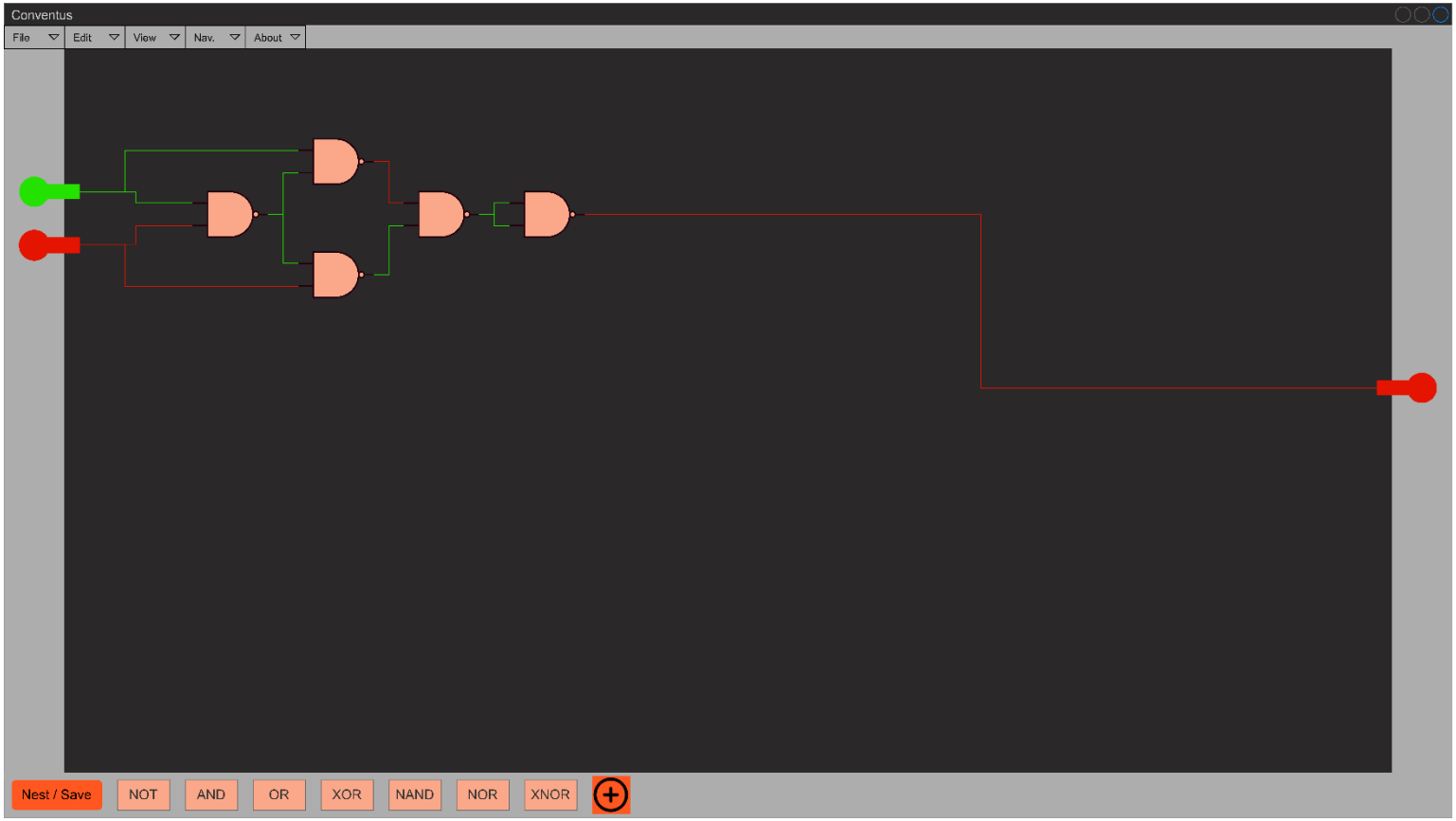
Contrairement à mon Pré-TPI, cette application n'aura qu'une seule et unique *scène*. Cela me simplifie beaucoup la gestion des managers, et cela me demande également moins de travail sur le design et la création de l'interface car tout est regroupé au même endroit.

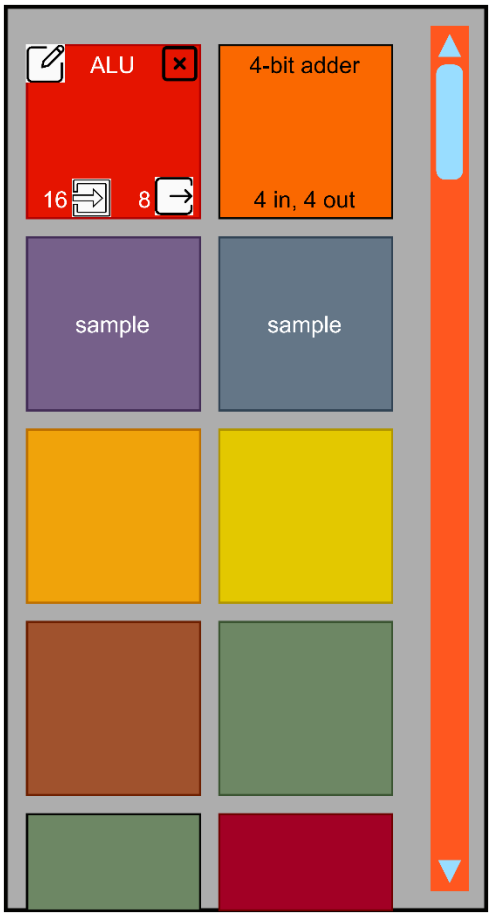
J'ai néanmoins plusieurs maquettes car j'ai également conceptualisé les panneau « pop-up » (*nesting* et *editing*) et le menu de gestion des composants.

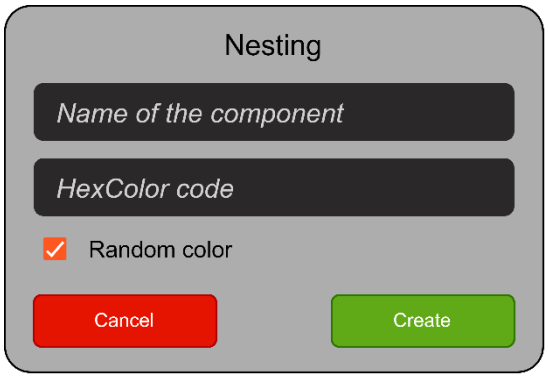
Sur les deux pages suivantes se trouvent mes maquettes que je viens de

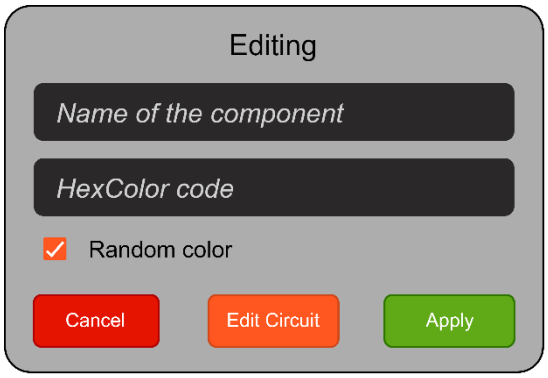
mentionner.

Présentation des maquettes




ALU
16
8
4-bit adder
4 in, 4 out
sample
sample


Nesting
Name of the component
HexColor code
Random color
Cancel
Create


Editing
Name of the component
HexColor code
Random color
Cancel
Edit Circuit
Apply

Palette de couleurs

Afin de s'assurer que le visuel de l'application soit agréable à l'œil, j'ai utilisé le site *coolors.co* pour générer une palette de couleurs cohérente. J'ai ensuite « peint » mes maquettes avec cette dernière. Je trouve que le résultat est satisfaisant. Voici la palette en question :

Raisin Black

Utilisé pour le canevas et pour le fond ce certains Input Fields

Silver Chalice

Pour le "cadre" de l'application. Permet de délimiter le canevas et de servir de toile de fond aux portes logiques, inputs et outputs

Light Salmon

Couleur de base des portes logiques et des boutons permettant de sélectionner celles-ci

International Orange Aerospace

Coches, boutons nest / save, bouton edit...

Apple Green

Signal-on color

Apple Red

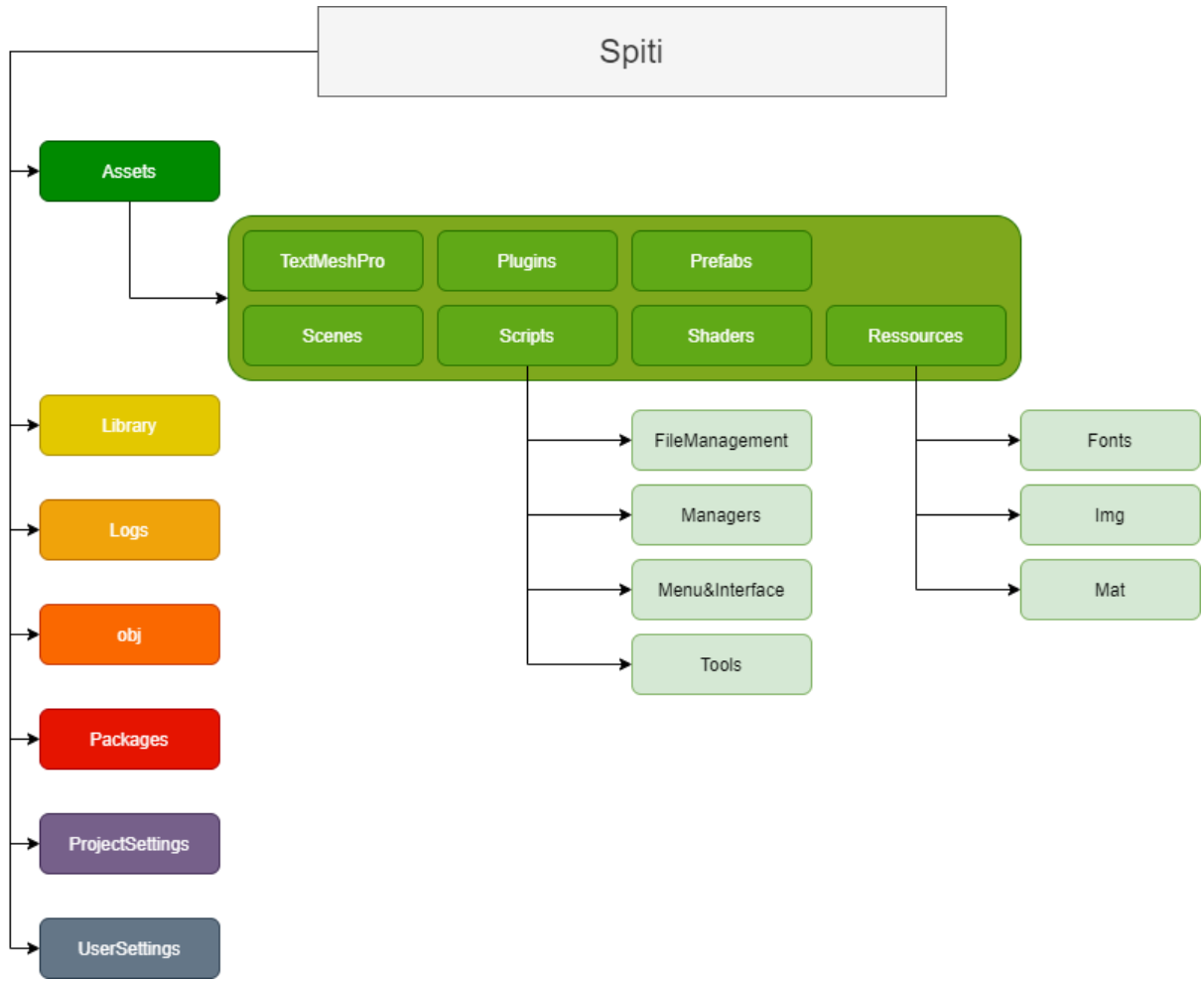
Signal-off color

Réalisation

Structure du projet

La structure de ce projet est plutôt simple et classique. À la racine du projet, il y a beaucoup de dossiers mais un seul nous intéresse, le dossier « *Assets* ». C'est dans celui-ci que se trouve toute notre application : *scripts*, images, polices, *shaders*, *scenes*, etc.

Le schéma ci-dessous devrait être assez explicite.



Je vais quand-même expliquer ce qui se trouve dans les dossiers principaux :

Assets : Contient tout ce qui se trouve dans l'application. C'est le contenu de

l'application.

TextMeshPro : C'est un plugin permettant d'avoir des textes sous forme de *meshs* (maillages / modèle 3D). Donne également plus de fonctionnalités sur la customisation des textes.

Plugins : Contient les plugins. Il y a notamment le *SVG* Importer.

Prefabs : Les *prefabs* sont des objets Unity stockés afin d'être ré-instanciés plus tard. Typiquement, on retrouve les canevas des *câbles, portes logiques* et des *components*.

Scenes : Ce dossier contient les *scènes* utilisées dans l'app. Il y en a que deux qui nous intéresse : le menu et la *scène* principale.

Scripts : Contient tous les *scripts* utilisés dans l'app. Je les ai classés par catégories qui me parlaient. Il y aurait plein d'autres manières de les classer…

Shaders : Dans ce projet, les *shaders* n'ont pas une très grande importance. Mais en général, c'est dans de dossier qu'on les stocke.

Ressources : C'est ici que se trouvent les images utilisées, les polices et les matériaux.

Library : Dossier où se trouvent toutes les librairies utilisées. Je n'ai pas eu besoin d'utiliser des libraires externes à *Unity*.

ProjectSettings : Paramètres du projet

UserSettings : Paramètres utilisateur

Description de l'application

Les components

La logique utilisée pour les *portes logiques* est assez simple, presque archaïque. En fait, c'est un *GameObject* auquel est attaché la *classe <Component >*. Cette dernière est commune à tous les éléments présents sur le canevas (à l'exception des câbles). Les *portes logiques* sont en quelques sortes un « cas particulier » de *component*. Cette classe à un *enum* intitulé sobrement « Type ». Il y a 10 types de *components : Buffer, NOT, AND, OR, XOR, NAND, NOR, XNOR*, Custom et Unset. Chaque *porte logique* à son type propre et il y a le type « Custom » pour

les *components* créés par les utilisateurs. Le type « Unset » est utilisé lorsque l'on ne sait pas quel rôle est sensé joué le *component*.

Déroulement du développement

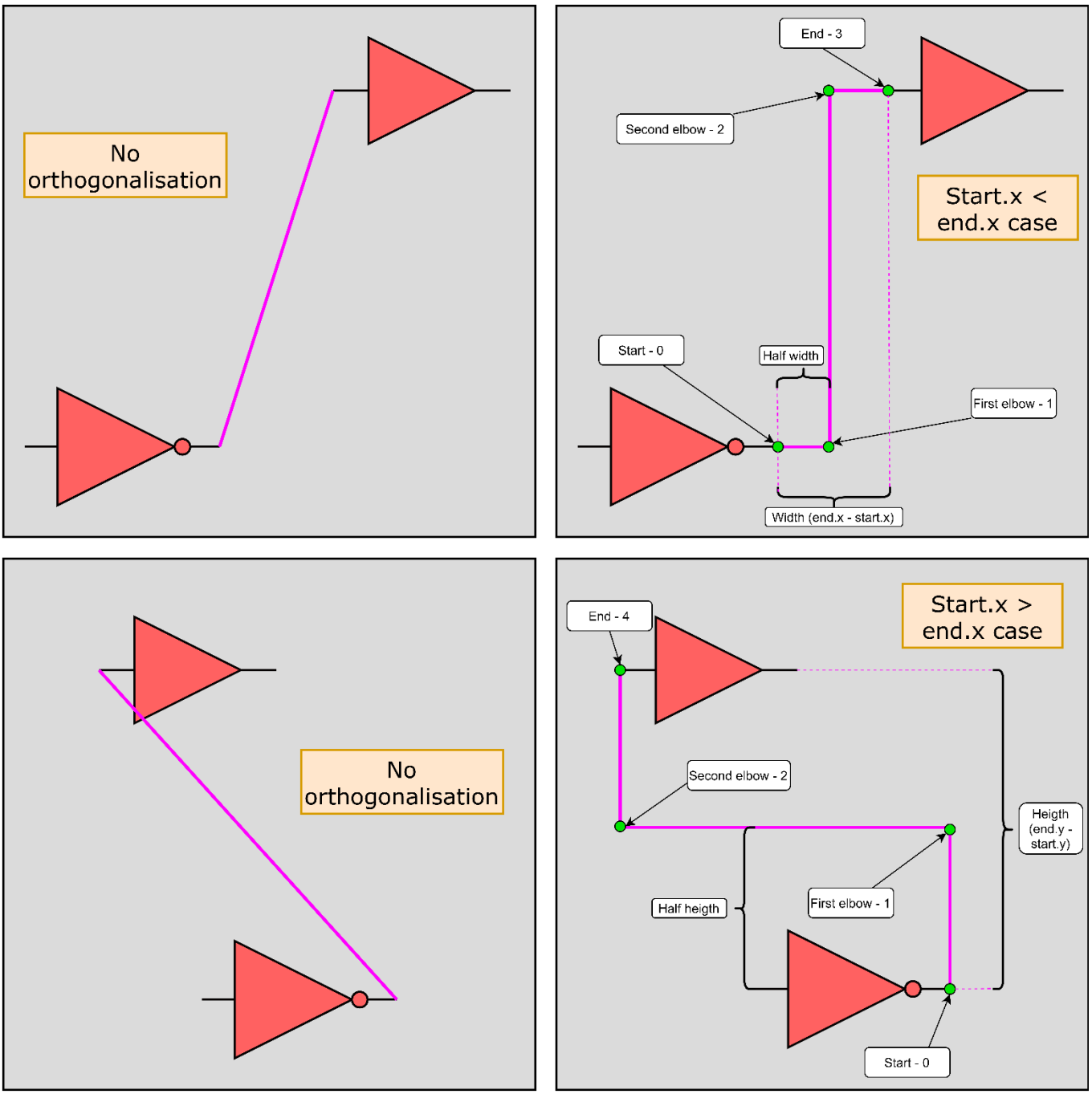
Interface de base

Portes logiques

Originellement, je me suis dit qu'il serait plus qualitatif d'utiliser des fichiers *SVG* pour représenter les *portes logiques*. J'ai trouvé une page WikiCommons mettant à disposition toutes les portes logiques des normes ANSI, IEL et DIN en téléchargement gratuitement. J'ai donc utilisé ces fichiers pour la première version des portes logiques dans l'application. Afin de pouvoir afficher du *SVG* dans *Unity*, il est nécessaire d'ajouter un *asset* s'appelant *SVG Importer*. C'est un *asset* développé par l'équipe d'*Unity* qui permet d'importer et de modifier les fichiers *SVG*. Malheureusement, c'est *asset* est encore ne développement et il existe que lamentablement peu de documentation dessus. J'ai essayé de faire des opérations simples avec comme par exemple changer la couleur des traits ou encore leur largeur, mais je n'y suis pas arrivé. Décidant de ne pas perdre du temps avec une technologie encore en développement, je suis passé sur du *PNG*. Avec le *PNG*, on perd en qualité et en flexibilité mais on gagne en facilité de manipulation.

Câbles

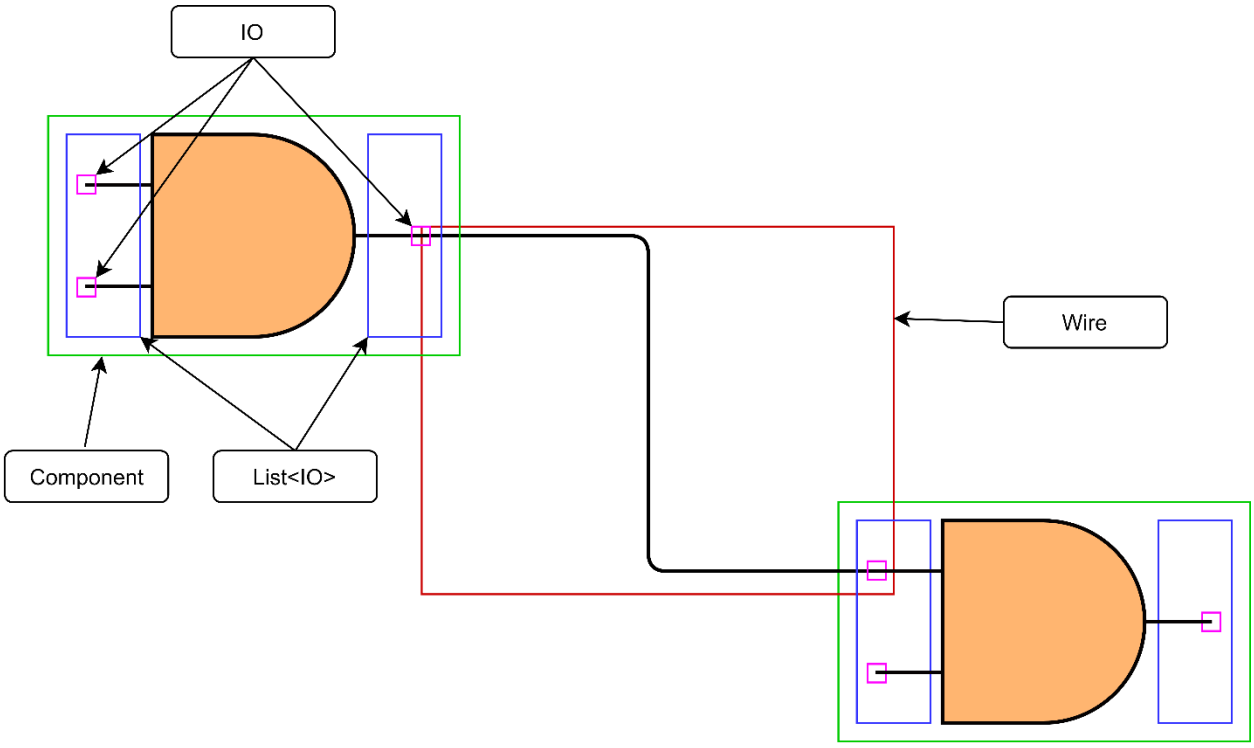
Les câbles relient deux IO entre eux. Ils adoptent automatiquement une forme orthogonale afin d'obtenir un rendu plus facilement lisible. Le *Line Renderer* permet de changer dynamiquement et facilement les propriétés du câble comme la couleur, l'épaisseur ou alors la position. On peut également utiliser un dégradé pour colorer le câble ; c'est pratique pour afficher visuellement la « progression » du signal dans ceux-ci. Ci-dessous un schéma expliquant comment fonctionne l'orthogonalisation des câbles (ne pas confondre avec l'orthogonalisation vectorielle via la méthode de Gram-Schmidt).



L'algorithme d'implémentation est très simple. Il suffit de prendre la position latérale entre l'IO de départ et celui d'arrivée et d'en faire la différence. Si le premier est plus grand que le second, on effectue une orthogonalisation sur l'axe Y, sinon sur l'axe X. Lorsque l'on déplace une porte reliée à une autre, la transition entre les deux types d'orthogonalisation se fait sans qu'on la remarque parce que pour passer de l'un à l'autre on passe par un stade où la différence vaut 0.

Synergie câbles – components

Un des premiers problèmes sérieux rencontré lors du développement de l'application, c'est le fait que les câbles ne se déplaçaient en même temps que les *components*, ce qui pose un problème de taille. Après avoir essayé plusieurs méthodes de fonctionnement différentes, une solution a finalement été trouvée. La première solution envisagée était bonne sur le concept mais très complexe dans les faits, rendant la solution difficile à débugger. Voici un schéma représentant le fonctionnement premièrement envisagé :



IO	
bool	input
bool	state
int	index
Component	component

Component	
List<Component>	components
bool[,]	truthTable
List<IO>	inputs
List<IO>	outputs
string	booleanExpression
Component.Type	type

AppManager	
List<Component>	canvasComponents
List<Component>	selectedComponents
List<IO>	inputs
List<IO>	outputs

Wire	
IO	inputIO
IO	outputIO
bool	state
Component	inComponent
Component	outComponent
LineRenderer	wire

Explication des classes :

IO

- Input : booléen permettant de savoir si ce *script* est attaché sur un *component* en tant qu'*input* ou en tant qu'*output*.

- State : état de la broche. Soit allumé (reçoit ou envoie le signal « true ») soit éteint (reçoit ou envoie le signal « false »)
- Index : donne la position du *script* dans la liste
- Component : indique quel est le *component* auquel est relié cet IO. Valeur « null » si pas relié

Component

- Components : Liste de tous les composants qui compose ce composant. Cela permet d'avoir des *components* fait par d'autres *components*, et ce de manière récursive.
- Truth Table : tableau en deux dimensions permettant de stocké la table de vérité du composant.
- Inputs : liste de tous les *scripts* IO jouant le rôle d'*inputs* sur ce composant
- Outputs : liste de tous les *scripts* IO jouant le rôle d'*outputs* sur ce composant
- Boolean Expression : expression booléenne sous forme de texte représentant le composant
- Type : donne le type du composant. C'est-à-dire, soit une *porte logique* soit un composant « Custom »

AppManager :

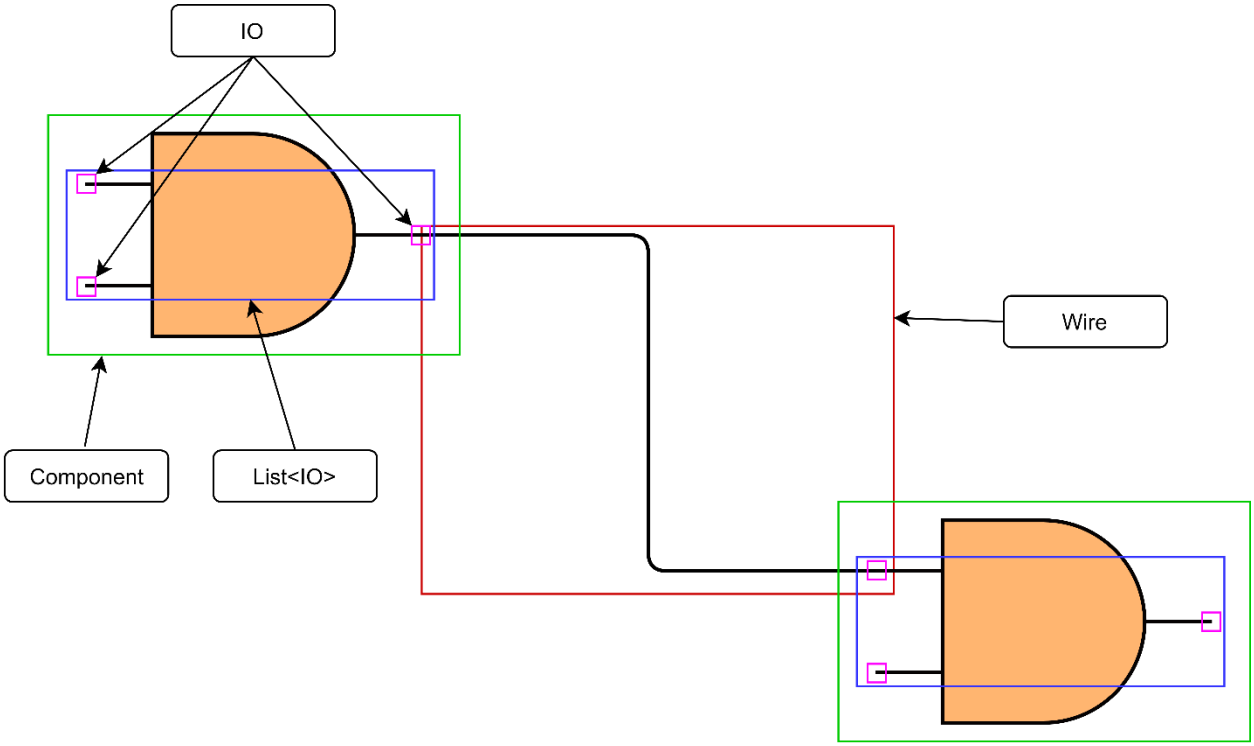
- Canvas Components : liste contenant tous les *components* sur le canevas
- Selected Components : liste contenant tous les *components* actuellement sélectionnés
- Inputs : liste de toutes les entrées du canevas
- Outputs : liste de toutes les sorties du canevas

Wire :

- Input IO : référence vers l'IO relié en entrée
- Output IO : référence vers l'IO relié en sortie
- State : état du câble (allumé étant « true » et éteint étant « false »)
- In Component : composant relié à l'entrée du câble
- Out Component : composant relié à la sortie du câble

- Wire : un *Line Renderer* est un composant *Unity* permettant de dessiner des lignes. Il y a beaucoup de paramètres utiles

Après avoir constaté l'inefficacité de ce modèle, j'ai décidé de supprimer les IO et de directement relier les *components* entre eux. Je me suis dit que ça allait simplifier le tout. Grave erreur ! en effet ne pas avoir de *classe* « parapluie » qui stocke toutes les infos relatives à une entrée ou une sortie signifie qu'il est obligatoire de stocker toutes ces informations dans la classe *component* directement. Après avoir essayé, sans succès, j'ai décidé de réintégrer la classe « IO » au système mais en simplifiant un peu la synergie entre les différentes classes. Voici un schéma montrant le nouveau fonctionnement ainsi que les nouvelles *classes* et leurs *attributs* :



IO	
bool	input
bool	state
Vector2	pos
IO	linkedIO

Component	
BoxCollider2D	boxCollider
bool[,]	truthTable
List<IO>	ios
string	booleanExpression
Component.Type	type

AppManager	
List<Component>	canvasComponents
List<Component>	selectedComponents
List<IO>	inputs
List<IO>	outputs

Wire	
IO	start
IO	end
bool	state
LineRenderer	wire

Explication des classes :

Je ne vais pas réexpliquer ce qui l'a déjà été par soucis de concision.

IO :

- Pos : position du *GameObject* sur la *scène* auquel le *script* est attaché
- Linked IO : référence à l'IO auquel celui-ci est lié. Valeur nulle s'il n'est pas relié

Component :

- Box Collider : *collider* correspondant à la taille de la *porte logique*
- IOS : liste de toutes les entrées et sorties du composant

Wire :

- Start : référence vers l'IO relié en entrée
- End : référence vers l'IO relié en sortie

Analyse du journal de travail

Problèmes rencontrés

Rétrospective

Erreurs restantes

Pour aller plus loin

Tests effectués

Documents fournis

Conclusions

Annexes

Manuel d'utilisation

Archives du projet