



UNIVERSITÉ DE BORDEAUX

PROJET SEMESTRE 8

PROJET INFORMATIQUE INDIVIDUEL

ENSC DEUXIÈME ANNÉE

Simon AUDRIX

Table des matières

I		3
1.	Rappel du besoin	4
2.	Description du projet & Choix de conception	4
2.1.	La filière sèche	4
2.2.	Le modèle poteau poutre	4
II		5
1.	Architecture globale	6
2.	OBJ Parsing	6
2.1.	Le choix du format	6
2.2.	L'OBJ comment ça marche	7
2.3.	Pseudo code	8
2.4.	Défauts et améliorations	9
3.	Positionnement de la vue	10
4.	Un format de donnée Back-End	11
4.1.	Building Blocs	11
4.2.	Méthodes et spécificités	12
4.3.	L'héritage des éléments	13
5.	Persistance des données	13
5.1.	Architecture BDD	14
5.2.	Utilisation de modèles	14
5.3.	Défauts et améliorations	14
6.	Communication entre Script et les arcanes d'Unity	15
6.1.	HashTable	15
6.2.	custom events	16
7.	Contrôles utilisateurs	17
7.1.	touches and clics - RayTracing	17
7.2.	Affichage modal et utilisation des prefabs	18
7.3.	Plan de coupe et Transparence - Shader	19
8.	Descente des charges	19
8.1.	Les charges	19
8.2.	Implémentation	21
9.	Modèle poutres	22
9.1.	Introduction	22
9.2.	Calculs nécessaires	22
9.3.	Retour aux valeurs limites et vérifications	24
10.	Modèle poteau	24
10.1.	Introduction	24
10.2.	calculs nécessaires	25
10.3.	Retour aux valeurs limites et vérifications	26
11.	Conclusion technique	26

III		27
1.	Vue principale et architecture	28
1.1.	Chargement d'un asset	28
1.2.	boutons & contrôles	29
1.3.	sélection	29
2.	objectPanel	29
2.1.	Affichage modal	29
2.2.	Mise à jour possible des objets	30
3.	Update panel	30
3.1.	Gestion par onglets et chargements dynamiques	30
4.	Conclusion	31
IV		32
1.	Retour sur le projet	33
2.	Améliorations envisagées et pistes de recherches	33



Première partie

INTRODUCTION & CONTEXTE

1. Rappel du besoin

En effet, un architecte travail rarement seul, il à pour objectif la conception et la réalisation d'un bâtiment mais doit aussi s'assurer que sa création tiendra debout une fois réalisée. Il lui faut choisir des matériaux et une structure qui au delà de leur esthétique respectent des normes de sécurité évitant par exemple qu'une maison s'effondre sur ses habitants, ou qu'un pont croule littéralement sous le poids des voitures.

L'architecte ou son cabinet n'est donc jamais seul lors de ces vérifications. Ce travail se fait de pair avec un bureau d'étude regroupant des ingénieurs en génie civil qui vont s'assurer dans les moindres détails qu'un plan est viable avant d'en autoriser la construction. Une part de ce travail de structure revient néanmoins à l'architecte qui réalise lors de la création de son projet ce que l'on appelle la phase de dimensionnement. Il doit en effet choisir des matériaux et des structures adaptés à ses envies conceptuelles et à la portance structurelle du bâtiment. Le choix de l'architecte n'est pas toujours accepté dans les bureaux d'étude. Ces erreurs créent alors des va-et-vient entre les cabinets et les bureaux d'études ce qui engendre régulièrement du travail supplémentaire, des coûts et des délais qui reviennent parfois cher et peuvent entraîner jusqu'au report ou à l'annulation de certains projets.

C'est dans ce cadre que vient s'inscrire le projet CiViLi.

2. Description du projet & Choix de conception

Le projet CiViLi est réalisé dans le cadre d'un projet informatique individuel de l'ENSC. Il a pour objectif de venir assister les architectes lors de la phase de dimensionnement afin d'assurer une meilleure flexibilité et un plus grand tôt d'acceptation de projet dans les bureaux d'études.

L'objectif est de permettre à un architecte de donner un modèle 3D au logiciel CiViLi et lui donner la possibilité de renseigner les informations sur la structure afin de lui fournir en retour des informations quand à la résistance et à la faisabilité de l'ensemble. L'objectif et de prévenir un maximum au défauts de conception afin de diminuer les risques structurels du projet.

Le projet actuellement lié à ce document, c'est à dire CiViLi-V1 n'est cependant pas un produit final. Il s'agit en effet d'un projet en cours. Sur une durée de moins de six mois, il paraît improbable de reprendre toute les problématiques relevant à la fois de la conception des bâtiments, et de la résistance des matériaux. C'est pourquoi de nombreux choix de conception ont du être fin afin de produire une *proof of concept* raisonnable et faisable en une durée si courte.

2.1. La filière sèche

L'un des premiers choix qui a été effectué est celui de se concentrer sur ce que l'on appelle la filière sèche. Il s'agit de méthode de construction dites sèches, par opposition aux anciennes méthodes dites "humides". Ces dernières nécessitent peu d'eau et utilisent des matériaux avec une faible empreinte carbone comme le bois notamment. Cette technique s'inscrit donc dans une démarche éco-responsable et semble être bien partie pour devenir l'une des principales méthodes de construction employées dans un futur proche. On note principalement l'apparition des premiers gratte-ciels en structure bois. C'est une architecture propre, et plus rapide car ne nécessitant pas de temps de séchage. Cela paraît donc un bon compromis pour piocher des matériaux modernes et inscrire le projet dans un développement en accord avec les problématiques de l'architecture moderne.

2.2. Le modèle poteau poutre

De même, le champ de l'ingénierie civile est très étendu. Afin de ne pas s'y perdre, il faut réduire le champ des possibles et ne prendre pour démarrer que le strict minimum. J'ai choisi de me focaliser sur ce que l'on appelle le modèle poteau poutre qui est très largement utilisé dans la construction classique notamment dans les maisons personnelles ainsi que dans les immeubles d'habitations par exemple. En plus d'être très répandu ce modèle présente l'avantage d'être relativement simple à appliquer d'un point de vue physique.



Deuxième partie

DÉVELOPPEMENT & FONCTIONNALITÉS

Dans cette partie, je vais reprendre l'intégralité des fonctionnalités mise en place en détaillant leur fonctionnement, leur implémentations et si besoin leur bases physiques et/ou mathématiques.

1. Architecture globale

La solution CiViLi est réalisé à l'aide du moteur Unity et de C#. Elle prend la forme d'une application bureau déconnectée. Elle se compose d'un mélange d'éléments Unity (matériaux, textures, UI...), d'une base de données embarquée et de script C# permettant le fonctionnement de l'ensemble.

Ci-dessous une rapide description des différents scripts réalisés et de leur objectifs. Chacun de ces scripts sera revu plus en détails afin d'en montrer les objectifs et les subtilités de fonctionnement.

Parser Il s'agit d'une classe statique contenant l'ensemble du code nécessaire au parsing des fichiers importés par l'utilisateur.

UIManager Il s'agit du code principal qui permet de lier les actions de l'utilisateur dans Unity au reste du modèle et aux données.

BuildingBloc Il s'agit d'un format de donnée *back-end* contenant les dénominateurs communs à tout les éléments porteurs d'une structure.

Poutre Héritant de *BuildingBloc*, poutre permet au logiciel de maintenir les données nécessaires aux calculs de contraintes sur les éléments de type poutre du modèle.

Poteau Héritant de *BuildingBloc*, poteau permet au logiciel de maintenir les données nécessaires aux calculs de contraintes sur les éléments de type poteau du modèle.

PoutreModel Il s'agit d'une classe permettant de lier les données enregistrées dans la base de données de CiViLi pour les injecter dans nos éléments de structure.

MaterialModel C'est la classe permettant de charger les différentes données des matériaux dans la base pour effectuer des calculs et de l'affichage dans le logiciel.

2. OBJ Parsing

La première étape du projet était d'afficher à l'utilisateur des éléments 3D dans le viewport de l'application . Il fallait donc permettre d'importer un fichier fait par l'utilisateur. Dès lors, les problèmes avec le framework proposé par Unity ont commencés. En effet, il est très simple d'importer un fichier dans Unity lors du développement. Il s'agit de glisser-déposer un fichier dans les *Assets* d'Unity pour qu'il soit automatiquement chargé (géométrie, textures, matériaux...). Cependant pour cela, il faut disposer à l'avance du modèle 3D ce qui n'est pas le cas dans CiViLi, en effet le modèle doit être affiché "*at runtime*" depuis un fichier de l'architecte.

Ce problème m'a donc obligé à développer un système permettant de parser des données 3D et de les reconstruire dans Unity.

2.1. Le choix du format

Pour commencer, la question du format à choisir s'est posée. Il a fallu faire un choix parmi de nombreux formats de données 3D existants (wavefront, collada, FBX, STL, IGES...). Chacun dispose de ses avantages et de ses inconvénients. Après quelques recherches, le format OBJ à présenté deux principaux avantages :

- Il est possible d'exporter ses créations en OBJ facilement depuis la grande majorité des logiciels de modélisation en architecture (AutoCad, Maya, Sketch-up, Blender...)
- Les fichiers OBJ sont particulièrement simples à comprendre et à utiliser et donc par extension assez simple à parser.

2.2. L'OBJ comment ça marche

Comme mentionné précédemment, les fichiers OBJ sont très simple à comprendre, ils prennent la forme de fichier texte et utilisent un identifiant à chaque ligne pour décrire les données qu'elle contient.

Ci-dessous, vous trouverez une description des lignes les plus importantes que l'on peut retrouver dans un fichier OBJ, de leur identifiant et de la manière dont elle stocke les données.

Commentaire

structure : # "Lorem Ipsum"

La ligne de commentaire est identifiée par un # et suivi de texte elle permet de contenir des données utiles pour l'utilisateur ou le développeur mais pas à la description de l'objet

Objet

structure : o "Name"

La ligne est identifiée par un o et suivi du nom de l'objet. Elle permet de signifier que l'on commence la définition d'un nouveau mesh et peut se retrouver plusieurs fois dans un même fichier si ce dernier décrit un modèle complexe.

Groupes

structure : g "Name"

La ligne est identifiée par un g et suivi du nom d'un objet elle permet de signifier le début de la déclaration d'un nouvel objet ou groupe d'objet.

Vertex

structure : v float float float

La ligne de description d'un vertex¹ permet de décrire un point. Elle est identifiée par un v suivi des trois coordonnées x, y et z du point séparées par des espaces.

UV

structure : vt float float

La ligne de description d'un UV² permet de décrire un point de texture. Elle est identifiée par un vt (le t signifiant texture) et suivie des deux coordonnées u et v du point séparées par des espaces.

Les normales

structure : vn float float float

La ligne de description d'une normale³ est identifiée par un vn et se compose des trois coordonnées servant à décrire le vecteur normal. Cependant Unity et ses outils nous permettent de négliger les données apportées par le fichier lors de notre lecture de celui-ci.

1. Un Vertex est le nom d'un point dans l'espace 3D

2. Un UV ou point d'UV est le nom donné à un point associé à une texture

3. En modélisation 3D, la normale à un plan représente un vecteur perpendiculaire à ce dernier, c'est à dire dont le produit vectoriel avec cette face est nul. Cette donnée est principalement utilisé par les logiciels de modélisation ou les moteurs de jeux comme Unity pour gérer des problématiques liées aux lumières et à l'éclairage. Les normales sont également utilisées pour des questions de performances.

Les faces

```
structure : f v1/vt1/vn1 v2/vt2/vn2 v3/vt3/vn3
```

La ligne de description d'une face permet de décrire un polygone d'un mesh. Elle est identifiée par un **f** et se compose de trois triplets. Chaque triplet contenant chacun une référence entière à respectivement :

- v : un vertex
- vt : un point de texture
- vn : un point de normal

Les matériaux

```
structure : usemtl "Name"
```

La ligne matériaux est identifiée par le code **usemtl** et est suivi d'un nom de fichier. Ce code sert à référencer un fichier attenant à l'OBJ le *.mlt* qui contient des données sur les matériaux appliqués aux objets lors de la phase de modélisation. Ces données sont pour l'instant ignorées par le parser et par le logiciel pour différentes raisons qui seront expliquées plus tard dans ce document.

2.3. Pseudo code

Le parser effectue donc trois tâches. Premièrement, il parcours et traite les données du fichier OBJ. Avec ces données il crée des Mesh. Pour finir il encapsule chacun des Mesh dans des *GameObject*⁴ en ajoutant les composants nécessaires à son bon fonctionnement. De plus, le Parser à pour spécificité d'être une classe statique, c'est à dire qu'il peut être simplement utilisé dans le code sans être instancié. L'UIManager qui l'appelle n'a donc pas besoin de s'encombrer d'une référence à un objet qui n'est utilisé qu'une fois. Il lui suffit d'appeler **Parser.ParserFile(Path)** pour lancer la lecture d'un fichier.

Le fonctionnement du parser est décrit ci dessous de manière simplifié :

```
chargerFichier(chemin d'accès)  
offset = 0  
  
pour chaque ligne dans le fichier  
#on regarde le code de ligne  
{  
    v -> on ajoute un Vector3 dans la liste de Vertex  
    vt -> on ajoute un Vector2 dans la liste d'UVs  
    f -> on ajoute l'index du point référencé par le fichier moins l'offset  
    o ou g -> on créer un nouvel objet avec les données actuelles  
    offset += nombre de points de l'objet que l'on vient de créer  
}
```

Traiter un vertex ou un uv est simple. Il s'agit de créer un Vector3 ou Vector2 qui sont les format Unity permettant de décrire des points dans l'espace (3D ou 2D) et de les ajouter dans une liste qui est un attribut du parser. La partie la plus compliquée est celle du traitement des faces et de la création des objets.

Tout d'abord le traitement des faces. Une face référence des index de points par exemple la face **f 1/_/_ 2/_/_ 3/_/_** référence les points 1, 2 et 3 de la liste de points. Cette indice est cependant global. Lorsque l'on arrive dans le second objet qui n'utilise par exemple que les points de 15 à 20 on n'a qu'une liste de 5 points. Il faut donc retirer un "offset" de 15. A chaque fois que l'on change d'objet, l'offset augmente d'autant que le nombre de point que contenait l'objet précédent.

4. Les GameObjects sont les composants de bases d'Unity. Ils sont utilisés pour représenter n'importe quel objet que l'on souhaite utiliser. On peut ajouter à un GameObject des Scripts ou des Composants pour modifier ou personnaliser son comportement

Une fois que l'on a créé les paramètres nécessaires on peut alors créer un Mesh et un GameObject pour permettre à Unity de représenter notre bâtiment. L'étape n'est pas simple non plus en effet pour que l'objet fonctionne correctement, s'affiche à l'écran interagisse avec la lumière... Il faut lui attribuer un certain nombre de composants.

Le code est affiché et expliqué ci-dessous :

```

129     static GameObject CreateGameObject(string name)
130     {
131         Vector2[] uvs = CreateUVs();
132
133         // On crée le mesh représenté par le fichier
134         Mesh mesh = new Mesh();
135         mesh.Clear();
136         mesh.name = name;
137         // On insère les données dans le mesh
138         mesh.vertices = CurrentPoints.ToArray();
139         mesh.uv = uvs;
140         mesh.triangles = Faces.ToArray();
141         mesh.RecalculateNormals();
142
143         // On crée le gameObject et ses composants nécessaires
144         GameObject obj = new GameObject();
145         obj.AddComponent<MeshFilter>(); // permet de donner un mesh à l'objet
146         obj.AddComponent<MeshRenderer>(); // permet d'afficher le mesh qui constitue l'objet
147         obj.AddComponent<MeshCollider>(); // permet de donner des pp de collision au mesh
148
149         obj.GetComponent<MeshFilter>().mesh = mesh;
150         obj.GetComponent<MeshCollider>().sharedMesh = mesh;
151         obj.name = name;
152
153         return obj;
154     }
155 #endregion
156 }
```

Ici on crée dans un premier temps le **Mesh** c'est lui qui va contenir les données propres à la géométrie. Il faut lui donner *vertices* un tableau de points (Vector3) il contient autant de points qu'en contient l'objet. Ensuite il faut lui donner *triangles* un tableau d'entier. Ce dernier doit absolument avoir une longueur multiple de trois. En effet, Unity comme la plupart des moteur de jeux ne fonctionne qu'avec des Mesh triangulés c'est à dire dont l'ensemble des faces sont des triangles. Si le tableau n'est pas de taille adéquate alors le logiciel déclenchera une erreur. Et enfin on doit lui donner *uv* un tableau de points (Vector2) qui doit contenir autant de point qu'en contient l'objet. Pour finir, on utilise la méthode **RecalculateNormals()** ; qui nous permet de régénérer les normales du mesh à partir de sa géométrie. Cela évite d'avoir à récupérer les normales du fichier de données permettant un gain de temps.

Dans un second temps on instancie un nouveau **GameObject** c'est lui qui permettra à Unity d'afficher nos données. Nous devons attacher à cet objet des composants pour permettre son fonctionnement :

MeshFilter Ce dernier permet de donner un ou plusieurs Mesh à l'objet que l'on créer grâce à sa propriété *mesh*

MeshRenderer Comme son nom l'indique, il permet de *rendre*, c'est à dire d'afficher à l'écran les données du Mesh-Filter. Sans ce composant impossible de voir le Mesh que contient le GameObject.

MeshCollider enfin, le MeshCollider permet de créer certaines propriétés physiques dans le moteur Unity. Il est nécessaire de l'ajouter pour avoir accès plus tard à certaines propriétés des objets nécessaires aux calculs comme par exemple les *BoundingBox*

L'objet dispose également d'un nom qui nous permet de le repérer dans la hiérarchie d'Unity et de le retrouver dans le code. Ce nom est celui contenu dans les fichiers OBJ et donc choisi par l'utilisateur lors de la modélisation.

2.4. Défauts et améliorations

Pour le projet CiViLi, il a été nécessaire de créer un parser d'OBJ cependant ce dernier reste sommaire et ne comprend que très peu de fonctionnalités. Il est seulement suffisant pour afficher des objets et retranscrire leur géométrie. De nombreux problèmes subsistent je ne vais pas tous les énumérer, mais certains sont assez importants pour mériter

d'être mentionnés.

Premièrement, pour des raisons de simplicité le traitement des matériaux n'est pas effectué à cette étape du code car elle nécessiterait des modifications lors du pré-traitement des données qui prendrait du temps à réaliser. Cependant, il serait possible de charger dès l'étape de parsage les données liées aux matériaux de l'utilisateur et de les lier dans un second temps avec les matériaux "physique" utilisés par le logiciel pour les différents calculs de résistance.

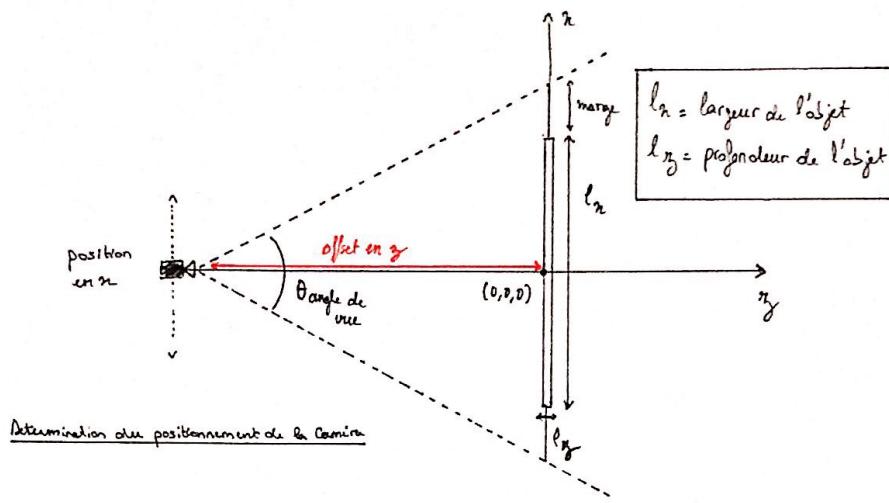
Deuxièmement, et ce point est important, les UV actuellement donnés lors du chargement d'un fichier sont incorrects. En effet, ils sont calculés en fonction des coordonnées 3D du point et pas des données du fichier car le format OBJ est incompatible en l'état avec le fonctionnement d'Unity. J'ai mentionné précédemment qu'il fallait fournir à Unity autant de coordonnées d'UV que de points ou dans un fichier OBJ on se retrouve souvent avec un plus grand nombre d'UV. Cela vient du fonctionnement de l'UV mapping⁵ en modélisation. En effet dans ce mapping, un point dans l'espace 3D peut avoir plusieurs locations dans l'espace des UV (espace 2D dans lequel vit la texture). Les données enregistrées dans le fichier OBJ ne peuvent donc pas être utilisées brutes dans Unity, elles nécessitent un traitement supplémentaire complexe qui a été mis de côté dans le cadre du PII pour des questions de délais. Cependant ce choix engendre des problèmes lors de l'affichage des textures sur le modèle dans le viewport. Une première solution pour améliorer le rendu sans gérer l'intégralité des UV pourrait être de normaliser les coordonnées utilisées pour les ramener dans la dimension de la texture.

3. Positionnement de la vue

Une autre problématique est celle de positionner la vue et donc la caméra. En effet les différentes créations importées dans le logiciel peuvent prendre différentes formes il faut donc positionner dynamiquement la caméra afin de permettre à l'utilisateur de visualiser l'ensemble de son modèle.

Pour placer dynamiquement la caméra, nous utilisons le Parser. Ce dernier retient les plus petites et les plus grandes coordonnées de points rencontrés lors du traitement du fichier. Il garde donc en mémoire les maximums et les minimums sur chaque axes ce qui nous permet de mesurer la taille maximale que fait l'objet.

Il nous faut donc calculer deux choses la position en x pour centrer la caméra sur l'objet et la distance en z et en y nécessaire à ce que l'objet rentre dans le champs. Le schéma ci-dessous exprime la situation :



Rappels de trigonométrie

Grâce à quelques bases en trigonométrie, on peut facilement calculer les données qui nous intéressent. C'est à dire la position en x et l'offset en z. Pour la position en x on l'obtient en divisant par deux la longueur de l'objet. Pour obtenir

5. L'UV mapping est le procédé qui vise à projeter un modèle 3D sur une texture 2D pour pouvoir l'appliquer à ce modèle. Comme déplier le patron d'un cube pour pouvoir dessiner dessus plus simplement.

notre offset en z c'est un petit plus compliqué. Unity nous permet de récupérer le *fieldOfView* c'est à dire l'angle de la prise de vue associé à la caméra principale (Angle Θ). Si on aligne la caméra avec le centre de l'objet, alors on crée un triangle rectangle. Grâce aux formules trigonométriques, on retrouve donc $\tan(\theta) = \frac{\text{opposé}}{\text{adjacent}}$. Ici on à θ qui vaut l'angle moitié de la prise de vue soit $\theta = \Theta/2$. Le coté adjacent est évidemment la longueur d'offset que l'on recherche et le coté opposé se mesure en prenant le maximum entre la largeur et la profondeur du bâtiment divisé par deux et en y ajoutant une marge.

On obtient donc facilement la relation suivante :

$$O_z = \frac{\frac{l}{2} + \frac{l}{6}}{\tan \theta} O_z = \frac{\frac{2l}{3}}{\tan \Theta/2}$$

ou O_z est l'offset en z, $l = \max(l_x, l_z)$ et Θ l'angle de vue de la caméra. La marge vaut $\frac{l}{6}$ cela permet de la rendre dépendante de la taille de l'objet. Plus un bâtiment est grand plus on a besoin de marges importantes.

Code

Ci dessous, on retrouve le code utilisé pour positionner la caméra Il est assez simple à comprendre, mais il me permet d'aborder la notion de *transform* :

```

710     void SetUpCamera()
711     {
712         // Recherche de l'axe principal et des tailles
713         float mainAxesMaxLength = Math.Max(Math.Abs(Parser.MaxX-Parser.MinX), Math.Abs(Parser.MaxZ-Parser.MinZ));
714         float secondaryAxisMaxLength = Math.Min(Math.Abs(Parser.MaxX-Parser.MinX), Math.Abs(Parser.MaxZ-Parser.MinZ));
715         float fov = Camera.main.fieldOfView;
716         float totalHeight = Math.Abs(Parser.MaxY - Parser.MinY);
717         //Calcul de l'offset
718         float widthNeededZ = (float)(Math.Tan(fov/2) * mainAxesMaxLength /2f );
719         // Calcul des position
720         float centerX = Parser.MaxX - secondaryAxisMaxLength / 2;
721         float posZ = mainParent.transform.position.z - Math.Abs(widthNeededZ);
722         float posY = Parser.MinY + totalHeight / 2f;
723
724         Camera.main.transform.position = new Vector3(centerX, posY, posZ);
725     }

```

Ici pour accéder au position de la caméra, on utilise le code : **Camera.main.transform.position** on peut aussi accéder à **transform.rotation**. L'objet transform est un composant qui est attaché à tous les GameObject. Ce dernier contient les informations relatives à la position (Vector3) et à la rotation (Quaternion) de l'objet dans l'espace. C'est donc la transform qu'il faudra manipuler pour déplacer ou tourner des objets dans l'espace. On peut notamment noter les fonction *transform.translate* et *transform.rotate* qui sont particulièrement utiles.

4. Un format de donnée Back-End

Une fois toutes ces étapes accomplies, on peut commencer à s'intéresser au calculs physiques. Bien que très pratique pour afficher et manipuler des objets graphiques, les GameObject d'Unity restant limités. En effet, ils est impossible de leur rajouter des données personnalisées. Pour contenir toutes les données nécessaires au calculs et au représentations physique telles que le volume, le poids ou encore le matériau il a fallu créer des classes métier adaptées.

4.1. Building Blocs

Ces classes métier sont au nombre de trois. Il s'agit de **BuildingBloc**, **poutre** et **poteau**. La première, BuildingBloc contient l'ensemble des données communes à tous les objets structurels et aux fonctionnement de ces derniers dans le logiciel.

Il contient les attributs privés suivant :

- *materialName*
- *isSelected*
- *isStructural*

materialName comme son nom l'indique sert à garder en mémoire le nom du matériau qui constitue l'élément. *isSelected* est un booléen qui sert à savoir si oui ou non l'élément est actuellement sélectionné par l'utilisateur dans le viewPort pour permettre de gérer l'affichage et les retours utilisateurs. *isStructural* permet de retenir si oui ou non l'objet

fait partie de la structure principale du bâtiment affiché. Par exemple une poutre est partie prenante de la structure alors que on le sait bien certains murs ne sont pas porteurs. Cela nous permet donc de modifier les propriétés d'un élément et de gérer l'affichage en conséquence.

La classe BuildingBloc contient également de nombreuses propriétés publiques qui sont affichées dans le code ci-dessous :

```

19  #region Properties
20  // Properties relatives to display
21  public string MaterialName
22  {
23      get{ return materialName; }
24      set{
25          materialName = value;
26          SetMaterial(value);
27      }
28  }
29  public GameObject RepresentedObject { get; set; }
30  // Properties relatives to calculation
31  public List<BuildingBloc> On { get; set; }
32  public List<BuildingBloc> Under { get; set; }
33
34  public float VolumicMass { get; set; }
35  public float Volume { get; set; }
36
37  public float P { get; set; } // Poids propre
38  public float G { get; set; } // Charges Globales
39  public float Q { get; set; } // Charges d'exploitation
40
41  public float ELS { get { return G + Q; } }
42  public float ELU { get { return (float)(1.35 * G + 1.5 * Q); } }
43
44  #endregion

```

Ici, on retrouve P, G et Q qui représente respectivement son poids (appelé poids propre en descente des charges), sa charge permanente et ses charges variables (aussi appelées charges d'exploitations). C'est trois grandeurs sont exprimées en Newton (N).

Il contient également V son volume mesuré en m^3 , et sa masse volumique (ρ) en kg/m^3

On retrouve aussi l'**ELS** et l'**ELU**. Ces deux grandeurs sont la charge limite de service et la charge limite ultime. Elle sont exprimées en Newton (N) et seront détaillées dans la section descente de charges.

Enfin chaque objet dispose de deux liste **On** et **Under** elle sont utilisées dans le procédé de descente des charges et contiennent les objets directement sur ou sous l'objet en question.

4.2. Méthodes et spécificités

Création et ajout du matériau

Tout élément structurel d'un bâtiment est constitué d'un matériau spécifique. Cependant, avec Unity nous sommes face à un problème de terminologie. En effet, en modélisation 3D, on assigne au Mesh des *matériaux* qui servent à gérer l'affichage et ne dispose que de propriétés visuelles (texture de surface, illumination, spécularité...) cependant il ne nous renseigne en rien sur les données physiques d'un matériau qui sont nécessaires pour effectuer les calculs physiques requis par le logiciel CiViLi. Nous devons donc assigner à notre objet deux "matériaux" un dans l'affichage qui permet à l'utilisateur de comprendre de quel matériau est constitué son objet et un au moteur de calcul. Le dénominateur commun entre ces deux matériaux est le nom qu'il porte. Dans les deux cas j'ai unifié les données afin que le matériaux visuels d'Unity et le matériaux stockés par le logiciel portent le même cela nous permet d'unifier les fonctionnement.

```

54  public void SetMaterial(string matName)
55  {
56      MeshRenderer renderer = RepresentedObject.GetComponent<MeshRenderer>();
57      renderer.material = GetUnityMaterial(matName);
58      float transparencyValue = isStructural ? 1f : 0.2f ;
59      renderer.material.SetFloat("transparency", transparencyValue);
60  }
61
62  public Material GetUnityMaterial(string materialName)
63  {
64      return (Material)Resources.Load("Materials/" + materialName, typeof(Material));
65  }
66

```

Sur cette figure, on peut voir la fonction **SetMaterial** celle-ci est appellée dans le *setter* de *materialName*, ainsi à chaque fois que l'on modifie le nom du matériau on effectue cette démarche.

Il se passe deux choses intéressantes dans cette fonction. D'abord on note que l'accès au matériaux d'Unity se fait via le composant *MeshRenderer*. On voit également que le matériau est chargé par une fonction annexe (*GetUnityMaterial*) cette dernière permet d'accéder aux ressources de l'application. Unity nous permet de charger dans nos scripts des ressources à condition que celle-ci soit rangées correctement dans l'arborescence dans un fichier Resources. Une fois

chargée, il faut la transtyper en l'objet voulu ici, on utilise :

```
Resources.Load("Materials/" + materialName, typeof(Material));
```

Le fichier se trouve donc au chemin : *Ressources/Materials/bois* par exemple. Cette ligne de code peut également s'utiliser sous la forme :

```
Resources.Load<Material>("Materials/" + materialName)
```

Enfin, la modification de la transparence du matériau dépend du statut structurel de la pièce elle implique la le shader⁶ du matériau qui sera décrit dans la partie portant sur ce sujet.

Calcul du poids propre

Pour calculer le poids propre de l'objet, rien de plus simple, il suffit de faire appel à ses souvenirs de physique élémentaire. On peut alors se rappeler de la formule du poids : $P = m \times g$ et de la formule de la masse volumique : $\rho = \frac{m}{V}$ en les combinant, on obtient facilement :

$$P = \rho \times V \times g$$

avec g la constante d'intensité de pesanteur qui vaut environ $9.8N/km^{-1}$, ρ la masse volumique du matériau constituant l'objet en kg/m^3 et V le volume en m^3 . On obtient donc le poids P en Newton.

Calcul du Volume

Il ne nous reste plus qu'à calculer le volume de l'élément. Pour ce faire, on utilise un processus itératif qui consiste à parcourir l'ensemble des triangles (faces) qui constituent le Mesh et à calculer le volume signé de ce triangle. En effet un triangle en soit n'a pas de volume on construit donc artificiellement un tétraèdre qui à pour sommet l'origine du plan (le point (0,0,0)). En ajoutant l'ensemble de ces volume on obtient alors le volume de n'importe quel Mesh même de forme complexe.

4.3. L'héritage des éléments

Différentes parties d'un bâtiment sont soumises à différentes contraintes, il faut donc différentier les murs des sols, la charpente des portes et ainsi de suite. Pour l'instant le programme est capable d'utiliser les poutres et les poteaux. Ils sont chacun représentés par une classe qui Hérite de BuildingBloc. Cela permet d'utiliser à notre avantage les processus de l'héritage et de donner aux poteaux et aux poutres toutes les propriété d'un bloc de construction détaillées dans la partie précédente et de leur rajouter des propriété et des méthodes spécifiques.

Ces deux classes sont donc créées pour supporter les modèles poutre et poteau qui sont des modèles physiques de résistance des matériaux. Les différentes fonctions et propriété découle en grande partie des nécessités de calcul de ces modèles. Le détail de ces classes sera donc donné dans la partie décrivant les calculs liés à ces deux modèles.

5. Persistance des données

Pour conserver toutes les données relatives notamment aux matériaux mais aussi on le verra aux différents éléments structurels, il est nécessaire de prendre en compte un système de persistance des données. C'est alors que se pose la question du format du **SGBDR**⁷ utilisé. J'ai opté ici pour le système **SQLite** permettant de réaliser un système embarqué et donc autonome. En effet, on peut peser le pour et le contre d'un système inclus dans l'application. Ses principaux point faibles sont l'impossibilité de modifier les données après *release*, d'avoir un système complexe pour gérer les données en tant qu'administrateur. A contrario, ce système permet de créer une application indépendante et sans connexion, plus légère et rapide à l'exécution. Pour un produit destiné à des Architectes qui peuvent se retrouver à travailler dans des lieux déconnecté (In situ, en visite, sur chantier...) et qui sont souvent amenés à travailler avec de grosse contraintes de temps le choix d'un système embarqué paraissait le plus adapté. J'ai donc choisi de gérer les données avec le système SQLite, les principaux problèmes des SGBDR intégré pouvant être résolus avec de systèmes d'updates ou de patch du logiciel.

6. Un shader est un script permettant d'altérer le rendu d'un objet

7. Système de Gestion de Bases de Données Relationnelles

5.1. Architecture BDD

La base de données est construite sur le modèle de données suivant :

Materials	PoutreSquare	Poutrelpn
Id (int) Identifiant unique du matériaux	Id (int) Identifiant unique de la poutre	Id (int) Identifiant unique de la poutre
Nom (string) Nom du matériaux	Nom (string) Nom de la poutre	Nom (string) Nom de la poutre
vM (float) Masse volumique en kg/m ³	a (float) Largeur de la section	l (float) Largeur de la poutre en mm
yM (float) Module de young en GigaPascal		h (float) Hauteur de la poutre en mm
Re (float) Résistance élastique en MegaPascal		e (float) Epaisseur de l'âme en mm
		e' (float) Epaisseur de la semelle en mm

Ce modèle est principalement construit autour des données nécessaires au calculs physiques du logiciel.

5.2. Utilisation de modèles

J'utilise des **classes modèles** pour gérer les appels à la base de données. Elles sont actuellement aux nombre de deux dans l'application : le modèle des matériaux et le modèle des poutres. Chacun dispose de l'url de connexion et d'une série de méthode permettant de requêter les données dans la base. Ces classes sont donc un mélange de C# et de SQL⁸

Ci dessous, le code d'une fonction d'un modèle et son explication :

Ici on à une fonction du modèle des matériaux que récupère tous les noms de matériaux enregistrés dans la base de données. On commence par instancier la liste des résultat et ensuite on utilise un bloc using(IDbConnection) cela nous permet de créer un espace ou l'on génère une ressource qui sera automatiquement détruite à la sortie du bloc. Cela permet de gérer les connexions avec la base de données sans oublier de les refermer. On reconnaît ensuite la textit SQLQuery qui contient le texte de notre requête ici rechercher tous les noms dans la table *materials*. Ensuite on recrée un bloc using dans lequel on fait cette fois appel à un objet *IDataReader*. C'est lui qui va nous permettre de récupérer l'ensemble de nos données. On exécute un while qui fait autant d'itération que de nombre de ligne retournés et dans chaque itération on peut indexer notre reader pour accéder aux données. Le reader contient autant d'indices que de données requêtées ici, 0 est l'index de la colonne "nom" (la seule).

```

19   public List<string> GetAll()
20   {
21       List<string> result = new List<string>();
22       using ( IDbConnection dbConnection = new SQLiteConnection(_connectionString))
23       {
24           dbConnection.Open();
25           using ( IDbCommand dbCmd = dbConnection.CreateCommand())
26           {
27               string SQLQuery = "SELECT name FROM materials";
28               dbCmd.CommandText = SQLQuery;
29
30               using ( IDataReader reader = dbCmd.ExecuteReader())
31               {
32                   while (reader.Read())
33                   {
34                       Debug.Log(reader.FieldCount);
35                       result.Add(reader.GetString(0));
36                   }
37               }
38           }
39       }
40   }
41   return result;
42 }
43 }
```

5.3. Défauts et améliorations

Pour l'instant les modèles retournent seulement des liste de données. Le mélange SQLite C# a un fonctionnement faible car l'indexation des reader fait que dès le moindre changement dans l'architecture des données tout est à refaire

8. Le SQL est le Structured Query Language qui est un langage largement utilisé dans le requêtage de bases de données relationnelles.

dans les modèles.

De nombreuses améliorations pourraient être apportées à ce système je ne vais mentionner ici que les deux plus importantes.

Hydratation

Le fait que les modèles ne renvoient que des données de types de bas niveau (string, float...) est un réel défaut. En effet cela nécessite un traitement des données après la sortie du modèle. Un système plus performant serait de créer directement les objets métier dans les modèles en hydratant les données dedans. Cela permettrait une plus grande flexibilité une meilleure maintenabilité et une simplification du code. Cela n'est cependant pas possible du à l'architecture actuelle du code. En effet, les données nécessaires à la création d'une classe métier ne sont pas seulement les données récupérées par les modèles ce qui empêche ce système. Il faudrait éventuellement repenser l'architecture via un système autre que l'héritage ou avec une amélioration des constructeurs.

Modèle parent

La création d'une classe mère permettant de gérer automatiquement la récupération des données permettrait également un allègement conséquent du code. En effet, on pourrait réaliser une classe mère disposant de quelques fonctions qui permettraient de gérer les problématiques des connexions à la base de données et la récupération dynamique de données. On pourrait ensuite appeler ces fonctions dans un modèle enfant qui en hériterait.

Le système pourrait alors fonctionner de la sorte :

```
getAll<T>("param1", "param2", "param3", "nomTable")
{
    List<T> result = Parent.request(param1, param2, param3, nomTable)
}
```

L'appel à un parent pourrait permettre de créer dynamiquement la requête en fonction d'un type (GET, POST, PUT..) d'un nom de table et d'un nombre de paramètre et de formater le résultat sous la forme d'objets métier associés. Tout cela pourrait se faire en prenant avantage des types génériques

Ce système n'a pas été implémenté car il n'est pas nécessaires au fonctionnement de CiViLi dans son état actuel et également par manque de temps mais il reste une piste notable d'amélioration.

6. Communication entre Script et les arcanes d'Unity

Unity est très performant quand il s'agit de faire ce pour quoi il est construit, c'est à dire réaliser des jeux. Lorsque l'on cherche un petit peu à modifier cette utilisation courante, on vient alors vite se frotter à des problèmes épineux. En effet la plupart du temps chaque GameObject se voit attribuer son script qui réalise des actions assez locales. Dans le cadre du projet CiViLi on se retrouve dans le cas de long scripts qui centralisent de nombreuses informations pour différents objets.

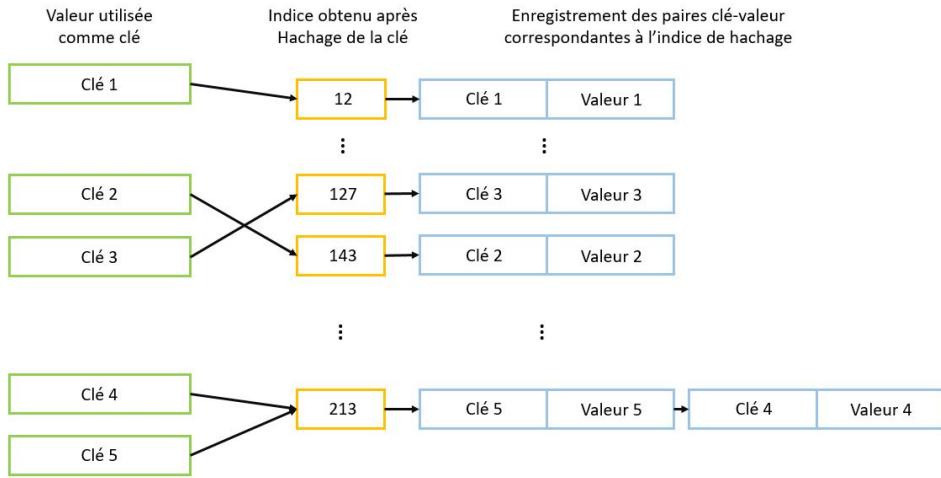
Unity à l'avantage qui est à la fois un inconvénient d'utiliser les attributs publics d'une classe pour assigner facilement des objets sans les instancier directement dans ces scripts. Cependant lorsque l'on veut faire transiter proprement des données sans se retrouver avec des ribambelles d'attributs publics dans tous les sens, il est nécessaires de trouver d'autre fonctionnement.

6.1. HashTable

Le premier objectif de communication était d'associer les formats de données back-end, c'est à dire les BuildingBloc avec les Mesh qu'il représentent. Dans un sens c'est simple, chaque BuildingBloc dispose d'une propriété *GameObject RepresentedObject get; set;* qui contient le gameObject encapsulé. Il suffit de lui donner au moment de la construction du mesh en sortie du Parser. Cependant lorsque l'utilisateur interagit avec la vue, Unity ne peut retourner qu'une référence vers les gameObject. Le lien ne marche pas dans les deux sens !

Pour pallier ce problème, j'ai eu recours à l'utilisation d'une table de hachage. Ce dispositif permet comme un dictionnaire de lier des paires clé-objet mais il s'agit souvent de clés de types complexes. Le fonctionnement est le suivant :

On ajoute un paire clé objet. La clé est hachée devient un index pour la paire. Contrairement au dictionnaire, on a un index qui référence à la fois la clé et l'objet. En utilisant comme clé nos gameObject et comme valeur nos BuildingBloc on crée un lien bidirectionnel de navigation. Cette table de hachage est un attribut statique de la classe BuildingBloc elle est remplie après le parsage du fichier et peut donc ensuite être utilisée à n'importe quel endroit dans le code.



Un autre point à mentionner est celui des collisions. EN effet, le résultat du Hachage peut amener deux objets vers le même indice de stockage. On parle alors de collision. Pour résoudre ces problèmes, différentes technique existe la plus courante étant de chaîner les valeurs dans l'espace de stockage assigné à l'indice. C'est ce que l'on voit dans le schéma ci-dessus avec les Clé 4 et 5.

Pour récupérer un élément dans la table, il suffit d'indexer notre table de hachage avec une clé. La clé est alors hachée, on trouve l'indice de stockage associé. On regarde ensuite les valeurs associées et on boucle dessus si il y en a plusieurs jusqu'à obtenir notre clé on peut donc ensuite récupérer la paire enregistrée.

6.2. custom events

Une autre méthode très simple pour faire communiquer des scripts ou des fonctions sont les évènements.

```

11  private delegate void OnSelectionChangedHandler();
12  private static event OnSelectionChangedHandler _onSelectionChanged;
13
14  private delegate void OnSelectionModifiedHandler();
15  private static event OnSelectionModifiedHandler _onSelectionModified;

```

Ici, je n'ai créé que deux simple évènements ainsi que des délégués pour gérer leur actions. Ces événement sont utilisés quand la sélection change ou quand la sélection actuelle est modifiée.

```

49  _onSelectionChanged += new OnSelectionChangedHandler(UpdateObjectPanel);
50  _onSelectionModified += new OnSelectionModifiedHandler(UpdateObjectPanel);

```

On abonne ensuite la méthode *UpdateObjectPanel* à ces évènements.

Dès lors que l'on cliquera sur un objet ou que l'on modifiera ses propriétés, ces évènements seront *raise* et on appellera notre fonction de mise à jour. Cependant ce systèmes est très réduit ici et mériterai à être étendu pour améliorer le fonctionnement et simplifier du code.

7. Contrôles utilisateurs

Unity fait par défaut hériter ses scripts de monoBehavior. Cet héritage nous permet d'avoir accès à deux événements importants. Le premier est la méthode *start* qui s'exécute comme son nom l'indique au démarrage de l'application. La deuxième, est la fonction *Update* qui est appelé toute les frames. Celle-ci est la fonction de référence lorsque l'on veut gérer les inputs utilisateurs.

7.1. touches and clics - RayTracing

Pour une fois, Unity nous rend la vie facile. En effet la gestion des contrôles utilisateurs et notamment des touches du clavier est une part prépondérante des jeux vidéos. Unity permet donc facilement d'accéder à ces informations.

```
143     void ManageInputs()
144     {
145         float transSpeed = 15f;
146         // Gestion du Scroll
147         if (Input.GetAxisRaw("Mouse ScrollWheel") > 0)
148         {
149             mainParent.transform.Translate(Vector3.forward * transSpeed * Time.deltaTime, Space.World);
150         }
151         else if (Input.GetAxisRaw("Mouse ScrollWheel") < 0)
152         {
153             mainParent.transform.Translate(Vector3.back * transSpeed * Time.deltaTime, Space.World);
154         }
155         // Gestion des touches
156         float speed = 50f;
157         if (Input.GetKey(KeyCode.Q))
158         {
159             mainParent.transform.Rotate(Vector3.up * speed * Time.deltaTime);
160         }
161         else if (Input.GetKey(KeyCode.D))
162         {
163             mainParent.transform.Rotate(-Vector3.up * speed * Time.deltaTime);
164         }
165         else if (Input.GetKey(KeyCode.Z))
166         {
167             mainParent.transform.Rotate(Vector3.right * speed * Time.deltaTime);
168         }
169         else if (Input.GetKey(KeyCode.S))
170         {
171             mainParent.transform.Rotate(-Vector3.right * speed * Time.deltaTime);
172         }
173         else if (Input.GetKey(KeyCode.LeftControl) && Input.GetKey(KeyCode.G))
174         {
175             mainParent.transform.position = new Vector3(0, 0, 0);
176         }
177         else if (Input.GetKey(KeyCode.LeftControl) && Input.GetKey(KeyCode.R))
178         {
179             mainParent.transform.rotation = Quaternion.identity;
180         }
181     }
```

Ici on retrouve le code permettant de gérer les touches et les scrolls. On peut accéder facilement au commande de l'utilisateur en utilisant : `Input.GetKeyDown(KeyCode.A)` pour le A par exemple. On accède aux scrolls en utilisant `Input.GetAxis("Mouse ScrollWheel") > 0` pour le scroll vers le haut par exemple. Ensuite on effectue les différentes actions associées aux touches. On remarque par exemple que le scroll utilise la méthode `transform.translate` mentionnée précédemment pour déplacer l'affichage et que les touches utilisent la méthode `transform.rotate` pour faire tourner la vue. Ensuite on utilise la formulation `Vector.direction * speed * Time.deltaTime` ceci nous permet plusieurs chose la première est de faire appel au propriétés statiques de la classe `Vector3` qui nous permettent de sélectionner des direction. Ensuite la vitesse ici un nombre flottant nous permet de gérer la vitesse de déplacement et enfin la gestion du temps ! `Time.deltaTime` nous permet d'effectuer l'action à un pas régulier moins rapide que le nombre de frame (qui est le rate d'appel de la fonction `Update`). En effet, notre mouvement étant assez lent il n'a pas besoin d'être recalculé ou ré-affiché exactement toutes les frames. Cela améliore les performances et diminue le coût d'exécution.

Ci dessous on retrouve la gestion des clics. Ce fonctionnement est similaire à celui des touches, il s'agit d'une cascade de if qui regarde les différents cas possible cependant la gestion est un peu différentes et un peu plus complexe.

Une fois que l'on détecte un clic, avec `Input.GetMouseButton(0)` (permet de détecter un clic gauche). La procédure est la suivante on utilise l'outil de *RayCasting* d'Unity qui permet de faire du tracé de rayon. On crée un rayon fictif qui part de la caméra et qui traverse les coordonnées de la souris. Une fois cela fait, on projette le rayon dans l'espace 3D et on regarde si l'on traverse un objet. Si c'est le cas le processus de *RayCast* nous renvoie un `hit` c'est à dire un contact on peut ensuite accéder à l'objet cliqué grâce à `hit.transform.GameObject`. On se sert ensuite de la table de hachage

sus-mentionné pour récupérer le BuildingBloc associé.

```

181     // Gestions des clics
182     if (Input.GetMouseButtonUp(0))
183     {
184         // Cast a ray to mouse coordinates
185         Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
186         if (Physics.Raycast(ray, out RaycastHit hit, 1000.0f))
187         {
188             if (hit.transform != null) // If the ray exist
189             {
190                 BuildingBloc clickedBloc = (BuildingBloc)BuildingBloc.accessTable[hit.transform.gameObject];
191                 // Manage selection
192                 if (!currentSelection.Contains(clickedBloc))
193                 {
194                     currentSelection.Remove(clickedBloc);
195                     clickedBloc.ToggleSelection();
196                 }
197                 else if (Input.GetKey(KeyCode.LeftShift))
198                 {
199                     // selection multiple
200                     currentSelection.Add(clickedBloc);
201                     clickedBloc.ToggleSelection();
202                 }
203                 else
204                 {
205                     // selection simple
206                     if (currentSelection.Count > 0)
207                     {
208                         foreach (BuildingBloc bloc in currentSelection) bloc.ToggleSelection();
209                     }
210                     currentSelection = new List<BuildingBloc>() { clickedBloc };
211                     clickedBloc.ToggleSelection();
212                 }
213                 Debug.Log(currentSelection.Count);
214                 _onSelectionChanged();
215             }
216         }
217     }

```

7.2. Affichage modal et utilisation des prefabs

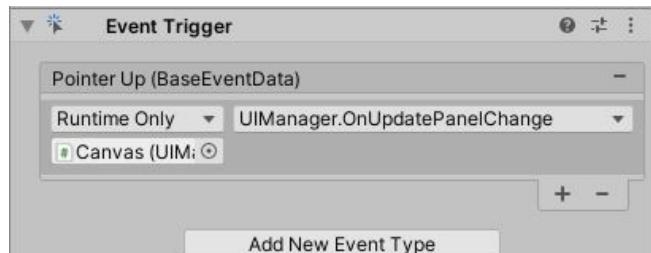
instancier des menu dynamiques

Pour permettre à l'utilisateur d'accéder à des données on passe via différents menus. Pour cela on peut utiliser un autre concept d'Unity les **prefabs**. Ils s'agit d'objet enregistrés à l'avance dans Unity et qui peuvent être chargés dynamiquement. On utilise la ligne **Instantiate(nomPrefab, Position, Rotation)**. Cela nous permet de créer dynamiquement des affichage et de leur donner des valeurs calculées par le moteur physique ou encore des valeurs récupérées dans la base de données. Cela nous permet également de créer un affichage dynamique c'est à dire avec un nombre d'objet inconnu à l'avance ce qui n'est pas possible autrement dans l'éditeur Unity. Les prefabs sont donc des objets définis à l'avance avec leur composants et leurs propriétés mais ajoutés dynamiquement à la scène.

Panneau de sélection

Pour gérer le panneau de sélection dont le fonctionnement est décrit dans la partie 3, on utilise trois **panels**, au-quel sont ajoutés des composants **EventTrigger**

Ce composant nous permet de référencer un Script et une fonction pour l'activer quand l'objet détecte un certain comportement de la part de l'utilisateur. Ici on donne à chaque bouton la même fonction *UpdateOnPanelChange* et celle-ci reçoit la référence du bouton et agit en conséquence. Ici il modifie le contenu d'une zone d'affichage le *DisplayPort* et charge des données dans la base pour les afficher à l'utilisateur.



Améliorations

Cependant ce fonctionnement n'est pas optimal. En effet le mieux serait de créer un script indépendant prenant la référence des boutons et qui serait capable de gérer un groupe de panels. Il pourrait associer dynamiquement un nombre

quelconque de boutons à leurs actions et pourrait remplir le DisplayPort grâce à des événements à part du gestionnaire d'interface principal. Cela permettrait d'alléger le code et de se rapprocher du fonctionnement classique d'Unity.

7.3. Plan de coupe et Transparence - Shader

Objet non structurels

Pour permettre à l'utilisateur de bien différencier les objets relevant de la structure principale ou non, on affiche de manière semi-translucide les objet non structurel. Déjà brièvement mentionné plus tôt, c'est le paramètre de **Transparency** que l'on passe au matériau qui est modifié. Pour altérer le matériau, on passe par un shader. Un shader est un script qui permet de modifier le rendu d'un matériau pixels par pixels. Il suffit de lui donner la propriété `isStructural` d'un objet qui vaut vraie ou faux pour activer ou non ce comportement.

Plan de Coupe

Pour permettre à l'utilisateur d'afficher l'intérieur d'un bâtiment j'ai également décidé d'ajouter un plan de coupe. Ce plan doit également servir d'input au shader pour permettre de modifier l'affichage pour n'afficher que ce qui est au dessus à gauche ou devant la plan. Cela fonctionne en créant une *Slice*, c'est à dire une coupe. On donne au shader un point d'application qui représente le centre de la coupe (`Vector3`) et une normale qui représente le sens de la coupe. Cette normale nous permet de réaliser un produit vectoriel avec l'ensemble des points/pixels du mesh. Selon le signe de ce produit on peut déterminer si le point se trouve au dessus ou en dessous du plan et donc l'afficher ou non en conséquence.

Amélioration possibles

A ce stade du projet le shader est entièrement développé et fonctionnel. Cependant créer un plan de coupe dynamique permettant à l'utilisateur d'interagir avec le modèle s'est révélé un peu plus compliqué que prévu. Notamment due aux Quaternions qui sont le système de représentation des rotations dans Unity qui s'est avéré assez complexe à appréhender. Pour que cela deviennent fonctionnel, il ne reste qu'à réussir à créer dynamiquement un plan suivant le curseur en prenant l'inclinaison de l'objet sur lequel pointe la souris.

8. Descente des charges

Une fois son plan réalisé, l'architecte l'envoie à un bureau d'étude. Son travail et de détecter l'ensemble des éléments porteurs que l'on appelle **système porteur** ou **squelette**. Dans ce système, on distingue les porteurs horizontaux et les porteurs verticaux. Le but de la descente de charges est de dimensionner la structure en trouvant les forces qui s'y appliquent. La plupart des procédés et des nomenclatures qui vont suivre sont issus de la littérature qui se base principalement sur les **EUROOCODE** qui sont des textes de normes européennes sur les structures dans le bâtiment.

8.1. Les charges

On note un certain nombres de forces qui s'appliquent aux éléments d'une structure :

Actions permanentes

Les charges permanentes ont pour symbole générale **G**. Elles représentent principalement le poids des éléments porteurs ou non. On en distingue quatre grandes catégories :

- G_1 : poids propre de la structure
- G_2 : poids des autres éléments
- G_3 : poussée des terres
- G_4 : action dues aux déformations différées

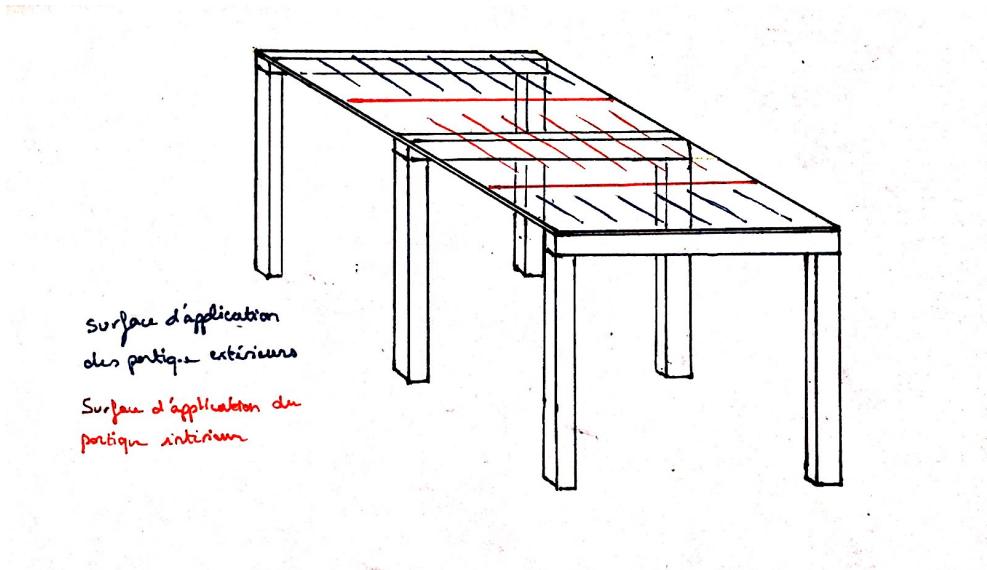
Actions Variables

Celles-ci peuvent varier au cours de la vie d'une structure. Il s'agit principalement des **charges d'exploitations** comme le poids des personnes, des meubles,... et des **charges climatiques**. Leur symbole générale est **Q**.

- Q_1 : charges d'exploitations
- Q_2 : charges climatiques (neige notée S_n ou le vent noté \mathbf{S})
- Q_3 : actions de la température (symbole T)
- Q_4 : actions passagères en cours d'exécution

Dans le cadre du projet informatique et au vu de la contrainte de temps, le projet ne prend pour l'instant en compte que les actions G_1 et G_2 . Le code est également prévu pour implémenter les charges Q_1 qui ne sont cependant pas encore intégrées faute de documentation et de données solides.

Le procédé de la descente des charges vise à faire littéralement descendre les charges verticales pour connaître les contraintes appliquées aux différents éléments d'une structure. Prenons l'exemple simple suivant :



Ici on a une couverture fine qui repose sur une série de portique. Un portique est une structure poteau-poutre. La poutre étant la partie horizontale, elle repose toujours sur deux poteaux ou plus. Les détails de leur fonctionnement sera décrit dans les parties suivantes.

Normalement, chaque élément porteur se voit attribuer une zone d'influence. En effet chacune des poutres de ce schéma ne porte pas exactement la même surface de la couche supérieure. On différencie alors des poutres situées en bordure au centre ou dans les coins d'une structure pour définir leur zone d'influence. Ici prenons la poutre centrale par exemple elle supporte la zone rouge. Soit deux fois plus que les autres portiques (zones bleues). Ensuite on peut facilement calculer la descente des charges ici. Le poids appliqué à la poutre est le poids de la couverture ainsi que son propre poids et éventuellement des charges d'exploitation. Chaque poteau qui la supporte porte donc la moitié de ce poids.

La descente des charges est en grande partie implémentée dans le logiciel. Cependant la descente de charge de CiViLi à ce stade néglige certains processus comme le calcul des zones d'influence par exemple pour des raisons de complexité d'implémentation. De même le procédé de descente des charges actuellement utilisé ne s'applique qu'à des structures classiques de bâtiment et ne sont en aucun cas valables pour des structures complexes comme notamment des structures en porte-à-faux ou des dalles suspendues. Pour l'instant, les actions des charges sont reportés sur la surface haute d'un porteur. Il ne sont donc pas pondérés par la surface de leur zone d'influence.

ELS et ELU

Dans cette section je fait aussi un point rapide sur les charges appelées **ELS** et **ELU**. Cela signifie **Etat Limite de Service** et **Etat Limite Ultime**. Une structure peut être soumise à un grand nombre d'actions qui peuvent se combiner. Quand on étudie la résistance, on doit le faire en se plaçant dans le scénario le plus défavorable. On calcule donc l'**ELS** et l'**ELU** qui sont des pondérations des charges appliquées aux bâtiments.

- Les **états limites de service** correspondent aux conditions normales d'exploitation. Il s'agit de la déformation élastique des structures, on regarde si un élément ne se déforme pas trop.

- les états limites ultimes correspondent à un critère de ruine conventionnel. C'est à dire de sollicitation maximum du matériaux. On vérifie si un élément ne se romps pas.

on à les formules suivantes :

$$ELS = G + Q$$

et

$$ELU = 1,35G_{max} + 1,5Q$$

8.2. Implémentation

Ci dessous, se trouve les différentes parties du code nécessaires à la descente de charge et leurs explication :

```

683     public void ComputeLoadButton()
684     {
685         mainParent.transform.localRotation = Quaternion.identity;
686
687         // Set layered structure
688         foreach (BuildingBloc obj1 in BuildingBloc.accessTable.Values)
689         {
690             obj1.RepresentedObject.GetComponent<MeshFilter>().mesh.RecalculateBounds();
691             foreach (BuildingBloc obj2 in BuildingBloc.accessTable.Values)
692             {
693                 obj2.RepresentedObject.GetComponent<MeshFilter>().mesh.RecalculateBounds();
694                 if (obj1 != obj2)
695                 {
696                     if (IsOn(obj1.RepresentedObject, obj2.RepresentedObject))
697                     {
698                         Debug.Log(obj1.RepresentedObject.name + " is on " + obj2.RepresentedObject.name);
699                         obj1.On.Add(obj2);
700                         obj2.Under.Add(obj1);
701                     }
702                 }
703             }
704         }
705         // Compute Lifted Weight
706         List<BuildingBloc> orderedObjects = BuildingBloc.accessTable.Values.Cast<BuildingBloc>().ToList().OrderByDescending(o => o.RepresentedObject.GetComponent<Collider>().bounds.max.y).ToList();
707         foreach (BuildingBloc data in orderedObjects)
708         {
709             data.ComputeLiftedWeight();
710         }
711     }

```

La première étape pour appliquer la descente de charges est de trouver quel objet repose sur quel autre. On créer donc une double boucle qui parcours tous les objets et regarde comparativement tous les autres. Si on à deux objets différents alors on s'intéresse à savoir si ils sont l'un sur l'autre. C'est l'appel à la fonction **IsOn**

```

635     #region WeightDescent
636     // return true if obj1 is on obj2 false otherwise
637     bool IsOn(GameObject obj1, GameObject obj2)
638     {
639         Bounds obj1Bounds = obj1.GetComponent<Collider>().bounds;
640         Bounds obj2Bounds = obj2.GetComponent<Collider>().bounds;
641
642         Vector3 minObj1 = obj1Bounds.min;
643         Vector3 maxObj1 = obj1Bounds.max;
644         Vector3 minObj2 = obj2Bounds.min;
645         Vector3 maxObj2 = obj2Bounds.max;
646
647         if (Math.Round(minObj1.y, 1) == Math.Round(maxObj2.y, 1))
648         {
649             //Debug.Log("Contact point");
650             if (IsPlaneOverlap(minObj1, minObj2, maxObj1, maxObj2))
651             {
652                 //Debug.Log("Plane overlap");
653                 return true;
654             }
655         }
656         return false;
657     }

```

Unity étant un moteur de jeux, il dispose de certains outils de physique simple aux-quel je me suis intéressé le plus prometteur pour notre problème étant les **BoundingBox** en jeu vidéo on parle des **AABB** ou Axis-Aligned Bounding Box par opposition aux **OB** les Oriented Bounding Box. Une bounding box est une encapsulation de notre Mesh dans une forme simple par exemple un parallélépipède. Elle permet souvent de gérer les collision. Une bounding box permet de détecter une collision si deux objets ont des intersection sur les trois axes du plan. Cependant dans notre cas, les objets ne se recoupe pas en effet une poutre est sur un poteau et non pas dedans. Ces boites ne sont donc pas suffisante.

J'ai pu utiliser à mon avantages les propriétés **obj.GetComponent<Collider>().bounds.min** et **obj.GetComponent<Collider>().bounds.max** qui m'ont permis de récupérer simplement le point le plus haut et le plus bas des bouding box des objets. Cela permet de facilement savoir en comparant le haut d'un objet et le bas d'un autre si ces deux se trouve sur le même plan en y. C'est à dire si ils sont à des hauteur cohérente avec le fait d'être l'un sur l'autre.

Cependant, la hauteur d'un objet par rapport à un autre n'est pas la seule condition pour qu'il se trouve physiquement au dessus. Il faut aussi qu'il soit aligné c'est à dire qu'il y ai un recouvrement. Ici le recouvrement qui nous intéresse est sur la plan X Z. Pour cela, il faut qu'il y ai un recouvrement à la fois sur l'axe X ET sur l'axe Z. Pour dire qu'on à un recouvrement sur un axe, il suffit de regarder si la coordonnées la "plus à gauche" d'un objet sur cet axe est plus petite que la coordonnée la plus à droite d'un autre objet sur ce même axe et vice versa.

```

659     bool IsPlaneOverlap(Vector3 minObj1, Vector3 minObj2, Vector3 maxObj1, Vector3 maxObj2)
660     {
661         // mins for object one on all axis
662         float minX1 = minObj1.x;
663         float minZ1 = minObj1.z;
664         // mins for object two on all axis
665         float minX2 = minObj2.x;
666         float minZ2 = minObj2.z;
667         // maxs for object one on all axis
668         float maxX1 = maxObj1.x;
669         float maxZ1 = maxObj1.z;
670         // maxs for object two on all axis
671         float maxX2 = maxObj2.x;
672         float maxZ2 = maxObj2.z;
673
674         if (IsAxisOverlap(minX1, minX2, maxX1, maxX2)
675             && IsAxisOverlap(minZ1, minZ2, maxZ1, maxZ2))
676             ) return true; // X & z
677         return false;
678     }
679
680     bool IsAxisOverlap(float min1, float min2, float max1, float max2)
681     {
682         if (max1 >= min2 || min1 <= max2) return true;
683         else return false;
684     }

```

Une fois que l'on réuni ces deux critère alors on sait qu'un objet supporte l'autre. On ajoute donc l'un dans la liste **On** et l'autre dans la liste **Under**. Cela nous permet de connaître la **Chaîne structurelle**. Dès lors, il ne nous reste plus qu'à calculer le poids porté de chaque objet. On part de l'objet les plus hauts sur l'axe Y. C'est à dire les objets qui ne porte personne et on descend ensuite à partir de la. Chaque objet porte alors son propre poids et un poids additionnel qui équivaut au poids porté des éléments sur lui que divise le nombre d'appuis de ces éléments. Par exemple prenons une chape de béton sur trois poutre chaque poutre son poids plus un tiers du poids porté de la chape (car celle ci repose sur trois appuis son poids est donc répartis.).

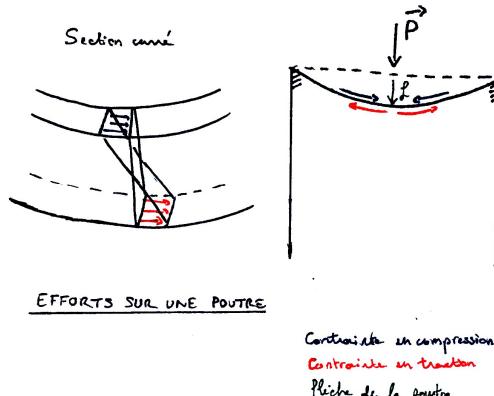
Une fois que l'on connaît le poids porté de chaque élément, on peut passer aux modèles physiques permettant de vérifier leurs résistance sous ce poids.

9. Modèle poutres

9.1. Introduction

Une poutre représente un objet structurel qui repose sur des appuis verticaux et se trouve dans le plan horizontal. Elle est donc soumise à des efforts de flexion et de compression

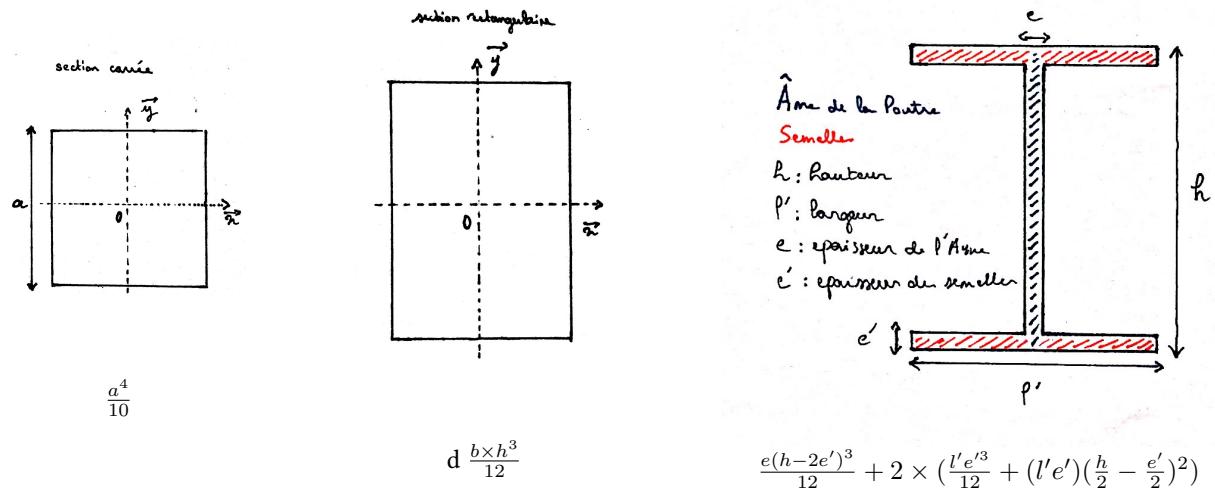
Les deux grandeurs principales qui nous intéressent sont la flèche f_{max} qui est la déformation vers le bas d'une poutre soumises à un force. Ainsi que la contrainte en traction σ_{max} qui s'exerce dans la poutre. En effet, lorsque une poutre fléchie vers le bas, alors il s'exerce une traction et une compression dans le sens des fibres (le sens orthogonal au poids). On ne s'intéresse qu'à la contrainte de traction car elle est toujours plus critique que celle en compression.



9.2. Calculs nécessaires

Moment quadratique

Le **moment quadratique** est un grandeur qui est relative à la **section** d'une poutre. C'est à dire que sa formule varie en fonction de la section. La section est le profil de coupe d'une poutre orthogonalement à sa ligne médiane. Le moment quadratique est représenté par la lettre I et s'exprime en m^4 . Ci dessous les formules de moment pour quelques type de section courantes :



Moment de flexion ou moment fléchissant

Le **Moment en flexion** ou plus souvent appelé **Moment fléchissant** en architecture M_f représente la force de flexion appliquée à la poutre. C'est la résultante créée par l'application du poids. Il s'exprime par la formule :

$$M_f = \frac{p \times L^2}{8}$$

avec L la longueur de la poutre en m et p la **charge linéique** appliquée à la poutre.

La charge linéique exprimée en N/m est un représentation de la charge appliquée sur un mètre de poutre. Or jusqu'à présent nous ne possédions que la charge portée globale en N . Il nous faut donc la convertir pour assurer l'homogénéité de l'équation. La charge linéique vaut la charge surfacique que multiplie la largeur de la poutre : $p = P_s * l$. En effet la charge qui nous intéresse doit se répartir dans la longueur qui est la dimension de la poutre avec la plus grande contrainte.

La charge surfacique est l'expression de l'application d'un poids sur une surface. Il se mesure en N/m^2 et on le calcule en divisant le poids par sa surface d'application. C'est la fameuse surface d'influence de la poutre. Comme mentionnée précédemment, on néglige cette donnée et on considère comme surface d'application l'aire haute de la poutre : $A = l \times L$. La charge surfacique sur une poutre est donc obtenue par le calcul : $P_s = \frac{P}{A}$

Retournons au moment, on se retrouve avec :

$$M_f = \frac{p \times L^2}{8} \quad M_f = \frac{P_s * l \times L^2}{8} \quad M_f = \frac{\frac{P}{A} * l \times L^2}{8} \quad M_f = \frac{\frac{P}{L} * l \times L^2}{8} \quad M_f = \frac{P \times L^2}{8}$$

on obtient donc la formule simplifiée et homogène du moment quadratique :

$$M_f = \frac{P \times L}{8}$$

avec P en *Newton* et L la longueur de la poutre en *mètres* on à bien M_f en **Nm**.

Contrainte en traction

Une fois que l'on obtient ce moment, on peut en déduire la **contrainte en traction** σ qui en résulte dans les fibres de la poutres. Sa formule générale est :

$$\sigma = \frac{M_f}{I} \times V$$

ou V est l'ordonnée du point où la traction est la plus forte. Dans le cas simplifié du portique (une poutre sur deux appuis) on peut donc estimer ce point. Il s'agit du point le plus bas de la poutre (l'endroit où la déformation en flexion

est la plus importante). On regard l'écart de ce point par rapport à la ligne moyenne de la poutre⁹. Dans un poutre de hauteur H la distance entre le point le plus bas et la corde est donc $\frac{H}{2}$

On peut donc exprimer la formule de la contrainte max simplifiée :

$$\sigma_{max} = \frac{M_f}{I} \times \frac{H}{2}$$

avec le moment fléchissant en **Nm**, le moment quadratique en m^4 et H la hauteur de la poutre en m , on obtient donc

$$\frac{Nm}{m^4} \times \frac{m}{\emptyset} = Nm^{-2}$$

Flèche

La flèche est la mesure de l'écart entre la ligne moyenne au repos et sous contrainte. Elle est simple à calculer grâce à la formule suivante :

$$f = \frac{5 \times p \times L^4}{384 \times E \times I}$$

ou p est la charge linéique en **N/m**, L la longueur de la poutre en **m**, E le module de Young du matériaux considéré en **Gpa** (Giga Pascal), et I le moment quadratique en m^4 on doit donc ramener le module de Young en N/m^2 pour obtenir un résultat de la flèche cohérent en **mètres**. Or on sait que $1 \text{ Pascal} = 1 \text{ N/m}^2$ et on a un rapport d'ordre 10^9 . On obtient donc :

$$m = \frac{\emptyset \times N/m \times m^4}{\emptyset \times N/m^2 \times m^4}$$

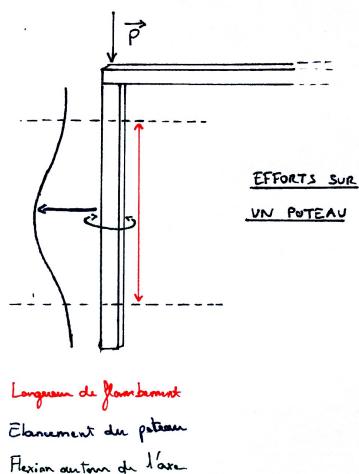
9.3. Retour aux valeurs limites et vérifications

Une fois ces quelques calculs compris, on peu simplement les appliquer pour obtenir : σ_{ELS} et σ_{ELU} . Ces deux valeurs se calcurent en employant la formule de σ_{max} et en remplaçant la valeur de P dans la formule du moment quadratique par nos charge pondérées à l'ELS ou à l'ELU. Une fois que l'on obtient la contrainte en traction, on peut la comparer à R_e la valeur de la résistance élastique du matériau. Cette valeur est enregistrée dans la base de données en **MPa** (Mega Pascal). Après des conversions on peut donc les comparer. Si σ_{max} à l'ELS ou à l'ELU est inférieure à la limite élastique, alors on respecte les critères d'usage et de ruine.

10. Modèle poteau

10.1. Introduction

Un poteau par opposition à une poutre représente un objet structurel dans le plan vertical. Les contraintes qui s'y applique sont toujours identique. On s'intéresse principalement à la compression des éléments cependant, la résultante de ses actions est différentes. En effet, les déformations ne s'effectue plus colinéairement au poids mais à sa perpendiculaire. Il s'agit principalement des efforts de flambement et d'élanement qui limite la charge possible en tête du poteau.



9. La ligne moyenne est l'axe qui permet de "résumer la poutre" il s'agit souvent du point central de la section

10.2. calculs nécessaires

Longueur de flambement

Lorsque l'on applique une charge sur un poteau, il s'exerce sur ce poteau des forces partant du centre vers l'extérieur le poussant à se déformer. Dans une trop grande mesure ces déformations peuvent aller jusqu'à rompre le poteau entraînant des évènements graves pour la structure (créés par les contraintes de cisaillement dues aux vents, le flambement est l'une des principales causes des effondrements de ponts par exemple).

Ces efforts apportent une déformation sur une partie ou sur toute la longueur de notre poteau. On peut calculer cette **longueur de flambement** grâce à un **coefficients de flambement**. Ce dernier dépend des types d'attaches du poteau (encastrement, rotule, libre...) dans le cadre du projet CiViLi, on simplifie les calculs en supposant l'ensemble des attaches de poteau (et de poutre) comme des doubles encastrement. Cela nous donne donc :

$$L_{cr} = 0.5 \times L$$

Où L_{cr} est la longueur du flambement.

Rayon de giration

Les fortes contraintes n'apportent pas que des déformations axiales comme le flambement, mais peuvent aussi amener à des déformations radiales. On calcule donc également le rayon de giration qui traduit le mouvement de la section autour de son centre de gravité.

$$i_z = \sqrt{\frac{I}{A}}$$

avec I le **moment quadratique de la section** et A l'**aire de la section**.

Cette valeur doit être calculée afin de pouvoir mesurer l'élancement des poteaux.

Coefficient d'élancement

L'élancement est la rapport de distance entre le milieu de la section au repos et le milieu de la section sous contrainte. L'élancement est le plus souvent critique au milieu de la zone de flambement. Pour connaître la charge maximale que l'on peut mettre sur notre structure, il faut mesurer l'élancement réduit. Ce dernier s'obtient grâce à l'élancement et à l'élancement critique.

La formule de l'élancement est la suivante :

$$\lambda = \frac{L_{cr}}{i_z}$$

avec i_z le rayon de giration et L_{cr} en mètre l'élancement est sans unité.

Élancement critique - Formule d'Euler

L'élancement critique λ_{crit} se mesure grâce à la formule d'Euler. Elle traduit la résistance maximale de la poutre. Cependant, elle se base sur une hypothèse d'homogénéité parfaite du matériaux. Les imperfections courantes dans les matériaux de construction (noeuds dans le bois, bulles dans les aciers...) amène une sous-estimation de la ruine des éléments, d'où la nécessité d'utiliser l'élancement réduit.

La formule d'Euler est la suivante :

$$\lambda_{crit} = \sqrt{\frac{\Pi^2 \times E}{\sigma_e}}$$

où E est le module de Young du matériaux utilisé et σ_e sa limite élastique. L'élancement critique est donc également une grandeur sans unité.

Élancement relatif ou élancement réduit

On définit l'**élancement réduit** comme le ratio de l'élancement et de l'élancement critique. On a $\lambda_{red} = \frac{\lambda}{\lambda_{crit}}$. Ici on a une estimation de l'élancement limite réel du poteau. Cependant comme mentionné, il s'agit d'une estimation et on doit donc chercher un *coefficient de sécurité* pour assurer les calculs de dimensionnement.

Coefficient de réduction

Le **coefficent de réduction** χ est le facteur de flambage. Il est issu de simulation sur des échantillons de matériaux. En temps normal, ils se lit sur des courbes d'abaques ou dans des tables en fonction de la valeur de l'élancement critique. Pour nous ils est impossible d'utiliser ce mécanisme. J'ai donc du rechercher une méthode numérique pour le calculer.

Après quelque recherche, on peut trouver que χ s'exprime :

$$\chi = \frac{1}{\Phi + \sqrt{\Phi^2 - \lambda_{red}^2}}$$

ici Φ est dépendant du matériaux utilisé :

$$\Phi = \frac{1 + \alpha(\lambda_{red} - 0.2) + \lambda_{red}^2}{2}$$

On retrouve λ_{red} notre élancement réduit dans les deux formules et α qui est uniquement dépendant du type de matériaux et de l'objet utilisé. Cette valeur α est associée avec les courbes d'abaques précédemment mentionnées. Il existe un coefficient par courbe. Il suffit ensuite de trouver dans des tableaux quelle courbe est associé avec le matériaux et le type de structure que l'on utilise. Pour l'instant le logiciel CiViLi utilise la plus petit valeur : $\alpha = 0.13$. Ce facteur d'imperfection dépend donc uniquement de la courbe de flambement et du type de structure.

Charges maximale en tête

On obtient ensuite l'expression de la **charge maximale en tête** de poteau notée Q_{max} qui se calcule simplement :

$$Q_{max} = \chi \times A \times \sigma_e$$

avec A l'aire de la section et σ_e la limite élastique du matériaux utilisé. Cette charge utilise χ le **coefficent de réduction** elle est donc bien dépendante à la fois des risque de flambement et de l'élancement du poteau ainsi que du rayon de giration de la section.

10.3. Retour aux valeurs limites et vérifications

Une fois ces calculs effectués, ils est donc plus simple que pour une poutre de vérifier les contraintes. Il suffit de vérifier que le poids (G) calculée lors de la descente des charge et majoré (ELS ou ELU) n'est pas supérieur au poids maximal acceptable en tête du poteau.

11. Conclusion technique

L'ensemble des deux modèles présenté ci-dessus à été implémenté dans le logiciel CiViLi. Le plus complexe dans ce type de projet n'est pas forcément l'implémentation finale, mais bien la collecte de donnée et la compréhension des modèles physiques. Le réel challenge ici est de créer une architecture permettant de concilier les fonctionnement d'Unity avec les besoins stricte d'une application métier de calcul. Dans le futur le projet pourrait avoir à gagner à s'extraire du moteur Unity et utiliser un moteur custom réalisé à l'aide de technologie de plus bas niveau tel que le pont SharpGL donnant une interface OpenGL en C#. Cela permettrait un meilleur contrôle du code et de ses fonctionnalités, ainsi qu'une architecture plus propre. Cependant ce type d'approche nécessiterai la création d'un moteur de rendu 3D ainsi que le développement d'un système d'interface qui bien que passionnant n'est pas envisageable sur un projet d'une durée aussi limitée.



A large, modern building with a curved facade featuring glass panels and wooden slats. The building has multiple levels with balconies and a glass-enclosed entrance area.

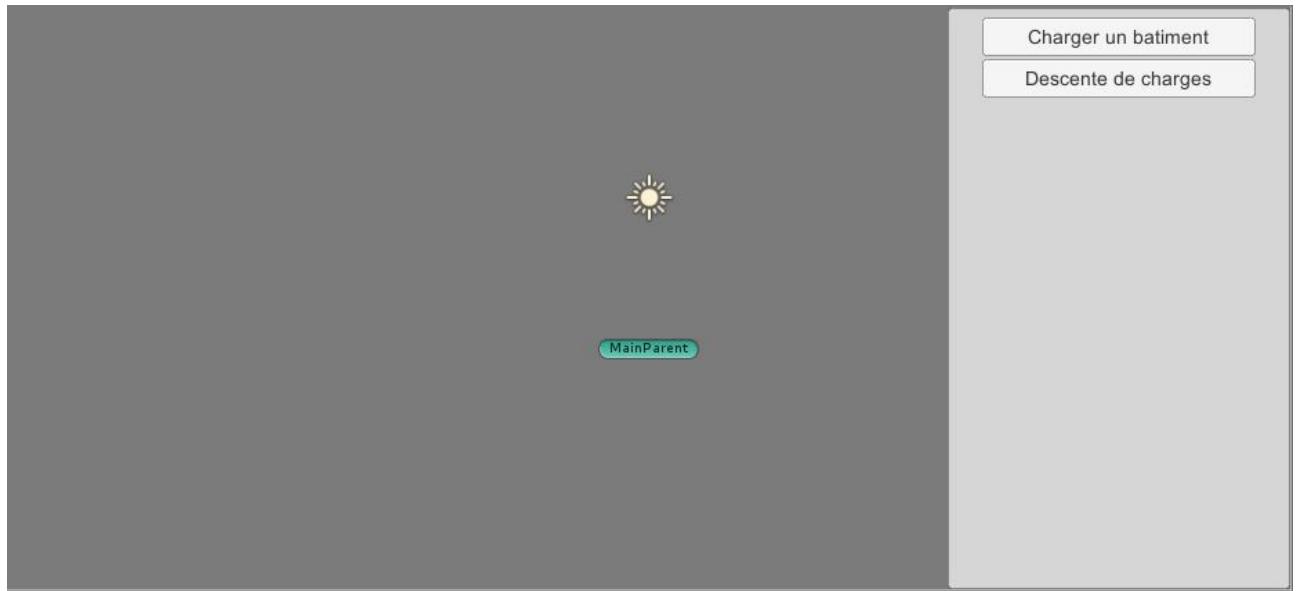
Troisième partie

UTILISATION DU LOGICIEL

1. Vue principale et architecture

Cette partie a pour objectif de détailler le coté "front" du logiciel et d'en expliquer brièvement les spécificités et le fonctionnement. J'y aborderais également ses principaux défauts et de possibles améliorations.

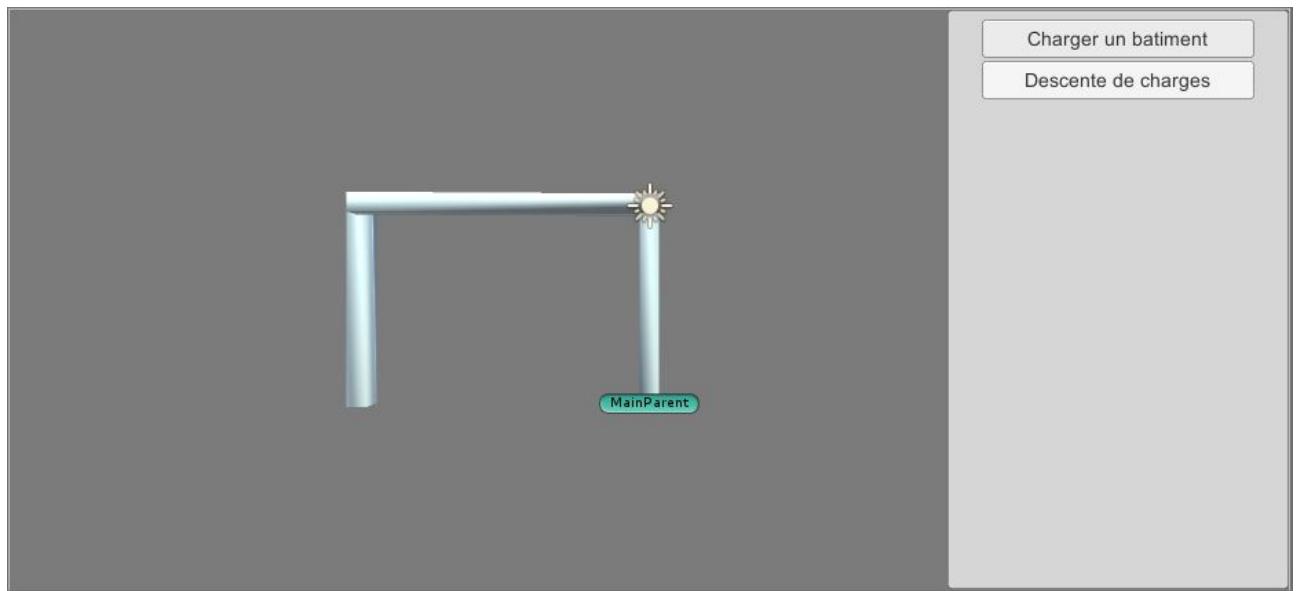
Lorsque vous ouvrez le logiciel CiViLi, vous vous retrouvez face à cette interface.



Elle se constitue de deux zones principales. Sur la gauche c'est le ViewPort. C'est là que s'affiche le contenu 3D. A droite se trouve le panneau de contrôle qui permettra de réaliser la plupart des actions possibles dans l'application et d'étudier la structure. Pour l'instant on y retrouve deux boutons. Le premier pour charger un bâtiment le second pour effectuer une descente de charges.

1.1. Chargement d'un asset

Lorsque l'on clic sur le bouton **charger un bâtiment**, on se retrouve avec l'affichage suivant :



On rencontre alors le premier défaut du logiciel. En effet pour l'instant il ne permet de charger qu'un seul fichier de test dont l'adresse est inscrite en dur dans le code. Cela vient du fait que malgré de nombreuse recherche je n'ai pas

été en mesure de créer un explorateur de fichier depuis Unity afin de permettre à l'utilisateur de choisir son dossier. Ce problème vient probablement du fait que Unity étant un système exportable sur de nombreuse plate-forme et notamment core Windows et Unix mais aussi téléphone et autre... il ne met pas à disposition d'interface de communication avec le système hôte.

Je n'ai malheureusement pas encore trouvé de solution à ce problème.

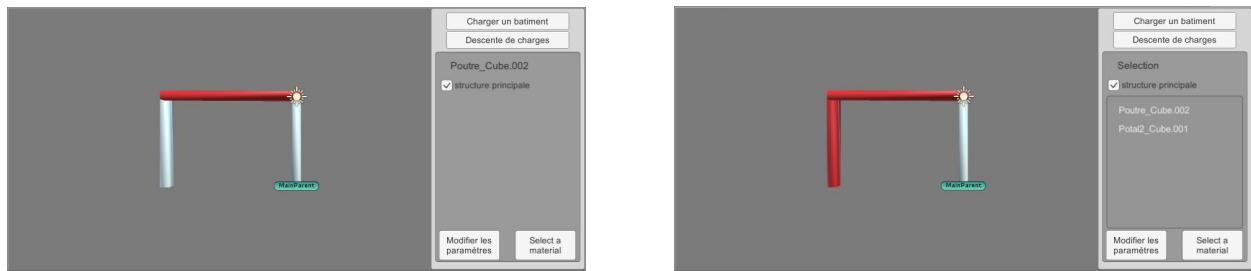
1.2. boutons & contrôles

Une fois qu'un objet est chargé cependant, on peut commencer à utiliser pleinement le logiciel. L'utilisation principale se fait à travers l'interface mais il est bon de noter l'ajout des contrôles claviers qui permettent de gérer notamment le coté affichage. En effet, les touches Z,Q,S et D sont utilisées pour modifier la rotation du Mesh. On peut également utiliser le scroll pour avancer ou reculer la vue (zoom).

Les commandes Ctrl + R et Ctrl + G ont également été ajoutées, elles permettent respectivement de remettre à zéro la rotation de l'objet et sa translation. Si l'on perd de vu l'objet on peut donc facilement le remettre dans son état initial.

1.3. sélection

En utilisant la souris, il est possible de sélectionner une partie du bâtiment. Dès qu'un objet est sélectionné, son matériau change pour permettre à l'utilisateur de le repérer. On voit également apparaître sur la droite le panneau d'objet qui permet d'accéder aux paramètres de l'objet et de le modifier. Cependant, sélectionner un à un les éléments est suffisant pour un Mesh de test qui ne comprend que trois objets. Mais dès lors que la complexité de l'objet augmente, il devient très vite lassant de devoir modifier un à un les objets. J'ai donc également implémenté une sélection et une modification multiple des éléments. Il suffit de maintenir Shift en cliquant pour sélectionner une série d'objets.

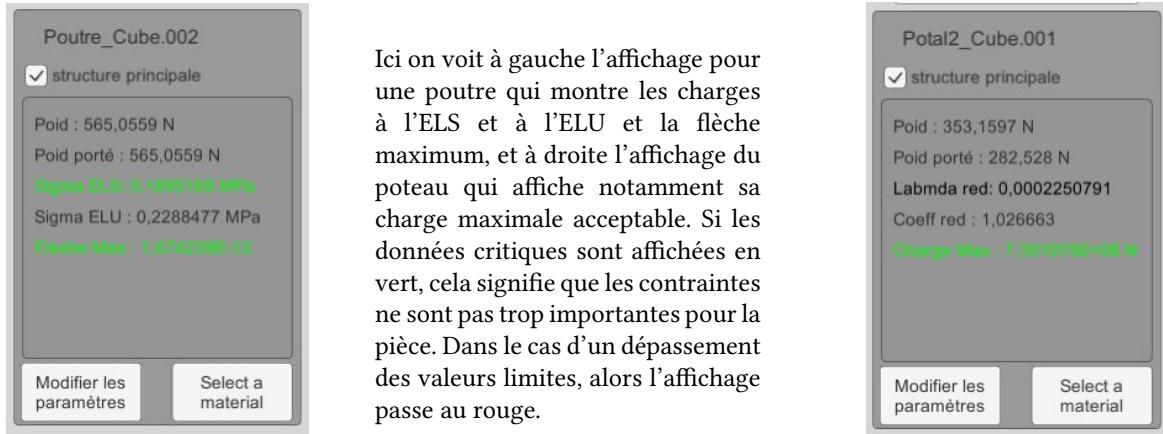


2. objectPanel

Le panneau de description des objets, qui se trouve sur la droite de l'écran est le principal élément qui permet de connaître les détails sur les objets. Si on a une sélection unique, alors le panneau affiche en titre le nom de l'objet sélectionné. Si la sélection est multiple, alors on affiche la liste des objets qui seront affectés par d'éventuelles modifications.

2.1. Affichage modal

Pour ce qui est de l'affichage des données, il est dynamique. Pour une sélection dont le type n'est pas référencé, alors on n'affiche que son nom. Sinon, on affiche en fonction de l'objet représenté. Selon si on est face à une poutre ou à un poteau, les données utiles sont différentes. Le code sélectionne donc l'affichage en conséquence.

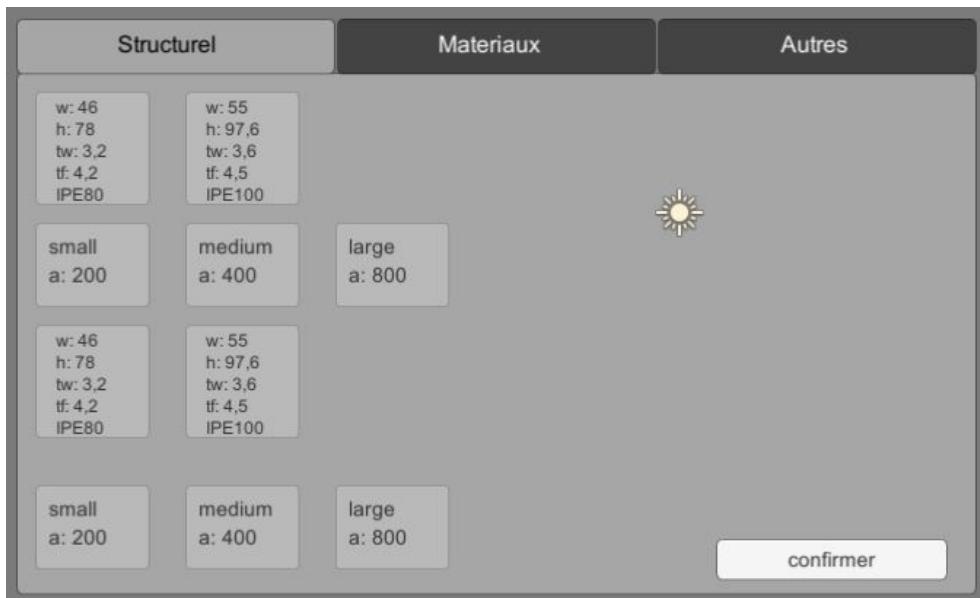


2.2. Mise à jour possible des objets

Une fois que l'utilisateur a fait une sélection simple ou multiple, il peut cliquer en bas à droite de l'écran pour afficher le panneau de mise-à-jour qui permet de modifier les paramètres de sa sélection. Dans ce nouvel affichage, il pourra alors modifier ou le type d'objet, ou le matériau assigné à sa sélection. De plus il peut à tout moment re-cliquer sur un objet sélectionné pour le dé-sélectionner.

3. Update panel

Le panneau de mise-à-jour est présenté ci-dessous :



Lorsqu'il s'affiche, il remplace la vue centrale et prend la place du viewPort masquant la vue de l'objet 3D pendant la sélection. Bien que laissant plus de place à l'affichage, cela empêche l'utilisateur de voir en temps réel les modifications apportées à son objet. Le format de ce menu pourrait donc être revisité pour une solution plus *user friendly*.

3.1. Gestion par onglets et chargements dynamiques

Lorsque l'on se retrouve dans ce menu, on se retrouve face à trois onglets. Le premier permet d'assigner un type d'objet spécifique à la structure comme le montre la capture d'écran ci dessus. Le menu manque encore pour l'instant cruellement d'explications. Mais les données étant chargées dynamiquement dans la base de données au moment de

l'ouverture du menu et les boutons permettant d'afficher ces données étant créées en cours d'exécution, cela rend très compliqué toute tentative de design.

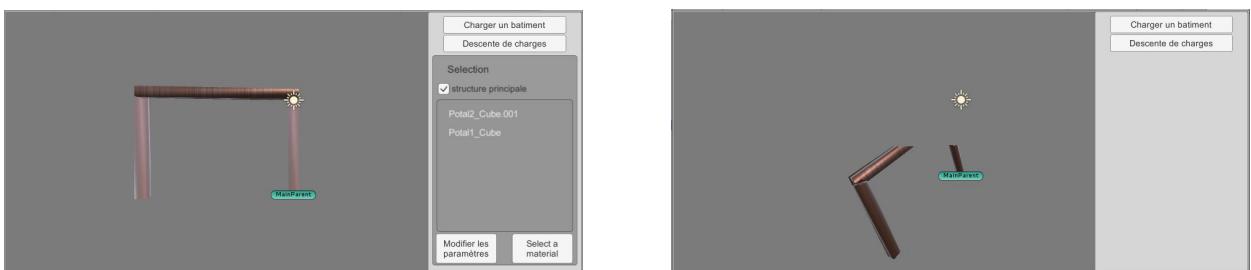
Pour plus d'explication, on retrouve ici quatre lignes. La première représente les poutres de types IPE et la seconde représente des poutres à section carrée. Les deux du dessous présente les même données mais pour des poteaux. On se trouve la encore face à une limitation du système actuel. En effet, les élément utilisé en poutre ou en poteau peuvent être similaire mais comme le système ne sait pas encore reconnaître seul lequel est lequel, on est dans l'obligation de dédoubler notre affichage pour gérer séparément la création d'un poteau ou d'une poutre. une solution serait la gestion dynamique des différents objets utilisables dans le logiciel.



Sur cette seconde image, on voit le menu matériaux qui permet d'assigner les matériaux aux objets. Pour l'instant on ne dispose que d'un type de bois et d'un type d'acier. Mais ces matériaux nécessiterait à un seul des menus complets car ils viennent sous de nombreuses compositions différents (arbres différents, alliages divers...) cependant pour l'instant seul ces deux matériaux précis sont utilisés. Ici on voit les deux "boutons" qui permettent de les utiliser. J'affiche dessus une version modifier de l'image de texture utilisé dans le matériaux. Cela permet de ne pas seulement nommer le matériau dans le but d'aider l'utilisateur dans son choix.

L'onglet autres est pour l'instant vide. Il est prévu pour accueillir les données relatives aux éléments non structurels comme par exemple les planchers, les couvertures et les systèmes isolants par exemple qui ne prennent pas part à la portance de la structure mais on définitivement un impact sur son poids. Cela n'est cependant possible que par l'ajout d'une classe métier pour gérer les actions de ce type d'élément. Bien évidemment, à chaque élément ajouté il nous faut également renseigner dans la base de données ses propriétés et donc lui créer une table et un modèle pour utiliser ces données. L'ajout de n'importe quel élément est donc coûteux en temps et en ressources. C'est pourquoi à ce stade aucun élément non essentiel à la création de portiques n'a été ajouté.

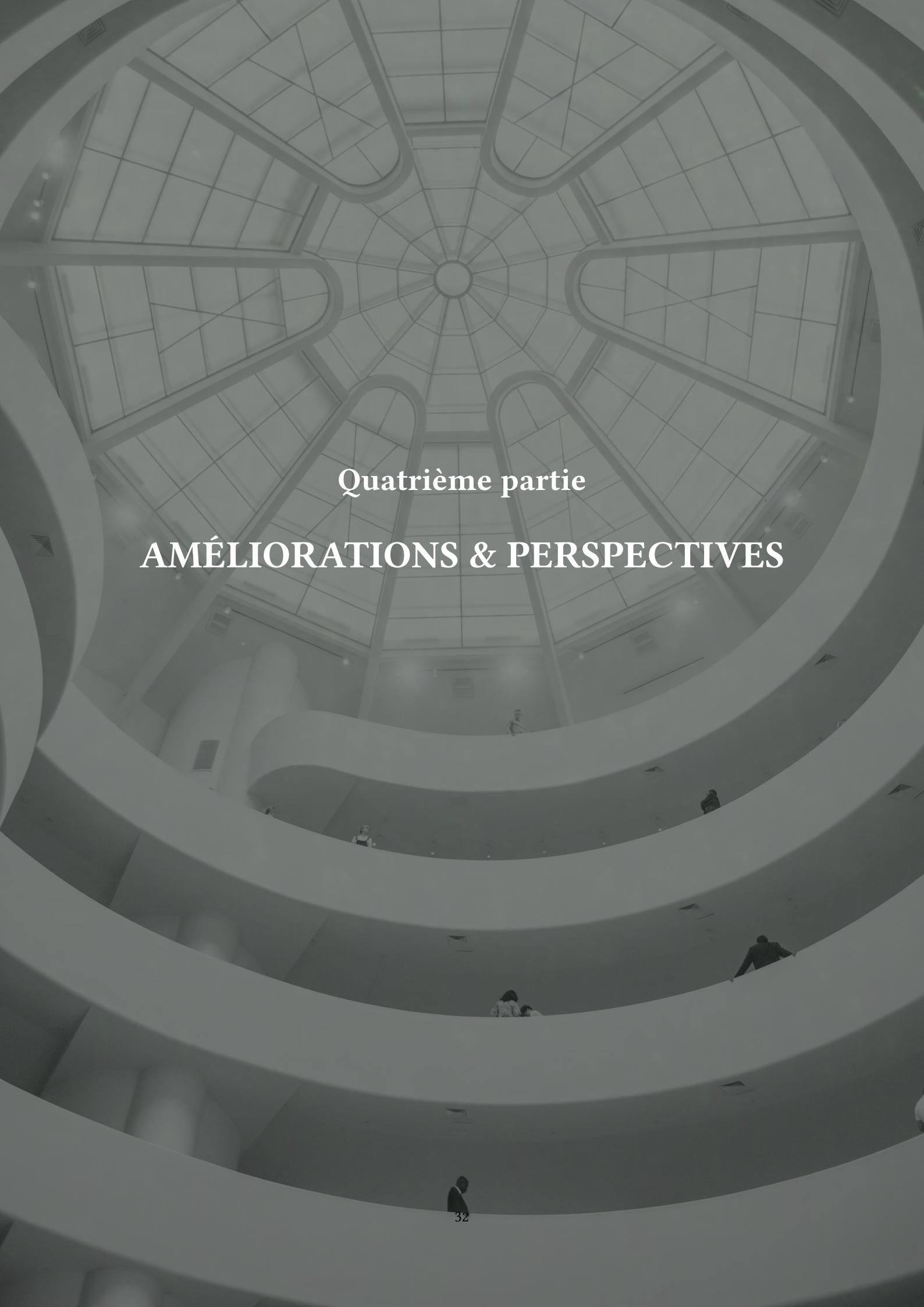
autre gestion des objets



Ici, on peut aussi voir à gauche l'ajout de la transparence à un élément pour signifier qu'il ne fait pas partie de la structure principale, et à droite l'effet du plan de coupe qui n'est malheureusement pas réalisable par l'utilisateur en l'état. Les données du Shader ne pouvant être passé à ce dernier que via l'éditeur d'Unity.

4. Conclusion

Pour l'instant, l'interface est suffisante pour faire les tests sommaires nécessaires à la *proof of concept* néanmoins, elle nécessite un travail d'améliorations à part entière afin de la rendre digne d'une école de cognitique. Cependant comme énoncé précédemment l'ajout de chaque élément est coûteux en recherches et en temps. De plus les améliorations de l'interface vont de pairs avec une augmentation proportionnelle des capacités du code qui la supporte.



Quatrième partie

AMÉLIORATIONS & PERSPECTIVES

1. Retour sur le projet

Développer ce projet dans le cadre du projet informatique individuel m'a permis de progresser sur différents points. En programmation d'abord, ou il m'a permis de découvrir de nouveaux concepts ainsi que le nouvel environnement de développement Unity. Sur le monde de l'architecture dont j'ai pu découvrir la phase intéressante est indispensable du dimensionnement. Cela m'a également permis d'avancer dans ma vision du travail de l'ingénieur cogniticien. C'est lorsque l'on travail seul que l'on se rend compte du réel atout qu'est une équipe. En tant qu'ingénieurs cogniticiens, nous devons pouvoir nous approprier un domaine et le comprendre assez pour proposer des idées innovantes et gérer la conception de produits ou de service développés autour de l'idée d'aider des utilisateurs. C'est ce que j'ai essayé de faire en créant le projet CiViLi qui avec un investissement plus important et en ajoutant dans la conception la démarche *centrée utilisateur*, peut je pense avoir une réelle plus-value dans le domaine de l'architecture. L'objectif dans le futur serait d'étendre avec l'aide de professionnels du domaine cette solution afin de créer un logiciel à la fois puissant, précis et simple d'utilisation.

2. Améliorations envisagées et pistes de recherches

Bien que non optimal, le résultat montre qu'il est possible de réaliser un logiciel d'aide au dimensionnement. Désormais, l'objectif est de continuer à l'étendre. Pour ça de nombreuses pistes s'offrent à moi.

L'un des premiers objectifs, serait d'ajouter la gestion des classes de bâtiments. Dans mes recherches, j'ai trouvé qu'il existait des catégories de bâtiments (habitations, bureau, stockage...) permettant d'utiliser des approximations fortes sur les charges d'exploitations. Ces dernières jouant un rôle important dans le poids des objets notamment des sols et surfaces vivantes il serait très intéressant de pouvoir ajouter cette gestion des catégories de bâtiments.

En suivant, l'ajout d'un plus grands nombres de données dans la base pour les différents types de poteaux et de poutres ainsi que de matériaux serait nécessaire pour commencer à réaliser des agencements structurels plus complexes. De plus la prise en compte d'un plus grand nombre de sections et les recherches sur leurs moment quadratique et leurs calcul de volume et d'aire serait intéressant pour prendre en compte notamment des poutres à structure asymétriques tels que des HEA ou autres.

La recherche sur les systèmes d'explorateurs de fichier semble également indispensable à la réalisation d'un prototype pleinement fonctionnel en *stand-alone*. De plus l'utilisation améliorée de la base de données pourrait permettre de gérer des fonctionnements tels que l'enregistrement de projet et la gestion d'une interface personnalisable pour les différents utilisateurs qui est un grand dénominateur commun de beaucoup de logiciels de modélisation et permettrait d'inscrire CiViLi dans la continuité des outils de travail des architectes.

La reconnaissance et l'ajout des matériaux de manière dynamique lors du chargement de l'objet serait également un grand plus pour le prototype. Cela permettrait à la fois une utilisation plus simple et moins rébarbative et également une amélioration des performances. Cependant, ceci nécessite de se plonger plus en avant dans la compréhension des fichiers *.mlt* ce que je n'ai pour l'instant fait qu'en surface.

De nombreuses choses permettant d'améliorer ou de préciser les calculs peuvent également être implémentées. Notamment la gestion de la localisation des poutres dans les structures, des zones d'application et des points d'application des charges. La différentiation des poutres primaires et secondaires. La gestion d'efforts autres que la compression tels que la traction par exemple pour gérer les structures en porte-à-faux voire les structures suspendues. Étendre à des modèles de calculs plus complexes que le modèle poteau-poutre, ajouter la prise en compte de contreventements et des actions climatiques serait également un gros plus. Le domaine est vaste et les idées ne manquent pas. La liste est sans fin.

Finalement et je m'arrêterai ici promis, mais il faut envisager la possibilité de s'extraire du framework Unity qui bien qu'intéressant pour la création rapide d'une application fonctionnelle reste contraignante sur de nombreux points. Envisager d'utiliser OpenGL ou une autre technologie de bas niveaux semble être l'un des meilleurs compromis pour repousser les limites possibles et augmenter drastiquement en flexibilité, hors du carcan des GameObject et des prefabs. De plus le changement de langage vers un langage de calcul plus performant reste également une piste très probable d'évolution.