# BilleGateCoin - BGC

Scattered Corporation

# RAPPORT DE SOUTENANCE

réalisé par

aurele.oules • leo.gervoson • raphael.brenn

Projet EPITA 2020

# Table des matières

**1 Introduction** **3**

**2 CLI** **3**

**3 Génération du portefeuille** **3**
3.1 Génération d'une clé privée . . . 3
3.2 Clé publique . . . 3
3.3 Adresse . . . 4
3.4 Chiffrement . . . 5

**4 Sérialisation** **6**
4.1 Contrats . . . 6
4.1.1 Placement de billes . . . 6
4.1.2 StartContract . . . 6
4.1.3 ThrowContract . . . 8
4.1.4 TransactionContract . . . 9
4.1.5 ClaimContract . . . 9
4.2 Block . . . 9
4.2.1 Block header . . . 9
4.2.2 Block . . . 10

**5 Minage** **11**
5.1 Implémentation . . . 11
5.2 Difficulté variable . . . 12

**6 Signatures** **13**

**7 Persistance** **14**

**8 Réseau** **15**
8.1 Architecture . . . 15
8.2 Peer to peer (P2P) . . . 15
8.3 Supernodes . . . 15
8.4 Messages . . . 16

**9 Plateau de jeu** **16**
9.1 Première version 9,90 Mo . . . 16
9.2 Deuxième version 447,45 Ko . . . 17

**10 Design des billes** **17**
10.1 Modèles 3D . . . 18
10.2 Icônes . . . 18

**11 Site web** **19**
11.1 Développement . . . 19
11.2 Documentation . . . 19

**12 Conclusion** **21**

# 1 Introduction

BilleGateCoin est une plateforme décentralisée pour un jeu de billes. Dans ce rapport de soutenance nous allons évoquer les différentes fonctions et tâches implémentées dans notre projet. Nous discuterons également de ce qu'il reste à faire.

# 2 CLI

Une implémentation d'une CLI[1] est nécessaire pour le test du protocole de la blockchain. Elle s'avère également utile lorsqu'un utilisateur souhaite communiquer directement avec la blockchain, plutôt que de passer par l'interface graphique qui sera créée sur Unity. Quelques commandes ont été implémentées par **Aurèle** :

- $init$ : initialise la blockchain
- $signcontract$ : prend en argument un contrat en héxadécimal et le sign avec la clé privée du joueur
- $decodecontract$ : prend en argument un contract en héxadécimal et le décode
- $createwallet$ : prend en argument un mot de passe de chiffrement et génère un nouveau porte-feuille
- $deletewallet$ : supprime le porte-feuille
- $inventory$ : pas encore implémentée. Cette fonction doit retourner l'inventaire d'un joueur en parcourant le **world state**
- $start$ : démarre la node pour communiquer en TCP/IP
- $send$ : permet d'envoyer à une node un message texte (pour tester)

# 3 Génération du portefeuille

Un portefeuille consiste d'une clé privée et d'une clé publique correspondante. Celui-ci permet de signer des contrats sur la plate-forme and remplace le système basique d'email/mot de passe. La génération d'un portefeuille se fait en plusieurs étapes.

ECDSA est l'algorithme cryptographique utilsé par BGC pour signer les contrats. BilleGateCoin utilise la courbe elliptique **secp256k1**. Nous avons décidé d'utiliser le même algorithme de signature ainsi que la même courbe elliptique que le Bitcoin afin de pouvoir comparer nos résultats et également utiliser les normes déjà établies par la communauté.

La création d'une clé privée, clé publique et d'une adresse a été réalisé par **Léo**. **Aurèle** a travaillé sur le chiffrement de la clé privée.

## 3.1 Génération d'une clé privée

Une clé privée est un nombre aléatoire compris entre $1$ et $2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$. Ce nombre doit être stocké de manière très sécurisée.

## 3.2 Clé publique

Une clé publique est dérivée de la clé privée avec l'algorithme ECDSA sur la courbe **secp256k1**.

1. Command Line Interface

## 3.3 Adresse

Voici les étapes de génération d'une adresse.

- Générer une clé privée
- Dériver la clé publique
- Réaliser un hash SHA-256 sur la clé publique
- Réaliser un hash RIPE-MD sur le hash SHA-256
- Ajouter l'octet de version devant le hash RIPE-MD (0x8A)
- Réaliser le hash double hash SHA-256 du hash RIPE-MD versionné
- Ajouter les 4 premiers octets du double hash (checksum) à la fin du hash RIPE-MD versionné

Exemple d'adresse : xZ7h1HUEcraAAHiJBqm1aDnr9nXJBh6yz8

```
public byte[] Address() {
    SHA256 sha526 = SHA256.Create();

    // Step 2: SHA256 hash
    byte[] hash = sha526.ComputeHash(PublicKey);

    // Step 3: RIPEMD-160 hash
    RIPEMD160 ripemd = RIPEMD160Managed.Create();
    byte[] ripemdHash = ripemd.ComputeHash(hash);

    // Step 4: Add version byte (0x00)
    byte[] versioned = new byte[ripemdHash.Length + 1];
    versioned[0] = Version;

    for (int i = 0; i < ripemdHash.Length; i++)
    {
        versioned[i + 1] = ripemdHash[i];
    }

    // Step 5 & 6: SHA256 twice
    hash = sha526.ComputeHash(versioned);
    hash = sha526.ComputeHash(hash);

    // Step 7 & 8: add trimmed hash at the end of the versioned hash
    byte[] address = new byte[ripemdHash.Length + 5];
    {
        int i = 0;
        for (; i < versioned.Length; i++)
        {
            address[i] = versioned[i];
        }
        for (int j = 0; j < 4; j++)
        {
            address[i + j] = hash[j];
        }
    }

    return address;
}
```

## 3.4 Chiffrement

Une clé privée doit être stockée et chiffrée. Pour se faire, nous avons adopter l'algorithme de chiffrement **AES**. Lors de la création d'un nouveau porte-feuille, l'utilisateur est demandé d'entrer un mot de passe pour chiffrer sa clé privée. Celle-ci est sauvegardé en tant que $wallet.dat$ et ne peut être déchiffrée uniquement avec le mot de passe de l'utilisateur.

# 4 Sérialisation

La sérialisation est le codage d'une structure de données dans un format qui peut être lu et restitué plus tard. BilleGateCoin doit passer par ce processus de sérialisation afin de pouvoir stocker les différents contrats et block de la blockchain sur un disque dur, de manière compacte et optimisée. **Aurèle** a implémenté ces différentes fonctions.

## 4.1 Contrats

### 4.1.1 Placement de billes

Un placement de billes est une structure de données réutilisée dans plusieurs contrats. Un placement permet à un joueur de placer une ou plusieurs de ses billes dans un contrat pour les jouer, ou simplement les échanger.

| Octets | Valeur |
|---|---|
| 1 | Nombre de types |
| 1 | Type |
| 4 | Quantité |
| ... | ... |
| 1 | Type |
| 4 | Quantité |

### 4.1.2 StartContract

Ce contrat doit être signé par les deux joueurs afin de créer une nouvelle partie. Celui-ci contient les billes mises en jeu des deux joueurs, leur adresse respective, et leur signature. La première bille du placement est celle jouée par le joueur.

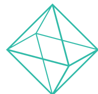
| Octets | Valeur |
|---|---|
| 1 | Version |
| 1 | Type = 0 |
| ? | Fee (Placement) |
| ? | Placement du joueur 1 |
| ? | Placement du joueur 2 |
| 25 | Adresse du joueur 1 |
| 25 | Adresse du joueur 2 |
| 4 | Nonce du joueur 1 |
| 65 | Signature du joueur 1 |
| 4 | Nonce du joueur 2 |
| 65 | Signature du joueur 2 |

Un contrat sérialisé puis converti en héxadécimal ressemble à ceci :

010002000C000000010F00000002024000000000420000000002057A000000080200000008A9B9E6AD7633E9
F498D044D22B8F6466EEA73DCD727E966458A4BA642E726EAB822B91C6FEA82802466AC6BA4E00115D2C2
04000000D59C2D1DA1F26883303C2541F35F406ADCB7EAB78DAA5B1A15975B0DBE3DB0C62B149E7B93F83
430173FEFABC8F3A2F6E680F5A236F07424B5BA86F293B7747000080000007A95F34FA2752E1588139BEB
68042D24A7B42D75EB012BD0A9A6F67984C92F4647B42F8B9D2EC0C94CD997CDA463C5067E488EFB39163
1C8EEB8820F6CCF5A0D00

Celui-ci peut être converti en QR Code pour pouvoir être envoyé à un joueur facilement.

Ce contrat peut ensuite facilement être lu par le protocole.



```
    Version: 1
Type: 0
Fee:
        Type: 0
        Amount: 12

        Type: 1
        Amount: 15

Player One Placement:
        Type: 2
        Amount: 64

        Type: 4
        Amount: 32

Player Two Placement:
        Type: 5
        Amount: 122

        Type: 8
        Amount: 2

Player One Address: xmq5PkTAp21ajwLsM6VEuofwUjrNXeix6p
Player Two Address: xeYEjQwU3Q35zCp4GCxHDPHWphHRh1FB1j
Player One Nonce: 4
Player One Sig: 213 156 45 29 161 242 104 131 48 60 37 65 243 95 64 106 220 183 234
183 141 170 91 26 21 151 91 13 190 61 176 198 43 20 158 123 147 248 52 48 23 63 239
171 200 243 162 246 230 128 245 162 54 240 116 36 181 186 134 242 147 183 116 112 0
Player Two Nonce: 8
Player Two Sig: 122 149 243 79 162 117 46 21 136 19 155 235 104 4 45 36 167 180 45 117
235 1 43 208 169 166 246 121 132 201 47 70 71 180 47 139 157 46 192 201 76 217 151 205
164 99 197 6 126 72 142 251 57 22 49 200 238 184 130 15 108 207 90 13 0
```

### 4.1.3 ThrowContract

Ce contrat doit être créé dès qu'un joueur lance sa bille. Il contient les informations du joueur (sa clé publique), son vecteur de lancé $(X, Z)$, le hash de la partie en cours, et sa signature.

| Octets | Valeur |
|---|---|
| 1 | Version |
| 1 | Type = 1 |
| ? | Fee (Placement) |
| 1 | Vec.X |
| 1 | Vec.Z |
| 32 | Hash de la partie |
| 4 | Nonce de la partie |
| 4 | Nonce du joueur |
| 65 | Signature |

### 4.1.4 TransactionContract

Ce contrat doit être signé par les deux joueurs lorsqu'ils souhaitent échanger instantanément des billes. Il possède la même structure que le **StartContract** sauf que son octet de version est $2$.

### 4.1.5 ClaimContract

Ce contrat n'est pas encore implémenté. Celui-ci devra être signé par le vainqueur de la partie afin de récupérer les mises des joueurs.

## 4.2 Block

Un block est une liste de contrats. Une fois qu'un block est ajouté à la blockchain, celui-ci ne peut être ni modifié, ni supprimé, il existe pour toujours.

### 4.2.1 Block header

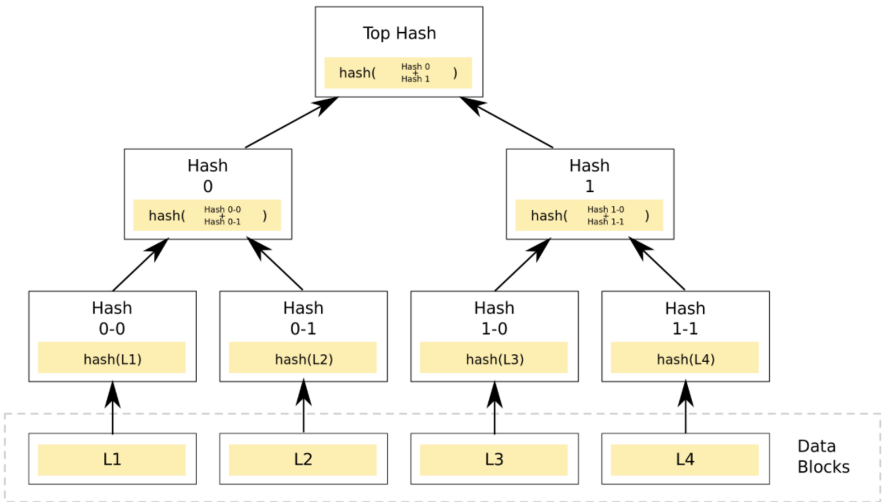
| Octets | Valeur |
|---|---|
| 1 | Version |
| 32 | Hash précédent |
| 32 | Merkle root |
| 4 | Date |
| 4 | Difficulté (target) |
| 4 | Nonce |

### 4.2.2 Block

| Octets | Valeur |
|---|---|
| 77 | Block header |
| 4 | Nombre de contrats |
| ? | Contrats sérialisés |

Chaque block contient la racine du Merkle Tree (Merkle Root). C'est un arbre binaire qui contient les hash SHA-256 de tous les contrats du block. Celui-ci permet de vérifier rapidement si un contrat appartient à un block et asssure son authenticité lorsqu'il est transmis.

```
public MerkleTree(byte[][] data) {
    List<MerkleNode> nodes = new List<MerkleNode>();

    // If tree branches is odd, duplicate last branch
    if (data.Length % 2 != 0) {
        data.Append(data[data.Length - 1]);
    }

    for (uint i = 0; i < data.Length; i++) {
        MerkleNode node = new MerkleNode(null, null, data[i]);
        nodes.Add(node);
    }

    for (uint i = 0; i < data.Length / 2; i++) {
        List<MerkleNode> level = new List<MerkleNode>();
        for (int j = 0; i < nodes.Count; j++) {
            MerkleNode node = new MerkleNode(nodes[j], nodes[j + 1], null);
            level.Add(node);
        }

        nodes = level;
    }

    Root = nodes[0];
}
```




# 5 Minage

Le minage assure que la blockchain reste immuable. Pour ajouter un block à la blockchain, chaque mineur doit essayer de résoudre un puzzle. Ce puzzle est un hash du block qui doit être compris entre $1$ et la cible variable de difficulté. Un hash est un nombre aléatoire entre $1$ et $2^{256} - 1$. Lorsqu'un mineur trouve un hash valide, le mineur forme un block avec un contrat qui lui permet de récupérer sa récompense. Cette récompense est le coeur de l'économie de BilleGateCoin (voir cahier des charges). Chaque ordinateur se synchronise avec la chaîne la plus longue, celle contenant le plus de block valides.

Ce hash assure la stabilité de la chaîne car cette tâche est très difficile à résoudre pour un individuel, mais simple pour le réseau entier. De plus, si un mineur malveillant voulait altérer le contenu d'un block il devrait miner tous les blocks suivants jusqu'à posséder la chaîne la plus longue et ainsi être accepté par les nodes. Cette attaque est donc très peu problable.

## 5.1 Implémentation

Dans cette implémentation de **Aurèle** du protocole BilleGateCoin, le minage s'effectue sur le processeur de l'utilisateur, ce qui n'est pas très optimisé. Une autre implémentation pourrait être effectuée pour être compatible avec les cartes graphiques modernes, ou mêmes sur des circuits intégrés tels que les ASIC. De plus, le C# ne nous permet pas d'obtenir de grandes performances.

```
public (uint, byte[]) Run() {
    byte[] hash = new byte[32];

    uint nonce = 0;
    while (nonce < UInt32.MaxValue) {
        byte[] data = InitData(nonce);

        // Compute hash of block
        SHA256 sha = SHA256.Create();
        hash = sha.ComputeHash(data);

        // SHA256 returns an unsigned byte array
        // But BigInteger only accepts signed byte arrays
        // Append zero-byte to make it positive
        BigInteger intHash = new BigInteger(hash, true);
        BigInteger target = new BigInteger(Target, true);
        // If block hash is below target, it is valid
        if (intHash.CompareTo(target) == -1) {
            break;
        }

        // Compute a different hash by changing the nonce
        nonce++;
    }
    return (nonce, hash);
}
```

## 5.2 Difficulté variable

La difficulté de minage est une variable qui assure que les mineurs ajoutent des blocks à la blockchain de manière régulière même si le nombre de mineurs, donc de puissance de calcul, augmente.

Cette difficulté permet aux nodes de prévoir l'espace disque nécessaire pour stocker la blockchain et assure une économie stable (voir section suivante). Elle doit donc être ajustée régulièrement.

Le *block time* est une constante qui définit le temps qu'un mineur met pour ajouter un block à la chaîne.

Sur la plateforme de jeu de billes, le block time doit être assez court (entre 20s et une minute) afin de permettre des exécutions de contrats (créations de parties, et lancés de billes) rapides. Par exemple, le Bitcoin utilise un block time de 10 minutes, et Ethereum un block time de 20 secondes.

Un intervalle trop court peut causer des problèmes sur le délai de distribution des nouveaux blocks entre les nodes.

La difficulté de minage est au départ 1. Celle-ci est ajustée tous les X blocks. Chaque node va calculer le temps que X block prennent pour être minés en théorie, divisé par la moyenne de temps que les X derniers blocks ont pris pour être minés. La difficulté est ainsi multipliée par ce coefficient.

$$nouvelle\ difficulté = difficulté \times \frac{temps\ attendu}{temps\ réel}$$

```
public static byte[] Target() {
    // No need to compute if blockchain height is below 180
    // Or height is not modulo 180
    if (Blockchain.Blockchain.Height < AdjustDifficultyBlocks ||
          Blockchain.Blockchain.Height % AdjustDifficultyBlocks != 0) {
          return Blockchain.Blockchain.LastTarget();
    }


    Iterator iterator = new Iterator();
    uint blocks = 0;

    uint startTime = 0;
    uint endTime = 0;

    while (iterator.CanIterate) {
    Blockchain.Block block = iterator.Next();
    if (blocks == 0) {
    endTime = block.BlockHeader.Timestamp;
    }
    if (blocks == AdjustDifficultyBlocks) {
    startTime = block.BlockHeader.Timestamp;
    break;
    }

    blocks++;
    }

    uint expectedTime = Consensus.BlockTime * AdjustDifficultyBlocks;

    BigInteger targetUint = new BigInteger(Blockchain.Blockchain.LastTarget(), true);
    return BigInteger.Multiply(targetUint, ((endTime - startTime) / expectedTime)).ToByteArray();
}
```

# 6 Signatures

Les signatures permettent d'assurer l'authenticité d'un contrat et de la propriété des billes. Dans un système décentralisé, la manière de procéder à une authentification est celle de signature digitale. **Aurèle** et **Léo** ont travaillé sur cette tâche.

Chaque contrat est signé par une clé privée, et chaque utilisateur peut simplement vérifier son authenticité en faisant correspondre la signature avec la clé publique du joueur.

Pour signer un contrat il suffit de le sérialiser, effectuer un hash SHA-256 de celui-ci et signer ce hash. La signature est concaténée au contrat sérialisé et peut être diffusé sur la blockchain.

Dans le cas d'un contrat mettant en relation deux joueurs tels qu'un StartContract ou un TransactionContract, celui-ci devra être signé par les deux joueurs séquentiellement. Le premier joueur hash le contrat, signe ce contrat et ajoute sa signature au contrat. Le deuxième joueur hash le contrat signé et ajoute sa signature. Ce contrat à double signature peut maintenant être publié sur la blockchain.

```
public bool PartialSign(byte[] privateKey, uint playerOneNonce) {
    PlayerOneNonce = playerOneNonce;
    byte[] serialized = Serialize(ContractHelper.SerializationType.NoSig);

    // Compute signature using serialized byte array
    SHA256 sha256 = new SHA256Managed();
    byte[] hash = sha256.ComputeHash(serialized);

    (byte[] sig, bool valid) = Utils.SignData(hash, privateKey);
    if (!valid) {
        return false;
    }

    PlayerOneSignature = sig;

    return true;
}

public bool Sign(byte[] privateKey, uint playerTwoNonce) {
    PlayerTwoNonce = playerTwoNonce;
    byte[] serialized = Serialize(ContractHelper.SerializationType.Partial);
    // Compute signature using serialized byte array
    SHA256 sha256 = new SHA256Managed();
    byte[] hash = sha256.ComputeHash(serialized);

    (byte[] sig, bool valid) = Utils.SignData(hash, privateKey);
    if (!valid) {
        return false;
    }

    PlayerTwoSignature = sig;
    return true;
}
```

# 7 Persistance

La blockchain est une base de donnée qui ne cesse de croître en taille alors il est primordiale d'avoir un accès rapide à chaque block et contrat précédent tout en optimisant l'espace disque. **Aurèle** était chargé de s'occuper de la persistance de la blockchain.

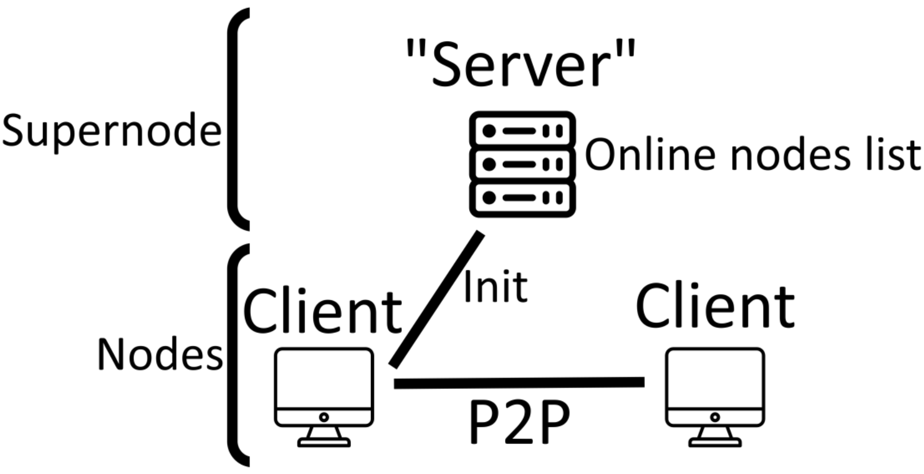
BilleGateCoin utilise **LevelDB** pour stocker les blocks. Cette base de données rapide clé/valeur a été inventé par Google en C++. Elle permet d'immortaliser les données de la chaîne et tenir l'inventaire de chaque joueur de BilleGateCoin, c'est le **world state**.

```
// Save block
DB.Put(block.Hash, block.Serialize());
```

# 8 Réseau

L'implémentation d'un protocole de communication est majeur dans une blockchain. **Léo** s'est occupé de la partie réseau et de la construction des messages via TCP/IP.

## 8.1 Architecture




Le réseau est constitué de :

- Supernodes : serveurs (ports ouverts) permettant aux nodes de communiquer entre elles.
- Nodes : clients

## 8.2 Peer to peer (P2P)

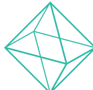
Une blockchain est basée sur la communication P2P entre les nodes. Cependant, les clients utilisent la plupart du temps un routeur NAT dont les ports sont fermés. Il faut donc des supernodes intermediaires afin d'initialiser une connection P2P. On utilisera la méthode du hole punching pour ouvrir les ports des clients.

## 8.3 Supernodes

Les supernodes auront des addresses IP fixes, hardcodées dans la blockchain. Il faut au minimum une supernode en ligne pour garantir le fonctionnement du réseau. Les supernodes recensent toutes les nodes en ligne. Toute initiation de connection P2P entre nodes se fait en passant par une supernode :

- La node initiant la connection contacte la supernode pour obtenir l'addresse IP de la node visée
- Elle envoie ensuite un paquet à la cible, qui sera rejeté par le pare-feu. Cette étape a pour but d'ouvrir un port acceptant les paquets venant de la cible
- Elle envoie l'ID du port nouvellement ouvert à la supernode, qui le communique à la cible
- La cible envoie un paquet au port ouvert, qui cette fois est accepté
- Les 2 nodes ont à présent chacune un port ouvert acceptant les paquets venant de l'autre





## 8.4 Messages

Les paquets envoyés sur le réseau, appelés messages, possèdent une architecture prédéfinie. Ils suivent l'architecture suivante :

| Bytes | Valeur |
|---|---|
| 1 | Magic (réseau) |
| 1 | Commande |
| 4 | Taille du contenu |
| 4 | Checksum du contenu |
| ? | Contenu |

# 9 Plateau de jeu

Le jeu de billes est traditionnellement relié à l'univers des enfants et de l'école, nous nous sommes donc rapidement mis d'accord sur la nature du plateau de jeu : une cour de récréation. **Raphäel** était chargé du design de la map.

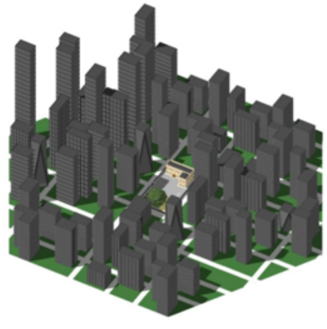
## 9.1 Première version 9,90 Mo

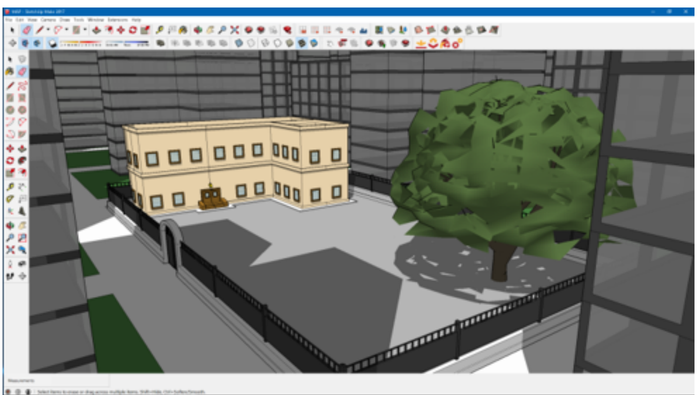
Logiciels utilisés :

— Google Sketchup 2017 (modélisation => export en .dae vers Blender)

— Blender 2.81 (export en .fbx vers Unity 2019.3.4

Cet environnement se compose d'une partie principale (la cour de récréation rectangulaire avec des grilles sur chaque face, un arbre d'un côté et l'école de l'autre) et d'une partie secondaire, le décor (un centre-ville de 70 buildings environ, plus d'une dizaine d'étages chacun). Le modèle est constitué presque exclusivement de composants originaux modélisés pour l'occasion, seul le bâtiment de l'école est un fichier téléchargé sur internet qui a été retravaillé ensuite pour l'incorporer dans le reste de la scène. Cette première version était finie fin janvier, sauf qu'au moment de l'incorporer au repo Git nous nous sommes rendus compte que le modèle était difficile à gérer dans Unity car la taille du fichier était beaucoup trop importante... le travail a donc été repris pour alléger le fichier.





## 9.2 Deuxième version 447,45 Ko

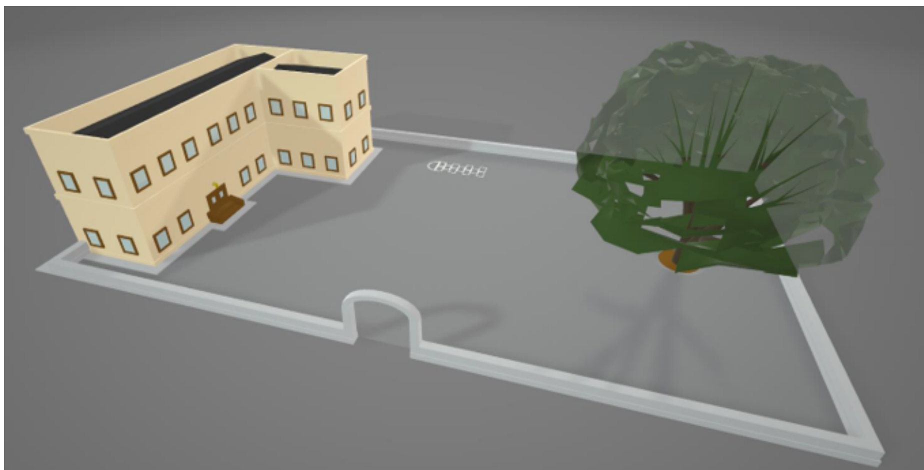
Logiciels utilisés :

— Google Sketchup 2017 (modélisation => export en .dae vers Blender)

— Blender 2.81 (export en .fbx vers Unity 2019.3.4

Cette deuxième version ne contient plus que la partie principale dénuée des grilles trop complexes (donc seulement la cour de récréation rectangulaire avec un arbre d'un côté et l'école de l'autre).

Elle fut finie début février, environ deux semaines après, et ne faisait plus que 4,8% de sa taille originale (qui a engendré un gain considérable en performances). Cette version requiert encore un peu de travail, surtout par rapport aux grilles qu'il va falloir remettre (en version simplifiée bien sûr).



# 10 Design des billes

C'est une partie du travail qui évolue lentement mais sûrement depuis le début du projet, et qui a (depuis fin février environ) atteint un stade assez satisfaisant pour qu'elle puisse être considéré comme temporairement terminée. Le jeu compte aujourd'hui six classes de billes, cinq d'entre elles correspondent chacune à cinq modèles différents (chaque modèle étant lui même disponible en six

couleurs différentes), et la dernière classe étant la classe « spéciale » qui ne contient aujourd'hui qu'un modèle. **Raphäel** s'occupe du design et de la définition des billes.

## 10.1 Modèles 3D

Logiciels utilisés :
- Google Sketchup 2017 (modélisation => export en .dae vers Blender)
- Blender 2.81 (export en .fbx vers Unity 2019.3.4)

Les modèles de billes actuels sont tous des modèles originaux créés pour l'occasion, un autre modèle de test avait été pris d'internet mais n'a pas pris place dans la collection finale.

| Name | Rarity | Blue | Green | Orange | Red | Purple | Dark |
|---|---|---|---|---|---|---|---|
| Elastic | Common | | | | | | |
| Layers | Uncommon | | | | | | |
| Ribbon | Uncommon | | | | | | |
| Stripes | Rare | | | | | | |
| Whirlwind | Extra rare | | | | | | |

Les modèles « pèsent » respectivement 61 Ko, 210 Ko, 53 Ko, 127 Ko et 50 Ko (de haut en bas). Malgré les efforts qui ont été faits (et qui continuent d'être faits) pour rendre les ressources les plus légères possibles, certains modèles sont nettement plus lourds que les autres mais cela ne semble pour l'instant pas trop affecter les performances du jeu, ce n'est donc pas ici considéré comme un problème.

## 10.2 Icônes

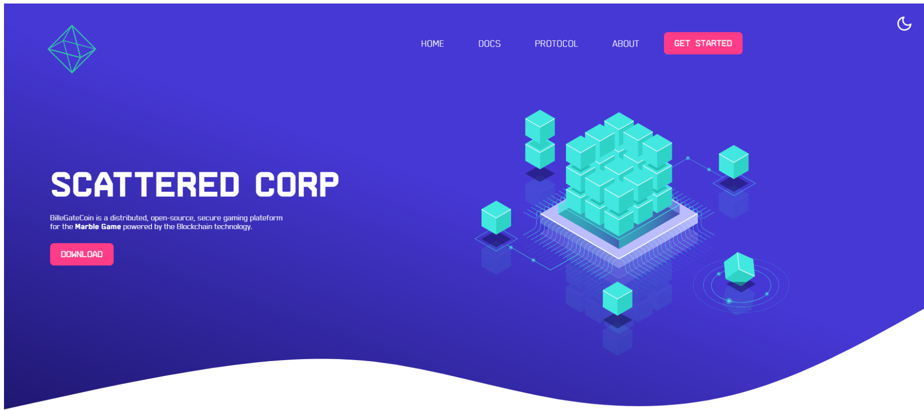
Logiciels utilisés :
- Double Commander 0.9.8 pour la normalisation des fichiers
- 3DBrowser 14.25 pour la génération des miniatures dans le format souhaité

Les fichiers 3D des billes ont d'abord reçu un traitement qui consistait à normaliser leurs noms (sous la forme « [modèle] - [couleur] »). Ensuite, un deuxième traitement a été appliqué sur le groupe de fichiers : deux miniatures, une en .png (sur fond blanc) et l'autre en .gif (animation rotative antihoraire – 24 fps – fond blanc) ont été créées pour chaque bille. Ces miniatures ont pu servir à l'élaboration d'un tableau inventaire animé qui viendra enjoliver le site internet, et serviront également sûrement pour les menus du jeu (un tableau de GIF est moins gourmand en ressources que le même tableau de fichiers 3D complets).

# 11 Site web

## 11.1 Développement

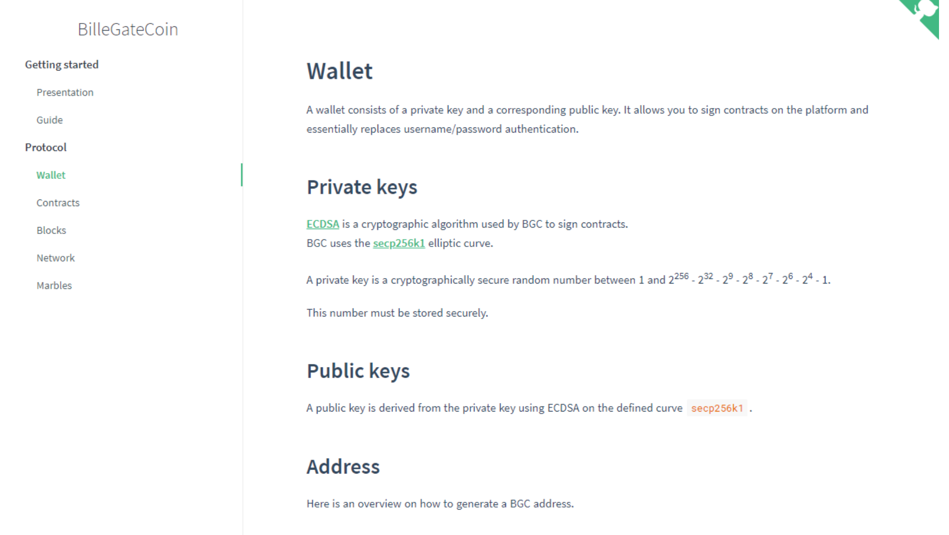
La construction du site web a été commencé par Aurèle. Il est fait en JavaScript, React.js et Sass. Disponible à l'adresse **www.scatteredcorp.tech**.




Le type de design employé est "isometric". On peut y retrouver les valeurs du projet et un lien vers le code source du projet. Le bouton "Download" ne fonctionne pas encore.

## 11.2 Documentation

Une page documentation est disponible depuis le site principal en cliquant sur le bouton "DOCS". Cette page donne des informations sur la partie technique du protocole de BilleGateCoin tels que la construction d'un porte-feuille ou d'un message sur le réseau.



### Step 1

Generate a private key.

`36487317F7C3A171B8499114104A611CACB18E4FE03CB2BAB07C9B7C1AD83411`

### Step 2

Compute the public key.

`04B047580BD5F1E6D2113FFFD79A86B968343D241D0E963AEC0CB6363388C089887029CE4B9275C8A4A836B8258A663FAEEBA2F99C9EBC714C3108AF347410BBBF`

### Step 3

Perform a SHA-256 hash on the public key.

`D43E2A1B29E8E6FE84FD0C11FA627136FAD3AF27E6FFE8BA6FC16FA2067594C7`

### Step 4

Perform a RipeMD hash on the SHA-256 hash.

`10292A5EB397372B47E6715C8F9418D7B0F8A3BB`

### Step 5

Add version byte in front of RipeMD hash ( `0x8A` ).

`8A10292A5EB397372B47E6715C8F9418D7B0F8A3BB`

### Step 6

Perform double SHA-256 hash of the versioned hash (Step 5).

`FB446099A2167287F02265E8BA2D8A460F13A4E35F437D53C525E331B541FC76`

## Message

Below is the minimal structure for a network message:

| Bytes | Value |
|---|---|
| 1 | Magic (network) |
| 1 | Command |
| 4 | Total size |
| 4 | Checksum |
| ? | Message content |

## 12 Conclusion

En conclusion, le groupe est toujours aussi motivé pour terminer notre projet ambitieux. Nous avons pris un peu de retard sur la partie jeu Unity car un membre de **Scattered Corp**, Frédéric est parti. Le projet avance très bien dans la partie blockchain, et nous sommes tous très satisfait des résultats obtenus.