



Documentation - Jumeau numérique de la ferme de Badevel

- 1) Modifier la carte
- 2) Base de données
 - b) Ajouter de nouveaux objets
 - c) Créer de nouvelles fonctions
- 3) Interface utilisateur
 - a) Créer un popup intra-diégétique
 - b) Créer un popup extra-diégétique

1) Modifier la carte

Pour modifier la carte il suffit de l'ouvrir avec Unity, il sera ainsi possible de rajouter des objets voir de modifier des objets qui existent actuellement et de les déplacer. Cela permettrait de mettre à jour la ferme voir d'adapter l'application à d'autre fermes.

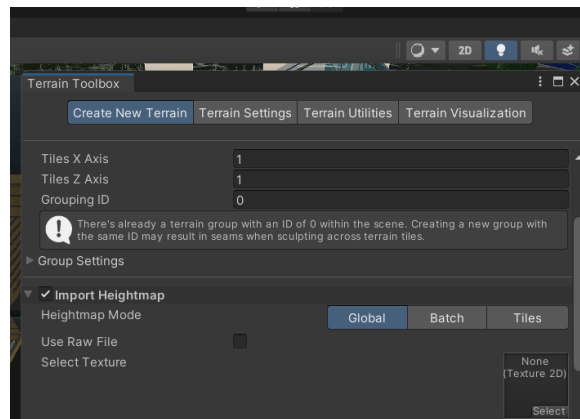
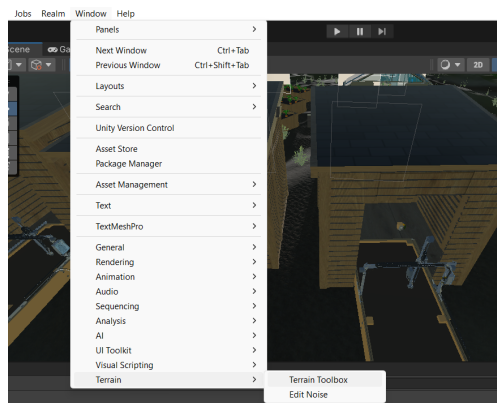
Pour modifier le relief du terrain, plusieurs solutions sont possibles. Toutes les solutions nécessite l'instalation du plugin Terrain Tools.

- La première manière de faire consiste à recréer un terrain de zéro en remplaçant un à un les éléments de la ferme. On sélectionne alors Terrain et on indique sélectionne la texture de terrain au format désiré.



Attention

Pour un format image, il est nécessaire d'utiliser une image carrée.

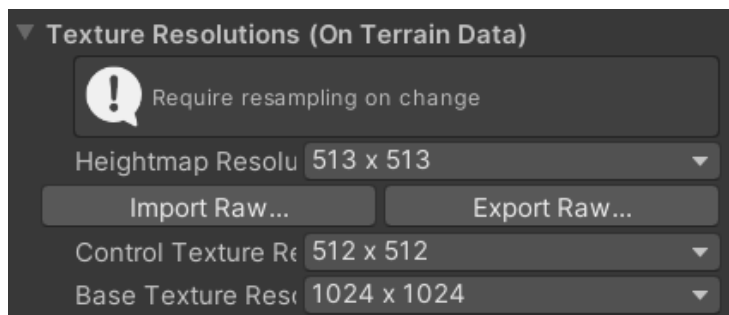
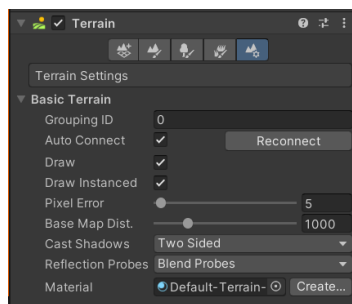


- On peut également modifier la height map du terrain directement en cliquant dessus. On sélectionne ensuite les options du terrain et on modifie le terrain.



Attention

Pour modifier le terrain, on ne peut qu'utiliser des fichiers .raw en échelles de gris. De tels fichiers peuvent être générés par des logiciels tels que Bryce et édités avec Gimp.



2) Base de données

a) Les fonctions

Toutes les fonctions liées à la base de données se retrouvent dans le script `RealmController.cs` trouvable dans `Assets/Script/RealmController.cs`.

Les fonctions peuvent être séparées en deux catégories: les fonctions de récupération et celle d'envoi. Chacune de ses fonctions prend en entrée ou en sortie un objet, dans le cas d'un envoi quel que soit l'objet la base de données va créer une collection correspondante. Cependant dans le cas d'une réception il faut s'assurer que la collection possède les mêmes champs que l'objet.

Pour modifier les objets ils faut changer directement dans leur fichiers; par exemple si je veux rajouter un champ "taille" à `UserLogin`, voilà à quoi cela va ressembler:

```

public class UserLogin : RealmObject
{
    [PrimaryKey]
    [MapTo("_id")]
    public string id { get; set; }


    [MapTo("password")]
    public string password { get; set; }

    //-----nouveau-----
    [MapTo("taille")]
    public string taille { get; set; }
    //----- fin nouveau-----

    public UserLogin() {}
    //-----nouvel argument-----
    public UserLogin(string id,string password,string taille){

        this.id=id;
        this.password = password;
        //-----nouveau-----
        this.taille = taille;
        //----- fin nouveau-----
    }
}

```

 Les fonctions de récupération de données et de permanence ne sont pas conçu pour effectuer des recherches sur plus d'une semaine. Si vous souhaitez le modifier il suffit de changer les 168 (profondeur de recherche en nombre d'heure) dans le fichier par le nombre d'heures de votre choix.

b) Ajouter de nouveaux objets

Dans l'architecture actuel de nouveaux objets pourrait être représenter comme de nouveau champ dans l'objet datablock cependant si il faut créer une nouvelle collection dans la base de donnée il faut tout d'abord créer une nouvelle collection dans la base de donnée puis un nouvel objet dans les script en suivant le modèles des autres objets: des getter/ setter, des constructeurs.

c) Créer de nouvelles fonctions

Pour créer une nouvelle fonction il faut suivre le pattern suivant:

```

public async Task<List<Perm>> GetPerm()
{
    //-----laisser tel quel dans toute les fonctions (sert à se connecter)
    if (email != "" && password != "")
    {
        _realmApp = App.Create(new AppConfiguration(RealmAppId)
        {
            MetadataPersistenceMode = MetadataPersistenceMode.NotEncrypted
        });
        try
        {
            if (_realmUser == null)
            {
                _realmUser = await _realmApp.LogInAsync(Credentials.EmailPassword(email, password));
            }
        }
    }
}

```

```

        _realm = await Realm.GetInstanceAsync(new SyncConfiguration(email, _realmUser));
    }
    else
    {
        _realm = Realm.GetInstance(new SyncConfiguration(email, _realmUser));
    }
}
catch (ClientResetException clientResetEx)
{
    if (_realm != null)
    {
        _realm.Dispose();
    }
    clientResetEx.InitiateClientReset();
}
//-----laisser tel quel dans toute les fonctions (sert à se connecter)(fin)
//----- code + return de la fonction
    return permlog;
//----- code + return de la fonction (fin)
}
//----- null check
    return null;
}

```

Pour interagir avec la base de données seul deux fonctions était implémenté avec le SDK:

- La fonction permettant de recherche par id :

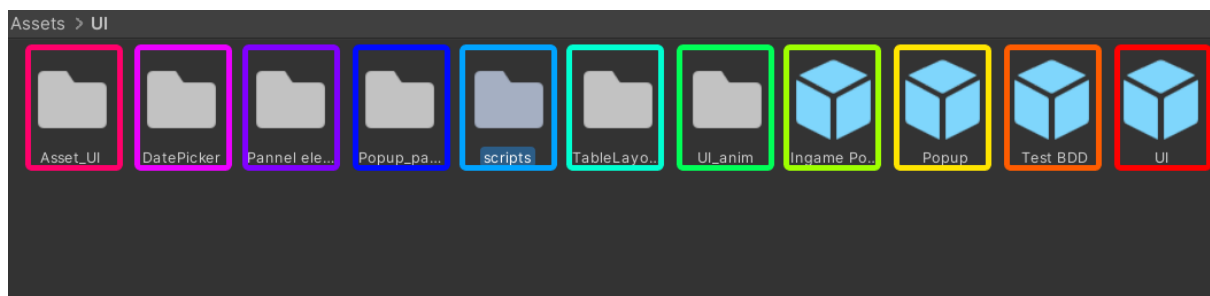
```
_realm.Find<Object>(id)
```

- La fonction permettant d'envoyer une collection vers la base de données :

```
_realm.Add(Object);
```

3) Interface utilisateur

Tous les éléments liés à l'interface utilisateur sont tous placés dans le dossier UI des Assets :



- L'UI



Il s'agit d'un Prefab Correspondant à l'interface utilisateur affichée en HUD. Il comprend notamment le bouton pour sortir du logiciel.



- **Test BDD**

Il s'agit d'un prefab utilisé pour réaliser des tests sur la base de données. Il est notamment utilisé pour tester la connexion de l'utilisateur au serveur.



- **Popup**

Il s'agit d'un prefab pour faire apparaître des informations extra diégétiques sur la fenêtre de l'utilisateur. Il s'adapte automatiquement à son contenu. Il n'est pas directement placé dans la scène (cf. **INDIQUER PARTIE**)



- **Ingame Popup**

Ce prefab permet de contenir des données et de les afficher diégétiquement, il peut être placé sur la carte de la ferme à n'importe quel endroit pour donner des informations contextuelles.



- **UI_anim**

Le dossier contenant toutes les animations liées à l'interface utilisateur.



- **TableLayout**

Les fichiers générés par le paquet TableLayout.



- **scripts**

Contient tous les scripts utilisés par les éléments d'interface.



- **Popup_panel**

Contient tous les panels qui sont affichés sur les popups. Dès que l'on désire ajouter un layout de popup, il faut le placer dans ce dossier.



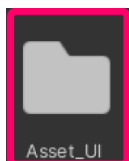
- **Pannel elements**

Contient tous les éléments d'interface placés à l'intérieur des popups. Par exemple, les barres de progression, les boutons, les tableaux, etc.



- **DatePicker**

Les fichiers générés par le paquet DatePicker. Ce paquet permet d'avoir un sélectionneur de date directement dans Unity.



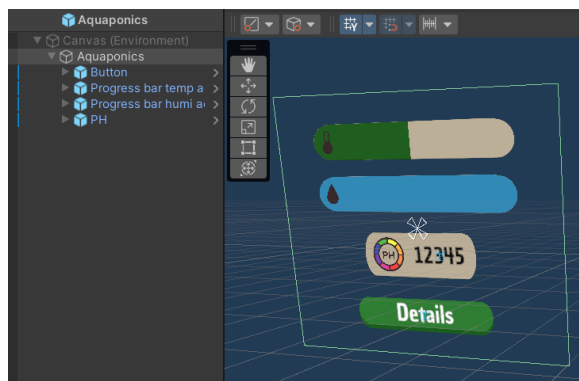
- **Asset_UI**

Contient tous les assets utilisés dans les prefabs de l'interface.

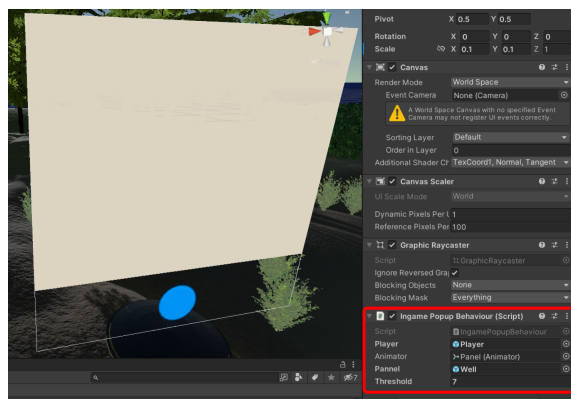
a) Créer un popup intra-diégétique

Pour créer un popup intra-diégétique, il faut :

- Concevoir un prefab placé dans un canevas en tant qu'environnement
- Placer le prefab dans la propriété Panel du Ingame Popup Behavior
- Ajouter le player en tant que propriété Player (cela sert à orienter le panel correctement en fonction de la position du joueur)
- Ajuster le Threshold (la distance d'affichage).



Contenu du Popup



Propriétés du Popup



Les panels à l'intérieur des popups intra-diégétiques ont une taille fixe. Nous conseillons de prendre appui sur un panel déjà existant pour en concevoir un nouveau.



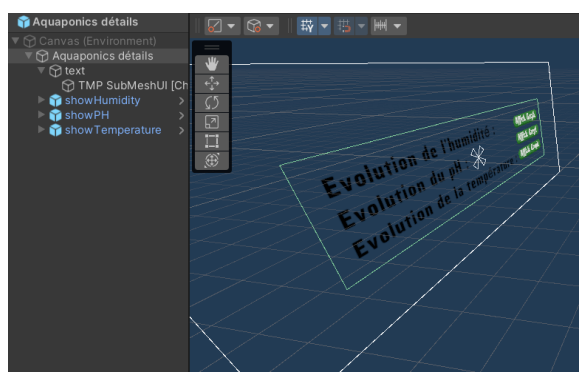
Attention

Si vous rajoutez du texte sur le canevas utilisez un objet/component **TextMeshProUGUI**

b) Créer un popup extra-diégétique

Pour créer un popup extra-diégétique, il faut :

- Concevoir un prefab placé dans un canevas en tant qu'environnement (la forme et la taille ne sont pas limitées)
- Attacher un Popup Generator à un GameObject
- Lier la fonction `displayPopup` à une action ou tout autre type d'événement.
- Glisser le prefab Popup dans la propriété Popup
- Ajouter un titre à la fenêtre de Popup (si vide = pas de titre)
- Ajouter le prefab du panel précédemment conçu.



Canevas du popup



Script Popup Generator sur un bouton

**Attention**

Si vous rajoutez du texte sur le canevas utilisez un objet/component **TextMeshProUGUI**