



eBook Gratuit

APPRENEZ

unity3d

eBook gratuit non affilié créé à partir des
contributors de Stack Overflow.

#unity3d

Table des matières

À propos.....	1
Chapitre 1: Démarrer avec unity3d.....	2
Remarques.....	2
Versions.....	2
Examples.....	5
Installation ou configuration.....	5
Vue d'ensemble.....	5
Installation.....	6
Installation de plusieurs versions d'Unity.....	6
Éditeur de base et code.....	6
Disposition.....	6
Disposition Linux.....	7
Utilisation de base.....	7
Script de base.....	8
Disposition de l'éditeur.....	8
Personnalisation de votre espace de travail.....	10
Chapitre 2: API CullingGroup.....	13
Remarques.....	13
Examples.....	13
Culling object distances.....	13
Suppression de la visibilité des objets.....	15
Distances limites.....	16
Visualiser les distances limites.....	16
Chapitre 3: Asset Store.....	18
Examples.....	18
Accéder à la banque de biens.....	18
Achats d'actifs.....	18
Importation de biens.....	19
Actifs de publication.....	19
Confirmer le numéro de facture d'un achat.....	20

Chapitre 4: Collision	21
Examples	21
Les collisionneurs	21
Box Collider	21
Propriétés	21
Exemple	22
Sphère Collider	22
Propriétés	22
Exemple	23
Collisionneur de capsules	23
Propriétés	24
Exemple	24
Collisionneur de roue	24
Propriétés	24
Ressort de suspension	25
Exemple	25
Collisionneur de maille	25
Propriétés	26
Exemple	27
Collisionneur de roue	27
Déclencheurs collisionneurs	29
Les méthodes	29
Script de déclenchement du collisionneur	30
Exemple	30
Chapitre 5: Comment utiliser les packages d'actifs	31
Examples	31
Paquets d'actifs	31
Importer un paquet .unity	31
Chapitre 6: Communication avec le serveur	33
Examples	33
Obtenir	33

Simple Post (Post Fields).....	33
Poster (Télécharger un fichier).....	34
Télécharger un fichier zip sur le serveur.....	34
Envoi d'une requête au serveur.....	34
Chapitre 7: Coroutines.....	38
Syntaxe.....	38
Remarques.....	38
Considérations de performance.....	38
Réduisez les erreurs en mettant en cache YieldInstructions.....	38
Examples.....	38
Coroutines.....	38
Exemple.....	40
Mettre fin à une coroutine.....	40
Méthodes MonoBehaviour qui peuvent être Coroutines.....	42
Enchaînement de coroutines.....	42
Façons de céder.....	44
Chapitre 8: Couches.....	47
Examples.....	47
Utilisation de la couche.....	47
Structure du masque de calque.....	47
Chapitre 9: Développement multiplateforme.....	49
Examples.....	49
Définitions du compilateur.....	49
Organisation des méthodes spécifiques de la plate-forme aux classes partielles.....	49
Chapitre 10: Éclairage Unity.....	51
Examples.....	51
Types de lumière.....	51
Lumière de zone.....	51
Lumière directionnelle.....	51
Point de lumière.....	52
Projecteur.....	53

Note sur les ombres	54
Émission	55
Chapitre 11: Extension de l'éditeur	57
Syntaxe	57
Paramètres	57
Exemples	57
Inspecteur Personnalisé	57
Tiroir de propriété personnalisée	59
Articles de menu	62
Gizmos	67
Un exemple	67
Exemple deux	69
Résultat	69
Non sélectionné	70
Choisi	70
Fenêtre de l'éditeur	71
Pourquoi une fenêtre d'édition?	71
Créer un éditeur de baseWindow	71
Exemple simple	71
Aller plus loin	72
Sujets avancés	75
Dessin dans le SceneView	75
Chapitre 12: Implémentation de classe MonoBehaviour	80
Examples	80
Aucune méthode surchargée	80
Chapitre 13: Importateurs et processeurs (post)	81
Syntaxe	81
Remarques	81
Exemples	81
Postprocesseur de texture	81
Un importateur de base	82

Chapitre 14: Intégration des annonces	.86
Introduction	.86
Remarques	.86
Examples	.86
Unity Ads Basics en C #	.86
Unity Ads Basics en JavaScript	.87
Chapitre 15: La mise en réseau	.88
Remarques	.88
Mode sans tête dans l'unité	.88
Examples	.89
Créer un serveur, un client et envoyer un message	.89
La classe que nous utilisons pour sérialiser	.89
Créer un serveur	.89
Le client	.91
Chapitre 16: La physique	.93
Examples	.93
Corps rigides	.93
Vue d'ensemble	.93
Ajouter un composant Rigidbody	.93
Déplacement d'un objet Rigidbody	.93
Masse	.93
Traîne	.93
isKinematic	.94
Contraintes	.94
Les collisions	.94
Gravité dans un corps rigide	.95
Chapitre 17: Les attributs	.97
Syntaxe	.97
Remarques	.97
SerializeField	.97

Examples.....	98
Attributs communs d'inspecteur.....	98
Attributs de composant.....	100
Attributs d'exécution.....	101
Attributs de menu.....	102
Attributs de l'éditeur.....	104
Chapitre 18: Modèles de conception.....	108
Examples.....	108
Modèle de conception MVC (Model View Controller).....	108
Chapitre 19: Mots clés.....	112
Introduction.....	112
Examples.....	112
Création et application de balises.....	112
Définition de balises dans l'éditeur.....	112
Définition de balises via un script.....	112
Création de balises personnalisées.....	113
Recherche de GameObjects par tag:.....	114
Trouver un seul GameObject.....	114
Recherche d'un tableau d'instances GameObject.....	115
Comparaison de balises.....	115
Chapitre 20: Optimisation.....	117
Remarques.....	117
Examples.....	117
Vérifications rapides et efficaces.....	117
Contrôle de distance / distance.....	117
Chèques de Limites.....	117
Mises en garde.....	117
Puissance de coroutine.....	118
Usage.....	118
Fractionnement de routines de longue durée sur plusieurs images.....	118
Effectuer des actions coûteuses moins souvent.....	118

Pièges courants	119
Cordes	119
Les opérations de type string créent des ordures	119
Cachez vos opérations de chaîne	119
La plupart des opérations de chaîne sont des messages de débogage	120
Comparaison de chaîne	121
Références du cache	121
Évitez d'appeler des méthodes à l'aide de chaînes	122
Éviter les méthodes d'unité vide	123
Chapitre 21: Plates-formes mobiles	124
Syntaxe	124
Examples	124
Déetecter le toucher	124
TouchPhase	124
Chapitre 22: Plugins Android 101 - Une introduction	126
Introduction	126
Remarques	126
Commençant par les plugins Android	126
Aperçu de la création d'un plugin et de la terminologie	126
Choisir entre les méthodes de création de plugins	127
Examples	127
UnityAndroidPlugin.cs	127
UnityAndroidNative.java	127
UnityAndroidPluginGUI.cs	128
Chapitre 23: Préfabriqués	129
Syntaxe	129
Examples	129
introduction	129
Créer des préfabriqués	129
Inspecteur préfabriqué	130
Préfabrication	131

Instanciation du temps de conception	131
Instanciation du runtime	132
Préfabriqués imbriqués	132
Chapitre 24: Quaternions	137
Syntaxe	137
Examples	137
Introduction à Quaternion vs Euler	137
Quaternion Look Rotation	137
Chapitre 25: Raycast	139
Paramètres	139
Examples	139
Physique Raycast	139
Physics2D Raycast2D	139
Encapsulation des appels Raycast	140
Lectures complémentaires	141
Chapitre 26: Réalité virtuelle (VR)	142
Examples	142
Plateformes VR	142
SDK	142
Documentation	142
Activation du support VR	142
Matériel	143
Chapitre 27: Recherche et collecte de GameObjects	145
Syntaxe	145
Remarques	145
Quelle méthode utiliser	145
Aller plus loin	145
Examples	146
Recherche par nom de GameObject	146
Recherche par tags de GameObject	146
Inséré dans les scripts en mode édition	146

Recherche de scripts GameObjects par MonoBehaviour.....	146
Recherche de GameObjects par nom à partir d'objets enfants.....	147
Chapitre 28: Regroupement d'objets.....	148
Examples.....	148
Pool d'objets.....	148
Pool d'objets simple.....	150
Un autre pool d'objets simples.....	152
Chapitre 29: Ressources.....	154
Examples.....	154
introduction.....	154
Ressources 101.....	154
introduction.....	154
Mettre tous ensemble.....	155
Notes finales.....	155
Chapitre 30: ScriptableObject.....	157
Remarques.....	157
ScriptableObjects avec AssetBundles.....	157
Examples.....	157
introduction.....	157
Création d'actifs ScriptableObject.....	157
Créer des instances ScriptableObject via le code.....	158
ScriptableObjects sont sérialisés dans l'éditeur même dans PlayMode.....	158
Rechercher des ScriptableObjects existants lors de l'exécution.....	159
Chapitre 31: Se transforme.....	160
Syntaxe.....	160
Examples.....	160
Vue d'ensemble.....	160
Parenting et enfants.....	161
Chapitre 32: Singletons in Unity.....	163
Remarques.....	163
Lectures complémentaires.....	163

Examples.....	164
Implémentation à l'aide de RuntimeInitializeOnLoadMethodAttribute.....	164
Un simple MonoBehaviour Singleton dans Unity C #.....	164
Advanced Unity Singleton.....	165
Mise en œuvre de singleton via la classe de base.....	167
Motif Singleton utilisant le système Unitys Entity-Component.....	169
Classe Singleton basée sur MonoBehaviour & ScriptableObject.....	170
Chapitre 33: Système audio.....	175
Introduction.....	175
Examples.....	175
Classe audio - Lecture audio.....	175
Chapitre 34: Système d'interface utilisateur (UI).....	176
Examples.....	176
S'abonner à l'événement dans le code.....	176
Ajouter des écouteurs de souris.....	176
Chapitre 35: Système d'interface utilisateur graphique en mode immédiat (IMGUI).....	178
Syntaxe.....	178
Examples.....	178
GUILayout.....	178
Chapitre 36: Système de saisie.....	179
Examples.....	179
Clé de lecture Appui et différence entre GetKey, GetKeyDown et GetKeyUp.....	179
Capteur d'accéléromètre de lecture (Basic).....	180
Capteur d'accéléromètre de lecture (avance).....	180
Capteur d'accéléromètre de lecture (précision).....	181
Cliquez sur le bouton de la souris (gauche, milieu, droite).....	182
Chapitre 37: Unité d'animation.....	185
Examples.....	185
Animation de base pour la course.....	185
Création et utilisation de clips d'animation.....	186
Animation 2D Sprite.....	188
Courbes d'animation.....	190

Chapitre 38: Unity Profiler	193
Remarques	193
Utiliser Profiler sur différents périphériques	193
Android	193
iOS	194
Examples	194
Balisage du profileur	194
Utilisation de la classe de profileur	194
Chapitre 39: Utilisation du contrôle de source Git avec Unity	196
Examples	196
Utilisation de GFS Large File Storage (LFS) avec Unity	196
Avant-propos	196
Installation de Git & Git-LFS	196
Option 1: Utiliser une application Git GUI	196
Option 2: installer Git & Git-LFS	197
Configuration du stockage de fichiers Git Large sur votre projet	197
Mettre en place un dépôt Git pour Unity	197
Unité ignore les dossiers	197
Paramètres du projet Unity	198
Configuration supplémentaire	198
Scènes et préfabriqués fusionnant	199
Chapitre 40: Vector3	200
Introduction	200
Syntaxe	200
Examples	200
Valeurs statiques	200
Vector3.zero et Vector3.one	200
Directions statiques	201
Indice	203
Créer un vecteur3	203

Constructeurs	203
Conversion à partir d'un Vector2 ou Vector4	204
Mouvement d'application	204
Lerp et LerpUnclamped	205
MoveTowards	206
SmoothDamp	207
Crédits	210

À propos

You can share this PDF with anyone you feel could benefit from it, download the latest version from: [unity3d](#)

It is an unofficial and free unity3d ebook created for educational purposes. All the content is extracted from [Stack Overflow Documentation](#), which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official unity3d.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec unity3d

Remarques

Unity fournit un environnement de développement de jeux multi-plateformes pour les développeurs. Les développeurs peuvent utiliser le langage C # et / ou UnityScript basé sur la syntaxe JavaScript pour programmer le jeu. Les plates-formes de déploiement cibles peuvent être facilement modifiées dans l'éditeur. Tous les codes de jeu de base restent les mêmes, à l'exception de certaines fonctionnalités dépendant de la plate-forme. Une liste de toutes les versions et des téléchargements et notes de version correspondants peut être trouvée ici: <https://unity3d.com/get-unity/download/archive> .

Versions

Version	Date de sortie
Unité 2017.1.0	2017-07-10
5.6.2	2017-06-21
5.6.1	2017-05-11
5.6.0	2017-03-31
5.5.3	2017-03-31
5.5.2	2017-02-24
5.5.1	2017-01-24
5.5	2016-11-30
5.4.3	2016-11-17
5.4.2	2016-10-21
5.4.1	2016-09-08
5.4.0	2016-07-28
5.3.6	2016-07-20
5.3.5	2016-05-20
5.3.4	2016-03-15
5.3.3	2016-02-23

Version	Date de sortie
5.3.2	2016-01-28
5.3.1	2015-12-18
5.3.0	2015-12-08
5.2.5	2016-06-01
5.2.4	2015-12-16
5.2.3	2015-11-19
5.2.2	2015-10-21
5.2.1	2015-09-22
5.2.0	2015-09-08
5.1.5	2015-06-07
5.1.4	2015-10-06
5.1.3	2015-08-24
5.1.2	2015-07-16
5.1.1	2015-06-18
5.1.0	2015-06-09
5.0.4	2015-07-06
5.0.3	2015-06-09
5.0.2	2015-05-13
5.0.1	2015-04-02
5.0.0	2015-03-03
4.7.2	2016-05-31
4.7.1	2016-02-25
4.7.0	2015-12-17
4.6.9	2015-10-15
4.6.8	2015-08-26

Version	Date de sortie
4.6.7	2015-07-01
4.6.6	2015-06-08
4.6.5	2015-04-30
4.6.4	2015-03-26
4.6.3	2015-02-19
4.6.2	2015-01-29
4.6.1	2014-12-09
4.6.0	2014-11-25
4.5.5	2014-10-13
4.5.4	2014-09-11
4.5.3	2014-08-12
4.5.2	2014-07-10
4.5.1	2014-06-12
4.5.0	2014-05-27
4.3.4	2014-01-29
4.3.3	2014-01-13
4.3.2	2013-12-18
4.3.1	2013-11-28
4.3.0	2013-11-12
4.2.2	2013-10-10
4.2.1	2013-09-05
4.2.0	2013-07-22
4.1.5	2013-06-08
4.1.4	2013-06-06
4.1.3	2013-05-23

Version	Date de sortie
4.1.2	2013-03-26
4.1.0	2013-03-13
4.0.1	2013-01-12
4.0.0	2012-11-13
3.5.7	2012-12-14
3.5.6	2012-09-27
3.5.5	2012-08-08
3.5.4	2012-07-20
3.5.3	2012-06-30
3.5.2	2012-05-15
3.5.1	2012-04-12
3.5.0	2012-02-14
3.4.2	2011-10-26
3.4.1	2011-09-20
3.4.0	2011-07-26

Examples

Installation ou configuration

Vue d'ensemble

Unity fonctionne sous Windows et Mac. Il existe également une [version alpha Linux](#).

Il y a 4 plans de paiement différents pour Unity:

1. **Personnel** - gratuit (*voir ci-dessous*)
2. **Plus** - 35 USD par mois et par siège (*voir ci-dessous*)
3. **Pro** - \$ 125 USD par mois et par siège - Après vous être abonné au plan Pro pendant 24 mois consécutifs, vous avez la possibilité de ne plus vous abonner et de conserver la version que vous avez.
4. **Enterprise** - [Contact Unity](#) pour plus d'informations

*Selon le CLUF: les sociétés ou entités incorporées dont le chiffre d'affaires dépassait 100 000 USD au cours de leur dernier exercice financier doivent utiliser **Unity Plus** (ou une licence supérieure); Au-delà de 200 000 USD, ils doivent utiliser **Unity Pro** (ou Enterprise).*

Installation

1. Téléchargez l' [assistant de téléchargement Unity](#) .
2. Exécutez l'assistant et choisissez les modules que vous souhaitez télécharger et installer, tels que l'éditeur Unity, l'IDE MonoDevelop, la documentation et les modules de construction de plate-forme souhaités.

Si vous avez une version plus ancienne, vous pouvez [mettre à jour vers la dernière version stable](#)

Si vous souhaitez installer Unity sans l'assistant de téléchargement Unity, vous pouvez obtenir les **programmes d'installation des composants** à partir des [notes de version d' Unity 5.5.1](#) .

Installation de plusieurs versions d'Unity

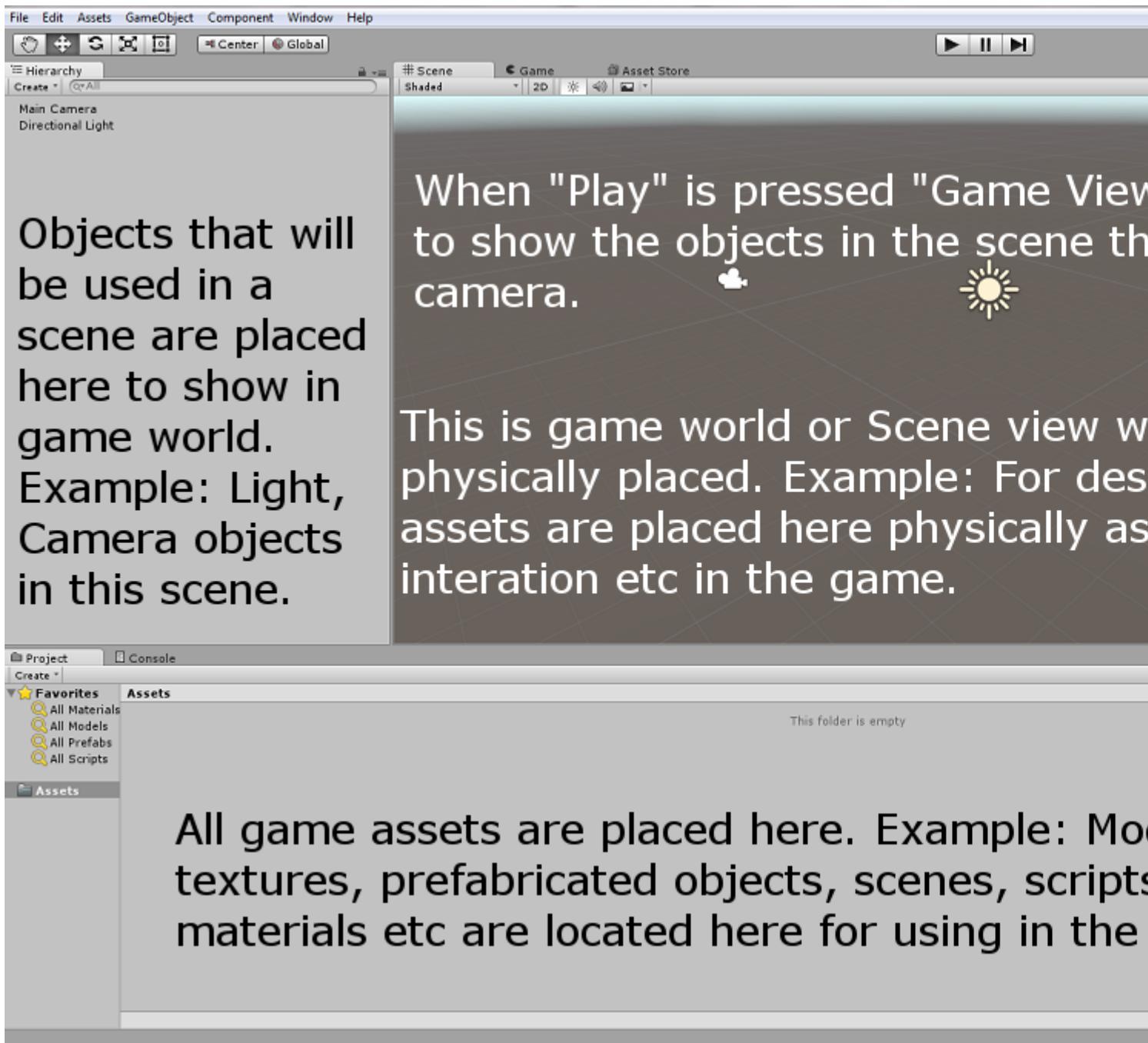
Il est souvent nécessaire d'installer plusieurs versions d'Unity en même temps. Faire cela:

- Sous Windows, remplacez le répertoire d'installation par défaut par un dossier vide que vous avez créé précédemment, tel que `Unity 5.3.1f1` .
- Sur Mac, le programme d'installation sera toujours installé sur `/Applications/Unity` . Renommez ce dossier pour votre installation existante (par exemple pour `/Applications/Unity5.3.1f1`) avant d'exécuter le programme d'installation pour la version différente.
- Vous pouvez maintenir `Alt` lors du lancement d'Unity pour le forcer à vous laisser choisir un projet à ouvrir. Sinon, le dernier projet chargé essaiera de se charger (si disponible) et il pourra vous demander de mettre à jour un projet que vous ne souhaitez pas mettre à jour.

Éditeur de base et code

Disposition

L'éditeur de base d'Unity sera comme ci-dessous. Les fonctionnalités de base de certaines fenêtres / onglets par défaut sont décrites dans l'image.



Disposition Linux

Il y a une petite différence dans la disposition des menus de la version Linux, comme la capture



Utilisation de base

Créez un `GameObject` vide en cliquant avec le bouton droit de la souris dans la fenêtre Hiérarchie et sélectionnez `Create Empty`. Créez un nouveau script en cliquant avec le bouton droit de la souris dans la fenêtre Projet et sélectionnez `Create > C# Script`. Renommez-le si nécessaire.

Lorsque le `GameObject` vide est sélectionné dans la fenêtre Hiérarchie, faites glisser le script

nouvellement créé dans la fenêtre Inspecteur. Le script est maintenant associé à l'objet dans la fenêtre Hiérarchie. Ouvrez le script avec l'EDI MonoDevelop par défaut ou votre préférence.

Script de base

Le code de base ressemblera à celui ci-dessous, à l'exception de la ligne `Debug.Log("hello world!!");`.

```
using UnityEngine;
using System.Collections;

public class BasicCode : MonoBehaviour {

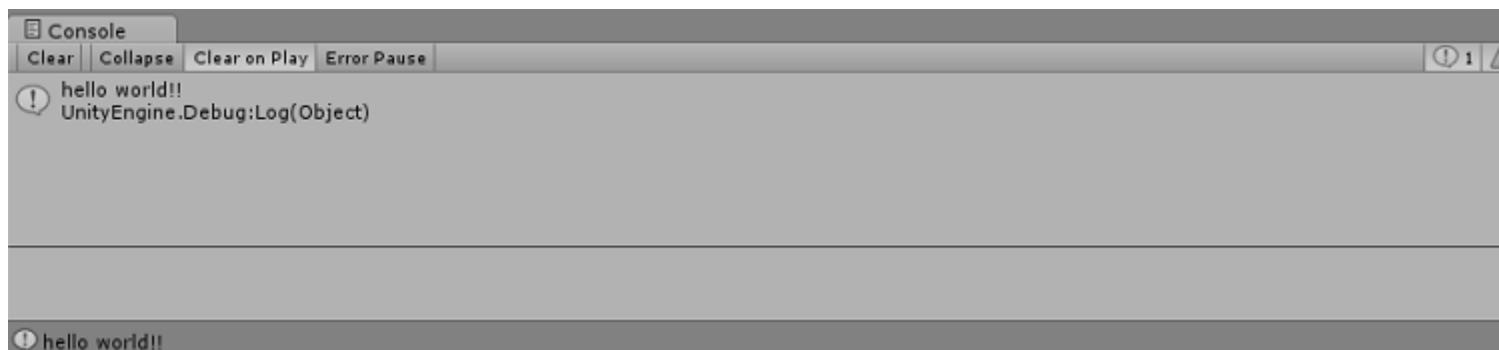
    // Use this for initialization
    void Start () {
        Debug.Log("hello world!!");
    }

    // Update is called once per frame
    void Update () {

    }
}
```

Ajoutez la ligne `Debug.Log("hello world!!");` dans la méthode `void Start()`. Enregistrez le script et revenez à l'éditeur. Exécutez-le en appuyant sur **Play** en haut de l'éditeur.

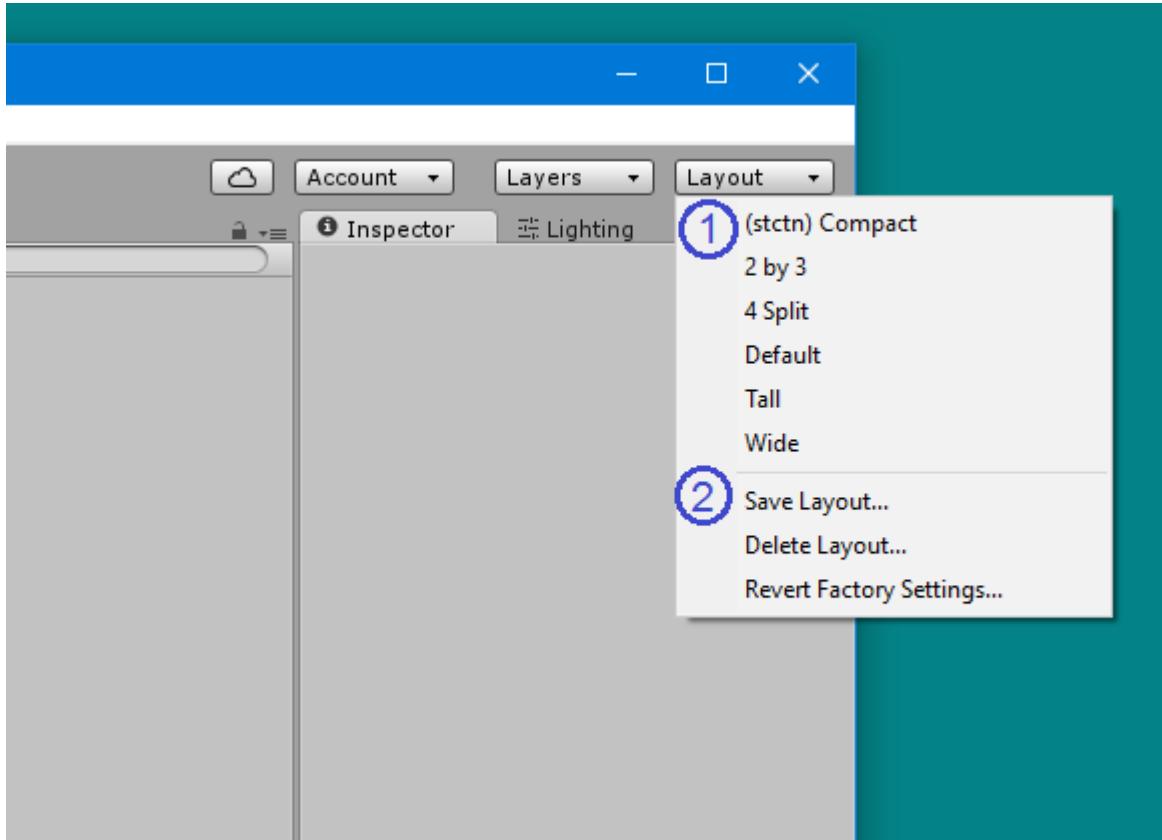
Le résultat doit être comme ci-dessous dans la fenêtre de console:



Disposition de l'éditeur

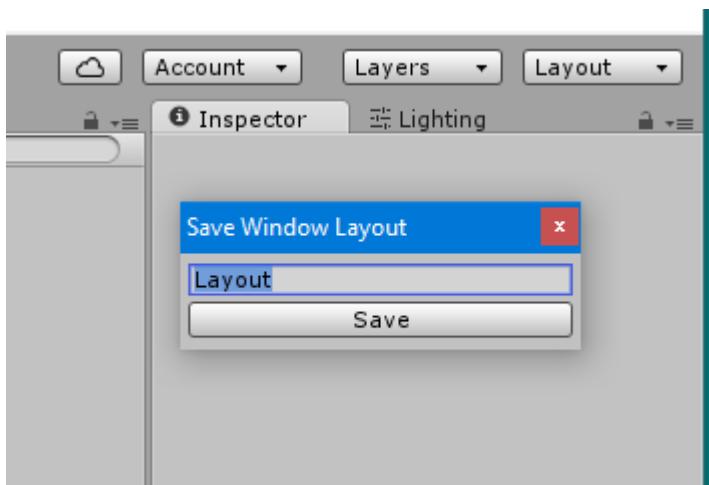
Vous pouvez enregistrer la disposition de vos onglets et fenêtres pour normaliser votre environnement de travail.

Le menu des mises en page se trouve dans le coin supérieur droit de l'éditeur Unity:

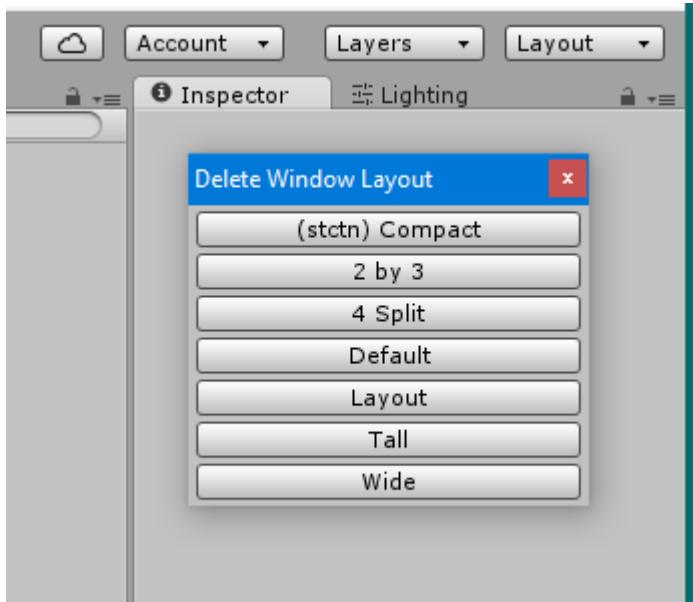


Unity est livré avec 5 dispositions par défaut (2 par 3, 4 Split, Default, Tall, Wide) (*marquées avec 1*) . Dans l'image ci-dessus, hormis les dispositions par défaut, il existe également une disposition personnalisée en haut.

Vous pouvez ajouter vos propres dispositions en cliquant sur le bouton "**Enregistrer la mise en page ...**" dans le menu (*marqué avec 2*) :



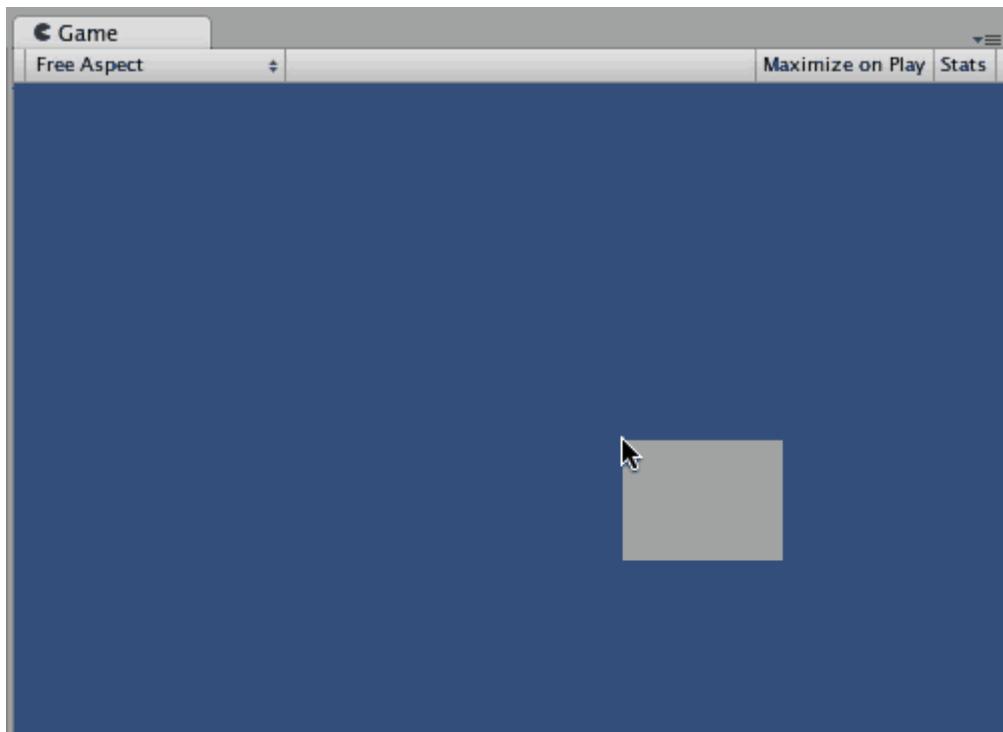
Vous pouvez également supprimer n'importe quelle mise en page en cliquant sur le bouton "**Supprimer la mise en page ...**" dans le menu (*marqué avec 2*) :



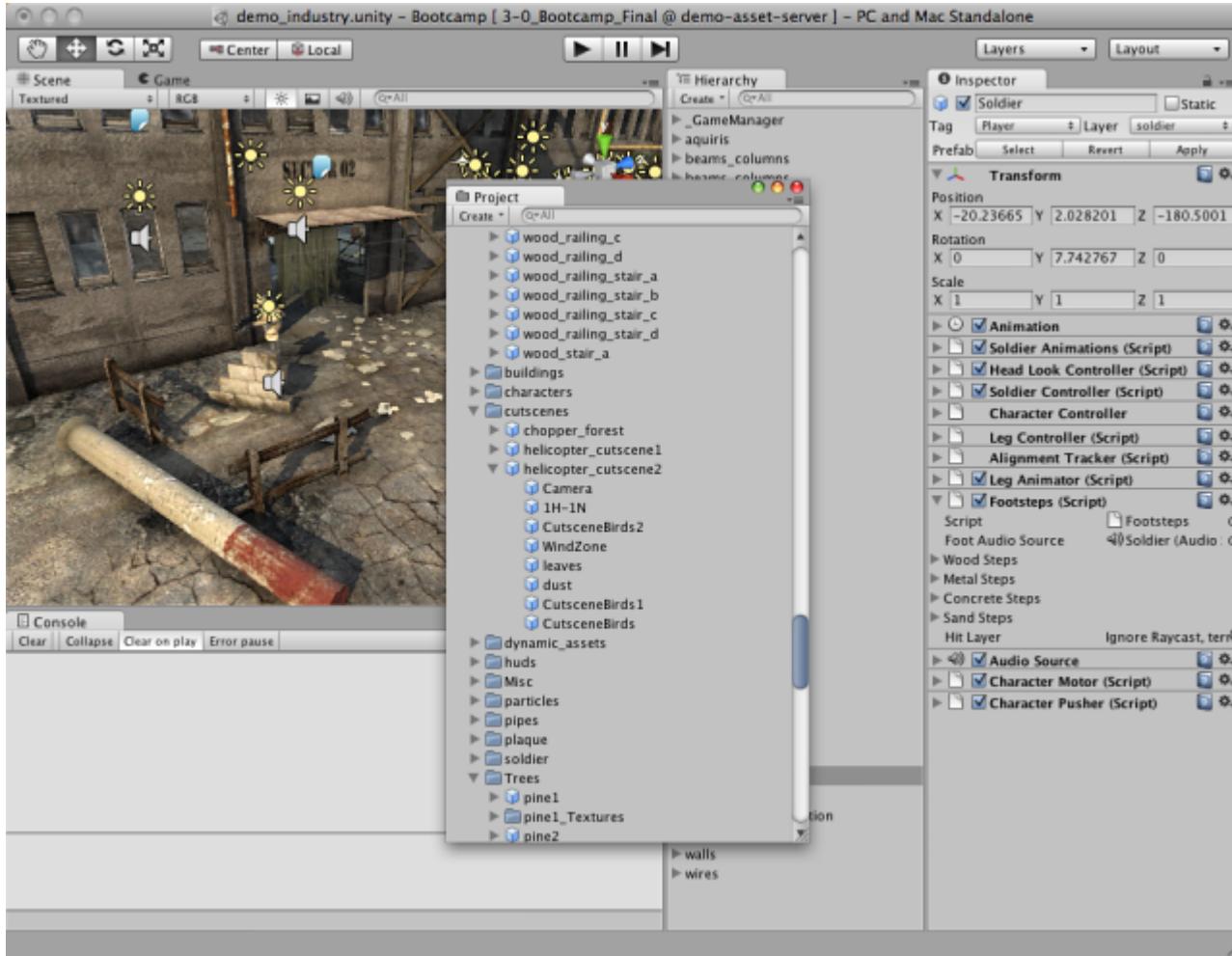
Le bouton "Rétablir les paramètres d'usine ..." supprime toutes les dispositions personnalisées et restaure les dispositions par défaut (*marquées avec 2*) .

Personnalisation de votre espace de travail

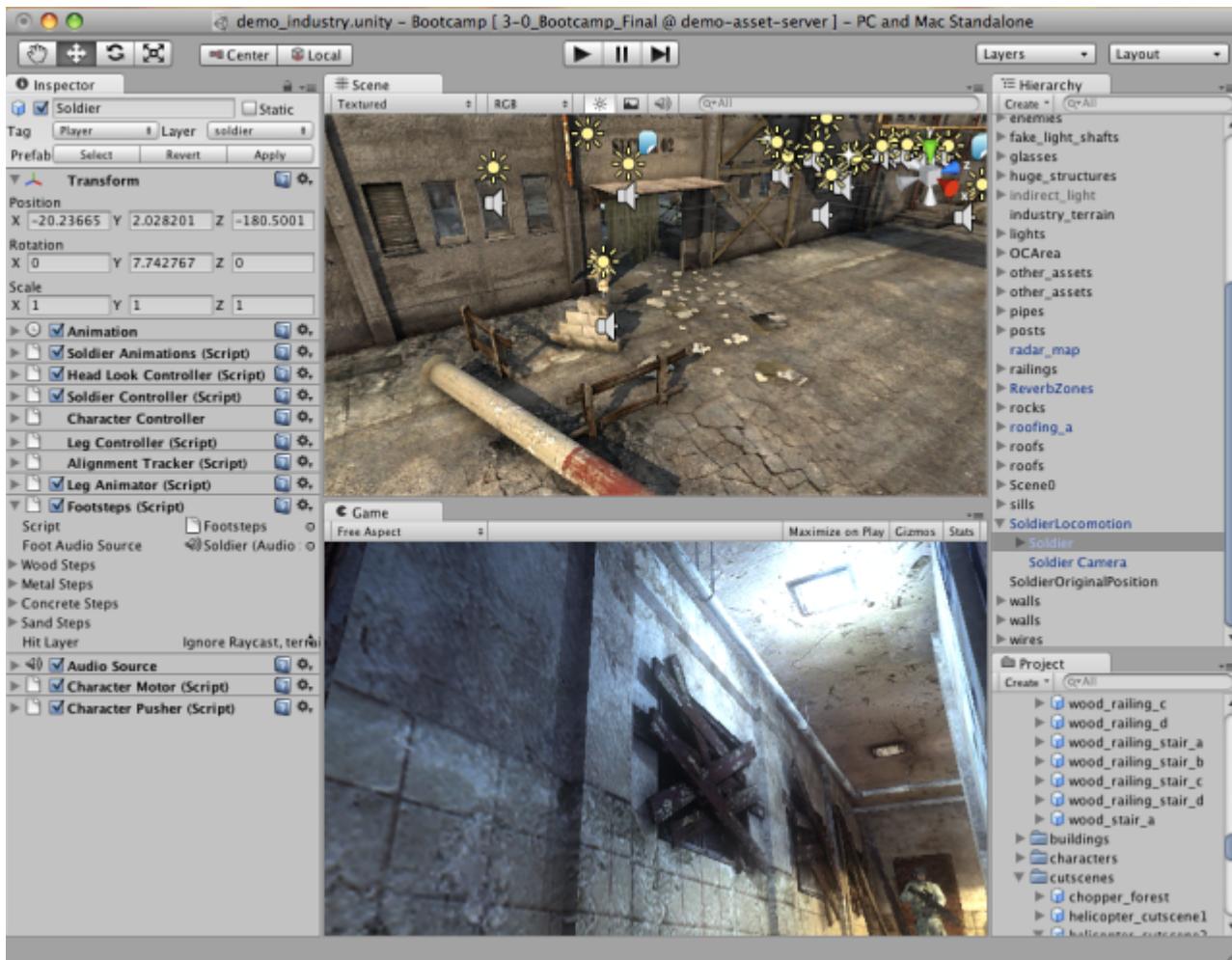
Vous pouvez personnaliser votre disposition de vues en cliquant-glissant sur l'onglet de n'importe quelle vue vers l'un de plusieurs emplacements. Supprimer un onglet dans la zone d'onglet d'une fenêtre existante ajoutera l'onglet à côté des onglets existants. Vous pouvez également déposer une tabulation dans une zone de quai pour ajouter la vue dans une nouvelle fenêtre.



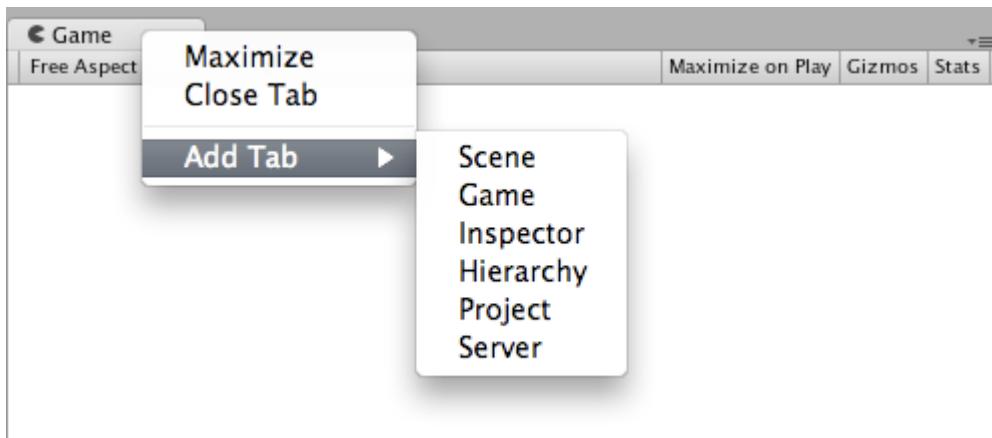
Les onglets peuvent également être détachés de la fenêtre de l'éditeur principal et disposés dans leurs propres fenêtres d'édition flottantes. Les fenêtres flottantes peuvent contenir des arrangements de vues et d'onglets, tout comme la fenêtre de l'éditeur principal.



Lorsque vous avez créé une mise en page d'éditeur, vous pouvez enregistrer la mise en page et la restaurer à tout moment. [Reportez-vous à cet exemple pour les dispositions de l'éditeur](#).



À tout moment, vous pouvez cliquer avec le bouton droit de la souris sur l'onglet de n'importe quelle vue pour afficher des options supplémentaires telles que Agrandir ou ajouter un nouvel onglet dans la même fenêtre.



Lire Démarrer avec unity3d en ligne: <https://riptutorial.com/fr/unity3d/topic/846/demarrer-avec-unity3d>

Chapitre 2: API CullingGroup

Remarques

Comme l'utilisation de CullingGroups n'est pas toujours simple, il peut être utile d'encapsuler l'essentiel de la logique derrière une classe de gestionnaire.

Vous trouverez ci-dessous un schéma de fonctionnement d'un tel gestionnaire.

```
using UnityEngine;
using System;
public interface ICullingGroupManager
{
    int ReserveSphere();
    void ReleaseSphere(int sphereIndex);
    void SetPosition(int sphereIndex, Vector3 position);
    void SetRadius(int sphereIndex, float radius);
    void SetCullingEvent(int sphereIndex, Action<CullingGroupEvent> sphere);
}
```

L'essentiel est que vous réserviez une sphère d'abattage du gestionnaire qui renvoie l'index de la sphère réservée. Vous utilisez ensuite l'index donné pour manipuler votre sphère réservée.

Exemples

Culling object distances

L'exemple suivant montre comment utiliser CullingGroups pour obtenir des notifications en fonction du point de référence de distance.

Ce script a été simplifié pour plus de brièveté et utilise plusieurs méthodes très performantes.

```
using UnityEngine;
using System.Linq;

public class CullingGroupBehaviour : MonoBehaviour
{
    CullingGroup localCullingGroup;

    MeshRenderer[] meshRenderers;
    Transform[] meshTransforms;
    BoundingSphere[] cullingPoints;

    void OnEnable()
    {
        localCullingGroup = new CullingGroup();

        meshRenderers = FindObjectsOfType<MeshRenderer>()
            .Where((MeshRenderer m) => m.gameObject != this.gameObject)
            .ToArray();
    }
}
```

```

cullingPoints = new BoundingSphere[meshRenderers.Length];
meshTransforms = new Transform[meshRenderers.Length];

for (var i = 0; i < meshRenderers.Length; i++)
{
    meshTransforms[i] = meshRenderers[i].GetComponent<Transform>();
    cullingPoints[i].position = meshTransforms[i].position;
    cullingPoints[i].radius = 4f;
}

localCullingGroup.onStateChanged = CullingEvent;
localCullingGroup.SetBoundingSpheres(cullingPoints);
localCullingGroup.SetBoundingDistances(new float[] { 0f, 5f });
localCullingGroup.SetDistanceReferencePoint(GetComponent<Transform>().position);
localCullingGroup.targetCamera = Camera.main;
}

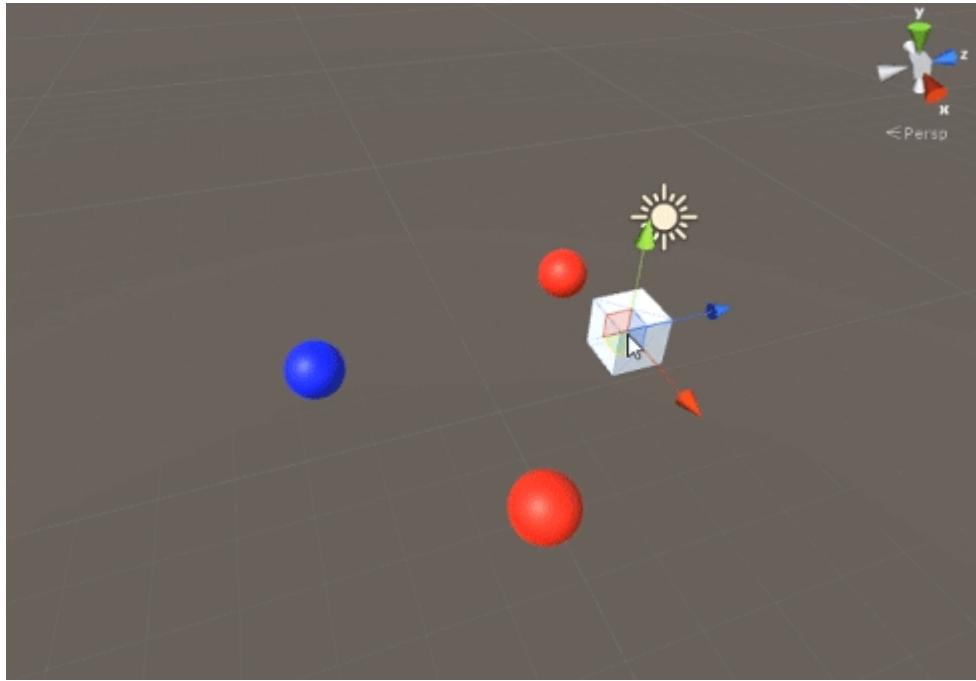
void FixedUpdate()
{
    localCullingGroup.SetDistanceReferencePoint(GetComponent<Transform>().position);
    for (var i = 0; i < meshTransforms.Length; i++)
    {
        cullingPoints[i].position = meshTransforms[i].position;
    }
}

void CullingEvent(CullingGroupEvent sphere)
{
    Color newColor = Color.red;
    if (sphere.currentDistance == 1) newColor = Color.blue;
    if (sphere.currentDistance == 2) newColor = Color.white;
    meshRenderers[sphere.index].material.color = newColor;
}

void OnDisable()
{
    localCullingGroup.Dispose();
}
}

```

Ajoutez le script à un GameObject (dans ce cas un cube) et appuyez sur Play. Tous les autres GameObject en scène changent de couleur en fonction de leur distance par rapport au point de référence.



Suppression de la visibilité des objets

Le script suivant illustre comment recevoir des événements en fonction de la visibilité sur une caméra définie.

Ce script utilise plusieurs méthodes performantes pour des raisons de brièveté.

```
using UnityEngine;
using System.Linq;

public class CullingGroupCameraBehaviour : MonoBehaviour
{
    CullingGroup localCullingGroup;

    MeshRenderer[] meshRenderers;

    void OnEnable()
    {
        localCullingGroup = new CullingGroup();

        meshRenderers = FindObjectsOfType<MeshRenderer>()
            .Where((MeshRenderer m) => m.gameObject != this.gameObject)
            .ToArray();

        BoundingSphere[] cullingPoints = new BoundingSphere[meshRenderers.Length];
        Transform[] meshTransforms = new Transform[meshRenderers.Length];

        for (var i = 0; i < meshRenderers.Length; i++)
        {
            meshTransforms[i] = meshRenderers[i].GetComponent<Transform>();
            cullingPoints[i].position = meshTransforms[i].position;
            cullingPoints[i].radius = 4f;
        }

        localCullingGroup.onStateChanged = CullingEvent;
        localCullingGroup.SetBoundingSpheres(cullingPoints);
        localCullingGroup.targetCamera = Camera.main;
```

```

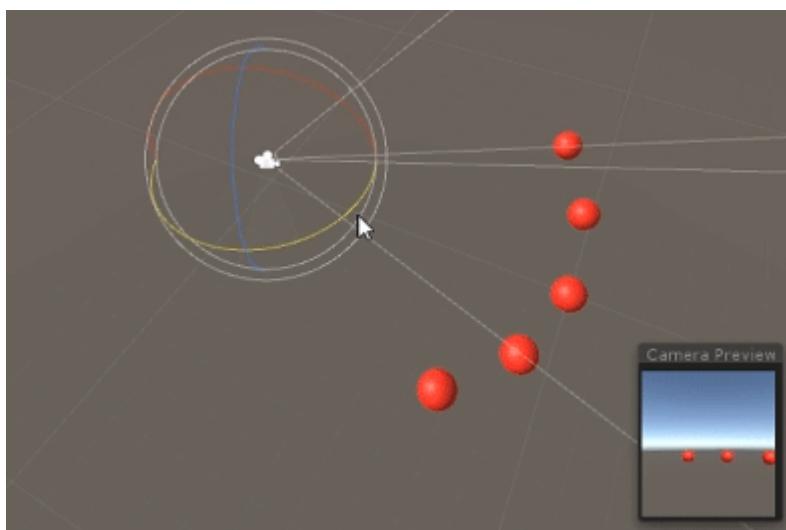
}

void CullingEvent (CullingGroupEvent sphere)
{
    meshRenderers[sphere.index].material.color = sphere.isVisible ? Color.red :
Color.white;
}

void OnDisable()
{
    localCullingGroup.Dispose();
}
}

```

Ajoutez le script à la scène et appuyez sur Play. Toute la géométrie en scène changera de couleur en fonction de leur visibilité.



Un effet similaire peut être obtenu à l'aide de la méthode `MonoBehaviour.OnBecameVisible()` si l'objet possède un composant `MeshRenderer`. Utilisez `CullingGroups` lorsque vous devez `Vector3` coordonnées `GameObjects`, `Vector3` vides ou une méthode centralisée de suivi des visibilités d'objet.

Distances limites

Vous pouvez ajouter des distances limites au-dessus du rayon du point de réforme. Ils sont en quelque sorte des conditions de déclenchement supplémentaires en dehors du rayon principal des points d'abattage, comme "fermer", "loin" ou "très loin".

```
cullingGroup.SetBoundingDistances (new float[] { 0f, 10f, 100f});
```

Les distances limites affectent uniquement l'utilisation d'un point de référence de distance. Ils n'ont aucun effet lors de l'abattage de l'appareil photo.

Visualiser les distances limites

Ce qui peut causer une confusion initiale est la façon dont les distances limites sont ajoutées au-dessus des rayons de la sphère.

Tout d'abord, le groupe de sélection calcule l' *aire* de la sphère de délimitation et de la distance limite. Les deux zones sont additionnées et le résultat est la zone de déclenchement de la bande de distance. Le rayon de cette zone peut être utilisé pour visualiser le champ d'effet de la distance limite.

```
float cullingPointArea = Mathf.PI * (cullingPointRadius * cullingPointRadius);  
float boundingArea = Mathf.PI * (boundingDistance * boundingDistance);  
float combinedRadius = Mathf.Sqrt((cullingPointArea + boundingArea) / Mathf.PI);
```

Lire API CullingGroup en ligne: <https://riptutorial.com/fr/unity3d/topic/4574/api-cullinggroup>

Chapitre 3: Asset Store

Examples

Accéder à la banque de biens

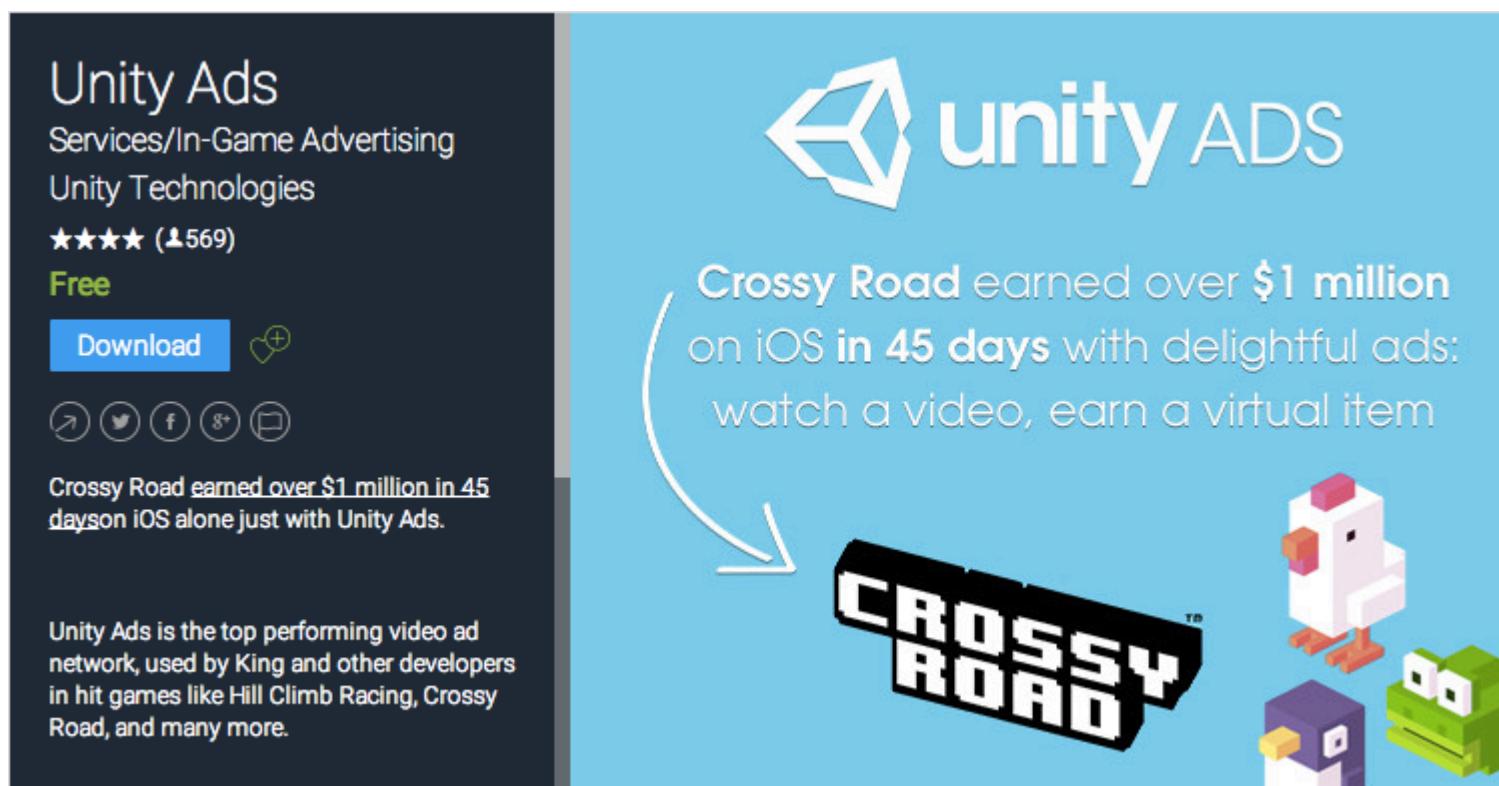
Vous pouvez accéder à Unity Asset Store de trois manières:

- Ouvrez la fenêtre Asset Store en sélectionnant Window → Asset Store dans le menu principal d'Unity.
- Utilisez la touche de raccourci (Ctrl + 9 sous Windows / 9 sous Mac OS)
- Parcourir l'interface Web: <https://www.assetstore.unity3d.com/>

Vous pouvez être invité à créer un compte utilisateur gratuit ou à vous connecter si vous accédez à Unity Asset Store pour la première fois.

Achats d'actifs

Après avoir accédé à la base de données des actifs et avoir consulté la ressource que vous souhaitez télécharger, cliquez simplement sur le bouton **Télécharger**. Le texte du bouton peut également être **Acheter maintenant** si l'actif a un coût associé.



Si vous visualisez Unity Asset Store via l'interface Web, le texte du bouton **Télécharger** peut s'afficher sous la forme **Open in Unity**. La sélection de ce bouton lance une instance de Unity et affiche l'actif dans la fenêtre Asset Store .

Vous pouvez être invité à créer un compte utilisateur gratuit ou à vous connecter si vous effectuez

votre premier achat auprès de Unity Asset Store.

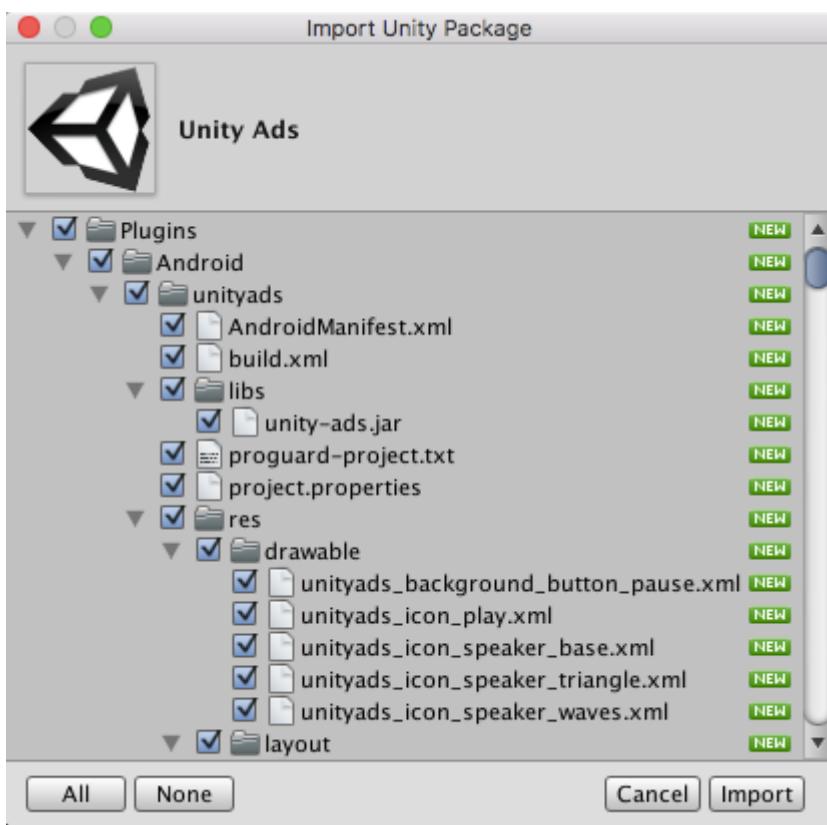
Unity procédera alors à l'acceptation de votre paiement, le cas échéant.

Importation de biens

Une fois le fichier téléchargé dans Unity, le bouton **Télécharger** ou **Acheter** est remplacé par **Importer**.

La sélection de cette option invitera l'utilisateur à accéder à la fenêtre *Importer Unity Package*, dans laquelle il pourra sélectionner les fichiers d'actifs qu'il souhaite importer dans son projet.

Sélectionnez **Importer** pour confirmer le processus, en plaçant les fichiers d'actifs sélectionnés dans le dossier Ressources affiché dans la fenêtre *Vue du projet*.



Actifs de publication

1. créer un compte éditeur
2. ajouter un actif dans le compte de l'éditeur
3. télécharger les outils de magasin de ressources (à partir du magasin de ressources)
4. allez dans "Outils Asset Store">> "Téléchargement de paquet"
5. sélectionnez le dossier correct du package et du projet dans la fenêtre des outils du magasin de ressources
6. cliquez sur télécharger
7. soumettre votre bien en ligne

TODO - ajouter des photos, plus de détails

Confirmer le numéro de facture d'un achat

Le numéro de facture est utilisé pour vérifier la vente pour les éditeurs. De nombreux éditeurs d'actifs ou de plug-ins payants demandent le numéro de facture sur demande d'assistance. Le numéro de facture est également utilisé comme clé de licence pour activer un actif ou un plug-in.

Le numéro de facture se trouve à deux endroits:

1. Après avoir acheté l'actif, vous recevrez un email dont le sujet est "Confirmation d'achat Unity Asset Store ...". Le numéro de facture est dans la pièce jointe PDF de cet email.



UNITY3D.COM

Unity Technologies ApS
Vendersgade 28
1363 København K
Danmark

INVOICE

Invoice No.	[REDACTED]
Date	[REDACTED]
Due Date	[REDACTED]
Order No.	[REDACTED]

2. Ouvrez <https://www.assetstore.unity3d.com/#!/account/transactions>, puis vous trouverez le numéro de facture dans la colonne *Description*.

Credit Card / PayPal		
Date	Action	Description
[REDACTED]	CREDIT CARD / PAYPAL	#30[REDACTED]80 Mesh Terrain Editor Pro

Lire Asset Store en ligne: <https://riptutorial.com/fr/unity3d/topic/5705/asset-store>

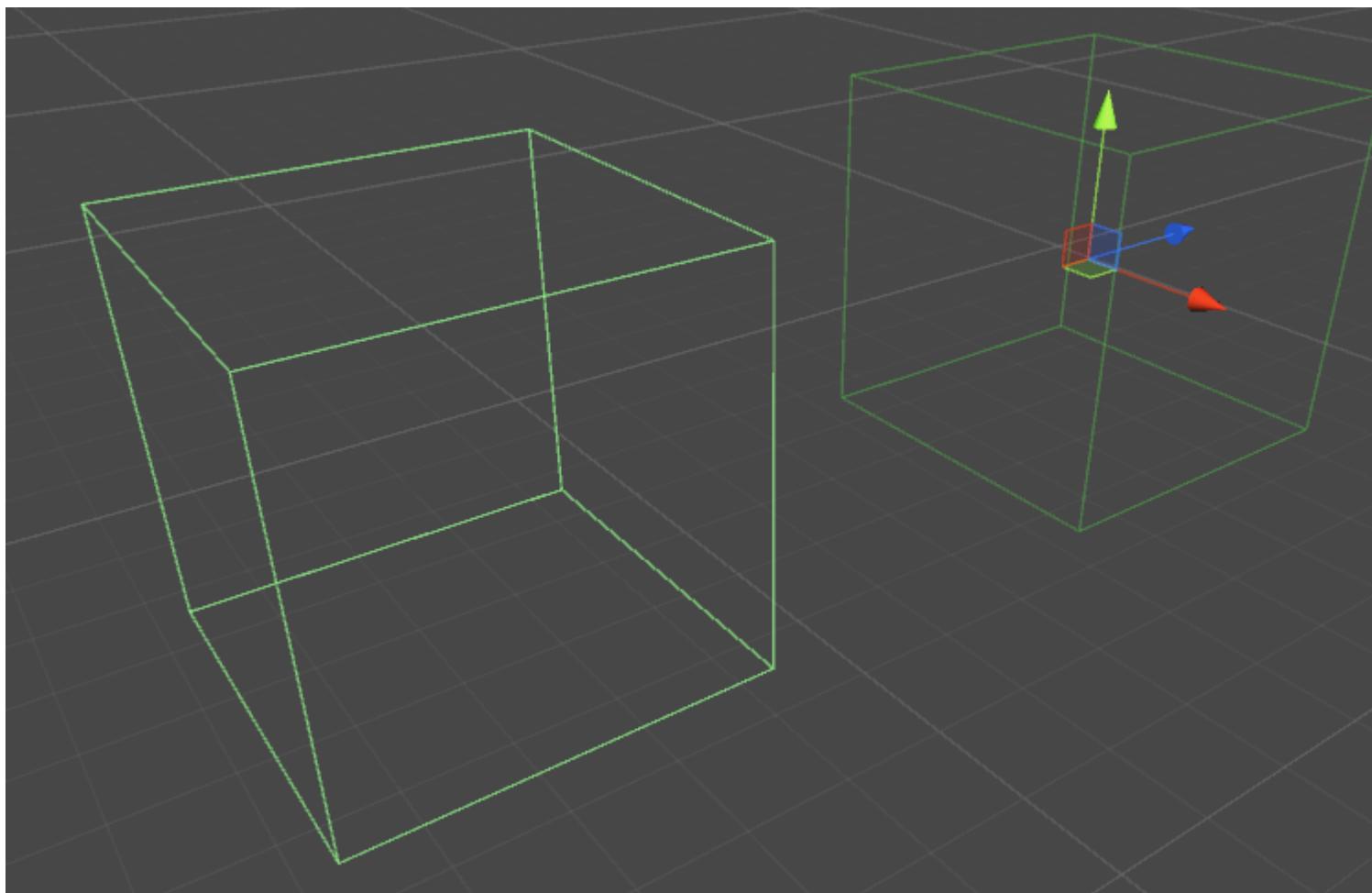
Chapitre 4: Collision

Examples

Les collisionneurs

Box Collider

Un Collider primitif en forme de cuboïde.



Propriétés

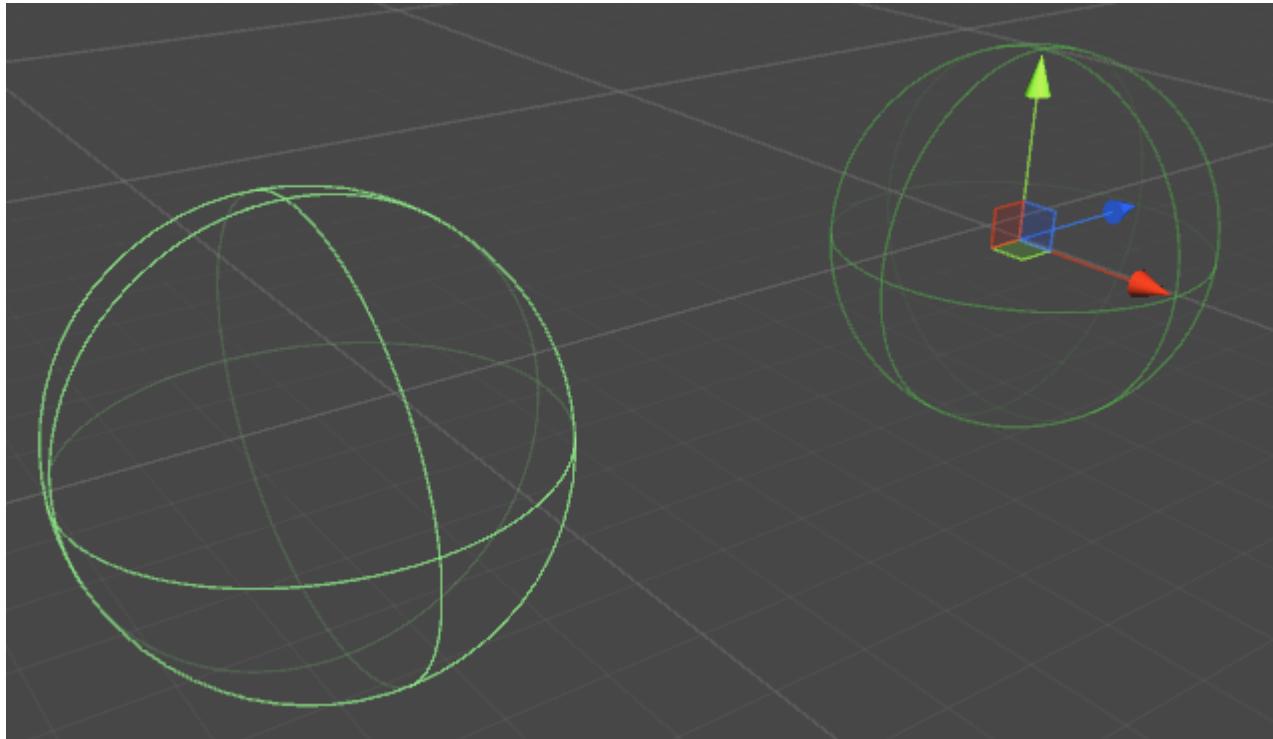
- **Is Trigger** - Si coché, le Box Collider ignore la physique et devient un collisionneur de déclenchement
- **Material** - Une référence, si spécifiée, au matériau physique du Box Collider
- **Centre** - La position centrale du Box Collider dans l'espace local
- **Size** - La taille du Box Collider mesurée dans l'espace local

Exemple

```
// Add a Box Collider to the current GameObject.  
BoxCollider myBC = BoxCollider)myGameObject.gameObject.AddComponent(typeof(BoxCollider));  
  
// Make the Box Collider into a Trigger Collider.  
myBC.isTrigger= true;  
  
// Set the center of the Box Collider to the center of the GameObject.  
myBC.center = Vector3.zero;  
  
// Make the Box Collider twice as large.  
myBC.size = 2;
```

Sphère Collider

Un Collider primitif en forme de sphère.



Propriétés

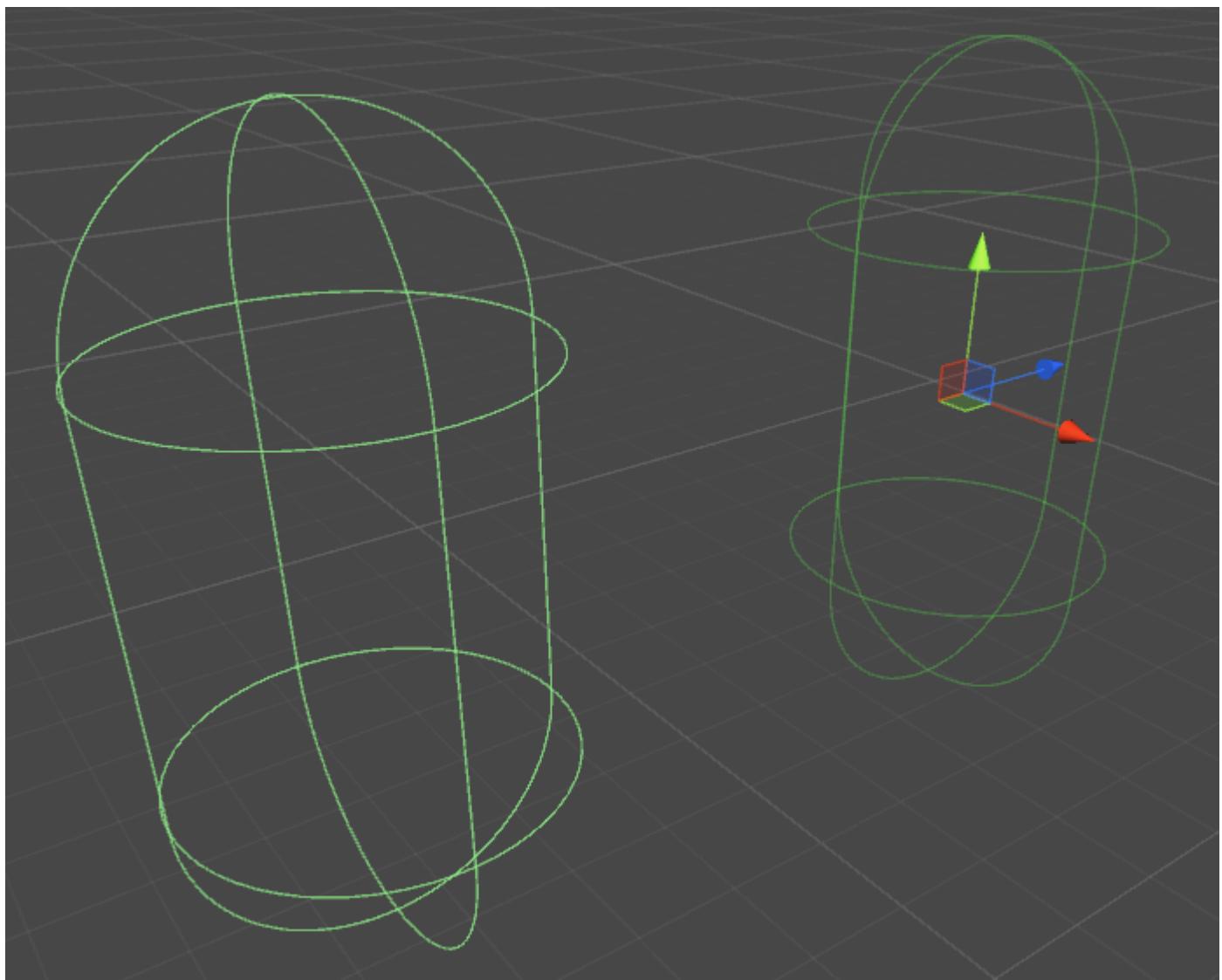
- **Is Trigger** - Si coché, le collisionneur Sphère ignorera la physique et deviendra un collisionneur de déclenchement
- **Material** - Une référence, si spécifiée, au matériau physique du collisionneur Sphère
- **Centre** - La position centrale du collisionneur Sphère dans l'espace local
- **Rayon** - Le rayon du collisionneur

Exemple

```
// Add a Sphere Collider to the current GameObject.  
SphereCollider mySC =  
SphereCollider)myGameObject.gameObject.AddComponent(typeof(SphereCollider));  
  
// Make the Sphere Collider into a Trigger Collider.  
mySC.isTrigger= true;  
  
// Set the center of the Sphere Collider to the center of the GameObject.  
mySC.center = Vector3.zero;  
  
// Make the Sphere Collider twice as large.  
mySC.radius = 2;
```

Collisionneur de capsules

Deux demi-sphères reliées par un cylindre.



Propriétés

- **Is Trigger** - Si coché, le collisionneur de capsule ignorera la physique et deviendra un collisionneur de déclenchement
- **Material** - Une référence, si spécifiée, au matériau physique du collisionneur de capsule
- **Centre** - La position centrale du collisionneur de capsules dans l'espace local
- **Rayon** - Le rayon dans l'espace local
- **Hauteur** - Hauteur totale du collisionneur
- **Direction** - L'axe d'orientation dans l'espace local

Exemple

```
// Add a Capsule Collider to the current GameObject.  
CapsuleCollider myCC =  
CapsuleCollider)myGameObject.gameObject.AddComponent(typeof(CapsuleCollider));  
  
// Make the Capsule Collider into a Trigger Collider.  
myCC.isTrigger= true;  
  
// Set the center of the Capsule Collider to the center of the GameObject.  
myCC.center = Vector3.zero;  
  
// Make the Sphere Collider twice as tall.  
myCC.height= 2;  
  
// Make the Sphere Collider twice as wide.  
myCC.radius= 2;  
  
// Set the axis of lengthwise orientation to the X axis.  
myCC.direction = 0;  
  
// Set the axis of lengthwise orientation to the Y axis.  
myCC.direction = 1;  
  
// Set the axis of lengthwise orientation to the Y axis.  
myCC.direction = 2;
```

Collisionneur de roue

Propriétés

- **Mass** - La masse du collisionneur de roue
- **Rayon** - Le rayon dans l'espace local

- **Taux d'amortissement des roues** - Valeur d'amortissement pour le collisionneur de roue
- **Distance de suspension** - Extension maximale le long de l'axe Y dans l'espace local
- **Force app distance de point** - Le point où les forces seront appliquées,
- **Centre** - Centre du collisionneur de roue dans l'espace local

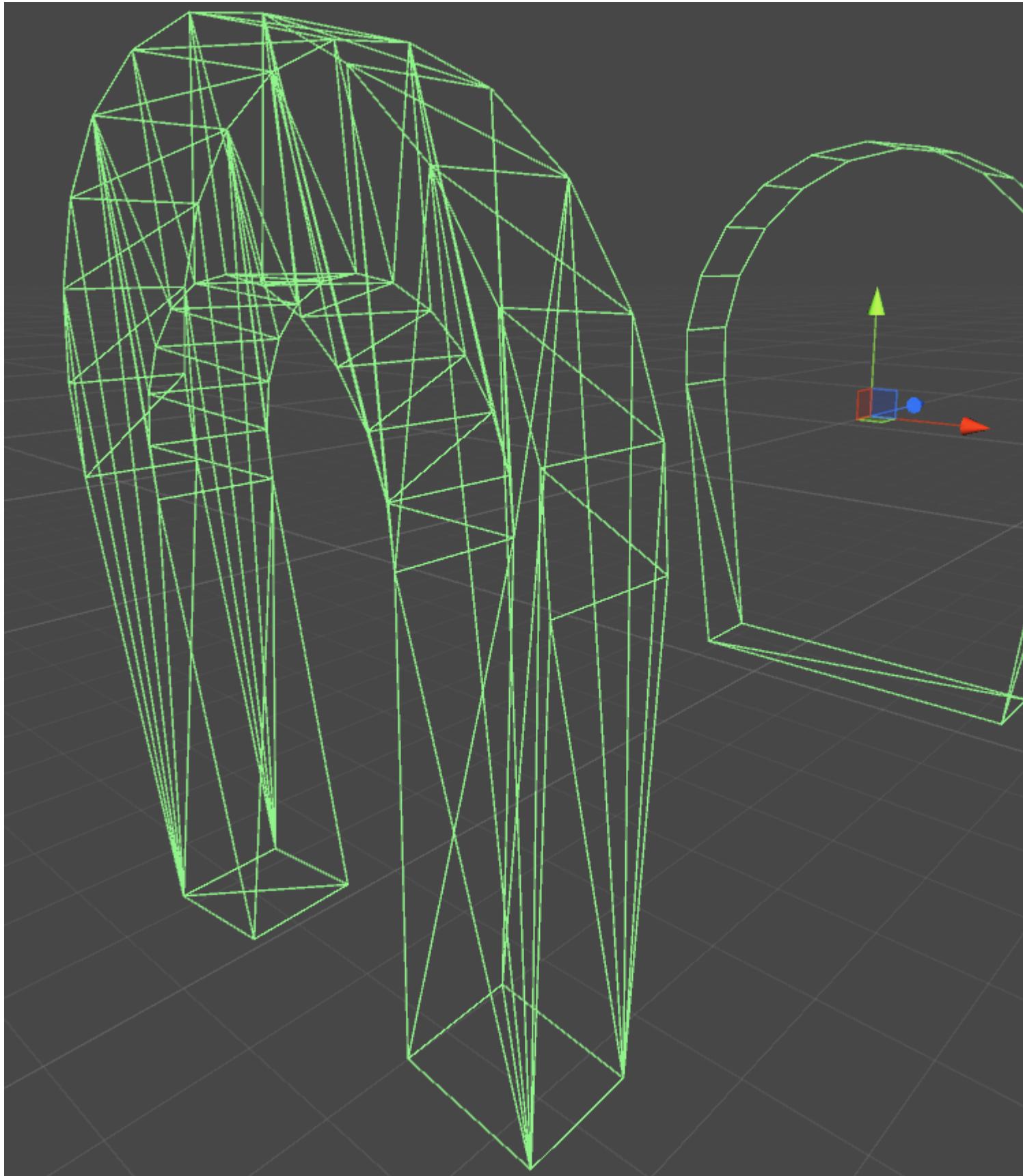
Ressort de suspension

- **Spring** - la vitesse à laquelle la roue tente de revenir à la position cible
- **Amortisseur** - Une valeur plus grande amortit davantage la vitesse et la suspension se déplace plus lentement
- **Position cible** - la valeur par défaut est 0,5, à 0 la suspension est expirée, à 1 elle est en extension complète
- **Frottement avant / latéral** - comment le pneu se comporte lorsqu'il roule en avant ou sur le côté

Exemple

Collisionneur de maille

Un collisionneur basé sur un actif maillé.



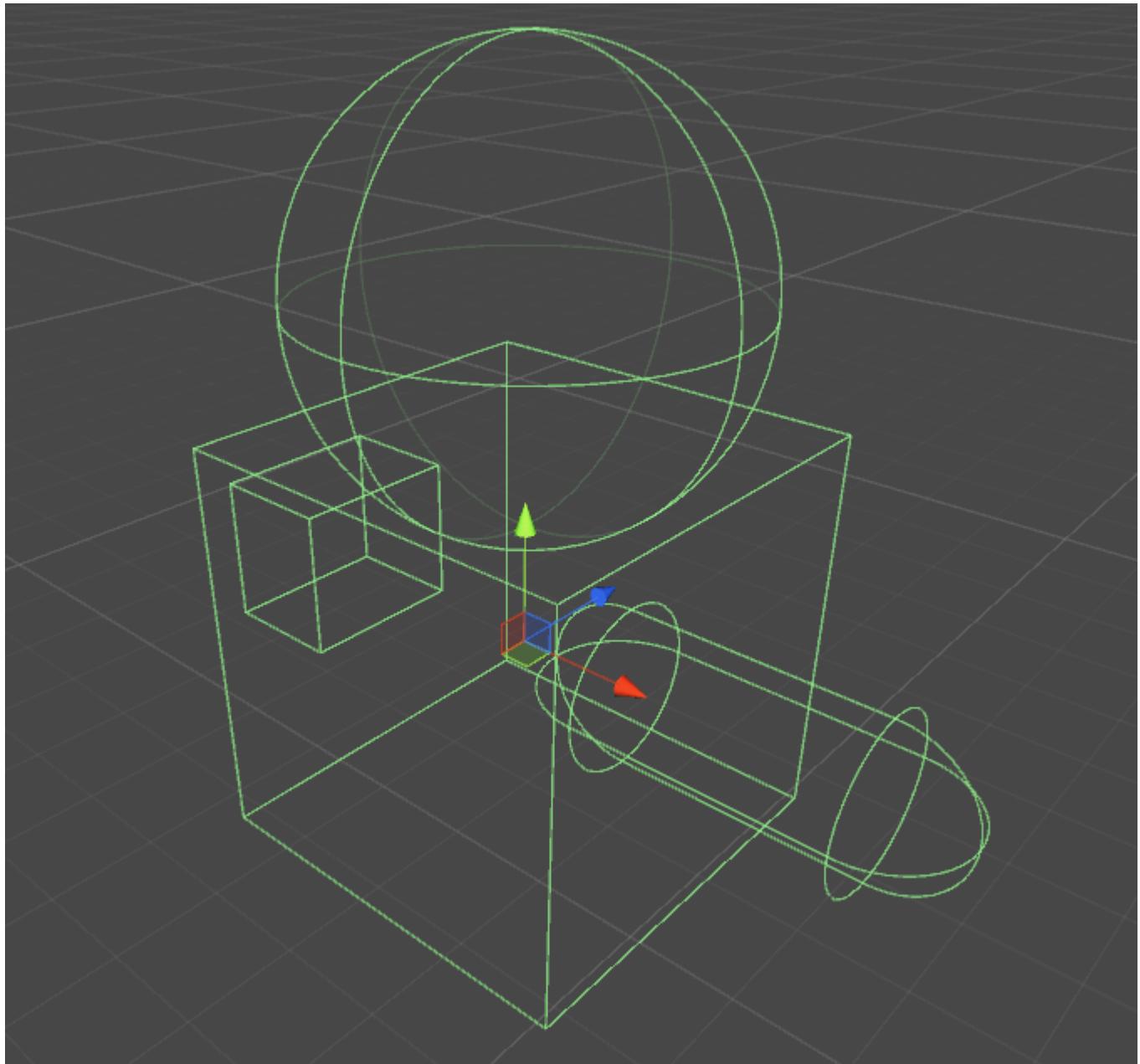
Propriétés

- **Is Trigger** - Si coché, le Box Collider ignore la physique et devient un collisionneur de déclenchement

- **Material** - Une référence, si spécifiée, au matériau physique du Box Collider
- **Mesh** - Une référence au maillage sur lequel le Collider est basé
- **Convexes** - Les collisionneurs de maillage convexes sont limités à 255 polygones - si activé, ce collisionneur peut entrer en collision avec d'autres collisionneurs de maillage

Exemple

Si vous appliquez plusieurs collisionneurs à un objet GameObject, nous l'appelons un collisionneur composé.



Collisionneur de roue

Le collisionneur de roues à l'intérieur de l'unité est construit sur le collisionneur de roues PhysX de Nvidia et partage donc de nombreuses propriétés similaires. Techniquement, l'unité est un

programme "sans unité", mais pour que tout soit logique, certaines unités standard sont nécessaires.

Propriétés de base

- Masse - le poids de la roue en kilogrammes, il est utilisé pour le moment de la roue et le moment de la rotation.
- Rayon - en mètres, le rayon du collisionneur.
- Wheel Damping Rate (Taux d'amortissement de la roue) - Règle la sensibilité des roues au couple appliqué.
- Distance de suspension - Distance totale en mètres parcourue par la roue
- Force App Point Distance - Où est la force de la suspension appliquée au corps rigide parent
- Centre - La position centrale de la roue

Paramètres de suspension

- Spring - C'est la constante de printemps, K, en newtons / mètre dans l'équation:

$$\text{Force} = \text{constante de printemps} * \text{distance}$$

Un bon point de départ pour cette valeur doit être la masse totale de votre véhicule, divisée par le nombre de roues, multipliée par un nombre compris entre 50 et 100. Par exemple, si vous avez une voiture de 2 000 kg à 4 roues, chaque roue devra soutenir 500 kg. Multipliez cela par 75, et votre constante de printemps devrait être 37 500 Newtons / mètre.

- Amortisseur - l'équivalent d'un amortisseur dans une voiture. Des taux plus élevés rendent le suspense "plus rigide" et les taux inférieurs le rendent "plus doux" et plus susceptible d'osciller.

Je ne connais pas les unités ou l'équation pour cela, mais je pense que cela concerne une équation de fréquence en physique.

Paramètres de frottement latéral

La courbe de frottement à l'unité a une valeur de glissement déterminée par le glissement de la roue (en m / s) par rapport à la position souhaitée par rapport à la position réelle.

- Extremum Slip - C'est la quantité maximale (en m / s) qu'une roue peut glisser avant de perdre de la traction
- Valeur extrême - C'est la quantité maximale de friction à appliquer à une roue.

Les valeurs pour Extremum Slip doivent être comprises entre 0,2 et 2 m / s pour la plupart des voitures现实的. 2 m / s est d'environ 6 pieds par seconde ou 5 mph, ce qui est beaucoup de glissement. Si vous estimatez que votre véhicule doit avoir une valeur supérieure à 2 m / s pour le glissement, vous devriez envisager d'augmenter la friction maximale (valeur extrême).

Max Fraction (Extremum Value) est le coefficient de frottement dans l'équation:

$$\text{Force de frottement (en newtons)} = \text{coefficient de frottement} * \text{force descendante (en newtons)}$$

Cela signifie qu'avec un coefficient de 1, vous appliquez toute la force de la voiture + la suspension opposée à la direction du glissement. Dans les applications du monde réel, les valeurs supérieures à 1 sont rares, mais pas impossibles. Pour un pneu sur asphalte sec, les valeurs comprises entre 0,7 et 0,9 sont réalistes, donc la valeur par défaut de 1,0 est préférable.

Cette valeur ne doit pas dépasser, de manière réaliste, 2,5, car un comportement étrange se produira. Par exemple, vous commencez à tourner à droite, mais parce que cette valeur est si élevée, une force importante est appliquée en face de votre direction, et vous commencez à glisser dans le virage au lieu de partir.

Si vous avez maximisé les deux valeurs, vous devriez alors commencer à augmenter le glissement asymptote et la valeur. Asymptote Slip doit être compris entre 0,5 et 2 m / s et définit le coefficient de frottement pour toute valeur de glissement au-delà du glissement Asymptote. Si vous trouvez que votre véhicule se comporte bien jusqu'à ce qu'il se casse, à quel point il agit comme s'il était sur la glace, vous devriez augmenter la valeur Asymptote. Si vous trouvez que votre véhicule ne peut pas dériver, vous devez baisser la valeur.

Friction Forward

Le frottement vers l'avant est identique au frottement latéral, sauf que cela définit la force de traction de la roue dans le sens du mouvement. Si les valeurs sont trop basses, vos véhicules feront des épuisements et essaieront les pneus avant de progresser lentement. Si elle est trop élevée, votre véhicule peut avoir tendance à essayer de faire un retournement grave, voire pire.

Notes complémentaires

Ne vous attendez pas à pouvoir créer un clone GTA ou un autre clone de course en ajustant simplement ces valeurs. Dans la plupart des jeux de conduite, ces valeurs sont constamment modifiées dans le script pour différentes vitesses, terrains et valeurs de virage. De plus, si vous appliquez simplement un couple constant aux collisionneurs de roue lorsque vous appuyez sur une touche, votre jeu ne se comportera pas de manière réaliste. Dans la réalité, les voitures ont des courbes de couple et des transmissions pour modifier le couple appliqué aux roues.

Pour de meilleurs résultats, vous devez régler ces valeurs jusqu'à ce que vous obteniez une réponse satisfaisante de la part de la voiture, puis modifier le couple de la roue, l'angle de rotation maximal et les valeurs de frottement dans le script.

Vous trouverez plus d'informations sur les collisionneurs de roues dans la documentation de Nvidia:

<http://docs.nvidia.com/gameworks/content/gameworkslibrary/physx/guide/Manual/Vehicles.html>

Déclencheurs collisionneurs

Les méthodes

- OnTriggerEnter()
- OnTriggerStay()
- OnTriggerExit()

Vous pouvez créer un collisionneur dans un **déclencheur** afin d'utiliser les `OnTriggerEnter()`, `OnTriggerStay()` et `OnTriggerExit()`. Un collisionneur de déclenchement ne réagira pas physiquement aux collisions, les autres GameObjects le traversent simplement. Ils sont utiles pour détecter lorsqu'un autre objet GameObject se trouve dans une zone donnée ou non, par exemple, lors de la collecte d'un élément, nous pouvons être en mesure de le parcourir, mais détecter quand cela se produit.

Script de déclenchement du collisionneur

Exemple

La méthode ci-dessous est un exemple d'écouteur de déclenchement qui détecte lorsqu'un autre collisionneur entre dans le collisionneur d'un objet GameObject (tel qu'un lecteur). Les méthodes de déclenchement peuvent être ajoutées à tout script affecté à un objet GameObject.

```
void OnTriggerEnter(Collider other)
{
    //Check collider for specific properties (Such as tag=item or has component=item)
}
```

Lire Collision en ligne: <https://riptutorial.com/fr/unity3d/topic/4405/collision>

Chapitre 5: Comment utiliser les packages d'actifs

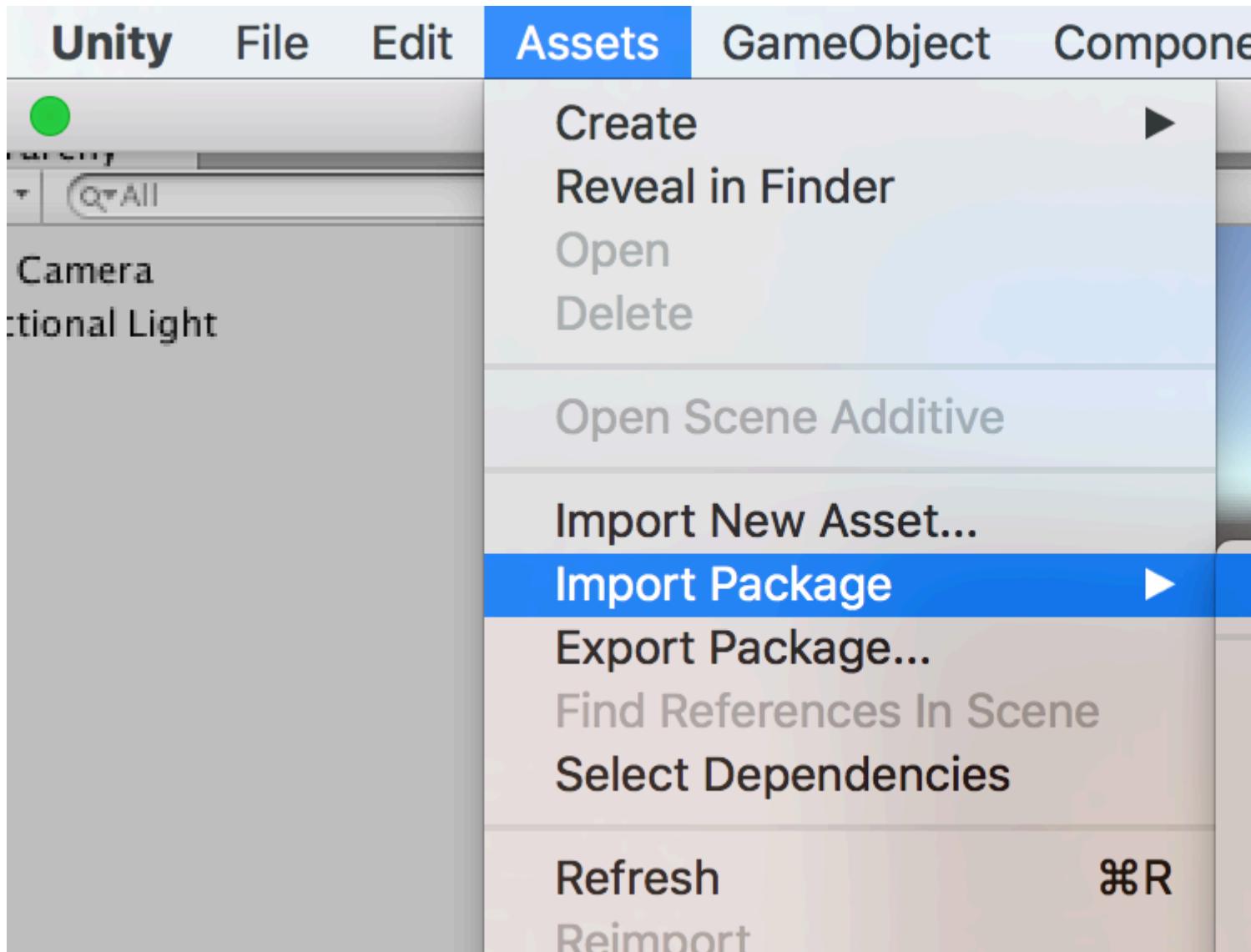
Exemples

Paquets d'actifs

Les packages d'actifs (avec le format de fichier de `.unitypackage`) sont un moyen couramment utilisé pour distribuer des projets Unity à d'autres utilisateurs. Lorsque vous travaillez avec des périphériques disposant de leurs propres SDK (par exemple, [Oculus](#)), vous pouvez être invité à télécharger et à importer l'un de ces packages.

Importer un paquet `.unity`

Pour importer un package, accédez à la barre de menus Unity et cliquez sur `Assets > Import Package > Custom Package...`, puis accédez au fichier `.unitypackage` dans le navigateur de fichiers qui apparaît.



Lire Comment utiliser les packages d'actifs en ligne:

<https://riptutorial.com/fr/unity3d/topic/4491/comment-utiliser-les-packages-d-actifs>

Chapitre 6: Communication avec le serveur

Examples

Obtenir

Get obtient des données du serveur Web. et `new WWW("https://urlexample.com")`; avec une url mais sans second paramètre fait un **Get**.

c'est à dire

```
using UnityEngine;
using System.Collections;

public class ExampleClass : MonoBehaviour
{
    public string url = "http://google.com";

    IEnumerator Start()
    {
        WWW www = new WWW(url); // One get.
        yield return www;
        Debug.Log(www.text); // The data of the url.
    }
}
```

Simple Post (Post Fields)

Chaque instance de **WWW** avec un deuxième paramètre est une *publication*.

Voici un exemple pour poster *un identifiant* et un *mot de passe* sur le serveur.

```
void Login(string id, string pwd)
{
    WWWForm dataParameters = new WWWForm();      // Create a new form.
    dataParameters.AddField("username", id);
    dataParameters.AddField("password", pwd);     // Add fields.
    WWW www = new WWW(url+"/account/login", dataParameters);
    StartCoroutine("PostdataEnumerator", www);
}

IEnumerator PostdataEnumerator(WWW www)
{
    yield return www;
    if (!string.IsNullOrEmpty(www.error))
    {
        Debug.Log(www.error);
    }
    else
    {
        Debug.Log("Data Submitted");
    }
}
```

Poster (Télécharger un fichier)

Télécharger un fichier sur le serveur est également un message. Vous pouvez facilement télécharger un fichier via **WWW** , comme ci-dessous:

Télécharger un fichier zip sur le serveur

```
string mainUrl = "http://server/upload/";
string saveLocation;

void Start()
{
    saveLocation = "ftp:///home/xxx/x.zip"; // The file path.
    StartCoroutine(PrepareFile());
}

// Prepare The File.
IEnumerator PrepareFile()
{
    Debug.Log("saveLoacation = " + saveLocation);

    // Read the zip file.
    WWW loadTheZip = new WWW(saveLocation);

    yield return loadTheZip;

    PrepareStepTwo(loadTheZip);
}

void PrepareStepTwo(WWW post)
{
    StartCoroutine(UploadTheZip(post));
}

// Upload.
IEnumerator UploadTheZip(WWW post)
{
    // Create a form.
    WWWForm form = new WWWForm();

    // Add the file.
    form.AddBinaryData("myTestFile.zip", post.bytes, "myFile.zip", "application/zip");

    // Send POST request.
    string url = mainUrl;
    WWW POSTZIP = new WWW(url, form);

    Debug.Log("Sending zip...");
    yield return POSTZIP;
    Debug.Log("Zip sent!");
}
```

Dans cet exemple, la **coroutine est** utilisée pour préparer et télécharger le fichier. Si vous souhaitez en savoir plus sur les coroutines de Unity, visitez le site de [Coroutines](#) .

Envoi d'une requête au serveur

Il y a plusieurs façons de communiquer avec les serveurs en utilisant Unity en tant que client (certaines méthodologies sont meilleures que d'autres selon votre objectif). Tout d'abord, il faut déterminer si le serveur doit pouvoir envoyer efficacement des opérations vers et depuis le serveur. Pour cet exemple, nous allons envoyer quelques données à notre serveur pour être validées.

Très probablement, le programmeur aura configuré une sorte de gestionnaire sur son serveur pour recevoir les événements et répondre au client en conséquence - toutefois, cela est hors de la portée de cet exemple.

C #:

```
using System.Net;
using System.Text;

public class TestCommunicationWithServer
{
    public string SendDataToServer(string url, string username, string password)
    {
        WebClient client = new WebClient();

        // This specialized key-value pair will store the form data we're sending to the
server
        var loginData = new System.Collections.Specialized.NameValueCollection();
        loginData.Add("Username", username);
        loginData.Add("Password", password);

        // Upload client data and receive a response
        byte[] opBytes = client.UploadValues(ServerIpAddress, "POST", loginData);

        // Encode the response bytes into a proper string
        string opResponse = Encoding.UTF8.GetString(opBytes);

        return opResponse;
    }
}
```

La première chose à faire est de lancer des instructions qui nous permettent d'utiliser les classes WebClient et NameValueCollection.

Pour cet exemple, la fonction SendDataToServer prend en 3 paramètres de chaîne (facultatifs):

1. Url du serveur avec lequel nous communiquons
2. Première donnée
3. Deuxième donnée que nous envoyons au serveur

Le nom d'utilisateur et le mot de passe sont les données facultatives que j'envoie au serveur. Pour cet exemple, nous l'utilisons pour une validation ultérieure à partir d'une base de données ou de tout autre stockage externe.

Maintenant que nous avons configuré notre structure, nous allons instancier un nouveau WebClient pour envoyer nos données. Maintenant, nous devons charger nos données dans notre NameValueCollection et télécharger les données sur le serveur.

La fonction UploadValues prend également 3 paramètres nécessaires:

1. Adresse IP du serveur
2. Méthode HTTP
3. Les données que vous envoyez (le nom d'utilisateur et le mot de passe dans notre cas)

Cette fonction renvoie un tableau d'octets de la réponse du serveur. Nous devons encoder le tableau d'octets renvoyés en une chaîne appropriée pour pouvoir manipuler et disséquer la réponse.

On pourrait faire quelque chose comme ça:

```
if (opResponse.Equals(ReturnMessage.Success))  
{  
    Debug.Log("Unity client has successfully sent and validated data on server.");  
}
```

Maintenant, vous pourriez encore être confus, donc je suppose que je vais donner une brève explication sur la façon de gérer un serveur de réponse face.

Pour cet exemple, je vais utiliser PHP pour gérer la réponse du client. Je recommande d'utiliser PHP comme langage de script principal car il est très polyvalent, facile à utiliser et surtout rapide. Il existe certainement d'autres moyens de gérer une réponse sur un serveur, mais à mon avis, PHP est de loin l'implémentation la plus simple et la plus simple dans Unity.

PHP:

```
// Check to see if the unity client send the form data  
if(!isset($_REQUEST['Username']) || !isset($_REQUEST['Password']))  
{  
    echo "Empty";  
}  
else  
{  
    // Unity sent us the data - its here so do whatever you want  
  
    echo "Success";  
}
```

C'est donc la partie la plus importante - l'écho. Lorsque notre client télécharge les données sur le serveur, le client enregistre la réponse (ou la ressource) dans ce tableau d'octets. Une fois que le client a la réponse, vous savez que les données ont été validées et que vous pouvez passer au client une fois que cet événement est arrivé. Vous devez également réfléchir au type de données que vous envoyez (dans une certaine mesure) et à la manière de minimiser le montant réellement envoyé.

Il ne s'agit donc que d'une seule façon d'envoyer / recevoir des données de la part d'Unity. Selon votre projet, il existe d'autres moyens plus efficaces pour vous.

[Lire Communication avec le serveur en ligne:](#)

<https://riptutorial.com/fr/unity3d/topic/5578/communication-avec-le-serveur>

Chapitre 7: Coroutines

Syntaxe

- Coroutine StartCoroutine publique (routine IEnumerator);
- public Coroutine StartCoroutine (string methodName, valeur d'objet = null);
- StopCoroutine publique vide (string methodName);
- StopCoroutine publique annulée (routine IEnumerator);
- annulation publique StopAllCoroutines ();

Remarques

Considérations de performance

Il est préférable d'utiliser les coroutines avec modération, car la flexibilité entraîne un coût de performance.

- Coroutines en grand nombre exige plus de la part du processeur que les méthodes de mise à jour standard.
- Il y a un problème dans certaines versions d'Unity où les coroutines produisent des ordures à chaque cycle de mise à jour en raison de la boîte Unity `MoveNext` valeur de retour `MoveNext`. Cela a été observé pour la dernière fois en 5.4.0b13. ([Rapport de bogue](#))

Réduisez les erreurs en mettant en cache YieldInstructions

Une astuce courante pour réduire les déchets générés dans les coroutines consiste à mettre en cache l'`YieldInstruction`.

```
IEnumerator TickEverySecond()
{
    var wait = new WaitForSeconds(1f); // Cache
    while(true)
    {
        yield return wait; // Reuse
    }
}
```

Le fait de produire `null` ne produit pas de déchets supplémentaires.

Examples

Coroutines

Tout d'abord, il est essentiel de comprendre que les moteurs de jeu (tels que Unity) fonctionnent

selon un paradigme basé sur les images.

Le code est exécuté à chaque image.

Cela inclut le code propre à Unity et votre code.

Lorsque l'on pense aux cadres, il est important de comprendre qu'il n'y a **absolument** aucune garantie que les images se produisent. Ils **ne se** produisent **pas** sur un rythme régulier. Les écarts entre les images peuvent être, par exemple, 0,02632 puis 0,021167 puis 0,029778, etc. Dans l'exemple, ils sont tous "environ" 1/50 de seconde, mais ils sont tous différents. Et à tout moment, vous pouvez obtenir un cadre beaucoup plus long ou plus court. et votre code peut être exécuté à tout moment dans le cadre.

Gardant cela à l'esprit, vous pouvez vous demander: comment accéder à ces cadres dans votre code, dans Unity?

Tout simplement, vous utilisez soit l'appel Update (), soit vous utilisez une coroutine. (En effet - ils sont exactement la même chose: ils permettent d'exécuter le code à chaque image.)

Le but d'une coroutine est que:

vous pouvez exécuter du code, puis "arrêtez et attendez" **jusqu'à une future image**.

Vous pouvez attendre jusqu'à **la prochaine image**, vous pouvez attendre **un certain nombre d'images** ou vous pouvez attendre un temps **approximatif** en secondes dans le futur.

Par exemple, vous pouvez attendre "environ une seconde", ce qui signifie qu'il faudra attendre environ une seconde, puis placer votre code dans une image à peu près une seconde. (Et en effet, dans ce cadre, le code pourrait être exécuté à tout moment, quel qu'il soit.) Répéter: ce ne sera pas exactement une seconde. Un timing précis n'a pas de sens dans un moteur de jeu.

Dans une coroutine:

Attendre un cadre:

```
// do something
yield return null; // wait until next frame
// do something
```

Attendre trois images:

```
// do something
yield return null; // wait until three frames from now
yield return null;
yield return null;
// do something
```

Attendre **environ** une demi-seconde:

```
// do something
yield return new WaitForSeconds (.5f); // wait for a frame in about .5 seconds
```

```
// do something
```

Faites quelque chose à chaque image:

```
while (true)
{
    // do something
    yield return null; // wait until the next frame
}
```

Cet exemple est littéralement identique à mettre simplement quelque chose à l'intérieur de l'appel "Mise à jour" d'Unity: le code à "faire quelque chose" est exécuté à chaque image.

Exemple

Attachez Ticker à un `GameObject`. Tant que cet objet de jeu est actif, le tick sera exécuté. Notez que le script arrête soigneusement la coroutine lorsque l'objet de jeu devient inactif; c'est généralement un aspect important de l'ingénierie correcte de l'utilisation des coroutines.

```
using UnityEngine;
using System.Collections;

public class Ticker:MonoBehaviour {

    void OnEnable()
    {
        StartCoroutine(TickEverySecond());
    }

    void OnDisable()
    {
        StopAllCoroutines();
    }

    IEnumerator TickEverySecond()
    {
        var wait = new WaitForSeconds(1f); // REMEMBER: IT IS ONLY APPROXIMATE
        while(true)
        {
            Debug.Log("Tick");
            yield return wait; // wait for a frame, about 1 second from now
        }
    }
}
```

Mettre fin à une coroutine

Souvent, vous concevez des coroutines qui se terminent naturellement lorsque certains objectifs sont atteints.

```
IEnumerator TickFiveSeconds()
{
```

```

var wait = new WaitForSeconds(1f);
int counter = 1;
while(counter < 5)
{
    Debug.Log("Tick");
    counter++;
    yield return wait;
}
Debug.Log("I am done ticking");
}

```

Pour arrêter une coroutine de "l'intérieur" de la coroutine, vous ne pouvez pas simplement "revenir" comme vous le feriez plus tôt d'une fonction ordinaire. Au lieu de cela, vous utilisez la `yield break`.

```

IEnumerator ShowExplosions()
{
    ... show basic explosions
    if(player.xp < 100) yield break;
    ... show fancy explosions
}

```

Vous pouvez également forcer toutes les coroutines lancées par le script à s'arrêter avant de terminer.

```

void OnDisable()
{
    // Stops all running coroutines
    StopAllCoroutines();
}

```

La méthode pour arrêter une coroutine *spécifique* à partir de l'appelant varie selon la manière dont vous l'avez démarrée.

Si vous avez démarré une coroutine par nom de chaîne:

```
StartCoroutine("YourAnimation");
```

alors vous pouvez l'arrêter en appelant **StopCoroutine** avec le même nom de chaîne:

```
StopCoroutine("YourAnimation");
```

Vous pouvez également conserver une référence à la `IEnumerator` renvoyée par la méthode `coroutine`, ou l'`IEnumerator` objet retourné par `StartCoroutine` et appeler `StopCoroutine` sur l'un de ceux-ci :

```

public class SomeComponent : MonoBehaviour
{
    Coroutine routine;

    void Start ()
    {
        routine = StartCoroutine(YourAnimation());
    }
}

```

```

    }

    void Update () {
        // later, in response to some input...
        StopCoroutine(routine);
    }

    IEnumerator YourAnimation () { /* ... */ }
}

```

Méthodes MonoBehaviour qui peuvent être Coroutines

Il existe trois méthodes MonoBehaviour qui peuvent être réalisées en routines.

1. Début()
2. OnBecameVisible ()
3. OnLevelWasLoaded ()

Cela peut être utilisé pour créer, par exemple, des scripts qui ne s'exécutent que lorsque l'objet est visible pour une caméra.

```

using UnityEngine;
using System.Collections;

public class RotateObject : MonoBehaviour
{
    IEnumerator OnBecameVisible()
    {
        var tr = GetComponent<Transform>();
        while (true)
        {
            tr.Rotate(new Vector3(0, 180f * Time.deltaTime));
            yield return null;
        }
    }

    void OnBecameInvisible()
    {
        StopAllCoroutines();
    }
}

```

Enchaînement de routines

Les routines peuvent céder à l'intérieur d'elles-mêmes et attendre d'**autres routines**.

Vous pouvez donc enchaîner les séquences - "l'une après l'autre".

C'est très simple, et c'est une technique de base dans l'Unité.

Dans les jeux, il est absolument naturel que certaines choses se passent "dans l'ordre". Presque chaque "round" d'un jeu commence par une série d'événements se produisant, dans un certain temps, dans un certain ordre. Voici comment commencer un jeu de course automobile:

```
IEnumerator BeginRace()
{
    yield return StartCoroutine(PrepareRace());
    yield return StartCoroutine(Countdown());
    yield return StartCoroutine(StartRace());
}
```

Donc, quand vous appelez BeginRace ...

```
StartCoroutine(BeginRace());
```

Il exécutera votre routine "préparer la course". (Peut-être, en faisant clignoter des lumières et en faisant du bruit sur la foule, en réinitialisant les scores, etc.). Lorsque cela sera terminé, votre séquence "Compte à rebours" sera animée, vous permettant d'animer un compte à rebours sur l'interface utilisateur. Lorsque cela est terminé, il lancera votre code de départ de la course, où vous pourrez peut-être exécuter des effets sonores, démarrer des pilotes d'IA, déplacer l'appareil d'une certaine manière, etc.

Pour plus de clarté, comprenez que les trois appels

```
yield return StartCoroutine(PrepareRace());
yield return StartCoroutine(Countdown());
yield return StartCoroutine(StartRace());
```

doivent eux-mêmes **être dans** une coroutine. C'est-à-dire qu'ils doivent être dans une fonction du type `IEnumerator`. Donc, dans notre exemple, c'est `IEnumerator BeginRace`. Ainsi, à partir du code "normal", vous lancez cette coroutine avec l'appel `StartCoroutine`.

```
StartCoroutine(BeginRace());
```

Pour mieux comprendre le chaînage, voici une fonction qui enchaîne les coroutines. Vous passez dans un tableau de routines. La fonction exécute autant de routines que vous passez, dans l'ordre, l'une après l'autre.

```
// run various routines, one after the other
IEnumerator OneAfterTheOther( params IEnumerator[] routines )
{
    foreach ( var item in routines )
    {
        while ( item.MoveNext() ) yield return item.Current;
    }

    yield break;
}
```

Voici comment vous appelez cela ... disons que vous avez trois fonctions. Rappelez-vous qu'ils doivent tous être `IEnumerator`:

```
IEnumerator PrepareRace()
{
    // codesay, crowd cheering and camera pan around the stadium
```

```

        yield break;
    }

IEnumerator Countdown()
{
    // codesay, animate your countdown on UI
    yield break;
}

IEnumerator StartRace()
{
    // codesay, camera moves and light changes and launch the AIs
    yield break;
}

```

Tu appellerais ça comme ça

```
StartCoroutine( MultipleRoutines( PrepareRace(), Countdown(), StartRace() ) );
```

ou peut-être comme ça

```

IEnumerator[] routines = new IEnumerator[] {
    PrepareRace(),
    Countdown(),
    StartRace() };
StartCoroutine( MultipleRoutines( routines ) );

```

Pour répéter, l'une des exigences les plus fondamentales des jeux est que certaines choses se produisent les unes après les autres "dans un ordre" au fil du temps. Vous réalisez cela dans l'unité très simplement, avec

```

yield return StartCoroutine(PrepareRace());
yield return StartCoroutine(Countdown());
yield return StartCoroutine(StartRace());

```

Façons de céder

Vous pouvez attendre jusqu'à la prochaine image.

```
yield return null; // wait until sometime in the next frame
```

Vous pouvez avoir plusieurs de ces appels d'affilée, pour simplement attendre autant d'images que vous le souhaitez.

```

//wait for a few frames
yield return null;
yield return null;

```

Attendez **environ** n secondes. Il est extrêmement important de comprendre que ce n'est qu'un **temps très approximatif**.

```
yield return new WaitForSeconds(n);
```

Il n'est absolument pas possible d'utiliser l'appel "WaitForSeconds" pour toute forme de synchronisation précise.

Souvent, vous voulez enchaîner les actions. Alors, faites quelque chose, et quand c'est fini, faites autre chose et quand c'est fini, faites autre chose. Pour cela, attendez une autre coroutine:

```
yield return StartCoroutine(coroutine);
```

Comprenez que vous ne pouvez l'appeler que dans une coroutine. Alors:

```
StartCoroutine(Test());
```

C'est comme ça que vous commencez une coroutine à partir d'un morceau de code "normal".

Puis, dans cette coroutine en cours d'exécution:

```
Debug.Log("A");
StartCoroutine(LongProcess());
Debug.Log("B");
```

Cela imprimera A, démarrera le long processus et **imprimera immédiatement B**. Il n'attendra pas que le long processus se termine. D'autre part:

```
Debug.Log("A");
yield return StartCoroutine(LongProcess());
Debug.Log("B");
```

Cela imprimera A, lancera le long processus, **attendra la fin du processus**, puis imprimera B.

Il est toujours utile de se rappeler que les coroutines n'ont absolument aucun lien avec le threading. Avec ce code:

```
Debug.Log("A");
StartCoroutine(LongProcess());
Debug.Log("B");
```

il est facile de penser que c'est comme "démarrer" le LongProcess sur un autre thread en arrière-plan. Mais c'est absolument incorrect. C'est juste une coroutine. Les moteurs de jeux sont basés sur des cadres et les "coroutines" dans Unity vous permettent simplement d'accéder aux images.

Il est très facile d'attendre la fin d'une requête Web.

```
void Start() {
    string url = "http://google.com";
    WWW www = new WWW(url);
    StartCoroutine(WaitForRequest(www));
}
```

```
IEnumerator WaitForRequest(WWW www) {
    yield return www;

    if (www.error == null) {
        //use www.data);
    }
    else {
        //use www.error);
    }
}
```

Pour être complet: dans de très rares cas, vous utilisez une mise à jour fixe dans Unity; il y a un appel `WaitForFixedUpdate()` qui normalement ne serait jamais utilisé. Il existe un appel spécifique (`WaitForEndOfFrame()` dans la version actuelle d'Unity) qui est utilisé dans certaines situations pour générer des captures d'écran au cours du développement. (Le mécanisme exact change légèrement au fur et à mesure que l'Unity évolue, donc google pour les dernières informations s'il ya lieu.)

Lire Coroutines en ligne: <https://riptutorial.com/fr/unity3d/topic/3415/coroutines>

Chapitre 8: Couches

Examples

Utilisation de la couche

Les calques Unity sont similaires aux balises car ils peuvent être utilisés pour définir des objets devant interagir ou se comporter d'une certaine manière. Cependant, les calques sont principalement utilisés avec des fonctions de la classe `Physics` : [Unity Documentation - Physics](#)

Les calques sont représentés par un entier et peuvent être transmis aux fonctions de cette manière:

```
using UnityEngine;
class LayerExample {

    public int layer;

    void Start()
    {
        Collider[] colliders = Physics.OverlapSphere(transform.position, 5f, layer);
    }
}
```

L'utilisation d'un calque de cette manière n'inclut que les Colliders dont les GameObjects ont le calque spécifié dans les calculs effectués. Cela simplifie la logique et améliore les performances.

Structure du masque de calque

La structure `LayerMask` est une interface qui fonctionne presque exactement comme si on passait un entier à la fonction en question. Cependant, son plus grand avantage est de permettre à l'utilisateur de sélectionner la couche en question dans un menu déroulant de l'inspecteur.

```
using UnityEngine;
class LayerMaskExample{

    public LayerMask mask;
    public Vector3 direction;

    void Start()
    {
        if(Physics.Raycast(transform.position, direction, 35f, mask))
        {
            Debug.Log("Raycast hit");
        }
    }
}
```

Il possède également plusieurs fonctions statiques permettant de convertir les noms de couches en index ou index en noms de couche.

```
using UnityEngine;
class NameToLayerExample{

    void Start()
    {
        int layerindex = LayerMask.NameToLayer("Obstacle");
    }
}
```

Pour faciliter la vérification de la couche, définissez la méthode d'extension suivante.

```
public static bool IsInLayerMask(this GameObject @object, LayerMask layerMask)
{
    bool result = (1 << @object.layer & layerMask) == 0;

    return result;
}
```

Cette méthode vous permettra de vérifier si un objet de jeu se trouve dans un masque (sélectionné dans l'éditeur) ou non.

Lire Couches en ligne: <https://riptutorial.com/fr/unity3d/topic/4762/couches>

Chapitre 9: Développement multiplateforme

Exemples

Définitions du compilateur

Les définitions du compilateur exécutent un code spécifique à la plate-forme. En les utilisant, vous pouvez faire de petites différences entre les différentes plates-formes.

- Déclenchez les réalisations de Game Center sur les appareils Apple et les performances de Google Play sur les appareils Android.
- Changer les icônes dans les menus (logo Windows dans Windows, Linux pingouin dans Linux).
- Possiblement avoir une mécanique spécifique à la plate-forme en fonction de la plate-forme.
- Et beaucoup plus...

```
void Update() {  
  
#if UNITY_IPHONE  
    //code here is only called when running on iPhone  
#endif  
  
#if UNITY_STANDALONE_WIN && !UNITY_EDITOR  
    //code here is only ran in a unity game running on windows outside of the editor  
#endif  
  
//other code that will be ran regardless of platform  
  
}
```

[Une liste complète des définitions du compilateur Unity peut être trouvée ici](#)

Organisation des méthodes spécifiques de la plate-forme aux classes partielles

Les [classes partielles](#) permettent de séparer la logique de base de vos scripts des méthodes spécifiques à la plate-forme.

Les classes et méthodes partielles sont marquées avec le mot-clé `partial`. Cela signale au compilateur de laisser la classe "open" et de regarder dans les autres fichiers pour le reste de l'implémentation.

```
// ExampleClass.cs  
using UnityEngine;  
  
public partial class ExampleClass : MonoBehaviour  
{  
    partial void PlatformSpecificMethod();  
  
    void OnEnable()  
}
```

```
{  
    PlatformSpecificMethod();  
}  
}
```

Nous pouvons maintenant créer des fichiers pour nos scripts spécifiques à la plate-forme qui implémentent la méthode partielle. Les méthodes partielles peuvent avoir des paramètres (également `ref`) mais doivent renvoyer `void`.

```
// ExampleClass.Iphone.cs  
  
#if UNITY_IPHONE  
using UnityEngine;  
  
public partial class ExampleClass  
{  
    partial void PlatformSpecificMethod()  
    {  
        Debug.Log("I am an iPhone");  
    }  
}  
#endif
```

```
// ExampleClass.Android.cs  
  
#if UNITY_ANDROID  
using UnityEngine;  
  
public partial class ExampleClass  
{  
    partial void PlatformSpecificMethod()  
    {  
        Debug.Log("I am an Android");  
    }  
}  
#endif
```

Si une méthode partielle n'est pas implémentée, le compilateur omettra l'appel.

Astuce: Ce modèle est utile lors de la création de méthodes spécifiques à l'éditeur.

Lire Développement multiplateforme en ligne:

<https://riptutorial.com/fr/unity3d/topic/4816/developpement-multiplateforme>

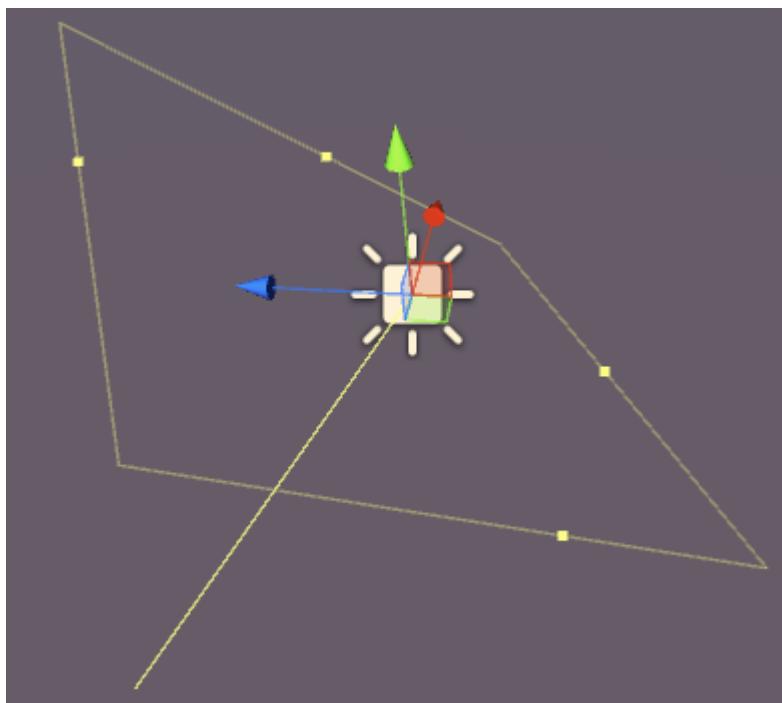
Chapitre 10: Éclairage Unity

Examples

Types de lumière

Lumière de zone

La lumière est émise à travers la surface d'une zone rectangulaire. Ils sont cuits seulement, ce qui signifie que vous ne serez pas en mesure de voir l'effet tant que vous n'avez pas cuit la scène.

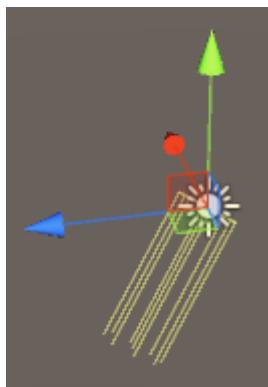


Les lumières de zone ont les propriétés suivantes:

- **Largeur** - Largeur de la zone claire.
- **Hauteur** - Hauteur de la zone claire.
- **Couleur** - Attribuez la couleur de la lumière.
- **Intensité** - Quelle est la puissance de la lumière de 0 à 8.
- **Intensité du rebond** - Quelle est la puissance de la lumière *indirecte* de 0 à 8.
- **Draw Halo** - Dessine un halo autour de la lumière.
- **Flare** - Permet d'affecter un effet de fusée éclairante à la lumière.
- **Mode de rendu** - Auto, Important, Not Important.
- **Culling Mask** - Permet d'allumer des parties d'une scène de manière sélective.

Lumière directionnelle

Les lumières directionnelles émettent de la lumière dans une seule direction (un peu comme le soleil). Peu importe où dans la scène le véritable objet GameObject est placé car la lumière est "partout". L'intensité lumineuse ne diminue pas comme les autres types de lumière.

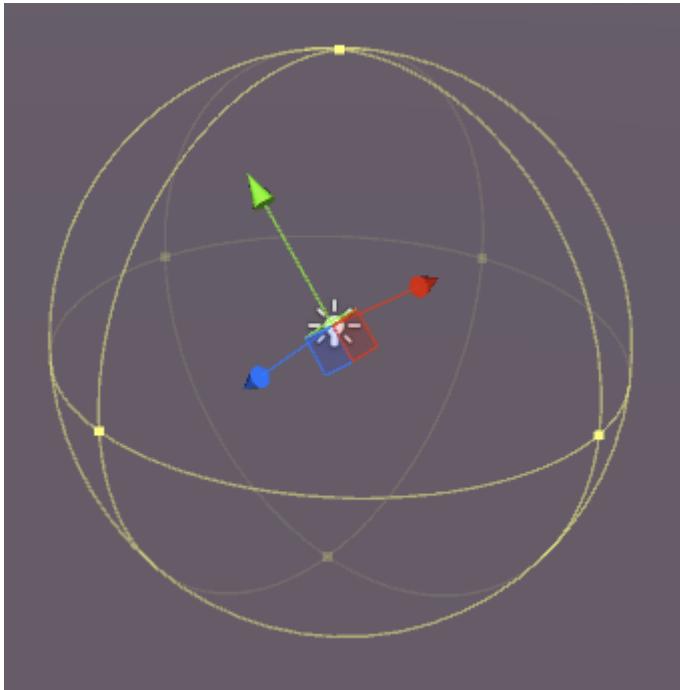


Un éclairage directionnel a les propriétés suivantes:

- **Cuisson en temps** réel, au four ou en mélange.
- **Couleur** - Attribuez la couleur de la lumière.
- **Intensité** - Quelle est la puissance de la lumière de 0 à 8.
- **Intensité du rebond** - Quelle est la puissance de la lumière *indirecte* de 0 à 8.
- **Type d'ombre** - Pas d'ombre, d'ombre dure ou d'ombre douce.
- **Cookie** - Permet d'attribuer un cookie à la lumière.
- **Taille du cookie** - La taille du cookie attribué.
- **Draw Halo** - **Dessine** un halo autour de la lumière.
- **Flare** - Permet d'affecter un effet de fusée éclairante à la lumière.
- **Mode de rendu** - Auto, Important, Not Important.
- **Culling Mask** - Permet d'allumer des parties d'une scène de manière sélective.

Point de lumière

Un point lumineux émet de la lumière à partir d'un point situé dans l'espace dans toutes les directions. Plus on s'éloigne du point d'origine, moins la lumière est intense.

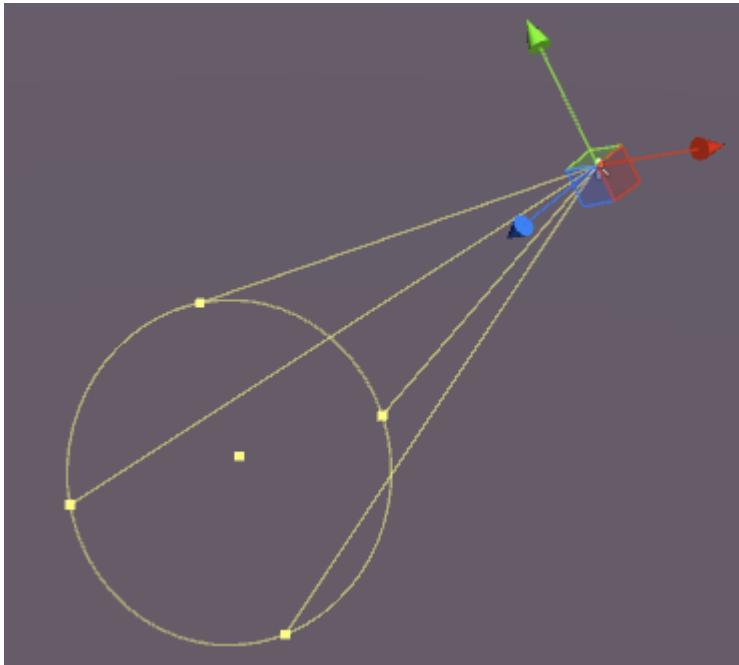


Les lumières ponctuelles ont les propriétés suivantes:

- **Cuisson en temps réel**, au four ou en mélange.
- **Range** - La distance du point où la lumière n'atteint plus.
- **Couleur** - Attribuez la couleur de la lumière.
- **Intensité** - Quelle est la puissance de la lumière de 0 à 8.
- **Intensité du rebond** - Quelle est la puissance de la lumière *indirecte* de 0 à 8.
- **Type d'ombre** - Pas d'ombre, d'ombre dure ou d'ombre douce.
- **Cookie** - Permet d'attribuer un cookie à la lumière.
- **Draw Halo - Dessine** un halo autour de la lumière.
- **Flare** - Permet d'affecter un effet de fusée éclairante à la lumière.
- **Mode de rendu** - Auto, Important, Not Important.
- **Culling Mask** - Permet d'allumer des parties d'une scène de manière sélective.

Projecteur

Un Spot Light ressemble beaucoup à un Point Light, mais l'émission est limitée à un angle. Le résultat est un "cône" de lumière, utile pour les phares de voiture ou les projecteurs.



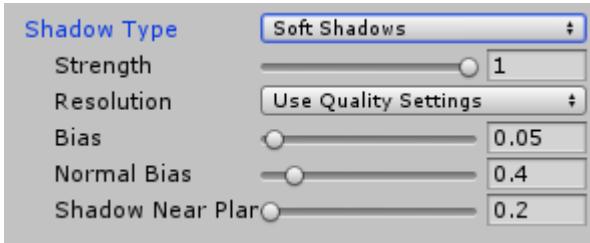
Les projecteurs ont les propriétés suivantes:

- **Cuisson en temps réel**, au four ou en mélange.
- **Range** - La distance du point où la lumière n'atteint plus.
- **Angle Spot** - Angle d'émission lumineuse.
- **Couleur** - Attribuez la couleur de la lumière.
- **Intensité** - Quelle est la puissance de la lumière de 0 à 8.
- **Intensité du rebond** - Quelle est la puissance de la lumière *indirecte* de 0 à 8.
- **Type d'ombre** - Pas d'ombre, d'ombre dure ou d'ombre douce.
- **Cookie** - Permet d'attribuer un cookie à la lumière.
- **Draw Halo - Dessine** un halo autour de la lumière.
- **Flare** - Permet d'affecter un effet de fusée éclairante à la lumière.
- **Mode de rendu** - Auto, Important, Not Important.
- **Culling Mask** - Permet d'allumer des parties d'une scène de manière sélective.

Note sur les ombres

Si vous sélectionnez Hard ou Soft Shadows, les options suivantes sont disponibles dans l'inspecteur:

- **Force** - Comme les ombres sont noires de 0 à 1.
- **Résolution** - Quelle est la précision des ombres?
- **Biais** - degré auquel les surfaces de projection de l'ombre sont éloignées de la lumière.
- **Biais normal** - Le degré auquel les surfaces de projection d'ombre sont poussées vers l'intérieur le long de leurs normales.
- **Shadow Near Plane** - 0.1 - 10.



Émission

L'émission se produit lorsqu'une surface (ou plutôt un matériau) émet de la lumière. Dans le panneau d'inspection d'un matériau sur un objet statique utilisant le Shader Standard, il existe une propriété d'émission:

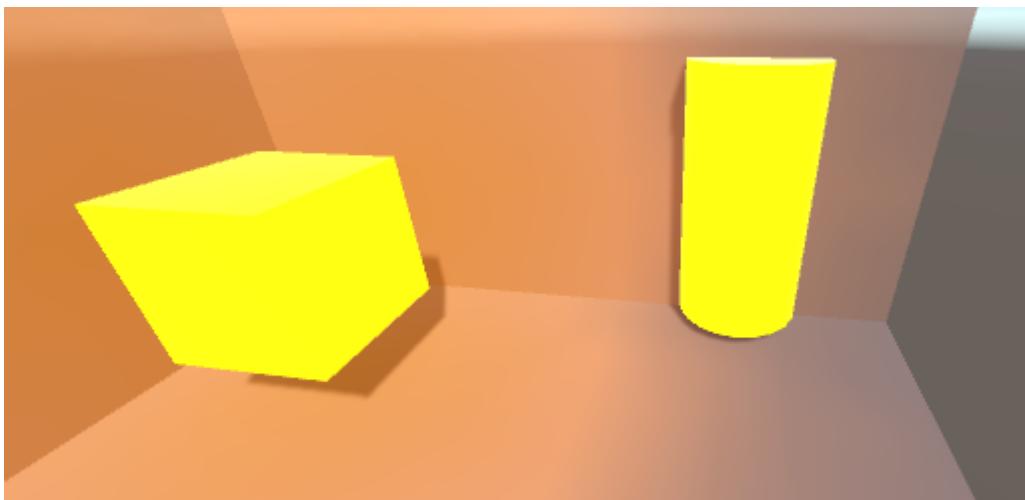


Si vous modifiez cette propriété à une valeur supérieure à la valeur par défaut, vous pouvez définir la couleur d'émission ou attribuer une **carte d'émission** au matériau. Toute texture affectée à cette fente permettra à l'émission d'utiliser ses propres couleurs.

Il existe également une option d'illumination globale qui vous permet de définir si l'émission est cuite sur des objets statiques proches ou non:

- **Baked** - L'émission sera cuite dans la scène
- **Realtime** - L'émission affectera les objets dynamiques
- **Aucun** - L'émission n'affectera pas les objets proches

Si l'objet n'est *pas* réglé sur statique, l'effet fera toujours apparaître l'objet "brillant" mais aucune lumière n'est émise. Le cube ici est statique, le cylindre n'est pas:



Vous pouvez définir la couleur d'émission dans un code comme celui-ci:

```
Renderer renderer = GetComponent<Renderer>();
Material mat = renderer.material;
mat.SetColor("_EmissionColor", Color.yellow);
```

La lumière émise tombera à un rythme quadratique et ne montrera que contre les matériaux statiques de la scène.

Lire Éclairage Unity en ligne: <https://riptutorial.com/fr/unity3d/topic/7884/eclairage-unity>

Chapitre 11: Extension de l'éditeur

Syntaxe

- [MenuItem (string itemName)]
- [MenuItem (string itemName, bool isValidateFunction)]
- [MenuItem (string itemName, bool isValidateFunction, int priority)]
- [ContextMenu (nom de chaîne)]
- [ContextMenuItem (nom de chaîne, fonction de chaîne)]
- [DrawGizmo (GizmoType gizmo)]
- [DrawGizmo (gizmoType, type drawnGizmoType)]

Paramètres

Paramètre	Détails
MenuCommand	MenuCommand permet d'extraire le contexte d'un MenuItem
MenuCommand.context	L'objet qui est la cible de la commande de menu
MenuCommand.userData	Un int pour transmettre des informations personnalisées à un élément de menu

Exemples

Inspecteur Personnalisé

L'utilisation d'un inspecteur personnalisé vous permet de modifier la manière dont un script est dessiné dans l'inspecteur. Parfois, vous souhaitez ajouter des informations supplémentaires dans l'inspecteur pour votre script, ce qui n'est pas possible avec un tiroir de propriétés personnalisé.

Vous trouverez ci-dessous un exemple simple d'objet personnalisé qui, avec l'utilisation d'un inspecteur personnalisé, peut afficher des informations plus utiles.

```
using UnityEngine;
#if UNITY_EDITOR
using UnityEditor;
#endif

public class InspectorExample : MonoBehaviour {

    public int Level;
    public float BaseDamage;

    public float DamageBonus {
        get {
            return Level / 100f * 50;
        }
    }
}
```

```

        }

    }

    public float ActualDamage {
        get {
            return BaseDamage + DamageBonus;
        }
    }

}

#endif UNITY_EDITOR
[CustomEditor( typeof( InspectorExample ) )]
public class CustomInspector : Editor {

    public override void OnInspectorGUI() {
        base.OnInspectorGUI();

        var ie = (InspectorExample)target;

        EditorGUILayout.LabelField( "Damage Bonus", ie.DamageBonus.ToString() );
        EditorGUILayout.LabelField( "Actual Damage", ie.ActualDamage.ToString() );
    }
}
#endif

```

Tout d'abord, nous définissons notre comportement personnalisé avec certains champs

```

public class InspectorExample : MonoBehaviour {
    public int Level;
    public float BaseDamage;
}

```

Les champs affichés ci-dessus sont automatiquement dessinés (sans inspecteur personnalisé) lorsque vous consultez le script dans la fenêtre Inspecteur.

```

public float DamageBonus {
    get {
        return Level / 100f * 50;
    }
}

public float ActualDamage {
    get {
        return BaseDamage + DamageBonus;
    }
}

```

Ces propriétés ne sont pas automatiquement dessinées par Unity. Pour afficher ces propriétés dans la vue Inspecteur, nous devons utiliser notre inspecteur personnalisé.

Nous devons d'abord définir notre inspecteur personnalisé comme ceci

```

[CustomEditor( typeof( InspectorExample ) )]
public class CustomInspector : Editor {

```

L'inspecteur personnalisé doit dériver de l' éditeur et nécessite l'attribut *CustomEditor* . Le

paramètre de l'attribut est le type d'objet pour lequel cet inspecteur personnalisé doit être utilisé.

Le suivant est la méthode `OnInspectorGUI`. Cette méthode est appelée chaque fois que le script est affiché dans la fenêtre de l'inspecteur.

```
public override void OnInspectorGUI() {
    base.OnInspectorGUI();
}
```

Nous appelons `base.OnInspectorGUI()` pour laisser Unity gérer les autres champs du script. Si nous n'appelions pas cela, nous devrions faire plus de travail nous-mêmes.

Ensuite sont nos propriétés personnalisées que nous voulons montrer

```
var ie = (InspectorExample)target;

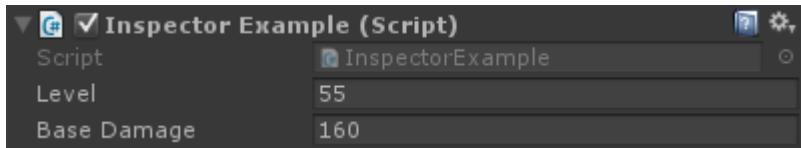
EditorGUILayout.LabelField("Damage Bonus", ie.DamageBonus.ToString());
EditorGUILayout.LabelField("Actual Damage", ie.ActualDamage.ToString());
```

Nous devons créer une variable temporaire contenant la cible convertie dans notre type personnalisé (la cible est disponible car nous dérivons de l'éditeur).

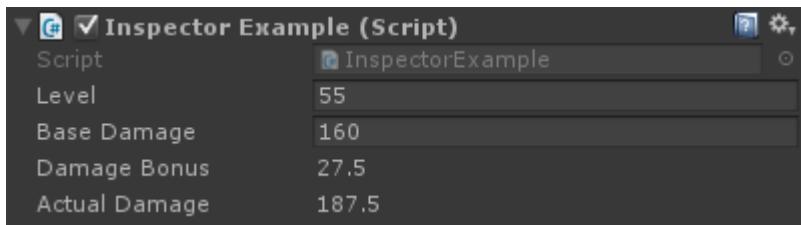
Ensuite, nous pouvons décider comment dessiner nos propriétés, dans ce cas deux labelfields sont suffisants car nous voulons juste montrer les valeurs et ne pas pouvoir les éditer.

Résultat

Avant



Après



Tiroir de propriété personnalisée

Parfois, vous avez des objets personnalisés qui contiennent des données mais ne dérivent pas de `MonoBehaviour`. L'ajout de ces objets en tant que champ dans une classe `MonoBehaviour` n'aura aucun effet visuel, sauf si vous écrivez votre propre tiroir de propriétés personnalisé pour le type de l'objet.

Vous trouverez ci-dessous un exemple simple d'objet personnalisé, ajouté à `MonoBehaviour` et un

tiroir de propriétés personnalisé pour l'objet personnalisé.

```
public enum Gender {
    Male,
    Female,
    Other
}

// Needs the Serializable attribute otherwise the CustomPropertyDrawer wont be used
[Serializable]
public class UserInfo {
    public string Name;
    public int Age;
    public Gender Gender;
}

// The class that you can attach to a GameObject
public class PropertyDrawerExample : MonoBehaviour {
    public UserInfo UIInfo;
}

[CustomPropertyDrawer( typeof( UserInfo ) )]
public class UserInfoDrawer : PropertyDrawer {

    public override float GetPropertyHeight( SerializedProperty property, GUIContent label ) {
        // The 6 comes from extra spacing between the fields (2px each)
        return EditorGUI.singleLineHeight * 4 + 6;
    }

    public override void OnGUI( Rect position, SerializedProperty property, GUIContent label ) {
        EditorGUI.BeginProperty( position, label, property );

        EditorGUI.LabelField( position, label );

        var nameRect = new Rect( position.x, position.y + 18, position.width, 16 );
        var ageRect = new Rect( position.x, position.y + 36, position.width, 16 );
        var genderRect = new Rect( position.x, position.y + 54, position.width, 16 );

        EditorGUI.indentLevel++;
        EditorGUI.PropertyField( nameRect, property.FindPropertyRelative( "Name" ) );
        EditorGUI.PropertyField( ageRect, property.FindPropertyRelative( "Age" ) );
        EditorGUI.PropertyField( genderRect, property.FindPropertyRelative( "Gender" ) );

        EditorGUI.indentLevel--;
        EditorGUI.EndProperty();
    }
}
```

Tout d'abord, nous définissons l'objet personnalisé avec toutes ses exigences. Juste une classe simple décrivant un utilisateur. Cette classe est utilisée dans notre classe `PropertyDrawerExample` que nous pouvons ajouter à un `GameObject`.

```
public enum Gender {
    Male,
    Female,
    Other
}
```

```

}

[Serializable]
public class UserInfo {
    public string Name;
    public int Age;
    public Gender Gender;
}

public class PropertyDrawerExample : MonoBehaviour {
    public UserInfo UIInfo;
}

```

La classe personnalisée a besoin de l'attribut `Serializable`, sinon `CustomPropertyDrawer` ne sera pas utilisé

Le suivant est le `CustomPropertyDrawer`

Nous devons d'abord définir une classe dérivée de `PropertyDrawer`. La définition de classe nécessite également l'attribut `CustomPropertyDrawer`. Le paramètre passé est le type de l'objet à utiliser pour ce tiroir.

```

[CustomPropertyDrawer( typeof( UserInfo ) )]
public class UserInfoDrawer : PropertyDrawer {

```

Ensuite, nous remplaçons la fonction `GetPropertyHeight`. Cela nous permet de définir une hauteur personnalisée pour notre propriété. Dans ce cas, nous savons que notre propriété aura quatre parties: étiquette, nom, âge et sexe. Par conséquent, nous utilisons `EditorGUIUtility.singleLineHeight * 4`, nous ajoutons 6 pixels supplémentaires car nous voulons espacer chaque champ de deux pixels.

```

public override float GetPropertyHeight( SerializedProperty property, GUIContent label ) {
    return EditorGUIUtility.singleLineHeight * 4 + 6;
}

```

Suivant est la méthode `OnGUI` réelle. Nous commençons avec `EditorGUI.BeginProperty ([...])` et terminons la fonction avec `EditorGUI.EndProperty ()`. Nous faisons cela pour que si cette propriété fait partie d'un préfabriqué, la logique de remplacement préfabriquée réelle fonctionnerait pour tout entre ces deux méthodes.

```

public override void OnGUI( Rect position, SerializedProperty property, GUIContent label ) {
    EditorGUI.BeginProperty( position, label, property );

```

Après cela, nous montrons une étiquette contenant le nom du champ et nous définissons déjà les rectangles pour nos champs.

```

EditorGUI.LabelField( position, label );

var nameRect = new Rect( position.x, position.y + 18, position.width, 16 );
var ageRect = new Rect( position.x, position.y + 36, position.width, 16 );
var genderRect = new Rect( position.x, position.y + 54, position.width, 16 );

```

Chaque champ est espacé de 16 + 2 pixels et la hauteur est de 16 (ce qui est identique à `EditorGUIUtility.singleLineHeight`)

Ensuite, indentez l'interface utilisateur avec un onglet pour une présentation un peu plus agréable, affichez les propriétés, désélectionnez l'interface graphique et *terminez* avec `EditorGUI.EndProperty`.

```
EditorGUI.indentLevel++;

EditorGUI.PropertyField( nameRect, property.FindPropertyRelative( "Name" ) );
EditorGUI.PropertyField( ageRect, property.FindPropertyRelative( "Age" ) );
EditorGUI.PropertyField( genderRect, property.FindPropertyRelative( "Gender" ) );

EditorGUI.indentLevel--;

EditorGUI.EndProperty();
```

Nous *affichons* les champs en utilisant `EditorGUI.PropertyField` qui nécessite un rectangle pour la position et un `SerializedProperty` pour la propriété à afficher. Nous acquérons la propriété en appelant `FindPropertyRelative ("...")` sur la propriété passée dans la fonction `OnGUI`. Notez que ces propriétés sont sensibles à la casse et que les propriétés non publiques sont introuvables!

Pour cet exemple, je ne sauvegarde pas les propriétés retournées par `property.FindPropertyRelative ("...")`. Vous devez les enregistrer dans des champs privés de la classe pour éviter les appels inutiles

Résultat

Avant



Après



Articles de menu

Les éléments de menu sont un excellent moyen d'ajouter des actions personnalisées à l'éditeur. Vous pouvez ajouter des éléments de menu à la barre de menus, les utiliser en tant que clics contextuels sur des composants spécifiques, ou même en tant que clics contextuels sur des champs de vos scripts.

Vous trouverez ci-dessous un exemple de la manière d'appliquer des éléments de menu.

```

public class MenuItemsExample : MonoBehaviour {

    [MenuItem( "Example/DoSomething %#&d" )]
    private static void DoSomething() {
        // Execute some code
    }

    [MenuItem( "Example/DoAnotherThing", true )]
    private static bool DoAnotherThingValidator() {
        return Selection.gameObjects.Length > 0;
    }

    [MenuItem( "Example/DoAnotherThing _PGUP", false )]
    private static void DoAnotherThing() {
        // Execute some code
    }

    [MenuItem( "Example/DoOne %a", false, 1 )]
    private static void DoOne() {
        // Execute some code
    }

    [MenuItem( "Example/DoTwo #b", false, 2 )]
    private static void DoTwo() {
        // Execute some code
    }

    [MenuItem( "Example/DoFurther &c", false, 13 )]
    private static void DoFurther() {
        // Execute some code
    }

    [MenuItem( "CONTEXT/Camera/DoCameraThing" )]
    private static void DoCameraThing( MenuCommand cmd ) {
        // Execute some code
    }

    [ContextMenu( "ContextSomething" )]
    private void ContentSomething() {
        // Execute some code
    }

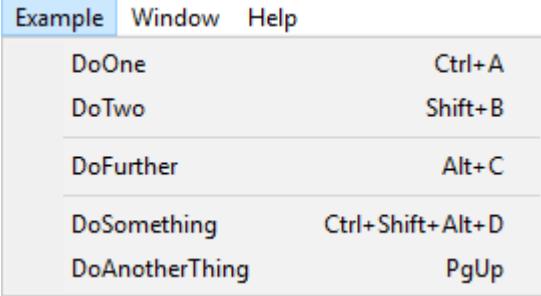
    [ContextMenu( "Reset", "ResetDate" )]
    [ContextMenu( "Set to Now", "SetDateToNow" )]
    public string Date = "";

    public void ResetDate() {
        Date = "";
    }

    public void SetDateToNow() {
        Date = DateTime.Now.ToString();
    }
}

```

Qui ressemble à ceci



Passons en revue l'élément de menu de base. Comme vous pouvez le voir ci-dessous, vous devez définir une fonction statique avec un attribut `MenuItem`, auquel vous transmettez une chaîne comme titre pour l'élément de menu. Vous pouvez mettre votre élément de menu à plusieurs niveaux en ajoutant un / dans le nom.

```
[MenuItem( "Example/DoSomething %#&d" )]  
private static void DoSomething() {  
    // Execute some code  
}
```

Vous ne pouvez pas avoir un élément de menu au niveau supérieur. Vos éléments de menu doivent être dans un sous-menu!

Les caractères spéciaux à la fin du nom de `MenuItem` sont des raccourcis clavier, ils ne sont pas obligatoires.

Il existe des caractères spéciaux que vous pouvez utiliser pour vos touches de raccourci, à savoir:

- % - Ctrl sous Windows, Cmd sous OS X
- # - Décalage
- & - Alt

Cela signifie que le raccourci % # & d signifie ctrl + shift + alt + D sur Windows et cmd + shift + alt + D sur OS X.

Si vous souhaitez utiliser un raccourci sans touche spéciale, par exemple juste la touche «D», vous pouvez ajouter le caractère de soulignement (_) au raccourci clavier que vous souhaitez utiliser.

Il existe d'autres clés spéciales qui sont prises en charge, à savoir:

- GAUCHE, DROITE, HAUT, BAS - pour les touches fléchées
- F1..F12 - pour les touches de fonction
- HOME, END, PGUP, PGDN - pour les touches de navigation

Les touches de raccourci doivent être séparées de tout autre texte avec un espace

Viennent ensuite les éléments de menu du validateur. Les éléments du menu du validateur permettent de désactiver les éléments de menu (grisés, non cliquables) lorsque la condition n'est pas remplie. Un exemple pourrait être que votre élément de menu agit sur la sélection actuelle de GameObjects, que vous pouvez vérifier dans l'élément de menu du validateur.

```
[MenuItem( "Example/DoAnotherThing", true )]
private static bool DoAnotherThingValidator() {
    return Selection.gameObjects.Length > 0;
}

[MenuItem( "Example/DoAnotherThing _PGUP", false )]
private static void DoAnotherThing() {
    // Execute some code
}
```

Pour qu'un élément de menu du validateur fonctionne, vous devez créer deux fonctions statiques, à la fois avec l'attribut MenuItem et le même nom (la touche de raccourci importe peu). La différence entre les deux est que vous les marquez comme une fonction de validation ou non en passant un paramètre booléen.

Vous pouvez également définir l'ordre des éléments du menu en ajoutant une priorité. La priorité est définie par un entier que vous passez en troisième paramètre. Plus le nombre est petit, plus le chiffre est élevé dans la liste. Vous pouvez ajouter un séparateur entre deux éléments de menu en vous assurant qu'il y a au moins 10 chiffres entre la priorité des éléments du menu.

```
[MenuItem( "Example/DoOne %a", false, 1 )]
private static void DoOne() {
    // Execute some code
}

[MenuItem( "Example/DoTwo #b", false, 2 )]
private static void DoTwo() {
    // Execute some code
}

[MenuItem( "Example/DoFurther &c", false, 13 )]
private static void DoFurther() {
    // Execute some code
}
```

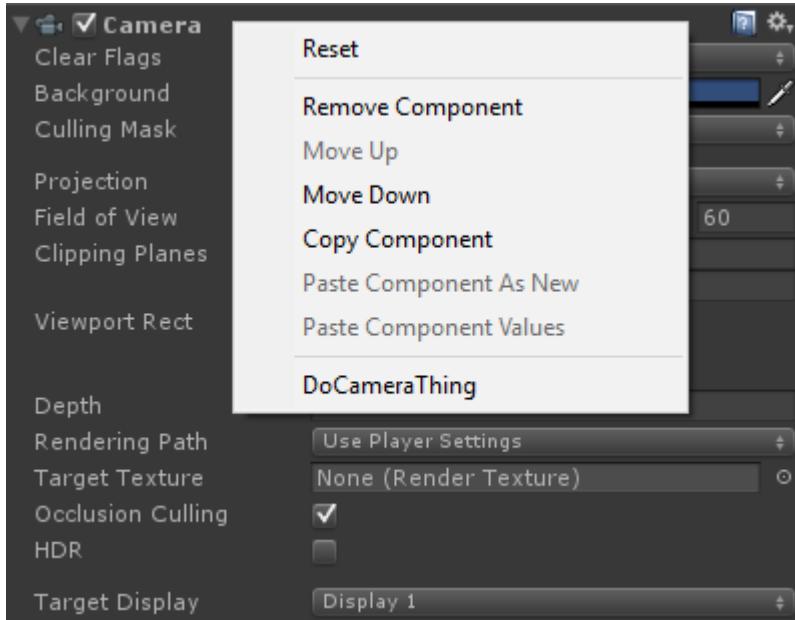
Si vous avez une liste de menus combinant des éléments prioritaires et non hiérarchisés, les éléments non prioritaires seront séparés des éléments prioritaires.

Ensuite, vous ajoutez un élément de menu au menu contextuel d'un composant déjà existant. Vous devez démarrer le nom du MenuItem avec CONTEXT (sensible à la casse) et faire en sorte que votre fonction prenne un paramètre MenuCommand.

L'extrait suivant ajoute un élément de menu contextuel au composant Caméra.

```
[MenuItem( "CONTEXT/Camera/DoCameraThing" )]
private static void DoCameraThing( MenuCommand cmd ) {
    // Execute some code
}
```

Qui ressemble à ceci

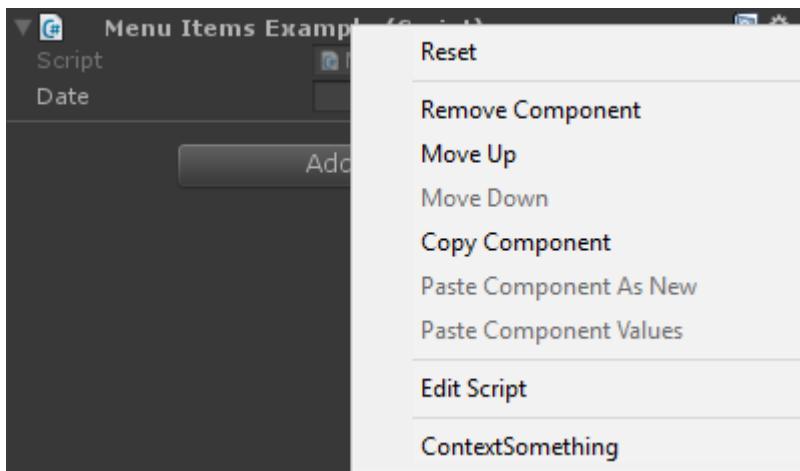


Le paramètre `MenuCommand` vous donne accès à la valeur du composant et à toute donnée utilisateur envoyée avec celui-ci.

Vous pouvez également ajouter un élément de menu contextuel à vos propres composants en utilisant l'attribut `ContextMenu`. Cet attribut ne prend qu'un nom, aucune validation ou priorité et doit faire partie d'une méthode non statique.

```
[ContextMenu( "ContextSomething" )]
private void ContentSomething() {
    // Execute some code
}
```

Qui ressemble à ceci



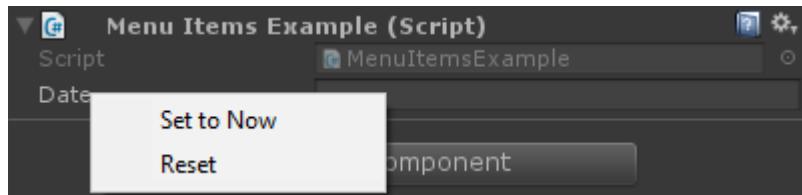
Vous pouvez également ajouter des éléments de menu contextuel aux champs de votre propre composant. Ces éléments de menu apparaissent lorsque vous cliquez sur le champ auquel ils appartiennent et que vous pouvez exécuter des méthodes que vous avez définies dans ce composant. De cette façon, vous pouvez ajouter par exemple des valeurs par défaut ou la date actuelle, comme indiqué ci-dessous.

```
[ContextMenu("Reset", "ResetDate")]
[ContextMenu("Set to Now", "SetDateToNow")]
public string Date = "";

public void ResetDate() {
    Date = "";
}

public void SetDateToNow() {
    Date = DateTime.Now.ToString();
}
```

Qui ressemble à ceci



Gizmos

Les gizmos sont utilisés pour dessiner des formes dans la vue de la scène. Vous pouvez utiliser ces formes pour dessiner des informations supplémentaires sur vos GameObjects, par exemple le frustum qu'ils possèdent ou la plage de détection.

Voici deux exemples sur la façon de procéder

Un exemple

Cet exemple utilise les *méthodes OnDrawGizmos* et *OnDrawGizmosSelected* (magic).

```
public class GizmoExample : MonoBehaviour {

    public float GetDetectionRadius() {
        return 12.5f;
    }

    public float GetFOV() {
        return 25f;
    }

    public float GetMaxRange() {
        return 6.5f;
    }

    public float GetMinRange() {
        return 0;
    }

    public float GetAspect() {
        return 2.5f;
    }
}
```

```

public void OnDrawGizmos() {
    var gizmoMatrix = Gizmos.matrix;
    var gizmoColor = Gizmos.color;

    Gizmos.matrix = Matrix4x4TRS( transform.position, transform.rotation,
transform.lossyScale );
    Gizmos.color = Color.red;
    Gizmos.DrawFrustum( Vector3.zero, GetFOV(), GetMaxRange(), GetMinRange(), GetAspect()
);

    Gizmos.matrix = gizmoMatrix;
    Gizmos.color = gizmoColor;
}

public void OnDrawGizmosSelected() {
    Handles.DrawWireDisc( transform.position, Vector3.up, GetDetectionRadius() );
}
}

```

Dans cet exemple, nous avons deux méthodes pour dessiner des gizmos, l'une qui dessine lorsque l'objet est actif (OnDrawGizmos) et l'autre lorsque l'objet est sélectionné dans la hiérarchie (OnDrawGizmosSelected).

```

public void OnDrawGizmos() {
    var gizmoMatrix = Gizmos.matrix;
    var gizmoColor = Gizmos.color;

    Gizmos.matrix = Matrix4x4TRS( transform.position, transform.rotation,
transform.lossyScale );
    Gizmos.color = Color.red;
    Gizmos.DrawFrustum( Vector3.zero, GetFOV(), GetMaxRange(), GetMinRange(), GetAspect() );

    Gizmos.matrix = gizmoMatrix;
    Gizmos.color = gizmoColor;
}

```

D'abord, nous sauvegardons la matrice et la couleur du gizmo parce que nous allons le changer et que nous voulons le rétablir lorsque nous n'avons plus d'effet sur le dessin du gizmo.

Ensuite, nous voulons dessiner le tronc de notre objet, cependant, nous devons changer la matrice des Gizmos pour qu'elle corresponde à la position, à la rotation et à l'échelle. Nous avons également mis la couleur des Gizmos au rouge pour souligner le frustum. Lorsque cela est fait, nous pouvons appeler *Gizmos.DrawFrustum* pour dessiner le frustum dans la vue de la scène.

Lorsque nous avons fini de dessiner ce que nous voulons dessiner, nous réinitialisons la matrice et la couleur des Gizmos.

```

public void OnDrawGizmosSelected() {
    Handles.DrawWireDisc( transform.position, Vector3.up, GetDetectionRadius() );
}

```

Nous voulons également dessiner une plage de détection lorsque nous sélectionnons notre objet GameObject. Cela se fait via la classe *Handles*, car la classe *Gizmos* ne possède aucune méthode pour les disques.

L'utilisation de cette forme de dessin donne des résultats dans la sortie indiquée ci-dessous.

Exemple deux

Cet exemple utilise l'attribut *DrawGizmo*.

```
public class GizmoDrawerExample {  
  
    [DrawGizmo( GizmoType.Selected | GizmoType.NonSelected, typeof( GizmoExample ) )]  
    public static void DrawGizmo( GizmoExample obj, GizmoType type ) {  
        var gizmoMatrix = Gizmos.matrix;  
        var gizmoColor = Gizmos.color;  
  
        Gizmos.matrix = Matrix4x4.TRS( obj.transform.position, obj.transform.rotation,  
obj.transform.lossyScale );  
        Gizmos.color = Color.red;  
        Gizmos.DrawFrustum( Vector3.zero, obj.GetFOV(), obj.GetMaxRange(), obj.GetMinRange(),  
obj.GetAspect() );  
  
        Gizmos.matrix = gizmoMatrix;  
        Gizmos.color = gizmoColor;  
  
        if ( ( type & GizmoType.Selected ) == GizmoType.Selected ) {  
            Handles.DrawWireDisc( obj.transform.position, Vector3.up, obj.GetDetectionRadius()  
);  
        }  
    }  
}
```

Cette méthode vous permet de séparer les appels de gizmo de votre script. La plupart utilise le même code que l'autre exemple, à l'exception de deux choses.

```
[DrawGizmo( GizmoType.Selected | GizmoType.NonSelected, typeof( GizmoExample ) )]  
public static void DrawGizmo( GizmoExample obj, GizmoType type ) {
```

Vous devez utiliser l'attribut *DrawGizmo* qui prend l'énumération *GizmoType* comme premier paramètre et *Type* comme second paramètre. Le *Type* doit être le type que vous souhaitez utiliser pour dessiner le gizmo.

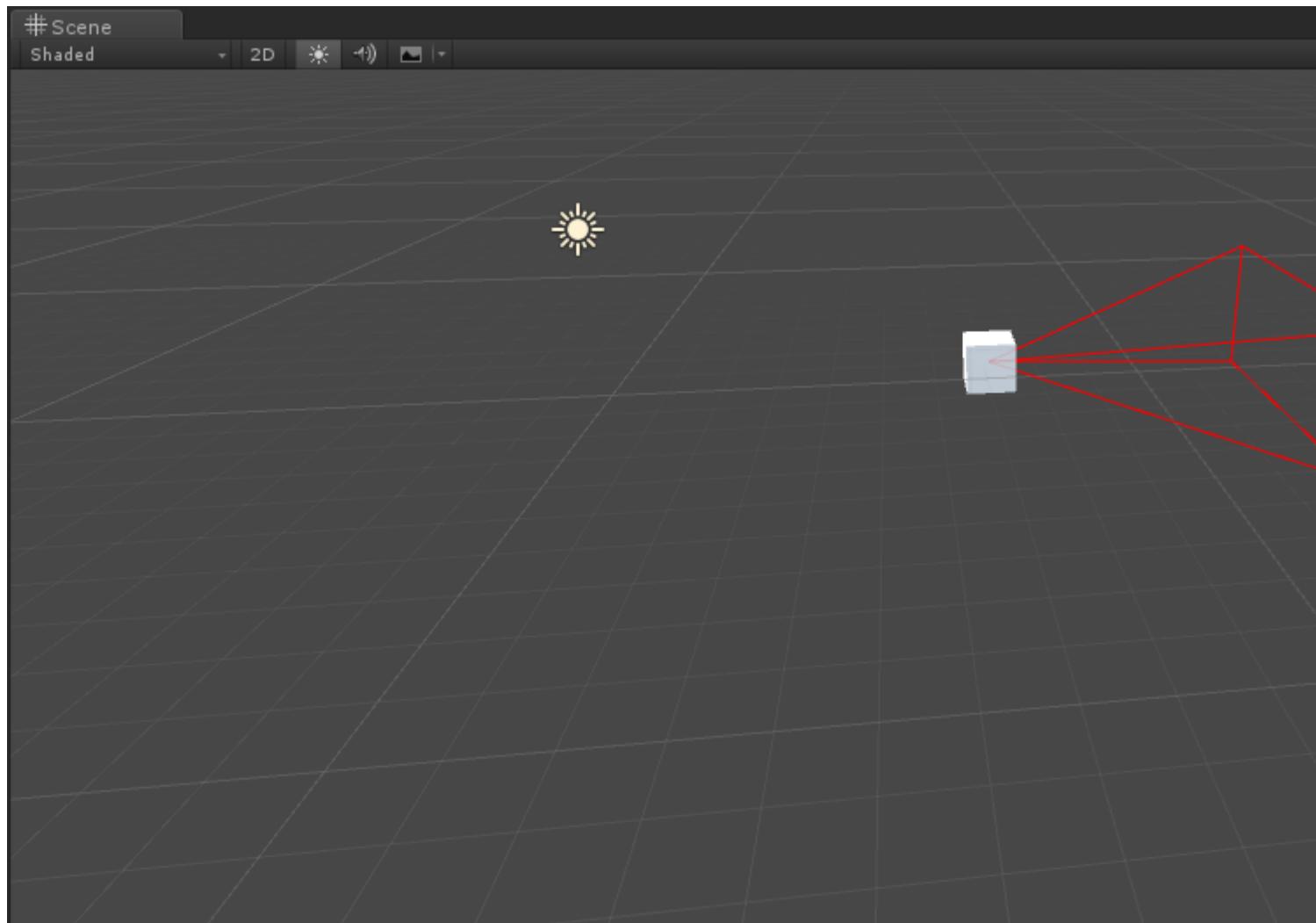
La méthode de dessin du gizmo doit être statique, publique ou non publique et peut être nommée comme vous le souhaitez. Le premier paramètre est le type, qui doit correspondre au type transmis en tant que second paramètre dans l'attribut, et le second paramètre est l'énumération *GizmoType* qui décrit l'état actuel de votre objet.

```
if ( ( type & GizmoType.Selected ) == GizmoType.Selected ) {  
    Handles.DrawWireDisc( obj.transform.position, Vector3.up, obj.GetDetectionRadius() );  
}
```

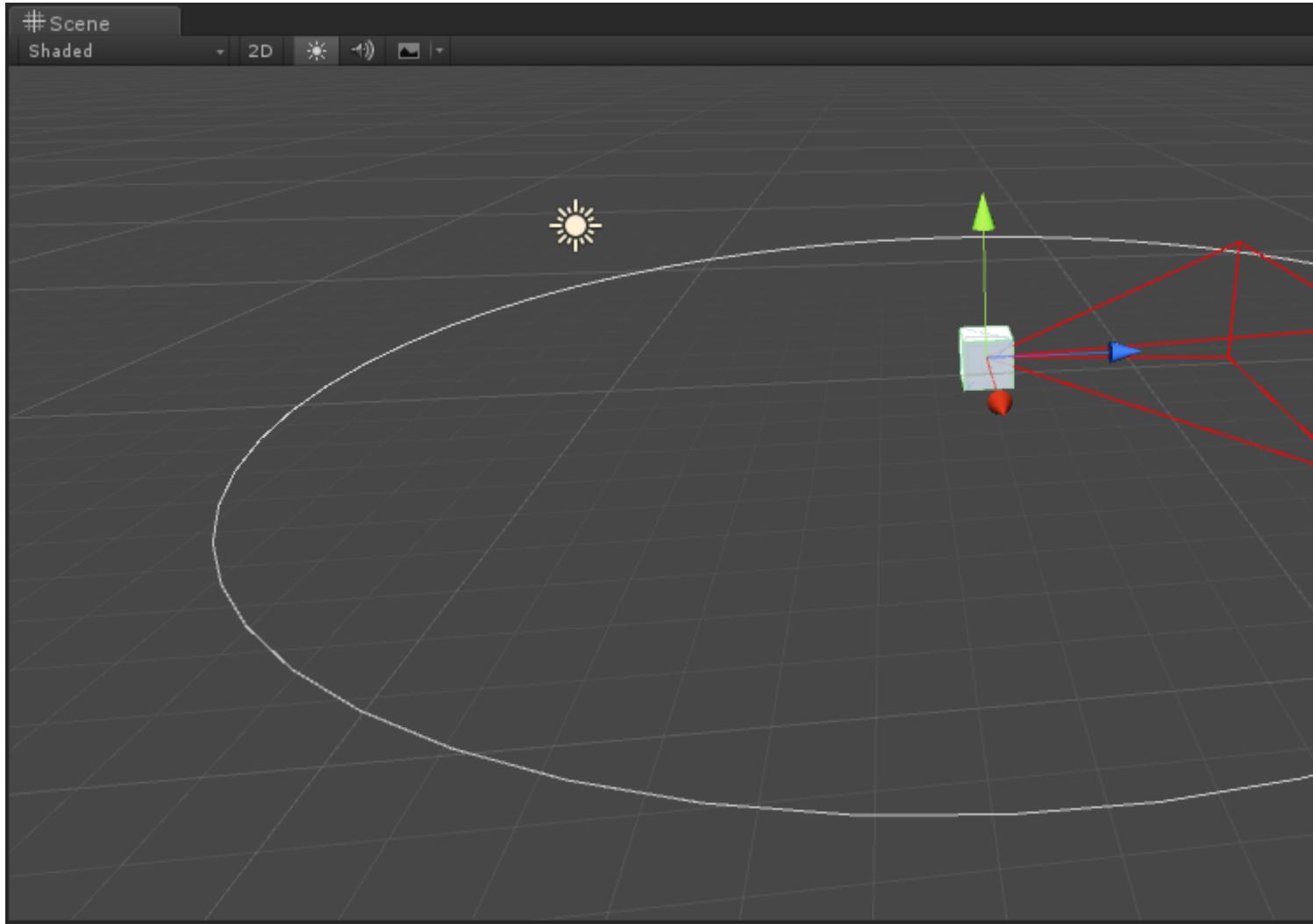
L'autre différence est que pour vérifier quel est le *GizmoType* de l'objet, vous devez faire une vérification AND sur le paramètre et le type que vous voulez.

Résultat

Non sélectionné



Choisi



Fenêtre de l'éditeur

Pourquoi une fenêtre d'édition?

Comme vous l'avez peut-être vu, vous pouvez faire beaucoup de choses dans un inspecteur personnalisé (si vous ne savez pas ce qu'est un inspecteur personnalisé, consultez l'exemple ici: <http://www.riptutorial.com/unity3d/topic/2506 / extend-the-editor> Mais, à un moment donné, vous souhaiterez peut-être implémenter un panneau de configuration ou une palette de ressources personnalisée, dans laquelle vous utiliserez une **fenêtre EditorWindow** . (généralement à travers la barre supérieure), les tabuler, etc.

Créer un éditeur de baseWindow

Exemple simple

La création d'une fenêtre d'édition personnalisée est assez simple. Tout ce que vous avez à faire est d'étendre la classe `EditorWindow` et d'utiliser les méthodes `Init()` et `OnGUI()`. Voici un exemple simple:

```
using UnityEngine;
```

```

using UnityEditor;

public class CustomWindow : EditorWindow
{
    // Add menu named "Custom Window" to the Window menu
    [MenuItem("Window/Custom Window")]
    static void Init()
    {
        // Get existing open window or if none, make a new one:
        CustomWindow window = (CustomWindow) EditorWindow.GetWindow(typeof(CustomWindow));
        window.Show();
    }

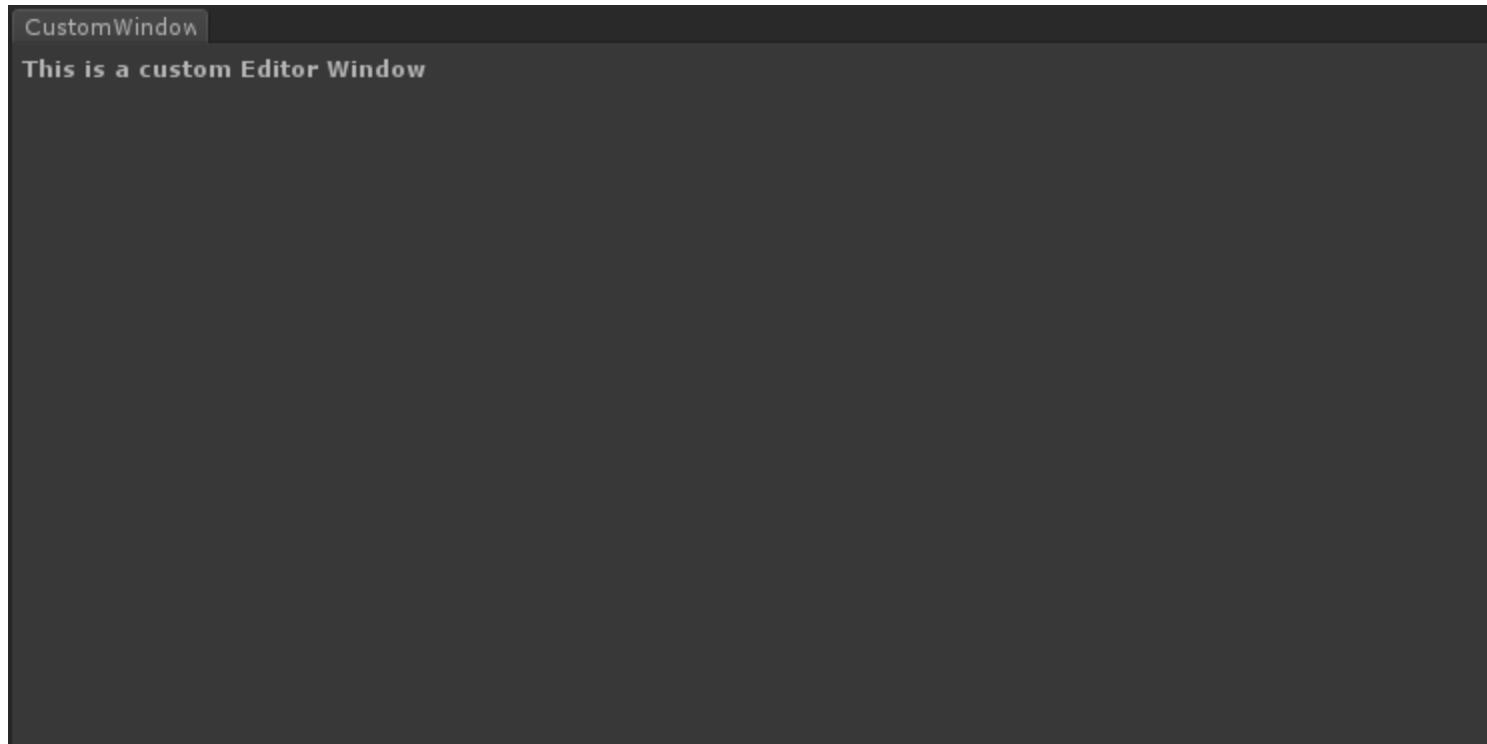
    void OnGUI()
    {
        GUILayout.Label("This is a custom Editor Window", EditorStyles.boldLabel);
    }
}

```

Les 3 points importants sont:

1. N'oubliez pas d'étendre `EditorWindow`
2. Utilisez `Init()` comme indiqué dans l'exemple. `EditorWindow.GetWindow` vérifie si une `CustomWindow` est déjà créée. Sinon, cela créera une nouvelle instance. En utilisant ceci vous vous assurez que vous n'avez pas plusieurs instances de votre fenêtre en même temps
3. Utilisez `OnGUI()` comme d'habitude pour afficher des informations dans votre fenêtre

Le résultat final ressemblera à ceci:



Aller plus loin

Bien sûr, vous voudrez probablement gérer ou modifier certains actifs en utilisant cette

EditorWindow. Voici un exemple d'utilisation de la classe [Selection](#) (pour obtenir la sélection active) et modification des propriétés d'actif sélectionnées via [SerializedObject](#) et [SerializedProperty](#).

```
using System.Linq;
using UnityEngine;
using UnityEditor;

public class CustomWindow : EditorWindow
{
    private AnimationClip _animationClip;
    private SerializedObject _serializedClip;
    private SerializedProperty _events;

    private string _text = "Hello World";

    // Add menu named "Custom Window" to the Window menu
    [MenuItem("Window/Custom Window")]
    static void Init()
    {
        // Get existing open window or if none, make a new one:
        CustomWindow window = (CustomWindow) EditorWindow.GetWindow(typeof(CustomWindow));
        window.Show();
    }

    void OnGUI()
    {
        GUILayout.Label("This is a custom Editor Window", EditorStyles.boldLabel);

        // You can use EditorGUI, EditorGUILayout and GUILayout classes to display
        anything you want
        // A TextField example
        _text = EditorGUILayout.TextField("Text Field", _text);

        // Note that you can modify an asset or a gameobject using an EditorWindow. Here
        is a quick example with an AnimationClip asset
        // The _animationClip, _serializedClip and _events are set in OnSelectionChange()

        if (_animationClip == null || _serializedClip == null || _events == null) return;

        // We can modify our serializedClip like we would do in a Custom Inspector. For
        example we can grab its events and display their information

        GUILayout.Label(_animationClip.name, EditorStyles.boldLabel);

        for (var i = 0; i < _events.arraySize; i++)
        {
            EditorGUILayout.BeginVertical();

            EditorGUILayout.LabelField(
                "Event : " +
                _events.GetArrayElementAtIndex(i).FindPropertyRelative("functionName").stringValue,
                EditorStyles.boldLabel);

            EditorGUILayout.PropertyField(_events.GetArrayElementAtIndex(i).FindPropertyRelative("time"),
            true,
            GUILayout.ExpandWidth(true));

            EditorGUILayout.PropertyField(_events.GetArrayElementAtIndex(i).FindPropertyRelative("functionName"),
```

```

        true, GUILayout.ExpandWidth(true));

EditorGUILayout.PropertyField(_events.GetArrayElementAtIndex(i).FindPropertyRelative("floatParameter"),
                           true, GUILayout.ExpandWidth(true));

EditorGUILayout.PropertyField(_events.GetArrayElementAtIndex(i).FindPropertyRelative("intParameter"),
                           true, GUILayout.ExpandWidth(true));
EditorGUILayout.PropertyField(
    _events.GetArrayElementAtIndex(i).FindPropertyRelative("objectReferenceParameter"), true,
                           GUILayout.ExpandWidth(true));

    EditorGUILayout.Separator();
    EditorGUILayout.EndVertical();
}

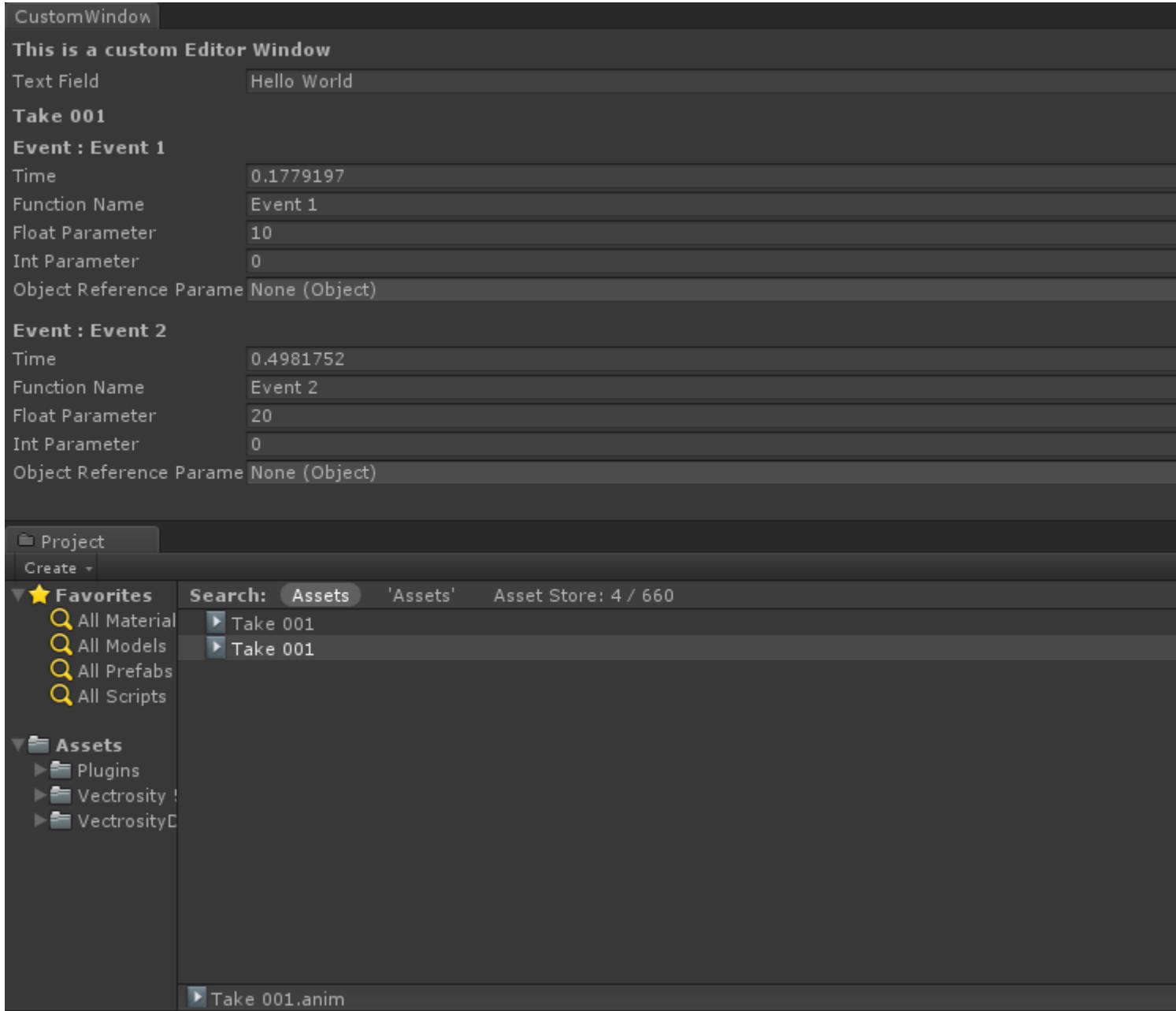
// Of course we need to Apply the modified properties. We don't our changes won't
be saved
_serializedClip.ApplyModifiedProperties();
}

/// This Message is triggered when the user selection in the editor changes. That's
when we should tell our Window to Repaint() if the user selected another AnimationClip
private void OnSelectionChange()
{
    _animationClip =
        Selection.GetFiltered(typeof(AnimationClip),
SelectionMode.Assets).FirstOrDefault() as AnimationClip;
    if (_animationClip == null) return;

    _serializedClip = new SerializedObject(_animationClip);
    _events = _serializedClip.FindProperty("m_Events");
    Repaint();
}
}

```

Voici le résultat:



Sujets avancés

Vous pouvez faire des choses vraiment avancées dans l'éditeur, et la classe `EditorWindow` est parfaite pour afficher une grande quantité d'informations. La plupart des ressources avancées de Unity Asset Store (telles que NodeCanvas ou PlayMaker) utilisent `EditorWindow` pour afficher les vues personnalisées.

Dessin dans le SceneView

Une chose intéressante à faire avec un `EditorWindow` est d'afficher des informations directement dans votre `SceneView`. De cette façon, vous pouvez créer un éditeur de carte / monde entièrement personnalisé, par exemple, en utilisant votre `EditorWindow` personnalisée en tant que palette de ressources et en écoutant les clics dans `SceneView` pour instancier de nouveaux objets. Voici un exemple :

```

using UnityEngine;
using System;
using UnityEditor;

public class CustomWindow : EditorWindow {

    private enum Mode {
        View = 0,
        Paint = 1,
        Erase = 2
    }

    private Mode CurrentMode = Mode.View;

    [MenuItem ("Window/Custom Window")]
    static void Init () {
        // Get existing open window or if none, make a new one:
        CustomWindow window = (CustomWindow)EditorWindow.GetWindow (typeof (CustomWindow));
        window.Show();
    }

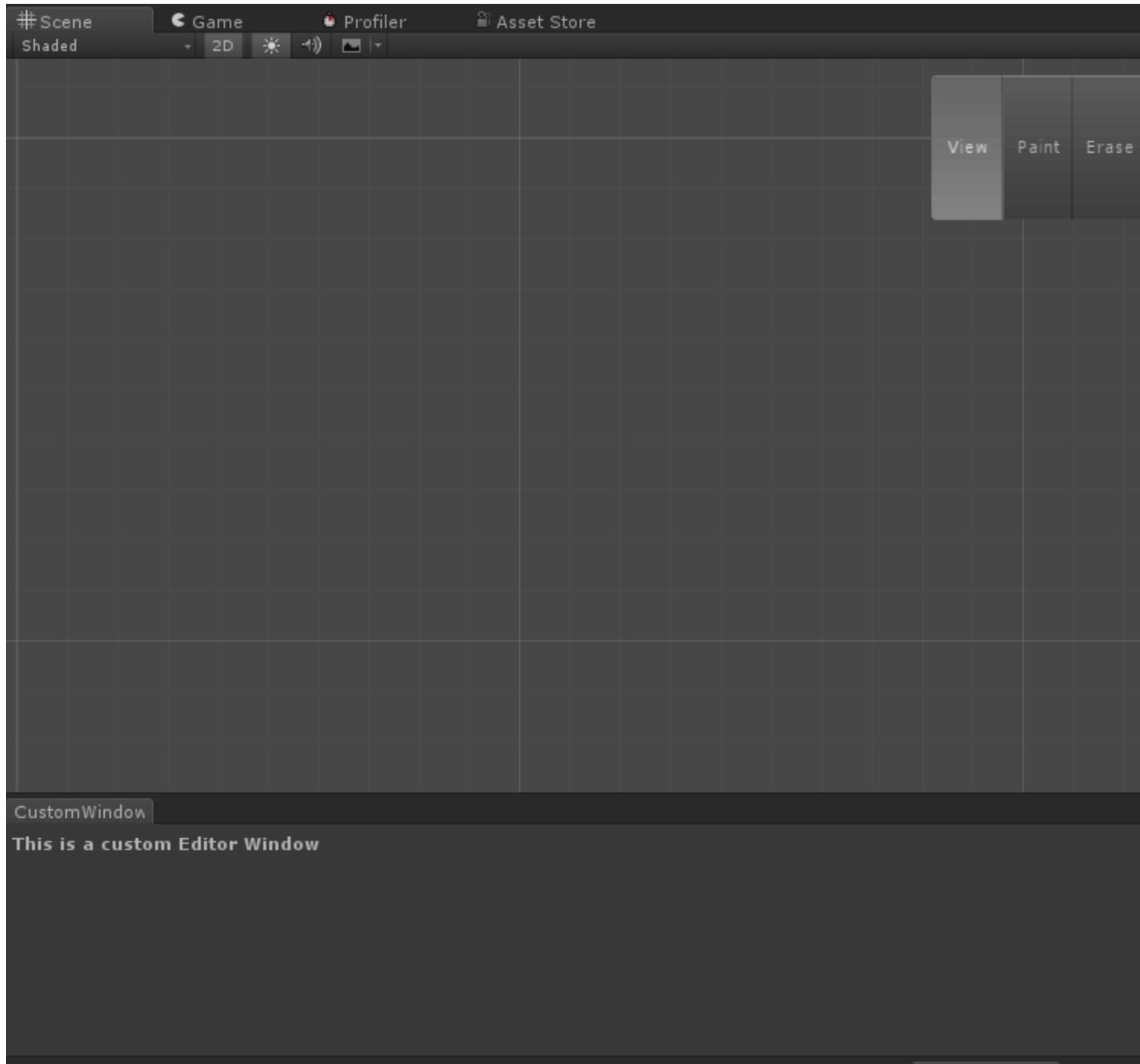
    void OnGUI () {
        GUILayout.Label ("This is a custom Editor Window", EditorStyles.boldLabel);
    }

    void OnEnable() {
        SceneView.onSceneGUIDelegate = SceneViewGUI;
        if (SceneView.lastActiveSceneView) SceneView.lastActiveSceneView.Repaint();
    }

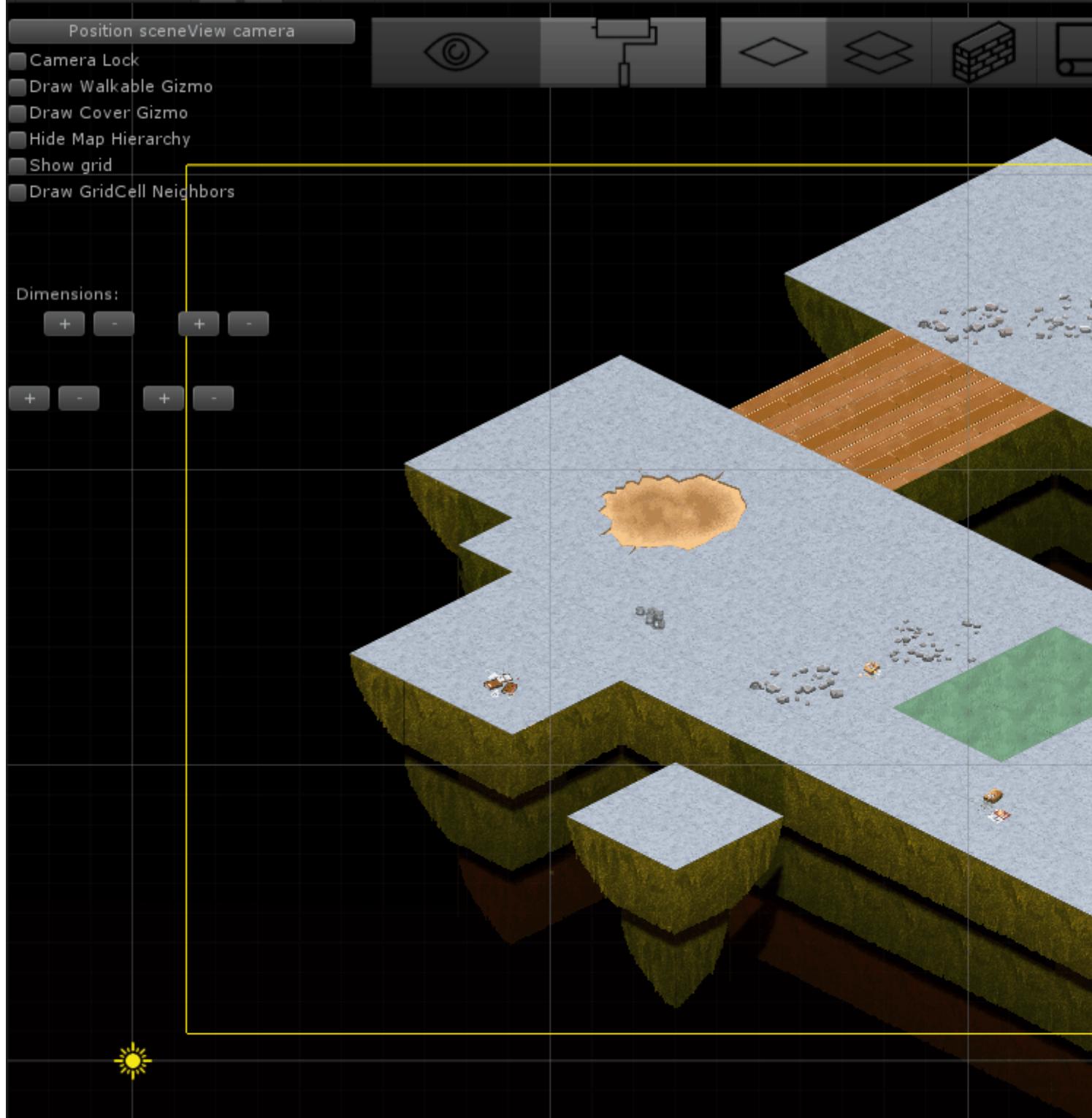
    void SceneViewGUI(SceneView sceneView) {
        Handles.BeginGUI();
        // We define the toolbars' rects here
        var ToolBarRect = new Rect((SceneView.lastActiveSceneView.camera.pixelRect.width / 6),
10, (SceneView.lastActiveSceneView.camera.pixelRect.width * 4 / 6) ,
SceneView.lastActiveSceneView.camera.pixelRect.height / 5);
        GUILayout.BeginArea(ToolBarRect);
        GUILayout.BeginHorizontal();
        GUILayout.FlexibleSpace();
        CurrentMode = (Mode) GUILayout.Toolbar(
            (int) CurrentMode,
            Enum.GetNames(typeof(Mode)),
            GUILayout.Height(ToolBarRect.height));
        GUILayout.FlexibleSpace();
        GUILayout.EndHorizontal();
        GUILayout.EndArea();
        Handles.EndGUI();
    }
}

```

Cela affichera la barre d'outils directement dans votre SceneView



Voici un aperçu de la distance que vous pouvez parcourir:

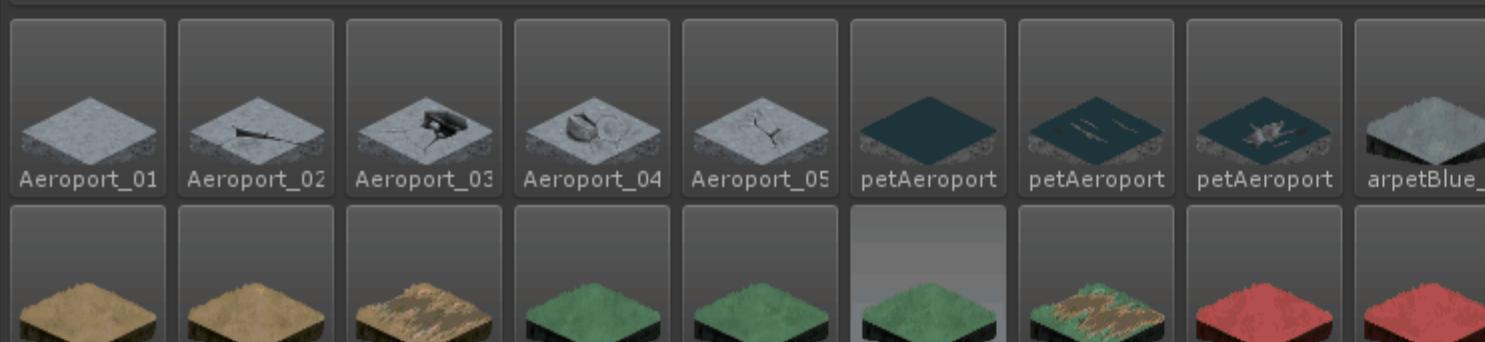


Map Editor Project Console Pro 3

Palette

Search Term :

[Grid]



<https://riptutorial.com/fr/unity3d/topic/2506/extension-de-l-editeur>

Chapitre 12: Implémentation de classe MonoBehaviour

Exemples

Aucune méthode surchargée

La raison pour laquelle vous n'avez pas besoin de remplacer `Awake`, `Start`, `Update` et d'autres méthodes est qu'elles ne sont pas des méthodes virtuelles définies dans une classe de base.

Lors de la première utilisation de votre script, le moteur d'exécution de script examine le script pour voir si certaines méthodes sont définies. Si tel est le cas, ces informations sont mises en cache et les méthodes sont ajoutées à leur liste respective. Ces listes sont ensuite simplement retransmises à des moments différents.

La raison pour laquelle ces méthodes ne sont pas virtuelles est due aux performances. Si tous les scripts avaient `Awake`, `Start`, `OnEnable`, `OnDisable`, `Update`, `LateUpdate` et `FixedUpdate`, ils seraient tous ajoutés à leurs listes, ce qui signifierait que toutes ces méthodes seraient exécutées.

Normalement, cela ne poserait pas de gros problème, cependant, tous ces appels de méthodes vont du côté natif (C++) au côté géré (C#), ce qui entraîne un coût de performance.

Maintenant, imaginez ceci, toutes ces méthodes sont dans leurs listes et certaines / la plupart d'entre elles n'ont peut-être même pas un corps de méthode réel. Cela signifierait qu'il y a une énorme quantité de performance gaspillée sur les méthodes d'appel qui ne font même rien. Pour éviter cela, Unity a choisi de ne pas utiliser de méthodes virtuelles et a créé un système de messagerie qui vérifie que ces méthodes ne sont appelées que lorsqu'elles sont réellement définies, ce qui évite les appels de méthodes inutiles.

Vous pouvez en savoir plus sur le sujet sur un blog Unity ici: [10000 Update \(\) Appels](#) et plus sur IL2CPP ici: [Une introduction à IL2CPP Internals](#)

Lire **Implémentation de classe MonoBehaviour en ligne:**

<https://riptutorial.com/fr/unity3d/topic/2304/implementation-de-classe-monobehaviour>

Chapitre 13: Importateurs et processeurs (post)

Syntaxe

- AssetPostprocessor.OnPreprocessTexture ()

Remarques

Utilisez `String.Contains()` pour traiter uniquement les actifs ayant une chaîne donnée dans leurs chemins d'actifs.

```
if (assetPath.Contains("ProcessThisFolder"))
{
    // Process asset
}
```

Exemples

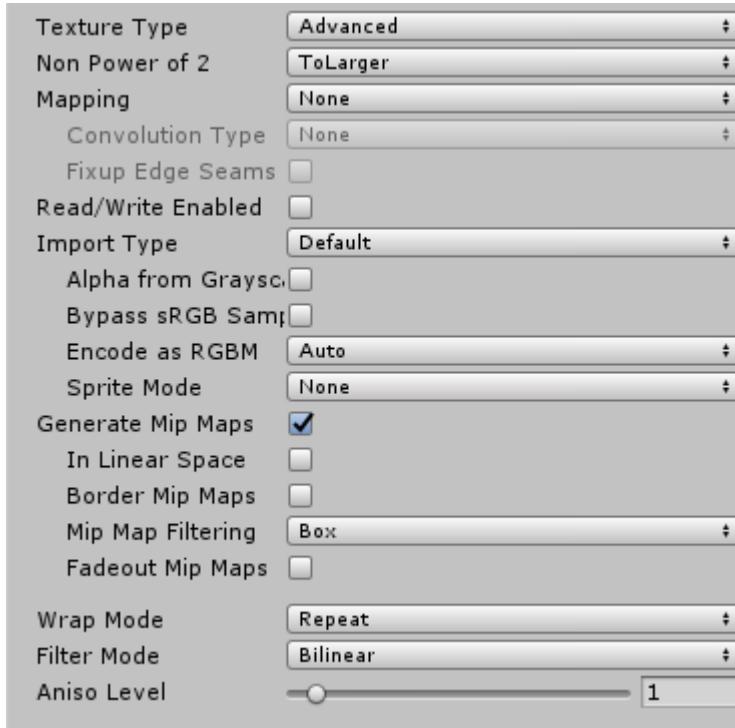
Postprocesseur de texture

Créez le fichier `TexturePostProcessor.cs` n'importe où dans le dossier **Assets** :

```
using UnityEngine;
using UnityEditor;

public class TexturePostProcessor : AssetPostprocessor
{
    void OnPostprocessTexture(Texture2D texture)
    {
        TextureImporter importer = assetImporter as TextureImporter;
        importer.anisoLevel = 1;
        importer.filterMode = FilterMode.Bilinear;
        importer.mipmapEnabled = true;
        importer.npotScale = TextureImporterNPOTScale.ToLarger;
        importer.textureType = TextureImporterType.Advanced;
    }
}
```

Maintenant, chaque fois que Unity importe une texture, elle aura les paramètres suivants:



Si vous utilisez le postprocesseur, vous ne pouvez pas modifier les paramètres de texture en manipulant les paramètres d'**importation** dans l'éditeur.

Lorsque vous appuyez sur le bouton **Appliquer**, la texture est réimportée et le code post-processeur est exécuté à nouveau.

Un importateur de base

Supposons que vous ayez un fichier personnalisé pour lequel vous souhaitez créer un importateur. Ce pourrait être un fichier .xls ou autre. Dans ce cas, nous allons utiliser un fichier JSON car il est facile, mais nous allons choisir une extension personnalisée pour faciliter l'identification des fichiers?

Supposons que le format du fichier JSON est

```
{
    "someValue": 123,
    "someOtherValue": 456.297,
    "someBoolValue": true,
    "someStringValue": "this is a string",
}
```

Sauvegardons cela comme `Example.test` quelque part en *dehors* des actifs pour le moment.

Ensuite, créez un `MonoBehaviour` avec une classe personnalisée uniquement pour les données. La classe personnalisée est uniquement destinée à faciliter la déserialisation du JSON. Vous n'avez PAS besoin d'utiliser une classe personnalisée, mais cet exemple est plus court. Nous allons enregistrer cela dans `TestData.cs`

```
using UnityEngine;
using System.Collections;
```

```

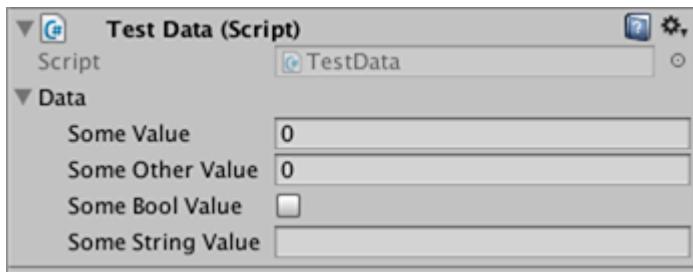
public class TestData : MonoBehaviour {

    [System.Serializable]
    public class Data {
        public int someValue = 0;
        public float someOtherValue = 0.0f;
        public bool someBoolValue = false;
        public string someStringValue = "";
    }

    public Data data = new Data();
}

```

Si vous deviez ajouter manuellement ce script à un GameObject, vous verriez quelque chose comme



Ensuite, créez un dossier `Editor` quelque part sous `Assets`. Je peux être à n'importe quel niveau. Dans le dossier `Editor`, créez un fichier `TestDataAssetPostprocessor.cs` et insérez-le.

```

using UnityEditor;
using UnityEngine;
using System.Collections;

public class TestDataAssetPostprocessor : AssetPostprocessor
{
    const string s_extension = ".test";

    // NOTE: Paths start with "Assets/"
    static bool IsFileWeCareAbout(string path)
    {
        return System.IO.Path.GetExtension(path).Equals(
            s_extension,
            System.StringComparison.OrdinalIgnoreCase);
    }

    static void HandleAddedOrChangedFile(string path)
    {
        string text = System.IO.File.ReadAllText(path);
        // should we check for error if the file can't be parsed?
        TestData.Data newData = JsonUtility.FromJson<TestData.Data>(text);

        string prefabPath = path + ".prefab";
        // Get the existing prefab
        GameObject existingPrefab =
            AssetDatabase.LoadAssetAtPath(prefabPath, typeof(GameObject)) as GameObject;
        if (!existingPrefab)
        {
            // If no prefab exists make one
            GameObject newGameObject = new GameObject();
            newGameObject.AddComponent<TestData>();
        }
    }
}

```

```

        PrefabUtility.CreatePrefab(prefabPath,
                                    newGameObject,
                                    ReplacePrefabOptions.Default);
        GameObject.DestroyImmediate(newGameObject);
        existingPrefab =
            AssetDatabase.LoadAssetAtPath(prefabPath, typeof(GameObject)) as GameObject;
    }

    TestData testData = existingPrefab.GetComponent<TestData>();
    if (testData != null)
    {
        testData.data = newData;
        EditorUtility.SetDirty(existingPrefab);
    }
}

static void HandleRemovedFile(string path)
{
    // Decide what you want to do here. If the source file is removed
    // do you want to delete the prefab? Maybe ask if you'd like to
    // remove the prefab?
    // NOTE: Because you might get many calls (like you deleted a
    // subfolder full of .test files you might want to get all the
    // filenames and ask all at once ("delete all these prefabs?").
}

static void OnPostprocessAllAssets (string[] importedAssets, string[] deletedAssets,
string[] movedAssets, string[] movedFromAssetPaths)
{
    foreach (var path in importedAssets)
    {
        if (IsFileWeCareAbout(path))
        {
            HandleAddedOrChangedFile(path);
        }
    }

    foreach (var path in deletedAssets)
    {
        if (IsFileWeCareAbout(path))
        {
            HandleRemovedFile(path);
        }
    }

    for (var ii = 0; ii < movedAssets.Length; ++ii)
    {
        string srcStr = movedFromAssetPaths[ii];
        string dstStr = movedAssets[ii];

        // the source was moved, let's move the corresponding prefab
        // NOTE: We don't handle the case if there already being
        // a prefab of the same name at the destination
        string srcPrefabPath = srcStr + ".prefab";
        string dstPrefabPath = dstStr + ".prefab";

        AssetDatabase.MoveAsset(srcPrefabPath, dstPrefabPath);
    }
}
}

```

Avec cela sauvegardé, vous devriez pouvoir faire glisser le fichier `Example.test` que nous avons créé ci-dessus dans votre dossier Unity Assets et vous devriez voir le préfabriqué correspondant créé. Si vous modifiez `Example.test` vous verrez que les données du préfabriqué sont immédiatement mises à jour. Si vous faites glisser le préfabriqué dans la hiérarchie de la scène, vous le verrez aussi bien que les modifications de `Example.test`. Si vous déplacez `Example.test` vers un autre dossier, le préfabriqué correspondant se déplacera avec lui. Si vous modifiez un champ sur une instance, modifiez le fichier `Example.test`. Seuls les champs que vous n'avez pas modifiés sur l'instance sont mis à jour.

Améliorations: Dans l'exemple ci-dessus, après avoir `Example.test` dans votre dossier `Assets`, vous verrez à la fois un `Example.test` et un `Example.test.prefab`. Il serait bon de savoir que cela fonctionne plus comme les importateurs de modèles fonctionnent, nous ne verrons que par magie `Example.test` et c'est un `AssetBundle` ou quelque chose du genre. Si vous savez comment s'il vous plaît fournir cet exemple

Lire Importateurs et processeurs (post) en ligne:

<https://riptutorial.com/fr/unity3d/topic/5279/importateurs-et-processeurs--post->

Chapitre 14: Intégration des annonces

Introduction

Cette rubrique concerne l'intégration de services de publicité tiers, tels que Unity Ads ou Google AdMob, dans un projet Unity.

Remarques

Ceci s'applique aux [Unity Ads](#).

Assurez-vous que le mode test pour Unity Ads est activé pendant le développement

En tant que développeur, vous n'êtes pas autorisé à générer des impressions ou à installer en cliquant sur les annonces de votre propre jeu. Cela viole l'accord [de service de Unity Ads](#), et vous serez banni du réseau Unity Ads pour tentative de fraude.

Pour plus d'informations, lisez les [conditions d'utilisation de Unity Ads](#).

Exemples

Unity Ads Basics en C

```
using UnityEngine;
using UnityEngine.Advertisements;

public class Example : MonoBehaviour
{
    #if !UNITY_ADS // If the Ads service is not enabled
    public string gameId; // Set this value from the inspector
    public bool enableTestMode = true; // Enable this during development
    #endif

    void InitializeAds () // Example of how to initialize the Unity Ads service
    {
        #if !UNITY_ADS // If the Ads service is not enabled
        if (Advertisement.isSupported) { // If runtime platform is supported
            Advertisement.Initialize(gameId, enableTestMode); // Initialize
        }
        #endif
    }

    void ShowAd () // Example of how to show an ad
    {
        if (Advertisement.isInitialized || Advertisement.IsReady()) { // If the ads are ready
        to be shown
            Advertisement.Show(); // Show the default ad placement
        }
    }
}
```

Unity Ads Basics en JavaScript

```
#pragma strict
import UnityEngine.Advertisements;

#if !UNITY_ADS // If the Ads service is not enabled
public var gameId : String; // Set this value from the inspector
public var enableTestMode : boolean = true; // Enable this during development
#endif

function InitializeAds () // Example of how to initialize the Unity Ads service
{
    #if !UNITY_ADS // If the Ads service is not enabled
    if (Advertisement.isSupported) { // If runtime platform is supported
        Advertisement.Initialize(gameId, enableTestMode); // Initialize
    }
    #endif
}

function ShowAd () // Example of how to show an ad
{
    if (Advertisement.isInitialized && Advertisement.IsReady()) { // If the ads are ready to
be shown
        Advertisement.Show(); // Show the default ad placement
    }
}
```

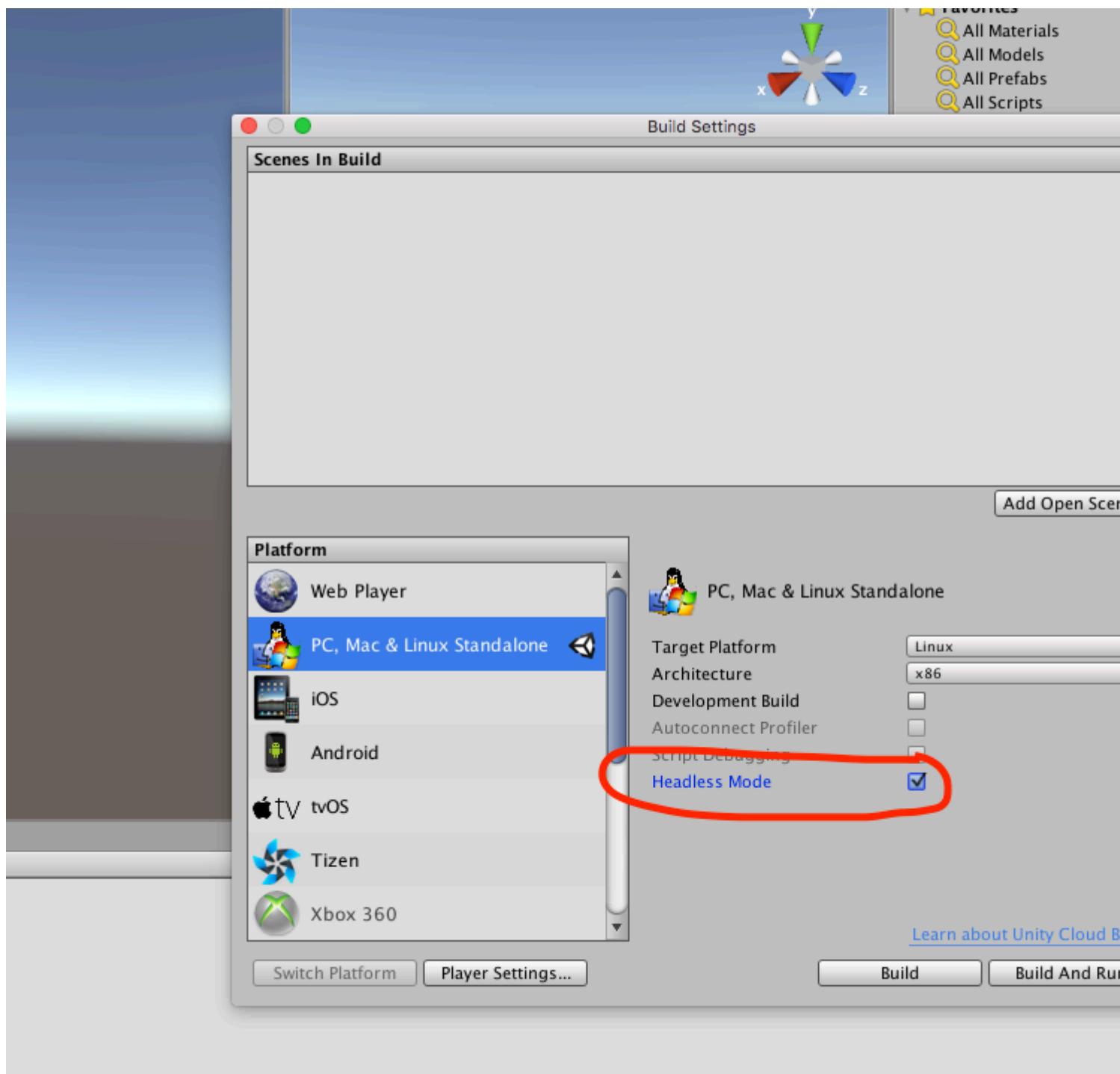
Lire Intégration des annonces en ligne: <https://riptutorial.com/fr/unity3d/topic/9796/integration-des-annonces>

Chapitre 15: La mise en réseau

Remarques

Mode sans tête dans l'unité

Si vous créez un serveur à déployer sous Linux, les paramètres de génération ont une option "Mode sans tête". Une application créée avec cette option n'affiche rien et ne lit pas les entrées de l'utilisateur, ce qui est généralement ce que nous souhaitons pour un serveur.



Exemples

Créer un serveur, un client et envoyer un message.

La mise en réseau d'Unity fournit l'API de haut niveau (HLA) pour gérer les communications de réseau en s'extrayant des implémentations de bas niveau.

Dans cet exemple, nous allons voir comment créer un serveur pouvant communiquer avec un ou plusieurs clients.

Le HLA nous permet de sérialiser facilement une classe et d'envoyer des objets de cette classe sur le réseau.

La classe que nous utilisons pour sérialiser

Cette classe doit hériter de `MessageBase`, dans cet exemple, nous allons simplement envoyer une chaîne dans cette classe.

```
using System;
using UnityEngine.Networking;

public class MyNetworkMessage : MessageBase
{
    public string message;
}
```

Créer un serveur

Nous créons un serveur qui écoute le port 9999, autorise un maximum de 10 connexions et lit les objets du réseau de notre classe personnalisée.

Le HLA associe différents types de messages à un identifiant. Il existe des types de messages par défaut définis dans la classe `MsgType` de Unity Networking. Par exemple, le type de connexion a l'ID 32 et il est appelé dans le serveur lorsqu'un client s'y connecte ou dans le client lorsqu'il se connecte à un serveur. Vous pouvez enregistrer des gestionnaires pour gérer les différents types de messages.

Lorsque vous envoyez une classe personnalisée, comme dans notre cas, nous définissons un gestionnaire avec un nouvel identifiant associé à la classe que nous envoyons sur le réseau.

```
using UnityEngine;
using System.Collections;
using UnityEngine.Networking;

public class Server : MonoBehaviour {
    int port = 9999;
```

```

int maxConnections = 10;

// The id we use to identify our messages and register the handler
short messageID = 1000;

// Use this for initialization
void Start () {
    // Usually the server doesn't need to draw anything on the screen
    Application.runInBackground = true;
    CreateServer();
}

void CreateServer() {
    // Register handlers for the types of messages we can receive
    RegisterHandlers ();

    var config = new ConnectionConfig ();
    // There are different types of channels you can use, check the official documentation
    config.AddChannel (QosType.ReliableFragmented);
    config.AddChannel (QosType.UnreliableFragmented);

    var ht = new HostTopology (config, maxConnections);

    if (!NetworkServer.Configure (ht)) {
        Debug.Log ("No server created, error on the configuration definition");
        return;
    } else {
        // Start listening on the defined port
        if(NetworkServer.Listen (port))
            Debug.Log ("Server created, listening on port: " + port);
        else
            Debug.Log ("No server created, could not listen to the port: " + port);
    }
}

void OnApplicationQuit() {
    NetworkServer.Shutdown ();
}

private void RegisterHandlers () {
    // Unity have different Messages types defined in MsgType
    NetworkServer.RegisterHandler (MsgType.Connect, OnClientConnected);
    NetworkServer.RegisterHandler (MsgType.Disconnect, OnClientDisconnected);

    // Our message use his own message type.
    NetworkServer.RegisterHandler (messageID, OnMessageReceived);
}

private void RegisterHandler(short t, NetworkMessageDelegate handler) {
    NetworkServer.RegisterHandler (t, handler);
}

void OnClientConnected(NetworkMessage netMessage)
{
    // Do stuff when a client connects to this server

    // Send a thank you message to the client that just connected
    MyNetworkMessage messageContainer = new MyNetworkMessage();
    messageContainer.message = "Thanks for joining!";

    // This sends a message to a specific client, using the connectionId
}

```

```

NetworkServer.SendToClient(netMessage.conn.connectionId,messageID,messageContainer);

// Send a message to all the clients connected
messageContainer = new MyNetworkMessage();
messageContainer.message = "A new player has conencted to the server";

// Broadcast a message a to everyone connected
NetworkServer.SendToAll(messageID,messageContainer);
}

void OnClientDisconnected(NetworkMessage netMessage)
{
    // Do stuff when a client dissconnects
}

void OnMessageReceived(NetworkMessage netMessage)
{
    // You can send any object that inherence from MessageBase
    // The client and server can be on different projects, as long as the MyNetworkMessage
    or the class you are using have the same implementation on both projects
    // The first thing we do is deserialize the message to our custom type
    var objectMessage = netMessage.ReadMessage<MyNetworkMessage>();
    Debug.Log("Message received: " + objectMessage.message);

}
}

```

Le client

Maintenant, nous créons un client

```

using System;
using UnityEngine;
using UnityEngine.Networking;

public class Client : MonoBehaviour
{
    int port = 9999;
    string ip = "localhost";

    // The id we use to identify our messages and register the handler
    short messageID = 1000;

    // The network client
    NetworkClient client;

    public Client ()
    {
        CreateClient();
    }

    void CreateClient()
    {
        var config = new ConnectionConfig ();

        // Config the Channels we will use
        config.AddChannel (QosType.ReliableFragmented);
    }
}

```

```

config.AddChannel (QosType.UnreliableFragmented);

// Create the client ant attach the configuration
client = new NetworkClient ();
client.Configure (config,1);

// Register the handlers for the different network messages
RegisterHandlers();

// Connect to the server
client.Connect (ip, port);
}

// Register the handlers for the different message types
void RegisterHandlers () {

    // Unity have different Messages types defined in MsgType
    client.RegisterHandler (messageID, OnMessageReceived);
    client.RegisterHandler(MsgType.Connect, OnConnected);
    client.RegisterHandler(MsgType.Disconnect, OnDisconnected);
}

void OnConnected(NetworkMessage message) {
    // Do stuff when connected to the server

    MyNetworkMessage messageContainer = new MyNetworkMessage();
    messageContainer.message = "Hello server!";

    // Say hi to the server when connected
    client.Send(messageID,messageContainer);
}

void OnDisconnected(NetworkMessage message) {
    // Do stuff when disconnected to the server
}

// Message received from the server
void OnMessageReceived(NetworkMessage netMessage)
{
    // You can send any object that inherence from MessageBase
    // The client and server can be on different projects, as long as the MyNetworkMessage
    or the class you are using have the same implementation on both projects
    // The first thing we do is deserialize the message to our custom type
    var objectMessage = netMessage.ReadMessage<MyNetworkMessage>();

    Debug.Log("Message received: " + objectMessage.message);
}
}
}

```

Lire La mise en réseau en ligne: <https://riptutorial.com/fr/unity3d/topic/5671/la-mise-en-reseau>

Chapitre 16: La physique

Examples

Corps rigides

Vue d'ensemble

Le composant Rigidbody donne à un GameObject une *présence physique* dans la scène en ce qu'il est capable de répondre aux forces. Vous pouvez appliquer des forces directement sur le GameObject ou lui permettre de réagir à des forces externes telles que la gravité ou un autre corps rigide qui le frappe.

Ajouter un composant Rigidbody

Vous pouvez ajouter un Rigidbody en cliquant sur **Composant> Physique> Rigidbody**

Déplacement d'un objet Rigidbody

Si vous utilisez un Rigidbody sur un GameObject, il est recommandé d'utiliser des forces ou un couple pour le déplacer plutôt que de le transformer. Utilisez les `AddForce()` ou `AddTorque()` pour cela:

```
// Add a force to the order of myForce in the forward direction of the Transform.  
GetComponent<Rigidbody>().AddForce(transform.forward * myForce);  
  
// Add torque about the Y axis to the order of myTurn.  
GetComponent<Rigidbody>().AddTorque(transform.up * torque * myTurn);
```

Masse

Vous pouvez modifier la masse d'un objet Rigidbody GameObject pour influer sur la façon dont il réagit avec les autres corps rigides et les autres forces. Une masse plus élevée signifie que le GameObject aura plus d'influence sur d'autres GameObjects basés sur la physique et nécessitera une plus grande force pour se déplacer. Les objets de masse différente tomberont au même rythme s'ils ont les mêmes valeurs de glissement. Modifier la masse en code:

```
GetComponent<Rigidbody>().mass = 1000;
```

Traîne

Plus la valeur de glissement est élevée, plus un objet ralentira en se déplaçant. Pensez-y comme une force adverse. Pour modifier le glisser dans le code:

```
GetComponent<Rigidbody>().drag = 10;
```

isKinematic

Si vous marquez un Rigidbody comme **Kinematic**, il ne peut pas être affecté par d'autres forces, mais il peut toujours affecter d'autres GameObjects. Pour modifier dans le code:

```
GetComponent<Rigidbody>().isKinematic = true;
```

Contraintes

Il est également possible d'ajouter des contraintes à chaque axe pour geler la position ou la rotation du corps rigide dans l'espace local. La valeur par défaut est `RigidbodyConstraints.None` comme indiqué ici:



Un exemple de contraintes dans le code:

```
// Freeze rotation on all axes.  
GetComponent<Rigidbody>().constraints = RigidbodyConstraints.FreezeRotation  
  
// Freeze position on all axes.  
GetComponent<Rigidbody>().constraints = RigidbodyConstraints.FreezePosition  
  
// Freeze rotation and motion on all axes.  
GetComponent<Rigidbody>().constraints = RigidbodyConstraints.FreezeAll
```

Vous pouvez utiliser l'opérateur OR au niveau du bit | pour combiner plusieurs contraintes comme ceci:

```
// Allow rotation on X and Y axes and motion on Y and Z axes.  
GetComponent<Rigidbody>().constraints = RigidbodyConstraints.FreezePositionZ |  
    RigidbodyConstraints.FreezeRotationX;
```

Les collisions

Si vous souhaitez qu'un GameObject avec un Rigidbody y réponde, vous devrez également ajouter un collisionneur. Les types de collisionneur sont:

- Boîte collisionneur
- Sphère collisionneur
- Collisionneur de capsules
- Roue collisionneur
- Collisionneur de maille

Si vous appliquez plusieurs collisionneurs à un objet GameObject, nous l'appelons un collisionneur composé.

Vous pouvez créer un collider dans un **déclencheur** afin d'utiliser les `OnTriggerEnter()`, `OnTriggerStay()` et `OnTriggerExit()`. Un collisionneur de déclenchement ne réagira pas physiquement aux collisions, les autres GameObjects le traversent simplement. Ils sont utiles pour détecter lorsqu'un autre objet GameObject se trouve dans une zone donnée ou non, par exemple, lors de la collecte d'un élément, nous pouvons être en mesure de le parcourir, mais détecter quand cela se produit.

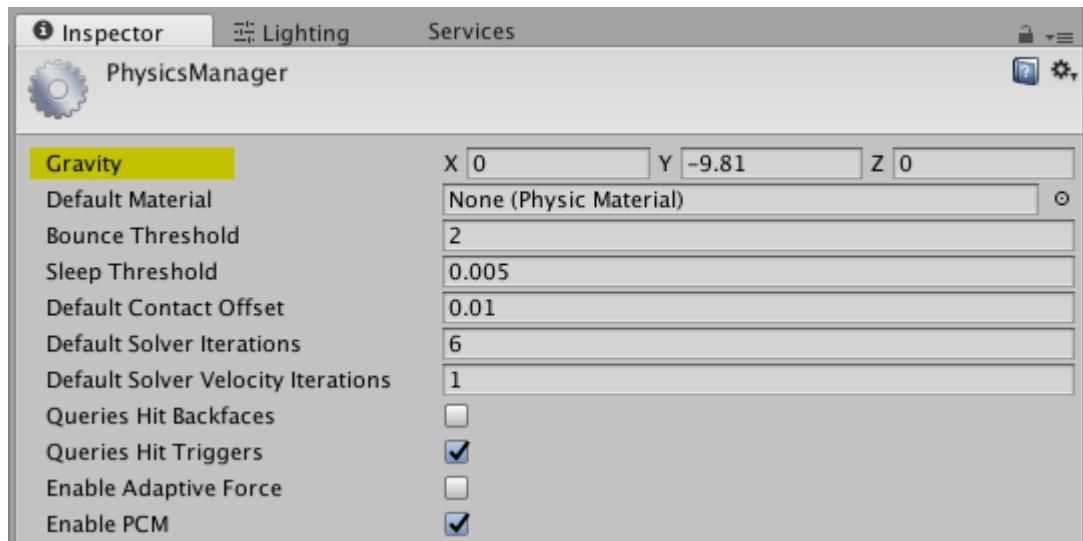
Gravité dans un corps rigide

La propriété `useGravity` d'un `Rigidbody` contrôle si la gravité l'affecte ou non. Si défini sur `false` le `Rigidbody` se comportera comme si dans l'espace (sans qu'une force constante lui soit appliquée dans une certaine direction).

```
GetComponent<Rigidbody>().useGravity = false;
```

C'est très utile dans les situations où vous avez besoin de toutes les autres propriétés de `Rigidbody` à l'exception du mouvement contrôlé par gravité.

Lorsque cette `Rigidbody` est activée, le `Rigidbody` sera affecté par une force gravitationnelle définie **sous Physics Settings**:



La gravité est définie en unités mondiales par seconde au carré et est entrée ici en tant que vecteur tridimensionnel: avec les paramètres de l'image d'exemple, tous les `RigidBodies` avec la propriété `useGravity` définie sur `True` auront une force de 9,81 unités mondiales par seconde. *par seconde* dans la direction vers le bas (lorsque les valeurs Y négatives dans le système de coordonnées d'Unity sont orientées vers le bas).

Lire La physique en ligne: <https://riptutorial.com/fr/unity3d/topic/3680/la-physique>

Chapitre 17: Les attributs

Syntaxe

- [AddComponentMenu (string nomMenu)]
- [AddComponentMenu (string menuName, int order)]
- [CanEditMultipleObjects]
- [ContextMenuItem (nom de chaîne, fonction de chaîne)]
- [ContextMenu (nom de chaîne)]
- [CustomEditor (Type inspectedType)]
- [CustomEditor (Type inspectedType, bool editorForChildClasses)]
- [CustomPropertyDrawer (type de type)]
- [CustomPropertyDrawer (type type, bool useForChildren)]
- [DisallowMultipleComponent]
- [DrawGizmo (GizmoType gizmo)]
- [DrawGizmo (gizmoType, type drawnGizmoType)]
- [ExecuteInEditMode]
- [En-tête (en-tête de chaîne)]
- [HideInInspector]
- [InitializeOnLoad]
- [InitializeOnLoadMethod]
- [MenuItem (string itemName)]
- [MenuItem (string itemName, bool isValidateFunction)]
- [MenuItem (string itemName, bool isValidateFunction, int priority)]
- [Multiligne (lignes int)]
- [PreferenceItem (nom de chaîne)]
- [Plage (float min, float max)]
- [RequireComponent (type type)]
- [RuntimeInitializeOnLoadMethod]
- [RuntimeInitializeOnLoadMethod (RuntimeInitializeLoadType loadType)]
- [SerializeField]
- [Espace (hauteur du flotteur)]
- [TextArea (int minLines, int maxLines)]
- [Info-bulle (info-bulle de chaîne)]

Remarques

SerializeField

Le système de sérialisation d'Unity peut être utilisé pour:

- **Peut** sérialiser des champs non statiques publics (de types sérialisables)
- **Peut** sérialiser des champs non statiques non publics marqués avec l'attribut [SerializeField]

- **Impossible de** sérialiser les champs statiques
- **Impossible de** sérialiser les propriétés statiques

Votre champ, même s'il est marqué avec l'attribut `SerializeField`, ne sera attribué que s'il est d'un type que Unity peut sérialiser, à savoir:

- Toutes les classes héritant de `UnityEngine.Object` (par exemple, `GameObject`, `Component`, `MonoBehaviour`, `Texture2D`)
- Tous les types de données de base comme `int`, `string`, `float`, `bool`
- Certains types intégrés comme `Vector2` / `3/4`, `Quaternion`, `Matrix4x4`, `Color`, `Rect`, `LayerMask`
- Tableaux d'un type sérialisable
- Liste d'un type sérialisable
- Enums
- Structs

Examples

Attributs communs d'inspecteur

```
[Header( "My variables" )]
public string MyString;

[HideInInspector]
public string MyHiddenString;

[Multiline( 5 )]
public string MyMultilineString;

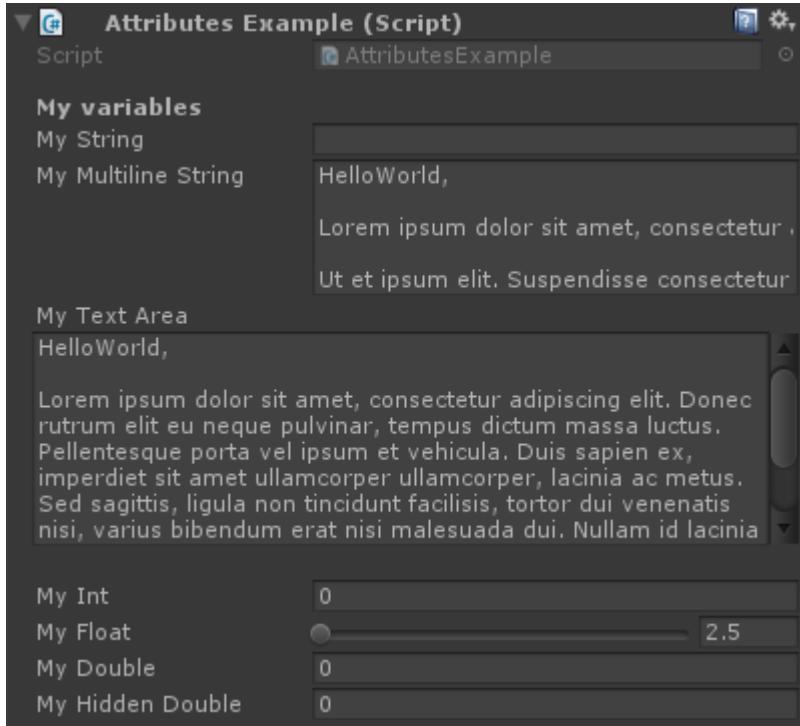
[TextArea( 2, 8 )]
public string MyTextArea;

[Space( 15 )]
public int MyInt;

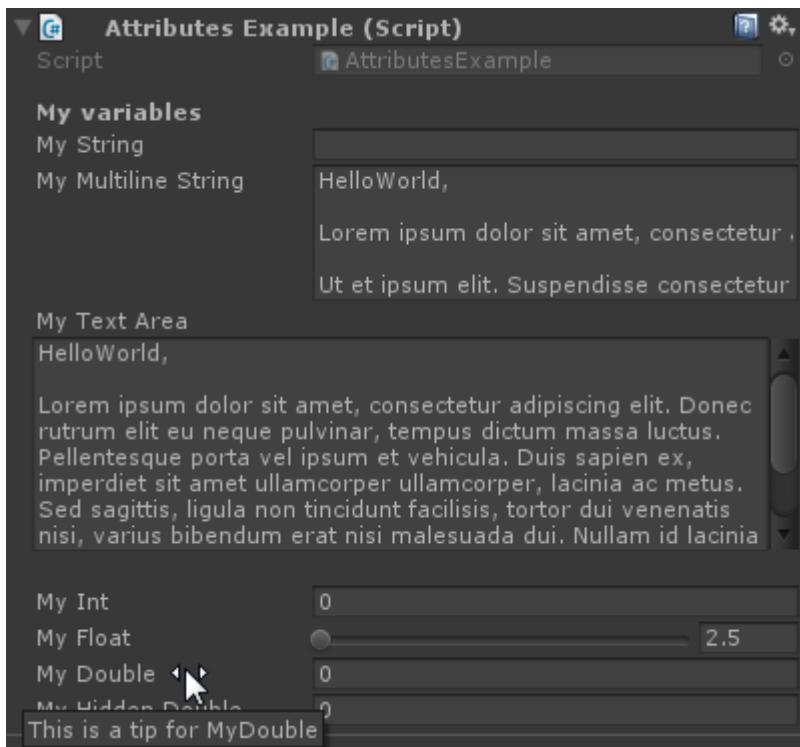
[Range( 2.5f, 12.5f )]
public float MyFloat;

[Tooltip( "This is a tip for MyDouble" )]
public double MyDouble;

[SerializeField]
private double myHiddenDouble;
```



En survolant l'étiquette d'un champ:



```
[Header( "My variables" )]
public string MyString;
```

En-tête place une étiquette en gras contenant le texte au-dessus du champ attribué. Ceci est souvent utilisé pour étiqueter les groupes afin de les différencier des autres labels.

```
[HideInInspector]
public string MyHiddenString;
```

HideInInspector empêche l'affichage des champs publics dans l'inspecteur. Ceci est utile pour accéder aux champs d'autres parties du code où ils ne sont pas visibles ou mutables.

```
[Multiline( 5 )]  
public string MyMultilineString;
```

Multiline crée une zone de texte avec un nombre spécifié de lignes. Si vous dépassiez ce montant, la boîte ne sera ni étendue ni enroulée.

```
[TextArea( 2, 8 )]  
public string MyTextArea;
```

TextArea permet un texte de style multiligne avec des barres de défilement et de **retour automatique à la ligne** si le texte dépasse la zone allouée.

```
[Space( 15 )]  
public int MyInt;
```

L'espace oblige l'inspecteur à ajouter un espace supplémentaire entre les éléments précédents et actuels, ce qui facilite la distinction et la séparation des groupes.

```
[Range( 2.5f, 12.5f )]  
public float MyFloat;
```

Range force une valeur numérique entre un minimum et un maximum. Cet attribut fonctionne également sur les entiers et les doubles, même si min et max sont spécifiés comme des flottants.

```
[Tooltip( "This is a tip for MyDouble" )]  
public double MyDouble;
```

L'infobulle affiche une description supplémentaire chaque fois que l'étiquette du champ est survolée.

```
[SerializeField]  
private double myHiddenDouble;
```

SerializeField force Unity à sérialiser le champ - utile pour les champs privés.

Attributs de composant

```
[DisallowMultipleComponent]  
[RequireComponent( typeof( Rigidbody ) )]  
public class AttributesExample : MonoBehaviour  
{  
    [...]  
}
```

```
[DisallowMultipleComponent]
```

L'attribut `DisallowMultipleComponent` empêche les utilisateurs d'ajouter plusieurs instances de ce composant à un seul `GameObject`.

```
[RequireComponent( typeof( Rigidbody ) )]
```

L'attribut `RequireComponent` vous permet de spécifier un autre composant (ou plusieurs) en tant que conditions requises pour l'ajout de ce composant à un `GameObject`. Lorsque vous ajoutez ce composant à un `GameObject`, les composants requis seront automatiquement ajoutés (s'ils ne sont pas déjà présents) et ces composants ne peuvent pas être supprimés tant que celui qui les nécessite n'est pas supprimé.

Attributs d'exécution

```
[ExecuteInEditMode]
public class AttributesExample : MonoBehaviour
{
    [RuntimeInitializeOnLoadMethod]
    private static void FooBar()
    {
        [...]
    }

    [RuntimeInitializeOnLoadMethod( RuntimeInitializeLoadType.BeforeSceneLoad )]
    private static void Foo()
    {
        [...]
    }

    [RuntimeInitializeOnLoadMethod( RuntimeInitializeLoadType.AfterSceneLoad )]
    private static void Bar()
    {
        [...]
    }

    void Update()
    {
        if ( Application.isEditor )
        {
            [...]
        }
        else
        {
            [...]
        }
    }
}
```

```
[ExecuteInEditMode]
public class AttributesExample : MonoBehaviour
```

L'attribut `ExecuteInEditMode` force Unity à exécuter les méthodes magiques de ce script même lorsque le jeu n'est pas en cours de lecture.

Les fonctions ne sont pas constamment appelées comme en mode lecture

- La mise à jour est uniquement appelée lorsque quelque chose a changé dans la scène.
- OnGUI est appelée lorsque la vue de jeu reçoit un événement.
- OnRenderObject et les autres fonctions de rappel de rendu sont appelées à chaque repeindre de la vue de la scène ou de la vue de jeu.

```
[RuntimeInitializeOnLoadMethod]
private static void FooBar()

[RuntimeInitializeOnLoadMethod( RuntimeInitializeLoadType.BeforeSceneLoad )]
private static void Foo()

[RuntimeInitializeOnLoadMethod( RuntimeInitializeLoadType.AfterSceneLoad )]
private static void Bar()
```

L'attribut RuntimeInitializeOnLoadMethod permet d'appeler une méthode de classe d'exécution lorsque le jeu charge le moteur d'exécution, sans aucune interaction de l'utilisateur.

Vous pouvez spécifier si vous souhaitez que la méthode soit appelée avant ou après le chargement de la scène (la valeur par défaut est après). L'ordre d'exécution n'est pas garanti pour les méthodes utilisant cet attribut.

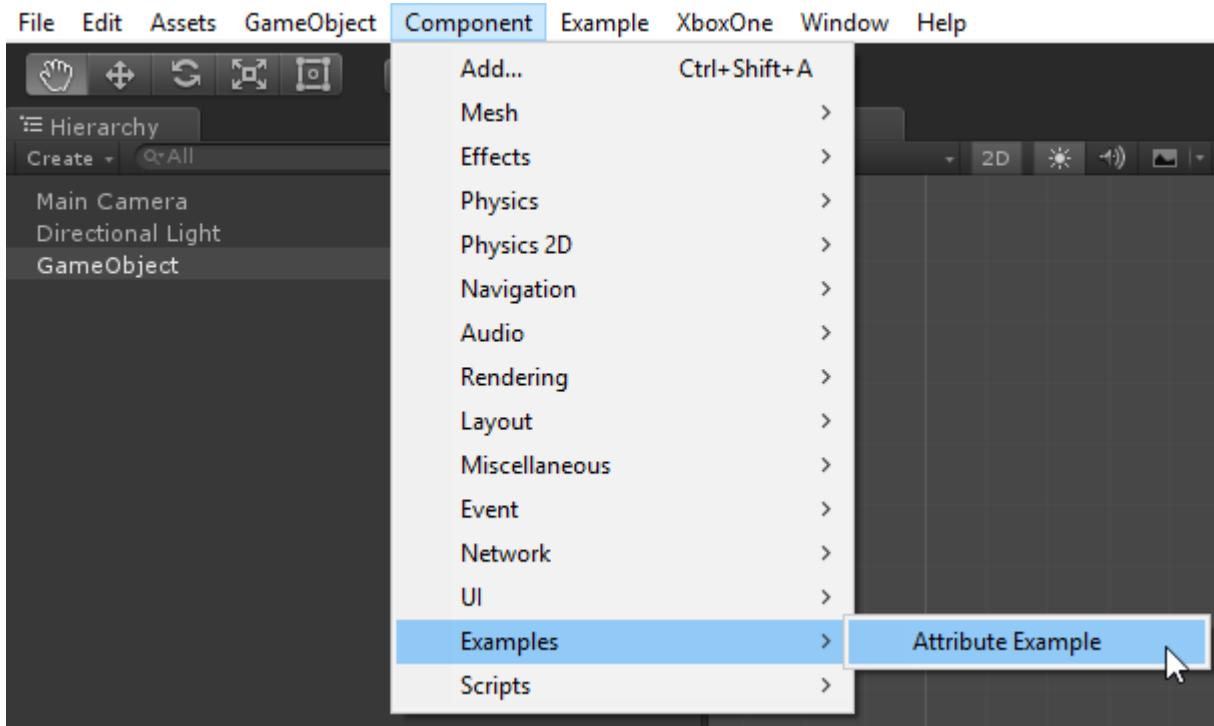
Attributs de menu

```
[AddComponentMenu( "Examples/Attribute Example" )]
public class AttributesExample : MonoBehaviour
{
    [ContextMenuItem( "My Field Action", "MyFieldContextMenuAction" )]
    public string MyString;

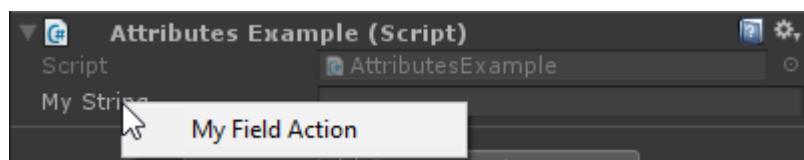
    private void MyFieldContextMenuAction()
    {
        [...]
    }

    [ContextMenu( "My Action" )]
    private void MyContextMenuAction()
    {
        [...]
    }
}
```

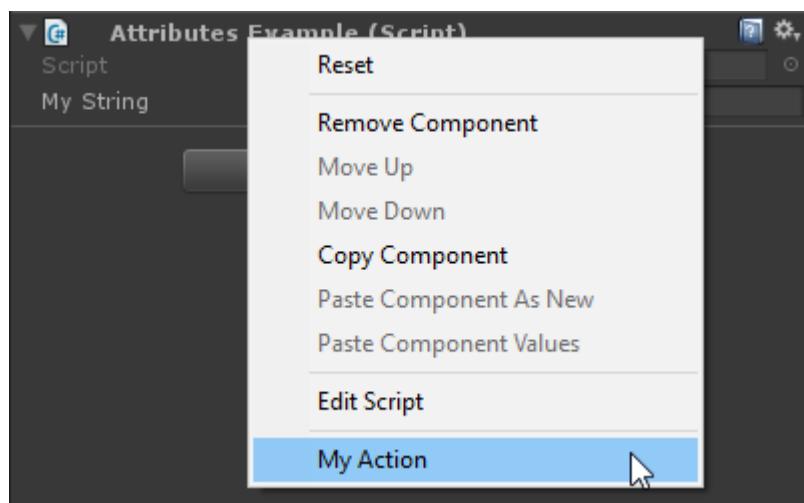
Le résultat de l'attribut [AddComponentMenu]



Le résultat de l'attribut [ContextMenuItem]



Le résultat de l'attribut [ContextMenu]



```
[AddComponentMenu( "Examples/Attribute Example" )]
public class AttributesExample : MonoBehaviour
```

L'attribut AddComponentMenu vous permet de placer votre composant n'importe où dans le menu Component au lieu du menu Component-> Scripts.

```
[ContextMenu( "My Field Action", "MyFieldContextAction" )]
public string MyString;
```

```
private void MyFieldContextMenu()
{
    [...]
}
```

L'attribut `ContextMenu` vous permet de définir des fonctions pouvant être ajoutées au menu contextuel d'un champ. Ces fonctions seront exécutées lors de la sélection.

```
[ContextMenu( "My Action" )]
private void MyContextMenuAction()
{
    [...]
}
```

L'attribut `ContextMenu` vous permet de définir des fonctions pouvant être ajoutées au menu contextuel du composant.

Attributs de l'éditeur

```
[InitializeOnLoad]
public class AttributesExample : MonoBehaviour
{
    static AttributesExample()
    {
        [...]
    }

    [InitializeOnLoadMethod]
    private static void Foo()
    {
        [...]
    }
}
```

```
[InitializeOnLoad]
public class AttributesExample : MonoBehaviour
{
    static AttributesExample()
    {
        [...]
    }
}
```

L'attribut `InitializeOnLoad` permet à l'utilisateur d'initialiser une classe sans aucune interaction de l'utilisateur. Cela se produit chaque fois que l'éditeur se lance ou sur une recompilation. Le constructeur statique garantit que cela sera appelé avant toute autre fonction statique.

```
[InitializeOnLoadMethod]
private static void Foo()
{
    [...]
}
```

```
}
```

L'attribut `InitializeOnLoad` permet à l'utilisateur d'initialiser une classe sans aucune interaction de la part de l'utilisateur. Cela se produit chaque fois que l'éditeur se lance ou sur une recompilation. L'ordre d'exécution n'est pas garanti pour les méthodes utilisant cet attribut.

```
[CanEditMultipleObjects]
public class AttributesExample : MonoBehaviour
{
    public int MyInt;

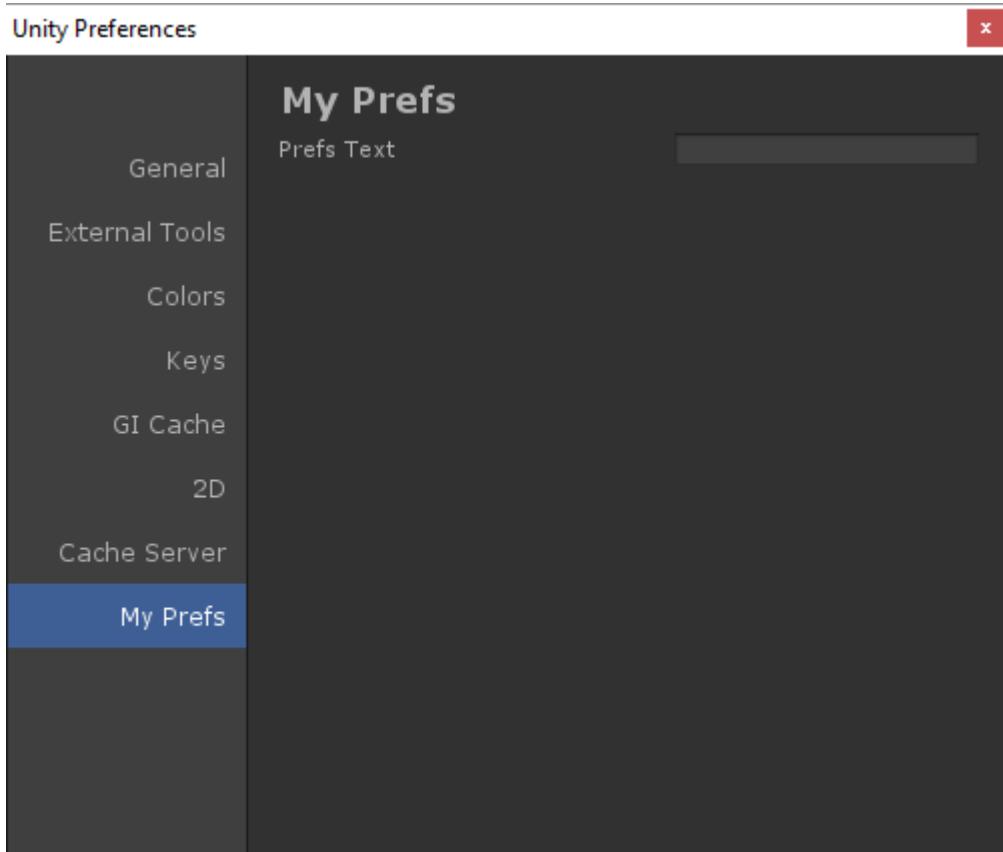
    private static string prefsText = "";

    [PreferenceItem( "My Prefs" )]
    public static void PreferencesGUI()
    {
        prefsText = EditorGUILayout.TextField( "Prefs Text", prefsText );
    }

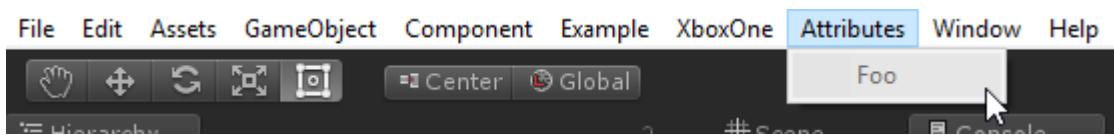
    [MenuItem( "Attributes/Foo" )]
    private static void Foo()
    {
        [...]
    }

    [MenuItem( "Attributes/Foo", true )]
    private static bool FooValidate()
    {
        return false;
    }
}
```

Le résultat de l'attribut `[PreferenceItem]`



Le résultat de l'attribut [MenuItem]



```
[CanEditMultipleObjects]
public class AttributesExample : MonoBehaviour
```

L'attribut CanEditMultipleObjects vous permet de modifier les valeurs de votre composant sur plusieurs GameObjects. Sans ce composant, vous ne verrez pas votre composant apparaître comme normal lors de la sélection de plusieurs GameObjects, mais à la place, vous verrez le message "Modification multi-objets non prise en charge"

Cet attribut est destiné aux éditeurs personnalisés pour prendre en charge la modification multiple. Les éditeurs non personnalisés prennent automatiquement en charge l'édition multiple.

```
[PreferenceItem( "My Prefs" )]
public static void PreferencesGUI()
```

L'attribut PreferenceItem vous permet de créer un élément supplémentaire dans le menu des préférences d'Unity. La méthode de réception doit être statique pour pouvoir être utilisée.

```
[MenuItem( "Attributes/Foo" )]
private static void Foo()
```

```

{
    [...]
}

[MenuItem( "Attributes/Foo", true )]
private static bool FooValidate()
{
    return false;
}

```

L'attribut MenuItem vous permet de créer des éléments de menu personnalisés pour exécuter des fonctions. Cet exemple utilise également une fonction de validation (qui renvoie toujours false) pour empêcher l'exécution de la fonction.

```

[CustomEditor( typeof( MyComponent ) )]
public class AttributesExample : Editor
{
    [...]
}

```

L'attribut CustomEditor vous permet de créer des éditeurs personnalisés pour vos composants. Ces éditeurs seront utilisés pour dessiner votre composant dans l'inspecteur et devront dériver de la classe Editor.

```

[CustomPropertyDrawer( typeof( MyClass ) )]
public class AttributesExample : PropertyDrawer
{
    [...]
}

```

L'attribut CustomPropertyDrawer vous permet de créer un tiroir de propriétés personnalisé dans l'inspecteur. Vous pouvez utiliser ces tiroirs pour vos types de données personnalisés afin qu'ils soient visibles dans l'inspecteur.

```

[DrawGizmo( GizmoType.Selected )]
private static void DoGizmo( AttributesExample obj, GizmoType type )
{
    [...]
}

```

L'attribut DrawGizmo vous permet de dessiner des gadgets personnalisés pour vos composants. Ces gadgets seront dessinés dans la vue de la scène. Vous pouvez décider quand dessiner le gizmo en utilisant le paramètre GizmoType dans l'attribut DrawGizmo.

La méthode de réception nécessite deux paramètres, le premier est le composant pour lequel dessiner le gizmo et le second est l'état dans lequel se trouve l'objet qui a besoin du gizmo dessiné.

Lire Les attributs en ligne: <https://riptutorial.com/fr/unity3d/topic/5535/les-attributs>

Chapitre 18: Modèles de conception

Exemples

Modèle de conception MVC (Model View Controller)

Le contrôleur de vue de modèle est un modèle de conception très courant qui existe depuis un certain temps. Ce modèle vise à réduire le code des *spaghettis* en séparant les classes en parties fonctionnelles. Récemment, j'ai expérimenté ce motif dans Unity et je voudrais donner un exemple de base.

Une conception MVC se compose de trois parties principales: modèle, vue et contrôleur.

Modèle: Le modèle est une classe représentant la partie données de votre objet. Cela pourrait être un joueur, un inventaire ou un niveau entier. Si programmé correctement, vous devriez pouvoir prendre ce script et l'utiliser en dehors de Unity.

Notez quelques points à propos du modèle:

- Il ne devrait pas hériter de Monobehaviour
- Il ne doit pas contenir de code spécifique à Unity pour la portabilité
- Comme nous évitons les appels d'API Unity, cela peut entraîner des choses comme les convertisseurs implicites dans la classe Model (des solutions de contournement sont requises)

Player.cs

```
using System;

public class Player
{
    public delegate void PositionEvent(Vector3 position);
    public event PositionEvent OnPositionChanged;

    public Vector3 position
    {
        get
        {
            return _position;
        }
        set
        {
            if (_position != value) {
                _position = value;
                if (OnPositionChanged != null) {
                    OnPositionChanged(value);
                }
            }
        }
    }
    private Vector3 _position;
```

```
}
```

Vector3.cs

Une classe Vector3 personnalisée à utiliser avec notre modèle de données.

```
using System;

public class Vector3
{
    public float x;
    public float y;
    public float z;

    public Vector3(float x, float y, float z)
    {
        this.x = x;
        this.y = y;
        this.z = z;
    }
}
```

Affichage: la vue est une classe représentant la partie de visualisation liée au modèle. Ceci est une classe appropriée à dériver de Monobehaviour. Cela devrait contenir du code qui interagit directement avec les API spécifiques à Unity, y compris `OnCollisionEnter`, `Start`, `Update`, etc.

- Hérite généralement de Monobehaviour
- Contient un code spécifique à Unity

PlayerView.cs

```
using UnityEngine;

public class PlayerView : MonoBehaviour
{
    public void SetPosition(Vector3 position)
    {
        transform.position = position;
    }
}
```

Contrôleur: le contrôleur est une classe qui lie le modèle et la vue. Les contrôleurs conservent à la fois le modèle et la vue en synchronisation, ainsi que l'interaction entre les partenaires. Le contrôleur peut écouter les événements de l'un ou l'autre partenaire et les mettre à jour en conséquence.

- Lie le modèle et la vue en synchronisant l'état
- Peut générer une interaction entre les partenaires
- Les contrôleurs peuvent être ou ne pas être portables (vous devrez peut-être utiliser le code Unity ici)
- Si vous décidez de ne pas rendre votre contrôleur portable, envisagez d'en faire un Monobehaviour pour faciliter l'inspection de l'éditeur.

PlayerController.cs

```
using System;

public class PlayerController
{
    public Player model { get; private set; }
    public PlayerView view { get; private set; }

    public PlayerController(Player model, PlayerView view)
    {
        this.model = model;
        this.view = view;

        this.model.OnPositionChanged += OnPositionChanged;
    }

    private void OnPositionChanged(Vector3 position)
    {
        // Sync
        Vector3 pos = this.model.position;

        // Unity call required here! (we lost portability)
        this.view.SetPosition(new UnityEngine.Vector3(pos.x, pos.y, pos.z));
    }

    // Calling this will fire the OnPositionChanged event
    private void SetPosition(Vector3 position)
    {
        this.model.position = position;
    }
}
```

Utilisation finale

Maintenant que nous avons toutes les pièces principales, nous pouvons créer une usine qui générera les trois parties.

PlayerFactory.cs

```
using System;

public class PlayerFactory
{
    public PlayerController controller { get; private set; }
    public Player model { get; private set; }
    public PlayerView view { get; private set; }

    public void Load()
    {
        // Put the Player prefab inside the 'Resources' folder
        // Make sure it has the 'PlayerView' Component attached
        GameObject prefab = Resources.Load<GameObject>("Player");
        GameObject instance = GameObject.Instantiate<GameObject>(prefab);
        this.model = new Player();
        this.view = instance.GetComponent<PlayerView>();
        this.controller = new PlayerController(model, view);
    }
}
```

```
    }  
}
```

Et enfin, nous pouvons appeler l'usine d'un directeur ...

Manager.cs

```
using UnityEngine;  
  
public class Manager : MonoBehaviour  
{  
    [ContextMenu("Load Player")]  
    private void LoadPlayer()  
    {  
        new PlayerFactory().Load();  
    }  
}
```

Joignez le script Manager à un GameObject vide dans la scène, cliquez avec le bouton droit sur le composant et sélectionnez "Load Player".

Pour une logique plus complexe, vous pouvez introduire l'héritage avec des classes de base abstraites et des interfaces pour une architecture améliorée.

Lire Modèles de conception en ligne: <https://riptutorial.com/fr/unity3d/topic/10842/modeles-de-conception>

Chapitre 19: Mots clés

Introduction

Une balise est une chaîne pouvant être appliquée pour marquer les types `GameObject`. De cette façon, il est plus facile d'identifier des objets `GameObject` particuliers via du code.

Une balise peut être appliquée à un ou plusieurs objets de jeu, mais un objet de jeu ne comporte toujours qu'une seule balise. Par défaut, la balise "*Untagged*" est utilisée pour représenter un objet `GameObject` qui n'a pas été intentionnellement balisé.

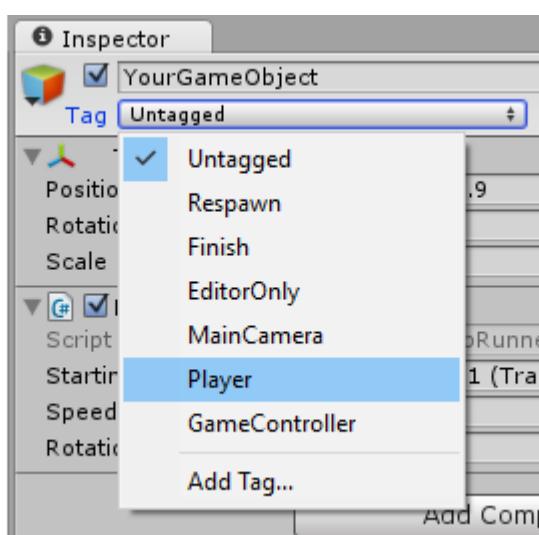
Exemples

Création et application de balises

Les balises sont généralement appliquées via l'éditeur; Cependant, vous pouvez également appliquer des balises via un script. Toute balise personnalisée doit être créée via la fenêtre *Balises et calques* avant d'être appliquée à un objet de jeu.

Définition de balises dans l'éditeur

Avec un ou plusieurs objets de jeu sélectionnés, vous pouvez sélectionner une étiquette de l'inspecteur. Les objets de jeu porteront toujours une seule balise; Par défaut, les objets du jeu seront marqués comme "*non marqués*". Vous pouvez également passer à la fenêtre *Balises et calques* en sélectionnant "*Ajouter une balise ...*"; Cependant, il est important de noter que cela vous amène uniquement à la fenêtre *Balises et calques*. Tout tag que vous créez *ne s'appliquera pas* automatiquement à l'objet de jeu.



Définition de balises via un script

Vous pouvez directement modifier une balise d'objets de jeu via un code. Il est important de noter que vous devez fournir une balise à partir de la liste des balises actuelles. Si vous fournissez un tag qui n'a pas encore été créé, cela entraînera une erreur.

Comme détaillé dans d'autres exemples, l'utilisation d'une série de variables de `static string`, par opposition à l'écriture manuelle de chaque balise, peut garantir la cohérence et la fiabilité.

Le script suivant montre comment modifier une série de balises d'objets de jeu, en utilisant `static string` références de `static string` pour assurer la cohérence. Notez que chaque `static string` représente une balise déjà créée dans la fenêtre *Balises et calques*.

```
using UnityEngine;

public class Tagging : MonoBehaviour
{
    static string tagUntagged = "Untagged";
    static string tagPlayer = "Player";
    static string tagEnemy = "Enemy";

    /// <summary>Represents the player character. This game object should
    /// be linked up via the inspector.</summary>
    public GameObject player;
    /// <summary>Represents all the enemy characters. All enemies should
    /// be added to the array via the inspector.</summary>
    public GameObject[] enemy;

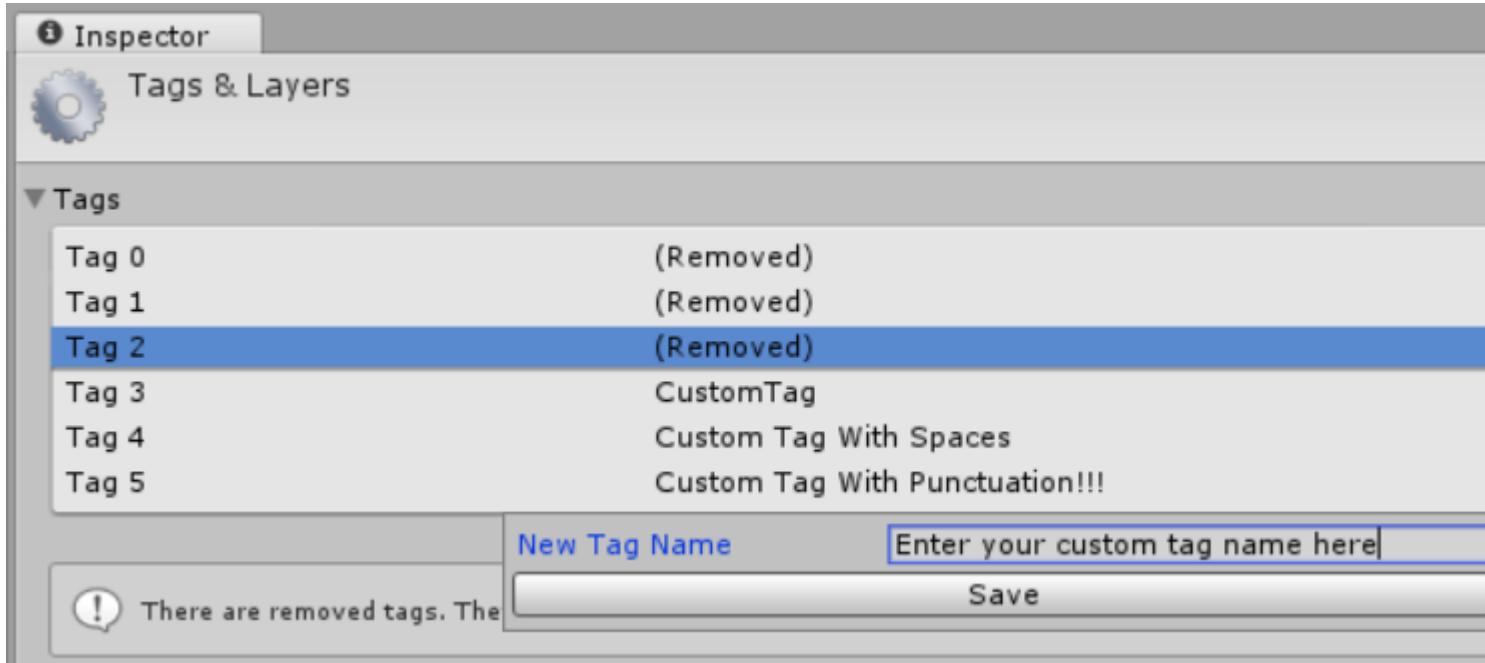
    void Start ()
    {
        // We ensure that the game object this script is attached to
        // is left untagged by using the default "Untagged" tag.
        gameObject.tag = tagUntagged;

        // We ensure the player has the player tag.
        player.tag = tagUntagged;

        // We loop through the enemy array to ensure they are all tagged.
        for(int i = 0; i < enemy.Length; i++)
        {
            enemy[i].tag = tagEnemy;
        }
    }
}
```

Création de balises personnalisées

Que vous définissiez des balises via l'inspecteur ou via un script, les balises doivent être déclarées via la fenêtre *Balises et calques* avant d'être utilisées. Vous pouvez accéder à cette fenêtre en sélectionnant "Ajouter des balises ..." dans le menu déroulant des balises d'objets de jeu. Vous pouvez également trouver la fenêtre sous **Edition> Paramètres du projet> Balises et calques**.



Sélectionnez simplement le bouton + , entrez le nom souhaité et sélectionnez Enregistrer pour créer un tag. En sélectionnant le bouton - , vous supprimez l'étiquette en surbrillance. Notez que de cette manière, le tag sera immédiatement affiché comme "(Supprimé)" , et sera complètement supprimé lors du prochain rechargeement du projet.

Sélectionner l'engrenage / cog en haut à droite de la fenêtre vous permettra de réinitialiser toutes les options personnalisées. Cela supprimera immédiatement toutes les balises personnalisées, ainsi que tout calque personnalisé que vous pouvez avoir sous "Tri des calques" et "Calques" .

Recherche de GameObjects par tag:

Les balises facilitent la localisation d'objets de jeu spécifiques. Nous pouvons rechercher un seul objet de jeu ou en chercher plusieurs.

Trouver un seul GameObject

Nous pouvons utiliser la fonction statique `GameObject.FindGameObjectWithTag(string tag)` pour rechercher des objets de jeu individuels. Il est important de noter que, de cette manière, les objets de jeu ne sont interrogés dans aucun ordre particulier. Si vous recherchez une étiquette qui est utilisé sur des objets de jeu multiples dans la scène, cette fonction ne sera pas en mesure de garantir quel objet jeu est retourné. En tant que tel, il est plus approprié quand on sait que seul *un* objet de jeu utilise une telle étiquette, ou lorsque nous ne sommes pas préoccupés par l'instance exacte de `GameObject` qui est retourné.

```
//<summary>We create a static string to allow us consistency.</summary>
string playerTag = "Player"
```

```
//<summary>We can now use the tag to reference our player GameObject.</summary>
GameObject player = GameObject.FindGameObjectWithTag(playerTag);
```

Recherche d'un tableau d'instances GameObject

Nous pouvons utiliser la fonction statique `GameObject.FindGameObjectsWithTag(string tag)` pour rechercher *tous* les objets de jeu qui utilisent une balise particulière. Ceci est utile quand on veut parcourir un groupe d'objets de jeu particuliers. Cela peut également être utile si vous souhaitez trouver un *seul* objet de jeu, mais que *plusieurs* objets de jeu peuvent utiliser la même balise. Comme nous ne pouvons pas garantir l'instance exacte renvoyée par `GameObject.FindGameObjectWithTag(string tag)`, nous devons plutôt récupérer un tableau de toutes les instances potentielles de `GameObject` avec `GameObject.FindGameObjectsWithTag(string tag)` et analyser plus en détail le tableau résultant à la recherche de.

```
///<summary>We create a static string to allow us consistency.</summary>
string enemyTag = "Enemy";

///<summary>We can now use the tag to create an array of all enemy GameObjects.</summary>
GameObject[] enemies = GameObject.FindGameObjectsWithTag(enemyTag);

// We can now freely iterate through our array of enemies
foreach(GameObject enemy in enemies)
{
    // Do something to each enemy (link up a reference, check for damage, etc.)
}
```

Comparaison de balises

Lorsque vous comparez deux `GameObjects` par des balises, il convient de noter que ce qui suit entraînerait une surcharge de Garbage Collector car une chaîne est créée à chaque fois:

```
if (go.Tag == "myTag")
{
    //Stuff
}
```

Lorsque vous effectuez ces comparaisons dans `Update()` et dans un autre callback Unity (ou une boucle), vous devez utiliser cette méthode sans allocation de tas:

```
if (go.CompareTag("myTag"))
{
    //Stuff
}
```

De plus, il est plus facile de conserver vos tags dans une classe statique.

```
public static class Tags
{
    public const string Player = "Player";
    public const string MyCustomTag = "MyCustomTag";
}
```

Ensuite, vous pouvez comparer en toute sécurité

```
if (go.CompareTag(Tags.MyCustomTag)
{
    //Stuff
}
```

De cette façon, vos chaînes de mots-clés sont générées au moment de la compilation et vous limitez les implications des fautes d'orthographe.

Tout comme conserver les balises dans une classe statique, il est également possible de les stocker dans une énumération:

```
public enum Tags
{
    Player, Ennemis, MyCustomTag;
}
```

et puis vous pouvez le comparer en utilisant la `ToString()` `enum ToString()`:

```
if (go.CompareTag(Tags.MyCustomTag.ToString()))
{
    //Stuff
}
```

Lire Mots clés en ligne: <https://riptutorial.com/fr/unity3d/topic/5534/mots-cles>

Chapitre 20: Optimisation

Remarques

1. Si possible, désactivez les scripts sur les objets lorsqu'ils ne sont pas nécessaires. Par exemple, si vous avez un script sur un objet ennemi que les utilisateurs recherchent et déclenchent sur le joueur, envisagez de désactiver ce script lorsque l'ennemi est trop éloigné du joueur, par exemple.

Exemples

Vérifications rapides et efficaces

Évitez les opérations inutiles et les appels de méthodes partout où vous le pouvez, en particulier dans une méthode appelée plusieurs fois par seconde, comme la `Update` à `Update`.

Contrôle de distance / distance

Utilisez `sqrMagnitude` au lieu de `magnitude` lorsque vous comparez les distances. Cela évite inutiles `sqrt` opérations. Notez que lorsque vous utilisez `sqrMagnitude`, le côté droit doit également être mis au carré.

```
if ((target.position - transform.position).sqrMagnitude < minDistance * minDistance)
```

Chèques de Limites

Les intersections d'objets peuvent être grossièrement vérifiées en vérifiant si leurs limites `Collider` / `Renderer` croisent. La structure `Bounds` a également une méthode pratique `Intersects` qui permet de déterminer si deux bornes se croisent.

`Bounds` nous aident également à calculer une approximation proche de la distance *réelle* (surface à surface) entre les objets (voir `Bounds.SqrDistance`).

Mises en garde

La vérification des limites fonctionne très bien pour les objets convexes, mais les vérifications des limites sur les objets concaves peuvent entraîner des inexactitudes beaucoup plus importantes selon la forme de l'objet.

L'utilisation de `Mesh.bounds` n'est pas recommandée car elle renvoie des limites d'espace locales. Utilisez `MeshRenderer.bounds` place.

Puissance de coroutine

Usage

Si vous exécutez une opération longue qui repose sur l'API Unity non [compatible](#) avec les [threads](#) , utilisez [Coroutines](#) pour la diviser sur plusieurs cadres et garder votre application réactive.

[Les coroutines](#) permettent également d'effectuer des actions coûteuses à chaque nième image au lieu d'exécuter cette action à chaque image.

Fractionnement de routines de longue durée sur plusieurs images

Coroutines permet de distribuer des opérations de longue durée sur plusieurs cadres afin de maintenir le framerate de votre application.

Les routines qui peignent ou génèrent du terrain de manière procédurale ou génèrent du bruit sont des exemples qui peuvent nécessiter le traitement Coroutine.

```
for (int y = 0; y < heightmap.Height; y++)
{
    for (int x = 0; x < heightmap.Width; x++)
    {
        // Generate pixel at (x, y)
        // Assign pixel at (x, y)

        // Process only 32768 pixels each frame
        if ((y * heightmap.Height + x) % 32 * 1024) == 0)
            yield return null; // Wait for next frame
    }
}
```

Le code ci-dessus est un exemple facile à comprendre. Dans le code de production , il est préférable d'éviter le contrôle par pixel qui vérifie quand `yield return` le `for` `yield return` (peut - être faire tous les 2-3 lignes) et d' effectuer une pré-calcul `for` la longueur de boucle à l' avance.

Effectuer des actions coûteuses moins souvent

Les Coroutines vous aident à effectuer des actions coûteuses moins fréquemment, de sorte qu'elles ne sont pas aussi importantes que si elles étaient exécutées à chaque image.

En prenant l'exemple suivant directement à partir du [manuel](#) :

```

private void ProximityCheck()
{
    for (int i = 0; i < enemies.Length; i++)
    {
        if (Vector3.Distance(transform.position, enemies[i].transform.position) <
dangerDistance)
            return true;
    }
    return false;
}

private IEnumerator ProximityCheckCoroutine()
{
    while(true)
    {
        ProximityCheck();
        yield return new WaitForSeconds(.1f);
    }
}

```

Les tests de proximité peuvent être optimisés en utilisant l' [API CullingGroup](#) .

Pièges courants

Une erreur courante des développeurs est d'accéder aux résultats ou aux effets secondaires des coroutines en *dehors* de la coroutine. Coroutines retourne le contrôle à l'appelant dès qu'un relevé de `yield return` est rencontré et que le résultat ou l'effet secondaire peut ne pas encore être exécuté. Pour contourner les problèmes où vous *devez* utiliser le résultat / effet secondaire en dehors de la coroutine, cochez [cette réponse](#) .

Cordes

On pourrait faire valoir qu'il ya plus de porcs dans l'Unité que l'humble cordelette, mais c'est l'un des aspects les plus faciles à résoudre dès le départ.

Les opérations de type string créent des ordures

La plupart des opérations sur les chaînes génèrent des quantités infimes de déchets, mais si ces opérations sont appelées plusieurs fois au cours d'une seule mise à jour, elles se cumulent. Au fil du temps, cela déclenchera le nettoyage de la mémoire automatique, ce qui peut entraîner une pointe de processeur visible.

Cachez vos opérations de chaîne

Prenons l'exemple suivant.

```

string[] StringKeys = new string[] {
    "Key0",
    "Key1",
    "Key2"
};

void Update()
{
    for (var i = 0; i < 3; i++)
    {
        // Cached, no garbage generated
        Debug.Log(StringKeys[i]);
    }

    for (var i = 0; i < 3; i++)
    {
        // Not cached, garbage every cycle
        Debug.Log("Key" + i);
    }

    // The most memory-efficient way is to not create a cache at all and use literals or
    constants.
    // However, it is not necessarily the most readable or beautiful way.
    Debug.Log("Key0");
    Debug.Log("Key1");
    Debug.Log("Key2");
}

```

Cela peut sembler idiot et redondant, mais si vous travaillez avec Shaders, vous pourriez rencontrer de telles situations. Cacher les clés fera la différence.

Veuillez noter que les *littéraux de chaîne* et les *constantes* ne génèrent aucune erreur, car ils sont injectés de manière statique dans l'espace de pile du programme. Si vous générez des chaînes au moment de l'exécution et que vous générez les **mêmes** chaînes à chaque fois, comme dans l'exemple ci-dessus, la mise en cache sera certainement utile.

Pour les autres cas où la chaîne générée n'est pas la même à chaque fois, il n'y a pas d'autre alternative à la génération de ces chaînes. En tant que tel, le pic de mémoire avec la génération manuelle de chaînes à chaque fois est généralement négligeable, à moins que des dizaines de milliers de chaînes ne soient générées à la fois.

La plupart des opérations de chaîne sont des messages de débogage

Faire des opérations de chaîne pour les messages de débogage, c.-à-d. `Debug.Log("Object Name: " + obj.name)` **est** `Debug.Log("Object Name: " + obj.name)` et ne peut être évité pendant le développement. Il est toutefois important de veiller à ce que les messages de débogage non pertinents ne se retrouvent pas dans le produit publié.

L'une des méthodes consiste à utiliser l' **attribut Conditionnel** dans vos appels de débogage. Cela non seulement supprime les appels de méthode, mais aussi toutes les opérations de chaîne qui y entrent.

```

using UnityEngine;
using System.Collections;

public class ConditionalDebugExample: MonoBehaviour
{
    IEnumerator Start()
    {
        while(true)
        {
            // This message will pop up in Editor but not in builds
            Log("Elapsed: " + Time.timeSinceLevelLoad);
            yield return new WaitForSeconds(1f);
        }
    }

    [System.Diagnostics.Conditional("UNITY_EDITOR")]
    void Log(string Message)
    {
        Debug.Log(Message);
    }
}

```

Ceci est un exemple simplifié. Vous voudrez peut-être investir du temps dans la conception d'une routine de journalisation à part entière.

Comparaison de chaîne

Ceci est une optimisation mineure, mais cela vaut la peine d'être mentionné. La comparaison des chaînes est légèrement plus complexe qu'on pourrait le penser. Le système essaiera de prendre en compte les différences culturelles par défaut. Vous pouvez choisir d'utiliser une simple comparaison binaire, plus rapide.

```

// Faster string comparison
if (strA.Equals(strB, System.StringComparison.OrdinalIgnoreCase)) {...}
// Compared to
if (strA == strB) {...}

// Less overhead
if (!string.IsNullOrEmpty(strA)) {...}
// Compared to
if (strA == "") {...}

// Faster lookups
Dictionary<string, int> myDic = new Dictionary<string, int>(System.StringComparer.OrdinalIgnoreCase);
// Compared to
Dictionary<string, int> myDictionary = new Dictionary<string, int>();

```

Références du cache

Cacher les références pour éviter les appels coûteux, en particulier dans la fonction de mise à jour. Cela peut être fait en mettant en cache ces références au démarrage si elles sont disponibles ou lorsqu'elles sont disponibles et en recherchant null / bool flat pour éviter d'obtenir à nouveau la référence.

Exemples:

Références du composant de cache

changement

```
void Update()
{
    var renderer = GetComponent<Renderer>();
    renderer.material.SetColor("_Color", Color.green);
}
```

à

```
private Renderer myRenderer;
void Start()
{
    myRenderer = GetComponent<Renderer>();
}

void Update()
{
    myRenderer.material.SetColor("_Color", Color.green);
}
```

Références d'objet cache

changement

```
void Update()
{
    var enemy = GameObject.Find("enemy");
    enemy.transform.LookAt(new Vector3(0, 0, 0));
}
```

à

```
private Transform enemy;

void Start()
{
    this.enemy = GameObject.Find("enemy").transform;
}

void Update()
{
    enemy.LookAt(new Vector3(0, 0, 0));
}
```

De plus, cachez les appels coûteux comme les appels à Mathf dans la mesure du possible.

Évitez d'appeler des méthodes à l'aide de chaînes

Évitez d'appeler des méthodes en utilisant des chaînes pouvant accepter des méthodes. Cette

approche utilisera la réflexion qui peut ralentir votre jeu, en particulier lorsqu'il est utilisé dans la fonction de mise à jour.

Exemples:

```
//Avoid StartCoroutine with method name  
this.StartCoroutine("SampleCoroutine");  
  
//Instead use the method directly  
this.StartCoroutine(this.SampleCoroutine());  
  
//Avoid send message  
var enemy = GameObject.Find("enemy");  
enemy.SendMessage("Die");  
  
//Instead make direct call  
var enemy = GameObject.Find("enemy") as Enemy;  
enemy.Die();
```

Éviter les méthodes d'unité vide

Évitez les méthodes d'unité vide. En plus d'être un mauvais style de programmation, les scripts d'exécution impliquent une surcharge minime. Dans de nombreux cas, cela peut augmenter et affecter les performances.

```
void Update  
{  
}  
  
void FixedUpdate  
{  
}
```

Lire Optimisation en ligne: <https://riptutorial.com/fr/unity3d/topic/3433/optimisation>

Chapitre 21: Plates-formes mobiles

Syntaxe

- public static int Input.touchCount
- Touch statique public Input.GetTouch (int index)

Exemples

Déetecter le toucher

Pour détecter un contact dans Unity, il est assez simple d'utiliser `Input.GetTouch()` et de lui transmettre un index.

```
using UnityEngine;
using System.Collections;

public class TouchExample : MonoBehaviour {
    void Update() {
        if (Input.touchCount > 0 && Input.GetTouch(0).phase == TouchPhase.Began)
        {
            //Do Stuff
        }
    }
}
```

ou

```
using UnityEngine;
using System.Collections;

public class TouchExample : MonoBehaviour {
    void Update() {
        for(int i = 0; i < Input.touchCount; i++)
        {
            if (Input.GetTouch(i).phase == TouchPhase.Began)
            {
                //Do Stuff
            }
        }
    }
}
```

Ces exemples prennent le contact de la dernière image de jeu.

TouchPhase

À l'intérieur du `TouchPhase` enum, il existe 5 types différents de `TouchPhase`

- A commencé - un doigt a touché l'écran
- Déplacé - un doigt déplacé sur l'écran
- Stationnaire - un doigt est sur l'écran mais ne bouge pas
- Terminé - un doigt a été levé de l'écran
- Annulé - le système a annulé le suivi pour le toucher

Par exemple, pour déplacer l'objet auquel ce script est attaché sur l'écran en fonction du toucher.

```
public class TouchMoveExample : MonoBehaviour
{
    public float speed = 0.1f;

    void Update () {
        if(Input.touchCount > 0 && Input.GetTouch(0).phase == TouchPhase.Moved)
        {
            Vector2 touchDeltaPosition = Input.GetTouch(0).deltaPosition;
            transform.Translate(-touchDeltaPosition.x * speed, -touchDeltaPosition.y * speed,
0);
        }
    }
}
```

Lire Plates-formes mobiles en ligne: <https://riptutorial.com/fr/unity3d/topic/6285/plates-formes-mobiles>

Chapitre 22: Plugins Android 101 - Une introduction

Introduction

Cette rubrique est la première partie d'une série sur la création de plug-ins Android pour Unity. Commencez ici si vous avez peu ou pas d'expérience dans la création de plug-ins et / ou le système d'exploitation Android.

Remarques

A travers cette série, j'utilise intensivement des liens externes que je vous encourage à lire. Bien que les versions paraphrasées du contenu concerné soient incluses ici, il peut arriver que la lecture supplémentaire aide.

Commencant par les plugins Android

Actuellement, Unity propose deux méthodes pour appeler un code Android natif.

1. Écrivez le code natif d'Android en Java et appelez ces fonctions Java en utilisant C #
2. Ecrire du code C # pour appeler directement les fonctions faisant partie du système d'exploitation Android

Pour interagir avec du code natif, Unity fournit des classes et des fonctions.

- [AndroidJavaObject](#) - Il s'agit de la classe de base fournie par Unity pour interagir avec le code natif. Presque tous les objets renvoyés à partir du code natif peuvent être stockés en tant que et `AndroidJavaObject`
- [AndroidJavaClass](#) - Hérite de `AndroidJavaObject`. Ceci est utilisé pour référencer les classes dans votre code natif
- [Obtenir / Définir](#) les valeurs d'une instance d'un objet natif et les [versions](#) statiques de [GetStatic](#) / [SetStatic](#)
- [Appel / CallStatic](#) pour appeler des fonctions natives non statiques et statiques

Aperçu de la création d'un plugin et de la terminologie

1. Ecrire du code Java natif dans [Android Studio](#)
2. Exportez le code dans un fichier JAR / AAR (étapes ici pour les [fichiers JAR](#) et les [fichiers AAR](#))
3. Copiez le fichier JAR / AAR dans votre projet Unity sur **Assets / Plugins / Android**

4. Ecrire du code dans Unity (C # a toujours été la voie à suivre ici) pour appeler des fonctions dans le plugin

Notez que les trois premières étapes s'appliquent UNIQUEMENT si vous souhaitez avoir un plugin natif!

À partir de là, je ferai référence au fichier JAR / AAR en tant que **plug - in natif** et au script C # en tant que **wrapper C #**

Choisir entre les méthodes de création de plugins

Il est immédiatement évident que la première façon de créer des plugins est longue, donc choisir votre itinéraire semble inutile. Cependant, la méthode 1 est la seule façon d'appeler le code personnalisé. Alors, comment choisir?

En termes simples, votre plugin

1. Impliquer le code personnalisé - Choisir la méthode 1
2. Invoquer uniquement des fonctions Android natives? - Choisissez la méthode 2

S'il vous plaît, n'essayez **PAS** de "mélanger" (c'est-à-dire une partie du plugin en utilisant la méthode 1, et l'autre en utilisant la méthode 2) les deux méthodes! Bien que cela soit tout à fait possible, il est souvent difficile et difficile à gérer.

Exemples

UnityAndroidPlugin.cs

Créez un nouveau script C # dans Unity et remplacez son contenu par le suivant

```
using UnityEngine;
using System.Collections;

public static class UnityAndroidPlugin {

}
```

UnityAndroidNative.java

Créez une nouvelle classe Java dans Android Studio et remplacez son contenu par le suivant

```
package com.aks.unityandroidplugin;
import android.util.Log;
import android.widget.Toast;
import android.app.ActivityManager;
import android.content.Context;
```

```
public class UnityAndroidNative {  
  
}
```

UnityAndroidPluginGUI.cs

Créez un nouveau script C # dans Unity et collez ces contenus

```
using UnityEngine;  
using System.Collections;  
  
public class UnityAndroidPluginGUI : MonoBehaviour {  
  
    void OnGUI () {  
  
    }  
  
}
```

Lire Plugins Android 101 - Une introduction en ligne:

<https://riptutorial.com/fr/unity3d/topic/10032/plugins-android-101---une-introduction>

Chapitre 23: Préfabriqués

Syntaxe

- objet statique public PrefabUtility.InstantiatePrefab (Object target);
- objet statique public AssetDatabase.LoadAssetAtPath (string assetPath, type type);
- objet statique public Instantiate (objet d'origine);
- public static Object Resources.Load (chemin de chaîne);

Examples

introduction

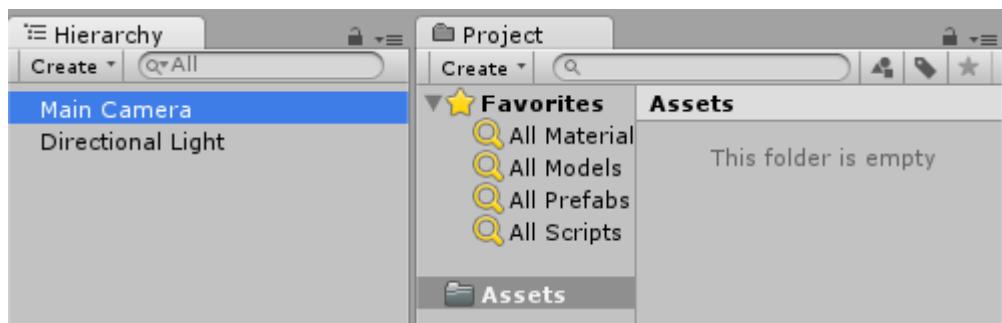
Les **préfixes** sont un type d'actif qui permet de stocker un objet GameObject complet avec ses composants, propriétés, composants associés et valeurs de propriété sérialisées. Il existe de nombreux scénarios où cela est utile, notamment:

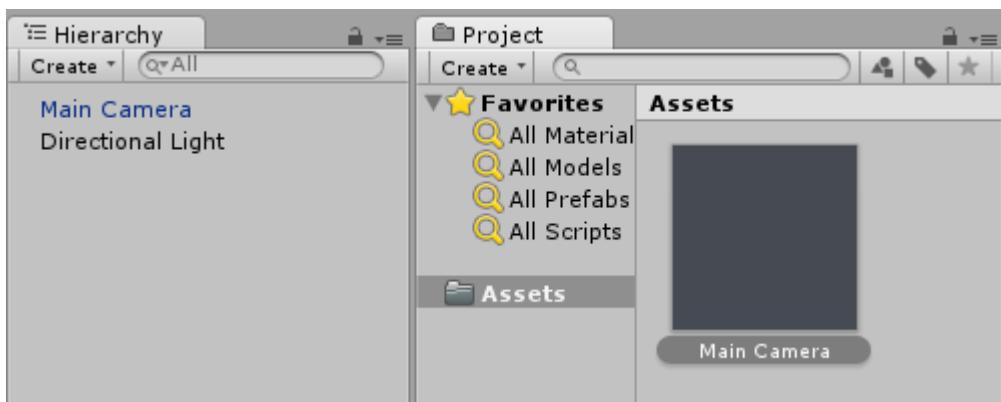
- Dupliquer des objets dans une scène
- Partage d'un objet commun sur plusieurs scènes
- Pouvoir modifier une pièce préfabriquée une seule fois et appliquer les modifications à plusieurs objets / scènes
- Créer des objets dupliqués avec des modifications mineures, tout en ayant les éléments communs modifiables à partir d'une base préfabriquée
- Instanciation de GameObjects à l'exécution

Il y a une règle de base dans l'Unité qui dit "tout devrait être préfabriqué". Bien que cela soit probablement exagéré, cela encourage la réutilisation du code et la création de GameObjects de manière réutilisable, ce qui est à la fois efficace en termes de mémoire et de conception.

Créer des préfabriqués

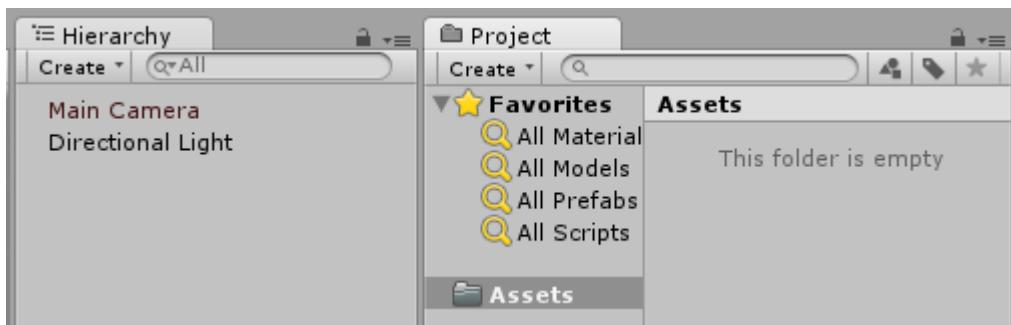
Pour créer un préfabriqué, faites glisser un objet de jeu de la hiérarchie de la scène dans le dossier ou sous-dossier **Assets** :





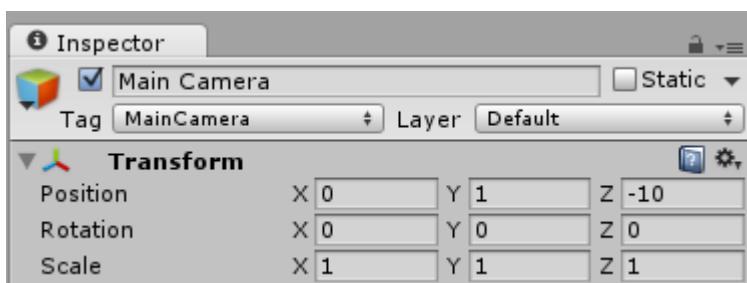
Le nom de l'objet du jeu devient bleu, indiquant qu'il est **connecté à un préfabriqué**. Maintenant, cet objet est une **instance préfabriquée**, tout comme une instance d'objet d'une classe.

Un préfabriqué peut être supprimé après instantiation. Dans ce cas, le nom de l'objet de jeu précédemment connecté devient rouge:

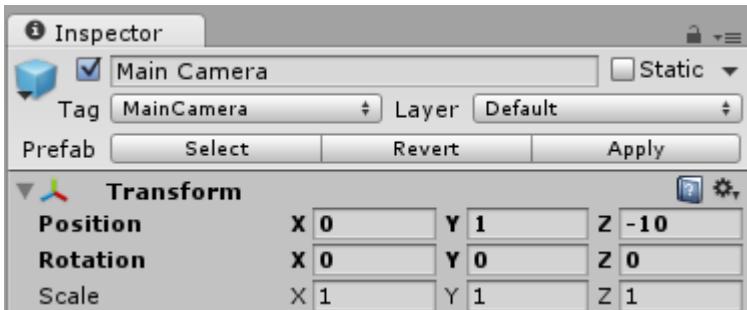


Inspecteur préfabriqué

Si vous sélectionnez un préfabriqué dans la vue hiérarchique, vous remarquerez que son inspecteur diffère légèrement d'un objet de jeu ordinaire:



contre



Les propriétés en gras signifient que leurs valeurs diffèrent des valeurs préfabriquées. Vous pouvez modifier toute propriété d'un préfabriqué instancié sans affecter les valeurs préfabriquées d'origine. Lorsqu'une valeur est modifiée dans une instance préfabriquée, elle devient en gras et toute modification ultérieure de la même valeur dans le préfabriqué ne sera pas répercutée dans l'instance modifiée.

Vous pouvez revenir aux valeurs préfabriquées d'origine en cliquant sur le bouton **Rétablir**, qui reflétera également les changements de valeur dans l'instance. De plus, pour rétablir une valeur individuelle, vous pouvez cliquer avec le bouton droit de la souris et appuyer sur **Rétablir la valeur dans Préfab**. Pour rétablir un composant, cliquez dessus avec le bouton droit de la souris et appuyez sur **Revenir sur Préfab**.

Un clic sur le bouton **Appliquer** remplace les valeurs de propriété préfabriquées par les valeurs de propriété d'objet de jeu actuelles. Il n'y a pas de bouton "Annuler" ou de dialogue de confirmation, donc manipulez ce bouton avec soin.

Le bouton **Sélectionner** met en évidence le préfabriqué connecté dans la structure de dossiers du projet.

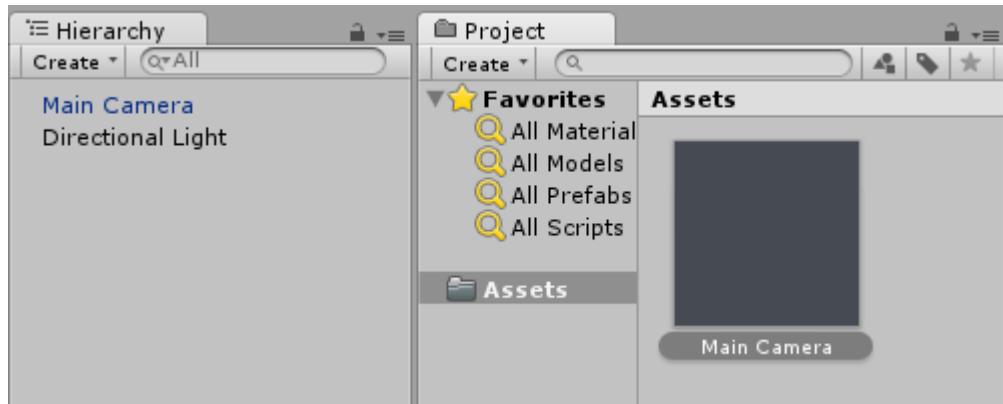
Préfabrication

Il y a deux façons d'instancier les préfabriqués: pendant la **conception** ou l' **exécution** .

Instanciation du temps de conception

L'instanciation des préfabriqués au moment de la conception est utile pour placer plusieurs instances du même objet (par exemple, *placer des arbres lors de la conception d'un niveau de votre jeu*).

- Pour instancier visuellement un préfabriqué, faites-le glisser de la vue du projet vers la hiérarchie de la scène.



- Si vous écrivez une [extension d'éditeur](#), vous pouvez également instancier une méthode `PrefabUtility.InstantiatePrefab()` appelant par programmation:

```
GameObject gameObject =
(GameObject)PrefabUtility.InstantiatePrefab(AssetDatabase.LoadAssetAtPath("Assets/MainCamera.pref
typeof(gameObject)));
```

Instanciation du runtime

L'instanciation de préfabriqués à l'exécution est utile pour créer des instances d'un objet en fonction d'une logique (par exemple, *générer un ennemi toutes les 5 secondes*).

Pour instancier un préfabriqué, vous avez besoin d'une référence à l'objet préfabriqué. Cela peut être fait en ayant un champ `public GameObject` dans votre script `MonoBehaviour` (et en définissant sa valeur à l'aide de l'inspecteur dans l'éditeur Unity):

```
public class SomeScript : MonoBehaviour {
    public GameObject prefab;
}
```

Ou en plaçant le préfabriqué dans le dossier [Resource](#) et en utilisant `Resources.Load`:

```
GameObject prefab = Resources.Load("Assets/Resources/MainCamera");
```

Une fois que vous avez une référence à l'objet préfabriqué, vous pouvez l'instancier en utilisant la fonction `Instantiate` n'importe où dans votre code (par exemple, *dans une boucle pour créer plusieurs objets*):

```
GameObject gameObject = Instantiate<GameObject>(prefab, new Vector3(0, 0, 0),
Quaternion.identity);
```

Remarque: Le terme *préfabriqué* n'existe pas au moment de l'exécution.

Préfabriqués imbriqués

Les préfabriqués imbriqués ne sont pas disponibles dans Unity pour le moment. Vous pouvez faire

glisser un préfabriqué vers un autre et l'appliquer, mais les modifications apportées au préfabriqué enfant ne seront pas appliquées à celui qui est imbriqué.

Mais il existe une solution de contournement simple: **vous devez ajouter au parent prefab un script simple, qui instanciera un enfant.**

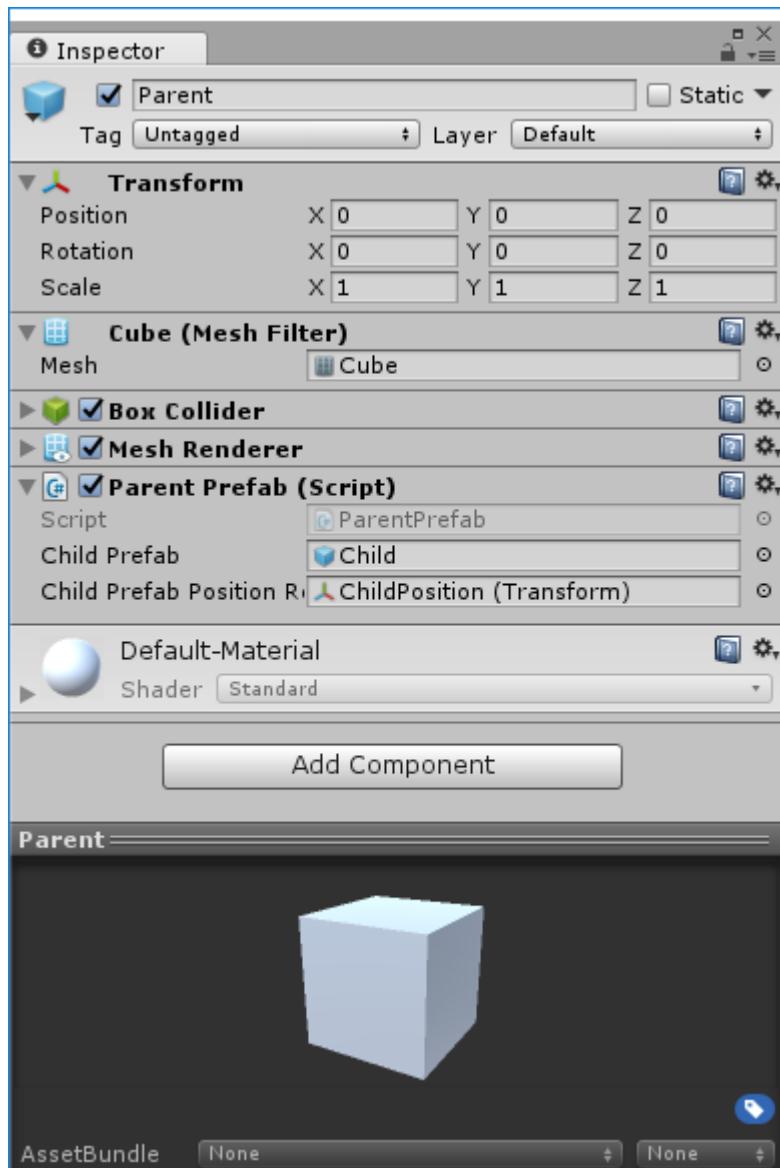
```
using UnityEngine;

public class ParentPrefab : MonoBehaviour {

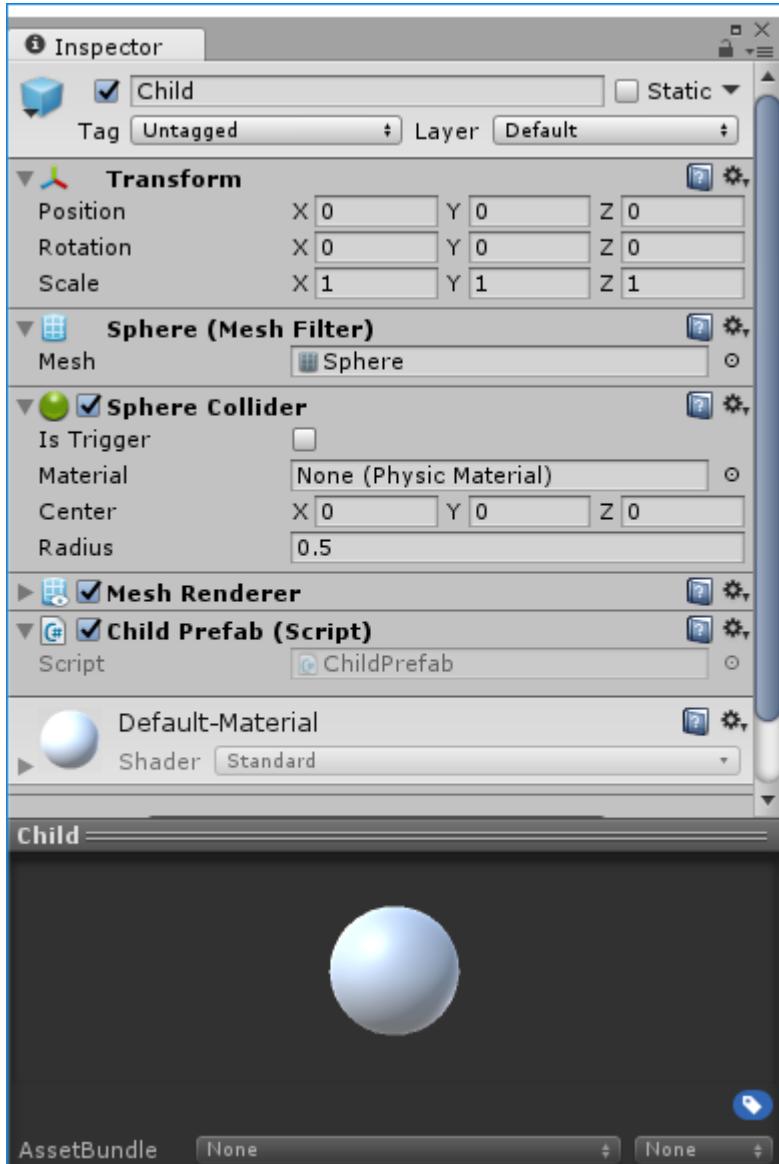
    [SerializeField] GameObject childPrefab;
    [SerializeField] Transform childPrefabPositionReference;

    // Use this for initialization
    void Start () {
        print("Hello, I'm a parent prefab!");
        Instantiate(
            childPrefab,
            childPrefabPositionReference.position,
            childPrefabPositionReference.rotation,
            gameObject.transform
        );
    }
}
```

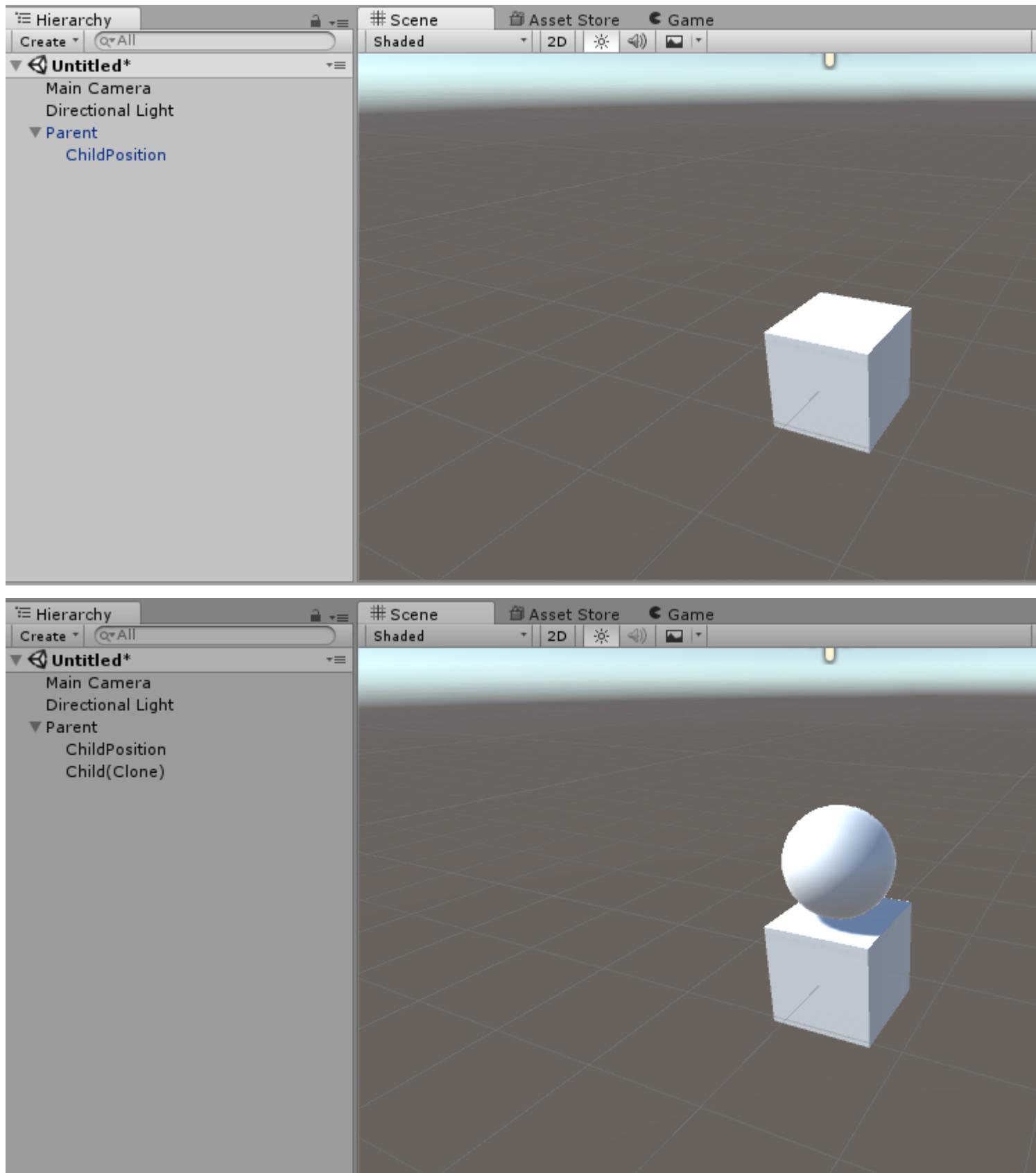
Parent préfabriqué:



Enfant préfabriqué:



Scène avant et après le début:



Lire Préfabriqués en ligne: <https://riptutorial.com/fr/unity3d/topic/2133/prefabriques>

Chapitre 24: Quaternions

Syntaxe

- Quaternion.LookRotation (Vector3 forward [, Vector3 up]);
- Quaternion.AngleAxis (angles flottants, Vector3 axisOfRotation);
- angle flottantEntre = Quaternion.Angle (Quaternion rotation1, Quaternion rotation2);

Examples

Introduction à Quaternion vs Euler

Les angles d'Euler sont des "angles de degré" comme 90, 180, 45, 30 degrés. Les quaternions diffèrent des angles d'Euler en ce sens qu'ils représentent un point sur une sphère d'unité (le rayon est de 1 unité). Vous pouvez voir cette sphère comme une version 3D du cercle Unit que vous apprenez en trigonométrie. Les quaternions diffèrent des angles d'Euler en ce sens qu'ils utilisent des nombres imaginaires pour définir une rotation 3D.

Bien que cela puisse paraître compliqué (et c'est sans doute le cas), Unity possède d'excellentes fonctions intégrées qui vous permettent de basculer entre les angles et les quaternions d'Euler, ainsi que des fonctions permettant de modifier les quaternions, sans connaître les mathématiques.

Conversion entre Euler et Quaternion

```
// Create a quaternion that represents 30 degrees about X, 10 degrees about Y
Quaternion rotation = Quaternion.Euler(30, 10, 0);

// Using a Vector
Vector3 EulerRotation = new Vector3(30, 10, 0);
Quaternion rotation = Quaternion.Euler(EulerRotation);

// Convert a transforms Quaternion angles to Euler angles
Quaternion quaternionAngles = transform.rotation;
Vector3 eulerAngles = quaternionAngles.eulerAngles;
```

Pourquoi utiliser un quaternion?

Les quaternions résolvent un problème connu sous le nom de blocage du cardan. Cela se produit lorsque l'axe principal de rotation devient colinéaire avec l'axe de rotation tertiaire. Voici un [exemple visuel](#) @ 2:09

Quaternion Look Rotation

Quaternion.LookRotation(Vector3 forward [, Vector3 up]) créera une rotation de Quaternion qui «avance» le vecteur avant et dont l'axe Y est aligné avec le vecteur «up». Si le vecteur ascendant n'est pas spécifié, Vector3.up sera utilisé.

Faites pivoter cet objet de jeu pour regarder un objet de jeu cible

```
// Find a game object in the scene named Target
public Transform target = GameObject.Find("Target").GetComponent<Transform>();

// We subtract our position from the target position to create a
// Vector that points from our position to the target position
// If we reverse the order, our rotation would be 180 degrees off.
Vector3 lookVector = target.position - transform.position;
Quaternion rotation = Quaternion.LookRotation(lookVector);
transform.rotation = rotation;
```

Lire Quaternions en ligne: <https://riptutorial.com/fr/unity3d/topic/1782/quaternions>

Chapitre 25: Raycast

Paramètres

Paramètre	Détails
origine	Le point de départ du rayon dans les coordonnées du monde
direction	La direction du rayon
maxDistance	La distance maximale à laquelle le rayon doit vérifier les collisions
layerMask	Masque de calque utilisé pour ignorer de manière sélective les collisionneurs lors de la projection d'un rayon.
queryTriggerInteraction	Spécifie où cette requête doit frapper les déclencheurs.

Exemples

Physique Raycast

Cette fonction projette un rayon depuis l' `origin` du point dans la `direction` de la longueur `maxDistance` contre tous les collisionneurs de la scène.

La fonction prend la `direction` `origin` `maxDistance` et calcule s'il y a un collider devant le `GameObject`.

```
Physics.Raycast(origin, direction, maxDistance);
```

Par exemple, cette fonction imprimerà `Hello World` sur la console s'il y a quelque chose dans les 10 unités du `GameObject` attaché:

```
using UnityEngine;

public class TestPhysicsRaycast : MonoBehaviour
{
    void FixedUpdate()
    {
        Vector3 fwd = transform.TransformDirection(Vector3.forward);

        if (Physics.Raycast(transform.position, fwd, 10))
            print("Hello World");
    }
}
```

Physics2D Raycast2D

Vous pouvez utiliser les raycasts pour vérifier si un ai peut marcher sans tomber du bord d'un niveau.

```
using UnityEngine;

public class Physics2dRaycast : MonoBehaviour
{
    public LayerMask LineOfSightMask;
    void FixedUpdate()
    {
        RaycastHit2D hit = Physics2D.Raycast(raycastRightPart, Vector2.down, 0.6f * heightCharacter, LineOfSightMask);
        if(hit.collider != null)
        {
            //code when the ai can walk
        }
        else
        {
            //code when the ai cannot walk
        }
    }
}
```

Dans cet exemple, la direction est correcte. La variable raycastRightPart est la partie droite du personnage, de sorte que le raycast se produira dans la partie droite du personnage. La distance est de 0,6 fois la hauteur du personnage, donc le rayon ne donne pas un coup quand il touche le sol beaucoup plus bas que le sol sur lequel il se trouve. Assurez-vous que le Layermask est défini sur la masse uniquement, sinon il détectera également d'autres types d'objets.

RaycastHit2D lui-même est une structure et non une classe, donc hit ne peut pas être nul; Cela signifie que vous devez vérifier le collisionneur d'une variable RaycastHit2D.

Encapsulation des appels Raycast

Si vous appelez directement vos scripts `Raycast` vous risquez de rencontrer des problèmes si vous devez modifier les matrices de collision à l'avenir, car vous devrez suivre tous les champs `LayerMask` pour les adapter. Selon la taille de votre projet, cela peut devenir une énorme entreprise.

Encapsuler les appels `Raycast` peut vous simplifier la vie.

À partir d'un principe [SoC](#), un objet de jeu ne devrait vraiment pas connaître ou se soucier de `LayerMasks`. Il suffit d'une méthode pour scanner son environnement. Que le résultat de raycast renvoie ceci ou cela ne devrait pas avoir d'importance pour le gameobject. Il ne doit agir que sur les informations reçues et ne pas faire d'hypothèses sur l'environnement dans lequel il se trouve.

Une façon de procéder consiste à déplacer la valeur `LayerMask` vers des instances [ScriptableObject](#) et à les utiliser comme forme de services raycast que vous injectez dans vos scripts.

```
// RaycastService.cs
using UnityEngine;
```

```
[CreateAssetMenu(menuName = "StackOverflow")]
public class RaycastService : ScriptableObject
{
    [SerializeField]
    LayerMask layerMask;

    public RaycastHit2D Raycast2D(Vector2 origin, Vector2 direction, float distance)
    {
        return Physics2D.Raycast(origin, direction, distance, layerMask.value);
    }

    // Add more methods as needed
}
```

```
// MyScript.cs
using UnityEngine;

public class MyScript : MonoBehaviour
{
    [SerializeField]
    RaycastService raycastService;

    void FixedUpdate()
    {
        RaycastHit2D hit = raycastService.Raycast2D(Vector2.zero, Vector2.down, 1f);
    }
}
```

Cela vous permet de créer un certain nombre de services raycast, le tout avec différentes combinaisons LayerMask pour différentes situations. Vous pourriez en avoir un qui ne touche que les collisionneurs au sol, et un autre qui touche les collisionneurs au sol et les plates-formes à sens unique.

Si vous devez apporter des modifications radicales à vos configurations LayerMask, il vous suffit de mettre à jour ces ressources RaycastService.

Lectures complémentaires

- [Inversion de contrôle](#)
- [Injection de dépendance](#)

Lire Raycast en ligne: <https://riptutorial.com/fr/unity3d/topic/2826/raycast>

Chapitre 26: Réalité virtuelle (VR)

Examples

Plateformes VR

Il existe deux plates-formes principales en VR, l'une est la plate-forme mobile, comme **Google Cardboard** , **Samsung GearVR** , l'autre est la plate-forme PC, comme **HTC Vive**, **Oculus**, **PS VR** ...

Unity prend officiellement en charge les **systèmes Oculus Rift** , **Google Carboard** , **Steam VR** , **Playstation VR** , **Gear VR** et **Microsoft Hololens** .

La plupart des plates-formes ont leur propre support et sdk. Habituellement, vous devez télécharger le SDK en tant qu'extension d'abord pour l'unité.

SDK:

- [Google carton](#)
- [Plateforme Daydream](#)
- [Samsung GearVR](#) (intégré depuis Unity 5.3)
- [Oculus Rift](#)
- [HTC Vive / Open VR](#)
- [Microsoft Hololens](#)

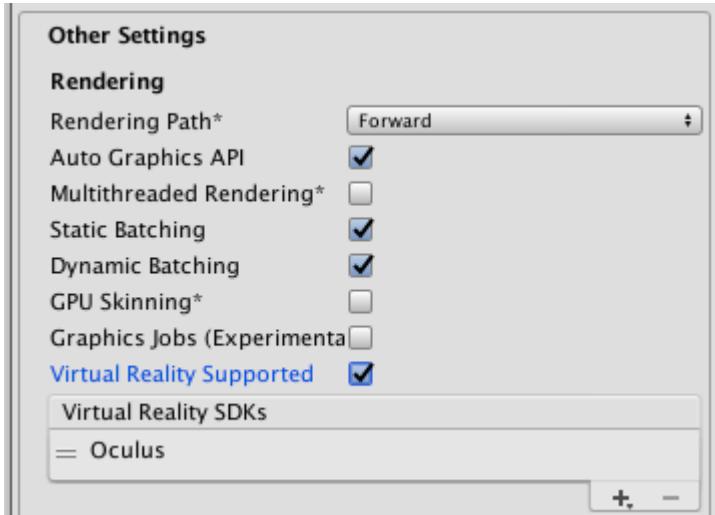
Documentation:

- [Google carton / rêverie](#)
- [Samsung GearVR](#)
- [Oculus Rift](#)
- [HTC Vive](#)
- [Microsoft Hololens](#)

Activation du support VR

Dans Unity Editor, ouvrez **Paramètres du lecteur** (Edition> Paramètres du projet> Lecteur).

Sous **Autres paramètres** , cochez *Réalité virtuelle prise en charge* .



Ajoutez ou supprimez des périphériques VR pour chaque cible de génération dans la liste *SDK de réalité virtuelle* sous la case à cocher.

Matériel

Il existe une dépendance matérielle nécessaire pour une application VR, qui dépend généralement de la plate-forme que vous créez. Il existe 2 grandes catégories de périphériques matériels en fonction de leurs capacités de mouvement:

1. 3 DOF (degrés de liberté)
2. 6 degrés de liberté

3 DOF signifie que le mouvement de l'affichage monté sur la tête (HMD) est contraint de fonctionner en 3 dimensions qui tournent autour des trois axes orthogonaux centrés sur le centre de gravité du HMD - les axes longitudinal, vertical et horizontal. Le mouvement autour de l'axe longitudinal est appelé roulis, le mouvement autour de l'axe latéral est appelé tangage et le mouvement autour de l'axe perpendiculaire est appelé lacet, des principes similaires régissent le mouvement de tout objet en mouvement comme un avion ou une voiture. capable de voir dans toutes les directions X, Y, Z par le mouvement de votre HMD dans l'environnement virtuel, mais vous ne pourriez rien déplacer ou toucher (le mouvement d'un contrôleur Bluetooth supplémentaire n'est pas le même).

Cependant, 6 DOF permet une expérience à l'échelle de la pièce dans laquelle vous pouvez également vous déplacer sur les axes X, Y et Z en dehors des mouvements de roulis, tangage et tangage autour de son centre de gravité, d'où le degré de liberté 6.

Actuellement, un système VR à l'échelle de la salle pour 6 DOF nécessite des performances de calcul élevées avec une carte graphique et une RAM haut de gamme que vous n'obtiendrez probablement pas de vos ordinateurs portables standard et nécessitera un ordinateur de bureau aux performances optimales espace libre, alors qu'une expérience de 3 DOF peut être réalisée par un simple smartphone standard avec un gyroscope intégré (intégré dans la plupart des smartphones modernes, qui coûte environ 200 dollars ou plus).

Certains dispositifs courants disponibles sur le marché sont les suivants:

- [Oculus Rift \(6 DOF\)](#)
- [HTC Vive \(6 DOF\)](#)
- [Rêverie \(3 DOF\)](#)
- [Gear VR Powered by Oculus \(3 DOF\)](#)
- [Google Cardboard \(3 DOF\)](#)

Lire Réalité virtuelle (VR) en ligne: <https://riptutorial.com/fr/unity3d/topic/5787/realite-virtuelle--vr->

Chapitre 27: Recherche et collecte de GameObjects

Syntaxe

- public statique GameObject Find (nom de chaîne);
- statique publique GameObject FindGameObjectWithTag (balise de chaîne);
- statique publique GameObject [] FindGameObjectsWithTag (balise de chaîne);
- objet statique public FindObjectOfType (type type);
- public static Object [] FindObjectsOfType (type type);

Remarques

Quelle méthode utiliser

Soyez prudent lorsque vous recherchez GameObjects à l'exécution, car cela peut nécessiter beaucoup de ressources. Surtout: ne lancez pas `FindObjectOfType` ou `Find` in `Update`, `FixedUpdate` ou plus généralement dans une méthode appelée une ou plusieurs fois par frame.

- Appelez les méthodes d'exécution `FindObjectOfType` et `Find` uniquement lorsque cela est nécessaire
- `FindGameObjectWithTag` a de très bonnes performances par rapport aux autres méthodes basées sur les chaînes. Unity conserve des onglets séparés sur les objets marqués et interroge ceux-ci au lieu de la scène entière.
- Pour les GameObjects "statiques" (tels que les éléments d'interface utilisateur et les préfabriqués) créés dans l'éditeur, utilisez la [référence sérialisable de GameObject](#) dans l'éditeur.
- Gardez vos listes de GameObjects dans List ou Arrays que vous gérez vous-même
- En général, si vous instanciez beaucoup de GameObjects du même type, jetez un coup d'œil au [pool d'objets](#)
- Mettez vos résultats de recherche en cache pour éviter les méthodes de recherche coûteuses.

Aller plus loin

Outre les méthodes fournies par Unity, il est relativement facile de concevoir vos propres méthodes de recherche et de collecte.

- Dans le cas de `FindObjectsofType()`, vos scripts peuvent conserver une liste d'eux-mêmes dans une collection `static`. Il est beaucoup plus rapide d'itérer une liste d'objets que de rechercher et inspecter des objets de la scène.
- Ou créez un script qui stocke leurs instances dans un `Dictionary` basé sur des chaînes, et

vous pourrez développer un système de marquage simple.

Exemples

Recherche par nom de GameObject

```
var go = GameObject.Find("NameOfTheObject");
```

Avantages	Les inconvénients
Facile à utiliser	Les performances se dégradent avec le nombre d'objets de jeu dans la scène
	Les chaînes sont <i>des références faibles</i> et suspectent les erreurs de l'utilisateur

Recherche par tags de GameObject

```
var go = GameObject.FindGameObjectWithTag("Player");
```

Avantages	Les inconvénients
Possibilité de rechercher des objets simples et des groupes entiers	Les chaînes sont des références faibles et suspectent les erreurs de l'utilisateur.
Relativement rapide et efficace	Le code n'est pas portable car les balises sont codées en dur dans les scripts.

Inséré dans les scripts en mode édition

```
[SerializeField]  
GameObject[] gameObjects;
```

Avantages	Les inconvénients
Belle performance	La collection d'objets est statique
Code portable	Ne peut se référer qu'à GameObjects de la même scène

Recherche de scripts GameObjects par MonoBehaviour

```
ExampleScript script = GameObject.FindObjectOfType<ExampleScript>();  
GameObject go = script.gameObject;
```

`FindObjectOfType()` renvoie `null` si aucun n'est trouvé.

Avantages	Les inconvénients
Fortement typé	Les performances se dégradent avec le nombre d'objets de jeu nécessaires pour évaluer
Possibilité de rechercher des objets simples et des groupes entiers	

Recherche de GameObjects par nom à partir d'objets enfants

```
Transform tr = GetComponent<Transform>().Find("NameOfTheObject");
GameObject go = tr.gameObject;
```

Find renvoie null si aucun n'est trouvé

Avantages	Les inconvénients
Portée de recherche limitée et bien définie	Les chaînes sont des références faibles

Lire Recherche et collecte de GameObjects en ligne:

<https://riptutorial.com/fr/unity3d/topic/3793/recherche-et-collecte-de-gameobjects>

Chapitre 28: Regroupement d'objets

Examples

Pool d'objets

Parfois, lorsque vous créez un jeu, vous devez créer et détruire de nombreux objets du même type, encore et encore. Vous pouvez simplement faire cela en créant un préfabriqué et en l'instanciant / détruisant chaque fois que vous en avez besoin, mais cela est inefficace et peut ralentir votre jeu.

Un moyen de contourner ce problème est le regroupement d'objets. En gros, cela signifie que vous avez un pool (avec ou sans limite de quantité) d'objets que vous allez réutiliser chaque fois que vous le pouvez pour éviter une instantiation ou une destruction inutiles.

Voici un exemple de pool d'objets simple

```
public class ObjectPool : MonoBehaviour
{
    public GameObject prefab;
    public int amount = 0;
    public bool populateOnStart = true;
    public bool growOverAmount = true;

    private List<GameObject> pool = new List<GameObject>();

    void Start()
    {
        if (populateOnStart && prefab != null && amount > 0)
        {
            for (int i = 0; i < amount; i++)
            {
                var instance = Instantiate(Prefab);
                instance.SetActive(false);
                pool.Add(instance);
            }
        }
    }

    public GameObject Instantiate (Vector3 position, Quaternion rotation)
    {
        foreach (var item in pool)
        {
            if (!item.activeInHierarchy)
            {
                item.transform.position = position;
                item.transform.rotation = rotation;
                item.SetActive( true );
                return item;
            }
        }

        if (growOverAmount)
        {
```

```

        var instance = (GameObject)Instantiate(prefab, position, rotation);
        pool.Add(instance);
        return instance;
    }

    return null;
}
}

```

Voyons d'abord les variables

```

public GameObject prefab;
public int amount = 0;
public bool populateOnStart = true;
public bool growOverAmount = true;

private List<GameObject> pool = new List<GameObject>();

```

- `GameObject prefab` : il s'agit du préfabriqué que le pool d'objets utilisera pour instancier de nouveaux objets dans le pool.
- `int amount` : C'est le nombre maximum d'éléments pouvant se trouver dans le pool. Si vous souhaitez instancier un autre élément et que le pool a déjà atteint sa limite, un autre élément du pool sera utilisé.
- `bool populateOnStart` : vous pouvez choisir de remplir le pool au démarrage ou non. Cela remplira le pool d'instances du préfabriqué de sorte que la première fois que vousappelez `Instantiate` vous obtiendrez un objet déjà existant.
- `bool growOverAmount` : `bool growOverAmount` cette valeur sur `true` pour permettre au pool de croître chaque fois que le montant demandé est supérieur à un certain montant. Vous n'êtes pas toujours en mesure de prédire avec précision la quantité d'articles à placer dans votre piscine, ce qui ajoutera plus à votre piscine en cas de besoin.
- `List<GameObject> pool` : c'est le pool, l'endroit où tous vos objets instanciés / détruits sont stockés.

Maintenant, vérifions la fonction `Start`

```

void Start()
{
    if (populateOnStart && prefab != null && amount > 0)
    {
        for (int i = 0; i < amount; i++)
        {
            var instance = Instantiate(Prefab);
            instance.SetActive(false);
            pool.Add(instance);
        }
    }
}

```

Dans la fonction de démarrage, nous vérifions si nous devons remplir la liste au début et le faire si le `prefab` a été défini et que le montant est supérieur à 0 (sinon, nous créerions indéfiniment).

Ceci est juste une simple boucle pour instancier de nouveaux objets et les placer dans le pool.

Une chose à laquelle il faut faire attention est que toutes les instances sont inactives. De cette façon, ils ne sont pas encore visibles dans le jeu.

Ensuite, il y a la fonction `Instantiate`, qui est la plus grande partie de la magie

```
public GameObject Instantiate (Vector3 position, Quaternion rotation)
{
    foreach (var item in pool)
    {
        if (!item.activeInHierarchy)
        {
            item.transform.position = position;
            item.transform.rotation = rotation;
            item.SetActive(true);
            return item;
        }
    }

    if (growOverAmount)
    {
        var instance = (GameObject) Instantiate(prefab, position, rotation);
        pool.Add(instance);
        return instance;
    }

    return null;
}
```

La `Instantiate` fonction ressemble propre de l' unité `Instantiate` fonction, à l' exception de la maison préfabriquée a déjà été fournie ci - dessus en tant que membre de la classe.

La première étape de la fonction `Instantiate` consiste à vérifier si un objet inactif se trouve actuellement dans le pool. Cela signifie que nous pouvons réutiliser cet objet et le rendre au demandeur. S'il y a un objet inactif dans le pool, nous définissons la position et la rotation, configurez-le pour qu'il soit actif (sinon il pourrait être réutilisé par accident si vous oubliez de l'activer) et renvoyez-le au demandeur.

La deuxième étape ne se produit que s'il n'y a pas d'éléments inactifs dans le pool et que le pool est autorisé à dépasser le montant initial. Ce qui se passe est simple: une autre instance du préfabriqué est créée et ajoutée au pool. Permettre la croissance de la piscine vous aide à avoir la bonne quantité d'objets dans la piscine.

La troisième "étape" ne se produit que s'il n'y a pas d'éléments inactifs dans le pool et que le pool n'est pas autorisé à se développer. Lorsque cela se produit, le demandeur recevra un objet `GameObject` nul, ce qui signifie que rien n'était disponible et devrait être géré correctement pour empêcher les `NullReferenceExceptions`.

Important!

Pour vous assurer que vos objets reviennent dans le pool, vous ne devez pas détruire les objets du jeu. La seule chose à faire est de les désactiver et de les rendre utilisables dans le pool.

Pool d'objets simple

Vous trouverez ci-dessous un exemple de pool d'objets permettant de louer et de renvoyer un type d'objet donné. Pour créer le pool d'objets, une fonction Func pour la fonction de création et une action pour détruire l'objet sont requises pour donner à l'utilisateur une certaine flexibilité. Lors de la demande d'un objet lorsque le pool est vide, un nouvel objet sera créé et, sur demande, lorsque le pool contient des objets, les objets seront supprimés du pool et renvoyés.

Pool d'objets

```
public class ResourcePool<T> where T : class
{
    private readonly List<T> objectPool = new List<T>();
    private readonly Action<T> cleanUpAction;
    private readonly Func<T> createAction;

    public ResourcePool(Action<T> cleanUpAction, Func<T> createAction)
    {
        this.cleanUpAction = cleanUpAction;
        this.createAction = createAction;
    }

    public void Return(T resource)
    {
        this.objectPool.Add(resource);
    }

    private void PurgeSingleResource()
    {
        var resource = this.Rent();
        this.cleanUpAction(resource);
    }

    public void TrimResourcesBy(int count)
    {
        count = Math.Min(count, this.objectPool.Count);
        for (int i = 0; i < count; i++)
        {
            this.PurgeSingleResource();
        }
    }

    public T Rent()
    {
        int count = this.objectPool.Count;
        if (count == 0)
        {
            Debug.Log("Creating new object.");
            return this.createAction();
        }
        else
        {
            Debug.Log("Retrieving existing object.");
            T resource = this.objectPool[count - 1];
            this.objectPool.RemoveAt(count - 1);
            return resource;
        }
    }
}
```

Utilisation de l'échantillon

```
public class Test : MonoBehaviour
{
    private ResourcePool<GameObject> objectPool;

    [SerializeField]
    private GameObject enemyPrefab;

    void Start()
    {
        this.objectPool = new ResourcePool<GameObject>(Destroy, () =>
Instantiate(this.enemyPrefab) );
    }

    void Update()
    {
        // To get existing object or create new from pool
        var newEnemy = this.objectPool.Rent();
        // To return object to pool
        this.objectPool.Return(newEnemy);
        // In this example the message 'Creating new object' should only be seen on the frame
call
        // after that the same object in the pool will be returned.
    }
}
```

Un autre pool d'objets simples

Un autre exemple: une arme qui tire des balles.

L'arme agit comme un pool d'objets pour les puces qu'elle crée.

```
public class Weapon : MonoBehaviour {

    // The Bullet prefab that the Weapon will create
    public Bullet bulletPrefab;

    // This List is our object pool, which starts out empty
    private List<Bullet> availableBullets = new List<Bullet>();

    // The Transform that will act as the Bullet starting position
    public Transform bulletInstantiationPoint;

    // To spawn a new Bullet, this method either grabs an available Bullet from the pool,
    // otherwise Instantiates a new Bullet
    public Bullet CreateBullet () {
        Bullet newBullet = null;

        // If a Bullet is available in the pool, take the first one and make it active
        if (availableBullets.Count > 0) {
            newBullet = availableBullets[availableBullets.Count - 1];

            // Remove the Bullet from the pool
            availableBullets.RemoveAt(availableBullets.Count - 1);

            // Set the Bullet's position and make its GameObject active
            newBullet.transform.position = bulletInstantiationPoint.position;
        }
    }
}
```

```

        newBullet.gameObject.SetActive(true);
    }
    // If no Bullets are available in the pool, Instantiate a new Bullet
    else {
        newBullet newObject = Instantiate(bulletPrefab, bulletInstantiationPoint.position,
Quaternion.identity);

        // Set the Bullet's Weapon so we know which pool to return to later on
        newBullet.weapon = this;
    }

    return newBullet;
}

}

public class Bullet : MonoBehaviour {

    public Weapon weapon;

    // When Bullet collides with something, rather than Destroying it, we return it to the
pool
    public void ReturnToPool () {
        // Add Bullet to the pool
        weapon.availableBullets.Add(this);

        // Disable the Bullet's GameObject so it's hidden from view
        gameObject.SetActive(false);
    }

}

```

Lire Regroupement d'objets en ligne: <https://riptutorial.com/fr/unity3d/topic/2276/regroupement-d-objets>

Chapitre 29: Ressources

Examples

introduction

Avec la classe Ressources, il est possible de charger dynamiquement des actifs qui ne font pas partie de la scène. C'est très utile lorsque vous devez utiliser des ressources à la demande, par exemple localiser des audios multilingues, des textes, etc.

Les actifs doivent être placés dans un dossier nommé **Ressources**. Il est possible d'avoir plusieurs dossiers de ressources répartis dans la hiérarchie du projet. `Resources` classe de `Resources` inspectera tous les dossiers de ressources que vous pourriez avoir.

Chaque actif placé dans les ressources sera inclus dans la version, même s'il n'est pas référencé dans votre code. N'insérez donc pas d'actifs dans Resources sans distinction.

```
//Example of how to load language specific audio from Resources

[RequireComponent(typeof(AudioSource))]
public class loadIntroAudio : MonoBehaviour {
    void Start () {
        string language = Application.systemLanguage.ToString();
        AudioClip ac = Resources.Load(language + "/intro") as AudioClip; //loading intro.mp3
        //specific for user's language (note the file file extension should not be used)
        if (ac==null)
        {
            ac = Resources.Load("English/intro") as AudioClip; //fallback to the english
            //version for any unsupported language
        }
        transform.GetComponent< AudioSource >().clip = ac;
        transform.GetComponent< AudioSource >().Play();
    }
}
```

Ressources 101

introduction

Unity dispose de quelques dossiers «spécialement nommés» qui permettent une variété d'utilisations. Un de ces dossiers est appelé "Ressources"

Le dossier 'Resources' est l'un des deux seuls moyens de charger des ressources lors de l'exécution dans Unity (l'autre étant [AssetBundles \(Unity Docs\)](#)).

Le dossier 'Resources' peut résider n'importe où dans votre dossier Assets et vous pouvez avoir plusieurs dossiers nommés Resources. Le contenu de tous les dossiers "Ressources" est fusionné au moment de la compilation.

Le principal moyen de charger un actif à partir d'un dossier Ressources consiste à utiliser la fonction `Resources.Load`. Cette fonction prend un paramètre de chaîne qui vous permet de spécifier le chemin du fichier par rapport au dossier Resources. Notez que vous n'avez pas besoin de spécifier des extensions de fichier lors du chargement d'un actif

```
public class ResourcesSample : MonoBehaviour {  
  
    void Start () {  
        //The following line will load a TextAsset named 'foobar' which was previously placed  
        //under 'Assets/Resources/Stackoverflow/foobar.txt'  
        //Note the absence of the '.txt' extension! This is important!  
  
        var text = Resources.Load<TextAsset>("Stackoverflow/foobar").text;  
        Debug.Log(string.Format("The text file had this in it :: {0}", text));  
    }  
}
```

Les objets composés de plusieurs objets peuvent également être chargés à partir de Ressources. Les exemples sont de tels objets sont des modèles 3D avec des textures cuites ou un sprite multiple.

```
//This example will load a multiple sprite texture from Resources named "A_Multiple_Sprite"  
var sprites = Resources.LoadAll("A_Multiple_Sprite") as Sprite[];
```

Mettre tous ensemble

Voici l'une de mes classes d'assistance que j'utilise pour charger tous les sons pour n'importe quel jeu. Vous pouvez attacher ceci à n'importe quel GameObject dans une scène et charger les fichiers audio spécifiés à partir du dossier 'Resources / Sounds'

```
public class SoundManager : MonoBehaviour {  
  
    void Start () {  
  
        //An array of all sounds you want to load  
        var filesToLoad = new string[] { "Foo", "Bar" };  
  
        //Loop over the array, attach an Audio source for each sound clip and assign the  
        //clip property.  
        foreach(var file in filesToLoad) {  
            var soundClip = Resources.Load<AudioClip>("Sounds/" + file);  
            var audioSource = gameObject.AddComponent< AudioSource >();  
            audioSource.clip = soundClip;  
        }  
    }  
}
```

Notes finales

1. L'unité est intelligente quand il s'agit d'inclure des actifs dans votre build. Tout élément qui n'est pas sérialisé (c'est-à-dire utilisé dans une scène incluse dans une génération) est exclu d'une génération. TOUTEFOIS, cela ne s'applique à aucun élément du dossier Ressources. Par conséquent, n'allez pas trop loin en ajoutant des actifs à ce dossier
2. Les actifs chargés à l'aide de `Resources.Load` ou `Resources.LoadAll` peuvent être déchargés ultérieurement à l'aide de `Resources.UnloadUnusedAssets` ou `Resources.UnloadAsset`.

Lire Ressources en ligne: <https://riptutorial.com/fr/unity3d/topic/4070/ressources>

Chapitre 30: ScriptableObject

Remarques

ScriptableObjects avec AssetBundles

Faites attention lorsque vous ajoutez des prefabs à AssetBundles s'ils contiennent des références à ScriptableObjects. Étant donné que ScriptableObjects sont essentiellement des actifs, Unity en crée des doublons avant de les ajouter à AssetBundles, ce qui peut entraîner un comportement indésirable lors de l'exécution.

Lorsque vous chargez un objet GameObject à partir d'un AssetBundle, il peut être nécessaire de réinjecter les actifs ScriptableObject dans les scripts chargés, en remplacement des fichiers groupés. Voir [Injection de dépendance](#)

Examples

introduction

ScriptableObjects sont des objets sérialisés qui ne sont pas liés à des scènes ou des objets de jeu comme MonoBehaviours. Pour le dire d'une certaine manière, il s'agit de données et de méthodes liées à des fichiers d'actif dans votre projet. Ces actifs ScriptableObject peuvent être transmis à MonoBehaviours ou à d'autres ScriptableObjects, où leurs méthodes publiques sont accessibles.

En raison de leur nature de ressources sérialisées, elles constituent d'excellentes classes de gestionnaires et de sources de données.

Création d'actifs ScriptableObject

Vous trouverez ci-dessous une implémentation simple de ScriptableObject.

```
using UnityEngine;

[CreateAssetMenu(menuName = "StackOverflow/Examples/MyScriptableObject")]
public class MyScriptableObject : ScriptableObject
{
    [SerializeField]
    int mySerializedNumber;

    int helloWorldCount = 0;

    public void HelloWorld()
    {
        helloWorldCount++;
        Debug.LogFormat("Hello! My number is {0}.", mySerializedNumber);
        Debug.LogFormat("I have been called {0} times.", helloWorldCount);
    }
}
```

```
    }
}
```

En ajoutant l'attribut `CreateAssetMenu` à la classe, Unity le répertoriera dans le sous-menu **Assets / Create**. Dans ce cas, il se trouve sous **Assets / Create / StackOverflow / Examples**.

Une fois créées, les instances de `ScriptableObject` peuvent être transmises à d'autres scripts et à `ScriptableObjects` via l'inspecteur.

```
using UnityEngine;

public class SampleScript : MonoBehaviour {

    [SerializeField]
    MyScriptableObject myScriptableObject;

    void OnEnable()
    {
        myScriptableObject.HelloWorld();
    }
}
```

Créer des instances `ScriptableObject` via le code

Vous créez de nouvelles instances `ScriptableObject` via `ScriptableObject.CreateInstance<T>()`

```
T obj = ScriptableObject.CreateInstance<T>();
```

Où `T` étend `ScriptableObject`.

Ne créez pas `ScriptableObjects` en appelant leurs constructeurs, c.-à-d. `new ScriptableObject()`.

La création de `ScriptableObjects` par code pendant l'exécution est rarement requise car leur utilisation principale est la sérialisation des données. Vous pourriez aussi bien utiliser des classes standard à ce stade. C'est plus fréquent lorsque vous écrivez des extensions d'éditeur.

`ScriptableObjects` sont sérialisés dans l'éditeur même dans PlayMode

Des précautions supplémentaires doivent être prises lors de l'accès aux champs sérialisés dans une instance `ScriptableObject`.

Si un champ est marqué comme `public` ou sérialisé via `SerializeField`, sa modification est permanente. Ils ne se réinitialisent pas lorsque vous quittez le mode de lecture, comme le fait `MonoBehaviours`. Cela peut être utile parfois, mais cela peut aussi causer des dégâts.

Pour cette raison, il est préférable de rendre les champs sérialisés en lecture seule et d'éviter les champs publics.

```
public class MyScriptableObject : ScriptableObject
{
```

```

[SerializeField]
int mySerializedValue;

public int MySerializedValue
{
    get { return mySerializedValue; }
}
}

```

Si vous souhaitez stocker des valeurs publiques dans un `scriptableObject` réinitialisé entre les sessions de lecture, envisagez d'utiliser le modèle suivant.

```

public class MyScriptableObject : ScriptableObject
{
    // Private fields are not serialized and will reset to default on reset
    private int mySerializedValue;

    public int MySerializedValue
    {
        get { return mySerializedValue; }
        set { mySerializedValue = value; }
    }
}

```

Rechercher des `ScriptableObjects` existants lors de l'exécution

Pour rechercher *des ScriptableObjects actifs* pendant l'exécution, vous pouvez utiliser `Resources.FindObjectsOfTypeAll()`.

```
T[] instances = Resources.FindObjectsOfTypeAll<T>();
```

Où `T` est le type de l'occurrence de `ScriptableObject` que vous recherchez. *Actif* signifie qu'il a été chargé en mémoire sous une forme quelconque auparavant.

Cette méthode est très lente, alors n'oubliez pas de mettre en cache la valeur de retour et d'éviter de l'appeler fréquemment. Le référencement direct de `ScriptableObjects` dans vos scripts devrait être votre option préférée.

Conseil: Vous pouvez gérer vos propres collections d'instances pour des recherches plus rapides. Demandez à vos `ScriptableObjects` de s'inscrire eux-mêmes à une collection partagée pendant `OnEnable()`.

Lire `ScriptableObject` en ligne: <https://riptutorial.com/fr/unity3d/topic/3434/scriptableobject>

Chapitre 31: Se transforme

Syntaxe

- void Transform.Translate (traduction de Vector3, Space relativeTo = Space.Self)
- void Transform.Translate (float x, float y, float z, espace relatif à = espace.Self)
- void Transform.Rotate (Vector3 eulerAngles, Space relativeTo = Space.Self)
- void Transform.Rotate (float xAngle, float yAngle, float zAngle, Space relativeTo = Space.Self)
- void Transform.Rotate (axe Vector3, angle flottant, espace relatif To = Space.Self)
- void Transform.RotateAround (point Vector3, axe Vector3, angle flottant)
- void Transform.LookAt (cible de transformation, Vector3 worldUp = Vector3.up)
- void Transform.LookAt (Vector3 worldPosition, Vector3 worldUp = Vector3.up)

Exemples

Vue d'ensemble

Les transformations contiennent la majorité des données relatives à un objet, y compris ses parents, ses enfants, sa position, sa rotation et son échelle. Il a également des fonctions pour modifier chacune de ces propriétés. Chaque objet GameObject a une transformation.

Traduire (déplacer) un objet

```
// Move an object 10 units in the positive x direction
transform.Translate(10, 0, 0);

// translating with a vector3
vector3 distanceToMove = new Vector3(5, 2, 0);
transform.Translate(distanceToMove);
```

Faire pivoter un objet

```
// Rotate an object 45 degrees about the Y axis
transform.Rotate(0, 45, 0);

// Rotates an object about the axis passing through point (in world coordinates) by angle in
degrees
transform.RotateAround(point, axis, angle);
// Rotates on it's place, on the Y axis, with 90 degrees per second
transform.RotateAround(Vector3.zero, Vector3.up, 90 * Time.deltaTime);

// Rotates an object to make it's forward vector point towards the other object
transform.LookAt(otherTransform);
// Rotates an object to make it's forward vector point towards the given position (in world
coordinates)
transform.LookAt(new Vector3(10, 5, 0));
```

Vous trouverez plus d'informations et d'exemples sur la [documentation d'Unity](#).

Notez également que si le jeu utilise des corps rigides, il ne faut pas interagir directement avec la transformation (sauf si le corps rigide a `isKinematic == true`). Dans ces cas, utilisez `AddForce` ou d'autres méthodes similaires pour agir directement sur le corps rigide.

Parenting et enfants

Unity travaille avec des hiérarchies pour que votre projet reste organisé. Vous pouvez affecter des objets à une place dans la hiérarchie à l'aide de l'éditeur, mais vous pouvez également le faire via du code.

La parentalité

Vous pouvez définir le parent d'un objet avec les méthodes suivantes

```
var other = GetOtherGameObject();
other.transform.SetParent( transform );
other.transform.SetParent( transform, worldPositionStays );
```

Chaque fois que vous définissez un parent de transformation, les objets resteront en position mondiale. Vous pouvez choisir de rendre cette position relative en transmettant `false` pour le paramètre `worldPositionStays`.

Vous pouvez également vérifier si l'objet est un enfant d'une autre transformation avec la méthode suivante

```
other.transform.IsChildOf( transform );
```

Obtenir un enfant

Comme les objets peuvent être parentés, vous pouvez également trouver des enfants dans la hiérarchie. La méthode la plus simple consiste à utiliser la méthode suivante

```
transform.Find( "other" );
transform.FindChild( "other" );
```

Note: Les appels `FindChild` Trouver sous le capot

Vous pouvez également rechercher des enfants plus bas dans la hiérarchie. Vous faites cela en ajoutant un "/" pour spécifier un niveau plus profond.

```
transform.Find( "other/another" );
transform.FindChild( "other/another" );
```

Une autre façon de récupérer un enfant consiste à utiliser `GetChild`

```
transform.GetChild( index );
```

`GetChild` nécessite un entier en tant qu'`index` qui doit être inférieur au nombre total d'enfants

```
int count = transform.childCount;
```

Modification de l'index des frères et sœurs

Vous pouvez changer l'ordre des enfants d'un GameObject. Vous pouvez le faire pour définir l'ordre de dessin des enfants (en supposant qu'ils sont sur le même niveau Z et le même ordre de tri).

```
other.transform.SetSiblingIndex( index );
```

Vous pouvez également définir rapidement l'index des frères et sœurs sur la première ou la dernière méthode en utilisant les méthodes suivantes

```
other.transform.SetAsFirstSibling();
other.transform.SetAsLastSibling();
```

Détacher tous les enfants

Si vous souhaitez libérer tous les enfants d'une transformation, vous pouvez le faire:

```
foreach(Transform child in transform)
{
    child.parent = null;
}
```

De plus, Unity fournit une méthode à cet effet:

```
transform.DetachChildren();
```

Fondamentalement, looping et `DetachChildren()` définissent les parents des enfants de première profondeur sur null - ce qui signifie qu'ils n'auront pas de parents.

(*enfants de première profondeur: les transformées qui sont directement enfants de transformée*)

Lire Se transforme en ligne: <https://riptutorial.com/fr/unity3d/topic/2190/se-transforme>

Chapitre 32: Singletons in Unity

Remarques

Bien qu'il existe des écoles de pensée qui expliquent pourquoi l'utilisation non [restreinte](#) de Singletons est une mauvaise idée, par exemple [Singleton sur gameprogrammingpatterns.com](#), il peut [arriver](#) que vous souhaitiez conserver un objet GameObject dans Unity sur plusieurs scènes (par exemple pour une musique de fond transparente). tout en veillant à ce qu'il n'existe pas plus d'une instance; un cas d'utilisation parfait pour un Singleton.

En ajoutant ce script à un objet GameObject, une fois instancié (par exemple en l'incluant n'importe où dans une scène), il restera actif dans les scènes et une seule instance existera jamais.

Les instances [ScriptableObject](#) ([UnityDoc](#)) fournissent une alternative valide à Singletons pour certains cas d'utilisation. Bien qu'ils n'appliquent pas implicitement la règle d'instance unique, ils conservent leur état entre les scènes et fonctionnent bien avec le processus de sérialisation Unity. Ils favorisent également l'[inversion du contrôle au fur et à mesure que les dépendances sont injectées via l'éditeur](#).

```
// MyAudioManager.cs
using UnityEngine;

[CreateAssetMenu] // Remember to create the instance in editor
public class MyAudioManager : ScriptableObject {
    public void PlaySound() {}
}
```

```
// MyGameObject.cs
using UnityEngine;

public class MyGameObject : MonoBehaviour
{
    [SerializeField]
    MyAudioManager audioManager; //Insert through Inspector

    void OnEnable()
    {
        audioManager.PlaySound();
    }
}
```

Lectures complémentaires

- [Implémentation Singleton en C #](#)

Examples

Implémentation à l'aide de RuntimeInitializeOnLoadMethodAttribute

Depuis **Unity 5.2.5**, il est possible d'utiliser `RuntimeInitializeOnLoadMethodAttribute` pour exécuter la logique d'initialisation en contournant l' [ordre d'exécution MonoBehaviour](#). Cela permet de créer une implémentation plus propre et plus robuste:

```
using UnityEngine;

sealed class GameDirector : MonoBehaviour
{
    // Because of using RuntimeInitializeOnLoadMethod attribute to find/create and
    // initialize the instance, this property is accessible and
    // usable even in Awake() methods.
    public static GameDirector Instance
    {
        get; private set;
    }

    // Thanks to the attribute, this method is executed before any other MonoBehaviour
    // logic in the game.
    [RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.BeforeSceneLoad) ]
    static void OnRuntimeMethodLoad()
    {
        var instance = FindObjectOfType<GameDirector>();

        if (instance == null)
            instance = new GameObject("Game Director").AddComponent<GameDirector>();

        DontDestroyOnLoad(instance);

        Instance = instance;
    }

    // This Awake() will be called immediately after AddComponent() execution
    // in the OnRuntimeMethodLoad(). In other words, before any other MonoBehaviour's
    // in the scene will begin to initialize.
    private void Awake()
    {
        // Initialize non-MonoBehaviour logic, etc.
        Debug.Log("GameDirector.Awake()", this);
    }
}
```

L'ordre d'exécution résultant:

1. `GameDirector.OnRuntimeMethodLoad()` démarré ...
2. `GameDirector.Awake()`
3. `GameDirector.OnRuntimeMethodLoad()` terminé.
4. `OtherMonoBehaviour1.Awake()`
5. `OtherMonoBehaviour2.Awake()` , etc.

Un simple MonoBehaviour Singleton dans Unity C

Dans cet exemple, une instance statique privée de la classe est déclarée au début.

La valeur d'un champ statique est partagée entre les instances, donc si une nouvelle instance de cette classe est créée, le `if` trouvera une référence au premier objet Singleton, détruisant la nouvelle instance (ou son objet de jeu).

```
using UnityEngine;

public class SingletonExample : MonoBehaviour {

    private static SingletonExample _instance;

    void Awake() {

        if (_instance == null){

            _instance = this;
            DontDestroyOnLoad(this.gameObject);

            //Rest of your Awake code

        } else {
            Destroy(this);
        }
    }

    //Rest of your class code
}
```

Advanced Unity Singleton

Cet exemple combine plusieurs variantes de MonoBehaviour singletons trouvées sur Internet en un seul et vous permet de modifier son comportement en fonction de champs statiques globaux.

Cet exemple a été testé avec Unity 5. Pour utiliser ce singleton, il suffit de l'étendre comme suit:

`public class MySingleton : Singleton<MySingleton> {}`. Vous devrez peut-être également remplacer `AwakeSingleton` pour l'utiliser au lieu de `Awake` habituel. Pour plus de réglages, changez les valeurs par défaut des champs statiques comme décrit ci-dessous.

1. Cette implémentation utilise l'attribut `DisallowMultipleComponent` pour conserver une instance par `GameObject`.
2. Cette classe est abstraite et ne peut être étendue. Il contient également une méthode virtuelle `AwakeSingleton` qui doit être remplacée au lieu d'implémenter un `Awake` normal.
3. Cette implémentation est thread-safe.
4. Ce singleton est optimisé. En utilisant un drapeau `instantiated` au lieu d'une instance null, nous évitons la surcharge liée à l'implémentation de l'opérateur `==` par Unity. ([Lire la suite](#))
5. Cette implémentation n'autorise aucun appel à l'instance singleton lorsqu'elle est sur le point d'être détruite par Unity.
6. Ce singleton est livré avec les options suivantes:
 - `FindInactive` : recherche d'autres instances de composants du même type attachées à

GameObject inactif.

- Persist : Persist si le composant doit rester actif entre les scènes.
- DestroyOthers : DestroyOthers -il détruire tout autre composant du même type et n'en conserver qu'un seul?
- Lazy : s'il faut définir l'instance singleton "à la volée" (dans Awake) ou seulement "à la demande" (quand getter est appelé).

```
using UnityEngine;

[DisallowMultipleComponent]
public abstract class Singleton<T> : MonoBehaviour where T : Singleton<T>
{
    private static volatile T instance;
    // thread safety
    private static object _lock = new object();
    public static bool FindInactive = true;
    // Whether or not this object should persist when loading new scenes. Should be set in
    Init().
    public static bool Persist;
    // Whether or not destroy other singleton instances if any. Should be set in Init().
    public static bool DestroyOthers = true;
    // instead of heavy comparision (instance != null)
    // http://blogs.unity3d.com/2014/05/16/custom-operator-should-we-keep-it/
    private static bool instantiated;

    private static bool applicationIsQuitting;

    public static bool Lazy;

    public static T Instance
    {
        get
        {
            if (applicationIsQuitting)
            {
                Debug.LogWarningFormat("[Singleton] Instance '{0}' already destroyed on
                application quit. Won't create again - returning null.", typeof(T));
                return null;
            }
            lock (_lock)
            {
                if (!instantiated)
                {
                    Object[] objects;
                    if (FindInactive) { objects = Resources.FindObjectsOfTypeAll(typeof(T)); }
                    else { objects = FindObjectsOfType(typeof(T)); }
                    if (objects == null || objects.Length < 1)
                    {
                        GameObject singleton = new GameObject();
                        singleton.name = string.Format("{0} [Singleton]", typeof(T));
                        Instance = singleton.AddComponent<T>();
                        Debug.LogWarningFormat("[Singleton] An Instance of '{0}' is needed in
                        the scene, so '{1}' was created{2}", typeof(T), singleton.name, Persist ? " with
                        DontDestroyOnLoad." : ".");
                    }
                    else if (objects.Length >= 1)
                    {
                        Instance = objects[0] as T;
                        if (objects.Length > 1)
```

```

        {
            Debug.LogWarningFormat("[Singleton] {0} instances of '{1}'!", objects.Length, typeof(T));
            if (DestroyOthers)
            {
                for (int i = 1; i < objects.Length; i++)
                {
                    Debug.LogWarningFormat("[Singleton] Deleting extra '{0}' instance attached to '{1}'", typeof(T), objects[i].name);
                    Destroy(objects[i]);
                }
            }
            return instance;
        }
    }
    return instance;
}
protected set
{
    instance = value;
    instantiated = true;
    instance.AwakeSingleton();
    if (Persist) { DontDestroyOnLoad(instance.gameObject); }
}
}

// if Lazy = false and gameObject is active this will set instance
// unless instance was called by another Awake method
private void Awake()
{
    if (Lazy) { return; }
    lock (_lock)
    {
        if (!instantiated)
        {
            Instance = this as T;
        }
        else if (DestroyOthers && Instance.GetInstanceID() != GetInstanceID())
        {
            Debug.LogWarningFormat("[Singleton] Deleting extra '{0}' instance attached to '{1}'", typeof(T), name);
            Destroy(this);
        }
    }
}

// this might be called for inactive singletons before Awake if FindInactive = true
protected virtual void AwakeSingleton() {}

protected virtual void OnDestroy()
{
    applicationIsQuitting = true;
    instantiated = false;
}
}

```

Mise en œuvre de singleton via la classe de base

Dans les projets comportant plusieurs classes singleton (comme c'est souvent le cas), il peut être propre et pratique d'abstraire le comportement singleton dans une classe de base:

```
using UnityEngine;
using System.Collections.Generic;
using System;

public abstract class MonoBehaviourSingleton<T> : MonoBehaviour {

    private static Dictionary<Type, object> _singletons
        = new Dictionary<Type, object>();

    public static T Instance {
        get {
            return (T)_singletons[typeof(T)];
        }
    }

    void OnEnable() {
        if (_singletons.ContainsKey(GetType())) {
            Destroy(this);
        } else {
            _singletons.Add(GetType(), this);
            DontDestroyOnLoad(this);
        }
    }
}
```

Un MonoBehaviour peut alors implémenter le modèle singleton en étendant MonoBehaviourSingleton. Cette approche permet d'utiliser le modèle avec une empreinte minimale sur le Singleton lui-même:

```
using UnityEngine;
using System.Collections;

public class SingletonImplementation : MonoBehaviourSingleton<SingletonImplementation> {

    public string Text= "String Instance";

    // Use this for initialisation
    IEnumerator Start () {
        var demonstration = "SingletonImplementation.Start()\n" +
            "Note that the this text logs only once and\n"
            "only one class instance is allowed to exist.";
        Debug.Log(demonstration);
        yield return new WaitForSeconds(2f);
        var secondInstance = new GameObject();
        secondInstance.AddComponent<SingletonImplementation>();
    }
}
```

Notez que l'un des avantages du modèle singleton est qu'il est possible d'accéder à une référence à l'instance de manière statique:

```
// Logs: String Instance
Debug.Log(SingletonImplementation.Instance.Text);
```

Gardez à l'esprit que cette pratique doit être minimisée afin de réduire le couplage. Cette approche a également un faible coût de performance en raison de l'utilisation de Dictionary, mais comme cette collection ne peut contenir qu'une seule instance de chaque classe de singleton, le compromis en termes de principe DRY (Don't Repeat Yourself), de lisibilité et la commodité est petite.

Motif Singleton utilisant le système Unity Entity-Component

L'idée principale est d'utiliser GameObjects pour représenter les singletons, ce qui présente de nombreux avantages:

- Réduit au minimum la complexité mais supporte des concepts comme l'injection de dépendance
- Les singletons ont un cycle de vie Unity normal dans le cadre du système Entity-Component
- Les singletons peuvent être chargés et mis en cache localement lorsque cela est nécessaire (par exemple dans les boucles de mise à jour)
- Pas de champs statiques nécessaires
- Pas besoin de modifier les MonoBehaviours / Composants existants pour les utiliser comme Singletons
- Facile à réinitialiser (il suffit de détruire le Singletons GameObject), sera à nouveau chargé lors de la prochaine utilisation
- Mock facile à injecter (juste l'initialiser avec la maquette avant de l'utiliser)
- Inspection et configuration avec l'éditeur Unity normal et peut déjà se produire à l'heure de l'éditeur ([Capture d'écran d'un Singleton accessible dans l'éditeur Unity](#))

Test.cs (qui utilise l'exemple singleton):

```
using UnityEngine;
using UnityEngine.Assertions;

public class Test : MonoBehaviour {
    void Start() {
        ExampleSingleton singleton = ExampleSingleton.instance;
        Assert.IsNotNull(singleton); // automatic initialization on first usage
        Assert.AreEqual("abc", singleton.myVar1);
        singleton.myVar1 = "123";
        // multiple calls to instance() return the same object:
        Assert.AreEqual(singleton, ExampleSingleton.instance);
        Assert.AreEqual("123", ExampleSingleton.instance.myVar1);
    }
}
```

ExampleSingleton.cs (qui contient un exemple et la classe Singleton actuelle):

```
using UnityEngine;
using UnityEngine.Assertions;

public class ExampleSingleton : MonoBehaviour {
    public static ExampleSingleton instance { get { return Singleton.get<ExampleSingleton>(); } }
    public string myVar1 = "abc";
    public void Start() { Assert.AreEqual(this, instance, "Singleton more than once in
```

```

        scene");
    }

/// <summary> Helper that turns any MonoBehaviour or other Component into a Singleton
</summary>
public static class Singleton {
    public static T get<T>() where T : Component {
        return GetOrAddGo("Singletons").GetOrAddChild("") + typeof(T)).GetOrAddComponent<T>();
    }
    private static GameObject GetOrAddGo(string goName) {
        var go = GameObject.Find(goName);
        if (go == null) { return new GameObject(goName); }
        return go;
    }
}

public static class GameObjectExtensionMethods {
    public static GameObject GetOrAddChild(this GameObject parentGo, string childName) {
        var childGo = parentGo.transform.FindChild(childName);
        if (childGo != null) { return childGo.gameObject; } // child found, return it
        var newChild = new GameObject(childName);           // no child found, create it
        newChild.transform.SetParent(parentGo.transform, false); // add it to parent
        return newChild;
    }

    public static T GetOrAddComponent<T>(this GameObject parentGo) where T : Component {
        var comp = parentGo.GetComponent<T>();
        if (comp == null) { return parentGo.AddComponent<T>(); }
        return comp;
    }
}

```

Les deux méthodes d'extension pour GameObject sont également utiles dans d'autres situations, si vous n'en avez pas besoin, déplacez-les dans la classe Singleton et rendez-les privées.

Classe Singleton basée sur MonoBehaviour & ScriptableObject

La plupart des exemples Singleton utilisent MonoBehaviour comme classe de base. Le principal inconvénient est que cette classe Singleton ne vit que pendant l'exécution. Cela a des inconvénients:

- Il n'y a aucun moyen de modifier directement les champs singleton autres que la modification du code.
- Aucun moyen de stocker une référence à d'autres actifs sur le Singleton.
- Aucun moyen de définir le singleton comme destination d'un événement Unity UI. Je finis par utiliser ce que j'appelle des "composants proxy" dont la seule proposition est d'avoir 1 méthode de ligne appelant "GameManager.Instance.SomeGlobalMethod ()".

Comme indiqué dans les remarques, il existe des implémentations qui tentent de résoudre ce problème en utilisant ScriptableObjects comme classe de base, mais perdent les avantages d'exécution du MonoBehaviour. Cette implémentation résout ce problème en utilisant un scriptableObject en tant que classe de base et un comportement mono associé lors de l'exécution:

- C'est un atout pour que ses propriétés puissent être mises à jour dans l'éditeur comme

n'importe quel autre actif Unity.

- Il joue bien avec le processus de sérialisation Unity.
- Est-il possible d'attribuer des références sur le singleton à d'autres actifs de l'éditeur (les dépendances sont injectées via l'éditeur).
- Les événements Unity peuvent appeler directement des méthodes sur le Singleton.
- Peut l'appeler de n'importe où dans la base de code en utilisant "SingletonClassName.Instance"
- A accès aux événements et méthodes MonoBehaviour en cours d'exécution, tels que: Mise à jour, Réveil, Démarrer, FixedUpdate, StartCoroutine, etc.

```
/*
 * Better Singleton by David Darias
 * Use as you like - credit where due would be appreciated :D
 * Licence: WTFPL V2, Dec 2014
 * Tested on Unity v5.6.0 (should work on earlier versions)
 * 03/02/2017 - v1.1
 */

using System;
using UnityEngine;
using SingletonScriptableObjectNamespace;

public class SingletonScriptableObject<T> :
SingletonScriptableObjectNamespace.BehaviourScriptableObject where T :
SingletonScriptableObjectNamespace.BehaviourScriptableObject
{
    //Private reference to the scriptable object
    private static T _instance;
    private static bool _instantiated;
    public static T Instance
    {
        get
        {
            if (_instantiated) return _instance;
            var singletonName = typeof(T).Name;
            //Look for the singleton on the resources folder
            var assets = Resources.LoadAll<T>("");
            if (assets.Length > 1) Debug.LogError("Found multiple " + singletonName + "s on
the resources folder. It is a Singleton ScriptableObject, there should only be one.");
            if (assets.Length == 0)
            {
                _instance = CreateInstance<T>();
                Debug.LogError("Could not find a " + singletonName + " on the resources
folder. It was created at runtime, therefore it will not be visible on the assets folder and
it will not persist.");
            }
            else _instance = assets[0];
            _instantiated = true;
            //Create a new game object to use as proxy for all the MonoBehaviour methods
            var baseObject = new GameObject(singletonName);
            //Deactivate it before adding the proxy component. This avoids the execution of
the Awake method when the proxy component is added.
            baseObject.SetActive(false);
            //Add the proxy, set the instance as the parent and move to DontDestroyOnLoad
scene
            SingletonScriptableObjectNamespace.BehaviourProxy proxy =
baseObject.AddComponent<SingletonScriptableObjectNamespace.BehaviourProxy>();
            proxy.Parent = _instance;
```

```

        Behaviour = proxy;
        DontDestroyOnLoad(Behaviour.gameObject);
        //Activate the proxy. This will trigger the MonoBehaviourAwake.
        proxy.gameObject.SetActive(true);
        return _instance;
    }
}

//Use this reference to call MonoBehaviour specific methods (for example StartCoroutine)
protected static MonoBehaviour Behaviour;
public static void BuildSingletonInstance() {
SingletonScriptableObjectBehaviour i = Instance; }
private void OnDestroy(){ _instantiated = false; }
}

// Helper classes for the SingletonScriptableObject
namespace SingletonScriptableObjectNamespace
{
#if UNITY_EDITOR
//Empty custom editor to have cleaner UI on the editor.
using UnityEditor;
[CustomEditor(typeof(BehaviourProxy))]
public class BehaviourProxyEditor : Editor
{
    public override void OnInspectorGUI(){}
}

#endif

public class BehaviourProxy : MonoBehaviour
{
    public IBehaviour Parent;

    public void Awake() { if (Parent != null) Parent.MonoBehaviourAwake(); }
    public void Start() { if (Parent != null) Parent.Start(); }
    public void Update() { if (Parent != null) Parent.Update(); }
    public void FixedUpdate() { if (Parent != null) Parent.FixedUpdate(); }
}

public interface IBehaviour
{
    void MonoBehaviourAwake();
    void Start();
    void Update();
    void FixedUpdate();
}

public class BehaviourScriptableObject : ScriptableObject, IBehaviour
{
    public void Awake() { ScriptableObjectAwake(); }
    public virtual void ScriptableObjectAwake() { }
    public virtual void MonoBehaviourAwake() { }
    public virtual void Start() { }
    public virtual void Update() { }
    public virtual void FixedUpdate() { }
}
}

```

Ici, il y a un exemple de classe singleton GameManager utilisant SingletonScriptableObject (avec beaucoup de commentaires):

```

using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

//this attribute is optional but recommended. It will allow the creation of the singleton via
//the asset menu.
//the singleton asset should be on the Resources folder.
[CreateAssetMenu(fileName = "GameManager", menuName = "Game Manager", order = 0)]
public class GameManager : SingletonScriptableObject<GameManager> {

    //any properties as usual
    public int Lives;
    public int Points;

    //optional (but recommended)
    //this method will run before the first scene is loaded. Initializing the singleton here
    //will allow it to be ready before any other GameObjects on every scene and will
    //will prevent the "initialization on first usage".
    [RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.BeforeSceneLoad)]
    public static void BeforeSceneLoad() { BuildSingletonInstance(); }

    //optional,
    //will run when the Singleton Scriptable Object is first created on the assets.
    //Usually this happens on edit mode, not runtime. (the override keyword is mandatory for
    this to work)
    public override void ScriptableObjectAwake() {
        Debug.Log(GetType().Name + " created.");
    }

    //optional,
    //will run when the associated MonoBehavioir awakes. (the override keyword is mandatory
    for this to work)
    public override void MonoBehaviourAwake() {
        Debug.Log(GetType().Name + " behaviour awake.");
    }

    //A coroutine example:
    //Singleton Objects do not have coroutines.
    //if you need to use coroutines use the attached MonoBehaviour
    Behaviour.StartCoroutine(SimpleCoroutine());
}

//any methods as usual
private IEnumerator SimpleCoroutine() {
    while(true) {
        Debug.Log(GetType().Name + " coroutine step.");
        yield return new WaitForSeconds(3);
    }
}

//optional,
//Classic runtime Update method (the override keyword is mandatory for this to work).
public override void Update() {

}

//optional,
//Classic runtime FixedUpdate method (the override keyword is mandatory for this to work).
public override void FixedUpdate() {

}

```

```
}

/*
*  Notes:
*  - Remember that you have to create the singleton asset on edit mode before using it. You
have to put it on the Resources folder and of course it should be only one.
*  - Like other Unity Singleton this one is accessible anywhere in your code using the
"Instance" property i.e: GameManager.Instance
*/
```

Lire Singletons in Unity en ligne: <https://riptutorial.com/fr/unity3d/topic/2137/singletons-in-unity>

Chapitre 33: Système audio

Introduction

Ceci est une documentation sur la lecture audio dans Unity3D.

Examples

Classe audio - Lecture audio

```
using UnityEngine;

public class Audio : MonoBehaviour {
    AudioSource audioSource;
    AudioClip audioClip;

    void Start() {
        audioClip = (AudioClip)Resources.Load("Audio/Soundtrack");
        audioSource.clip = audioClip;
        if (!audioSource.isPlaying) audioSource.Play();
    }
}
```

Lire Système audio en ligne: <https://riptutorial.com/fr/unity3d/topic/8064/systeme-audio>

Chapitre 34: Système d'interface utilisateur (UI)

Exemples

S'abonner à l'événement dans le code

Par défaut, il convient de s'abonner à un événement en utilisant l'inspecteur, mais il est parfois préférable de le faire en code. Dans cet exemple, nous vous abonnons pour cliquer sur l'événement d'un bouton afin de le gérer.

```
using UnityEngine;
using UnityEngine.UI;

[RequireComponent(typeof(Button))]
public class AutomaticClickHandler : MonoBehaviour
{
    private void Awake()
    {
        var button = this.GetComponent<Button>();
        button.onClick.AddListener(HandleClick);
    }

    private void HandleClick()
    {
        Debug.Log("AutomaticClickHandler.HandleClick()", this);
    }
}
```

Les composants de l'interface utilisateur fournissent généralement facilement leur auditeur principal:

- Bouton: `onClick`
- Dropdown: `onValueChanged`
- InputField: `onEndEdit` , `onValidateInput` , `onValueChanged`
- Barre de défilement: `onValueChanged`
- ScrollRect: `onValueChanged`
- Slider: `onValueChanged`
- Toggle: `onValueChanged`

Ajouter des écouteurs de souris

Parfois, vous souhaitez ajouter des écouteurs sur des événements particuliers non fournis de manière native par les composants, en particulier des événements de souris. Pour ce faire, vous devrez les ajouter vous-même en utilisant un composant `EventTrigger`:

```
using UnityEngine;
using UnityEngine.EventSystems;
```

```
[RequireComponent(typeof(EventTrigger))]
public class CustomListenersExample : MonoBehaviour
{
    void Start()
    {
        EventTrigger eventTrigger = GetComponent<EventTrigger>();
        EventTrigger.Entry entry = new EventTrigger.Entry();
        entry.eventID = EventTriggerType.PointerDown;
        entry.callback.AddListener( ( data ) => { OnPointerDownDelegate(
(PointerEventData)data ) } );
        eventTrigger.triggers.Add( entry );
    }

    public void OnPointerDownDelegate( PointerEventData data )
    {
        Debug.Log( "OnPointerDownDelegate called." );
    }
}
```

Divers eventID sont possibles:

- PointeurEnter
- PointerExit
- PointerDown
- Pointeur
- PointerClick
- Traîne
- Laissez tomber
- Faire défiler
- UpdateSelected
- Sélectionner
- Désélectionner
- Bouge toi
- InitializePotentialDrag
- BeginDrag
- EndDrag
- Soumettre
- Annuler

Lire Système d'interface utilisateur (UI) en ligne:

<https://riptutorial.com/fr/unity3d/topic/2296/systeme-d-interface-utilisateur--ui->

Chapitre 35: Système d'interface utilisateur graphique en mode immédiat (IMGUI)

Syntaxe

- public statique void GUILayout.Label (string text, options params GUILayoutOption [])
- bool statique public GUILayout.Button (string text, options params GUILayoutOption [])
- chaîne statique publique GUILayout.TextArea (texte de chaîne, options params GUILayoutOption [])

Exemples

GUILayout

Ancien outil système UI, maintenant utilisé pour le prototypage rapide et simple ou le débogage dans le jeu.

```
void OnGUI ()  
{  
    GUILayout.Label ("I'm a simple label text displayed in game.");  
  
    if ( GUILayout.Button("CLICK ME") )  
    {  
        GUILayout.TextArea ("This is a \n  
                           multiline comment.");  
    }  
}
```

La fonction **GUILayout** fonctionne dans la fonction **OnGUI** .

Lire Système d'interface utilisateur graphique en mode immédiat (IMGUI) en ligne:

<https://riptutorial.com/fr/unity3d/topic/6947/systeme-d-interface-utilisateur-graphique-en-mode-immédiat--imgui->

Chapitre 36: Système de saisie

Exemples

Clé de lecture Appui et différence entre GetKey, GetKeyDown et GetKeyUp

L'entrée doit lire depuis la fonction de mise à jour.

Référence pour tous les énumérations **KeyCode** disponibles.

1. Appuyez sur la touche avec `Input.GetKeyDown` :

`Input.GetKeyDown` **plusieurs fois** `true` si l'utilisateur maintient la touche spécifiée. Cela peut être utilisé pour tirer à **plusieurs reprises** une arme tout en maintenant la touche spécifiée. Vous trouverez ci-dessous un exemple de déclenchement automatique de la puce lorsque la touche Espace est enfoncée. Le joueur n'a pas besoin d'appuyer et de relâcher la touche encore et encore.

```
public GameObject bulletPrefab;
public float shootForce = 50f;

void Update()
{
    if (Input.GetKeyDown(KeyCode.Space))
    {
        Debug.Log("Shooting a bullet while SpaceBar is held down");

        //Instantiate bullet
        GameObject bullet = Instantiate(bulletPrefab, transform.position, transform.rotation)
as GameObject;

        //Get the Rigidbody from the bullet then add a force to the bullet
        bullet.GetComponent<Rigidbody>().AddForce(bullet.transform.forward * shootForce);
    }
}
```

2. Appuyez sur la touche avec `Input.GetKeyDown` :

`Input.GetKeyDown` ne sera vrai qu'une **fois** lorsque la touche spécifiée est pressée. C'est la différence clé entre `Input.GetKey` et `Input.GetKeyDown`. Un exemple d'utilisation de cette utilisation est d'activer / désactiver une interface utilisateur ou une lampe de poche ou un élément.

```
public Light flashLight;
bool enableFlashLight = false;

void Update()
{
    if (Input.GetKeyDown(KeyCode.Space))
    {
        //Toggle Light
        enableFlashLight = !enableFlashLight;
        if (enableFlashLight)
        {
```

```

        flashLight.enabled = true;
        Debug.Log("Light Enabled!");
    }
    else
    {
        flashLight.enabled = false;
        Debug.Log("Light Disabled!");
    }
}
}

```

3. Appuyez sur la touche avec `Input.GetKeyDown` :

C'est l'exact opposé de `Input.GetKeyDown`. Il est utilisé pour détecter le relâchement / le relâchement de la pression d'une touche. Tout comme `Input.GetKeyDown`, il ne retourne `true` qu'une seule fois . Par exemple, vous pouvez enable lumière lorsque la touche est enfoncée avec `Input.GetKeyDown` puis désactiver la lumière lorsque la clé est libérée avec `Input.GetKeyUp` .

```

public Light flashLight;
void Update()
{
    //Disable Light when Space Key is pressed
    if (Input.GetKeyDown(KeyCode.Space))
    {
        flashLight.enabled = true;
        Debug.Log("Light Enabled!");
    }

    //Disable Light when Space Key is released
    if (Input.GetKeyUp(KeyCode.Space))
    {
        flashLight.enabled = false;
        Debug.Log("Light Disabled!");
    }
}

```

Capteur d'accéléromètre de lecture (Basic)

`Input.acceleration` est utilisée pour lire le capteur de l'accéléromètre. Il renvoie `Vector3` qui contient les valeurs des axes `x` , `y` et `z` dans un espace 3D.

```

void Update()
{
    Vector3 acclerometerValue = rawAccelValue();
    Debug.Log("X: " + acclerometerValue.x + " Y: " + acclerometerValue.y + " Z: " +
acclerometerValue.z);
}

Vector3 rawAccelValue()
{
    return Input.acceleration;
}

```

Capteur d'accéléromètre de lecture (avance)

L'utilisation de valeurs brutes directement à partir du capteur de l'accéléromètre pour déplacer ou faire pivoter un objet `GameObject` peut entraîner des problèmes tels que des mouvements saccadés ou des vibrations. Il est recommandé de lisser les valeurs avant de les utiliser. En fait, les valeurs du capteur de l'accéléromètre doivent toujours être lissées avant utilisation. Cela peut être accompli avec un filtre passe-bas et c'est là que `Vector3.Lerp` entre en place.

```
//The lower this value, the less smooth the value is and faster Accel is updated. 30 seems
fine for this
const float updateSpeed = 30.0f;

float AccelerometerUpdateInterval = 1.0f / updateSpeed;
float LowPassKernelWidthInSeconds = 1.0f;
float LowPassFilterFactor = 0;
Vector3 lowPassValue = Vector3.zero;

void Start()
{
    //Filter Accelerometer
    LowPassFilterFactor = AccelerometerUpdateInterval / LowPassKernelWidthInSeconds;
    lowPassValue = Input.acceleration;
}

void Update()
{
    //Get Raw Accelerometer values (pass in false to get raw Accelerometer values)
    Vector3 rawAccelValue = filterAccelValue(false);
    Debug.Log("RAW X: " + rawAccelValue.x + " Y: " + rawAccelValue.y + " Z: " +
rawAccelValue.z);

    //Get smoothed Accelerometer values (pass in true to get Filtered Accelerometer values)
    Vector3 filteredAccelValue = filterAccelValue(true);
    Debug.Log("FILTERED X: " + filteredAccelValue.x + " Y: " + filteredAccelValue.y + " Z: " +
filteredAccelValue.z);
}

//Filter Accelerometer
Vector3 filterAccelValue(bool smooth)
{
    if (smooth)
        lowPassValue = Vector3.Lerp(lowPassValue, Input.acceleration, LowPassFilterFactor);
    else
        lowPassValue = Input.acceleration;

    return lowPassValue;
}
```

Capteur d'accéléromètre de lecture (précision)

Lisez le capteur de l'accéléromètre avec précision.

Cet exemple alloue de la mémoire:

```
void Update()
{
    //Get Precise Accelerometer values
    Vector3 accelValue = preciseAccelValue();
```

```

        Debug.Log("PRECISE X: " + accelValue.x + " Y: " + accelValue.y + " Z: " + accelValue.z);

    }

Vector3 preciseAccelValue()
{
    Vector3 accelResult = Vector3.zero;
    foreach (AccelerationEvent tempAccelEvent in Input.accelerationEvents)
    {
        accelResult = accelResult + (tempAccelEvent.acceleration * tempAccelEvent.deltaTime);
    }
    return accelResult;
}

```

Cet exemple n'alloue **pas de** mémoire:

```

void Update()
{
    //Get Precise Accelerometer values
    Vector3 accelValue = preciseAccelValue();
    Debug.Log("PRECISE X: " + accelValue.x + " Y: " + accelValue.y + " Z: " + accelValue.z);

}

Vector3 preciseAccelValue()
{
    Vector3 accelResult = Vector3.zero;
    for (int i = 0; i < Input.accelerationEventCount; ++i)
    {
        AccelerationEvent tempAccelEvent = Input.GetAccelerationEvent(i);
        accelResult = accelResult + (tempAccelEvent.acceleration * tempAccelEvent.deltaTime);
    }
    return accelResult;
}

```

Notez que cela n'est pas filtré. S'il vous plaît regardez [ici](#) comment lisser les valeurs de l'accéléromètre pour supprimer le bruit.

Cliquez sur le bouton de la souris (gauche, milieu, droite)

Ces fonctions sont utilisées pour vérifier les clics du bouton de la souris.

- `Input.GetMouseButton(int button);`
- `Input.GetMouseDown(int button);`
- `Input.GetMouseUp(int button);`

Ils prennent tous le même paramètre.

- 0 = clic gauche de la souris.
- 1 = Clic droit de la souris.
- 2 = Clic de souris du milieu.

`GetMouseButton` est utilisé pour détecter quand le bouton de la souris est *maintenu* enfoncé. Il renvoie `true` lorsque le bouton de la souris spécifié est maintenu enfoncé.

```

void Update()
{
    if (Input.GetMouseButton(0))
    {
        Debug.Log("Left Mouse Button Down");
    }

    if (Input.GetMouseButton(1))
    {
        Debug.Log("Right Mouse Button Down");
    }

    if (Input.GetMouseButton(2))
    {
        Debug.Log("Middle Mouse Button Down");
    }
}

```

`GetMouseButton` est utilisé pour détecter le clic de souris. Il retourne `true` s'il est pressé **une fois**. Il ne reviendra pas `true` jusqu'à ce que le bouton de la souris est relâché et appuyez à nouveau.

```

void Update()
{
    if (Input.GetMouseButton(0))
    {
        Debug.Log("Left Mouse Button Clicked");
    }

    if (Input.GetMouseButton(1))
    {
        Debug.Log("Right Mouse Button Clicked");
    }

    if (Input.GetMouseButton(2))
    {
        Debug.Log("Middle Mouse Button Clicked");
    }
}

```

`GetMouseButtonUp` est utilisé pour détecter le moment où le bouton de la souris spécifié est libéré. Cela ne renverra `true` lorsque le bouton de la souris spécifié sera relâché. Pour revenir à nouveau, il faut appuyer à nouveau et relâcher.

```

void Update()
{
    if (Input.GetMouseButtonUp(0))
    {
        Debug.Log("Left Mouse Button Released");
    }

    if (Input.GetMouseButtonUp(1))
    {
        Debug.Log("Right Mouse Button Released");
    }

    if (Input.GetMouseButtonUp(2))
    {
        Debug.Log("Middle Mouse Button Released");
    }
}

```

```
    }  
}
```

Lire Système de saisie en ligne: <https://riptutorial.com/fr/unity3d/topic/3413/systeme-de-saisie>

Chapitre 37: Unité d'animation

Exemples

Animation de base pour la course

Ce code montre un exemple simple d'animation dans Unity.

Pour cet exemple, vous devriez avoir 2 clips d'animation; Run et Idle. Ces animations doivent être des mouvements sur place. Une fois les clips d'animation sélectionnés, créez un contrôleur Animator. Ajoutez ce contrôleur au lecteur ou à l'objet de jeu que vous souhaitez animer.

Ouvrez la fenêtre Animator à partir de l'option Windows. Faites glisser les 2 clips d'animation dans la fenêtre Animator et 2 états seront créés. Une fois créé, utilisez l'onglet des paramètres de gauche pour ajouter 2 paramètres, tous deux en tant que bool. Nommez-en un comme "PerformRun" et un autre comme "PerformIdle". Définissez "PerformIdle" sur true.

Effectuez des transitions à partir de l'état inactif pour exécuter et exécuter au ralenti (reportez-vous à l'image). Cliquez sur Idle-> Run transition et dans la fenêtre Inspector, désélectionnez HasExit. Faites la même chose pour l'autre transition. Pour Idle-> Run transition, ajoutez une condition: PerformIdle. Pour Run-> Idle, ajoutez une condition: PerformRun. Ajoutez le script C # ci-dessous à l'objet de jeu. Il devrait fonctionner avec l'animation à l'aide du bouton Haut et faire pivoter avec les boutons Gauche et Droite.

```
using UnityEngine;
using System.Collections;

public class RootMotion : MonoBehaviour {

    //Public Variables
    [Header("Transform Variables")]
    public float RunSpeed = 0.1f;
    public float TurnSpeed = 6.0f;

    Animator animator;

    void Start()
    {
        /**
         * Initialize the animator that is attached on the current game object i.e. on which you
         * will attach this script.
         */
        animator = GetComponent<Animator>();
    }

    void Update()
    {
        /**
         * The Update() function will get the bool parameters from the animator state machine and
         * set the values provided by the user.
         * Here, I have only added animation for Run and Idle. When the Up key is pressed, Run
         */
    }
}
```

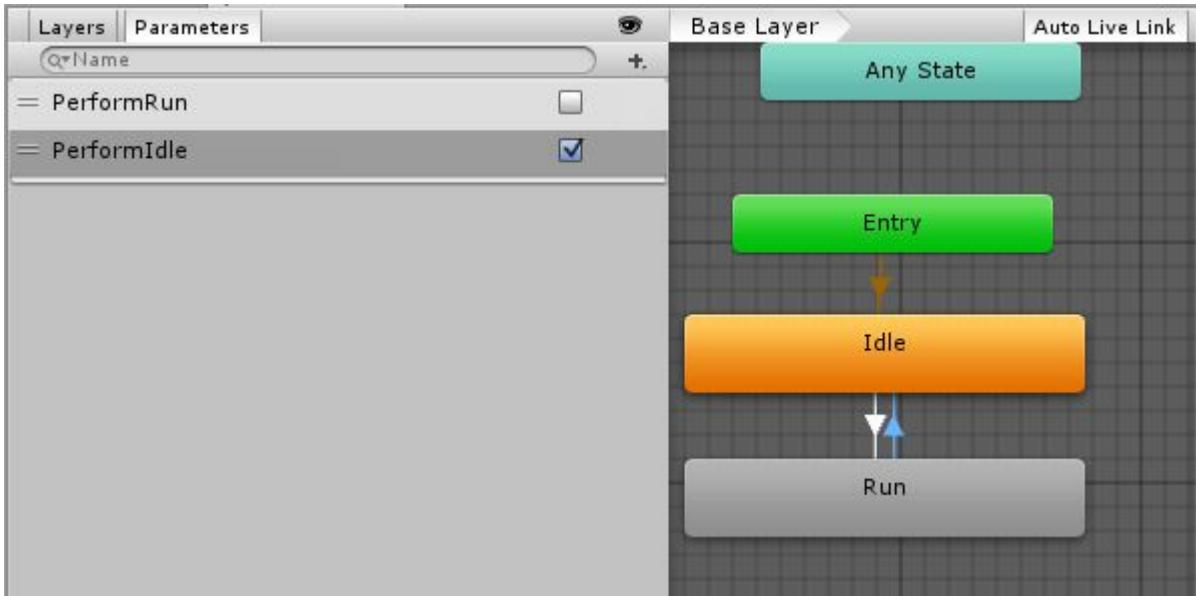
```

animation is played. When we let go, Idle is played.
 */

if (Input.GetKey (KeyCode.UpArrow)) {
    animator.SetBool ("PerformRun", true);
    animator.SetBool ("PerformIdle", false);
} else {
    animator.SetBool ("PerformRun", false);
    animator.SetBool ("PerformIdle", true);
}
}

void OnAnimatorMove()
{
    /**
     * OnAnimatorMove() function will shadow the "Apply Root Motion" on the animator. Your
     game objects position will now be determined
     * using this function.
     */
    if (Input.GetKey (KeyCode.UpArrow)) {
        transform.Translate (Vector3.forward * RunSpeed);
        if (Input.GetKey (KeyCode.RightArrow)) {
            transform.Rotate (Vector3.up * Time.deltaTime * TurnSpeed);
        }
        else if (Input.GetKey (KeyCode.LeftArrow)) {
            transform.Rotate (-Vector3.up * Time.deltaTime * TurnSpeed);
        }
    }
}
}
}

```

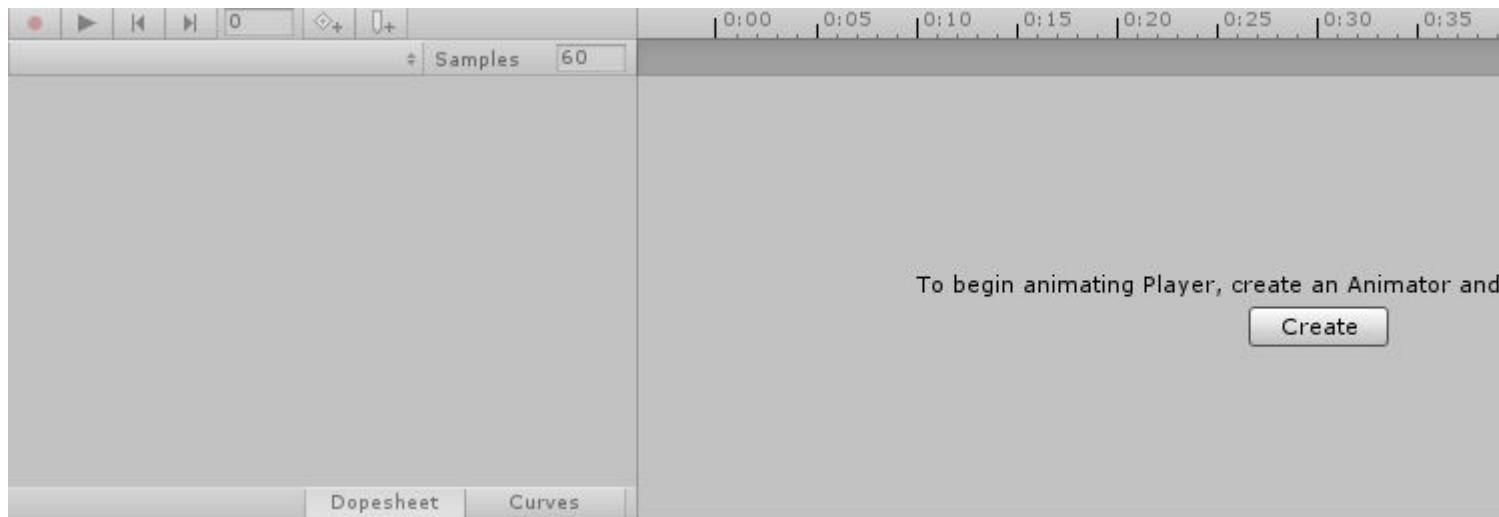


Création et utilisation de clips d'animation

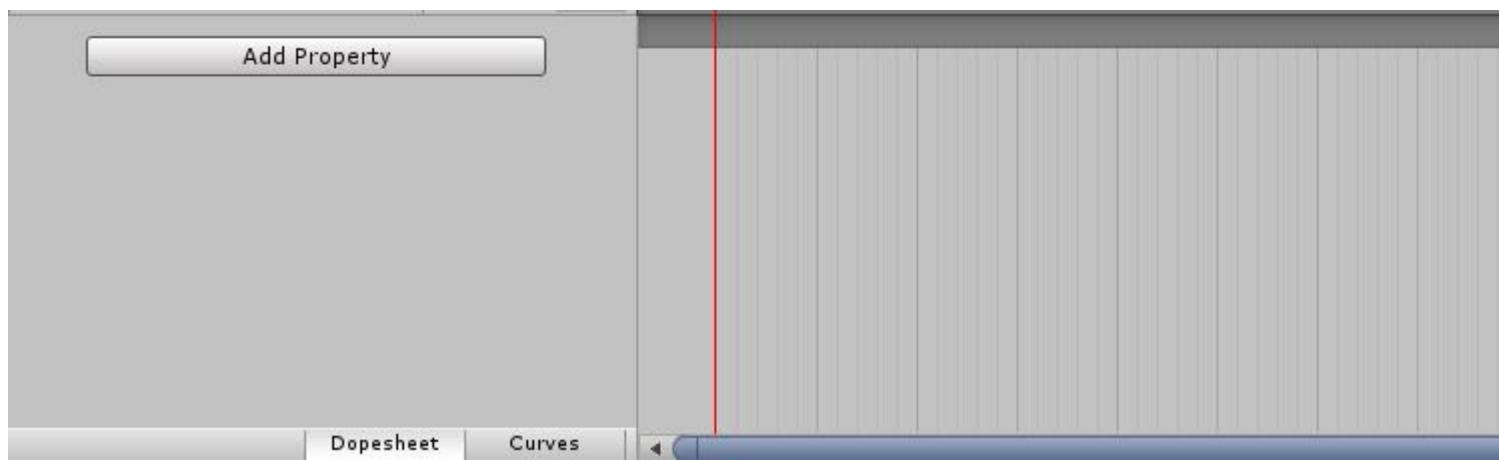
Cet exemple montre comment créer et utiliser des clips d'animation pour des objets de jeu ou des joueurs.

Remarque: les modèles utilisés dans cet exemple sont téléchargés depuis Unity Asset Store. Le lecteur a été téléchargé à partir du lien suivant:
<https://www.assetstore.unity3d.com/en/#!/content/21874>.

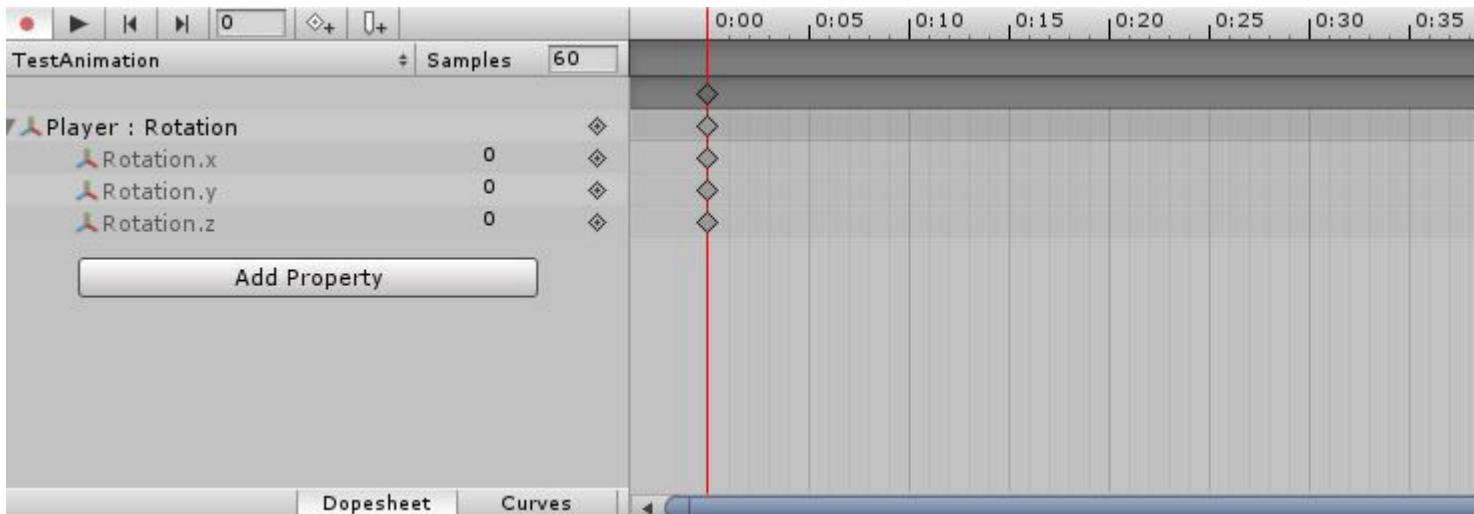
Pour créer des animations, ouvrez d'abord la fenêtre d'animation. Vous pouvez l'ouvrir en cliquant sur Window et Select Animation ou appuyez sur Ctrl + 6. Sélectionnez l'objet de jeu auquel vous souhaitez appliquer le clip d'animation dans la fenêtre Hiérarchie, puis cliquez sur le bouton Créer de la fenêtre d'animation.



Nommez votre animation (comme IdlePlayer, SprintPlayer, DyingPlayer, etc.) et enregistrez-la. Maintenant, à partir de la fenêtre d'animation, cliquez sur le bouton Ajouter une propriété. Cela vous permettra de modifier la propriété de l'objet ou du joueur dans le temps. Cela peut inclure des formes Tran comme la rotation, la position et l'échelle et toute autre propriété attachée à l'objet de jeu, par exemple Collider, Mesh Renderer, etc.



Pour créer une animation en cours d'exécution pour un objet de jeu, vous aurez besoin d'un modèle 3D humanoïde. Vous pouvez télécharger le modèle à partir du lien ci-dessus. Suivez les étapes ci-dessus pour créer une nouvelle animation. Ajoutez une propriété Transform et sélectionnez Rotation pour l'un des segments de caractère.



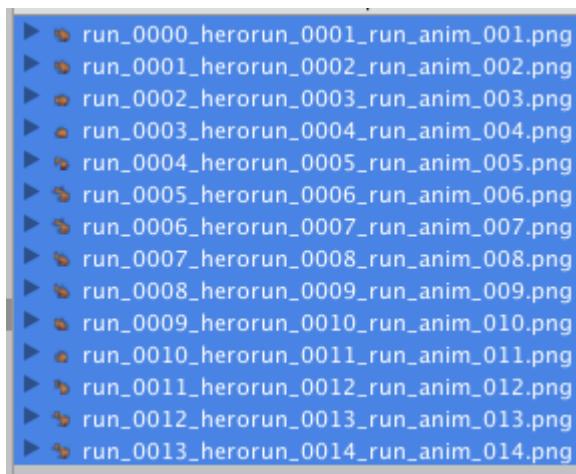
À ce moment, votre bouton de lecture et vos valeurs de rotation dans la propriété objet du jeu seraient devenus rouges. Cliquez sur la flèche déroulante pour voir les valeurs de rotation X, Y et Z. Le temps d'animation par défaut est défini sur 1 seconde. Les animations utilisent des images clés pour interpoler entre les valeurs. Pour animer, ajoutez des clés à différents moments et modifiez les valeurs de rotation à partir de la fenêtre Inspecteur. Par exemple, la valeur de rotation à l'instant 0.0s peut être 0.0. Au temps 0.5s, la valeur peut être 20.0 pour X. Au moment 1.0s, la valeur peut être 0.0. Nous pouvons terminer notre animation à 1.0s.

La longueur de votre animation dépend des dernières clés ajoutées à l'animation. Vous pouvez ajouter d'autres touches pour rendre l'interpolation plus fluide.

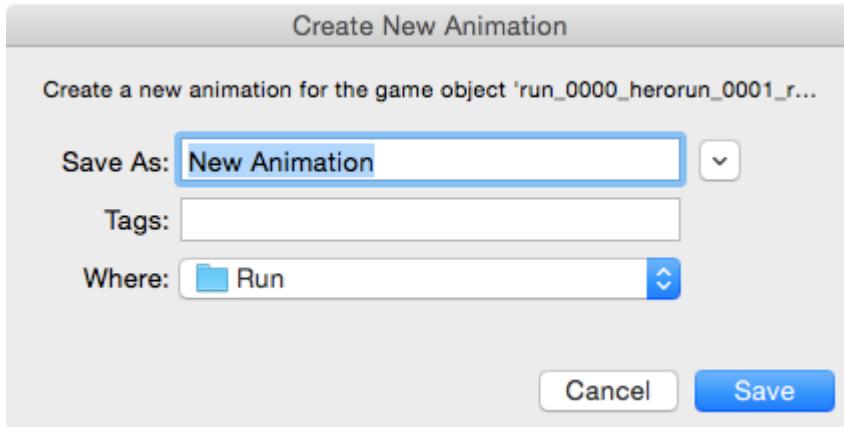
Animation 2D Sprite

L'animation de sprite consiste à afficher une séquence d'images ou de cadres existante.

Commencez par importer une séquence d'images dans le dossier des ressources. Soit créer des images à partir de zéro ou en télécharger à partir de l'Asset Store. (Cet exemple utilise [cet actif gratuit](#).)

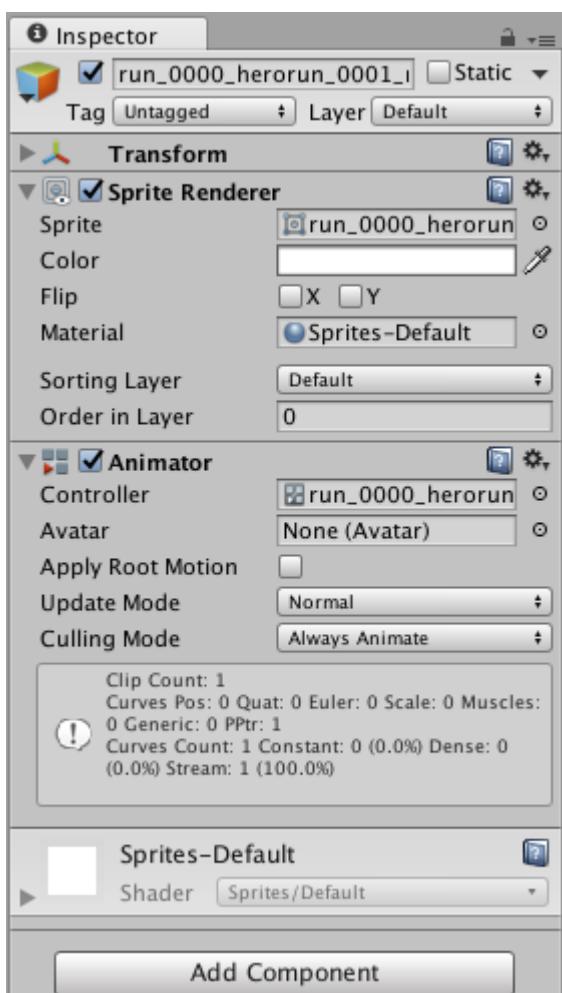


Faites glisser chaque image individuelle d'une animation unique du dossier des ressources vers la vue de la scène. Unity affichera une boîte de dialogue pour nommer le nouveau clip d'animation.



Ceci est un raccourci utile pour:

- créer de nouveaux objets de jeu
- assigner deux composants (un moteur de rendu Sprite et un animateur)
- créer des contrôleurs d'animation (et y associer le nouveau composant Animator)
- créer des clips d'animation avec les images sélectionnées



Prévisualisez la lecture dans l'onglet Animation en cliquant sur Lecture:



La même méthode peut être utilisée pour créer de nouvelles animations pour le même objet de jeu, puis supprimer le nouvel objet de jeu et le nouveau contrôleur d'animation. Ajoutez le nouveau clip d'animation au contrôleur d'animation de cet objet de la même manière qu'avec l'animation 3D.

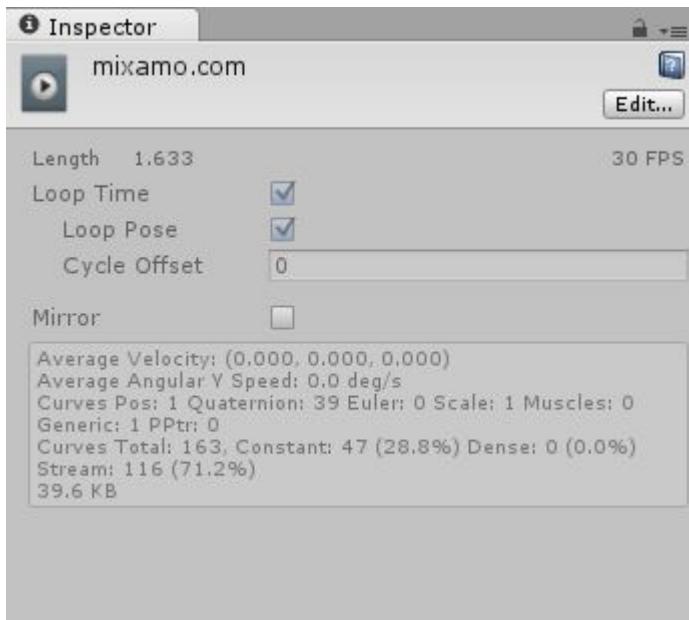
Courbes d'animation

Les courbes d'animation vous permettent de modifier un paramètre flottant au fur et à mesure de la lecture de l'animation. Par exemple, s'il y a une animation de longueur 60 secondes et que vous voulez une valeur / paramètre float, appelez-la X, pour varier dans l'animation (comme au temps d'animation = 0.0s; X = 0.0 au moment de l'animation = 30.0s); X = 1,0, au moment de l'animation = 60,0s; X = 0,0).

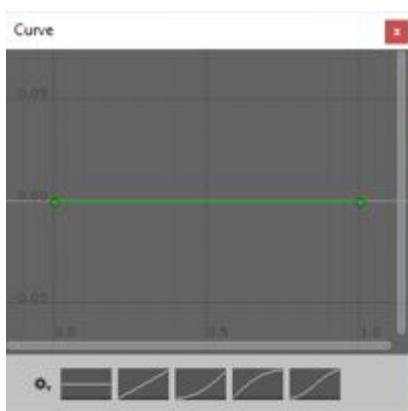
Une fois que vous avez la valeur flottante, vous pouvez l'utiliser pour traduire, faire pivoter, mettre à l'échelle ou l'utiliser de toute autre manière.

Pour mon exemple, je montrerai un objet de jeu de joueur en cours d'exécution. Lorsque l'animation de la lecture est lancée, la vitesse de traduction du lecteur doit augmenter à mesure que l'animation se poursuit. Lorsque l'animation arrive à son terme, la vitesse de traduction doit diminuer.

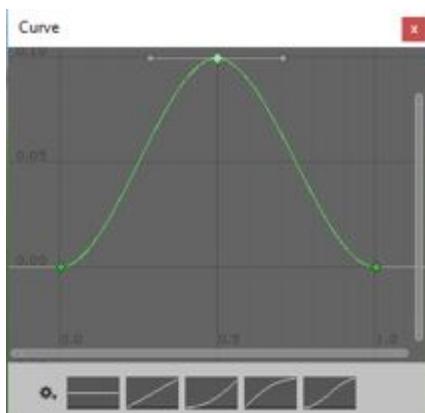
J'ai un clip d'animation en cours de création. Sélectionnez le clip puis dans la fenêtre de l'inspecteur, cliquez sur Modifier.



Une fois là-bas, faites défiler jusqu'à Curves. Cliquez sur le signe + pour ajouter une courbe. Nommez la courbe, par exemple ForwardRunCurve. Cliquez sur la courbe miniature à droite. Il ouvrira une petite fenêtre avec une courbe par défaut.



Nous voulons une courbe de forme parabolique là où elle monte puis diminue. Par défaut, il y a 2 points sur la ligne. Vous pouvez ajouter plus de points en double-cliquant sur la courbe. Faites glisser les points pour créer une forme similaire à la suivante.



Dans la fenêtre de l'animateur, ajoutez le clip en cours d'exécution. Ajoutez également un paramètre float avec le même nom que la courbe, à savoir ForwardRunCurve.

Lors de la lecture de l'animation, la valeur flottante change en fonction de la courbe. Le code suivant montre comment utiliser la valeur float:

```
using UnityEngine;
using System.Collections;

public class RunAnimation : MonoBehaviour {

    Animator animator;
    float curveValue;

    void Start()
    {
        animator = GetComponent<Animator>();
    }

    void Update()
    {
        curveValue = animator.GetFloat("ForwardRunCurve");

        transform.Translate (Vector3.forward * curveValue);
    }
}
```

La variable `curveValue` contient la valeur de la courbe (`ForwardRunCurve`) à un moment donné. Nous utilisons cette valeur pour modifier la vitesse de la traduction. Vous pouvez attacher ce script à l'objet de jeu du joueur.

Lire Unité d'animation en ligne: <https://riptutorial.com/fr/unity3d/topic/5448/unite-d-animation>

Chapitre 38: Unity Profiler

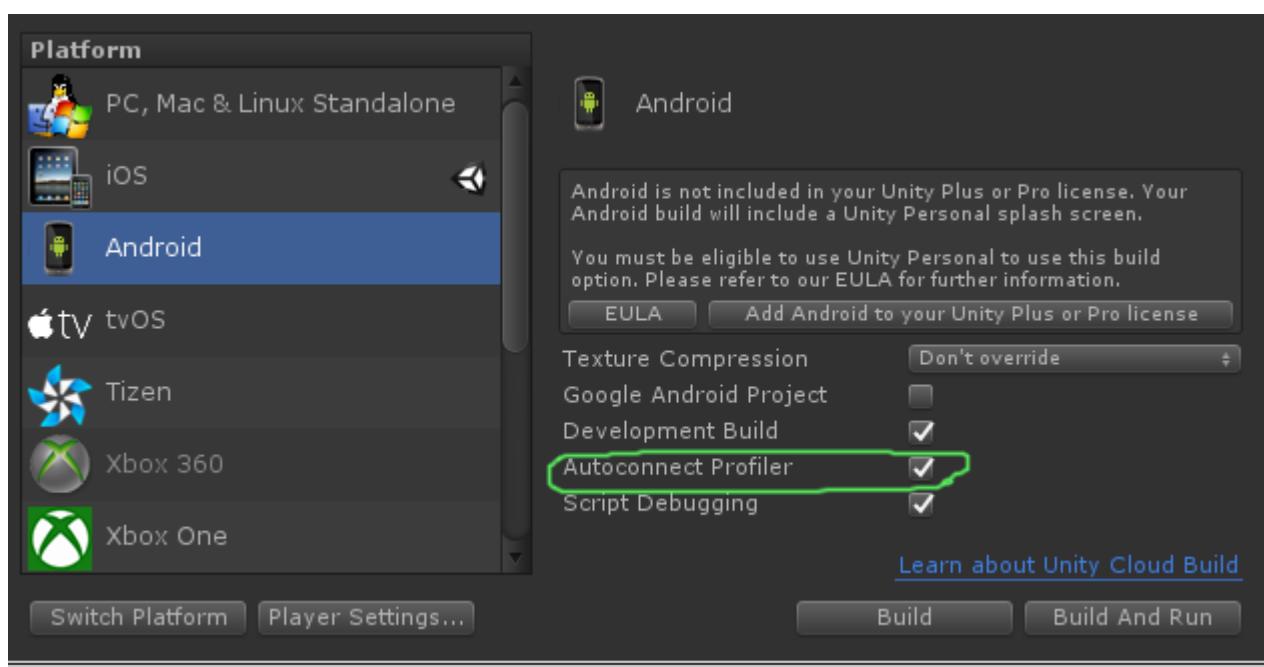
Remarques

Utiliser Profiler sur différents périphériques

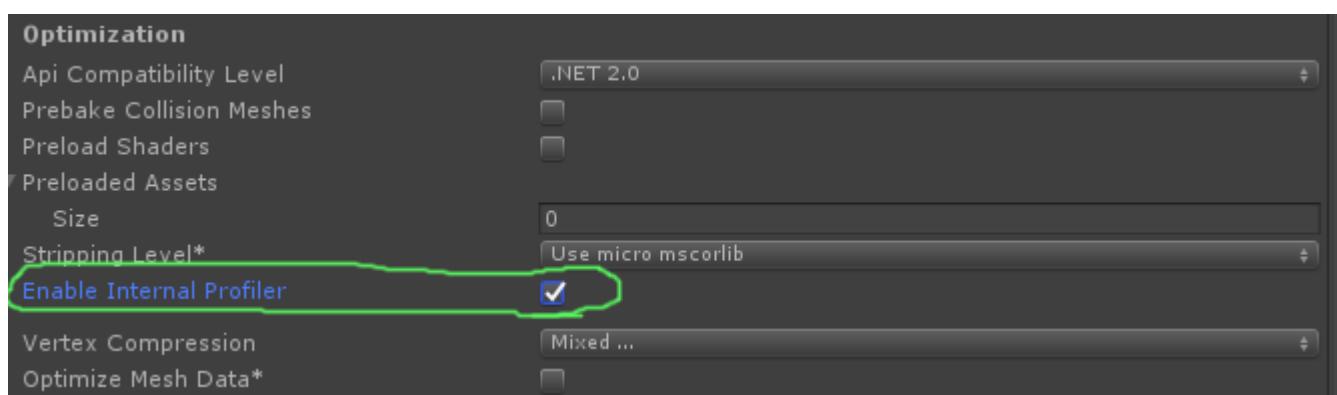
Il y a peu de choses importantes à savoir pour raccorder correctement Profiler sur différentes plates-formes.

Android

Pour bien attacher le profil, il faut utiliser le bouton "Build and Run" de la fenêtre Build Settings avec l'option **Autoconnect Profiler** cochée.



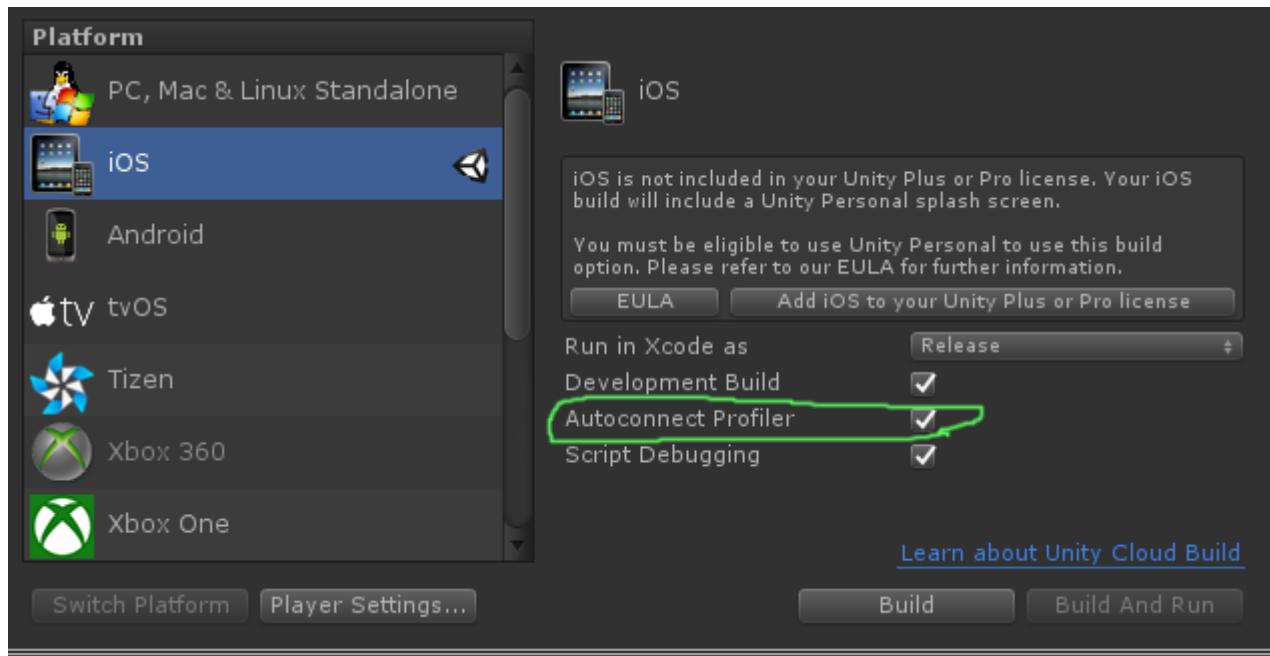
Une autre option obligatoire, dans l'inspecteur de paramètres de lecteur Android dans l'onglet Autres paramètres, est une case à cocher **Activer le profileur interne** qui doit être vérifiée pour que LogCat affiche les informations du profileur.



Utiliser uniquement "Générer" ne permettra pas à l'éditeur de profil de se connecter à un périphérique Android car les "Build and Run" utilisent des arguments de ligne de commande spécifiques pour le démarrer avec LogCat.

iOS

Pour attacher correctement le profil, le bouton "Build and Run" de la fenêtre Build Settings avec l'option **Autoconnect Profiler** cochée doit être utilisé lors de la première exécution.



Sur iOS, aucune option dans les paramètres du lecteur ne doit être définie pour que le profileur soit activé. Il devrait fonctionner hors de la boîte.

Exemples

Balisage du profileur

Utilisation de la classe de profileur

Une bonne pratique consiste à utiliser `Profiler.BeginSample` et `Profiler.EndSample` car il aura sa propre entrée dans la fenêtre du profileur.

De plus, ces balises seront supprimées lors de la construction sans développement en utilisant `ConditionalAttribute`, vous n'avez donc pas besoin de les supprimer de votre code.

```
public class SomeClass : MonoBehaviour
{
    void SomeFunction()
    {
        Profiler.BeginSample("SomeClass.SomeFunction");
        // Various call made here
        Profiler.EndSample();
    }
}
```

```
    }  
}
```

Cela créera une entrée "SomeClass.SomeFunction" dans la fenêtre du profileur qui facilitera le débogage et l'identification du col de la bouteille.

Lire Unity Profiler en ligne: <https://riptutorial.com/fr/unity3d/topic/6974/unity-profiler>

Chapitre 39: Utilisation du contrôle de source Git avec Unity

Examples

Utilisation de GFS Large File Storage (LFS) avec Unity

Avant-propos

Git peut travailler avec le développement de jeux vidéo. Cependant, le principal inconvénient est que le contrôle des versions de fichiers multimédias volumineux (> 5 Mo) peut être un problème à long terme, car votre historique de validation bloque - Git n'a tout simplement pas été conçu pour gérer les fichiers binaires.

La bonne nouvelle est que depuis la mi-2015, GitHub a publié un plug-in pour Git appelé [Git LFS](#), qui traite directement de ce problème. Vous pouvez maintenant facilement et efficacement versionner de gros fichiers binaires!

Enfin, cette documentation est axée sur les exigences et les informations spécifiques nécessaires pour garantir que votre vie Git fonctionne bien avec le développement de jeux vidéo. Ce guide ne couvrira pas la manière d'utiliser Git lui-même.

Installation de Git & Git-LFS

En tant que développeur, vous disposez d'un certain nombre d'options et le premier choix consiste à savoir s'il faut installer la ligne de commande Git principale ou laisser l'une des applications d'interface graphique Git les plus prisées en charge pour vous.

Option 1: Utiliser une application Git GUI

Ceci est vraiment une préférence personnelle, car il existe plusieurs options en termes d'interface graphique de Git ou d'utilisation d'une interface graphique. Vous avez un certain nombre d'applications à choisir, voici 3 des plus populaires:

- [Sourcetree \(gratuit\)](#)
- [Github Desktop \(gratuit\)](#)
- [SmartGit \(commercial\)](#)

Une fois que vous avez installé l'application de votre choix, consultez Google et suivez les instructions pour vous assurer qu'elle est configurée pour Git-LFS. Nous allons sauter cette étape dans ce guide car il est spécifique à l'application.

Option 2: installer Git & Git-LFS

C'est assez simple - [Installez Git](#). Alors, [Installez Git LFS](#).

Configuration du stockage de fichiers Git Large sur votre projet

Si vous utilisez le plug-in Git LFS pour mieux prendre en charge les fichiers binaires, vous devrez définir certains types de fichiers à gérer par Git LFS. Ajoutez ce qui suit à votre fichier `.gitattributes` à la racine de votre référentiel pour prendre en charge les fichiers binaires courants utilisés dans les projets Unity:

```
# Image formats:  
*.tga filter=lfs diff=lfs merge=lfs -text  
*.png filter=lfs diff=lfs merge=lfs -text  
*.tif filter=lfs diff=lfs merge=lfs -text  
*.jpg filter=lfs diff=lfs merge=lfs -text  
*.gif filter=lfs diff=lfs merge=lfs -text  
*.psd filter=lfs diff=lfs merge=lfs -text  
  
# Audio formats:  
*.mp3 filter=lfs diff=lfs merge=lfs -text  
*.wav filter=lfs diff=lfs merge=lfs -text  
*.aiff filter=lfs diff=lfs merge=lfs -text  
  
# 3D model formats:  
*.fbx filter=lfs diff=lfs merge=lfs -text  
*.obj filter=lfs diff=lfs merge=lfs -text  
  
# Unity formats:  
*.sbsar filter=lfs diff=lfs merge=lfs -text  
*.unity filter=lfs diff=lfs merge=lfs -text  
  
# Other binary formats  
*.dll filter=lfs diff=lfs merge=lfs -text
```

Mettre en place un dépôt Git pour Unity

Lors de l'initialisation d'un référentiel Git pour le développement d'Unity, il y a plusieurs choses à faire.

Unité ignore les dossiers

Tout ne devrait pas être versionné dans le référentiel. Vous pouvez ajouter le modèle ci-dessous à votre fichier `.gitignore` dans la racine de votre référentiel. Ou bien, vous pouvez vérifier le code source libre [Unity .gitignore sur GitHub](#) et en générer un en utilisant [gitignore.io pour l'unité](#).

```
# Unity Generated
```

```

[Tt]emp/
[Ll]ibrary/
[Oo]bj/

# Unity3D Generated File On Crash Reports
sysinfo.txt

# Visual Studio / MonoDevelop Generated
ExportedObj/
obj/
*.csproj
*.unityproj
*.sln
*.suo
*.tmp
*.user
*.userprefs
*.pidb
*.boopproj
*.svd

# OS Generated
desktop.ini
.DS_Store
.DS_Store?
.Spotlight-V100
.Trashes
ehthumbs.db
Thumbs.db

```

Pour en savoir plus sur la configuration d'un fichier .gitignore, [consultez ici](#).

Paramètres du projet Unity

Par défaut, les projets Unity ne sont pas configurés pour prendre en charge le contrôle de version correctement.

1. (Ignorez cette étape dans la version 4.5 et ultérieure) Activer l'option `External` dans `Unity -> Preferences -> Packages -> Repository`.
2. Basculez vers les `Visible Meta Files` dans `Edit -> Project Settings -> Editor -> Version Control Mode`.
3. Basculer vers le `Force Text` dans `Edit -> Project Settings -> Editor -> Asset Serialization Mode`.
4. Enregistrez la scène et le projet dans le menu `File`.

Configuration supplémentaire

L'un des rares désagréments majeurs liés à l'utilisation de Git avec les projets Unity est que Git ne se soucie pas des répertoires et laissera volontiers des répertoires vides après avoir supprimé les fichiers. Unity va créer des fichiers `*.meta` pour ces répertoires et peut causer une bataille entre les membres de l'équipe lorsque Git se `*.meta` pour continuer à ajouter et à supprimer ces fichiers

méta.

Ajoutez ce hook post-fusion Git au dossier `/.git/hooks/` pour les référentiels contenant des projets Unity. Après chaque extraction / fusion de Git, il examinera quels fichiers ont été supprimés, vérifie si le répertoire dans lequel il se trouve est vide et, le cas échéant, le supprime.

Scènes et préfabriqués fusionnant

Un problème courant lorsque vous travaillez avec Unity est lorsque 2 développeurs ou plus modifient une scène ou un préfabriqué Unity (fichiers * .unity). Git ne sait pas comment les fusionner correctement. Heureusement, l'équipe d'Unity a déployé un outil appelé [SmartMerge](#), qui permet de fusionner automatiquement les données. La première chose à faire est d'ajouter les lignes suivantes à votre fichier `.git` ou `.gitconfig` : (Windows: `%USERPROFILE%\gitconfig`, Linux / Mac OS X: `~/.gitconfig`)

```
[merge]
tool = unityyamlmerge

[mergetool "unityyamlmerge"]
trustExitCode = false
cmd = '<path to UnityYAMLMerge>' merge -p "$BASE" "$REMOTE" "$LOCAL" "$MERGED"
```

Sous **Windows**, le chemin d'accès à UnityYAMLMerge est le suivant:

```
C:\Program Files\Unity\Editor\Data\Tools\UnityYAMLMerge.exe
```

ou

```
C:\Program Files (x86)\Unity\Editor\Data\Tools\UnityYAMLMerge.exe
```

et sur **MacOSX** :

```
/Applications/Unity/Unity.app/Contents/Tools/UnityYAMLMerge
```

Une fois cela fait, le mergetool sera disponible lorsque des conflits surviendront lors de la fusion / rebase. N'oubliez pas de lancer `git mergetool` manuellement pour déclencher UnityYAMLMerge.

Lire [Utilisation du contrôle de source Git avec Unity en ligne](#):

<https://riptutorial.com/fr/unity3d/topic/2195/utilisation-du-controle-de-source-git-avec-unity>

Chapitre 40: Vector3

Introduction

La structure `Vector3` représente une coordonnée 3D et constitue l'une des structures de base de la bibliothèque `UnityEngine`. La structure `Vector3` se trouve le plus souvent dans la composante `Transform` de la plupart des objets de jeu, où elle est utilisée pour conserver la *position* et l' *échelle*. `Vector3` fournit de bonnes fonctionnalités pour effectuer des opérations vectorielles courantes.

[Vous pouvez en savoir plus sur la structure `Vector3` dans l'API Unity.](#)

Syntaxe

- vecteur public3 ();
- public Vector3 (float x, float y);
- public Vector3 (float x, float y, float z);
- Vector3.Lerp (Vector3 startPosition, Vector3 targetPosition, float movementFraction);
- Vector3.LerpUnclamped (Vector3 startPosition, Vector3 targetPosition, float movementFraction);
- Vector3.MoveTowards (Vector3 startPosition, Vector3 targetPosition, distance flottante);

Exemples

Valeurs statiques

La structure `Vector3` contient des variables statiques qui fournissent des valeurs `Vector3` couramment utilisées. La plupart représentent une *direction* , mais ils peuvent toujours être utilisés de manière créative pour fournir des fonctionnalités supplémentaires.

`Vector3.zero` et `Vector3.one`

`Vector3.zero` et `Vector3.one` sont généralement utilisés en connexion avec un `Vector3` *normalisé* ; c'est-à-dire un `Vector3` où les valeurs `x` , `y` et `z` ont une magnitude de 1. En tant que tel, `Vector3.zero` représente la valeur la plus faible, tandis que `Vector3.one` représente la valeur la plus élevée.

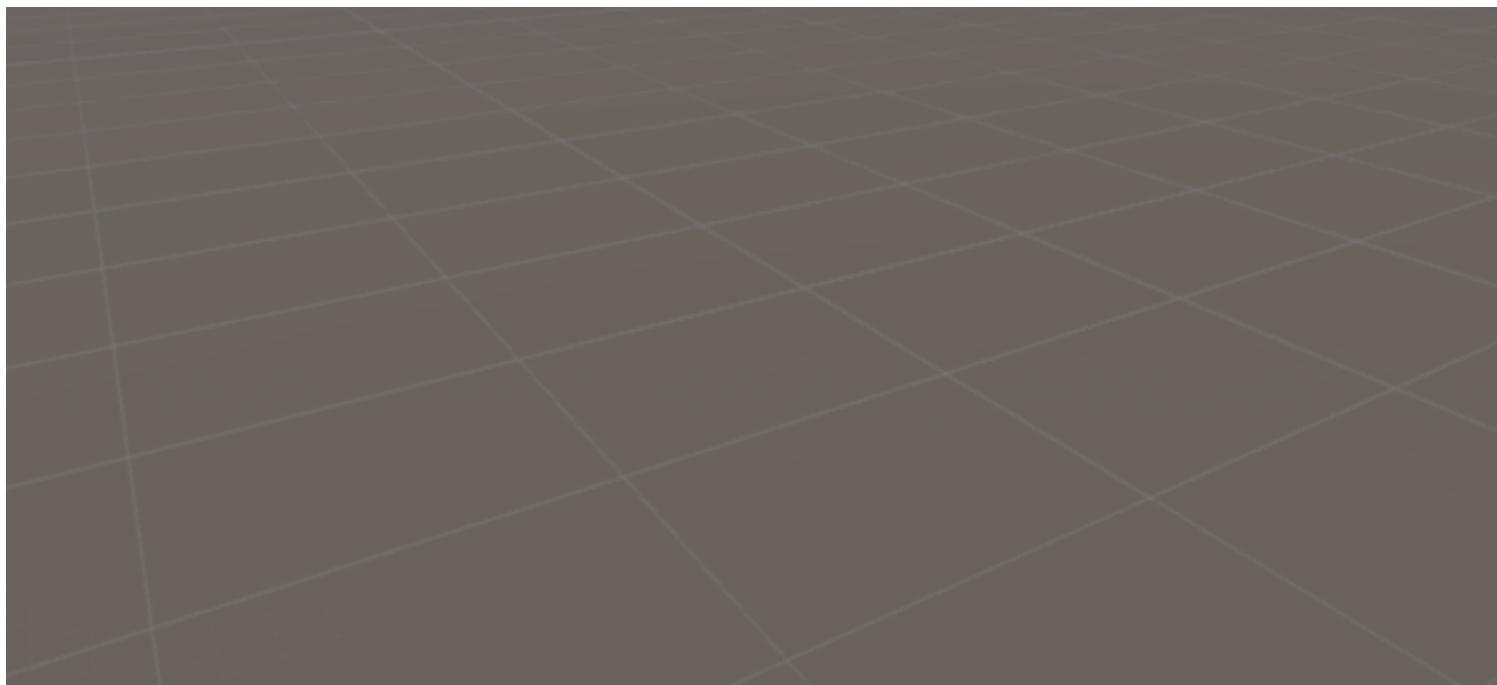
`Vector3.zero` est également couramment utilisé pour définir la position par défaut sur les transformations d'objets.

La classe suivante utilise `Vector3.zero` et `Vector3.one` pour gonfler et dégonfler une sphère.

```
using UnityEngine;
```

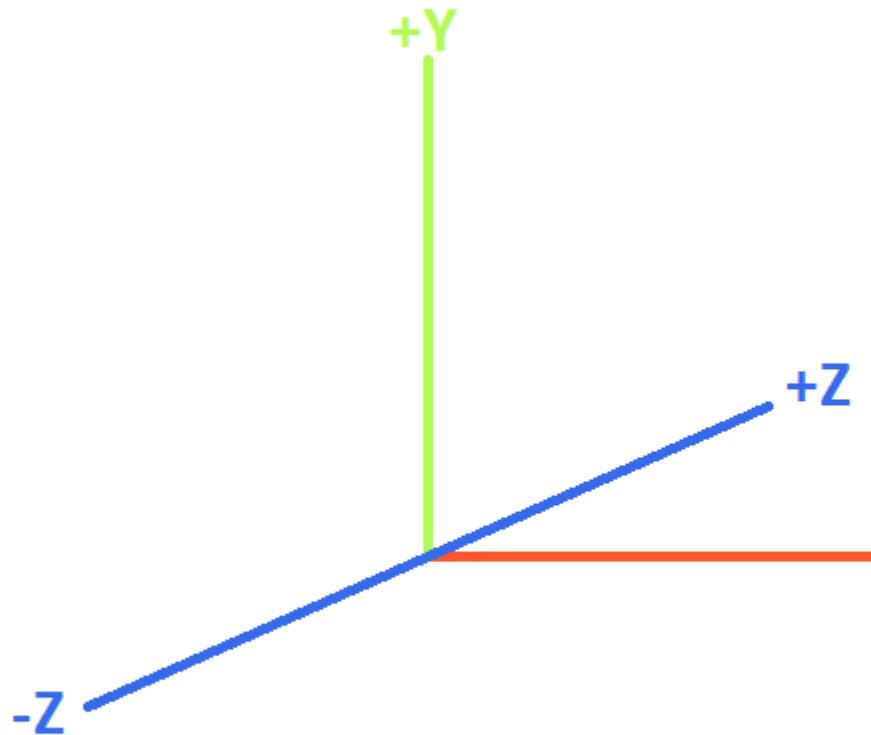
```
public class Inflater : MonoBehaviour
{
    <summary>A sphere set up to inflate and deflate between two values.</summary>
    public ScaleBetween sphere;

    ///<summary>On start, set the sphere GameObject up to inflate
    /// and deflate to the corresponding values.</summary>
    void Start()
    {
        // Vector3.zero = Vector3(0, 0, 0); Vector3.one = Vector3(1, 1, 1);
        sphere.setScale(Vector3.zero, Vector3.one);
    }
}
```



Directions statiques

Les directions statiques peuvent être utiles dans un certain nombre d'applications, la direction étant positive et négative sur les trois axes. Il est important de noter que Unity utilise un système de coordonnées pour gaucher, ce qui a une incidence sur la direction.



LEFT-HANDED COORDINATE SYSTEM

La classe suivante utilise les directions statiques `Vector3` pour déplacer des objets le long des trois axes.

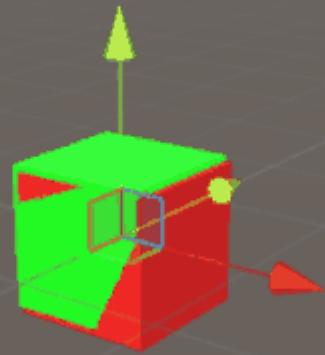
```
using UnityEngine;

public class StaticMover : MonoBehaviour
{
    <summary>GameObjects set up to move back and forth between two directions.</summary>
    public MoveBetween xMovement, yMovement, zMovement;

    ///<summary>On start, set each MoveBetween GameObject up to move
    /// in the corresponding direction(s).</summary>
    void Start()
    {
        // Vector3.left = Vector3(-1, 0, 0); Vector3.right = Vector3(1, 0, 0);
        xMovement.SetDirections(Vector3.left, Vector3.right);

        // Vector3.down = Vector3(0, -1, 0); Vector3.up = Vector3(0, 0, 1);
        yMovement.SetDirections(Vector3.down, Vector3.up);

        // Vector3.back = Vector3(0, 0, -1); Vector3.forward = Vector3(0, 0, 1);
        zMovement.SetDirections(Vector3.back, Vector3.forward);
    }
}
```



Indice

Valeur	X	y	z	<code>new Vector3()</code> méthode <code>new Vector3()</code> équivalente
<code>Vector3.zero</code>	0	0	0	<code>new Vector3(0, 0, 0)</code>
<code>Vector3.one</code>	1	1	1	<code>new Vector3(1, 1, 1)</code>
<code>Vector3.left</code>	-1	0	0	<code>new Vector3(-1, 0, 0)</code>
<code>Vector3.right</code>	1	0	0	<code>new Vector3(1, 0, 0)</code>
<code>Vector3.down</code>	0	-1	0	<code>new Vector3(0, -1, 0)</code>
<code>Vector3.up</code>	0	1	0	<code>new Vector3(0, 1, 0)</code>
<code>Vector3.back</code>	0	0	-1	<code>new Vector3(0, 0, -1)</code>
<code>Vector3.forward</code>	0	0	1	<code>new Vector3(0, 0, 1)</code>

Créer un vecteur3

Une structure `Vector3` peut être créée de plusieurs manières. `Vector3` est une structure et, en tant que telle, devra généralement être instanciée avant utilisation.

Constructeurs

Il y a trois constructeurs intégrés pour instancier un `Vector3` .

Constructeur	Résultat
<code>new Vector3()</code>	Crée une structure <code>Vector3</code> avec les coordonnées de (0, 0, 0).
<code>new Vector3(float x, float y)</code>	Crée une structure <code>Vector3</code> avec les coordonnées <code>x</code> et <code>y</code> . <code>z</code> sera mis à 0.
<code>new Vector3(float x, float y, float z)</code>	Crée une structure <code>Vector3</code> avec les <code>x</code> , <code>y</code> et <code>z</code> .

Conversion à partir d'un `Vector2` ou `Vector4`

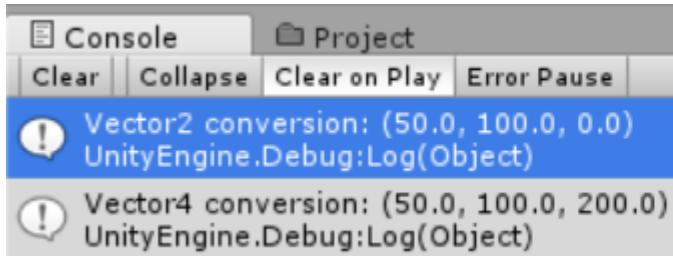
Bien que cela soit rare, vous pouvez rencontrer des situations où vous devrez traiter les coordonnées d'une structure `Vector2` ou `Vector4` tant que `Vector3` . Dans de tels cas, vous pouvez simplement passer le `Vector2` ou le `Vector4` directement dans le `Vector3` , sans l'instancier au préalable. Comme il faut le supposer, une structure `Vector2` ne transmettra que les valeurs `x` et `y` , tandis qu'une classe `Vector4` omettra son `w` .

Nous pouvons voir une conversion directe dans le script ci-dessous.

```
void VectorConversionTest()
{
    Vector2 vector2 = new Vector2(50, 100);
    Vector4 vector4 = new Vector4(50, 100, 200, 400);

    Vector3 fromVector2 = vector2;
    Vector3 fromVector4 = vector4;

    Debug.Log("Vector2 conversion: " + fromVector2);
    Debug.Log("Vector4 conversion: " + fromVector4);
}
```



Mouvement d'application

La structure `Vector3` contient des fonctions statiques pouvant être utiles lorsque vous souhaitez appliquer un mouvement au `Vector3` .

Lerp et LerpUnclamped

Les fonctions lerp fournissent un mouvement entre deux coordonnées basées sur une fraction fournie. Lorsque `Lerp` ne permet que le mouvement entre les deux coordonnées, `LerpUnclamped` permet des fractions qui se déplacent en dehors des limites entre les deux coordonnées.

Nous fournissons la fraction de mouvement comme un `float`. Avec une valeur de `0.5`, on trouve le point milieu entre les deux coordonnées `Vector3`. Une valeur de `0` ou `1` renverra le premier ou le deuxième `Vector3`, respectivement, car ces valeurs sont soit corrélées à aucun mouvement (renvoyant ainsi le premier `Vector3`), soit au mouvement terminé (ceci renvoyant le deuxième `Vector3`). Il est important de noter qu'aucune des deux fonctions ne permettra de modifier la fraction de mouvement. C'est quelque chose que nous devons prendre en compte manuellement.

Avec `Lerp`, toutes les valeurs sont bloquées entre `0` et `1`. Ceci est utile lorsque vous souhaitez vous déplacer vers une direction et que vous ne souhaitez pas dépasser la destination.

`LerpUnclamped` peut prendre n'importe quelle valeur et peut être utilisé pour se déplacer à partir de la destination ou au-*dès* de la destination.

Le script suivant utilise `Lerp` et `LerpUnclamped` pour déplacer un objet à un rythme constant.

```
using UnityEngine;

public class Lerping : MonoBehaviour
{
    /// <summary>The red box will use Lerp to move. We will link
    /// this object in via the inspector.</summary>
    public GameObject lerpObject;
    /// <summary>The starting position for our red box.</summary>
    public Vector3 lerpStart = new Vector3(0, 0, 0);
    /// <summary>The end position for our red box.</summary>
    public Vector3 lerpTarget = new Vector3(5, 0, 0);

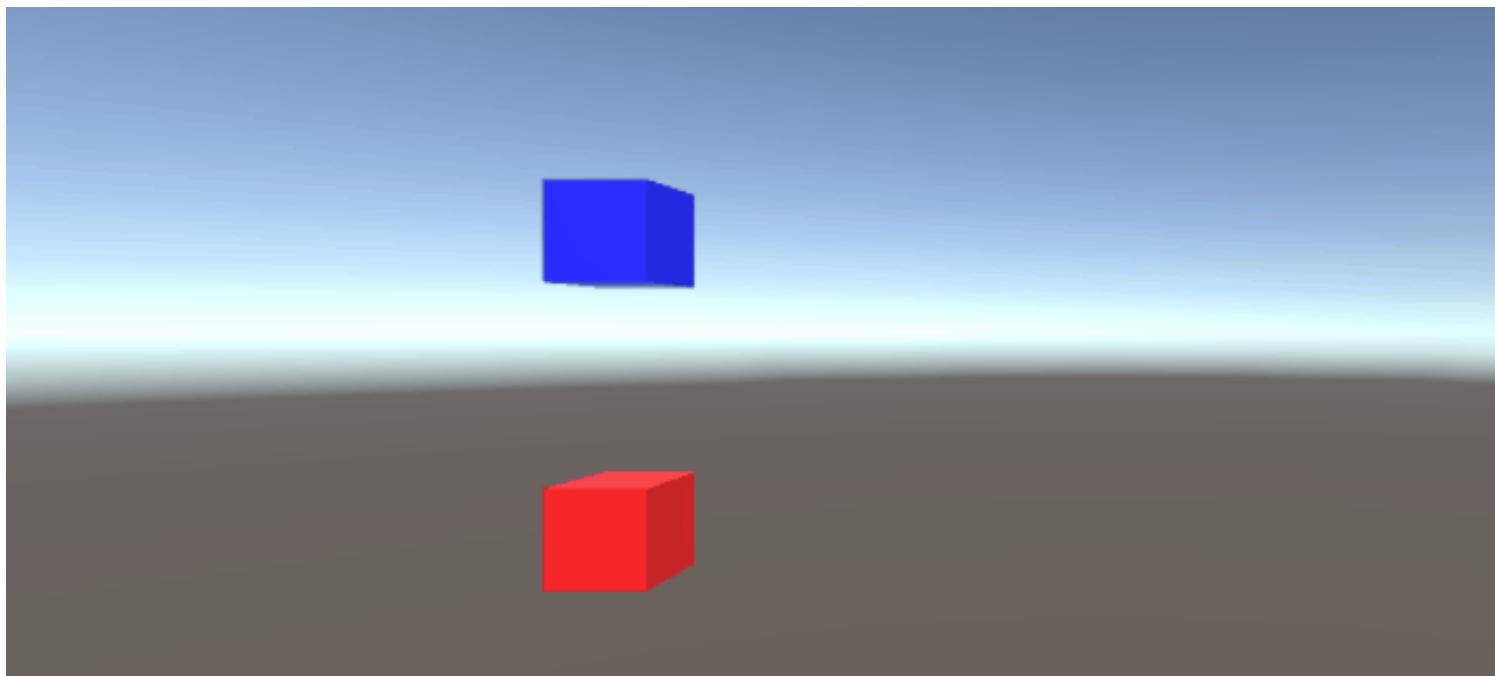
    /// <summary>The blue box will use LerpUnclamped to move. We will
    /// link this object in via the inspector.</summary>
    public GameObject lerpUnclampedObject;
    /// <summary>The starting position for our blue box.</summary>
    public Vector3 lerpUnclampedStart = new Vector3(0, 3, 0);
    /// <summary>The end position for our blue box.</summary>
    public Vector3 lerpUnclampedTarget = new Vector3(5, 3, 0);

    /// <summary>The current fraction to increment our lerp functions by.</summary>
    public float lerpFraction = 0;

    private void Update()
    {
        // First, I increment the lerp fraction.
        // deltaTime * 0.25 should give me a value of +1 every second.
        lerpFraction += (Time.deltaTime * 0.25f);

        // Next, we apply the new lerp values to the target transform position.
        lerpObject.transform.position
            = Vector3.Lerp(lerpStart, lerpTarget, lerpFraction);
        lerpUnclampedObject.transform.position
            = Vector3.LerpUnclamped(lerpUnclampedStart, lerpUnclampedTarget, lerpFraction);
    }
}
```

```
    }  
}
```



MoveTowards

`MoveTowards` se comporte très similaire à `Lerp` ; La différence principale est que nous fournissons une *distance* réelle à déplacer, au lieu d'une *fraction* entre deux points. Il est important de noter que `MoveTowards` ne dépassera pas le `Vector3`.

Tout comme avec `LerpUnclamped` , nous pouvons fournir une valeur de distance négative pour nous éloigner du `Vector3` cible. Dans de tels cas, nous ne `Vector3` jamais le `Vector3` , et le mouvement est donc indéfini. Dans ces cas, nous pouvons traiter la cible `Vector3` comme une "direction opposée"; tant que le `Vector3` pointe dans la même direction, par rapport au `Vector3` , le mouvement négatif doit se comporter normalement.

Le script suivant utilise `MoveTowards` pour déplacer un groupe d'objets vers un ensemble de positions en utilisant une distance lissée.

```
using UnityEngine;  
  
public class MoveTowardsExample : MonoBehaviour  
{  
    /// <summary>The red cube will move up, the blue cube will move down,  
    /// the green cube will move left and the yellow cube will move right.  
    /// These objects will be linked via the inspector.</summary>  
    public GameObject upCube, downCube, leftCube, rightCube;  
    /// <summary>The cubes should move at 1 unit per second.</summary>  
    float speed = 1f;  
  
    void Update()  
    {
```

```

// We determine our distance by applying a deltaTime scale to our speed.
float distance = speed * Time.deltaTime;

// The up cube will move upwards, until it reaches the
//position of (Vector3.up * 2), or (0, 2, 0).
upCube.transform.position
    = Vector3.MoveTowards(upCube.transform.position, (Vector3.up * 2f), distance);

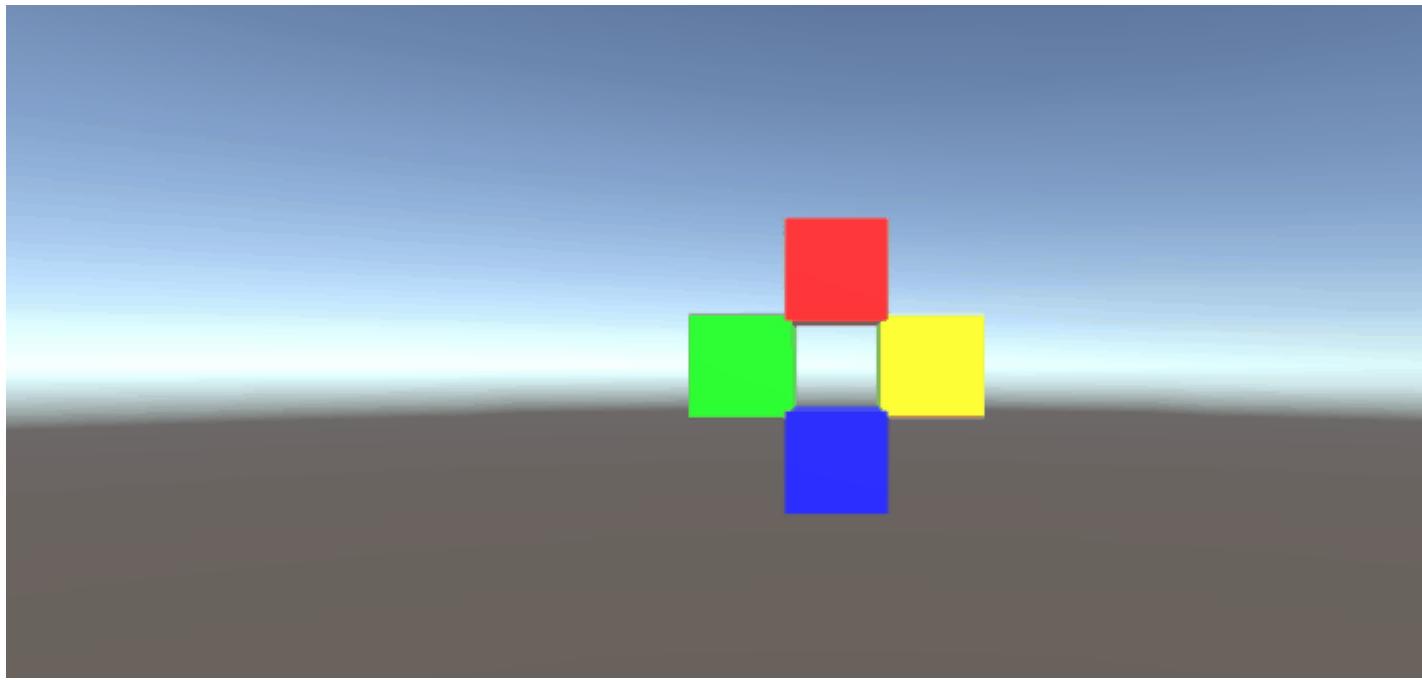
// The down cube will move downwards, as it enforces a negative distance..
downCube.transform.position
    = Vector3.MoveTowards(downCube.transform.position, Vector3.up * 2f, -distance);

// The right cube will move to the right, indefinitely, as it is constantly updating
// its target position with a direction based off the current position.
rightCube.transform.position = Vector3.MoveTowards(rightCube.transform.position,
    rightCube.transform.position + Vector3.right, distance);

// The left cube does not need to account for updating its target position,
// as it is moving away from the target position, and will never reach it.
leftCube.transform.position
    = Vector3.MoveTowards(leftCube.transform.position, Vector3.right, -distance);
}

}

```



[SmoothDamp](#)

Considérez `SmoothDamp` comme une variante de `MoveTowards` avec un lissage intégré. Selon la documentation officielle, cette fonction est la plus utilisée pour effectuer des suivis de caméra en douceur.

En `Vector3` coordonnées `Vector3` départ et cible, nous devons également fournir un `Vector3` pour représenter la vitesse et un `float` représentant le temps *approximatif* nécessaire pour terminer le mouvement. Contrairement aux exemples précédents, nous fournissons la vitesse comme référence, à incrémenter, en interne. Il est important de prendre note de ceci, car la modification

de la vitesse en dehors de la fonction pendant que nous exécutons encore la fonction peut avoir des résultats indésirables.

En plus des variables *requises*, nous pouvons également fournir un `float` pour représenter la vitesse maximale de notre objet et un `float` pour représenter l'intervalle de temps écoulé depuis le précédent appel `SmoothDamp` à l'objet. Nous n'avons pas *besoin* de fournir ces valeurs; par défaut, il n'y aura pas de vitesse maximale et l'intervalle de temps sera interprété comme `Time.deltaTime`. Plus important encore, si vous appelez la fonction un par objet dans une fonction `MonoBehaviour.Update()`, vous *ne devriez pas* avoir à déclarer un intervalle de temps.

```
using UnityEngine;

public class SmoothDampMovement : MonoBehaviour
{
    /// <summary>The red cube will imitate the default SmoothDamp function.
    /// The blue cube will move faster by manipulating the "time gap", while
    /// the green cube will have an enforced maximum speed. Note that these
    /// objects have been linked via the inspector.</summary>
    public GameObject smoothObject, fastSmoothObject, cappedSmoothObject;

    /// <summary>We must instantiate the velocities, externally, so they may
    /// be manipulated from within the function. Note that by making these
    /// vectors public, they will be automatically instantiated as Vector3.Zero
    /// through the inspector. This also allows us to view the velocities,
    /// from the inspector, to observe how they change.</summary>
    public Vector3 regularVelocity, fastVelocity, cappedVelocity;

    /// <summary>Each object should move 10 units along the X-axis.</summary>
    Vector3 regularTarget = new Vector3(10f, 0f);
    Vector3 fastTarget = new Vector3(10f, 1.5f);
    Vector3 cappedTarget = new Vector3(10f, 3f);

    /// <summary>We will give a target time of 5 seconds.</summary>
    float targetTime = 5f;

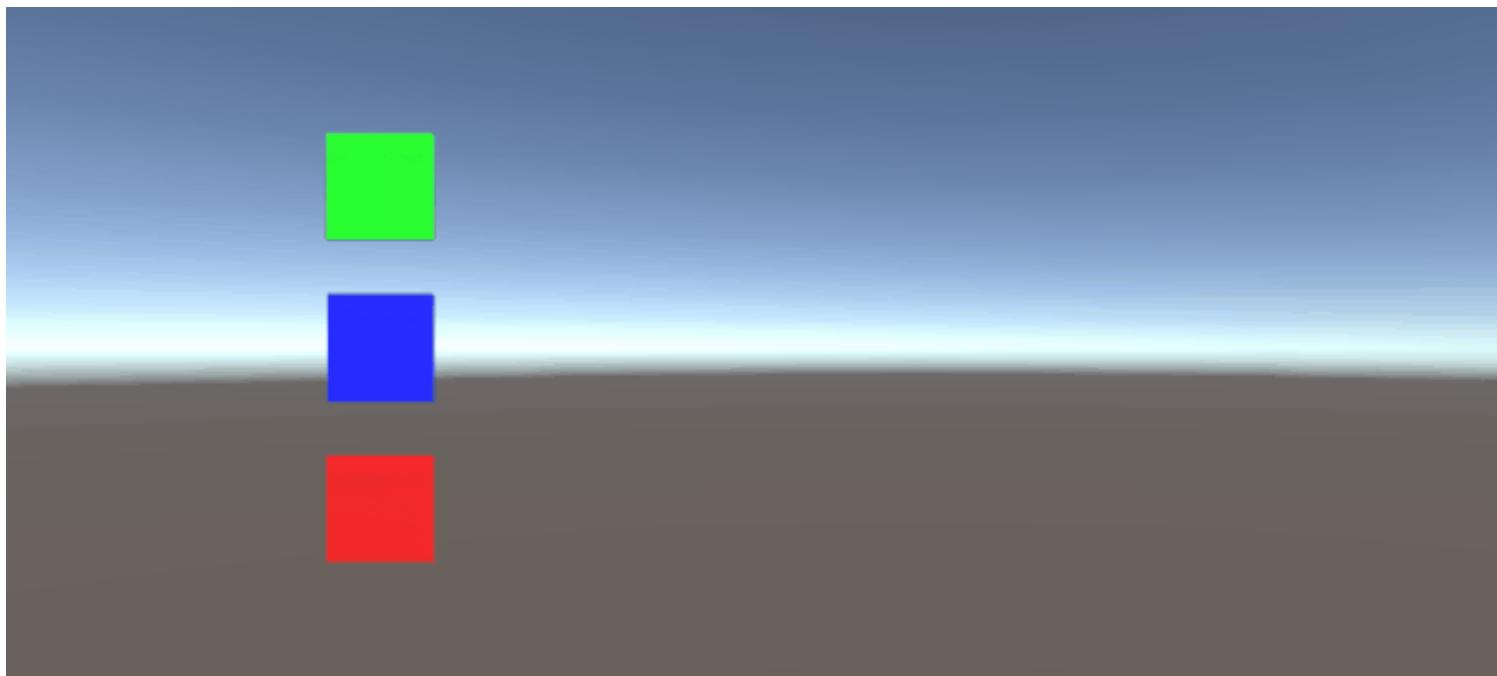
    void Update()
    {
        // The default SmoothDamp function will give us a general smooth movement.
        smoothObject.transform.position = Vector3.SmoothDamp(smoothObject.transform.position,
            regularTarget, ref regularVelocity, targetTime);

        // Note that a "maxSpeed" outside of reasonable limitations should not have any
        // effect, while providing a "deltaTime" of 0 tells the function that no time has
        // passed since the last SmoothDamp call, resulting in no movement, the second time.
        smoothObject.transform.position = Vector3.SmoothDamp(smoothObject.transform.position,
            regularTarget, ref regularVelocity, targetTime, 10f, 0f);

        // Note that "deltaTime" defaults to Time.deltaTime due to an assumption that this
        // function will be called once per update function. We can call the function
        // multiple times during an update function, but the function will assume that enough
        // time has passed to continue the same approximate movement. As a result,
        // this object should reach the target, quicker.
        fastSmoothObject.transform.position = Vector3.SmoothDamp(
            fastSmoothObject.transform.position, fastTarget, ref fastVelocity, targetTime);
        fastSmoothObject.transform.position = Vector3.SmoothDamp(
            fastSmoothObject.transform.position, fastTarget, ref fastVelocity, targetTime);

        // Lastly, note that a "maxSpeed" becomes irrelevant, if the object does not
    }
}
```

```
// realistically reach such speeds. Linear speed can be determined as
// (Distance / Time), but given the simple fact that we start and end slow, we can
// infer that speed will actually be higher, during the middle. As such, we can
// infer that a value of (Distance / Time) or (10/5) will affect the
// function. We will half the "maxSpeed", again, to make it more noticeable.
cappedSmoothObject.transform.position = Vector3.SmoothDamp(
    cappedSmoothObject.transform.position,
    cappedTarget, ref cappedVelocity, targetTime, 1f);
}
}
```



Lire Vector3 en ligne: <https://riptutorial.com/fr/unity3d/topic/7827/vector3>

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec unity3d	Alexey Shimansky, Chris McFarland, Community, Desutoroiya, driconmax, Fjárníng órbíe, James Radvan, josephsw, Linus Juhlin, Luís Fonseca, Maarten Bicknese, martinholder, matiaslauriti, Mike B, Minzkraut, PlanetVaster, R.K123, S. Tarık Çetin, Skyblade, SourabhV, SP., tenpn, tim, user3071284
2	API CullingGroup	volvis
3	Asset Store	JakeD, Trent, zwcloud
4	Collision	Fjárníng órbíe, jjhavokk, Xander Luciano
5	Comment utiliser les packages d'actifs	Fjárníng órbíe
6	Communication avec le serveur	David Martinez, devon t, Fjárníng órbíe, Maxim Kamalov, tim
7	Coroutines	agiro, Fattie, Fehr, Giuseppe De Francesco, Problematic, Skyblade, Thulani Chivandikwa, Thundernerd, ɬolæz əhl qoq, volvis
8	Couches	Arijoon, dreadnought, Light Drake, RamenChef, Skyblade
9	Développement multiplateforme	user3797758, volvis
10	Éclairage Unity	Fjárníng órbíe
11	Extension de l'éditeur	Pierrick Bignet, Skyblade, Thundernerd, ɬolæz əhl qoq, volvis
12	Implémentation de classe MonoBehaviour	matiaslauriti, Skyblade, Thundernerd, user3797758
13	Importateurs et processeurs (post)	gman, Skyblade, volvis
14	Intégration des annonces	bqf the Zéralot

15	La mise en réseau	David Martinez , driconmax , Rafiwui , RamenChef
16	La physique	eunoia , Ḟálmínḡ óm̊bíé , jack jay
17	Les attributs	4444 , Thundernerd
18	Modèles de conception	Ian Newland
19	Mots clés	Arijoon , Augure , glaubergft , Gnemlock , MadJlzz , Skyblade , Trent
20	Optimisation	Ed Marty , EvilTak , Ḟálmínḡ óm̊bíé , Grigory , JohnTube , Skyblade , Thulani Chivandikwa , volvis
21	Plates-formes mobiles	Airwarfare , Skyblade
22	Plugins Android 101 - Une introduction	Venkat at Axiom Studios
23	Préfabriqués	Brandon Mintern , Dávid Florek , Ḟálmínḡ óm̊bíé , gman , Gnemlock , Guglie , James Radvan , Jean Vitor , josephsw , Lich , matiaslauriti , Skyblade , Thulani Chivandikwa , Iolæz æþt qoq , Woltus , yummypasta
24	Quaternions	matiaslauriti , Tiziano Coroneo , Xander Luciano , yummypasta
25	Raycast	driconmax , Meinkraft , Skyblade , user3570542 , volvis , wouterrobot
26	Réalité virtuelle (VR)	4444 , Airwarfare , Guglie , pew. , Pratham Sehgal , tim
27	Recherche et collecte de GameObjects	Pierrick Bignet , S. Tarık Çetin , volvis
28	Regroupement d'objets	Chris McFarland , Ed Marty , lase , matiaslauriti , S. Tarık Çetin , Thulani Chivandikwa , Thundernerd , Iolæz æþt qoq , volvis
29	Ressources	glaubergft , MadJlzz , Skyblade , Venkat at Axiom Studios
30	ScriptableObject	volvis
31	Se transforme	ADB , Jean Vitor , matiaslauriti , S. Tarık Çetin , Skyblade , Thundernerd , Xander Luciano
32	Singltons in Unity	David Darias , Fehr , James Radvan , JohnTube , matiaslauriti , Maxim Kamalov , Simon Heinen , SP. , Tiziano Coroneo , Umair M , volvis , Zze ,

33	Système audio	R4mbi, ɬolæz øħø qoq
34	Système d'interface utilisateur (UI)	Hellium, matiaslauriti, Maxim Kamalov, Programmer, RamenChef, Skyblade, Umair M
35	Système d'interface utilisateur graphique en mode immédiat (IMGUI)	Skyblade, Soaring Code
36	Système de saisie	Programmer, Skyblade, ɬolæz øħø qoq
37	Unité d'animation	4444, Fiery Raccoon, Guglie
38	Unity Profiler	Amitayu Chakraborty, ForceMagic, RamenChef, Skyblade
39	Utilisation du contrôle de source Git avec Unity	Commodore Yournero, Hacky, James Radvan, matiaslauriti, Max Yankov, Maxim Kamalov, Pierrick Bignet, Ricardo Amores, S. Tarık Çetin, S.Richmond, Skyblade, Thulani Chivandikwa, YsenGrimm, yummypasta
40	Vector3	diconmax, Fjárníng ómblíe, Gnemlock