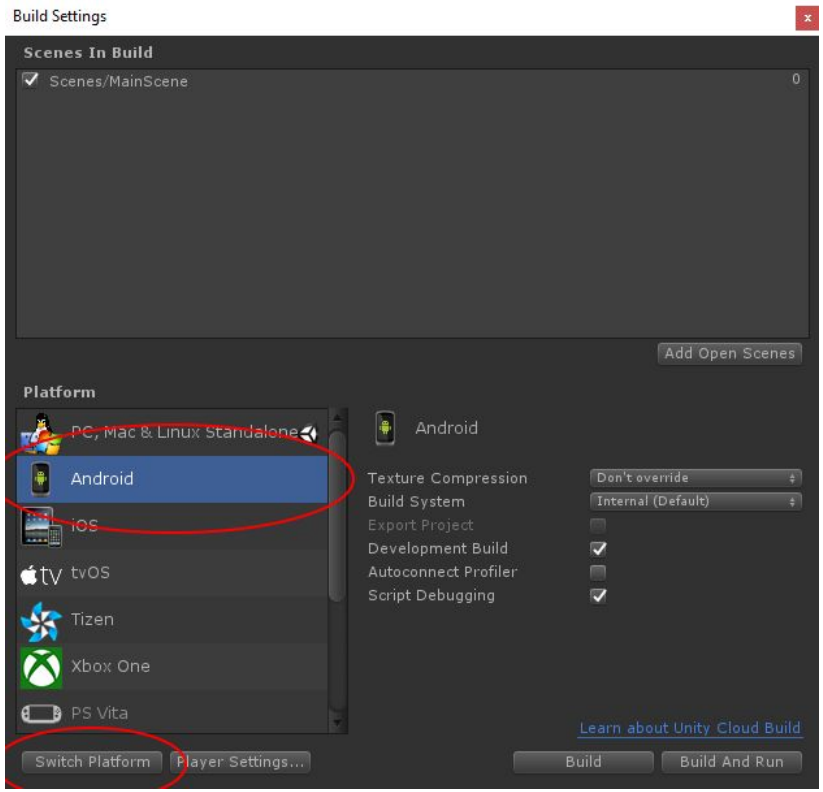
Cours 11

Unity et Android

- Unity est l’engin multiplateforme avec le plus de présence sur Android.
- Il y a d’autres présences, telles que SDL, Unreal, Adobe AIR, IceWave, etc., mais la facilité de développer avec Unity et la gratuité du développement sur Android en ont fait une combinaison gagnante.

- Pour ajouter le support Android à votre projet Unity, il suffit d'aller dans les "Build Settings" (Ctrl+Shift+B) et de choisir la plateforme Android en la sélectionnant et appuyant sur "Switch Platform".



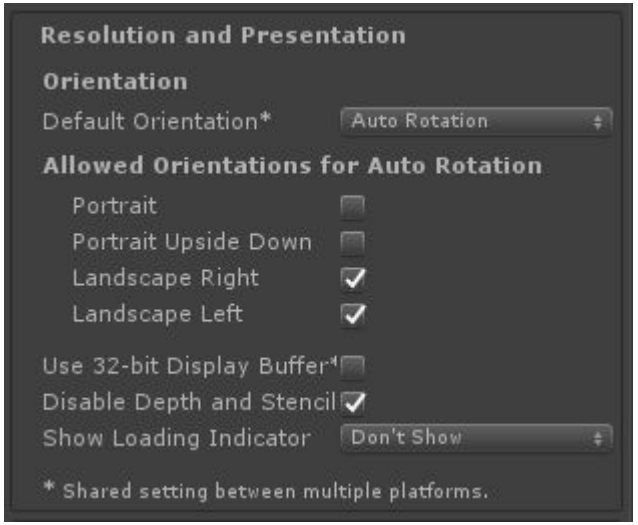
- Vous aurez également besoin de lui indiquer où est situé votre SDK Android dans “Edit - Preferences... - External Tools - Android - SDK”.
- Si vous avez une installation régulière de Android Studio, votre SDK devrait être:
  `%HOME%\AppData\Local\Android\sdk`

- Vous aurez peut-être à spécifier où est votre JDK si vous en avez plusieurs. Vous pouvez utiliser celui de Android Studio:
  `C:\Program Files\Android\Android Studio\jre`

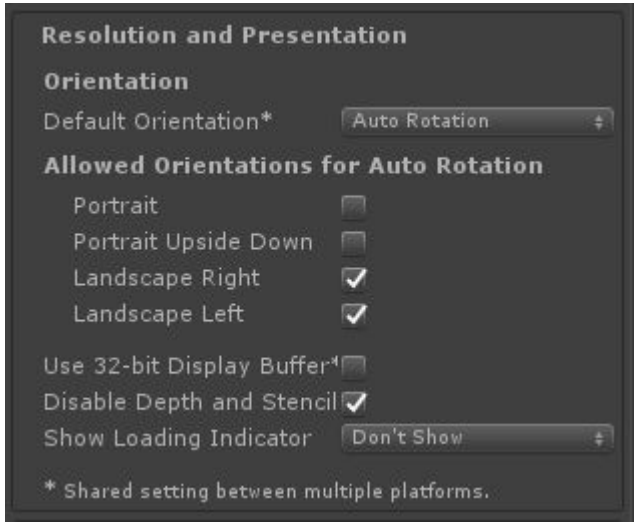
- Les autres champs des “Build Settings” Android sont les suivants:
  - **Texture Compression**: Puisque la mémoire est restreinte sur Android, les textures sont automatiquement compressées en format ETC. Cette option vous permet de changer ceci au risque d’une perte de compatibilité.
  - **Build System & Export Project**: Nous utiliserons uniquement le système interne de Unity, mais il serait possible ici d’utiliser le même système de Android Studio (gradle).
  - **Development Build**: Principalement, ceci affecte l’option **`android:debuggable`** de votre manifeste.
  - **Autoconnect Profiler**: Ouvre et connecte automatiquement le “Profiler” sur un “Build & Run”.
  - **Script Debugging**:Permet le debogage C#.

- Si vous appuyez sur “Player Settings…”, vous aurez également un panneau vous permettant de configurer votre application Android.
- Pour Android, vous aurez 5 volets:
  - Resolution and Presentation
  - Icon
  - Splash Image
  - Other Settings
  - Publishing Settings

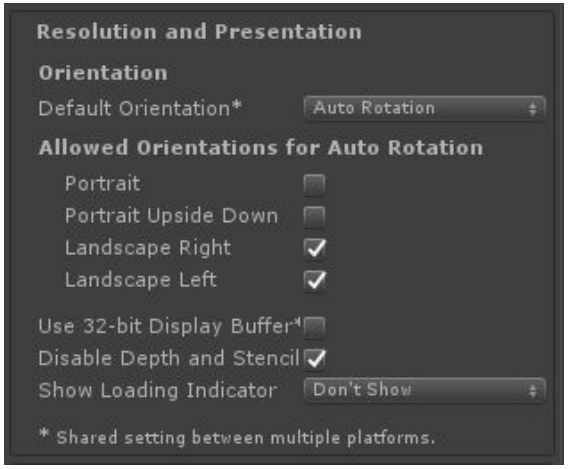
- **Default Orientation**: Normalement, il est préférable de mettre “Auto Rotation” et de restreindre avec les options suivantes. Sauf pour cas particuliers, la seule autre option qui devrait être utile est pour forcer “Portrait”, car on devrait toujours supporter les deux côtés (“Left” et “Right”) pour “Landscape”.



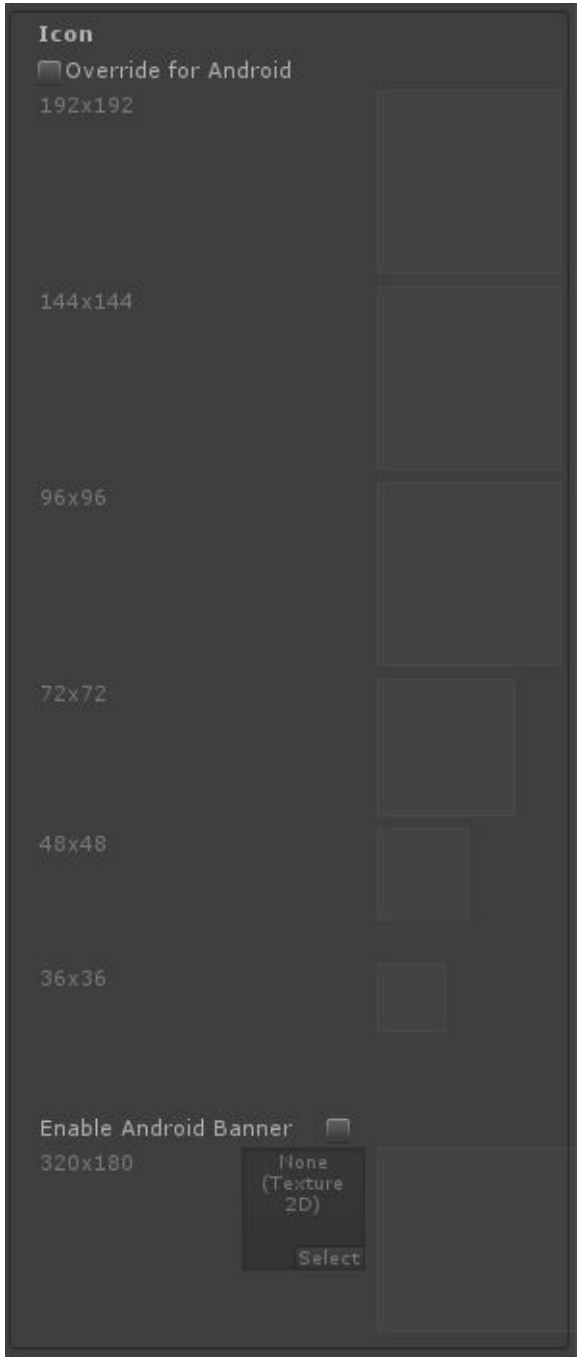
- **Use 32-bit Display Buffer**: Utilisez ceci si vous voulez supporter de l'alpha ou avez des dégradés qui ont des problèmes de bandes. Pour sauver de la mémoire, la valeur par défaut est 16-bit RGB 565.



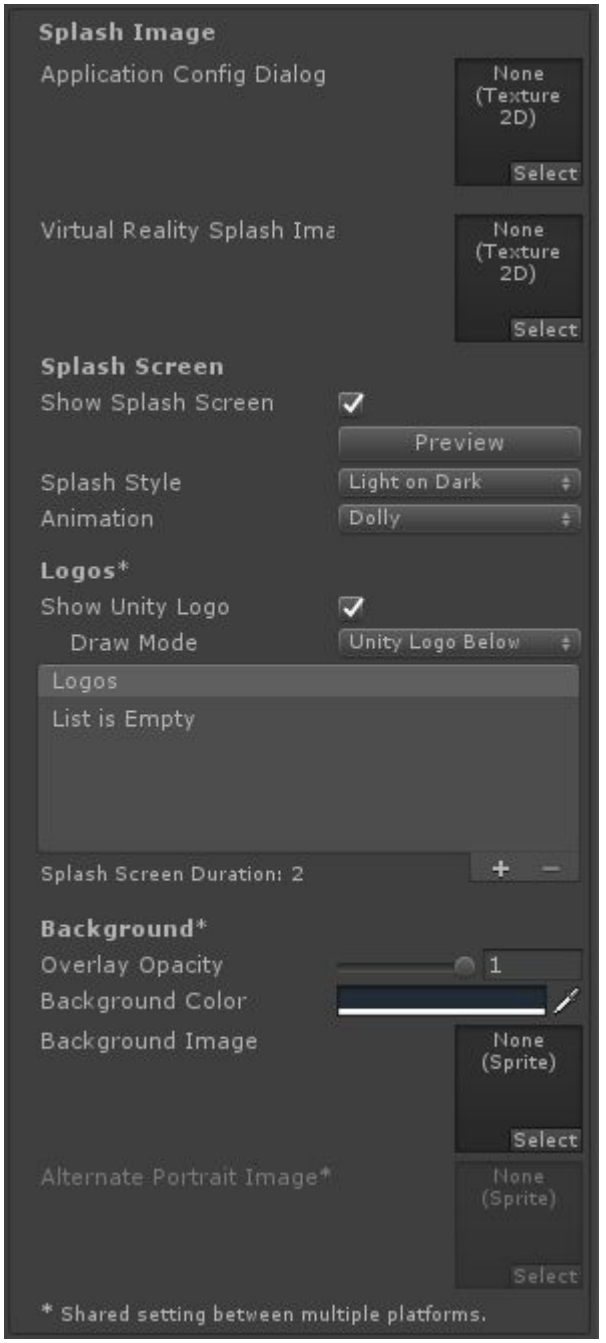
- **Disable Depth and Stencil**: Cette option permet de sauver davantage de mémoire si vous n'utilisez pas la profondeur et le stencil.
- **Show Loading Indicator**: Puisque les temps de chargement sont beaucoup plus significatifs sur Android, il peut être intéressant l'indicateur de Unity. Par contre, les designers UI/UX préfèrent généralement vous le fournir.



- La section “Icon” sert à changer l’icône de lancement de l’application. La valeur par défaut est le logo de Unity. Les tailles 192, 144, 96, 72, 48 et 36 correspondent à **xxxhdpi**, **xxhdpi**, **xhdpi**, **hdpi**, **mdpi**, et **ldpi**.
- **Enable Android Banner**: Ceci est destiné à Android TV qui utilise des “*banners*” au lieu d’icônes.

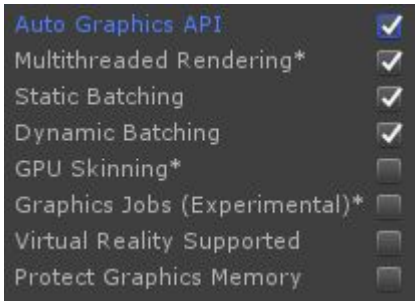


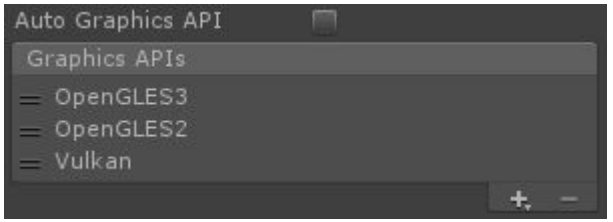
- **Splash Image**: Ceci est une catégorie commune à bien d'autres plateformes. Unity vous permet d'y mettres des images à afficher en succession au lancement de l'application.



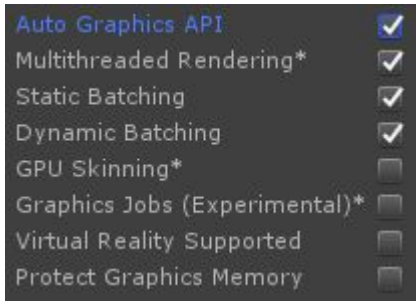
- **Other Settings**: Cette section est la plus importante et détaillée. Elle est aussi celle qui change le plus d'une version Unity à l'autre. Nous allons nous concentrer sur les points le plus importants et spécifiques à Android.

- **Rendering**:
  - **Auto Graphics API**: Unity supporte OpenGLES 2 et 3 sur Android, ainsi que le tout nouveau Vulkan. Si vous gardez l'option automatique activé, Unity détectera au lancement lequel utiliser. Par contre, ceci implique que Unity devra inclure 3 versions de chaque shader, matériel, etc.

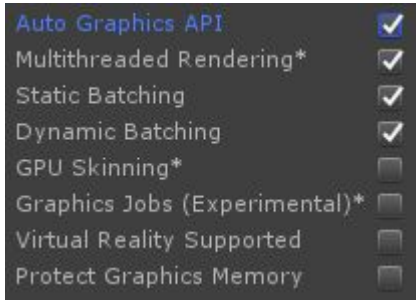




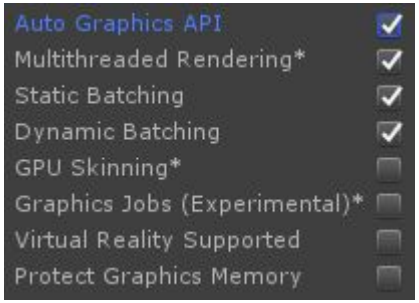
- **Multithreaded Rendering**: En cochant cette option, la majorité du rendu sera fait sur un autre fil d'exécution et pourra être fait en même temps que le "*Behavior Update*" de vos "*game objects*". Sauf s'il vous cause des problèmes, ceci est généralement désiré.



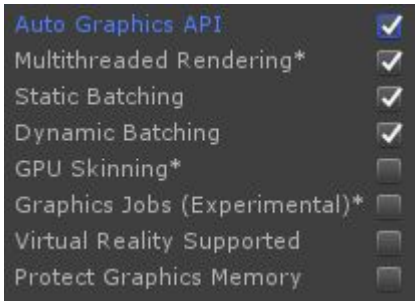
- **Static Batching & Dynamic Batching**: Diminuer le nombre de “*draw calls*” sur Android est un élément clé de la performance. Ceci devrait pratiquement toujours être activé.



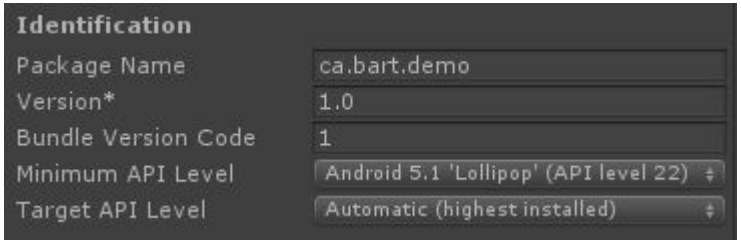
- **GPU Skinning & Virtual Reality Supported**: Uniquement possible à partir de OpenGLES3 et pour les applications VR.

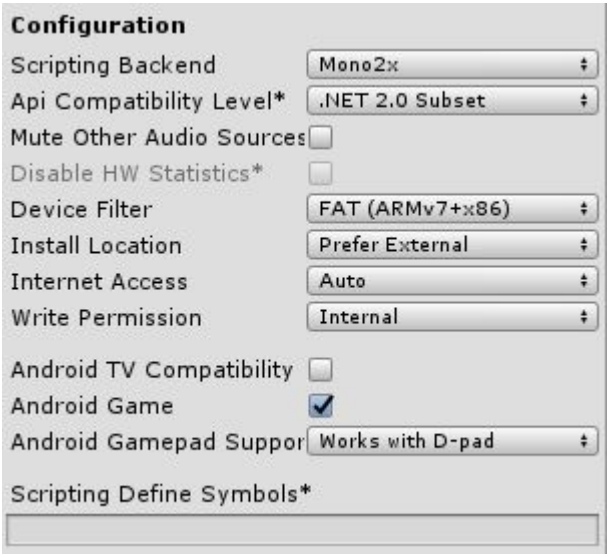


- **Graphics Jobs (Experimental)**: Ceci remplacera ultimement "*Multithreaded Rendering*", mais est uniquement possible avec Vulkan, donc très peu d'appareils peuvent en bénéficier et la fonction est expérimentale.

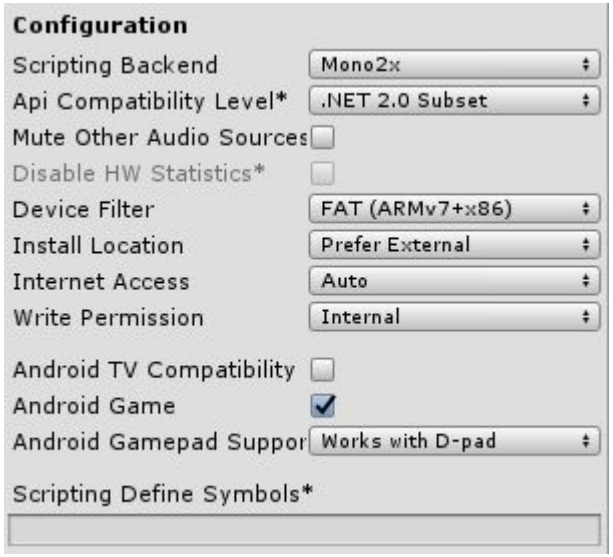


- La section d'identification permet d'assigner des valeurs à quelques champs du manifeste Android:
  - **Package Name** : **`package`** de la balise **`manifest`**
  - **Version*** : **`android:versionName`** de la balise **`manifest`**
  - **Bundle Version Code** : **`android:versionCode`** de la balise **`manifest`**
  - **Minimum API Level** : **`android:minSdkVersion`** de la balise **`uses-sdk`**
  - **Target API Level** : **`android:targetSdkVersion`** de la balise **`uses-sdk`**

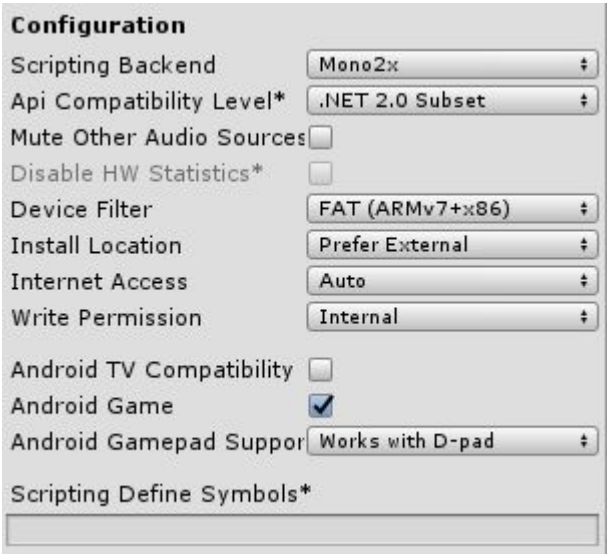




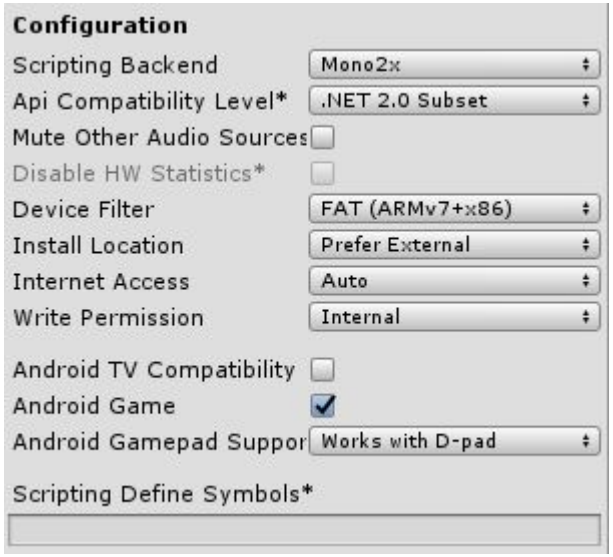
- **Scripting Backend**:
  - **Mono2x** : C# compilé en IL
  - **IL2CPP** : Le code compilé C# se fait transformer en C++ et re-compilé en natif (plus rapide à l'exécution, mais beaucoup plus long à la compilation et difficile au débogage).
- **Api Compatibility Level*** : Choisir entre tout .NET 2.0 ou seulement un sous-ensemble de fonctionnalité. "*.NET 2.0 Subset*" est généralement utilisé sauf si le contraire est nécessaire pour sauver de l'espace.



- **Mute Other Audio Sources** : Empêche d'autres applications de jouer du son pendant que la vôtre est en avant-plan.
- **Disable HW Statistics*** : Désactive les analytiques envoyées à Unity.
- **Device Filter** : Permet de choisir entre ARMv7, x86 ou la combinaison des deux. Ceci double la taille de l'exécutable, mais permet d'installer sur plus d'appareils.

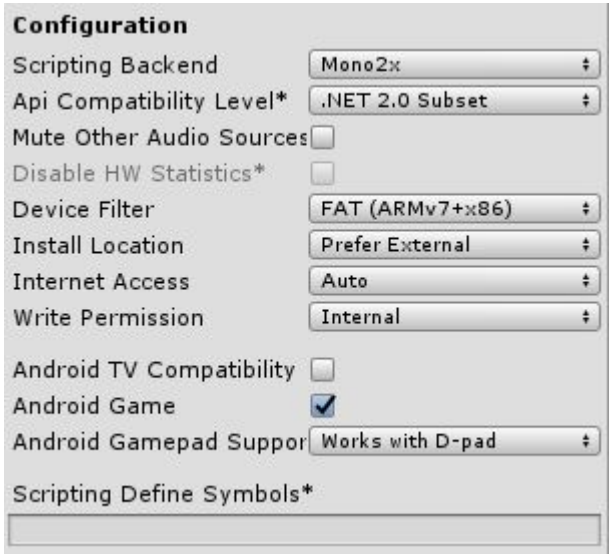


- **Install Location** : ceci correspond à `android:installLocation` de la balise `manifest`
- **Internet Access** : Ajoute la permission `INTERNET` au manifeste automatiquement ou de façon forcée.
- **Write Permission** : Si mis à “**External**”, ceci va ajouter la permission au manifeste et faire en sorte de Unity utilisera le stockage externe pour ses fichiers (cache, etc.)



- **Android TV Compatibility** : Pour les jeux voulant être disponible sur Android TV, mais vous devez supporter la contrôle à distance du Android TV ou un *Gamepad*.
- **Android Game** : `android:isGame` de la balise `application`
- **Android Gamepad Support** : Ceci sert à dire quel type de contrôleur est supporté ou nécessaire sur Android TV.

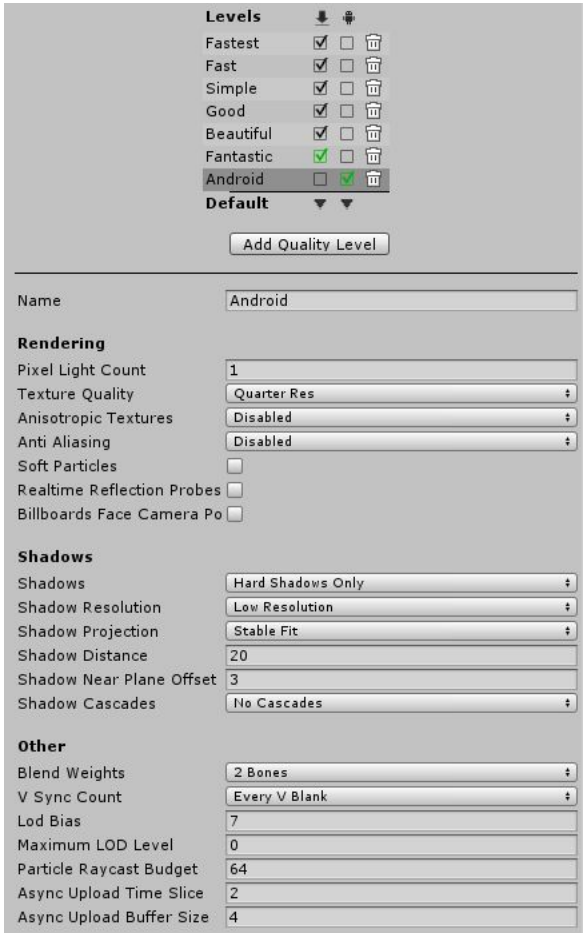
- **Scripting Define Symbols*** : Si vous voulez activer des symboles sur Android. Unity ajoutera automatiquement `UNITY_ANDROID` à la liste pour que vous puissiez avoir du code uniquement utilisé sur Android.



- Les “***Publishing Settings***” permettent de configurer la signature de votre application pour distribution sur Google Play.
- Normalement, le “*keystore*” (trousseau de clés) sera fourni par le “*publisher*” de l’application et il doit obligatoirement demeurer le même pour toutes les versions de l’application.

- Cet écran a également l'option "***Split Application Binary***" puisque Google Play impose une limite de 100 Mo sur les fichiers APK. Si vous activez cette option, Unity ne gardera que les ressources nécessaires pour la première scène dans votre APK, et le reste sera mis dans 1 ou 2 fichiers OBB (d'un maximum de 2 Go chacun). Ces fichiers peuvent être envoyés sur Google Play avec votre APK, permettant d'avoir des jeux jusqu'à environ 4.1 Go. Voir : https://developer.android.com/google/play/expansion-files.html

- Il est également important sur Android de bien choisir le niveau de qualité de Unity. Dans “***Edit > Project Settings > Quality***”, vous pouvez choisir quels niveaux sont permis sur Android. Dans bien des cas, les développeurs préfèrent créer un niveau spécifique à Android pour l’optimiser vers une expérience “*mobile*” plaisante.



- Lorsque vous construisez le projet, vous avez le choix entre “**Build**” et “**Build And Run**”.
- Dans les deux cas, sur Android, il commence par faire votre APK. “**Build And Run**”, par contre, va également le déplacer vers votre appareil (ou simulateur) ***S’IL Y A UNIQUEMENT UN APPAREIL PRÉSENT***. Après l’avoir installé, Unity lancera également un *Intent* pour démarrer votre jeu.

- Vous remarquerez que vous n'avez pas eu à créer un manifeste Android. Unity s'occupe d'en générer un grâce aux valeurs dans ses options et en y ajoutant les entrées trouvées dans son répertoire de *"plugins"*. Si à un moment vous en avez besoin, immédiatement après un *"build"*, vous le trouverez ici:
  `Temp\StagingArea\AndroidManifest.xml`

Cours 11

Débogage Unity et Android

- Les développeurs Unity sur Windows sont habitués de pouvoir essayer leurs jeux directement avec le bouton *Play* de Unity. Ceci est différent sur Android, puisque vous pouvez uniquement utiliser vos *plugins* sur l'appareil. Vous êtes donc obligés de faire "**Build And Run**" et d'essayer votre jeu sur l'appareil.
- Il existe l'application **Unity Remote 5** qui permet d'envoyer l'affichage de l'éditeur vers un appareil, mais celle-ci ne permet pas réellement d'exécuter le code sur l'appareil et d'y attacher l'éditeur, donc a très peu d'utilité.

- Si ce qui vous intéresse est le “*profiler*” de Unity, celui-ci est capable de détecter un appareil connecté en USB et de s’y connecter par ADB sans problème.
- Par contre, si vous voulez déboguer votre code C#, vous allez avoir besoin d’une connexion TCP/IP à votre appareil.
- Dans bien des cas, les réseaux corporatifs isolent les appareils mobiles du reste des ordinateurs, empêchant un lien direct entre le débogueur de Unity et l’appareil.

- ADB nous donne un outil utile pouvant régler tous ces cas. Il est possible de rediriger un port local vers l'appareil par USB, permettant de déboguer votre application Unity même si le réseau n'est pas atteignable.
- Lorsque Unity est lancé sur l'appareil, vous remarquerez une ligne comme celle-ci dans les *logs*:

```
Using monoOptions --debugger-agent=transport=dt_socket,embedding=1,defer=y,address=0.0.0.0:56354
```

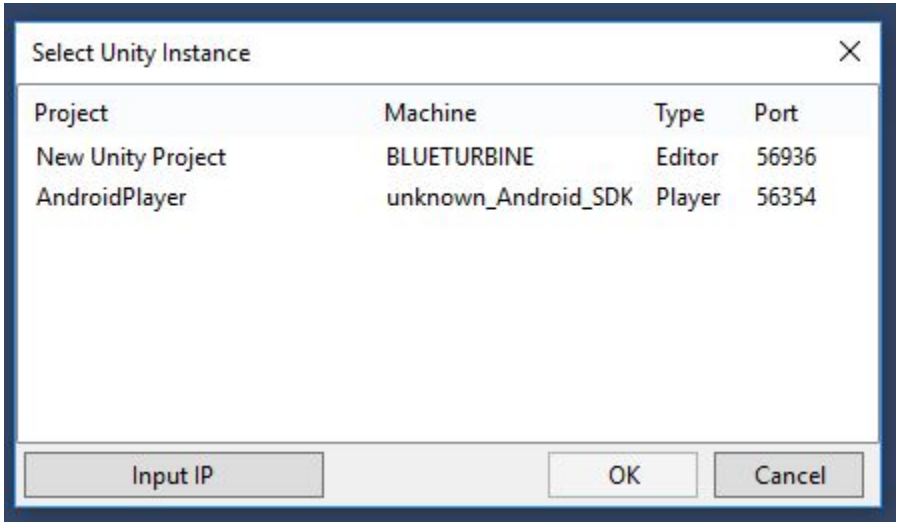
- Ce qui est important c’est de remarquer que Unity a ouvert un port 56xxx pour le débogage. D’autres ports sont également ouverts, mais c’est celui-ci qui servira à connecter Visual Studio à votre appareil.
- Vous pouvez également le trouver à l’aide de la commande suivante dans votre répertoire **`platform-tools`** du SDK:
  `adb shell "netstat -a | grep 0.0.0.0:56"`

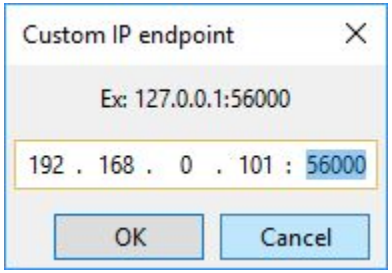
```
D:\Users\fguimond\AppData\Local\Android\sdk\platform-tools>adb shell "netstat -a | grep 0.0.0.0:56"
 tcp       0      0 0.0.0.0:56354           0.0.0.0:*               LISTEN

D:\Users\fguimond\AppData\Local\Android\sdk\platform-tools>
```

- Avec cette information, vous pouvez établir le pont vers votre application avec la commande suivante:
  `adb forward tcp:56000 tcp:56354`

- Le premier nombre devrait toujours être 56000 pour faciliter la connexion dans Visual Studio, et le deuxième devrait être le nombre découvert dans l'étape suivante. Ce port changera à chaque fois que l'application est lancée, alors vous devez refaire la démarche à chaque fois.

- Dans Visual Studio, vous pouvez maintenant faire **Attach Unity Debugger**.
- Vous verrez votre éditeur, et possiblement votre appareil Android si votre réseau le permet ou votre émulateur. Dans le cas de l’ émulateur, vous ne serez probablement pas capable de vous y connecter.





- Par contre, si vous faites “**Input IP**”, Visual Studio mettra automatiquement votre adresse IP et le port 56000 (que vous avez fait un pont USB vers votre appareil par ADB). Vous devriez pouvoir vous y connecter sans problème.
- L’avantage de cette méthode est que vous pouvez l’utiliser avec tous les réseaux et tous les appareils si vous avez un cable USB et les pilotes ADB de l’appareil.

Cours 11

Astuces Unity et Android

- Si votre application a été créée pour utiliser une souris (sans l'utilisation de clique droit ou de combinaison avec d'autres touches du clavier), il y a de fortes chances qu'elle fonctionne "*out-of-the-box*" sur Android. Les composantes 2D de UI de Unity fonctionnent aussi bien avec une souris qu'avec le toucher.
- Le bouton "*back*" de Android est reconnu comme une touche de clavier régulière dans Unity et assigné à la touche échape ("*ESC*").

- L’accéléromètre et le support de contrôleurs *bluetooth* sont gérés automatiquement avec la classe `Input` et `InputManager` de Unity.
- La navigation par contrôle à distance sur Android TV provoque des événements de clavier comme les flèches sur un clavier.
- La classe `UnityPlayerActivity` est utilisée comme activité principale de Android mais elle peut être remplacée par une classe de votre choix. Cette classe devra s’occuper de gérer le `UnityPlayer` elle-même.

- **`UnityPlayerActivity`** met automatiquement le **`UnityPlayer`** sur pause dès qu'un événement **`onPause`** est reçu. Ceci arrêtera tout code C# de s'exécuter dès qu'elle n'a plus le focus (par exemple, si un "*pop-up*" vient partiellement cacher l'écran). Vous ne pouvez donc pas exécuter de code C# dans le **`onStop`**, **`onDestroy`**, **`onRestart`**, **`onStart`**, etc., sans modifier la classe **`UnityPlayerActivity`**.

- La classe `PlayerPrefs` de Unity sauvegarde directement avec la classe `SharedPreferences` de Android. Toutes ses valeurs sont donc visibles à la fois en Java et C#, et sauvegardées dans un simple fichier XML sur l'appareil.
- Il est recommandé de vérifier vos applications sur appareils tout au long du développement. Plusieurs développeurs font l'erreur de reporter "*l'optimisation*" à la fin, mais la réalité est que certaines choses ne pourront pas fonctionner si elles sont implémentées pour des systèmes comme Windows.

Cours 11

Invocation de Java à partir du C#

- Le C# de Unity s'exécute dans sa propre machine virtuelle avec son propre espace de mémoire.
- Les appels au Java doivent se faire par réflexion et les informations passées et retournées sont des copies.
- Ces deux points rendent les appels au Java particulièrement coûteux lorsque comparés aux appels natifs C# et C++.

- Les appels par réflexion ajoutent également d'autres risques de bogues. Lorsque le code est compilé, normalement, le compilateur vérifie les noms des symboles utilisés. Ceci n'est pas le cas par réflexion. Les symboles sont référés par des chaînes de caractères et évalués à l'exécution final de l'application. Il est donc très fréquent d'avoir des problèmes apparaître à cause de fautes de frappes ou d'un changement de nom.

- La communication du C# vers le Java est faite à l'aide de JNI (*Java Native Interface*).
- Unity expose le JNI à l'aide des classes **`AndroidJNI`** et **`AndroidJNIHelper`**.
- L'utilisation directe de JNI peut permettre de mieu optimizer la communication, mais ceci vient avec un niveau de complexité bien supérieur.
- La meilleure optimization est de limiter les appels au Java.

- Généralement, bien des problèmes peuvent se faire régler avec un seul appel au Java à l'aide d'une méthode statique. Ceci permet de le faire avec un seul échange et un minimum d'informations interprétées (donc moins de chances de bogues).
- En JNI traditionnel, vous avez besoin de savoir, au minimum:
  - le nom de la classe avec la méthode;
  - le nom de la méthode à appeler;
  - la signature de la méthode (c-à-d les types de ses paramètres ainsi que le type de sa valeur de retour).

- Unity vous offre des classes pour simplifier ces échanges:
  - **`AndroidJavaClass`** : Représente une classe Java et permet d'accéder tous ses membres statiques.
  - **`AndroidJavaObject`** : Représente un objet Java et permet d'accéder tous ses membres d'instance.
- Ces classes s'occupent d'inférer une partie des informations par introspection plutôt que de vous forcer de tout déclarer.
- Elles s'occupent également de la gestion des références locales et globales qui est souvent problématique.

- Les appels avec ces classes font automatiquement la conversion entre types primitifs et chaînes de caractères.
- Par contre, il est nécessaire de passer par **`AndroidJavaObject`** pour tous les autres types ou de les sérialiser en chaînes de caractères.
- Unity s'occupe de déterminer la signature avec le tableau d'argument reçu, et le type de retour est déterminé par le paramètre passé à la méthode générique.

```
AndroidJavaClass jc = new AndroidJavaClass("com.unity3d.player.UnityPlayer");
// jni.FindClass("com.unity3d.player.UnityPlayer");
AndroidJavaObject jo = jc.GetStatic<AndroidJavaObject>("currentActivity");
// jni.GetStaticFieldID(classID, "Ljava/lang/Object;");
// jni.GetStaticObjectField(classID, fieldID);
// jni.FindClass("java.lang.Object");

Debug.Log(jo.Call<AndroidJavaObject>("getCacheDir").Call<string>("getCanonicalPath"));
// jni.GetMethodID(classID, "getCacheDir", "()Ljava/io/File;"); // or any baseclass thereof!
// jni.CallObjectMethod(objectID, methodID);
// jni.FindClass("java.io.File");
// jni.GetMethodID(classID, "getCanonicalPath", "()Ljava/lang/String;");
// jni.CallObjectMethod(objectID, methodID);
// jni.GetStringUTFChars(javaString);
```

- Pour **`AndroidJavaClass`**:
  - Vous obtenez une référence à une classe Java en faisant **`new AndroidJavaClass(className)`**.

```
AndroidJavaClass jc = new AndroidJavaClass("java.lang.System");
```

  - Faites attention que vous devez utiliser “**`$`**” au lieu de “**`.`**” lorsque vous faites référence à une classe imbriquée: **`android.view.ViewGroup$LayoutParams`**
  - Vous pouvez appeler des méthodes statiques avec **`void CallStatic(methodName, ...params)`** et **`T CallStatic<T>(methodName, ...params)`**.
  - Vous pouvez accéder et modifier des variables statiques avec **`T GetStatic<T>(fieldName)`** et **`void SetStatic<T>(fieldName, value)`**.

- Pour **`AndroidJavaObject`**:
  - Faire **`new AndroidJavaObject(className, ...params)`** est l'équivalent de créer un nouvel objet en appelant son constructeur.
  - Vous pouvez appeler des méthodes d'instance avec **`void Call(methodName, ...params)`** et **`T Call<T>(methodName, ...params)`**.
  - Vous pouvez accéder et modifier des variables d'instance avec **`T Get<T>(fieldName)`** et **`void Set<T>(fieldName, value)`**.
  - Vous ne pouvez pas faire de "*cast*" mais puisque **`AndroidJavaObject`** représente tout ce qui hérite de **`java.lang.Object`** (tout sauf un des 8 types primitifs), ceci n'est jamais nécessaire.

- Chaque instance de **`AndroidJavaClass`** ou **`AndroidJavaObject`** contient une référence globale JNI vers un objet correspondant du côté de Java. Ceci veut dire que si l'objet C# n'est plus utilisé, cette référence sera uniquement libérée après que le *garbage collector* C# soit passé. Seulement après ce moment qu'il sera possible pour le *garbage collector* Java de s'exécuter, qui prend ultimement prendre beaucoup de temps.

- Pour remédier la situation, ces objets ont une méthode **`Dispose()`** qui libère toutes les ressources utilisées par le côté C#. Il est recommandé de toujours l'appeler lorsque vous avez terminé avec un objet.
- Vous pouvez utiliser "**`using`**" pour automatiquement faire ceci, ou bien utiliser un bloc "**`try … finally`**".

```
//Getting the system language safely
void Start () {
    using (AndroidJavaClass cls = new AndroidJavaClass("java.util.Locale")) {
        using(AndroidJavaObject locale = cls.CallStatic<AndroidJavaObject>("getDefault")) {
            Debug.Log("current lang = " + locale.Call<string>("getDisplayLanguage"));

        }
    }
}
```

- … ou bien…

```
void Start () {
    AndroidJavaClass cls = null;
    AndroidJavaObject locale = null;
    try {
        cls = new AndroidJavaClass("java.util.Locale");
        locale = cls.CallStatic<AndroidJavaObject>("getDefault");
        Debug.Log("current lang = " + locale.Call<string>("getDisplayLanguage");
    } finally {
        if (cls != null) cls.Dispose();
        if (locale != null) locale.Dispose();
    }
}
```

- Faites particulièrement attention aux objets temporaires que vous n'avez pas assignés à une variable. Dans le code suivant, deux objets Java seront non-éligibles au *garbage collector* Java tant que celui C# ne s'est pas exécuté.

```
Debug.Log("current lang = " +
    new AndroidJavaClass("java.util.Locale")
        .CallStatic<AndroidJavaObject>("getDefault")
            .Call<string>("getDisplayLanguage"));
```

- Puisque le code C# ne s'exécute pas sur le fil d'exécution principal de Java, il est fréquent d'appeler la méthode **`runOnUiThread`** de l'activité active. Par contre, ceci est difficile à partir du C# puisque tous les appels au Java sont synchrones.
- Unity offre alors la classe **`AndroidJavaRunnable`** qui permet de passer une action C# à se faire exécuter en tant que **`Runnable`** Java.

```
using UnityEngine;
using System.Collections;

public class ExampleClass : MonoBehaviour
{
    // Pass execution context over to the Java UI thread.
    void Start()
    {
        AndroidJavaClass unityPlayer = new AndroidJavaClass("com.unity3d.player.UnityPlayer");
        AndroidJavaObject activity = unityPlayer.GetStatic<AndroidJavaObject>("currentActivity");
        activity.Call("runOnUiThread", new AndroidJavaRunnable(runOnUiThread));
    }

    void runOnUiThread()
    {
        Debug.Log("I'm running on the Java UI thread!");
    }
}
```

- Faites attention avec cette technique puisque la majorité des programmeurs Unity n'utilisent jamais de fils d'exécution autre que celui principal de Unity et ont très peu d'expérience dans un environnement "*multi-thread*". Puisque le code dans le **`AndroidJavaRunnable`** est C# mais ne s'exécute pas sur le fil principal de Unity, il pourrait accidentellement appeler du code C# qui n'est pas destiné pour du "*multi-thread*".