Logique des shaders

Table des Matières Simplifiés :

Qu’est-ce qu’un shader ?

Basé sur la version 6.3 de Unity.

Définition

Les shaders sont des **scripts qui s'exécutent sur le GPU** (Graphics Processing Unit). Ils se codent avec des langages comme l’HLSL ou le WGSL. Sur Unity, on utilise plutôt le HLSL.

Le GPU est spécialisé pour faire les mêmes **calculs simples, mais des centaines de fois en même temps ;** à l’inverse des CPUs, plutôt spécialisé pour faire des calculs complexes et différents, mais peu en parallèle.
Les GPUs proposent des étapes plus ou moins définies pour coder via des shaders.

Du fait des capacités du GPU, les shaders sont plutôt **utilisés pour générer des images, des textures**, certaines simulations ou toutes sortes de résultats nécessitant ces calculs en parallèle.
Dans les jeux vidéo et autres logiciels 3D, on les utilise le plus souvent avec des objets 3D ou 2D pour les afficher à l’écran en temps réel.

Materials

Sur Unity, vous utilisez un shader en combinaison avec un material. Le material est utilisé par le CPU et est **appliqué sur les objets et relie leurs propriétés au shader**. Il contiendra tous leurs paramètres, les propriétés de l'objet sur lequel il est appliqué, les textures, les float et autres à envoyer sur le GPU pour être utilisé par le shader.

Shadergraph

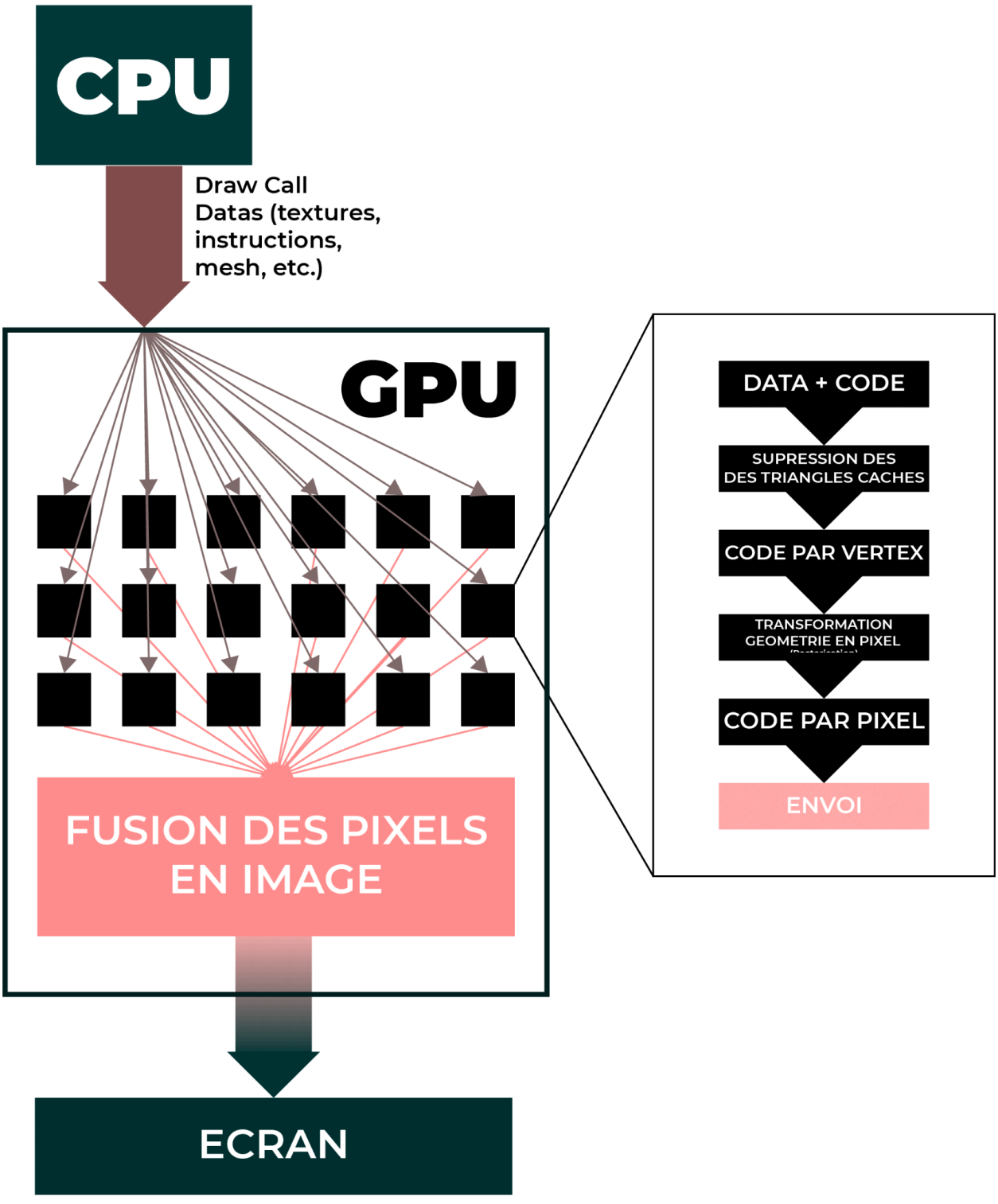
Heureusement pour vous, on peut également créer des shaders avec des **systèmes nodaux** plus visuels. Sur Unity, on a donc **Shadergraph** qui **représente visuellement le code du shader** et ses étapes internes.

Shadergraph utilise deux étapes pour rendre vos pixels : vertex et fragments.
Le calcul créé par l’assemblage de node s’effectue **pour chaque vertex ou chaque pixel**.

C’est le sujet de ce document.

Modèle Mental

Le CPU envoie les données et le code à exécuter au GPU. Le GPU exécute tout son code en parallèles : chaque pixel est exécuté séparément, ainsi des milliers de pixels sont calculés en même temps. Leurs résultats sont ensuite fusionnés pour faire l'image globale.



Vertex / Fragments

Vertex et fragment sont les deux stages accessible et modifiable depuis shadergraph. Ils permettent d'effectuer des opérations sur les vertex et les pixels (fragments) de vos objets.

Dans chaque partie, vous trouverez des paramètres qui changent en fonction du type de shader sélectionné dans graph settings.

Chaque suite de nodes branchés sur les inputs seront exécutées une fois pour chaque vertex si elles sont branchées au stage des vertex ou une fois pour chaque pixel si elles sont branchées au stage des pixels (fragments).

Vertex et fragment ne peuvent pas se partager des nodes : aucune node ne peut être branchée au deux. Pour faire passer des infos entre les deux, vous pouvez voir les interpolateurs.

Paramètres d’un Shader

Les graphs setting sont accessibles dans la fenêtre ouvrable en haut à droite () “Graph Inspector” puis dans l’onglet “Graph Settings”, ils permettent de modifier les paramètres de tout le shader.

Active Targets
Sélectionner les pipelines installés que vous voulez utiliser avec ce shader.

Material
Sélectionne le type de shader.
Cela peut rajouter des outputs dans la partie vertex et fragments du graph.

Surface
Opaque ou transparent. Change l’ordre de rendu des objets : les objets transparents seront rendus après les objets opaques. L’ordre de rendu et une approximation est peu amené à des glitch.

Blend
Apparaît si vous avez sélectionné transparent. Détermine comment est gérée la transparence.

> **Alpha :** transparence classique.
> Si alpha est à 0 ne change pas les couleurs déjà sur l'écran, une valeur de 1 sera rendue opaque. Entre 0 et 1 fait la transition entre les couleurs de l'écran et celle de votre shader.
>
> **Additive :** additionne les couleurs de votre shader au reste de l'écran. Éclaircie les couleurs. Rouge et bleu produiront du magenta, du noir ne va rien changer à l'écran, etc. Légèrement plus rapide qu'Alpha, sur les GPU moderne la différence est négligeable.
> Cette technique était utilisée couramment dans les jeux vidéos ancienne génération pour des effets faux transparence, les effets technologiques, les hologrammes dans mass effect par exemple.
>
> **Premultiply** : quand votre alpha est haut agit comme le mode alpha et quand la transparence est bas, agit comme additif. N’applique pas l’alpha sur la

réflexion et conserve l'éclat de la réflexion. Plus rapide qu'Alpha, sur les GPU moderne la différence est négligeable.

Multiply : multiplie les couleurs de votre objet avec celles de l'écran. Va assombrir les couleurs. N'applique pas l'alpha sur la réflexion et conserve l'éclat de la réflexion. Légèrement plus rapide qu'Alpha, sur les GPU moderne la différence est négligeable.

Preserve Specular Lighting
Option pour les options blend : additif et alpha. permet de choisir si les réflexions sont multipliées ou non par l'alpha (**conserve l'éclat de la réflexion même sur un objet très transparent**). Multiply et Premultiply le font par défaut.

Alpha Clip
Permet de tronquer l'alpha. Le port "Alpha Clip Threshold" va apparaître dans vos outputs. Peut-être appeler cutout mode. Concrètement, les pixels avec une transparence en dessous de la valeur d'alpha clip ne seront pas calculés et ceux au-dessus seront calculés et opaque. Très pratique pour simuler de la transparence sur les hardwares faible (cf. dither). Utiliser souvent pour des Sprites ou pour toutes textures à afficher avec une transparence nette (feuillage, particule, GIF…).

Render Face
Sélectionne les faces qui seront rendues à l'écran.
Si vous rendez sur la face arrière, il se peut que vos normals soient inversées. Peut également causer des erreurs de lighting ou du z-fighting.
Il existe une node dans shadergraph "is front face" pour changer le comportement de votre graph en fonction de quelle face le pixel actuelle est rendu.

Depth Write
Permet de forcer ou non les tests de profondeur sur cet objet. Concrètement, Unity génère pour ces objets une image (buffer) de la profondeur en plus de l'image à l'écran. Par défaut, seulement les objets opaques génèrent une profondeur.

Depth Test
Permet de sélectionner le calcul utilisé pour comparer la profondeur. Chaque option compare la profondeur du pixel que nous sommes en train de calculer avec le pixel déjà sur l'image afin de savoir si le pixel déjà sur l'image est devant ou si on peut calculer notre pixel actuel.

Allow Material Override
Permet d'afficher la plupart des options du Graph Settings directement dans le material pour pouvoir les modifier par material.

Support VFX Graph
Permet de rendre ce shader utilisable par les Visual Effect Graph pour faire des effets à particules. Le shader est à mettre dans les blocs "Output Particle Shader Graph". Dans vos assets, cela vous généra un shadergraph spéciale trouvable en développant la flèche du shadergraph de base dans l'explorateur d'assets de Unity.

Types de Material

Accessible dans *Graph settings > Material* ou *Assets > clic droit > Create > Shader Graph*.

Vous avez plusieurs types de shadergraph. Ils changent le rendu final, le type de node utilisées et les outputs du graph. Ils sont tous créés pour un contexte spécifique.

Blank Shader Graph
shader de base, ne fonctionnera pas tant qu'un type de material est sélectionné dans les Graph Settings.

Subgraph
Permet de créer un **graph utilisable dans d'autres shadergraph. Apparait sous forme d'une node** dans les autres shader. Souvent utilisé pour des calculs répétitifs. Voir ça comme l'équivalent d'une fonction en code.
Vous devez y sélectionner vous-mêmes les valeurs et les types de sorties sur les options de la node output.
Vous pouvez convertir une sélection de nodes en subgraph avec le menu clic droit dans vos shaders.

Sur URP & HDRP :

Lit Shader Graph
Permet de créer des objets qui gèrent la lumière et la réflexion. Les calculs de lighting sont de complexité moyenne. Pour du lighting plus complexe ou plus simple, regarder les materials existant pour chaque pipeline. (Pour URP)

Unlit Shader Graph
Calculs la couleur et l'émission seulement et ne fait aucun calculs de lighting.

Fullscreen Shader Graph
Crée un graph de shader plein écran pour appliquer un material à l'ensemble de l'écran en post-process ou pour un rendu personnalisé. (plus d'info ici)

Canvas Graph
À utiliser avec des éléments UGUI de l'UI des canvas (Image,Raw Image, Button, etc.)

Decal Graph
Permet de faire des decals. Utilisable avec un decal projector pour projeter des textures contre des surfaces (impacte de balle, tâche, logo, etc.)

Six Way Graph
Défini des materials qui réagissent à la lumière provenant de six directions. Ceci est utile pour simuler des effets volumétriques réalistes tels que la fumée, la poussière et les nuages. (pour plus d'info)

Custom render texture
À utiliser avec une custom render texture et à mettre dans ses paramètres, permet de faire des itérations sur la texture, comme laisser des traces derrière soi dans la neige. (cf. Doc)

Sur URP :

Sprite Lit Shader Graph
Permet d'afficher un sprite qui peut être éclairé par des light 2D si votre projet utilise bien un renderer settings sur 2D.

Sprite custom Lit Shader Graph
Permet d'afficher un sprite qui peut être éclairé par un lighting custom.

Sprite Unlit Shader Graph
Pour afficher un sprite sans éclairage.

UI Shader Graph
Permet d'utiliser des shaders sur des éléments du nouveau système d'UI Toolkit. Doit être utilisé avec le "Render Type Branch node" pour sélectionner le type d'objet destiner à être rendu.

Sur HDRP :

Hair
Permet de rendre les cheveux et la fourrure. (cf. Doc)

Eye
Permet de créer des yeux personnalisés basés sur la physique. Il modélise un material à deux couches : la première décrit la cornée et les fluides à sa surface, et la seconde la sclère et l'iris, visibles à travers la première. Il prend en charge divers effets, tels que la réfraction cornéenne, les caustiques, la dilatation pupillaire, l'assombrissement du limbe et la diffusion sous-cutanée. (cf. Doc)

Fabric
Permet de rendre différents types de tissus. Il utilise soit du coton, soit de la soie anisotrope comme base, et prend en charge divers effets supplémentaires tels que la diffusion sous la surface pour créer des tissus d'apparence réaliste. (cf. Doc)

StackLit
Permet de rendre des materials plus complexes que le Lit même déjà complet d'HDRP. Il inclut toutes les fonctionnalités disponibles dans le shader Lit et, dans certains cas, propose des versions plus avancées ou de meilleure qualité. (cf. Doc)

Fog
Le Fog Volume Shadergraph permet de créer un fog volumétrique et des effets environnementaux.

Terrain
Permet de rendre les terrains d'Unity avec ce shader. (cf. Doc)

Water
Simule un milieu aqueux volumétrique et inclut des propriétés pour restituer des effets tels que les caustiques, la diffusion et la réfraction

SubSurface Scattering
Simule la manière dont la lumière interagit avec et pénètre les objets translucides, tels que la peau ou les feuilles des plantes. Lorsque la lumière pénètre la surface d'un material diffusant sous la surface, elle se disperse et se brouille avant de quitter la surface à un point différent.

Anisotropy
Changent d'apparence lorsque vous visualisez le material sous différents angles.

Iridescences
Semble changer progressivement de couleur à mesure que l'angle de vue ou l'angle d'éclairage change. Utilisez ce type de material pour créer des materials tels que des bulles de savon, du métal irisé ou des ailes d'insectes.

Physically Based Sky
Simule une planète sphérique dotée d'une atmosphère dont la densité diminue avec l'altitude. Plus on s'élève, moins l'atmosphère est dense. Le modèle simule une couche d'ozone. (cf. Understand Sky)

Données

Propriété / Variables

Les propriétés sont des **variables** avec une valeur. Elles peuvent être glissées directement en tant que node pour être utilisé. Vous pouvez convertir un node d'un type compatible en propriété et vice-versa, cela permet de donner accès en propriété à des valeurs qui étaient hardcoded dans le shadergraph au paravent par exemple.

Pour créer une propriété, appuyer sur le bouton "*+*" du **Blackboard (**  **).**

Le material permet d'accéder aux propriétés cochées comme "exposed" or "Show in Inspector" depuis l'inspecteur.
Ces propriétés sont sinon modifiables depuis des scripts C#.

Chaque propriété à des options.

Les options de la propriété sont trouvables dans la section "*Node Settings*" de la fenêtre du *"Graph Inspector"* ().

Options en commun

Nom
Le nom affiché de la propriété dans l'inspecteur. Ex : Wave1

Référence
Le "nom" utilisé par les scripts pour récupérer ou modifier cette propriété. Ex : sur la propriété "Wave1" la référence appelable par script sera "_Wave1".

Précision
Cette option vous permet de **choisir la taille de vos valeurs**.
Cela va rendre vos valeurs plus ou moins précises en fonction de l'option choisie.

Je vous conseille de changer la précision de vos shaders, nodes, et propriétés qu'après que vous ayez fini le shader et tous ses tests. Pour être sûr de ne pas perdre du temps à chercher des problèmes créés par une mauvaise précision mis sur une mauvaise valeur.

4 possibilités :

- **Graph Précision :** la propriété prend le mode directement de shadergraph, changeable dans graph settings
- **Inherit** : Prend la précision des nodes précédentes ou sinon du shadergraph.

- **Half & Single :** Ce sont tous les deux des **nombres décimaux**.

  Mais avec une **taille et une qualité différentes** : Single et deux fois plus grand et précis que Half.

  Certain hardware peuvent silencieusement changer l'option que vous choisissez ici.

  Si la précision choisie n'est pas supportée, Unity bascule automatiquement sur une précision compatible.

  Ne testez pas les précisions quand vous êtes toujours en train de tester votre shader, car elles peuvent changer la qualité, faire des estimations ou apporter des bugs.

  Ces floats sont stockés et gérés d'une façon particulière :

  - ils ont une valeur max au-dessus duquel le nombre est considéré infini : si le nombre max est 55 000, 55 001 sera considéré comme infini par le GPU et les calculs faits avec la valeur.
  - ils ont une valeur basse en dessous de laquelle le nombre est considéré comme 0 : si le nombre max est 0.00005, 0.00004 sera considéré comme 0 par le GPU et les calculs faits avec la valeur.
  - Les floats peuvent être positifs ou négatifs dans shadergraph.

  Pour plus de précison.

Scope
Spécifie comment la valeur est partagée entre différents objets :

- **Global :** Permet de modifier la propriété globalement, sur tous les shaders avec cette propriété partagent la même valeur. La valeur de la propriété n'est

pas partagée entre des shaders différents. Modifiable uniquement via un script C#.
- **Per Material** : la propriété devient indépendante entre chaque material.
- **Hybrid Per Instance**: même effet que Per Material sauf si vous utilisez DOTS.

Show In Inspector
Affiche la propriété dans l'inspecteur. Ne fonctionne pas pour les propriétés avec un scope globales, les Gradients et Les Samplers.

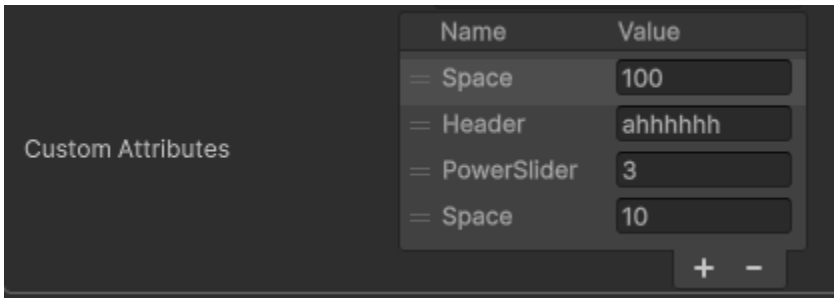
Read Only
Affiche la propriété comme grisés dans l'inspecteur et empêche sa modification depuis l'inspecteur du material.

Default Value
La valeur par défaut que prend la propriété si vous créez un nouveau material.

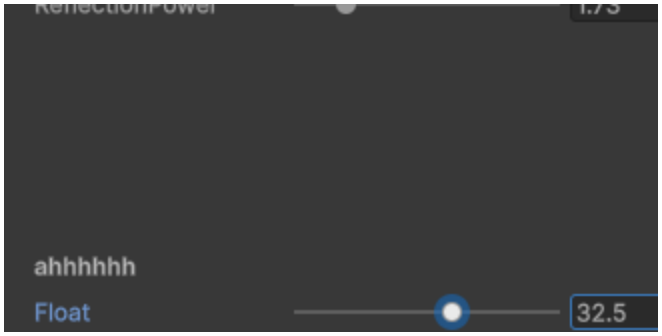
Custom Attributes
Une liste de MaterialPropertyDrawer qui vous permet de mettre en forme vos propriétés dans l'inspecteur.

Il y en a deux types :
- les drawers : qui affecte la façon dont est convertie dans le shader la valeur entrée dans l'inspecteur. Il ne peut y avoir qu'un drawer par propriété.
  Il en existe déjà par défaut : Toggle, ToggleOff, KeywordEnum, Enum, PowerSlider, IntRange. Voir ici pour plus de détails.
- Les decorators : qui affecte la mise en forme autour de la propriété.
  Il en existe déjà par défaut : Space, Header. Voir ici pour plus de détails.

Exemples dans shadergraph :



Dans l'inspecteur :



Types de Propriétés

Float

1 channel, Connecteurs Bleus clairs

Modes
comment la valeur apparaît dans l'inspecteur.

- **Default :** apparaît comme un nombre décimal.

- **Integer** : apparaît comme un nombre entier.

- **Slider** : apparaît comme un curseur coulissable sur une **barre limitée par les valeurs min et max**.
  Default value vient définir la position de base du curseur sur la barre.
  Plusieurs modes pour le slider : **Default** pour des nombres décimaux;
  Power pour que les valeurs suivent une courbe exponentielle au lieu de suivre une courbe linéaire;
  Integer, pour des nombres entiers.

- **Enum** : apparaît comme une liste sélectionnable d'options dans l'inspecteur. Vous nommez les possibilités et les associer à un nombre dans les options de la variable dans le shadergraph. Quand un membre de la liste est sélectionné dans l'inspecteur du material, alors le nombre qui lui est associé est utilisé.

Vector2,3,4

2 channels, Connecteurs vertes
3 channels, Connecteurs jaunes
4 channels, Connecteurs roses

Permet de créer un vecteur avec 2,3,4 valeurs xyzw ou rgba.
Les composants des vecteurs peuvent être appelés rgba ou xyzw.
Souvent utiliser avec des directions, des positions, ou des UVs.

Les vecteurs peuvent être utilisés dans plusieurs contextes différents, comme des positions ou des directions, mais peuvent également contenir des données totalement arbitraires.

On ne mélange jamais des vecteurs de différents espaces sans les transformer dans un même referentiel.

Color

4 channels, Connecteurs roses

Permet d'obtenir une couleur. Dans les shaders les couleurs sont dans un espace linéaire et non pas sRGB et vont de 0 à 1. Linéaire : (0.42,1,0.5) sRGB : (173,187,255).
Les shaders ne font pas la différence entre des couleurs et des vecteurs, ce sont juste des types de données pour eux, de ce fait, color et vector4 sont affichés de la même couleur.

Modes

- **Default** : Une **couleur classique**. À partir de l'inspecteur de materials, il ne peut être défini que sur des **valeurs comprises entre 0 et 1.**
- **HDR** (High Dynamic Range) : Peut-être **en dehors de la plage 0-1**. L'inspecteur le montre en ajoutant un curseur d'intensité à l'interface utilisateur du sélecteur de couleurs. En ce qui concerne la configuration à partir de C#, vous pouvez multiplier la couleur avec un flottant si vous souhaitez un résultat plus lumineux, par exemple. material.SetColor("_Color", Color.red * 5f). Si vous souhaitez conserver le même résultat que l'interface utilisateur, multipliez-le plutôt par 2 ^ Intensité (alias color * Mathf.Pow (2, intensité).

  Cela permet une plage de valeurs beaucoup plus large qui représente plus précisément la façon dont l'œil humain perçoit la couleur et la luminosité. À utiliser **pour l'émission** ou **pour plus de précision** dans vos couleurs, notamment pour les couleurs très sombres ou très claires.

L’HDR est légèrement plus long à calculer et à stocker que les couleurs de base.

Gradient

Connecteurs rouges

Le gradient permet de créer un dégradé que vous pouvez ensuite sample avec une *Sample Gradient Node*.
Il n’est pas affichable dans l’inspecteur, ni changeable depuis les scripts c# le rendant très peu pratique et polyvalent. En plus de cela, vous serez limité par le traitement de couleur et la technique de transitions entre elle choisi par Unity. Faire son propre gradient sur Photoshop ou autre et l’importer sur Unity vous donnera plus de possibilités et de profondeur à vos couleurs.

Textures2D

Connecteurs rouges

C’est une texture. Vous pouvez avoir jusqu'à 128 textures par shaders.
Avec clic droit > “Set as Main Texture” sur la propriété, vous pouvez marquer la texture comme principale. Si vous voulez utiliser des spritesRenderer ou des imageRenderer, vous pouvez appeler la texture “MainTex” pour récupérer directement la texture depuis le component.

Modes
La valeur par défaut que prend la propriété si rien n’est setup dans l’inspecteur du material.

Filter Modes
Quelle technique utilisée pour changer la taille de la texture ?

- **Point** (or Nearest-Point) : La couleur est tirée du pixel le plus proche. Le résultat est en blocs/pixelisé. Pour du pixel art.
- **Linear / Bilinear** : La couleur est faite en faisant la moyenne des couleurs.
- **Trilinear** : Idem, mais utilise également les mipmap levels.

Wrap Modes

- **Repeat** : les valeurs UV en dehors de 0-1 entraîneront la texture en mosaïque/répétition.

- **Clamp** : les valeurs UV en dehors de 0-1 sont limités, provoquant l'étirement des bords de la texture.
- **Mirror** : La texture se répète et se reflète régulièrement.
- **Mirror Once** : La texture est reflétée une fois, puis limite les valeurs UV inférieures à -1 et supérieures à 2.

Textures3D

Connecteurs rouges

3D, il a donc une profondeur ainsi qu'une largeur et une hauteur. À utiliser avec une node sample texture 3D. Il doit donc être échantillonné avec une coordonnée 3D afin de sélectionner le pixel à sortir. Shader Graph l'appelle toujours un UV, bien qu'il soit courant de l'appeler aussi une coordonnée UVW, où W fait référence à la profondeur, comme l'axe Z dans un espace 3D normal.

Textures2D Array

Connecteurs rouges

Similaire à Texture3D, c'est une **collection de plusieurs textures 2D**. Il est aussi à échantillonner avec une coordonnée "3D", bien que les fonctions de la bibliothèque URP Shader les divisent en deux variables : une coordonnée UV 2D et un index sur le node Sample Texture 2D Array, avec les ports Vector2 « UV » et Float « Index ». À utiliser avec la node Sample Texture 2D Array.

Cubemap

Connecteurs rouges

Un Cubemap (aussi appelé TextureCube en HLSL) est un autre type de textures 2D array qui contient 6 tranches, une pour chaque côté d'un cube. Il est échantillonné avec un vecteur 3D qui pointe du centre du cube de texture. La couleur renvoyée est le résultat de l'intersection de ce vecteur. Utilisez souvent pour les skybox, des probes volumes ou des reflection probes.

Matrix

2x2 / 3x3 / 4x4 channels, Connecteurs bleus

Un Matrix4x4 signifie qu'elle comporte 4 lignes et 4 colonnes, soit 16 composants au total. De même, un Matrix3x3 a 9 composants et Matrix2x2 a 4 composants.
Les matrices sont utilisées pour appliquer des transformations pour de la translation (ajustement de la position), rotation et scale. Shader Graph possède cependant quelques matrices utiles déjà intégrées aux nodes et utilisable avec des nodes comme la node Transfrom.

SamplerState

Connecteurs gris

Paramètres génériques pour définir comment une texture est rendu (filtre pour le redimensionnage, mode de répétition, qualité dans des angles serrés)
Par défaut, claque texture définit dans ses options la façon dont elle est rendu, un sampler va override ceci. Vous êtes limité à 16 samplers par shader, mais vous pouvez toujours réutiliser plusieurs samplers pour différentes textures, ce qui rend cette limite difficilement atteignable.

Boolean

Connecteurs violets

À ne pas confondre avec le keyword boolean. Retourne 1 quand cocher et 0 autrement. Peut également être utilisé avec la node Branch pour des conditions, mais la node branch calculant les deux parties vraies et faux du shader peut devenir gourmande en ressource.

Keyword

Les Keywords sont un type spécial qui définit des états (vrai / faux, rouge/vert/bleu/jaune, etc.). 64 max dans un shader.
Chaque état va créer une variante de shader. Les keyword ne **calcules que la parties du graph qui est relié à l'état sélectionné**. Il sont **performant mais rajoute beaucoup de temps de build et prennent de la memoire** (cf. Variant pour plus d'info).

Chaque propriété keyword du blackboard peut être glissé dans le graphique pour l'ajouter en tant que node. La node créée ressemblera à d'autres nodes et aura des

ports d'entrée (On/Off pour le keyword booléen, les inputs nommés pour le keyword Enum). Seuls les types Floats et Vector peuvent être connectés.

Propriété :

- **Promote to final Shader (**subgraph seulement**)** : Rend le keyword disponible dans l'ensemble du shader plutôt qu'uniquement dans le subgraph.
- **Definition** : Définit le type de compilation du keyword, qui détermine la manière dont Unity compile le code du shader. Cela vous permet d'optimiser l'équilibre entre le temps de build, le temps d'exécution et la taille des fichiers.

  Les options sont :

  Shader Feature : Unity compile seulement les variantes de shader pour les combinaisons de keyword utilisées par les materials de votre build et supprime les autres variantes de shader.

  Multi Compile : Unity compile des variantes de shader pour toutes les combinaisons de keyword, que la build utilise ou non ces variantes. Utilisez cette option si vous voulez switcher entre différents mots-clés en jeu avec C#.

  Predefined : Permet d'utiliser des keywords déjà existant, définie par vous plus tôt ou Unity.

  Dynamic Branch : Aucune variantes n'est créée tout le code reste dans le même shader (similaire à la node branch + boolean).

- **Is Overridable :** indique comment le keyword peut être accédé via script.
- **Generate Material Property** : Affiche le keyword dans l'inspecteur.
- **Stages :** Restreint les stages de rendu auquel le keyword est accessible.

  Options : All, Vertex ou Fragment.

Types de Keywords

Boolean : deux conditions : vrai ou faux.

Enum :

Un énumerateur affiche un drop down dans l'inspecteur et permet de sélection l'une de ses options. Le shader utilisera seulement les nodes connectés à la sélection de l'inspecteur.

Material Property
Permet d'agir en fonction de la qualité actuelle du projet set pour les materials.

Dropdown

Spécifique à subgraph, permet de créer un menu déroulant pour sélectionner des options voulu sur la node du subgraph. Glissable comme une node keyword.

Nodes : Définition

Une node est une représentation d'un calcul fait dans le shader. Elles peuvent être reliées entre elles pour faire une chaîne de calculs. Espace est la touche par défaut pour accéder au menu de création de node dans shadergraph.

Note : Les nodes ne sont calculées que si elles sont utilisées.

Ports
Les ports sont les entrées et sorties des valeurs dans les nodes.

Promotion / Tronquage
Les vecteurs et les matrices peuvent être promus et tronqués en fonction des ports. Pour convenir à la valeur attendue d'un port.
Le troncage enlève les canaux en trop.
La "promotion" remplie avec ces valeurs par défauts : (0,0,0,1) sauf pour le float qui remplit chaque nouveau channel avec sa propre valeur.
Ex : un Vector2 -> (3,2) multiplié par des vector4 -> (0,1,1,1) sera promu en tant que vector4 -> (3,2,0,1). Un Float -> (2) mis dans un port vector4 deviendra un vector4 -> (2,2,2,2).

Les ports d'une node ont un type qui leur est généralement associé :

- un seul type de variables existant ;
- accepte tous les types (dynamic). La plupart du temps, quand un port dynamique est connecté avec un certain type, tous les autres ports dynamiques sur la node prennent ce type.
- accepte n'importe quel type de vecteur. Tronque tous les vecteurs au vecteur le plus petit.
- accepte n'importe quel type de matrices ;
- Spéciale. Règle spéciale sur certaines nodes.

Propriétés

Vous pouvez glisser vos propriétés pour les utiliser en tant que nodes et les reliés au reste des nodes.

Node : Input de Données

C’est nodes permettent de récupérer des informations de la scène ou de l’objet. Elles sont exportées automatiquement par les renderers.

Vertex Color
Contient la couleur de chaque vertex de votre objet. Chaque mesh contient cette donnée qui peut être utilisé pour y mettre n’importe quelle data qui rentre dans un vector3 RGB. On y met souvent une texture, on peut y mettre des informations particulières à chaque vertex, etc. Unity l’utilise parfois pour les components qui affichent des images (sprite, image,raw image, etc.) pour vous permettre de récupérer la couleur utilisée pour teindre (tint) votre objet dans les paramètres du component.

UV
Coordonnées 2D (x, y) servant à échantillonner les textures (récupérer les pixels en fonction des coordonnées). Plage standard [0–1]. Modifier les UV (tiling, offset, distorsion) affecte directement le sampling et vous permet de modifier comment s’affiche votre texture. Les UV hors plage (hors 0-1) dépendent du Wrap Mode, de la texture. Distordre les UV avec du noise peut provoquer du stretching.

Sur Unity, les coordonnées des UV vont de (0,0) en bas à gauche à (1,1) en haut à droite.

Les UVs sont souvent offsets avec Add ou substract et peuvent être scale avec Multiply ou Divide. La node Tiling And Offset fait le cumule des deux. Ils existent d’autre nodes qui peuvent modifier vos UVs retrouvable dans la catégorie UV de votre shadergraph favori.

Screen position
Position du pixel en coordonnées d’écran. Fréquemment utilisée pour des effets fullscreen à l’écran (dissolve, réfraction, vignette). Attention aux différences de projection et au rendu transparent : les valeurs peuvent varier selon l’ordre de rendu.

Normal
Direction de la face actuelle. À normaliser après toute opération mathématique, sinon le lighting devient incorrect. Une normale mal normalisée, inversé ou utilisée dans un mauvais espace provoque des erreurs visibles : éclairage inversé, artefacts, reflets instables.

Position
Retourne la position courante (vertex ou fragment) dans l'espace choisi. En World/View space, les grandes valeurs nécessitent une précision élevée (Single). Objet : Coordonnées locales au maillage. (0,0,0) est l'origine/le pivot du maillage. L'axe suit le gizmo « Local ».

Chaque espace définie comment les positions sont gérées et par rapport à quoi.

> **Absolute World** : Coordonnées basées sur le centre de la scène (0,0,0). Des valeurs similaires sont utilisées pour positionner les GameObjects/Transforms lorsqu'ils n'ont pas de parent.
>
> **World** : Dans URP, identique à Absolute World.
> Dans HDRP, par défaut, l'origine (0,0,0) est définie selon la position dans le monde de la caméra.
>
> **View** : Cette espace est également par rapport à la caméra, mais l'axe Z négatif est orienté selon l'avant de la caméra. Si le Z de notre pixel/vertex est positif, il se trouve derrière la caméra en dehors de la vue. L'axe Z, inversé par un node Negate, donne la distance à la caméra.
>
> **Tangent** : cet espace est relatif à la surface du mesh.
>
> Cet espace est construit à partir des vecteurs tangent et normal du mesh. Un vecteur bitangent/binormal est créé perpendiculairement au deux.
>
> L'axe X représente le vecteur tangent, l'axe Y le vecteur bitangent et l'axe Z le vecteur normal. Le vecteur tangent suit l'axe horizontal des UV, tandis que le vecteur bitangent suit l'axe vertical.
>
> L'espace tangent est couramment utilisé pour le mappage normal/bump maping. Ces textures définissent leurs vecteurs dans l'espace tangent, qui sont ensuite convertis dans l'espace monde pour les calculs d'éclairage et d'ombrage.

View direction
Direction du pixel vers la caméra. Utile pour des effets dépendants de l'angle de vue (faux reflets, distorsion, paillettes, etc.). Sensible aux changements de projection et de FOV.

Camera
Permet d'accéder aux propriétés de la camera

Position : Position de la caméra en World space. Sert aux calculs de distance, de direction, de la camera par rapport à l'objet. Coûte peu, mais implique souvent des opérations vectorielles supplémentaires.
Direction : donne la direction vers laquelle pointe la caméra dans la scène. À ne pas confondre avec view direction.

Object

Donne accès aux propriétés spatial de l'objet.

Position : la position de l'objet en world dans la scène.
Scale : donne la scale globale de l'objet.
World bound Min/Max : donne la position de la face de la Bounding Box, la plus près / loin, de l'objet.
Bound Size : donne la taille de la Bounding Box.

Time

Permet d'accéder aux informations de temps d'Unity. Permet de faire des opérations time sensitive comme les animations ou les transitions.

Constant

Donnes des constants classiques d'utilisation.
Coute extrêmement peu au GPU.

PI = π → Utile pour trigonométrie, oscillations, rotations.

TAU = 2 x π → deux fois pi est fréquemment utilisé.

PHI = φ (nombre d'or), → défini un ratio régulier et vu comme parfait, agréable à l'œil.

e = 2.718282 (nombres exponentiels)(Ex : 10 x e^5) → utilisé pour faire des fonctions, exponentielle ou stagnante.

SQRT2 = $\sqrt{2}$ → utiliser pour la géométrie ou la trigonométrie.

Scene depth

Donne la profondeur de la scène vu depuis la caméra.
Utilisée pour des effets de depth, fade, intersection, soft particles, brouillard.
Fonctionne uniquement si la depth texture est activée et disponible sur la plateforme.

Scene color

Couleur de la scène déjà rendue de la caméra. Sert aux effets de distorsion, réfraction ou post-process-like. Coûteux et dépend fortement du pipeline et de l'ordre de rendu.

Math de Base

Node : Calculs

Multiply / Divide
Opérations de base. Multiply est omniprésent (masques, intensités). Divide est plus coûteux et peut produire des infinis si le diviseur est proche de zéro. Sur des coordonnées ou des directions multiply et divide vont scale l'échelle spatiale. Ex : UV multiplié par 2 va doubler les coordonnées X et Y et rendre plus d'images, 4; x4, il rendra 8 images, etc.

Add / Substract
Addition et soustraction. Utilisées pour offsets, accumulations.
Sur un système de cordonnée va décaler les coordonnées dans un sens ou l'autre.
Additionné avec Time, il permet de décaler les cordonnées avec le temps.
Ex : UV.x + time va décaler la texture vers la droite à l'infinie.
Attention aux dépassements de plage sur les couleurs ou l'alpha (en dehors (1,1,1,1).

Tiling And Offset

Cumule les effets de multiply puis de add. Utiliser surtout avec les UVs.

Maximum

Donne la plus grande des deux valeurs.

Minimum

Donne la plus petite des deux valeurs.

Split
Sépare les canaux d'un vecteur. Gratuit conceptuellement, mais améliore la lisibilité du graph.

Combine, Vector 2,3,4
Assemble plusieurs float en vecteur. Attention à l'ordre des canaux. Pratique et lisible.

Float, Color, Gradient, Etc
Nodes de données constantes. Peu coûteux.

Swizzle
Réorganise ou duplique les canaux d'un vecteur. Très utile pour éviter des Split/Combine inutiles.

Clamp
Limite une valeur entre Min et Max. Sécurise les calculs avec des erreurs ou des infinies, mais peut cacher ainsi des erreurs logiques.

saturate
Clamp optimisé entre 0 et 1. Utilise pour limiter vos valeurs comme l'alpha, la smoothness ou autre. Ne pas hésiter à en mettre pour empêcher des valeurs trop fortes d'arriver dans les outputs [0-1] du graph ce qui emmènerait à des comportements imprévisibles. Moins coûteux que Clamp générique. Sécurise les calculs avec des erreurs ou des infinies, mais peut cacher ainsi des erreurs logiques.

Lerp
Permet de **transitionner entre des valeurs A et B selon T**.
T à 0 donne 100% de A et 0% de B,
T = 0.5 → 50% A + 50% B
T = 1 → 0% A et 100% B.
Souvent utilisé avec des masques noir et blanc. Très dépendant de la précision des floats. TRES TRES UTILISE. Vidéo explicative en anglais ici.

Inverse Lerp
Convertit une valeur dans un intervalle donné vers un range [0–1]. Idéal pour distances ou hauteurs. Sensible aux valeurs min/max inversées.
Ex : si je veux 1 quand objet à 200m.
Et 0 quant à 5 m.
InverseLerp(5 , 200 , distance de l'objet réel) on obtient les valeurs voulues 0 et 1 pour 5 m à 200m.
Ne pas oublier de limiter la valeur de sortie si la valeur T peut dépasser l'étendu A-B.

Step
Comparaison binaire. Retourne 0 ou 1. Génère des transitions dures, aliasing (pixellisation des limite net). Permet de comparer deux valeurs. Si la valeur d'entrée est supérieure ou égale à la valeur d'entrée, la fonction renvoie 1 ; sinon, elle renvoie 0.
Utilisable ainsi en tant que condition if.

Smoothstep
Version adoucie de Step. Transition progressive. Légèrement plus coûteux, mais visuellement plus stable.

One minus
Retourne 1 – X. Classique pour inverser un masque. Attention si X sort de [0–1].

Negate
Inverse le signe. Équivalent à multiplier par -1. Utilisé pour inverser des directions.

Normalize
Force un vecteur à une longueur de 1. Indispensable pour des calculs de normales et de directions. Permet de standardiser des directions en les mettant à longueur 1 et ainsi les rendre comparables entre elles. Coûteux si sur-utilisé.

Fraction
Retourne seulement la partie décimale (0.5 → 0.5, 1 → 0, 1.2 → 0.2, etc.). Découpe en répétitions de 0 à 1. Peut provoquer des discontinuités visibles.
Utile pour visionner le tiling d'UV ou visionner des grandes valeurs.

Absolute
Supprime le signe. Les valeurs négatives deviennent positives. Utile pour symétrie ou distances. Perd l'information directionnelle.
Ex : Absolute(-1) = Absolute(1) = 1.

Remap
Convertit une plage de valeurs vers une autre. Très utilisé pour normaliser des données. Attention aux divisions par zéro internes si la plage source est nulle.

Masques

Qu’est-ce qu’un mask dans Shader Graph ?

Un mask est une information en niveaux de gris :

- Noir (0) → l’effet ne s’applique pas
- Blanc (1) → l’effet s’applique à 100 %
- Gris → mélange progressif

Dans Shader Graph, un mask est presque toujours, un canal R/G/B/A d’une texture ou une valeur calculée (noise, height, fresnel, etc.)

Le cas le plus courant : texture “Mask Map”

En workflow standard (URP / HDRP), on utilise souvent une seule texture qui contient plusieurs masques appelée Mask Map.

Canal	Utilisation classique
R	Metallic
G	Occlusion
B	Detail / Height / unused
A	Smoothness

Avantage : une seule texture compacte et économe en mémoire et une seule texture à calculer au lieu de 4 !

Dans Shader Graph : Sample Texture 2D > Split > Chaque sortie (R, G, B, A) devient un mask.

Utilisation typique d’un mask

Le node clé du workflow :

Lerp (Linear Interpolate)

→ Result = Lerp(A, B, Mask)

- A → valeur sans effet (0)
- B → valeur avec effet (1)
- Mask (T) → contrôle du mélange

Exemple concret

➡ Peindre de la rouille sur un métal :
- A = métal propre
- B = métal rouillé
- Mask = texture noire/blanc peinte

➡ Varier une texture :
- A = texture de sol
- B = texture d'herbe
- Mask = texture noire/blanc de patch aléatoire fait à la main

Mais du coup Lerp de Lerp permet de cumuler et de masqué successivement.

Ajuster un mask

Les masks bruts sont rarement utilisés tels quels.

Des nodes permettent de corriger leurs valeurs avant de les branchés dans un lerp :

One Minus → inverser le mask ;

Multiply → intensité du mask ;

Power → durcir ou adoucir la transition ;

Smoothstep → transitions propres ;

Saturate → sécuriser entre 0 et 1.

Exemple : Mask → Power(2) → Multiply(Strength) → Saturate -> Lerp

Masques procéduraux (sans texture)

Un mask peut être calculé :

Sources courantes :

- Noise / Simple Noise
- Gradient
- Fresnel
- Position (Object / World)
- Normal direction (Dot Product)

Exemple :

- Fresnel → mask pour un effet de bord
- Noise → salissures aléatoires, mélanger deux textures pour de la variation
- World Y Position → neige sur le dessus

Un mask est rarement branché directement sur l'input (sauf Alpha), mais utilisé pour mélanger des valeurs et surtout des couleurs avec lerp.

Bonnes pratiques

- Toujours garder les masks en Linear (pas sRGB)
- Réutiliser un même mask pour plusieurs propriétés
- Grouper les paramètres (Strength, Contrast)
- Visualiser le mask seul (Preview ou branché sur Base Color)
- Optimiser : 1 texture > 4 textures

Logique Continue

Il n'y a pas de boucle, pas de if/else au sens traditionnel, pas d'état conservé entre deux pixels.
Chaque pixel (ou vertex) exécute exactement la même suite d'opérations mathématiques, en parallèle, sans savoir ce que font les autres.

La logique dans un shader est donc mathématique.

Au lieu de dire :
si A alors B, sinon C

on calcule A, B et C, puis on choisit lequel est visible via des opérations continues (0 → 1).

Les booléens n'existent pas vraiment

Dans Shader Graph, une "condition" est presque toujours représentée par un float :

0 → faux

1 → vrai

toute valeur entre 0 et 1 → transition partielle

On garde par convention est praticité nos valeurs entre 0 et 1 la plupart du temps.

C'est une différence fondamentale avec le CPU :
Une condition peut être progressive, floue, ou physiquement interprétable.

Condition ET (AND)

→ Multiply

Logique classique : A ET B pour valider la condition

Équivalent shader : **A * B (multiply)**

Pourquoi ça marche :

si A = 0 → résultat = 0;
si B = 0 → résultat = 0;

si A = 1 et B = 1 → résultat = 1.

Et surtout :

Si A = 0.5 et B = 0.5 → résultat = 0.25.

Ce comportement est extrêmement utile :

- masques combinés
- empilement de conditions
- atténuation progressive

Exemples d'usage :

- un effet visible uniquement dans une zone et selon l'angle de vue.
- un dissolve contrôlé par plusieurs critères
- combiner un mask de texture avec un mask procédural

Attention
Multiplier plusieurs valeurs < 1 réduit très vite l'intensité.
C'est mathématiquement correct, mais visuellement parfois trop agressif.

Condition OU (OR)

→ Add / Max

Logique classique : A OU B donne une condition valide

Approches shader possibles :

- **A + B (Add):** simple, fonctionne bien pour des effets additifs (lumière, glow), peut dépasser 1 → nécessite souvent un Saturate
  Add = accumulation d'énergie

- **max(A, B) (Max) :** correspond mieux à un OU logique strict, évite l'accumulation, très utilisé pour des masques binaires ou semi-binaires
  Max = sélection dominante

Exemples d'usage :

- afficher un effet si au moins une condition est vraie
- combiner plusieurs zones d'influence
- masques de terrain, decals, UI

Condition if / else

Un if CPU arrête un chemin de calcul.
Un shader, lui, calcule tout, puis sélectionne.

Approches shader :

- **Step** : créer une condition
  Step (edge, value), retourne 0 ou 1, sert à créer une condition nette.

  Exemple :

  condition = step(50m, distance)

- **Lerp** : choisir entre deux valeurs
  lerp(A, B, condition)

  si condition = 0 → A
  si condition = 1 → B

  entre les deux → transition

  Ensemble, ça donne l'équivalent de :

```
if (value >= threshold)
   return B
else
   return A
```

Node : Effets et Texture générés :

Sample Texture 2D
Échantillonne une texture via des coordonnée UV. Sélectionner l'espace de coordonnée et si la texture est une normal ou une texture classique.
L'input UV permet d'y branché des UVs modifié pour déformer la texture ou la déplacer. 128 textures max par shaders.
Par défaut utilise le samplerState de la texture, ce qui est limité à 16 si vous ne raccordez pas ensemble des sample 2D avec des samplerState.
Sample texture possède plusieurs types modifiables depuis les nodes settings pour des cas particuliers. Ils existent d'autre sample nodes qui permettent d'afficher les autres types de propriétés ou d'afficher une texture en fonction du monde et pas de la position de l'objet (triplanar).

Triplanar
Projette une texture sur les trois axes pour éviter les UVs. Coûteux (3 samples minimum). Peut créer des transitions floues ou des limites visibles si mal paramétrées. Cher en perf.

Fresnel
Calcule un masque basé sur l'angle entre la normale et la vue. Très utilisé pour rim light, outlines, effets énergétiques, eau, faux reflet. Sensible à la qualité des normales et à leur espace.

Simple Noise
Bruit simple procédural. Rapide, mais peu détaillé. Répétitif à grande échelle. Souvent utilisé pour du breakup ou des *masks*.

Voronoi
Génère des cellules, fracturées, pour de la pierre, des motifs organiques, etc. Plus coûteux que le noise simple.

Gradient noise
Bruit plus lisse et continu. Adapté aux déformations, nuages, gradients naturels. Plus cher que Noise, mais visuellement plus stable.

Dither
Crée un motif de tramage pour simuler une transparence ou un cutout progressif. Très utilisé avec Alpha Clip. Peut scintiller en mouvement ou en VR.

Erreurs et Pièges

Types d’Erreurs

Shadergraph **hérite des limitations des shaders** et en rajoute également.
De plus, le code de génération de shader est souvent critiqué, car **mal optimisé et instable**. Cela change à chaque mise à jour, mais lentement.

En dehors de l’interface et de l’implémentation de shadergraph qui peut donner des erreurs, un shader peut aussi avoir des erreurs et **shadergraph hérite des règles et limite des shaders**.

Un shader est constitué d’instruction simple fait à la chaine et mathématique. **Peu d’erreurs sont possibles dans les faits, car il est difficile de faire crasher des fonctions mathématiques.**

Il y a donc trois types d’erreur dû à :

Un problème de Unity

vous ne pouvez généralement rien y faire, mais un redémarrage suffit habituellement. Sinon, il faudra peut-être refaire la node ou le shader qui pose un problème.

Des erreurs de syntaxe

Arrivent si vous codez vous-mêmes le shader ou si vous utilisez une custom node dans shadergraph. Il faudra corriger l’erreur de code.

Des erreurs mathématiques

Ce sont des erreurs les plus courantes, mais les plus vicieuses. Comme ce sont des **erreurs de logiques**, le script ne vous les marque pas forcément comme une erreur, avec un message, mais donnera un **faux résultat et échouera silencieusement**.

Apparences des Erreurs

Une node responsable d'une erreur affichera un point d'exclamation rouge.

Quand un shader est en erreur sur Unity, il est affiché en magenta sur le material, vous indiquant un problème. Il peut arriver qu'il y est des détails dans la console.

Le magenta arrive pour des problèmes de syntaxe, mais arrivera surtout pour des erreurs mathématiques :

- Si un **infini** arrive dans un résultat (
    - additions trop grosses ;
    - division par zéro ;
    - exponentiels trop gros ;
    - accumulation (time par exemple)
    - Etc.

- Si un **NaN (Not an Number)** arrive dans les résultats (
    - propriété vide ;
    - racine négative ;
    - log négatif ;
    - acos avec des valeurs en dehors du -1 - 1 ;
    - Normalize mais avec (0,0,0) ;
    - incompatibilité et utilisation de fonctions non prises en charge par la pipeline ou le type de material
    - Etc.

Le material peut devenir également **cyan quand il load**. Quand cela est visible en build, cela brise l'immersion aux joueurs. Il est possible de précharger les shaders pour éviter ça (warming).

L’Enfer de la Transparence

Arrive souvent dans les logiciels de rendu en temps réel dû à la façon dont les logiciels simplifie les calculs de transparence.

Les objets opaques n’ont pas ce problème, car ils écrivent dans le *Depth Buffer* (ZWrite / depth write), ce qui permet le *Depth Testing* (ZTest / depth test) afin de trier les pixels individuellement. Unity rend aussi les objets opaques les plus proches de la caméra en premier, ce qui évite de perdre du temps à calculer des pixels cachés derrière d’autres.

Solutions :

- Il est possible de **forcer les objets transparents à écrire dans le *depth buffer*** avec l’option Graph Settings > Depth Write > Force Enable.
- Toujours **séparer les parties opaques et transparentes** d’un modèle en meshes distincts (ou au minimum en sous-meshes séparés) et leur appliquer des materials différents.
- Si votre objet n’a pas besoin de transparence partielle, envisagez d’utiliser un **material opaque avec *Alpha Clipping* à la place**. Cela peut aussi être combiné avec du *Dithering* pour simuler une certaine transparence. (cf. ce tuto).
- Dans certains cas, il est possible de **passer à un autre mode d’alpha blend** (*blend mode*). Les modes de fusion additive (Blend One One) et multiplicatif (par exemple Blend Zero SrcColor) produisent notamment le même résultat quel que soit l’ordre de rendu des objets.
- Les objets transparents sont triés en fonction de l’origine de leur mesh. Si vous savez qu’un material transparent particulier doit toujours apparaître derrière ou devant les autres, vous pouvez **utiliser la *Render Queue* ou la *Sorting Priority* du material** pour forcer l’ordre de rendu.
- Lors du rendu d’un mesh, les faces sont dessinées dans l’ordre de leur liste d’indices/triangles. Dans les cas où il n’y a pas de géométrie qui s’intersecte et où la forme conserve son ordre quel que soit l’angle de la caméra (comme des sphères superposées), il peut être possible de **pré-trier les triangles** afin de forcer l’ordre d’affichage. Cette méthode peut être plus performante que les précédentes, mais nécessite une configuration dans le logiciel de modélisation.

  Cela peut varier selon le logiciel de modélisation 3D, mais lorsque plusieurs meshes sont combinés, on part du principe que les triangles sont ajoutés à la

suite. **Il faut ajouter les formes du plus proche de la camera au plus loin.** Cela ne fonctionnera que pour les faces avant tandis que pour les faces arrière l'ordre sera inversé. Il faudra peut-être également décocher la case “Optimise Mesh : Polygon Order” lors de l'import du modèle dans Unity. Car cela pourrait réorganiser l’ordre du triangle.

Problèmes courants (FAQ)

FAQ

Mon image est teintée / perd sa teinte
Si vous utilisez des renderers d’image, ils peuvent teinter vos sprites avec l’option tint du renderer. Cela peut être désactivé dans les options des shadergraph d’UI, canvas et sprites dans graph settings > Disable Color Tint.

Vous pouvez accéder à la couleur paramétrée dans le renderer avec la node “Vertex Color”. L'image est accessible si ça référence dans ses options et “_MainTex”.

Shaders magenta après changement de pipeline : Il est fréquent que vos materials deviennent magenta si vous essayez de changer de pipeline de rendu. Cela est dû au fait que chaque pipeline utilise des paramètres et techniques différents. Vous pouvez convertir automatiquement tous les materials avec le convertisseur adaptés (cf. doc).

Le shader ne s’anime pas dans l’éditeur / scene view
Par défaut, la vue de la Scène ne s’actualise pas régulièrement afin d'optimiser les performances de l'éditeur. Si nécessaire, vous pouvez activer l'option « Always Refresh » dans le menu déroulant en haut à droite de la fenêtre de la scène.

Material grisé / Propriétés non modifiables
Pour les modèles 3D : Pour modifier les materials des modèles importés dans Unity, vous devez les extraire au préalable (option d’importation des modèles 3D). Unity peut également attribuer automatiquement un material par défaut à votre mesh, un material non modifiable. Vous devez alors créer votre propre material.

Pour ShaderGraph : Le material par défaut du shadergraph est aussi non modifiable. Pour en créer un nouveau, faites un clic droit sur votre shader et sélectionnez « Create > Material ».

Agitement / instabilité du rendu / saccadement

Peut-être dû à beaucoup de choses :

- Utilisation de coordonnée world au lieu de absolute world > instabilité lorsque la camera bouge.
- dû à une accumulation et des valeurs énormes
    - UV x exponentiel ;
    - Sine (time x 1000) ;
    - Etc.

  Vous atteigniez alors la limite des valeurs possibles de l'ordinateur, plus vous allez dans les grands nombre moins l'ordinateur peut faire des transitions fluides entre chaque nombre et on se retrouve avec un rendu haché ou saccadé (cf. half & single).

Le shader apparaît éteint/noir

Cause :

- Éclairage désactivé ;
- Couleur de base ;
- Normal manquante ;
- Normal Inversé ;
- Smoothness / alpha /ambiante occlusion avec des valeurs en dehors de 0-1.

Le shader Brille en blanc ou noir

Votre émission est trop forte ou votre alpha n'a pas des valeurs entre 0 et 1.

Erreurs de compilation/génération

Ces erreurs apparaissent généralement dans la console et empêchent la compilation du shader. Elles sont souvent dues à un bug Unity ou une limite d'une ressource atteinte.

“Shader error in ‘...’: syntax error”
Le code du Shader généré est incorrect ou incompatible.

Causes fréquentes :

- Le code du node de fonction personnalisée contient des erreurs ;
- Utilisation d’un HLSL non pris en charge par le pipeline de rendu actuel.

Solution : Vérifiez les nodes de fonction personnalisée et assurez-vous que le shader correspond au pipeline (URP/HDRP/Intégré).

“Failed to compile shader”
Des nodes produisent une sortie invalide. Recherchez les nodes surlignés en rouge et Déconnectez les nodes un par un pour trouver la source.

Erreurs de pipeline de rendu

Très fréquentes lors du changement de pipeline.

“Graph is not supported / compatible by the pipeline”
Le shader a été créé pour un autre pipeline ou n’a pas la bonne pipeline affecté dans ses options.

Solutions :

- Passer à URP/HDRP ;
- utiliser un shader existant ;
- Vérifier les erreurs de la console ;
- Assurer que la ressource de pipeline appropriée est affectée ;
- Utiliser l'outil de conversion de material de la pipeline actuelle.

Erreurs de nodes

Ces erreurs apparaissent dans le graphe de shader.

Nodes ou ports rouges

Dû à une connexion invalide ou manquantes.

- Compléter les nodes ;
- Vérifiez les types de données (Vector, Float, Color);
- Utilisez les nodes de conversion (Split, Combine, swizzle, transform, vector2,3,4, transform, Etc.) adaptées.

“Incompatible Port”
Connexion de types incompatibles donc faites correspondre les types d’entrée/sortie ou convertissez-les.

“Missing Input”
Port requis non connecté donc, ajoutez une valeur ou un node.

Erreurs spécifiques à la plateforme

Fonctionne dans l'éditeur, mais pas sur mobile

Dû à des nodes non pris en charge ou un modèle de shader trop complexe.

Réduisez la complexité du shader, évitez les calculs complexes, les boucles et les fonctions personnalisées.

“The shader required derivative, but there are not avalable”
Utilisation de nodes comme DDX/DDY sur des plateformes non prises en charge.

Debugging

Déboguer un shader n'a rien à voir avec déboguer du code CPU. Il n'y a :

- pas de console,
- pas de logs,
- pas de breakpoints,
- pas d'exécution séquentielle observable.

Un shader s'exécute des milliers de fois en parallèle, et ne peut communiquer qu'une seule chose : une couleur. La seule sortie observable est le rendu final à l'écran.

Déboguer un shader consiste donc presque toujours à visualiser des données.
Tout debug passe par la transformation d'une donnée en signal visuel

Voici quelques techniques de plus pour ShaderGraph :

Main Preview

La preview principale du Shader Graph est le **premier outil de debug**.

- Elle affiche le résultat final du shader dans un contexte simplifié
- Elle permet de vérifier rapidement si un calcul "fonctionne" ou non

Clic droit sur la preview :

- permet de changer le mesh (sphère, cube, plan, custom)
- essentiel pour détecter des erreurs dépendantes des normales, UVs ou tangentes

Attention

La preview ne reflète pas toujours :

- l'éclairage réel de la scène
- les conditions de transparence
- la profondeur ou le post-process

Preview Node

La Preview node est l'outil de debug local le plus important.

Elle permet de :

- visualiser le résultat d'un calcul intermédiaire
- sans l'envoyer vers l'output
- l'utiliser comme un breakpoint visuel

Debug via Base Color / Emission
Brancher temporairement une valeur sur Base Color ou Emission est une technique de debug standard.

Exemples :
- afficher une normale → (normal * 0.5 + 0.5)
- afficher une position → Fraction(position)
- afficher un masque → sortie directe en gris

L'émission est souvent préférable :
- elle ignore le lighting
- le résultat est plus lisible

Fraction et Step comme outils de debug
Certaines nodes ne servent pas qu'à l'effet final.

Fraction
- permet de visualiser des valeurs > 1
- transforme une valeur continue en motif répétitif 0–1

Très utile pour visualiser :
- positions world
- distances
- accumulations

Step
- permet de vérifier si une condition est vraie ou fausse
- transforme une comparaison en signal binaire visible

Exemple :
- vérifier si une distance dépasse un seuil
- vérifier si une coordonnée est dans une zone donnée

View Generated Shader

Shader Graph est une abstraction.

Quand un bug est :
- incompréhensible,
- lié à la précision,

- dépendant du pipeline,

il faut regarder le code généré spécifique d'Unity (ShaderLab) via le bouton "Generate ou " "View Generated Code" sur l'inspecteur du shader sélectionné dans les assets.

Compile And Show Code

via le bouton "Compile And Show Code" sur l'inspecteur du shader sélectionné dans les assets. Vous pouvez y sélectionner l'encodage utilisé (différentes plateformes, non pas le même encodage). Il faut impérativement cliquer sur "Generate" à chaque fois que vous voulez vérifier un changement, sinon le code affiché n'est pas à jour.

Permet de :
- voir le HLSL final réellement compilé
- comprendre ce que Shader Graph a optimisé, fusionné ou dupliqué

détecter :
- conversions implicites
- normalizations cachées
- variantes inattendues
- changements de précision

Ce n'est pas un outil pour débutants, mais c'est un outil clé pour comprendre les performances et expliquer certains comportements "magiques" de Shader Graph.

Déboguer, c'est simplifier

Règle d'or du debugging shader : **Si tu ne comprends pas ton graph, le GPU non plus.**

Bons réflexes :
- isoler un calcul
- remplacer des textures par des constantes
- tester avec des valeurs simples (0, 0.5, 1)
- supprimer tout ce qui n'est pas nécessaire au bug

Un shader se débogue rarement en ajoutant des nodes.

Il se débogue presque toujours en enlevant des nodes.

Optimisation

Node : Organisation

sticky note
Ajoute des commentaires visuels. Aucun impact sur les performances. Essentiel pour la maintenance.

Groupes
Vous pouvez regrouper des sections de nodes et les nommer. Encapsule visuellement des ensembles de nodes. Améliore la lecture, mais n'a aucun effet fonctionnel. Sélectionnez vos nodes, *clic droit > Group Selection* ou *Ctrl+G*.

Redirect
Permet de réorganiser les connexions sans tirer des longs câbles. Purement ergonomique.

Catégories
Organisent les propriétés dans l'inspecteur. Crucial pour la lisibilité côté artistes et designers. Aucun impact runtime. À créer au même endroit que les propriétés sur le "+". Vous pouvez glisser ou créer vos propriétés à l'intérieur pour les associer à la catégorie. La catégorie est refermable dans l'inspecteur ou dans le shadergraph.

Précision

La précision est la capacité d'une variable à représenter une valeur voulue.

Toutes les valeurs ne sont pas possibles avec Single et Half:

- Un certain nombre de chiffres est possible dans un nombre : si le float est limité à 5 chiffres, 553 212 et 553 210 deviennent équivalents.
- Tous les nombres n'existent pas donc il y a des écarts entre un nombre et le nombre d'après. Cet écart est plus important sur les nombres élevés. Sur les half 0.000061 est la valeur possible la plus proche de 0, toutes valeurs entre les deux, devient 0. Pour 60 000, la valeur la plus proche possible est 60 032.

Tableau comparatif entre Single et Half :

Nom	**Single**		**Half**	
Propriétés	**Valeurs**	**Exemples**	**Valeurs**	**Exemples**
Taille et type	32-bit float		16-bit float	
Valeur Min avant 0	0.000000000000000001 ($\sim10^{-38}$)	*Il n'y a aucune autre valeur possible entre 0 et 10^{-38}.*	0.000061	*Il n'y a aucune autre valeur possible entre 0 et 0.000061.*
Valeur Max avant infinie	99 999 999 999 999 999 (3.40×10^{38})	*Il n'y a aucune autre valeur possible entre infinie et 3.40×10^{38}.*	65504	*Il n'y a aucune autre valeur possible entre infinie et 65504.*
Nombre de chiffres possibles	~7	*1.50585451 est convertie en 1.5058540.*	~3	*10.856 est convertie en 10.9.*
Écart moyen vers 0	~0.000000119 (1.19×10^{-7})	*Il n'y a pas d'autres nombres possibles entre 1 et 1.00000012.*	0.00098	*Il n'y a pas d'autres nombres possibles entre 1 et 1.00098.*
Nombre à partir duquel l'écart entre les chiffres est supérieur à 1	~8 300 000	*Il n'y a pas d'autres nombres possibles entre 8 388 608 et 8 388 609.* *8 388 608 + 0.5 = 8 388 608*	~1024	*Il n'y a pas d'autres nombres possibles entre 1024 et 1025.* *1024 + 0.3 = 1024*
Usages	- N'importe quoi avec des grandes valeurs possibles. - World-space et screen-space position math - Lighting - Depth - Temporal effects - Accumulation et boucle - Matrix math - Depth - Accumulation, loops - Calculs avec du temps - émission - Pour n'importe quoi d'important -		- Pour des valeurs de couleur dans un espace linéaire ou de RGB01. - Sur des Normals ou des vecteurs normalisés. - pour faire des calculs ou des déplacements d'UV. - pour faire du Lerp. - Afin d'utiliser des masks. - Pour le Mobile ou des contextes avec peu de bandwidth. -	

Commentaire	Précision par défaut sur la plupart des GPU d'ordinateurs. Deux fois plus grand que Half en taille, mais donne accès à **beaucoup de détails**.	Valeurs deux fois plus petites qu'un Single. La quantification en Half est très agressive. Les petites différences disparaissent. Les cumuls dérivent. Tout processus itératif (sommes d'éclairage, évaluations BRDF, intégration du brouillard) peut se dégrader visiblement s'il est maintenu trop longtemps en Half . Cela se traduit par des bandes, des scintillements ou des reflets instables. Considérez ça comme une **compression agressive**.

Shader Variants & Build Time

Dans Unity, les **variantes de shader** sont générées en fonction des keywords utilisés dans un shader. Shader Graph, en particulier, peut ajouter un grand nombre de keywords automatiquement, notamment selon le **Render Pipeline** utilisé (URP, HDRP, Built-in).

Génération des variantes

Chaque keyword booléen (activé ou désactivé) double le nombre de variantes possibles. Ex : vrai va créer un shader avec que les nodes qui y sont branchées et faux fait de même pour les nodes qui l'utilise.

Par exemple :

- **7 keywords booléens**
- Chaque keyword a **2 états** (On / Off)
- Nombre total de variantes générées : 2^7 = 128 variantes

Unity impose une **limite maximale de 128 variantes par shader**. Dans cet exemple, la limite est atteinte exactement. Si vous dépassez cette limite, Unity ne génère pas les variantes supplémentaires.

En pratique, le nombre total de variantes peut être approximé par :

Pow (2 , nb keywords + nb keywords de Unity)

Les keywords ajoutés automatiquement par Unity (liés aux lights, shadows, fog, etc.) sont souvent sous-estimés et peuvent faire exploser le nombre de variantes sans que le développeur s'en rende compte.

Impact sur le temps de build

Le nombre de variantes **n'est réellement visible qu'au moment du build.**

Plus vous avez de variantes :

- plus le **temps de build** augmente de **beaucoup**,
- plus la **taille du build** est importante,
- plus la **compilation des shaders** est coûteuse.

En Éditeur, Unity compile souvent les variantes à la volée, ce qui peut masquer le problème jusqu'au build final.

Shader stripping et erreurs

Lors du build, Unity tente de **stripper** (supprimer) les variantes de shader qui ne semblent pas nécessaires afin de réduire la taille finale. Cependant :

- Si Unity ne trouve pas la variante exacte dont il a besoin à l'exécution, il essaie de sélectionner une **variante similaire.**
- Si aucune variante compatible n'est trouvée, Unity utilise le **shader d'erreur magenta.**

Cela se traduit par des objets roses dans la scène, signe qu'une variante de shader est manquante.

Always Included Shaders

Attention importante :

Si vous ajoutez un shader dans la liste **Always Included Shaders** (Graphics Settings) :

- Unity inclut **toutes les variantes possibles** de ce shader dans le build,
- même celles déclarées avec *shader_feature*.

Cela peut annuler complètement les bénéfices du shader stripping et faire exploser le temps de build et la taille du projet.

Problème des keywords multiples dans un même set

Si vous utilisez *shader_feature* et que **plusieurs keywords du même set/enum sont activés en même temps** (possible avec le code), Unity :

- tente de choisir la variante la plus proche,
- ce qui peut provoquer des **résultats visuels inattendus** ou des comportements incorrects du shader.

Bonnes pratiques

- Limiter le nombre de keywords
- Éviter les combinaisons inutiles
- Surveiller les keywords ajoutés automatiquement par Shader Graph
- Préférer les **keywords locaux** (Local Keywords)

Scripting C# avec Shader Graph

Le scripting C# permet de contrôler dynamiquement les shaders créés avec Shader Graph. Cela inclut la modification des materials, l'activation de mots-clés (*keywords*), ainsi que l'optimisation des performances lors de l'utilisation de plusieurs instances d'objets.

Material

Un **Material** est l'instance d'un shader. Modifier un material via script affecte généralement **tous les objets** qui utilisent ce material, sauf si une instance unique est créée.

material.SetFloat("_Intensity", 1.0f);
material.SetColor("_Color", Color.red);

Attention :

- L'accès à *renderer.material* crée **une nouvelle instance** du material en mémoire.
- L'accès à *renderer.sharedMaterial* modifie le material partagé par tous les objets.

Une mauvaise gestion peut entraîner une **création excessive d'instances** et des problèmes de performance.

Les noms utilisés en C# doivent correspondre au ***Reference* Name** défini dans Shader Graph (et non au nom affiché).

Keywords

Lorsqu'un Enum est contrôlé par script, il faut utiliser le format suivant :

{REFERENCE}_{REFERENCESUFFIX}

Exemple : Si le Reference Name est *MYENUM* et que l'option choisie est *OPTION1:*
material.EnableKeyword("MYENUM_OPTION1");

Important :
Lorsque vous activez une option d'un Enum, vous devez **désactiver manuellement les autres options**, sinon le shader ne se mettra pas à jour correctement.

```
material.DisableKeyword("MYENUM_OPTION2");
material.DisableKeyword("MYENUM_OPTION3");
```

Shader branching et multi_compile

Si vous avez besoin que le shader change de comportement **à l'exécution** (runtime), vous devez utiliser *multi_compile*.

Exemple **:** Activer ou désactiver le brouillard, Activer des effets visuels optionnels, Donner un contrôle dynamique au joueur, etc.

Attention : Un usage excessif de *multi_compile* peut augmenter le temps de compilation et la taille du build.

Performance : LocalKeyword vs string

Les fonctions comme *EnableKeyword(string)* sont **plus lentes** que celles utilisant un type *LocalKeyword*.

Bonne pratique :

- Créer un *LocalKeyword*
- Le mettre en cache au start

```
LocalKeyword myKeywordRef;

void Start()
{
   myKeywordRef = new LocalKeyword(shader, "MYKEYWORD");
}
```

```
void Update()
{
   material.EnableKeyword(myKeywordRef);
}
```

Cela est particulièrement important si le keyword est activé/désactivé fréquemment.

MaterialPropertyBlock

Le **MaterialPropertyBlock** est la méthode recommandée pour modifier des propriétés de shader **par objet**, sans créer de nouvelles instances de material.

Avantages :

- Pas de duplication de material
- Meilleures performances
- Idéal pour les objets instanciés (ennemis, particules, props)

```
MaterialPropertyBlock block = new MaterialPropertyBlock();
Renderer renderer = GetComponent<Renderer>(); block.SetFloat("_Intensity", 1.0f);
renderer.SetPropertyBlock(block);
```

À privilégier lorsque plusieurs objets partagent le même material mais les valeurs doivent varier par instance.

Pour aller plus loin

Il existe des vidéos de toutes les nodes sur shadergraph.
Beaucoup de nodes sont traduisibles entre **blender**, **Unreal**, **Unity**, **Substance Designer** et **Texture Lab,** regarder les tutos de ces logiciels marche également.

Liens Utiles :

- Docs :
  doc :
  https://docs.unity3d.com/Packages/com.unity.shadergraph@17.3/manual/index.html
  shadergraph doc :
  https://docs.unity3d.com/6000.3/Documentation/Manual/materials-and-shaders.html

- Book :
  https://jettelly.com/store/the-unity-shaders-bible
  https://jettelly.com/store/visualizing-equations-vol-2
  https://jettelly.com/store/visualizing-equations-vol-1
  https://unity.com/resources/tech-artists-key-toolsets

- ShaderGraph :
  https://www.youtube.com/@RemyMaetz
  https://www.youtube.com/@MinionsArt
  https://www.youtube.com/@BenCloward

- Math :
  https://www.youtube.com/@acegikmo
  https://www.desmos.com/calculator?lang=fr

Table des Matières Complète :