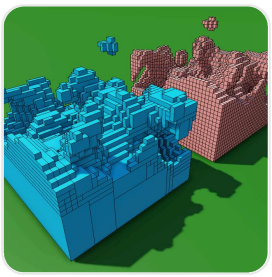
ChatGPT

Optimisations avancées pour un jeu voxel Unity HDRP

Représentation et gestion efficace des voxels

Structures de données optimisées : Représenter les voxels de manière compacte et efficace est crucial. Évitez les structures lourdes comme les dictionnaires pour stocker les blocs – elles ajoutent du surcoût en mémoire et en temps d'accès reddit.com . Préférez un simple tableau 1D indexé pour chaque chunk, en combinant les coordonnées `(x,y,z)` en un indice unique (bit shifting) reddit.com . Cette approche réduit la fragmentation et améliore la cohérence de cache reddit.com , ce qui accélère les itérations sur les voxels. Choisir une **taille de chunk** adaptée est également important : des chunks plus grands réduisent le coût relatif du meshing aux bordures. Par exemple, des chunks de **16×(hauteur du monde)×16** sont un choix courant pour équilibrer nombre de chunks et taille de maillages reddit.com (similaire à Minecraft). Cela limite le nombre d'objets à gérer tout en maintenant un bon niveau de détail.

Génération de maillages (meshing) efficace : Pour un monde voxel destructible, il faut convertir les données de voxels en maillages 3D de manière performante. Une technique de base est de **ne générer que les faces "exposées"** des voxels, c'est-à-dire celles qui ne sont pas complètement entourées par d'autres blocs solides ou hors du champ medium.com . Ainsi, les faces internes cachées ne sont pas rendues du tout, ce qui réduit d'emblée le nombre de polygones à dessiner. On peut pousser plus loin avec le **greedy meshing** (*maillage glouton*), qui consiste à fusionner en une seule grande surface les faces contiguës de même type sur un même plan medium.com . En combinant de nombreux petits triangles en quelques grands, on réduit drastiquement le nombre de sommets et de triangles à dessiner, ainsi que les appels de rendu.



devforum.roblox.com *Exemple d'optimisation par greedy meshing.* La scène rouge (droite) rend chaque face de voxel séparément (~15 931 éléments), tandis que la scène bleue (gauche) utilise le greedy meshing et ne comporte plus que ~508 éléments à dessiner, sans perte visuelle devforum.roblox.com .

En pratique, le greedy meshing allège énormément la charge GPU en réduisant les polygones à traiter. Un développeur rapporte être passé d'une grille de 200^3 voxels à seulement ~2 000 polygones rendus grâce à ces optimisations – soit **0,025%** du total initial reddit.com . Ce gain monumental illustre à quel point diminuer le nombre de triangles améliore le framerate sur des gros mondes voxels. Notez que le greedy meshing peut légèrement augmenter le temps de génération de chunk (algorithme plus complexe) reddit.com , mais ce coût CPU est largement compensé par le soulagement du GPU ensuite.

Niveau de détail (*LOD*) adaptatif : Mettre en place un système de **LOD** pour les voxels permet de maintenir un framerate élevé même avec un très grand horizon de vue. L'idée est d'utiliser des maillages moins détaillés pour les zones lointaines. Par exemple, à grande distance, on peut afficher un chunk sur deux (échelle 2:1) ou fusionner plusieurs voxels en un seul polygone plus grand. Des approches avancées vont jusqu'à **diviser ou fusionner dynamiquement les chunks** en fonction de la distance à la caméra, de sorte que la taille apparente d'un chunk reste similaire à l'écran reddit.com . Ainsi, près du joueur on a des petits chunks très détaillés, et au loin de gros chunks englobants avec moins de détails. Cette stratégie permet des distances de vue quasi infinies sans faire exploser le nombre de polygones, tout en évitant les changements brusques : la transition de LOD peut être gérée en fond (threads) pour ne pas bloquer le rendu reddit.com . Unity ne fournit pas de LOD automatique pour des terrains destructibles, mais vous pouvez vous inspirer des systèmes de terrain classiques : par exemple, utiliser les **Groupes de LOD** sur les objets lointains ou un système d'**HLOD** (LOD hiérarchique) qui combine plusieurs chunks éloignés en une seule mesh simplifiée medium.com . Des assets comme *AutoLOD* (gratuit) peuvent générer des maillages simplifiés automatiquement medium.com , mais pour un terrain voxel destructible, il faudra souvent coder une solution sur mesure. L'essentiel est de réduire la géométrie rendue au loin, soit en omettant les petits détails, soit en les remplaçant par des formes plus simples (par ex. utiliser un voxel pour une zone de 2×2×2 à distance, etc.).

Rendu performant en HDRP

Matériaux et shaders optimisés : L'utilisation de l'**HDRP** (High Definition Render Pipeline) offre une qualité visuelle élevée, mais chaque shader et matériau doit être optimisé pour un rendu massivement voxelisé. Idéalement, utilisez **un matériau unique avec atlas de textures ou textures array** pour tous les types de blocs opaques. De cette manière, tous les voxels d'un chunk se dessinent en un seul draw call, sans changement de matériau. Deux approches existent pour gérer les multiples textures de blocs avec un seul shader : (1) combiner les textures dans un *Texture3D/Array* et calculer l'index de texture dans le shader en fonction d'un ID de matériau stocké par face reddit.com , ou (2) ne fusionner en greedy mesh que les faces de même type de bloc, puis dessiner chaque matériau ID séparément (ce qui donne plusieurs sous-maillages, donc plus de draw calls) reddit.com . La première approche (atlas ou texture array) est recommandée pour réduire au minimum les appels de rendu, au prix d'un shader un peu plus complexe. Assurez-vous que le shader utilise bien le **SRP Batcher** d'Unity (la plupart des shaders HDRP par défaut le supportent), afin que l'envoi des données des nombreux objets soit bien optimisé côté CPU. En outre, activez le **GPU Instancing** sur le matériel si possible (bien que dans le cas d'un mesh combiné par chunk, instancer chaque chunk apporte peu, c'est surtout utile si vous aviez encore de multiples petits objets séparés). Pour les blocs transparents (eau, verre), essayez de les regrouper également par chunk et matériau pour éviter de multiplier les passes. N'abusez pas des matériaux transparents car ils empêchent certaines optimisations (pas d'écriture dans le depth buffer, pas de culling par GPU en amont) et peuvent provoquer de l'**overdraw** (sur-dessin de pixels multiples fois). Si vous avez de vastes plans d'eau, envisagez de les représenter par des meshes spécialisés (par ex. un plan d'eau par chunk, au lieu de milliers de petits voxels d'eau) afin de rendre la transparence plus efficacement.

Éclairage et ombrages : En HDRP, l'éclairage réaliste peut être coûteux, surtout avec un monde entier dynamique (donc non pré-calculé). Pour un **monde voxel destructible**, le baking d'éclairage global n'est pas possible sur les éléments modifiables en temps réel. Il faut donc s'appuyer sur de l'éclairage **temps-réel optimisé**. Privilégiez un nombre limité de lumières dynamiques: typiquement une **lumière directionnelle** (soleil) avec des shadows cascades pour l'éclairage principal, et utilisez des **Light Probes/Reflection Probes** ou les **Probe Volumes** d'Unity pour fournir un éclairage d'ambiance aux zones non directement éclairées. Les Probe Volumes (introduits dans Unity 2022+ HDRP) permettent d'obtenir un éclairage global approximatif à grande échelle avec un coût modéré, mais ils sont statiques par défaut – dans un monde destructible, vous pourriez les rafraîchir ou recalculer partiellement si de gros changements se produisent, mais c'est complexe. À défaut, l'**Ambient Occlusion** en temps réel (SSAO) peut ajouter du relief visuel pour les cavernes ou coins, sans coût trop élevé. Évitez d'utiliser le **Ray Tracing DXR** de l'HDRP pour l'illumination globale ou les ombres dans un tel projet, car bien que tentant pour la qualité, cela exploserait le coût GPU (sauf peut-être en offrant une option aux joueurs haut de gamme). Préférez les techniques écran (comme le **SSGI** – Screen Space Global Illumination) pour un léger GI en temps réel, en sachant que le SSGI reste gourmand. **Limitez la distance des ombres** dynamiques et le nombre d'objets qui les projettent : marquez éventuellement les petits débris ou blocs lointains pour qu'ils ne projettent pas d'ombre, ou utilisez le système de *shadow caching* d'HDRP pour ne recalculer les ombres des objets statiques que lorsque nécessaire discussions.unity.com . En réglant intelligemment la qualité des ombres (résolution, nombre de cascades, distance) et en désactivant les lumières inutiles, vous pouvez gagner beaucoup de ms de rendu.

Transparence et effets post-traitement : Comme mentionné, les surfaces transparentes doivent être utilisées avec parcimonie. En HDRP, le coût des objets transparents s'ajoute après l'opacité, et il n'y a pas de culling aussi efficace. Utilisez éventuellement le **Z-Prepass** (pré-pass de profondeur) d'HDRP pour que les objets opaques occultent les pixels des transparents (réduisant l'overdraw). Pour les *post-process*, réglez-les au strict nécessaire. Des effets comme le Bloom, Depth of Field, Motion Blur, etc., apportent du confort visuel mais peuvent coûter cher sur un grand écran à haute résolution. Dans une optique "framerate maximal", on peut réduire leur qualité ou les désactiver sur les machines moins puissantes. Un développeur indique par exemple qu'en désactivant ou ajustant certains post-process lourds, il a pu passer son projet HDRP à ~80 FPS au lieu d'un framerate bien plus bas auparavant discussions.unity.com . Profitez du **Profiling GPU** (via l'outil *Frame Debugger* ou *Profiler* d'Unity) pour identifier les passes les plus coûteuses et affiner les réglages. Par exemple, si le *Volumetric Fog* ou les *Clouds* HDRP consomment trop, envisagez de les désactiver ou de diminuer leur qualité à distance. De même, si votre jeu est surtout en extérieur, le **Sky Lighting** peut être pré-calculé (cubemap d'éclairage) plutôt que recalculé en temps réel. En somme, chaque feature HDRP (éclairage volumétrique, réflexions planaires, etc.) doit être évaluée: **désactivez celles qui ne contribuent pas assez au visuel par rapport à leur coût**.

Optimisation de l'utilisation CPU et GPU

Multi-threading C# et Burst : Unity 6.1 offre un **Job System** C# et le compilateur **Burst** pour exploiter les CPU multi-cœurs à fond. Il est fortement conseillé d'en tirer parti pour le calcul des chunks – génération du terrain (bruit procédural), mise à jour des données de voxels, calcul des maillages, etc., peuvent être effectués sur des threads de fond au lieu de bloquer le thread principal (render thread). En partitionnant le travail (par exemple, traiter N chunks par frame en tâche de fond), on évite les pics de calcul. Le compilateur Burst traduit le code C# structuré en *jobs* en code natif ultra-optimisé, offrant souvent un gain de performance très important (on parle d'améliorations pouvant aller jusqu'à **x10 sur les tâches de génération** reddit.com par rapport à du C# mono-thread classique). En pratique, cela signifie que la création d'un chunk ou sa modification (minage d'un voxel, explosion qui modifie le terrain) peut passer de plusieurs millisecondes à quelques centaines de microsecondes grâce à Burst, rendant les mises à jour quasi transparentes pour le joueur. Pour en bénéficier, utilisez les conteneurs de données natifs d'Unity (par ex. `NativeArray` pour vos tableaux de voxels) et marquez les jobs avec `[BurstCompile]`. Pensez aussi au **Job de type IJobParallelFor** pour générer plusieurs portions de mesh en parallèle (par ex. diviser un chunk en sections et traiter chaque section en parallèle, ou traiter plusieurs chunks simultanément). Un article détaille comment offloader le *meshing* sur des threads puis ne renvoyer le mesh fini au main thread qu'une fois prêt reddit.com, ce qui évite tout gel visuel.

Shaders de calcul (Compute Shaders) : Le GPU peut aussi être mis à contribution pour certaines tâches massives via des compute shaders (HLSL). Par exemple, on pourrait envisager de générer la géométrie voxel directement sur GPU, ou de calculer l'érosion/terrains avec un compute. L'avantage est le parallélisme massif du GPU, mais il faut faire attention **aux retours de données vers le CPU**, qui sont lents. Comme le note un développeur, *"le read-back GPU est lent, je préfère utiliser des jobs Burst pour tout ce qui doit revenir au CPU, et réserver les compute shaders aux données qui restent sur le GPU"* reddit.com. Ainsi, si vous utilisez un compute shader pour, disons, calculer une texture de bruit ou une carte d'occlusion, utilisez-la directement dans un shader de rendu sans rapatrier les données. En revanche, pour construire un Mesh Unity (qui est une structure CPU à la fin), passer par le GPU n'est pas forcément gagnant car il faudrait copier le résultat en RAM. Un compromis moderne consiste à utiliser le GPU pour filtrer ou marquer des voxels (par ex. un compute pourrait déterminer quelles zones du monde nécessitent un LOD plus fin, ou quels voxels sont exposés) et ne rapatrier qu'une petite liste de résultats, pendant que le détail du mesh est fait côté CPU en parallèle.

Culling et gestion de la visibilité : Profitez des mécanismes de **culling** d'Unity afin d'éviter tout travail inutile de rendu. Par défaut, Unity fait du **frustum culling** – les MeshRenderers (vos chunks) en dehors de la caméra ne seront pas dessinés. Assurez-vous que les Bounds de vos meshes de chunk sont correctement définis (Unity les calcule automatiquement d'après les vertices, généralement). Pour un grand monde voxel, il est aussi vital d'introduire un **culling d'occlusion** – c'est-à-dire ne pas dessiner les chunks complètement cachés derrière d'autres. Unity propose un Occlusion Culling basé sur des données pré-calculées (baking) pour les objets statiques medium.com . Si de larges portions de votre monde sont relativement statiques, vous pouvez les marquer *Static Occluder/Occludee* et utiliser cette fonctionnalité pour que, par exemple, les caves profondes ou l'intérieur des montagnes ne rendent pas les chunks derrière. Cependant, dans un monde destructible, le pré-calcul d'occlusion devient obsolète dès que le joueur modifie le terrain (ouvrir un mur crée une nouvelle ligne de visée). Il faut donc soit rebaker régulièrement (peu pratique), soit implémenter un culling d'occlusion dynamique simplifié. Par exemple, vous pouvez programmer un test de visibilité entre chunks voisins: si un chunk est entouré de voisins opaques plus hauts, il est caché. Ce genre d'astuce est complexe à généraliser, mais au minimum **ne rendez pas les faces de voxels qui sont adossées à d'autres voxels opaques**, ce qui est déjà assuré par le face culling évoqué plus haut. En résumé, grâce au culling, idéalement **seuls les chunks visibles par la caméra et exposés** sont réellement soumis au rendu.

Autres optimisations CPU/GPU utiles : Exploitez le **burst de Groupes d'instances** avec `DrawMeshInstancedIndirect` si vous souhaitez dessiner énormément d'objets répétitifs via le GPU en une seule fois – ceci est moins applicable aux voxels (qu'on combine déjà en meshes), mais peut servir pour des effets comme des végétations voxelisées ou des particules de débris. Utilisez le **profiling Unity** pour trouver les goulots d'étranglement CPU (par ex. trop de GameObjects, Garbage Collector qui s'active fréquemment, etc.) et adressez-les. Par exemple, si la collecte de garbage cause des à-coups, passez en **Garbage Collector incrémental** (Player Settings) et évitez d'allouer pendant le jeu (d'où l'intérêt du pooling plus bas). Côté GPU, surveillez l'occupation via le *Profiler* GPU ou des outils comme NSight/RenderDoc afin de voir si vous êtes bound par le rasterisation, le calcul shader, la bande passante mémoire, etc., et ajustez les shaders/effets en conséquence. Enfin, n'oubliez pas d'activer le mode **"Graphics Jobs"** dans Unity (option de player) qui permet d'exécuter certaines commandes de rendu sur des threads worker, libérant un peu le main thread en scénario CPU-bound.

Streaming des chunks et gestion mémoire

Chargement asynchrone des chunks : Pour un monde constitué de milliers de chunks, il est indispensable de **charger et décharger dynamiquement** les régions en fonction de la position du joueur. Ne tentez pas d'instancier tous les chunks 100×100 d'un coup au début – même si cela tient en mémoire, le rendu de 10 000 chunks simultanément serait inutilement coûteux si le joueur n'en voit qu'une partie. À la place, adoptez une stratégie de **streaming** spatial : définissez un rayon actif autour du joueur (par ex. 10 chunks de distance). À chaque fois que le joueur se déplace et entre dans un nouveau chunk, calculez les chunks qui entrent dans le champ et ceux qui en sortent. Unity propose pour cela son système d'**Addressables** qui peut faciliter le chargement d'assets de manière asynchrone (utile surtout si vos chunks sont pré-construits ou stockés sous forme de scènes ou d'asset bundles). Comme le suggère un développeur, *« Unity a un excellent package Addressables pour le streaming de contenu, à ajuster pour votre usage »* reddit.com . Vous pouvez par exemple créer un prefab ou une scène par chunk de terrain et demander son chargement via Addressables lorsque nécessaire reddit.com . Si vos chunks sont purement générés procéduralement à la volée, l'intérêt d'Addressables est moindre (pas d'asset à charger), mais vous pouvez quand même utiliser des *Coroutine* ou des *Async Operations* Unity pour étaler le coût de génération sur plusieurs frames.

Pooling et cycles de vie des chunks : La création et destruction fréquente de GameObjects et Meshes peut causer des ralentissements (allocations, fragmentation, et surcoût du GC). Pour y remédier, mettez en place un **pool de chunks** réutilisables. Par exemple, instanciez au lancement un certain nombre de chunks "vides" (objets contenant un MeshFilter, MeshRenderer, etc.) et conservez-les inactifs. Quand un chunk doit apparaître, prenez-en un du pool, positionnez-le et assignez-lui le mesh généré, plutôt que d'en créer un nouveau medium.com medium.com . Inversement, quand un chunk sort de la zone active, désactivez-le et renvoyez-le dans le pool au lieu de le détruire medium.com . Cette technique de pooling réduit drastiquement les *spikes* de frame lors des phases de chargement/déchargement, et garde la mémoire plus stable. Veillez aussi à réutiliser les buffers de mesh si possible : vous pouvez garder en mémoire des tableaux de vertices/indices de taille fixe et les remplir pour chaque nouveau mesh, au lieu d'en allouer des nouveaux à chaque fois.

Gestion mémoire et stockage des données : Représenter un monde destructible de grande taille peut consommer beaucoup de mémoire si on n'y prend garde. Quelques astuces pour réduire l'empreinte : stockez les types de voxels sur le type numérique le plus petit possible (par ex. `byte` si <256 types de blocs, ou `ushort` si <65k types) plutôt qu'un `int` 32-bit par voxel – vous diviserez par 4 ou 2 la RAM utilisée. Vous pouvez également compresser les chunks qui sont entièrement vides ou uniformes en ne stockant qu'un "remplissage" au lieu d'un tableau complet (technique des *sparse chunks* ou des *run-length encoding* sur la hauteur). Si le monde dépasse la taille de la mémoire vive, envisagez de stocker sur disque les zones lointaines ou déjà visitées (système de *pagefile* maison ou en s'appuyant sur des *asset bundles*). Unity 6 introduit peut-être un meilleur support des **mondes massifs** avec des coordonnées 64-bit, mais en attendant il est courant d'utiliser la technique du **floating origin** : recaler périodiquement l'origine du monde autour du joueur pour éviter les problèmes de précision flottante lorsque les coordonnées deviennent très grandes. Par exemple, tous les 1000 unités de déplacement, recentrez le monde en soustrayant ce décalage aux positions de tous les objets (Unity ne le fait pas nativement pour vous en 6.1, donc c'est à implémenter). Cette astuce permet d'éviter le scintillement des objets lointains si jamais votre monde dépasse ~10 km de largeur en coordonnées Unity.

Exemple de stratégie de streaming : Le moteur de Minecraft est souvent cité comme référence en gestion de chunks. Il utilise un système de **file d'attente prioritaire** pour le streaming : les chunks proches du joueur ont un haut niveau de priorité de chargement/meshing reddit.com . Une implémentation courante consiste à avoir deux files : une pour les chargements (génération des données de voxel) et une pour le meshing (génération des Mesh). À chaque frame, vous traitez quelques éléments de chaque file pour lisser la charge reddit.com . Cela évite de gros à-coups quand le joueur bouge rapidement. Vous pouvez également prédire le mouvement du joueur (par exemple sa direction) et **pré-charger** en priorité les chunks vers lesquels il se dirige, tout en déchargeant ceux qu'il s'éloigne de façon sûre. En procédant ainsi, certains moteurs parviennent à un streaming presque invisible, sans stuttering, même en augmentant la distance de vue reddit.com . L'important est de **mesurer le temps de ces opérations** et d'ajuster le nombre de chunks traités par frame pour ne pas saturer le budget. Profitez du Job System pour effectuer les calculs de sélection de chunks à charger (distance, visibilité) en tâche parallèle également, surtout si vous devez parcourir des centaines de positions de chunks chaque fois – en C# pur, cela peut causer des ralentissements notables reddit.com .

Astuces spécifiques à Unity 6.1 HDRP pour les grands mondes destructibles

Restez à jour avec les dernières fonctionnalités : Unity 6.1 (et les versions avoisinantes) apporte des améliorations notables en performance et en capacités pour les grands mondes. Par exemple, HDRP a migré vers un **Render Graph** interne qui optimise l'utilisation du CPU et du GPU en ordonnançant mieux les passes de rendu, tout en réduisant les bugs et en améliorant les outils discussions.unity.com . Assurez-vous d'activer et d'utiliser ces nouveautés (consultez les notes de version Unity 6.0/6.1). De même, le pipeline de données DOTS (Entities 1.0+) est désormais stable et peut être considéré si vous avez besoin de gérer **plusieurs millions de voxels actifs** efficacement. DOTS permet de traiter des *chunks* d'entités de manière vectorisée et cache-friendly, ce qui est exactement le cas d'usage d'un monde voxel (des milliers de blocs mis à jour/rendus). Des retours d'expérience indiquent que DOTS excelle quand on a un très grand nombre d'objets simples discussions.unity.com , ce qui correspond aux voxels. Cependant, migrer tout un projet en ECS est un investissement lourd – on peut obtenir d'excellents résultats avec le mono-comportement classique + jobs/burst bien conçu, donc évaluez le rapport bénéfice/effort selon votre projet.

Outils utiles de l'écosystème Unity : Profitez des **Asset Store** et ressources de la communauté. Par exemple, des frameworks open-source comme *HertzVox 2* ou *Vloxy Engine* démontrent comment structurer un moteur voxel autour de Burst et du GPU github.com . Le plugin *Voxel Play 2* (Asset Store) offre une solution clé en main de monde voxel infini avec streaming, pouvant servir de référence. Pour le streaming de mondes, des assets comme *World Streamer 2* ou *SECTR* permettent de diviser la scène et de charger/décharger des sections géographiquement. Côté optimisation graphique, n'hésitez pas à utiliser le **Profiler** et le **Frame Debugger** d'Unity pour traquer les problèmes spécifiques à HDRP. Le **Memory Profiler** peut vous aider à visualiser l'utilisation mémoire de vos chunks et détecter d'éventuelles fuites ou fragmentation. Unity 6.1 étant relativement récent, gardez un œil sur les forums et le *Issue Tracker* Unity pour tout bug de performance ou optimisation manquante qui serait corrigé dans une patch (par exemple, certains utilisateurs ont rapporté des *spikes* de culling en HDRP dans le passé discussions.unity.com – ces problèmes tendent à se résoudre avec les updates).

Équilibrer qualité et performances : Même avec un moteur optimisé, un **monde voxel HDRP** pousse le hardware dans ses retranchements. Il peut être judicieux d'offrir des **options graphiques** aux joueurs. Unity HDRP supporte nativement le **Dynamic Resolution Scaling (DRS)** et les techniques de sur-échantillonnage comme **AMD FSR** ou **NVIDIA DLSS**. Celles-ci permettent de réduire la résolution de rendu interne (par ex. rendre à 75% ou 50% de la résolution) tout en upscalant l'image de façon assez propre, ce qui augmente énormément le framerate sur les GPU limités. Un développeur suggère même d'*"y aller à fond avec le DRS, par exemple rendre à 50% avec FSR, puis ajuster la netteté pour compenser"* discussions.unity.com , ce qui peut littéralement doubler le fps (au détriment d'un léger flou). En HDRP 6.x, Unity intègre ces options assez facilement via le *Dynamic Resolution Handler*. De plus, pensez à gérer le **VSync** de manière appropriée (désactivé si vous visez le framerate maximal non bridé, ou activable en option pour éviter le tearing). Enfin, testez sur différentes configurations et **profilez sur de véritables builds** (le mode Play en Éditeur est toujours moins performant). Vous pourriez découvrir que vous êtes GPU-bound sur certaines cartes (nécessitant de baisser la densité de voxels ou la qualité HDRP), ou au contraire CPU-bound (dans ce cas, réduire le nombre de chunks actifs ou optimiser davantage les jobs). Ajustez alors vos techniques en conséquence.

En synthèse, combiner ces optimisations – structures de données compactes, génération de mesh intelligente, rendering HDRP ajusté, utilisation judicieuse du multi-threading/GPU, streaming asynchrone – vous permettra de réaliser un **monde voxel destructible, visuellement riche et fluide**. Le tableau ci-dessous récapitule les techniques clés et leur impact estimé sur les performances du jeu.

Tableau récapitulatif des techniques et de leur impact sur les performances

Technique d'optimisation	Description	Impact sur performances
Face culling des voxels (faces exposées seulement)	Ne générer que les faces visibles (air/libres) de chaque bloc au lieu de tous les côtés.	Réduction majeure du nombre de polygones ~6× moins), allégeant fortement la charge G medium.com .
Greedy meshing (fusion de faces contiguës)	Regrouper les faces adjacentes de même type en grandes surfaces.	Énorme baisse du nombre de triangles et dra ex: ~0,025% des triangles initiaux restants re Amélioration drastique du rendu GPU (fps +-
LOD et HLOD pour voxels	Utiliser des maillages moins détaillés pour les chunks lointains; fusionner/découper dynamiquement les chunks selon la distance.	Réduction du nombre de chunks/triangles re loin, maintien d'un FPS élevé même pour de grandes distances de vue reddit.com .

Technique d'optimisation	Description	Impact sur performances
Atlas de textures/Matériau unique	Unifier les textures de blocs (atlas ou Texture Array) sous un seul shader.	Réduction importante des draw calls (1 par c lieu d'une par matériau). Meilleure utilisation **Batcher** d'Unity.
Job System + Burst (multi-threading)	Offloader la génération de terrain/mesh sur des jobs C# compilés en natif.	Gros gain CPU (jusqu'à x10 sur les temps de chunks) reddit.com , ce qui élimine les saccades modifications/chargements.
Compute shaders ciblés	Utiliser le GPU pour des calculs massifs parallèles (ex: bruit 3D, filtres) sans retour CPU.	Peut accélérer certains calculs gourmands (G CPU), mais **faible impact si readback CPU né** (ralentissement) reddit.com .
Frustum & Occlusion culling	Ne pas dessiner les chunks hors caméra ou occultés derrière d'autres.	Évite de faire travailler CPU/GPU sur des obje invisibles. Gain élevé en scènes denses (jusqu dizaines de % de tris en moins) medium.com .
Streaming de chunks (load/unload)	Charger en mémoire uniquement les chunks proches du joueur, décharger les lointains.	Empêche l'explosion du nombre d'objets ren l'utilisation mémoire. Indispensable pour gar FPS stable en monde ouvert.
Pooling d'objets	Réutiliser les GameObjects/Mesh des chunks au lieu de détruire/créer constamment.	Réduit les pics de lag dus au Garbage Collect allocations. Améliore la **fluidité** (micro-stutte moins lors du streaming) medium.com .
Floating origin (origine flottante)	Recentrer régulièrement l'origine du monde autour du joueur pour limiter les coordonnées élevées.	Assure la stabilité numérique et la précision d rendu/physique sur de très grandes distance direct sur FPS négligeable, mais évite des art pourraient forcer à baisser la qualité.
Ajustements HDRP (qualité)	Diminuer ou désactiver les features lourdes (ombres lointaines, volumétriques, post-process) sur configurations modestes.	Peut **grandement améliorer le framerate** en réduisant la charge GPU (plusieurs ms gagné frame). Ex: post-process optimisés → +30% F discussions.unity.com .
Résolution dynamique / Upscaling (DLSS/FSR)	Rendre à résolution inférieure puis upscaler via algorithme AI ou filtre.	**Augmentation massive du FPS** en cas de bin Ex: rendu à 50% + FSR peut presque doubler framerate discussions.unity.com , au prix d'une légèr de netteté.

Sources : Optimisations tirées de forums Unity, Reddit (*VoxelGameDev*), documentations Unity et retours d'expérience de développeurs reddit.com reddit.com reddit.com , etc. Les références citées en indice 【】 renvoient aux ressources détaillées correspondantes. En combinant ces techniques, vous maximiserez les performances de votre jeu voxel sous Unity HDRP tout en conservant une qualité graphique élevée.

Citations

How to handle massive numbers of chunks : r/VoxelGameDev
https://www.reddit.com/r/VoxelGameDev/comments/ss7cr4/how_to_handle_massive_numbers_of_chunks

How to handle massive numbers of chunks : r/VoxelGameDev
https://www.reddit.com/r/VoxelGameDev/comments/ss7cr4/how_to_handle_massive_numbers_of_chunks

How to handle massive numbers of chunks : r/VoxelGameDev
https://www.reddit.com/r/VoxelGameDev/comments/ss7cr4/how_to_handle_massive_numbers_of_chunks

How to handle massive numbers of chunks : r/VoxelGameDev
https://www.reddit.com/r/VoxelGameDev/comments/ss7cr4/how_to_handle_massive_numbers_of_chunks

Building a High-Performance Voxel Engine in Unity: A Step-by-Step Guide Part 2: M...
https://medium.com/@adamy1558/building-a-high-performance-voxel-engine-in-unity-a-step-by-step-guide-part-2-mesh-generation-bcf1401a5b4b

Building a High-Performance Voxel Engine in Unity: A Step-by-Step Guide Part 2: M...
https://medium.com/@adamy1558/building-a-high-performance-voxel-engine-in-unity-a-step-by-step-guide-part-2-mesh-generation-bcf1401a5b4b

Consume everything - how greedy meshing works - Community Tutorials - Develop...
https://devforum.roblox.com/t/consume-everything-how-greedy-meshing-works/452717

Voxel game optimizations? : r/VoxelGameDev
https://www.reddit.com/r/VoxelGameDev/comments/1f93uke/voxel_game_optimizations/

Voxel game optimizations? : r/VoxelGameDev
https://www.reddit.com/r/VoxelGameDev/comments/1f93uke/voxel_game_optimizations/

How to handle massive numbers of chunks : r/VoxelGameDev
https://www.reddit.com/r/VoxelGameDev/comments/ss7cr4/how_to_handle_massive_numbers_of_chunks

How to handle massive numbers of chunks : r/VoxelGameDev
https://www.reddit.com/r/VoxelGameDev/comments/ss7cr4/how_to_handle_massive_numbers_of_chunks

How I Optimize Large Scenes in Unity HDRP Using Three Steps | by Liberty Depriest...
https://medium.com/@youngchae.depriest/how-i-optimize-large-scenes-in-unity-hdrp-using-three-steps-796f75df2f11

How I Optimize Large Scenes in Unity HDRP Using Three Steps | by Liberty Depriest...
https://medium.com/@youngchae.depriest/how-i-optimize-large-scenes-in-unity-hdrp-using-three-steps-796f75df2f11

Voxel game optimizations? : r/VoxelGameDev
https://www.reddit.com/r/VoxelGameDev/comments/1f93uke/voxel_game_optimizations/

Voxel game optimizations? : r/VoxelGameDev
https://www.reddit.com/r/VoxelGameDev/comments/1f93uke/voxel_game_optimizations/

Mega runtime Performance tips thread (unity & HDRP) - Guide to ...
https://discussions.unity.com/t/mega-runtime-performance-tips-thread-unity-hdrp-guide-to-better-runtime-unity-performance/855438

How to optimize graphics performance with HDRP - Unity Discussions
https://discussions.unity.com/t/how-to-optimize-graphics-performance-with-hdrp/318741

Using Unity Jobs & Burst To Improve Performance By 10 TIMES!!!
https://www.reddit.com/r/VoxelGameDev/comments/rm8fv6/using_unity_jobs_burst_to_improve_perform

How to handle massive numbers of chunks : r/VoxelGameDev
https://www.reddit.com/r/VoxelGameDev/comments/ss7cr4/how_to_handle_massive_numbers_of_chunks

Voxel game optimizations? : r/VoxelGameDev
https://www.reddit.com/r/VoxelGameDev/comments/1f93uke/voxel_game_optimizations/

How I Optimize Large Scenes in Unity HDRP Using Three Steps | by Liberty Depriest...
https://medium.com/@youngchae.depriest/how-i-optimize-large-scenes-in-unity-hdrp-using-three-steps-796f75df2f11

So I'm doing an large open world game, how to I seamlessly load ...
https://www.reddit.com/r/Unity3D/comments/sttm6d/so_im_doing_an_large_open_world_game_how_to_

So I'm doing an large open world game, how to I seamlessly load ...
https://www.reddit.com/r/Unity3D/comments/sttm6d/so_im_doing_an_large_open_world_game_how_to_

Building a High-Performance Voxel Engine in Unity: A Step-by-Step Guide Part 5: A...
https://medium.com/@adamy1558/building-a-high-performance-voxel-engine-in-unity-a-step-by-step-guide-part-5-advanced-chunk-7060b4b6275c

Building a High-Performance Voxel Engine in Unity: A Step-by-Step Guide Part 5: A...
https://medium.com/@adamy1558/building-a-high-performance-voxel-engine-in-unity-a-step-by-step-guide-part-5-advanced-chunk-7060b4b6275c

Building a High-Performance Voxel Engine in Unity: A Step-by-Step Guide Part 5: A...
https://medium.com/@adamy1558/building-a-high-performance-voxel-engine-in-unity-a-step-by-step-guide-part-5-advanced-chunk-7060b4b6275c

How to handle massive numbers of chunks : r/VoxelGameDev
https://www.reddit.com/r/VoxelGameDev/comments/ss7cr4/how_to_handle_massive_numbers_of_chunks

How to handle massive numbers of chunks : r/VoxelGameDev
https://www.reddit.com/r/VoxelGameDev/comments/ss7cr4/how_to_handle_massive_numbers_of_chunks

How to handle massive numbers of chunks : r/VoxelGameDev
https://www.reddit.com/r/VoxelGameDev/comments/ss7cr4/how_to_handle_massive_numbers_of_chunks

How to handle massive numbers of chunks : r/VoxelGameDev
https://www.reddit.com/r/VoxelGameDev/comments/ss7cr4/how_to_handle_massive_numbers_of_chunks

HDRP roadmap in 2024 - Unity Discussions
https://discussions.unity.com/t/hdrp-roadmap-in-2024/929302

What is DOTS good for so far? - Unity Engine
https://discussions.unity.com/t/what-is-dots-good-for-so-far/857098

GitHub - Hertzole/HertzVox-2: An efficient and easy to use voxel framework for Uni...
https://github.com/Hertzole/HertzVox-2

HUGE Performance Spike When Culling (HDRP) Something to do ...
https://discussions.unity.com/t/huge-performance-spike-when-culling-hdrp-something-to-do-with-loading-textures/1545338

Best performance in HDRP, if I don't care about quality, how do I ...
https://discussions.unity.com/t/best-performance-in-hdrp-if-i-dont-care-about-quality-how-do-i-maximize-fps/889587

Voxel game optimizations? : r/VoxelGameDev
https://www.reddit.com/r/VoxelGameDev/comments/1f93uke/voxel_game_optimizations/

Toutes les sources

reddit medium devforum.roblox discussions.unity github