



Guide d'optimisation d'un moteur voxel haute performance sous Unity 6 HDRP

Les jeux **voxel** (à la Minecraft) posent des défis uniques en termes de performance et de rendu visuel. Pour obtenir un monde voxel **fluide** et **visuellement beau** dans Unity (notamment avec HDRP pour un rendu haute fidélité), il faut optimiser tous les aspects : **organisation mémoire**, **gestion des maillages**, **multithreading**, **rendu et éclairage**, **optimisations CPU/GPU**, **physique**, **génération procédurale** et **visibilité**. Voici un guide structuré des meilleures pratiques, avec exemples et techniques utilisées par les grands jeux voxel, pour construire un moteur voxel ultra-efficace sous Unity.

Organisation de la mémoire des voxels

Une bonne organisation des données en mémoire est cruciale pour la performance. Chaque voxel contient généralement des informations comme son type (ou matériau), et éventuellement des données supplémentaires (lumière, santé du bloc, etc.). Les structures de données efficaces permettent de **minimiser l'utilisation mémoire** et de **maximiser la localisation des données en cache** (data locality) pour accélérer les accès.

- **Tableaux contigus plutôt que multidimensionnels** : Évitez les tableaux 3D natifs de C# pour stocker les voxels. Il est recommandé d'utiliser un tableau **unidimensionnel** linéaire avec un schéma d'indexation calculé. Un tableau 3D de type `voxel[x][y][z]` entraîne en interne des références multiples peu amies du cache. À la place, un tableau linéaire `voxels[]` de taille `width*height*depth` peut stocker tous les voxels, avec une fonction d'indexation pour convertir une coordonnée 3D en index unique `voxel.wiki`. Par exemple, avec `width`, `height`, `depth` du chunk : `index = x + z*width + y*width*depth` donnera un index unique pour `(x,y,z)` `voxel.wiki`. Ceci garantit des données contiguës en mémoire, améliorant le parcours séquentiel et le caching CPU.
- **Choix de la taille de chunk** : Un **chunk** est un groupe de voxels (souvent cubique, e.g. $16 \times 16 \times 16$) traité comme une unité. Utiliser des dimensions de chunk en **puissance de deux** est courant, car cela s'aligne bien avec d'éventuels systèmes de LOD ou octrees. Par exemple, 16 ou 32 voxels de côté. Un chunk 16^3 contient 4096 voxels. Garder la taille modérée (16 ou 32) équilibre la granularité : trop grand, les maillages sont lourds à mettre à jour ; trop petit, il y aura trop de chunks à gérer. **7 Days to Die** utilise des chunks de 16m de côté (le monde est divisé en régions de 16×16 blocs). Assurez-vous aussi de gérer les voxels en bordure de chunk pour éviter les doublons ou incohérences (surtout si vous faites du terrain lissé nécessitant du « voxel supplémentaire » pour les normales).
- **Structures compactes et bitmasks** : Stockez les informations de voxel de façon compacte. Par exemple, si vous n'avez qu'une poignée de types de blocs, un seul octet peut suffire pour le type. Certains moteurs stockent un voxel en 2–3 octets seulement (ex. type de bloc, et quelques bits pour l'éclairage ou d'autres propriétés). Utiliser des **bitmasks** et des **énumérations en byte** permet de représenter un voxel sur peu de mémoire, réduisant la cache miss et permettant d'en charger plus d'un coup. Un chunk de 16^3 avec 3 octets par voxel fait ~12 Ko, ce qui est très léger en RAM. On peut regrouper ces données dans des **struct C# non managées (blittable)** pour utilisation avec le Job System et Burst.

- **Gestion du vide et des zones sparses** : Si votre monde est très vaste et majoritairement vide (air), envisagez des structures **sparses** pour stocker les voxels. Des octrees creux (Sparse Voxel Octree, SVO) ou des *grilles multi-niveaux* peuvent accélérer l'accès aux voxels en évitant de stocker chaque voxel vide individuellement. Par exemple, un SVO alloue dynamiquement des nœuds seulement où la matière est présente, ce qui réduit la mémoire pour des caves ou un ciel ouvert. Cependant, ce genre de structure est plus complexe à mettre en œuvre, surtout dans Unity (il faut alors un bon soutien de l'allocation mémoire natif et du Job System pour traverser ces structures). Pour débuter, la méthode du **chunk statique** (tableau plein) est plus simple et souvent suffisante, surtout avec les techniques de chargement/déchargement de chunks vus plus loin.

Exemple d'implémentation – Struct de voxel et indexation : vous pouvez définir un struct voxel minimal et une fonction d'indexation comme suit :

csharp

Copier

```
public struct Voxel {  
    public byte type;          // Type ou matériau du voxel  
    public byte light;         // Niveau de Lumière (0-15 par ex, si nécessaire)  
    public byte data;          // Autres données (ex: durabilité, variante)  
}  
  
// dimensions du chunk  
int width = 16, height = 16, depth = 16;  
Voxel[] voxels = new Voxel[width * height * depth];  
  
// Fonction d'indexation  
int Idx(int x, int y, int z) => x + z * width + y * width * depth;  
  
// Lecture/écriture  
Voxel v = voxels[Idx(x,y,z)];  
voxels[Idx(x,y,z)] = new Voxel { type = 1, light = 15, data = 0 };
```

Cette organisation contiguë est **cache-friendly** et s'intègre bien avec le Burst (on peut convertir voxels en NativeArray<Voxel> pour les jobs).

Gestion des maillages de voxels

L'optimisation du **maillage (mesh)** qui représente visuellement les voxels est l'élément central de la performance d'un moteur voxel. Plutôt que d'avoir un cube **séparé** pour chaque voxel (ce qui serait désastreux en draw calls et polycount), on génère un maillage combiné par chunk. Voici les techniques clés :

- **Culling des faces cachées** : Ne pas générer les faces des cubes qui sont en contact avec d'autres cubes solides. Dans un environnement voxel dense, la plupart des faces sont complètement cachées par des voisines et n'ont pas besoin d'être rendues. On parcourt chaque voxel du chunk, et pour chaque face (haut, bas, gauche, droite, avant, arrière), on vérifie si le voxel voisin dans cette direction est *air* (ou en dehors du monde). **Si oui**, on ajoute la face au maillage, **sinon** on la skip. Cela réduit radicalement le nombre de polygones. Dans un volume massif comme 16^3 voxels, sans culling on aurait $4096 \text{ cubes} \times 6 \text{ faces} = 24576 \text{ faces}$. Avec culling, une scène naturelle (terrain) n'en a qu'une fraction (seulement la « peau » de la forme). *Exemple* : Ce principe est utilisé dans Minecraft et tous les moteurs voxels : seules les faces exposées à l'air ou à la frontière d'un chunk sont émises.
- **Greedy meshing (maillage glouton)** : Technique avancée consistant à fusionner plusieurs faces adjacentes coplanaires en un seul grand quadrilatère. Plutôt que de dessiner chaque face de 1×1 unité séparément, on cherche les *rectangles* de voxels exposés qui se touchent et on les combine. Par exemple, une grande paroi plate de 16×16 voxels sans interruptions peut devenir 2 grands polygones (un pour chaque côté) au lieu de 256 petits. Le greedy meshing peut réduire drastiquement le nombre de vertices et triangles nécessaires. Moins de triangles = **moins de charge GPU** et **moins de draw calls**. C'est une optimisation majeure adoptée dans de nombreux moteurs (il existe des implémentations publiques, ex: algorithme de Mikola Lysenko de 0fps). **Attention** toutefois à bien gérer les *T-junctions* (points où des grands polygones en croisent d'autres plus petits) – ces artefacts peuvent apparaître, mais en pratique ils n'affectent pas beaucoup le rendu [medium.com](#) (Minecraft s'en accommode très bien). Résultat : des mondes visuellement identiques, mais avec un maillage beaucoup plus **léger** à dessiner.

- **Fusion de maillages et draw calls** : Unity supporte la fusion de maillages via `Mesh.CombineMeshes`. Dans un jeu voxel, le plus simple est de construire directement un maillage unique par chunk regroupant toutes les faces visibles. Ainsi, chaque chunk est dessiné en 1 *draw call*. Veillez à n'utiliser qu'un seul matériau par chunk (par exemple, un atlas de textures pour tous les types de blocs) afin que tout le chunk soit rendu en une passe. Si vous avez plusieurs matériaux (par ex. blocs translucides vs opaques), Unity devrait faire un second dessin du même mesh pour les parties avec un autre matériau. Préférez donc regrouper les voxels par propriétés pour minimiser les passes.
- Astuce* : Certaines implémentations utilisent plusieurs sous-maillages (`Mesh SubMesh`) pour séparer par matériau, tout en gardant un `GameObject` par chunk. À évaluer selon vos besoins (eau transparente, etc.).
- **Streaming et mise à jour du mesh** : Construisez vos maillages de chunk de façon dynamique à l'exécution. Lorsqu'un chunk est généré ou modifié, vous devez recalculer son mesh (et possiblement ceux des chunks voisins affectés sur la bordure). Cette opération doit idéalement être faite hors du thread principal (voir section multithreading) pour éviter les saccades. Une fois le mesh calculé (listes de vertices, d'UV, de triangles), on l'applique sur un `MeshFilter/MeshRenderer` du chunk. Unity permet la mise à jour de mesh à la volée via `mesh.Clear()` puis assignation des nouvelles données.
- Pour les **modifications localisées** (ex: le joueur casse un bloc), une stratégie consiste à ne recalculer que le chunk courant et ses voisins immédiats (puisque casser un bloc expose possiblement des faces dans les chunks adjacents). Avec des chunks de 16^3 , recalculer ~4096 voxels est très rapide, surtout en C# bursted. Évitez de recalculer *tous* les chunks à chaque modification, ce n'est pas nécessaire.
- **Exemple de boucle de meshing** : pour chaque voxel actif du chunk, on teste ses 6 voisines. En pseudo-code:

csharp

Copier

```

for x,y,z in chunk:
    if (!voxel[x,y,z].isActive) continue;
    if (voxel[x,y,z+1] est Air) AddFace(Direction.Front, x,y,z);
    if (voxel[x,y,z-1] est Air) AddFace(Direction.Back, x,y,z);
    if (voxel[x+1,y,z] est Air) AddFace(Direction.Right, x,y,z);
    ... (Left, Up, Down)
  
```

La fonction `AddFace` va ajouter 4 vertices et 2 triangles (ou moins si on fusionne avec greedy meshing) pour la face spécifiée, en assignant les UV appropriés (vers l'atlas de texture du bloc, par ex).

En appliquant ces techniques, **notre moteur ne construit que la géométrie strictement nécessaire**. Par exemple, un test montre qu'on peut rendre de **vaste mondes** voxelisés beaucoup plus efficacement : « *ne générer les faces de mesh que pour les voxels exposés à l'air ou en bord de chunk réduit significativement le nombre de faces à rendre* ». De plus, le greedy meshing aide à **diminuer les appels de rendu** (draw calls) qui sont un goulot d'étranglement courant.

Multithreading et jobs pour les voxels

Tirer parti de tous les cœurs CPU est indispensable pour un jeu voxel qui génère du terrain, calcule des maillages et met à jour de nombreux objets. Unity offre deux approches principales : les **C# Jobs (DOTS)** et les threads/classiques (Tasks .NET). La combinaison C# Jobs + Burst est particulièrement puissante dans Unity 6:

- **Unity Job System + Burst Compiler** : C'est l'approche recommandée pour paralléliser les tâches lourdes. Le Job System permet de programmer des jobs (structures implémentant `IJob`, `IJobParallelFor`, etc.) s'exécutant sur des **threads de travail** gérés par Unity. Le **Burst Compiler** vient *transcompiler* ce code C# en code natif ultra-optimisé (LLVM) en respectant certaines contraintes. Résultat : des gains de performance souvent ×10 (voire ×100) par rapport à du C# normal [medium.com](#). Comme le dit un expert : « *Burst est un nouveau compilateur qui peut massivement optimiser le code... Combinés (jobs + Burst) ils accélèrent le code de façon incroyable* » [medium.com](#). En pratique, cela permet de calculer des chunks en parallèle sans bloquer le thread principal, amortissant le coût sur plusieurs cœurs.
- **Travaux adaptés au Job System** : Dans un moteur voxel, plusieurs tâches sont hautement parallélisables :

- *Génération de hauteur/bruit* : calculer le type de chaque voxel via du *noise* (Perlin/Simplex) peut être fait en parallèle pour tous les voxels d'un chunk. On peut lancer un job `IJobParallelFor` qui assigne un type (air, terre, pierre...) à chaque position. Par exemple, le job parcourt une plage d'index sur un `NativeArray<Voxel>` et remplit les données selon la fonction de terrain.
- *Meshing* : la génération de la liste de triangles/vertices d'un chunk peut aussi être faite en job. On peut même découper le travail par face ou par portion du chunk (`IJobParallelFor` sur chaque colonne par exemple). Toutefois, la construction d'un mesh complet implique d'accumuler des listes – il faut gérer la synchronisation ou utiliser des structures concurrentes (comme `NativeList`) compatibles avec Burst.
- *Physique simplifiée* : détection des voisins pour le support des blocs (gravité) ou propagation de lumière peut aussi être distribuée.

L'idéal est de chaîner les jobs : exécuter un job de *voxel generation* (bruit) puis, une fois terminé, un job de *meshing* pour ce chunk, etc. Unity permet de chaîner via les dépendances de JobHandles.

- **Exemple de job Burst – Génération de voxels par bruit :**

csharp

Copier

```
[BurstCompile]
struct VoxelGenerationJob : IJobParallelFor {
    public NativeArray<Voxel> voxels;
    [ReadOnly] public int chunkSize;
    [ReadOnly] public float noiseScale;
    [ReadOnly] public float baseHeight;
    [ReadOnly] public float3 chunkWorldPos; // position monde du chunk
    public void Execute(int index) {
        int3 pos = IndexToXYZ(index, chunkSize);
        float worldX = chunkWorldPos.x + pos.x;
        float worldZ = chunkWorldPos.z + pos.z;
        // Générer hauteur via bruit (ex: Perlin)
        float heightValue = noise.snoise(new float2(worldX, worldZ) * noiseScale);
        int terrainHeight = (int)math.floor(heightValue * baseHeight);
        byte type = (pos.y <= terrainHeight) ? (byte)1 : (byte)0; // 1 = air, 0 = water
        voxels[index] = new Voxel { type = type };
    }
}
```

```
// Programmation du job:  
var job = new VoxelGenerationJob { voxels = voxArr, chunkSize=16, noiseScale = 0.01 };  
JobHandle handle = job.Schedule(voxArr.Length, 64);  
// ... plus tard:  
handle.Complete();
```

Ici, le job calcule pour chaque index si le voxel est de type "sol" ou "air" en fonction d'une hauteur de terrain procédurale. Le tout se fait en parallèle sur N threads et Burst vectorise/optimise le code mathématique. On obtient ainsi un remplissage de chunk très efficace – « *le Job System exécute ce processus en parallèle sans bloquer le thread principal, améliorant significativement la performance pour la génération de terrain à grande échelle* ». ◀

- **Threads classiques et Tâches .NET** : Alternativement, on peut utiliser `System.Threading` (`ThreadPool`) ou les `Task` asynchrones pour paralléliser. Par exemple, lancer un `Task.Run(() => GenerateChunk(chunkCoord))` qui calcule un chunk en arrière-plan, puis envoie les données au thread principal pour créer le mesh Unity. Cette approche fonctionne, mais il faut être très prudent de ne pas toucher aux API Unity en arrière-plan (pas de Mesh ni `GameObject` manipulation hors du main thread). Vous devrez synchroniser les résultats (via `TaskAwaiter` ou des file d'événements). **Unity Jobs** a l'avantage de l'intégration : il gère tout cela et évite les erreurs (pas d'accès interdits car tout paramètre du job doit être natif/blittable). De plus, Burst ne s'applique pas aux threads manuels, ce qui fait qu'ils seront moins rapides en calcul brut. En somme, **Jobs/Burst est généralement supérieur** en performance et sécurité de thread.
- **Burst et Data-Oriented Design** : Pour exploiter pleinement Burst, structurez vos données de façon contiguë (comme vu plus haut) et évitez les appels virtuels ou allocations pendant les jobs. Unity DOTS (ECS) pousse à représenter chaque chunk ou chaque voxel comme des entités et composants, ce qui peut être extrême (1 entité par voxel = des millions, peu pratique sauf pour des démos spécifiques). Une approche raisonnable est *1 entité par chunk*, contenant un composant avec un gros blob de voxels ou un `NativeArray`. Cette approche ECS facilite le scheduling de systèmes (un système pour meshing, un pour génération, etc. qui itèrent sur tous les chunks entités). Toutefois, DOTS/ECS a une courbe d'apprentissage et Unity 6 HDRP classique permet déjà beaucoup avec les jobs sans adopter 100% ECS.

- **Ne pas sur-synchroniser** : Lorsque vous schedulez des jobs, évitez d'appeler `Complete()` tout de suite, sinon vous perdez le bénéfice du parallélisme. Idéalement, lancez un job de génération de chunk, puis laissez-le tourner pendant que le jeu continue (peut-être un frame ou deux), et récupérez le résultat quand nécessaire. Par exemple, vous pouvez lancer les jobs de plusieurs chunks et ne les compléter que quand le joueur s'en rapproche suffisamment. Cette approche pipeline les calculs et maintient la fluidité. Bien entendu, avant d'utiliser les données (ex: assigner le Mesh), il faudra s'assurer du `Complete`.

En résumé, **exploitez le multi-cœur** autant que possible. Un moteur voxel doit constamment calculer ou mettre à jour des données (monde infini, IA, physique...); répartir ces charges sur plusieurs threads est indispensable pour que le **framerate reste stable**. Unity fournit les outils (C# Jobs, Burst) pour écrire du code multi-thread *sûr et ultra-rapide*. Comme le souligne un développeur : « *Unity a des APIs haute performance en C#, en particulier avec Burst et DOTS* », ce qui permet d'éviter de tout coder en C++ natif comme on le ferait dans un moteur custom.

Rendu haute qualité (HDRP/URP) pour voxels

Obtenir un rendu **beau** pour un monde voxel implique de soigner l'éclairage, les matériaux et les effets, tout en restant performant. Unity HDRP offre un pipeline riche pour un rendu PBR (physically based rendering) réaliste, tandis que URP ou Built-in peuvent suffire pour un style plus cartoon ou pixel-art. Voici des techniques pour un rendu de qualité, **adapté aux voxels** :

- **Matériaux et textures** : Utilisez des **matériaux PBR** pour vos voxels afin de bénéficier de l'éclairage HDRP (albédo, métallicité, roughness, normals...). Par exemple, un voxel de pierre peut avoir une texture tileable avec une normal map pour du relief. Il est courant d'utiliser un **atlas de textures** regroupant toutes les faces de blocs (comme Minecraft). Vous pouvez ainsi appliquer une seule matière à tout le chunk, et positionner les UV de chaque face sur la bonne tuile de l'atlas. Sous HDRP, assurez-vous que l'atlas soit importé avec un filtre de mipmaps approprié pour éviter le moiré à distance.

Astuce : vous pouvez augmenter la résolution de textures vu la puissance HDRP, mais attention à la VRAM si votre monde a beaucoup de variétés de blocs.

- **Éclairage global et GI** : Par défaut, un monde voxel dynamique ne peut pas bénéficier du **baking** (pré-calculation) d'éclairage statique. Il faut donc s'appuyer sur l'éclairage temps-réel. HDRP propose le **Screen-Space Global Illumination (SSGI)** et la **Ray-Traced Global Illumination** (si matériel RTX) pour obtenir de l'indirect lighting. Cependant, ces techniques peuvent être coûteuses avec un grand nombre de surfaces planes comme un monde voxel. Une approche classique dans les moteurs voxels est d'implémenter un système de **lumière voxel** simplifié : par exemple, propager une valeur de lumière dans les voxels (comme Minecraft avec ses niveaux de lumière de 0 à 15 pour les torches et le soleil). On peut alors encoder cette lumière dans les sommets du mesh (couleur de sommet) ou via une texture 3D d'éclairage. Cela permet une **illumination globale approximative** mais très performante – Minecraft appelle cela "Smooth Lighting". Cette technique fait une sorte d'**Ambient Occlusion voxel** en dur : chaque coin de face de cube est ombré en fonction des blocs adjacents. Le résultat est un éclairage doux dans les coins, évitant l'aspect trop plat. *Exemple* : Minecraft a introduit l'AO dans son "smooth lighting" pour améliorer la fidélité visuelle, en calculant l'occlusion de chaque coin de cube selon la présence des trois blocs adjacents.
- **Ambient Occlusion et post-traitements** : Même avec un éclairage de base, l'ajout d'un **SSAO** (écran-space ambient occlusion) renforce beaucoup la profondeur visuelle des scènes voxel. Unity HDRP intègre un SSAO efficace. Cela assombrit les recoins et jointures entre blocs pour un rendu plus réaliste des cavités. D'autres post-process HDRP utiles : la **tonemapping** (pour un bel HDR), une légère **bloom** pour les émetteurs de lumière (lave, torches), et le **fog volumétrique** pour les effets de brouillard de distance ou de profondeur dans les grottes. Attention à régler la distance du fog pour cacher l'éventuelle distance de rendu limitée.
Avec HDRP, vous pouvez aussi activer les **Contact Shadows** sur votre lumière principale (soleil) pour que les petits détails, comme les bordures de blocs, projettent des ombres courtes qui ajoutent du relief.

- **Gestion des lumières dynamiques** : Un écueil dans un monde voxel peut être le nombre de **lumières** (par ex., torches placées par le joueur). Sous HDRP (en rendu différé), on peut supporter plus de lumières que sur l'URP forward, mais il y a tout de même un coût par lumière influençant de nombreux objets. Il est souvent préférable de **limiter le rayon** et l'influence des lumières dynamiques et d'utiliser l'éclairage voxel mentionné plus haut. Par exemple, plutôt que de placer une Light Unity pour chaque torche, on peut tricher en augmentant la valeur d'émission du matériau du bloc de torche et en comptant sur le GI temps-réel (SSGI) pour éclairer autour. Si ce n'est pas suffisant, on peut envisager une solution hybride : calculer la propagation de lumière soi-même et injecter un terme d'**émission** dans les matériaux des faces de voxels voisins (imite la diffusion de la lumière). C'est complexe mais certains moteurs voxels custom le font.
- **Transparence et reflets** : Si votre jeu a de l'eau ou du verre, leur rendu peut être coûteux. Préférez un **shader simplifié** pour l'eau voxel (par exemple une simple plane animée pour la surface infinie, plutôt que chaque voxel d'eau). De même, si vous utilisez un **skybox HDRI** sous HDRP, vous obtenez des reflets environnementaux réalistes sur les matériaux brillants (ex: minerai métallique). Vous pouvez ajouter des **Reflection Probes** statiques dans certaines zones (comme l'intérieur de grandes cavernes) pour simuler la GI locale sans calculer du full raytracing.
- **Exemple – Voxel Global Illumination** : Des solutions existent sur l'Asset Store (comme **Lumina GI** pour HDRP) qui utilisent une **voxelisation de la scène** pour calculer une GI temps-réel. Ce principe rapproche du moteur **Teardown** (qui utilise le lancer de rayons sur voxels pour un éclairage incroyablement réaliste). Intégrer un tel système est ambitieux, mais sachez que HDRP offre aussi la possibilité d'activer le **Path Tracing** pour du rendu offline (captures d'écran ultra réalistes, par ex.). Pour le temps réel jouable, un mélange de techniques d'occlusion (AO), de lumières dynamiques limitées, et de l'utilisation maline de l'émission matérielle donnera souvent le meilleur rapport qualité-perf.

En résumé, Unity HDRP peut produire un rendu **triple A** même pour un jeu voxel, à condition d'optimiser les effets. Un **lighting pass** bien pensé (éventuellement en calculant un facteur d'occlusion voxel offline) peut grandement améliorer l'apparence sans coût énorme. N'oubliez pas qu'avec un style voxel, le **style artistique** (couleurs vibrantes, contraste) compte autant que la technologie de rendu. N'hésitez pas à ajuster le **profil post-processing** pour obtenir le rendu souhaité (par exemple, un léger grain pour casser l'aspect trop lisse, ou un Filmic tonemapper pour des couleurs équilibrées).

Optimisations CPU/GPU (Draw calls, Culling, LOD)

Même avec un mesh optimisé et du multithreading, il faut penser globalement aux performances CPU/GPU de la scène. Voici les points à surveiller pour que le moteur reste fluide dans des mondes possiblement infinis.

- **Réduction des draw calls** : Comme évoqué, combinez les voxels en un minimum de maillages. Chaque chunk devrait idéalement être un seul mesh, donc un seul **draw call**. Si vous avez des entités séparées (arbres, objets posés), envisagez de les instancier ou de les fusionner aussi. Gardez un œil sur le compteur de batchs dans le Frame Debugger. Par exemple, 200 chunks visibles = 200 draw calls, ce qui est gérable. Mais 1000 chunks commencerait à être lourd (bien qu'Unity puisse trier par matière efficacement). **Instancing** ne s'applique pas directement aux cubes uniques (car tous positionnés différemment), mais si vous avez des motifs répétés (comme une forêt d'arbres identiques), le GPU Instancing peut aider.
- **Frustum Culling (culling de frustum)** : Unity culle automatiquement les objets dont le bounding box est hors de la vue de la caméra. Assurez-vous que chaque chunk a un **Collider** ou **Renderer.bounds** bien calculé englobant ses voxels, pour que ce culling soit efficace. Ainsi, seuls les chunks dans le cône de vision du joueur seront envoyés au rendu. Cela réduit fortement la charge quand le joueur regarde dans une direction (les chunks derrière lui ou trop loin sur les côtés ne sont pas rendus). Par exemple, sur 200 chunks chargés autour, peut-être seulement 80 sont dans le champ de vision : les autres ne consommeront pas de temps de rendu.

- **Occlusion Culling (culling d'occlusion)** : C'est plus complexe pour un monde destructible, car l'occlusion statique baked d'Unity n'est pas applicable (elle suppose des géométries fixes). Cependant, on peut implémenter un occlusion culling dynamique simplifié. L'idée est de détecter si un chunk est **totalement masqué** par d'autres chunks plus proches de la caméra. Un moteur comme **Voxel Farm** procède en calculant pour chaque chunk des grands polygones occlusifs et en rasterisant ceux-ci dans une petite depth map pour tester la visibilité des chunks lointains. C'est un système pointu, mais qui a montré des excellents résultats (jeu comme Everquest Landmark à l'époque). Pour une approche plus simple : vous pouvez considérer que les voxels forment naturellement de grosses masses opaques ; ainsi, si le joueur est à l'extérieur, **les caves souterraines** (chunks profonds) n'ont pas besoin d'être rendues jusqu'à ce qu'il y entre. On peut donc choisir de **ne pas générer/rendre les chunks entièrement sous la surface** tant qu'il n'y a pas de trou visible vers eux. Ce culling "par hauteur" est imparfait (il suffit d'un trou pour devoir afficher), mais c'est une heuristique. Unity n'aide pas nativement ici, donc ce sera du système "maison".
- **Niveaux de détail (LOD) pour voxels** : Le LOD classique (plus bas poly quand on s'éloigne) est difficile à appliquer aux voxels car la transition se voit fort. Des algorithmes spécialisés comme **Transvoxel** ou **chunked LOD** existent pour le terrain voxel lissé, mais pour du bloc, c'est plus simple : on vise à garder une taille de bloc apparent constante à l'écran. Une technique efficace utilisée par un développeur Unity est de **fusionner les voxels distants en blocs plus gros**. Autrement dit, à une certaine distance, au lieu de représenter chaque cube, on représente des **agrégats** de $2 \times 2 \times 2$ voxels par un seul "cube" dans le mesh distant. Ainsi, la taille en pixels du voxel agrandi reste similaire à celle d'un voxel normal proche. Cela évite d'avoir un maillage ultra détaillé au loin que le GPU de toute façon ne peut pas distinguer précisément. Cette approche LOD se fait au moment du **meshing** : on peut réduire la résolution de la grille pour les chunks lointains (ex: un chunk 16^3 loin pourrait être généré à partir d'un sampling 8^3 , chaque voxel englobant en réalité 2^3 voxels). Une autre méthode plus brute est de désactiver purement les chunks trop lointains (et les remplacer éventuellement par une simple heightmap ou un fond de décor). Mais dans un monde infini, on affiche généralement une distance limitée. **Note:** Les techniques automatiques des moteurs (Nanite d'Unreal, LOD Group Unity) ne s'appliquent pas bien ici, car nos maillages voxels ne sont pas des surfaces continues classiques à décimer – il vaut mieux contrôler nous-même la simplification du terrain.

- **Limite de distance de rendu** : Pour éviter d'afficher des milliers de chunks, on définit souvent une distance max (par ex., afficher les chunks jusqu'à 512 mètres autour du joueur). Au-delà, on peut plonger le monde dans un brouillard épais ou un ciel vide. Minecraft utilise le "fog" pour masquer la coupe franche du rendu lointain. Dans Unity, paramétrez le **Clipping Plane far** de la caméra en conséquence (ou adaptez via un plan de découpe personnalisé en shader pour fondre avec le skybox).
- **Optimisations GPU diverses** : Utilisez des shaders simples pour les blocs opaques (lit shader standard suffit). Évitez les calculs de vertex inutiles en shader – nos meshes voxels ont beaucoup de sommets, donc un vertex shader ultra léger aide. Le fragment shader lui peut être plus lourd si HDRP, mais on peut tolérer grâce à la réduction de faces via greedy meshing. Essayez de maintenir vos **batches** cohérents : tri par matière est naturel si un chunk = un matériau. Donc la charge GPU sera bien occupée à dessiner de gros meshes optimaux plutôt qu'une myriade de petits.

En appliquant ces optimisations, on obtient un rendu très scalable. Par exemple, **Arkenhammer** (développeur d'un jeu voxel Unity) explique qu'ils n'ont pas pu utiliser les LOD automatiques, mais que générer des *blocs plus gros à distance* marche bien mieux pour conserver de bonnes performances visuelles. De même, ils soulignent que beaucoup d'optimisations sont *maison*, en travaillant au plus bas niveau du moteur pour gagner en efficacité. Cela rejoint l'idée qu'un jeu voxel doit souvent traiter la scène différemment d'un jeu standard, et ne pas hésiter à **sortir des sentiers battus** des outils standards (NavMesh, terrains Unity, etc.) pour implémenter du sur-mesure.

Optimisation de la physique et des collisions

La physique dans un monde voxel présente des défis de performance. Un terrain voxel consiste en un très grand nombre de surfaces collidables (chaque face de bloc). Utiliser naïvement le moteur physique d'Unity sur des milliers de colliders peut devenir un goulet d'étranglement. Voici des pratiques pour garder la simulation fluide :

- **Colliders de chunk** : L'approche classique est de générer un **MeshCollider** pour chaque chunk, en utilisant le même maillage que le rendu (ou un maillage collision plus simplifié). Unity pourra alors détecter les collisions du joueur et des objets avec le terrain. Ceci fonctionne, mais **attention** : mettre à jour un **MeshCollider** est coûteux. À chaque fois qu'un chunk est modifié et que son mesh change, Unity doit recalculer l'arbre de collision (BVH) de ce **MeshCollider**, ce qui peut provoquer des pics de lag si fait trop souvent. Pour atténuer cela, vous pouvez :
 - Limiter la fréquence de mise à jour des colliders (par ex., appliquer un délai ou regrouper plusieurs modifications avant de recalculer la collision).
 - Utiliser des **chunks plus petits** pour la collision que pour le rendu, afin de ne recalculer que des zones localisées. Par exemple, un chunk de rendu 16^3 pourrait être découpé en quatre colliders 8^3 .
 - Désactiver complètement la collision des chunks lointains (le joueur ne peut pas y être, donc inutile de les avoir en physique).
- **Collision custom (détéction bloc)** : Certains moteurs choisissent de **ne pas utiliser MeshCollider** du tout. À la place, on effectue la collision par requête manuelle sur la grille voxel. Par exemple, pour le mouvement du joueur, on peut prendre sa position souhaitée, calculer quelles cellules voxels il occupe et voir si l'une est solide. Si oui, on corrige la position (simplement empêcher de rentrer dans le bloc). Cette méthode, utilisée dans Minecraft, est très efficace pour le player controller et quelques entités basiques. Elle évite d'impliquer le solver PhysX d'Unity pour le terrain. De même, un tir de projectile peut être géré par un raycast **voxel** (pas `Physics.Raycast` Unity, mais un parcours Bresenham 3D dans la grille de voxels jusqu'à toucher un bloc). Cela élimine beaucoup de surcharge. **Arkenhammer** indique par exemple avoir carrément abandonné la physics Unity pour le terrain : « *Nous n'utilisons pas la physique Unity parce que générer des colliders pour le terrain est trop coûteux* ». À la place, tout est fait via des calculs custom plus efficents.

- **Rigidbodies et blocs dynamiques** : Si votre jeu a des blocs qui tombent (gravité), réfléchissez-y à deux fois. Simuler des centaines de blocs en chute libre avec des rigidbodies peut saturer PhysX. Souvent, on "triche" : on détache un bloc, on le passe en objet physique *temporairement* pour l'effet, puis on le détruit après quelques secondes ou on le fige. Teardown excelle là-dessus en autorisant la destruction mais en **agrémentant** les débris (des milliers de petits voxels qui tombent sont groupés en fragments convexes plus gros pour la simulation). Dans Unity, vous pouvez utiliser les **Rigidbodies composites** ou **articulations** pour lier des blocs ensemble quand ils tombent, afin de réduire le compte d'objets.
- **Navigation IA** : Unity NavMesh ne gère pas nativement les terrains modifiables voxel. Il faudrait recarver le navmesh à chaque changement, ce qui est impraticable en temps réel. Deux options :
 1. **Navmesh statique + adaptation** : si le terrain de base est fixe (par ex, seulement de petites constructions sont modifiées), on peut pré-baker le navmesh sur le terrain initial puis avoir des agents capables de sauter ou s'adapter aux changements mineurs.
 2. **Pathfinding grille** : implémenter un pathfinding A* directement sur la grille voxel (ou une grille 2D projetée du terrain). Par exemple, pour des ennemis au sol, projeter le voxel world en une grille de marche (sol ou pas sol à chaque x,z), éventuellement avec plusieurs niveaux pour les caves, et utiliser A*. Certes, la grille peut être grande, mais on peut la limiter à une zone autour du joueur et la mettre à jour dynamiquement.
- De nombreux jeux voxels ont leur propre système de pathfinding.
Arkenhammer note avoir un pathfinding custom car le NavMesh Unity ne marchait pas sur leur terrain voxel. Cela leur permet d'intégrer la logique de sauts, d'échelles, d'effondrements de terrain plus facilement que d'essayer de plier le NavMesh Unity à ces besoins.
- **Trigger et détection simples** : Pour certains mécanismes (ex: ramasser un objet dans un bloc), il peut être plus simple d'utiliser un petit Trigger collider sur l'objet ramassable que d'analyser voxel par voxel. Utilisez la physique Unity là où ça a du sens (objets discrets, personnages), mais pas pour le terrain massif.

En bref, la physique Unity peut être utilisée partiellement mais doit souvent être supplée par des méthodes maison pour un résultat performant. L'objectif est de réduire le nombre de colliders actifs et de contourner le recalcul fréquent de MeshColliders. L'expérience de jeux existants montre qu'il faut parfois sortir de PhysX : « *nous avons notre propre pathfinding... Nous n'utilisons pas le navmesh Unity ni sa physique pour le terrain* », ce qui souligne la nécessité de solutions sur mesure. Vous pouvez cependant utiliser la physique Unity pour les personnages et véhicules, en combinant avec des tests voxels pour le contact sol, afin d'obtenir le meilleur des deux mondes.

Gestion de la génération procédurale (chunks dynamiques et mondes infinis)

La plupart des jeux voxels offrent des mondes étendus voire infinis, générés de manière procédurale. Pour que cela tourne en temps réel, l'**optimisation du streaming de monde** est vitale. Il s'agit de charger/décharger les chunks autour du joueur intelligemment, sans pause perceptible.

- **Changement progressif des chunks** : Ne générez pas tout le monde d'un coup au démarrage. Concentrez-vous sur la zone autour du joueur. Définez un **rayon de chargement** (en chunks) autour de la position du joueur, par ex. 10 chunks de rayon (ainsi, 21×21 chunks potentiellement chargés). À mesure que le joueur se déplace, activez les nouveaux chunks qui entrent dans le rayon et détruisez (ou mettez en pool) ceux qui en sortent. Cela requiert un suivi de la **position du chunk courant** du joueur (par ex: `playerChunkX = floor(playerWorldX / chunkSize)`). Ensuite, c'est un simple test des coordonnées de chunk à charger par rapport à une liste de ceux déjà chargés.

- **File d'attente et lazy loading** : Pour éviter les pics de calcul quand le joueur bouge rapidement, utilisez une **file d'attente** de chargement de chunks. Au lieu de tout créer instantanément, on ajoute les positions de chunks à charger dans une queue, et on en traite quelques-uns par frame. Par exemple, on peut décider de charger au maximum 4 chunks par frame et d'espacer ces chargements tous les N frames. *Principe* : « *plutôt que de charger tous les chunks d'un coup, on les distribue sur plusieurs frames pour lisser la charge CPU et éviter les bottlenecks* ». Cette technique de **chargement paresseux** garantit que même si le joueur déclenche le chargement de 100 chunks, ils seront générés progressivement, limitant l'impact sur le framerate instantané. Bien sûr, essayez d'anticiper un peu la direction du joueur pour charger en priorité ce vers quoi il se dirige.
- **Pool d'objets Chunk** : Créer et détruire des GameObjects et composants a un coût non négligeable (allocation, garbage collection). Mieux vaut **réutiliser** les chunks. Implémentez un **Chunk Pool** global : une réserve de X objets chunk préfabriqués prêts à l'emploi. Quand un chunk sort de la zone active, au lieu de `Destroy()`, on le désactive et le remet dans la pool. Inversement, pour charger un chunk, on prend dans la pool un objet libre (ou on en instancie un nouveau si vide). Avant réutilisation, pensez à réinitialiser son état (mesh vidé, colliders remis à zéro, etc.). Un exemple de pool manager crée une dizaine de chunks au départ et les recycle à la demande. Ceci permet d'éviter des allocations constantes et de stabiliser l'utilisation mémoire.
- **Déchargement des chunks lointains** : Mécaniquement, si on charge de nouveaux chunks, il faut *décharger* ceux devenus trop loin. Vous pouvez définir un **rayon de déchargement** légèrement plus grand que le rayon de chargement, pour éviter des chargements/déchargements trop fréquents lorsque le joueur oscille à la frontière. Par ex., charge jusqu'à 10 chunks, décharge au-delà de 12 chunks. Cela crée une petite hystérésis. Le déchargement peut simplement consister à désactiver le GameObject du chunk et/ou le renvoyer au pool, et retirer son entrée de vos dictionnaires de chunks actifs. Si vous avez des données modifiées que vous souhaitez conserver (ex: un joueur a creusé un tunnel), il faudra les stocker (en mémoire ou sur disque) avant de décharger, afin de les recharger plus tard. C'est un autre sujet (sauvegarde) mais crucial pour un monde persistant.

- **Mondes infinis et Floating Origin** : Unity utilise des coordonnées en flottants 32 bits, qui perdent en précision quand on s'éloigne trop de l'origine (0,0,0). Au-delà de ~10 km, on commence à avoir du jitter graphique. Dans un monde potentiellement infini, il faut recourir à la technique du **Floating Origin** : recentrer périodiquement le joueur et les chunks autour de (0,0,0). Concrètement, lorsque le joueur atteint par ex. 1000 unités sur X, on peut soustraire 1000 à toutes les positions (le joueur revient vers 0, et on déplace l'origine du monde). Le joueur ne s'aperçoit de rien, mais on évite les énormes coordonnées. On doit déplacer tous les objets actifs égaux (chunks, entités) en une fois pour que le référentiel reste cohérent. Unity 2021+ et DOTS facilitent cela avec des positions 64-bit ou des systèmes d'origine, mais en pratique, c'est plus simple de faire une translation globale de la scène.
- **Stockage et génération déterministe** : Pour que deux joueurs voient le même terrain ou pour recharger un terrain plus tard, utilisez un **seed** constant pour votre bruit aléatoire. Ainsi, la fonction de génération produira toujours le même chunk donné ses coordonnées. Cela vous évite de stocker chaque voxel. On ne stocke que les modifications apportées par le joueur. Ce concept est ce qui permet à Minecraft d'avoir des mondes de 30 millions de blocs de côté générés à la volée. Si votre jeu a différents biomes ou des structures (villages, etc.), envisagez une génération en plusieurs passes (d'abord le terrain, puis un pass qui place des objets en se basant sur un autre bruit ou sur le terrain généré).

Mise en œuvre – Chargement autour du joueur :

Supposons un rayon de vue de 8 chunks. À chaque frame, on peut faire :

csharp

 Copier

```
Vector3Int playerChunk = PlayerChunkCoord();
if (playerChunk != lastPlayerChunk) {
    // Charger les nouveaux chunks dans le rayon
    for (int dx = -8; dx <= 8; ++dx) {
        for (int dz = -8; dz <= 8; ++dz) {
            Vector3Int cpos = playerChunk + new Vector3Int(dx, 0, dz);
            if (!chunksActive.ContainsKey(cpos)) {
                queueLoad.Enqueue(cpos);
            }
        }
    }
    // Décharger les chunks trop loin
```

```
foreach (var c in chunksActive.Keys) {
    if (math.abs(c.x - playerChunk.x) > 10 || math.abs(c.z - playerChunk.z)
        queueUnload.Enqueue(c);
}
lastPlayerChunk = playerChunk;
}

// Puis, à chaque Update, on traite un peu la queue:
if (Time.frameCount % loadInterval == 0 && queueLoad.Count > 0) {
    for (int i = 0; i < chunksPerFrame && queueLoad.Count > 0; i++) {
        Vector3Int cpos = queueLoad.Dequeue();
        Chunk chunk = chunkPool.Get(); // prendre dans le pool
        chunk.Init(cpos, chunkSize);
        chunksActive[cpos] = chunk;
    }
}
if (Time.frameCount % unloadInterval == 0 && queueUnload.Count > 0) {
    for (int i = 0; i < chunksPerFrameUnload && queueUnload.Count > 0; i++) {
        Vector3Int cpos = queueUnload.Dequeue();
        if (chunksActive.TryGetValue(cpos, out Chunk chunk)) {
            chunkPool.Release(chunk);
            chunksActive.Remove(cpos);
        }
    }
}
```

Cet exemple illustre un mécanisme de **chargement différé** (par intervalle de frames) et de maintien d'un **pool**. L'essentiel est que le travail se fait de façon incrémentale plutôt que tout à la fois. L'utilisation d'une *Queue* permet de prioriser et d'organiser le streaming.

En conclusion, un monde infini doit être géré intelligemment pour ne pas saturer la mémoire et le CPU. En limitant la zone active, en chargeant progressivement et en recyclant les ressources, on obtient un streaming fluide. Les joueurs ne verront alors quasiment jamais de « pop » de terrain, ou de freeze de génération, le monde semblant se dérouler continuellement autour d'eux. C'est ainsi que des titres comme Minecraft, ou les moteurs open-source comme **Manic Digger**, parviennent à gérer d'immenses mondes en temps réel.

Techniques de visibilité et culling spécifiques

Nous avons déjà abordé plusieurs techniques de **visibilité** (face culling, frustum, occlusion). Récapitulons celles-ci et d'autres astuces pour s'assurer que le moteur ne dessine que ce qui est nécessaire :

- **Hidden Face Culling** (élimination des faces cachées) : **Rappel** – ne construire que les faces exposées à l'air ou à la frontière du chunk. C'est le premier niveau de culling, opéré au niveau de la génération du mesh. Cela réduit typiquement l'ordre de grandeur du nombre de polygones.
- **Back-face Culling GPU** : Pensez à définir vos triangles de faces de voxels avec la bonne orientation (sens horaire vs anti-horaire) et à **désactiver le rendu des faces arrière** (back-face). Cela est généralement par défaut actif dans Unity (matériaux opaques) et évite au GPU de dessiner l'intérieur des cubes (même s'ils étaient visibles par transparence). Donc, même pour une face exposée, on ne rendra que sa face avant tournée vers la caméra, pas l'arrière (inutile car toujours occulté par le cube lui-même). C'est un gain classique GPU.
- **Frustum Culling** : Géré par Unity comme mentionné – chaque chunk étant un Renderer séparé, Unity fera le tri pour nous. Assurez-vous d'**échelonner la hauteur des chunks** de sorte qu'ils englobent toute la verticale nécessaire (par exemple, si votre monde fait 256 voxels de haut, vous pouvez découper verticalement aussi, ou encapsuler la hauteur dans le bounds).
- **Distance Culling** : On peut manuellement désactiver les chunks trop loin (même avant la limite de clipping) pour améliorer les perfs, en jouant sur le **Layer Cull Distance** de la caméra par exemple (assigner les chunks lointains à un layer et régler une distance max pour ce layer). Mais cela rejoint l'idée de distance de rendu déjà couverte.

- **Occlusion par voxel** : Cas particulier, **greedy meshing** aide indirectement l'occlusion : en fusionnant les faces, on crée de larges polygones qui peuvent servir d'occludeurs plus efficaces pour Unity. Toutefois, Unity Occlusion Culling (baké) n'est pas utilisable dynamiquement. Si votre jeu est archétypal (genre Minecraft avec de grandes montagnes pleines), il peut être utile de développer un système d'occlusion. Par exemple, le blog ProcWorld illustre un système où pour chaque chunk on trouve de grands rectangles pleins qui peuvent cacher ce qu'il y a derrière. Ces surfaces calculées peuvent ensuite servir à un test de recouvrement de chunks derrière. C'est un système complexe mais potentiellement payant pour de vastes scènes avec beaucoup de voxels invisibles. **Dans un premier temps**, vous pouvez ne pas avoir d'occlusion avancée ; concentrez-vous sur le culling par distance et frustum, déjà très efficaces, surtout couplés à la réduction de polygones.
- **Culling des entités internes** : Si le joueur est en surface, vous pourriez décider de ne pas générer du tout l'intérieur des grottes tant qu'il n'y entre pas. Inversement, si le joueur est sous terre, inutile de conserver les chunks de surface très lointains (on pourrait réduire le rayon de chargement en hauteur). Certains jeux gèrent des "caches" de volume visibles : par exemple, dans Minecraft, l'éclairage global gère les zones non éclairées (souterraines) différemment. Ce n'est pas vraiment du culling, mais ça influence la décision de rendre ou non certaines choses.

En pratique, la combinaison de **faces exposées uniquement** et de **greedy meshing** fait déjà qu'un chunk éloigné composé essentiellement de blocs pleins sera très léger à dessiner, voire un seul quad. Donc même sans occlusion culling sophistiqué, le coût de rendre quelques chunks cachés derrière une montagne reste modeste. Il vaut parfois mieux **ne pas trop complexifier le culling** et accepter de dessiner 10–20% de « trop » que de faire un test coûteux pour les éviter. L'adage en graphisme temps réel est qu'un culling mal dosé peut coûter plus cher que de juste dessiner l'objet. Trouvez le bon équilibre via des profils.

Enfin, une **astuce de confort** pour le joueur : faites en sorte que les transitions de chargement/déchargement soient discrètes. Le culling efficace aide à ça (pas de pop visible sur les côtés), et vous pouvez utiliser un léger fondu d'apparition pour les chunks nouvellement visibles (par exemple augmenter leur alpha graduellement ou un effet de brouillard local) afin de masquer l'apparition soudaine.

Études de cas : moteurs voxels Unity dans les grands jeux

Plusieurs jeux connus ou projets ont implémenté des moteurs voxels performants, chacun avec ses spécificités. S'inspirer de leurs techniques permet de valider les bonnes pratiques :

- **7 Days to Die** – Vaste jeu de survie en voxel sous Unity. Il utilise une approche **mixte bloc et terrain lissé**. Le sol et les montagnes sont lissés (algorithme de type Marching Cubes/Dual Contouring) pour éviter l'aspect trop cubique, tandis que les constructions peuvent être cubiques. Cela nécessite de générer des maillages plus complexes (normales interpolées) et d'introduire des **méta-données de forme** (demi-bloc, pente). Ils ont dû optimiser fortement le code natif, et le jeu souffrait encore de limitations de l'Unity d'alors. Par exemple, pendant longtemps, **7DtD** n'avait pas de distance de vue énorme pour rester jouable. Ils ont amélioré le support du **multithread** dans les dernières alphas, et adopté un système de stabilité physique (effondrement des structures) en parcourant la grille voxels plutôt qu'avec PhysX pur. Bien que détails internes ne soient pas publics, on constate en jeu l'emploi du greedy meshing (peu de surfaces inutiles) et un *level of detail* rudimentaire (les éléments éloignés sont moins détaillés visuellement). Il y a aussi un usage intensif du **Occlusion Culling baked** pour les bâtiments (non-terrain), mais pour le terrain lui-même, c'est géré via la transparence du mesh (les caves non découvertes ne sont pas rendues du tout).
- **Teardown** – Jeu de destruction voxel ultra réaliste (mais **pas sous Unity**, moteur custom en C++). Pourquoi en parler ? Parce qu'il démontre ce qu'on peut faire en poussant les techniques à l'extrême : il utilise la **path tracing** (voxel cone tracing) pour un éclairage global en temps réel (reflets, ombres douces), rendu possible grâce à une **structure hiérarchique (grille mipmap)** sur GPU pour accélérer le lancer de rayons. Teardown simule la physique de milliers de voxels en temps réel en ne rendant dynamiques que les morceaux détachés (sinon tout ce qui est statique est traité comme du décor figé, donc pas de calcul physique inutile). Leur approche montre que si on contrôle entièrement le pipeline, on peut obtenir des graphismes époustouflants dans un monde voxel. Sous Unity HDRP, on ne peut pas (encore) refaire Teardown facilement, mais l'arrivée du ray tracing et de techniques de GI temps réel s'en rapproche. On peut par exemple activer le **Ray Traced Reflections** et **Global Illumination** HDRP pour des cartes graphiques haut de gamme et avoir un rendu voxel similaire (à petit échelle).

- **Vintage Story** – Un jeu voxel “sandbox” issu d’un moteur open-source **Manic Digger** (C# OpenGL). Il n’utilise pas Unity, mais est un excellent exemple d’optimisation : ils ont mis en place un **multithreading massif** (génération de monde, pathfinding, IA sur threads séparés), un **système de rendu par chunks très propre**, et ont ajouté des fonctionnalités comme l’**Ambient Occlusion par voxel** pour le lighting, la gestion des micro-blocs, etc. Fait notable, Vintage Story et même Hytale initialement ont préféré partir d’un moteur custom parce que « *les jeux voxel ont de nombreuses optimisations uniques indisponibles immédiatement dans les moteurs généraux* ». En clair, Unity/Unreal apportent un surcroît d’outils inutiles et manquent de contrôle fin pour les voxels, ce qui a poussé ces devs à tout recoder sur mesure. Cela confirme que pour avoir un moteur voxel ultra-performant, il faut parfois s’écarte des sentiers battus, mais notre chance avec Unity 6 est d’avoir DOTS/Burst qui réduisent cet écart (on peut écrire du code très optimisé en C# natif).
- **Hytale** – En développement (qui est passé de Java à C++), vise à être un “Minecraft++”. Son moteur voxel comportera certainement du **greedy meshing**, un éclairage type Minecraft amélioré, et de nombreux effets (eau, particules) optimisés pour du voxel. Ils mettent l’accent sur le **modding** et la facilité, donc possiblement moins de complexité technique visible côté moddeurs, mais beaucoup de travail en coulisse dans le moteur. On sait qu’Hytale avait démarré sur le code de Manic Digger comme Vintage Story, preuve que ce vieux code open-source contenait de bonnes bases (mais le fait qu’ils le réécrivent en C++ montre qu’ils veulent plus de performance).
- **Exemple Unity avec Burst** – Un utilisateur Unity (Arkenhammer sur Reddit) a partagé son expérience d’un prototype voxel sous Unity : il a utilisé énormément d’**API bas niveau** (Mesh API, C# jobs, compute), au point que « *beaucoup de notre code de rendu bas niveau fait ressembler le projet plus à un moteur custom qu’à un jeu Unity conventionnel* ». Il mentionne aussi avoir implémenté son propre **pathfinding** et évité le NavMesh Unity, et ne pas utiliser la physique Unity pour le terrain. Par contre, il profite de la **commodité d’Unity** pour tout le reste (animation, éditeur, etc.). Cela montre qu’on peut allier le meilleur des deux mondes : utiliser Unity comme cadre, mais réinventer certaines roues pour coller aux besoins voxel. Il note enfin que « *peu importe le moteur, beaucoup de choses seront custom de toute façon, l’avantage d’Unity c’est de pouvoir le faire en C# haute performance plutôt qu’en C++* ».

En s'inspirant de ces exemples, on retient que **les fondamentaux** (chunks, greedy meshing, culling, multithreading) sont omniprésents. Chaque projet ajoute sa touche : éclairage sophistiqué pour l'un, voxel lissé pour l'autre, gameplay de destruction pour un troisième. À vous de voir les besoins de votre jeu et d'adapter ces techniques. Mais ce qui est certain, c'est que **la performance naît de la combinaison** de toutes ces optimisations à chaque niveau du moteur.

Bonnes pratiques de développement et structure de projet

Pour terminer, voici quelques **bonnes pratiques générales** lors de la construction de votre moteur voxel sous Unity :

- **Organisation du code** : Séparez clairement les systèmes – par exemple, un module **WorldManager** qui gère les chunks chargés, un module **ChunkGenerator** (peut-être en singleton ou monobehaviour central) qui lance les jobs de génération/meshing, un **ChunkPool** pour la réutilisation, etc. Une architecture claire vous aidera à intégrer progressivement chaque optimisation (il vaut mieux dès le début penser asynchrone et chunk-based, même si on n'implémente pas tout de suite le multithreading).
- **Tests de performance réguliers** : Profilez fréquemment en augmentant l'échelle (plus grande distance de vue, plus de modifications) pour traquer les goulets d'étranglement. Par exemple, testez le coût de génération d'un chunk complet avec votre code Burst et assurez-vous qu'il est acceptable en le comparant à Time.maximumDeltaTime visé.
- **Gestion mémoire** : Faites attention aux allocations : utilisez des **NativeArrays** réutilisables pour les jobs, ou des pools d'objets. Les mondes voxels tendent à générer beaucoup de garbage (surtout si on n'y prend pas garde avec des listes/linq etc.), ce qui cause des GC spikes. Privilégiez les conteneurs natifs (NativeList, etc.) ou les structures fixed-size.
- **Taille du build** : Un jeu voxel peut avoir beaucoup d'assets (textures d'atlas, modèles de végétation, etc.). Utilisez les outils d'Unity pour optimiser la taille et le temps de chargement : compressions de texture, Addressables si besoin pour charger certaines ressources à la demande (par ex, biomes rares).

- **Évolutivité** : Essayez dès le départ d'imaginer comment vous ajouterez de nouvelles fonctionnalités (biomes, entités, eau, etc.) sans casser l'optimisation. Par exemple, pour l'eau : prévoyez un type de voxel "eau" qui ne soit pas rendu dans le même mesh que le terrain (car semi-transparent nécessite une passe séparée). Peut-être gérez un mesh distinct par chunk pour les voxels liquides, et triés à part. Ce genre de décision doit être réfléchi tôt pour éviter de grosses refontes.
- **Utiliser le Burst même hors jobs** : Parfois, certaines fonctions utilitaires peuvent être décorées de `[BurstCompile]` et appelées depuis un job ou via `Unity.Burst.BurstCompiler.CompileFunctionPointer` pour gagner en vitesse. Par exemple, une fonction de bruit Perlin custom peut être burstiée.
- **Limiter le debug en mode play** : Les Gizmos de debug (afficher chaque cube par un Gizmo cube) sont très utiles en phase initiale pour valider le chunking, mais désactivez-les ensuite car ils ralentissent beaucoup l'éditeur. Prévoyez des booléens ou un niveau de log paramétrable pour activer/désactiver le debug visuel sans enlever le code.

En suivant ce guide et ces conseils, vous êtes armé pour construire un moteur voxel sur Unity performant et visuellement attrayant. Chaque aspect du moteur – des bits en mémoire jusqu'aux pixels à l'écran – doit être pensé pour le **contexte voxel**. C'est un travail d'optimisation global passionnant, qui vous permettra de créer des mondes riches où performance et beauté cohabitent. Avec Unity 6 HDRP, vous disposez d'outils puissants (Jobs, Burst, rendu physique) pour y parvenir, comme l'ont montré d'autres projets ambitieux. **Bonne construction de monde voxel !**

Sources : Les conseils compilés ci-dessus s'appuient sur les retours d'expériences et analyses de multiples développeurs et projets voxels, notamment les tutoriels de CodeComplex, des discussions techniques sur Reddit (ex. Arkenhammer) et les articles de référence sur les moteurs type Minecraft. Ces références soulignent l'importance d'optimisations telles que le greedy meshing, le multithreading Unity, ou encore des LODs adaptés aux voxels, qui ont été intégrées aux recommandations de ce guide.

Citations



Data Structures - Voxel.Wiki

<https://voxel.wiki/wiki/datastructures/>



Building a High-Performance Voxel Engine in Unity: A Step-by-Step Guide Pa...

<https://medium.com/@adamy1558/building-a-high-performance-voxel-engine-in-unity-a-step-by-step-guide-part-2-mesh-generation-bcf1401a5b4b>

Ⓜ Intro to Jobs/Burst/DoD. I often talk about performance gains I... | by Jason ...

https://medium.com/@jasonbooth_86226/intro-to-jobs-burst-dod-66c6b81c017f

Toutes les sources



voxel



medium