Voxel World Roadmap — Unreal Engine 5.6

This roadmap breaks the voxel world project into phased milestones. Each phase includes: Goal, Implementation summary, Acceptance criteria / End result, Tasks, Risks / Notes, and Quick checklist you must satisfy before moving to the next phase. We will only move on when the current phase is 100% functional.

Engine: Unreal Engine 5.6 (C++ primary; Blueprints where convenient). Use only 100% free plugins. Prefer in-house C++ code for all world generation.

Key Features:

- World generation created on a chosen world size (small, medium, large)
- Chunk saving/loading so players can traverse large worlds without overhead
- Saving so if the world is altered in any way, the world can be exited and loaded with the saved changes
- Unique biomes (mountainous, flat, heavy-tree spawns, lake area, etc)

High Level Architecture (modules)

1. Core / Types
    a. FBlockType / EBlockID (small integer ID, palette-driven)
    b. Block properties: isSolid, isTransparent, lightOpacity, materialIndex, any extra metadata
2. Chunk data
    a. FChunkKey (chunk X/Z, maybe Y if using vertical chunking)
    b. FVoxelChunkData (raw block storage, biome info, heightmap cache, modification delta map)
3. World Manager
    a. Manages which chunks are loaded, streaming radius, LRU cache, player-centered loading/unloading
4. Generator
    a. Deterministic generator that given seed + chunk coords produces the base chunk contents (terrain + biome)
    b. Starts with a naive visible-face approach, possibly upgrade to greedy meshing for performance
5. Mesher
    a. Takes FVoxelChunkData and produces combined mesh for a chunk (vertices, indices, uvs, normals, collision)
    b. Starts with a naive visible-face approach, possibly upgrade to greedy meshing for performance
6. Renderer / Mesh Component
    a. Uses a runtime generated mesh component. Options are Unreal's UProceduralMeshComponent or a better-performing RuntimeMeshComponent which has been widely used for voxel work.
7. Storage / IO

    a. Per-chunk saving (only store diffs/modified blocks, compression, versioning, async IO worker threads
8. Tools / Editor
    a. Debug drawing, save inspection, chunk viewer, generator parameter UI
9. (Future) Multiplayer
    a. Server authoritative chunk changes, delta replication to clients. Design save format and replication-friendly change logs from day one.

Recommended baseline technical choices (subject to change):

- **Chunk size (recommended start):** 16 × 16 × 128 (X × Y × Z). This gives a good balance between mesh granularity and performance.
- **Block storage:** uint8 block IDs (change to uint16 if you need >255 block types).
- **Default/generated terrain** is implicit; only deltas (modified blocks) are saved. - Meshing approach: Start with a naive face-culling mesher, then possibly implement greedy meshing to drastically reduce triangles.
- **Noise:** Use FastNoiseLite (free, permissive license) for deterministic noise generation.
- **Mesh component:** Start with UProceduralMeshComponent (built-in) for prototyping, switch to a free community Runtime Mesh Component if necessary for performance.
- **IO / threading:** All heavy generation and disk IO must be async/background; mesh creation and actor changes must be on game thread.

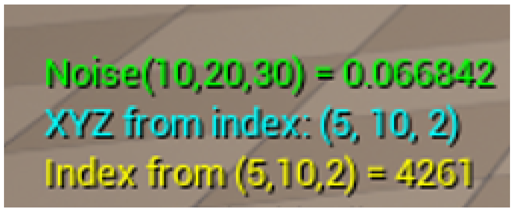
Phase 1 - Project & Tooling Setup (Foundations)

Goal: Create a clean C++ plugin/module skeleton, integrate FastNoiseLite, establish basic block types and helper utilities, and confirm build process for UE5.6.

Implementation: Create a new C++ plugin named **VoxelCore.** Add a **ThirdParty/FastNoiseLite** folder. Implement core types: **EBlockId, FBlockInfo, block palette, block indexing helpers IndexFromXYZ / XYZFromIndex, FChunkKey, struct with hashing.** Add a **Config** header with constants: **Chunk_X = 16, Chunk_Y = 128, Chunk_Z = 16,** seed configuration, default material atlas slot indices.

End Goal: Project compiles cleanly in UE5.6 (Win+Editor) - plugin/module is visible in Editor and loads with project. A simple unit-test C++ command **Voxel.TestSetup** chunk indexing tests and returns deterministic FastNoiseLite output for a few coordinates.

End Result (Voxel.TestSetup console command):



```
Noise(10,20,30) = 0.066842
XYZ from index: (5, 10, 2)
Index from (5,10,2) = 4261
```

The end result shows that the chunk index math is working (XYZ -> Index and back give sane results). FastNoiseLite is integrated properly (deterministic noise values). The console command system is working. This marks the end of Phase 1's acceptance criteria.

Phase 2 - Chunk Data Model & Deterministic Generator

Goal: Implement in-memory chunk data structure, deterministic base generator (seeded), and a debug visualizer that can print chunk heightmap and biome info to log.

Implementation: Implement **FVoxelChunkData** with **TArray<uint8>** blocks and **TMap<int32, uint16>** Modified Blocks for deltas. Implement **FVoxelGenerator** that uses FastNoiseLite to produce a base chunk from **FChunkKey + seed.** Start with a basic heightmap terrain (stone below height, dirt/grass top layer, air above). Implement **FVoxelChunk** and memory layout helpers. Implement **FVoxelGenerator::GenerateBaseChunk(const FChunkKey&, FVoxelChunkData&)**. Add debug visualizer and console commands to request generation and print results. Add a lightweight debug actor **AVoxelDebugVisualizer** that can request single-chunk generation and print a human friendly summary (topmost block height per X/Z column, and a small ASCII map) to output log.

End Goal: For a chosen seed and chunk coordinates, repeated runs produce identical chunk block arrays. The debug visualizer prints the expected heightmap and top-block distribution - **ModifiedBlocks** is empty for freshly generated chunks.

End Result (Voxel.GenChunk 0 0 1337), and (Voxel.GenChunk 0 0 12345):

```
LogTemp: === VoxelDebug: Chunk (0,0) Seed=1337 ===
LogTemp: 6666666666666666
LogTemp: 6666666666666677
LogTemp: 6666666666666777
LogTemp: 6666666666667777
LogTemp: 6666666666777777
LogTemp: 6666666667777777
LogTemp: 6666666677777777
LogTemp: 6666666777777777
LogTemp: 6666677777777777
```

```
LogTemp: 6666777777777777
LogTemp: 6667777777777777
LogTemp: 6777777777777777
LogTemp: 7777777777777777
LogTemp: 7777777777777777
LogTemp: 7777777777777777
LogTemp: 7777777777777777
LogTemp: TopBlock 2 (Grass): 256 columns
LogTemp: ModifiedBlocks count (should be zero): 0
LogTemp: === End VoxelDebug ===
LogTemp: === VoxelDebug: Chunk (0,0) Seed=12345 ===
LogTemp: 6666666666666666
LogTemp: 6666666666666677
LogTemp: 6666666666666777
LogTemp: 6666666666667777
LogTemp: 6666666666777777
LogTemp: 6666666667777777
LogTemp: 6666666677777777
LogTemp: 6666666777777777
LogTemp: 6666677777777777
LogTemp: 6666777777777777
LogTemp: 6667777777777777
LogTemp: 6777777777777777
LogTemp: 7777777777777777
LogTemp: 7777777777777777
LogTemp: 7777777777777777
LogTemp: 7777777777777777
LogTemp: TopBlock 2 (Grass): 256 columns
LogTemp: ModifiedBlocks count (should be zero): 0
LogTemp: === End VoxelDebug ===
```

The end result showed a gradient of 6s turning into 7s across the columns where 6 = stone, 7 = dirt, and TopBlock 2 (Grass) confirms the surface cap is still grass. This means noise is being sampled correctly in world space, heights are varying across X/Z, not locked, and column filling logic (stone to dirt to grass) is working as designed. The two seeds ran (1337 and 12345) have nearly identical patterns because the chunk size is small and noise frequency is only 0.01f. Both seeds still drive the noise generator, but the values might be smoothed out at that scale. We will see bigger seed variations once we move to different chunk sizes and increase NoiseFrequency.

Phase 3 - Naive Meshing & Single-Chunk Rendering

Goal: Render one generated chunk in the world using a per-chunk mesh actor. Implement a simple face-culling mesher (no greedy yet).

Implementation: Create a **AVoxelChunkActor** with a **UProceduralMeshComponent.** Implement **FVoxelMesher_Naive** which emits quads only for block faces that have air/transparent neighbors. Generate a single chunk, pass chunk data to the mesher, and create the procedural mesh on the game thread. Hook up simple UVs (texture atlas) and basic materials.

End Goal: A generated chunk is visible in the editor and game view with correct block shapes and textures. Faces between solid blocks are culled (you shouldn't see interior faces).

Texture Atlas Map

A texture map of size 256x64 was created. This is a simple atlas with 4 tiles horizontally. We are currently using 3 of the 4 in this order: Grass, Dirt, Stone. Create a material **M_VoxelAtlas** in the Editor that samples the texture with a default UV. The atlas map lives in **/Game/Textures/M_VoxelAtlas.M_VoxelAtlas.**

End Result (Voxel.SpawnChunk 0 0 1337):

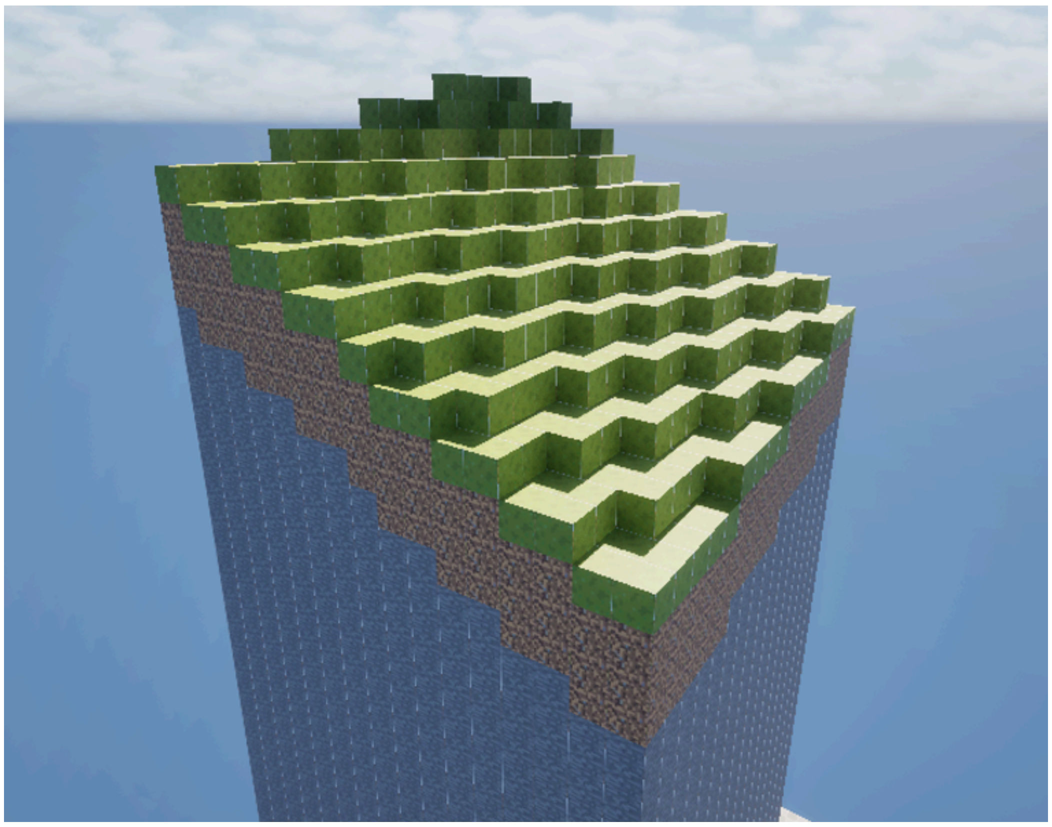
The command successfully spawns a chunk at world origin with visible voxels textured via the atlas map. Faces between solid blocks are culled.

Phase 4 - Chunk Streaming Async Generation (Spawning/Despawning)

Goal: Implement player-centered chunk streaming (load/unload by radius). Chunks load and unload cleanly based on player movement. No visible frame hitches during streaming. Memory usage stays stable regardless of exploration distance.

Implementation: Add **AVoxelWorldManager** to track the pawn, build a nearest-first desired set, and queue thread pool jobs that run **GenerateBaseChunk + FVoxelMesher_Naive** off-thread. Results are drained on the game thread to spawn/update **AVoxelChunkActor::BuildFromBuffers**, while out-of-range chunks are despawned. Uses a pending queue (no stalls), no forced actor names, and exposes radius/seed/material as settings.

End Goal: Player movement within the world triggers chunk load/unload with no major frame hitches. Modified blocks persist across sessions: editing a block, quitting and reloading results in the same modification. Disk size for long play sessions remains reasonable thanks to delta-only saving.

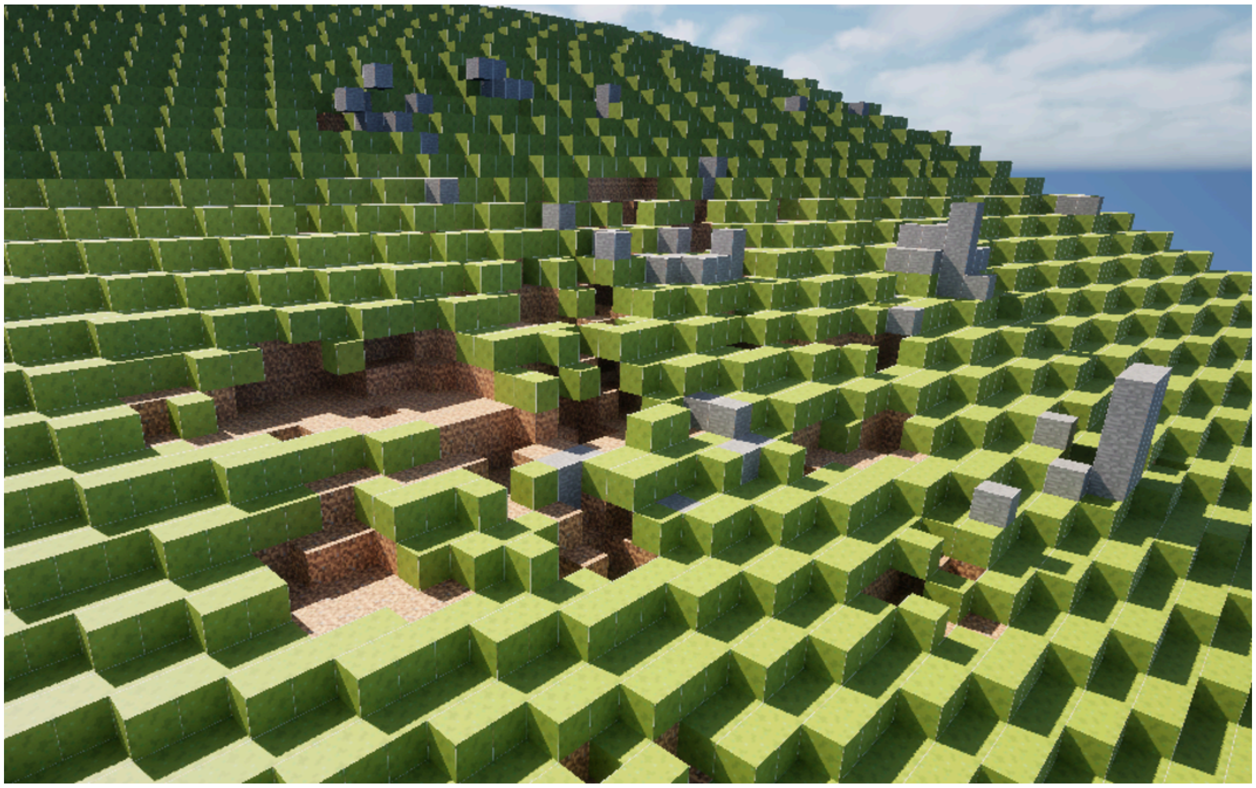


Phase 5 - Altering Chunks (Placing/Destroying)

Goal: Enable deterministic, real-time terrain editing by clicking to remove or place a single voxel. Targeting must be accurate on all faces and chunk seams; edits must persist after unload/save/reload without hitches.

Implementation: Add two tools: **AVoxelRemoveTool (RMB)** and **AVoxelPlaceTool (LMB),** selectable block **PlaceBlockID.** Each tool line-traces from the camera, nudges a hair along/opposite the view ray, then maps that point via **WorldToVoxel_Centered (X to X, Z to Y, Y to Z; centered rounding floor(v/BS + 0.5))**. The manager exposes **RemoveBlock_Local** and **PlaceBlock_Local** to mutate the chunk, mark it dirty and **KickBuild** through the existing async meshing path. No dependence on surface normals; works across chunk borders and negative coords.

End Goal: Clicking a face removes that voxel or places the chosen block precisely where expected. Edits persist after despawn and across sessions (same seed). Streaming remains smooth because rebuild stays on the background queue. In the future, this seed saving will need to save to a map name of the same seed to prevent seeing your world if you load up just the same seed after making a change. Furthermore, this system will need to be moved from actors to **per-player editing component** bound to input and later replicated for multiplayer.



Phase 6 - Chunk Save/Load By Name/Seed

Goal: Implement persistent chunk saving (delta-based) with async disk writes. Create the ability to “Create New World” from UI with a map name and a seed option. Create an option to “Load World” from UI that displays all of your saved map names. When world is created and then you leave the game, that world will be saved to the disk and you can load it from the Load World UI. Ensure that loading worlds specifically happen from world name and not seed.

Implementation: Not established

End Goal: Not established

Phase 7 - Biomes and Extended Terrain Features

Goal: Add biome system (temperature & moisture maps), data-driven biome definitions, and additional terrain features (caves, overhangs, simple caves). Make biome distribution configurable.

Implementation: Add additional noise layers: temperature, moisture, and continent masks. Implement **FBiomeDefinition data** (DataTable or JSON) to specify layer rules: vertical strata, surface blocks, modifiers (e.g., more stone in mountains), spawn weights. - Integrate biome selection during generation; allow seeded biome frequency and biome size parameters. - Add optional cave generation using 3D noise and carve-out rules.

End Goal: Different biomes are visually distinguishable in the world (mountain vs f lat) and spawn according to configured weights. - Caves and overhangs appear in plausible places and do not break chunk meshing or save format

Phase 8 - Polish, Tools & Profiling

Goal: Add developer tools (chunk inspector, save viewer), polish materials & texture atlas, implement optimized collision strategy, and fully profile the world for memory & performance.

Implementation: Implement Editor tools: chunk debugger window, persistent save viewer, generator parameter controls. - Replace temporary materials with a texture atlas + Material Instance setup for chunk materials. - For collision: generate simplified collision geometry (merged convex or simplified height-based collision) where useful to reduce physics cost. - Run Unreal Insights and fix hotspots.

End Goal: Editor tools allow devs to inspect, force-generate, and save/load specific chunks. - Collision performance acceptable for expected player counts in local testing. - Memory budget (document target MB) is consistent with project goals

Phase 9 - Greedy Mesh Rendering If Needed (Optional)

Goal: Implement Greedy Mesh Rendering if Naive mesh rendering is found to have a performance impact.

Phase 10 - Multiplayer Readiness and Replication

Goal: Design and prototype a multiplayer strategy where the server is authoritative for world changes and clients receive only deltas for modified regions.

Implementation (brief): - Design patch/delta operation messages: per-chunk change lists or per-block ops batched into network-safe messages. - Server persists authoritative changes and broadcasts change sets to connected clients (or clients request chunk deltas on load). - Implement client-side prediction for simple interactions (e.g., immediate local block break visuals that get reconciled)

End result (acceptance criteria): - Clear replication API exists and a prototype demonstrates one client breaking a block and the server persisting and informing others.

Save File Format

• Per-chunk binary file: header (magic + version + seed + coords) + compressed payload that contains (localIndex:uint32, blockId:uint16) entries for only modified blocks. Use UE FArchive and FCompression for serialization and compression.

AI Source Code:

Public\ChunkConfig.h

```
#pragma once

// Chunk size config - change these constants project-wide if needed.
constexpr int32 CHUNK_SIZE_X = 16;
constexpr int32 CHUNK_SIZE_Y = 128; // vertical axis
constexpr int32 CHUNK_SIZE_Z = 16;
constexpr int32 CHUNK_VOLUME = CHUNK_SIZE_X * CHUNK_SIZE_Y * CHUNK_SIZE_Z;

constexpr int32 DEFAULT_WORLD_SEED = 1337;
```

Public\ChunkHelpers.h

```
#pragma once
#include "CoreMinimal.h"
#include "ChunkConfig.h"

// Indexing convention:
// X in [0,CHUNK_SIZE_X), Y in [0,CHUNK_SIZE_Y) vertical, Z in [0,CHUNK_SIZE_Z)
// Index = X + Z * CHUNK_SIZE_X + Y * (CHUNK_SIZE_X * CHUNK_SIZE_Z)

inline int32 IndexFromXYZ(int32 X, int32 Y, int32 Z)
{
    return X + Z * CHUNK_SIZE_X + Y * (CHUNK_SIZE_X * CHUNK_SIZE_Z);
}



inline void XYZFromIndex(int32 Index, int32& OutX, int32& OutY, int32& OutZ)
```

```
{
    const int32 XYPlane = CHUNK_SIZE_X * CHUNK_SIZE_Z; // blocks per vertical layer
    OutY = Index / XYPlane;
    int32 Rem = Index - (OutY * XYPlane);
    OutZ = Rem / CHUNK_SIZE_X;
    OutX = Rem - (OutZ * CHUNK_SIZE_X);
}

// Simple chunk key (2D chunks: chunk X and chunk Z)
struct FChunkKey
{
    int32 X;
    int32 Z;


    FChunkKey() : X(0), Z(0) {}
    FChunkKey(int32 InX, int32 InZ) : X(InX), Z(InZ) {}


    bool operator==(const FChunkKey& Other) const
    {
        return X == Other.X && Z == Other.Z;
    }
};


// Hash function so we can use FChunkKey in TMap/TSet
FORCEINLINE uint32 GetTypeHash(const FChunkKey& Key)
{
    // Combine the two 32-bit ints into a hash
    return HashCombine(::GetTypeHash(Key.X), ::GetTypeHash(Key.Z));
}
```

Public\VoxelChunk.h

```
#pragma once

#include "CoreMinimal.h"
#include "ChunkConfig.h"      // CHUNK_SIZE_*
#include "ChunkHelpers.h"     // FChunkKey, IndexFromXYZ(...)
#include "VoxelTypes.h"       // EBlockId

// One chunk�s voxel data: base array + delta overrides (flat index -> id).
struct FVoxelChunkData
{
    // Identity
    FChunkKey Key; // chunk coords (X,Z)

    // Base blocks (generated). Size = CHUNK_VOLUME. Stores EBlockId as uint8.
    TArray<uint8> Blocks;

    // Deltas (persisted): localIndex -> blockId (16-bit for future-proofing).
    TMap<int32, uint16> ModifiedBlocks;

    // Ctors
    FVoxelChunkData() = default;

    explicit FVoxelChunkData(const FChunkKey& InKey)
        : Key(InKey)
    {
        Blocks.SetNumZeroed(CHUNK_VOLUME);      // default Air (0)
        ModifiedBlocks.Empty();
    }

    // Bounds check
    FORCEINLINE bool IsInBounds(int32 X, int32 Y, int32 Z) const
    {
        return X >= 0 && X < CHUNK_SIZE_X
            && Y >= 0 && Y < CHUNK_SIZE_Y
            && Z >= 0 && Z < CHUNK_SIZE_Z;
    }

    // Indexing (delegates to helpers: X + Z*X + Y*(X*Z))
    FORCEINLINE int32 IndexFromXYZLocal(int32 X, int32 Y, int32 Z) const
    {
        return IndexFromXYZ(X, Y, Z);
    }

    // Read with delta fallback
    FORCEINLINE EBlockId GetBlockAt(int32 X, int32 Y, int32 Z) const
    {
        if (!IsInBounds(X, Y, Z)) return EBlockId::Air;
        const int32 Index = IndexFromXYZLocal(X, Y, Z);

        if (const uint16* Ptr = ModifiedBlocks.Find(Index))
        {
            return static_cast<EBlockId>(*Ptr);
        }
        return static_cast<EBlockId>(Blocks.IsValidIndex(Index) ? Blocks[Index] : 0);
    }

    // Write: if equal to base value, drop delta; else store 16-bit delta.
    FORCEINLINE void SetBlockAt(int32 X, int32 Y, int32 Z, EBlockId NewId, bool
bMarkModified = true)
    {
        if (!IsInBounds(X, Y, Z)) return;
        const int32 Index = IndexFromXYZLocal(X, Y, Z);
        if (!Blocks.IsValidIndex(Index)) return;

        const uint8 Raw = static_cast<uint8>(NewId);
        if (bMarkModified)
        {
            // If the new id equals the base Blocks value, remove the delta
            if (Blocks[Index] == Raw)
            {
                ModifiedBlocks.Remove(Index);
            }
            else
            {
                ModifiedBlocks.Add(Index, static_cast<uint16>(Raw));
            }
        }
        else
        {
            Blocks[Index] = Raw;
        }
    }

    FORCEINLINE void ClearDeltas()
    {
        ModifiedBlocks.Empty();
    }
};
```

Public\VoxelChunkActor.h

```
#pragma once
#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "ProceduralMeshComponent.h"
#include "VoxelChunk.h" // <-- add this include for FVoxelChunkData
#include "VoxelChunkActor.generated.h"

UCLASS()
class AVoxelChunkActor : public AActor
{
    GENERATED_BODY()
public:
    AVoxelChunkActor();

    UPROPERTY(VisibleAnywhere)
    UProceduralMeshComponent* ProcMesh;

    UPROPERTY(EditAnywhere, Category = "Voxel")
    float BlockSize = 100.f;
    // Existing:
    void BuildFromBuffers(const TArray<FVector>& Vertices,
        const TArray<int32>& Triangles,
        const TArray<FVector>& Normals,
        const TArray<FVector2D>& UVs,
        const TArray<FLinearColor>& Colors,
        const TArray<FProcMeshTangent>& Tangents,
        UMaterialInterface* UseMaterial);

    // NEW: used by VoxelChunkSpawnCommand.cpp
    void BuildFromChunk(const FVoxelChunkData& Chunk, float InBlockSize,
UMaterialInterface* UseMaterial);
};
```

Public\VoxelCore.h

```
// Copyright Epic Games, Inc. All Rights Reserved.

#pragma once

#include "Modules/ModuleManager.h"

class FVoxelCoreModule : public IModuleInterface
{
public:

    /** IModuleInterface implementation */
    virtual void StartupModule() override;
    virtual void ShutdownModule() override;
};
```

Public\VoxelDebugVisualizer.h

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "VoxelChunk.h"
#include "VoxelGenerator.h"
#include "VoxelDebugVisualizer.generated.h"

/**
 * Debug actor you can place in the world. In the Details panel set ChunkX/ChunkZ and click
 * GenerateChunk (BlueprintCallable) to produce and log a small ASCII heightmap + summary.
 */
UCLASS(Blueprintable, ClassGroup = (Voxel), meta = (BlueprintSpawnableComponent))
```

```
class AVoxelDebugVisualizer : public AActor
{
    GENERATED_BODY()

public:
    AVoxelDebugVisualizer();

    // Chunk coords to generate
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Voxel Debug")
    int32 ChunkX = 0;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Voxel Debug")
    int32 ChunkZ = 0;

    // Optional seed override. If zero, uses DEFAULT_WORLD_SEED from ChunkConfig.h
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Voxel Debug")
    int32 Seed = 0;

    // Generate and print chunk summary to Output Log (callable from BP/Editor)
    UFUNCTION(CallInEditor, BlueprintCallable, Category = "Voxel Debug")
    void GenerateAndLogChunk();

protected:
    virtual void BeginPlay() override;
};
```

Public\VoxelEditPlaceTool.h

```
#pragma once
#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "VoxelEditPlaceTool.generated.h"

class AVoxelWorldManager;

UCLASS()
class AVoxelEditPlaceTool : public AActor
{
    GENERATED_BODY()
public:
    AVoxelEditPlaceTool();

    UPROPERTY(EditAnywhere) AVoxelWorldManager* WorldManager = nullptr;
    UPROPERTY(EditAnywhere) float MaxRange = 5000.f;
    UPROPERTY(EditAnywhere) uint8  PlaceBlockId = 3;   // Set to the block you want (e.g.
Grass = 3)
    UPROPERTY(EditAnywhere) bool   bDebug = true;
protected:
    virtual void BeginPlay() override;
    virtual void Tick(float DeltaSeconds) override;

private:
    void TryPlace();

    // optional quick hotkeys 1..9 to switch block id
    void HandleHotkeys();
};
```

Public\VoxelEditRemoveTool.h

```
#pragma once
#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "VoxelEditRemoveTool.generated.h"

class AVoxelWorldManager;

UCLASS()
class AVoxelEditRemoveTool : public AActor
{
    GENERATED_BODY()
public:
    AVoxelEditRemoveTool();

    UPROPERTY(EditAnywhere) AVoxelWorldManager* WorldManager = nullptr;
    UPROPERTY(EditAnywhere) float MaxRange = 5000.f;
    UPROPERTY(EditAnywhere) bool  bDebug = true;

protected:
    virtual void BeginPlay() override;
    virtual void Tick(float DeltaSeconds) override;

private:
    void TryRemove();
};
```

Public\VoxelGenerator.h

```
#pragma once

#include "CoreMinimal.h"
#include "ChunkHelpers.h"
#include "ChunkConfig.h"
#include "VoxelChunk.h"
```

```
#include "FastNoiseLite.h"

/**
 * Deterministic chunk generator using FastNoiseLite.
 * - Produces a base terrain: stone deep, dirt a few layers, grass on top, air above.
 * - Deterministic based on seed + chunk coords.
 */
class FVoxelGenerator
{
public:
    FVoxelGenerator(int32 InSeed = DEFAULT_WORLD_SEED);

    /** Generate base chunk contents into OutChunk. Does not apply deltas. */
    void GenerateBaseChunk(const FChunkKey& Key, FVoxelChunkData& OutChunk);

    /** Generator parameters (tweakable) */
    void SetHeightScale(float InScale) { HeightScale = InScale; }
    void SetHeightOffset(float InOffset) { HeightOffset = InOffset; }
    void SetNoiseFrequency(float InFreq) { NoiseFrequency = InFreq; }

private:
    int32 Seed;
    FastNoiseLite NoiseHeight;
    float HeightScale;    // multiplier to convert noise to height
    float HeightOffset;   // additive offset
    float NoiseFrequency; // frequency scale for noise inputs

    FORCEINLINE int32 WorldHeightFromNoise(float NoiseValue) const
    {
        // noise expected in [-1,1]. Map to [0, CHUNK_SIZE_Y-1]
        float Norm = (NoiseValue + 1.0f) * 0.5f; // 0..1
        float TotalMaxHeight = static_cast<float>(CHUNK_SIZE_Y - 1);
        float H = Norm * HeightScale + HeightOffset;
        // Clamp to chunk height bounds
        int32 HeightInt = FMath::Clamp(static_cast<int32>(FMath::RoundToInt(H)), 1,
CHUNK_SIZE_Y - 1);
        return HeightInt;
    }
};
```

Public\VoxelMesher.h

```
#pragma once

#include "CoreMinimal.h"
#include "VoxelChunk.h"
#include "VoxelTypes.h"
#include "ProceduralMeshComponent.h"
```

```
/**
 * Naive mesher that emits visible faces only.
 * - Produces scaled vertex positions (in world units) given BlockSize.
 * - Produces simple UVs suitable for a tiled atlas.
 */
class FVoxelMesher_Naive
{
public:
    /** Build mesh arrays from the chunk.
     * BlockSize = size of one cube along each axis in Unreal units (e.g. 100)
     */
    static void BuildMesh(const FVoxelChunkData& Chunk, float BlockSize,
        TArray<FVector>& OutVertices,
        TArray<int32>& OutTriangles,
        TArray<FVector>& OutNormals,
        TArray<FVector2D>& OutUVs,
        TArray<FLinearColor>& OutColors,
        TArray<FProcMeshTangent>& OutTangents);

private:
    // helper: returns true if neighbor at world-local (x+nx,y+ny,z+nz) is empty (air)
    static bool IsAirNeighbor(const FVoxelChunkData& Chunk, int32 X, int32 Y, int32 Z,
int32 NX, int32 NY, int32 NZ);

    // Simple atlas mapping: returns bottom-left UV and tile size (uTile,vTile)
    static void GetAtlasUVForBlock(EBlockId Id, FVector2D& OutUV0, FVector2D& OutTileSize);
};
```

Public\VoxelSaveSystem.h

```
#pragma once
#include "CoreMinimal.h"

struct FVoxelChunkData;

namespace VoxelSaveSystem
{
    // Apply saved edits (if any) into Data (keyed by Data.Key and Seed).
    bool LoadDelta(int32 Seed, FVoxelChunkData& Data);

    // Persist edited cells from Data.ModifiedBlocks for this chunk.
    void SaveDelta(int32 Seed, const FVoxelChunkData& Data);
}
```

Public\VoxelTypes.h

```
#pragma once

#include "CoreMinimal.h"
#include <cstdint>

// Basic block ID list (start small, expand later)
enum class EBlockId : uint8
{
    Air = 0,
    Dirt = 1,
    Grass = 2,
    Stone = 3,
    Sand = 4,
    Water = 5,
    // Add more block types here
    Max
};


struct FBlockInfo
{
    uint8 Id; // small numeric id (same as EBlockId cast)
    bool bIsSolid;
    bool bIsTransparent;
    uint8 MaterialIndex; // index into a texture atlas or material array


    FBlockInfo()
        : Id(static_cast<uint8>(EBlockId::Air))
        , bIsSolid(false)
        , bIsTransparent(true)
        , MaterialIndex(0)
    {
    }
};
```

Public\VoxelWorldManager.h

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "ProceduralMeshComponent.h"     // FProcMeshTangent
#include "ChunkHelpers.h"                // FChunkKey
#include "VoxelChunk.h"                  // FVoxelChunkData
#include "VoxelWorldManager.generated.h"

class AVoxelChunkActor;
```

```
UENUM(BlueprintType)
enum class EVoxelWorldSize : uint8
{
    Small  UMETA(DisplayName = "Small"),
    Medium UMETA(DisplayName = "Medium"),
    Large  UMETA(DisplayName = "Large")
};

/** Off-thread result: mesh buffers + data. */
struct FChunkMeshResult
{
    FChunkKey Key;
    float     BlockSize = 100.f;

    TSharedPtr<FVoxelChunkData> Data;

    TArray<FVector>          V;
    TArray<int32>            I;
    TArray<FVector>          N;
    TArray<FVector2D>        UV;
    TArray<FLinearColor>     C;
    TArray<FProcMeshTangent> T;
};

USTRUCT()
struct FChunkRecord
{
    GENERATED_BODY()

    TSharedPtr<FVoxelChunkData>      Data;
    TWeakObjectPtr<AVoxelChunkActor> Actor;
    bool bDirty = false;
};

UCLASS(Blueprintable)
class AVoxelWorldManager : public AActor
{
    GENERATED_BODY()
public:
    AVoxelWorldManager();

    /** Material for chunks (your atlas). */
    UPROPERTY(EditAnywhere, Category = "Voxel|Streaming")
    UMaterialInterface* ChunkMaterial = nullptr;

    /** Size of a voxel in UU. */
    UPROPERTY(EditAnywhere, Category = "Voxel|Streaming")
    float BlockSize = 100.f;
    /** Square radius (in chunks). */
    UPROPERTY(EditAnywhere, Category = "Voxel|Streaming", meta = (ClampMin = "1", ClampMax
= "32"))
    int32 RenderRadiusChunks = 6;

    /** Soft world clamp in chunks from origin. */
    UPROPERTY(EditAnywhere, Category = "Voxel|Streaming")
    EVoxelWorldSize WorldSize = EVoxelWorldSize::Small;

    /** Seed for deterministic world generation. */
    UPROPERTY(EditAnywhere, Category = "Voxel|Streaming")
    int32 WorldSeed = 1337;

    /** How often to update streaming (s). */
    UPROPERTY(EditAnywhere, Category = "Voxel|Streaming", meta = (ClampMin = "0.02",
ClampMax = "2.0"))
    float UpdateIntervalSeconds = 0.15f;

    /** Max background jobs. */
    UPROPERTY(EditAnywhere, Category = "Voxel|Streaming", meta = (ClampMin = "1", ClampMax
= "64"))
    int32 MaxConcurrentBackgroundTasks = 8;

    /** Optional: explicit actor to track; else Player0 pawn. */
    UPROPERTY(EditAnywhere, Category = "Voxel|Streaming")
    TWeakObjectPtr<AActor> TrackedActor;

    // Public wrapper to the existing private WorldToVoxel used by streaming
    bool WorldToVoxel_ForEdit(const FVector& World,
        FChunkKey& OutKey, int32& OutX, int32& OutY, int32& OutZ) const;

    bool WorldToVoxel_Centered(const FVector& World,
        FChunkKey& OutKey, int32& OutX, int32& OutY, int32& OutZ) const;


    // Mutate a single cell in a loaded chunk and trigger rebuild
    bool RemoveBlock_Local(const FChunkKey& Key, int32 X, int32 Y, int32 Z);

    // Sets a voxel to BlockId and schedules a rebuild for that chunk.
    bool PlaceBlock_Local(const FChunkKey& Key, int32 X, int32 Y, int32 Z, uint8 BlockId);


protected:
    virtual void BeginPlay() override;
    virtual void Tick(float DeltaSeconds) override;
    virtual void EndPlay(const EEndPlayReason::Type EndPlayReason) override;

private:
    // Loaded chunk records
    TMap<FChunkKey, FChunkRecord> Loaded;

    // Work queues
    TSet<FChunkKey> Pending;
    TQueue<TSharedPtr<FChunkMeshResult>, EQueueMode::Mpsc> Completed;

    float TimeAcc = 0.f;

    // --- Streaming helpers ---
    int32 GetWorldRadiusLimit() const;
    bool  IsWithinWorldLimit(const FChunkKey& Key) const;

    FIntPoint WorldToChunkXZ(const FVector& World) const;

    /** Convert world?chunk key + local voxel coords. Robust for negatives. */
    bool WorldToVoxel(const FVector& World, FChunkKey& OutKey, int32& OutX, int32& OutY,
int32& OutZ) const;

    void RecomputeDesiredSet(const FIntPoint& Center, TSet<FChunkKey>& OutDesired) const;
    void BuildDesiredOrdered(const FIntPoint& Center, TArray<FChunkKey>& OutOrdered) const;

    void KickBuild(const FChunkKey& Key, TSharedPtr<FVoxelChunkData> Existing);
    void SpawnOrUpdateChunkFromResult(const TSharedPtr<FChunkMeshResult>& Res);
    void UnloadNoLongerNeeded(const TSet<FChunkKey>& Desired);

    UWorld* GetWorldChecked() const { check(GetWorld()); return GetWorld(); }

    void FlushAllDirtyChunks();
};
```

Private\VoxelChunkActor.cpp

```
#include "VoxelChunkActor.h"
#include "VoxelMesher.h" // <-- for FVoxelMesher_Naive

AVoxelChunkActor::AVoxelChunkActor()
{
    PrimaryActorTick.bCanEverTick = false;

    ProcMesh = CreateDefaultSubobject<UProceduralMeshComponent>(TEXT("ProcMesh"));
    SetRootComponent(ProcMesh);

    ProcMesh->bUseAsyncCooking = true;
    ProcMesh->SetCollisionEnabled(ECollisionEnabled::QueryAndPhysics);
    ProcMesh->SetCollisionObjectType(ECC_WorldStatic);
    ProcMesh->SetCollisionResponseToAllChannels(ECR_Block);
}
```

```
void AVoxelChunkActor::BuildFromBuffers(const TArray<FVector>& Vertices,
    const TArray<int32>& Triangles,
    const TArray<FVector>& Normals,
    const TArray<FVector2D>& UVs,
    const TArray<FLinearColor>& Colors,
    const TArray<FProcMeshTangent>& Tangents,
    UMaterialInterface* UseMaterial)
{
    ProcMesh->ClearAllMeshSections();
    ProcMesh->CreateMeshSection_LinearColor(0, Vertices, Triangles, Normals, UVs, Colors,
Tangents, true);

    ProcMesh->bUseAsyncCooking = true;
    ProcMesh->SetCollisionEnabled(ECollisionEnabled::QueryAndPhysics);
    ProcMesh->SetCollisionObjectType(ECC_WorldStatic);
    ProcMesh->SetCollisionResponseToAllChannels(ECR_Block);

    if (UseMaterial)
    {
        ProcMesh->SetMaterial(0, UseMaterial);
    }
}

void AVoxelChunkActor::BuildFromChunk(const FVoxelChunkData& Chunk, float InBlockSize,
UMaterialInterface* UseMaterial)
{
    BlockSize = InBlockSize;

    TArray<FVector> V;
    TArray<int32>   I;
    TArray<FVector> N;
    TArray<FVector2D> UV;
    TArray<FLinearColor> C;
    TArray<FProcMeshTangent> T;

    // Naive mesher (Phase 3 path)
    FVoxelMesher_Naive::BuildMesh(Chunk, BlockSize, V, I, N, UV, C, T);

    BuildFromBuffers(V, I, N, UV, C, T, UseMaterial);
}
```

Private\VoxelChunkCommands.cpp

```
#include "VoxelCore.h"
#include "VoxelGenerator.h"
#include "VoxelChunk.h"
#include "Misc/CommandLine.h"
```

```
#include "Misc/Paths.h"
#include "FastNoiseLite.h"
#include "HAL/PlatformFilemanager.h"
#include "Misc/FileHelper.h"

static FAutoConsoleCommand CmdGenChunk(
    TEXT("Voxel.GenChunk"),
    TEXT("Generate a chunk and log its ascii map. Usage: Voxel.GenChunk <chunkX> <chunkZ>
[seed]"),
    FConsoleCommandWithArgsDelegate::CreateLambda([](const TArray<FString>& Args)
        {
            if (Args.Num() < 2)
            {
                UE_LOG(LogTemp, Warning, TEXT("Usage: Voxel.GenChunk <chunkX> <chunkZ>
[seed]"));
                return;
            }

            int32 ChunkX = FCString::Atoi(*Args[0]);
            int32 ChunkZ = FCString::Atoi(*Args[1]);
            int32 UseSeed = DEFAULT_WORLD_SEED;
            if (Args.Num() >= 3)
            {
                UseSeed = FCString::Atoi(*Args[2]);
            }

            FChunkKey Key(ChunkX, ChunkZ);
            FVoxelGenerator Generator(UseSeed);
            FVoxelChunkData Chunk(Key);
            Generator.GenerateBaseChunk(Key, Chunk);

            // Build simple heights and log (reuse logic similar to debug actor)
            TArray<int32> TopHeights; TopHeights.SetNumZeroed(CHUNK_SIZE_X * CHUNK_SIZE_Z);
            for (int32 Z = 0; Z < CHUNK_SIZE_Z; ++Z)
            {
                for (int32 X = 0; X < CHUNK_SIZE_X; ++X)
                {
                    int32 TopY = -1;
                    for (int32 Y = CHUNK_SIZE_Y - 1; Y >= 0; --Y)
                    {
                        if (Chunk.GetBlockAt(X, Y, Z) != EBlockId::Air)
                        {
                            TopY = Y; break;
                        }
                    }
                    TopHeights[Z * CHUNK_SIZE_X + X] = TopY;
                }
            }
            UE_LOG(LogTemp, Log, TEXT("=== Voxel.GenChunk (%d,%d) seed=%d ==="), ChunkX,
ChunkZ, UseSeed);
            for (int32 Z = 0; Z < CHUNK_SIZE_Z; ++Z)
            {
                FString Row;
                for (int32 X = 0; X < CHUNK_SIZE_X; ++X)
                {
                    int32 H = TopHeights[Z * CHUNK_SIZE_X + X];
                    int32 Scaled = (H < 0) ? 0 : FMath::Clamp(FMath::RoundToInt(((float)H /
(float)CHUNK_SIZE_Y) * 12.0f), 0, 12);
                    TCHAR C = (Scaled <= 9) ? TCHAR('0' + Scaled) : TCHAR('A' + (Scaled -
10));
                    Row.AppendChar(C);
                }
                UE_LOG(LogTemp, Log, TEXT("%s"), *Row);
            }

            UE_LOG(LogTemp, Log, TEXT("ModifiedBlocks count (should be zero): %d"),
Chunk.ModifiedBlocks.Num());
            UE_LOG(LogTemp, Log, TEXT("=== End Voxel.GenChunk ==="));
        })
);
```

Private\VoxelChunkSpawnCommand.cpp

```
#include "VoxelCore.h"
#include "VoxelGenerator.h"
#include "VoxelChunk.h"
#include "Misc/CommandLine.h"
#include "Misc/Paths.h"
#include "FastNoiseLite.h"
#include "HAL/PlatformFilemanager.h"
#include "Misc/FileHelper.h"

static FAutoConsoleCommand CmdGenChunk(
    TEXT("Voxel.GenChunk"),
    TEXT("Generate a chunk and log its ascii map. Usage: Voxel.GenChunk <chunkX> <chunkZ>
[seed]"),
    FConsoleCommandWithArgsDelegate::CreateLambda([](const TArray<FString>& Args)
        {
            if (Args.Num() < 2)
            {
                UE_LOG(LogTemp, Warning, TEXT("Usage: Voxel.GenChunk <chunkX> <chunkZ>
[seed]"));
                return;
            }

            int32 ChunkX = FCString::Atoi(*Args[0]);
            int32 ChunkZ = FCString::Atoi(*Args[1]);
            int32 UseSeed = DEFAULT_WORLD_SEED;
            if (Args.Num() >= 3)
            {
                UseSeed = FCString::Atoi(*Args[2]);
            }

            FChunkKey Key(ChunkX, ChunkZ);
            FVoxelGenerator Generator(UseSeed);
            FVoxelChunkData Chunk(Key);
            Generator.GenerateBaseChunk(Key, Chunk);

            // Build simple heights and log (reuse logic similar to debug actor)
            TArray<int32> TopHeights; TopHeights.SetNumZeroed(CHUNK_SIZE_X * CHUNK_SIZE_Z);
            for (int32 Z = 0; Z < CHUNK_SIZE_Z; ++Z)
            {
                for (int32 X = 0; X < CHUNK_SIZE_X; ++X)
                {
                    int32 TopY = -1;
                    for (int32 Y = CHUNK_SIZE_Y - 1; Y >= 0; --Y)
                    {
                        if (Chunk.GetBlockAt(X, Y, Z) != EBlockId::Air)
                        {
                            TopY = Y; break;
                        }
                    }
                    TopHeights[Z * CHUNK_SIZE_X + X] = TopY;
                }
            }

            UE_LOG(LogTemp, Log, TEXT("=== Voxel.GenChunk (%d,%d) seed=%d ==="), ChunkX,
ChunkZ, UseSeed);
            for (int32 Z = 0; Z < CHUNK_SIZE_Z; ++Z)
            {
                FString Row;
                for (int32 X = 0; X < CHUNK_SIZE_X; ++X)
                {
                    int32 H = TopHeights[Z * CHUNK_SIZE_X + X];
                    int32 Scaled = (H < 0) ? 0 : FMath::Clamp(FMath::RoundToInt(((float)H /
(float)CHUNK_SIZE_Y) * 12.0f), 0, 12);
                    TCHAR C = (Scaled <= 9) ? TCHAR('0' + Scaled) : TCHAR('A' + (Scaled -
10));
                    Row.AppendChar(C);
                }
                UE_LOG(LogTemp, Log, TEXT("%s"), *Row);
            }

            UE_LOG(LogTemp, Log, TEXT("ModifiedBlocks count (should be zero): %d"),
Chunk.ModifiedBlocks.Num());
```

```
            UE_LOG(LogTemp, Log, TEXT("=== End Voxel.GenChunk ==="));
        })
);

```

Private\VoxelCoreModule.cpp

```
// Copyright Epic Games, Inc. All Rights Reserved.

#include "VoxelCore.h"

#define LOCTEXT_NAMESPACE "FVoxelCoreModule"

void FVoxelCoreModule::StartupModule()
{
    // This code will execute after your module is loaded into memory; the exact timing is
specified in the .uplugin file per-module
}

void FVoxelCoreModule::ShutdownModule()
{
    // This function may be called during shutdown to clean up your module.  For modules
that support dynamic reloading,
    // we call this function before unloading the module.
}

#undef LOCTEXT_NAMESPACE

IMPLEMENT_MODULE(FVoxelCoreModule, VoxelCore)
```

Private\VoxelCoreTests.cpp

```
#include "VoxelCore.h"
#include "ChunkConfig.h"
#include "VoxelTypes.h"
#include "FastNoiseLite.h"

static FAutoConsoleCommand CmdVoxelTestSetup(
    TEXT("Voxel.TestSetup"),
    TEXT("Runs voxel core setup test to validate indexing + noise"),
    FConsoleCommandDelegate::CreateStatic([]()
        {
            int32 X = 5, Y = 10, Z = 2;
            int32 Index = X + Y * CHUNK_SIZE_X + Z * CHUNK_SIZE_X * CHUNK_SIZE_Y;

            FString IndexMsg = FString::Printf(TEXT("Index from (5,10,2) = %d"), Index);
            UE_LOG(LogTemp, Log, TEXT("%s"), *IndexMsg);
            if (GEngine) GEngine->AddOnScreenDebugMessage(-1, 5.f, FColor::Yellow,
IndexMsg);
```

```
            int32 OutX = Index % CHUNK_SIZE_X;
            int32 OutY = (Index / CHUNK_SIZE_X) % CHUNK_SIZE_Y;
            int32 OutZ = Index / (CHUNK_SIZE_X * CHUNK_SIZE_Y);

            FString CoordMsg = FString::Printf(TEXT("XYZ from index: (%d, %d, %d)"), OutX,
OutY, OutZ);
            UE_LOG(LogTemp, Log, TEXT("%s"), *CoordMsg);
            if (GEngine) GEngine->AddOnScreenDebugMessage(-1, 5.f, FColor::Cyan, CoordMsg);

            FastNoiseLite Noise;
            Noise.SetSeed(DEFAULT_WORLD_SEED);
            Noise.SetNoiseType(FastNoiseLite::NoiseType_OpenSimplex2);
            float Value = Noise.GetNoise(10.0f, 20.0f, 30.0f);

            FString NoiseMsg = FString::Printf(TEXT("Noise(10,20,30) = %f"), Value);
            UE_LOG(LogTemp, Log, TEXT("%s"), *NoiseMsg);
            if (GEngine) GEngine->AddOnScreenDebugMessage(-1, 5.f, FColor::Green,
NoiseMsg);
        })
);
```

Private\VoxelDebugVisualizer.cpp

```
#include "VoxelDebugVisualizer.h"

AVoxelDebugVisualizer::AVoxelDebugVisualizer()
{
    PrimaryActorTick.bCanEverTick = false;
}

void AVoxelDebugVisualizer::BeginPlay()
{
    Super::BeginPlay();
}

void AVoxelDebugVisualizer::GenerateAndLogChunk()
{
    const int32 UseSeed = (Seed != 0) ? Seed : DEFAULT_WORLD_SEED;
    FVoxelGenerator Generator(UseSeed);
    FChunkKey Key(ChunkX, ChunkZ);
    FVoxelChunkData Chunk(Key);
    Generator.GenerateBaseChunk(Key, Chunk);

    // Build a simple heightmap and top-block map
    TArray<int32> TopHeights;
    TopHeights.SetNumZeroed(CHUNK_SIZE_X * CHUNK_SIZE_Z);
    // For each column find highest non-air block
    for (int32 Z = 0; Z < CHUNK_SIZE_Z; ++Z)
    {
        for (int32 X = 0; X < CHUNK_SIZE_X; ++X)
        {
            int32 TopY = -1;
            for (int32 Y = CHUNK_SIZE_Y - 1; Y >= 0; --Y)
            {
                EBlockId Id = Chunk.GetBlockAt(X, Y, Z);
                if (Id != EBlockId::Air)
                {
                    TopY = Y;
                    break;
                }
            }
            TopHeights[Z * CHUNK_SIZE_X + X] = TopY;
        }
    }

    // Log header
    UE_LOG(LogTemp, Log, TEXT("=== VoxelDebug: Chunk (%d,%d) Seed=%d ==="), ChunkX, ChunkZ,
UseSeed);

    // ASCII map of heights (scaled): print rows from Z=0..Z-1 with X increasing
    FString Row;
    for (int32 Z = 0; Z < CHUNK_SIZE_Z; ++Z)
    {
        Row.Empty();
        for (int32 X = 0; X < CHUNK_SIZE_X; ++X)
        {
            int32 H = TopHeights[Z * CHUNK_SIZE_X + X];
            // Map height to an ASCII char: use 0..9 and letters for higher values
            int32 Scaled = (H < 0) ? 0 : FMath::Clamp(FMath::RoundToInt(((float)H /
(float)CHUNK_SIZE_Y) * 12.0f), 0, 12);
            TCHAR C;
            if (Scaled <= 9) C = TCHAR('0' + Scaled);
            else C = TCHAR('A' + (Scaled - 10));
            Row.AppendChar(C);
        }
        UE_LOG(LogTemp, Log, TEXT("%s"), *Row);
    }

    // Print some example top-block counts
    TMap<uint8, int32> TopCounts;
    for (int32 Z = 0; Z < CHUNK_SIZE_Z; ++Z)
    {
        for (int32 X = 0; X < CHUNK_SIZE_X; ++X)
        {
            int32 Y = TopHeights[Z * CHUNK_SIZE_X + X];
            uint8 Id = 0;
            if (Y >= 0) Id = static_cast<uint8>(Chunk.GetBlockAt(X, Y, Z));
            TopCounts.FindOrAdd(Id)++;
        }
    }

    for (auto& Pair : TopCounts)
    {
        uint8 Id = Pair.Key;
        int32 Count = Pair.Value;
        FString Name = TEXT("Unknown");
        switch (static_cast<EBlockId>(Id))
        {
        case EBlockId::Air: Name = TEXT("Air"); break;
        case EBlockId::Dirt: Name = TEXT("Dirt"); break;
        case EBlockId::Grass: Name = TEXT("Grass"); break;
        case EBlockId::Stone: Name = TEXT("Stone"); break;
        default: break;
        }
        UE_LOG(LogTemp, Log, TEXT("TopBlock %d (%s): %d columns"), Id, *Name, Count);
    }

    UE_LOG(LogTemp, Log, TEXT("ModifiedBlocks count (should be zero): %d"),
Chunk.ModifiedBlocks.Num());
    UE_LOG(LogTemp, Log, TEXT("=== End VoxelDebug ==="));
}
```

Private\VoxelEditPlaceTool.cpp

```
#include "VoxelEditPlaceTool.h"
#include "VoxelWorldManager.h"
#include "Kismet/GameplayStatics.h"
#include "DrawDebugHelpers.h"

AVoxelEditPlaceTool::AVoxelEditPlaceTool()
{
    PrimaryActorTick.bCanEverTick = true;
    PrimaryActorTick.bStartWithTickEnabled = true;
}

void AVoxelEditPlaceTool::BeginPlay()
{
    Super::BeginPlay();
    EnableInput(UGameplayStatics::GetPlayerController(this, 0));
}

void AVoxelEditPlaceTool::Tick(float)
```

```
{
    if (!WorldManager) return;

    APlayerController* PC = UGameplayStatics::GetPlayerController(this, 0);
    if (!PC) return;

    HandleHotkeys();

    // Use LMB for place on this tool (simple & consistent)
    if (PC->WasInputKeyJustPressed(EKeys::LeftMouseButton))
    {
        TryPlace();
    }
}

void AVoxelEditPlaceTool::HandleHotkeys()
{
    APlayerController* PC = UGameplayStatics::GetPlayerController(this, 0);
    if (!PC) return;

    // Quick-select ids 1..9 (optional)
    if (PC->WasInputKeyJustPressed(EKeys::One))   PlaceBlockId = 1;
    if (PC->WasInputKeyJustPressed(EKeys::Two))   PlaceBlockId = 2;
    if (PC->WasInputKeyJustPressed(EKeys::Three)) PlaceBlockId = 3;
    if (PC->WasInputKeyJustPressed(EKeys::Four))  PlaceBlockId = 4;
    if (PC->WasInputKeyJustPressed(EKeys::Five))  PlaceBlockId = 5;
    if (PC->WasInputKeyJustPressed(EKeys::Six))   PlaceBlockId = 6;
    if (PC->WasInputKeyJustPressed(EKeys::Seven)) PlaceBlockId = 7;
    if (PC->WasInputKeyJustPressed(EKeys::Eight)) PlaceBlockId = 8;
    if (PC->WasInputKeyJustPressed(EKeys::Nine))  PlaceBlockId = 9;
}

void AVoxelEditPlaceTool::TryPlace()
{
    if (!WorldManager) return;

    // Camera ray
    APlayerController* PC = UGameplayStatics::GetPlayerController(this, 0);
    FVector O; FRotator R; PC->GetPlayerViewPoint(O, R);
    const FVector D = R.Vector().GetSafeNormal();
    const FVector End = O + D * MaxRange;

    // Raycast against chunk meshes (to get the face we�re targeting)
    FHitResult Hit;
    FCollisionQueryParams Params(SCENE_QUERY_STAT(VoxelPlaceTrace), false, this);
    Params.bTraceComplex = true;
    if (!GetWorld()->LineTraceSingleByChannel(Hit, O, End, ECC_Visibility, Params))
        return;
    // Persist the click ray
    DrawDebugLine(GetWorld(), O, Hit.ImpactPoint, FColor::Green, /*bPersistent=*/true,
/*LifeTime=*/0.f, 0, 2.0f);

    // Place on the air side: nudge a hair opposite the ray into empty space
    const float   Eps = FMath::Max(1.f, WorldManager->BlockSize * 0.01f);
    const FVector POutside = Hit.ImpactPoint - D * Eps;
    DrawDebugPoint(GetWorld(), POutside, 8.f, FColor::Yellow, /*bPersistent=*/true,
/*LifeTime=*/0.f);

    // Map world ? (chunk key, local x/y/z) using the centered grid
    FChunkKey Key; int32 X = 0, Y = 0, Z = 0;
    if (!WorldManager->WorldToVoxel_Centered(POutside, Key, X, Y, Z))
        return;

    const bool OK = WorldManager->PlaceBlock_Local(Key, X, Y, Z, PlaceBlockId);

    if (bDebug)
    {
        FString Msg = FString::Printf(TEXT("Placed ID %u @ (%d,%d,%d) in Chunk (%d,%d)"),
            (uint32)PlaceBlockId, X, Y, Z, Key.X, Key.Z);
        GEngine->AddOnScreenDebugMessage(-1, 1.2f, OK ? FColor::Green : FColor::Red, OK ?
Msg : TEXT("Place failed"));
    }
}
```

Private\VoxelEditRemoveTool.cpp

```
#include "VoxelEditRemoveTool.h"
#include "VoxelWorldManager.h"
#include "Kismet/GameplayStatics.h"
#include "DrawDebugHelpers.h"

AVoxelEditRemoveTool::AVoxelEditRemoveTool()
{
    PrimaryActorTick.bCanEverTick = true;
    PrimaryActorTick.bStartWithTickEnabled = true;
}

void AVoxelEditRemoveTool::BeginPlay()
{
    Super::BeginPlay();
    EnableInput(UGameplayStatics::GetPlayerController(this, 0));
}

void AVoxelEditRemoveTool::Tick(float)
{
    if (!WorldManager) return;
    APlayerController* PC = UGameplayStatics::GetPlayerController(this, 0);
    if (!PC) return;

    if (PC->WasInputKeyJustPressed(EKeys::RightMouseButton))
    {
        TryRemove();
    }
}

void AVoxelEditRemoveTool::TryRemove()
{
    if (!WorldManager) return;

    APlayerController* PC = UGameplayStatics::GetPlayerController(this, 0);
    if (!PC) return;

    // Camera ray
    FVector O; FRotator R;
    PC->GetPlayerViewPoint(O, R);
    const FVector D = R.Vector().GetSafeNormal();
    const FVector End = O + D * MaxRange;

    // Raycast against chunk meshes
    FHitResult Hit;
    FCollisionQueryParams Params(SCENE_QUERY_STAT(VoxelRemoveTrace), false, this);
    Params.bTraceComplex = true;
    if (!GetWorld()->LineTraceSingleByChannel(Hit, O, End, ECC_Visibility, Params))
        return;

    // Persist the click ray
    DrawDebugLine(GetWorld(), O, Hit.ImpactPoint, FColor::Red, /*bPersistentLines=*/true,
/*LifeTime=*/0.f, 0, 2.0f);

    // Push a hair *into* the block along the ray ? deterministic interior sample
    const float   Eps = FMath::Max(1.f, WorldManager->BlockSize * 0.01f);
    const FVector PInside = Hit.ImpactPoint + D * Eps;   // +D = inside the clicked voxel
    DrawDebugPoint(GetWorld(), PInside, 8.f, FColor::Yellow, /*bPersistent=*/true,
/*LifeTime=*/0.f);

    // Map world ? (chunk key, local x/y/z) using centered grid
    FChunkKey Key; int32 X = 0, Y = 0, Z = 0;
    if (!WorldManager->WorldToVoxel_Centered(PInside, Key, X, Y, Z))
        return;

    // Clear that cell and rebuild
    const bool OK = WorldManager->RemoveBlock_Local(Key, X, Y, Z);

    if (bDebug)
    {
        GEngine->AddOnScreenDebugMessage(
            -1, 1.2f, OK ? FColor::Green : FColor::Red,
            OK ? TEXT("Removed") : TEXT("Remove failed"));
    }
}
```

Private\VoxelGenerator.cpp

```
#include "VoxelGenerator.h"
#include "ChunkConfig.h"
#include "VoxelTypes.h"

FVoxelGenerator::FVoxelGenerator(int32 InSeed)
    : Seed(InSeed)
    , HeightScale(static_cast<float>(CHUNK_SIZE_Y) * 0.6f) // ~60% of vertical range
    , HeightOffset(static_cast<float>(CHUNK_SIZE_Y) * 0.2f) // base offset
    , NoiseFrequency(0.05f) // << increased frequency so noise varies more
{
    NoiseHeight.SetSeed(Seed);
    NoiseHeight.SetNoiseType(FastNoiseLite::NoiseType_OpenSimplex2);
    NoiseHeight.SetFrequency(NoiseFrequency);
}

void FVoxelGenerator::GenerateBaseChunk(const FChunkKey& Key, FVoxelChunkData& OutChunk)
{
    OutChunk.Key = Key;

    if (OutChunk.Blocks.Num() != CHUNK_VOLUME)
    {
        OutChunk.Blocks.SetNumZeroed(CHUNK_VOLUME);
    }
    OutChunk.ClearDeltas();

    for (int32 LocalZ = 0; LocalZ < CHUNK_SIZE_Z; ++LocalZ)
    {
        for (int32 LocalX = 0; LocalX < CHUNK_SIZE_X; ++LocalX)
        {
            // World coordinates for noise input
            int32 WorldX = Key.X * CHUNK_SIZE_X + LocalX;
            int32 WorldZ = Key.Z * CHUNK_SIZE_Z + LocalZ;

            // Noise input (scaled floats)
            float NX = static_cast<float>(WorldX) * NoiseFrequency;
            float NZ = static_cast<float>(WorldZ) * NoiseFrequency;
            float NoiseVal = NoiseHeight.GetNoise(NX, NZ);

            // Map noise to usable chunk height
            int32 ColumnTopY = WorldHeightFromNoise(NoiseVal);

            for (int32 LocalY = 0; LocalY < CHUNK_SIZE_Y; ++LocalY)
            {
                int32 Index = IndexFromXYZ(LocalX, LocalY, LocalZ);

                if (LocalY > ColumnTopY)
                {
                    OutChunk.Blocks[Index] = static_cast<uint8>(EBlockId::Air);
                }
                else
                {
                    int32 Depth = ColumnTopY - LocalY;
                    if (Depth == 0)
                    {
                        OutChunk.Blocks[Index] = static_cast<uint8>(EBlockId::Grass);
                    }
                    else if (Depth <= 3)
                    {
                        OutChunk.Blocks[Index] = static_cast<uint8>(EBlockId::Dirt);
                    }
                    else
                    {
                        OutChunk.Blocks[Index] = static_cast<uint8>(EBlockId::Stone);
                    }
                }
            }
        }
    }
}
```

Private\VoxelMesher.cpp

```
#include "VoxelMesher.h"
#include "ChunkConfig.h"
#include "ChunkHelpers.h"
#include "Math/UnrealMathUtility.h"
#include "ProceduralMeshComponent.h" // for FProcMeshTangent

void FVoxelMesher_Naive::BuildMesh(const FVoxelChunkData& Chunk, float BlockSize,
    TArray<FVector>& OutVertices,
    TArray<int32>& OutTriangles,
    TArray<FVector>& OutNormals,
    TArray<FVector2D>& OutUVs,
    TArray<FLinearColor>& OutColors,
    TArray<FProcMeshTangent>& OutTangents)
{
    OutVertices.Reset();
    OutTriangles.Reset();
    OutNormals.Reset();
    OutUVs.Reset();
    OutColors.Reset();
    OutTangents.Reset();

    const float Half = BlockSize * 0.5f;

    for (int32 X = 0; X < CHUNK_SIZE_X; ++X)
    {
        for (int32 Z = 0; Z < CHUNK_SIZE_Z; ++Z)
        {
            for (int32 Y = 0; Y < CHUNK_SIZE_Y; ++Y)
            {
                EBlockId Id = Chunk.GetBlockAt(X, Y, Z);
                if (Id == EBlockId::Air) continue;

                const float WorldX = static_cast<float>(X) * BlockSize;
                const float WorldY = static_cast<float>(Z) * BlockSize;
                const float WorldZ = static_cast<float>(Y) * BlockSize;

                const FVector Min(WorldX - Half, WorldY - Half, WorldZ - Half);
                const FVector Max(WorldX + Half, WorldY + Half, WorldZ + Half);

                const FVector
                    B00(Min.X, Min.Y, Min.Z), B10(Max.X, Min.Y, Min.Z),
                    B11(Max.X, Max.Y, Min.Z), B01(Min.X, Max.Y, Min.Z),
                    T00(Min.X, Min.Y, Max.Z), T10(Max.X, Min.Y, Max.Z),
                    T11(Max.X, Max.Y, Max.Z), T01(Min.X, Max.Y, Max.Z);

                FVector2D UV0, Tile;
                GetAtlasUVForBlock(Id, UV0, Tile);

                FLinearColor Color = FLinearColor::White;
                switch (Id)
                {
                case EBlockId::Grass: Color = FLinearColor(0.1f, 0.8f, 0.1f); break;
                case EBlockId::Dirt:  Color = FLinearColor(0.45f, 0.28f, 0.13f); break;
                case EBlockId::Stone: Color = FLinearColor(0.5f, 0.5f, 0.5f); break;
                default: break;
                }

                auto PushFace = [&](const FVector& A, const FVector& B, const FVector& C, const FVector& D, const FVector& Normal, bool bFlipTopUVs = false)
                    {
                        const int32 Base = OutVertices.Num();

                        OutVertices.Add(A); // 0
                        OutVertices.Add(B); // 1
                        OutVertices.Add(C); // 2
                        OutVertices.Add(D); // 3

                        // Flipped winding order for outward normals
                        OutTriangles.Add(Base + 0);
                        OutTriangles.Add(Base + 2);
                        OutTriangles.Add(Base + 1);
                        OutTriangles.Add(Base + 0);
                        OutTriangles.Add(Base + 3);
                        OutTriangles.Add(Base + 2);

                        OutNormals.Add(Normal);
                        OutNormals.Add(Normal);
                        OutNormals.Add(Normal);
                        OutNormals.Add(Normal);

                        // UVs
                        if (!bFlipTopUVs)
                        {
                            const FVector2D U00(UV0.X, UV0.Y);
                            const FVector2D U10(UV0.X + Tile.X, UV0.Y);
                            const FVector2D U11(UV0.X + Tile.X, UV0.Y + Tile.Y);
                            const FVector2D U01(UV0.X, UV0.Y + Tile.Y);
                            OutUVs.Add(U00);
                            OutUVs.Add(U10);
                            OutUVs.Add(U11);
                            OutUVs.Add(U01);
                        }
                        else
                        {
                            // ? Corrected orientation for grass top
                            const FVector2D U00(UV0.X, UV0.Y + Tile.Y);
                            const FVector2D U10(UV0.X + Tile.X, UV0.Y + Tile.Y);
                            const FVector2D U11(UV0.X + Tile.X, UV0.Y);
                            const FVector2D U01(UV0.X, UV0.Y);
                            OutUVs.Add(U00);
                            OutUVs.Add(U10);
                            OutUVs.Add(U11);
                            OutUVs.Add(U01);
                        }

                        OutColors.Add(Color);
                        OutColors.Add(Color);
                        OutColors.Add(Color);
                        OutColors.Add(Color);

                        const FVector TangentDir = FVector::CrossProduct(FVector::UpVector,
Normal).GetSafeNormal();
                        const FProcMeshTangent Tangent(TangentDir, false);
                        OutTangents.Add(Tangent);
                        OutTangents.Add(Tangent);
                        OutTangents.Add(Tangent);
                        OutTangents.Add(Tangent);
                    };

                auto IsAir = [&](int32 NX, int32 NY, int32 NZ)
                    {
                        if (NX < 0 || NX >= CHUNK_SIZE_X ||
                            NY < 0 || NY >= CHUNK_SIZE_Y ||
                            NZ < 0 || NZ >= CHUNK_SIZE_Z) return true;
                        return Chunk.GetBlockAt(NX, NY, NZ) == EBlockId::Air;
                    };

                if (IsAir(X + 1, Y, Z)) PushFace(B10, B11, T11, T10, FVector(1, 0, 0));
// +X
                if (IsAir(X - 1, Y, Z)) PushFace(B01, B00, T00, T01, FVector(-1, 0, 0));
// -X
                if (IsAir(X, Y, Z + 1)) PushFace(B11, B01, T01, T11, FVector(0, 1, 0));
// +Y (north)
                if (IsAir(X, Y, Z - 1)) PushFace(B00, B10, T10, T00, FVector(0, -1, 0));
// -Y (south)
                if (IsAir(X, Y + 1, Z)) PushFace(T00, T10, T11, T01, FVector(0, 0, 1),
true); // +Z (top, flip UVs)
                if (IsAir(X, Y - 1, Z)) PushFace(B01, B11, B10, B00, FVector(0, 0, -1));
// -Z (bottom)
            }
        }
    }
}

bool FVoxelMesher_Naive::IsAirNeighbor(const FVoxelChunkData& Chunk, int32 X, int32 Y,
int32 Z, int32 NX, int32 NY, int32 NZ)
{
    int32 NXAbs = X + NX;
    int32 NYAbs = Y + NY;
    int32 NZAbs = Z + NZ;

    if (NXAbs < 0 || NXAbs >= CHUNK_SIZE_X ||
        NYAbs < 0 || NYAbs >= CHUNK_SIZE_Y ||
        NZAbs < 0 || NZAbs >= CHUNK_SIZE_Z)
    {
        return true;
    }

    EBlockId Neighbor = Chunk.GetBlockAt(NXAbs, NYAbs, NZAbs);
    return (Neighbor == EBlockId::Air);
}
```

```cpp
//void FVoxelMesher_Naive::GetAtlasUVForBlock(EBlockId Id, FVector2D& OutUV0, FVector2D&
OutTileSize)
//{
//    const int32 TilesAcross = 4;
//    const float TileSize = 1.0f / static_cast<float>(TilesAcross);
//
//    int32 Slot = 0;
//    switch (Id)
//    {
//    case EBlockId::Grass: Slot = 0; break;
//    case EBlockId::Dirt:  Slot = 1; break;
//    case EBlockId::Stone: Slot = 2; break;
//    default: Slot = 3; break;
//    }
//
//    OutUV0 = FVector2D(Slot * TileSize, 0.0f);
//    OutTileSize = FVector2D(TileSize, 1.0f);
//}

void FVoxelMesher_Naive::GetAtlasUVForBlock(EBlockId Id, FVector2D& OutUV0, FVector2D&
OutTileSize)
{
    // Atlas layout (adjust to your real atlas)
    constexpr int32 TilesAcross = 4;
    constexpr int32 TilesDown = 1;
    constexpr int32 AtlasResX = TilesAcross * 64; // 256 if tiles are 64px wide
    constexpr int32 AtlasResY = TilesDown * 64; // 64  if tiles are 64px tall

    // Inset (gutter) in texels to avoid bleeding from neighbors
    constexpr int32 PadPx = 2; // try 2; if you still see seams at distance, try 3�4

    const float TileW = 1.0f / float(TilesAcross);
    const float TileH = 1.0f / float(TilesDown);
    const float PadU = float(PadPx) / float(AtlasResX);
    const float PadV = float(PadPx) / float(AtlasResY);

    int32 slotX = 0, slotY = 0;
    switch (Id)
    {
    case EBlockId::Grass: slotX = 0; break;
    case EBlockId::Dirt:  slotX = 1; break;
    case EBlockId::Stone: slotX = 2; break;
    default:              slotX = 3; break;
    }

    // Start of the tile + inset
    OutUV0 = FVector2D(slotX * TileW + PadU, slotY * TileH + PadV);
    // Shrink tile so we don't touch the outer border
    OutTileSize = FVector2D(TileW - 2 * PadU, TileH - 2 * PadV);
}
```

Private\VoxelSaveSystem.cpp

```
#include "VoxelSaveSystem.h"
#include "VoxelChunk.h"          // FVoxelChunkData, EBlockId, Data.Key, ModifiedBlocks,
SetBlockAt
#include "ChunkConfig.h"         // CHUNK_SIZE_X/Y/Z
#include "Misc/Paths.h"
#include "Misc/FileHelper.h"
#include "HAL/PlatformFilemanager.h"
#include "Serialization/BufferArchive.h"
#include "Serialization/MemoryReader.h"

static constexpr uint32 VCD_MAGIC = 0x44435631; // 'VCD1'
static constexpr uint16 VCD_VER = 1;

static FString ChunkPath(int32 Seed, const FChunkKey& Key)
{
    const FString Root = FPaths::Combine(FPaths::ProjectSavedDir(), TEXT("Voxel"),
FString::Printf(TEXT("Seed_%d"), Seed));
    const FString Chunks = FPaths::Combine(Root, TEXT("Chunks"));
    IFileManager::Get().MakeDirectory(*Chunks, /*Tree=*/true);
    return FPaths::Combine(Chunks, FString::Printf(TEXT("%d_%d.vcd"), Key.X, Key.Z));
}

namespace VoxelSaveSystem
{
    bool LoadDelta(int32 Seed, FVoxelChunkData& Data)
    {
        const FString Path = ChunkPath(Seed, Data.Key);
        TArray<uint8> Bytes;
        if (!FFileHelper::LoadFileToArray(Bytes, *Path) || Bytes.Num() < 10) return false;

        FMemoryReader Ar(Bytes);
        uint32 Magic = 0; uint16 Ver = 0; uint32 Count = 0;
        Ar << Magic; Ar << Ver; Ar << Count;
        if (Magic != VCD_MAGIC || Ver != VCD_VER) return false;

        for (uint32 i = 0; i < Count; ++i)
        {
            int32 I; uint8 Id;
            Ar << I; Ar << Id;

            //const int32 X = I % CHUNK_SIZE_X;
            //const int32 Y = (I / CHUNK_SIZE_X) % CHUNK_SIZE_Y;
            //const int32 Z = I / (CHUNK_SIZE_X * CHUNK_SIZE_Y);
            const int32 X = I % CHUNK_SIZE_X;
            const int32 Z = (I / CHUNK_SIZE_X) % CHUNK_SIZE_Z;
            const int32 Y = I / (CHUNK_SIZE_X * CHUNK_SIZE_Z);

            Data.SetBlockAt(X, Y, Z, static_cast<EBlockId>(Id));
        }
        return true;
    }

    void SaveDelta(int32 Seed, const FVoxelChunkData& Data)
    {
        if (Data.ModifiedBlocks.Num() == 0) return;

        FBufferArchive Ar;
        uint32 Magic = VCD_MAGIC; uint16 Ver = VCD_VER;
        uint32 Count = (uint32)Data.ModifiedBlocks.Num();
        Ar << Magic; Ar << Ver; Ar << Count;

        //for (const TPair<int32, uint8>& P : Data.ModifiedBlocks)
        //{
        //  int32 I = P.Key; uint8 Id = P.Value;
        //  Ar << I; Ar << Id;
        //}

        for (const TPair<int32, uint16>& P : Data.ModifiedBlocks) // value is uint16 in
your data
        {
            int32 I = P.Key;
            uint8 Id = static_cast<uint8>(P.Value);               // file stays compact
(8-bit ids)
            Ar << I; Ar << Id;
        }

        const FString Path = ChunkPath(Seed, Data.Key);
        FFileHelper::SaveArrayToFile(Ar, *Path);
    }
}
```

Private\VoxelWorldManager.cpp

```
#include "VoxelWorldManager.h"
#include "VoxelChunkActor.h"
#include "VoxelGenerator.h"
#include "VoxelMesher.h"              // FVoxelMesher_Naive
#include "VoxelTypes.h"
#include "ChunkConfig.h"
#include "VoxelSaveSystem.h"
```

```
#include "Kismet/GameplayStatics.h"
#include "Async/Async.h"

AVoxelWorldManager::AVoxelWorldManager()
{
    PrimaryActorTick.bCanEverTick = true;
    PrimaryActorTick.bStartWithTickEnabled = true;
}

void AVoxelWorldManager::BeginPlay()
{
    Super::BeginPlay();
}

int32 AVoxelWorldManager::GetWorldRadiusLimit() const
{
    switch (WorldSize)
    {
    case EVoxelWorldSize::Small:  return 16;
    case EVoxelWorldSize::Medium: return 64;
    case EVoxelWorldSize::Large:  return 256;
    default: return 16;
    }
}

bool AVoxelWorldManager::IsWithinWorldLimit(const FChunkKey& Key) const
{
    const int32 R = GetWorldRadiusLimit();
    return FMath::Abs(Key.X) <= R && FMath::Abs(Key.Z) <= R;
}

FIntPoint AVoxelWorldManager::WorldToChunkXZ(const FVector& W) const
{
    const double CSX = (double)CHUNK_SIZE_X * BlockSize;
    const double CSZ = (double)CHUNK_SIZE_Z * BlockSize;
    const int32 Cx = FMath::FloorToInt(W.X / CSX);
    const int32 Cz = FMath::FloorToInt(W.Y / CSZ); // chunk Z -> world Y
    return FIntPoint(Cx, Cz);
}

// Robust world->voxel using a global grid (handles negatives).
bool AVoxelWorldManager::WorldToVoxel(const FVector& W, FChunkKey& OutKey, int32& x, int32&
y, int32& z) const
{
    const double invBS = 1.0 / (double)BlockSize;
    const int32 GX = FMath::FloorToInt(W.X * invBS); // world X -> voxel X
    const int32 GZ = FMath::FloorToInt(W.Y * invBS); // world Y -> voxel Z
    const int32 GY = FMath::FloorToInt(W.Z * invBS); // world Z -> voxel Y
    const int32 CX = FMath::FloorToInt((double)GX / (double)CHUNK_SIZE_X);
    const int32 CZ = FMath::FloorToInt((double)GZ / (double)CHUNK_SIZE_Z);

    const int32 LX = GX - CX * CHUNK_SIZE_X;
    const int32 LZ = GZ - CZ * CHUNK_SIZE_Z;
    const int32 LY = FMath::Clamp(GY, 0, CHUNK_SIZE_Y - 1);

    OutKey = FChunkKey(CX, CZ);
    x = FMath::Clamp(LX, 0, CHUNK_SIZE_X - 1);
    z = FMath::Clamp(LZ, 0, CHUNK_SIZE_Z - 1);
    y = LY;
    return true;
}

void AVoxelWorldManager::RecomputeDesiredSet(const FIntPoint& Center, TSet<FChunkKey>&
OutDesired) const
{
    OutDesired.Reset();
    const int32 R = RenderRadiusChunks;
    for (int32 dz = -R; dz <= R; ++dz)
    {
        for (int32 dx = -R; dx <= R; ++dx)
        {
            const FChunkKey Key(Center.X + dx, Center.Y + dz);
            if (IsWithinWorldLimit(Key)) OutDesired.Add(Key);
        }
    }
}

void AVoxelWorldManager::BuildDesiredOrdered(const FIntPoint& Center, TArray<FChunkKey>&
OutOrdered) const
{
    OutOrdered.Reset();
    const int32 R = RenderRadiusChunks;
    for (int32 dz = -R; dz <= R; ++dz)
    {
        for (int32 dx = -R; dx <= R; ++dx)
        {
            const FChunkKey K(Center.X + dx, Center.Y + dz);
            if (IsWithinWorldLimit(K)) OutOrdered.Add(K);
        }
    }
    OutOrdered.Sort([&](const FChunkKey& A, const FChunkKey& B) {
        const int32 dA = FMath::Abs(A.X - Center.X) + FMath::Abs(A.Z - Center.Y);
        const int32 dB = FMath::Abs(B.X - Center.X) + FMath::Abs(B.Z - Center.Y);
        return dA < dB;
        });
}
```

```
void AVoxelWorldManager::KickBuild(const FChunkKey& Key, TSharedPtr<FVoxelChunkData>
Existing)
{
    if (Pending.Contains(Key)) return;
    Pending.Add(Key);

    const int32 Seed = WorldSeed;
    const float BS = BlockSize;

    Async(EAsyncExecution::ThreadPool, [this, Key, Existing, Seed, BS]()
        {
            TSharedPtr<FVoxelChunkData> Data = Existing;
            if (!Data.IsValid())
            {
                Data = MakeShared<FVoxelChunkData>(Key);
                FVoxelGenerator Gen(Seed);
                Gen.GenerateBaseChunk(Key, *Data);
                VoxelSaveSystem::LoadDelta(Seed, *Data);
            }

            TSharedPtr<FChunkMeshResult> R = MakeShared<FChunkMeshResult>();
            R->Key = Key;
            R->BlockSize = BS;
            R->Data = Data;

            FVoxelMesher_Naive::BuildMesh(*Data, BS, R->V, R->I, R->N, R->UV, R->C, R->T);

            Completed.Enqueue(R);
        });
}

void AVoxelWorldManager::SpawnOrUpdateChunkFromResult(const TSharedPtr<FChunkMeshResult>&
Res)
{
    if (!Res) return;
    if (!IsWithinWorldLimit(Res->Key)) return;

    FChunkRecord& Rec = Loaded.FindOrAdd(Res->Key);
    Rec.Data = Res->Data;

    AVoxelChunkActor* Actor = Rec.Actor.Get();
    if (!Actor || !IsValid(Actor))
    {
        const FVector Origin(
            (double)Res->Key.X * CHUNK_SIZE_X * Res->BlockSize,
            (double)Res->Key.Z * CHUNK_SIZE_Z * Res->BlockSize,
            0.0
        );
        FActorSpawnParameters SP;
        SP.SpawnCollisionHandlingOverride =
ESpawnActorCollisionHandlingMethod::AlwaysSpawn;
        Actor = GetWorldChecked()->SpawnActor<AVoxelChunkActor>(Origin,
FRotator::ZeroRotator, SP);
        if (!Actor) return;
        Actor->BlockSize = Res->BlockSize;
        Rec.Actor = Actor;
    }

    Actor->BuildFromBuffers(Res->V, Res->I, Res->N, Res->UV, Res->C, Res->T,
ChunkMaterial);
}

void AVoxelWorldManager::UnloadNoLongerNeeded(const
    TSet<FChunkKey>& Desired)
{
    for (auto& Pair : Loaded)
    {
        const FChunkKey& Key = Pair.Key;
        FChunkRecord& Rec = Pair.Value;

        if (!Desired.Contains(Key))
        {
            if (AVoxelChunkActor* A = Rec.Actor.Get())
            {
                A->Destroy(); Rec.Actor = nullptr;
            }

            if (Rec.Data.IsValid() && Rec.Data->ModifiedBlocks.Num() > 0)
            {
                VoxelSaveSystem::SaveDelta(WorldSeed, *Rec.Data);
            }
        }
    }
}


void AVoxelWorldManager::Tick(float DeltaSeconds)
{
    Super::Tick(DeltaSeconds);

    // Drain a few completed jobs per frame to avoid hitches
    {
        int32 Drain = 0;
        const int32 MaxDrainPerFrame = 4;
        TSharedPtr<FChunkMeshResult> Res;
        while (Drain < MaxDrainPerFrame && Completed.Dequeue(Res))
        {
            Pending.Remove(Res->Key);
            SpawnOrUpdateChunkFromResult(Res);

            // If edits landed while job was running, resubmit now
            if (FChunkRecord* Rec = Loaded.Find(Res->Key))
            {
                if (Rec->bDirty)
                {
                    Rec->bDirty = false;
                    KickBuild(Res->Key, Rec->Data);
                }
            }
            ++Drain;
        }
    }

    TimeAcc += DeltaSeconds;
    if (TimeAcc < UpdateIntervalSeconds) return;
    TimeAcc = 0.f;

    AActor* Target = TrackedActor.IsValid() ? TrackedActor.Get()
        : UGameplayStatics::GetPlayerPawn(GetWorldChecked(), 0);
    if (!Target) return;

    const FIntPoint Center = WorldToChunkXZ(Target->GetActorLocation());

    TSet<FChunkKey> Desired;
    RecomputeDesiredSet(Center, Desired);

    TArray<FChunkKey> DesiredOrdered;
    BuildDesiredOrdered(Center, DesiredOrdered);

    UnloadNoLongerNeeded(Desired);

    int32 Slots = FMath::Max(0, MaxConcurrentBackgroundTasks - Pending.Num());
    for (const FChunkKey& K : DesiredOrdered)
    {
        if (Slots <= 0) break;

        const FChunkRecord* Rec = Loaded.Find(K);
        const bool bHasActor = Rec && Rec->Actor.IsValid();
        if (bHasActor) continue;
        if (Pending.Contains(K)) continue;

        TSharedPtr<FVoxelChunkData> Existing = Rec ? Rec->Data : nullptr;
        KickBuild(K, Existing);
        --Slots;
    }
}
```

```
void AVoxelWorldManager::FlushAllDirtyChunks()
{
    for (auto& Pair : Loaded)
    {
        FChunkRecord& Rec = Pair.Value;
        if (Rec.Data.IsValid() && Rec.Data->ModifiedBlocks.Num() > 0)
        {
            VoxelSaveSystem::SaveDelta(WorldSeed, *Rec.Data);
            Rec.bDirty = false;
        }
    }
}

bool AVoxelWorldManager::WorldToVoxel_Centered(const FVector& World,
    FChunkKey& OutKey, int32& OutX, int32& OutY, int32& OutZ) const
{
    const double BS = (double)BlockSize;

    const int32 GX = FMath::FloorToInt(World.X / BS + 0.5);
    const int32 GZ = FMath::FloorToInt(World.Y / BS + 0.5);
    const int32 GY = FMath::FloorToInt(World.Z / BS + 0.5);

    const int32 CX = FMath::FloorToInt((double)GX / (double)CHUNK_SIZE_X);
    const int32 CZ = FMath::FloorToInt((double)GZ / (double)CHUNK_SIZE_Z);

    OutX = GX - CX * CHUNK_SIZE_X;
    OutZ = GZ - CZ * CHUNK_SIZE_Z;
    OutY = FMath::Clamp(GY, 0, CHUNK_SIZE_Y - 1);

    OutKey = FChunkKey(CX, CZ);
    return true;
}


bool AVoxelWorldManager::WorldToVoxel_ForEdit(const FVector& World,
    FChunkKey& OutKey, int32& OutX, int32& OutY, int32& OutZ) const
{
    return WorldToVoxel(World, OutKey, OutX, OutY, OutZ); // forward to your existing
mapper
}

bool AVoxelWorldManager::RemoveBlock_Local(const FChunkKey& Key, int32 X, int32 Y, int32 Z)
{
    FChunkRecord* Rec = Loaded.Find(Key);
    if (!Rec || !Rec->Data.IsValid()) return false;

    Rec->Data->SetBlockAt(X, Y, Z, EBlockId::Air);
    Rec->bDirty = true;
    if (!Pending.Contains(Key))
        KickBuild(Key, Rec->Data);

    return true;
}

bool AVoxelWorldManager::PlaceBlock_Local(const FChunkKey& Key, int32 X, int32 Y, int32 Z,
uint8 BlockId)
{
    FChunkRecord* Rec = Loaded.Find(Key);
    if (!Rec || !Rec->Data.IsValid()) return false;

    Rec->Data->SetBlockAt(X, Y, Z, static_cast<EBlockId>(BlockId));
    Rec->bDirty = true;

    if (!Pending.Contains(Key))
        KickBuild(Key, Rec->Data);

    return true;
}


void AVoxelWorldManager::EndPlay(const EEndPlayReason::Type EndPlayReason)
{
    // Save edits for all still-loaded chunks before we go away
    FlushAllDirtyChunks();

    Super::EndPlay(EndPlayReason);
}
```