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.

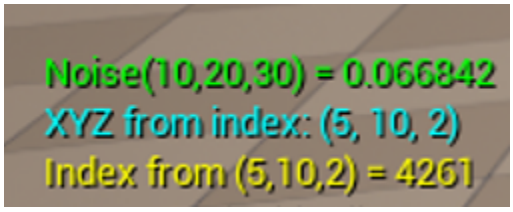
Phase 1 - Project & Tooling Setup (Foundations)

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):



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: 6666667777777777
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):

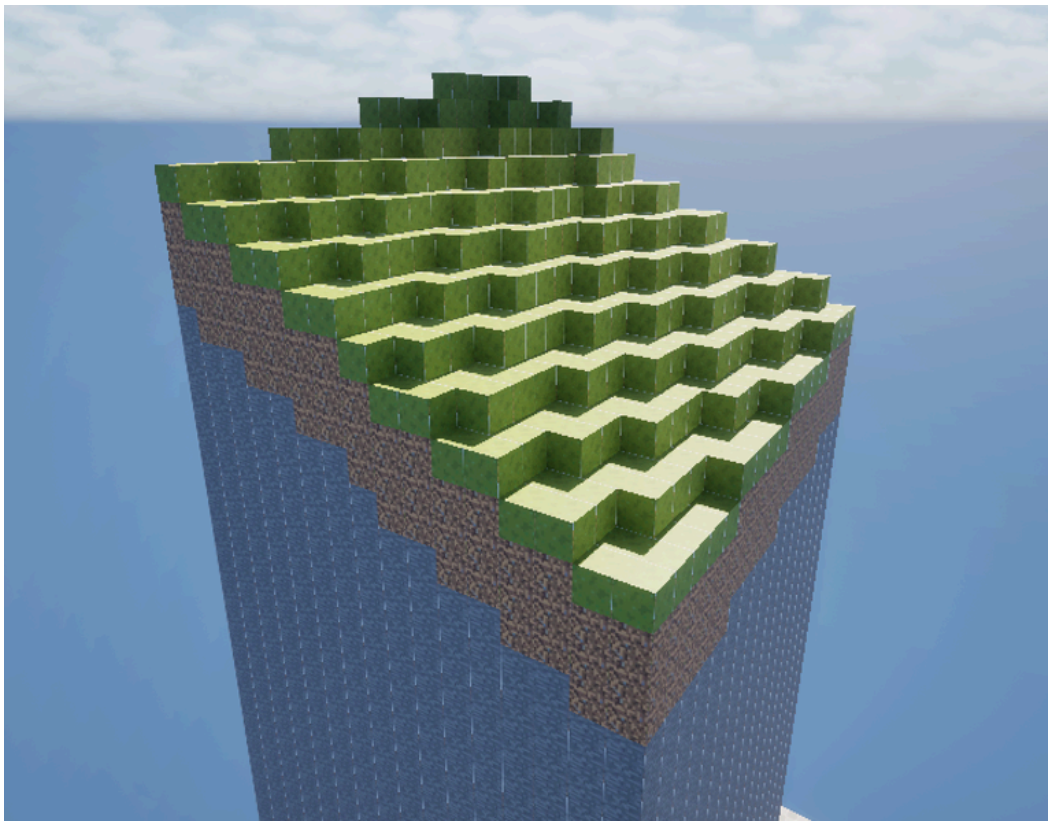
The command successfully spawns a chunk at world origin with visible voxels textured via the atlas map. Faces between solid blocks are culled.

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 - World Saving/Loading From Menu (Player Persistence) & Client-Side Chunk Rendering

Goal: Persist worlds by **world name** (not seed), including all chunk data and per-player positions/state, and render chunks client-side while keeping the server authoritative for collision and data. Late joins resume exactly where the player left off; hosts and clients see identical terrain once their visual world streams in.

Implementation: World persistence by name: Introduce a world folder layout under Saved/Voxel/Worlds/<WorldName>/. Chunk edits are stored as **delta-only** files keyed by chunk (x, z) ; player blobs. (Players/PlayerID.sav stores the last Transform and BP SaveGame bytes for each player. Helper API (UWorldPersistenceLibrary) exposes:

- WorldExists, ListWorlds, CreateWorldFolders
- SavePlayerFromCharacter (WorldName, PlayerID, Character)
- GetLastPlayerTransform(WorldName, PlayerID, OutTransform)
- GetSurfaceSpawnAtOrigin(Seed, BlockSize, OutTransform)
- AreChunksReadyAroundLocationClamped(WorldManager, Location, ClampRadius)

Blueprint Breakdown (GameMode and Widgets):

- Main Menu: Create world -> validate name uniqueness -> store **WorldName** and **Seed** in Game Instance.
- Level Load (GameMode BeginPlay): Cast to Game Instance and retrieve the WorldName and Seed that we set in the main menu. Then spawn actor BP_VoxelWorldManager with the given seed and world name as well as update interval of 0.05 and max concurrent background tasks of 32. Set bool Client Visual Instance and bool Render Meshes to false so it's only used for collision and physics on all players ends. We keep the tracked actors array empty for now. Promote this worldmanager to a variable for later use, and set it to replicate.
- (GameMode) OnPostLogin: First switch on has authority, and if authority grab the player controller and cast to BP_FirstPersonPlayerController. Add this to an array of PCs called All PCs. Then, call the "Get Last Player Transform" function, taking in the world name and the "Get Object Name" of the PC (for now, this works, but when we expand to Steam we will change that to the player's Steam ID). We branch on this, where true means a player with that name has played before and false means this player has not played this world. True - We call Spawn Actor From Class BP_FirstPersonCharacter at the transform we got from the "Get Last Player Transform" node. Then we immediately add it to the world manager's tracked actor array using the "Add Tracked Actor" function. Then, we call the 'Are Chunks Ready Around Location Clamped' taking in that spawned character's get actor location node, as well as a clamped radius of 5. We branch from this where if false we delay for 1 second and try again, and true we continue to then possess the player controller and call a function on that controller called "CLT Spawn Local World Generator" (explained later). False - We call 'Get Surface Spawn at Origin' taking in the seed and block size of 50. We branch on this output, where true means we found the surface spawn at 0,0 on that chunk. If false we just print 'couldn't locate surface spawn at origin'. On true we immediately spawn actor from class BP_FirstPersonCharacter at the transform we receive from the 'Get Surface Spawn at Origin'. Then we immediately add it to the world manager's tracked actor array using the "Add Tracked Actor" function. Then, we call the 'Are Chunks Ready Around Location Clamped' taking in that spawned character's get actor location node, as well as a clamped radius of 5. We branch from this where if false we delay for 1 second and try again, and true we continue to then possess the player controller and call a function on that controller called "CLT Spawn Local World Generator" (explained later). CLT Spawn Local World Generator - This function runs on client only (reliable). It takes Block Size, World Seed and World Name as inputs. First we branch off is Local Controller to ensure it runs locally only. Then, we spawn our own BP_VoxelWorldManager inputting block size, world seed and world name. We set Client Visual Instance and Render Meshes to true. Then we set collision off and hide the actor in game. This will create a visualized client-side render of the chunks that specific client is near, allowing players to only view the chunks they are each around. So all players rely on collision and physics from the

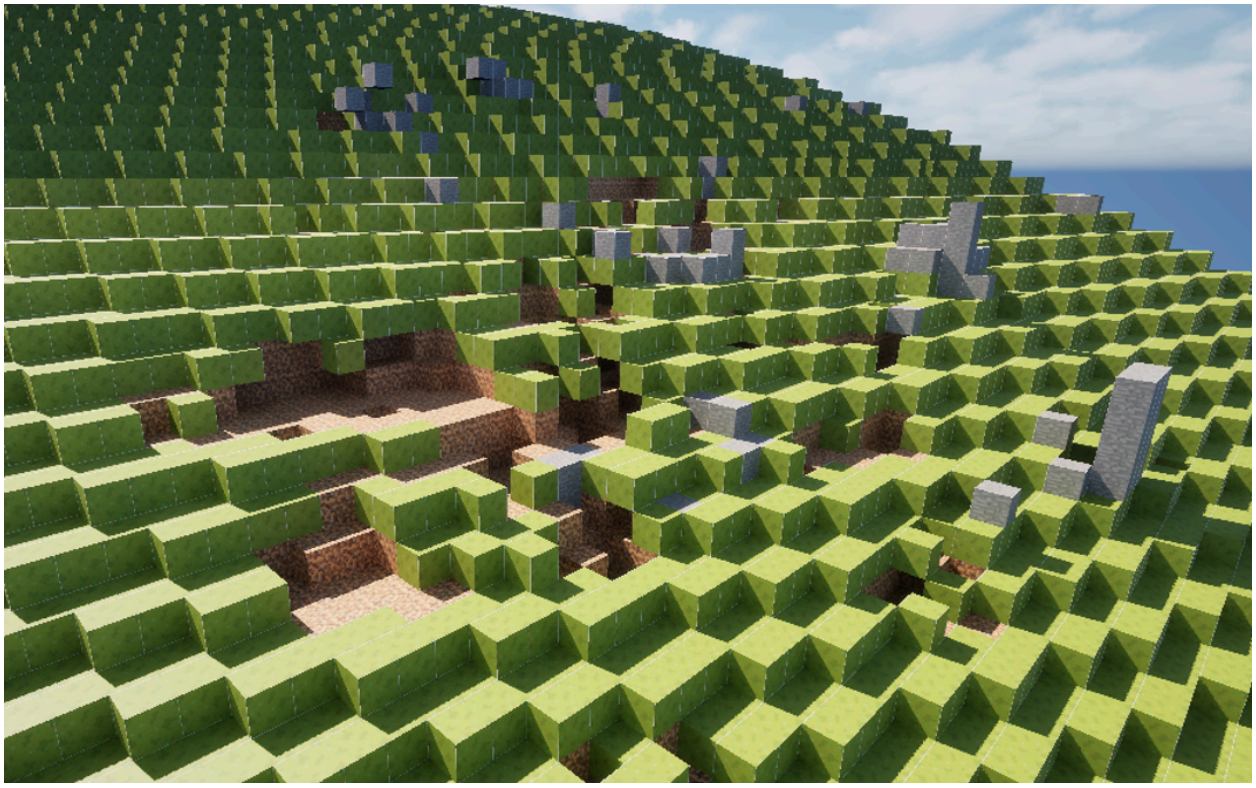
main VoxelWorldManager spawned in the GameMode, but that is visibly hidden, they use the client-side spawned one for visuals.

Overview: After possession, PlayerController runs a client-only RPC to spawn a local BP_VoxelWorldManager with Client Visual Instance and Render Meshes to true, collision disabled, hidden actor - this instance renders chunks around that player only.

Delta Loading: On world manager startup and as chunks stream, base terrain is generated (GenerateBaseChunk) then call LoadDeltaByWorld(WorldName, Chunk) is applied so visuals and collision reflect edits.

Player Saving: For now, periodically call SavePlayerFromCharacter (as well as OnLogout) to persist the player's latest transform and SaveGame fields.

End Goal: Selecting/creating a world in the main menu loads that world's terrain and player states. Each joining player spawns at their last saved position (or surface at origin for first-time joiners). The server keeps authoritative chunk data and collision while each client renders its own nearby chunks with a local world manager for smooth visuals. System has been set up to make it so edits saved in prior sessions appear consistently once chunks stream in; memory remains stable thanks to delta-only chunk persistence.

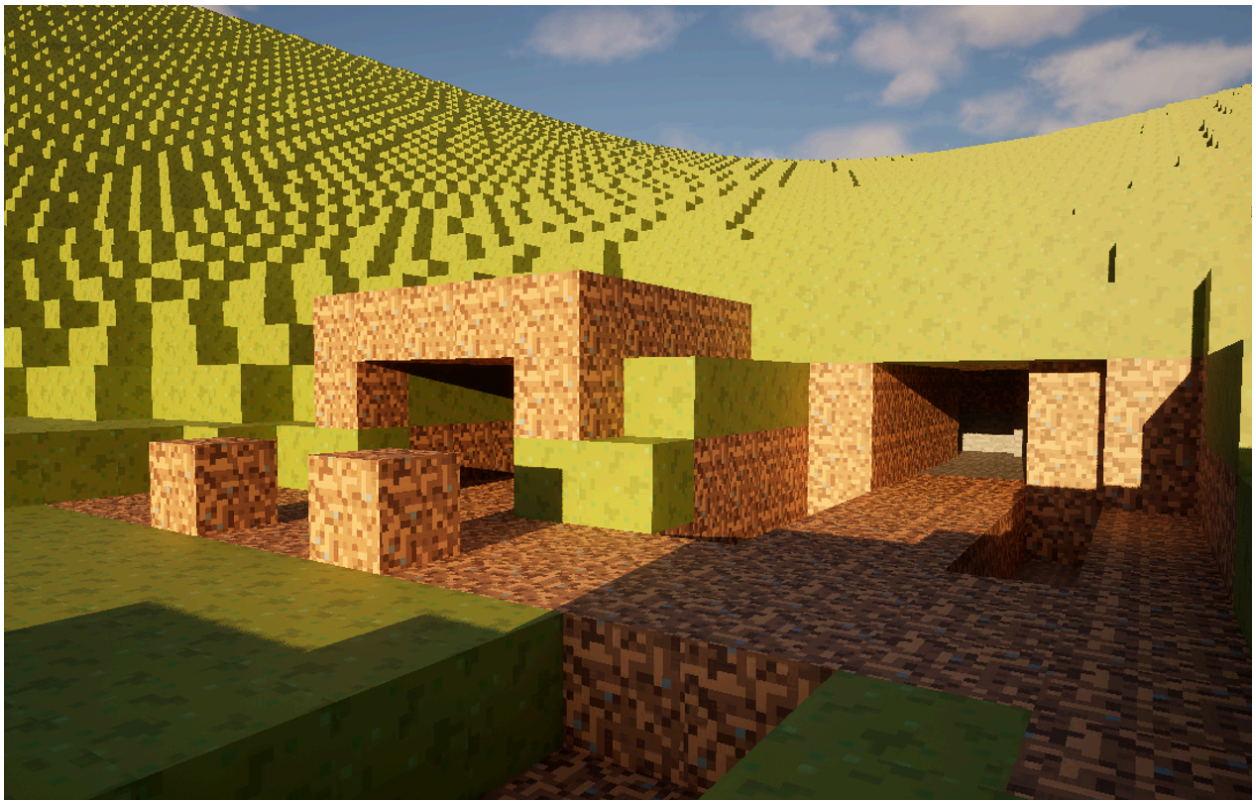


Phase 6 - Altering Chunks (Placing/Destroying)

Goal: Create a function that we can use on possessed BP_FirstPersonPlayerCharacter to left click to place a block of choice. Line trace from player to hit placement location, spawn block of choice in correct location on face accurately. Right click should complete a similar task, but simply remove the block that is hit with the line trace. Apply to the authoritative server world (collision/physics) and replicate to all clients' local visual worlds immediately. Late joiners see all edits made before they connect. Persist exactly like other world data using delta savers to the disk. Stay in sync under rapid actions - no desync, no bugs.

Implementation: Make the server the single source of truth for voxel data and collision and let each player render a client-visual world for representation as done previously. A click is resolved to a cell with VoxelWorldManager::ResolveVoxelFromHit (face-biased, centered mapping) and applied on the server via ApplyBlockEdit_Server which loads/generates chunk data on demand (EnsureChunkDataLoaded_ForEdit), marks the chunk dirty, rebuilds (KickBuild), and invalidates border neighbors. Each streamed chunk on the server now owns a lightweight replicated actor, AVoxelChunkNetState, which carries a FastArray of modified cells (FModifiedCellArray). The server records edits with ServerApplyCellChange(LocalIndex, BlockID) and FastArray replication drives ApplyCellOnClient on recipients, which forwards to the client-visual manager ApplyVisualOpOrQueue(ChunkXZ, LocalIndex, BlockID) - queueing if the chunk hasn't streamed yet. On clients, when a chunk mesh finishes (SpawnOrUpdateChunkFromResult), we first apply any pending replicated edits: if we apply any, we skip drawing the stale buffers and immediately rebuild so visuals always match authoritative data. For late joiners and for clients arriving to far-away edits, the PlayerController exposes RequestInitialChunkSnapshotsBP which calls Server_RequestInitialChunkSnapshots() / Server_RequestInitialChunkSnapshotsForArea(), which the server answers with Client_ReceiveChunkSnapshot and the client ingests those ops with ApplyOrQueueClientOps(). To eliminate rare prediction drift during rapid clicking, after each server apply we also send back the authoritative cell values and the client reconciles immediately via IngestReplicatedCells (ChunkZX, Cells). Persistence remains delta-only and deltas are written on unload or shutdown.

End Goal: Players can left click to place blocks and right click to remove blocks. As blocks are placed/removed, changes are shown visibly to all connected players alike. Changes apply to server-side world managers to reflect accurate collision and physics as well as everyone's individual client-side world manager to reflect accurate visuals. The world can be closed, re-loaded and all changes made are retained and visible.



Phase 6.5 - Improved FPS While Distant Chunks Are Rendering

Goal: Eliminate brief frame-time dips that occurred when a new ring of chunks flipped from **queued** to **meshed** to **spawned**, enabling smooth play at larger render radii (12-20) without changing visuals, collisions, or shadows.

Implementation: Time-budget draining of completed chunk meshes on the game thread; stop draining once per-frame budgets are hit (time, vertex count and item count). **Capped enqueues per tick** so we don't start too many builds at once when a new ring appears (prevents many jobs completing in the same frame). **Async unload saves** moves delta-save writes off the game thread during chunk unload (final synchronous flush still happens on EndPlay). **Ray tracing opt-out for voxel chunks** to avoid BLAS memory churn at high radii.

End Goal: Maintain near-baseline FPS while traversing the world with higher render radii; make chunk pop-in gradual over a few frames instead of a single-frame spike and remove RT memory warnings at large view distances.

Results: FPS drops at ring boundaries are significantly reduced, play is smooth at 12-16 radii, and radii 18-20 is very playable with only rare, tiny micro-stutters. Overall frame pacing improved without visual compromises; future tuning is now simple via per-instance budgets.