Awesome! I will create a detailed, step-by-step roadmap specifically for building a **classic Minecraft clone in C++** on **Windows** using **Visual Studio Community Edition**, with **existing libraries** (like OpenGL, SDL2, etc.), focused on making a **minimal custom game engine** that is **expandable**, and tuned for a **mid-level programmer**.

I'll cover:

* A structured learning path (what to learn in what order)
* Key systems to implement (rendering, world generation, physics, etc.)
* Architectural choices (how to organize your code)
* Library and tool recommendations
* How to split development into phases (small milestones)

I'll get started and let you know when it's ready!

**Building a Minecraft-Style Voxel Game in C++: A Step-by-Step Roadmap**

This roadmap will guide you through creating a **minimal Minecraft-like game** (voxel engine) from scratch in C++ on Windows, using **Visual Studio Community Edition** and libraries like **SDL2** and **OpenGL**. We’ll follow a phased approach with clear milestones, focusing on practical steps and good practices. By the end, you’ll have a basic engine that can render a blocky world, handle player movement and block interactions, and that you can expand with more features.

**Note:** This project is complex but very achievable by iterating in small steps. Even large voxel games are built incrementally – *“It is easy to start with something very basic and slowly build up to a more complex and thorough voxel engine with lots of little steps”* ([Let's Make a Voxel Engine](https://sites.google.com/site/letsmakeavoxelengine/home#:~:text=engine%20takes%20time%20and%20effort%2C,tasks)). We will break down tasks into phases and sub-tasks so you can tackle them one by one.

**Phase 1: Learning and Setup (Preparation)**

Before coding the game, invest time in **learning the key technologies** and setting up your development environment. As a mid-level programmer, you should be comfortable with C++ basics; now you’ll focus on the specific skills and tools for a 3D voxel game.

* **Install and Configure Visual Studio:** Ensure you have **Visual Studio Community Edition** installed with the C++ development workload. Familiarize yourself with creating a new C++ project, setting include/lib directories, and using the Visual Studio debugger.
* **Learn SDL2 (Simple DirectMedia Layer):** SDL2 will handle window creation, input, and possibly audio. Learn how to initialize SDL and open a window, how to poll for input events (keyboard/mouse), and how to play sounds (SDL\_mixer if needed). A good resource is the official SDL documentation and tutorials. (For example, Lazy Foo’s SDL2 tutorials cover window creation, input, and more ([GitHub - jmparis/sdl2-lazy-foo: SDL2 Tutorial from Lazy foo site](https://github.com/jmparis/sdl2-lazy-foo#:~:text=50%20SDL%20and%20OpenGL%202,1%20program)).) Make sure you can write a simple SDL program that opens a window and captures keyboard input.
* **Learn Modern OpenGL (3D Rendering):** Our rendering will use OpenGL for drawing the 3D world. You should understand the basics of the OpenGL **graphics pipeline** (how 3D geometry is transformed into pixels on screen). In modern OpenGL, you will write shaders (in GLSL) that run on the GPU to handle these transformations ([LearnOpenGL - Hello Triangle](https://learnopengl.com/Getting-started/Hello-Triangle" \l ":~:text=In%20OpenGL%20everything%20is%20in,advantage%20to%20create%20fancy%20pixels)). Key topics to learn:
  + Creating an OpenGL **context** (we’ll use SDL for this). The steps are: **initialize SDL**, **set SDL GL attributes** (context version, depth buffer, etc.), **create a window with SDL\_WINDOW\_OPENGL**, and **create an OpenGL context** with SDL\_GL\_CreateContext ([Tutorial1: Creating a Cross Platform OpenGL 3.2 Context in SDL (C / SDL) - OpenGL Wiki](https://www.khronos.org/opengl/wiki/Tutorial1:_Creating_a_Cross_Platform_OpenGL_3.2_Context_in_SDL_(C_/_SDL)#:~:text=Creating%20a%20window%20in%20SDL,to%20it%20uses%20these%20steps)).
  + **Drawing primitives** (triangles) using Vertex Buffer Objects (VBOs) and Vertex Array Objects (VAOs).
  + Writing a basic **vertex and fragment shader** to render a colored triangle.
  + Enabling **depth testing** (so nearer objects obscure farther ones).

*Recommended resources:* the official OpenGL tutorials (e.g. the [OpenGL Wiki tutorial on SDL context creation] ([Tutorial1: Creating a Cross Platform OpenGL 3.2 Context in SDL (C / SDL) - OpenGL Wiki](https://www.khronos.org/opengl/wiki/Tutorial1:_Creating_a_Cross_Platform_OpenGL_3.2_Context_in_SDL_(C_/_SDL)#:~:text=Creating%20a%20window%20in%20SDL,to%20it%20uses%20these%20steps))) and the **LearnOpenGL** guide for modern OpenGL, which covers creating a window, drawing a triangle, shaders, etc.

* **Math and 3D Concepts:** Refresh your knowledge of **3D math** – vectors, matrices, and transformations. You’ll use these for camera movement and placing blocks in the world. It’s recommended to use a library like **GLM (OpenGL Mathematics)** for matrix and vector operations so you don’t have to write your own. Ensure you understand coordinate systems (world space, view/camera space, screen space) and how a **perspective projection** works to simulate 3D depth.
* **Familiarize with Voxel Engine Basics:** Read up on what a voxel engine is and how Minecraft-style worlds work:
  + A **voxel** is a 3D block (like a 1x1x1 cube in the world). The game world is a grid of voxels.
  + Voxels are often managed in 3D chunks (e.g. 16×16×16 blocks) for efficiency.
  + Typical goals: being able to render many cubes efficiently, modify blocks in real-time, and have a large (potentially infinite) world ([Let's Make a Voxel Engine](https://sites.google.com/site/letsmakeavoxelengine/home#:~:text=The%20main%20goals%20and%20ambitions,voxel%20engine%20are%20as%20follows)).

You can skim articles or guides on voxel engines for conceptual understanding. For instance, knowing that you should only render the *visible faces* of blocks (not the ones hidden against other blocks) will be important for performance ([Let's Make a Voxel Engine - 06 - Chunk Optimizations](https://sites.google.com/site/letsmakeavoxelengine/home/chunk-optimizations#:~:text=Don%E2%80%99t%20Render%20Unseen%20Triangles)).

* **Plan Project Structure:** At this stage, outline how you will organize your code. Plan to create modules/classes for major systems: e.g., *Window/Graphics*, *Input*, *World/Chunk*, *Player*, etc. (We will detail this in Phase 3.) The idea is to keep the code modular and manageable.
* **Set Up Libraries:** Using a package manager can simplify this. Consider using **vcpkg** (Microsoft’s C++ package manager) to install SDL2, GLM, and GLEW/GLAD (for OpenGL function loading). For example, after installing vcpkg, you can install SDL2 with a single command, and integrate it with Visual Studio easily. Alternatively, manually download these libraries:
  + *SDL2:* Get the development libraries for Visual C++ from the SDL website. Link against SDL2.lib and include the SDL headers.
  + *OpenGL loader:* Windows’ default OpenGL is outdated, so use **GLEW** or **GLAD** to access modern OpenGL functions. (GLEW is a library you can initialize after creating the GL context; GLAD is a set of headers you generate for the GL version you need.)
  + *GLM:* Header-only, just include it.
  + *SDL\_mixer (optional):* for audio, if you plan to add sound.
* **Small Validation Project:** As a milestone for Phase 1, create a trivial program to ensure everything is set up: for example, open an SDL window, create an OpenGL context, clear the screen with a color, and handle an event to close the window. If this works, you’re ready to move on.

*Milestone:* You have a working C++ project in Visual Studio that opens an OpenGL window using SDL2. You have basic knowledge of how rendering and input work, and you’ve prepared the libraries needed. All further development will happen in this project.

**Phase 2: Prototype – Window, Graphics & Input Demo**

In this phase, you will build a simple prototype application to verify that your learning translates into working code. The goal is to create a minimal program that opens a window, renders a simple 3D object, and lets you control a camera. This “hello world” of the engine ensures that SDL2 and OpenGL are correctly integrated and that you can move on to building the engine proper.

**Step 2.1: Create a Window with OpenGL Context** – Start by writing code to initialize SDL and OpenGL:

* **Initialize SDL:** Call SDL\_Init(SDL\_INIT\_VIDEO | SDL\_INIT\_AUDIO) (include audio if you plan to use sound). Set SDL attributes for OpenGL context version (e.g., 3.3 core) using SDL\_GL\_SetAttribute. Also request a double-buffered window and a depth buffer. For example:
* SDL\_GL\_SetAttribute(SDL\_GL\_CONTEXT\_MAJOR\_VERSION, 3);
* SDL\_GL\_SetAttribute(SDL\_GL\_CONTEXT\_MINOR\_VERSION, 3);
* SDL\_GL\_SetAttribute(SDL\_GL\_CONTEXT\_PROFILE\_MASK, SDL\_GL\_CONTEXT\_PROFILE\_CORE);
* SDL\_GL\_SetAttribute(SDL\_GL\_DOUBLEBUFFER, 1);
* SDL\_GL\_SetAttribute(SDL\_GL\_DEPTH\_SIZE, 24);

Then create the window: SDL\_Window\* window = SDL\_CreateWindow("Voxel Game", SDL\_WINDOWPOS\_CENTERED, SDL\_WINDOWPOS\_CENTERED, 1280, 720, SDL\_WINDOW\_OPENGL | SDL\_WINDOW\_RESIZABLE); ([Tutorial1: Creating a Cross Platform OpenGL 3.2 Context in SDL (C / SDL) - OpenGL Wiki](https://www.khronos.org/opengl/wiki/Tutorial1:_Creating_a_Cross_Platform_OpenGL_3.2_Context_in_SDL_(C_/_SDL)#:~:text=Creating%20a%20window%20in%20SDL,to%20it%20uses%20these%20steps)). After that, create the GL context: SDL\_GLContext glContext = SDL\_GL\_CreateContext(window); and ensure it’s made current. If any of these steps fail, print an error (SDL provides SDL\_GetError()).

* **Load OpenGL Functions:** If using GLEW, call glewInit() after creating the context. If using GLAD, ensure you included the generated loader before any GL calls. This gives access to functions like glCreateShader etc.
* **Basic OpenGL Setup:** Set a clear color with glClearColor(r,g,b,1.0f) and enable depth testing (glEnable(GL\_DEPTH\_TEST)). Also enable face culling (glEnable(GL\_CULL\_FACE) and glCullFace(GL\_BACK)) – this means OpenGL will skip drawing back-faces of polygons, which is good since we never need to see the inside faces of cubes. We will also rely on it when rendering voxel faces.

**Step 2.2: Render a Test Object** – Verify that rendering works by drawing a simple shape:

* Write a basic vertex shader and fragment shader, compile and link them into a shader program. The vertex shader can output a fixed color or pass through a color attribute; the fragment shader outputs the color.
* Define a simple geometry to draw. Start with a triangle (the classic OpenGL demo). Define 3 vertices in a VBO and configure a VAO. Alternatively, to test 3D, you can define a cube’s 8 vertices and 12 triangles. But a single triangle is enough to validate the pipeline.
* In your main loop (discussed below), call glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT), then bind your shader program and VAO, and call glDrawArrays(GL\_TRIANGLES, 0, 3) (for example) to draw the triangle. Swap the window buffers with SDL\_GL\_SwapWindow(window).

If you see the triangle on the screen, congrats – OpenGL is rendering correctly! If not, use this stage to debug any issues (shader compilation errors, wrong coordinate causing triangle not visible, etc.). Common mistakes include missing calls to SDL\_GL\_SwapWindow (so nothing shows), forgetting to compile/link shaders, or not using the correct GL clear flags for color/depth.

**Step 2.3: Basic Input & Camera Control** – Now integrate input so you can interact:

* **Input Loop:** Use SDL event polling (SDL\_PollEvent). Handle SDL\_QUIT event or Escape key to close the window. Also handle keyboard events for movement. For now, you can implement a simple scheme: e.g., arrow keys or WASD to move the camera position on the XZ plane, and maybe keys to move up/down (for flying). You might also capture the mouse motion (SDL\_MouseMotionEvent) for looking around.
* **Camera Setup:** For now, implement a simple **free-fly camera**. Maintain variables for camera position (camX, camY, camZ) and orientation angles (yaw, pitch). Each frame, adjust these based on input (e.g., W moves forward in the direction of yaw, A strafes left, etc., and mouse movement changes yaw/pitch). Then construct a basic view matrix: e.g., use GLM’s lookAt() with eye = camera pos, center = pos + forwardVector, up = world up. Also set up a projection matrix (perspective) once, using e.g. glm::perspective(fov, aspect, near, far).
* **Integrate with Rendering:** Pass the view and projection matrices to your shader (as uniform matrices). In the vertex shader, multiply them with your vertex positions so the triangle (or cube) appears in the correct place relative to the camera. This way, moving the camera with input will appear to move around the object.

For simplicity in the prototype, you could skip actual matrix math by just moving the triangle’s coordinates manually or something, but it’s better to set up the proper camera now as practice. With GLM, it’s not too difficult.

Now you have a loop that processes input, updates camera, and renders a frame each time – essentially a minimal **game loop**. The core pattern of a game loop is: **process input, update game state, render output**, repeatedly ([Programming Patterns for Games: Game Loop - DEV Community](https://dev.to/zigzagoon1/programming-patterns-for-games-game-loop-4goc#:~:text=The%20simplest%20possible%20game%20loop,looks%20something%20like%20this)). We will formalize this loop in the engine design soon.

**Prototype Milestone:** You should now have a program where you can see a triangle (or simple object) and use keyboard/mouse to move the camera around. This confirms that:

* SDL2 can create a window and capture input events.
* OpenGL is rendering to the window.
* Your development environment is correctly set up for C++ compilation and linking with these libraries.
* You have a grasp of the basic runtime loop (input -> update -> render).

Take time to clean up and organize this prototype code a bit, as it will evolve into your engine. Once the prototype works, commit the working version in source control (we highly recommend using Git from this point – *“Yes, you should always use source control. Yes, even if you’re a solo developer working on a hobby project.”* ([What is the best Source Control for game development ? - DEV Community](https://dev.to/kahncode/what-is-the-best-source-control-for-game-development-5g0p#:~:text=Let%27s%20get%20one%20thing%20out,if%20you%20are%20a%20beginner)) It will save you from losing progress and help track changes).

**Phase 3: Designing the Core Engine Structure**

With the groundwork laid, it’s time to design a simple *game engine architecture* for your clone. This means deciding on how to structure your code into logical systems and establishing the main loop that ties everything together. The engine will remain minimal – just enough to support Minecraft-like gameplay – but we’ll set it up in a modular way so you can extend it later.

**3.1 – Core Engine Architecture Overview:** The engine will be built around a continuous **game loop** that runs until the player exits. Each iteration of the loop will:

1. Handle **Input** (read player commands like movement or block placement),
2. **Update Game State** (move the player, apply physics like gravity, generate or unload world chunks as needed, etc.),
3. **Render** the current state (draw the world and the player’s view of it to the screen).

This is the classic game loop pattern ([Programming Patterns for Games: Game Loop - DEV Community](https://dev.to/zigzagoon1/programming-patterns-for-games-game-loop-4goc#:~:text=The%20simplest%20possible%20game%20loop,looks%20something%20like%20this)). We will likely also regulate the loop timing (to limit framerate or to keep physics updates at a fixed timestep) – more on that shortly.

**3.2 – Separating Systems (Modules):** Organize the code into classes or modules for clarity:

* **Application/Engine**: High-level class (or just your main) that initializes everything and runs the loop.
* **Renderer/Graphics System**: Responsible for all OpenGL calls. It should know how to draw the world (chunks/blocks) and any UI elements. It can encapsulate the shader programs, textures, and camera matrices. For example, you might have a Renderer class with a render(World& world, Player& player) method.
* **World/Chunk Manager**: Manages the world data. We will create a World class that contains a collection of Chunk objects. The world provides access to block data at given coordinates, generates new chunks as needed, and handles chunk updates (like when blocks change). We’ll define chunk shortly.
* **Player/Entity**: The player is an entity with position, orientation, and possibly velocity (for physics). A Player class can hold these and have methods to process input or apply gravity. Later, you might generalize to an Entity system if adding mobs, but for now a single Player class is fine.
* **Input Handler**: You can keep input handling in the main loop, but it’s good to wrap it in a module (even if it’s just a set of functions) that reads SDL events and updates flags (e.g., bool movingForward, bool jumping, etc., in the Player or a separate InputState structure).
* **Audio System** (optional now): If you plan sound, an Audio manager can initialize SDL\_mixer, load sound files, and expose functions to play a sound (like block break sound).

Think of the **engine architecture** as a collection of subsystems working together. For instance: the Input system tells the Player to jump; the Player’s movement is checked against the World (collision); the World provides the Renderer with chunk meshes to draw; the Renderer draws them. Keeping these concerns separate makes the code cleaner and easier to maintain.

**3.3 – Game Loop and Timing:** Implement the main loop in a controlled manner:

* Decide on a frame rate or tick rate. You might allow rendering to run uncapped (as fast as possible) or vsynced to monitor refresh, but you may want the game logic (updates) to run at a fixed rate (e.g., 60 updates per second) for consistent physics. A common approach is a fixed timestep for updates and interpolation for render, but as an initial implementation, you can use a simpler method: tie logic and rendering together with a small delta time.
* For example:
* Uint32 prevTime = SDL\_GetTicks();
* while (running) {
* Uint32 currTime = SDL\_GetTicks();
* float delta = (currTime - prevTime) / 1000.0f;
* prevTime = currTime;
* handleInput();
* updateGame(delta);
* renderFrame();
* }

This uses a variable timestep (delta) for updates, which is easier to start with. The player movement and other physics will be scaled by delta to be frame-rate independent. (Later, you can refine this with a fixed timestep loop to avoid inconsistencies.)

* **VSync or Frame Cap:** Using SDL\_GL\_SetSwapInterval(1) can enable vsync to cap the frame rate and reduce CPU usage. If not using vsync, you might manually delay or cap the loop to ~60 FPS. In any case, ensure the loop doesn’t run unbounded and consume 100% CPU without need.

**3.4 – Implementing the Skeleton:** Start coding the structure:

* Create the classes (Engine, World, Chunk, Player, etc.) with just basic members. For example, struct Chunk { /\* will hold blocks and mesh data \*/ };, class World { std::map<ChunkCoord, Chunk> chunks; /\* and methods \*/ };, class Player { glm::vec3 position; glm::vec3 velocity; ... };.
* Setup an instance of World, Player, Renderer in main. Initialize the World (maybe create one dummy chunk for now), set Player starting position (e.g., at the world’s surface).
* **Input handling:** perhaps create a simple function or class that processes SDL events and updates the Player’s desired movement (e.g., setting a velocity or directly adjusting position). For now, you can move the player noclip (ignoring collisions) – we will add proper collisions in Phase 4.
* **Renderer:** Prepare it to handle drawing the world. For now, you can reuse the code from the prototype to draw something, but soon we’ll give it actual chunk data to draw.
* The Renderer might need the camera’s position/orientation (from Player) to build the view matrix each frame. So decide if the Renderer will compute that internally given the Player, or if Player produces a camera matrix for the renderer.
* Make sure to also handle window events like resize (you might call glViewport to adjust to new window size if needed).

At this point, you should have the basic engine loop running: an empty world, a controllable player camera, and a renderer clearing the screen each frame (maybe still drawing the test triangle or nothing yet). The code is structured into logical units (even if the implementations are still stubbed out). This is a good time to double-check that your code organization makes sense and that it will support the next steps where we add real game features.

**Good Practices at this stage:**

* **Keep it Simple:** Don’t over-engineer the architecture. A few well-defined classes are enough. (Avoid unnecessary complex patterns until you need them.)
* **Use Source Control:** Commit your skeleton engine. This way, if a future change breaks something, you can diff and revert easily ([What is the best Source Control for game development ? - DEV Community](https://dev.to/kahncode/what-is-the-best-source-control-for-game-development-5g0p#:~:text=Let%27s%20get%20one%20thing%20out,if%20you%20are%20a%20beginner)).
* **Logging:** Consider adding a basic logging utility (even just writing to stdout or a file) to record events, which helps debugging. For example, log when chunks are created or when the player’s position updates.
* **Error Handling:** Check for errors in initialization (SDL or OpenGL) and either handle or output clear messages. It’s easier to debug early if you get clear error logs.

**Phase 4: Implementing Core Game Mechanics**

Now comes the fun part – turning the engine into a Minecraft-style game! We will implement features in a sensible order, focusing on one piece at a time. The priority order of features will be:

1. **World Representation & Generation** – how blocks are stored and world terrain is created.
2. **Rendering the World** – drawing the blocks efficiently.
3. **Player Movement & Physics** – moving in the world with gravity and collisions.
4. **Block Interactions** – breaking and placing blocks.
5. **Basic Game Mechanics** – (optional additions) multiple block types, simple lighting, etc.

By adding these one by one, you can test and ensure each works before moving on.

**4.1 World Representation and Terrain Generation**

* **Chunk Data Structure:** Define the size of a chunk (Minecraft uses 16×256×16, but you can choose a smaller height for simplicity, e.g. 16×64×16). Create a Chunk class that contains a 3D array of block identifiers. Each **block** can be represented by a small type (e.g., uint8\_t or an enum) that indicates what kind of block it is (0 = air, 1 = dirt, 2 = grass, etc.). Also include a flag or separate structure to mark if a block is active/solid; however, if 0 is air, then "non-zero means active" is enough.
* **Coordinate System:** Decide how you refer to blocks in the world. Commonly:
  + Each chunk has an (cx, cy, cz) index in chunk-space (where cx,cz cover horizontal plane, cy might be always 0 if you don't stack chunks vertically initially).
  + Within a chunk, blocks have (bx, by, bz) from 0 to CHUNK\_SIZE-1.
  + You can map global world coordinates to chunk + local index via division/modulus. E.g., given world (wx, wy, wz), cx = floor(wx/CHUNK\_SIZE), bx = wx mod CHUNK\_SIZE (with care for negatives).
  + For now, you might keep it simple by only having one chunk or a fixed grid of chunks, but planning for coordinate translation now will help expand later.
* **Terrain Generation:** Start with something simple:
  + **Flat world:** E.g., fill blocks below a certain Y level with stone/dirt and above that with air. For instance, generate a flat ground at y=0 or y=10 within the chunk.
  + Once a flat world works, you can introduce a heightmap using noise: use a 2D noise function (Perlin or Simplex noise) to determine ground height at each (x,z). You could use an existing noise library or write a simple Perlin noise implementation. For example, the **Simplex noise** algorithm is popular for terrain and was used in some open-source clones ([GitHub - swr06/Minecraft: A Tiny Minecraft clone made with C++ and OpenGL.](https://github.com/swr06/Minecraft#:~:text=,such%20as%20Glass%20and%20models)). You input (x,z) and get a pseudo-random height value. By tweaking frequency and amplitude you can create rolling hills. Fill blocks below that height with ground material and above with air.
  + You can also add variation like different block types at different heights (e.g., top layer grass, beneath that dirt, then stone).
  + Ensure that chunk generation is deterministic (use the chunk’s world coordinates as seed input to noise) so it always produces the same terrain for the same location.
* **Implement World/Chunk in Code:** Add a method in World like generateChunk(int cx, int cz) that allocates a new Chunk, fills its block array according to your terrain generation logic, and stores it in the world’s chunk map. Call this for the chunks you want initially (e.g., generate a 3×3 grid of chunks around the origin, or even just one chunk at (0,0) to start). As a test, you could hardcode one chunk with some interesting shape (like a column or a small hill) to verify rendering later.

At this stage, you have chunks with data but they’re not visible yet. You should implement a way to inspect the chunk data (maybe a debug print that outputs the top layer height map or count of solid blocks) just to verify generation logic. Once confident, move to rendering.

**4.2 Chunk Meshing and Rendering the World**

Rendering every cube by drawing all six faces individually would be extremely slow in pure immediate mode. Instead, we use a **mesh** (collection of vertices) for each chunk, which combines many cubes into a single set of drawing calls. The process is:

* **Mesh Generation:** For each chunk, iterate over each block. If the block is air (empty), skip it. If it’s a solid block, determine which faces are exposed and should be drawn:
  + Check each of the 6 neighbors (±X, ±Y, ±Z). If a neighbor is out of world bounds or is air, then the corresponding face of our block is visible and should be drawn; if the neighbor is solid, then that face is hidden and we skip it ([Let's Make a Voxel Engine - 06 - Chunk Optimizations](https://sites.google.com/site/letsmakeavoxelengine/home/chunk-optimizations#:~:text=One%20easy%20optimization%20we%20can,this%20optimization%20is%20as%20follows)) ([Let's Make a Voxel Engine - 06 - Chunk Optimizations](https://sites.google.com/site/letsmakeavoxelengine/home/chunk-optimizations#:~:text=Don%E2%80%99t%20Render%20Unseen%20Triangles)).
  + For each face that is visible, add its vertices to the chunk’s vertex list. Each face is two triangles (6 vertices if not using indexing, or 4 vertices if using indexed quads). You’ll also add texture coordinates for that face (if using textures) or a color value (if using vertex colors).
  + By doing this, we **don’t create any geometry for faces between two solid blocks**, saving a lot of rendering work ([Let's Make a Voxel Engine - 06 - Chunk Optimizations](https://sites.google.com/site/letsmakeavoxelengine/home/chunk-optimizations#:~:text=One%20easy%20optimization%20we%20can,this%20optimization%20is%20as%20follows)). For example, in a solid 16×16×16 chunk, only the outer surface generates triangles.
  + You can further optimize by **greedy meshing** (merging adjacent faces into one large quad), but that’s an advanced step. Initially, generating face by face is fine.
* **Store Mesh Data:** Common approach is to use a structure holding arrays of vertices, maybe separate array for indices if you use element drawing. For simplicity, you can generate a single large std::vector<float> containing interleaved vertex attributes for all faces in the chunk. Each vertex might contain position (x,y,z), a normal vector (for lighting), and texture UV coordinates.
* **OpenGL Buffer:** Create a VBO for each chunk’s mesh data and a VAO to describe the layout. Load the vertex data into the VBO (glBufferData). This might be done at chunk generation time or at first render. If the world is static, you can do it once. If it can change, you’ll need to regenerate and update the buffer when blocks change (we will handle that when adding block interactions).
* **Shader for Blocks:** Write a shader that uses a model-view-projection matrix to place the chunk geometry in the world. Likely you’ll use world coordinates directly in the vertex positions, so your vertex shader just multiplies proj \* view \* vec4(position,1.0). For fragment shader, apply a texture or color. Initially, you can assign a constant color or use the block type to choose a color (e.g., dirt = brown, grass = green). This can be done by passing a color per face as a vertex attribute. Later, you can use a texture atlas: a single texture containing all block faces tiled, and give each face an appropriate UV range.
* **Drawing:** In your Renderer’s render function, iterate over all chunks in the World (that are in view) and issue a draw call for each chunk’s mesh (e.g., glDrawArrays(GL\_TRIANGLES, 0, vertexCount) after binding that chunk’s VAO and shader). This way, if you have N chunks loaded, you’ll roughly do N draw calls per frame (which is fine for moderate N).

After this step, if everything is wired up, you should see a blocky landscape on the screen! 🎉 The entire chunk (or multiple chunks) of terrain generated should now be visible. Move the camera (player) around to explore it.

**Troubleshooting Rendering:**

* If you see nothing or weird artifacts, check that face culling is correct (you might need to consistent winding order for your face vertices – typically define them clockwise or counterclockwise so that the back faces are culled properly).
* If the terrain looks inside-out, you might be drawing back-faces instead; change the order or disable face culling to debug.
* Ensure the projection matrix is set (a common mistake is to forget to update the shader with camera matrices each frame).
* Check that your chunk vertex buffer was uploaded correctly (maybe start with a very tiny chunk or even a manually defined cube to ensure the pipeline is correct).

**Frustum Culling (optional now):** As you add more chunks, consider not drawing those completely outside the camera’s view. You can compute a chunk’s bounding box and do a frustum check. This can save draw calls. However, if your view distance is not huge and chunk count is limited, you can skip this initially and add it when optimizing.

Now your game shows a static world. Next, we’ll make the player a real part of that world.

**4.3 Player Movement, Physics, and Collision**

Moving the camera freely is nice for debugging, but to mimic Minecraft gameplay, the player needs proper physics: gravity pulling them down, collision so they can walk on blocks and not pass through walls, and jumping. This will greatly enhance the feel of the game.

* **Gravity:** Each frame, if the player is not on the ground, apply a downward velocity (e.g., vy -= 9.81 \* delta for gravity acceleration). The player is essentially a point or capsule that falls until hitting terrain.
* **Collision Detection:** Represent the player as a simple **axis-aligned bounding box (AABB)** – e.g., 0.6 × 1.8 × 0.6 (Minecraft player size). For collision with the voxel world, a common approach is: after moving the player based on input and gravity, check the blocks around the player’s new position. The player’s feet position corresponds to some block (floor) and the head to another.
  + Determine which blocks the player’s AABB overlaps. Because blocks are also axis-aligned and on a grid, you can derive which block indices range the player covers. For example, if player X coordinate is 10.2 and width is 0.6, then they span from x=10 to x=10 (still within one block in x). If width were 1.2 at that position, they might span x=10 and x=11. So, you could iterate over all blocks in the range [floor(x - width/2) .. floor(x + width/2)] for X, similar for Y and Z, and test if any of those blocks is solid.
  + A simpler approximation: check the block at the player’s feet (position rounded down) as the ground, and maybe a few around. If any solid block is intersecting the player’s position:
    - If the player is below or at that block’s top, push the player up (stand on the block) and zero out the downward velocity (land).
    - If trying to move into a wall horizontally, cancel or slide the movement.
  + Essentially, resolve collisions axis by axis: first handle X movement (push out of blocks on X), then Y, then Z. This is a typical approach for AABB vs grid collisions.
  + It doesn’t have to be perfect (stair and corner cases can be refined later), but ensure the player cannot fall through the ground or walk through a wall.
* **Implement Jumping:** Allow the player to press Space (for example) to jump. If the player is on ground (you’d have a flag for “isOnGround”), give the player an upward velocity (e.g., +5 m/s). Gravity will then pull them down naturally. Ensure the player can’t multi-jump in mid-air by only allowing jump when onGround is true.

Test this thoroughly in your world. Spawn the player a few blocks above ground and see if they fall and land. Walk off an edge, do they drop? Can you climb up a 1-block high step? (Minecraft’s logic allows stepping up 1 block height without jumping. You can implement that later by auto-lifting the player if the next block is one higher. Initially, you might require jumping for any height difference.)

**Collisions can be tricky**, so don’t despair if it’s imperfect at first. Even a simple approach where you prevent movement if the next block in the path is solid (stop the player against walls) and set the player’s y to the ground when falling is enough for now. As a basic test: the player should be able to walk on a flat plane and not sink into it, and if you create a pillar in front of them, they shouldn’t be able to pass through.

Having implemented this, you now essentially have an FPS camera that walks on a voxel terrain. This transforms the experience: you can **explore your block world as an avatar**.

*(Tip: For debugging collisions, you can draw a simple representation of the player’s AABB or print the block coordinates the player is touching. This helps to tune the logic.)*

**4.4 Block Interactions (Breaking and Placing Blocks)**

One of the defining features of Minecraft is the ability to mine (remove) blocks and place blocks to build. Now that the player can move around, add this interactivity:

* **Selecting a Block (Raycasting):** When the player left-clicks, you need to determine which block they are aiming at. This requires a raycast from the camera position in the direction of the camera’s look vector:
  + Implement a simple **ray marching**: start at the player’s eye position, and step along the ray direction in small increments (e.g., 0.1 or smaller) until you either hit a solid block or reach some max distance (like 5 or 6 blocks, which is the reach distance).
  + A more efficient approach is **DDA (Digital Differential Analysis)** for grid traversal, which essentially calculates which block grid boundaries the ray will cross in what order (there are known algorithms for voxel raycast). But a simple incremental approach can work if steps are small and max distance is limited.
  + Once you find a block that the ray intersects (the first non-air block along the ray), that is the targeted block for removal. Also determine the face of that block that was hit (this can be deduced from the ray intersection normal or by tracking which neighbor turned from air to solid in the ray march).
* **Removing (Breaking) Blocks:** Remove the targeted block by setting it to air in your World’s data structure. Then **update the chunk mesh** for that chunk. You’ll need to regenerate the mesh for that chunk (and perhaps neighboring chunk if the block was on a boundary, because a face in the neighbor chunk might become visible now). Ideally, have a function World::updateBlock(x,y,z, newType) that:
  + Finds which chunk the coordinates belong to,
  + Updates the block in the chunk,
  + Re-meshes that chunk (and optionally its neighbors if edges are affected).
  + For re-meshing, you can either regenerate from scratch or be smart and only update around that block. Given performance concerns are low right now, regenerating the whole chunk mesh on a block change is fine for simplicity.
* **Placing Blocks:** Similar, but instead of removing, you add a block. Typically, when you right-click, you place a block *adjacent* to the targeted block on the face you clicked. E.g., if you’re pointing at the top face of a block and right-click, you place a new block on top of it (assuming that space is empty). Use the hit face normal to determine the neighbor cell in which to add a block (if that neighbor is air, set it to e.g. a “stone” or currently selected block type).
* **Block Types and Inventory:** For now, you can default to placing a common block type (say, always dirt, or have a variable for “selectedBlockType” that you can change with number keys to simulate a hotbar). Keeping a full inventory system is beyond minimal scope, but you could allow toggling the block type to place for variety.

Test breaking and placing: you should be able to dig into the ground and also pillar up by placing blocks. See that the meshes update and the changes persist in memory. This makes your world **editable**, a huge step towards a Minecraft-like experience.

Be mindful of performance: continuously digging blocks shouldn’t freeze the game. A naive mesh regeneration of a 16³ chunk is fine (it’s not too many faces after optimization of hidden ones). But if you dig super fast or run and load many chunks at once, there could be hitches – this is something to watch for in expansion phase (e.g., you might move chunk mesh generation to a background thread later).

**4.5 Additional Core Features and Polish**

By now, you have the core gameplay loop: explore the world and modify it. There are a few more mechanics to consider as core features:

* **Multiple Block Types and Textures:** Expand your block definitions to include various types (grass, dirt, stone, wood, etc.). Create a texture atlas image file containing all the block faces (for example, find a free texture pack or extract Minecraft’s textures if licensing allows for personal use). Update your mesh generator to assign UV coordinates based on block type and face (e.g., a grass block has grass texture on top, dirt on bottom, and grass\_dirt on sides). This will make the world look more interesting than one uniform block. You might also implement a very basic lighting shade: e.g., make faces pointing upward slightly brighter than faces pointing north/south (a form of fixed light shading) to give blocks some contrast.
* **Lighting Engine (Basic):** A full dynamic lighting system (with torches, day/night) is complex. As a start, implement **ambient lighting** or a simple sunlight model:
  + Sunlight: maybe define “sky” blocks where if a column has no block above, it’s lit. You can propagate a light value downwards until a block stops it. This is approaching Minecraft’s smooth lighting, which is advanced. Instead, an easier trick: compute light level = max height - current y (so deeper is darker) or use a flood-fill from top of chunks marking light. If too complex, skip dynamic lighting for now.
  + Even without dynamic lights, you can use vertex colors to make underground areas dark (if you decide to dig caves, you’d notice no lighting difference unless you handle it).
  + Many clones simply make all faces the same brightness initially (full bright).
  + Mark this as a future improvement if not done now.
* **User Interface:** Minimal UI is needed: perhaps a crosshair in the center (draw a simple GUI texture or even using SDL to draw lines) to help aim at blocks. Also, you might want to display the selected block type (like a hotbar). For debug, showing coordinates or FPS is helpful. You can use SDL\_ttf to render text or use an imgui overlay for quick debugging info. But at minimum, a crosshair (two small perpendicular lines or a plus sign in the middle) greatly improves targeting.
* **Sound Effects:** If you have time, integrate SDL\_mixer and play sounds for actions:
  + Footstep sound when walking (check if player moved and is on ground).
  + A dig sound when breaking a block, and a placement sound for placing.
  + Possibly background music or ambient sound. This greatly improves immersion but ensure the basics are stable first.
* **Saving/Loading World:** It might be important to save the world state to disk, especially since you can edit it. A simple approach is to save each chunk to a file (or one file) listing block IDs. Because this is a clone, you could even aim to serialize to a format like Minecraft’s Anvil, but that’s overkill. Instead, implement a basic save: e.g., for each chunk, write out coordinates and block array to a binary file or JSON. Then on startup, generate chunks as before but if a chunk file exists, load it instead of generating new. This way, your changes persist. If this is too much for now, you can skip or do a trivial approach (like a single file with all modifications list).

Take a moment to play your game now: you can walk around an infinite (or large) voxel landscape, dig tunnels, build structures. **This is essentially a Minecraft creative mode clone at its core!** Many advanced features can be added, but you have reached a minimal, working game loop.

Commit this version in source control with a tag like "v1.0 core gameplay complete". This will be a baseline to refer back to.

**Phase 5: Expansion and Polish**

With the core in place, you can enhance the game in many directions. Below are suggestions for improvements, optimizations, and additional features. Tackle these in any order that interests you, but be mindful of complexity – some are easier than others:

* **Infinite World & Chunk Streaming:** If your world is currently just a preset area, implement **dynamic loading/unloading of chunks** so the world can be theoretically infinite. This involves:
  + Keeping track of the player’s chunk position. When the player moves into a new chunk, generate neighboring chunks that come into range and delete or cache chunks that are far away.
  + You might maintain a radius (e.g., load all chunks within 8 chunks distance of the player).
  + Ensure thread safety or do generation during time slices to avoid stutter. Many engines generate terrain in a background thread and then push the mesh to the main thread when ready.
  + With infinite terrain, you’ll need to be careful with coordinates (they could become very large). Using 32-bit ints for block positions is fine since even ±2 billion is huge (each unit is 1 meter, that’s 2 billion meters range).
  + A key challenge is preventing a spiral of lag when exploring quickly; hence consider generating only a few chunks per frame or using threads.
* **Performance Optimizations:** Profile the game to find bottlenecks:
  + **Rendering performance:** If you notice low FPS with many chunks, consider optimizing the mesh:
    - Implement the **greedy meshing** algorithm to reduce number of faces by combining adjacent faces in the same plane ([Let's Make a Voxel Engine - 06 - Chunk Optimizations](https://sites.google.com/site/letsmakeavoxelengine/home/chunk-optimizations#:~:text=One%20easy%20optimization%20we%20can,this%20optimization%20is%20as%20follows)). This can drastically cut vertex count (Minecraft does this).
    - Use **indexed drawing** to reuse vertices between adjacent faces (though greedy meshing might eliminate many shared faces anyway).
    - Consider **LOD or draw distance**: perhaps limit how far you render chunks (set far plane accordingly). Distant chunks could also be drawn with simplified geometry (or just not at all, covered by fog).
  + **Updating performance:** If block updates (break/place) cause noticeable lag, optimize the mesh update path. Perhaps only update a 16x16 column of blocks rather than entire chunk, or defer meshing to a separate thread so the game isn’t stuck (the chunk might disappear or show a placeholder while remeshing).
  + **Memory:** Monitor memory usage of chunk data and meshes. It can grow quickly if you generate a lot. Freeing far away chunks helps. You might implement a cache that writes them to disk and frees memory (like Minecraft does) if you roam far.
* **Graphics Enhancements:**
  + **Lighting and Shadows:** Improve the lighting model. You could implement a simplified ambient occlusion (like Minecraft’s smoother lighting where edges between blocks are slightly darkened) or even shadow mapping for a sun. For a classic feel, you might implement a day-night cycle by changing a global light color over time, and using that in the shader to tint everything darker at night. Dynamic lights (e.g., a torch the player carries) would require a more complex system (like a light-propagation in the voxel grid or using OpenGL light/shader tricks).
  + **Textures and details:** Add **transparent blocks** (glass, water). These require drawing in a second pass after opaque blocks and blending enabled. Water could be a semi-transparent blue plane in the water blocks, possibly with an animated texture. Also consider adding **sprites** for things like fire or particles (simple billboards).
  + **Skybox and Fog:** Implement a skybox (a big cube around the world with a sky texture) or simpler, a gradient sky. Fog can be done by adding fog calculation in the fragment shader (interpolating color to a fog color based on distance). This not only adds atmosphere but hides pop-in of far terrain and reduces the visible far detail.
  + **GUI:** If you want a proper UI, you could integrate a library like **Dear ImGui** (which the tiny C++ clone used for its menu ([GitHub - swr06/Minecraft: A Tiny Minecraft clone made with C++ and OpenGL.](https://github.com/swr06/Minecraft#:~:text=,Connected%20textures%2Fblocks%20support))) or create simple in-game UI for health, inventory, etc. ImGui is great for debugging and even could be used as a rudimentary menu or block selection interface.
* **Gameplay Extensions:**
  + **Mobs or NPCs:** Introduce simple AI characters (like a moving cube or a textured sprite that wanders) to simulate animals or monsters. This requires an update loop for entities and possibly pathfinding if you make them navigate the blocks.
  + **Crafting/Inventory:** For a more survival experience, implement an inventory to pick up blocks you break, and only allow placing if you have blocks. This needs UI to show inventory and controls to select items. Crafting can be as simple as converting wood to planks in code, or a full recipe system which is a project of its own.
  + **Health and Damage:** Add a health attribute to the player and implement fall damage (if velocity on impact > threshold, reduce health). If mobs exist, they could hurt the player. Add a death condition (respawn or game over).
  + **Multiplayer:** Very advanced, but ultimately a huge feature. You’d need to create a server-client model, synchronize world and players over a network. If this is a personal project, you might skip it, but it’s a learning path in itself (sockets, serialization, client-side prediction, etc.).
* **Polish and Bug Fixing:** As you add features, periodically take time to **refactor and clean up**. For example:
  + Organize code into files logically (e.g., World.cpp, Player.cpp, etc.).
  + Remove magic numbers (define constants for chunk size, gravity, etc. at the top).
  + Fix any known bugs (collision oddities, lighting quirks, etc.).
  + Write comments for complex sections (future you will appreciate it).
  + Make the engine more robust (handle edge cases like if player tries to dig below bedrock (maybe define a bottom layer), or if they reach the world edge (if not infinite)).
* **Tools for Improvement:** Now that your project is larger:
  + Use the **Visual Studio Debugger** extensively to diagnose issues (set breakpoints in the update loop, inspect variables like player position or block values when something goes wrong).
  + Use a **Profiler** to find performance issues. Visual Studio has a profiler (for CPU usage) you can run. Check which functions consume time – e.g., if rendering dominates, see if it’s the draw calls or mesh generation.
  + For GPU specifics, tools like **RenderDoc** can capture a frame and help debug rendering issues (like seeing what geometry was drawn). This can be overkill for simple projects but is very educational.
  + **Version control**: Continue making commits at logical milestones. If trying an experimental feature, consider making a Git branch so you can revert easily if it fails.
  + **Testing**: Try unusual player actions to test stability (e.g., dig the entire ground under you, or spawn to extreme coordinates). Also test on another PC if possible to ensure it runs (maybe package the SDL2 dlls, etc., for distribution).

Finally, **have fun expanding your game**! Building a Minecraft clone is an extensive project, and there's always one more feature you could add. Prioritize features that teach you something new or that you find enjoyable to implement. Remember to keep your code clean as you grow it – messy code can slow you down as features accumulate.

**Tools and Best Practices Recap**

As you develop this project, keep these general tips in mind to avoid common pitfalls:

* **Use Source Control from Day 1:** As mentioned, Git (with platforms like GitHub or GitLab) is your best friend. Commit often with descriptive messages. This not only safeguards your code (backup) but lets you trace when a bug was introduced by checking diffs ([What is the best Source Control for game development ? - DEV Community](https://dev.to/kahncode/what-is-the-best-source-control-for-game-development-5g0p#:~:text=Let%27s%20get%20one%20thing%20out,if%20you%20are%20a%20beginner)). Even for a solo project, this is invaluable.
* **Iterative Development:** Build and test in small increments (feature by feature). Don’t code massive changes for days without running the game – run it after each small feature to catch bugs early. This aligns with the iterative approach we outlined (phase by phase) and is how voxel engines are built in practice ([Let's Make a Voxel Engine](https://sites.google.com/site/letsmakeavoxelengine/home#:~:text=engine%20takes%20time%20and%20effort%2C,tasks)).
* **Leverage the Debugger:** Instead of relying only on std::cout for debugging, use the Visual Studio debugger. Set breakpoints inside the chunk generation to verify values, step through the collision logic to see why the player might be stuck, etc. It’s a faster way to find logic errors.
* **Profile Before Optimizing:** If the game feels slow, gather data on what’s slow. Avoid premature optimization without evidence ([Let's Make a Voxel Engine - 06 - Chunk Optimizations](https://sites.google.com/site/letsmakeavoxelengine/home/chunk-optimizations#:~:text=The%20optimization%20goal%20that%20I,be%20seen%20by%20the%20player)). Use profiling tools to see if, for example, chunk meshing is the bottleneck or if rendering too many triangles is the issue, then address that specific hotspot.
* **Memory Management:** Keep track of allocated memory. Use RAII (smart pointers or containers) so that you don’t leak memory when generating or destroying chunks. For instance, when you unload a chunk, delete its VBO/VAO to free GPU memory. Tools like Visual Studio’s memory diagnostics or external ones can help catch leaks if you suspect any.
* **Code Quality and Organization:** As the codebase grows:
  + Refactor duplicated code into functions (e.g., neighbor checking for faces).
  + Keep functions to a reasonable length and focused. For instance, a function that generates a chunk mesh shouldn’t also handle input – separate concerns.
  + Use descriptive names for variables and functions (generateChunkMesh() is clearer than doMesh()).
  + Document assumptions (e.g., “assuming chunk size is power of 2 for bit tricks” in comments if you do that).
* **Community and Resources:** If you get stuck on a particular implementation (say, the raycast or collision math), don’t hesitate to consult resources:
  + Online forums like Stack Exchange (gamedev section) or Reddit’s /r/gamedev or /r/opengl for specific questions.
  + Open-source projects: Reading code from similar projects (e.g., the C++ clone on GitHub ([GitHub - swr06/Minecraft: A Tiny Minecraft clone made with C++ and OpenGL.](https://github.com/swr06/Minecraft" \l ":~:text=,such%20as%20Glass%20and%20models)) ([GitHub - swr06/Minecraft: A Tiny Minecraft clone made with C++ and OpenGL.](https://github.com/swr06/Minecraft" \l ":~:text=,Fast%20world%20saving%20and%20loading)) or the Minetest engine which is C++ ([Minetest, an Open Source Minecraft Alternative](https://itsfoss.com/minetest/" \l ":~:text=Minetest%2C%20put%20simply%2C%20is%20a,or%20build%2031%2C000%20blocks%20up%E2%80%9D))) can provide insight into how they structure things. Be careful not to copy large portions blindly – use it to learn how others solve problems.
  + Official documentation: SDL Wiki, OpenGL reference pages, and library docs for GLM, etc., are very helpful for correct usage.
* **Common Pitfalls to Avoid:**
  + *Overcomplicating Early:* You might be tempted to introduce a complex engine pattern (ECS, multithreading, etc.) early. Make sure you truly need it. Often, a straightforward approach is sufficient for a basic clone.
  + *Ignoring Edge Cases:* Some issues (like what happens if the player digs the block they’re standing on) can crash or glitch the game. Try to handle such cases (e.g., if player is inside a block after an update, push them up/out).
  + *Too Large Steps:* Don’t jump to implement infinite world or multiplayer from scratch without having the basics solid. Tackle features in increasing order of complexity.
  + *Neglecting user experience:* Even if it’s a clone for learning, little things like smooth camera controls, a proper exit when pressing ESC, etc., make it more enjoyable to test. For example, implement mouse capture (hide cursor and confine it when looking around, like an FPS, and release it when menu or on ESC).
  + *Not using tools:* Many new programmers print debug info and slog through issues. While prints are useful, a debugger can show you exactly what’s happening step-by-step. Similarly, don’t guess why FPS is low – profile it. These tools exist to make development easier.

By adhering to these practices, you’ll not only build the game successfully but also grow as a developer. Building a Minecraft clone is an **extremely rewarding project** – it touches on graphics, physics, world generation, and more. Take pride in each feature you add: from the first spinning cube to a vast, textured terrain, to a fully interactive world.

Good luck, and enjoy the process of creating your C++ voxel game! Every big game starts from a simple prototype, and you’re well on your way from a blank window to a blocky universe of your own. Happy coding! **🏗️🕹️**

**Sources:**

* Minetest (open-source Minecraft-like) is written in C++, illustrating that C++ is a suitable choice for voxel games ([Minetest, an Open Source Minecraft Alternative](https://itsfoss.com/minetest/" \l ":~:text=Minetest%2C%20put%20simply%2C%20is%20a,or%20build%2031%2C000%20blocks%20up%E2%80%9D)).
* Voxel engine development can be approached iteratively, adding features like chunks, culling, and collision step by step ([Let's Make a Voxel Engine](https://sites.google.com/site/letsmakeavoxelengine/home" \l ":~:text=engine%20takes%20time%20and%20effort%2C,tasks)).
* SDL2 can create an OpenGL context, allowing use of OpenGL for rendering within an SDL window ([Tutorial1: Creating a Cross Platform OpenGL 3.2 Context in SDL (C / SDL) - OpenGL Wiki](https://www.khronos.org/opengl/wiki/Tutorial1:_Creating_a_Cross_Platform_OpenGL_3.2_Context_in_SDL_(C_/_SDL)" \l ":~:text=Creating%20a%20window%20in%20SDL,to%20it%20uses%20these%20steps)).
* A typical game loop processes input, updates the game state, and renders each frame ([Programming Patterns for Games: Game Loop - DEV Community](https://dev.to/zigzagoon1/programming-patterns-for-games-game-loop-4goc" \l ":~:text=The%20simplest%20possible%20game%20loop,looks%20something%20like%20this)).
* Optimization tip: avoid rendering faces of a block that touch another solid block (those faces are never seen) ([Let's Make a Voxel Engine - 06 - Chunk Optimizations](https://sites.google.com/site/letsmakeavoxelengine/home/chunk-optimizations" \l ":~:text=One%20easy%20optimization%20we%20can,this%20optimization%20is%20as%20follows)).
* An example C++ voxel engine implemented features such as simplex noise terrain, frustum culling, and AABB collision detection for the player ([GitHub - swr06/Minecraft: A Tiny Minecraft clone made with C++ and OpenGL.](https://github.com/swr06/Minecraft" \l ":~:text=,such%20as%20Glass%20and%20models)) ([GitHub - swr06/Minecraft: A Tiny Minecraft clone made with C++ and OpenGL.](https://github.com/swr06/Minecraft" \l ":~:text=,Fast%20world%20saving%20and%20loading)).
* Even for solo projects, using source control (like Git) is highly recommended for tracking changes and reverting mistakes ([What is the best Source Control for game development ? - DEV Community](https://dev.to/kahncode/what-is-the-best-source-control-for-game-development-5g0p" \l ":~:text=Let%27s%20get%20one%20thing%20out,if%20you%20are%20a%20beginner)).