ChatGPT

FEATURE:

Design and implement a **lightweight, high-performance media player** that can display video (e.g. Opencast recordings) in a **borderless window** with **audio playback**. The window should be **movable by dragging** (using Alt + left mouse drag) and **resizable**, despite having no standard window frame. Scrolling the mouse wheel while the window is focused will adjust the **audio volume** up or down. The application must utilize **GPU acceleration** to ensure smooth playback (targeting 60 FPS or higher), and support multiple graphics backends – specifically **DirectX**, **OpenGL**, and **Metal** – within one codebase/executable for easy portability. Version 1.0 will target **Windows 11**, but the architecture should be cross-platform to facilitate Linux and macOS support in version 2.0.

Key requirements and considerations for the feature include: - **Borderless, draggable window**: The main window should be created without default chrome (no title bar or borders) and still allow user-initiated move and resize actions. Alt + left-click drag will act as a substitute for dragging the title bar to move the window. The window edges (or an alternative mechanism) should allow resizing. - **Mouse wheel volume control**: When the user scrolls the mouse wheel over the window (or it's focused), it adjusts the playback volume (e.g. scroll up = volume up, scroll down = volume down). - **GPU-accelerated rendering**: Video frames (or other 2D/3D content) should be rendered using GPU APIs for efficiency. The solution should not be tied to a single graphics API – it should support Direct3D (for Windows), OpenGL (for broad compatibility), and Metal (for macOS) **within the same application**. This likely means using an abstraction or engine that can target multiple APIs. For example, **bgfx** is a suitable rendering library that is *"graphics API agnostic"* and supports DirectX 11/12, OpenGL (including ES), Metal, Vulkan, etc., under the hood [1]. Using such a library or a similar cross-platform 3D engine will simplify multi-backend support. - **High FPS, vsync**: The player should aim for vsynced 60fps playback. This means using rendering techniques that avoid stutter and taking advantage of hardware decoding if possible. Efficient frame buffering and timing are important for smooth video. - **Audio playback**: The audio from the video needs to be output through an appropriate audio API. Initially on Windows, this could be via a cross-platform audio library (e.g. SDL2's audio subsystem, which can also run on other OSes) or Windows-specific API. The volume control will interface with this audio output to adjust playback volume (e.g., by altering the audio stream volume or applying a gain). - **Portability and modular design**: Although only Windows is targeted in v1.0, the code should be written with portability in mind. Platform-specific functionality (window creation, event handling, etc.) should be abstracted. Where possible, use libraries that are themselves cross-platform (e.g. SDL for windowing/input/audio, and a graphics abstraction like bgfx) to minimize per-OS code. This will ease the transition to Linux and macOS in the next version.

In summary, the feature is essentially a **custom video player window** with unconventional window controls and broad hardware support. It should behave smoothly (no laggy UI even at 60fps video), and be packaged in a convenient way for end-users (possibly a single installer or self-contained executable that can fetch dependencies as needed).

EXAMPLES:

To aid development, the `examples/` folder contains sample code illustrating key aspects of the project:

- `examples/sdl_borderless_resize.cpp` – *Borderless window with drag and resize.* This example uses SDL2 to create a borderless, resizable window and demonstrates how to implement custom dragging and resizing. It sets up an **SDL_SetWindowHitTest** callback to designate certain regions of the window as draggable or resizable. When the user holds Alt and clicks/drags the window, the callback marks the interior as draggable, allowing the window to be moved by the user [2]. The edges of the window are marked as resize zones, enabling the user to resize the window by dragging just inside the window boundaries (since there is no OS-provided border). This example renders a solid color background for simplicity while you drag/resize the window. It shows how to track the Alt key state and integrate that with SDL's hit-testing API. (On platforms that support it, SDL's hit-test will let the OS perform the drag/resize for a smoother experience, rather than manually moving the window.)

- `examples/bgfx_sdl_example.cpp` – *Using bgfx for cross-API rendering.* This example creates a window (also borderless via SDL) and initializes the **bgfx** rendering library on it. It demonstrates providing bgfx with the native window handle obtained from SDL (using `SDL_GetWindowWMInfo`) and setting up bgfx's platform data for different platforms (e.g., HWND for Windows, Display and Window for X11, etc.). The code selects `bgfx::RendererType::Count` to allow bgfx to auto-choose the best backend for the running platform (Direct3D on Windows by default, Metal on macOS, etc.), but you could explicitly choose one if needed. The example then clears the screen to a color each frame and processes basic events:

- It handles window **resize events** (using `SDL_WINDOWEVENT_RESIZED`) to call `bgfx::reset(...)` with the new width and height so that bgfx knows about the new framebuffer size.
- It handles **mouse wheel events** to adjust a volume variable, printing the new volume level (this simulates how we'll adjust audio volume in the real application).
- It handles the quit event to exit the loop.

This example doesn't play actual video; instead, it shows a render loop with bgfx clearing the window each frame (which would be where video frame drawing or other 3D rendering occurs). It verifies that bgfx is set up correctly for **GPU acceleration** and that the integration with SDL for windowing works. Notably, by using bgfx, the same code can run with DirectX, OpenGL, or Metal underneath – fulfilling the multi-API requirement without us writing separate render code for each API (bgfx handles that internally) [1].

Both examples are simplified prototypes focusing on specific aspects (window management and rendering). They are meant to be building blocks: in the full application, the functionality will be combined (e.g., creating a borderless SDL window, initializing bgfx on it, using a similar hit-test or alternative approach for Alt-drag, and then decoding video frames to display via bgfx while playing audio).

Download the examples: You can download a ZIP archive of the `examples/` folder here: **【51†download】 examples.zip** (this contains the two `.cpp` files described above). *(If clicking doesn't download, you may right-click and "Save Link As...")*

DOCUMENTATION:

The following documentation and resources will be useful during development (and can be pulled in via the MCP for reference):

- **BGFX (Cross-Platform Rendering Library) – GitHub and Docs**: The bgfx GitHub README [1] [3] lists supported backends and platforms (showing it covers DirectX 11/12, OpenGL, Metal, etc., and runs on Windows, macOS, Linux, etc.), and the official docs provide guidance on integration. This will help with setting up bgfx, choosing a renderer, and handling things like view clearing, frame submission, resetting on resize, etc.
- **SDL2 Documentation**: The SDL wiki is a great reference for using SDL features:
- *Window creation and management*: Creating an SDL window with flags like `SDL_WINDOW_BORDERLESS` and `SDL_WINDOW_RESIZABLE`, and using `SDL_SetWindowHitTest` for custom drag/resize regions [2].
- *Event handling*: SDL events for keyboard (`SDL_KEYDOWN/UP` for Alt), mouse (`SDL_MOUSEBUTTONDOWN`, `SDL_MOUSEWHEEL` for scroll), window events (`SDL_WINDOWEVENT_RESIZED`).
- *Audio*: Using SDL's audio API (e.g. `SDL_OpenAudioDevice`, or the higher-level SDL_mixer if needed) to play sound, and functions like `SDL_QueueAudio` to feed decoded audio frames to the audio device.
- SDL's licensing (zlib license) is very permissive [4], meaning it can be freely used in commercial applications, so we can safely include it.
- **FFmpeg – libavcodec/libavformat API Documentation**: FFmpeg will likely be used to decode video and audio from media files or streams. The official FFmpeg docs [5] and **API examples** (such as the decoding example in FFmpeg's documentation) are crucial for implementing the demuxing and decoding. We'll need to use **libavformat** to read video files/streams and **libavcodec** to decode video frames and audio samples. Key aspects include opening the format context, finding streams, selecting a decoder, using `avcodec_send_packet` / `avcodec_receive_frame` to get decoded frames, and converting those frames (e.g. from YUV to RGB for rendering, using libswscale if needed). Also, retrieving audio PCM samples to send to SDL's audio device. FFmpeg's wiki and examples (like `ffplay.c` or tutorials) can provide guidance on A/V sync and decoding loops.
- **FFmpeg Licensing and Deployment**: Documentation on FFmpeg's LGPL licensing requirements [5] [6] will be important since we don't want to "break licenses." We should compile FFmpeg with only LGPL components (no GPL codecs) and **dynamically link** to its libraries to comply with LGPL [6]. The **FFmpeg License and Legal Considerations** page is referenced to ensure we distribute FFmpeg in compliance (for example, not enabling non-free codecs, providing attribution, etc.). This also suggests an installer strategy: we could have the installer download FFmpeg DLLs at setup (to avoid statically linking or including them in a way that complicates licensing).
- **Win32 and X11 specifics for borderless windows**: In case we need to dive deeper into platform-specific implementations for moving a borderless window:
- For Windows, the Win32 API allows simulating a title-bar drag by sending a `WM_NCLBUTTONDOWN` message with `HTCAPTION` after releasing the mouse capture. (This is a known trick to move a window programmatically when the user drags inside it.)
- For X11/Linux, resources on using Xlib (if not relying on SDL's hit test) might be needed. An older blog post is noted where moving a borderless SDL window required X11 calls because grabbing and moving relative to window coordinates caused jumpy behavior [7]. With SDL2's hit-test, we likely won't need manual Xlib calls, but it's good to know the underlying issue: the need for **absolute**

coordinates when dragging a window (SDL's relative coords can cause jumps) [7]. SDL's hit-test effectively solves this by letting the window manager handle the drag when possible.

- The SDL forums/Stack Overflow also discuss borderless window dragging. One can also implement a fallback: on Windows, if not using hit-test, handle Alt+LMB drag by calling `ReleaseCapture()` and `SendMessage(hwnd, WM_NCLBUTTONDOWN, HTCAPTION, 0)` using the window handle (obtained via SDL's `GetWindowWMInfo`) – this delegates the drag to the OS [8].
- **Documentation for multi-platform video**: For the future (Linux/macOS), having references for platform-specific video backends might help. For example, **Apple's AVFoundation** (for macOS) or **GStreamer** (Linux) if we ever consider alternatives to FFmpeg. But since FFmpeg is cross-platform, we can use it on all OSes. Still, keeping an eye on platform-native hardware decoding APIs (like VideoToolbox on Mac, DXVA2/D3D11VA on Windows, VAAPI on Linux) via FFmpeg could be beneficial for performance.

All these documents and links will help ensure the coding agent has the necessary information on hand: from **SDL2 usage**, to **bgfx integration**, to **FFmpeg decoding and licensing**.

OTHER CONSIDERATIONS:

In building this project, there are several additional considerations and potential pitfalls to keep in mind:

- **Choosing a Graphics Framework vs Native APIs**: Since the goal is to support multiple graphics APIs (DirectX, OpenGL, Metal) with minimal effort, using a **cross-platform graphics wrapper** is optimal. The examples and docs lean towards **bgfx**, which is BSD-2 licensed (very permissive) and explicitly supports all those backends in one library [1]. This saves us from writing separate rendering code for each API. An alternative could be using a game engine or higher-level framework, but that would likely be overkill and add complexity. If, for some reason, bgfx is not used, an interim approach for v1.0 could be to use **OpenGL** on Windows (via a library like SDL or GLFW to create a GL context) which would also run on Linux, and use Apple's **Metal** or MoltenVK on Mac. However, maintaining multiple code paths is exactly what bgfx is designed to avoid. Thus, the plan is to incorporate bgfx for rendering and let it choose the appropriate backend per platform. This needs an initial setup (building or fetching bgfx libraries, which we might automate via the build system or during installation).

- **Video Decoding and Performance**: Decoding video can be CPU intensive, especially for high-resolution content at 60fps. We should consider using **hardware-accelerated decoding** when available. FFmpeg can utilize hardware decoders (through DXVA2/D3D11VA on Windows, VideoToolbox on macOS, etc.) if configured, which would offload the heavy work to the GPU/ASIC and ensure the CPU isn't a bottleneck. If using these, make sure the frames can be transferred to our rendering context efficiently (some APIs can render directly to DX11 textures, etc., which might align well with bgfx's backends). In v1.0, a simpler route is to use software decoding for simplicity and only 1080p or so, then optimize later with hardware decode in v2.0. Regardless, structure the video playback loop to be robust: use separate threads for decoding vs rendering if necessary (e.g., a decoding thread that feeds frames to a queue, and the rendering/main thread picks them up to display). This prevents decode latency from stalling the rendering loop. We should also manage buffering to handle momentary slowdowns (buffer a few frames ahead).

- **Audio-Video Synchronization**: Proper A/V sync is crucial in a media player. We'll need to use timestamps from the video frames and audio packets. Typically, audio is driven by the audio device (or a master clock), and video is synced to it. We may use the audio playback timestamps as the master (since audio hardware clock is steady) and adjust video frame timing accordingly (dropping or delaying frames to maintain sync). FFmpeg's `AVFrame` provides PTS (presentation timestamps); we'll use those to schedule frames. For v1, using SDL's audio callback or a separate audio thread to drive sync (e.g., get the current audio play position in time and make sure the next video frame's PTS is on or before that) is a strategy. This is a bit complex, but there are resources (like FFmpeg's tutorial on syncing video) we can consult. In testing, ensure that video doesn't drift relative to audio.

- **Volume Control Implementation**: The scroll-wheel volume control should ideally provide feedback to the user (e.g., an on-screen volume indicator or at least console log as in the example). Volume can be implemented by adjusting the SDL audio device volume if using SDL (SDL itself doesn't have a direct volume API for the default device, but we can scale the audio samples before enqueueing, or use SDL_mixer which has volume control). A simple approach is to keep a software volume factor (0-100%) and multiply decoded audio samples by that factor before sending to output. SDL_mixer (if allowed) could simplify this, but it's an extra library. We can also tie the volume to the system mixer if desired, but it's probably better to keep it app-specific. Ensure the volume doesn't go out of range or cause distortion (clamp values, etc.). Also, note that on Windows, the scroll wheel might be detected even if the window is not focused (depending on OS settings like "scroll inactive windows"); we should probably only change volume when our window is active to avoid confusion.

- **Borderless Window UX Details**: When implementing the borderless window dragging on different OSes, there are a few things to be mindful of:

- On **Windows**, the Alt key is not reserved by the OS for window dragging (Alt+Drag in Windows usually does nothing by default). We must implement it. Using the SDL hit-test (as in the example) is the high-level solution. Alternatively, using the Win32 trick (ReleaseCapture + send WM_NCLBUTTONDOWN) is a more manual solution if needed. SDL's hit-test will internally call the appropriate OS behaviors (e.g., on Windows it might handle the message for us). We should also ensure that Alt+F4 still works to close the window (SDL will post an SDL_QUIT when Alt+F4 is pressed, which we handle).
- On **Linux**, many desktop environments **already use Alt+Left-Drag as a global window move** (even for windows with decorations). This means our application might conflict or be redundant. For example, on Ubuntu, Alt+Drag will move any window by default. If our app tries to do the same, it could either conflict or be unnecessary. We may need to either choose a different modifier on Linux (perhaps Ctrl+Drag or Super+Drag) or document that the user should disable the window manager's Alt-drag for this app. Alternatively, if using SDL's hit-test on Linux, it might override or work in conjunction with the window manager. It's something to test. For v1 (Windows only), this isn't an issue yet, but keep it in mind for v2 on Linux – we might need to make the modifier configurable or detect the OS/WM.
- On **macOS**, the concept of Alt-drag moving a window is not standard. We'll likely implement the same approach (maybe using the Option key as Alt). SDL's hit-test should work on macOS (internally it uses Cocoa calls to allow dragging in a borderless window region). We should test that when we get to macOS support. Also, macOS typically has a different fullscreen paradigm (and a green window button, etc. which we won't have). Borderless on macOS may behave like a tool window. We

should ensure it plays nicely (e.g., Spaces/Mission Control might treat it as a normal app window even without title bar).

- **Resizing**: Because there's no window border, we have to give some visual hint or at least make it easy to resize. In the example, we used a 10px drag region at edges. We might consider changing the cursor icon when near edges (SDL has `SDL_SetCursor` and we can use `SDL_SYSTEM_CURSOR_SIZEWE`/`SIZENS` etc.). SDL's hit-test does not automatically change the cursor, so for polish we should handle that (there is an SDL issue about cursor feedback in hit-test regions [9]). It's a minor UI detail but improves usability.

- **Fetching Dependencies in Installer**: The user expressed that it would be nice if the solution was **"a single file installer that fetches what it needs from GitHub during setup."** This implies:

- We should not statically bundle large dependencies that could cause licensing issues or bloat. For example, we can have the installer download the precompiled SDL2 and FFmpeg binaries. SDL2 is zlib-license, so it's fine to include, but its size isn't big anyway. FFmpeg is larger and trickier with licensing, so downloading an LGPL-compliant build separately is a good idea (ensuring we also provide source or link to FFmpeg source to comply with LGPL). We can perhaps host a known-good FFmpeg binary or direct the installer to FFmpeg's official builds.
- Similarly, for bgfx, we could either include its source and build it as part of our app (bgfx is not too large, and BSD license is fine to compile in), or fetch a compiled library for the platform. bgfx doesn't have official binary releases as far as I know, so compiling from source (maybe via CMake or genie) might be part of our build process rather than the installer. For ease, we might vendor bgfx as a submodule.
- We need to decide on an installer creation tool that can perform downloads. Alternatively, the app itself on first run could check for these external DLLs and download them. However, that complicates first run. It might be cleaner to ship what we legally can (SDL, our code, bgfx since it's BSD) inside the installer, and fetch only things like FFmpeg DLLs if we don't want to include them directly. Since FFmpeg LGPL allows inclusion as long as we allow relinking or provide object files, an easier path is dynamic linking – we can include the DLLs and just ensure we offer a way for users to swap them (that could be simply documenting how to replace the DLL with another build if desired). So, in summary: **no GPL code included**, use dynamic linking for FFmpeg, and make it easy to replace or update those binaries.

- Automatic updates/downloads should be done over a secure connection (HTTPS) and with verified sources to avoid tampering – a consideration for production deployment.

- **Testing on Windows 11**: We should test the application thoroughly on Windows 10/11 for the features:

- Does Alt-drag reliably move the window? (Ensure no other app or game overlay is intercepting Alt-click, e.g., some graphics drivers have alt-click shortcuts)
- Does the scroll wheel adjust volume smoothly and within range?
- When the window loses focus, do we want to still receive scroll events (probably not; typically you'd require focus)?
- If the window is at an edge of the screen, can the user still grab it to move/resize? (Might need some padding or an alternate mechanism if it becomes hard to target the 1px edge on a screen boundary)

- High-DPI displays: On Windows 11 with HiDPI, SDL might scale the window or not depending on settings. We might have to ensure DPI awareness is set (SDL by default is DPI aware). The hit-test regions should scale with DPI accordingly.

- Fullscreen mode: Not explicitly required, but maybe consider if borderless fullscreen (window maximized without border) is something we want to easily allow (perhaps as a double-click or keybind feature). For now, probably out of scope unless user requests.

- **Future features and extensibility**: While not needed in v1, keep the code extensible for things like play/pause controls, timeline seeking, etc. Because we have no default window chrome, if we later want to add UI controls (play/pause buttons, a seek bar, etc.), we might either draw them ourselves (e.g., using a UI library or custom rendering with bgfx) or rely on keyboard shortcuts. It's fine to start with just the basics (maybe use Space bar for pause, Esc to quit, etc., as hidden shortcuts). But structuring the code so that input events can be mapped to actions will make it easier to add such features. Also consider if multiple videos or playlists might be a requirement later (for now, one video at a time is assumed).

- **Memory management and cleanup**: Ensure that on exit, we properly shut down subsystems: stop audio playback, free decoder contexts, free bgfx (which we do via `bgfx::shutdown()`), and destroy the SDL window. Memory leaks in a video player that might run for hours could be problematic, so pay attention to freeing frames (FFmpeg `av_frame_free`), packets (`av_packet_free`), etc., and deleting any buffers. Use smart pointers or RAII where possible in C++ to manage this. Also handle error cases gracefully (e.g., if a video file can't be opened or a decoder isn't found, show an error and exit cleanly, rather than just crashing or hanging).

- **Patents and codecs**: One more licensing consideration: certain codecs (like H.264, H.265) are patent-encumbered. If this is just a personal/internal project, it's fine, but if distributing widely, one should be aware of patent licensing. Using the OS's native decoders (Media Foundation, etc.) can offload that liability to the OS. FFmpeg itself doesn't license patents for you. This is more of a legal/business consideration than coding, but worth noting. For an open-source or internal tool, it's usually not an issue.

Given all these considerations, the plan is to proceed with a **C++ implementation** using **SDL2 for windowing/input (and possibly audio)** and **bgfx for rendering**, and **FFmpeg for decoding**. The coding agent should scaffold the project accordingly: setting up these libraries (perhaps via vcpkg, Conan, or submodules), implementing the main loop, and integrating the examples provided into the final solution. Testing will be key to ensure the user experience meets the description (smooth 60fps video, easy window manipulation, responsive volume control) and that the code is structured to allow future expansion to other platforms with minimal changes.

---

[1] [3] GitHub - bkaradzic/bgfx: Cross-platform, graphics API agnostic, "Bring Your Own Engine/Framework" style rendering library.
https://github.com/bkaradzic/bgfx

[2] SDL2/SDL_SetWindowHitTest - SDL2 Wiki
https://wiki.libsdl.org/SDL2/SDL_SetWindowHitTest

4 SDL2/FAQLicensing - SDL2 Wiki
https://wiki.libsdl.org/SDL2/FAQLicensing

5 6 FFmpeg License and Legal Considerations
https://www.ffmpeg.org/legal.html

7 8 The Developer’s Cry
https://devcry.heiho.net/html/2009/20090923-dragging-borderless-window.html

9 Borderless window with hit testing does not toggle system cursors in ...
https://github.com/libsdl-org/SDL/issues/4790