📌 Project: HyperReal Game Engine

🔷 SECTION 1: General Architecture & Philosophy

Q1. What is the architecture of your engine?
The architecture follows a modular pattern centered around an `Application` class and a `LayerStack`. The `Application` runs the main loop and delegates rendering, updates, and events to layers. Each layer encapsulates a separate concern—e.g., game logic, UI, debugging. This modularity makes it easy to insert or remove functionality without touching the core.

Q2. Why did you create this engine? What gap were you solving?
I wanted to create an engine that strips away the intimidating learning curve of traditional engines like Unreal or Unity. Many beginners are discouraged by overly technical APIs or bloated UIs. HyperReal is minimal yet powerful—it makes 2D game development accessible without sacrificing structure or performance.

Q3. How does the engine's game loop work?
The loop consists of three main phases: `OnUpdate`, `OnEvent`, and `OnImGuiRender`. Every frame, the engine calculates delta time, updates active layers with game logic, dispatches events like input or window resize, and finally renders both game graphics and UI. This loop continues until the window is closed.

Q4. What are "Layers" in this engine?
Layers are self-contained modules that implement a common interface: `OnAttach`, `OnDetach`, `OnUpdate`, `OnImGuiRender`, and `OnEvent`. This pattern allows developers to compartmentalize their code. For example, the UI can exist in one layer, game logic in another, and debugging in a third.

Q5. How do you add/remove Layers during runtime?
The `LayerStack` manages this. Layers are added via `PushLayer` or `PushOverlay`. You can remove them by calling `PopLayer` or `PopOverlay`. Internally, it's just a vector of smart pointers, so ownership and lifecycle are tightly controlled.

🔷 SECTION 2: Rendering & Graphics

Q6. What rendering API does HyperReal use? Why?
I use OpenGL (via GLAD) for the rendering backend. OpenGL is cross-platform, easy to prototype with, and well-supported. I chose it over Vulkan or DirectX for simplicity and speed of iteration, especially for a 2D engine.

Q7. How does your `Renderer2D` class work?
`Renderer2D` is a static class responsible for batching quads and pushing draw calls. It holds vertex/index buffers, tracks a dynamic vertex array, and groups quads with similar texture bindings to reduce GPU overhead. The `BeginScene()` call resets the batch, `DrawQuad()` pushes vertex data, and `EndScene()` submits it.

Q8. Is batching implemented? How?
Yes. I use dynamic vertex buffers that accumulate vertex data until either a texture changes or the buffer limit is hit. At that point, the batch is flushed, and a new one begins. This significantly reduces draw calls per frame.

Q9. How do you handle rotated quads?
Rotation is handled via a model matrix composed of translation * rotation * scale. The resulting transform is applied to each vertex before it's added to the vertex buffer.

Q10. How are textures managed?
Textures are bound to numbered slots in the shader. Each quad carries a texture slot index. If the current texture isn't already bound, it gets a slot and the renderer tracks it. If all slots are full, the batch is flushed and the texture set is reset.

Q11. Any texture atlasing used?
Not yet, but the renderer supports it conceptually. A future improvement could add a TextureAtlas class that manages regions of a large image and updates texture coordinates accordingly.

🔷 SECTION 3: Events, Input, and ImGui

Q12. What event system does your engine use?

Events are based on a base `Event` class with subclasses for keyboard, mouse, window, etc. They are dispatched through a dispatcher pattern and propagated through layers from top to bottom, unless handled.

Q13. How do you manage input?

Input is polled via GLFW. I created an `Input` singleton that wraps GLFW calls and provides methods like `IsKeyPressed()` and `IsMouseButtonPressed()`. This allows decoupling input queries from the GLFW context.

Q14. How is ImGui integrated?

ImGui is integrated through the OpenGL backend and GLFW input system. I created a dedicated `ImGuiLayer`, which is pushed as an overlay in the `LayerStack`. Any layer can override `OnImGuiRender()` to inject custom ImGui windows.

Q15. Can users create ImGui-based UI? How?

Yes. In their custom layer, they override `OnImGuiRender()` and use standard ImGui calls. The `ImGuiLayer` sets up the context, handles frame start/end, and draws everything on top of the game frame.

🔷 SECTION 4: Engine Capabilities and Systems

Q16. What are the main systems implemented so far?

- **Rendering (2D only)**
- **Input Handling**
- **Event Dispatching**
- **ImGui UI Overlay**
- **Layer-based Game Structure**
- **Basic Timekeeping / Delta Time**
- **Camera System (Orthographic)**

Q17. What systems are planned but not yet implemented?

- Entity Component System (ECS)
- Physics (likely Box2D)
- 3D Rendering Support
- Scene Serialization & Editor Tools
- Audio
- Resource Manager (for shaders/textures)

🔷 SEC

Q18. Can I build a full 2D game with this engine?

Yes. It includes all necessary tools to build and run a 2D game, like layer management, rendering, input, and basic game loop support. It doesn't have scripting or prefab systems yet, so currently it's C++ only.

Q19. How does camera movement work?

The engine includes a `Camera` class with a transform matrix. You can set position/zoom, and the camera's view matrix is multiplied with each object's transform to get the final position in screen space.

Q20. What challenges did you face building the engine?
Biggest challenges were abstraction boundaries. Deciding how to decouple rendering from gameplay logic, or how to structure a game engine to be extendable without becoming bloated. Also managing GPU state manually through OpenGL taught me a lot about performance bottlenecks.

🔷 SECTION 5: Technical Edge Cases / Pitfalls

Q21. What kind of bugs did you face during development?

- Memory leaks from not detaching/deleting layers
- ImGui context mismatch during multiple layer renders
- Depth sorting issues when rendering overlapping quads
- Event propagation bugs—events being handled multiple times
- Frame tearing due to improper VSync config

Q22. What happens if too many textures are bound in a frame?
OpenGL has a texture slot limit (usually 32). If the limit is hit, the engine flushes the current batch and starts a new one. This ensures rendering correctness at the cost of slightly more draw calls.

Q23. How are frame timings calculated?
Using `std::chrono::high_resolution_clock`. The time delta between frames is used to scale movement and animation, ensuring smooth rendering across systems.

Q24. How is the engine debugged?
I added logging macros (`HR_CORE_INFO`, etc.), OpenGL error callbacks, and visual debug overlays with ImGui for draw call count, frame rate, and GPU memory usage.

Q25. What would you do differently if you started again?
I'd start with a proper ECS from day one. That would make systems like physics, animation, and audio easier to manage. Also, I'd abstract the rendering API sooner so I can plug in Vulkan/DirectX later.

🔷 SECTION 6: Future-Proofing & Goals

Q26. Will you add 3D support?
Eventually. That would involve a complete rework of the renderer, shaders, and transform systems. I plan to wrap OpenGL behind a rendering API abstraction so switching to Vulkan later is viable.

Q27. What makes your engine different from others like Raylib or LÖVE2D?
Mine offers a structured C++ OOP architecture with layering, batching, and modularity. Raylib and LÖVE2D are great but lean heavily on scripting and flat APIs. HyperReal is a better fit for C++ developers who want more control.

Q28. Why should someone use your engine over Unity?
If you're a beginner developer who wants to understand the inner workings of a game engine, Unity is too abstract. HyperReal is low-level enough to teach you how real-time rendering, events, and input systems work, but high-level enough to get games running fast.

Q29. Is the engine cross-platform?
Yes, through GLFW and OpenGL. It compiles and runs on Windows, macOS, and Linux. No platform-specific code is used beyond GLFW.

Q30. How easy is it to contribute or extend the engine?
Very. The codebase is organized, readable, and modular. Each system is separated into headers and translation units. Developers can create new layers, inject custom rendering logic, or modify core systems with minimal friction.