Technical Deep Dive: AI Features & 3D Implementation

This document provides a detailed technical explanation of how the AI-powered features (Quiz Grading, Chat) and the 3D model integration work in the Concept Master LMS.

1. AI Quiz Grading System

The AI grading system allows for automated evaluation of "Short Answer" questions, providing instant feedback and scores to students.

Workflow Overview

1. **Student Submission**: The student submits their answers via the frontend.
2. **Backend Processing**: The backend receives the answers and prepares them for the AI.
3. **AI Evaluation**: The Google Gemini API evaluates each answer against the question.
4. **Result Storage**: The backend stores the AI's feedback and score.
5. **Feedback Display**: The frontend displays the detailed results to the student.

Implementation Details

Frontend (

AttemptQuiz.jsx)

- **Submission**: When the user clicks "Submit", the
  submit function constructs a payload containing `question_id` and `answer_text` for each question.
- **Loading State**: A specific loading overlay ("AI is Verifying Your Answers") is displayed to indicate that complex processing is happening.
- **API Call**: `api.post('/quizzes/${id}/attempts', payload)` sends the data to the backend.

Backend (

quizzes.controller.js &

ai.controller.js)

- **Controller Logic**: The `submitAttempt` function in
  quizzes.controller.js detects if the quiz type is `SHORT_ANSWER`.
- **Data Preparation**: It maps the student's answers to the corresponding question text.
- **AI Interaction**: It calls
  gradeQuiz from
  ai.controller.js.
- **Prompt Engineering**: The
  gradeQuiz function constructs a structured prompt for the Gemini API.
  - **Role**: "You are an expert educator..."
  - **Task**: "Evaluate if the answer is factually correct..."

    - **Output Format**: It strictly requests a JSON array containing `status` ("correct"/"incorrect"), `feedback`, and `marks_awarded`.
- **Gemini API Call**: Uses `genAI.models.generateContent` with the `gemini-2.5-flash` model for fast and cost-effective processing.
- **Result Parsing**: The JSON response from AI is parsed, and `QuizResult` records are created in the database with the `ai_feedback` and correctness status.

2. AI Chat System (Concept Master AI)

The AI Chat acts as a personal tutor, allowing students to ask questions and receive explanations in real-time.

Workflow Overview

1. **User Input**: Student types a question in the chat interface.
2. **API Request**: Frontend sends the prompt to the backend.
3. **AI Generation**: Backend forwards the prompt to Gemini API.
4. **Response Rendering**: Frontend receives the text and renders it using Markdown.

Implementation Details

Frontend (

ConceptMasterAI.jsx)

- **State Management**: Uses `useState` to manage the array of `messages` (user and assistant).
- **API Call**:
  handleSubmit sends a POST request to `/api/ai/generate` with the user's message.
- **Markdown Rendering**: Uses `react-markdown` and `remark-gfm` to render the AI's response. This supports code blocks, tables, and rich text formatting, making the explanations easy to read.
- **UI/UX**: Features a chat interface with auto-scrolling (`messagesEndRef`), loading animations (`Loader2`), and a clean, modern design using Tailwind CSS.

Backend (

ai.controller.js)

- **Endpoint**: `generateContent` handles the request.
- **Gemini Integration**: It initializes the `GoogleGenAI` client with the API key.
- **Model**: Uses `gemini-2.5-flash` to generate a text response based on the student's prompt.
- **Response**: Returns the generated text in a JSON object `{ text: "..." }`.

3. 3D Model Implementation

The 3D model on the home page adds a premium, interactive visual element to the application.

Technology Stack

- **Three.js**: The core 3D library.
- **@react-three/fiber**: A React renderer for Three.js, allowing 3D scenes to be built as React components.
- **@react-three/drei**: A collection of useful helpers for `@react-three/fiber` (e.g., `OrbitControls`, `useGLTF`).

Implementation Details

Component (

Scene3D.jsx)

- **Canvas**: The `<Canvas>` component creates the WebGL context. It's configured with `alpha: true` for a transparent background, allowing the website's background to show through.
- **Lighting**: A combination of `ambientLight`, `directionalLight`, and `pointLight` is used to illuminate the model and create depth.
- **Model Loading**:
  - **`useGLTF` Hook**: Loads the 3D model file (e.g., `/robot.glb`) from the public directory.
  - **RobotModel Component**: Specifically handles the robot model. It uses `useAnimations` to play any embedded animations in the GLB file automatically.
  - **Fallback**: A PlaceholderModel (a torus knot) is rendered if no model path is provided or if loading fails.
- **Controls**: `<OrbitControls>` allows the user to rotate the camera around the model. It's configured with `enableZoom={false}` to prevent scrolling interference and `autoRotate={false}` (though configurable) to keep it steady by default.
- **Responsive Design**: The canvas is wrapped in a `div` with `w-full h-full`, allowing it to resize responsively based on the parent container's dimensions defined in Home.jsx.

Integration (

Home.jsx)

- The Scene3D component is placed within a `motion.div` in the hero section.
- It's passed the `autoRotate={true}` prop (though Scene3D implementation shows it might be hardcoded to false in `OrbitControls`, this is where you'd control it).
- The container size changes based on screen breakpoints (`h-[350px]` to `lg:h-[600px]`), ensuring the model looks good on all devices.
-