**RhinoWebApp Phase 2 Design Document**

**1. Introduction**

**Project Overview**:  
RhinoWebApp is a web-based application designed to visualize and interact with Rhino 3D models within a Three.js environment. The project leverages the rhino3dm.js library for handling Rhino files and aims to provide a user-friendly interface for exploring model data and interacting with geometry.

**Phase 2 Objectives**:  
Phase 2 focuses on achieving our MVP (Minimum Viable Product) by implementing core functionalities that allow users to upload Rhino models, interact with them, and display essential information such as object names, colors, layers, and user-defined data.  
  
Separate RhinoViewer.vue (Three.js scene initialization) from Rhino management.

**2. System Overview**

**Architecture**:

* **Frontend**: Built with Vue.js, the frontend handles the UI/UX and interacts with the Three.js scene.
* **Backend**: Although not heavily involved in Phase 2, Rhino.Compute may be used in future phases for server-side processing.
* **Libraries**:
  + **Vue:**
  + **Three.js**: For rendering 3D models and handling user interactions.
  + **rhino3dm.js**: For managing Rhino 3DM files in the browser.
  + **Rhino3dmLoader:** For parsing and loading Rhino Objects to Three.js geometry

**2.1. Design Strategy:**

* **Single File Handling**: The design is centered around handling one Rhino file at a time. This simplifies the architecture, as the system doesn’t need to manage multiple file states or switch contexts. The process will be linear: upload, process, render.
* **Component Responsibilities**: Each component in the application will have a clear, well-defined responsibility. RhinoManagement.vue will handle the Rhino file’s loading, data extraction, and conversion to Three.js objects, while ThreeJSScene.vue will focus solely on managing and rendering the scene.

**2.2. Control Flow:**

* **Step 1: File Upload**:
  + The user selects a Rhino file using an input element in RhinoManagement.vue.
  + The file is read, and the necessary Rhino data (including geometry and user-defined attributes) is extracted.
* **Step 2: Data Processing**:
  + Once the Rhino file is loaded, RhinoManagement.vue processes it to extract the geometry, object attributes, and any user-defined data.
  + The data is stored in memory for immediate use and later applied to the scene.
* **Step 3: Geometry Conversion**:
  + The geometry from the Rhino file is converted into Three.js objects. This step includes any necessary transformations, scaling, or adjustments to align with the Three.js scene.
* **Step 4: Scene Management**:
  + The processed Three.js objects are added directly to the scene managed by ThreeJSScene.vue.
  + ThreeJSScene.vue handles rendering, camera controls, and any interactions with the scene.
* **Step 5: User Interaction**:
  + After the geometry is rendered, users can interact with the scene. For example, they might select objects, which could trigger actions like displaying object details, changing colors, or updating the scene based on the current phase of the project.

**2.3. Architecture:**

* **Component-Based Structure**: The architecture leverages Vue’s component-based approach to isolate responsibilities and keep the codebase modular and maintainable. Each component has a single responsibility, making it easier to debug, test, and extend.
* **Data Flow**:
  + Data flows from RhinoManagement.vue to ThreeJSScene.vue. This is primarily a one-way data flow where RhinoManagement.vue processes and prepares data, which is then handed off to ThreeJSScene.vue for rendering.
  + This structure ensures that ThreeJSScene.vue is focused solely on rendering and doesn’t need to concern itself with the complexities of file handling or data extraction.
* **State Management**:
  + State is managed locally within the components to keep things simple. For example, the scene is stored in ThreeJSScene.vue, while the Rhino file data is managed in RhinoManagement.vue.
  + For now, there's no need for a global state management solution like Vuex, given the scope and structure of the MVP.
* **Extensibility**:
  + The architecture is designed to be extensible. For instance, while SQLite is deferred to Design phase 2.1, the current setup allows for the future integration of a lightweight database without significant refactoring.
  + Similarly, additional features like more complex user interactions, advanced scene controls, or even multi-file management could be added incrementally.

**2.4. Scalability Considerations:**

* **User Interaction**:
  + As the application grows, user interactions with the Three.js scene (e.g., object selection, data display) can be expanded. The current design allows for these features to be added without disrupting the core flow.
* **Data Management**:
  + Although we’re starting with in-memory management of data extracted from Rhino files, the architecture can be expanded to include persistent data storage (like SQLite) if the application needs to scale to handle more complex data scenarios or persist state between sessions.
* **Performance**:
  + Given the linear flow and direct manipulation of the scene, performance should be strong for the initial MVP. If performance issues arise, they can typically be addressed by optimizing the data extraction and rendering pipeline.

**2.5. User-Defined Data Handling:**

* **Extraction**:
  + User-defined data will be extracted from the Rhino model during the data processing step. This data could include custom attributes, annotations, or any other metadata defined within the Rhino model.
* **Application**:
  + This data will be applied to the Three.js objects. For instance, objects could be colored or labeled based on this user-defined data, or certain attributes could trigger specific behaviors in the Three.js scene.
* **Future Enhancements**:
  + As the application evolves, this user-defined data could also be used to drive more complex interactions or integrations, such as dynamically adjusting the scene based on user inputs, or linking to external data sources.

**Next Steps:**

* **Solidify the Flow**: Ensure that the flow from file upload to scene rendering is well-documented and that each step is clear.
* **Refine Component Interactions**: Make sure that the way components interact with each other is efficient and logical, minimizing the chances for errors or unexpected behaviors.
* **Prepare for Coding**: Once the design is fully fleshed out and everyone is aligned on the approach, we can move into the coding phase with confidence.

**3. Detailed Design**

**3.1 Modular Components**

**Objective**: Modularize the functionality to ensure maintainability and scalability as the project grows.

Existing Classes

* **RhinoDocumentHandler.ts**:
  + Responsible for loading Rhino 3DM files and providing access to the Rhino document object.
  + Manages the initialization of the rhinoModule and handles file parsing.
* **RhinoObjectIDRetriever.ts**:
  + Retrieves and logs the unique IDs of objects in the Rhino document.
* **RhinoColorRetriever.ts**:
  + Retrieves the display colors of objects and ensures they are passed to the corresponding Three.js geometry.
* **RhinoObjectNameRetriever.ts**:
  + Extracts and passes the object names from the Rhino document to the corresponding Three.js geometry.
* **RhinoLayerRetriever.ts**:
  + Retrieves the layer information associated with objects, which will be displayed in the UI.

**New Class to Build**

* **UserDefinedDataRetriever.ts**:
  + Retrieves predefined user-defined data (key: value pairs) and
* **RhinoObjectAttributeLoader**
  + Passes Rhino Object ID, Name, and Color to the corresponding Three.js geometry.
* **RhinoUserDefinedDataLaoader**
  + Passes Rhino Object User-Defined data to the corresponding Three.js geometry.

**New RhinoManagement.vue file**

* Move Rhino related calls to their own file.

**New RhinoManagement.vue file**

Move Rhino related calls to their own file.

**Integration**:

* During parsing, the above retrievers will be used to assign Rhino Object Name, Color, and user-defined data to Three.js geometry. This ensures that each piece of geometry in the scene contains the correct attributes for further interaction.

**3.2 Three.js Scene Design**

* Add a visual grid on xy plane. Minor grid = 1, major grid = 10

**Objective:** Separate the RhinoViewer.vue code (Three.js scene initialization) from

**3.3 UI/UX Design**

**Objective**: Simplify the user interface to focus on core interactions while making the application visually appealing and user-friendly.

* **Canvas Setup**:
  + The 3D canvas will be made full screen to maximize the viewing area for the Rhino model.
  + The existing 'File Upload Button' will remain as the primary method for users to upload Rhino files.
* **Interaction Design**:
  + **Geometry Selection**:
    - Users will be able to click on any geometry in the scene. Upon selection, the application will display a pop-up box in the upper left corner of the screen.
    - This pop-up will show:
      * **Object Name**: The name of the selected object.
      * **Rhino Layer**: The layer from which the object originated.
      * **User-Defined Data**: Key: Value pairs associated with the selected object.
    - The pop-up will have a minimalistic design, with a clear and concise presentation of the data.
* **Pop-Up Implementation**:
  + The pop-up will be a simple HTML element styled with CSS for clarity. It will appear when an object is selected and disappear when the user clicks elsewhere or selects a different object.
  + The pop-up will be positioned relative to the viewport, ensuring it is always visible regardless of the camera angle or scene content.

**4. Implementation Plan**

**Task Breakdown**:

1. **Modular Component Development**:
   * Extend the existing Rhino retrievers to handle object names, colors, and user-defined data.
   * Ensure these components work together to pass the necessary data to Three.js geometry.
2. **UI/UX Implementation**:
   * Modify the existing Vue.js component to make the canvas full screen.
   * Implement the pop-up UI for displaying object information.
   * Integrate the object selection functionality with the Three.js scene.
3. **Testing and Validation**:
   * Unit tests for each modular component to ensure correct data extraction and assignment.
   * Integration tests to validate that object selection and pop-up display work as intended.

**5. Testing Plan**

**Unit Testing**:

* Test each retriever module to ensure accurate data extraction from Rhino files.
* Validate that the data is correctly assigned to Three.js geometry.

**Integration Testing**:

* Verify that the UI elements (canvas, pop-up) function correctly across different screen sizes.
* Test object selection and ensure the correct data is displayed in the pop-up.