ChatGPT

Supporting 3D Model Loading in the Language vs. Using External Modules

Introduction

Building 3D and XR applications requires robust support for loading and managing 3D models. A key design question is **whether the programming language itself should natively support 3D model loading or if this functionality should be provided via libraries/engine integration.** We want to enable *liveness* (live coding and real-time updates) for 3D assets without compromising the core language features planned. Below we explore how to approach 3D model loading in the project, covering where, when, and how to implement it, and how to achieve editable, hot-swappable 3D models at runtime.

Language-Level Support vs. Library/Engine Integration

Instead of baking 3D model loading directly into the language syntax or core, it's often better to **decouple the language from specific 3D APIs**. Many successful systems use a *bridge* or library approach: the language remains general-purpose, and 3D functionality is accessed through a dedicated API or engine module. For example, the LRP robotics language avoided hard-coding robotics APIs; instead, it connects to different robotics libraries via separate bridge software [1]. This decoupling means the language core stays focused (keeping planned features intact), while domain-specific capabilities like 3D model loading are handled by pluggable components. By analogy, our language can remain lean and flexible, interfacing with a 3D rendering engine or library for model handling rather than implementing all model parsing and rendering logic internally.

Advantages of a library/engine approach:

- **Maintainability:** The core language implementation stays simpler, and updates to support new model formats or graphics techniques can happen in the engine/library without altering language semantics.
- **Flexibility:** Users could swap out or update the 3D engine if needed, as long as the language's interface (the "bridge") to that engine is well-defined. This mirrors how *LRP* could work with ROS or Lego Mindstorms by switching its bridge [1].
- **Focus on Language Features:** You can continue to develop planned language features (e.g. type system, concurrency, etc.) without intertwining them with heavy 3D logic. The language would call on an API to load or manipulate models, but the intricate details (parsing file formats, handling GPU resources) are done in the engine or support library.

In practice, this is how most environments handle 3D: for example, **Unity** uses C# as the language while the Unity engine handles model import/serialization; **WebXR/A-Frame** uses JavaScript/HTML with the three.js library doing the actual model loading under the hood. The programming language provides the logic and "glue," but *model loading is done by the engine's API*. Our project should follow a similar separation of concerns.

Maintaining Core Language Features

A primary concern is ensuring that adding 3D support doesn't derail the language's own roadmap. By keeping 3D loading as an extension or library, we preserve all planned language features – the language remains powerful and general, with 3D as an add-on. The language can define abstract interfaces or classes for things like 3D objects, scenes, or animations, but delegate the heavy lifting to engine code.

This approach also keeps the language **lively and dynamic** without being bogged down by large static subsystems. The language runtime could, for instance, expose a function like `loadModel("file.gltf")` that returns a live object representing the model. Internally that function calls the engine/renderer to actually parse the file and create a scene graph object. But from the language user's perspective, it feels like the language supports models as first-class entities – they can store them in variables, inspect them, modify them – even though the grunt work happens in the integrated engine.

By designing a clean API, we ensure the **liveness** principle carries through: the interpreter or runtime keeps running while models are loaded or manipulated. This is similar to dynamic environments in other domains – for example, Guile-2D (a Scheme game library) *"provides a dynamic environment in which a developer can build a game incrementally as it runs via the REPL"* [2]. We want our language runtime to allow incremental changes to 3D content on the fly. Importantly, focusing on language core features means things like the interpreter or compiler optimizations are handled independently, and the 3D support lives in a module that can evolve separately (or even be optional for non-3D projects).

Where and How to Implement 3D Model Loading

Where: The 3D model loading functionality should live in the *runtime library or engine* that accompanies the language, rather than in the language parser or core VM itself. For instance, we might have a built-in module (or standard library) called something like `XR3D` that provides classes like `Model`, `Scene`, and functions to load or manipulate them. This module would interface with a rendering engine (which could be custom-built for the language or a third-party engine bound to the language). By confining 3D support to a module, we encapsulate complexity – the engine can be written in a performant systems language (C++/Rust, etc.) and the language just calls into it.

When: Since 3D/XR is a critical feature for our project's goals, planning for it early is wise, but implementation can come *after establishing the core language* functionality. In other words, define the language's syntax, semantics, and other planned features first; concurrently, design the engine interface. Once the language can call external APIs reliably (via FFI or bridging mechanism), you can implement the 3D loader module. It's important to prototype early (to ensure the language <-> engine interface works as expected), but the full feature set (supporting multiple formats, complex scenes) can be iterative. This way, we **"keep all of the features planned for the language itself"** on track and integrate 3D support once the foundation is solid.

How: A pragmatic path is to leverage existing **standard 3D formats and libraries**. In modern XR development, **glTF** has emerged as a de facto standard for model delivery at runtime. glTF is *"an open standard for 3D scenes and models that's designed for efficient runtime use"*, often dubbed the *"JPEG of 3D"* [3]. It is lightweight, fast to load, and includes rich features like materials and animations [4]. We should strongly consider making glTF the primary supported format in our system. That means our 3D module would include a glTF loader (potentially using an existing library from the Khronos Group or a open-source parser) to import models.

By focusing on glTF we get multiple benefits: - **Efficiency:** glTF is optimized for runtime loading (binary .glb versions pack data efficiently, minimizing parsing overhead) [4] . - **Features:** glTF supports embedded materials, PBR textures, skeletal animations, morph targets, etc., which are essential for XR experiences [4] . This aligns with the need to leverage *animation primitives* that assets expose. - **Ecosystem:** Many tools export to glTF, and many engines support it. Adopting it means we can utilize existing converters and perhaps even engine routines (for example, if using a graphics engine that already knows how to handle glTF).

Of course, developers might have models in other formats (OBJ, FBX, USD, etc.). We don't have to natively handle every format. A sensible approach is to support one or two key runtime-friendly formats (glTF for general use, possibly USD/USDZ if AR workflows on certain platforms demand it) and document a conversion pipeline for others. For example, OBJ can be converted to glTF easily [4] . This keeps our implementation scope reasonable.

Within the engine layer, loading a model would parse the file into an **internal scene graph** structure (nodes, meshes, materials, animations). The language runtime then gets a handle to the resulting object, which it can manipulate via defined methods. This way, the *model exists as a live object in the language*, and the programmer can interact with its parts.

Example: If a model is loaded as `model = loadModel("tree.glb")`, the `model` might have properties or methods like `model.position` (transform in the scene), `model.findNode("Leaves")` to get a sub-part, or `model.playAnimation("Shake")` to trigger an embedded animation. These APIs would be designed in the language's standard library but implemented in the engine backend. The language itself doesn't need new syntax for this – it's regular function calls and object properties – but from the user's perspective it feels native and *live*.

Notably, **exposing animation and structure** is crucial. If a 3D asset contains animations (say a character model with walk/run animations or an object with moving parts), our system should let the developer target those. For instance, if `model.animations` is a list, the developer could choose an animation clip to play or even adjust its speed. Similarly, if the model has named sub-components (bones, meshes), they should be accessible for transformations or material changes. By providing this introspection, the language environment supports deep interaction with the asset's "inside structure".

This need for introspection is echoed in real-world XR development. In Apple's RealityKit, developers found that directly working with a complex USD 3D asset required digging through entity names and hierarchies to tweak materials or parts – a *"tedious, fiddly and repetitive"* process [5] . The lesson is that **our language's 3D API should abstract some of that tedious work**. We can offer conveniences like automatically exposing child entities by name or allowing queries for components (e.g., list all materials on a model). The goal is to make targeting those animation primitives or sub-objects straightforward in code, rather than forcing manual traversal of low-level data structures for each tweak.

Liveness and Hot-Swappable Models at Runtime

One of the project's core ideas is *liveness*: the ability to update code or assets on the fly without restarting the whole environment. We want this for 3D models as well – to be able to edit or swap models during runtime seamlessly.

Achieving **hot-swappable 3D models** involves both engine capability and smart design of our development workflow:

- **Dynamic Loading/Unloading:** The engine should support loading a model into a scene at runtime (which we plan via `loadModel`). It should also allow removing or replacing that model object. For example, if the developer wants to swap model A with model B while the app is running, they could call something like `scene.remove(modelA)` then `scene.add(modelB)` or simply `modelA.replaceWith(modelB)`. Under the hood, the engine destroys or hides the old model and instantiates the new one. This operation should be safe to do while the program runs (especially if tied to a UI action or a code hot-reload).

- **Edit and Refresh:** If the developer edits the *asset itself* (for instance, tweaking the 3D model in an external modeling tool or changing a texture file), we can make the environment pick up those changes. Some modern engines and frameworks support file-watching to auto-reload assets. For instance, *Bevy*, a Rust game engine, has an opt-in ability where *"at runtime, if you modify the file of an asset that is loaded into the game, Bevy can detect that and reload the asset automatically... You can edit your assets while the game is running and see the changes instantly in-game."* [6]. This kind of **hot-reload** is fantastic for rapid iteration. We should consider a similar mechanism: the development environment (or engine) could monitor asset files and refresh the model in-place. This means the world doesn't restart; the model just updates (e.g., your rock model's new shape or texture appears immediately where the old one was).

- **Live Tuning:** Apart from reloading whole assets, we want to tune parts of the model live. Because our language objects give access to sub-components, a developer in a REPL or live coding session could, for example, adjust a material color or an animation parameter in real time. If the language has a watcher on its data structures or is event-driven, these changes propagate to rendering immediately. In practice, this might involve calling engine functions to update GPU buffers or rebind materials, but again that's hidden behind the language API. The effect is that *small edits don't require a full reload*. For instance, toggling the visibility of a part (turning a layer of the model on/off) or moving a sub-node's position can be done by setting a property on the model object.

- **Limitations and Considerations:** It's important to note not every asset is equally amenable to hot-swapping in a live setting. The user correctly suspected that *not all might support this*. Complex assets with intricate state might need special handling. The Bevy engine notes that typical simple assets (textures, simple models) reload easily, but *"complex GLTF or scene files, or assets involving custom logic, might not"* refresh perfectly [7]. In our context, that means if a model has very complex interdependencies (scripts attached, physics, etc.), hot-swapping it could be more involved. We should design for the common case (geometry and material changes) and document that truly fundamental changes (like altering a skeleton hierarchy) might still require a restart of that model's portion of the scene. However, since our language environment is under our control, we could also manage some of this by resetting the model's state when needed (e.g., if a hot-reload fails gracefully, we can fall back to reloading that model's scene graph from scratch without killing the whole app).

To maximize liveness, we can incorporate features like: - **Asset Versioning:** Each loaded asset could have a version tag; when a file changes or a new asset is swapped in, the version updates and the engine knows to use the new data for subsequent frames. - **State Preservation:** If we reload a model, perhaps preserve any runtime state that makes sense. For example, if an object had moved or the user changed some parameters via code, one could attempt to reapply those to the new model if the structure is similar. This is a stretch goal – initially, simply reloading entirely is fine. - **Hot-Reload Toggle:**

Similar to Bevy's opt-in approach, allow developers to enable/disable automatic model reloads. During development it can be on, but in production it might be off for stability/performance.

The **animation primitives** in assets deserve special mention. Many 3D models (especially character or interactive ones) come with animation data – whether skeletal animations or morph targets. We should ensure our runtime can *play, pause, seek,* or *swap* these animations on the fly. For example, if a character has "idle" and "run" animations, the developer should be able to trigger those via the language API without reloading the model. Moreover, if the asset supports it, blending between animations or adjusting the playback speed are features that make XR experiences fluid. All of this can be handled by the underlying engine's animation system, exposed through our language's model/animation objects.

Finally, liveness also implies an interactive development experience. We could provide a simple scene inspector or debugging output for models – e.g., the ability to query the model object for its list of nodes or animations. This complements hot-swapping by giving developers insight into what can be changed at runtime. In summary, **the runtime should treat 3D models as live objects that can be inspected and modified on the go.**

Conclusion

In conclusion, **3D model loading should be supported via the language's runtime/library rather than hard-coded into the language core.** We recommend designing a clear API in the language for loading and manipulating models, while delegating the actual loading/parsing to a 3D engine or module. This approach preserves all the planned language features by avoiding entanglement with complex 3D logic, aligning with the principle of keeping the core language clean.

By leveraging standard formats like glTF (built for efficient runtime use [3]), we ensure the model pipeline is efficient and feature-rich (including animations and PBR materials [4]). The integration can happen as a bridge where the language calls engine functions to load assets and returns live objects to the user.

Crucially, the design should embrace **liveness**: allow developers to edit and swap 3D assets during runtime without resetting the entire application. Through techniques like dynamic loading/unloading, file-watchers for auto-reload, and exposing model internals for fine-grained control, we can achieve a workflow where a change to a 3D model or its parameters is reflected immediately in the XR experience. This boosts productivity and creativity, as seen in other live-programming systems [2] [6].

Finally, while not every model format or scenario will support seamless live editing (some complex cases may need careful handling [7]), focusing on a robust core approach will cover the vast majority of use cases. In summary, **implement 3D model support as a well-structured module of the language's ecosystem** – one that interfaces with powerful 3D libraries – and ensure it supports hot-swapping and editing to keep in spirit with the language's live, dynamic nature. This will provide the needed 3D/XR capabilities without sacrificing the integrity of the language's design or the developer's rapid feedback cycle.

Sources

- Fleischauer, M. *Web Game Development Tools* – mentions Three.js providing 3D scene graph and model loading as an external library in JS [8].

- VR Software Wiki – *glTF File Format* (Brown University) – describes glTF as an efficient runtime 3D format, “JPEG of 3D,” supporting materials/animations [9] [4].
- Báez, M. – *LRP: Live Robot Programming* (dissertation) – discusses decoupling language from specific APIs via bridges [1].
- Holmes, T. – *Customizing 3D assets in visionOS* – notes the difficulty of manually targeting sub-entities in 3D assets at runtime without dedicated support [5].
- Bevy Cheatbook – *Hot-Reloading Assets* – explains auto-reloading changed assets during runtime for instant feedback (with some format limitations) [6] [7].
- Thompson, D. – *Live Asset Reloading with Guile-2D* – highlights a live coding game environment allowing incremental development via REPL [2].

---

[1] repositorio.uchile.cl
https://repositorio.uchile.cl/xmlui/bitstream/handle/2250/175437/Mapping-state-machines-to-developers%27-mental-model.pdf?sequence=1&isAllowed=y

[2] Live Asset Reloading with guile-2d - dthompson
https://dthompson.us/posts/live-asset-reloading-with-guile-2d.html

[3] [4] [9] VR Software wiki - glTF File Format
https://www.vrwiki.cs.brown.edu/scientific-data/3d-model-file-types/gltf-file-format

[5] A SwiftUI-like DSL for customising 3D assets in visionOS at runtime | by Tom Holmes | Medium
https://medium.com/@tommy_holmes_/a-swiftui-like-dsl-for-customising-3d-assets-in-visionos-at-runtime-bfeecb7677eb

[6] [7] Hot-Reloading Assets - Unofficial Bevy Cheat Book
https://bevy-cheatbook.github.io/assets/hot-reload.html

[8] Web Game Development Tools by Mike Fleischauer
https://gitnation.com/contents/choosing-a-game-engine-or-framework-for-html-game-development