Cardano Game Engine Wallet - Godot Integration

Proof of Achievement & Research report - Milestone 1

Executive summary

The Cardano Game Engine Wallet is a project with the goal of developing and integrating light wallet and more general blockchain functionality into the Godot engine (what is normally called “offchain” code in the field), with the aim of significantly lowering the barrier for developers who want to power their own entertainment experiences with the Cardano Blockchain and Godot engine.

As part of the first milestone of the project, our team evaluated the different approaches to integrating blockchain functionality into the Godot engine and divided them into two axes: the *extension mechanism* and the *integrated library*. As a result of our inquiries and some experimentation, we settled on using Godot’s **GDExtension** as the extension mechanism and the **Cardano Serialization Library (CSL)** as the integrated library. Mediating both is **godot-rust**, a binding layer, since Godot and CSL use different programming languages (C++ and Rust, respectively).

According to our assessment, this technology stack should allow us to develop **both a Cardano light wallet and a more featured Software Development Kit (SDK)**, fulfilling in that way both the primary and the second, optional, goal that was laid out in the project’s proposal (*“Providing a Godot SDK for constructing Cardano dApps / games”*).

As proof that this approach is feasible, **we provide an interactive demonstration that allows a user to 1. Import an already existing wallet 2. Transfer ADA to an arbitrary address**. This proves that we have a mechanism for calling into our integrated library (CSL) from GDScript and performing both wallet and general off-chain logic.

The extension mechanism

Godot is a fully-featured, cross-platform game engine that supports many different platforms, both for development and as final running targets for the produced games. Since the goal of the project is to extend the capabilities of the software to allow for Cardano dApp development, one of our first concerns was to investigate the different extension mechanisms that the engine allows and the ways these interact with the distinct targets the engine supports.
In this regard, Godot provides two avenues for extension:

1. **Modules**
2. **GDExtension**

The first one, **modules**, is a way of customizing the engine by writing C++ code that is **statically** linked during build time. This method is in principle the most powerful, since it allows one to interact directly with the engine's internals. A testimony to this is that GDScript (the domain-specific language used for scripting game logic in the engine) is implemented as a module and not as part of the engine core, contrary to what its importance might suggest.

The second, **GDExtension**, is a mechanism for loading shared libraries during **runtime**. To interact with the engine itself, these libraries must employ the GDExtension API, a (currently) unstable C interface automatically generated by the engine. This method allows the user to download an extension and use it at any time, **without needing a custom built version of the engine** with the required Cardano functionality; a significant advantage for distribution and ease-of-use. **It also allows us to work in any programming language** that supports the C ABI.

We consider that these last two reasons weigh far more than what the modules system can provide, so **we prioritized getting a working demonstration using the GDExtension system**. We left the module system as a last resource, in case technical difficulties with integrating the desired library could not let us achieve our goals using the GDExtension system.

The integrated library

Introduction

When developing applications for the blockchain, it is natural for a split to occur in the codebase. The decentralized nature of the chain is due to the validation logic that network nodes execute before incorporating a transaction into the immutable ledger. This logic is usually called the "on-chain" code, which is quite different from the rest of the code and **is not in the scope of this project**. We expect users of our SDK to leverage any of the already existing technologies to develop this part of their dApps (such as: PlutusTx, Plutarch, Aiken, etc.)

What this project aims to provide is support for writing the "off-chain" code. As the proposal says, the goal of the Cardano Game Engine Wallet is to provide an integrated wallet and *optionally* a SDK for writing the off-chain code. The reason we emphasize the optional goal now instead of the mandatory wallet part will become clear soon.

The off-chain code is the one immediately adjacent to the blockchain. As such, it is somewhat ill-defined: it is code related to the blockchain, but that it does not run in the blockchain itself (in contrast to the smart contracts and other on-chain code). We may say it is the code that is absolutely necessary to read and write data from the blockchain.

Some of the responsibilities the off-chain code handles:

1. Querying the blockchain to get information about its entities: unspent outputs in an address, metadata of a transaction, delegated stake, etc.
2. Building a correct transaction, using as inputs the information queried previously
3. Submitting a transaction to the blockchain.

These are precisely the same things any wallet must also do to support even the most basic features (e.g: transferring assets to an address). Because of this, it follows that **the wallet is also off-chain code**. The main difference is that, in a classic dApp architecture, the user brings their own wallet. With the Cardano Game Engine Wallet, this is no longer the case: the game installation can bring its own wallet that the user may use while interacting with it.

Because of this strong link between wallets and off-chain frameworks, we focused our attention on how to deliver an off-chain framework for the Godot engine, knowing that the wallet specific functionality (such as seed-phrase generation, account management, importing, etc.) **was not going to be very difficult to implement from scratch if necessary**. We envision that game developers employing our framework will customize the game-embedded wallet to their needs, probably activating/deactivating or even extending its functions.

Difficulties in integrating an existing framework

To reduce the amount of code to be written we evaluated the possibility of integrating an already existing framework into the Godot engine. Our main focus was the **Cardano Transaction Library (CTL)** and **Lucid**, both written in Javascript or languages compiled to Javascript. Re-using a framework would have allowed us to lift many useful features completely and make them available to the final user, with us just having to deliver the wallet-specific functionality. The first framework also had recently acquired support for generating BIP-39 key-phrases, making the wallet-specific tasks even fewer.

Integrating a Javascript library by embedding a Javascript runtime into Godot is possible, and there is in fact a **GodotJs** binding layer that does precisely this by leveraging the **QuickJS** framework. However, after some inquiries, we realized that the main obstacle to this approach was going to be the common dependency in all off-chain frameworks: the **cardano-serialization-library (CSL)** .

This last library is very common in all frameworks because it handles the serialization and deserialization of the Cardano ledger's data types, a task that is very error-prone and tedious, but also crucial to realize a correct implementation of the off-chain code. It is implemented in Rust, and used on the web as a WASM module. This last detail was the main complication: if we wanted to use any of these frameworks, we needed to be able to run Javascript **and** WASM. This technical hurdle seemed very difficult to achieve, as there are currently no mature WASM extensions or modules available for the Godot engine. Because of this, the idea of integrating an already-existing framework **had to be dropped**.

Results of experimenting with CSL / CML

With the difficulties expressed above, our team decided to start writing a small demonstration of an integration of CSL in the Godot engine (PR), with the aim of just developing a light wallet integration. The goal was to discover any possible issues that could arise and also experiment with different approaches for communicating values between GDScript and Rust/CSL.

godot-rust was used as a binding layer between the Rust code and Godot and its GDExtension API. Early on, we settled on a design that involved wrapping CSL with a minimal Rust layer that exposed a few native classes to the GDScript side of the framework, which godot-rust allows to do by calling a small number of Rust macros.

The results were good: there were no issues in using these wrapper classes in GDScript and calling methods for generating wallets and building transactions. After more iterations on the demonstration and researching other frameworks (particularly Lucid), we also came to the conclusion that **we can deliver the required SDK functionality by re-implementing the necessary features on top of CSL**, just like Lucid and CTL do.

As part of our experimentation, we also tried using the newer **cardano-multiplatform-library (CML)** (commit). This library aims to achieve the same as CSL, with the difference that it employs **cddl-codegen** to keep the ledger's data types up to date automatically. Porting the demo from CSL to CML was not a difficult task, and the libraries presented minimal differences between their APIs.

Because of this, **we considered the choice of library to be contingent on 1. Which is the best maintained and 2. Which is most used**. While the emphasis on code generation promises to make maintenance better for CML, we are also wary of choosing a serialization library that is different to what most other tools use. With this in mind, **we chose to continue our work with CSL.**

The demo

We present a description of the demo, as well as instructions for setting it up and executing it, in our GitHub repository: godot-cardano.

Paima integration design

Introduction

Our team held a meeting with the Paima team at dcSpark to address different approaches for the integration.

We are aware of two main ways of achieving this integration:

- Leveraging the JavascriptBridge class available in web exports of games to communicate with the browser context.
- Use the GodotJs binding layer to execute the Paima SDK Javascript libraries directly from the engine.

Of these two, the second approach **provides a path for supporting all of the platforms** we aim for with our wallet and SDK. The first approach is strictly restricted to web exports and, therefore, **only suitable for web games**.

GodotJs was considered before for integrating CTL or Lucid and discarded due to the complications that running WASM introduced; this is something we do not need to concern ourselves with when considering a Paima integration. There are, however, other factors that introduce problems when trying to run Javascript code in Godot with GodotJs:

1. GodotJs is implemented as an engine *module*. As we discussed before, this implies **a less than ideal experience for compilation and distribution of our work**.
2. Running Javascript code inside Godot implies using a non-standard runtime. Most Javascript code is designed to be run in the browser or with the Node.js runtime. **This means that there might be compatibility issues when running the Paima libraries with GodotJs**.

Because of these, we considered following the GodotJs approach for integrating the Paima libraries **too risky**, as we might run into issues that we cannot foresee or fix in the limited time-frame of Milestone 5, where most of the work in this area is scheduled to happen.

With this in mind our design for the Paima integration is based on the 1st approach, the one employing the JavascriptBridge.

JavascriptBridge design

A design involving the JavascriptBridge consists in exposing the Paima SDK's functions and objects to the developer with GDScript wrappers. Thismay be performed easily, as the JavascriptBridge class allows us to access any object in the web page, as well as its methods and properties.

Furthermore, there is support for passing callbacks to the Javascript runtimes, meaning that it is possible to also trigger signals and do event-oriented programming, as it would be expected from a game developer.

For reference, we plan to use other projects already following this approach, in particular the Metamak addon for Godot 3.5: https://github.com/nate-trojian/MetamaskAddon .