Scenechain Whitepaper

v0.2.1 - Oct 23 2021

Website: https://scenechain.org

> Please note that until the release of the **Scenechain mainnet** the **Scenechain whitepaper** should be considered under development.

Scenechain is a collaborative generative-art NFT creation platform built on Solana. It is using:

- **composability** to achieve collaboration
- **code-space scarcity** to incentivize both high quality and the use of composition

The long-term vision of Scenechain is to make high-quality audiovisual generative art a store of value.

About generative art and the demoscene

If you are not familiar with generative art and the demoscene this is an important chapter to understand this whitepaper. Otherwise, you can skip ahead to the *Scenechain basics* chapter.

Quoting from Wikipedia:

> Generative art refers to art that in whole or in part has been created with the use of an autonomous system. ... it often refers to algorithmic art (algorithmically determined computer generated artwork) and synthetic media (general term for any algorithmically-generated media).

There are 2 quite distinct communities involved with generative art: the more traditional art scene and the demoscene.

Generative art in the art scene

The generative art movement started to take off in the 1960s in the art scene. Artists used computers to generate art pieces. Here is an example from 1966:

Picture 1 - Walk Through Raster, Frieder Nake 1966

The demoscene

Quoting from Wikipedia:

> The demoscene: is an international computer art subculture focused on producing demos: self-contained, sometimes extremely small, computer programs that produce audiovisual presentations.

The demoscene started in the early 1980s. In the demoscene, there are demo categories that artificially restrict the program size of the demos. The most widespread categories are the 64Kbyte demos and the 4KByte demos. Because of the small code size, these artworks rely very heavily on algorithmic generation. The most beautiful 64Kbyte or 4Kbyte demos are basically high-quality generative artworks.

This is one of the best 64kbyte demos of all time:

https://www.youtube.com/watch?v=MAnhcUNHRW0

And here is one of the most amazing 4kbyte ones:

https://www.youtube.com/watch?v=jB0vBmiTr6o

Scenechain basics

Scenechain demos work on a virtual demo machine which we call the **Scenechain Virtual Machine**, which is basically a webassembly virtual machine with some audio-visual APIs that are thin webassembly wrappers of browser APIs like `WebGL`, `Web-Audio` and the `Canvas API`. In practice, Scenechain demos work in a web browser.

The Scenechain demo code lives in the **Scenechain code space**. The Scenechain code space is limited: Anyone can upload code to the code space in a permissionless way, but it costs SCODE tokens to upload code. SCODE is a scarce deflationary asset. This way creators are incentivized to achieve the desired audio-visual effect using as small code-size as possible. The created demos are NFTs that can be sold for Solana.

Composability

The above-mentioned mechanism would create an interesting NFT marketplace, but the secret sauce and core innovation of Scenechain are that it also incentivizes creative collaboration through composability (otherwise known by programmers as *code reuse*). Creating useful reusable open-source code libraries always suffered from the **tragedy-of-commons** problem: people gain tremendous value from these libraries but their creators are not financially rewarded for it. In Scenechain in order to create amazing demos with small code size, creators are incentivized to refer to other already uploaded code libraries and call functions from them, because as we will see it is cheaper to refer to code than to duplicate it. The creators of the libraries are automatically rewarded when other creators reuse their code. We will present an example in the next chapter on how this composability can be used to achieve great things using other people's work. We believe that in the future it will also be used in ways that we cannot anticipate today.

Scenechain package basics

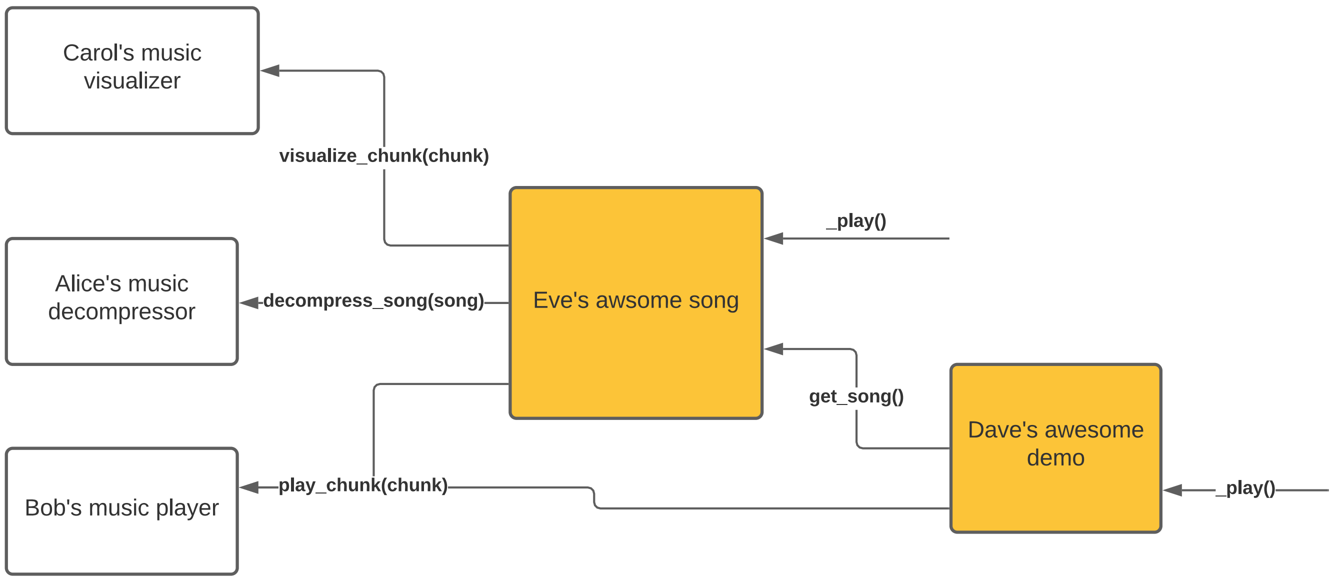
Creators can upload so-called **Scenechain packages** to the Scenechain code space. A Scenechain package is a webassembly module plus a little metadata.

Package upload costs SCODE tokens: we will discuss the details of cost calculation later. Scenechain packages are immutable: once uploaded cannot be modified and are there forever. The package is the unit of ownership in Scenechain. When a package is uploaded its uploader becomes its first owner. Packages can be traded as NFT-s.

A SCENECHAIN PACKAGE - being a webassembly module - contains webassembly **functions**. Some of these functions are declared **exported functions**. An exported function in one package can be **imported** and called from any other (later) package in the code space: this is the way creators can reuse already uploaded code.

> Note that exported and imported functions are not a SCENECHAIN invention, they are part of the webassembly format, SCENECHAIN just uses a naming convention that the package id (which is a Solana account id) is the name used as webassembly module name when importing.

A SCENECHAIN PACKAGE can be a **demo** or a **library**. A demo can still contain exported functions, its only speciality is that it contains a function called `_play()` which is the entry point of the audio-visual demo. Although the `_play()` function is exported to be accessed by the demo runtime, other packages can never import this function. (The general rule is that any function with a name starting with the underscore character is exported for the runtime and cannot be imported by other packages.)



Picture 2 - Scenechain packages referring to each other

In the above diagram we can see a use-case with 3 libraries and 2 demos (one of which is not really a demo but a song). The arrows represent imports in the direction of function call (the arrow goes from the importer to the exported function). Eve is a composer who composed a song. She offers this song technically as a demo, because her package has a `_play()` function. She is not a coder though: her package contains just a few lines of code and her compressed music. In the short code she calls Alice's library to decompress the music from the compressed representation stored in her package, calls Carol's music visualizer to display something on the screen during the music and calls Bob's music player to play the music. In addition to this she exports a `get_song()` function so that demo coders who are not composers can create demos based on her music. This is what Dave does with his demo: Uses Eve's music during his demo and also uses Bob's music player library to play the music.

SCODE TOKENOMICS

SCODE is a standard spl token on Solana. Its token address is: `HWnfNCDHWJqqXQBpF379ubyLPhk8sdvFAa2mBsn4T6Bz`

1,000,000,000 SCODE tokens has been minted in the beginning. SCODE is a deflationary token, because as we will see a lot of it will be burnt when used, and no SCODE token is released besides the genesis 1,000,000,000 ever.

The cost of a package upload is calculated as follows:

```
upload_cost = base_cost + dependency_cost
```

`base_cost` is 1 SCODE for each byte using the **net package size**.

> The net package size of a package is the size of the webassembly module inside the package minus the import and export sections plus 1 byte for each import and export. The reason that we count the imports and exports as only one byte each is that these sections (and fortunately only these sections) contain the human-readable names of imported and exported functions. (At other places only the function index is used inside binary webassembly). If we would count these sections into the cost as is, creators would use very short (minified) names, which would be confusing.

98% of `base_cost` is simply burned, 2% is `operator_reward`.

> operator_reward is payed to the operator of the webapp (or other kind of off-chain app) the user is directly interacting with to access SCENECHAIN.

DEPENDENCY ROYALTIES

Designing a good system to distribute royalties to dependencies is not trivial. The problem is that if the system is designed in a naive way there will be profitable strategies to upload packages in a way that is not good for the ecosystem. The most obvious of these behaviors is to create copycats of every interesting library. If we design algorithms to discourage this then we get other kinds of more nuanced vulnerabilities like actors managing to pay lots of royalties to themselves.

We considered multiple complex systems to solve these problems purely with algorithms. Each of these systems has some drawbacks and some of them are very complicated. Eventually, we decided to go with a relatively simple system and not only rely on algorithms but also on **social contracts** in the community.

First of all, `dependency_cost` is 0 for library packages. (packages that do not have a `_play()` function)

For demo packages, the `dependency_cost` is calculated based on the functions called from the `_play()` function of this package and the functions called by them recursively. First, a list of all these called functions is assembled, which we call dependencies. Then for each such dependency, a cost contribution is calculated as the size of the function multiplied by a `p` quotient.

```
dependency_cost = for_each_dependency_function_F(size(F) * p)
```

where

```
p = 0.03 + (position_in_codespace(F) / current_filled_codespace_size) * 0.02
```

This cost is simply transferred to the owners of the dependency functions.

In addition to the math, the social contract used here is that the community values originality and reuse of original code where possible. Any demo which has a library that has obvious copycats in their dependency tree will be worth less by NFT collectors than the ones that use the ecosystem as intended. We will discuss Scenechain social contracts in more detail in the *Scenechain manifesto* chapter.

Implications of the dependency royalties calculation

The mechanism above means that the cost of using a package is somewhere between 3% and 5% of its size. The formula incentivizes the usage of functions earlier in the code space: besides the social contracts mentioned above, this is also something that incentivizes original work instead of copycats.

This mechanism implies that it is profitable to publish a library if its functions will be called (directly or indirectly) from at least 20-33 demos on average. (Actually, sometimes a package can be profitable with significantly less use because as we will see the dependencies of a package also get some royalties from package sales.) Anyway, 20-33 usage can be easily achieved in the case of certain generic functions. Also, note that if a creator needs to write a function for a demo anyway it might be a good idea to export it even if only 1 or 2 packages would use it.

Tokenomics implications to the size of the code space

Because 0.98 SCODE from the 1,000,000,000 SCODE is always burnt for each uploaded byte the maximum capacity of the code space is around 1 GByte. The code space seems extremely scarce, and it is, but people familiar with 4Kbyte demos know what very creative people can achieve in small code sizes. Also here the possibility to cheaply reuse other people's code means that we will see traditional 4Kbyte demo quality in much smaller sizes. The decision for the 1,000,000,000 initial SCODE tokens was a tradeoff. We want to simulate Renaissance Florence: There must be enough code space to have a vibrant ecosystem for the years to come and to be profitable in creating amazing art and useful libraries in the early years. On the other hand, we need extreme scarcity for the conservative long-term art NFT collectors.

NFT trading cost and royalties

Once a package is uploaded it can be traded for SOL tokens. There is a 0.2% `trading_fee` involved which is paid to the operator (similarly to `operator_reward` in case of package upload).

We have seen that dependent functions get royalties from package uploads. There is also a mechanism that dependencies get a little bit of profit from high-value package sales. It works the following way: For each package, its highest-ever sale price is recorded. (all-time-high price). When a package is sold for a new all-time high price, a bonus needs to be paid by the buyer. This bonus is distributed among the dependencies of the package weighted using the already known weights (`size(F) * p`). The bonus is calculated as follows:

```
ath_royalty_bonus = (new_ath - previous_ath) * 0.05
```

Scenechain manifesto

The Scenechain community can thrive if most of the community members behave according to this manifesto:

```
The ultimate goal of the Scenechain community is to protect the ability of valuable
Scenechain packages to be a long-term store of value.

We believe that the above goal can be achieved through valuing innovation and
collaboration.

We value the original instead of the copycat.

With regards to copycats, not all cases are black-and-white.

Because packages are immutable each package is responsible for the entirety of its
dependency tree.
```

Let's discuss the points:

> The ultimate goal of the Scenechain community is to protect the ability of valuable Scenechain packages to be a long-term store of value.

Obviously, the value of SCODE tokens will be higher as a consequence of the success of the above goal but the value of the SCODE token cannot be more important than the ability of quality packages to be a long-term stable store of value. Without the packages having a stable long-term value SCODE tokens can only have vulnerable short-term speculative value.

> We believe that the above goal can be achieved through valuing innovation and collaboration.

For this reason, Scenechain is designed to incentivize innovation and collaboration via composition. On the other hand, this must be done without violating our first point. An example is the SCODE token supply. Higher supply might foster more innovation on a longer time horizon than the 1 billion supply but we needed a tradeoff here because we need significant scarcity to achieve our first goal.

> We value the original instead of the copycat.

Original ideas first introduced to Scenechain should be valued highly. Both when investing and also when importing. When someone blatantly copies and reuploads a library already on Scenechain (maybe changing trivialities to make it harder to notice the duplication) we call such a package (or part of a package) a copycat. Copycat authors might advertise their package both for investment and for importing. If they could succeed it would demotivate other authors to create original content. We envision that the community will create tools and databases where those packages can be flagged (with explanation).

> With regards to copycats, not all cases are black-and-white

We don't like when an exported function is just copied instead of imported from a new package. On the other hand, if the function is not exported but solves an important general problem, it should be ok to copy. (Example: Someone writes a nice short generic function to do Fast Fourier Transform on sound data inside a demo, but they do not export the function. Other people should copy and export it then.) But just copying every private function of a demo and publishing a very similar demo is not valuable.

Some copies have some improvements on the original, in this case, it is also ok to copy, but it can be a fine line. We certainly don't ban anything and we can value things just a little bit more or a little bit less, we don't need black and white judgment all the time.

> Because packages are immutable each package is responsible for the entirety of its dependency tree.

When importing a function from a package we should examine that package the same way as when we buy a package. Importing a blatant copycat makes the importer package also less valuable because 'unpunished' this behavior would lead to profitable unproductive behaviour by copycat creators.

As we mentioned there will be databases where creators can check whether a library has been flagged as a copycat suspect or not. Still, it will probably happen that someone will import from a copycat accidentally. This should not render the importer package worthless, just a little bit less worthy. Human judgment is important in these cases, these are things that would be hard to solve only with algorithms.

Scenechain off-chain code overview

The Scenechain on-chain code is responsible for the management of package uploads, cost calculation, royalty distribution, and NFT trading. It never actually executes the webassembly code inside the packages. The webassembly code is executed off-chain in the browser (or any environment that adheres to the *Scenechain Virtual Machine Specification*). Similarly to the on-chain code the Scenechain off-chain codebase is open-source. It contains a web application in which users can upload, browse and trade packages, and run demos. The web app will be run by the Scenechain team, but anyone else can run the web app parallelly. Also, anyone can create similar web apps: it is not part of the Scenechain protocol. Strictly speaking, the Scenechain protocol consists of the on-chain code and the *Scenechain Virtual Machine Specification*. (For example, this specification will contain the audio-visual APIs available for the demos.) The Scenechain off-chain code can be treated as a **reference implementation** of the off-chain parts of the protocol.

The Scenechain web app is a blockchain explorer and a lightweight development environment. It provides the following capabilities to the users:

- Users can browse packages in the order of upload. It is possible to filter for certain attributes like code size.
- It is possible to run the demos.
- The webassembly code in packages can be viewed converted to textual webassembly.
- Possibly we will store other optional information off-chain for packages and they can be also viewed. (Example: source code if the code was compiled from a higher-level language, Readme file, etc...)
- Creators can author and upload new packages. Also, they can test the packages before uploading.

Smart contract tech details

Writing smart contracts for tasks that handle a relatively large amount of data or do a non-trivial amount of calculation is not a simple feat on any blockchain. Solana has remarkable throughput but the fixed calculation limit in an instruction and the fixed account data size limit need careful consideration about how we arrange data and calculations to always fit into the limits.

From the data-access perspective, the Solana blockchain can be treated as a key-value store: Programs can own any number of accounts: the account's public key is the key, and the data stored in an account is the value.

Scenechain will create one account for each uploaded package. Each package will contain its own position in bytes in the Scenechain code space.

There will be an additional global Solana account to store global information such as:

- position in bytes in the code space of the last-added package
- the public key of the last-added package

On Solana SOL needs to be paid for an account creation if we want the account to be rent-exempt forever. Luckily this cost is not too big for the data size we deal with in the case of Scenechain. The package uploader will have to pay this SOL fee in addition to the SCODE cost of an upload.

Dependency royalty calculation in the smart contracts

The most resource-intensive calculation that needs to be executed in smart contracts is the calculation of royalties. This wouldn't even be feasible on most other chains but needs an optimized algorithm even on Solana to scale to complex Scenechain packages with large dependency trees.

The naive approach would be recursively traversing the whole dependency tree of a new package (using depth-first search for example) in one transaction. This would be prohibitive: we would need to process a huge amount of packages in one transaction.

What we do instead is that we precalculate the set of transitive dependencies of each function when its package is uploaded. A package upload will consist of several transactions:

```
actual_package_upload()
precalc_deps(F1)
precalc_deps(F2)
...
commit_package()
```

So the key smart contract function is `precalc_deps(F)`. It will simply take the precalculated dependency set of the functions called from `F`, unions them together, and stores the result as the precalculated dependency set of `F`.

The above-mentioned algorithm would be fast but would need significant storage for the precalculated dependency sets, so there is one more optimization: we only store the hash of the precalculated dependency set for each function in the blockchain. The actual dependency sets of the functions called by `F` have to be provided by the off-chain code to the `precalc_deps(F)` function. (The smart contract checks whether they are correct by comparing their hash to the stored hash.)

Scenechain Virtual Machine and web app tech details

Scenechain demo developers will need to create packages that contain webassembly functions which will run in the browser in a sandboxed environment. In these functions, they will be able to use the **Scenechain Demo API**. The Scenechain Demo API contains the low-level functions to create visuals and sound effects programmatically, and also will contain functions to gain user input to be able to create interactive experiences. The Scenechain Demo API will be mostly a thin wrapper around `WebGL`, `Web-Audio`, and the `Canvas API` which we will expose from the embedding Javascript environment. The webassembly code will be heavily sandboxed: not being able to access anything which is not part of the Scenechain Demo API.

The demos can decide the viewport size they use and they can communicate this to the embedding environment, so the embedding environment may decide what resolution it uses during playback. Demos are encouraged to use the **1080p** resolution today, but this is not a requirement in the protocol, as the community may decide to prefer higher resolutions in the case of later demos.

The smart contracts do not and cannot decide whether a webassembly demo is buggy or not. People will simply not collect demos that are deemed not worthy. The client code only needs to ensure that sandboxing works well and the web app does not hang when the webassembly code is not responsive.

The Scenechain web app will be developed using Nodejs and Javascript on the server-side. In the browser, we will use Reactjs with Javascript.

Please note that the webassembly modules in the packages cannot be used as they are in the browser. They lack the Scenechain Demo API imports and also only one module can be executed at one time in the browser. For this reason, when running a demo package, the web app will recursively download all its dependencies, and merge them into an **executable** webassembly module. In the case of the packages in `Picture 2`, if we want to run the demo `Dave's awesome demo`, all the packages on the picture will be downloaded and merged. During the merge, the following happens:

- All the functions in all the packages are appended together
- Intermediary exports and imports are eliminated, the only exports remaining are the exports of the demo package starting with an underscore.
- Function indices are reshuffled (for example in call instructions)
- Imports are added from the embedding environment to support the SceneChain Demo API.

Roadmap

Happended so far:

- September 2021: The start of Scenechain development
- Oktober 25, 2021: The 1,000,000,000 SCODE tokens have been minted

Future plans:

- November 2021: airdrops and token sale(s)
- March 2022: Mainnet release

Development Roadmap

Before the mainnet release, the main focus is on creating robust fundamentals on which the team and the community can build. The UX will be a bit bare-bones and the only programming language supported will be textual webassembly. Some of the efforts will be partly parallelized, but the rough order is the following:

- Webassembly format operations (like merging packages)
- Smart contract development
- Scenechain Demo API development
- Webapp development

After the mainnet release, we can concentrate more on higher-level convenience features which are implemented in off-chain code. Examples:

- good development environment UX
- possibly using higher level languages.
- library discoverability features

Token allocation

- 25% - Team allocation
- 25% - Airdrops
- 25% - Seed funds and/or public sale
- 25% - Ecosystem funds

Team allocation

Part of it (16%) gradually unlocks over 3 years for existing team members. Part of it (9%) is reserved for future team members. For existing team members the 3 year lock period will start before the public token sales.

Seed funds and/or Public sale

Funds collected from this will be used for **Scenechain** development.

Airdrops

Some airdrop campaigns will happen before the public sale and some will happen after it. In the airdrop campaigns before the public sale, people will be able to apply for tokens, but those tokens will be actually airdropped only after the public sale. This way airdropped funds will not participate in the initial price discovery of SCODE.

Ecosystem funds

Grants will be decided by the **Scenechain** team in the beginning. Eventually, grants will be decided by a decentralized governance mechanism.