Lesson 8 - Cairo / Starknet continued

Contract Classes

A recent addition to starknet, since version 0.9
From medium article
"Taking inspiration from object-oriented programming, we distinguish between the contract code and its implementation. We do so by separating contracts into classes and instances."

The way it works is similar to the proxy pattern in Ethereum.

A **contract class** is the definition of the contract: Its Cairo bytecode, hint information, entry point names, and everything necessary to unambiguously define its semantics. Each class is identified by its class hash.

A **contract instance**, is a deployed contract corresponding to some class. Note that only contract instances behave as contracts, i.e., have their own storage and are callable by transactions/other contracts.
A contract class does not necessarily have a deployed instance in StarkNet.

The declare transaction type declares a class but does not deploy an instance of that class.

The deploy system call takes 3 arguments

- The class hash
- Salt
- Constructor arguments

This will deploy a new instance of the contract whose address depends on the above arguments, this is similar to the CREATE2 op code on Ethereum.

Contract Extensibility

See Forum post

Currently

- Cairo has no explicit smart contract extension mechanisms such as inheritance or composability
- There's no function overloading making function selector collisions very likely – more so considering selectors do not take function arguments into account
- Any `@external` function defined in an imported module will be automatically re-exposed by the importer (i.e. the smart contract)
- Builtins cannot be imported more than once in the entire imports hierarchy, resulting in errors on import (or errors on compilation if not added) – and most contracts will need the same common set of builtins such as `pedersen`, `range_check`, etc.

There is ongoing discussion how to resolve these issues

Games on starknet

"I define the on-chain gaming thesis as this: **on-chain games will be the frontier for technological advancement of blockchain for the next five years.** Web3 game developers like MatchboxDAO are building the toolkit for future consumer-facing Dapps."

See Matchbox DAO blog
Matchbox DAO repo

Solve2Mint

Solve 2 Mint is a framework for NFT emission where each NFT maps to a unique solution to an equation or puzzle.

Topology

Topology are also very involved in this area.
Their philosophy is that usually the gane developers are 'gods' but with blockchain and zkps, we can prevent this and allow a common exploration of problems.
They have maxims such as "Compute more, store less" which makes economic sense on a layer 2.
They also believe that humans are at a disadvantage to bots, since humans need to go through devices to interact with a game.
They think that games should be designed around NP-hard problems so that an algorithm cannot be written to solve it.
They also advise to add randomness from the user, because if the game is deterministic, then someone could simulate theh future off chain.

Game interaction has been helped by account abstraction and the use of session keys for example with the Argentx wallet.

Fountain

A 2-dimensional physics engine written in Cairo
repo
See tweet

Roll Your Own

Roll Your Own - A Dope Wars open universe project.

A modular game engine architecture for the StarkNet L2 roll-up, where computation is cheap and new game styles are being explored. Roll your own module and join the ecosystem.

Influence

Influence is an immersive, persistent space strategy game where players compete through multiple avenues, including: mining, building, trading, researching, and fighting, and interact in a 3D universe through third-person control of their ships and the installations they build.

Advantages using starknet

1. High Scalability & Low Fees
   Moving to Starknet, they expect a reduction of 100X in fees
2. Instant Actions
   Actions are regarded as final once they are submitted to the sequencer.
3. More complex transactions at the same price

Loot Realms

Advantages

1. Cheap Fees
2. Heavy computation possible
3. Composability of game items and liquidity

Dope War

Design

Physics Engine on Starknet

Video

Also Physics puzzle

Dark Forest (Not on Starknet)

Dark Forest is an massively multiplayer online real-time strategy (MMORTS) space conquest game build on top of the Ethereum blockchain.

From a medium article describing the game

Dark Forest game V0.3 uses zero-knowledge proof technology to prove 2 operations regarding the planet’s location:
1/ planet initialization (init)
2/ planet movement (move).
The Circuit logic is implemented in darkforest-v0.3/circuits/.
They are implemented with circom, and the circuit proof uses Groth16.

Circuits used in Dark Forest

init Circuit

init Circuit ensures the coordinate falls in certain range during the creation of planet. Both x and y coordinates cannot exceed 2^{32}.

mimc(x,y) hash calculates on the coordinates.
x/y is private input, while hash value is public input.

move Circuit

During the planet’s movement, the move circuit checks the moving range does not exceed a circular area of radius distMax:

We need to consolidate the hash value of both original coordinates and post-moving coordinates.

Dark Forest uses Circom to create its proofs.
Circom is a novel domain-specific language for defining arithmetic circuits that can be used to generate zero-knowledge proofs. `Circom compiler` is a circom language compiler written in Rust that can be used to generate a R1CS file with a set of associated constraints and a program (written either in C++ or WebAssembly) to efficiently compute a valid assignment to all wires of the circuit.

See circuits

- `/circuits/init`: Proof for initializing a player into the universe
- `/circuits/move`: Proof for initiating a move between two planets
- `/circuits/reveal`: Proof for broadcasting/revealing the coordinates of a planet. Note that nothing in the broadcast action needs to happen in "zero-knowledge"; we just

found it easier to implement verification of MiMC hash preimage via a ZK verifier than via a Solidity verifier.
- `/circuits/biomebase`: Proof that a planet has a given `biomebase`, which in combination with the planet's `spacetype` will specify the planet's biome.

There are two additional subdirectories for auxiliary utility circuits:

- `/circuits/perlin`: Perlin Noise ZK Circuit.
- `/circuits/range_proof`: Proof that an input, or list of inputs, has an absolute value that is at most a user-provided upper bound.

There are some play to earn features, you can earn DAI for revealing your planets locations.

Deploying contracts

If you are using the starknet CLI you can use

`starknet deploy --contract compiled_contract.json --address SELECTED_ADDRESS`

If you are using protostar, see docs

`protostar deploy ./build/main.json --network alpha-goerli`

Tips and best practices

There are some useful tips here

1. If you contract functions are not marked external or view, then they are internal.
2. Functions marked as view can be invoked at present
3. Reverted transactions are a problem (see yesterday's notes)
4. You can deploy contracts from your contract, see deploy

Optimisation

This is still a new area, obviously the usual good practices of measuring, then optimising the low hanging fruit, one at a time.
A general guide is as mentioned above, computation is cheap, storage is expensive.
so

- **Cheap things**:
    - Transaction calldata.
    - Computation (adding, multiplying, calling functions, etc) — this all happens off-chain in a batch and is relatively cheaply verified on-chain.
    - Reading from a storage var — this happens off-chain.
- **Expensive things**:
    - Modifying a storage var: these modifications have to be written to L1 calldata, which is expensive.

@RoboTeddy gives some estimation of costs

A basic write that modifies a single storage_var slot

Say you define a storage var like this:

```
@storage_var
func _balances(addr: felt) -> (res: felt):
end
```

Let's examine much it might cost to execute `_balances.write(1, 123456)`.

- **Base cost:** Writing a single storage slot costs ~$0.60 (as of Nov 2021). The math:
    - The write causes a state diff of 64 bytes (32 bytes for the slot index number, and 32 bytes for the slot storage value)
    - L1 calldata costs 16 gas/byte
    - Gas price is ~130 gwei (Nov 2021)
    - Eth price is ~$4200 (Nov 2021)
    - `64 bytes * 16 gas/byte * 130e-9 eth/gas * $4200/eth = ~$0.60`
    - This is still 20x cheaper than an Ethereum SSTORE
- **'Batching rebate':** If a particular storage slot is written to `n` times within a single StarkNet batch (~1-4 hours), each write costs only `1/n`th as much. This is true even if the writes to the storage slot were caused by different tx, calls from other contracts, etc.
- **'Compression rebate':** If the value you're storing is a common one, it will likely compress well and take a bit less calldata. It might be complicated for StarkNet to pass these savings on to you, so I wouldn't rely on it happening soon.