Illuvium Land Sale Protocol

On-chain Architecture Version 1.1.2_03/31/22

Table of Contents

Introduction

Illuvium Land Sale Protocol, also known as Land Sale Protocol, or Land Sale, powers the series of sales (a.k.a sale events), selling in total up to around 100,000 Land NFTs. The protocol operates either completely on Layer 1, or in a mixed Layer 1/2 mode (when sale happens in Layer 1 and NFT minting happens in Layer 2), and consists of a Land ERC721 token itself, sale supporting smart contracts (helpers), backend and frontend services powering the initial sale of the token and simplifying the interaction with it later on.

This document concentrates on the blockchain part of the protocol, also known as on-chain protocol architecture.

General Considerations

The protocol is modeled via *tokens* – tradable entities, ERC721 compatible and ERC20 compatible smart contracts, and *helpers* – helper smart contracts which provide interactions with and between tokens in a decentralized manner.

Tokens

Tokens are tradable, transferable entities, representing a value. Basic design approach – minimalism and immutability, tokens embed as less information as possible and, ideally, have immutable data structures.

Non-fungible tokens hold the essential data model of the protocol.

Current design approach is to store both owner information and essential token data related to the protocol mechanics in the token data structure. Some data may still be stored off-chain and in the registry smart contracts (registry helpers).

In the Land Sale Protocol v1 tokens are: Land ERC721, Escrowed Illuvium 2 [sILV2] ERC20, and Illuvium [ILV] ERC20. Since the old original sILV token is not used in the protocol, for simplicity sILV2 is denoted as sILV.

Land ERC721 token smart contract is upgradable, and follows the “EIP-1822: Universal Upgradeable Proxy Standard (UUPS)” [2]. Illuvium [ILV] ERC20 and Escrowed Illuvium 2 [sILV2] ERC20 were released into mainnet in spring 2021 and winter 2022, land sale interacts with these tokens to support sILV payments.

Helpers

Helpers power token sales, pools, exchanges, marketplaces, auctions, upgrades and other token interaction logic – i.e. token data modification, ideally, without token data structure modifications. Helpers also provide storage for token data structures.

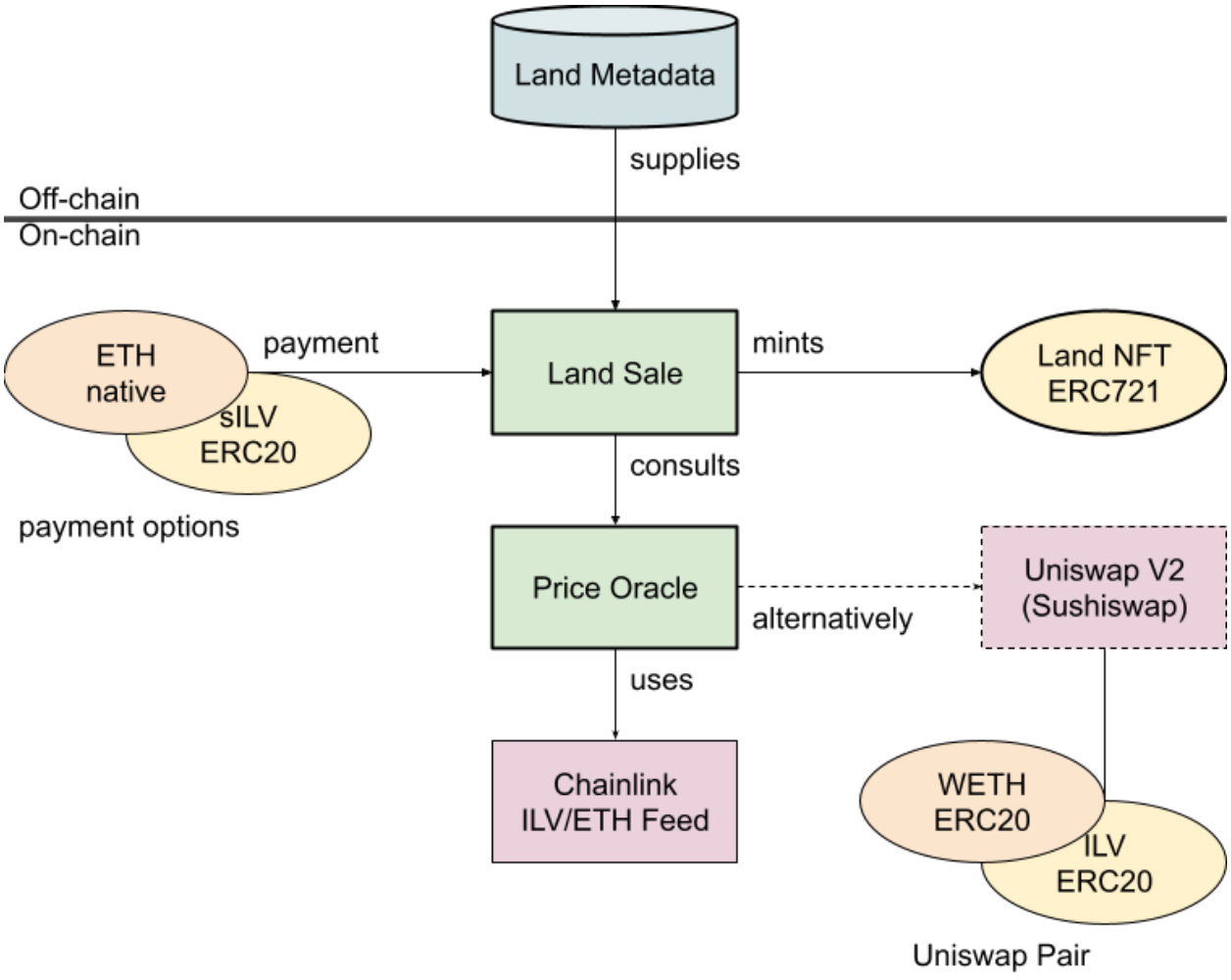
Helper smart contracts are the backbone of the off-chain interaction, serving as bridges between massive off-chain and limited on-chain protocol state changes.

Helpers may not be necessarily immutable, they may be upgraded – i.e. some of them may get deprecated and disabled, other ones may appear and introduce new features or modify existing features.

In the Land Sale Protocol v1 helpers are: Land Sale, and [Land Sale] Price Oracle smart contracts. Price Oracle is an external component of the protocol, its implementation can be ad-hocked, replaced on the go. It can be built on top of Uniswap/Sushiswap protocols, Chainlink, and others.

Land Sale smart contract is upgradable, and follows the “EIP-1822: Universal Upgradeable Proxy Standard (UUPS)” [2].

The diagram below provides a high-level overview of the protocol.



Land Sale Protocol Overview

Diagram 1. High-level view of the protocol. Land Sale smart contract sells the Land ERC721 tokens. It accepts ETH as a primary payment option, and sILV as an alternative payment option. If sILV is used, it consults with the [Land Sale] Price Oracle to convert ETH/sILV. Price Oracle is connected to the Chainlink ILV/ETH price feed (or, alternatively, consists of several smart contracts, hosted outside the land sale protocol, eventually consulting the price via ETH/ILV liquidity pair). Land Sale smart contract expects buyers to supply land metadata stored off-chain. Metadata contains land plot coordinates, rarity information, etc. Land Sale, Price Oracle, Land ERC721 contracts, as well as Land Metadata off-chain storage are part of the development scope for v1 release (marked with bold 2px border). Other components, such as sILV, ILV tokens, Chainlink Feed, Uniswap V2 contracts, WETH token, ETH/ILV liquidity pair, etc. already exist in mainnet and are used by the protocol as is (maked with regular 1 px border).

Access Control

Access Control is a base parent contract for the most smart contracts of the protocol. It provides an API to check if a specific operation is permitted globally and/or if a particular user has a permission to execute it.

It deals with two main entities: features and roles. Features are designed to be used to enable/disable public functions of the smart contract (used by a wide audience). User roles are designed to control the access to restricted functions of the smart contract (used by a limited set of maintainers).

All public functions are controlled with the feature flags and can be enabled/disabled during smart contact deployment, setup process, and, optionally, during contract operation.

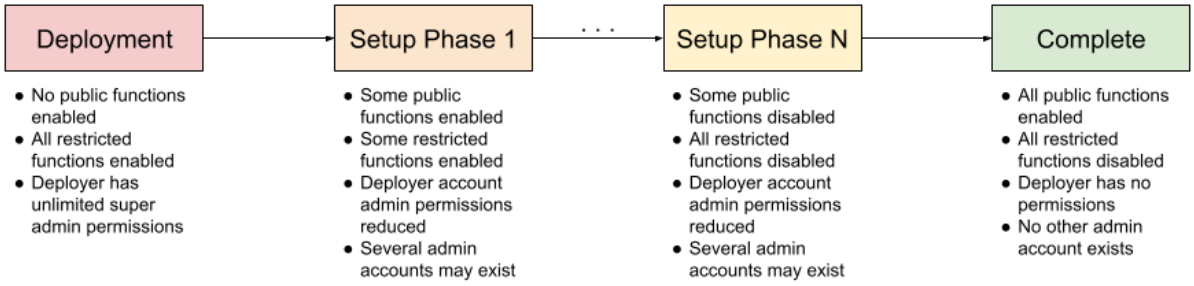
Restricted access functions are controlled by user roles/permissions and can be executed by the deployer during smart contract deployment and setup process.

After deployment is complete and smart contract setup is verified the deployer enables the feature flags and revokes own permissions to control these flags, as well as permissions to execute restricted access functions.

It is possible that smart contract functionalities are enabled in phases, but eventually smart contract is set to be uncontrolled by anyone and be fully decentralized.

It is also possible that the deployer shares its admin permissions with other addresses during the deployment and setup process, but eventually all these permissions are revoked from all the addresses involved.

Following diagram summarizes stated below:



Access Control and Deployment Stages/Phases

Diagram 2. Smart contract deployment and setup phases. It evolves from the fully controlled in the initial phases of the setup process to the fully decentralized and uncontrolled in the end.

Special Permissions Mapping

Special permissions mapping, `userRoles`, stores special permissions of smart contract administrators and helpers. The mapping is a part of Access Control and is inherited by the smart contracts using it.

The value stored in the mapping is a 256 bits unsigned integer, each bit of that integer represents a particular permission. We call a set of permissions a role. Usually, roles are defined as 32 bits unsigned integer constants, but extension to 255 bits is possible.

Permission with the bit 255 set is a special one. It corresponds to the access manager role `ROLE_ACCESS_MANAGER` defined on the Access Control smart contract and allows accounts having that bit set to grant/revoke their permissions to other addresses and to enable/disable corresponding features of the smart contract (to update zero address role – see below).

“This” address mapping is a special one. It represents the deployed smart contract itself and defines features enabled on it. Features control what public[1] functions are enabled and how they behave. Usually, features are defined as 32 bits unsigned integer constants, but extension to 255 bits is possible.

Access Control is a shared parent for other smart contracts which are free to use any strategy to introduce their features and roles. Usually, smart contracts use different values for all the features and roles (see the table in the next section).

Access manager may revoke its own permissions, including the bit 255 one. Eventually that allows an access manager to let the smart contract “float freely” and be controlled only by the community (via DAO) or by noone at all.

[1] Not to be confused with Solidity `public` function modifier. Publicly accessed functions are functions designed to be used by any user, with or without any special permission(s).

Comparing with OpenZeppelin

Both our and OpenZeppelin AccessControl [3] implementations feature a similar API to check/know "who is allowed to do this thing".

Zeppelin implementation is more flexible:

- it allows setting unlimited number of roles, while current is limited to 256 different roles
- it allows setting an admin for each role, while current allows having only one global admin

Our implementation is more lightweight:

- it uses only 1 bit per role, while Zeppelin uses 256 bits
- it allows setting up to 256 roles at once, in a single transaction, while Zeppelin allows setting only one role in a single transaction

Upgradeability

Upgradeable Access Control is an Access Control extension supporting the OpenZeppelin UUPS Proxy upgrades. Smart contracts inheriting from the `UpgradeableAccessControl` can be deployed behind the ERC1967 proxy and will get the upgradeability mechanism setup.

Upgradeable Access Control introduces another “special” permission bit 254 which is reserved for an upgrade manager role `ROLE_UPGRADE_MANAGER` which is allowed to and is responsible for implementation upgrades of the ERC1967 Proxy.

Land ERC721

Land ERC721, also known as Land NFT, Land Token, or Land is an ERC721 token smart contract representing the land in the Illuvium world. We refer to its token identifier, a record associating a token ID with its owner address, as land plot, land token, or token.

The ERC721 standard implementation is very basic – based on OpenZeppelin ERC721/EIP-1822 (UUPS) implementation. Custom upgradeable AccessControl (see Access Control section) is used instead of OpenZeppelin impl.

Land NFT features rich on-chain immutable metadata, defining unique properties of each land plot (token).

Functional Requirements

1. Fully ERC721 compliant token according to “EIP-721: ERC-721 Non-Fungible Token Standard”, with both optional ERC-721 extensions included (metadata and enumerable)
2. Implements “EIP-2981: NFT Royalty Standard”
    a. Authorized address(es) should be able to set up royalty related information
    b. Authorized address(es) should be able to set up contact level metadata (OpenSea collections support)

3. Implements “EIP-1822: Universal Upgradeable Proxy Standard (UUPS)”
    a. Authorized address(es) should be able to do the smart contract upgrade
4. Implements the “rescue” of assets functionality:
    a. Authorized address(es) should be able to transfer ERC20 tokens accidentally sent to the token smart contract outside to any address
5. Each token should have following Land Plot Metadata associated with it, stored on-chain:
    a. Region ID, defines a Region
    b. Coordinates (x, y) within the Region
    c. Tier ID, defines land Tier
    d. Plot Size, defines an internal coordinate system (isomorphic grid) within a land plot
        i. If size of a plot is H, implying $0 \leq x \leq H;\ 0 \leq y \leq H$, areas at the corners of the plot are “invalid” (see Diagram 3):
            1. $x + y < H/2$
            2. $x + y > H/2 + H$
            3. $x - y > H/2$
            4. $y - x > H/2$
        ii. Valid area of the plot is the intersection of (see Diagram 3):
            1. $x + y \geq H/2$
            2. $x + y \leq H/2 + H$
            3. $x - y \leq H/2$
            4. $y - x \leq H/2$
    e. Internal Land Structure
        i. Collection of Resource Sites, each site having its
            1. Type ID, defines site Type
            2. Coordinates (x, y) within the land plot
        ii. Landmark Type ID, defines the landmark type if present
6. Resource Site represents an element or a fuel resource, and is positioned inside the land plot on isomorphic grid
    a. There are three element type sites and three fuel type sites:
        i. Carbon (element),
        ii. Silicon (element),
        iii. Hydrogen (element),
        iv. Crypton (fuel),
        v. Hyperion (fuel),
        vi. Solon (fuel)
    b. Site size is implied to be 2x2 tiles
7. Landmark is a special site, present in the land plots of higher tiers (3, 4, 5), and positioned in the plot center
    a. There are three element type landmarks, three fuel type landmarks, and Arena
        i. Carbon Landmark,
        ii. Silicon Landmark,
        iii. Hydrogen Landmark (Eternal Spring),

iv. Crypton Landmark,
v. Hyperion Landmark,
vi. Solon Landmark (Fallen Star),
vii. Arena

b. Landmark size is implied to range from 2x2 to 4x4 tiles

8. Internal Land Structure, depending on the Tier ID, may contain different amount of resource sites (element, fuel), may contain the landmark or not
    a. These amounts are expected to be supplied externally (see Functional Requirements subsection in Land Sale section)
    b. For an element site its type is randomly picked from (Carbon, Silicon, Hydrogen)
    c. For a fuel site its type is randomly picked from (Crypton, Hyperion, Solon)
    d. Resource site coordinates within a plot are picked randomly, sites cannot coincide
9. The metadata should be immutable:
    a. It should be possible to predefine it and modify it for non-existing token
    b. It should not be possible to change it for an existing token
10. It should not be possible to mint a token without metadata
11. It should not be possible to mint two or more tokens with the same coordinates (x, y) in the same region, in other words (x, y, Region ID) combination must be unique for a token
12. It should not be possible to mint a token with coinciding sites; site size is implied to be 2x2 tiles (up to 4x4 for a landmark)

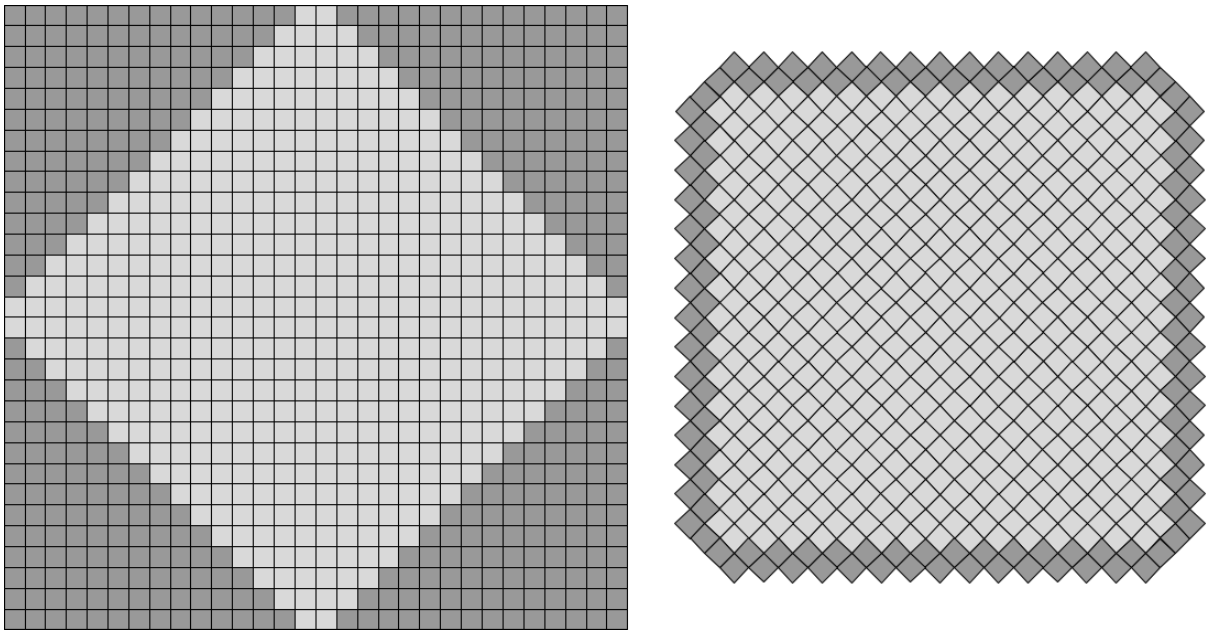


Diagram 3. Isomorphic coordinate grid. Corners of the square grid on the left are “cut off” (dark gray), the resulting isomorphic grid is on the right (light gray).

Non-functional Requirements

1. Land ERC721 smart contract expects the following metadata values to be supplied externally from the trusted source (Land Sale smart contract) when minting a Land ERC721 token:
    a. Region ID
    b. Coordinates (x, y)
    c. Tier ID
    d. Plot Size
    e. Landmark Type ID
    f. Number of Element Sites
    g. Number of Fuel Sites
    h. Generator Version
    i. Seed
2. Region ID, Coordinates (x, y), Tier ID, and Plot Size are predefined properties of the land, pre-stored off-chain before sale starts and available during the sale to be used to mint Land ERC721 tokens;
    a. These values should be written on-chain “as is” when tokens are minted
3. Internal Land Structure, on the other hand, is not defined until the plot is minted;
    a. Landmark Type ID, Number of Element Sites, and Number of Fuel Sites should be supplied externally (generated by Land Sale smart contract based on the Tier ID)
    b. Collection of Resource Sites should be deterministically derived from Number of Element/Fuel Sites, Plot Size, Generator Version (optionaly), and random Seed
4. Collection of Resource Sites should not be stored on-chain, the Seed which unambiguously defines it (together with Number of Element/Fuel Sites, Plot Size, and Generator Version) should be stored instead
5. Seed generation requirements are defined in Non-functional Requirements of the Land Sale section.
6. To simplify the Internal Land Structure generation algorithm, following non-functional limitations for the resource sites positioning are accepted:
    a. Grid size used for resource site generation can be rounded down from the plot size to be a multiple of 4
    b. Resource sites can be positioned on the even coordinates (x, y) only, odd coordinates (x, y) can be dropped
    c. Isomorphic grid form can be defined as per Diagram 3, with roughly 50% tiles in use

Data Structures

Due to some limitations Solidity has (ex.: allocating array of structures in storage), and due to the specific nature of internal land structure (landmark and resource sites data is deterministically derived from a pseudo-random seed), it is convenient to separate data

structures used to store the metadata on-chain, and data structures used to present the metadata via Land ERC721 ABI.

The table below summarizes the data structures used in the Land ERC721 smart contract. Note that primitive types are frequently used for storage purposes directly without defining the wrapper structure (marked with asterisk).

Solidity Name	Description
`PlotView`	Land Plot View, represents the Land Plot Metadata in a human-readable format, contains Version, Region ID, Coordinates (x, y), Tier ID, Plot Size, Number of Element/Fuel Sites, and Internal Land Structure
`SiteView`	Resource Site View, represents a Resource Site metadata in a human-readable format, contains Type ID, and Coordinates (x, y)
`PlotStore`	Land Plot Store, represents the land token metadata as it is stored in the smart contract, contains Version, Region ID, Coordinates (x, y), Tier ID, Plot Size, Number of Element/Fuel Sites, and Seed used to derive Internal Land Structure
`*uint256 PlotStore.pack()`	Land Plot Store tightly packed into a primitive – 256-bit integer: Version | Region ID | x | y | Tier ID | Plot Size | Number of Element Sites | Number of Fuel Sites | Seed
`*uint40 PlotView.loc()`	Land Plot Location, represents the land plot location – Region ID, Coordinates (x, y); tightly packed into a primitive
`*uint32 SiteView.loc()`	Resource Site Location, represents the resource site location – Coordinates (x, y); tightly packed into a primitive

Table 1. Summary of the data structures used in the Land Token smart contract.

Regions, resource site types, and landmark types are stored as numbers (IDs). The tables below describe the conventions used for resource sites and landmarks.

Resource Site Type ID	Resource Site Type Name
`1`	Carbon (element)
`2`	Silicon (element)
`3`	Hydrogen (element)
`4`	Crypton (fuel)

5	Hyperion (fuel)
6	Solon (fuel)

Table 2. Summary of the Resource Type ID definitions.

Landmark Type ID	Landmark Type Name
0	*No Landmark*
1	Carbon Landmark
2	Silicon Landmark
3	Hydrogen Landmark (Eternal Spring)
4	Crypton Landmark
5	Hyperion Landmark
6	Solon Landmark (Fallen Star)
7	Arena

Table 3. Summary of the Landmark Type ID definitions.

Implementation Overview

Land Token smart contract implements core ERC721 interface with optional interfaces defined in the ERC721 specification – metadata and token enumeration.

OpenZeppelin based implementation for ERC721 and EIP-1822 is extended with custom AccessControl (see Access Control section), EIP-2981, and land metadata support. Refer to OpenZeppelin documentation for the description of the ERC721 standard and EIP-1822 draft implementations, functions, state variables, etc.

Implementation Details

This section describes the Land Token smart contract implementation details. Note, that OpenZeppelin contracts are not described and not discussed here.

Internal Land Structure (Land Gen)

Each plot of land stores pseudo-random seed associated with it during the sale/minting process by the trusted Land Sale smart contract. The seed is expanded into a sequence of pseudo-random numbers by recursively hashing it with the built-in `keccak256`. This allows for easy and cheap generation of uniformly distributed pseudo-random numbers on any given segment to generate things like Resource Site Type IDs, or Landmark Type ID.

Some complexity arises with the uniformly random positioning of the resource sites on the isomorphic grid. We could, of course, use the original square grid and simply drop and regenerate resource sites in the “invalid” area in the corners. There would be on average 50% of such sites, but the algorithm behaves randomly and there is a possibility for an even bigger number of regenerations required.

But since the exact number of “valid” tiles is $S = H \cdot (1 + H/2)$ the algorithm can be improved by generating the sites on a rectangular area of size $H \times (1 + H/2)$ and then moving the sites from “invalid” areas (in the corners) into the “valid” ones of the same size (in the center). Diagram below illustrates the idea.

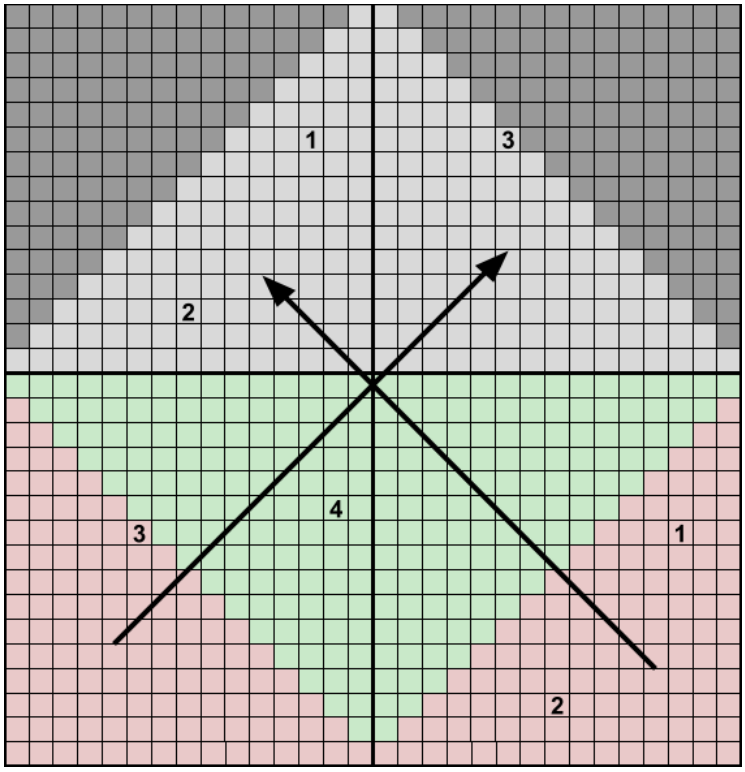


Diagram 4. Uniform rectangular area (bottom) transformed into the isomorphic grid by moving the “invalid” areas in the corners (red) onto the “valid” areas in the center (light gray). Resource sites “1” and “2” are moved from the right-bottom corner, resource site “3” is moved from left-bottom corner, and resource site “4” is not moved since it is already positioned correctly.

Landmarks are special sites positioned in the center of the grid, meaning the resource sites should not occupy the center of the plot. We reserve four spots in the center of the plot and make sure the resource sites are never placed in this area. This is done by decreasing the number of valid tiles by four $S = H \cdot (1 + H/2) - 4$ and relocating the tiles from the center into the free space. Diagram below illustrates the idea.

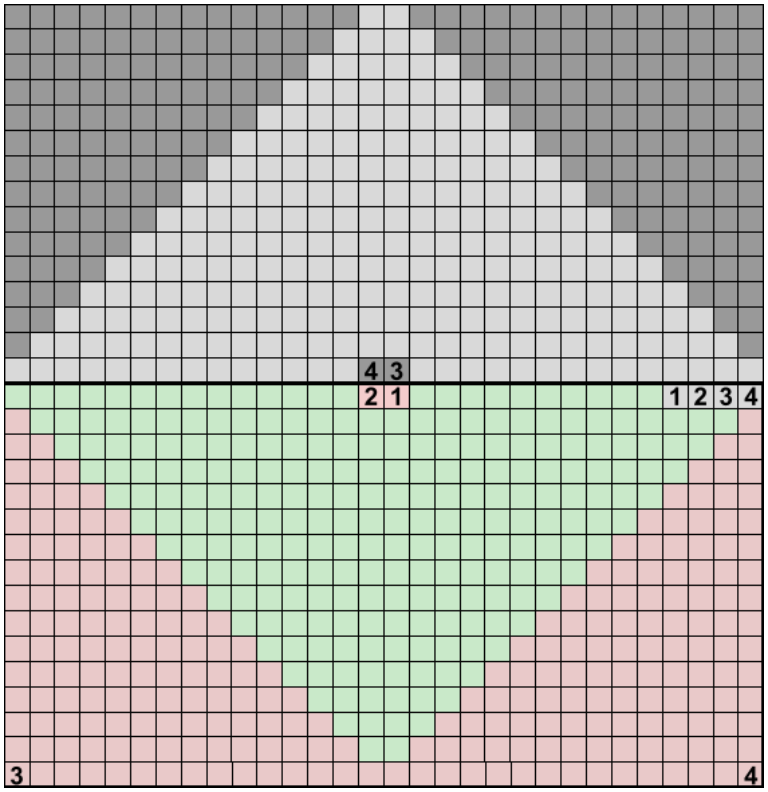


Diagram 5. Reserving the space for a landmark in the center of the plot: four tiles in the center are relocated into the reserved area on the rectangle.

Landmark coordinates are not defined in the Land Token smart contract, it is implied the landmark position is inside the free area in the center of the plot.

To simplify dealing with the 2x2 size of the sites, the algorithms described above are performed on the “normalized” isomorphic grid which is 2 times smaller than the original one. After the positioning is complete, the grid is “stretched” back to its original size. Resource site coordinate stored in the contract is the left-bottom corner (0, 0) of the site. Diagram below illustrates the idea.

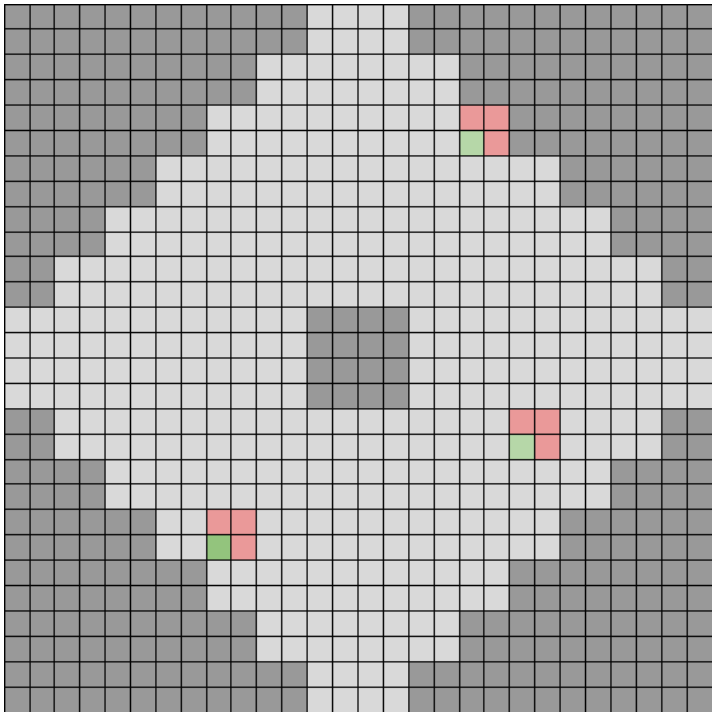


Diagram 6. 2x2 resource sites are positioned on the “normalized” grid, meaning, effectively, that resource sites coordinates are always even, for example: (10, 8), (22, 12), (20, 24).

Refer to the `getResourceSites()` function in the `LandLib.sol` library for exact implementation code for the ideas discussed above.

State Variables

State variables represent the global state of the smart contract. Following state variables are defined in the Land Token smart contract:

Solidity Name	Description
`owner` `* isOwner()`	Fake “owner” to support OpenSea integration
`royaltyReceiver`	EIP-2981 royalty receiver
`royaltyPercentage`	EIP-2981 royalty percent
`contractURI`	Contract level metadata to support OpenSea integration

Table 4. Summary of the state variables in the Land Token smart contract.

Note that smart contract “owner” address doesn’t actually own anything and cannot even change itself without having the `OWNER_MANAGER` permission (see below). This public variable is used only by OpenSea to determine the address eligible for editing token collection on OpenSea.

Mappings

Mappings store special permissions, land metadata, supporting constraints. Following mappings are defined on the Land Token smart contract:

Solidity Name	Description
`plots` `* getMetadata()`	Stores token metadata information $Token\ ID \Rightarrow Land\ Plot\ Metadata$
`plotLocations`	Enforces location uniqueness constraint $Land\ Plot\ Location \Rightarrow Token\ ID$
`userRoles`	Special permissions mapping

Table 5. Summary of the mappings in the Land Token smart contract.

Features and Roles

Features control the behavior of publicly available functions, such as token transfers and burning. In most cases enabling/disabling a feature allows to enable/disable corresponding functionality. Features are stored in `userRoles` mapping for the smart contract itself `userRoles[address(this)]`.

The table below summarizes the features defined in the Land Token smart contract.

Feature	Bit	Description
`TRANSFERS`	0	Enables own tokens transfers, `transferFrom`
`TRANSFERS_ON_BEHALF`	1	Enables transfers on behalf, `transferFrom`
`OWN_BURNS`	3	Enables burning own tokens, `burn`
`BURNS_ON_BEHALF`	4	Enables burning on behalf, `burn`

Table 6. Summary of the Land Token smart contract features.

Roles control access to smart contract functions not to be used publicly (restricted functions). These include minting, burning, etc. Roles are stored in `userRoles` mapping (excluding self address which stores features).

The table below summarizes the roles defined in the Land Token smart contract.

Role	Bit	Description
`TOKEN_CREATOR`	16	Allows minting tokens, `mint`
`TOKEN_DESTROYER`	17	Allows burning any tokens, `burn`

`URI_MANAGER`	18	Allows setting base URI, token URI, `setBaseURI`, `setTokenURI`
`RESCUE_MANAGER`	19	Allows “rescuing” (sending away) ERC20 tokens “lost” (sent by mistake), `rescueErc20`
`ROYALTY_MANAGER`	20	Allows setting EIP-2981 royalty info, `setRoyaltyInfo`
`OWNER_MANAGER`	21	Allows setting smart contract “owner”
`METADATA_PROVIDER`	22	Allows setting token (land) metadata
`UPGRADE_MANAGER`	254	Reserved in `UpgradeableAccessControl` smart contract, allows implementation upgrade via proxy
`ACCESS_MANAGER`	255	Reserved in `UpgradeableAccessControl` smart contract, allows modifying features and roles

Table 7. Summary of the Land Token smart contract special permissions (roles).

Public Functions

The table below summarizes public functions designed to transfer tokens between accounts, burn tokens and authorize other addresses to perform these operations on token owner behalf.

Function Name	Description
`transferFrom`	ERC721 transfer, transfers own tokens, or transfers tokens on behalf of address which approved the transfer
`safeTransferFrom`	Same as `transferFrom`, but checks if the receiver is either an EOA or `ERC721TokenReceiver` smart contract; executes `onERC721Received` and validates the response
`approve`	ERC721 approve, approves single token transfer on behalf and burning a single token on behalf of the account which invoked `approve`
`setApprovalForAll`	ERC721 operator approval, approves/revokes token transfers on behalf and burning tokens on behalf of the account which invoked `setApprovalForAll`
`burn`	Burns own token, or burns token on behalf of address which approved burning; deletes metadata

Table 8. Summary of public functions

Restricted Functions

The table below summarizes restricted access functions to be used by smart contract administrators and approved helpers to interact with the Land Token smart contract.

Function Name	Description
`mint`	Mints single token which metadata is pre-stored
`safeMint`	Same as `mint`, but checks if the receiver is either an EOA or `ERC721TokenReceiver` smart contract; executes `onERC721Received` and validates the response
`mintWithMetadata`	Sets token metadata and mints the token
`mintFor`	IMX-compatible version of `mintWithMetadata`, accepts the `mintingBlob` param containing `tokenId` and `metadata` in the `{tokenId}:{metadata}` format
`burn`	Burns tokens on behalf of any address, independently of if it approved operation or not; deletes metadata
`setMetadata`	Pre-sets token metadata (for non-existing token)
`removeMetadata`	Removes token metadata (for non-existing token)
`setBaseURI`	Sets the base URI used to construct token URIs for the tokens without URIs explicitly set
`setTokenURI`	Sets individual token URI
`setContractURI`	Sets the contractURI (contract level metadata)
`setRoyaltyInfo`	Sets the EIP-2981 royalty receiver and percent
`transferOwnership`	Sets the fake “owner” public variable
`rescueErc20`	“Rescues” ERC20 tokens accidentally sent into the NFT smart contract (transfers them)

Table 9. Summary of the functions with restricted access.

Note that Land Token smart contract admin may revoke the permission to access any of the restricted functions from any address, including its own. This means that any of the restricted functions can be disabled forever.

Tests

Resource site positioning on the isomorphic grid with the free area in the center of the plot is a non-trivial algorithm and to guarantee the resource sites are positioned correctly there are several tests performed on a large amount of the generated plots:

1. 10,000 seeds a picked randomly
2. For each seed picked, plots of various sizes and tiers are generated:
    a. Tiers: 1, 2, 3, 4, 5
    b. Sizes: 16 – 128
3. For each generated plot following categories of tests are made:
    a. Non-collision test checks that there are no colliding sites
    b. Isomorphic test checks there are no sites spanning outside the isomorphic grid
    c. Free center test checks there are no sites spanning inside the area in the center of the plot
    d. Randomization test checks the resource site locations “look random” (see below)

Randomization test “stacks” all the generated plots of the given size into one and examines how many resource sites reside (on average) in the different areas of the plot. It picks a random rectangle on the grid, estimates what would be the average amount of the resource sites inside and compares the value with the real one calculated by “stacking” the generated plots. Allowed deviation is 15%.

ASCII illustration output from the test is represented below.

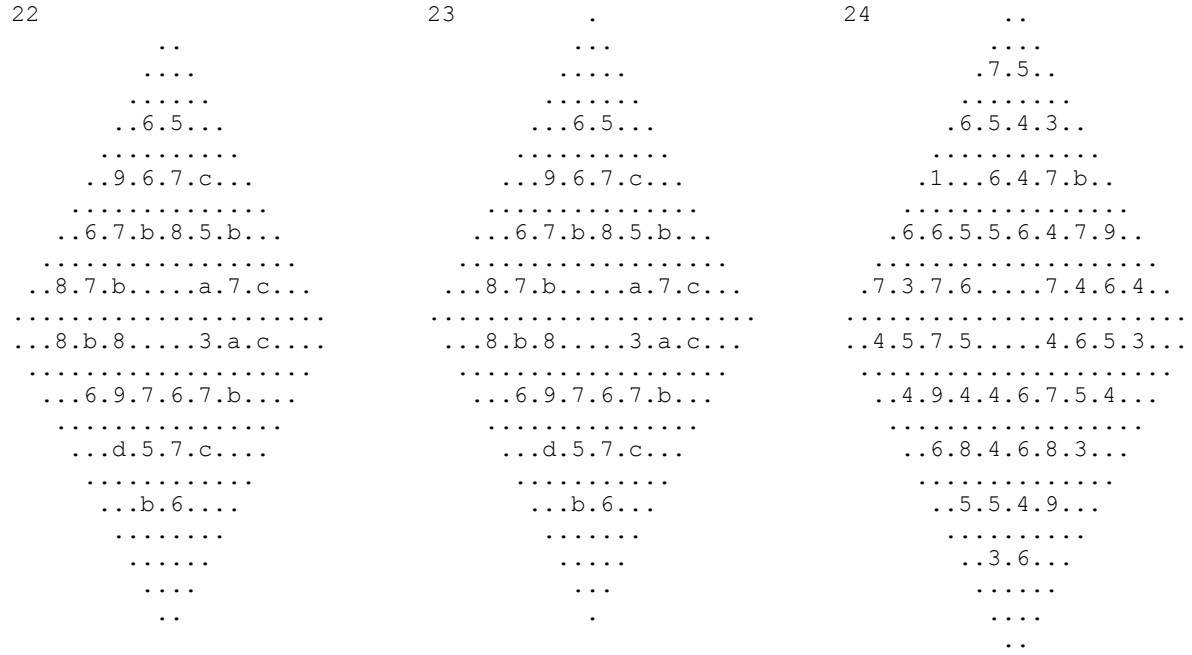


Diagram 7. ASCII illustration for the series of tests for 100 tier 3 plots of sizes 22, 23, and 24 stacked into one plot of the corresponding size. Dots denote an area on the isomorphic grid. Numbers (letters) denote cumulative number of resource sites found at a particular coordinate (x, y). Y-axis is oriented from top to bottom (ASCII console output). Sites occupy 2x2 area on the

grid, spreading one dot to the right and one dot to the bottom from the number (letter) denoting the site.

Refer to the `isomorphic_grid.js` for exact test implementation code of the discussed above.

Land Sale

Land Sale smart contract powers land sale campaigns, and enables the minting and purchase of Land NFTs (also referred to as items on sale, or items). It uses Dutch auction model pricing with the exponential price decline, when price halves every 34 minutes (configurable). Sale smart contract is reusable, it can be used to run several similar campaigns. First sale campaign sells 20,000 tokens, 100,000 in total (both values are configurable).

Since minting a plot requires at least 32 bytes of data (one EVM storage slot) and due to a significant amount of plots to be minted (about 100,000), pre-storing this data on-chain is not a viable option (2,000,000,000 of gas only to pay for the storage). Instead, the whole land plot data collection on sale is represented as a Merkle tree structure, the root of the Merkle tree is stored on-chain in the sale smart contract. To buy a particular plot, the buyer must know the entire collection and be able to generate and present the Merkle proof for this particular plot.

The input data is a collection of `PlotData` structures; the Merkle tree is built out from this collection, and the tree root is stored on the contract by the data manager. When buying a plot, the buyer also specifies the Merkle proof for a plot data to mint.

Functional Requirements

1. Sale contract sells the predefined collection of land tokens via the Dutch auction model:
    a. The sale duration is three days, start time is to be defined
    b. Tokens are sold in sequences, sequence groups tokens which are sold in parallel
    c. One sequence includes/groups 278 tokens
    d. Sequences start selling every hour, one sequence is on sale for two hours
    e. The price of a token on sale declines by 2% every minute passed
    f. The starting price for the tokens is defined in ETH:
        i. Tier 1: 0.5 ETH
        ii. Tier 2: 1.5 ETH
        iii. Tier 3: 5 ETH
        iv. Tier 4: 20 ETH
        v. Tier 5: N/A[2]
2. All the configuration parameters mentioned above are configurable
3. Land plots can be bought with either ETH or sILV, assuming the price of sILV is equal to ILV price

[2] Although LandSale smart contract supports Tier 5 plots, it won't be used to sell them; these plots will be pre-minted manually and sold via Silent Auction mechanism

4. Land tokens collection metadata is stored off-chain, it is written on-chain for each token individually when the token is sold, as a part of token minting process
5. Each metadata record in this predefined collection has the following structure:
    a. Token ID, defines a land token to buy/mint
    b. Sequence ID, defines the timeframe when a particular token is sold
    c. Region ID, [1–7], one of seven regions
    d. Coordinates (x, y) within the Region
    e. Tier ID, [1–5], one of five tiers
    f. Size, land plot size limits the (x, y) coordinates for the resource sites
6. Plot metadata above is supplied by the buyer together with the Merkle proof generated from the entire collection
7. Sale contract validates the supplied land plot belongs to the collection on sale by validating the proof against the pre-stored Merkle root of the entire collection
8. Sale contract enriches the metadata supplied with:
    a. Landmark Type ID, Number of Resource Sites, derived from the Tier ID:
        i. Tier 1: no landmark, 3 element sites, 1 fuel site
        ii. Tier 2: no landmark, 6 element sites, 3 fuel sites
        iii. Tier 3: element landmark, 9 element sites, 6 fuel sites
            1. Element landmark type is randomly picked from (Carbon, Silicon, Hydrogen)
        iv. Tier 4: fuel landmark, 12 element sites, 9 fuel sites
            1. Fuel landmark type is randomly picked from (Crypton, Hyperion, Solon)
        v. Tier 5: arena landmark, 15 element sites, 12 fuel sites
    b. The pseudo-random seed used by Land ERC721 contract to derive the Collection of Resource Sites
9. Authorized address(es) should be able to set up the sale (initialize the sale) with the following parameters required to run the sale:
    a. Sale Duration, start/end dates – time period when the sale is operational, this should be in sync with the sequence parameters (default duration is 3 days)
    b. Halving Time – token price halving time (default is 34 minutes to simulate 2% drop every minute passed)
    c. Sequence Duration – time limit of how long a token / sequence can be available for sale (default is 2 hours)
    d. Sequence Offset – time interval between the start of two adjacent sequences (default is 1 hour)
    e. Start Prices – starting price of the token for each Tier ID available on sale
10. Authorized address(es) should be able to update sale setup (reinitialize the sale) at any time – before, during, or after the sale event
11. Authorized address(es) should be able to set all the sale parameters or any subset of these parameters in a single transaction
12. Authorized address(es) should be able to pause the sale at any time, and resume it later on at any time, without losing any tokens on sale, and without losing their price:
    a. Pausing and resuming before the scheduled Sale Start doesn't have any effect

    b. Pausing before the scheduled Sale Start and resuming after it, shifts the Sale Duration, so that it starts immediately when resumed; this doesn't change time interval between sale start/end time
    c. Pausing after the Sale Start for any period of time, expands the Sale Duration by that period of time, so that it ends later; tokens available on sale when paused remain available on sale when resumed, at the same price
    d. Updating the Sale Start (sale initialization) resets sale pause state
13. Authorized address(es) should be able to set or update the Merkle root for the entire land plot collection on sale at any time – before, during, or after the sale event
14. It should be possible to buy a token on sale if
    a. the sale is properly initialized
        i. halving time is set
        ii. sequence duration is set
        iii. start price for the tier to buy is set
        iv. land plot collection Merkle root is set
    b. sale is active (current time is within Sale Duration)
    c. sale sequence for a token is active (current time is within Sequence Duration)
    d. the token metadata is correct (belongs to the collection on sale, verified using the proof supplied)
    e. buyer supplies enough funds – ETH in the transaction, or allows enough sILV to be taken
15. "Buy" transaction is expected to fail (revert) if the requirements above are not met
    a. The transaction is expected to succeed if ETH supplied exceeds the expected value; the excess amount is returned back to sender
16. Authorized address(es) should be able to set the "beneficiary" – an address which receives the funds obtained from the token sale – via push on every buy transaction
17. If the beneficiary address is not set – funds obtained from the sale should accumulate on the sale smart contract
18. Authorized address(es) should be able to withdraw funds accumulated on the sale contract
19. Authorized address(es) should be able to rescue ERC20 tokens accidentally sent to the sale smart contract
20. Implements "EIP-1822: Universal Upgradeable Proxy Standard (UUPS)"
    a. Authorized address(es) should be able to do the smart contract upgrade
21. Sale contract supports L1 sales (attached mode) and L2 sales (detached mode)
    a. In the attached mode tokens get minted immediately as part of the sale transaction
    b. In the detached mode tokens don't get minted immediately, sale transaction only emits an event containing token metadata and owner;
        i. this event is then picked by the off-chain worker (daemon) which mints the token in Layer 2 network (like IMX, or other)
        ii. description of the off-chain minting process is out of scope for this document

Non-functional Requirements

Initialization and Reinitialization

1. Initialization and reinitialization is done with the single function, taking the following parameters as inputs:
    a. Sale Start, unix timestamp, uint32, $[1,\ 2^{32} - 2]$
    b. Sale End, unix timestamp, uint32, $[1,\ 2^{32} - 2]$
    c. Halving Time, uint32, $[1,\ 2^{32} - 2]$
    d. Time Flow Quantum, uint32, $[1,\ 2^{32} - 2]$
    e. Sequence Duration, uint32, $[1,\ 2^{32} - 2]$
    f. Sequence Offset, uint32, $[1,\ 2^{32} - 2]$
    g. Start Prices, uint96 array, each element range $[1,\ 2^{96} - 2]$, maximum price is about $7.9 \cdot 10^{18}$ ETH
2. “Time Flow Quantum”, also referred to as “Price Update Interval”, is the time interval (seconds), used to round down current time value when doing price calculations
3. “Sale Duration” is a time interval between Sale Start (inclusive) and Sale End (exclusive)
4. “Sequence Duration”, when referred to a particular sequence [ID], is a time interval between Sequence Start (inclusive) and Sequence End (exclusive):
$$SequenceDuration_i = [SequenceStart_i,\ SequenceEnd_i),$$
where i is a zero-based sequence ID
    a. Sequence Start is derived from its ID, and Sequence Offset:
$$SequenceStart_i = i \cdot SequenceOffset$$
    b. Sequence End is derived from Sequence Start and Sequence Duration (global parameter):
$$SequenceEnd_i = SequenceStart_i + SequenceDuration$$
5. Any parameters can be “skipped” during initialization/reinitialization. “Skipping” means the values remain as were previously set. “Skipping” is achieved using the special value(s) for the parameters to be omitted:
    a. Sale Start: value $2^{32} - 1$, `0xFFFFFFFF`
    b. Sale End: value $2^{32} - 1$, `0xFFFFFFFF`
    c. Halving Time: value $2^{32} - 1$, `0xFFFFFFFF`
    d. Time Flow Quantum: value $2^{32} - 1$, `0xFFFFFFFF`
    e. Sequence Duration: value $2^{32} - 1$, `0xFFFFFFFF`
    f. Sequence Offset: value $2^{32} - 1$, `0xFFFFFFFF`
    g. Start Prices: single element array with value $2^{96} - 1$, `[0xFFFFFFFFFFFFFFFFFFFFFFFF]`

6. Setting the Merkle root for the land tokens collection on sale is done separately from the parameters discussed above

Seed Generation

1. The pseudo-random seed used to generate Internal Land Structure is not required to be VRF based, or supplied externally, and can be derived from minting transaction parameters, such as address of the land plot buyer, block timestamp, etc.
2. The pseudo-random seed stored should be at least 160 bits long

Data Structures

Land Sale smart contact introduces a single data structure – Off-chain Land Plot Metadata – `PlotData`. It is somewhat similar to the Land Plot View structure used in the Land ERC721 smart contract. It doesn't have Internal Land Structure, and binds metadata records to the Token ID and Sequence ID.

Solidity Name	Description
`PlotData`	Land Plot Data, represents the Off-chain Land Plot Metadata, in a human-readable format, contains Token ID, Sequence ID, Region ID, Coordinates (x, y), Tier ID, and Plot Size

Table 10. Summary of the data structures used in the Land Sale smart contract.

Constructing the Merkle Proof

When buying a particular plot, Merkle proof needs to be constructed and supplied together with the plot metadata.
Merkle tree and proof can be constructed using the `web3-utils`, `merkletreejs`, and `keccak256` npm packages:

1. Hash the plot data collection elements via `web3.utils.soliditySha3`, making sure the packing order and types are exactly as defined in `PlotData` struct
2. Create a sorted MerkleTree (`merkletreejs`) from the hashed collection, use `keccak256` from the `keccak256` npm package as a hashing function, do not hash leaves (already hashed in step 1); Ex. MerkleTree options: `{hashLeaves: false, sortPairs: true}`
3. For any given plot data element the proof is constructed by hashing it (as in step 1), and querying the MerkleTree for a proof, providing the hashed plot data element as a leaf

Refer to the Solidity source code and JavaScript test cases for the code examples on how the proof is verified and how it should be generated.

Implementation Overview

Land Sale uses custom upgradeable Access Control similarly to the Land ERC721 Token implementation. See Access Control section for more details.

"Buy" public functions (`buyL1` and `buyL2`) are also protected with the feature flags. The sale can also be effectively inactivated at any time by "corrupting" the initialization params, for example, by setting halving time to zero, unsetting starting prices, unsetting the Merkle root, etc.

Buy Flow

When buying, the Sale performs the following actions:

1. State and input verification
    a. Check: sale is enabled (feature flag check)
    b. Check: sale is properly initialized
        i. halving time is set
        ii. sequence duration is set
        iii. start price for the tier to buy is set
        iv. land plot collection Merkle root is set
    c. Check: sale is active (current time is within Sale Duration)
    d. Check: sale sequence for a token is active (current time is within Sequence Duration)
    e. Check: the token metadata is correct (belongs to the collection on sale, verified using the proof supplied)
    f. Check: ETH supplied in tx, or sILV allowed to be transferred, covers token price
2. Payment processing
    a. Calculates land token price (ETH) based on its tier ID, starting price for that tier, sequence ID, and current time
        i. If sale was paused for some period of time, the current time is adjusted to count for the time sale was paused
    b. If no ETH is supplied in the tx, processes sILV payment
        i. Converts ETH price into sILV price via Uniswap V2 Price Oracle for ETH/ILV pair
        ii. Transfers the sILV amount to the "beneficiary" (if set), or to the sale smart contract otherwise
    c. Otherwise, if ETH is supplied in the tx, processes ETH payment
        i. Transfers the ETH amount to the "beneficiary" (if set), or leaves it in the sale smart contract otherwise
        ii. Transfers the excess ETH amount back to tx sender
        iii. ETH transfers (if applicable) are done via 2,300 gas transfer[3]
3. Land plot token creation

[3] Land Sale contract is designed to operate for several days; it can be reused several times, but this is not required, and if gas cost change in such a way that using 2,300 gas transfer becomes an issue this will be addressed once such change takes place

a. Generates a pseudo-random seed by hashing the Token ID, current timestamp, and buyer address
b. Derives the Landmark Type ID from the Tier ID (randomizing it based on the generated seed for tiers 3 and 4)
c. Derives the Number of Element and Fuel Sites from the Tier ID
d. Mints the token with the metadata specified, and a hash of the generated pseudo-random seed as the token metadata Seed
    i. In the detached mode token is minted in a separate transaction executed by the off-chain worker (daemon)
e. Internal Land Structure is derived from the generated pseudo-random seed by the Land ERC721 token smart contract view function(s); it's generation is not part of the minting flow

Withdrawal Flow

When withdrawing accumulated ETH/sILV from the sale, the Land Sale performs the following actions:

1. State and input verification
    a. Check: tx sender is authorized to withdraw
    b. Check: recipient address is set
    c. Check: there is ETH/sILV to withdraw
2. Transfer accumulated ETH/sILV to the recipient specified

Implementation Details

This section describes the Land Sale smart contract implementation details, including Dutch auction price calculation algorithm.

Price Calculation

Implementing the "decrease price by x (percent) for every y (seconds) passed" as is, for example, with the "for" loop applying the x decrease several times can be dangerous in Solidity: for a time interval t passed big enough, $t \gg y$, this would result in the "for" loop of length $l \gg 1$, potentially increasing the gas price unreasonably high.

Storing the previously calculated price value for every sequence in a smart contract storage looks more predictable in terms of gas usage, but still far from ideal.

Instead, we approximate the price formula with the exponential decay function:

$$p(t) = p_0 \cdot 2^{-t/t_0},$$

where p_0 is the initial price, $t_0 = \frac{-y}{log_2(1-x)}$ is the price halving time, and $p(t)$ is a price at any given moment t.

Approximation error doesn't exceed the 0.1% value, which is illustrated on the diagrams below.

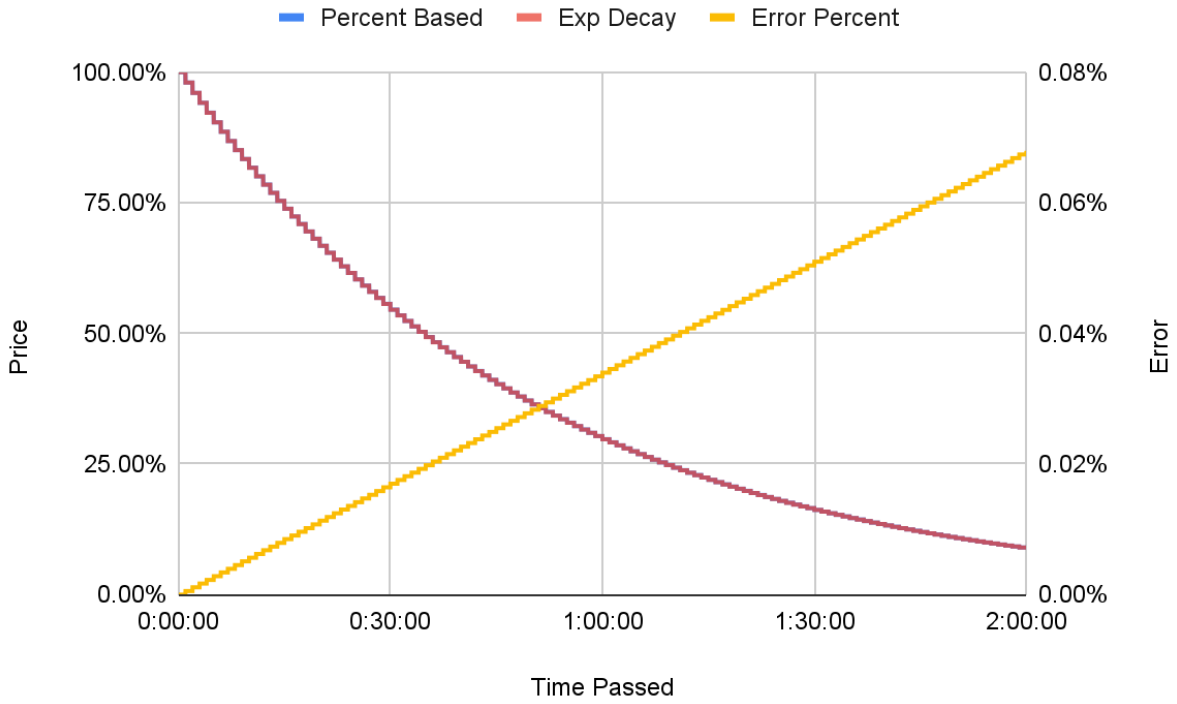


Diagram 6. Comparison of the Dutch auction price calculation done "as is" (blue line), and an exponential decay approximation (red line). The error (yellow line) is so small that red and blue lines are indistinguishable. Price drops by **2%** every **minute** during **2** hours. Calculated halving time t_0 is **2,058** seconds.

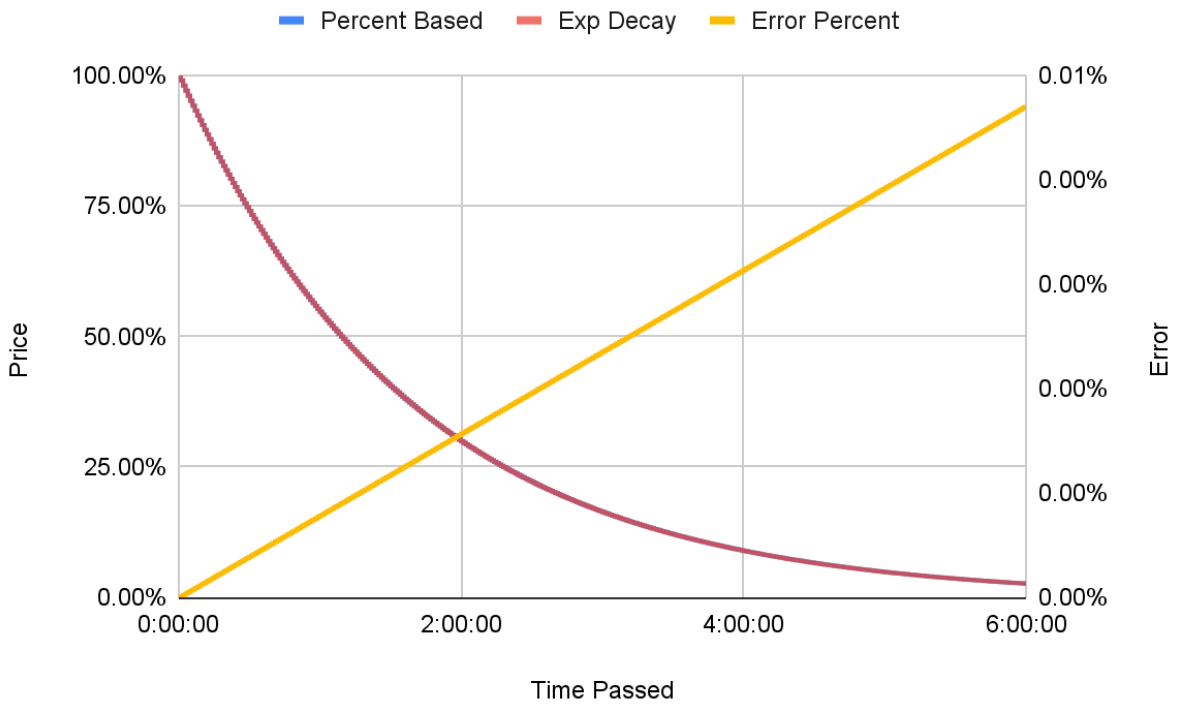


Diagram 7. Comparison of the Dutch auction price calculation done “as is” (blue line), and an exponential decay approximation (red line). The error (yellow line) is so small that red and blue lines are indistinguishable. Price drops by **1%** every **minute** during **6** hours. Calculated halving time t_0 is **4,138** seconds.

Solidity Implementation

Solidity has a limited support for numeric types (ex.: no floating points) and math functions (ex.: no negative power) and implementing the exponential decay formula “as is” is not possible.

It can be approximated, however, with any desired precision, based on the following sequence of considerations:

1. The price halves every t_0 seconds passed
2. The price drops by $\sqrt{2}$ every $\frac{t_0}{2}$ seconds passed
3. The price drops by $\sqrt[4]{2}$ every $\frac{t_0}{4}$ seconds passed
4. …
5. The price drops by $\sqrt[2^n]{2}$ every $\frac{t_0}{2^n}$ seconds passed

Based on this idea, we are able to calculate the price $p(t')$ at a time t' close any given point in time t:

$$p(t') = \sum_{0}^{n} a_i \cdot 2^{-1/2^i} \cdot p_0,$$

$$t' = \sum_{0}^{n} a_i \cdot \frac{t_0}{2^i},$$

where $a_i = \{0, 1\}$, and the deviation $\Delta t = |t - t'| \leq \frac{t_0}{2^n}$.

For any desired time deviation $\Delta t = |t - t'|$ we may choose n to satisfy it:

$$n = log_2\left(\frac{t_0}{\Delta t}\right)$$

In Ethereum mainnet blocks are mined on average every 14 seconds, and there is no need to make Δt smaller than that. $n = 8$ would give us enough precision to cover halving times up to one hour.

Eight roots of 2 are pre calculated as per table below:

i	Symbol	Value
1	$\sqrt{2}$	`1.414213562373095`
2	$\sqrt[4]{2}$	`1.189207115002721`
3	$\sqrt[8]{2}$	`1.090507732665257`
4	$\sqrt[16]{2}$	`1.044273782427413`
5	$\sqrt[32]{2}$	`1.021897148654116`
6	$\sqrt[64]{2}$	`1.010889286051700`
7	$\sqrt[128]{2}$	`1.005429901112802`
8	$\sqrt[256]{2}$	`1.002711275050202`

Table 11. Pre calculated square roots values for $n = 8$

Refer to the `price(p0, t0, t)` function in `LandSale.sol` for exact implementation code for the algorithm described above.

State Variables

State variables represent the global state of the smart contract. Following state variables are defined in the Sale smart contract:

Solidity Name	Description
`root`	Merkle tree root for the land plots, available on sale
`saleStart`	Sale start unix timestamp
`saleEnd`	Sale end unix timestamp
`halvingTime`	Dutch auction price halving time in seconds
`timeFlowQuantum`	Price update interval in seconds
`seqDuration`	Sequence duration in seconds
`seqOffset`	Sequence offset in seconds
`pausedAt`	Unix timestamp when sale was paused, or zero if sale is not paused
`pauseDuration`	Cumulative sale pause duration, total amount of time i seconds sale stayed in a paused state
`startPrices`	Starting prices(array) for each tier, starting from Tier 0, defined in wei
`beneficiary`	The address used to send the funds obtained on the sale via push mechanism (for each buying tx)
`targetNftContract`	Land ERC721 NFT smart contract address, immutable, set upon the deployment
`sIlvContract`	Escrowed Illuvium Token (sILV) smart contract address, immutable, set upon the deployment
`priceOracle`	Land Sale Price Oracle smart contract address, immutable, set upon the deployment

Table 12. Summary of the state variables in the Sale smart contract.

Mappings

Mappings store special permissions, land metadata, supporting constraints. Following mappings are defined on the Land Token smart contract:

Solidity Name	Description

`mintedTokens`	A bitmap of minted tokens, including tokens scheduled to be minted in L2 (detached mode)
`userRoles`	Special permissions mapping

Table 13. Summary of the mappings in the Land Token smart contract.

Features and Roles

Land Sale introduces a single feature to control the “buy” public function.

Feature	Bit	Description
`L1_SALE_ACTIVE`	0	Enables the L1 sale (attached mode), `buyL1`
`L2_SALE_ACTIVE`	1	Enables the L2 sale (attached mode), `buyL2`

Table 14. Summary of the Land Sale smart contract features.

Roles control access to smart contract functions not to be used publicly (restricted functions). These include initialization/reinitialization, ETH withdrawal, etc. Roles are stored in `userRoles` mapping (excluding self address which stores features).

The table below summarizes the roles defined in the Sale smart contract.

Role	Bit	Description
`PAUSE_MANAGER`	16	Allows pausing/resuming the sale, `pause/resume`
`DATA_MANAGER`	17	Allows setting the land token collection Merkle root, `setInputDataRoot`
`SALE_MANAGER`	18	Allows sale initialization and reinitialization, `initialize`
`RESCUE_MANAGER`	19	Allows “rescuing” (sending away) ERC20 tokens “lost” (sent by mistake), `rescueErc20`
`WITHDRAWAL_MANAGER`	20	Allows withdrawal of the accumulated funds (ETH/sILV), `withdraw`, `withdrawTo` Allows setting the beneficiary address, `setBeneficiary`
`UPGRADE_MANAGER`	254	Reserved in `UpgradeableAccessControl` smart contract, allows implementation upgrade via proxy
`ACCESS_MANAGER`	255	Reserved in `UpgradeableAccessControl` smart contract, allows modifying features and roles

Table 15. Summary of the Land Sale smart contract special permissions (roles).

Public Functions

The table below summarizes public functions designed to buy land plots.

Function Name	Description
`buyL1`	Sells Land NFT to tx sender in L1 (attached mode)
`buyL2`	Sells Land NFT to tx sender in L2 (detached mode)

Table 16. Summary of public functions.

Restricted Functions

The table below summarizes restricted access functions to be used by smart contract administrators and approved helpers to interact with the Land Sale smart contract.

Function Name	Description
`setInputDataRoot`	Sets the Merkle tree root for the land plots collection on sale
`initialize`	Updates sale parameters related to timings, prices
`pause`	Pauses the sale
`resume`	Resumes the sale
`setBeneficiary`	Sets the address receiving the funds obtained from sale
`withdraw`	Withdraws accumulated funds to tx sender
`withdrawTo`	Withdraws accumulated funds to the recipient specified
`rescueErc20`	“Rescues” ERC20 tokens accidentally sent into the sale smart contract (transfers them)

Table 17. Summary of the functions with restricted access.

Note that Land Sale smart contract admin may revoke the permission to access any of the restricted functions from any address, including its own. This means that any of the restricted functions can be disabled forever.

Land Sale Price Oracle

Land Sale Price Oracle is a helper smart contract which provides the Land Sale contract with the sILV price; this price is required by the Land Sale to sell land for sILV instead of ETH.

Functional Requirements

1. Given the ETH price of the item on sale (Land NFT), it should be possible to get sILV price of the item
    a. ETH price of the item is an amount of ETH required to buy the item, it is a primarily price of the item
    b. sILV price of the item is an amount of sILV required to buy the item, it is an alternative price of the item
    c. sILV price is considered to be equal to ILV price
2. sILV price should not be older than 30 hours

Non-functional Requirements

1. sILV price precision should not be worse than 5%
2. In case of inability to provide fresh and precise price, oracle should fail the price request

Implementation Overview

Land Sale Price Oracle implementation is straightforward and is based on the Chainlink price feed represented by `AggregatorV3Interface` smart contract instance powered by Chainlink.

It is also upgradeable, and follows the “EIP-1822: Universal Upgradeable Proxy Standard (UUPS)”, which allows switching the implementation to use alternative price sources like Uniswap, etc.

Smart contract doesn’t have any internal state variables, mappings, etc., but only the `AggregatorV3Interface` address.

About

Prepared by Basil Gorin for Illuvium.io to support Land Sale Protocol release 1 (release date April 15, 2022).

Based on the ideas and concepts by Kieran Warwick, Aaron Warwick, John Avery, and Illuvium Core Contributors.

Version 1.1.2 dated March 27, 2022.

References

1. EIP-721: ERC-721 Non-Fungible Token Standard
   https://eips.ethereum.org/EIPS/eip-721
2. EIP-1822: Universal Upgradeable Proxy Standard (UUPS)

https://eips.ethereum.org/EIPS/eip-1822
3. OpenZeppelin AccessControl
   https://docs.openzeppelin.com/contracts/4.x/api/access#AccessControl
4. Verifying Merkle Proofs with OpenZeppelin
   https://docs.openzeppelin.com/contracts/2.x/utilities#verifying_merkle_proofs
5. Using Chainlink Data Feeds
   https://docs.chain.link/docs/get-the-latest-price/
6. ILV/ETH Chainlink Price Feed
   https://market.link/feeds/95acf954-7848-4798-999c-a2c72a29a87e