Estimation of a minimal casino development on top of DECENTRALAND

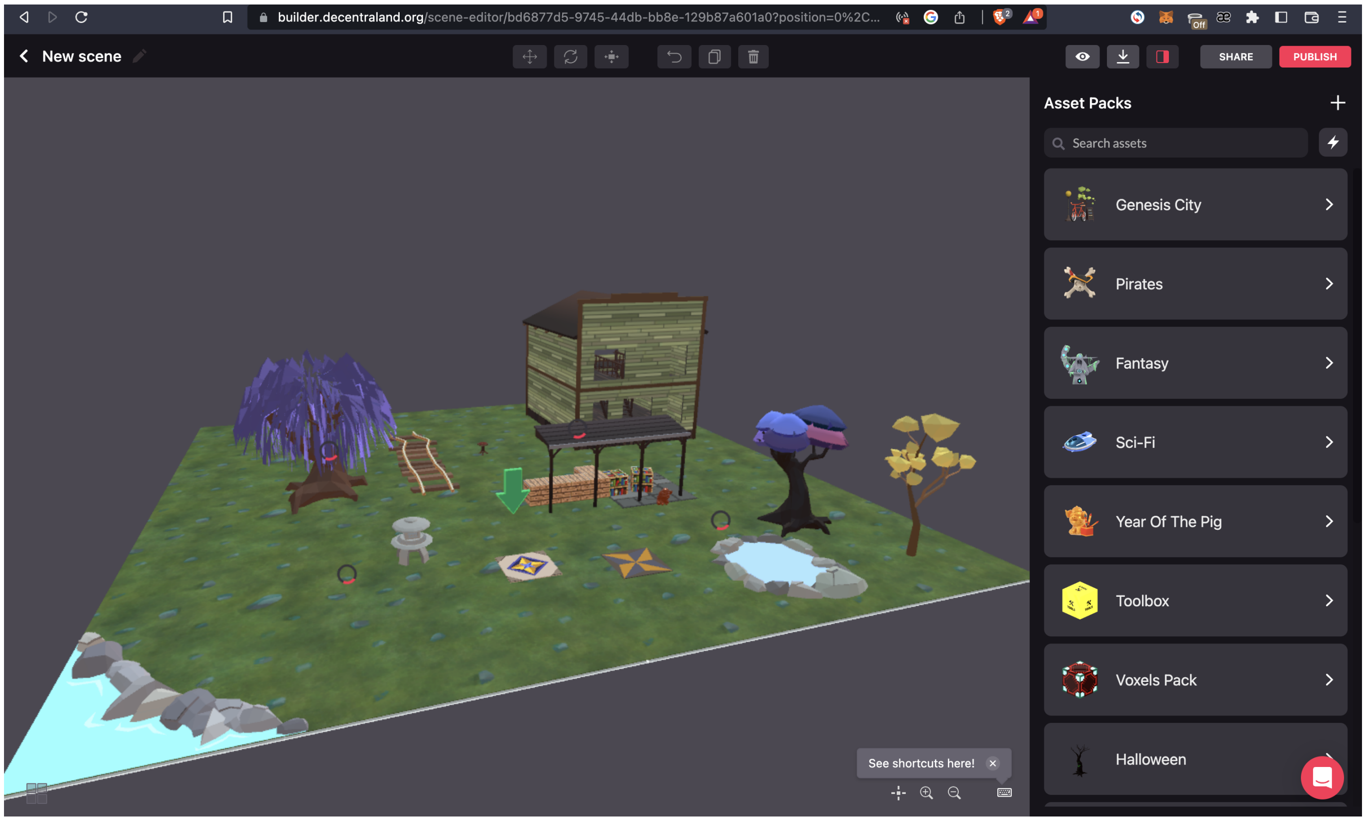
#decentraland #estimation #metaverse #solidity

Author: Nazar Havryliuk (na3aga)

---

1. Building a scene:

Building a scene can be done with decentralands editor. We are able to add our own assets, and graphics + animations can be added with their SDK



Game mechanics and Decentraland functionality can be developed with their SDK using Typescript

- SDK - https://docs.decentraland.org/creator/development-guide/coding-scenes/
- *Note*: they use Entity Component System design
- *Note*: scenes are deployed on 16x16 pieces of LAND called **parcels**, or combinations of them, called **estates** (but if you want a non-rectangular form, you need to use **SDK** instead of GUI **builder**)
- All the logic should be written in an async manner in Typescript and can be tested locally with their editor/builder/CLI

> - Take a look iif interested in card game : [Card Game Creation Kit](#)*
> - [A very useful list of examples & tutorials:](#)

---

2. Blockchain operations in the scene:

The following tools currently exist, all of them provided by Decentraland:

- The `Ethereum Controller` is a basic library that offers limited but simple functionality.
- The `eth-connect` library: A lower-level library to interface with Ethereum contracts and call their functions, for example, to trigger transactions or check balances.

Any operation with blockchain can be integrated into the game logic with these libraries, using TypeScript (as a game itself).

We can divide the operations needed for the game into 3 categories:

1. Reading and converting data/metadata from the blockchain (Or we can use other libraries or APIs)
2. Writing to the blockchain (submitting transactions to SmartContracs)
3. Signing the message by the user (We can do some actions off-chain and save the user's tokens if it's not critical actions and we can omit storing these on the blockchain)

---

3. Smart Contract purpose and example:

let's imagine a very simple Casino game where at least two players make their bets with the `play()` function and after, some special Service(with a private key) updates the `randomBlockHash` value, and any user can call the `winner()` function to determine the winner by random value, transfer the prize and refresh the contract for another game

> Note: this simplification is not perfect and can be vulnerable to various types of manipulation and does not guarantee a fair distribution of winnings. In real-world contracts, it's better to use Decentralized Oracles like **ChainLink** to generate random values. The contract needs to be tested very intensely also and more features like events/indexing and back-end are not presented in this example.

The main point to focus on is that all these functions can be called inside the game by the player for example if he interacts with some entity. Some system functions can be triggered by back-end services. And some data can be indexed and aggregated with other services to read them quicker by the user's client in the game.

Example Contract:

```
pragma solidity ^0.8.0;

contract Token {
    function transferFrom(address from, address to, uint256 value) public returns
(bool);
}

contract CasinoPoker {
    Token public token;
    uint256 public pot;
    uint256 public randomBlockHash;
    address public dealer;

    constructor(Token _token, address _dealer) public {
        token = _token;
        dealer = _dealer;
    }

    function play(uint256 bet) public {
        require(token.transferFrom(msg.sender, address(this), bet), "Token transfer
failed.");
        pot += bet;
    }

    function setRandomBlockHash(uint256 blockHash) public {
        require(msg.sender == dealer, "Only the dealer can set the random block
hash.");
        randomBlockHash = blockHash;
    }

    function winner() public {
        require(pot > 0, "The pot is empty.");

        // Determines the winner based on the random block hash
        uint256 mask = 2**256 - 1;
        address winner = address(uint160(keccak256(abi.encodePacked(randomBlockHash)))
& mask % address(this).balance);

        token.transferFrom(address(this), winner, pot);
        pot = 0;
    }
}
```

Conclusion

All of these parts can be combined together, with some back-end services to create a decentralized gaming experience on top of the DECENTRALIZED platform.

Here's a high-level overview of the steps involved:

1. Choose a blockchain platform: Decentraland supports at least two blockchain platforms: Ethereum, and Polygon (may have some limited functionality but transactions are way cheaper on Polygon). Choose the platform that best fits your needs.
2. Write the smart contract: Write a smart contract in Solidity that implements the game logic, such as betting, shuffling, dealing, and determining the winner.
3. Deploy the contract: Deploy the smart contract to the Testnet of the chosen blockchain platform, making it accessible for tests.
4. Create the Decentraland scene: Create a Decentraland scene that allows players to interact with the smart contract. This can be done using the Decentraland SDK, which provides a suite of tools and assets to help you create interactive 3D experiences.
5. Integrate the contract and scene: Integrate the smart contract and Decentraland scene by adding triggers and events that allow players to interact with the contract and see the results in the scene.
6. Test and deploy: Test your casino game to ensure it's functioning as intended, after, make the Audit of Smart Contracts (it's essential for production) and then deploy the Game to the Decentraland marketplace, where players can discover and play it.