Cryptopoly:
Using Ethereum State Channels for Decentralized Game Applications

by:

Michael Xu

Abstract

All modern multiplayer games are administered by having players connect to a remote server which is used to provide the ground truth for game state and player actions. This use of a central server provides a simple and intuitive way to administer game servers but also provides a single point of failure, as each server must be able to process all actions coming in and make a decision on whether the action is allowed or not, and how to update the game state accordingly. In cases where the server is under significant load, either from a very popular game release or from a deliberate attack, the game slows down or completely crashes. When there is a server action backlog, this can allow malicious actors to perform previously impossible actions. By instead using a decentralized platform, we can build a robust system that allows playing games through a P2P manner, filling in the need for central servers with consensus algorithms that provide the security on the part of a central authority. This project aims to show that a decentralized solution can be used to create a transparent, fully playable game of Monopoly with complex features that would be more scalable, reliable, and cost-effective compared to a centralized solution; meaning that games could be produced that costs pennies to publish and modify, taking seconds to propagate changes globally, and most importantly, cost nothing for upkeep. The codebase is available here: https://github.com/SirNeural/monopoly

Table of Contents

Introduction **3**
Background 3
Motivation 4
Existing Projects 5

Research **5**
Blockchain 5
Scaling 6
State Channels 8
Force Move 9
Randomness 10
Commit Reveal 11
Fate Channels 11

Implementation **11**
High-Level Overview 11
State Machine 13
Data Structures 14
Ethereum Smart Contract 15
Front End 16

Conclusion **19**

References **21**

Introduction

Background

The use of a central server to act as the designated authority to administer and preside over disputed actions within a game is a staple in almost any game genre. Using a central server offers many benefits: namely, since the game publisher controls the servers, they have ultimate control over what happens within their game. This simplifies a lot of tasks, for example with the standard central server, random number generation can be assumed “fair” because the server acts as an unbiased third party which generates numbers on behalf of the players, removing any possible player introduced bias from the generation task (Funfair Inc). There are numerous other benefits, such as allowing game developers to implement a server-side “anti-cheat” to make sure that players aren’t using so-called “hacked clients” to gain an unfair advantage in the game by sending malicious messages to the server which would grant the user previously unattainable stats in-game (*Don't Bear with the Cheaters*). Furthermore, having centralized servers allows game developers to roll out patches/updates to gameplay in one swift action, keeping everyone up to date on the latest “nerfs” or “buffs” fairly.

However, since the central server is responsible for the administration of the game, if it goes offline or becomes unavailable, the players connected to it cannot continue the game. This also means that game developers must invest a large amount of time, effort, and money into maintaining the servers and ensuring availability especially during times of burst activity (*The fall guys: why big multiplayer games almost always collapse at launch*). This can be either from sources of genuine activity spikes such as new game releases or revived game activity or from an attack from an adversary looking to take down the server for personal gain (*Battle.net Hit By DDoS Attack*).

A decentralized solution offers to solve these problems while ensuring the integrity of the game is maintained. In this way, instead of having players communicate everything with the central server, we can create a peer to peer network between each of the players participating in the game, creating a resilient network in which load is distributed between the players instead of all placed onto one server. The problem, in this case, is authority, since we need a way to be able to validate moves that are legal and distinguish them between malicious moves. This is where the blockchain comes in, offering a solution for decentralized consensus, with capabilities to host the game rules in a globally synchronized, immutable storage environment and act as an adjudicator to handle disputes that will inevitably arise through gameplay.

Motivation

The application of blockchains to production use cases has been a modern development, especially since the technology to allow complex, non-payment use has been pretty recent, there have only been a few games released that have been based around the blockchain and the benefits it can provide. Most of these games revolve around the use of ERC721 non-fungible tokens, meaning they represent unique items and are not interchangeable (*721*). This means that each of the game elements is tracked by these tokens, and transactions with these tokens between players make up the playable aspect of these games. However, there are many other types of games that can’t be represented by pure token transactions, and can also benefit from the distributed nature of the blockchain. Because of this, implementing a solid framework for complex games to demonstrate the possibilities of blockchain advancements outside of simple token exchanges is needed in order to show off all of the possibilities.

Existing Projects

As mentioned in the previous section, there are many existing blockchain games that use the ERC721 standard as mechanisms to power these trading games. The most popular of which is the CryptoKitties game, where ERC721 tokens represent virtual cats, which players can buy, sell, breed, and exchange (*CryptoKitties Technical Details*). There also exist card games similar to Hearthstone, with each card being a unique token that can be exchanged. The common scheme in all these existing projects is that they are based around ERC721 tokens that are exchangeable, similar to physically tradable cards such as Bakugan, Yu-Gi-Oh, or Pokemon, except virtually generated (*Gods Unchained Wiki*). To date, there hasn't been a wide release of a game that actually takes a more complex game such as Monopoly and converts it into a decentralized and playable version, inspiring this proof of concept.

Research

Blockchain

In order to replicate the security from a centralized server in a game intended for widespread use, there has to be a method to reach consensus between each of the game players as a surrogate for a central server. In order to do that, the rules of the game must be encoded in the blockchain so that applications can reference that, and use that to figure out which actions are valid and which ones are malicious. This brings up the first problem which needed to be solved, a way to represent individual games along with the rules governing that game within the blockchain.

The search for a suitable framework for implementing this game framework started from the basics, the underlying blockchain technology to use. There were two main choices, in this case, Bitcoin or Ethereum. Ethereum offered a more robust set of tools with its implementation of smart contracts and much faster block confirmation time, as well as being a more popular framework to experiment on with vastly more community resources than Bitcoin (*Ethereum Whitepaper*).

Scaling

The next problem to be tackled was a series of fundamental issues, preventing most games from being playable directly on the blockchain. The big issue with blockchain games is that the security of the game relies on confirmation from every other node in the network in order to progress the game. This presents multiple issues, since confirmation takes time, on the Bitcoin network one confirmation (usually more are needed to be completely safe) takes around 10 minutes (*Bitcoin: A Peer-to-Peer Electronic Cash System*), which would be completely unacceptable for any real-time game. Ethereum's confirmation time of 10 seconds is much better in comparison but still wouldn't work in a real-world setting where many people expect instant responses to a move, and where multiple moves could be made within the block confirmation time. The issue would be compounded as well if many decentralized games were released, as transaction volume would inevitably rise with the increased demand.

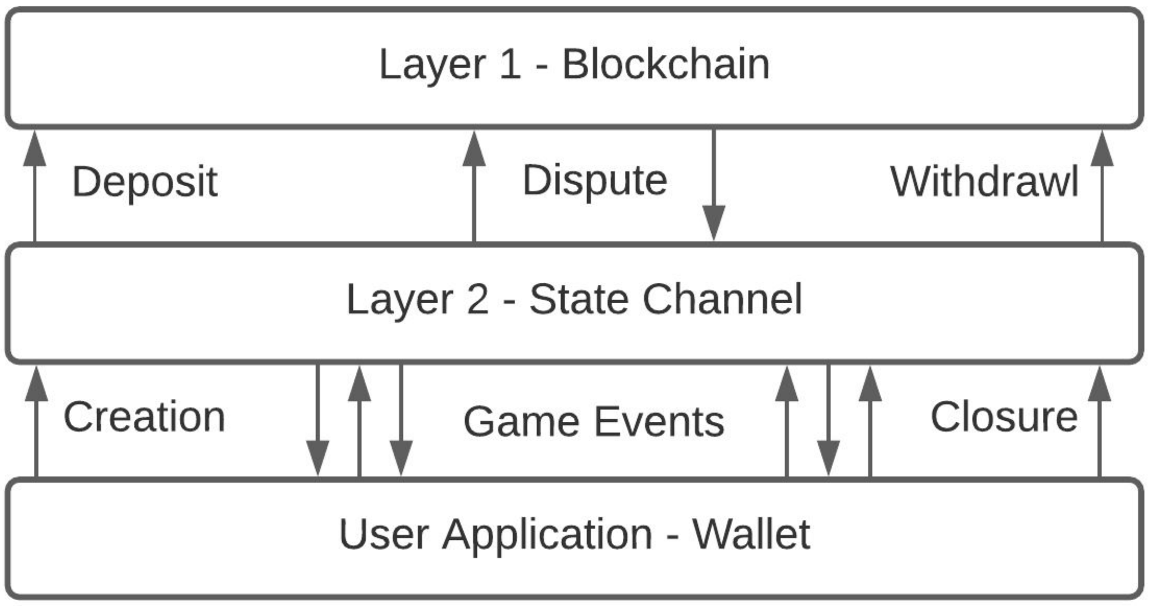
A secondary issue is the fees associated with blockchain transactions. Since each write operation to the blockchain (and smart contract invocation) costs fees to perform, this presents a potential problem since players would have to pay fees, while insignificant at first, add up over the course of multiple turns and across multiple hours of gameplay. This would also unfairly mean that players who may have already purchased a game would have to additionally pay to play. In addition, the fees are also tied to transaction volume, since higher fee transactions are prioritized in the blockchain network since miners are incentivized to include them. The same problem then occurs, with transaction volume rising fees would also rise.

The idea then is to use a Layer 2 scaling solution, as illustrated below. For Ethereum, there were two main viable candidates that are currently being researched and developed. The first to be evaluated was Plasma (*Plasma: Scalable Autonomous Smart Contracts*), a payment scaling solution for Ethereum that has the ability to create child blockchains based on the main chain. The main downside, however, is the arbitration window for withdrawals, meaning that any movement from funds on the child chain back to the main chain must wait for that window to pass. Typically, this is set between 7-14 days, which would be much too long for a game like this.

The other solution is much more flexible in regards to withdrawal time and is more intuitive to ration with: State Channels (*Counterfactual: Generalized State Channels*). State channels are a solution based akin to Bitcoin's Lightning Network (*The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments*), where a payment channel would be created, funds deposited, and after that any transactions in that channel could be done off-chain, only returning to the main blockchain once the channel was to be closed. The main operating principle about state channels is they emulate blockchain transactions, meaning that in case of any disputes, users can go straight back to the main blockchain to settle. State channels also offer a very important feature referred to as ‘instant finality’, where transactions are signed and take into effect immediately, meaning that game moves are approved and ‘valid’ immediately upon receipt. This property, combined with the flexible withdrawal times and simplicity of implementation, made it the choice of framework for this implementation.

State Channels

Since state channels emulate on-network transactions, they don’t need to communicate with the original blockchain layer (thus being called a Layer 2 solution), outside of the opening and closing of the channel itself. Any individual transactions between the opening and closing transactions don’t need to be reported, since the individual players communicate through the state channel. If however, a dispute is lodged which requires arbitration, then the participants in that state channel can go back to the main chain in order to resolve it. This way, in the vast majority of cases, the number of blockchain transactions (and thus fees we would pay, with the required confirmation time) can be drastically reduced, making blockchain games possible. The security of state channels is also important, and in order to create the game, both players must deposit funds into a contract where they are then locked, and only released once the game is over and the outcome is known (and proved). A diagram showing a sample state channel execution flow is included below.



The connection to the Ethereum main network would be hosted as an ‘anchor’ or ‘adjudicator’ contract, where funds could be deposited and withdrawn, and where the game rules would be encoded and stored. Players would read from this contract (this doesn’t incur any fees or wait time) and then create a state channel between themselves. By basing their initial state off what they read from the contract and advancing the state based on signed transactions from the other players, each player would independently come to a consensus about the game state. In case of disputes, if unresolved between the players themselves, they would instead go to the adjudicator contract, which would decide the outcome of the game based on the signed states provided by each of the parties involved in the dispute.

Force Move

Since Monopoly is a turn-based game, where each player proceeds in a set order one after the other, that makes it possible to implement a consensus paradigm called Force Move (*ForceMove: an n-party state channel protocol*). The need for this can be demonstrated as an example with two players, Alice and Bob. If Alice is one move away from winning the game and Bob decides to no longer cooperate with the rules of the game, then without Bob's final decision the game cannot officially be played out. In this case, if Bob suddenly disconnects, Alice cannot proceed, and thus the game is in a stalemate when it would have been Alice's win. If we implement force-move, we can then 'force' Bob to proceed with the game, or else forfeit the game. In this case, Alice can then proceed to create a transaction on the main chain, and provide her set of signed states from the previous moves and, after a set amount of time, Bob must produce a signed state which proceeds from that state, or forfeit his deposit. This keeps the game state flowing, and incentivizes each player to keep playing, even in an unfavorable position.

Randomness

Monopoly also poses a challenge to be implemented on the blockchain because it requires randomness. For example, each turn requires the roll of a fair die, and landing on either the Community Chest or Chance cards requires a random card to be drawn. Producing randomness especially in the blockchain is a highly debated process, and there exists no native way to generate randomness on the EVM. Since each smart contract must be executed on all Ethereum nodes, each node must be able to independently replicate the results of the smart contract, which makes the generation of random numbers usually a task relegated to Oracle services (*Verifiable Randomness for Blockchain Smart Contracts*) or through a different scheme. Since the core architecture is based around state channels, most of the Oracle services and other schemes that rely on block numbers or data derived from recently mined blocks won't work for this, since the state channel is not bound to the main network.

Commit Reveal

Commit-Reveal (*Commitment scheme*) is a tried and true solution to problems like this both on and off the blockchain, where players generate some random value on their side, produce a hash of that value, submit it to everyone else, and finally, everyone reveals all at once. Everyone's value is then combined together to create the fair ‘random’ number that is then used. Because each player must submit a hash of their random value, they cannot change their value and they do not know the value that anyone else has generated until the plaintext is revealed. This makes it extremely difficult for one party to unfairly influence the resulting random value but makes it trivial to validate that everyone is participating fairly.

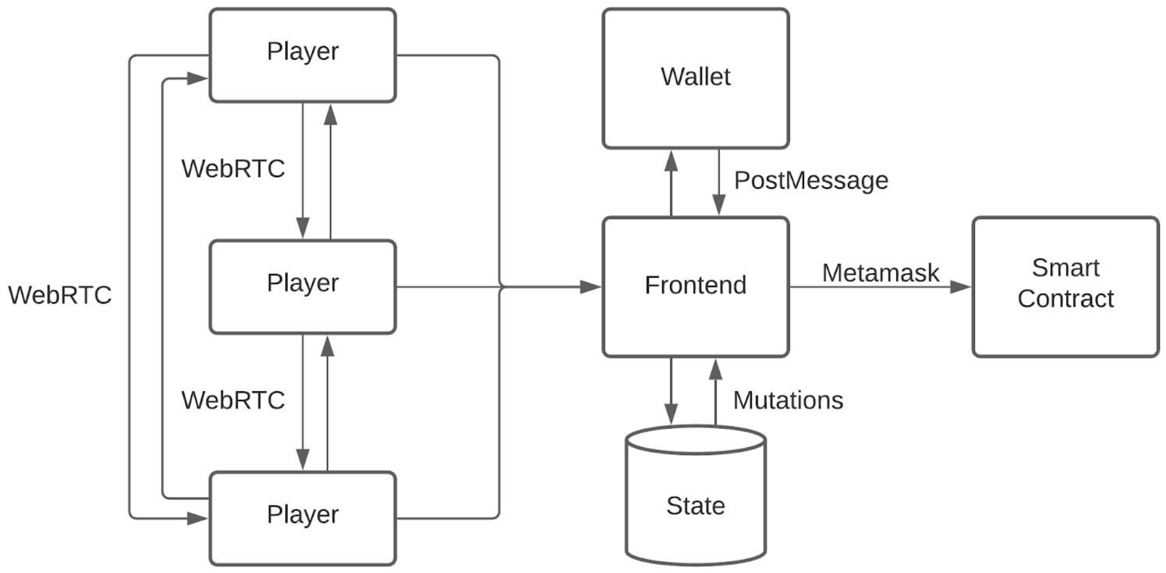
Fate Channels

The other option, the method used in FunFair’s Fate Channels (*Blockchain Solutions For Gaming Commercial Whitepaper v2.0 Draft*) implementation, based on repeated hashing of an initial seed, which is what our implemented approach used. The simplicity of this approach is that the initial value can be communicated between all players, then the game proceeds, and a nonce is updated and each turn receives an updated source of randomness automatically, reducing the surface area for griefing by other players.

Implementation

High-Level Overview

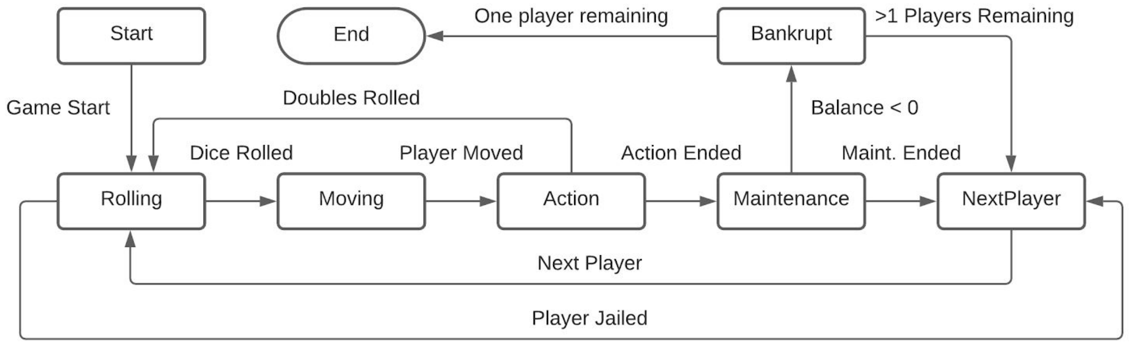
Cryptopoly is currently primarily powered by a WebRTC (*WebRTC*) data connection which enables peer-to-peer communication between each player and is established when joining a “room”. This backbone facilitates all state communication between players. Locally, a connection to the Ethereum network is set up with a widely used browser extension called Metamask (*MetaMask*), which allows a ‘light’ Ethereum node to be run in the browser and powers the connection to the adjudicator contract. Metamask provides an API interface that is then consumed by a dapp (distributed application) browser wallet implementation that supports state channels. The dapp wallet is state channel compatible and allows rapid signing of moves by creating a second pair of public/private keys to be used within the state channel application, eliminating the Metamask popup that would otherwise occur on every state update, allowing for much smoother user experiences.



State updates within the channel are first sent to the browser wallet, approved locally, sent out through WebRTC, and then verified against the adjudicator contract on the main Ethereum network once received. Note that this operation has no gas cost since we utilize “pure” functions that run locally on the node and do not require any updates to the blockchain. The wallet also updates the Vuex store whenever there is a new state update received from the state channel. The frontend reacts to the Vuex store and shows game state updates, which the players then see as changes to the rendered game board.

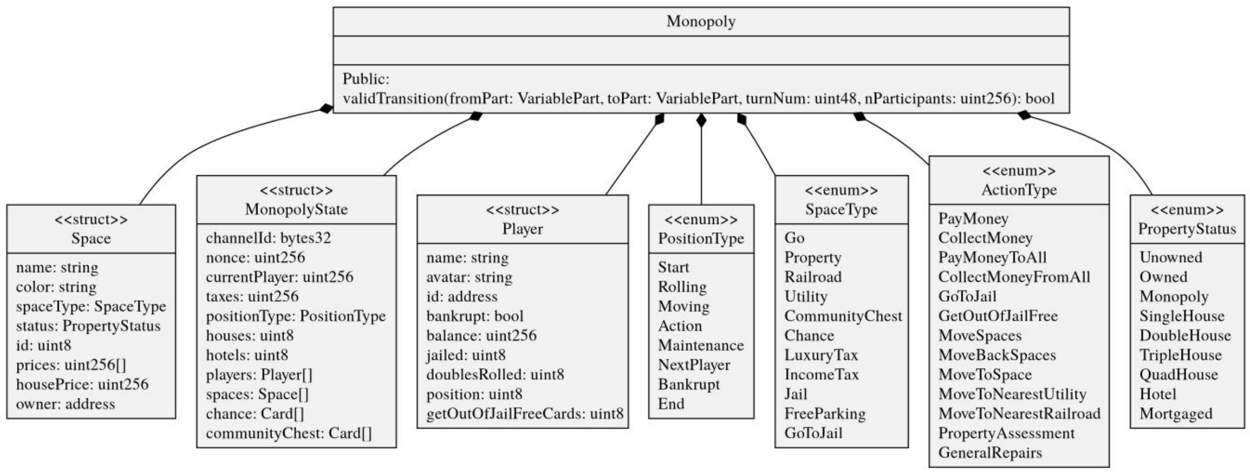
State Machine

Implementation of the application which would be supported by the state channel framework started with creating a state machine diagram, which represents the core of the application and shows how an application state progresses over time. For Monopoly, the game which was chosen, the associated state diagram is included below. Note that this is intended to be as general as possible, to allow for modifications to be made to the implementation, such as custom house rule implementations. One thing to note is that the state channel itself only gets updated in the NextPlayer state due to inherent limitations in the ForceMove protocol which forces updates to be made in a cyclical fashion by players one after another. This means that validation of the state by the official adjudicator contract is only done at the end of a players' turn, and not during. This compromise is done by the ForceMove protocol to ensure that certain properties are held while the state channel is running, meaning the updates shown in the frontend are provisional until they are verified by the contract and updated in the state channel. This does not compromise the integrity of the game however, since each update sent out is still validated locally on every other players' side and is still subject to the constraints laid out in the rules.



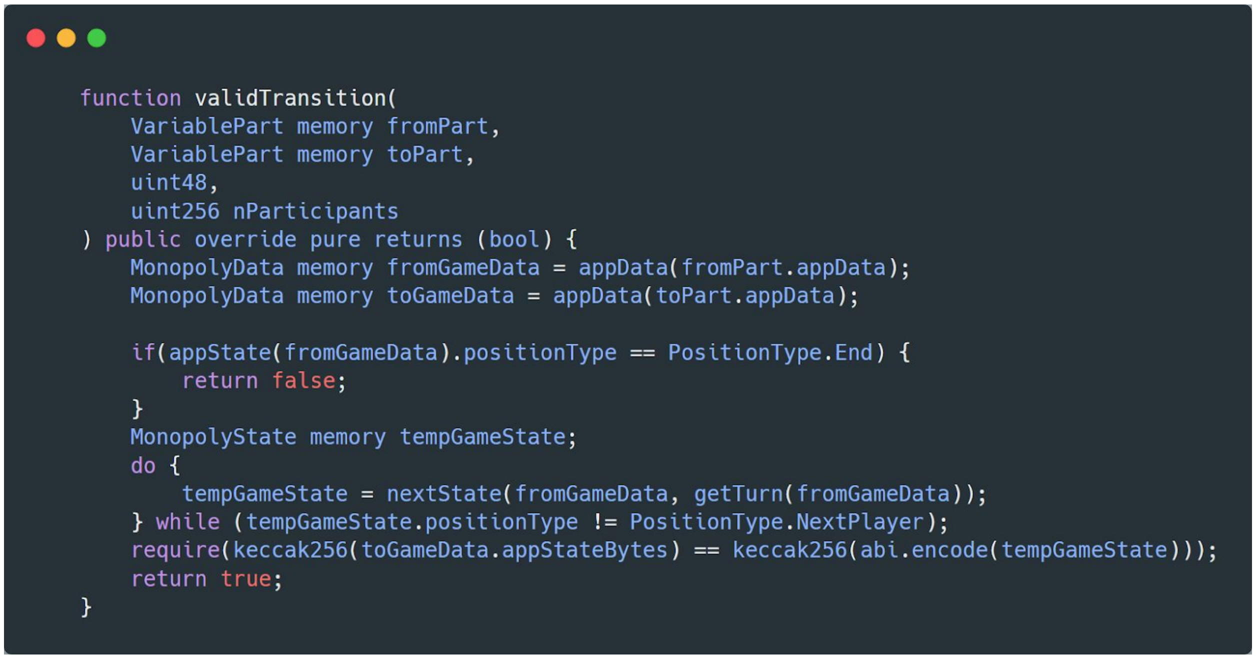
Data Structures

The game state itself is represented by a series of data structures, defined in the smart contract and replicated client-side on the frontend for consistency. Below is a UML class diagram representing the most important subset of structures, which hold the state of the game and are modified through the course of the game by certain player actions. There is a main MonopolyState structure that represents the game state, containing information such as the current position, information about the players, and game-specific data such as what properties there are, what cards can be chosen, and the seed for the random number generation algorithm. Player actions such as purchasing houses, rolling die, paying rent, etc. modify this state directly on each player's client-side cache of the state and are verified against the state contract for correctness at the end of each players' turn. The actions are broadcast by each player during their respective turn to every other connected, so each action appears synchronously on each players' screen with the same results.



Ethereum Smart Contract

Once this state diagram is created and the data structures decided upon, the next step is to create the Ethereum smart contract that will house all the required functionality for the creation and conclusion of state channels, handlers for disputes that may arise through gameplay, and transition validators which help understand current game state and assist in arbitration of disputes. Luckily in this case there are already well-researched implementations of the prior, however, the latter case with the state transition validators must be custom-tuned for each application and cannot easily be reused. In this implementation, we use a validation method based on a paradigm called applyChange. The transitions are validated by emulating the game state progression (applying the change) and computing a cryptographic hash of that final state. By then comparing that hash to the hash of the proposed next state, the contract can determine if a state transition is as expected. Compared to being validated by hard-coding rules within the contract, applyChange allows for a simpler process of writing game progression code, and in this case, works much better since Monopoly is such a complex game. Directly emulating game state evolution is a much more direct way to validate allowed moves, rather than having to think of every edge case to deny. These transition validators are written as pure functions, which in Solidity means that they can be called on a node and computed without a blockchain transaction, allowing integrations to progress game state via a service such as Infura instead of writing validation code client-side as well, although this was not done in the Cryptopoly implementation to reduce reliance on external web services.

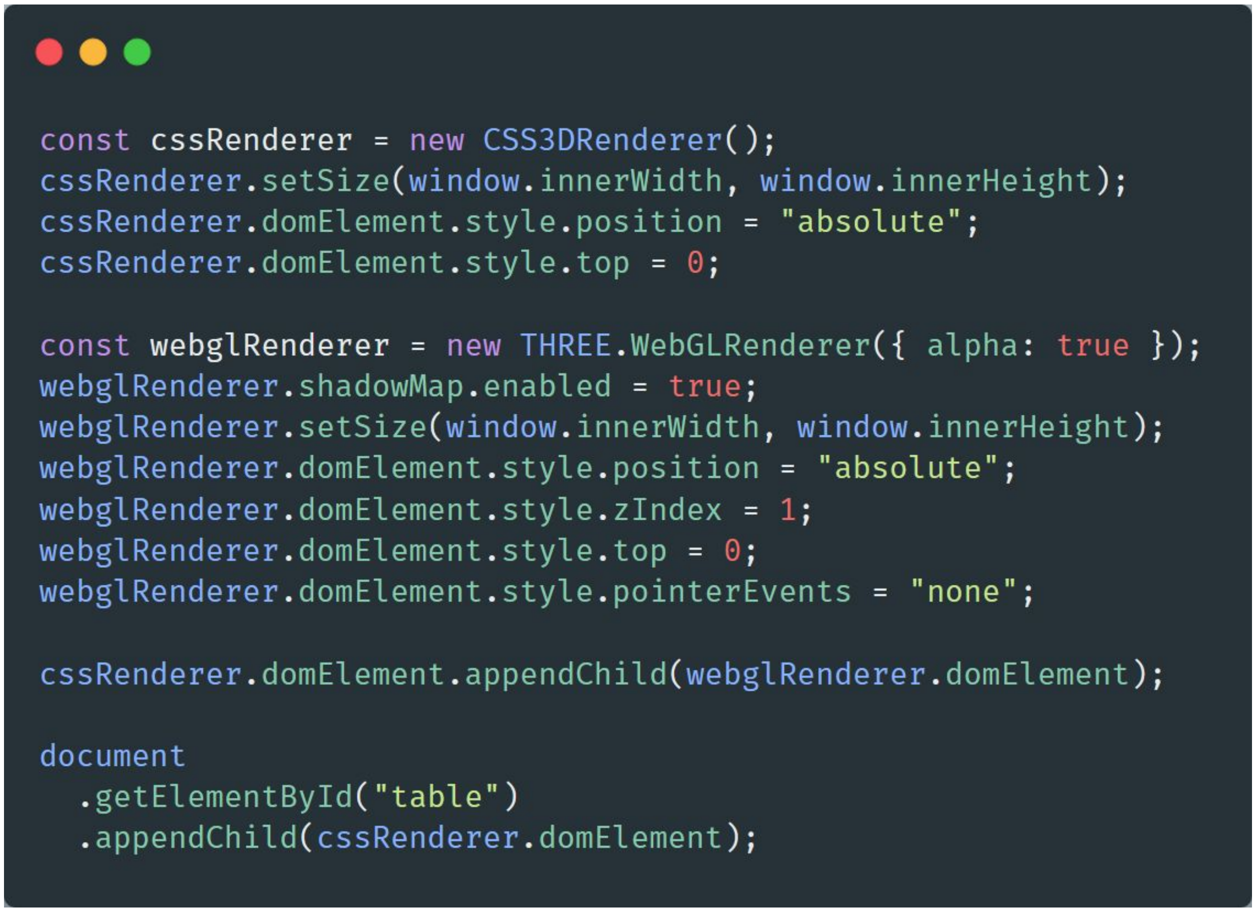


```solidity
function validTransition(
    VariablePart memory fromPart,
    VariablePart memory toPart,
    uint48,
    uint256 nParticipants
) public override pure returns (bool) {
    MonopolyData memory fromGameData = appData(fromPart.appData);
    MonopolyData memory toGameData = appData(toPart.appData);

    if(appState(fromGameData).positionType == PositionType.End) {
        return false;
    }
    MonopolyState memory tempGameState;
    do {
        tempGameState = nextState(fromGameData, getTurn(fromGameData));
    } while (tempGameState.positionType != PositionType.NextPlayer);
    require(keccak256(toGameData.appStateBytes) == keccak256(abi.encode(tempGameState)));
    return true;
}
```

Front End

3D rendering of the game board and the accompanying avatars is done with ThreeJS, a Javascript framework. A combination of both WebGL rendering and CSS3D rendering engines are used in order to blend together the CSS board elements with the WebGL avatars. These layers are blended by setting the following settings, which places the CSSRenderer underneath the WebGL renderer, sets any empty space in the WebGLRenderer to transparent, then passes any pointer-events to the CSS layer underneath, creating a seamlessly blended transition.



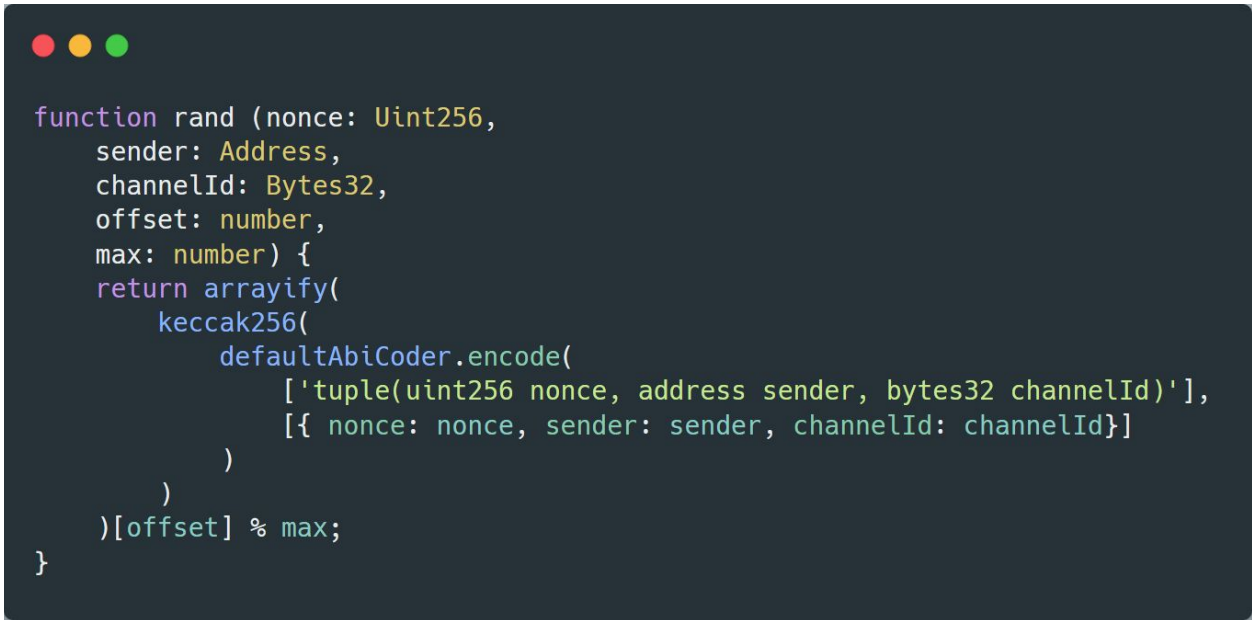
```
const cssRenderer = new CSS3DRenderer();
cssRenderer.setSize(window.innerWidth, window.innerHeight);
cssRenderer.domElement.style.position = "absolute";
cssRenderer.domElement.style.top = 0;

const webglRenderer = new THREE.WebGLRenderer({ alpha: true });
webglRenderer.shadowMap.enabled = true;
webglRenderer.setSize(window.innerWidth, window.innerHeight);
webglRenderer.domElement.style.position = "absolute";
webglRenderer.domElement.style.zIndex = 1;
webglRenderer.domElement.style.top = 0;
webglRenderer.domElement.style.pointerEvents = "none";

cssRenderer.domElement.appendChild(webglRenderer.domElement);

document
  .getElementById("table")
  .appendChild(cssRenderer.domElement);
```

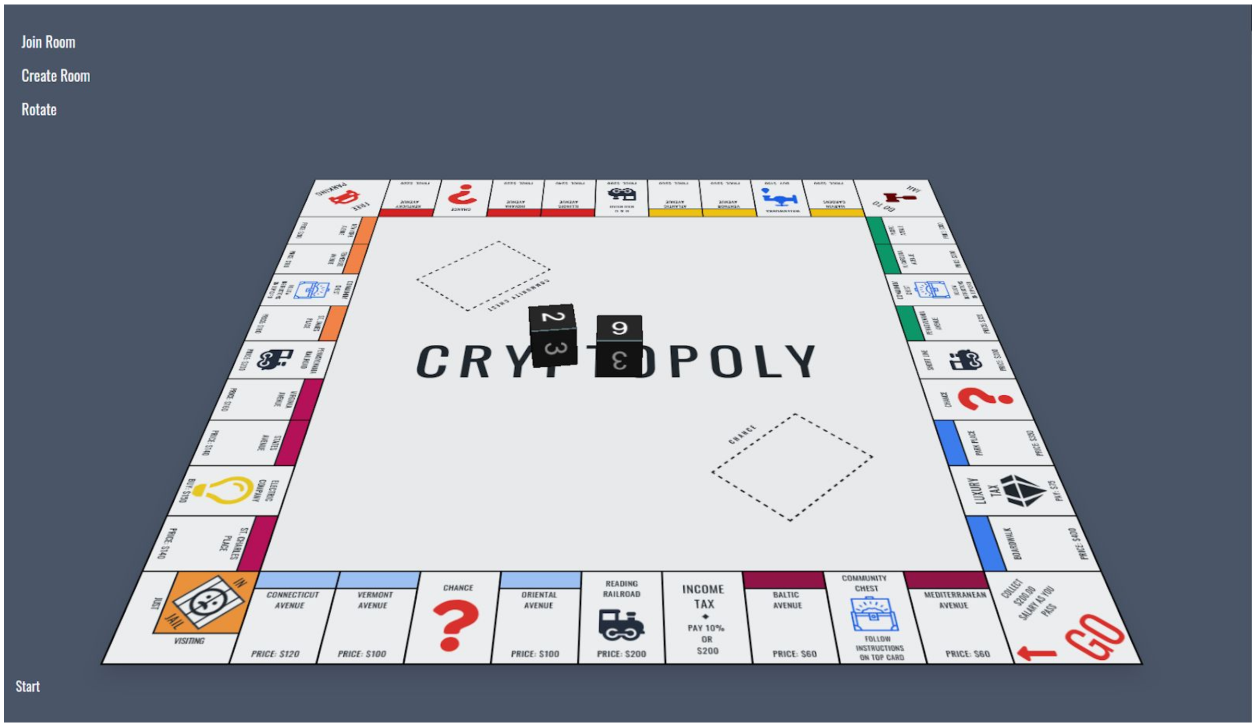
The physics of the rendered board is handled by CannonJS, whose main function is to serve up pre-baked dice rolls. The engine runs through a simulation first, then replaces the top facing side with a pre-set value calculated from our simplified Fate Channel code, shown below. This makes sure that each player in the game sees the same dice roll while preserving the random feel.



```
function rand (nonce: Uint256,
    sender: Address,
    channelId: Bytes32,
    offset: number,
    max: number) {
    return arrayify(
        keccak256(
            defaultAbiCoder.encode(
                ['tuple(uint256 nonce, address sender, bytes32 channelId)'],
                [{ nonce: nonce, sender: sender, channelId: channelId}]
            )
        )
    )[offset] % max;
}
```

The communications layer is handled using PeerJS, which is a WebRTC framework that allows players to communicate with each other and to ICE/TURN signal/relay servers. The communications layer is also an Event Emitter, which allows the application to listen in on certain asynchronous events and respond accordingly, i.e. rendering an avatar for a new player that joined, or updating the local state to be in sync with the state channel state.

The state is controlled through Vuex, a state management plugin for VueJS projects, which has a custom sync driver attached to both it and the communications layer which broadcasts actions performed by the current player to everyone else. This all comes together to create this proof of concept: Cryptopoly.



Conclusion

The viability of blockchain games is definitely present as shown in the Cryptopoly implementation, and while research is still ongoing into state channels the fact that this application works just shows the potential of the future. The cost-benefit of this system setup is immense, as it allows game studios to bring the ongoing costs of server hosting and maintenance to almost zero, making it now possible to shift business and development focus away from Operations toward Development.

While Monopoly is a good demonstration of a complex game that can be implemented on the blockchain, there are many more complex games that as of yet are incompatible with this framework. Monopoly works in this case because of its turn-based nature and its predetermined player order. In the case of more complex games that require dynamic players or even real-time games such as first-person shooters and real-time strategy games which don't have the constraint of having individual player actions contained within a single turn, this system might not work. In fact, this still remains a problem for games today with centralized servers as players connect in from many different locations, which results in a disparity between one player's perceived game state and another. This lag makes it very difficult to come to a consensus on who was correct, and many games solve this issue in different ways. Most first-person shooter games give priority to the player who claims an action, resulting in a bias called peekers advantage. The way forward, in this case, would be to devise a more general framework for these types of games which allows for decentralized consensus even with differing network lag.

References

"721." ERC. Web.

Buterin, Vitalik, and Joseph Poon. "Plasma: Scalable Autonomous Smart Contracts." Plasma: Scalable Autonomous Smart Contracts. 11 Aug. 2017. Web.

Buterin, Vitalik. "Ethereum Whitepaper." Ethereum.org. 2013. Web.

Chainlink. "Verifiable Randomness for Blockchain Smart Contracts." Chainlink. Chainlink, 16 June 2020. Web.

Coleman, Jeff, Liam Horne, and Li Xuanji. "Counterfactual: Generalized State Channels." 12 June 2018. Web.

"Commitment Scheme." Wikipedia. Wikimedia Foundation, 23 Aug. 2020. Web.

CryptoKitties. "CryptoKitties Technical Details." CryptoKitties. Web.

"Don't Bear with the Cheaters." Easy Anti-Cheat. Web.

"Ethereum API: IPFS API Gateway: ETH Nodes as a Service." Infura. Web.

"The Fall Guys: Why Big Multiplayer Games Almost Always Collapse at Launch." The Guardian. Guardian News and Media, 27 Aug. 2020. Web.

Funfair Inc. "Blockchain Solutions For Gaming Commercial Whitepaper V2.0 Draft." Blockchain Solutions For Gaming Commercial Whitepaper V2.0 Draft. Sept. 2018. Web.

Gods Unchained Wiki. "Gods Unchained Wiki." Gods Unchained Wiki. Gamepedia, 21 Feb. 2019. Web. 04 Sept. 2020.

Kain, Erik. "Battle.net Hit By DDoS Attack - 'Call Of Duty' 'Overwatch' And 'World Of Warcraft' Experiencing Issues." Forbes. Forbes Magazine, 03 June 2020. Web.

"MetaMask." MetaMask. Web.

Nakamoto, Satoshi. "Bitcoin: A Peer-to-Peer Electronic Cash System." Bitcoin: A Peer-to-Peer Electronic Cash System. 31 Oct. 2008. Web.

Poon, Joseph, and Thaddeus Dryja. "The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments." The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments. Jan. 2016. Web.

Reiff, Nathan. "Blockchain Explained." Investopedia. Investopedia, 31 Aug. 2020. Web.

Stewart, Andrew, and Tom Close. "ForceMove: Ann-party State Channel Protocol." 20 Nov. 2018. Web. 03 Sept. 2020.

"WebRTC." WebRTC. Web.