Game: Asteria

As part of the discovery process, it was decided that the use case for the partnerchain reference implementation would be a game. We decided on implementing Asteria which is a simple game about that showcases the capabilities of the eUTxO model. The game is simple: each player has its own ship that they can move according to a maximum speed to reach Asteria and claim a reward.

In this document we'll go over the mechanics of the game, to better understand the requirements, then we'll lay out the design decisions and technical aspects of the implementation, and finally, we'll detail the usage of the game via node commands.

Brief introduction to game mechanics

The game happens within a 2D grid through which ships can travel by moving a certain distance that does not surpass the maximum speed. The Asteria is at the center of the grid, at the (0,0) coordinates. Dispersed through the grid at fixed coordinates are pellets, which are freely available sources of fuel that ships can consume when they are sitting on those same coordinates. The distance between the Asteria and the ships, or the ship's before and after moving, is measured using Manhattan distance. The game is ruled over by on chain validators that are parameterised. Because of this, we can have various games with different settings. The most relevant parameters for our case are the following:

- Min_asteria_distance: minimum distance between the Asteria and a new ship on its creation.
- Max_ship_fuel: maximum amount of fuel that a ship can have.
- Max_asteria_mining: maximum percentage of the reward pool that a single ship can withdraw upon arriving at Asteria
- Max_speed: maximum distance a ship can travel in a certain time.
- Fuel_per_step: how much fuel is consumed when moving the ship 1 step.

And we have the following configuration:

- Min_asteria_distance: 10
- Max_ship_fuel: 30
- Max_asteria_mining: 50
- Max_speed: 1 step per 30 seconds.
- Fuel_per_step: 1 fuel per step.

Game Operations

Let's see the details of the operations that our node supports and their constraints:

- Create ship: Places a new ship on the map, according to the coordinates provided by the user. This initial placement needs to abide by the previously mentioned min_asteria_distance.
- Move ship: Moves a ship to the coordinates provided by the user if: the movement complies with the max_speed, and the ship has enough fuel to travel

- Gather fuel: Collects fuel from a pellet if the ship's coordinates overlap the pellet's.

- Mine Asteria: Collects a portion of the reward if the ship's coordinates overlap Asteria's and the portion to be redeemed satisfies max_asteria_mining.

There are more operations that we don't support, and there are other validations done upon these operations we mentioned, but for the sake of simplicity we don't go over every single one of the validations. For a more thorough explanation, you can check Asteria's official repository.

Technical details

Asteria is implemented utilizing the capabilities of the eUTxO model. It relies on three validators to check every operation of the game. It uses redeemers to specify game operations, and uses datums to keep the game's state. We won't discuss the overall design of the application in the eUTxO model as there is already a design document that excellently describes it. From now on we'll describe the technical aspects of implementing the game in our Substrate node.

Offchain implementation

As the first step, we decided on an offchain transaction building approach. This is the functionality that allows us to interact with the chain, and it is similar to that of regular offchains we see on Cardano. This allowed us to integrate the game easily into the node with minimal modifications to the node itself. The complexity arises as the system does not have a lot of usual features like a balancer, e.g., every output has to be constructed individually taking the coin expenses in consideration. Fortunately, there are no fees so that reduces the calculations to perform.

Node integration

The key step is to integrate the game into the node via the definition of a command line interface that allows the users to play the game and the modification of the genesis of the chain.

Genesis

An important decision made was to assume a fixed instance of the validators. This allows us to initialize the chain with the game board in the genesis. In the file that holds the initial UTxO set of the chain, we included the UTxOs for the Asteria and some pellets. This means that as soon as the chain starts, the user can enter the game by creating their ship.

Node commands

The main purpose of these functions is to simplify the usage for the user, as they could interact with the game using the tx-builder as well. The game commands are innate to the node, as we want the game to be the main functionality of the chain.

Game usage

Each of the following actions must be run with a running instance of the node.

Deploy Scripts

This command reads all the script parameters from game/src/deploy_params.json and applies them to the generic (parameterized) scripts, writing the resulting ones in their respective files, inside the scripts directory.

```
./target/release/griffin-partner-chains-node game deploy-scripts
        --params <SCRIPTS_PARAMS_PATH>
```

Arguments details:

params: path to the json file containing all the script parameters.

Create Ship

This command creates the player's Ship. The transaction also mints the initial ship's fuel, the ship and pilot tokens, and pays an inscription fee that is added to the total prize in the Asteria UTxO. The pilot token goes back to the wallet input owner, and serves as a proof of the ownership of the Ship.

```
./target/release/griffin-partner-chains-node game create-ship
        --input <WALLET_OUTPUT_REF>
        --witness <PUBLIC_KEY>
        --pos-x <POS_X>
        --pos-y <POS_Y>
        --ttl <TIME_TO_LIVE>
```

Arguments details:

- input: a wallet input that must be consumed to pay for the minimal amount of coin in the Ship output and the fee added to the Asteria accumulated rewards.
- witness: public key of the input owner. If omitted, Shawn's pub key is the default value, since this makes it easier to test transactions in a dev environment.
- pos-x: initial "x" coordinate of the Ship output.
- pos-y: initial "y" coordinate of the Ship output.
- ttl: the transaction's time-to-live. The resulting POSIX time of the validity interval is used to set the initial last-move-latest-time field in the Ship output datum.

Gather Fuel

This command moves fuel tokens from a pellet UTxO to a ship UTxO, only if they have the same position in the grid, as specified in the datums. The amount of fuel to gather is specified in the redeemer, and the total ship fuel must not exceed its maximum capacity.

```
./target/release/griffin-partner-chains-node game gather-fuel
        --ship <SHIP_OUTPUT_REF>
        --pellet <PELLET_OUTPUT_REF>
        --witness <PUBLIC_KEY>
        --fuel <FUEL_AMOUNT>
        --validity-interval-start <VALIDITY_INTERVAL_START>
```

Arguments details:

- ship: reference to the ship UTxO.
- pellet: reference to the pellet UTxO.
- witness: public key of the pilot token owner. This is necessary since the pilot UTxO must be provided as input to prove the ship ownership. If omitted, Shawn's pub key is the default value.
- fuel: the amount of fuel to transfer from the pellet to the ship.
- validity-interval-start: start of the transaction's validity interval. The corresponding POSIX must be greater than the last-move-latest-time field in the ship datum, in order to respect the speed limit of the last move.

Move Ship

This command moves the ship to a different point in the grid (updates de pos_x and pos_y fields in the ship datum). The transaction also burns the fuel tokens consumed.

```
./target/release/griffin-partner-chains-node game move-ship
        --ship <SHIP_OUTPUT_REF>
        --witness <PUBLIC_KEY>
        --pos-x <POS_X>
        --pos-y <POS_Y>
        --validity-interval-start <VALIDITY_INTERVAL_START>
        --ttl <TIME_TO_LIVE>
```

Arguments details:

- ship: reference to the ship UTxO. witness: public key of the pilot token owner. This is necessary since the pilot UTxO must be provided as input to prove the ship ownership. If omitted, Shawn's pub key is the default value.
- pos-x: new "x" coordinate of the ship.
- pos-y: new "y" coordinate of the ship.
- validity-interval-start: start of the transaction's validity interval. The corresponding POSIX must be greater than the last-move-latest-time field in the ship datum, in order to respect the speed limit of the last move.
- ttl: the transaction's time-to-live. The resulting POSIX time of the validity interval is used to set the initial last-move-latest-time field in the Ship output datum. The manhattan distance travelled divided by the POSIX validity range must be less or equal to the max speed.

Mine Asteria

This command can be triggered when the ship reaches Asteria, i.e., its coordinates are both zero. Then the ship owner can receive a percentage of the total prize given by (MAX_ASTERIA_MINING/100). This transaction also burns the ship and all remaining fuel tokens.

```
./target/release/griffin-partner-chains-node game mine-asteria
        --ship <SHIP_OUTPUT_REF>
        --witness <PUBLIC_KEY>
        --validity-interval-start <VALIDITY_INTERVAL_START>
```

Arguments details:

- ship: reference to the ship UTxO.
- witness: public key of the pilot token owner. This is necessary since the pilot UTxO must be provided as input to prove the ship ownership. If omitted, Shawn's pub key is the default value.
- validity-interval-start: start of the transaction's validity interval. The corresponding POSIX must be greater than the last-move-latest-time field in the ship datum, in order to respect the speed limit of the last move.

Dev activity log: Use case specific features

This is the deatiled log of the development activities required for the implementation of the use case specific features. The use case to develop was a game and we decided to implement Asteria. This is a game with 4 main operations that utilize the eUtxO model, so the task was to implement each of them. A more in-depth explanation of the game's operations and design can be found in the onchain documentation. The game commands' usage is thorougly explained in the game's README.

This log is divided in three parts:

- game integration: step-by-step modifications that were made to implement the game features,
- game commands in the node: add the operations as commands to the node,
- wallet refactor: particular to our node, we refactored the wallet to make it more modular.

Add Game Crate

1. Add the aiken (on-chain) code and its documentation in game/onchain.
2. Define the game crate and add it to the Cargo.toml as a member of the workspace:

partnerchain-reference-implementation/game/Cargo.toml
Lines 1 to 23 in b96196d

```
1   [package]
2   name = "game"
3   description = "Game implementation"
4   version = "0.1.0"
5   repository.workspace = true
6   edition.workspace = true
7
8   [dependencies]
9   anyhow = { workspace = true }
10  clap = { features = ["derive"], workspace = true }
11  griffin-core = { workspace = true }
12  griffin-wallet = { workspace = true }
```

partnerchain-reference-implementation/Cargo.toml
Lines 11 to 19 in b96196d

```
11  members = [
12      "demo/authorities",
13      "game",
14      "griffin-core",
15      "griffin-rpc",
16      "node",
17      "runtime",
18      "wallet",
19      "game",
```

partnerchain-reference-implementation/Cargo.toml
Line 82 in b96196d

```
82  game = { default-features = false, path = "game" }
```

3.Define new GameCommand struct and implement its run method:

We define an overarching GameCommand enum that will hold the actual subcommands enum, this structure allows us to integrate the command into the node.

partnerchain-reference-implementation/game/src/lib.rs
Lines 9 to 13 in 1b98b2b

```rust
#[derive(Clone, Debug, Subcommand)]
pub enum GameCommand {
    #[command(subcommand)]
    Game(Command),
}
```

The run method dictates the parsing of the commands:

partnerchain-reference-implementation/game/src/lib.rs
Lines 29 to 68 in 1b98b2b

```rust
impl GameCommand {
    pub async fn run(&self) -> sc_cli::Result<()> {
        let Context {
            cli,
            client,
            db,
            keystore,
            slot_config,
            ..
        } = Context::<GameCommand>::load_context().await.unwrap();
        match cli.command {
            Some(GameCommand::Game(cmd)) => match cmd {
```

4.Define each game command:

partnerchain-reference-implementation/game/src/lib.rs
Lines 15 to 27 in 1b98b2b

```rust
#[derive(Clone, Debug, Subcommand)]
pub enum Command {
    /// Create a ship to enter the game
    CreateShip(CreateShipArgs),
    /// Gather fuel using a ship and a fuel pellet
    GatherFuel(GatherFuelArgs),
    /// Move a ship to a new position
    MoveShip(MoveShipArgs),
    /// Mine Asteria using a ship
    MineAsteria(MineAsteriaArgs),
    /// Apply parameters and write game scripts
    DeployScripts,
```

5.Define each game commands' arguments:

i.Arguments for CreateShip:

partnerchain-reference-implementation/game/src/lib.rs
Lines 70 to 107 in 1b98b2b

```rust
#[derive(Debug, Args, Clone)]
pub struct CreateShipArgs {
    /// An input to be consumed by this transaction. This argument may be specified multiple times.
    #[arg(long, short, verbatim_doc_comment, value_parser = utils::input_from_string, required = true, value_name = "WALLET_
    pub input: Input,

    /// 32-byte H256 public key of an input owner.
    /// Their pk/sk pair must be registered in the wallet's keystore.
    #[arg(long, verbatim_doc_comment, value_parser = utils::h256_from_string, default_value = keystore::SHAWN_PUB_KEY, value
    pub witness: H256,

    #[arg(
```

ii.Arguments for MoveShip:

partnerchain-reference-implementation/game/src/lib.rs
Lines 109 to 139 in 1b98b2b

```rust
#[derive(Debug, Args, Clone)]
pub struct GatherFuelArgs {
    #[arg(long, short, verbatim_doc_comment, value_parser = utils::input_from_string, required = true, value_name = "SHIP_OU
    pub ship: Input,

    #[arg(long, short, verbatim_doc_comment, value_parser = utils::input_from_string, required = true, value_name = "PELLET_
    pub pellet: Input,

    /// 32-byte H256 public key of an input owner.
    /// Their pk/sk pair must be registered in the wallet's keystore.
    #[arg(long, short, verbatim_doc_comment, value_parser = utils::h256_from_string, default_value = keystore::SHAWN_PUB_KEY
    pub witness: H256,
```

iii.Arguments for GatherFuel:

partnerchain-reference-implementation/game/src/lib.rs
Lines 141 to 188 in 1b98b2b

```rust
#[derive(Debug, Args, Clone)]
pub struct MoveShipArgs {
    #[arg(long, short, verbatim_doc_comment, value_parser = utils::input_from_string, required = true, value_name = "SHIP_OU
    pub ship: Input,

    /// 32-byte H256 public key of an input owner.
    /// Their pk/sk pair must be registered in the wallet's keystore.
    #[arg(long, short, verbatim_doc_comment, value_parser = utils::h256_from_string, default_value = keystore::SHAWN_PUB_KEY
    pub witness: H256,

    #[arg(
        long,
```

iv.Arguments for MineAsteria:

partnerchain-reference-implementation/game/src/lib.rs
Lines 190 to 217 in 1b98b2b

```rust
#[derive(Debug, Args, Clone)]
pub struct MineAsteriaArgs {
    #[arg(long, short, verbatim_doc_comment, value_parser = utils::input_from_string, required = true, value_name = "SHIP_OU
    pub ship: Input,

    /// 32-byte H256 public key of an input owner.
    /// Their pk/sk pair must be registered in the wallet's keystore.
    #[arg(long, short, verbatim_doc_comment, value_parser = utils::h256_from_string, default_value = keystore::SHAWN_PUB_KEY
    pub witness: H256,

    #[arg(
        long,
```

6.Define the logic of each command (the functions called in lib.rs) in game.rs. This constitutes our off-chain code. These modifications are too long to add them all but you can check them out fully here.

We will higlight some key deatils of these implementations:

•Example of a redeemer:

partnerchain-reference-implementation/game/src/game.rs
Lines 130 to 141 in 1b98b2b

```rust
let asteria_redeemer = Redeemer {
    tag: RedeemerTag::Spend,
    index: ordered_inputs
        .iter()
        .position(|i| *i == asteria_input.input)
        .unwrap() as u32,
    data: PlutusData::from(PallasPlutusData::Constr(Constr {
        tag: 121,
        any_constructor: None,
        fields: Indef([].to_vec()),
    })),
};
```

•Example of a datum:

partnerchain-reference-implementation/game/src/game.rs
Lines 167 to 174 in 1b98b2b

```rust
let ship_datum = PallasPlutusData::from(ShipDatum::Ok {
    pos_x: args.pos_x,
    pos_y: args.pos_y,
    ship_token_name: ship_name.clone(),
    pilot_token_name: pilot_name.clone(),
    last_move_latest_time: slot_config.zero_time
        + args.ttl * slot_config.slot_length as u64,
});
```

•Example of an output:

partnerchain-reference-implementation/game/src/game.rs
Lines 192 to 198 in 1b98b2b

```
192     Output {
193         address: spacetime_address,
194         value: Value::Coin(10)
195             + Value::from((spacetime_hash, ship_name.clone(), 1))
196             + Value::from((pellet_policy, fuel_name.clone(), 30)),
197         datum_option: Some(Datum(PlutusData::from(ship_datum.clone()).0)),
198     },
```

•Example of asset burning:

partnerchain-reference-implementation/game/src/game.rs
Lines 764 to 767 in 1b98b2b

```
764     let burns = Some(
765         Multiasset::from((shipyard_policy, ship_token_name.clone(), -1))
766             + Multiasset::from((pellet_policy, fuel_name.clone(), -ship_fuel_i64)),
767     );
```

5.Add deploy_scripts function. This command allows the user to generate new instances of the parameterized scripts to create new games. This command searches for a file that has a structure like this:

partnerchain-reference-implementation/game/src/deploy_params.json
Lines 1 to 14 in 1b98b2b

```
1   {
2       "admin_policy": "516238dd0a79bac4bebe041c44bad8bf880d74720733d2fc0d255d28",
3       "admin_name": "asteriaAdmin",
4       "fuel_per_step": 1,
5       "initial_fuel": 30,
6       "max_speed": {
7           "distance": 1,
8           "time": 30000
9       },
10      "max_ship_fuel": 100,
11      "max_asteria_mining": 50,
12      "min_asteria_distance": 10,
```

This function is also quite long to icnlude it as a whole, but there are some details worth highlighting:

•Example of building the Plutus Data for the parameter application:

partnerchain-reference-implementation/game/src/game.rs
Lines 944 to 959 in 1b98b2b

```
944     let pellet_params = PallasPlutusData::Array(Indef(
945         [PallasPlutusData::Constr(Constr {
946             tag: 121,
947             any_constructor: None,
948             fields: Indef(
949                 [
950                     PallasPlutusData::BoundedBytes(BoundedBytes(
951                         hex::decode(params.admin_policy.clone()).unwrap(),
952                     )),
953                     PallasPlutusData::BoundedBytes(BoundedBytes(params.admin_name.clone().into())),
954                 ]
955                 .to_vec(),
```

•Example of applying parameters to a script:

partnerchain-reference-implementation/game/src/game.rs
Lines 961 to 967 in 1b98b2b

```
961     let pellet_script = PlutusScript(
962         apply_params_to_script(
963             pellet_params.encode_fragment().unwrap().as_slice(),
964             hex::decode(PELLET_PARAMETERIZED).unwrap().as_slice(),
965         )
966         .unwrap(),
967     );
```

Add game to node commands

1.Include the game crate as a dependency in the node's Cargo.toml.

partnerchain-reference-implementation/node/Cargo.toml
Line 25 in 1b98b2b

```
25      game = { workspace = true }
```

2.Add a new Game(GameCommand) item in the Subcommand enum, within node/src/cli.rs.

partnerchain-reference-implementation/node/src/cli.rs
Lines 112 to 114 in 1b98b2b

```
112         /// Commands to play the Asteria game
113         #[clap(flatten)]
114         Game(GameCommand),
```

3.Include the previous item in the subcommands match struct, within the run function in node/src/command.rs.

partnerchain-reference-implementation/node/src/command.rs
Lines 59 to 63 in 1b98b2b

```
59      Some(Subcommand::Game(cmd)) => {
60          let rt = Runtime::new().unwrap();
61          let _ = rt.block_on(cmd.run());
62          Ok(())
63      }
```

Wallet refactor

We'll briefly detail the modifications done to the wallet:

1.Move initial main() setup (cli parser, keystore and db paths, endpoints, sync with node) and auxiliary functions default_data_path and temp_dir to a new context.rs module.
2.Move the rest of auxiliary functions defined in main.rs into a new utils.rs module.
3.Define WalletCommand enum in wallet/src/cli.rs and implement its run() method, moving within the code previously in main().
4.Add library crate feature : include lib.rs.

This refactoring allowed us to include the wallet into the node, along with the game:

1.Include the griffin-wallet package in Cargo.toml.
2.Add a new Wallet(WalletCommand) item in the Subcommand enum, within node/src/cli.rs.
3.Include the previous item in the subcommands match struct, within the run function in node/src/command.rs.