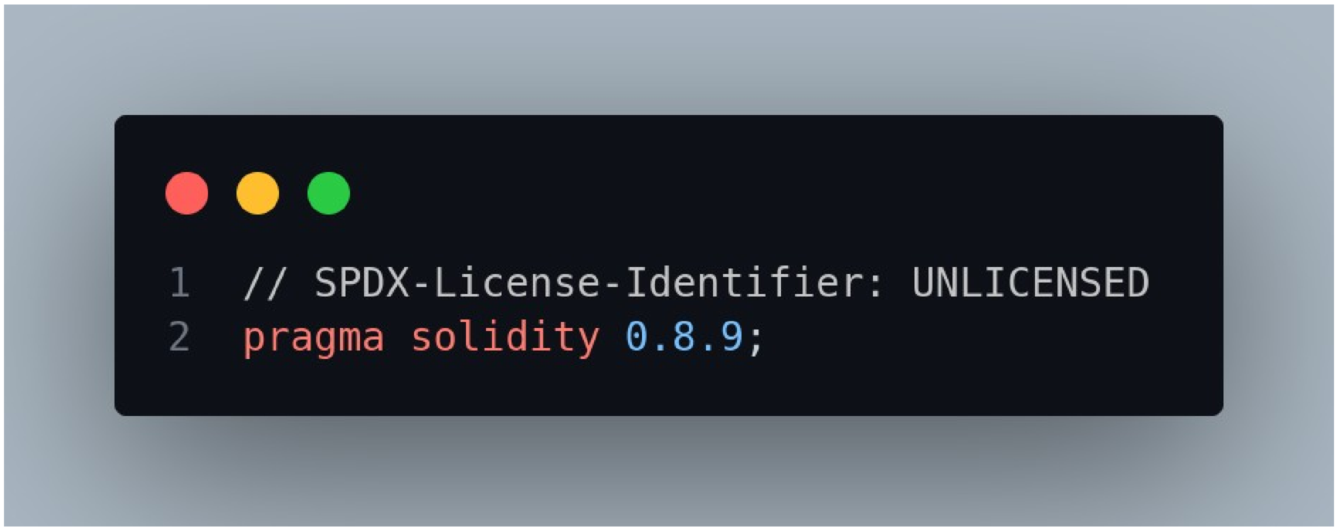
REALM FACET

Moving down a little deeper, one eventually finds the Gotchiverse Realm, which is located beyond the Human Realm, it's a place where smart contracts glitter. The ghost of a yield farmer who is wiped off in the Ether Realm travels to the Gotchiverse to become an Aavegotchi. The Aavegotchi are a unique species who like nothing more than farming and voting. With curiosity human want to know how the Realm was established which led human to checking the source of the Realm (RealmFacet.sol)

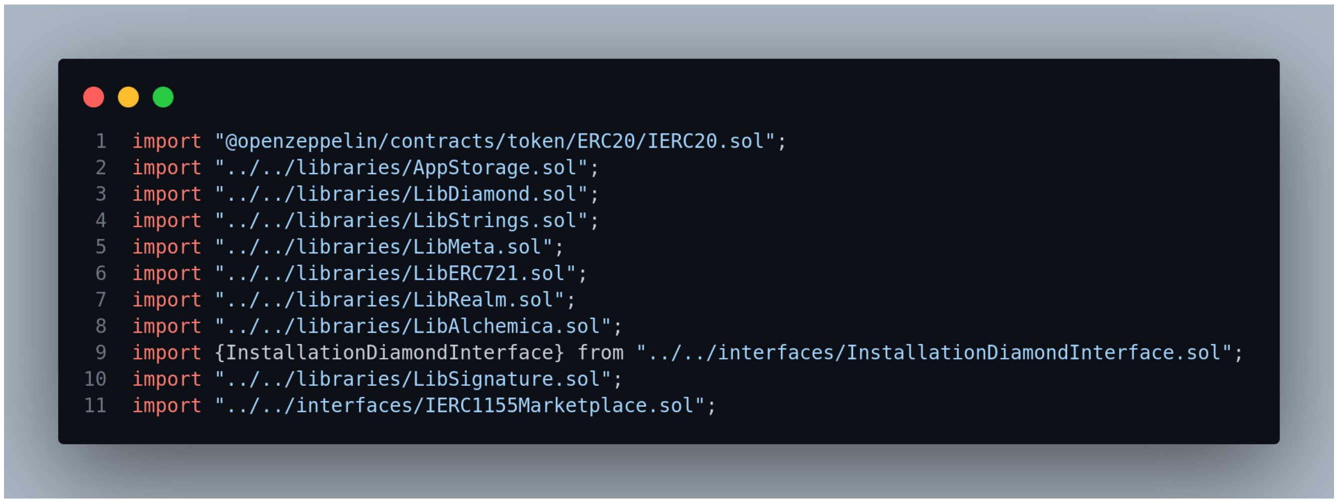
License and Compiler



```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.9;
```

The license Identifier used in the RealmFacet.sol is unlicensed which means anyone is free to copy, modify, publish, use, compile, sell, or distribute this software, either in source code form or as a compiled binary, for any purpose, commercial or non-commercial, and by any means.

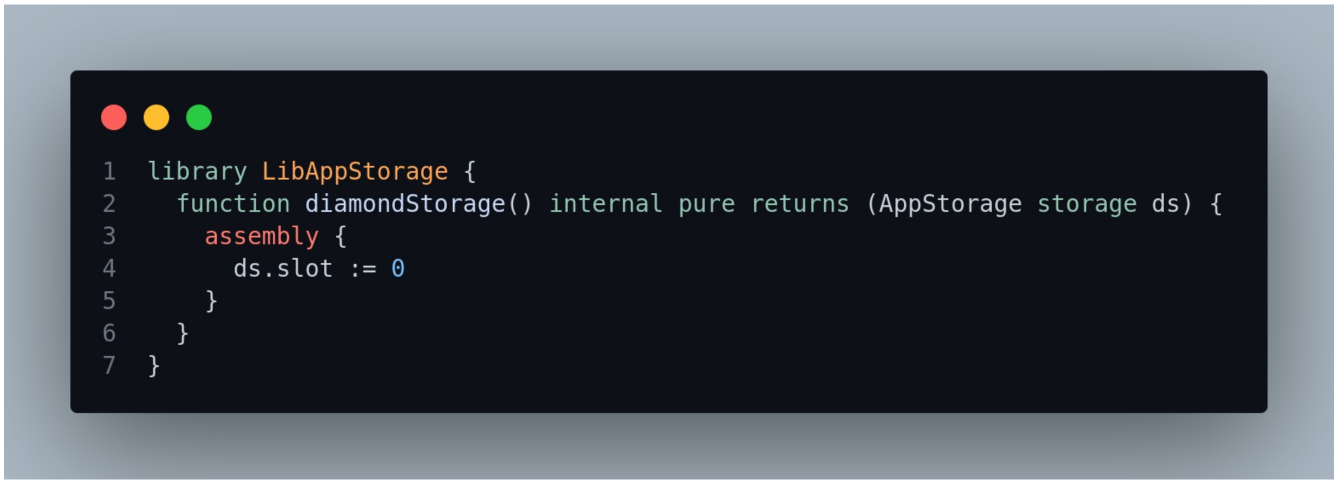
The Solidity version used in this contract is **0.8.9,** this require a compiler version of 0.8.9.



```solidity
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "../../libraries/AppStorage.sol";
import "../../libraries/LibDiamond.sol";
import "../../libraries/LibStrings.sol";
import "../../libraries/LibMeta.sol";
import "../../libraries/LibERC721.sol";
import "../../libraries/LibRealm.sol";
import "../../libraries/LibAlchemica.sol";
import {InstallationDiamondInterface} from "../../interfaces/InstallationDiamondInterface.sol";
import "../../libraries/LibSignature.sol";
import "../../interfaces/IERC1155Marketplace.sol";
```

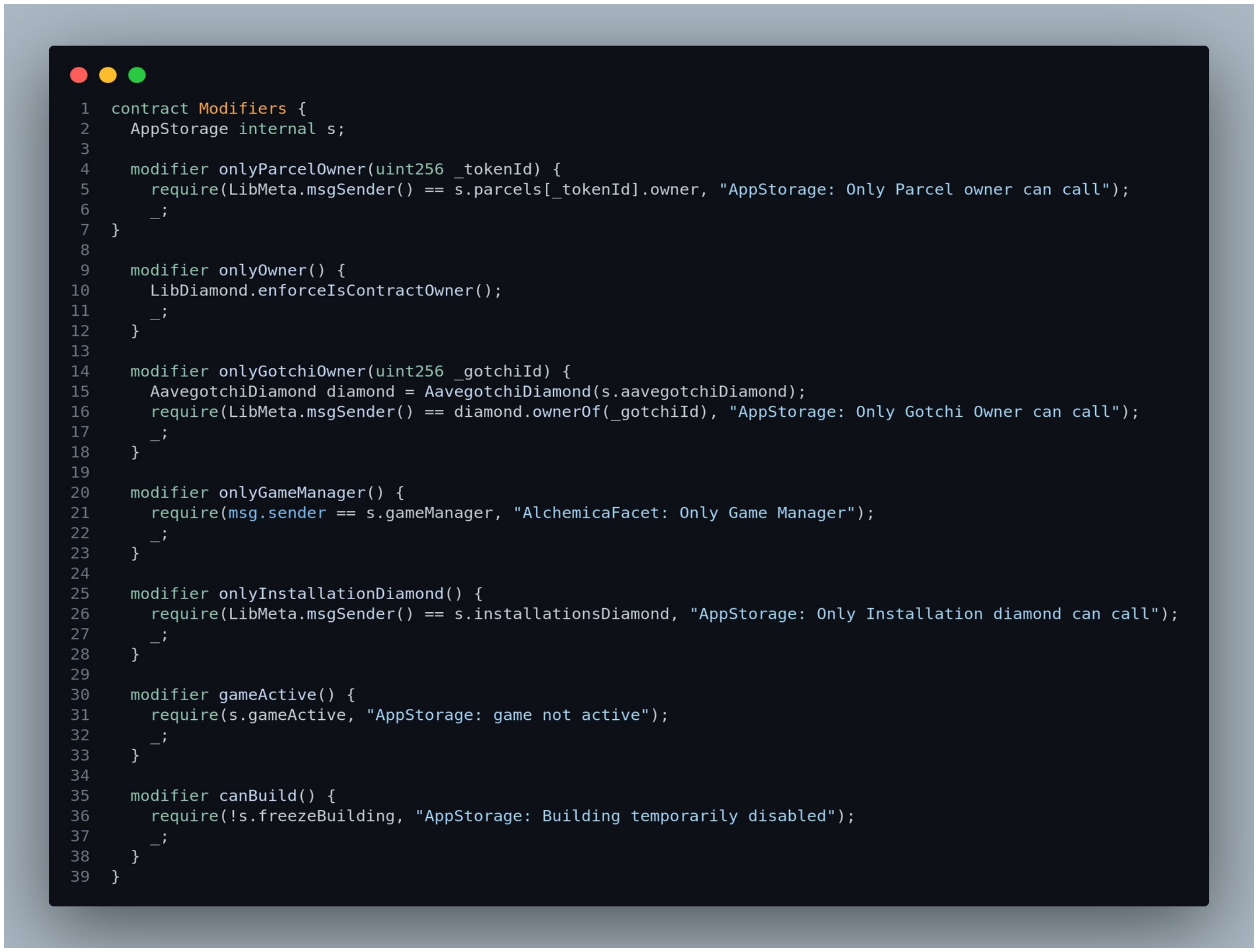
1. import "@open-zeppelin/contracts/token/ERC20/IERC20.sol";
The first import in this , it is an interface for ERC20, the interface gives a framework for the EIP20 standard.

2. import "../../libraries/AppStorage.sol";
The line two is a library import from AppStorage.sol. It is important to note that this contract is an upgradeable contract using the diamond standard for upgradability, the AppStorage.sol has some struct in it which act as a layout for state variables to be used by RealmFacet.sol, the library named LibAppStorage has a function in it pointing to slot zero which can be used within a contract or library function.



```solidity
library LibAppStorage {
  function diamondStorage() internal pure returns (AppStorage storage ds) {
    assembly {
      ds.slot := 0
    }
  }
}
```

Also in the AppStorage.sol is a contract only for modifiers which is inherited by the RealmFacet.sol and contains the usage of the state variable layout declared in slot 0 of the **contract Modifiers.**



```
contract Modifiers {
  AppStorage internal s;

  modifier onlyParcelOwner(uint256 _tokenId) {
    require(LibMeta.msgSender() == s.parcels[_tokenId].owner, "AppStorage: Only Parcel owner can call");
    _;
}

  modifier onlyOwner() {
    LibDiamond.enforceIsContractOwner();
    _;
  }

  modifier onlyGotchiOwner(uint256 _gotchiId) {
    AavegotchiDiamond diamond = AavegotchiDiamond(s.aavegotchiDiamond);
    require(LibMeta.msgSender() == diamond.ownerOf(_gotchiId), "AppStorage: Only Gotchi Owner can call");
    _;
  }

  modifier onlyGameManager() {
    require(msg.sender == s.gameManager, "AlchemicaFacet: Only Game Manager");
    _;
  }

  modifier onlyInstallationDiamond() {
    require(LibMeta.msgSender() == s.installationsDiamond, "AppStorage: Only Installation diamond can call");
    _;
  }

  modifier gameActive() {
    require(s.gameActive, "AppStorage: game not active");
    _;
  }

  modifier canBuild() {
    require(!s.freezeBuilding, "AppStorage: Building temporarily disabled");
    _;
  }
}
```

3. import "../../libraries/LibDiamond.sol";

A LibDiamond.sol library import is a default library that helps in the implementation of EIP2535 standard

4. import "../../libraries/LibStrings.sol";

This is a library import that uses open Zeppelin contract for converting decimals to ASCII string representation.

5. import "../../libraries/LibMeta.sol";

The LibMeta.sol library is helps in getting relayed call sender, a kind of call origin.

6. import "../../libraries/LibERC721.sol";

This is a library that implement some of the functions of the EIP721 standard

7. import "../../libraries/LibRealm.sol";

The LibRealm.sol is a library that helps in some functions used in the build up of the RealmFacet.sol which will later be implemented in some reviews to come.

8. import "../../libraries/LibAlchemica.sol";

This library importation is responsible for Gotchus Alchemica which are tokens that can be obtained in the Gotchiverse. Four tokens that can be obtained in the Gotchiverse are FUD, KUK, ALPHA, FOMO, these tokens can be used to buy installations like building of wall in the Gotchiverse or sold.

9. import {InstallationDiamondInterface} from "../../interfaces/InstallationDiamondInterface.sol";

An interface for installationDiamond.sol. Remember, installation in Gotchiverse are alter, harvester, reservoir, gotchi lodge, wall, NFT display, buildqueue booster

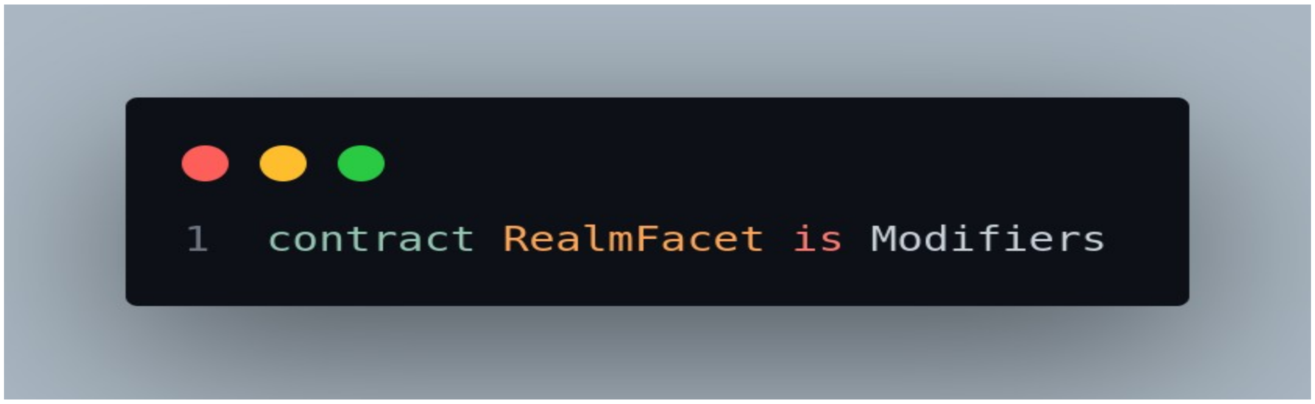
10. import "../../libraries/LibSignature.sol";

A library responsible for getting if a user is who he/she claimed to be, it is validated by telling a user to sign a hashed message.

11. import "../../interfaces/IERC1155Marketplace.sol";

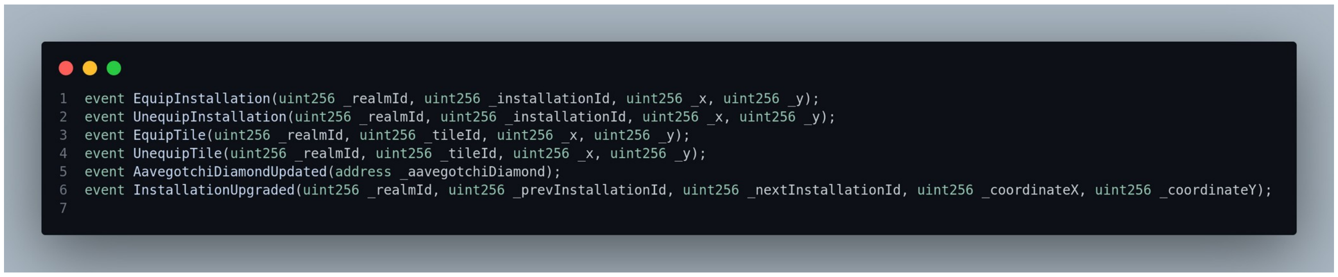
An interface for ERC1155 Market Place, this interface only has one function updateERC1155Listing.

This contract to be reviewed inherit contract Modifiers for its modifiers declaration.



```
contract RealmFacet is Modifiers
```

EVENTS

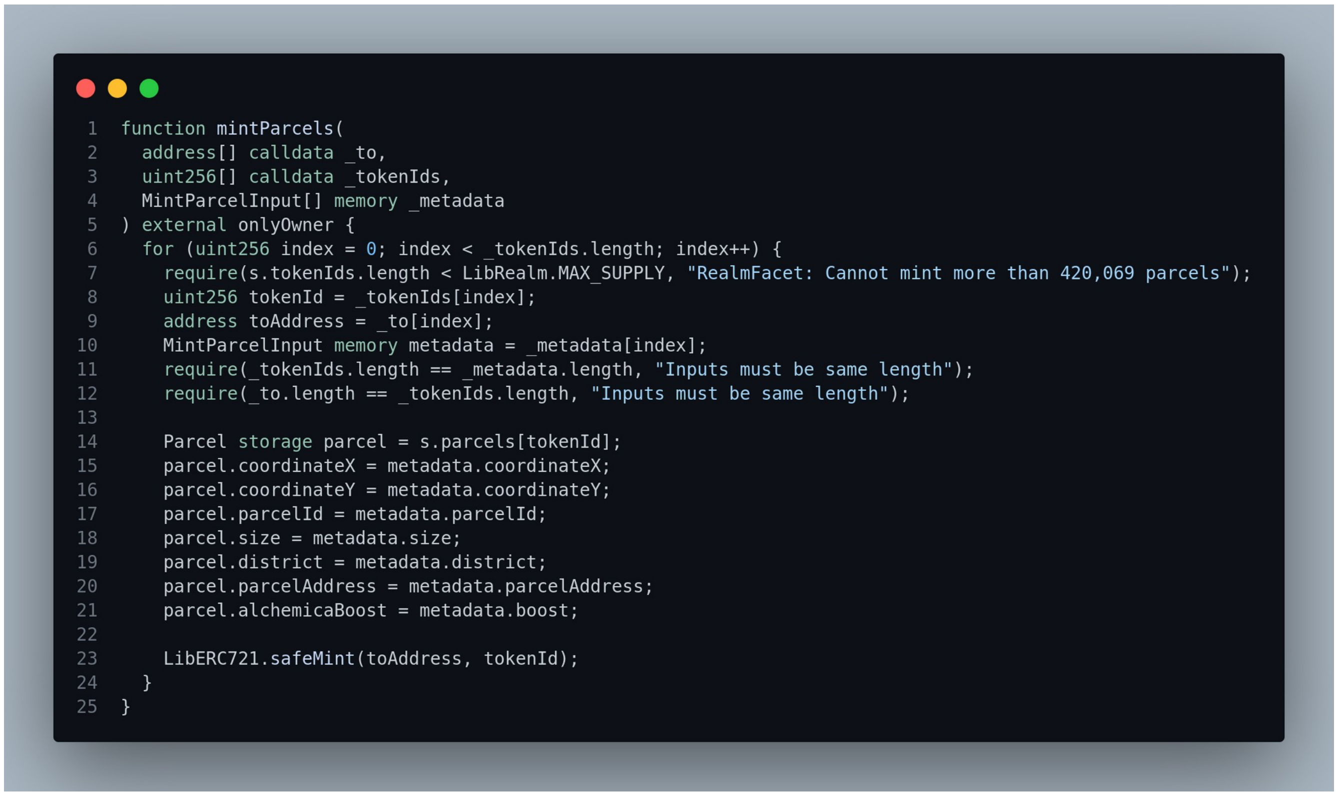


```
event EquipInstallation(uint256 _realmId, uint256 _installationId, uint256 _x, uint256 _y);
event UnequipInstallation(uint256 _realmId, uint256 _installationId, uint256 _x, uint256 _y);
event EquipTile(uint256 _realmId, uint256 _tileId, uint256 _x, uint256 _y);
event UnequipTile(uint256 _realmId, uint256 _tileId, uint256 _x, uint256 _y);
event AavegotchiDiamondUpdated(address _aavegotchiDiamond);
event InstallationUpgraded(uint256 _realmId, uint256 _prevInstallationId, uint256 _nextInstallationId, uint256 _coordinateX, uint256 _coordinateY);

```

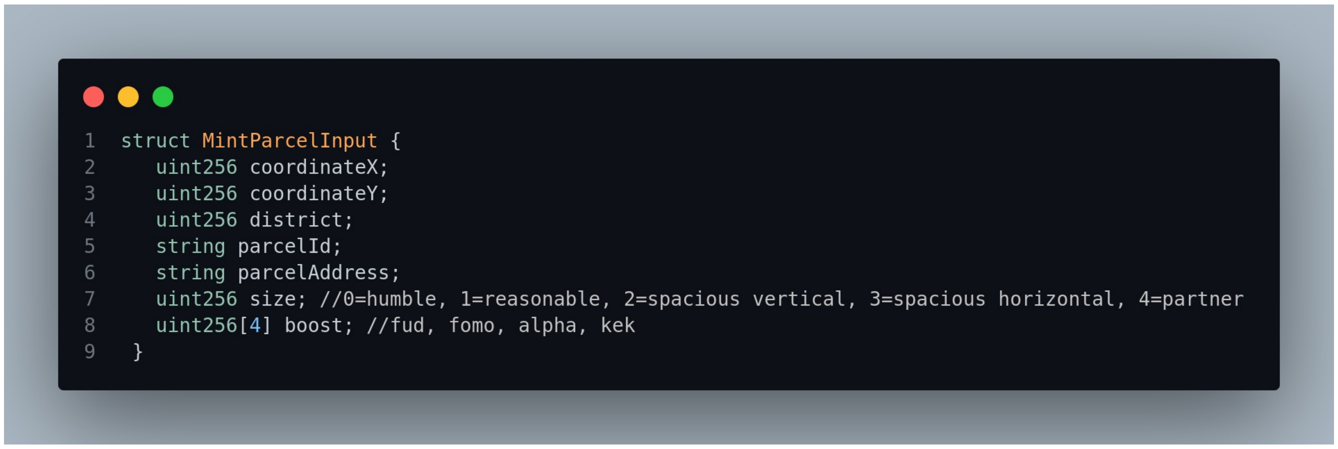
Event **EquipInstallation** will be emitted whenever an Aavegotchi equip an installation. The parameter realmId, the installationId of the user’s purchase and the coordinate x and y will be emitted,whenever an Aavegotchi unequipped an installation the **UnequipInstallation** event will be emitted. For the event **EquipTile,** the event will be logged when an Aavegotchi equipped a tile. Tiles are craft-able cosmetics also known as aesthetica, they can be placed on Gotchiverse land. Event unequipped is being logged out whenever a user unequipped a tile. The event **AavegotchiDiamondUpdated** is being emitted whenever there’s an update to the AavegotchiDiamond. Lastly, the event **InstallationUpgraded** is being emitted whenever there’s an upgrade to the installation, the upgrade in this case is like acquiring a higher installation compared to the previous, it takes in parameter realmId, the previous installationId, the next installationId, the coordinate x and y all in uint256.

FUNCTIONS EXPLANATION

Starting with the function mint parcels **Function mintParcels**



```
function mintParcels(
  address[] calldata _to,
  uint256[] calldata _tokenIds,
  MintParcelInput[] memory _metadata
) external onlyOwner {
  for (uint256 index = 0; index < _tokenIds.length; index++) {
    require(s.tokenIds.length < LibRealm.MAX_SUPPLY, "RealmFacet: Cannot mint more than 420,069 parcels");
    uint256 tokenId = _tokenIds[index];
    address toAddress = _to[index];
    MintParcelInput memory metadata = _metadata[index];
    require(_tokenIds.length == _metadata.length, "Inputs must be same length");
    require(_to.length == _tokenIds.length, "Inputs must be same length");

    Parcel storage parcel = s.parcels[tokenId];
    parcel.coordinateX = metadata.coordinateX;
    parcel.coordinateY = metadata.coordinateY;
    parcel.parcelId = metadata.parcelId;
    parcel.size = metadata.size;
    parcel.district = metadata.district;
    parcel.parcelAddress = metadata.parcelAddress;
    parcel.alchemicaBoost = metadata.boost;

    LibERC721.safeMint(toAddress, tokenId);
  }
}
```

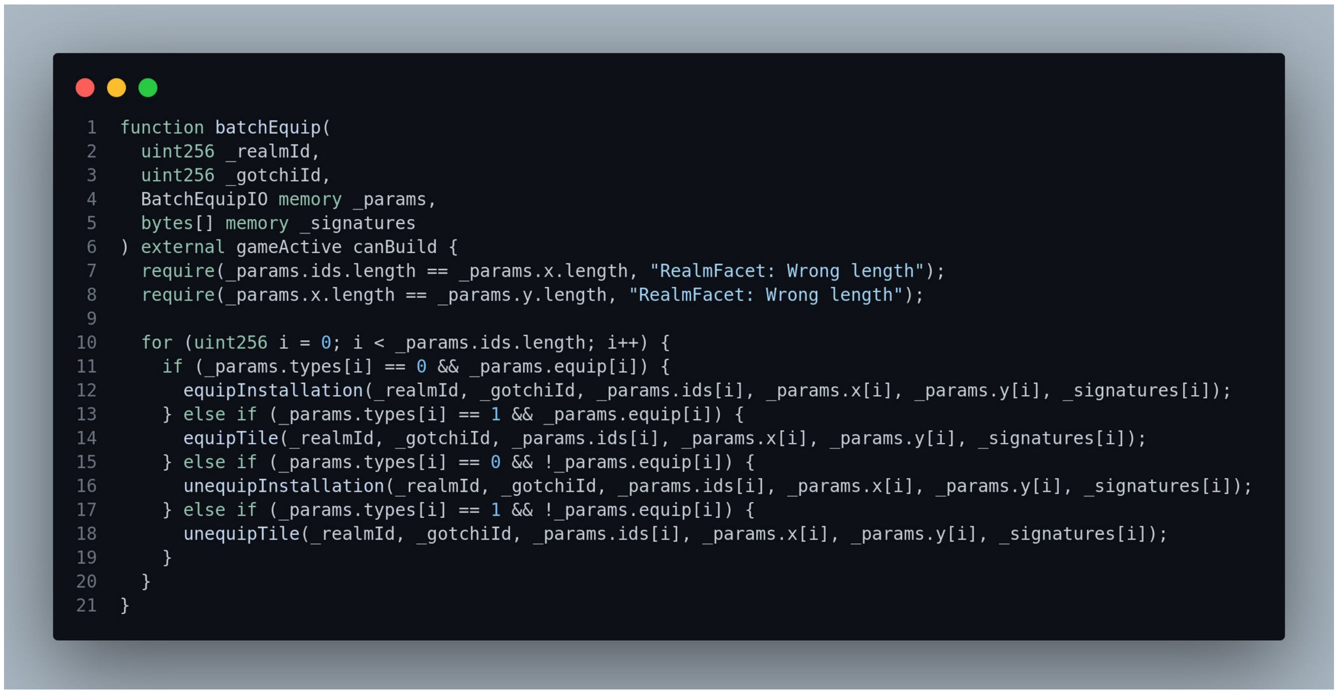


```
struct MintParcelInput {
   uint256 coordinateX;
   uint256 coordinateY;
   uint256 district;
   string parcelId;
   string parcelAddress;
   uint256 size; //0=humble, 1=reasonable, 2=spacious vertical, 3=spacious horizontal, 4=partner
   uint256[4] boost; //fud, fomo, alpha, kek
 }
```

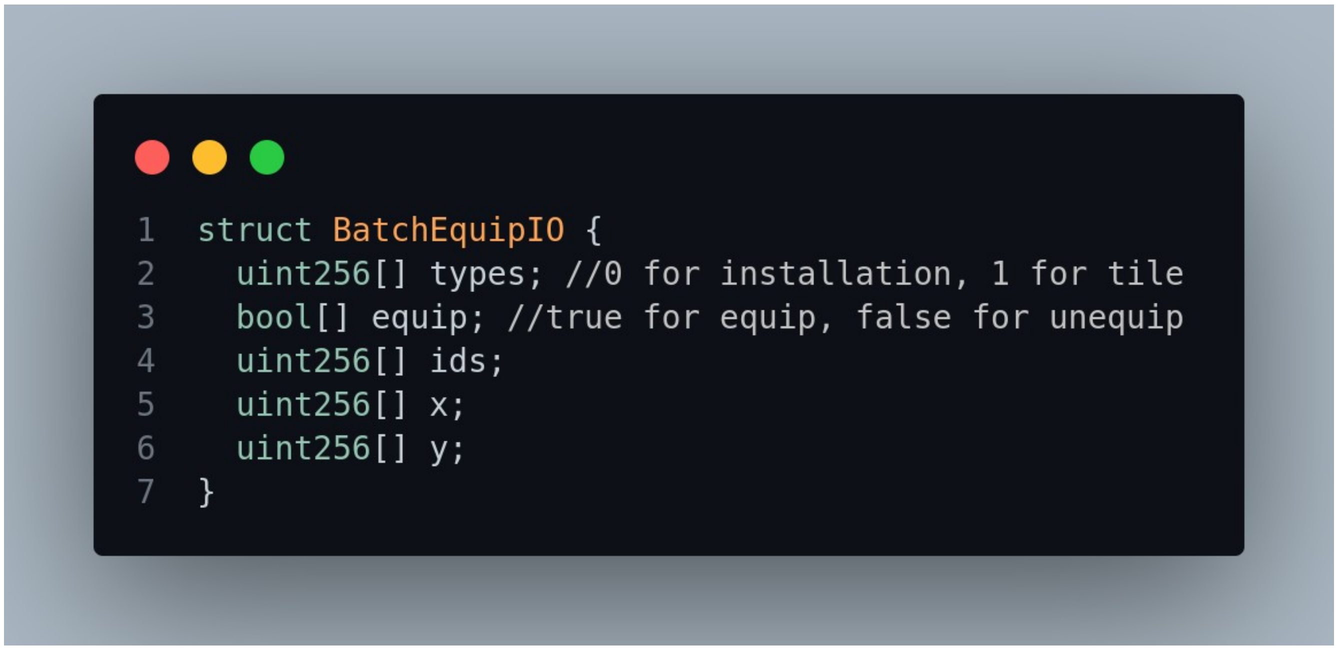
This function is an external function with an onlyOwner modifier that has been declared in the **contract modifier,** which means it can only be called externally by the diamond owner (restricted to the diamond owner). It uses the LibDiamond.enforceIsContractOwner to ensure the function is only called by the diamond owner.

The function allows the diamond owner to mint new parcels. It takes In 3 parameters, the array of addresses to mint to, the tokenIds of token (parcel) to mint and an array of structs containing the metadata of each parcel being minted. The struct contains the parcel coordinate both in x and y axis, the district in number, the parcelId, the parcelAddress, the size which can be either (humble, reasonable, spacious vertical, spacious horizontal or partner). Lastly, the boost which is an array of the gotchus alchemica. Parcel in Gotchiverse are piece of land which are categorized into humble, reasonable, spacious vertical, spacious horizontal and partner, the type of parcel determines the quantity of gotchus alchemica that will be minted on the parcel. Each parcel is represented as an ERC721 NFT and the tokenId of each parcel is used as a pointer to **struct Parcel** in **AppStorage.sol** which is used to store each parcel metadata before safe minting.

Function batchEquip



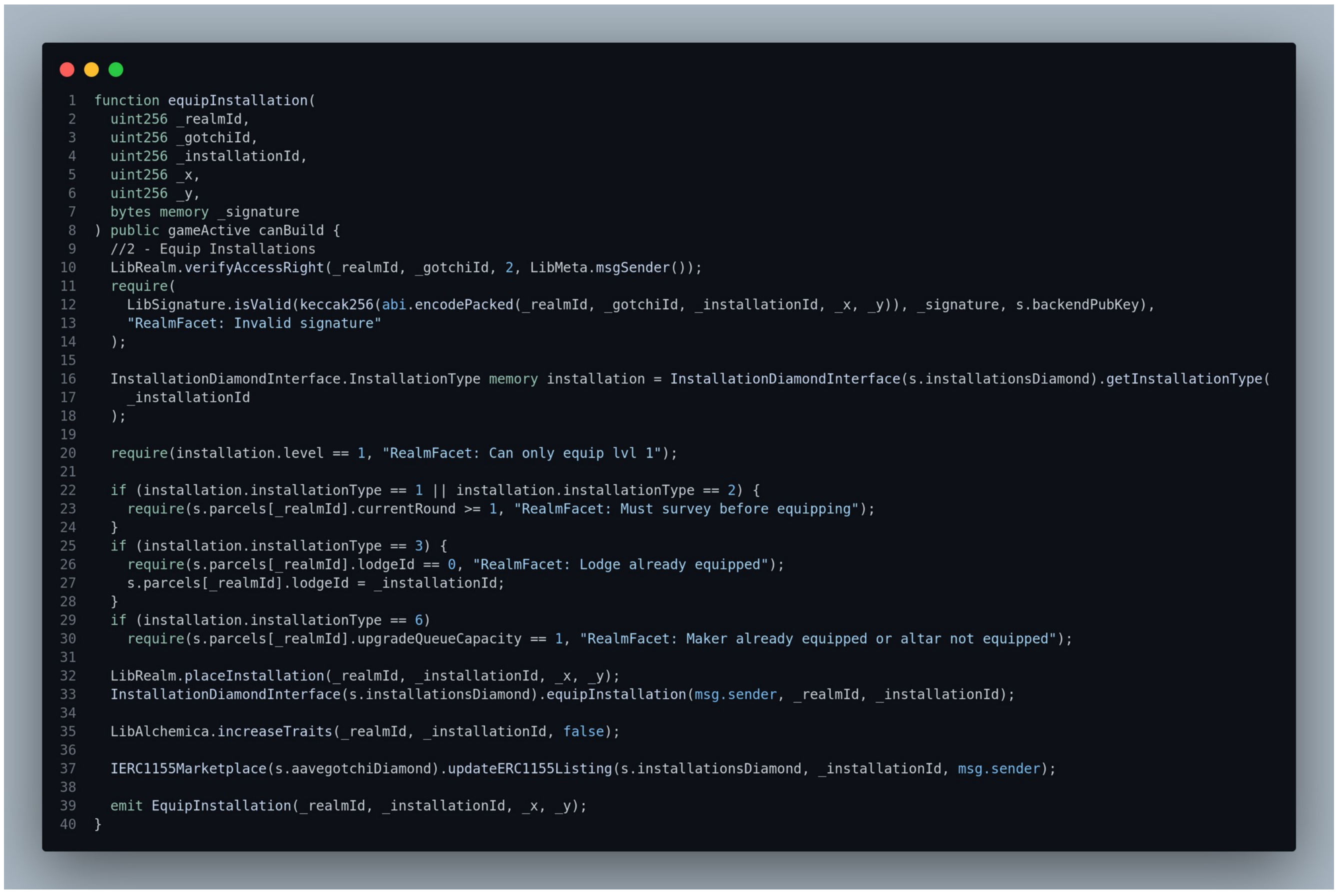
```
function batchEquip(
  uint256 _realmId,
  uint256 _gotchiId,
  BatchEquipIO memory _params,
  bytes[] memory _signatures
) external gameActive canBuild {
  require(_params.ids.length == _params.x.length, "RealmFacet: Wrong length");
  require(_params.x.length == _params.y.length, "RealmFacet: Wrong length");

  for (uint256 i = 0; i < _params.ids.length; i++) {
    if (_params.types[i] == 0 && _params.equip[i]) {
      equipInstallation(_realmId, _gotchiId, _params.ids[i], _params.x[i], _params.y[i], _signatures[i]);
    } else if (_params.types[i] == 1 && _params.equip[i]) {
      equipTile(_realmId, _gotchiId, _params.ids[i], _params.x[i], _params.y[i], _signatures[i]);
    } else if (_params.types[i] == 0 && !_params.equip[i]) {
      unequipInstallation(_realmId, _gotchiId, _params.ids[i], _params.x[i], _params.y[i], _signatures[i]);
    } else if (_params.types[i] == 1 && !_params.equip[i]) {
      unequipTile(_realmId, _gotchiId, _params.ids[i], _params.x[i], _params.y[i], _signatures[i]);
    }
  }
}
```



```
struct BatchEquipIO {
  uint256[] types; //0 for installation, 1 for tile
  bool[] equip; //true for equip, false for unequip
  uint256[] ids;
  uint256[] x;
  uint256[] y;
}
```

The function batchEquip is an external function with modifier **gameActive** and **canBuild.** These modifiers ensure the game is active **(s.gameActive must be true)** and can build **(s.freezeBuilding must be false)** following the modifier declaration in **contract Modifiers** before any batch equip of installation or tile. The function parameters takes the realmId in uint256, the gotchiId in uint256 and the _params in struct data type. The struct contains array of uint256 for types which can be 0 for installation or 1 for tile, also in the struct is an array of bool for variable equip, which can be true for equip or false for unequip, the array of uint256 for ids, the array of uint256 for coordinate x and the array of uint256 for coordinate y.
Basically, the function is used for batch equip of installation and tile, in the logic of this function is a call to another function depending on the condition each parameter meet, the function calls are **equipInstallation, equipTile, unequipInstallation,** and **unequipTile.**

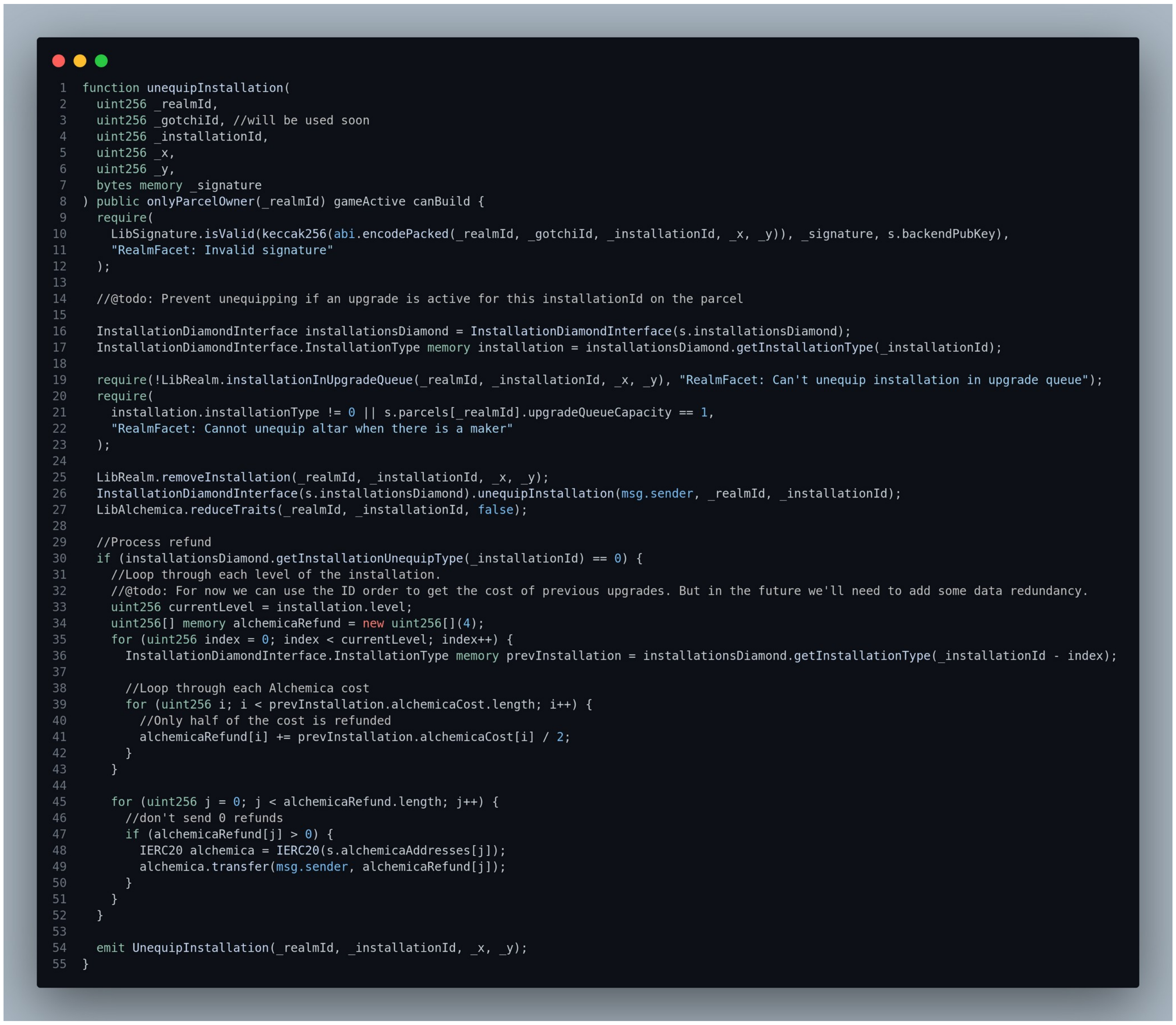
Function equipInstallation



```
function equipInstallation(
  uint256 _realmId,
  uint256 _gotchiId,
  uint256 _installationId,
  uint256 _x,
  uint256 _y,
  bytes memory _signature
) public gameActive canBuild {
  //2 - Equip Installations
  LibRealm.verifyAccessRight(_realmId, _gotchiId, 2, LibMeta.msgSender());
  require(
    LibSignature.isValid(keccak256(abi.encodePacked(_realmId, _gotchiId, _installationId, _x, _y)), _signature, s.backendPubKey),
    "RealmFacet: Invalid signature"
  );

  InstallationDiamondInterface.InstallationType memory installation = InstallationDiamondInterface(s.installationsDiamond).getInstallationType(
    _installationId
  );

  require(installation.level == 1, "RealmFacet: Can only equip lvl 1");

  if (installation.installationType == 1 || installation.installationType == 2) {
    require(s.parcels[_realmId].currentRound >= 1, "RealmFacet: Must survey before equipping");
  }
  if (installation.installationType == 3) {
    require(s.parcels[_realmId].lodgeId == 0, "RealmFacet: Lodge already equipped");
    s.parcels[_realmId].lodgeId = _installationId;
  }
  if (installation.installationType == 6)
    require(s.parcels[_realmId].upgradeQueueCapacity == 1, "RealmFacet: Maker already equipped or altar not equipped");

  LibRealm.placeInstallation(_realmId, _installationId, _x, _y);
  InstallationDiamondInterface(s.installationsDiamond).equipInstallation(msg.sender, _realmId, _installationId);

  LibAlchemica.increaseTraits(_realmId, _installationId, false);

  IERC1155Marketplace(s.aavegotchiDiamond).updateERC1155Listing(s.installationsDiamond, _installationId, msg.sender);

  emit EquipInstallation(_realmId, _installationId, _x, _y);
}
```

This function allows a parcel owner to equip installation, and it is important to note that installations in Gotchiverse are things built on a parcel like wall, alter, harvester, reservoir, gotchi lodge, NFT display, buildqueue booster. The function is a public function and takes in the same modifiers as **function batchEquip** which are the gameActive modifier and canBuild Modifier. Before an installation is equipped, the access right of a user is being checked using **function verifyAccessRight** from **library LibRealm** which uses a 2D mapping to check for user's access right with parcelId and actions as the key for the 2D mapping, Before equipping installation the LibSignature.isValid requirement must be passed. the LibSignature.isValid is used to validate that users are who they claimed to be by telling a user to sign an hashed message with their private key, in this manner real users can be verified from their signature. We also ensure the level of our installation gotten from the query of getInstallationType in **contract installationFacet** is equal to 1. if all of this requirement are met, the installation will be equipped depending on the condition passed. And the **event EquipInstallation** will be emmited.

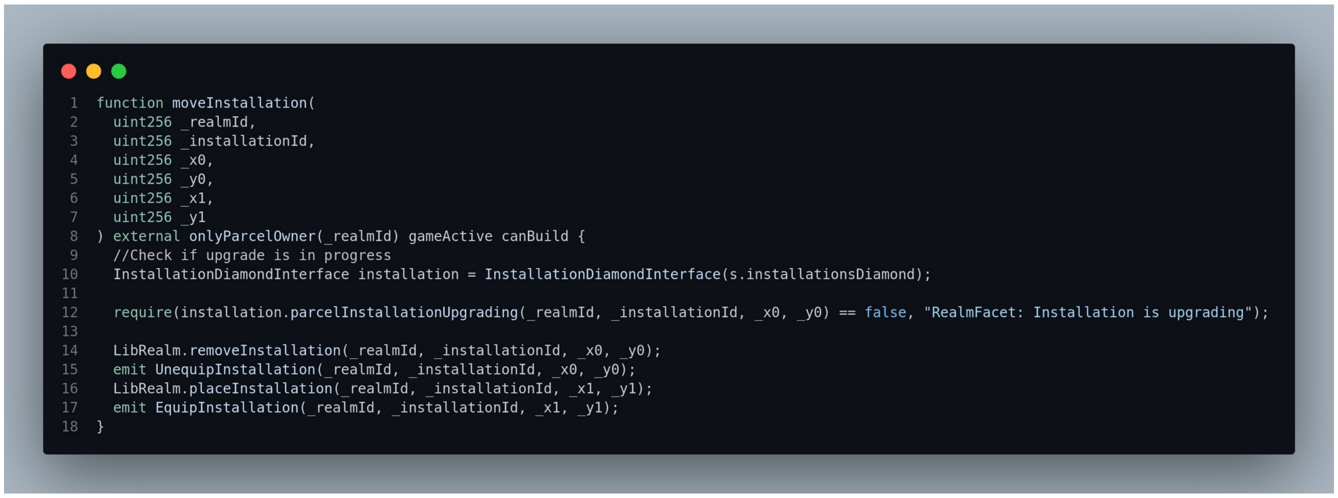
Function unequipInstallation



```solidity
function unequipInstallation(
  uint256 _realmId,
  uint256 _gotchiId, //will be used soon
  uint256 _installationId,
  uint256 _x,
  uint256 _y,
  bytes memory _signature
) public onlyParcelOwner(_realmId) gameActive canBuild {
  require(
    LibSignature.isValid(keccak256(abi.encodePacked(_realmId, _gotchiId, _installationId, _x, _y)), _signature, s.backendPubKey),
    "RealmFacet: Invalid signature"
  );

  //@todo: Prevent unequipping if an upgrade is active for this installationId on the parcel

  InstallationDiamondInterface installationsDiamond = InstallationDiamondInterface(s.installationsDiamond);
  InstallationDiamondInterface.InstallationType memory installation = installationsDiamond.getInstallationType(_installationId);

  require(!LibRealm.installationInUpgradeQueue(_realmId, _installationId, _x, _y), "RealmFacet: Can't unequip installation in upgrade queue");
  require(
    installation.installationType != 0 || s.parcels[_realmId].upgradeQueueCapacity == 1,
    "RealmFacet: Cannot unequip altar when there is a maker"
  );

  LibRealm.removeInstallation(_realmId, _installationId, _x, _y);
  InstallationDiamondInterface(s.installationsDiamond).unequipInstallation(msg.sender, _realmId, _installationId);
  LibAlchemica.reduceTraits(_realmId, _installationId, false);

  //Process refund
  if (installationsDiamond.getInstallationUnequipType(_installationId) == 0) {
    //Loop through each level of the installation.
    //@todo: For now we can use the ID order to get the cost of previous upgrades. But in the future we'll need to add some data redundancy.
    uint256 currentLevel = installation.level;
    uint256[] memory alchemicaRefund = new uint256[](4);
    for (uint256 index = 0; index < currentLevel; index++) {
      InstallationDiamondInterface.InstallationType memory prevInstallation = installationsDiamond.getInstallationType(_installationId - index);

      //Loop through each Alchemica cost
      for (uint256 i; i < prevInstallation.alchemicaCost.length; i++) {
        //Only half of the cost is refunded
        alchemicaRefund[i] += prevInstallation.alchemicaCost[i] / 2;
      }
    }

    for (uint256 j = 0; j < alchemicaRefund.length; j++) {
      //don't send 0 refunds
      if (alchemicaRefund[j] > 0) {
        IERC20 alchemica = IERC20(s.alchemicaAddresses[j]);
        alchemica.transfer(msg.sender, alchemicaRefund[j]);
      }
    }
  }

  emit UnequipInstallation(_realmId, _installationId, _x, _y);
}
```

This function allows a parcel owner to equip an installation, it has six parameters, realmId, gotchiId, installationId, the x and y coordinate and signature. It is a public function and has the **modifiers onlyParcelOwner(realmId), gameActive, canBuild,** the onlyParcelOwner(realmId) checks if the caller has parcel. The **function isValid** as explained above is required to be true, a check if an installation is not in upgrade queue must also be passed before an installation is removed. Whenever an installation is being unequipped, it is burnt and half of the gotchus alchemica used in the purchase of the installation will be refunded. When an installation is unequipped the **event UnequipInstallation** is being emitted whenever an installation is unequipped.

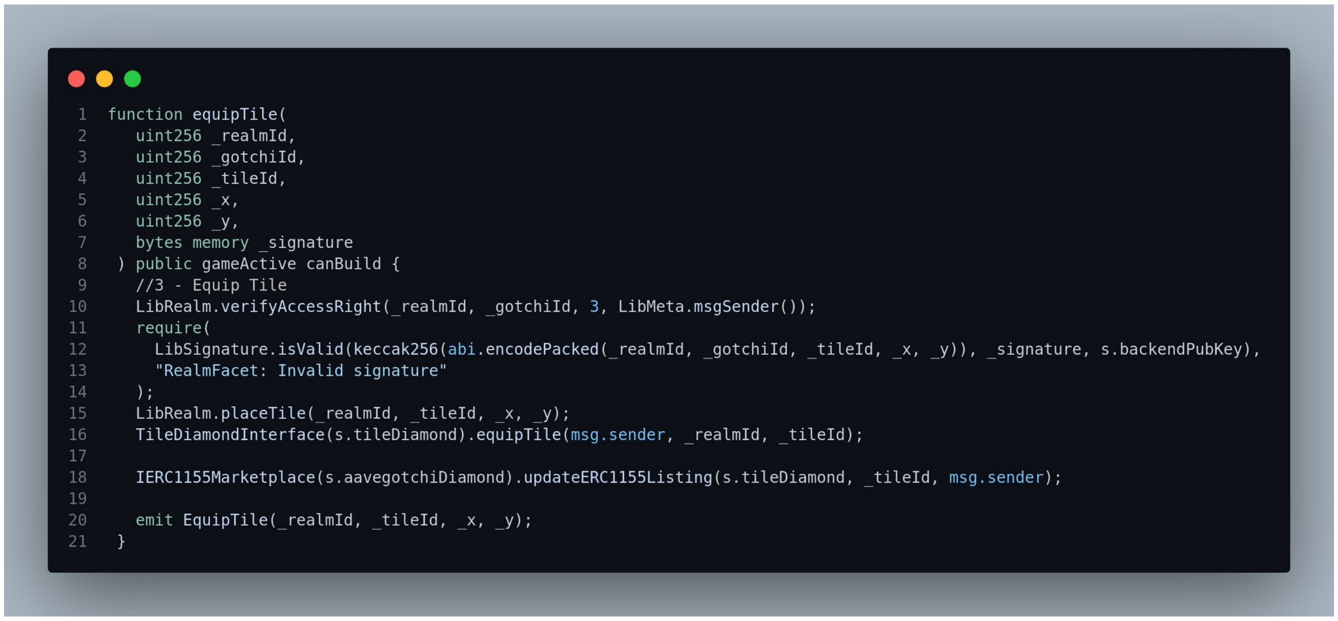
Function moveInstallation



```solidity
function moveInstallation(
  uint256 _realmId,
  uint256 _installationId,
  uint256 _x0,
  uint256 _y0,
  uint256 _x1,
  uint256 _y1
) external onlyParcelOwner(_realmId) gameActive canBuild {
  //Check if upgrade is in progress
  InstallationDiamondInterface installation = InstallationDiamondInterface(s.installationsDiamond);

  require(installation.parcelInstallationUpgrading(_realmId, _installationId, _x0, _y0) == false, "RealmFacet: Installation is upgrading");

  LibRealm.removeInstallation(_realmId, _installationId, _x0, _y0);
  emit UnequipInstallation(_realmId, _installationId, _x0, _y0);
  LibRealm.placeInstallation(_realmId, _installationId, _x1, _y1);
  emit EquipInstallation(_realmId, _installationId, _x1, _y1);
}
```

Function moveInstallation is simmilar to both using unequipInstallation and equipInstallation at the same time but with changes to the coordinate. The parameters that comes with function moveInstallation are the realmId which is the identifier of the parcel which the installation is being moved on, installationId which is the identifier of the installation being moved, initial x and y coordinate (the current x and y coordinate) and the coordinate a user intend to move to. It has the same modifiers as it is in **function unequipInstallation**. The **event UnequipInstallation** and **event EquipInstallation** are being emitted when an installation is moved from one coordinate to another coordinate.

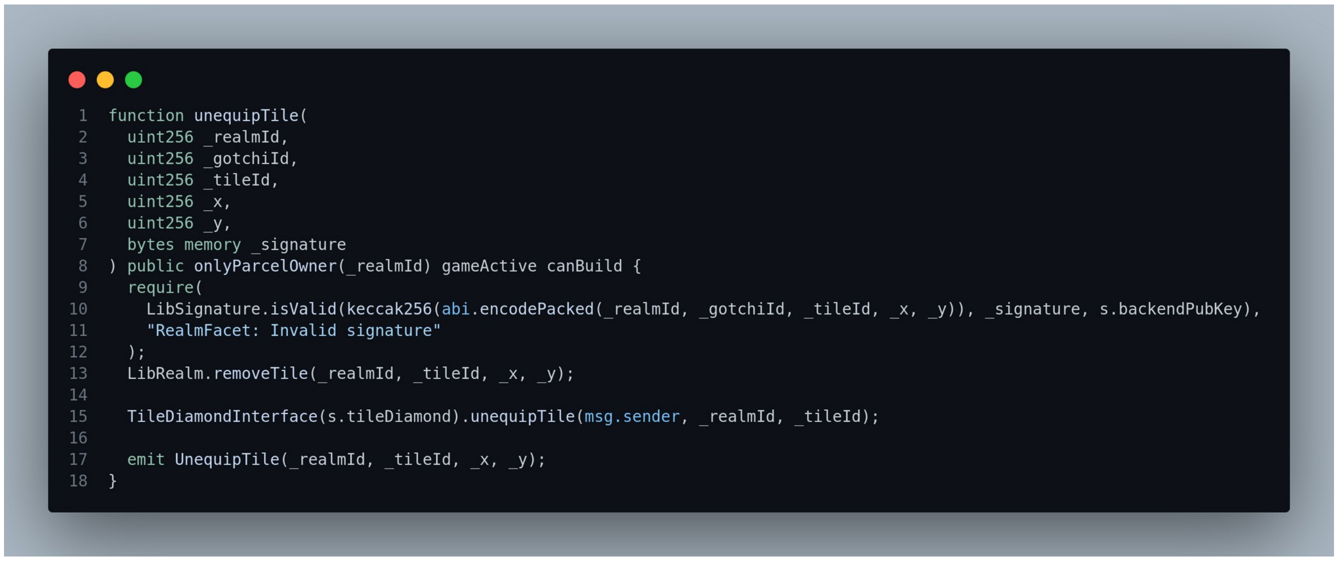
Function equipTile



```solidity
function equipTile(
    uint256 _realmId,
    uint256 _gotchiId,
    uint256 _tileId,
    uint256 _x,
    uint256 _y,
    bytes memory _signature
  ) public gameActive canBuild {
    //3 - Equip Tile
    LibRealm.verifyAccessRight(_realmId, _gotchiId, 3, LibMeta.msgSender());
    require(
      LibSignature.isValid(keccak256(abi.encodePacked(_realmId, _gotchiId, _tileId, _x, _y)), _signature, s.backendPubKey),
      "RealmFacet: Invalid signature"
    );
    LibRealm.placeTile(_realmId, _tileId, _x, _y);
    TileDiamondInterface(s.tileDiamond).equipTile(msg.sender, _realmId, _tileId);

    IERC1155Marketplace(s.aavegotchiDiamond).updateERC1155Listing(s.tileDiamond, _tileId, msg.sender);

    emit EquipTile(_realmId, _tileId, _x, _y);
  }
```

The equip tile allows a parcel owner to equip a tile, the parameters are realmId (which is the identifier of the parcel to which the tile is being equipped on), gotchiId, the tileId (which is the identifier of the tile being equipped, the x and y coordinate denote the starting coordinate of the tile and are used to make sure that slot is available on a parcel The logic in **function equipTile** is similar to that of **function equipInstallation**, they have the same modifiers gameActice and canBuild, it verifies user's access right as equipInstallation. User's signature is also being validated with the address they claim to be using. After passing through these requirements, a Tile can be equipped, ERC1155Listings is updated and **event EquipTile** is being emmited or logged to the frontend.

Function unequipTile

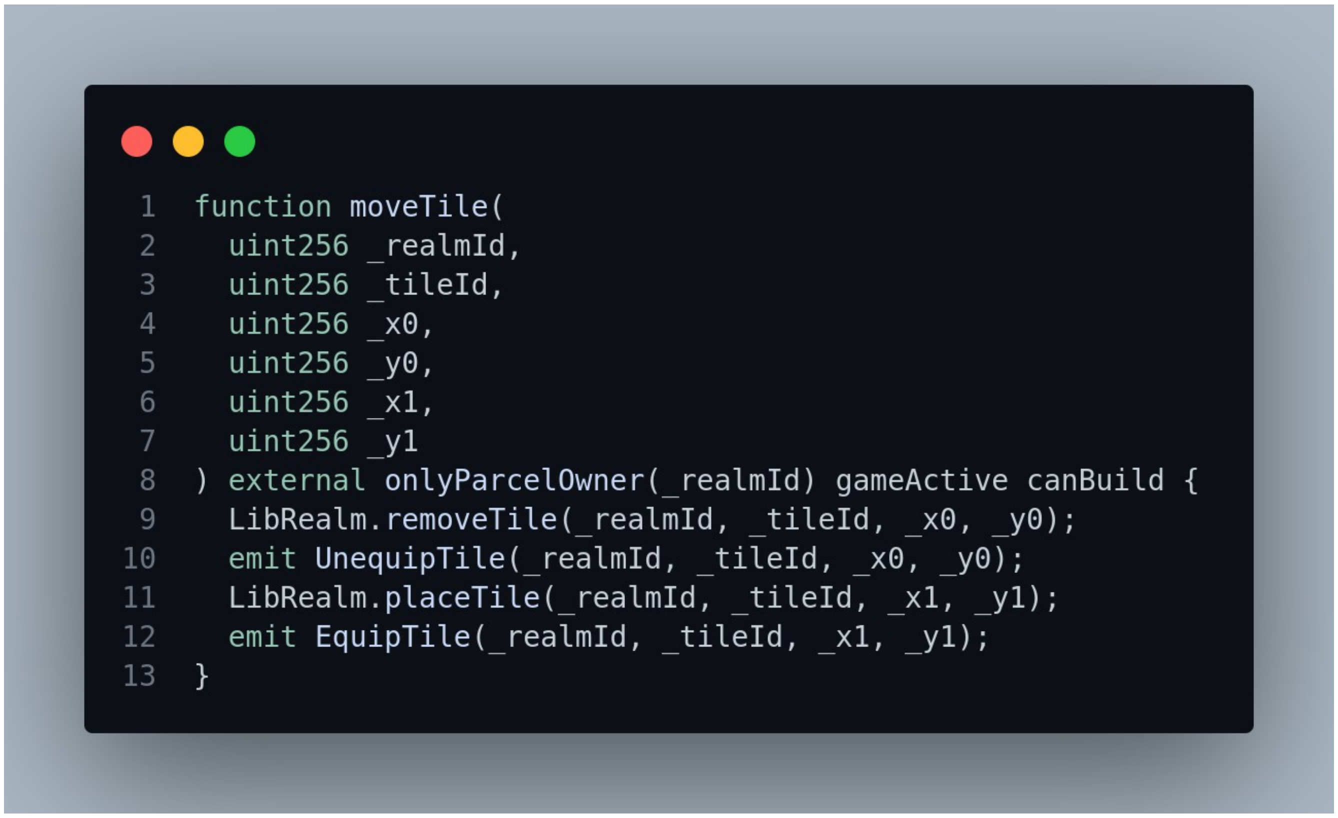


```
function unequipTile(
  uint256 _realmId,
  uint256 _gotchiId,
  uint256 _tileId,
  uint256 _x,
  uint256 _y,
  bytes memory _signature
) public onlyParcelOwner(_realmId) gameActive canBuild {
  require(
    LibSignature.isValid(keccak256(abi.encodePacked(_realmId, _gotchiId, _tileId, _x, _y)), _signature, s.backendPubKey),
    "RealmFacet: Invalid signature"
  );
  LibRealm.removeTile(_realmId, _tileId, _x, _y);

  TileDiamondInterface(s.tileDiamond).unequipTile(msg.sender, _realmId, _tileId);

  emit UnequipTile(_realmId, _tileId, _x, _y);
}
```

function unequipTile allow a parcel owner to unequip a tile, it has six parameters which are the realmId (identifier of the parcel which the tile is being equipped from), gotchiId, tileId (identifier of the tile being unequipped), the x coordinate of a tile, the y coordinate of a tile all in uint256, and signature in bytes (expected to be 65 bytes). All the parameters excluding the signature are being concatenated and hashed together to form a hashed message.

The unequipTile has the same modifiers **onlyParcelOwner(relamId), gameActive** and **canBuild** as that of unequipInstalation, after user claimed they are the real owner of an address a tile will then be unequipped and the unequipTile will be logged. The unequipTile log contains the realmId, tileId, and coordiante x and y.

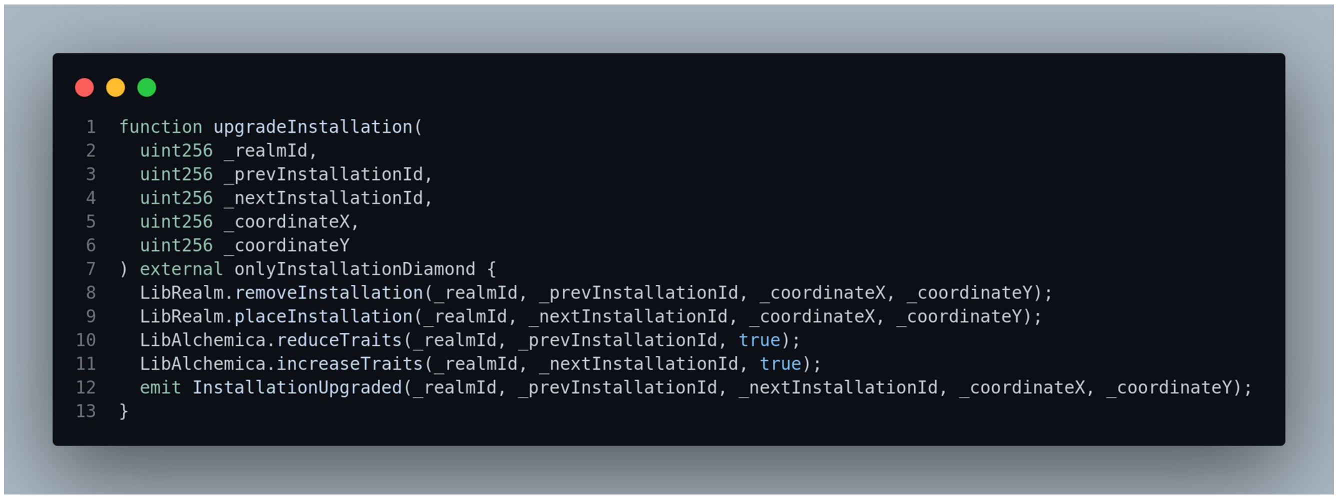
function moveTile



```
function moveTile(
  uint256 _realmId,
  uint256 _tileId,
  uint256 _x0,
  uint256 _y0,
  uint256 _x1,
  uint256 _y1
) external onlyParcelOwner(_realmId) gameActive canBuild {
  LibRealm.removeTile(_realmId, _tileId, _x0, _y0);
  emit UnequipTile(_realmId, _tileId, _x0, _y0);
  LibRealm.placeTile(_realmId, _tileId, _x1, _y1);
  emit EquipTile(_realmId, _tileId, _x1, _y1);
}
```

The function moveTile allows a parcel owner to move a tile from one coordinate to another coordinate, the move is done by removing the parcel owner's initial coordinate and adding a new coordinate for both x and y. The moveTile is an external function with modifiers **onlyParcelOwner(_realmId), gameActive** and **canBuild,** once this modifiers are passed users will be able to get a new coordinate for there tile after this is done two event is being emitted or logged. The event UnequipTile (having the coordinate x and y of the tile, the realmId and the tileId) and EquipTile (having the coordinate x and y parcel owner wants to move there tile to, the realmId and the tileId. Function moveTile is also similar to move installation in logic and modifiers.

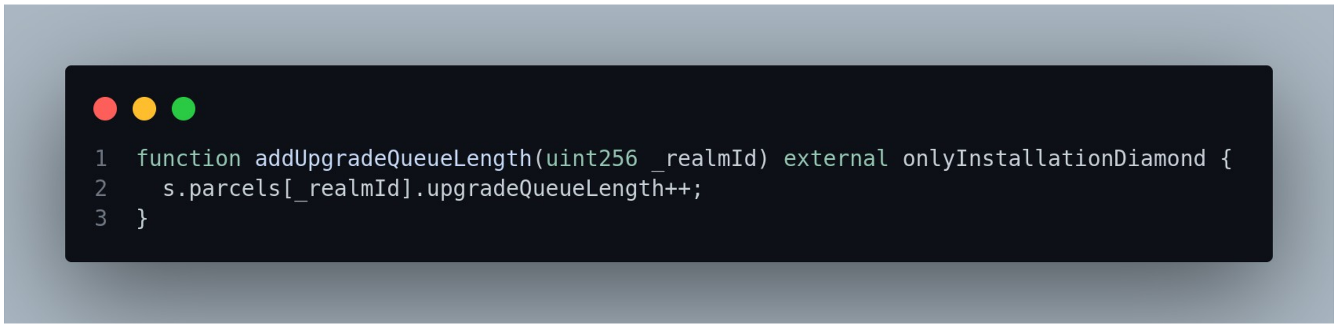
Function upgradeInstallation



```
function upgradeInstallation(
  uint256 _realmId,
  uint256 _prevInstallationId,
  uint256 _nextInstallationId,
  uint256 _coordinateX,
  uint256 _coordinateY
) external onlyInstallationDiamond {
  LibRealm.removeInstallation(_realmId, _prevInstallationId, _coordinateX, _coordinateY);
  LibRealm.placeInstallation(_realmId, _nextInstallationId, _coordinateX, _coordinateY);
  LibAlchemica.reduceTraits(_realmId, _prevInstallationId, true);
  LibAlchemica.increaseTraits(_realmId, _nextInstallationId, true);
  emit InstallationUpgraded(_realmId, _prevInstallationId, _nextInstallationId, _coordinateX, _coordinateY);
}
```

This function is responsible for upgrade to an installation equipped by parcel owner, the function is an external function and has a **modifier onlyInstallationDiamond** which means only installation diamond can call. Function upgradeInstallation has five parameters, _relamId, _prevInstallationId (previous installation Id), _nextInstallationId (next installation ID) and the coordinate x and y. this function is basically changing the installation Id from one to another and event installationUpgraded is logged or emitted.

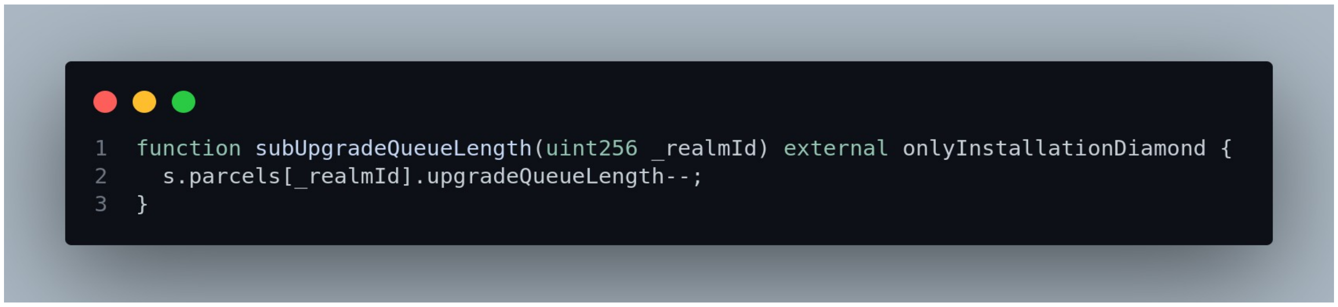
Function addUpgradeQueueLength



```
function addUpgradeQueueLength(uint256 _realmId) external onlyInstallationDiamond {
  s.parcels[_realmId].upgradeQueueLength++;
}
```

function addUpgradeQueueLength increases the length of upgrade queue length for a parcel, it is an external function and has a modifier onlyInstallationDiamond as we have in upgradeInstallation.

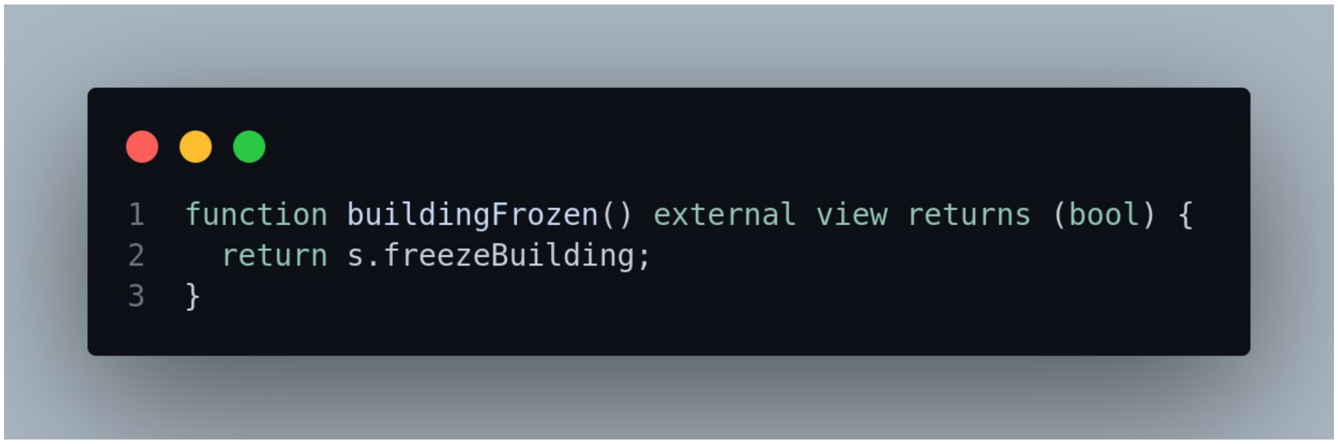
Function subUpgradeQueueLength



```
function subUpgradeQueueLength(uint256 _realmId) external onlyInstallationDiamond {
  s.parcels[_realmId].upgradeQueueLength--;
}
```

function subUpgradeQueueLength is an inverse of addUpgradeQueueLength, dong the same thing in an opposite direction with an external functional visibility (can only be call externally alone) and onlyInstallationDiamond modifier.

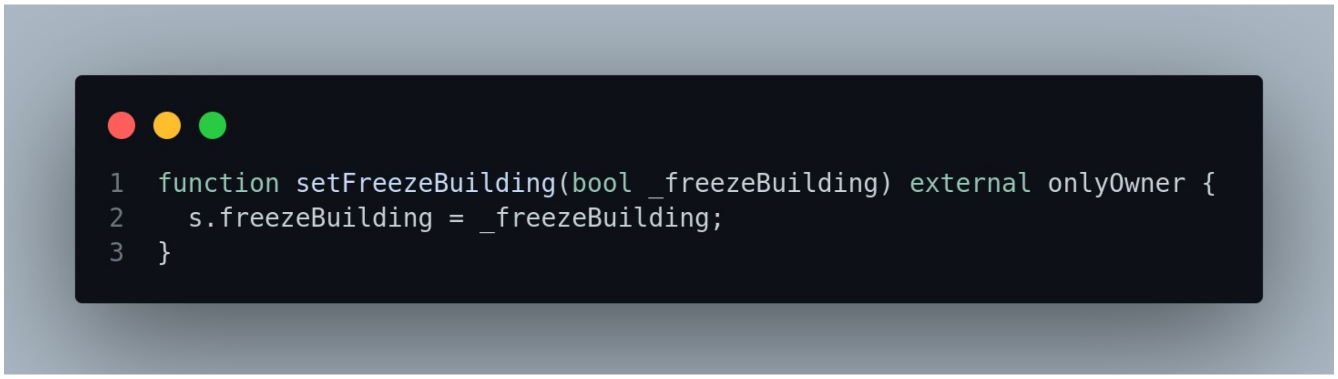
Function buildingFrozen



```
function buildingFrozen() external view returns (bool) {
  return s.freezeBuilding;
}
```

Function buildingFrozen returns true or false for s.freezeBuilding, which means s.freezBuilding has a boolean data type.

Function setFreezeBuilding



```
function setFreezeBuilding(bool _freezeBuilding) external onlyOwner {
  s.freezeBuilding = _freezeBuilding;
}
```

This function is responsible for set freeze building to either true or false, it can only be called by the diamond owner.