Cahier des Charges :

Steam Runner

Par CLARAS Damien (11203738) et Pierre Casati (11206386)

Table des matières :

I - Présentation du projet.

Nous sommes deux étudiants de l'université Lyon 1 à la Doua en licence math-info du cursus préparatoire intégré au réseau d'école *Polytech*. Notre projet s'inscrit dans le cursus de LIF7 du semestre 4. Le but est d'inventer et de réaliser un jeu dans un temps limité en utilisant nos connaissances en gestion de projet, pour le mener à bien, et nos connaissances en informatique pour le développer.

Nous avons choisi de réaliser un *rogue like steam punk post apocalyptique* que nous avons appelé Steam Runner.

Un *rogue like* est un jeu proposant des parties plus ou moins courtes, appelées *rush*, au cours desquelles on dirige un personnage à travers une série de niveaux. A la fin d'un *rush*, le personnage est supprimé et on recommence tout depuis le début lors d'un nouveau *rush*. Le but est donc généralement d'arriver le plus loin possible ou de faire le meilleur score au fil des *rush*. L'intérêt de ce type de jeu réside dans le coté aléatoire qui doit être très présent pour permettre une grande variété de *rush*.

Steam punk est un terme utilisé pour désigner un style d'univers "rétrofuturiste", dont l'action se déroule dans l'atmosphère de la société industrielle du XIX^{e} siècle. Le terme fait référence à l'utilisation massive des machines à vapeur au début de la révolution industrielle puis à l'époque victorienne, avec des composants contemporains comme la robotique, la connaissance scientifique avancée …

Post apocalytpique est un terme utilisé pour désigner un style d'univers où l'humanité se retrouve au bord de l'extinction suite à une catastrophe mondiale (généralement de type guerre atomique ou phénomène inexpliqué).

Notre jeu est donc un *rogue like* dont l'action se déroule dans un univers où l'humanité a presque disparue, et où la vapeur est la source d'énergie communément utilisée, notamment dans une robotique proche des plans les plus farfelus de Leonard de Vinci.

II - *Feature* du jeu.

Une grande rejouabilité :

La rejouabilité, pour un jeu vidéo, désigne l'intérêt pour le joueur de recommencer une partie. Il faut donc que le jeu contienne beaucoup d'éléments aléatoires, comme la génération du héros, des ennemis, et des niveaux, pour permettre au jeu de se renouveler. Un *rush* sera toujours différent du précédent. Il sera donc composé d'une série de salles, qui contiennent un ennemi ou un événement aléatoire.

Un scénario :

Le jeu doit comporter un scénario qui s'inscrit dans l'univers inventé. Ce scénario sera décrit au cours du jeu. Le joueur doit avoir envie de continuer à jouer non seulement parce que le jeu est amusant mais aussi pour découvrir le monde et l'histoire du héros. Le *rush* doit donc contenir des salles obligatoires pour faire avancer l'histoire. Le scénario doit avoir une fin alternative (voir les autres *features*).

Une difficulté choisie :

La difficulté doit pouvoir être choisie par le joueur au début de chaque rush pour lui permettre de jouer comme bon lui semble. Plus la difficulté est élevée, plus les récompenses sont élevées. Les récompenses sont sous forme d'or utilisé pour améliorer de façon persistante son héros (voir "De la persistance entre les *rush*"). De plus, seule la difficulté maximum permet de découvrir la fin alternative du jeu.

De la persistance entre les *rush* :

Le joueur doit pouvoir perfectionner son personnage avec des améliorations persistantes entre les *rush*. C'est-à-dire que le joueur recommence, certes depuis le début avec un nouveau héros à chaque nouveau *rush,* mais grâce à ces améliorations, le joueur progresse en puissance de *rush* en *rush*. Cela lui permet de faire des rush plus facilement ou d'augmenter la difficulté jusqu'à pouvoir survivre en difficulté maximale et découvrir la fin alternative du jeu. Cependant, pour permettre à tous les joueurs, y compris les moins bons, de pouvoir accéder à cette fin alternative, celle-ci est aussi visible à partir du moment où toutes les améliorations persistantes ont été débloquées (ce qui implique un certain temps de jeu).

Des combats en tour par tour et de la gestion de ressources :

Lorsque le joueur rencontre un ennemi dans une salle, un combat s'engage. Le joueur gagne de la pression à chaque tour, jusqu'à atteindre un niveau de pression maximum. Chaque action dépense de la pression. Le joueur peut programmer jusqu'à cinq actions à chaque tour. Le héros est composé de cinq parties (tête, torse, jambes, bras droit, bras gauche). Une action est une combinaison d'une de ces parties et d'une direction (haut, bas, gauche, droite). Par exemple, si le joueur programme Jambes - Droite, il se déplace à droite, ou encore Jambes -

Haut, il saute. Le joueur et l'ennemi se déplacent le long d'une ligne de cases donc sur un seul axe. Toute la pression utilisée pendant un combat sera déduite de la réserve de vapeur du héros à la fin du combat, s'il est toujours vivant. Il faudra donc que le joueur utilise judicieusement ses actions afin de ne pas manquer de vapeur. Sinon il sera obligé de convertir son or gagné jusqu'ici en vapeur. Et s'il vient à manquer d'or et de vapeur, le rush se termine tout simplement.

Des graphiques simples mais efficaces et une interface claire et détaillée :

Les graphiques doivent être simples mais efficaces pour ne pas passer trop de temps à les réaliser. Ils pourront faire l'objet d'améliorations si le temps et les ressources le permettent. L'interface doit être claire et détaillée pour permettre au joueur de bien comprendre les mécanismes du jeu. Par exemple, au début d'un rush, après la génération aléatoire du héros, le joueur peut choisir certaines parties de son héros pour les re-déterminer de manière aléatoire. Il faut donc qu'il puisse comprendre et savoir précisément ce que lui apporte chaque partie et ainsi pouvoir agir en combat de manière réfléchie.

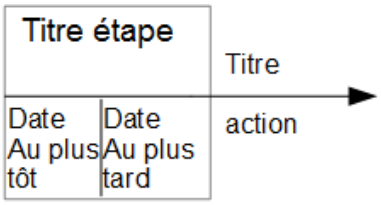
III - Organisation de travail.

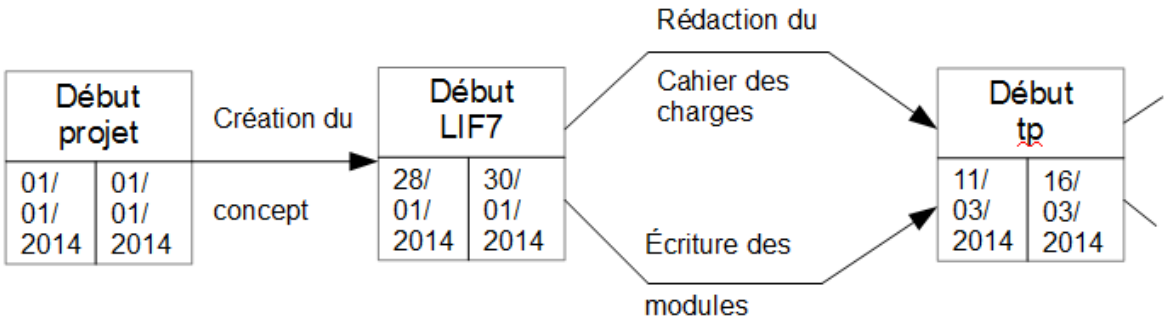
Pierre va s'occuper de la partie graphique du jeu, ce qui implique la programmation de l'affichage ainsi que la création des images du jeu. Damien va se consacrer au noyau du jeu, c'est-à-dire la création des règles du jeu et la programmation du noyau. Bien sûr, une constante communication entre les deux programmeurs est nécessaire pour aboutir à un code cohérent, clair et efficace.

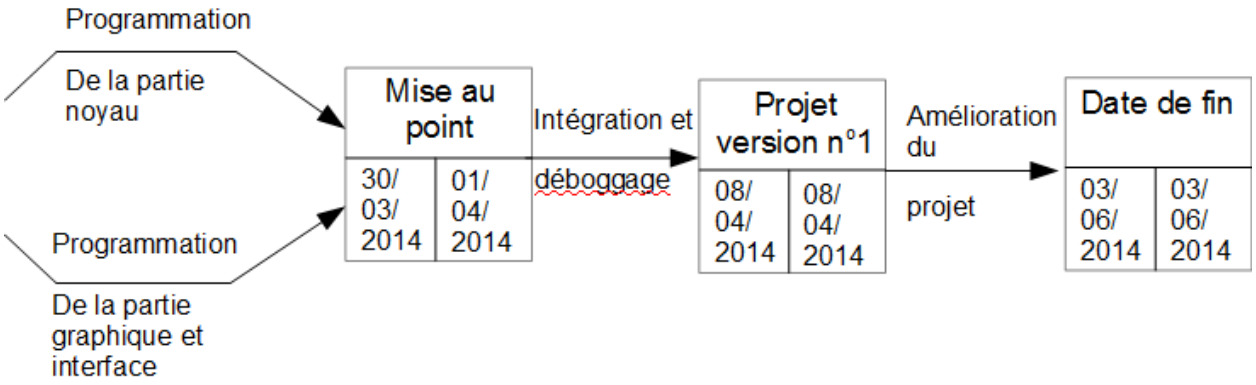
Nous allons commencer par l'essentiel, c'est à dire l'affichage et la gestion d'un personnage, d'un ennemi et d'un rush. Nous ajouterons ensuite un menu au jeu, puis les composantes aléatoires (création du personnage, des ennemis, des salles). Ensuite nous ajouterons les améliorations persistantes et les options. Et s'il nous reste du temps nous améliorerons l'aléatoire du jeu, l'IA (Intelligence Artificielle), l'interface et les graphismes.

Voici le PERT de notre projet :

Rappel d'un PERT :



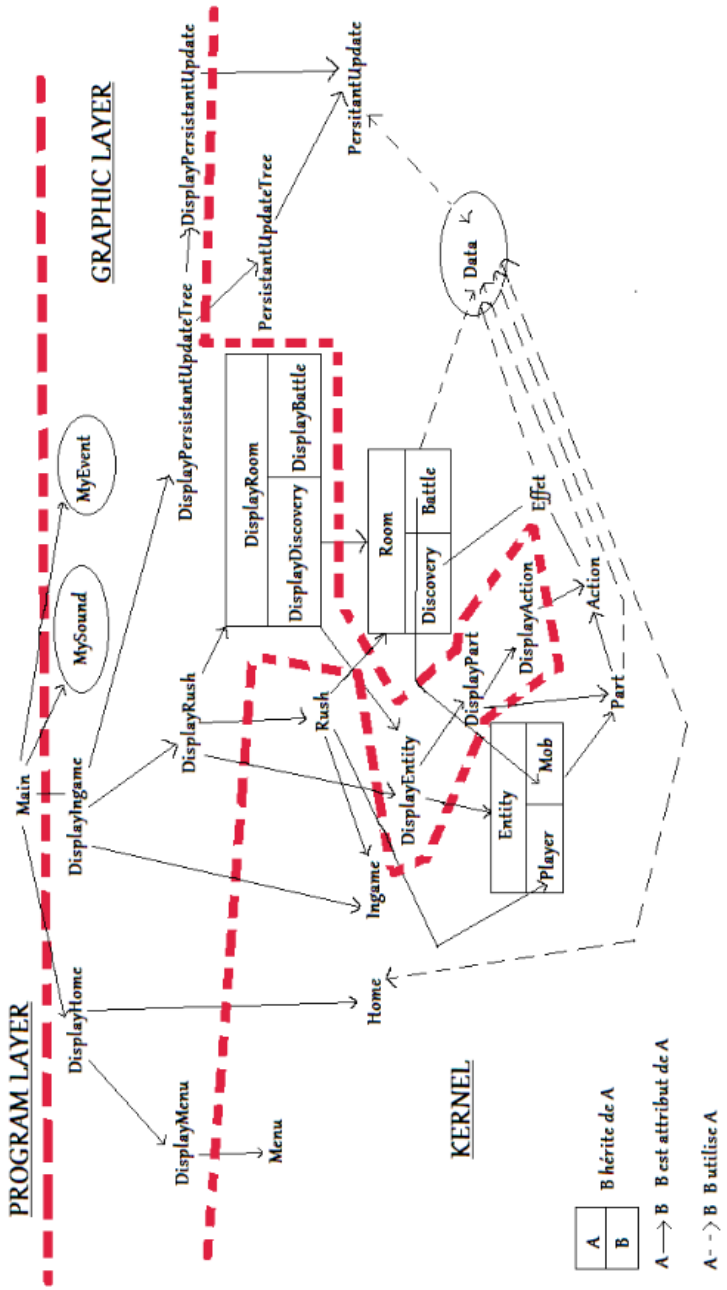




IV - Architecture du code.

Nous avons divisé le programme en deux parties : noyau et interface. Ainsi le corps du jeu est totalement indépendant de la manière dont nous allons l'afficher à l'utilisateur.

Le diagramme de dépendance actuel :



La liste des modules actuels :

Core :

Module Menu
Module Home
Module Ingame
Module Rush
Module Entity
    |_____Module Player
    |_____Module Mob
Module Part
Module Action
Module Effect
Module Room
    |_____Module Discovery
    |_____Module Battle
Module PersistantUpgradeTree
Module PersistantUpgrade
Module Data

Interface :

Module MySound
Module MyEvent
Module DisplayMenu
Module DisplayHome
Module DisplayIngame
Module DisplayRush
Module DisplayEntity
Module DisplayPart
Module DisplayAction
Module DisplayRoom
    |_____Module DisplayDiscovery
    |_____Module DisplayBattle
Module DisplayPersistantUpgradeTree
Module DisplayPersistantUpgrade

Module Menu

Class Menu

- vector<void*> _fonctions

+ Menu()
+ ~Menu()
+ void* GetFonction(unsigned short int i) const
+ void SetFonction(const void* value, unsigned short int i)

Module Home

Class Home

- Menu _menu
- string _pathBackground

+ Home()
+ ~Home()
+ Menu GetMenu() const
+ string GetPathBackground() const
+ void SetMenu(const Menu value)
+ void SetPathBackground(const string value)

Module Ingame

Class Ingame

- Rush _rush
- listLocalisation _localisation

+ Ingame()
+ ~Ingame()
+ Rush GetRush() const
+ listLocalisation GetLocalisation() const
+ void SetRush(const Rush value)
+ void SetLocalisation(const listLocalisation value)

Enumération

enum{PERSISTANTUpgradeTREE=0, RUSH=1}

Module Rush

Class Rush

- unsigned short int _difficulty
- Player _player
- Room* _CurrentRoom
- int _stockOre
- short int _stockVestige
- unsigned short int _maxDifficulty
- string _pathBackground

+ Rush()
+ ~Rush()
+ unsigned short int GetDifficulty() const
+ Player GetPlayer() const
+ Room* GetRomm(const unsigned short int i) const
+ int GetStockOre() const
+ short int GetStockVestige() const
+ unsigned short int GetMaxDifficulty() const
+ string GetPathBackground() const
+ void SetDifficulty(const unsigned short int value)
+ void SetPlayer(const Player value)
+ void SetRoom(const unsigned short int i, const Room* value)
+ void SetStockOre(const int value)
+ void SetStockVestige(const short int value)
+ void SetMaxdifficulty(const unsigned short int value)
+ void SetPathBackground(const string value)

Module Entity

Abstract Class Entity

- ± int _integrity
- ± short int _armor
- ± short int _valueTmpArmorModifier
- ± vector<Part*> _parts
- ± listTypeEntity _typeEntity
- ± bool _tmpArmorModifier
- ± bool _startDelayedAction
- ± bool _endDelayedAction
- ± Action* _tmpAction
- ± short int _tmpArmorModifierValue

- + Entity()
- + ~Entity()
- + int GetIntegrity() const
- + short int GetArmor() const
- + short int GetValueTmpArmorModifier() const
- + Part* GetPart(const unsigned short int i) const
- + listTypeEntity GetTypeEntity() const
- + bool GetTmpArmorModifier() const
- + bool GetStartDelayedAction() const
- + bool GetEndDelayedAction() const
- + short int GetTmpArmorModifierValue() const
- + Action* GetTmpAction() const
- + void SetIntegrity(const int value)
- + void SetArmor(const short int value)
- + void SetValueTmpArmorModifier(const short int value)
- + void SetPart(const unsigned short int i, const Part* value)
- + void SetTypeEntity(const listTypeEntity value)
- + void SetTmpArmorModifier(const bool value)
- + void SetStartDelayedAction(const bool value)
- + void SetEndDelayedAction(const bool value)
- + void SetTmpArmorModifierValue(const short int value)
- + void SetTmpAction(const Action* value)
- + void TakeDamage(const unsigned int damage, const bool armorIgnored)
- + void TakeHeal(const unsigned int heal)
- + void DoAction(const string part, const string direction, Entity* enemy)
- + void DoEndTurnAction(Entity* enemy)

Module Player

Class Player : Entity

- short int _pressurePerTurn
- int _steam
- int _maxPressure

+ Player(const int integrity, const short int armor, vector<Part*> parts, const short int pressurePerTurn, const int steam, const int maxPressure)
+ ~Player()
+ short int GetPressurePerTurn() const
+ int GetSteam() const
+ int GetMaxPressure() const
+ void SetPressurePerTurn(const short int value)
+ void SetSteam(const int value)
+ void SetMaxPressure(const int value)

Module Mob

Class Mob : Entity

+ Mob(const int integrity, const short int armor, const vector<Part*> parts)
+ ~Mob()
+ void DoTurn(Entity* enemy)

Module Part

Class Part

- string _type
- string _pathSprite
- vector<Action> _actions
- string _name

+ Part()
+ ~Part()
+ string GetType() const
+ string GetPathSprite() const
+ Action GetAction(const unsigned short int i) const
+ string GetName() const
+ void SetType(const string value)
+ void SetPathSprite(const string value)
+ void SetAction(const unsigned short int i, const Action value)
+ void SetName(const string value)
+ void LoadPart()

Module Action

Class Action

- unsigned short int _pressureCost
- unsigned short int _lineSprite
- vector<Effect> _effects
- string _flag
- listDirection direction

+ Action()
+ ~Action()
+ unsigned short int GetPressureCost() const
+ unsigned short int GetLineSprite() const
+ Effect GetEffect(const unsigned short int i)
+ string GetFlag() const
+ listDirection GetDirection() const
+ void SetPressureCost(const unsigned short int value)
+ void SetLineSprite(const unsigned short int value)
+ void SetEffect(const unsigned short int i, const Effect value)
+ void SetFlag(const string value)
+ void SetDirection(const listDirection value)
+ void DoAction()
+ void LoadAction()

Enumération

enum listDirection{UP=0, DOWN=1, LEFT=2, RIGHT=3}

Module Effect

Class Effect

- int _value
- string _typeEffect

+ Effect()
+ ~Effect()
+ int GetValue() const
+ string GetTypeEffect() const
+ void SetValue(const int value)
+ void SetTypeEffect(const string value)
+ void LoadEffect()

Module Room

Abstract Class Room

- ± string _background
- ± string _type

- \+ Room()
- \+ ~Room()
- \+ string GetBackground() const
- \+ string GetType() const
- \+ void SetBackground(const string value)
- \+ void SetType(const string value)

Module Discovery

Class Discovery : Room

- \- Effect _effect
- \- string _pathSprite

- \+ Discovery()
- \+ ~Discovery()
- \+ Effect GetEffect() const
- \+ string GetPathSprite() const
- \+ void SetEffect(const Effect value)
- \+ void SetPathSprite(const string value)

Module Battle

Class Battle : Room

- \- Entity _enemy

- \+ Battle()
- \+ ~Battle()
- \+ Entity GetEnemy() const
- \+ void SetEnemy(const Entity value)

Module PersistantUpgradeTree

Class PersistantUpgradeTree

- vector<PersistantUpgrade> _persistantUpgrades
- int _stockOre
- unsigned int _countNewUpgrade

+ PersistantUpgradeTree()
+ ~PersistantUpgradeTree
+ PersistantUpgrade GetPersistantUpgrade(const unsigned short int i) const
+ int GetStockOre() const
+ unsigned int GetCountNewUpgrade() const
+ void SetPersistantUpgrade(const PersistantUpgrade value, const unsigned short int i)
+ void SetStockOre(const int value)
+ void SetCountNewUpgrade(const unsigned int value)
+ void SavePersitantUpgradeTree() const

Module PersistantUpgrade

Class PersistantUpgrade

- unsigned short int _id
- unsigned short int _level
- unsigned short int _maxLevel
- int _cost
- vector<Part> _parts
- string _type
- string _name
- string _description

+ PersistantUpgrade(const unsigned short int i)
+ ~PersistantUpgrade()
+ unsigned short int GetId() const
+ unsigned short int GetLevel() const
+ unsigned short int GetMaxLevel() const
+ int GetCost() const
+ Part* GetPart(const unsigned short int i) const
+ string GetType() const
+ string GetName() const
+ string GetDescription() const
+ void SetId(const unsigned short int value)
+ void SetLevel(const unsigned short int value)
+ void SetMaxLevel(const unsigned short int value)
+ void SetCost(int value)
+ void SetPart(const Part* value, const unsigned short int i)

+ void SetType(const string value)
+ void SetName(const string value)
+ void SetDescription(const string value)
+ void SavePersistantUpgrade() const
+ void UpgradePersistantUpgrade()

Module Data

Class Data

+ Data()
+ ~Data()
+ string Read(const string type, string keys)
+ void Write(const string type, string keys, const string value)

Module MySound

Class MySound

Module MyEvent

Class MyEvent

Module DisplayMenu

Class DisplayMenu

- Menu* _menu
- sf::Sprite* _sprite
- int _x
- int _y

+ DisplayMenu()
+ ~DisplayMenu()
+ Menu* GetMenu() const
+ sf::Sprite GetSprite() const
+ int GetX() const
+ int GetY() const
+ void SetMenu(const Menu* value)
+ void SetSprite(const sf::Sprite* value, const int x, const int y)
+ void SetX(const int value)
+ void SetY(const int value)

+ void DisplayMenuOn(sf::RenderWindow* window) const

Module DisplayHome

Class DisplayHome

- Home* _home
- DisplayMenu* _displayMenu
- sf::Sprite* _background

+ DisplayHome()
+ ~DisplayHome()
+ Home* GetHome() const
+ DisplayMenu* GetDisplayMenu() const
+ sf::Sprite* GetBackground() const
+ void SetHome(const Home* value)
+ void SetDisplayMenu(const DisplayMenu* value)
+ void SetBackground(const sf::Sprite* value)
+ void DisplayHomeOn(sf::RenderWindow* window) const

Module DisplayIngame

Class DisplayIngame

- Ingame* _ingame
- DisplayRush* _displayRush
- DisplayPersistantUpgradeTree* _displayPersistantUpgradeTree

+ DisplayIngame()
+ ~DisplayIngame()
+ Ingame* GetIngame() const
+ DisplayRush* GetDisplayRush() const
+ DisplayPersistantUpgradeTree* GetDisplayPersistantUpgradeTree() const
+ void SetIngame(const Ingame* value)
+ void SetDisplayRush(const DisplayRush* value)
+ void SetDisplayPersistantUpgradeTree(const DisplayPersistantUpgradeTree* value)
+ void DisplayIngameOn(sf::RenderWindow* window) const

Module DisplayRush

Class DisplayRush

- Rush* _rush
- sf::Sprite* _background
- vector<DisplayRoom*> _displayRooms
- DisplayPlayer* _ displayPlayer
- short int _current room

+ DisplayRush()
+ ~DisplayRush()
+ Rush* GetRush() const
+ sf::Sprite* GetBackground() const
+ DisplayRoom* GetDisplayRoom(const unsigned short int i) const
+ DisplayPlayer* GetDisplayPlayer() const
+ short int GetCurrentRoom() const
+ void SetRush(const Rush* value)
+ void SetBackground(const sf::Sprite* value)
+ void SetDisplayRoom(const unsigned short int i, const DisplayRoom* value)
+ void SetDisplayPlayer(const DisplayPlayer* value)
+ void SetCurrentRoom(const short int value)
+ void DisplayRushOn(sf::RenderWindow* window)

Module DisplayEntity

Class DisplayEntity

- Entity* _entity
- sf::Sprite _sprite
- int _x
- int _y
- vector<DisplayPart*> _displayParts

+ DisplayEntity()
+ ~DisplayEntity()
+ Entity* GetEntity() const
+ sf::Sprite* GetSprite() const
+ int GetX() const
+ int GetY() const
+ DisplayPart* GetDisplayPart(const unsigned short int i) const
+ void SetEntity(const Entity* Entity)
+ void SetSprite(const Image* image)
+ void SetX(const int x)
+ void SetY(const int y)
+ void SetDisplayPart(const unsigned short int i, const DisplayPart* displayPart)

+ void CreateImage()
+ void DisplayImageOn(sf::RenderWindow* window) const

Module DisplayPart

Class DisplayPart

- Part* _part
- sf::Sprite* _sprite
- vector<DisplayAction*> _displayActions

+ DisplayPart(const string path, const Part* part)
+ ~DisplayPart()
+ Part* GetPart() const
+ sf::Sprite* GetSprite() const
+ DisplayAction* GetDisplayAction(const unsigned short int i) const
+ void SetPart(const Part* part)
+ void SetSprite(const sf::Sprite* image)
+ void SetDisplayAction(const DisplayAction* displayAction)
+ void DisplayPartOn(sf::RenderWindow* window) const

Module DisplayAction

Class DisplayAction

- Action* _action
- sf::Sprite* _sprite

+ DisplayAction(const string path, const Action* action)
+ ~DisplayAction()
+ Action* GetAction() const
+ sf::Sprite* GetSprite() const
+ void SetAction(const Action* action)
+ void SetSprite(const sf::Sprite* image)
+ void DisplayActionOn(sf::RenderWindow* window) const

Module DisplayRoom

Class DisplayRoom

± Room* _room
± sf::Sprite* _background

+ DisplayRoom()
+ ~DisplayRoom()
+ Room* GetRoom() const
+ sf::Sprite* GetBackground() const

+ void SetRoom(const Room* value)
+ void SetBackground(const sf::Sprite* value)
+ void DisplayRoomOn(sf::RenderWindow* window)

Module DisplayDiscovery

Class DisplayDiscovery : DisplayRoom

- Discovery* _discovery
- sf::Sprite* _sprite

+ DisplayDiscovery()
+ ~DisplayDiscovery()
+ Discovery* GetDiscovery() const
+ sf::Sprite* GetSprite() const
+ void SetDiscovery(const Discovery* value)
+ void SetSprite(const sf::Sprite* value)
+ void DisplayDiscoveryOn(sf::RenderWindow* window) const

Module DisplayBattle

Class DisplayBattle : DisplayRoom

- Battle* _battle
- DisplayEntity* _displayEntity

+ DisplayBattle()
+ ~DisplayBattle()
+ Battle* GetBattle() const
+ DisplayEntity* GetDisplayEntity() const
+ void SetBattle(const Battle* value)
+ void SetDisplayEntity(const DisplayEntity* value)
+ void DisplayBattleOn(sf::RenderWindow* window) const

Module DisplayPersistantUpgradeTree

Class DisplayPersistantUpgradeTree

- vector<DisplayPersistantUpgrade*> _displayPersistantUpgrades
- PersistantUpgradeTree* _persistantUpgradeTree
- sf::Sprite* _sprite

+ DisplayPersistantUpgradeTree(const string path)
+ ~DisplayPersistantUpgradeTree()
+ DisplayPersistantUpgrade* GetDisplayPersistantUpgrade(const unsigned short int i) const
+ PersistantUpgradeTree* GetPersistantUpgradeTree() const

+ sf::Sprite* GetSprite() const
+ void SetDisplayPersistantUpgrade(const DisplayPersistantUpgrade* displayPersistantUpgrade,
const unsigned short int i)
+ void SetPersistantUpgradeTree(const PersistantUpgradeTree* persistantUpgradeTree)
+ void SetSprite(const sf::Sprite* image)
+ void DisplayPersistantUpgradeTreeOn(sf::RenderWindow* window) const

Module DisplayPersistantUpgrade

Class DisplayPersistantUpgrade

- PersistantUpgrade* _persistantUpgrade
- sf::Text* _text
- sf::Sprite* _sprite
- int _x
- int _y

+ DisplayPersistantUpgrade(const string path, PersistantUpgrade* persitantUpgrade, const int x, const int y)
+ ~DisplayPersistantUpgrade()
+ PersistantUpgrade* GetPersistantUpgrade() const
+ sf::Text* GetText() const
+ sf::Sprite* GetSprite() const
+ int GetX() const
+ int GetY() const
+ void SetPersistantUpgrade(const PersistantUpgrade* persistantUpgrade)
+ void SetText(const sf::Text* text)
+ void SetSprite(const sf::Sprite* image)
+ void SetX(const int x)
+ void SetY(const int y)
+ void DisplayPersistantUpgrade(sf::RenderWindow* window) const