Ingénierie des systèmes orientés-objet

Laboratoire 1
Jeu DomeSupremacy

Table des matières

Introduction

Ce premier laboratoire consiste à réaliser un programme informatique implémentant un jeu simple.

Afin d'assurer la mise en pratique des notions introductives liées au paradigme orientée objet, vous serez fortement guidé vers l'implémentation à réaliser.

Objectifs

Les objectifs pédagogiques de ce laboratoire sont énumérés en ordre décroissant d'importance:

- Prendre contact avec le paradigme de la programmation orienté objet.
- Apprendre et mettre en pratique la syntaxe et la sémantique du langage `C++`.
- Autant conceptuellement que pour le développement en langage `C++`, comprendre :
    - la notion d'encapsulation (regroupement et intégrité)
    - faire les liens et comprendre les différences entre :
        - abstraction : concept
        - développement : classe
        - usage : objet
- Être capable de comprendre un schéma de conception existant :
    - faire les liens entre le jeu lui-même et la conception imposée (abstraction)
    - faire les liens entre la conception et le code à produire dans le contexte du langage `C++`: classes, objets et associations
- Apprécier l'utilisation d'une bibliothèque simplifiant la mise en place d'un tel projet : `EzGame`.
- Se familiariser avec l'environnement de développement Microsoft Visual Studio autant pour la rédaction de code que pour le débogage.
- Mettre en application une norme de codage et les bonnes pratiques associées.

Résumé du projet

On vous demande de réaliser un logiciel implémentant le jeu `DomeSupremacy`[1] : un jeu où deux joueurs accumulent des points en protégeant un dôme.

La conception logicielle vous est entièrement imposée. Le design à réaliser exploite pleinement la notion d'encapsulation du paradigme orienté objet. Vous pourrez ainsi développer votre sens critique sur cette approche du développement logiciel tout en vous familiarisant aux éléments techniques du langage `C++`.

L'utilisation de la bibliothèque `EzGame` permettra le développement avec une interface utilisateur graphique.

Les concepts de l'encapsulation sont particulièrement visibles dans ces deux aspects du projet :

a. Développement de `DomeSupremacy` : implémentation
b. Emploi de `EzGame` : utilisation

[1] Concept de jeu par François Côté-Raiche.

Présentation du jeu

Introduction

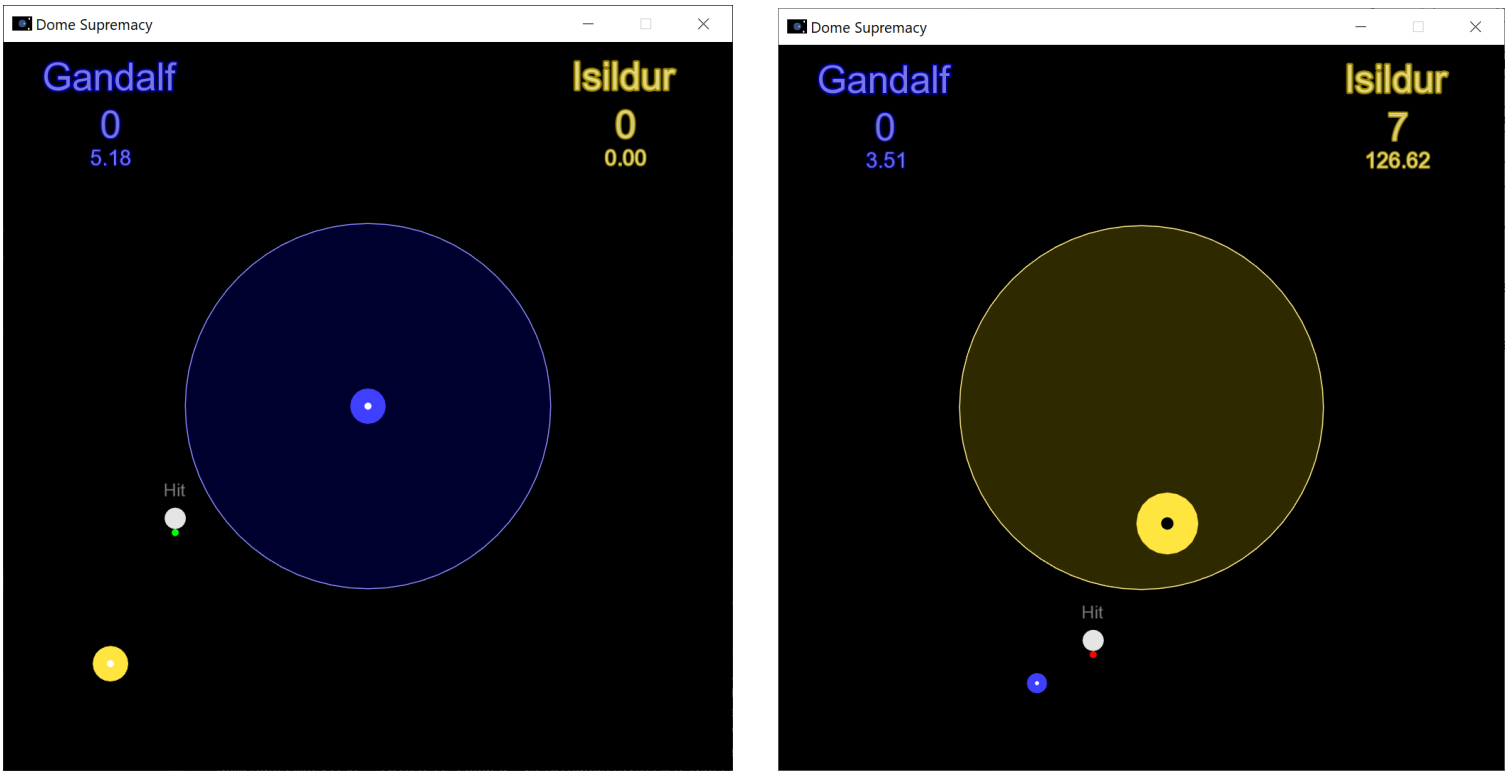
Le jeu `DomeSupremacy` consiste à accumuler un maximum de points en défendant le dôme convoité.

Les points saillants du jeu sont :

- Au centre de l'arène du jeu se trouve un dôme.
- Deux joueurs sont présents et chacun a un rôle :
  - le défenseur est le joueur qui défend le dôme
  - le prétendant est celui qui désire avoir le rôle de défenseur
- Un *match* consiste à faire un point (sans le modificateur) :
  - le défenseur obtient un point s'il touche le prétendant
  - le prétendant obtient un point s'il touche le dôme
- Lorsqu'un point de *match* est fait :
  - par le défenseur, les positions sont réassignées
  - par le prétendant, les rôles sont inversés et les positions sont réassignées
- Un modificateur est présent en tout temps sur l'arène :
  - Lorsqu'un joueur touche au modificateur, une modification est apportée à un joueur.
  - Un double système permet de déterminer l'impact de la modification :
    - type : bonus ou malus
      - bonus, favorise le joueur qui a attrapé le modificateur
      - malus, défavorise le joueur adverse
    - effet : la modification
      - vitesse : accélère ou ralenti le joueur
      - taille : augmente ou réduit la taille du joueur
      - bordures : téléportation ou bordures infranchissables
      - point de *match* : donne ou retire un point de *match*
- Le positionnement des éléments est déterminé ainsi :
  - dôme, au centre de l'arène
  - défenseur :
    - la position initiale est au centre du dôme
    - il peut se déplacer selon sa stratégie de gestion de la bordure
  - prétendant
    - la position initiale est déterminée aléatoirement sur le pourtour d'un cercle imaginaire centré sur le dôme avec un rayon de la moitié de la plus petite taille de l'arène
    - il peut se déplacer selon sa stratégie de gestion de la bordure
  - modificateur
    - la position initiale est déterminée aléatoirement à l'intérieur d'un anneau imaginaire centré sur le dôme avec un rayon intérieur équivalent au dôme et un rayon extérieur de la moitié de la plus petite taille de l'arène
- La partie évolue avec deux systèmes de pointage :
  - pointage de *match*
  - temps passé en tant que défenseur

    - ce double système permet d'avoir des objectifs différents selon les parties
- Le jeu se fait entièrement à l'aide du clavier avec les touches suivantes :
    - la touche `échappement` : le programme quitte (immédiatement)
    - la touche `entrer` : réinitialise le jeu (pointages, rôles et positions)
    - quatre touches personnalisées sont utilisées par chaque joueur :
        - déterminent le mouvement des joueurs : haut, gauche, bas et droit
        - ces touches sont à définir à votre guise selon votre clavier, par exemple : `WASD` et `IJKL`
- Un mode de gestion des bordures permet de définir le comportement lorsqu'un joueur dépasse les limites de l'arène :
    - restriction : la position du joueur est limitée par la taille de l'arène
    - téléportation : la position du joueur est téléportée du côté opposé de l'arène

Captures d'écran



Captures d'écran d'une partie – À gauche, le début d'une partie – À droite, l'avancement d'une partie

Conception orientée objet

Plusieurs diagrammes UML présentent la conception imposée et vous permettront de réaliser techniquement le projet.

Le projet est divisé en sept classes où chacune possède des responsabilités clairement identifiées. Les sections suivantes vous présentent chacune de ces classes en détail.

Vue d'ensemble

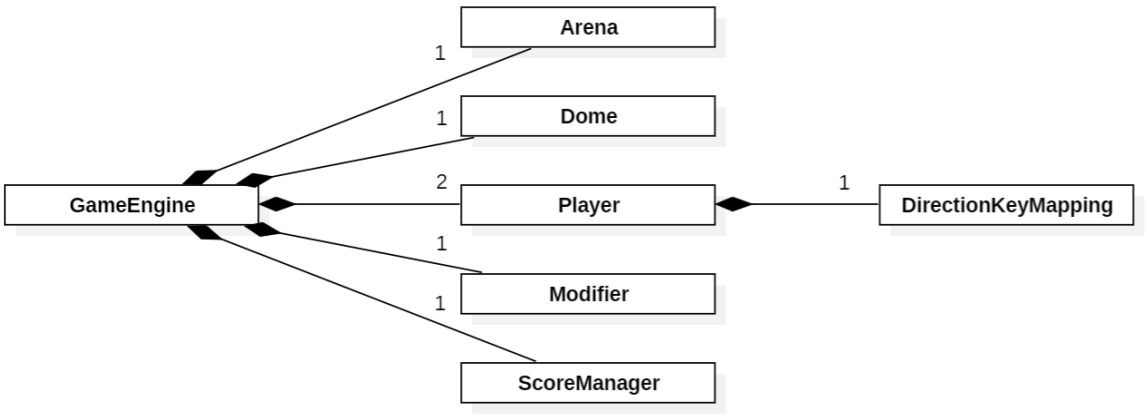


Diagramme de classe : Vue d'ensemble

Ce diagramme de classe présente une vue d'ensemble du projet et des relations existantes en chacune des classes.

Le détail des classes est présenté dans d'autres diagrammes.

Classe Arena

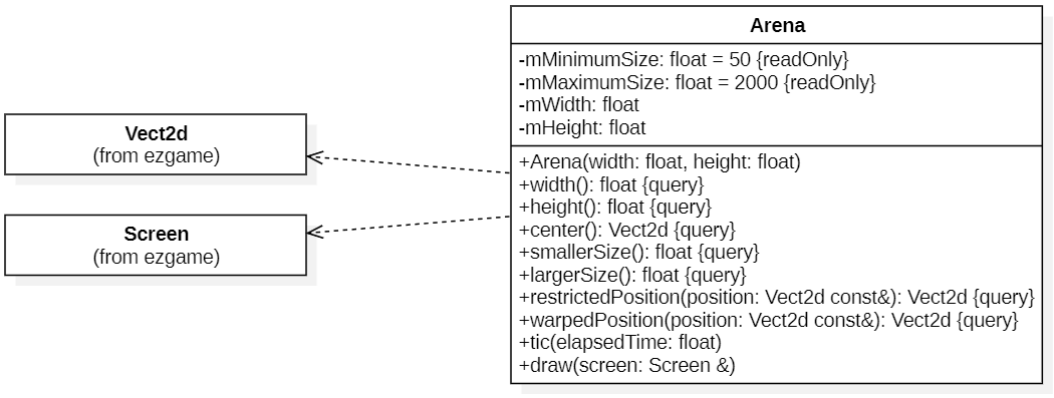


Diagramme de classe : Arena

Description :

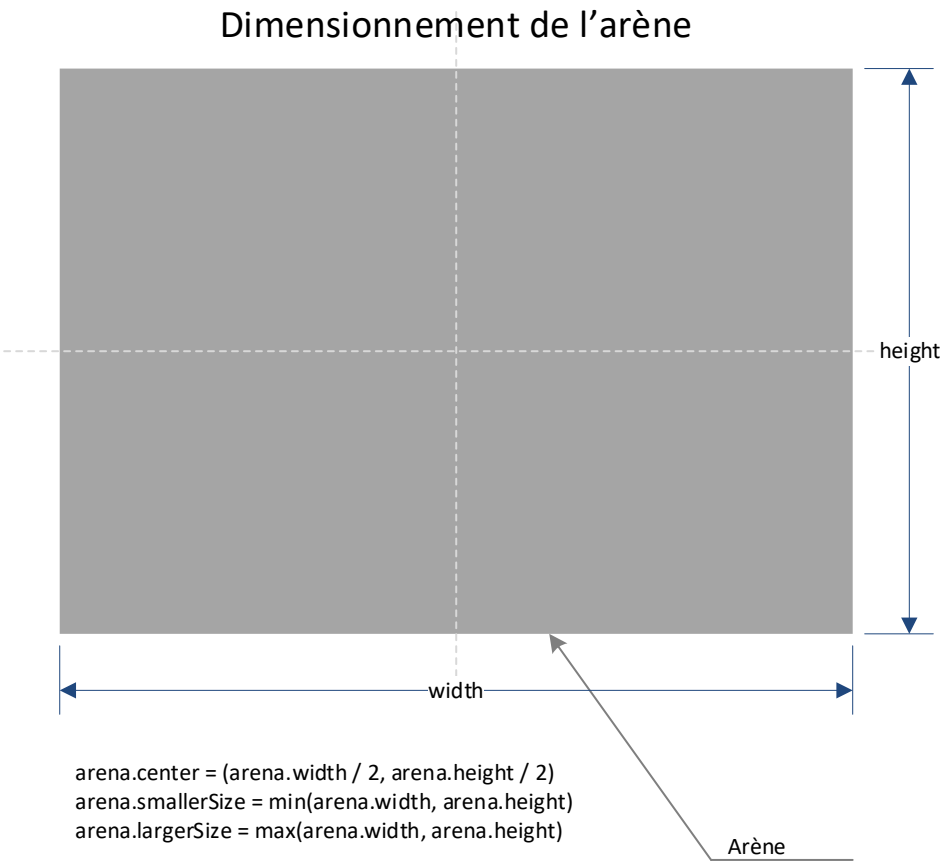
- Concept :
  - représente la zone de jeu
- Rôles :
  - détermine la dimension de la zone de jeu, par conséquent la taille de l'application (voir autres informations plus bas)
  - permet de déterminer la position du dôme (au centre)
  - facilite la gestion des bordures pour les joueurs
  - affiche le fond la scène
- Représentation graphique :
  - correspond au fond de l'arène et, conséquemment, de l'application graphique (voir autres informations plus bas).
  - le fond est noir (possibilité de faire des effets en option)
- Autres informations :
  - la taille de l'application est celle de l'arène

Détails sur les opérations :

- `Arena(width, height)`
  - Constructeur initialisant, dans l'ordre déclaré, les constantes et variables.
  - Les valeurs `mWidtht` et `mHeight` sont limitées dans l'intervalle [`mMinimumSize`, `mMaximumSize`].
- `width()`
  - Accesseur retournant la taille horizontale de l'arène.
- `height()`
  - Accesseur retournant la taille verticale de l'arène.
- `center()`
  - Accesseur retournant la position centrale de l'arène.
- `smallerSize()`
  - Accesseur retournant la plus petite dimension de l'arène.
- `largerSize()`
  - Accesseur retournant la plus grande dimension de l'arène.
- `restrictedPosition(position)`

- Fonction utilitaire retournant une position valide dans l'arène.
- Si la position est valide (à l'intérieur de l'arène), la même valeur est retournée.
- Si la position est invalide, retourne la position se trouvant à la limite de l'arène du côté où elle dépasse.

- `warpedPosition(position)`
  - Fonction utilitaire retournant une position valide dans l'arène.
  - Si la position est valide (à l'intérieur de l'arène), la même valeur est retournée.
  - Si la position est invalide, retourne la position se trouvant du côté opposé de l'arène où elle dépasse.
- `tic(elapsedTime)`
  - Fonction utilitaire gérant la progression de la simulation.
  - Ne fait rien.
  - Cette fonction est présente dans ce design par souci d'uniformité et pour faciliter d'éventuels ajouts.
- `draw(screen)`
  - Dessine l'arène.
  - Puisque c'est l'arène qui est dessinée en premier, on efface toute la surface graphique par une couleur uniforme (noire).



Représentation de l'arène

Classe Dome

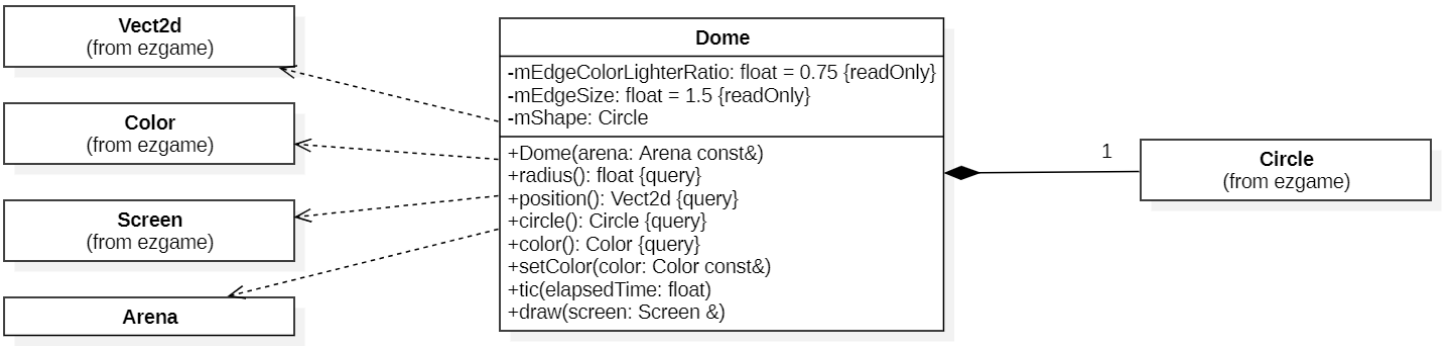


Diagramme de classe : Dome

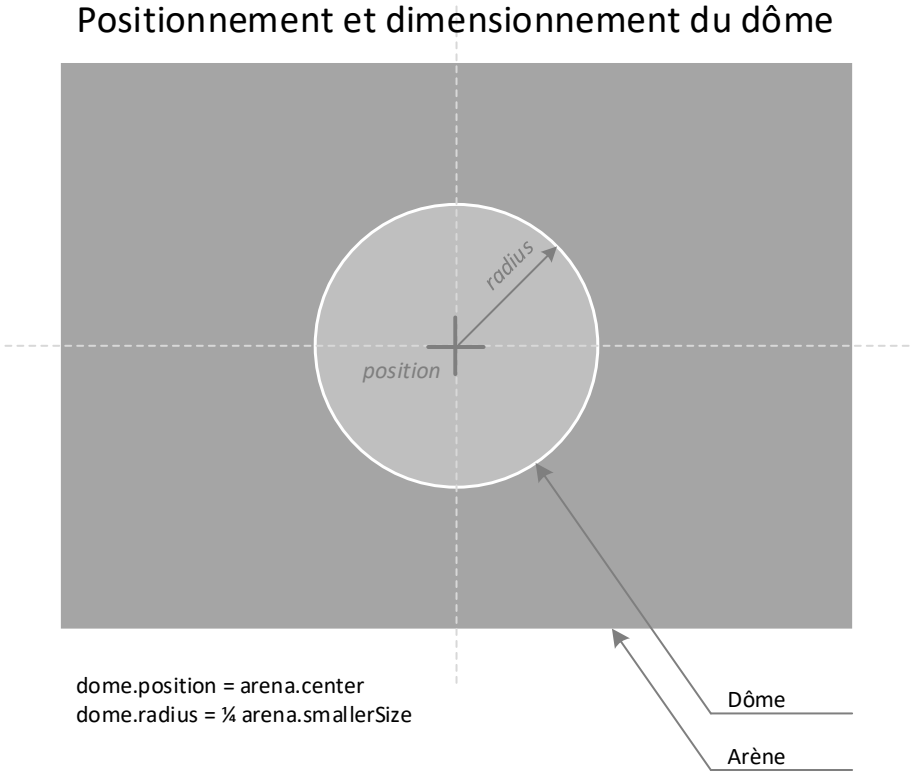
Description :

- Concept :
  - représente la dôme central
- Rôles :
  - permet de déterminer si le prétendant touche le dôme et si il y a une permutation de rôle
  - permet de déterminer les positionnements des joueurs et du modificateur
- Représentation graphique :
  - correspond au fond de l'application graphique.
  - le fond noir
- Autres informations:
  - -

Détails sur les opérations :

- `Dome(arena)`
  - Constructeur initialisant, dans l'ordre déclaré, les constantes et variables.
  - Le cercle est :
    - positionné au centre de l'arène
    - avec un rayon du quart de la taille le plus petite de l'arène
    - deux couleurs arbitraires (remplissage et contour) car elles seront initialisées plus tard par celle la couleur du joueur
    - la taille de la bordure est initialisée à `mEdgeSize`
- `radius()`
  - Accesseur retournant le rayon du dôme.
- `position()`
  - Accesseur retournant la position centrale du dôme.
- `circle()`
  - Accesseur retournant l'objet mShape.
- `color()`
  - Accesseur retournant la couleur de remplissage du cercle.
- `setColor(color)`
  - Mutateur modifiant la couleur du cercle :
  - la couleur de remplissage est celle donnée

    - la couleur de la bordure est celle donnée éclaircie du ratio `mEdgeColorLighterRatio`
- `tic(elapsedTime)`
    - Fonction utilitaire gérant la progression de la simulation.
    - Ne fait rien.
    - Cette fonction est présente dans ce design par souci d'uniformité et pour faciliter d'éventuels ajouts.
- `draw(screen)`
    - Dessine le dôme, l'objet `mCircle`.



Représentation et positionnement du dôme

Classe DirectionKeyMapping

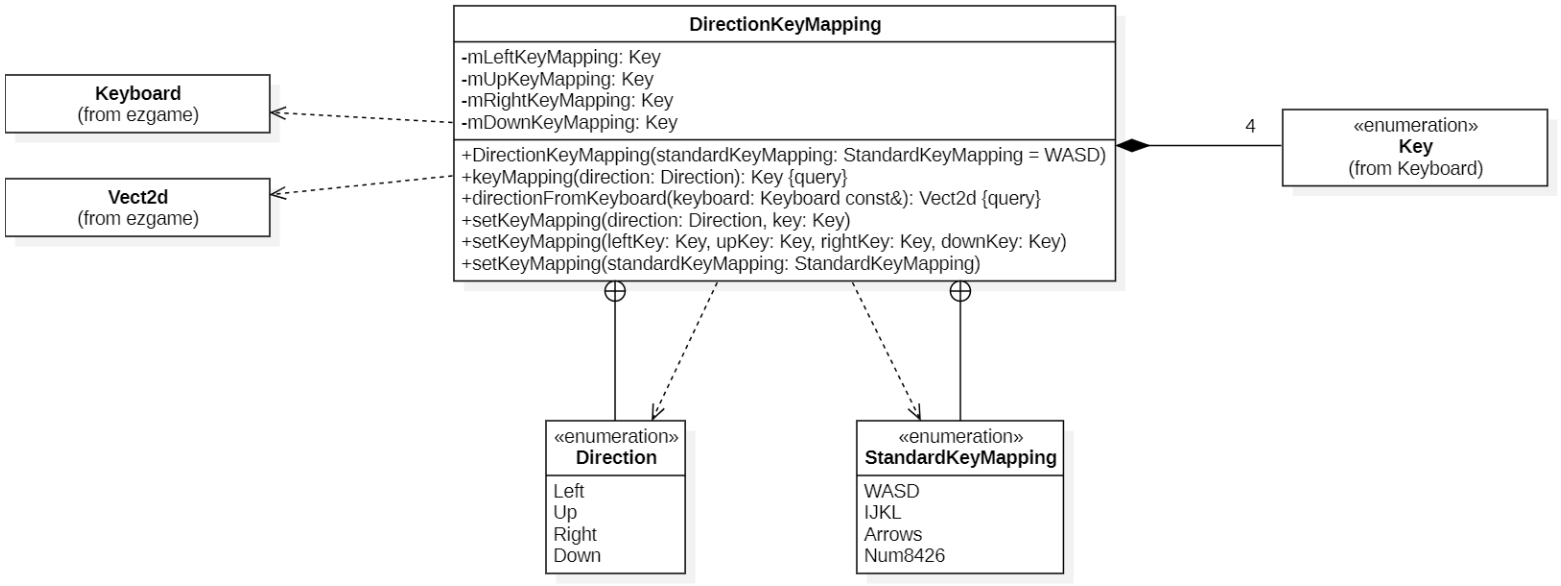


Diagramme de classe : DirectionKeyMapping

Description :

- Concept :
  - classe utilitaire simplifiant la relation entre la gestion des touches au clavier et une direction donnée
- Rôles :
  - permet d'établir les touches associées aux quatre directions cardinales
  - détermine le vecteur directeur direction d'un si le prétendant touche le dôme et si il y a une permutation de rôle
  - permet de déterminer la direction selon l'état du clavier
- Représentation graphique :
  - aucune
- Autres informations :
  - la direction retournée est un vecteur unitaire
- Sous-types :
  - `Direction` : énumération représentant les 4 directions
  - `StandardMapping` : énumération représentant 4 correspondances couramment utilisées pour déterminer la direction à l'aide du clavier.

Détails sur les opérations :

- `DirectionKeyMapping(standardKeyMapping)`
  - Constructeur initialisant les variables.
  - Redirige les assignations au mutateur `setKeyMapping(StandardKeyMapping)`
- `keyMapping(direction)`
  - Accesseur retournant la touche du clavier utilisée pour correspondre à la direction donnée.
- `directionFromKeyboard(keyboard)`

    - Fonction utilitaire retournant la direction selon l'état du clavier.
    - Cette fonction établit quelles touches sont appuyées parmi les quatre utilisées afin de déterminer la direction.
    - La direction retournée est :
        - un vecteur nul si aucune touche n'est appuyée : (0, 0)
        - un vecteur unitaire si au moins une touche est appuyée
- `setKeyMapping(direction, key)`
    - Mutateur assignant la touche `key` à la direction `direction`.
- `setKeyMapping(leftKey, upKey, rightKey, downKey)`
    - Mutateur assignant les quatre touches données aux quatre directions.
    - Cette surcharge de fonction appelle quatre fois la fonction `setKeyMapping(Direction, Key)`.
- `setKeyMapping(standardKeyMapping)`
    - Mutateur assignant les touches couramment utilisées.
    - Cette surcharge de fonction appelle la fonction `setKeyMapping(Key, Key, Key, Key)` selon l'énumération donnée.

Note importante

Attention au problème de « *n-key rollover* »!

Le terme « *n-key rollover* » désigne la capacité d'un clavier à enregistrer simultanément un certain nombre de touches pressées. Ce concept est crucial pour garantir que toutes les entrées sont correctement transmises au système informatique.

Le problème n'est pas de nature logicielle, mais plutôt liée aux capacités matérielles du clavier présent. Un clavier haut de gamme peut offrir la capacité d'enregistrer l'appui simultané de toutes les touches du clavier. Un clavier moins haut de gamme peut être limité au nombre de touches pouvant être détectées en même temps. Cette limitation se manifeste autant par certaines combinaisons de touches spécifiques que par le nombre de touches.

Lorsque le matériel ne possède pas une capacité de « *n-key rollover* » adéquate, il peut y avoir des omissions ou des confusions dans la transmission des signaux de touches au système informatique. Cette situation peut rendre l'application impossible à correctement opérer si l'appui de touches simultanées est nécessaire.

Il est essentiel de tester cette fonctionnalité pour assurer la fiabilité du clavier, surtout dans des contextes nécessitant des combinaisons de touches complexes. Les tests peuvent être effectués via des logiciels dédiés ou par des méthodes empiriques, où un utilisateur presse plusieurs touches simultanément pour évaluer la performance du clavier.

Voir ce lien pour faire un test manuel des possibilités du clavier concernant la situation du « *n-key rollover* ».

Classe Player

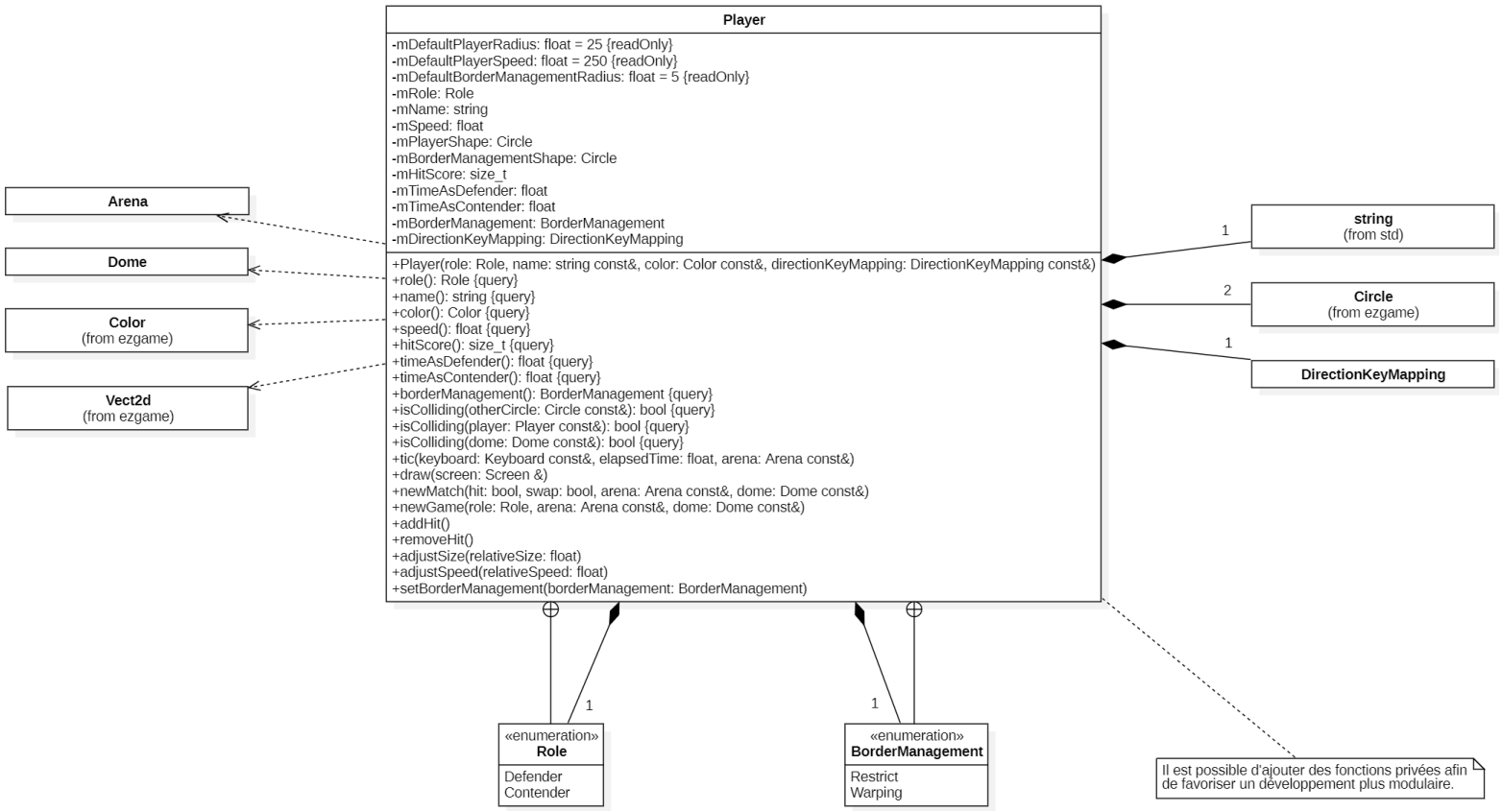
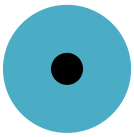


Diagramme de classe : Player

Description :

- Concept :
  - classe représentant un joueur
- Rôles :
  - possède l'état d'un joueur :
    - rôle : défenseur ou prétendant
    - nom
    - position
    - vitesse
    - taille
    - couleur
    - pointages : par point de *match* et par le temps écoulé
    - stratégie de gestion des bordures : restriction ou téléportation
    - touches du clavier associées au déplacement (voir `DirectionKeyMapping`)
  - détermine la position selon le rôle :
    - défenseur, au centre de l'arène
    - prétendant, une position aléatoire sur la bordure d'un cercle
  - effectue la détection de collision entre le joueur et :

        - un autre joueur
        - le dôme
        - le modificateur
- Représentation graphique :
    - deux cercles sont utilisés pour représenter le joueur
    - un cercle externe :
        - représentant le joueur
        - de la couleur du joueur
        - c'est le cercle utilisé pour la détection de collision
    - un cercle interne superposé au cercle externe :
        - centré sur le cercle interne
        - représentant la stratégie de gestion de la bordure
        - la couleur est mise à jour dès que la stratégie de gestion de la bordure change
        - la couleur est :
            - blanche pour restriction
            - noire pour téléportation
    - pour les deux cercles, aucune bordure n'est utilisée.



- Autres informations :
    - la vitesse est en pixel par seconde.
- Sous-types :
    - `Role` : le rôle du joueur, défenseur ou prétendant
    - `BorderManagement` : énumération représentant la stratégie de gestion de la bordure.

Détails :

- `Player(role, name, color, directionKeyMapping)`
    - Constructeur initialisant, dans l'ordre déclaré, les constantes et variables.
- `role()`
    - Accesseur retournant le rôle.
- `name()`
    - Accesseur retournant le nom.
- `color()`
    - Accesseur retournant la couleur du joueur.
- `speed()`
    - Accesseur retournant la vitesse.
- `hitScore()`
    - Accesseur retournant le pointage de *match*.
- `timeAsDefender()`
    - Accesseur retournant le temps passé en tant que défenseur.
- `timeAsContender()`
    - Accesseur retournant le temps passé en tant que prétendant.
- `borderManagement()`
    - Accesseur retournant la stratégie de gestion des bordures.
- `isColliding(otherCircle)`
    - Fonction utilitaire retournant si le cercle donné entre en collision avec le joueur.
- `isColliding(player)`
    - Fonction utilitaire retournant s'il existe une collision entre le joueur et le joueur donné.
    - Cette surcharge de fonction appelle la fonction `isColliding(Circle)`.
- `isColliding(dome)`

    - Fonction utilitaire retournant s'il existe une collision entre le joueur et le dôme.
    - Cette surcharge de fonction appelle la fonction `isColliding(Circle)`.
- `tic(keyboard, elapsedTime, arena)`
    - Fonction utilitaire gérant la progression de la simulation.
    - Effectue ces opérations :
        - augmente le temps écoulé selon le rôle
        - déplace le joueur selon l'état du clavier, en incluant :
            - la gestion de la bordure de l'arène
            - la synchronisation de position du cercle interne au cercle externe
- `newMatch(hit, swap, arena, dome)`
    - Fonction utilitaire réinitialisant l'état du joueur pour un nouveau *match*.
    - Effectue ces opérations :
        - si `hit`, fait un point de plus – sinon, aucun point supplémentaire
        - si `swap`, il y a permutation du rôle, le défenseur devient le prétendant et vice versa
        - réinitialise la position du joueur selon son rôle (s'il y a une permutation des rôles, selon le nouveau rôle)
- newGame(role, arena, dome)
    - Fonction utilitaire réinitialisant l'état du joueur pour une nouvelle partie.
    - Effectue ces opérations :
        - assigne le nouveau rôle
        - réinitialise la vitesse à la vitesse nominale
        - réinitialise la taille du joueur (incluant le marqueur au centre)
        - réinitialise le mode de gestion des bordures d'arène à `Restrict`
        - réinitialise le pointage de *match* à 0
        - réinitialise les temps cumulés en tant que défenseur et prétendant à 0
        - réinitialise la position selon le rôle
- `draw(screen)`
    - Dessine le joueur, les objets `mCircle`.
- `addHit()`
    - Ajoute un point de *match*.
- `removeHit()`
    - Retire un point de *match*.
    - Attention, la valeur ne peut être plus petite que 0.
- `adjustSize(relativeSize)`
    - Augmente ou réduit la taille du joueur selon `relativeSize`.
    - Attention, les rayons sont limités à ces valeurs minimums :
        - `mDefaultBorderManagementRadius` pour le cercle externe
        - 1 pour le cercle interne
- `adjustSpeed(relativeSpeed)`
    - Augmente ou réduit la vitesse du joueur selon `relativeSpeed`.
    - Attention, la vitesse minimum possible est 1
- `setBorderManagement(borderManagement)`
    - Mutateur déterminant la stratégie de gestion de la bordure.

Note importante

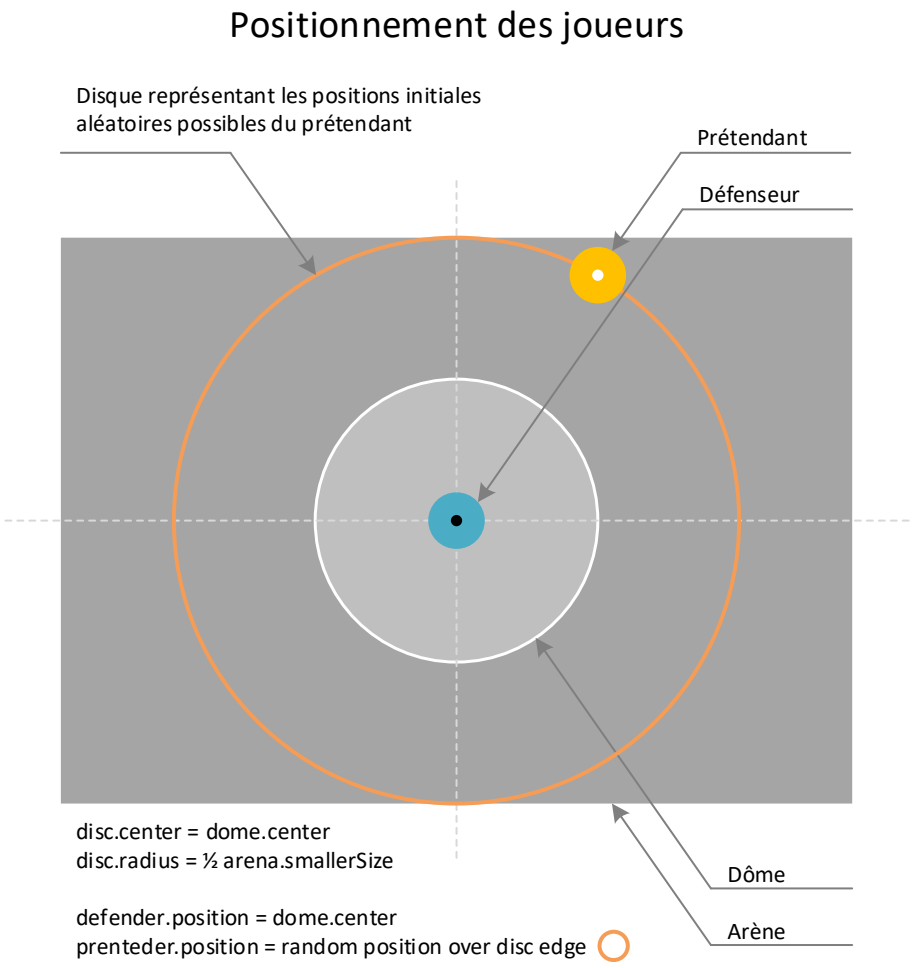
Cette conception ne présente aucune opération privée. Même si ces fonctions ne sont pas essentielles au fonctionnement de la classe, elles permettent souvent d'améliorer le code en

simplifiant le développement du projet. Sachez qu'il serait certainement préférable d'en ajouter pour augmenter la clarté et la modularité du code.

À titre d'exemple, l'implémentation de référence en possède 6. Cette information n'est donnée qu'à titre comparatif. C'est possible d'en faire plus ou moins. L'objectif est que ces ajouts soient pertinents et utiles.

La position du joueur est assignée selon le rôle :

- si le rôle est le défenseur, la position est au centre du dôme
- si le rôle est le prétendant, la position est aléatoire sur le pourtour d'un disque centré sur le dôme dont le rayon est la moitié de la taille la plus petite de l'arène



Représentation et positionnement des joueurs

Classe Modifier

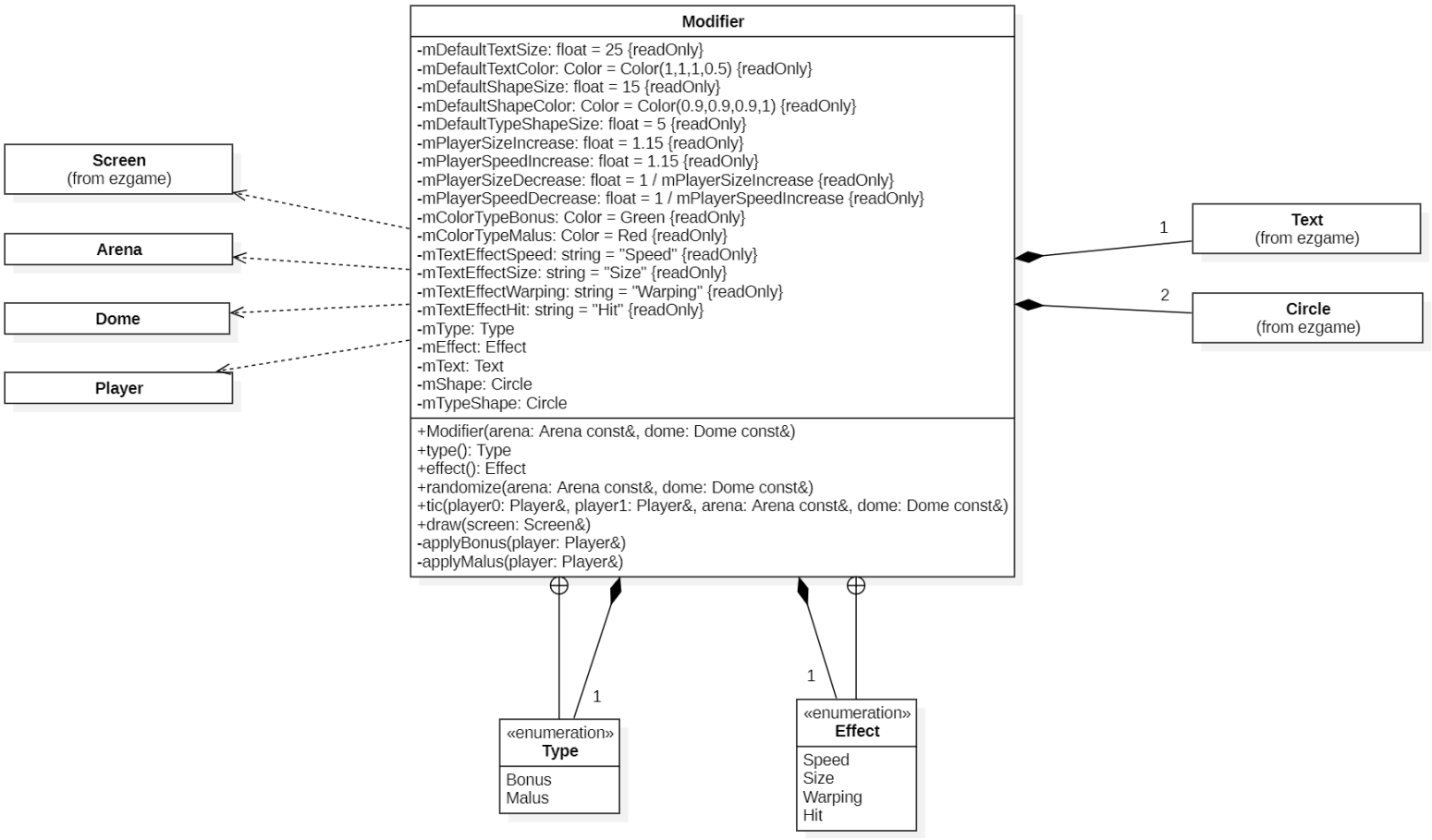
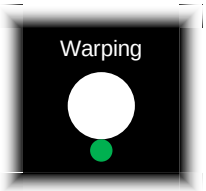


Diagramme de classe : Modifier

Description :

- Concept :
  - classe représentant le modificateur
- Rôles :
  - possède l'état du modificateur : le type, l'effet, sa position et sa présentation
- Représentation graphique :
  - deux cercles et un texte
  - couleur :
    - pour les trois, la couleur de remplissage est blanche
    - aucune bordure
  - le cercle principal :
    - représente le corps du modificateur
    - rayon de 15
    - c'est le cercle utilisé pour la détection de collision
  - le cercle secondaire :
    - représente le type du modificateur
    - la couleur est vert pour un bonus et rouge pour un malus

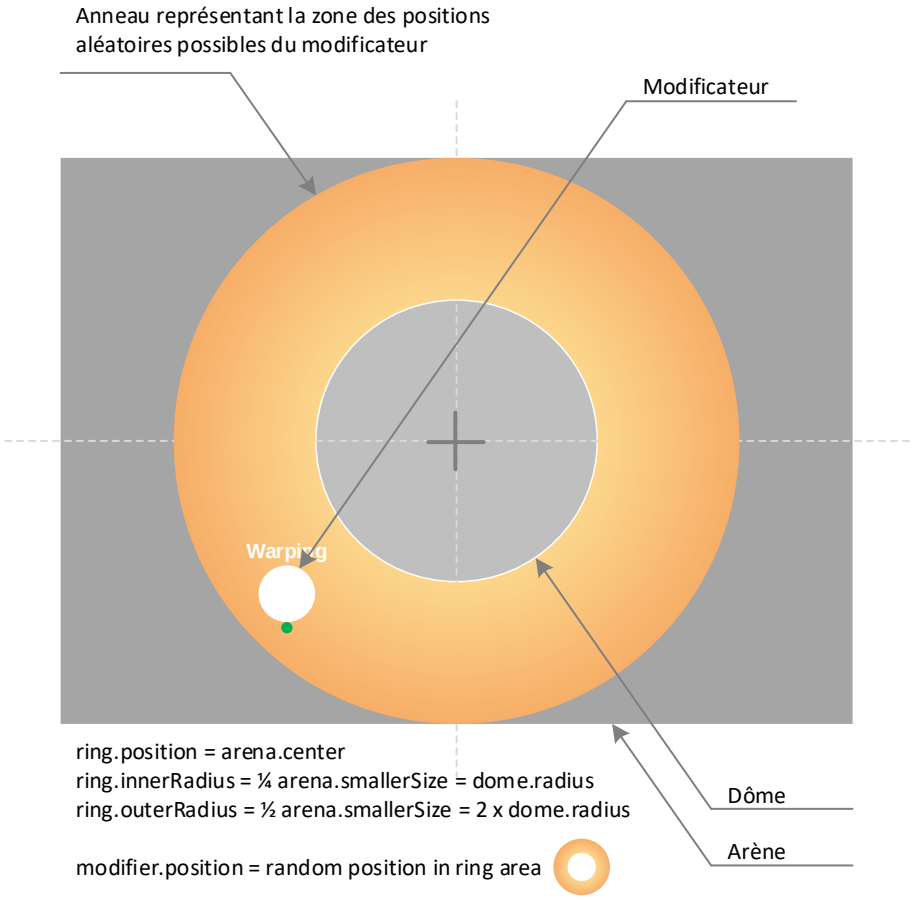


        - rayon de 5
        - se retrouve sous le cercle principal (centré et adjacent au cercle principal)
        - ne participe pas à la détection de contribution
    - le texte :
        - représente l'effet courant, soit l'un des textes suivants : `Size`, `Speed`, `Warping` et `Hit`
        - la taille du texte est de 15
        - le texte est centré au-dessus du cercle principal
- Sous-types :
    - `Type` : énumération représentant le type de modification (bonus ou malus)
    - `Effect` : énumération représentant l'une des quatre modifications possibles

Détails :

- `Modifier(arena, dome)`
    - Constructeur initialisant, dans l'ordre déclaré, les constantes et variables.
    - Appel la fonction `randomise`.
- `type()`
    - Accesseur retournant le type.
- `effect()`
    - Accesseur retournant l'effet.
- `randomize(arena, dome)`
    - Réinitialise aléatoirement le modificateur.
    - Les étapes sont les suivantes :
        1. détermine aléatoirement le type (voir `Random::enumeration`)
        2. déterminer la couleur du cercle secondaire selon le type
        3. détermine aléatoirement l'effet (voir `Random::enumeration`)
        4. détermine le texte selon l'effet
        5. détermine la position des trois éléments graphiques :
            - le cercle principal, aléatoirement dans l'anneau permis
            - le cercle secondaire, relative au cercle principal
            - le texte, relative au cercle principal
- `tic(player0, player1, arena, dome)`
    - Fonction utilitaire gérant la progression de la simulation.
    - Les étapes sont les suivantes :
        1. si les deux joueurs touchent au modificateur en même temps :
            - le modificateur n'est appliqué à personne
            - on réinitialise le modificateur (voir la fonction `randomize`)
        2. si l'un des joueurs touche au modificateur
            - si le type est bonus, on appel la fonction `applyBonus`
            - si le type est malus, on appel la fonction `applyMalus`
        3. si aucun contact, on ne fait rien
- `draw(screen)`
    - Dessine le modificateur, les objets (`mCircle` et `mText`).
- `applyBonus(player)`
    - selon l'effet, applique la modification positive au joueur
- `applyMalus(player)`
    - selon l'effet, applique la modification négative au joueur

Positionnement du modificateur



Représentation et positionnement du modificateur

Classe ScoreManager

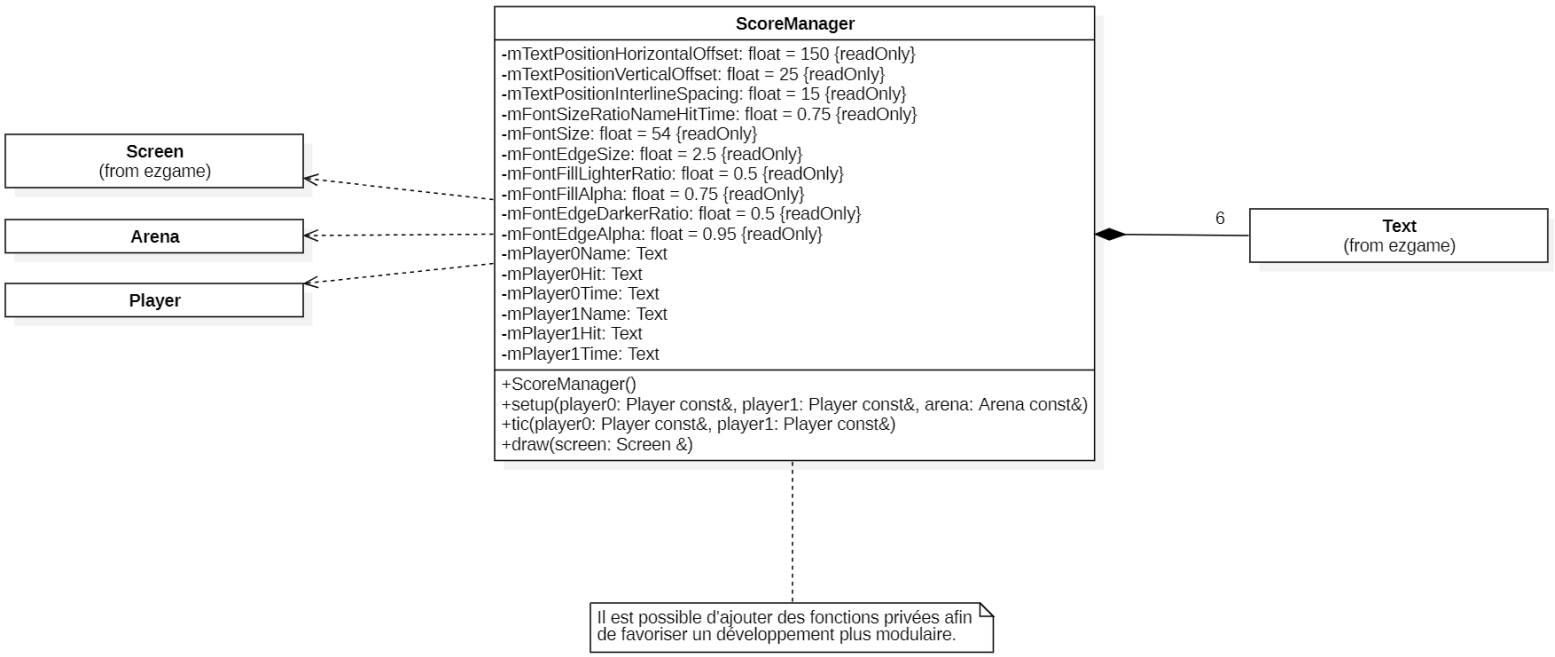


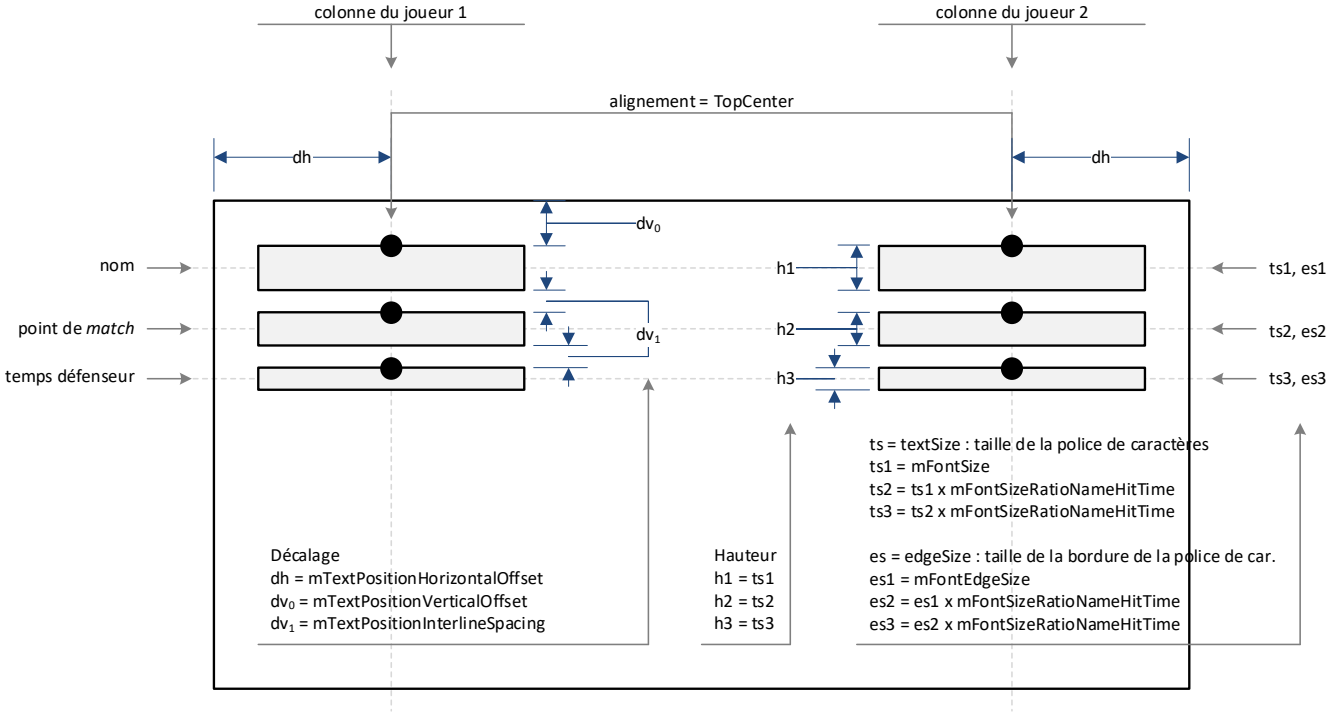
Diagramme de classe : ScoreManager

Description :

- Concept :
  - classe utilitaire simplifiant l'affichage des informations textuelles à l'écran
- Rôles :
  - maintient les informations de disposition des 6 textes présents sur la surface graphique
  - permet d'afficher simplement les informations des joueurs
- Représentation graphique :
  - 6 textes
  - disposition :
    - sur deux colonnes : joueur 1 et joueur 2
    - sur trois lignes : nom, pointage de *match* et temps en tant que défenseur
    - voir le schéma de la page suivante
  - la couleur de la police de caractères est :
    - remplissage : celle du joueur éclairci de 50%
    - contour : celle du joueur assombri de 50%
- Autres informations :
  - la disposition est calculée une fois au début du jeu par l'appel de la fonction `setup`
  - la mise à jour se fait à chaque appel de `processEvents` via la fonction `tic`
  - l'affichage ce fait à chaque appel de `processDisplay` via la fonction `draw`
- Sous-types :
  - aucun

Détails sur les opérations :

- `ScoreManager()`
  - Constructeur initialisant, dans l'ordre déclaré, les constantes et variables.
- `setup(player0, player1, arena)`
  - pour chacun des textes, établi :
    - la position
    - la taille de la police de caractère
    - la couleur de remplissage
    - la couleur de la bordure
    - la taille de la bordure
    - l'alignement
- `tic(player0, player1)`
  - met à jour le contenu des chaînes de caractères pour chacun des textes
- `draw(screen)`
  - dessine les six textes



Disposition de l'affichage

Note importante

Comme pour la classe `Player`, à la page 13, cette conception ne présente aucune opération privée. Même si ces fonctions ne sont pas essentielles au fonctionnement de la classe, elles permettent souvent d'améliorer le code en simplifiant le développement du projet. Sachez qu'il serait certainement préférable d'en ajouter pour augmenter la clarté et la modularité du code.

À titre d'exemple, l'implémentation de référence en possède 6. Cette information n'est donnée qu'à titre comparatif. C'est possible d'en faire plus ou moins. L'objectif est que ces ajouts soient pertinents et utiles.

Classe GameEngine

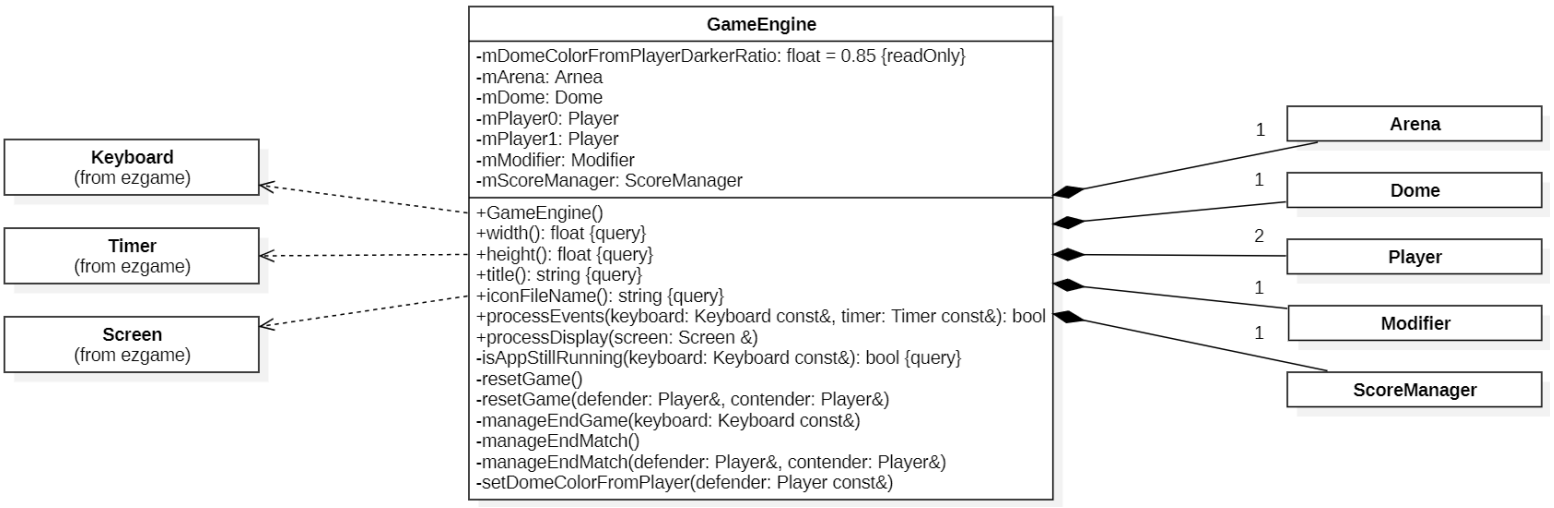


Diagramme de classe : GameEngine

Description :

- Concept :
  - classe intégratrice du projet
- Rôles :
  - permets une gestion centralisée :
    - en intégrant tous les constituants
    - appliquant la logique de haut niveau
  - elle est l'interface de programmation entre la bibliothèque `EzGame` et le reste du projet
- Représentation graphique :
  - aucune
- Autres informations :
  - aucune
- Sous-types :
  - aucun

Détails sur les opérations :

- `GameEngine()`
  - Constructeur initialisant, dans l'ordre déclaré, les constantes et variables.
  - Fait les appels suivants :
    - `ScoreManager::setup`
    - `GameEngine::resetGame`
- `width()`
  - Accesseur retournant la largeur de l'arène.
- `height()`
  - Accesseur retournant la hauteur de l'arène.
- `title()`

    - Accesseur retournant le titre de l'application.
- `iconFileName()`
    - Accesseur retournant le nom du fichier où se trouve l'icône de l'application.
- `processEvents(keyboard, timer)`
    - Fonction centrale de l'application où les données du modèle sont mises à jour.
    - L'utilisation du clavier et du temps écoulé sont disponibles.
- `processEvents(screen)`
    - Fonction centrale de l'application où la surface graphique est mise à jour.
- `isAppStillRunning(keyboard)`
    - Fonction utilitaire retournant si la touche du clavier pour quitter l'application a été appuyée.
    - La touche `escape`.
- `resetGame()`
    - Détermine quel joueur est le défenseur et quel joueur est le prétendant.
    - Appel la fonction `resetGame(defender, pretender)`
- `resetGame(defender, pretender)`
    - Appel la fonction `Player::newGame` pour chacun des joueurs.
    - Détermine la couleur du dôme.
    - Réinitialise le modificateur.
- `manageEndGame(keyboard)`
    - Si la touche du clavier déterminant la fin de la partie a été appuyée, appelle la fonction `resetGame()`.
    - La touche `enter`.
- `manageEndMatch()`
    - Selon l'état des joueurs, appelle la fonction `manageEndMatch(defender, contender)`.
- `manageEndMatch(defender, contender)`
    - si le prétendant entre en collision avec le dôme, appelle les fonctions suivantes :
        - contender.newMatch +1 point, avec permutation des rôles
        - defender.newMatch aucun point, avec permutation des rôles
        - setDomeColorFromPlayer
    - si le défenseur entre en collision avec le prétendant, appel les fonctions suivantes :
        - contender.newMatch aucun point, sans permutation des rôles
        - defender.newMatch +1 point, sans permutation des rôles
- `setDomeColorFromPlayer(defender)`
    - détermine la couleur du dôme selon la couleur du défenseur assombrie de 85%

EzGame

Vous avez à votre disposition une bibliothèque vous permettant de créer une application graphique facilement et rapidement.

En bref, cette bibliothèque permet :

- d'afficher une fenêtre graphique
- d'appeler en boucle deux fonctions :
  - une pour la mise à jour des données
  - une pour l'affichage à l'écran
- de consulter les informations suivantes:
  - l'état du clavier
  - le temps écoulé depuis le début du programme ou depuis le dernier tic
- d'afficher à l'écran :
  - une couleur uniforme pour toute la surface
  - un cercle
  - un texte

Vous trouverez dans le dossier du projet un petit site web présentant la documentation de la bibliothèque. Le fichier d'amorce est : `./EzGame/doc/index.html`.

Il est essentiel de consulter la documentation pour bien comprendre la structure des outils disponibles.

Solution Visual Studio mise à votre disposition

Vous avez à votre disposition une solution Visual Studio. Cette solution est préconfigurée de façon à pouvoir utiliser la bibliothèque `EzGame` directement.

Il est important de ne pas modifier ou effacer des éléments contenus dans le sous-dossier `./EzGame/`.

Contraintes

Voici les contraintes du projet :

- Contraintes techniques :
  - Vous devez respecter les noms des types, classes, attributs et opérations donnés.
  - Vous devez respecter la norme de codage imposée.
  - Il est attendu que votre code soit documenté adéquatement.
  - Votre code ne doit pas utiliser :
    - l'instruction `goto`;
    - des variables globales;
    - des littéraux sans variables symboliques (sauf lorsque trivial).
  - Il est obligatoire de réaliser le projet avec :
    - le langage de programmation `C++`
    - l'environnement de développement `Visual Studio`
    - la bibliothèque `EzGame`

    Pour chacun de ces éléments, on vous demande d'utiliser les versions les plus récentes.

- Contraintes liées au travail et à la remise :
  - Vous avez trois périodes pour réaliser ce laboratoire.
  - Vous devez le faire individuellement
    - sans faire de plagiat, l'entre-aide constructive est fortement encouragée entre les étudiants de la classe
    - si vous avez de l'aide, assurez-vous de maitriser ce qui vous est présenté
    - l'important est d'acquérir les compétences, pas d'où vos connaissances viennent
  - Assurez-vous de bien comprendre les éléments que vous remettez. Si le chargé de laboratoire vous pose une question sur le travail que vous avez remis et que vous n'êtes pas capable de répondre ou d'expliquer votre raisonnement, vous aurez systématiquement une note de zéro pour plagiat. Une telle situation suivra le protocole de discipline de l'ÉTS prévu à cet effet.
  - Il est attendu que la qualité générale du rapport final soit du niveau d'un étudiant en génie. La qualité du français est évaluée et une pénalité pouvant aller jusqu'à 10 % peut être appliquée.

Démarche suggérée

La démarche suivante vous aidera à mieux cibler l'ordre des tâches à faire pour réaliser un tel projet.

1. Avant de commencer :
    a. outils de développement :
        i. assurez-vous d'avoir installé la bonne version de `Visual Studio`
        ii. d'avoir le module `C++`
        iii. d'avoir fait la dernière mise à jour
    b. compréhension du projet :
        i. assurez-vous d'avoir fait une lecture de l'énoncé :
            1. on vous suggère de débuter par une lecture rapide
            2. pour ensuite approfondir progressivement les points techniques au besoin.
        ii. assurez-vous d'avoir une compréhension minimum de la bibliothèque `EzGame` qui vous est donnée :
        voir le sous-dossier : `./EzGame/doc/index.html`
        iii. assurez-vous de bien comprendre les éléments techniques présentés :
            1. faites-le aussitôt que possible
            2. poser vos questions rapidement afin d'éviter d'être pris à la dernière minute.
2. Démarrage du projet :
    a. Ouvrir le projet de démarrage donné dans `Visual Studio`.
    b. Assurez-vous qu'il compile et s'exécute en mode `Debug` et `Release`.
    c. Ajoutez au projet une classe `GameEngine` de votre cru et faire un test minimum de fonctionnement. Voir la documentation pour une suggestion.
3. Développement des classes :
    a. Assurez-vous de bien comprendre le rôle de chaque classe à développer afin de cerner les tâches nécessaires à réaliser. Il est aussi important de comprendre progressivement comment les classes collaborent et comment certaines s'imbriquent les unes dans les autres.
    b. Développez les parties une à la fois en débutant par les fonctions les plus simples et faites rapidement de petits tests validant votre travail au fur et à mesure (`CALTAL`).
    c. Voici une suggestion concernant l'ordre du développement à réaliser :
        i. `GameEngine` :
            1. débuter par une version minimale de départ, l'objectif étant de s'assurer que tout fonctionne bien et qu'il est possible d'avoir un point d'entrée dans le programme (voir l'exemple dans la documentation de `EzGame`).
            2. progressivement, au fur et à mesure que les autres classes sont développées, faites des tests sur ces dernières et intégrés progressivement les logiques associées dans le `GameEngine`
            3. à la toute fin, finaliser le développement des éléments manquants du projet
        ii. `Arena`
        iii. `Dome`
        iv. `DirectionKeyMapping`
        v. `Player`

vi. `Modifier`
vii. `ScoreManager`

4. Faire les tests globaux.
5. Remise du projet.
6. Vous préparer pour le quiz.

Rappelez-vous de mettre en pratique les recommandations exprimées par ces acronymes :

- `CALTAL` : « **C**ode **A** **L**ittle, **T**est **A** **L**ittle »
- `DRY` : « **D**on't **R**epeat **Y**ourself »
- `UMUD` : « yo**U** **M**ust **U**se the **D**ebugger »

Remise

La remise se fait sur Moodle :

- **avant** la première période du laboratoire 2 (la veille de cette période, vous aurez une pénalité équivalente à un jour de retard si votre remise n'est pas faite);
- le document ZIP contenant votre travail :
  - nom du document ZIP : `GPA434_Lab1_NomPrenom.zip`
  - le document ZIP doit contenir votre solution tout **en vous assurant de faire le nettoyage** de cette dernière. Voir le document présentant la procédure sur le site Moodle du cours. **N'effacez rien** du dossier `EzGame`.