



COMMENT IMPLÉMENTER DE NOUVELLES BIBLIOTHÈQUES DE JEUX

Projet Arcade : Axel Eckenberg & Marius Pain

Introduction

L'intégration de nouvelles bibliothèques de jeux pour le projet Arcade d'Epitech requiert une approche méthodique et structurée, étroitement alignée avec une architecture spécifique afin de garantir sa compatibilité avec le core du projet, notamment à travers l'utilisation d'interfaces préétablies. Dans ce document, nous explorerons les étapes essentielles pour implémenter de nouvelles bibliothèques de jeux dans le cadre du projet Arcade. Nous mettrons en lumière l'importance de respecter les interfaces et conventions établies dans le but de garantir la fonctionnalité et la compatibilité de ces bibliothèques avec notre projet, tout en assurant une expérience de développement harmonieuse.

Intégrer une nouvelle bibliothèque de jeu

Une fois la bibliothèque sélectionnée, il est temps de procéder à leur intégration dans l'architecture existante du projet Arcade. Tout d'abord, vous devrez récupérer l'ensemble des interfaces nécessaires pour la suite dans le dépôt GitHub suivant : [ArcadeShared](#).

1. Vos Classes

Vous devrez dans un premier temps créer (au moins) :

- Une classe principale pour votre bibliothèque, celle-ci devra hériter de l'interface : [IGame](#).
- Une classe d'erreur, celle-ci devra hériter de l'interface : [IGameError](#)

2. Votre classe principale

Afin d'implémenter votre classe principale vous devrez d'abord créer les méthodes héritées via l'interface, puis vous verrez comment vous pouvez créer différents éléments pour votre jeu et enfin ce qu'il vous faudra rajouter pour que votre bibliothèque puisse être chargée avec notre core.

a. Méthodes héritées

En héritant de l'interface [IGame](#), vous héritez des méthodes suivantes qu'il vous faudra implémenter :

- **init** : La méthode est appelée juste après que la bibliothèque ait été chargée par le core. Celle-ci doit au moins faire en sorte de stocker le pointeur partager (shared_ptr) [IArcade](#) qu'elle reçoit. Cela permet notamment d'utiliser les fonctions :
 - **display** : Appelle la fonction display de la bibliothèques graphiques actuellement chargée par le core, en lui envoyant l'entité que vous lui passer en paramètre.
 - **flipFrame** : Appelle la fonction flipFrame de la bibliothèques graphiques actuellement chargée par le core. Ne prend aucun paramètre.
 - **bindEvent** : Appelle la fonction bindEvent de la bibliothèques graphiques actuellement chargée par le core, en lui envoyant le type d'événement, la touche et la fonction a appelé lorsque l'événement est déclenché.
 - **setPreferredSize** : Appelle la fonction setPreferredSize de la bibliothèques graphiques actuellement chargée par le core, en lui envoyant la largeur et la hauteur minimal de l'écran.
 - **getDeltaTime** : Récupère le temps en seconde qu'il s'est écoulé entre les deux appels de la méthode run (méthode de la bibliothèque de jeu, voir plus bas).
 - **getTime** : Récupère l'heure actuelle de la machine en milliseconde.

- `getCurrentGameHighScore` : Récupère le meilleur score du jeu actuellement chargé par le core.

(Voir le pdf sur l'implémentation de bibliothèque graphique pour plus de détails sur les fonctions en lien avec les bibliothèques graphiques)

La fonction `init` prend en paramètre :


- `std::shared_ptr<IArcade> arcade` : Correspondant à un pointeur partagé sur le Core/Arcade.
- `start` : La méthode est appelée une première fois après la méthode `init` lorsque le jeu est chargé, puis elle est rappelée à chaque fois que l'utilisateur utilise la touche "relancer le jeu". Vous devez notamment réinitialiser le score du joueur dedans et faire d'autres actions (suivant la logique de votre jeu). La fonction ne prend pas de paramètre.
- `run` : La méthode est appelée à chaque tour de boucle (chaque frame), pendant que le jeu est lancé. Vous devez mettre dans la méthode la logique de votre jeu, les mouvements du joueur... La fonction ne prend pas de paramètre.
- `getScore` : La méthode doit retourner le score actuel du jeu. La fonction ne prend pas de paramètre.

b. Créer différents éléments

L'interface [`IDisplayable`](#) reçu par la méthode `display` du driver contient des méthodes utiles telles que des setters et des getters pour la position, la couleur, la scale (ou size) mais celle-ci possède également d'autres interfaces enfants tel que :

- [IEntity](#) : IEntity contient toutes les méthodes liées à la gestion de sprite/d'image.
- [IPrimitive](#) : IPrimitive est une interface qui regroupe tout ce qui peut être dessiné de façon simple par le driver, elle possède également plusieurs interfaces enfants telles que :
 - [ICircle](#) : ICircle contient toutes les méthodes nécessaires pour dessiner un cercle
 - [ILine](#) : ILine contient toutes les méthodes nécessaires pour dessiner une ligne
 - [ISquare](#) : ISquare contient toutes les méthodes nécessaires pour dessiner un carré
 - [IText](#) : IText contient toutes les méthodes nécessaires pour dessiner un text

Vous pouvez donc créer plusieurs classes qui héritent de certaines de ces interfaces et y mettre le contenu que vous souhaitez dedans, vous devez penser à implémenter au minimum les méthodes déclarées dans l'interface que vous utilisez.

 **Astuces** : Pour éviter de réécrire plusieurs fois les méthodes des interfaces que vous utilisez, vous pouvez créer des classes abstraites dans lesquelles vous regroupez le code qui est commun pour ces méthodes.

c. Externe "C"

Dans votre classe principale, vous allez devoir rajouter une structure "`extern "C"`" qui contient à l'intérieur obligatoirement au moins la fonction :

- "`std::unique_ptr<IGame> create_game(void)`" : La fonction doit créer une nouvelle instance de votre jeu et le retourner en tant qu'unique pointeur.

Vous pourrez également rajouter les fonctions suivantes dans la structure, mais elles ne sont pas obligatoires :

- “`const std::string &get_name(void)`” : La fonction doit retourner une chaîne de caractère correspondant au nom de votre jeu (ex : “Snake”...)
- “`__attribute__((constructor)) void load_lib()`” : Cette fonction sera appelée lorsque votre bibliothèque sera chargée par le core, vous ne devez mettre aucune logique nécessaire à votre bibliothèque dedans, cependant vous pouvez faire en sorte que celle-ci écrive un message pour dire que celle-ci est chargée.
- “`__attribute__((destructor)) void unload_lib()`” : Cette fonction sera appelée lorsque votre bibliothèque sera déchargée par le core, vous ne devez mettre aucune logique nécessaire à votre bibliothèque dedans, cependant vous pouvez faire en sorte que celle-ci écrive un message pour dire que celle-ci est déchargée.

3. Votre classe d'erreur

Afin d'implémenter votre classe d'erreur, vous devrez créer les méthodes héritées via l'interface, cela permettra que lorsqu'un problème se passe dans votre jeu, notre core puisse traiter cette erreur et afficher le message que vous lui donnez.

a. Méthodes héritées

En héritant de l'interface [IGameError](#), vous héritez des méthodes suivantes qu'il vous faudra implémenter :

- `getCode` : La méthode doit retourner un code d'erreur (int). Les codes ne sont pas prédéfinis et sont libres à chacun.
- `what` : La méthode retourne une chaîne de caractère contenant un message qui explique pourquoi et/ou comment et/ou où l'erreur s'est produite.

4. Conclusion

Vous avez désormais toutes les clefs en main pour implémenter une nouvelle bibliothèque de jeu pour notre arcade, n'hésitez pas à faire une pull request pour ajouter votre bibliothèque si vous souhaitez contribuer au projet. Nous vous remercions par avance de votre contribution.