



COMMENT IMPLÉMENTER DE NOUVELLES BIBLIOTHÈQUES GRAPHIQUES

Projet Arcade : Axel Eckenberg & Marius Pain

Introduction

L'intégration de nouvelles bibliothèques graphiques 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 graphiques dans le cadre du projet Arcade. Nous mettrons en lumière l'importance de respecter les interfaces et conventions établies afin 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 graphique

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 : [IDriver](#).
- Une classe d'erreur, celle-ci devra hériter de l'interface : [IDriverError](#)

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 afficher différents contenus 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 [IDriver](#), vous héritez des méthodes suivantes qu'il vous faudra implémenter :

- `bindEvent` : La méthode doit faire en sorte d'associer un événement et son `CallBack` (une fonction qui doit être appelée quand l'événement est déclenché). Il vous faut stocker dans un `map` (ou avec une autre méthode)

les événements et leurs Callbacks afin de pouvoir tester s'ils sont déclenchés pendant le jeu. La fonction prend en paramètre :

- `IEvent::EventType type` : Correspondant au type d'événement en fonction de l'énumération `IEvent`.
 - `EventKey key` : Correspondant à la touche à laquelle, on associe le type d'événement en fonction du typedef `EventKey`.
 - `EventCallback callback` : Correspondant à un pointeur sur la fonction qui doit être appelé lorsque l'événement est déclenché.
- `unbindAll` : La méthode doit faire en sorte de vider votre map (ou autre méthode) qui vous permettait de stocker les événements et leurs Callbacks. La fonction ne prend pas de paramètre.
- `setPreferredSize` : La méthode doit faire en sorte de modifier la taille actuelle de l'écran et/ou la taille minimum requise de l'écran (cela permet notamment de passer d'un écran vertical pour le menu à un écran plus carré pour le pacman ou tout autre jeux). La fonction prend en paramètre :
 - `std::size_t width` : Correspondant à la largeur minimale de l'écran
 - `std::size_t height` : Correspondant à la hauteur minimale de l'écran
- `display` : La méthode doit dessiner/charger l'élément qui lui est envoyé pour qu'ensuite celui-ci soit affiché lorsque la méthode `flipFrame` est appelée. La fonction prend en paramètre :
 - `const IDisplayable &displayable` : Correspondant à l'élément que l'on souhaite charger (Plus de détails sur comment le charger dans la section suivante)
- `flipFrame` : La méthode doit afficher l'ensemble du contenu qui a été précédemment chargé avec la méthode `display`, puis elle doit nettoyer l'affichage pour le prochain tour de boucle. Elle doit aussi vérifier si un événement est en cours et si c'est le cas s'il correspond à l'un des événements auquel un Callback est attaché. Cette fonction est appelée à


chaque tour de boucle du jeu afin d'afficher tous les éléments dessinés pendant la boucle. La fonction ne prend pas de paramètre.

b. Comment afficher différents contenus

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

Afin de savoir si l'élément que l'on vous envoie est une image, un texte, une ligne ou autre et donc de le dessiner en utilisant la bonne méthode, vous pouvez utiliser les "`dynamic_cast`" qui transforme IDisplayable en IEntity ou en IPrimitive et ensuite votre IPrimitive en ICircle, en ILine...

 **Astuce :** Votre "`dynamic_cast`" vous renvoie un `nullptr` s'il ne s'agit pas de la bonne interface.

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<IDriver> create_driver(void)"` : La fonction doit créer une nouvelle instance de votre driver 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 driver (ex : "SFML"...)
- `"__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 driver, notre core puisse traiter cette erreur et afficher le message que vous lui donnez.

a. Méthodes héritées

En héritant de l'interface [IDriverError](#), 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 graphique 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.