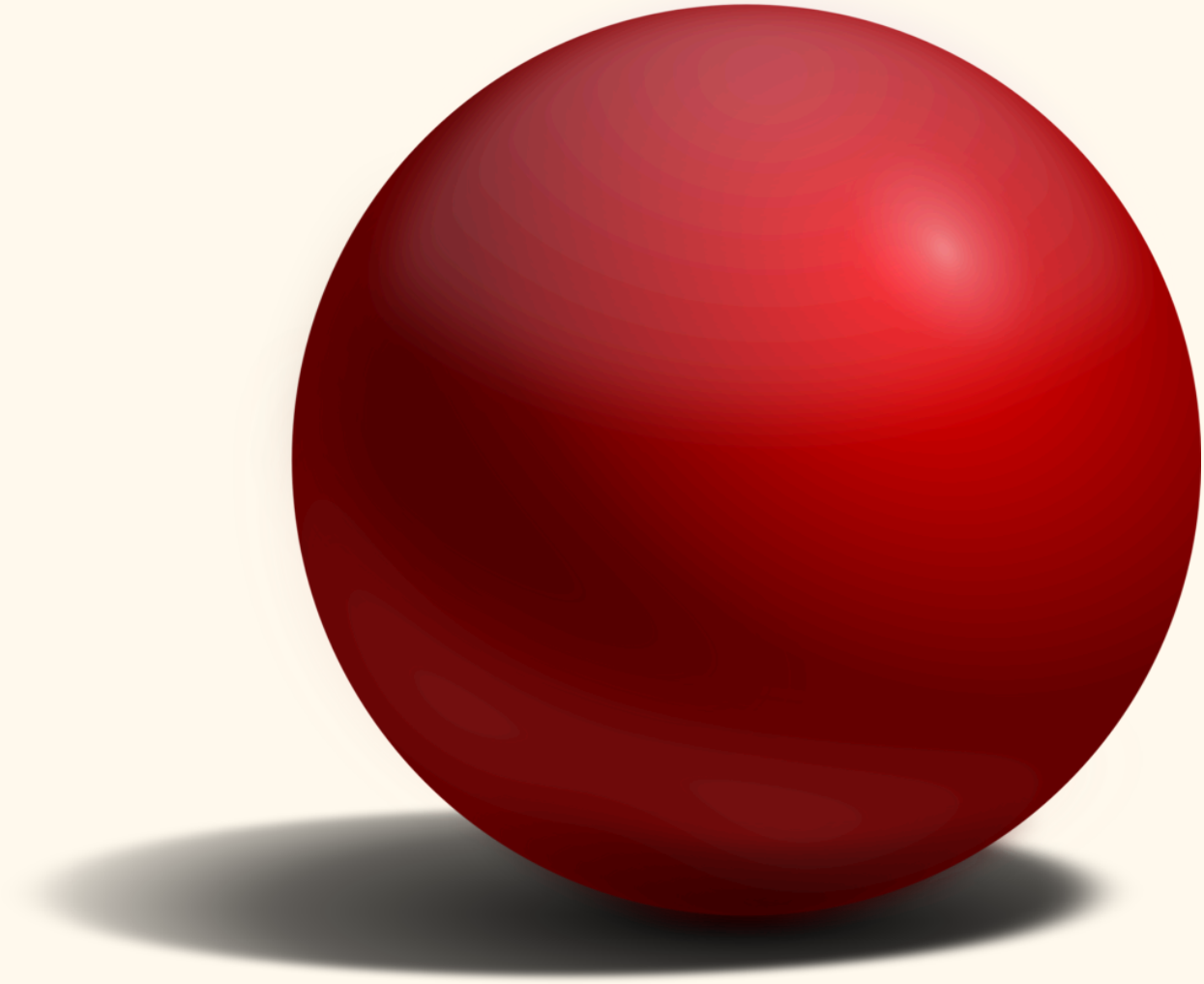
EPITECH.
L'ECOLE DE L'EXPERTISE INFORMATIQUE



COMMENT IMPLÉMENTER DE NOUVEAUX OBJETS PRIMITIFS

Projet RayTracer : Marius Pain, Landry Gigant, Thomas Boué & Aubane Nourry

Introduction

L'intégration de nouveaux objets primitifs pour le projet RayTracer d'Epitech requiert une approche méthodique et structurée, étroitement alignée avec une architecture spécifique afin de garantir sa compatibilité avec le corps du projet, notamment à travers l'utilisation d'interfaces préétablies. Dans ce document, nous explorerons les étapes essentielles pour implémenter de nouveaux objets primitifs dans le cadre du projet RayTracer. 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 objets avec notre projet, tout en assurant une expérience de développement harmonieuse.

Créer un nouveau objet primitif

Une fois l'objet sélectionné, il est temps de procéder à son intégration dans l'architecture existante du projet RayTracer.

1. Les Prérequis

Vous devrez dans un premier temps :

- Créer une classe principale pour votre objet, celle-ci devra hériter de la classe abstraite : APrimitive.

  La classe abstraite contient toutes les méthodes communes à chaque objet primitif. Elle hérite de l'interface IPrimitive.

- Créer un fichier permettant de compiler votre objet en librairie dynamique.
- Mettre à jour l'analyse syntaxique pour pouvoir créer votre objet.

2. Votre classe principale

Afin d'implémenter votre classe principale vous devrez créer les méthodes héritées via la classe abstraite, puis vous verrez ce qu'il vous faudra rajouter pour que votre bibliothèque puisse être chargée avec notre core.

a. Méthodes héritées

En héritant de la classe abstraite APrimitive, vous héritez des méthodes suivantes qu'il vous faudra implémenter :

- hits : Cette méthode permet de définir le comportement de notre objet primitif. En effet, cette méthode va renvoyer un booléen (Vrai ou Faux) en fonction de si un rayon (Par exemple un rayon lumineux) rentre en contact avec notre objet. Vous allez donc devoir définir quand est-ce que le rayon va entrer en contact avec votre objet ou non dans cette méthode.

  La fonction hits prend en paramètre :

  `const Math::Ray &ray` : Correspondant au rayon qui rentre en contact ou non avec votre objet

  La fonction hits prend renvoie :

  `bool` : Correspondant à Vrai si le rayon a touché l'objet, Faux sinon

- hitPoint : Cette méthode est liée à la méthode hits (ci-dessus). Dans le cas où un rayon (envoyé en paramètre de cette méthode) entre en contact avec votre objet, la méthode hitPoint renvoie le Point (dans un espace 3 dimensions) où le rayon est entré en contact avec votre objet primitif.

  La fonction hitPoint prend en paramètre :

  `const Math::Ray &ray` : Correspondant au rayon qui rentre en contact ou non avec votre objet

  La fonction hitPoint renvoie:

  `Math::Point3D` : Correspondant au point d'intersection entre le rayon et votre objet primitif.

- getNormalAt : Cette méthode permet d'obtenir un vecteur normal de votre objet à un point donné.

  La fonction getNormalAt prend en paramètre :

  `const Math::Point3D &point` : Correspondant au point pour lequel vous voulez un vecteur normal de votre objet

  La fonction getNormalAt renvoie:

  `Math::Vector3D` : Correspondant au vecteur normal à votre objet par rapport au point donné

b. Créer des objets primitifs spécifiques

La classe abstraite APrimitive contient uniquement certains attributs d'un objet. En effet, elle contient:

`Math::Point3D _pos` : Correspondant à la position de votre objet dans l'espace

`std::shared<RayTracer::IMaterial> _material` : Correspondant au pointeur vers le matériau de votre objet *(Comment créer un nouveau matériau)*

Par exemple, pour créer une Sphère, vous devrez ajouter un attribut rayon, car c'est ce qui permet de définir votre sphère. Ce qui signifie, que pour chaque objet ayant besoin de plus qu'une position et un matériau pour être défini, vous devrez définir les ou les attributs supplémentaires. De plus, pour pouvoir construire des objets avec ces attributs en question, vous devrez créer des Constructeurs pour vos classes qui permettent de définir ses attributs.

C. Implémenter les transformations

Chaque objet primitif à l'opportunité d'effectuer des transformations. La liste des transformations se trouve dans ce dossier. Lorsque vous souhaitez pouvoir appliquer une transformation à votre objet primitif, vous avez deux étapes à respecter:

- Faire hériter la classe principale de votre objet des interfaces des transformations voulues.
- Implémenter toutes les méthodes héritées des interfaces des transformations voulues

Par exemple, si vous voulez implémenter un objet Torus et faire en sorte qu'il puisse effectuer des rotations, la définition de la classe sera la suivante:

"`class Torus : public RayTracer::APrimitive, public RayTracer::ICanRotate {`

`…`

`}`"

Sans oublier d'implémenter les méthodes héritées de l'interface "`RayTracer::ICanRotate`".

d. Externe “C”

Dans votre classe principale, vous allez devoir rajouter une structure “`extern "C"`” qui contient à l’intérieur obligatoirement au moins ces deux fonctions :

- “`Primitive::NomDeVotreClasse *loadPrimitiveInstance(void)`” : La fonction doit créer une nouvelle instance de votre objet. Pour une sphère par exemple, le prototype de la fonction sera:

  “`Primitive::Sphere *loadPrimitiveInstance(void)`”

- “`const std::string &getName(void)`” : La fonction doit renvoyer le nom de l’objet primitif que vous avez créé. Par exemple, pour une sphère, le nom renvoyé sera “Sphere”

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

- “`__attribute__((constructor)) void initsharedlibrary()`” : Cette fonction sera appelée lorsque votre objet sera chargé par le corps, vous ne devez mettre aucune logique nécessaire à votre objet dedans, cependant vous pouvez faire en sorte que cette fonction écrive un message pour dire que votre objet primitif est chargé.
- “`__attribute__((destructor)) void destroysharedlibrary()`” : Cette fonction sera appelée lorsque votre objet sera déchargé par le corps, vous ne devez mettre aucune logique nécessaire à votre objet dedans, cependant vous pouvez faire en sorte que cette fonction écrive un message pour dire que votre objet primitif est déchargé.

3. Créer une librairie dynamique

Après de correctement implémenter votre objet primitif, vous devrez pouvoir compiler le code de votre objet (La classe principale ainsi que la partie externe “C”) pour en faire un seul et même fichier au format “.so” qui va représenter une librairie dynamique.

Pour ce faire, vous allez devoir réaliser un “Makefile”. Ce Makefile est un fichier qui va simplement permettre de compiler votre code de manière automatique et plus simplifiée (pas besoin d’exécuter des lignes de commandes). Ce Makefile devra donc compiler le code, générer un fichier “.so” qui se nommera:

“`libnom_de_votre_classe.so`”: Correspondant au nom du fichier que le corps du programme devra ouvrir pour récupérer votre objet. Pour une sphère par exemple, le nom du fichier sera “`libsphere.so`“.

Votre Makefile devra également copier le fichier d’en-tête de votre classe dans le dossier adéquat.

Votre Makefile devra enfin déplacer le fichier de librairie dynamique dans le dossier adéquat.

4. Mettre à jour l'analyse syntaxique

Une fois que votre fichier de librairie dynamique est créé, vous arrivez à la dernière étape. Il ne reste plus qu'à modifier la partie analyse syntaxique du programme pour pouvoir renseigner votre nouvel objet primitif dans les fichiers de configuration. *(Comment créer un fichier de configuration)*. Pour ce faire, rendez-vous dans les fichiers Scene.cpp et Scene.hpp.

Tout d'abord, vous devez ajouter une ligne dans le constructeur de la classe. Cette ligne permet de définir que ce type d'objet primitif existe. Elle permet également de spécifier la fonction que le programme va devoir utiliser. Par exemple, si vous voulez ajouter un Torus, vous devrez ajouter cette ligne:

```
"{"torus", [this](libconfig::Setting &primitive,
std::shared_ptr<Core> core), {createToruses(primitive,core);}"
```

Ensuite, vous allez devoir implémenter une méthode qui va permettre de créer autant de fois l'objet en question qu'il se trouve dans le fichier de configuration.

Par exemple, si vous voulez ajouter un Torus, vous devrez ajouter cette méthode:

```
"void RayTracer::Scene::createToruses(libconfig::Setting
&toruses, std::shared_ptr<Core> core)"
```

Cette méthode devra créer "`toruses.getLength()`" objets Torus. Pour ce faire, vous allez pouvoir réaliser la méthode ci-dessous.

(La méthode "`getLength()`" permet d'obtenir le nombre de fois que l'objet en question est requis dans le fichier de configuration)

Ensuite, vous devrez implémenter une méthode qui permet de créer une instance de votre objet. Par exemple, pour créer un objet Torus, vous devrez réaliser une méthode qui avec ce prototype:

```
“void RayTracer::Scene::createTorus(libconfig::Setting &primitive, std::shared_ptr<Core> core)“
```

Cette méthode va permettre d’ajouter un objet Torus à la liste des objets primitifs (“`_primitives`”). Pour ce faire, vous devrez:

- Récupérer les attributs de l’objet en question (ici le Torus) depuis le fichier en utilisant la libconfig *(Comment Installer la libconfig)*.
- Construire votre objet en appelant le bon constructeur.
- Enregistrer votre objet en utilisant son nom comme clé d’enregistrement dans la liste des objets primitifs.

5. Conclusion

Vous avez désormais toutes les clés en main pour implémenter un nouvel objet primitif pour notre RayTracer, n’hésitez pas à faire une pull-request pour ajouter votre objet si vous souhaitez contribuer au projet. Nous vous remercions par avance de votre contribution.