B-OOP-400 - RayTracer

Groupe :
Mathis EMAILLE - mathis.emaille@epitech.eu
Romain LABED - romain.labed@epitech.eu
Tristan CALARD - tristan.calard@epitech.eu
Vahan DUCHER - vahan.ducher@epitech.eu

Sommaire

Lancer le programme

Pour lancer le programme du raytracer, il y a plusieurs commandes préliminaires à exécuter :
Depuis le chemin suivant : `/[repository_raytracer]/`

Exécuter les commandes suivantes :
- `make fclean`
- `make`

Une fois la compilation terminée, lancer l'exécution avec la commande suivante :
`./raytracer [options] > [chemin_image_de_sortie]`

Les **options** possibles et leur utilité sont les suivantes :
- `-h / --help` : Afficher le menu d'aide

- `-s / --scene [chemin_fichier_de_configuration]` : Préciser le fichier de configuration cible

💡 Au lancement du programme, un fichier de configuration doit être précisé avec ce paramètre

- `-t / --threads [nombre_de_threads]` : Préciser le nombre de threads utilisés pour le rendu

💡 Par défaut, un seul thread est utilisé

- `-g / --gui` : Lancer le programme avec l'interface graphique

- `-q / --quality [nombre_de_samples]` : Préciser le nombre de samples (régler la qualité de l'image)

💡 Par défaut, le nombre de samples est égal à 100

Le **chemin de l'image de sortie** est le chemin du rendu final. Il est donc précisé après une redirection (« > »), car les pixels de l'image sont écrits dans la sortie standard.

💡 Il est inutile de préciser un chemin de sortie si le programme est lancé avec l'interface graphique (`-g / --gui`)

Ajouter une primitive

Pour ajouter une primitive, c'est-à-dire une forme pouvant être affichée dans le rendu, il est important de respecter plusieurs étapes.

- Créer un dossier spécifique pour les fichiers sources et le Makefile

Les fichiers contenant le code source de la primitive doivent être stockés à un endroit spécifique du repository. A partir du chemin `/[repository_raytracer]/plugins/primitives/`, créer un dossier portant le nom de la primitive. Dans ce dossier, créer les 2 dossiers suivant :

- `/[repository_raytracer]/plugins/primitives/[nom_de_la_primitive]/` **`includes`**`/`
- `/[repository_raytracer]/plugins/primitives/[nom_de_la_primitive]/`**`src`**`/`

Chacun de ces 2 dossiers contiendra un fichier source. Le dossier **`includes/`** contiendra le fichier `[nom_de_la_primitive].hpp`, et le dossier **`src/`** le ficher `[nom_de_la_primitive].cpp`.

Au chemin `/[repository_raytracer]/plugins/primitives/[nom_de_la_primitive]/`, il est aussi important de créer un `Makefile` qui permettra la compilation des fichiers sources de la primitive.

💡 : le nom du fichier compilé doit être « *raytracer_**[nom_de_la_primitive]**.so* »

- Appeler la compilation de la primitive

```
all:
	make -C sphere
	make -C plane
	make -C cylinder
	make -C cone
	make -C cube
	make -C wall

clean:
	make clean -C sphere
	make clean -C plane
	make clean -C cylinder
	make clean -C cone
	make clean -C cube
	make clean -C wall

fclean: clean
	make fclean -C sphere
	make fclean -C plane
	make fclean -C cylinder
	make fclean -C cone
	make fclean -C cube
	make fclean -C wall
```

De la même manière que pour les autres primitives, ajouter les lignes permettant la compilation de la primitive dans le fichier `/[repository_raytracer]/plugins/primitives/`**`Makefile.`**

Pour l'instruction **`all`**, on ajoute la ligne « `make -C [nom_de_la_primitive]` »
Pour l'instruction **`clean`**, on ajoute la ligne « `make clean -C [nom_de_la_primitive]`»
Pour l'instruction **`fclean`**, on ajoute la ligne « `make fclean -C [nom_de_la_primitive]`»

- **Remplir les fichiers source de la primitive**

Les deux fichier `.hpp` et `.cpp` doivent être remplis avec le bon code pour permettre le bon fonctionnement et affichage de la primitive lors du rendu.

Créer une classe
Créer une classe portant le nom de la primitive. Cette classe doit être située dans le namespace `RayTracer` et hériter de la classe abstraite `AShape`.

Définir les propriétés de la primitive
Il est important en créant une primitive de définir les propriétés nécessaires à son fonctionnement (rayon, dimensions, axe, …).
Il existe, suivant la propriété définie, des classe ou énumérations spécifiques à cet usage.
Pour définir un point dans l'espace, utiliser la classe `Math::Point3D`.
Pour définir un vecteur dans l'espace, utiliser la classe `Math::Vector3D`.
Pour définir un axis par rapport aux axes (X ; Y ; Z), utiliser l'énumération `ShapeConfig::AXIS`.

Pour permettre l'affichage et la gestion des matériaux, une primitive doit obligatoirement posséder une propriété `std::shared_ptr<RayTracer::Material::IMarterial>` **`_material`**.

Redéfinir les fonctions de la classe mère
Cette classe héritant de la classe `AShape`, héritant elle-même de l'interface `IShape`, certaines fonctions doivent être redéfinies pour cette classe spécifique.
Ces fonctions sont :

```
bool hasAllParameters(const RayTracer::ShapeConfig& config) const override
```

Cette fonction permet de vérifier que lors de la création d'une primitive de ce type, toutes les propriétés qu'elle contient aient été mentionnées. Pour cela, cette fonction se remplit de la manière suivante :

```
if (hasThisParameter(config, "x", "WALL") == false)
    return false;
if (hasThisParameter(config, "y", "WALL") == false)
    return false;
if (hasThisParameter(config, "z", "WALL") == false)
    return false;
if (hasThisParameter(config, "axis", "WALL") == false)
    return false;
if (hasThisParameter(config, "material", "WALL") == false)
    return false;
if (hasThisParameter(config, "width", "WALL") == false)
    return false;
if (hasThisParameter(config, "height", "WALL") == false)
    return false;
return true;
```

La fonction `hasThisParameter()`, définie dans la classe `AShape`, est appelée pour chaque propriété à vérifier. Elle renvoie `false` si la propriété n'a pas été définie dans la configuration (`config`), `true` le cas échéant. Le nom de la primitive concernée est passé en paramètre pour le message d'erreur.

```
void setup(const RayTracer::ShapeConfig& config) override
```

Cette fonction est appelée lors de la création d'une primitive de ce type. C'est elle qui appelle la vérification via la fonction `hasAllParameters()`. Elle va à partir du paramètre `config`, assigner aux propriété de la primitive les bonnes valeurs. Ces valeurs sont stockées dans un `std::map` du paramètre `config`.

Exemple pour la primitive `Cone`:

```
void setup(const RayTracer::ShapeConfig& config) override
{
    if (hasAllParameters(config) == false)
        throw ShapeException("CONE: Missing parameters in config file");
    setName(config._parameters.at("name"));
    _origin = Math::Point3D(atof(config._parameters.at("x").c_str()), atof(config._parameters.at("y").c_str()), atof(config._parameters.at("z").c_str()));
    _axis = getAxisFromString(config._parameters.at("axis"));
    _material = config._loadedMaterials.at(config._parameters.at("material"));
    _radius = atof(config._parameters.at("radius").c_str());
    // _height = atof(config._parameters.at("height").c_str());
    _angle = atof(config._parameters.at("angle").c_str());
    _vertex = Math::Point3D(_origin.x(), _origin.y(), _origin.z() + _height);
}
```

Ici, chaque propriétés de la primitive `Cone` est remplie à l'aide du `std::map` du paramètre `config`.

```
bool hit(const RayTracer::Ray& ray, RayTracer::Range ray_range, HitData& data) const override
```

Cette fonction est appelée en boucle par le `Core`, pour savoir si le rayon de lumière lancé, représenté par le paramètre `ray` passé en référence, a touché la primitive ou pas. Par ce paramètre `ray`, on peut connaitre l'origine du rayon, c'est-à-dire son point de départ, et son vecteur de direction.

Cette fonction renvoie `true` si le rayon a touché, `false` le cas inverse. Les données d'intersection sont décrites dans le paramètre `Data`, passé en référence.
Le paramètre `ray_range` permet de savoir l'intervalle dans laquelle doit se trouver le point d'intersection.

```
void rotate(const Math::Vector3D &rotation) override
```

Cette fonction est appelée lorsque l'utilisateur demande une transformation de type « rotation » sur la primitive. Son comportement est décrit dans cette fonction. La rotation est représentée par le vecteur `rotation` passé en paramètre.

Définir les fonctions pour le fonctionnement de la librairie

```
extern "C" RayTracer::IShape * initShape()
{
    std::cerr << "Sphere entryPoint" << std::endl;
    return new RayTracer::Sphere();
}

extern "C" int getType()
{
    return 0;
}
```

Ces deux fonctions sont nécessaires pour pouvoir charger la primitive sous forme de libraire.
`initShape()` revoit un instance de la primitive sous forme de `IShape *`.
`getType()` renvoie le numéro correspondant à la primitive.
Les numéros déjà assignés sont :

Sphere : 0
Plane : 1
Cone : 2
Cylinder : 3
Cube : 4
Wall : 5

Ajouter les informations aux fichiers externes
Certains fichiers externes ont besoin de certaines informations pour gérer la primitive créée.

La première est dans le fichier `/[repository_raytracer]/includes/Core.hpp.` Ajouter à l'énumération `LIBRARY_TYPE` le nom de la primitive avec le numéro associé (renvoyé par la fonction `getType()`)

```
enum LIBRARY_TYPE
{
    SPHERE = 0,
    PLANE = 1,
    CONE = 2,
    CYLINDER = 3,
    CUBE = 4,
```

Ensuite, dans le fichier `/[repository_raytracer]/src/Parser.cpp`, dans la fonction `getPrimitiveConfig()`. Cette fonction va lire dans le fichier de configuration toutes les propriétés qui lui sont mentionnées. Si la nouvelle primitive créée implémente des propriétés qui ne sont pas mentionnées dans cette fonction, il est nécessaire de les rajouter.

```
void RayTracer::Parser::getPrimitiveConfig(libconfig::Setting &primitive, RayTracer::ShapeConfig& config)
{
    const std::string &name = getPrimitiveName(primitive);
    if (name.empty())
        throw RayTracer::Parser::ParserException("Primitives: Invalid name parameter");
    config._parameters["name"] = name;
    libconfig::Setting &origin = primitive.lookup("origin");
    lookUpDoubleValue(origin, "x", config);
    lookUpDoubleValue(origin, "y", config);
    lookUpDoubleValue(origin, "z", config);
    lookUpStringValue(primitive, "material", config);
    lookUpDoubleValue(primitive, "radius", config);
    lookUpDoubleValue(primitive, "height", config);
    lookUpDoubleValue(primitive, "width", config);
    lookUpDoubleValue(primitive, "angle", config);
    lookUpStringValue(primitive, "axis", config);
    lookUpDoubleValue(primitive, "xDim", config);
    lookUpDoubleValue(primitive, "yDim", config);
    lookUpDoubleValue(primitive, "zDim", config);
}
```

Si cette propriété est un double, utiliser la fonction `lookUpDoubleValue()`. Si c'est un string, utiliser la fonction `lookUpStringValue()`. Ces deux fonctions prennent les même paramètres, à savoir la configuration dans laquelle se trouve le paramètre à lire, le nom du paramètre et une référence vers la `config` pour remplir son `std::map`.

Organiser le fichier de configuration

Chaque scène du programme raytracer est décrite dans un fichier de configuration qui suit un format bien particulier.

Dans ce fichier de configuration, l'utilisateur peut renseigner :
- Les paramètres de la caméra
- Les matériaux utilisés
- Les primitives à afficher
- Les transformations appliquées sur les primitives

Des commentaires peuvent être ajoutés avec le symbole '#'

Le fichier doit être nommé de la manière suivante : « [*nom_de_la_configuration*].scene ».

La caméra

```
camera:
{
    resolution:
    {
        width = 400;
        height = 400;
    }
    position:
    {
        x = 0.0;
        y = 0.0;
        z = 0.0;
    }
    fieldOfView = 45.0; # In degree
    maxDepth = 50;
    focusPoint:
    {
        x = 0.0;
        y = 2.0;
        z = 0.0;
    }
    sceneBackground:
    {
        r = 0.0;
        g = 0.0;
        b = 0.0;
    }
    samples = 1000;
}
```

Les paramètres de la caméra sont décrits de la manière ci-dessus. Ces derniers sont :
- La résolution de l'image (obligatoire)
- La position de la caméra (obligatoire)
- Le FOV (*field of view*) (optionnel, 45° par défaut)
- La profondeur maximale (optionnel, 50 par défaut)
- Le point de focus de la caméra (optionnel, (0 ; 2 ; 0) par défaut)
- L'arrière plan de la scène par défaut (optionnel, (0 ; 0 ; 0) à savoir noir par défaut)
- La qualité avec le nombre des samples (optionnel, 100 par défaut)

Les matériaux

```
materials:
{
    material_ground:
    {
        type = "lambertian";
        color = { r = 255.0; g = 255.0; b = 0.0; }
    }
    material_center:
    {
        type = "lambertian";
        color = { r = 0.0; g = 255.0; b = 255.0; }
    }
    material_left:
    {
        type = "metal";
        fuzziness = 0.0;
        color = { r = 255.0; g = 0.0; b = 0.0; }
    }
    material_right:
    {
        type = "lightDiffuse";
        color = { r = 255.0; g = 255.0; b = 255.0; }
    }
    material_cube:
    {
        type = "lambertian";
        color = { r = 255.0; g = 0.0; b = 0.0; }
    }
}
```

Dans cette section « materials », l'utilisateur doit définir tous les matériaux qui seront utilisés, Un matériau doit être définit ici s'il est utilisé pour une primitive. Pour définir un matériau, il y a 2 ou 3 paramètres à renseigner, en plus de son nom :
- Son nom (le nom de la section. Exemple : « *material_cube* »)
- Son type (3 disponibles : lambertian, metal ou lightDiffuse)
- Sa couleur (au format RGB)
- Pour le matériau « métal » uniquement : le facteur de fuzziness

Les primitives

```
primitives:
{
    # Sphere = 0
    # Plane = 1
    # Cone = 2
    # Cylinder = 3
    # Cube = 4
    # Wall = 5

    sphereLight1:
    {
        type = 0;
        origin = { x = 0.0; y = 20.0; z = 0.0; };
        material = "material_right";
        radius = 2.0;
    }
    sphereLight2:
    {
        type = 0;
        origin = { x = -20.0; y = 2.0; z = 50.0; };
        material = "material_right";
        radius = 2.0;
    }
```

La section « primitives » contient toutes les primitives (ou les formes) à afficher dans le rendu.
Pour déclarer une primitive :

- Assigner un nom (nom de la section. Exemple : « *sphereLight1* »)
- Définir un type à l'aide d'un numéro (c'est le numéro renvoyé par la fonction `getType()` définie dans le fichier source de la primitive).
- Assigner un matériau. Une primitive doit absolument contenir un matériau pour être affichée. Le matériau est assigné à la primitive à l'aide de son nom, définit dans la section « materials ».
- Définir toutes les propriétés nécessaires pour la primitive.

💡 : les propriétés de type vecteur, point ou code couleur peuvent être définies dans des sous-sections, comme la propriété « origin » par exemple.

Les transformations

```
transformations:
{
    qdheuxh: { type = "translate" ; vector: { x = 12.0 ; y = 67.0 ; z = 34.0 } } # a translation transfo. applied to sphere1
    cylinder1: { type = "rotate" ; vector: { x = 0.0 ; y = 0.0 ; z = 45.0 } } # a rotation transfo. applied to sphere2
}
```

La section « transformations » contient toutes les transformations qui seront appliquées à une primitive spécifique. Une transformation est définie de la manière suivante :

- Le nom de la primitive associée (nom de la section. Exemple : « *cylindre1* »). Ça doit être une primitive définie dans la section « primitives ».
- Le type (2 disponibles : « rotate » et « translate »)
- Le vecteur de transformation

Gérer les lumières

Pour ajouter une lumière à la scène, il n'y pas de section à proprement parler dans le fichier de configuration. Une lumière est en fait définie à l'aide d'une primitive et d'un matériau. Lors de la définition des matériaux, un type de matériau est disponible et s'appelle « lightDiffuse ». Une couleur peut également lui être associé.
Pour ajouter une lumière à la scène, il suffit de créer une primitive en lui associant ce matériau de lumière. Suivant la primitive choisit ainsi que ses paramètres, cela vient définir les rayons de direction de la lumière.

Exemple :

```
material_right:
{
    type = "lightDiffuse";
    color = { r = 255.0; g = 255.0; b = 255.0; }
}

sphereLight1:
{
    type = 0;
    origin = { x = 20.0; y = 0.0; z = 0.0; };
    material = "material_right";
    radius = 2.0;
}
```

Ici, le matériau « material_right » est un matériau de type « lightDiffuse », de couleur blanche, qui diffuse donc de la lumière. Il est associé à une primitive de type « sphère », qui permet ainsi la diffusion de la lumière dans de multiples directions.