Programmiersprachen
Sommersemester 2019

Aufgabensammlung 5

Die Aufgaben werden am **01. Juli** in der Übung bewertet. Diese Aufgabensammlung beschäftigt sich mit Vererbung in C++ und git.
Es gelten die Ausführungshinweise der vorherigen Aufgabenblätter (`const`-Korrektheit, member-initialization-list, Header/Source, CMakeLists, catch.hpp ...). Commiten Sie nach jedem implementieren Feature. Nutzen Sie neben dem Vorlesungsskript ausschließlich aktuelle Fachliteratur oder Online-Referenzen, z.B.

- Stroustrup, B.: Einführung in die Programmierung mit C++ (2010)
- http://en.cppreference.com/
- http://www.cplusplus.com/

Committen Sie Ihren Code nach jeder Aufgabe!
Bei Fragen und Anmerkungen schreiben Sie bitte eine Email an adrian.kreskowski@uni-weimar.de .

Aufgabe 5.1

Erstellen Sie sich einen Fork von folgendem Repository: https://github.com/vrsys/programmiersprachen-raytracer.
Clonen Sie anschließdas Repository mit `git` und konfigurieren es wie gewohnt mit `cmake`. Alle Ihre Klassen fügen Sie bitte zum Ordner `framework` hinzu. Ihre Tests schreiben Sie in die Datei `source/tests.cpp`. Testen Sie auch immer Randfälle! Erstellen Sie mindestens einmal nach jedem implementierten Feature einen Commit.
Erstellen Sie eine Klasse `Shape` als abstrakte Basisklasse für geometrische Objekte. Die Klasse-`Shape` hat die folgenden rein virtuellen (*pure virtual*) Methoden:

- `area` - berechnet die Oberfläche des Objekts
- `volume` - berechnet das Volumen des Objekts

[5 Punkte]

Aufgabe 5.2

Leiten Sie nun die Klassen `Sphere` und `Box` von `Shape` ab. Die Klasse `Sphere` besitzt einen Mittelpunkt vom Typ `glm::vec3` und einen Radius. Die achsenparallele (axis-aligned) Box besitzt ein Minimum und ein Maximum vom Typ `glm::vec3`.
Um glm-Vektoren zu benutzen inkludieren Sie den Header

```
#include <glm/vec3.hpp>
```

Implementieren Sie geeignete Konstruktoren und die Methoden `area()` und `volume()`. Testen Sie alle Methoden.

[10 Punkte]

Aufgabe 5.3

Erweitern Sie die Basisklasse `Shape` um die Attribute `name_` und `color_` und passen Sie die Konstruktoren entsprechend an. Beachten Sie, dass die Konstruktoren der abgeleiteten Klassen ebenfalls angepasst werden sollten und die Basisklasse korrekt initialisieren.

Hinweis: Die Initialisierung der Basisklasse erfolgt in der Initialisierungsliste des Konstruktors der abgeleiteten Klasse! Wo wird der Konstruktor der Basisklasse aufgerufen?

[10 Punkte]

Aufgabe 5.4

Implementieren Sie eine virtuelle Methode `Shape::print` für die Ausgabe von Objekten des Typs `Shape` und Überladen Sie den Stream-Operator `<<`, welcher `print` verwendet (siehe Slides zu Klassenmechaniken).

```
class Shape
{
public:
  //...
  virtual std::ostream& print(std::ostream& os) const;

  //...
};

std::ostream& operator<<(std::ostream& os, Shape const& s)
{
```

```
  // not implemented yet
}
```

[10 Punkte]

Aufgabe 5.5

Überschreiben Sie die Methode `print` für die abgeleiteten Klassen. Erzeugen Sie Objekte vom Typ `Sphere` und `Box` und geben Sie diese mit Hilfe des Out-Stream-Operators auf der Konsole aus.
Erklären Sie den Effekt des Schlüsselworts `override` im Kontext der Vererbung! Benutzen Sie das Schlüsselwort dementsprechend in Ihrer Methodendeklaration. Was passiert, wenn Sie das Schlüsselwort weglassen?

[12 Punkte]

Hinweis: Um den Namen und die Farbe auszugeben, sollte die Methode `Shape::print` explizit in der überschriebenen Methode aufgerufen werden.

Aufgabe 5.6

Fügen Sie folgenden Testcase zur Datei `tests/main.cpp` hinzu.

```
#include <glm/glm.hpp>
#include <glm/gtx/intersect.hpp>

TEST_CASE("intersect_ray_sphere", "[intersect]")
{
  // Ray
  glm::vec3 ray_origin{0.0f, 0.0f, 0.0f};1
  // ray direction has to be normalized !
  // you can use:
  //   v = glm::normalize(some_vector)
  glm::vec3 ray_direction{0.0f, 0.0f, 1.0f};

  // Sphere
  glm::vec3 sphere_center{0.0f ,0.0f, 5.0f};
  float sphere_radius{1.0f};

  float distance = 0.0f;
  auto result = glm::intersectRaySphere(
      ray_origin, ray_direction,
      sphere_center,
      sphere_radius * sphere_radius, // squared radius !!!
```

```
      distance);
  REQUIRE(distance == Approx(4.0f));
}
```

Fügen Sie einen neuen Datentypen `Ray` (DTO) zum Framework hinzu. Ein Ray-Objekt kodiert lediglich den Ursprung und die Richtung des Strahls im dreidimensionalen Raum.
Das Strahl-Objekt ermöglicht Ihnen über die parametrische Form eines Strahls unter Angabe einer skalaren Distanz t einen Punkt entlang des Strahls zu ermitteln.

$$\overrightarrow{p(t)} = \overrightarrow{o} + t * \overrightarrow{d} \tag{1}$$

```
struct Ray
{
  glm::vec3 origin    = {0.0f, 0.0f,  0.0f};
  glm::vec3 direction = {0.0f, 0.0f, -1.0f};
};
```

Erweitern Sie die Klasse Sphere um eine Methode `intersect` und implementieren Sie diese mit der Funktion `glm::intersectRaySphere`. Ihre Funktion soll ein Objekt vom Typ *Ray* entgegen nehmen, gegen die Kugel schneiden. Wenn ein Schnitt erfolgt ist, sollen folgende Attribute zurückgegeben werden:

1. ob ein Schnitt erfolgt ist
2. in welcher Distanz der Schnitt erfolgt ist (Parameter t)
3. den Namen des geschnittenen Objekts
4. die Farbe des geschnittenen Objekts
5. den 3D Punkt, an dem das Objekt geschnitten wurde
6. die Richtung, mit dem der Strahl das Objekt geschnitten hat

Legen Sie sich zur Rückgabe dieser Eigenschaften einen geeigneten Datentypen namens `HitPoint` in einer neuen Datei an. Überlegen Sie, ob Sie HitPoint als Klasse oder als Datentransferobjekt modellieren sollten. Wenn Sie bei der Entscheidung unsicher sind, nehmen Sie die Folien zum Klassendesign als Referenz!
Testen Sie Ihre Schnittmethode ausführlich mit verschiedenen Kugeln und für die verschiedenen Fälle.

Hinweis: Die Strahlrichtung muss normalisiert sein!
glm-Reference:
https://glm.g-truc.net/0.9.5/api/a00203.html#ga7773a235a18acb4f06cfe358d4453375

[5 Punkte]

Aufgabe 5.7

Sehen Sie sich folgenden Beispielcode an:

```
Color red{255, 0, 0};
glm::vec3 position{0.0f, 0.0f, 0.0f};

std::shared_ptr<Sphere> s1 =
    std::make_shared<Sphere>(position, 1.2f, red, "sphere0");
std::shared_ptr<Shape> s2 =
    std::make_shared<Sphere>(position, 1.2f, red, "sphere1");

s1->print(std::cout);
s2->print(std::cout);
```

Erklären Sie anhand des Beispiels die Begriffe „Statischer Typ einer Variablen“ und „Dynamischer Typ einer Variablen“.
Wann wird welche Art des Typs überprüft? Was sind die dynamischen und die statischen Typen der Variablen `s1` bzw `s2`?

[5 Punkte]

Aufgabe 5.8

In dieser Aufgabe geht es um das Schlüsselwort `virtual`. Deklarieren Sie den Destruktor der `Shape`-Klasse als virtuell.

```
Color red{255, 0, 0};
glm::vec3 position{0.0f, 0.0f, 0.0f};

Sphere* s1 = new Sphere{position, 1.2f, red, "sphere0"};
Shape*  s2 = new Sphere{position, 1.2f, red, "sphere1"};

s1->print(std::cout);
s2->print(std::cout);

delete s1;
delete s2;
```

Geben Sie im Funktionsrumpf der Kon- und Destruktoren der Klassenhierarchie deren Aufruf auf der Konsole aus.

Kompilieren Sie den gegebenen Programmcode (am besten innerhalb eines Testcases). In welcher Reihenfolge werden Konstruktoren und Destruktoren aufgerufen?

Entfernen Sie nun das Schlüsselwort `virtual` vom Destruktor der Basisklasse, testen Sie erneut und erklären Sie den Unterschied.

[10 Punkte]

Aufgabe 5.9

Erklären Sie die Unterschiede zwischen Klassenhierarchie vs. Objekthierarchie - Klassendiagramm vs. Objektdiagramm.

[5 Punkte]

Bei Fragen und Anmerkungen schreiben Sie bitte eine Email an adrian.kreskowski@uni-weimar.de .