Programmiersprachen
Sommersemester 2015

Aufgabensammlung 6

Die Aufgaben werden am **29. Juni** 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 ...). Die Benutzung von `git` stellen wir in der Übung vor. 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/

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

Aufgabe 6.1

Erstellen Sie sich einen Account bei https://github.com. Forken Sie sich das Repository https://github.com/vrsys/programmiersprachen-raytracer. Anschließend clonen Sie das Repository mit `git` und konfigurieren es mit `cmake`. Alle ihre Klassen fügen Sie bitte zum Ordner `framework` hinzu. Im Ordner `tests` finden Sie ein Programm welches Sie bitte mit ihren Tests erweitern. Erstellen Sie nach jedem implementierten Feature einen Commit.
Stellen Sie eine Klasse `Shape` als abstrakte Basisklasse für geometrische Objekte bereit. Die Klasse-`Shape` hat die folgenden rein virtuellen (*pure virtual*) Methoden:

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

[5 Punkte]

Aufgabe 6.2

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 Box besitzt ein Minimum und ein Maximum vom Typ `glm::vec3`. Includieren Sie dazu den Header

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

Implementieren Sie geeignete Konstruktoren, *get*-Methoden, die Methode `area` und die Methode `volume`. Testen Sie diese.

[10 Punkte]

Aufgabe 6.3

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

Hinweis: Die Initialisierung der Basisklasse erfolgt in der Initialisierungsliste des Konstruktors der abgeleiteten Klasse! **[10 Punkte]**

Aufgabe 6.4

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

```
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 6.5

Überschreiben Sie die Methode `print` für die abgeleiteten Klassen. Erzeugen Sie Objekte vom Typ `Sphere` und `Box` und geben Sie diese mit Hilfe des Stream-Operators auf der Konsole aus. **[10 Punkte]**

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

Aufgabe 6.6

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

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

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

  // Sphere
  glm::vec3 sphere_center(0.0,0.0,5.0);
  float sphere_radius(1.0);

  float distance(0.0);
  auto result = glm::intersectRaySphere(
      ray_origin, ray_direction,
      sphere_center, sphere_radius,
      distance);
  REQUIRE(distance == Approx(4.0f));
}
```

Fügen Sie ein struct Ray (DTO) zum Framework hinzu.

```
struct Ray
{
  glm::vec3 origin;
  glm::vec3 direction;
};
```

Erweitern Sie die Klasse Sphere um eine Methode `intersect` und implementieren Sie diese mit der Funktion `glm::intersectRaySphere`. Testen Sie die Methode.

Hinweis: Die Strahlrichtung muss normalisiert sein! **[10 Punkte]**

Aufgabe 6.7

Sehen Sie sich folgenden Beispielcode an:

```
Color red(255, 0, 0);
glm::vec3 position(0,0);

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

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

Erklären Sie anhand des Beispiels die Begriffe „Statischer Typ einer Variable“ und „Dynamischer Typ einer Variable“. **[5 Punkte]**

Aufgabe 6.8

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(0,0);

Sphere* s1 = new Sphere(position, 1.2, red, "sphere0");
Shape* s2  = new Sphere(position, 1.2, 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 (in einem Testcase am besten). 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 6.9

Erklären Sie die Unterschiede zwischen Klassenhierarchie vs. Objekthierarchie - Klassendiagramm vs. Objektdiagramm. **[5 Punkte]**

Aufgabe 6.10

Erweitern Sie die Klasse Box um eine Methode `intersect` und testen Sie diese. **[10 Punkte, optional]**

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