C++ Programmierung für Computergraphik

Übungsblatt 4: Aufbau einer modularen Spiele-Engine

Nachdem wir in der letzten Übung den Grundstein für unser Spiel gelegt haben, geht es in diesem Übungsblatt darum Ihr Programm zu einer leichtgewichtigen, modularen und leicht benutzbaren Engine auszubauen, beziehungsweise zu abstrahieren. Das erste Ziel besteht darin, die Engine modular zu gestalten, um neue Aspekte und Funktionalitäten leicht integrierbar zu machen. Dies sind im konkreten Fall die graphische Ansicht (`view::GlRenderer`), die Soundausgabe (`view::AlRenderer`) und die Spiellogik (`controller::Logic`) aber auch Netzwerkunterstützung und Ähnliches wäre denkbar.

Die unterschiedlichen Module sollen entsprechend des Model-View-Controller[1](MVC) Design-Patterns miteinander interagieren. Der Vorteil dieser Herangehensweise ist, dass die Gesamtkomplexität beschränkt bleibt, da jede Komponente klare Verantwortlichkeiten besitzt (Datenverwaltung im Modell, Interpretation und Darstellung der Daten durch die Renderer sowie Abbildung der Spielelogik im Controller). Durch die Entkopplung der einzelnen Aspekte einer Software gewinnt der Quelltext oft erheblich an Wartbarkeit und Verständlichkeit.

Wie im Pattern angedacht, sollten die Komponenten im Namespace `view::` nur lesend auf Klassen im Namespace `model::` zugreifen können, während Klassen im Namespace `controller::` Lese- und Schreibzugriff auf das Modell erlaubt ist.

Neben der Klasse `controller::GlutEngine`, in der alle Komponenten(, Delegierte[2] und das Modell) miteinander verbunden werden, sind die Klassen `model::Game` und `model::GameObject` von zentraler Bedeutung, da diese den Zustand des Spieles und der darin enthaltenen Objekte speichern und den anderen Komponenten zur Verfügung stellen. Die Klassen `view::GlRenderer`, `view::AlRenderer` und `controller::Logic` interpretieren beziehungsweise aktualisieren das Modell lediglich.

VORBEREITUNGEN

- Für die Vektor-/Matrixalgebra nutzen Sie bitte ihre Vektorklasse aus dem Aufgabenblatt 2 oder die Bibliothek „Eigen 3“[3]. Definieren Sie in der Datei `math.hpp` einen entsprechenden Typ `vec3_type` zur Repräsentation von Vektoren (siehe Quelltextausschnitt 1).

```
# include <eigen3/Eigen/Core>
# include <eigen3/Eigen/Dense>

template< typename T, unsigned int Dim >
using vec_type = Eigen::Matrix< T, Dim, 1 >;

typedef vec_type< double, 3 > vec3_type;
```

Quelltextausschnitt 1: Auf Eigen 3 basierende Definition des Typs `vec3_type`.

- Diese Übung baut auf den Klassen `controller::Engine` sowie `view::Window` aus den Aufgaben 3.2 und 3.3 auf. Die Klasse `view::Window` kann weiter verwendet werden, sollte jedoch in

1 MVC Pattern: de.wikipedia.org/wiki/Model_View_Controller
2 Delegate/Delegierter: de.wikipedia.org/wiki/Delegierter
3 Eigen 3 - Linear Algebra: eigen.tuxfamily.org

`view::GlutWindow` umbenannt werden, um die Abhängigkeit von der GLUT-Bibliothek zu verdeutlichen. Stellen sie sicher, dass die Semantik der Methoden übereinstimmt, falls sie ihre eigenes `GlutWindow` verwenden.

- Zur Steuerung der Soundausgabe sollen die die Bibliotheken OpenAL und freealut[4] zum Einsatz kommen, deren APIs weitestgehend analog zu OpenGL/GLUT strukturiert sind. Um die OpenAL und (free-)ALUT Bibliotheken benutzen zu können, inkludieren Sie den Header `AL/alut.h`.

  Die notwendigen Linkerspezikationen sollten bereits in ihrer `CMakeLists.txt` vorhanden sein. Es sollte daher genügen die Softwarepakete zu installieren.

- Die Aufgaben 4.1, 4.3 und 4.4 sollen durch Vervollständigung des mitgelieferten Quelltexts gelöst werden. Für jede dieser Aufgaben ist bereits eine seperate `main{41,43,44}.cpp` und ein Programm (`ex41`, `ex43`, `ex44`) in dem Projekt vorbereitet. Für die Aufgabe 4.4 müssen sie die von ihnen erstellten `.cpp` Dateien für die Klassen im Namespace `flappy_box::` zu dem Programm `ex44` hinzufügen.

- Halten sie Verzeichnisse und Namespaces sowie Datei und Klassennamen stets konsistent! Beispielsweise liegen die Dateien zu Klassen im Namespace `view::` im Verzeichnis `{include,src}/view/` und tragen einen vom Klassennamen abgeleiteten Dateinamen. Die Klasse `::flappy_box::model::Box` wird so zum Beispiel in der Datei `include/flappy_box/model/box.hpp` deklariert und in `src/flappy_box/model/box.cpp` definiert.

- Kommunizieren sie bei offenen Fragen miteinander (auch zwischen den Gruppen)!

Das Modell, Komponenten und deren Delegierte

Die Klasse `model::Game` stellt entsprechend des MVC Patterns die Schnittstelle zwischen dem Controller und den Ansichten dar. Es besteht im Grunde nur aus einer Liste von Pointern auf Objekte des abstrakten Typs `model::GameObject`. In der Implementierung eines Spieles werden die Spielobjekte von `model::GameObject` abgeleitet und mit den benötigten Zustandsvariablen versehen (siehe Aufgabe 4.4).

Die einzelnen Komponenten identifizieren über den RTTI-Mechanismus[5] den abgeleiteten Typ des jeweiligen `model::GameObject` und wählen eine passende „Interpretation“ (bzw. Delegierte-Klasse), um das Spielobjekt zu verarbeiten.

Im speziellen Fall des `view::GlRenderers` werden wir zum Beispiel den abgeleiteten Typ eines `model::GameObject` nutzen, um das passende Objekt der abstrakten Klasse des Delegierten dieser Komponente (eine von `view::GlRenderer::Drawable` abgeleitete Klasse) zurück zu liefern. Der gleiche Mechanismus wird auch für die Komponenten `view::AlRenderer` und `controller::Logic` benötigt.

Diese Zuordnung zwischen dem abgeleiteten Typ eines Spielobjekts und dem Delegierten für die Komponente, kann zum Beispiel durch die in Aufgabe 4.2 behandelte Klasse erfolgen.

4 OpenAL: openal.org
freealut: raw.githubusercontent.com/vancegroup/freealut/master/doc/alut.html
5 C++ Runtime Type Identification: en.wikibooks.org/wiki/C%2B%2B_Programming/RTTI

Aufgabe 4.1 – Umbau der Engine (3 Punkte)

Zunächst widmen wir uns jedoch dem Feinschliff der Klassen `controller::Engine`, `controller::GlutEngine` und `view::GlutWindow`.

Eine erste wichtige Änderung, stellt die Weiterleitung der Eingabeereignisse aus dem `view::GlutWindow` an einen sogenannten Event-Handler dar.

- Um hier keine Flexibilität einzubüßen, definieren wir eine abstrakte Klasse `controller::InputEventHandler` von der dann jede Klasse vererbt werden kann, welche Eingabeereignisse verarbeiten soll. Dieses Interface soll dazu eine rein-virtuelle Funktion `bool handle(keyboard_event const&)` bereitstellen, die später von der abgeleiteten Handler-Klasse überschrieben werden muss. (Der Rückgabewert soll angeben, ob das Event verarbeitet wurde)
- Die Struktur `controller::InputEventHandler::keyboard_event` soll die Eingabeereignisse abbilden und dabei unabhängig von GLUT bleiben, damit z.B. andere Window-Toolkits die `keyboard_events` befüllen können.

  Um die von GLUT bereitgestellten Informationen zu den Ereignissen zu speichern, benötigen wir einen Member `key` vom Typ `char`, `modifier_mask` vom Typ `int` sowie `mouse_pos` (Array aus zwei `double` Werten).

  Wie der Name andeutet soll `modifier_mask` als sogenannte Bitmaske genutzt werden, um den Zustand der Tasten (SHIFT, CTRL, ALT) darzustellen. Definieren die hierzu einen Enumerations-Typ der folgende Einträge enthält: `{SHIFT_ACTIVE=1, CTRL_ACTIVE=2, ALT_ACTIVE=4}`.

Die `view::GlutWindow` Klasse soll nunmehr ausschließlich das angezeigte Fenster repräsentieren. Die Anzeige und Tastatur-Ereignisse werden an Objekte vom Typ `view::GlRenderer` beziehungsweise `controller::InputEventHandler` weitergeleitet:

- Fügen sie der Window Klasse `shared_ptr` auf `view::GlRenderer` bzw. `controller::InputEventHandler` hinzu, die über entsprechende Argumente des Konstruktors initialisiert werden können.
- Rufen sie in den Funktionen `glutReshape`, `glutDisplay` und `glutKeyboard` die Funktionen `visualize_model` und `resize` in dem Objekt der Klasse `view::GlRenderer` sowie `controller::InputEventHandler::handle` aus. Beachten sie, dass sie zuerst ein `keyboard_event`-Objekt erzeugen müssen, das dann an die `handle` Methode übergeben wird.

Die `controller::Engine` Klasse enthält bereits Pointer-Referenzen auf die Spielelogik und das Modell. Die virtuelle `step` Funktion wird später in kurzen Zeitabständen durch die Klasse `controller::GlutEngine` aufgerufen, um das Spiel mithilfe des `controller::Logic` Objektes voranzutreiben. Abgeleitete Klassen können diese Funktion überschreiben, um neues Verhalten zu integrieren (wie etwa in Aufgabe 4.4).

Jetzt implementieren wir die Ereignisverarbeitung in der `controller::Engine` Klasse:

- Zunächst müssen wir von `controller::InputEventHandler` erben, damit die Klasse später als Event-Handler an das `view::GlutWindow` übergeben werden kann.

- In der Implementierung der Funktion `handle` soll lediglich die Funktion `step` mit dem übergebenen `keyboard_event` aufgerufen werden.

Beachten sie das durch dieses Vorgehen bei jeder Benutzereingabe zusätzliche Schritte im Spielablauf eingefügt werden. Da wir später jedoch alle zeitabhängigen Vorgänge von der Länge des letzten Zeitschritts (`model::Game::timestep`) oder dem Zeitstempel des Modells (`model::Game::timestamp`) abhängig machen, sollte dies keine Probleme verursachen.

Da die Fenster später in einer `Engine::run` überschreibenden Funktion erzeugt werden sollen, entsteht jedoch ein anderes Problem: Da der Konstruktor der Klasse `GlutWindow` einen `shared_ptr<InputEventHandler>` erwartet, benötigt die `run`-Methode den `shared_ptr` der den eigenen `this`-Pointer verwaltet. Diese Aufgabe kann mithilfe einer Klasse aus der Standardbibliothek gelöst werden:

- Fügen sie die Basisklasse `std::enable_shared_from_this<controller::Engine>` zur Klasse `controller::Engine` hinzu.

Dadurch erbt `controller::Engine` die Methode `shared_from_this` welche einen `shared_ptr<controller::Engine>` bereitstellt, der den `this`-Pointer des aufrufenden Objektes verpackt.

Aufgabe 4.2 – Eine Factory-Klasse zur Erzeugung von Delegierten (3 Punkte)

Studieren sie die `factory_map` Klasse in Quelltextausschnitt 1 bzw. der Datei `factory_map.hpp` im Detail. Lesen sie die Dokumentation der beteiligten STL-Funktionen[6]. Um die Funktion zu verstehen konzentrieren sie sich zuerst auf die Funktionsprototypen, um zu verstehen wie die Klasse benutzt werden kann. Schauen sie sich das Beispielprogramm in Quelltextausschnitt 3 an.

- Erklären sie den Tutoren welche Typen die Argumente und Rückgabewerte der beteiligten Funktionen besitzen und wie die Klasse funktioniert.
- Wie kann die Klasse helfen das auf Seite 2 beschriebene Problem zu lösen, den Spielobjekten die passenden Delegierten zuzuordnen?
- Stehen die Typen `InBase`, `Out` und `In` (in `register_module`) miteinander in Beziehung? Wenn ja, in welcher?
- Welchen Prototyp (Signatur) muss das aufrufbare Objekt zur Erzeugung der Delegiertenklasse (in Abhängigkeit von den Template-Parametern) anbieten, sodass es an die Funktion `register_module` übergeben werden kann?
- Geben sie jeweils den Aufruf der `register_module`-Methode an, der folgende Konstrukte zur Erzeugung des Ausgabetyps (Templateargument `Out`) anbindet: Eine statische Funktion, eine Membermethode und eine Klasse die einen Aufruf-Operator besitzt (einen Funktor). Definieren sie entsprechende Beispielkonstrukte. Sie können dabei zur Vereinfachung `Out` und `In` durch beliebige Typen ersetzen.
- Die in der Variable `outer_function` gespeicherte Lambda-Funktion enthält einen `std::static_pointer_cast`. Diese Art von Typkonvertierung sollte für polymorphe Objekte normaler-

6 C++ Type Support: en.cppreference.com/w/cpp/types (insb. `std::type_index`)
std::function: en.cppreference.com/w/cpp/utility/functional/function
std::map: en.cppreference.com/w/cpp/container/map

weise vermieden werden, da `static_casts` nicht überprüfen, ob der verlangte Typ mit dem abgeleiteten Typ des Objektes übereinstimmt. Können sie sich vorstellen warum hier dennoch ein `static_cast` genutzt wurde?

AUFGABE 4.3 – NUTZUNG DER DELEGIERTEN (3 PUNKTE)

Die Funktionalität in den Komponenten wird, wie in den der vorhergehenden Aufgabe beschrieben, anhand des abgeleiteten Typs der `model::GameObjects` zugeordnet. Daher muss für jede von `model::GameObject` abgeleitete Klasse eine Delegierter für jede Komponente (welche das Spielobjekt interpretieren soll) bereitgestellt werden. Die Komponenten und ihre Delegierten-Basisklassen, die in ihrem Spiel genutzt werden sollen, sind in der unten stehenden Tabelle 1 aufgelistet.

Komponente:	`controller::Logic`	`view::GlRenderer`	`view::AlRenderer`
Funktionalität:	Spielelogik	Grafikausgabe (mit OpenGL)	Soundausgabe (mit OpenAL)
Aufrufende Methode in Aufgabe 4.4:	`controller::Engine::step`	`view::GlutWindow::glutDisplay`	`flappy_box::controller::FlappyEngine::step`
Basisklasse für Delegierte:	`controller::Logic::ObjectLogic`	`view::GlRenderer::Drawable`	`view::AlRenderer::Audible`
Zugriffsrechte auf das Modell:	lesend/schreibend	lesend	lesend
Virtuelle Hauptfunktion der Delegierten:	`advance_model( Logic&, const keyboard_event& );`	`visualize_model( GlRenderer&, GlutWindow& );`	`auralize_model( AlRenderer& );`

Tabelle 1: Komponenten der Engine mit Zugriff auf `model::Game`.

In dieser Aufgabe sollen Sie die Methoden `controller::Logic::advance_model`, `view::AlRenderer::auralize_model` und `view::GlRenderer::visualize_model` implementieren.

- Setzen Sie zu Beginn der Methode `Logic::advance_model` als erstes den Zeitstempel des Modells auf die aktuelle Zeit. Nutzen sie die entsprechende Methode des Modells, die auch den Wert für den aktuellen Zeitschritt aktualisiert.
- Initialisieren Sie ihren OpenGL Kontext (`glClearColor`, `glClear`, `glMatrixMode`, `glLoadIdentity`, `gluLookAt`[7]) zu Beginn der Funktion `view::GlRender::visualize_model` und vergessen sie das Tauschen der Bildpuffer an deren Ende nicht.
- Führen Sie die Methoden `advance`/`auralize`/`visualize` der entsprechenden Delegierten für alle Objekte im Modell aus. Es gibt mindestens drei Möglichkeiten diese Delegierten zu verwalten:
    1. Erzeugung und Aufruf der Delegierten bei jedem Funktionsaufruf.
    2. Durch Nutzung der Funktionen `model::GameObject::getData` und `model::GameObject::registerData`[8] kann zunächst geprüft werden, ob ein Delegierter im `Game-`

7 Nutzen sie die selbe Kamerakonfiguration wie in Aufgabe 3.4.
8 Mit diesen Funktionen ist es möglich einen von `model::GameObject::Data` abgeleitetes Objekt zu speichern und durch Angabe des abgeleiteten Typs wieder abzurufen.

`Object` abgelegt wurde, falls nicht, legen Sie einen ab. Der so erlangte Delegierte wird dann ausgeführt.

3. Am effizientesten ist die Möglichkeit, dass die Komponenten jedes `model::GameObject` einmal durchlaufen und die Delegierten in ihnen ablegen. Die Methoden {`advance`,`auralize`, `visualize`}`_model` überprüfen dann nur noch, ob ein Delegierter für die Spielobjekte abgelegt wurde und führen diesen gegebenenfalls aus.

Erläutern sie einige markante Unterschiede und zählen sie mindestens einen Vor- und Nachteil für jeden der Ansätze auf?

- Nutzen sie den Quelltextausschnitt 3 als Inspiration, um ein kleines Testprogramm zu schreiben. Testen sie Ihre Komponenten mithilfe von Behelfsklassen. Erzeugen sie eine von `model::GameObject` abgeleitete Klasse als Test-Spielobjekt. Die Behelfsklassen für die Delegierten (die von den jeweiligen Delegierten-Basisklassen aus Tabelle 1 ableiten), sollten in den Funktionen `advance`, `auralize` bzw. `visualize` vorerst einfachen Text (z.B. `"Test Thinking!"`, `"Test Sound!"` und `"Test Image!"`) auf die Konsole ausgeben.

AUFGABE 4.4 – INTEGRATION EINER EINFACHEN SPIELLOGIK (4 PUNKTE)

Nachdem nun alle Teile der Basis-Engine komplett sind kommt der Moment der Wahrheit, fast schon ein richtiges Spiel. Die Idee hinter “flappy box“ ist simpel: Ein rotierender Würfel muss durch leichtes Anstoßen (ausgelöst durch das Drücken einer Taste) am herunter fallen gehindert werden. Ziel des Spiels ist es, die Kiste möglichst im Zentrum des Spielfeldes zu halten.

Ein wichtiger Aspekt wird die Multimodalität des Spielkonzeptes sein, das bedeutet das Spielziel kann sowohl über das graphische, als auch das akustische Feedback erreicht werden. Das modulare Konzept der Engine ermöglicht dabei ein sauber strukturiertes Vorgehen.

Zuerst müssen Sie mit der Klasse `flappy_box::model::Box` das Datenmodell erstellen:

- Definieren sie eine Klasse `::flappy_box::model::Box`, welche unser vorerst einziges Objekt im Spiel sein wird. Definieren sie in der Klasse Vektoren für Position [m], Geschwindigkeit [m/s] und Beschleunigung [m/s²] sowie einen Winkel [deg] die jeweils über Methoden ausgelesen und geändert werden können. Initialisieren sie diese Variablen mit Null(-vektoren).

Danach sollen die folgenden Delegierten-Klassen definiert werden:

- **`::flappy_box::controller::BoxObjectLogic`** – Delegierter von `::controller::Logic`

  Hier soll sowohl eine einfache Physiksimulation als auch die Benutzerinteraktion für das `Box`-Spielobjekt implementiert werden.

  - Dazu müssen wir die im Spielobjekt gespeicherte Position und Geschwindigkeit entsprechend der Newtonschen Bewegungsgleichung aktualisieren. Die Masse des Spielobjektes kann dabei `1`[kg] gleichgesetzt werden. Ein ausreichendes (aber für große Zeitschritte ungeeignetes Verfahren) um die Bewegungsgleichung zu integrieren ist das Euler Verfahren[9]. Nutzten sie bei der Umsetzung den von `model::Game::timestep` bereitgestellten Zeitschritt[10]. Die Beschleunigung in der Bewegungsgleichung soll

9 Euler Verfahren: de.wikipedia.org/wiki/Explizites_Euler-Verfahren

sich als Summe einer konstanten Gravitationskomponente (in negativer Richtung auf der z-Achse) und der benutzerdefinierten Beschleunigung im Spielobjekt ergeben.

Mit den derzeitigen Einstellungen sollte die Kiste mit linear steigender Geschwindigkeit herab fallen.

Nun ist die Interaktion mit dem Nutzer an der Reihe. Durch einen Tastaturdruck soll ein Kraftimpuls in Richtung der positiven Z-Achse auf das Spielobjekt angewendet werden:

- Werten sie das an die `ObjectLogic::advance` übergebene Tastaturereignis aus. Wurde die Taste `'w'` gedrückt, soll der Z-Wert der benutzerdefinierten Beschleunigungskomponente auf `20`[m/s] gesetzt werden.
- Um bei der Integration unabhängig vom Zeitschritt ein konsistentes Ergebnis zu erhalten, ist es notwendig den Beschleunigungswert als Funktion der Zeit abklingen zu lassen (und nicht gleich wieder auf `0`[m/s] zu setzen). Als entsprechende Vorschrift bietet sich hier ein exponentieller Zerfall[11] an.
- Welches Problem kann auftreten, wenn der letzte Punkt nicht beachtet wird?
- Sollte der Abfall der Beschleunigung vor oder nach der Verarbeitung des Tastaturereignisses angewendet werden? Begründen sie ihre Aussage.

- **`::flappy_box::view::BoxDrawable`** – Delegierter von `::view::GlRenderer`
  - Fügen Sie hier den rotierenden Kubus aus Aufgabenblatt 3 in die Funktion `visualize` ein.
  - Machen sie dann die Position des 3d-Kubus auf der Z-Achse von der im Box-Spielobjekt gespeicherten Position abhängig (nutzen sie dazu die OpenGL-Funktion `glTranslated`). Nutzten sie bei der Berechnung des Winkels den von `model::Game::timestamp` bereitgestellten Zeitstempel.
- **`::flappy_box::view::BoxAudible`** – Delegierter von `::view::AlRenderer`

  Diese Klasse ist für die zum Spielobjekt `flappy_box::model::Box` gehörige Soundausgabe zuständig. Diese sollte es ermöglichen den Abstand des Würfels vom Ursprung hörbar zu machen. Vorerst werden wir eine einfache Sinusschwingung ausgeben, deren Frequenz mit dem Abstand zur Zielhöhe variiert. Außerdem kann die Zielhöhe selbst über eine Schwingung repräsentiert werden.
  - Definieren sie in der Klasse zwei `double` Konstanten: die Basisfrequenz `base_frequency` soll mit Wert `400`[Hz] und die Zielfrequenz `target_frequency` mit einem Wert von `100`[Hz] initialisiert werden.

  Der Großteil des Quelltextes wird im Konstruktor (also nur einmal) ausgeführt:
  - Zunächst definieren wir mit den Funktionen `alGenSources` und `alSource3f` eine Geräuschquelle `_al_box_source` (speichern sie den Handle als Membervariable der

10 Der Rückgabewert (`std::chrono::duration<double>`) hat eine (tick-)`Periode` von `1`[s] und kann daher über die Methode `count` direkt in einen `double` Wert in der Einheit [s] umgewandelt werden.

11 Exponentielles Wachstum/Zerfall: de.wikipedia.org/wiki/Exponentielles_Wachstum

Klasse) die unser Spielobjekt repräsentiert und die Position `0.5`[m] auf der x-Achse (auf der rechten Seite) hat.

- Erzeugen sie einen Soundpuffer der eine `10`[s] lang anhaltende Sinusschwingung mit der Basisfrequenz enthalten soll. Hierzu nutzen sie `alGenBuffers`, `alutCreateBufferWaveform`. Lesen sie die entsprechende Dokumentation.
- Weisen sie den Soundpuffer der Geräuschquelle zu (`glSourcei`), aktivieren sie für diese Geräuschquelle die Wiedergabe in einer Endlosschleife (`AL_LOOPING`) aufrufen.
- Starten sie die Wiedergabe der Geräuschquelle (`alSourcePlay`).
- Analog sollen sie noch eine zweite Geräuschquelle und einen zweiten Geräusch-Puffer erzeugen der die Zielfrequenz von einer Position von `-0.5`[m] auf der x-Achse abspielt.

Nach dem die beiden Geräuschquellen bereits im Konstruktor gestartet wurden, muss in der Methode `auralize` nur noch die Frequenz beziehungsweise die Abspielgeschwindigkeit entsprechend des Positionsattributes des Spielobjektes variiert werden.

- Durch Aufruf der Funktion `alSourcef` mit dem Parameter `AL_PITCH`, kann ein `float` Wert übergeben werden, der einen Faktor für die Abspielgeschwindigkeit widerspiegelt. Das Verhältnis soll so gewählt werden das die Basisfrequenz des Box-Objektes auf eine gewünschte Frequenz geändert wird.

  Um dieses Verhältnis zu erhalten, muss die gewünschte Frequenz durch die Basisfrequenz geteilt werden. Die gewünschte Frequenz soll sich als Summe aus der Zielfrequenz sowie dem mit `10` multiplizierten Absolutwert der Position auf der Z-Achse ergeben.

Falls alles korrekt implementiert wurde, sollte die Frequenz der Box mit dem Abstand vom Ursprung steigen und um z=0 Schwebungen[12] mit dem Zielton hörbar werden.

Als letzter Schritt fehlt noch die Klasse `flappy_box::controller::FlappyEngine` in der die benötigten Komponenten der Engine erzeugt werden und die Delegierten bei den Factory-Objekten der Komponenten registriert werden.

- Vererben sie die Klasse von `controller::GlutEngine` und fügen sie Membervariablen vom Typ Pointer auf `view::GlRenderer` beziehungsweise auf `view::AlRenderer` hinzu.

Außerdem müssen die virtuellen Funktionen überschrieben werden:

- In der `init` Funktion muss zuerst die Initialisierung der Basisklasse aufgerufen werden, bevor wir mit der Funktion `alutInit` die Sound-Bibliotheken initialisieren.

  Nachdem alle Komponenten bereit sind, erzeugen wir unser Spielobjekt (`flappy::model::Box`) und fügen es dem dem Modell (`model::Game`) hinzu.

  Anschließend müssen sie noch die Delegierten der Komponenten bei den entsprechenden Factories(`GlRenderer::drawable_factory(),…`) registrieren.

12 Schwebung: de.wikipedia.org/wiki/Schwebung

- In der `run` Funktion muss nur das `view::GlutWindow` erzeugt und danach die `run`-Methode der Basisklasse aufgerufen werden, um das Spiel zu starten. Danach noch `alutExit` aufrufen und fertig! ;)

Viel Erfolg!

```
# include <memory>
# include <map>
# include <typeindex>

template< typename In, typename Out >
class factory_map
{
    typedef std::function< std::shared_ptr< Out > (const std::shared_ptr< In >&) >
         outer_function_type;

  public:

    // Register factory module for key/input type T.
    template< typename T >
    void register_module( const std::function< std::shared_ptr< Out > (const
std::shared_ptr<T>&) >& inner_function )
    {
      auto outer_function = [inner_function]( const std::shared_ptr< In >& in )
        {
          static_assert( std::is_base_of< In, T >::value
                       , "Key Type \"T\" must derive "
                         "from template parameter \"In\""
                       );
          auto derived_in = std::static_pointer_cast< T >( in );
          return inner_function( derived_in );
        };

      auto ins_result = _modules.insert( { typeid( T ), outer_function } );
      if( not ins_result.second )
        throw std::logic_error( "factory_map::register_module: "
                              + "A module with key "
                              + std::string( typeid( T ).name() )
                              + " is already registered."
                              );
    }

    // Find module for derived type of *in.
    std::shared_ptr< Out > create_for( const std::shared_ptr< In >& in )
    {
      auto found = _modules.find( typeid( *in ) );
      if( found == _modules.end() )
        throw std::out_of_range( "factory::create_for: "
                               + "Could not find module for key \""
                               + std::string( typeid( *in ).name() )
                               + "\"."
                               );
      return found->second( in );
    }

  private:
    std::map< std::type_index, outer_function_type > _modules;

}; // factory_map
```

Quelltextausschnitt 2: Inhalt der Datei `factory_map.hpp`.

```
# include "factory_map.hpp"
using namespace std;

//--- define model and delegates ---

struct Fruit { virtual ~Fruit() {}; };
struct Mango : public Fruit { double sweetness = 1000.; };
struct Lemon : public Fruit { bool has_worms = false; };

struct JuiceExtractor { virtual string make_juice() = 0; };
struct MangoExtractor : public JuiceExtractor
{
  MangoExtractor( const shared_ptr< Mango >& m ) : mango(m) {}
  virtual string make_juice() { return "Juice from a Mango with sweetness of "
                                  + to_string( mango->sweetness ) + " sugercubes";
                              }
  shared_ptr< Mango > mango;
};
struct LemonExtractor : public JuiceExtractor
{
  LemonExtractor( const shared_ptr< Lemon >& c ) : lemon( c ) {}
  virtual string make_juice() { return "Juice from a Lemon with "
                                  + string( !lemon->has_worms ? "no ":"" ) + "worms";
                              }
  shared_ptr< Lemon > lemon;
};

//--- prepare juice extraction setup ---

factory_map< Fruit, JuiceExtractor > extractor_toolbox;

extractor_toolbox.register_module< Mango >(
  []( const shared_ptr< Mango >& m ){ return make_shared< MangoExtractor >( m ); }
);
extractor_toolbox.register_module< Lemon >(
  []( const shared_ptr< Lemon >& l ){ return make_shared< LemonExtractor >( l ); }
);

//--- process (registered) fruits provided only via a base type (Fruit) ---

vector< shared_ptr< Fruit > > fruits;

fruits.emplace_back( new Lemon );
fruits.emplace_back( new Mango  );
auto bad_lemon = make_shared< Lemon >();
fruits.push_back( bad_lemon );
bad_lemon->has_worms = true;

for( auto f : fruits )
  cout << extractor_toolbox.create_for( f )->make_juice() << endl;
```

Quelltextausschnitt 3: Beispielnutzung der `factory_map`**:**
Zunächst wird eine Vererbungshirarchie für die Früchte und die Entsafterklassen definiert.
Danach erzeugen wir eine `factory_map` **die beliebige Früchte annimmt und den richtigen Entsafter für den abgeleiteten Fruchttyp zurückliefert. Zuerst muss dieser** `extractor_toolbox` **jedoch über die Methode** `register_module` **mitgeteilt werden, wie die unterschiedlichen Entsafter erzeugt und mit den passenden Fruchttypen bestückt werden können.**
Anschließend ruft die `extractor_toolbox` **für alle Früchte die richtige Erzeugungsfunktion auf, falls eine für den abgeleiteten Typ der Frucht registriert wurde. Diese Funktion erzeugt dann den passenden Entsafter und übergibt die Frucht. Beachten sie, dass den Entsaftern durch die vorhergehende Registrierung der abgeleitete Typ der Früchte verfügbar gemacht wird.**